Exploration of Alternative GPU Implementations of the Pair-HMMs Forward Algorithm

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

DOI
10.1109/BIBM.2016.7822645

Publication date
2016

Document Version
Accepted author manuscript

Published in
Proceedings 3rd International Workshop on High Performance Computing on Bioinformatics

Exploration of Alternative GPU Implementations of the Pair-HMMs Forward Algorithm

Shanshan Ren
Computer Engineering Lab
Delft University of Technology
2628CD Delft,The Netherlands
s.ren@tudelft.nl

Koen Bertels
Computer Engineering Lab
Delft University of Technology
2628CD Delft,The Netherlands
k.l.m.bertels@tudelft.nl

Zaid Al-Ars
Computer Engineering Lab
Delft University of Technology
2628CD Delft,The Netherlands
z.al-ars@tudelft.nl

***Abstract*—In order to handle the massive raw data generated by next generation sequencing (NGS) platforms, GPUs are widely used by many genetic analysis tools to speed up the used algorithms. In this paper, we use GPUs to accelerate the pair-HMMs forward algorithm, which is used to calculate the overall alignment probability in many genomics analysis tools. We firstly evaluate two different implementation methods to accelerate the pair-HMMs forward algorithm according to their effectiveness on GPU platforms. Based on these two methods, we present several implementations of the pair-HMMs forward algorithm. We execute these implementations on the NVIDIA Tesla K40 card using different datasets to compare the performance. Experimental results show that the intra-task implementation has the highest throughput in most cases, achieving pure computational throughput as high as 23.56 GCUPS for synthetic datasets. On a real dataset, the inter-task implementation achieves 4.82x speedup compared with a vectorized implementation executed on a 20-core POWER8 system.**

I. Introduction

Compared with first generation sequencing technology, Next Generation Sequencing (NGS) technology [1] is fast, low-cost and high-throughput, which brings great opportunities for new discoveries in disease diagnosis and personalized medicine. The raw DNA data produced by NGS platforms need to be processed by a series of complex genomic analysis tools to turn into meaningful data for genomic research, which is referred as to NGS data analysis. However, due to the large size of used datasets, it would take long time to perform NGS data analysis, even using high-performance systems or big clusters. Acceleration of genetic analysis tools is needed to render them more efficient.

Since many bioinformatics workloads are easy to execute in parallel, we can use dedicated hardware, such as GPUs and FPGAs, instead of general-purpose processors, to accelerate the computationally intensive algorithms and achieve large speedups. For example, the Smith-Waterman algorithm, widely used to identify optimal sequence alignment of two sequences, can be implemented on GPUs and FPGAs to improve the performance [2][5].

The pair-HMMs (pair hidden Markov models) forward algorithm (or PFA) is used to calculate the overall alignment probability of two sequences, making it a common algorithm in many genetic analysis tools, such as [6][7]. However, PFA consumes a large proportion of the overall execution time of the GATK HaplotypeCaller. In this paper, we implement PFA on GPUs in order to reduce its execution time and thus improve the overall performance.

In this paper, we present the following contributions: (1) evaluate two methods to implement PFA on GPUs; (2) present several implementations of PFA on GPUs based on these two methods; (3) discuss the performance of these implementations with different datasets on GPUs.

The rest of this paper is organized as follows: Section II presents a brief overview of related work and describes the details of PFA. This section also evaluates two acceleration methods on GPUs: inter-task and intra-task. Section III presents two implementations of the inter-task acceleration method, while Section IV shows three implementations of the intra-task acceleration method. Section V discusses several GPU-specific optimizations. Section VI presents the experimental results of different GPU implementations. Section VII concludes the paper and discusses future work.

II. Background

A. Pair-HMMs Forward Algorithm

In this paper, we mainly focus on PFA in the way it is defined in the GATK HaplotypeCaller, in which PFA takes a read and a haplotype as the input and calculates the overall alignment probability [3]. Equations (1) to (3) define how PFA is performed in the GATK HaplotypeCaller. In these equations, m and n are the length of the read R and the haplotype H, respectively. $M_{i,j}$ stands for the overall alignment probability of two sub-sequence $R_1 \dots R_i$ and $H_1 \dots H_j$. $I_{i,j}$ stands for the overall alignment probability of $R_1 \dots R_i$ and $H_1 \dots H_j$ when R_i aligned to gap. $D_{i,j}$ stands for the overall alignment probability of $R_1 \dots R_i$ and $H_1 \dots H_j$ when H_j aligned to gap.

Initialization:

$$\begin{cases} M_{i,0} = I_{i,0} = D_{i,0} = 0 & (0 \le i \le m) \\ M_{0,j} = I_{0,j} = 0 & (0 \le j \le n) \\ D_{0,j} = \frac{1}{n} & (0 \le j \le n) \end{cases} \quad (1)$$

Recurrence:

$$\begin{cases} M_{i,j} = \lambda_{i,j}(\alpha_i M_{i-1,j-1} + \beta_i I_{i-1,j-1} + \beta_i D_{i-1,j-1}) \\ I_{i,j} = \delta_i M_{i-1,j} + \epsilon_i I_{i-1,j} \\ D_{i,j} = \zeta_i M_{i,j-1} + \eta_i D_{i,j-1} \end{cases} \quad (2)$$

Termination:

$$Result = \sum_{j=1}^{n} M_{m,j} + I_{m,j} \quad (3)$$

where $\lambda_{i,j}$ is the emission probability, which has two different values, depending on the base value of the read at position i and the base value of the haplotype at position j. α_i, β_i, δ_i, ϵ_i, ζ_i and η_i are transmission probabilities that only depend on the read position i.

Algorithm 1 Pseudo code of PFA in the GATK HaplotypeCaller

procedure ALGORITHM($R[\,]$, $H[\,]$, $\alpha[\,]$, $\beta[\,]$, $\delta[\,]$, $\epsilon[\,]$, $\zeta[\,]$, $\eta[\,]$, $\lambda[\,]$)
  $M \leftarrow I \leftarrow D \leftarrow 0$
  $D_{0,0\ldots n} \leftarrow \frac{1}{n}$
  for $i \leftarrow 1, m$ **do**
    for $j \leftarrow 1, n$ **do**
      $M_{i,j} \leftarrow \lambda_{i,j} \cdot (\alpha_i \cdot M_{i-1,j-1} + \beta_i \cdot I_{i-1,j-1} + \beta_i \cdot D_{i-1,j-1})$
      $I_{i,j} \leftarrow \delta_i \cdot M_{i-1,j} + \epsilon_i \cdot I_{i-1,j}$
      $D_{i,j} \leftarrow \zeta_i \cdot M_{i,j-1} + \eta_i \cdot D_{i,j-1}$
    end for
  end for
  return $\sum_{j=1}^{n} M_{m,j} + I_{m,j}$
end procedure

Algorithm 1 shows the pseudo code of PFA used in the GATK HaplotypeCaller. It uses a two-layer loop to calculate every element of three matrices M, I and D, which results in the $O(mn)$ computational complexity of the algorithm. In the GATK HaplotypeCaller, this algorithm is performed millions of times, even in smaller datasets such as the 4.3 GB experimental dataset, where the algorithm is executed more than 3 million times [4]. Thus, acceleration of PFA is very important to improve the performance of the GATK HaplotypeCaller and other genetic analysis tools.

As shown by Equations (1) to (3) and Algorithm 1, each element of M, I and D only depends on the top, left and top-left neighbor elements in the three matrices. Fig. 1 shows the data dependency in PFA, which implies that all elements on the same anti-diagonal of each matrix can be computed in parallel as there is no data dependency among these elements. This defines the inherent parallelism of the algorithm, which could be exploited for algorithm acceleration.

B. Related Work

Previous research on GPU-based acceleration of algorithms related to HMMs (Hidden Markov models) in computational biology mainly focuses on HMMer, which is a software suite for protein sequence similarity calculation using probabilistic models called profile hidden Markov models (profile-HMMs) [8][9]. However, the input of the profile-HMMs forward algorithm in HMMer is composed of a sequence and a reference database, which is much different from the input of PFA in the GATK HaplotypeCaller. Here, the input is composed of many different sequence-pairs, which makes the data storage and data access on GPUs different from the GPU implementations of HMMer.

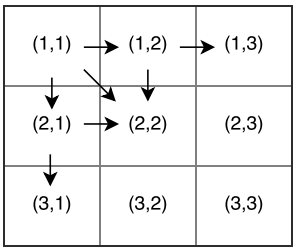

Fig. 1. Data dependency in the pair-HMMs forward algorithm

GPUs are also widely used to accelerate the Smith-Waterman alignment algorithm [2][10], which is similar to PFA. However, operations of Smith-Waterman are mainly integer based, while operations of PFA are in the floating-point domain. Moreover, the equations of PFA have several emission and transmission probabilities, which makes the register usage of PFA on GPUs larger than that of the Smith-Waterman alignment algorithm.

Research on increasing the speed of PFA by optimizations and acceleration can be found in [11][12][13][14]. Intel researchers employ vector instructions to reduce the execution time on Intel processors [11]. In the same way, IBM researchers provide a vectorized implementation with vector instructions on POWER processors [12].

In addition, [13] and [14] propose FPGA-based implementations of PFA on the Convey Computer platform and the IBM POWER8 platform, respectively. [13] employs a systolic array to map the algorithm on FPGAs. [14] proposes pipelined PEs (processing elements) of the systolic array, in addition to using the CAPI interface of the IBM POWER8 platform, which makes the data transfer more efficient.

In this paper, we analyze existing GPU acceleration strategies published in the literature to accelerate similar algorithms and compare their acceleration potential to PFA. Based on this, the best solution alternative is selected and used to accelerate the algorithm.

C. GPU Architecture

GPUs are originally designed to process graphics and images. Due to their highly parallel structure, modern GPUs are used as accelerators to implement algorithms, which process large data in parallel. NVIDIA GPUs have a number of multiprocessors, which are called Streaming Multiprocessors (SMs), and each SM consists of many cores. Taking the NVIDIA GeForce GTX TITAN X card as an example, the card has 24 SMs with each SM consisting of 128 cores. In

total, this card has 3072 cores that can execute in parallel for algorithms with abundant parallelism.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model that enables programmers to use NVIDIA GPUs for general purposed processing efficiently. In the context of CUDA, when the host processors launch a GPU kernel, a large number of threads are generated on the GPU device. Each thread executes an instance of the GPU kernel. CUDA introduces a two-level thread hierarchy to organize threads: blocks of threads and grids of blocks. A grid is a collection of all threads produced by a GPU kernel and a thread block is a collection of many threads; a grid consists of many thread blocks, each having the same number of threads. When mapping hardware architecture to CUDAs thread hierarchy, a single GPU chip executes one or more grids; an SM handles one or more thread blocks; a core handles one or more threads.

D. Two Acceleration Methods on GPUs

According to the characteristics of GPU platforms and the inherent parallelism of PFA, there are two methods to accelerate this algorithm on GPUs based on the parallelism granularity: inter-task parallelization and intra-task parallelization.

a) Inter-task parallelization: The general way to accelerate an algorithm on GPUs is to map the whole processing of the algorithm to a single thread, in order to have many copies of the algorithm running in parallel. Each thread implements the algorithm independently. This is called inter-task parallelization, which is also called coarse-grained parallelization.

b) Intra-task parallelization: As mentioned in Section II-A, in PFA, all the elements on the same anti-diagonal of the matrices can be computed in parallel. We exploit this inherent parallelism by mapping the algorithm on a single thread block. Each thread in the thread block computes the values of elements in the same column as all threads work along the wave-front from the top-left to the bottom-right corner of the matrix (Fig. 1). This method requires a whole block of threads to compute a single copy of the algorithm. This reduces the number of copies of the algorithm that can be executed in parallel on the GPUs compared to the inter-task parallelization. However, the computation complexity of the algorithm implemented using this method is reduced to a linear $O(m+n)$ instead of the quadratic $O(mn)$ required in the first method.

In the following sections, we will present several implementations of the two methods and compare the performance based on different datasets.

III. Inter-task Implementations

In this section, we firstly present a naive implementation of the inter-task method and later present an optimized implementation, which is based on tiling.

A. Naive Implementation

As mentioned in Section II-D, in the inter-task method, each thread implements PFA independently. Each thread computes the overall alignment probability of a read-haplotype pair as described in Algorithm 2.

Algorithm 2 Pseudo code of PFA using the inter-task method

procedure ALGORITHM($R[\,]$, $H[\,]$, $\alpha[\,]$, $\beta[\,]$, $\delta[\,]$, $\epsilon[\,]$, $\zeta[\,]$, $\eta[\,]$, $\lambda[\,]$)
  for $i \leftarrow 1, m$ **do**
    $MU \leftarrow IU \leftarrow 0$
    $DU \leftarrow \frac{1}{n}$
    $MID \leftarrow \beta_i \cdot DU$
    for $j \leftarrow 1, n$ **do**
      if $i > 1$ **then**
        $MU \leftarrow$ MM$_j$
        $IU \leftarrow$ II$_j$
        $DU \leftarrow$ DD$_j$
      end if
      $DN \leftarrow \zeta_i \cdot MN + \eta_i \cdot DN$
      $MN \leftarrow \lambda_{i,j} \cdot MID$
      $IN \leftarrow \delta_i \cdot MU + \epsilon_i \cdot IU$
      $MID \leftarrow \alpha_i \cdot MU + \beta_i \cdot IU + \beta_i \cdot DU$
      $MM_j \leftarrow MN$
      $II_j \leftarrow IN$
      $DD_j \leftarrow DN$
    end for
  end for
  return $\sum_{j=1}^{n} M_{m,j} + I_{m,j}$
end procedure

In Algorithm 2, each thread employs three vectors in the global memory to store the values of top neighbor elements, which are MM, II and DD, while MN, IN and DN are used to store the values of left neighbor elements. MU, DU and IU are used to store the data loaded from the MM, II and DD vectors. MID is used to store the intermediate values of top-left neighbor elements. By using MID, the algorithm avoids to load the value of top neighbor elements from the global memory twice.

Algorithm 2 has two loops to iterate on the all the bases of: 1. the read, and 2. the haplotype. We implement Algorithm 2 to iterate on the read bases in the outer loop and on the haplotype bases in the inner loop. This way, we need to load the emission and transmission probabilities only once. On the other hand, if we iterate on the haplotype bases in the outer loop, the emission and transmission probabilities need to be loaded in each iteration, which is equal to the length of haplotype. Thus, iterating on the read bases in the outer loop reduces significantly the number of times we need to access the memory.

B. Tile-based Implementation

In the naive implementation of the inter-task method, the number of global memory accesses is large. This is because calculating $M_{i,j}$, $I_{i,j}$ and Di,j involves six global memory

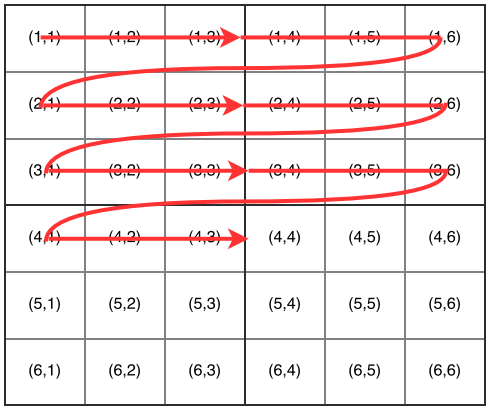

Fig. 2. Execution trace of the inter-task implementation *without tiling*

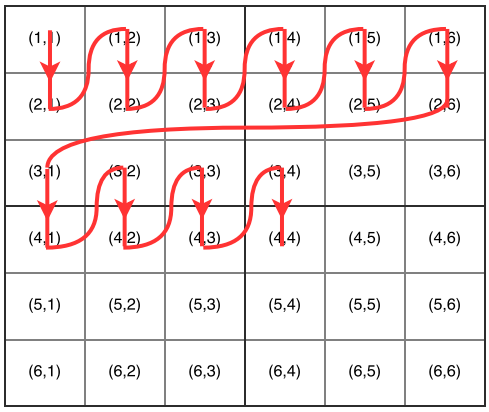

Fig. 3. Execution trace of the inter-task implementation *with tiling*

accesses, three for loading $M_{i-1,j}$, $I_{i-1,j}$ and $D_{i-1,j}$, and three for storing $M_{i,j}$, $I_{i,j}$ and Di, j, which amounts to $6 \times m \times n$ accesses.

In order to reduce the number of global memory accesses, we employ tiling technique [15]. A tile covers several successive elements in one column, where the number of elements is the size of the tile.

Fig. 2 shows the execution trace of each thread without tiling: it firstly calculates the elements in the first row, then calculates the elements in the second row, and so on. Fig. 3 shows the execution trace of each thread with tiling (tile size is 2): it firstly calculates the first tile in the first column, then calculates the first tile in the second column, until it finishes calculating the first tile in the last column, after which it starts calculating the second tile in the first column, and so on. In fact, the naive implementation could be considered as an implementation with tile size equal to 1.

When calculating the elements in one tile, each thread only loads three values used to calculate the first element in the tile and stores three values of the last element in the tile. Thus, by executing horizontally tile by tile, the number of total memory accesses is reduced to $(6 \times m \times n)$ / k, where k is the size of the tile.

The cost of reducing global memory accesses is that each thread uses more registers and shared memory to store the values of $M_{i,j}$ and $D_{i,j}$ of each element in one tile, which are used to calculate the elements in the neighboring tile. As registers and shared memory are scarce resources on GPUs, the tile size cannot continue to increase. We will discuss the optimal tile size in Section VI.

C. Optimization by sorting

In CUDA, threads in the same block are divided into multiple groups of threads called *warps*. For many NVIDIA GPUs, there are 32 threads in each warp. The threads in the same warp execute the same instruction at the same time, which is referred to as SIMT (Single Instruction Multiple Thread) execution model.

Although all threads implement PFA independently in this method, threads in the same warp have to wait for each other to finish their workload because of the SIMT execution model. In order to reduce the waiting time, we can sort the pairs according to the length of reads and haplotypes before sending them to the GPU.

IV. Intra-task Implementations

We firstly show a naive implementation of the intra-task method and then discuss two modified implementations: the warp-based implementation and tile-based implementation.

A. Naive Implementation

In the intra-task method, all the threads in the same block implement the pair-HMMs forward algorithm, shown in Fig. 4(a). We firstly assume the number of threads is equal to the length of the read. As mentioned before, all the threads work in a wavefront mode from top-left to bottom-right, which is implemented as: at the first step, thread 0 (T0) calculates the $M_{1,1}$, $D_{1,1}$ and $I_{1,1}$ matrix elements; at the second step, T0 calculates $M_{2,1}$, $D_{2,1}$ and $I_{2,1}$, while T1 calculates $M_{1,2}$, $D_{1,2}$ and $I_{1,2}$; at the third step, T0 calculates $M_{3,1}$, $D_{3,1}$ and $I_{3,1}$, T1 calculates $M_{2,2}$, $D_{2,2}$ and $I_{2,2}$, and T2 calculates $M_{1,3}$, $D_{1,3}$ and $I_{1,3}$, etc.

A synchronization function is called after each step to ensure that the different threads are synchronized. The synchronization function is supplied in CUDA to allow threads in the same block to cooperate with each other. There are in total $m + n - 1$ steps to finish the algorithm in the thread block, which means there are $m + n - 1$ synchronizations in the implementation.

Although there is no data dependency among elements on the same anti-diagonal, there are data dependencies among elements on adjacent anti-diagonals. For example, when T1 calculate $M_{1,2}$, $D_{1,2}$ and $I_{1,2}$, it needs the values of $M_{1,1}$ and $D_{1,1}$ from T0. The data dependencies are realized by storing data in the shared memory, which is read/written by all the threads in the same block. We apply vectors in the shared memory to store these value. The length of the vectors is equal to the number of the threads. At the beginning of each step, threads read data from the shared memory and at the end of

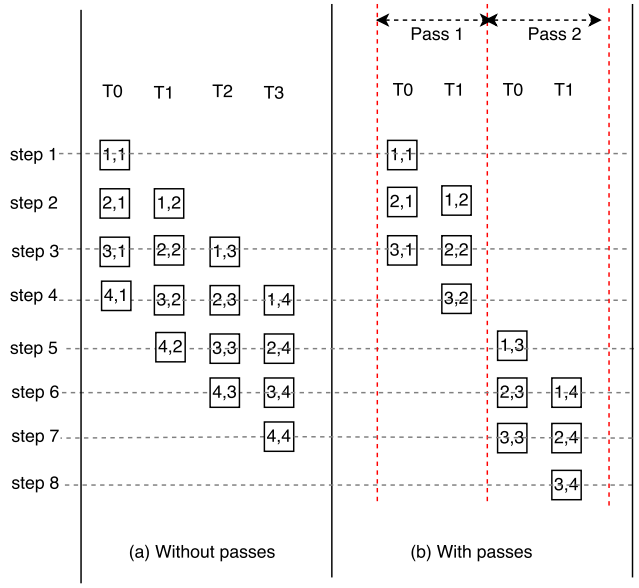

Fig. 4. Execution trace of the intra-task implementation (a) *without passes* when the number of threads is bigger than or equal to the length of the read (b) *with passes* when the number of threads is smaller than the length of the read.

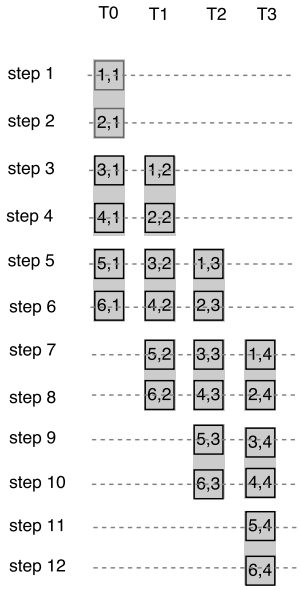

Fig. 5. Execution trace of the intra-task implementation with tiling

each step, threads write data into the shared memory. In this way, threads in the same block are able to exchange data.

When the number of threads is bigger than the length of the read, the threads are capable of fully processing the read. However, when the number of threads is smaller than the length of the read, we divide the calculation into several passes and store the intermediate data between passes into the global memory. Fig. 4(b) illustrates this implementation. We assume the number of threads is 2 and the length of the read is 4. Thus, there are two passes to implement this algorithm.

B. Warp-based Implementation

Since the synchronization function causes threads to stall and wait for other threads in the same block, it leads to lower GPU utilization efficiency. Thus, it is better to reduce the number of synchronization function calls. This can be addressed in two ways: warp-based implementation and tile-based implementation. This section introduces the warp-based implementation.

As mentioned in Section III-C, due to the SIMT execution model, threads in the same warp need to execute the same instruction at the same time. This means threads in the same warp synchronize with each other without calling a synchronization function. We can make use of this characteristics by setting the block size to 32, which is called warp-based implementation. In this way, there is no synchronization in the warp-based implementation.

In the warp-based implementation, we utilize the shuffle instructions to exchange data among threads instead of using shared memory. Shuffle instructions enable thread to directly read the registers of other threads in the same warp. Thus, shared memory is not needed for data exchange.

As the block size of the warp-based implementation is 32, which is in many cases smaller than the length of the read sequence, the warp-based implementation needs multiple passes to process the read. In this implementation, we store the intermediate results of each pass into the shared memory.

However, the warp-based implementation cannot fully utilize the resources on GPUs due to the small block size. One way to redeem this is to increase the block size and make each 32 threads in the same warp implements PFA independently, which we call improved warp-based implementation. For example, we set the block size to 128 and there are 4 warps in each block, each of which implements PFA independently. In this way, it can utilize more GPU resources. The disadvantage is that this implementation has to store the intermediate results into the global memory, which causes large latency. In Section VI-C, we will discuss the resource utilization and global memory latency of the warp-based implementations.

C. Tile-based Implementation

In the inter-task method, we use the tiling technique to reduce memory accesses. Here, we exploit the tiling technique to reduce the number of synchronization function calls.

Fig. 5 shows the tile-based implementation of the intra-task method. In the naive implementation, at each step, each thread only calculates the value of one element in each column. In contrast, in the tile-based implementation, at every k steps, each thread calculate the value of elements in one tile in each column. Take Fig. 5 for example, where tile size k is 2. At the first 2 steps, T0 calculates the values of the first tile in the first column; at the second 2 steps, T0 calculates the values of the second tile in the first column and T1 calculates the values of the first tile in the second column, etc.

There are $m + \lceil n/k \rceil - 1$ synchronizations in the tile-based implementation with tile size equal to k. However, in the tile-based implementation, we use more vectors in the shared memory to store the data exchanged among threads. As mentioned in Section III, shared memory is a scarce resource on GPUs. The tile size cannot continue to increase.

Moreover, it increases the number of execution steps. For example, in Fig. 5, we assume the length of the read and haplotype to be 4 and 6, respectively. In the naive implementation, it takes 9 steps to compute the result; while, it takes 12 steps to compute the result in the tile-based implementation. This is because the elements in each tile are calculated serially.

The tiling technique does not influence the execution steps in the inter-task method because there are $m \times n$ steps in the implementation with/without tiling. All the elements are calculated serially in the implementation with/without tiling.

D. Intra-task drawback

The drawback of the intra-task method is that it cannot fully utilize the GPU computation resources. This is because not all the threads in the block keep busy. In Fig. 4(a), for example, at the first 3 steps and at the last 3 steps, some threads are idle. Only at step 4 all threads are busy. Full utilization is only achieved while calculating the "widest" diagonals of the matrices.

V. GPU-Specific Optimizations

a) Coalesced Memory Access: On GPUs, coalesced global memory accesses can reduce the global memory transactions and make good use of the global memory bandwidth. Thus, global memory accesses are coalesced as much as possible in all the implementations.

b) Intrinsic Instructions: CUDA supplies intrinsic instructions, which work faster than native instructions. For example, $\_\_fmaf\_rn()$ computes the value of $x \times y + z$ as a single ternary operation instead of two dualism operations. Thus, the floating-point operations in all the implementations employ intrinsic instructions.

VI. Experimental Results and Discussion

A. Experimental Setup

All the experiments are performed on IBM Power System S824L (82478-42L). This system has two IBM Power8 processors (10 cores each) running at 3.42 GHz with 258 GB of DDR3 memory, and an NVIDIA Tesla K40 card, with 2880 cores, running at up to 745 MHz and offering compute capability 3.5.

The performance measure of cell updates per second (CUPS) is widely used in bioinformatics. A CUPS represents the time for a complete computation of one cell in the matrix, including all memory operations and corresponding overhead. The number of cells in a matrix can be calculated by $m \times n$, where m and n are the length of two sequences (here, read and haplotype). Given a chunk of read-haplotype pairs, the GCUPS (Giga cell updates per second) value is calculated by:

$$\frac{\sum_{i=1}^{s} m_i \times n_i}{t \times 10^9} \tag{4}$$

where s is the size of read-haplotype pairs in the chunk, m_i and n_i are length of ith read and ith haplotype, and t is the runtime in seconds. In this paper, except Section VI-E, the runtime t is the calculation time of the pair-HMMs forward algorithm on GPUs, which does not include the time to transfer data between host and GPU.

We firstly use synthetic datasets to compare the performance of implementations of the inter-task and intra-task methods. We then use a real dataset to compare the performance of the GPU-based implementations with the software implementations.

We generated 8 types of synthetic datasets, which are shown in Table I. Each one of the 8 datasets is built using read-haplotype pairs with a specific length. These read-haplotype pairs are chosen from chromosome 10 of the whole human genome dataset G15512.HCCI954.1 [14].

TABLE I
Synthetic datasets with various read-haplotype lengths

Dataset	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Read length	27	60	96	101	101	101	101	128
Haplotype length	32	64	96	128	160	192	224	256

B. Implementations of Inter-task Method

Fig. 6 shows the performance of various implementations of the inter-task method. We set the number of the read-haplotype pairs in each dataset to 10^6. Fig. 6 shows that the throughput of each implementation for all the datasets is comparable. For example, the throughput of the inter-task implementation with tile=2 remains around 13 GCUPS for all datasets.

For all the datasets, the throughput of the naive implementation is much lower than other implementations with tiling. It indicates that the tiling technique improves performance in the inter-task method.

The implementation with tile=4 achieves highest throughput over other implementations with tiling. It explains that the tile size cannot continue to increase to bigger values. This is because although the implementations with bigger tile sizes reduce global memory accesses, they increase the usage of shared memory and registers, which in turn decreases the number of active warps and active blocks running on each SM.

C. Implementations of Intra-task Method

Fig. 7 shows the performance of the implementations in the intra-task method. The number of the read-haplotype pairs in each dataset is 10^6. Unlike the implementations in the inter-task method, the throughput of the implements in the intra-task method increases when the length of the read and haplotype increases, except for the warp-based implementations. For example, the throughput of the naive implementation for the different datasets increases from 4.23 GCUPS up to 23.56 GCUPS as the sequence sizes increase.

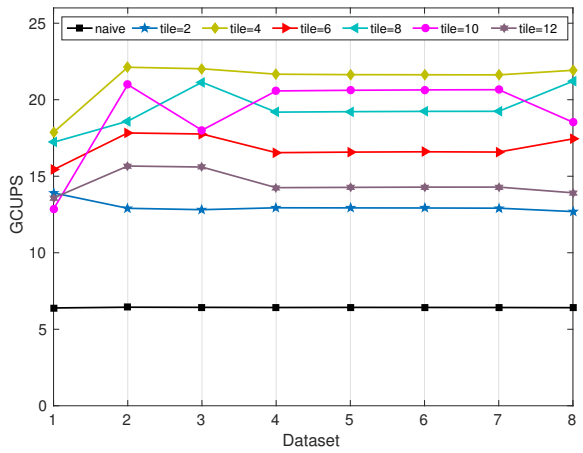

Fig. 6. Inter-task implementation performance comparison on different datasets

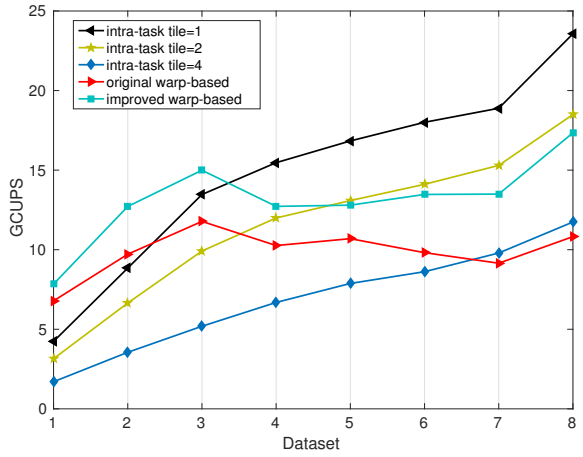

Fig. 7. Intra-task implementation performance comparison on different datasets

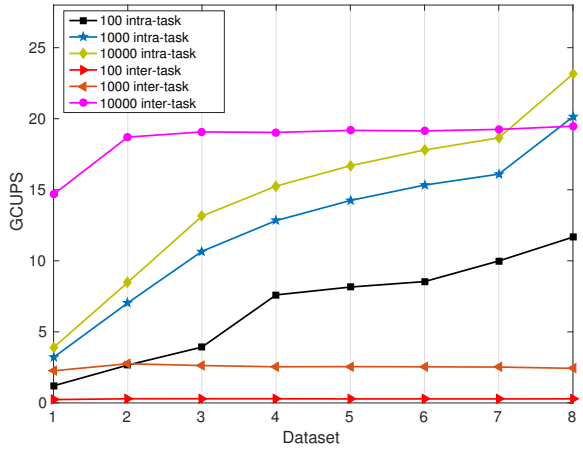

Fig. 8. Inter-task/intra-task implementation performance comparison on different datasets

This is because the computation complexity of the implementations in the intra-task method is $O(m+n)$, which means that the execution time increases linearly with length. However, the total number of cells computed is $10^6 \times m \times n$ for all the synthetic datasets, which increases quadratically with length. Thus, according to Equation (4), the throughput increases linearly with length.

Fig. 7 shows that the throughput of the naive implementation is bigger than the tile-based implementations. This is because the tiling technique in the intra-task method increases the number of execution steps that seems to outweigh the reduction in the number of synchronization calls.

The warp-based implementations achieve higher throughput than the naive implementation for the datasets with short sequence lengths. However, the throughput of the warp-based implementations become lower for the datasets with large sequence lengths. For the original warp-based implementation, the reason is that the number of the active warps on each SM is 16, which causes that GPU resource utilization is low. When sequence lengths are small, the advantage brought by reducing synchronization and shuffle instructions outweigh the resource under-utilization; however, when the sequence lengths are large, the situation is the opposite. For the improved warp-based implementation, the reason is that when the sequence length is bigger than 32 (warp size), it needs to put the intermediate result into the global memory. When sequence lengths are small, the total latency of global memory accesses is small, which does not outweigh the advantage brought by reducing synchronization and shuffle instructions; however, when sequence lengths are large, the situation is the opposite.

D. Intra-task vs Inter-task

This section compares the performance of the best implementations from these two methods. For the inter-task method, we use the implementation with tile=4; for intra-task method, we use the naive implementation. We use 8 types of datasets to measure the performance. The number of read-haplotype pairs in each type of datasets is set to 100, 1000 and 10000, respectively. Fig. 8 shows that the throughput of these two implementations increases when the number of pairs increases.

As shown in Fig. 8, the throughput of the inter-task implementation remains nearly the same for all datasets. However, the throughput of the warp-based implementation increases with increasing sequence lengths.

Moreover, the intra-task implementation achieves higher throughput when the number of read-haplotype pairs is 100 and 1000. The main reason is that when the number of read-haplotype pairs in the dataset is small, the intra-task implementation utilizes more resources than the inter-task method. In the inter-task method, each thread calculates the overall alignment probability of one read-haplotype pair. If the number of pairs in the dataset is 100, only 100 cores are utilized, while there are 2880 cores on the GPU card. With regards to the intra-task method, each block calculates the overall alignment probability of one read-haplotype pair. If there are 100 pairs in the dataset and each block has only

32 threads (the worst situation), all the cores on the GPU card would participate in the calculation theoretically.

E. Real dataset

The real dataset is produced by the GATK HaplotypeCaller version 3.6 from chromosome 10 of the whole human genome dataset G15512.HCCI954.1. There are two software implementations to compare the performance with. One is programmed in C++ using vector instructions, compiled with O3 optimization, and profiled on one POWER8 core; the other is a multicore implementation based on the first one using OpenMP and it is profiled on two POWER8 processors (20 cores in total).

The real dataset produced by the GATK HaplotypeCaller is divided into small chunks, which have different number of read-haplotype pairs ranging from 4 to 38912. As discussed before, the GPU-based implementation achieves higher throughput with bigger datasets. We reorganized the dataset and set the number of read-haplotype pairs in each chunk to 10^6.

We use the naive implementation of intra-task method and the tile=4 implementation of inter-task method to run real dataset. Here, the runtime t used to calculate GCUPS includes the calculation time of PFA on GPUs and the time used to transfer data between host and GPU.

TABLE II
PERFORMANCE OF IMPLEMENTATIONS WITH REAL DATASET

Implementations	Time (s)	Throughput (GCUPS)
Intra-task (original dataset)	47.82	5.28
Intra-task (reorganized dataset)	21.85	11.56
Inter-task (original dataset)	359.25	0.70
Inter-task (reorganized dataset)	19.75	12.79
Software-based (1 core)	1161	0.22
Software-based (OpenMP)	95.16	2.65

Table II shows the results of the six implementations. The inter-task implementation with reorganized dataset achieved the biggest throughput, which is 12.79 GCUPS. It is 58.78x faster than the vectorized implementation on one POWER8 core. It is also 4.82x faster than the vectorized implementation on a 20-core system. Moreover, the inter-task implementation with reorganized dataset is 18.19x faster than with original dataset.

The intra-task implementation with reorganized dataset is 53.14x faster than the software implementation on one POWER8 core. It is also 4.36x faster than the parallelized software implementation with OpenMP on 20 POWER8 cores. Reorganizing the dataset makes intra-task implementation 2.19x faster.

The GPU implementations with reorganized dataset achieve larger throughput than with original dataset. However, in most cases, reorganizing dataset involves the modification of the source codes of applications. With original dataset, it is better to use intra-task implementation. Shown in Table II, the intra-task implementation with original dataset is 7.5x faster than the inter-task implementation with original dataset.

VII. CONCLUSION

In this paper, we evaluated two methods to map the pair-HMMs forward algorithm on GPUs and present several implementations for each method. A number of architectural features have been employed to improve the performance, such as the tiling technique, warp-based, shuffle instructions and intrinsic instructions. We realized all the implementations on an NVIDIA Tesla K40 card. Experimental results show that intra-task implementation achieves the highest throughput over other implementations, achieving pure computational throughput as high as 23.56 GCUPS for synthetic datasets. By using a real dataset, the intra-task implementation is 4.82x faster than the vectorized implementation on a 20-core POWER8 system.

In the future, we plan to integrate the GPU-based implementation of the pair-HMMs forward algorithm into the GATK HaplotypeCaller to improve the overall performance of the tool.

REFERENCES

[1] Jay Shendure and Hanlee Ji, Next-generation DNA sequencing. Nature. Biotechnology 26, 2008, 1135-1145.

[2] Y. Liu, A. Wirawan and B. Schmidt, "Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions", BMC bioinformatics, vol. 14, no. 1, pp. 117, 2013.

[3] Genome Analysis Toolkit, https://software.broadinstitute.org/gatk/

[4] GCAT: Genome Comparison and Analytic Testing, http://www.bioplanet.com/gcat

[5] Li TI, Shum W, Truong K: 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). BMC Bioinformatics. 2007, 8: I85-10.1186/1471-2105-8-85.

[6] Bjarne Knudsen, Michael M. Miyamoto, Sequence Alignments and Pair Hidden Markov Models Using Evolutionary History. J. Mol. Biol., 2003, 333, 453-460.

[7] DePristo M, et al., A framework for variation discovery and genotyping using next-generation DNA sequencing data. 2011 NATURE GENETICS 43:491-498

[8] Xiaoqiang Li, Wenting Han, Gu Liu, Hong An, Mu Xu, Wei Zhou, Qi Li , A Speculative HMMER Search Implementation on GPU, 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, CHINA, 2012

[9] Jiang H, Ganesan N. CUDAMPF: a multi-tiered parallel framework for accelerating protein sequence search in HMMER on CUDA-enabled GPU. BMC Bioinformatics 2016;17:106.

[10] A. Bustamam, G. Ardaneswari, and D. Lestari. Implementation of cuda gpu-based parallel computing on smith-waterman algorithm to sequence database searches. In Advanced 17 CILAMCE 2014 Computer Science and Information Systems (ICACSIS), 2013 International Conference on, pages 137142, Sept 2013.

[11] https://www.biostars.org/p/96431/

[12] http://gatkforums.broadinstitute.org/gatk/discussion/4833/speed-up-haplotypecaller-on-ibm-power8-systems

[13] Ren, Shanshan, Sima, Vlad-Mihai and Al-Ars, Zaid. FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis, IEEE International Conference on Bioinformatics and Biomedicine (BIBM), 1465-1470, 2015.

[14] Ito, Megumi and Ohara, Moriyoshi. A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for pair-HMM algorithm. 2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX), 1-3, 2016.

[15] D. Hains, Z. Cashero, M. Ottenberg, W. Bohm, and S. Rajopadhye, Improving CUDASW, a parallelization of smith-waterman for CUDA enabled devices, in Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, Workshops and Phd Forum (IPDPSW' 11), pp. 490501, May 2011