2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Parallel Algorithms for Approximate String Matching with k Mismatches on CUDA

Yu Liu*,† Longjiang Guo* † §, Jinbao Li* †, Meirui Ren* † and Keqin Li‡
*School of Computer Science and Technology, Heilongjiang University, Harbin, China 150080
†Key Laboratory of Database and Parallel Computing, Heilongjiang Province, Harbin, China 150080
Email: longjiangguo@gmail.com, ly82642472@126.com
‡Department of Computer Science, State University of New York, New Paltz, New York, USA 12561
Email: lik@newpaltz.edu

***Abstract*—Approximate string matching using the k-mismatch technique has been widely applied to many fields such as virus detection and computational biology. The traditional parallel algorithms are all based on multiple processors, which have high costs of computing and communication. GPU has high parallel processing capability, low cost of computing, and less time of communication. To the best of our knowledge, there is no any parallel algorithm for approximate string matching with k mismatches on GPU. With a new parallel programming model based on CUDA, we present three parallel algorithms and their implementations on GPU, namely, the thread parallel algorithm, the block-thread parallel algorithm, and the OPT-block-thread parallel algorithm. The OPT-block-thread parallel algorithm can take full advantage of the powerful parallel capability of GPU. Furthermore, it balances the load among the threads and optimizes the execution time with the memory model of GPU. Experimental results show that compared with the traditional sequential algorithm on CPU, our best parallel algorithm on GPU in this paper achieves speedup of 40–80.**

***Keywords*-Approximate string matching; CUDA; GPU; Hamming distance; parallel algorithm;**

I. Introduction

Pattern matching of strings is one of the basic problems in computer science. Most researches in early times focused on accurate and exact pattern matching. With the development of modern information technology, it is found that exact string matching cannot satisfy the various requirements in real applications. Approximate string matching with some differences becomes necessary and more appropriate. For instances, in sequences analysis and genome comparison, which are important subjects of computational biology, approximate string matching is an important method and technique. Moreover, technologies of approximate string matching are widely used in virus diagnosis, voice recognition, file searching, pattern recognition, OCR correction, and many other areas.

The earliest method for approximate string matching is based on Hamming distance, which was introduced by Hamming in his fundamental paper of error detection and correction to count the number of different characters between two strings. He also defined the k-mismatch approximate string matching problem on the basis of Hamming distance, i.e., finding out all the starting positions of strings in a text whose Hamming distances to a pattern are no more than k.

With the explosive growth of the amount of information, traditional sequential algorithms for approximate string matching cannot meet the requirements of real applications. GPU has high parallel processing capability, for the problem of approximate string matching with k mismatches, its parallel algorithms need a lot of threads executing at the same time in order to obtain the best speedup. This is exactly the potential of the GPU.

In this paper, for the problem of k-mismatch approximate string matching, we propose three parallel algorithms based on the model of CUDA. The algorithms match the parallel processing capability of GPU, and balances the workload among the threads in the course of computation. The OPT-block-thread parallel algorithm balances the load among the threads and optimizes the execution time with the memory model of GPU. Experimental results show that compared with the traditional sequential algorithm on CPU, our best parallel algorithm on GPU in this paper achieves speedup of 40–80.

The rest of the paper is organized as follows. In Section II, some related works and development of GPU are introduced. In Section III, we define the problem of k-mismatch approximate string matching, and describe some background information about CUDA. In Section IV, for the problem of k-mismatch approximate string matching, we propose three parallel algorithms on GPU, namely, the thread parallel algorithm, the block-thread parallel algorithm, and the OPT-block-thread parallel algorithm. In Section V, we use real DNA sequences as experimental data in our experimental environment (NVIDIA GeForce GTX260), and the results demonstrate that the algorithms can achieve high speedups. Section VI summarizes the paper.

II. Related Work

For the research of the k-mismatch approximate string matching problem, a lot of sequential algorithms are proposed. Abrahamson, et al, indicated that the Hamming distance of a pattern string at every position of a text string

[1]§To whom correspondences should be addressed. Email: longjiangguo@gmail.com

can be calculated within the time of $O(n\sqrt{m \log m})$ [1], where n is the length of a text, and m is the length of a pattern. Landau and Galil et al proposed two methods can find all the substrings in a text string which differ from a pattern string at most k characters in $O(nk)$ time [2,3]. Landau introduced an algorithm based on a suffix tree, with extra space complexity of $O(n)$ [2]. Cormen, et al, proposed an algorithm based on FFT under the model of RAM, with time complexity of $O(n \log m)$ [4]. Amir, et al proposed an algorithm whose time complexity is $O(n\sqrt{k \log k})$ [5], which is considered as so far the best method.

Kaplan, et al, improved the result of [2] and achieved $O(n)$ space complexity and $O(nk^{\frac{2}{3}} \log^{\frac{1}{3}} m \log k)$ time complexity [6]. In addition, [6] proposed a new problem called *approximate k-mismatch*, by giving a new parameter ε. The problem is to find all the locations j in a text which satisfy the following condition, i.e., the error probability between the substring at j and a pattern is at least $(1-\varepsilon)k$ and at most $(1+\varepsilon)k$. For this new problem, the paper proposed a deterministic algorithm and a randomized algorithm, where the time complexity of the deterministic algorithm is $O((\frac{n}{\varepsilon^2})\sqrt{k}\log^3 m)$, and the time complexity of the randomized algorithm is $O(\frac{n}{\varepsilon^2}\log n \log^3 m \log k)$.

Many researchers have also studied variations of the traditional k-mismatch problem, similar to [6]. However, these problems are essentially still the calculation of Hamming distance. For example, Gabriele, et al, introduced a new parameter γ, and the problem is to find all substrings of a text string and a pattern string of length γ that differ at most k symbols [7]. Linhart and Clifford, et al, were the latest research results for *don't-care*, where the text and pattern strings contain *don't-care* characters which are not to be compared [8,9].

It is noticed that the time complexities for these sequential algorithms for k-mismatch approximate string matching are still high. In real applications, the length of a text is usually very large, so a computation could take a lot of time.

There is need for research on parallel algorithms of k-mismatch approximate string matching. C.Zhong provided a survey of existing parallel algorithms to compute Hamming distance [10]. The authors also proposed two parallel algorithms based on the LARPBS model, where one uses n processors with time complexity $O(m)$, and the other uses mn processors with constant time complexity. Although the algorithms reduce the communication time by using optical buses, they are still based on multiprocessors. The large number of processors makes these algorithms not suitable for actual implementation and real applications.

With the significant advancement and development of GPU, the computing speed of GPU is faster than that of CPU. In particular, NVIDIA has proposed the Compute Unified Device Architecture (CUDA) based on the G80 series in recent years. The parallel threads execution model and the technology of thread synchronization provide a brand-new research perspective and implementation method for enhancing the computation speed. The powerful parallel computing ability of GPU is suitable for large-scale parallel computing, and it has been used in many areas, such as image processing, scientific computing, and physical simulation. For string matching problems, many traditional exact string matching algorithms have achieved great speedup in the GPU [11].

III. Preliminaries

A. Problem Description

Definition 1: **Boolean function** fb. For two arbitrary characters $a, b \in \Sigma$, where Σ is a character set, the boolean function fb is defined as follows:

$$fb(a,b) = \begin{cases} 1, & \text{if } a \neq b; \\ 0, & \text{otherwise.} \end{cases}$$

Definition 2: **Hamming distance**. For string $X = X[0,...,n-1]$ and $Y = Y[0,...,n-1] \in \Sigma^*$, where Σ is a character set, the Hamming distance between X and Y is defined as follows:

$$ham(X,Y) = \sum_{i=0}^{n-1} fb(X[i], Y[i]).$$

Definition 3: **Problem of approximate string matching with k mismatches**. Given a pattern $P = P[0,...,m-1]$ and a text $T = T[0,...,n-1]$, where $m \ll n$, $P, T \in \Sigma^*$ and Σ is a character set, and a positive integer k, $0 \leq k \leq m$, find out all starting locations j in text T, such that $ham(P, T_j) \leq k$, where $T_j = T[j,...,j+m-1]$ and $0 \leq j \leq n-m$.

B. Introduction of CUDA

CUDA is a software and hardware system which treats GPU as a data parallel computing device. It is an entirely new software and hardware frame, where GPU can be considered as a device of parallel data calculation, which can assign and manage computations.

Program codes developed on CUDA can be divided into two types in actual execution. One is the *host code* which runs on CPU, and the other is the *device code* which runs on GPU. A parallel program that runs on GPU is called a *kernel*.

The parallel processing architecture of CUDA, namely, the threads organization architecture, is shown in Fig. 1. Threads of an executing kernel are organized as blocks, and blocks are organized as grids. A block is the execution unit of a kernel. A grid is a collection of blocks which can be executed in parallel. All blocks are executed in parallel. There is no communication and execution order among blocks. Within a block, all threads are also executed in parallel. The same kernel program can be executed in parallel by all the threads of the blocks which are contained in the same grid. Threads in the same block communicate with each other by using the shared memory and synchronize

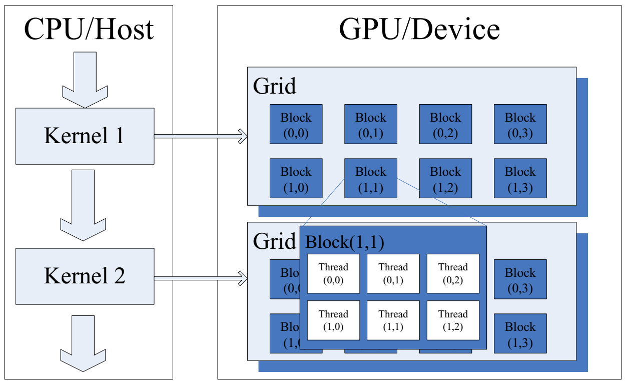


Figure 1. The threads organization architecture of CUDA.

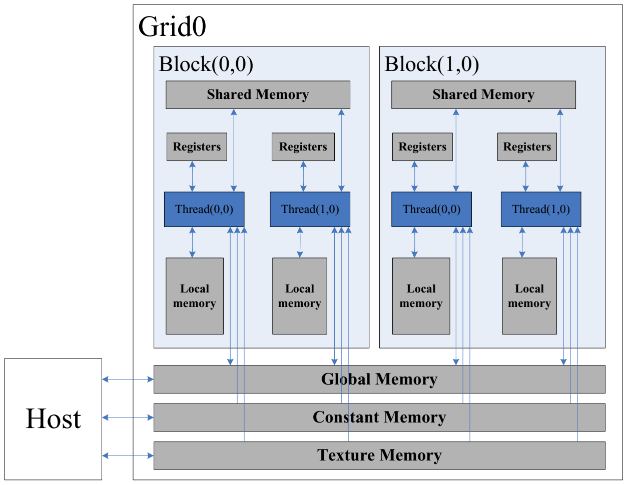


Figure 2. The memory model of CUDA.

by using the `__syncthreads()` instruction. This is the two-level block-thread parallel execution model of CUDA.

The CUDA parallel execution model is actually SIMD, i.e., all threads of a grid execute the same program – a kernel. The basic method of CUDA programming is to exploit and develop thread-level parallelism to the best of its potential. The threads can be dynamically scheduled and executed during runtime.

The memory space of CUDA is divided into registers, local memory, shared memory, global memory, constant memory, and texture memory (see Fig. 2 for illustration). Different memory has different location, accessibility, and lifetime (see Table I for details).

Every thread has its own memory, i.e., registers and local memory, which can be read and written directly. Every block has a shared memory, which can be read and written by the threads in the same block. All the threads in the same grid can access the same global memory. In addition, there are ROM accessible by all threads, i.e., constant memory and texture memory, which can help optimization for different applications.

Table I
THE CLASSIFICATION OF CUDA MEMORY

Memory	Location	Buffer	Access	Speed
Register	GPU	N/A	Device R/W	Fast
Share	GPU	N/A	Device R/W	Fast
Local	On-board Memory	No	Device R/W	Slow
Constant	On-board Memory	Yes	Device R, Host R/W	Faster
Texture	On-board Memory	Yes	Device R, Host R/W	Faster
Global	On-board Memory	No	Device R/W, Host R/W	Slow

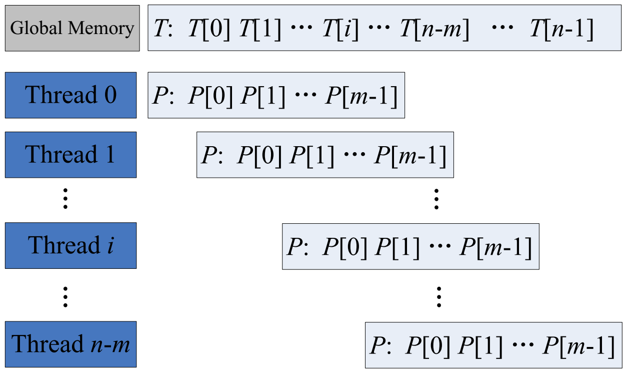


Figure 3. Execution of the thread parallel algorithm.

IV. PARALLEL ALGORITHMS WITH CUDA

In this section, we develop three parallel algorithms, namely, the thread parallel algorithm, the block-thread parallel algorithm, and the OPT-block-thread parallel algorithm.

A. Thread Parallel Algorithm

In this algorithm, there are $n-m+1$ threads. For a given pattern $P = P[0, ..., m-1]$ and a text $T = T[0, ..., n-1]$, $m \ll n$, the Hamming distance between P and the substring $T_i = T[i, ..., i+m-1]$ at location i of text T is calculated by Thread i, where $0 \le i \le n-m$. All threads are executed in parallel. The execution process is illustrated in Fig. 3.

The kernel function is described in Algorithm 1.

```
Algorithm 1 : Thread-Parallel-Kernel ($P$, $T$, $k$, $result[]$)
Require: A pattern $P$, a text $T$ and $k$, $0 \le k \le m$.
Ensure: A result vector $result[]$ of the global memory.
1: $i \leftarrow$ Thread ID;
2: Calculates the Hamming distance between $P$ and $T_i$ as follows:
       $Len \leftarrow ham(P, T_i)$;
       Where $T_i = T[i, ..., i+m-1]$ and $0 \le i \le n-m$.
3: IF $Len > k$, $result[i] \leftarrow -1$;
4: ELSE $result[i] \leftarrow i$.
```

The thread parallel algorithm is described in Algorithm 2.

This algorithm can take full advantage of GPU's powerful parallel processing capability, since multiple threads can

Algorithm 2 : Thread Parallel Algorithm

```
Require: A pattern $P$, a text $T$ and $k$, $0 \le k \le m$.
Ensure: A result vector $result[]$.
        If $ham(P, T_i) > k$, $result[i] = -1$;
        Else, $result[i] = i$.
      Where $T_i = T[i, ..., i+m-1]$ and $0 \le i \le n-m$.
1: Copy the text $T$ and the pattern $P$ from the main memory of the host to the global memory of the device.
2: Transfer the kernel function to start up $n-m+1$ threads.
3: CALL Thread-Parallel-Kernel($P$, $T$, $k$, $result[]$);
4: Copy the result vector $result[]$ from the global memory of the device to the main memory of the host.
```

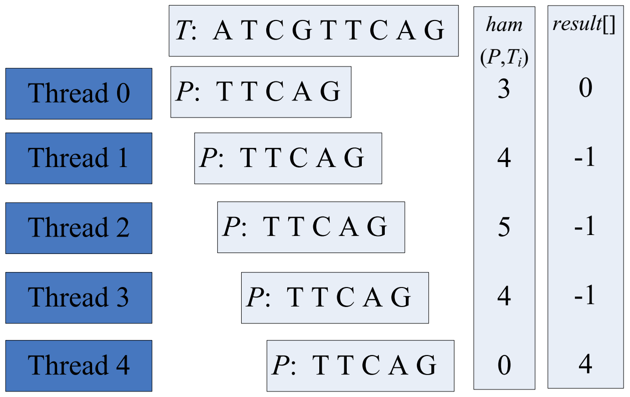


Figure 4. An example of the thread parallel algorithm.

calculate the Hamming distances between the pattern and the substrings of the text at different locations simultaneously. Moreover, it can balance load among threads during the course of calculation, and the execution time of the algorithm is a fixed value when k changes.

Fig. 4 shows an example and results of the algorithm when $T = ATCGTTCAG$, $P = TTCAG$ and $k = 3$. Let us take Thread 0 as an example. Thread 0 calculates the Hamming distance between P and $T_0 = T[0, ..., 4]$. The result is $ham(P, T_0) = 3$, and $3 \le k$, so $result[0] = 0$. According to this method, we can calculate all the elements in $result[]$, i.e., $result[1] = -1, result[2] = -1, result[3] = -1, result[4] = 4$. Here, the vector $result[]$ is stored in the global memory.

B. Block-Thread Parallel Algorithm

By using the two-level block-thread parallel model of CUDA which is introduced in Section III.B, the approximate string matching problem of the paper can be solved by the following parallel algorithm, which includes $n - m + 1$ blocks. The computation of the Hamming distance between $P = P[0, ..., m-1]$ and $T_i = T[i, ..., i+m-1]$ is assigned to Block i, where $0 \le i \le n - m$. There are m threads in each block, and every thread only calculates fb for one symbol. After such calculation, all the threads in the same block work together to sum up their results in parallel [12], i.e., the result is the Hamming distance between pattern P and the substring T_i of text T at location i. The execution process is illustrated in Fig. 5.

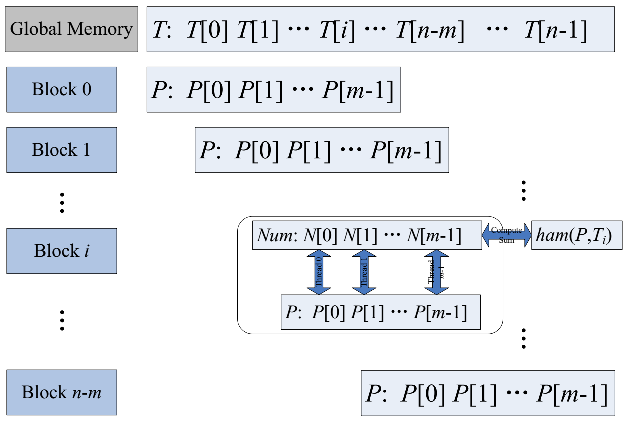


Figure 5. Execution of the block-thread parallel algorithm.

The kernel function is described in Algorithm 3.

Algorithm 3 : Block-Thread-Parallel-Kernel (P, T, k, $result[]$)

```
Require: A pattern $P$, a text $T$ and $k$, $0 \le k \le m$.
Ensure: A result vector $result[]$ of the global memory.
1: $i \leftarrow$ Block ID;
2: $j \leftarrow$ Thread ID in Block $i$;
3: $Num[j] \leftarrow fb(P[j], T[i+j])$;
   $Num[]$ is stored in the shared memory of Block $i$.
4: __syncthreads();
5: Elements in $Num[]$ are summed by all threads in Block $i$, and stored in $Len$;
   $Len$ is the Hamming distance between $P$ and $T_i$, where $T_i = T[i, ..., i+m-1]$ and $0 \le i \le n-m$.
6: IF $Len > k$, $result[i] \leftarrow -1$;
7: ELSE $result[i] \leftarrow i$.
```

The algorithm is described in Algorithm 4.

This algorithm considers the two-level block-thread parallel execution model of CUDA, and employs a lot of blocks in parallel. Threads in the same block are also executed in parallel. This algorithm reduces the workload of every thread and ensures balanced thread load in the course of parallel processing, and the execution time of the algorithm is a fixed value when k changes.

Fig. 6 shows an example and results of the algorithm when $T = ATCGTTCAG$, $P = TTCAG$ and $k = 3$. Let us take Block 0 as an example. Block 0 calculates the Hamming distance between P and $T_0 = T[0, ..., 4]$. Thread j in Block 0 calculates $fb(P[j], T[j])$, and the result is stored into the vector $Num[j]$ of the shared memory, $0 \le j \le 4$. Elements in the vector $Num[]$ are summed up in parallel

Algorithm 4 : Block-Thread Parallel Algorithm

```
Require: A pattern $P$, a text $T$ and $k$, $0 \le k \le m$.
Ensure: A result vector $result[]$.
        If $ham(P, T_i) > k$, $result[i] = -1$;
        Else, $result[i] = i$.
        where $T_i = T[i, ..., i+m-1]$ and $0 \le i \le n-m$.
1: Copy the text $T$ and the pattern $P$ from the main memory of the host to the global memory of the device.
2: Initiate $n - m + 1$ blocks, where every block uses $m$ threads.
3: CALL Block-Thread-Parallel-Kernel($P$, $T$, $k$, $result[]$);
4: Copy the result vector $result[]$ from the global memory of the device to the main memory of the host.
```

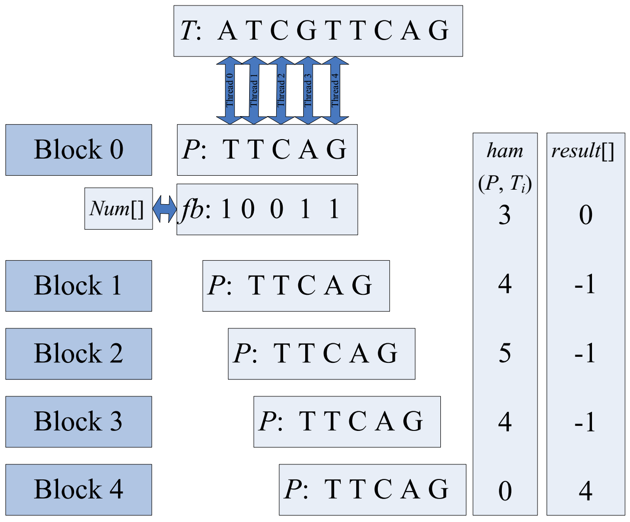


Figure 6. An example of the block-thread parallel algorithm.

by 4 threads in Block 0. The result is $ham(P, T_0) = 3$, and $3 \le k$, so $result[0] = 0$. According to this method, we can calculate all the elements in $result[]$, i.e., $result[1] = -1, result[2] = -1, result[3] = -1, result[4] = 4$.

C. OPT-Block-Thread Parallel Algorithm

In this section, we propose the OPT-block-thread parallel algorithm. Based on the above two parallel methods, this algorithm further explores the memory model of CUDA. While keeping the parallelism, this method accelerates data access speed in every thread by using the shared memory and improves the performance of the algorithm, and the execution time of the algorithm is a fixed value when k changes.

In the pure thread parallel algorithm, the main factor that affects its performance is frequent access of the global memory in every thread during the course of calculation. The speed of data access is the main restriction to performance. In Section III.B, we have introduced the memory model

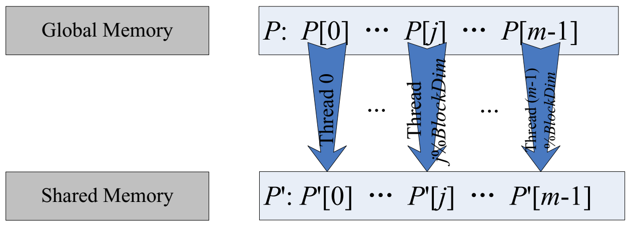


Figure 7. All threads in Block i read pattern P of the global memory into P' of their own shared memory in parallel.

of CUDA, where every block has its own shared memory, and all threads in the same block can access data in the shared memory. Furthermore, the access speed is very high. For the thread parallel algorithm, as every thread works independently, i.e., one thread cannot be affected by other threads when it is working, all threads in the same block can use the shared memory to accelerate the speed of data access. The basic idea can be described as follows. A pattern P and the substring of a text T which the threads in one block need in the course of calculation are read in parallel from the global memory to the shared memory of the block. Therefore, the speed of every thread for memory access in the course of calculation is increased.

A detailed implementation of the method is the following. Assume that every block has $BlockDim$ threads. There are $\lceil (n - m + 1)/BlockDim \rceil$ blocks. Every block uses $BlockDim$ threads, numbered as $0, 1, ..., BlockDim - 1$, to do the computations.

First, every block uses $BlockDim$ threads to read a pattern P of the global memory to P' of its shared memory in parallel. Taking Block i as an example, where $0 \le i \le \lceil (n-m+1)/BlockDim \rceil - 1$. Fig. 7 shows all the threads in Block i that read pattern P into P' in parallel, i.e., Thread (j mod $BlockDim$) in Block i reads the $P[j]$ into $P'[j]$, where $0 \le j \le m - 1$.

Second, because every thread needs to calculate m fb values, the length of the substring of a text T that is used by all the threads in a block is $BlockDim + m - 1$, see Fig. 8. Taking Block i as an example, where $0 \le i \le \lceil (n - m + 1)/BlockDim \rceil - 1$. The substring $T_i = T[i \times BlockDim, ..., i \times BlockDim + BlockDim - 1 + m - 1]$ of the text T of the global memory are read into $T' = T'[0, ..., BlockDim - 1 + m - 1]$ of the shared memory by all threads in Block i in parallel. Fig. 9 shows that $BlockDim$ threads in Block i read the substring T_i of the text T into T' in parallel, i.e., Thread (j mod $BlockDim$) in Block i reads the $T[i \times BlockDim + j]$ into $T'[j]$, where $0 \le j \le BlockDim - 1 + m - 1$. In the course of calculation, the data that the threads in the same block need is stored in the shared memory. This accelerates the speed for access and improves the performance of the algorithm.

Third, after reading the data from the global memory to the shared memory, every thread computes the Hamming

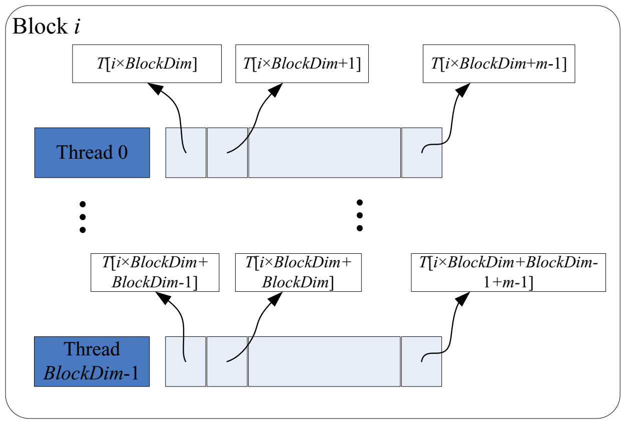

Block *i*
T[*i*×*BlockDim*]
T[*i*×*BlockDim*+1]
T[*i*×*BlockDim*+*m*-1]
Thread 0
T[*i*×*BlockDim*+ *BlockDim*-1]
T[*i*×*BlockDim*+ *BlockDim*]
T[*i*×*BlockDim*+*BlockDim*- 1+*m*-1]
Thread *BlockDim*-1

Figure 8. The size of the substring of text T when all threads in Block i calculate.

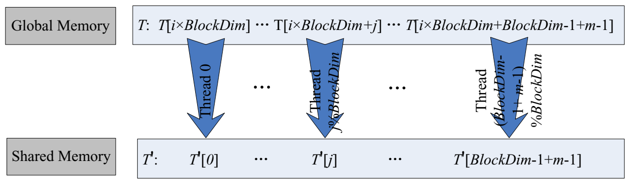

Global Memory
T: *T*[*i*×*BlockDim*] ··· T[*i*×*BlockDim*+*j*] ··· *T*[*i*×*BlockDim*+*BlockDim*-1+*m*-1]
Thread 0
Thread *j*%*BlockDim*
Thread (*BlockDim*-1+*m*-1) %*BlockDim*
Shared Memory
T': *T*'[*0*] ··· *T*'[*j*] ··· *T*'[*BlockDim*-1+*m*-1]

Figure 9. All threads in Block i read text T from the global memory into their own shared memory in parallel.

distance between one substring of the text T' and the pattern P', and stores the results to the vector $result[]$ of the global memory. The execution process of Block i is shown in Fig. 10, where Thread j calculates the Hamming distance between pattern P' and the substring $T'_j = T'[j, ..., j+m-1]$ of text T', for all $0 \le j \le BlockDim - 1$.

The kernel function is described in Algorithm 5.

The algorithm is described in Algorithm 6.

Fig. 11 shows an example and results of the algorithm when $T = ATCGTTCAGCA$, $P = TTCA$, $BlockDim = 4$ and $k = 2$. Since $n = 11$ and $m = 4$, it needs $\lceil (n-m+1)/BlockDim \rceil = 2$ blocks. Let us take Block 0 as an example. Assume the data have been read from the global memory to the shared memory. The four threads in Block 0 calculate the Hamming distances at all locations of

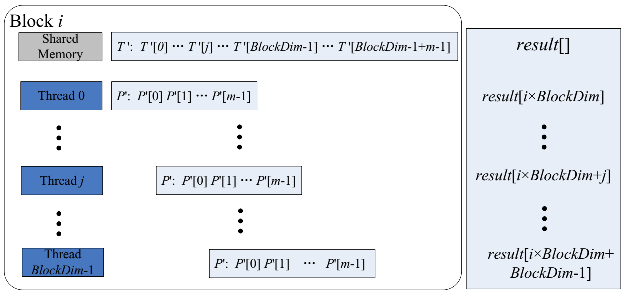

Block *i*
Shared Memory
T': *T*'[*0*] ··· *T*'[*j*] ··· *T*'[*BlockDim*-1] ··· *T*'[*BlockDim*-1+*m*-1]
Thread 0
P': *P*'[0] *P*'[1] ··· *P*'[*m*-1]
Thread *j*
P': *P*'[0] *P*'[1] ··· *P*'[*m*-1]
Thread *BlockDim*-1
P': *P*'[0] *P*'[1] ··· *P*'[*m*-1]
result[]
result[*i*×*BlockDim*]
result[*i*×*BlockDim*+*j*]
result[*i*×*BlockDim*+ *BlockDim*-1]

Figure 10. Execution of the OPT-block-thread parallel algorithm.

Algorithm 5 : OPT-Block-Thread-Parallel-Kernel (P', T', k, $result[]$)

```
Require: A pattern $P'$, a text $T'$ and $k$, $0 \le k \le m$.
Ensure: A result vector $result[]$ of the global memory.
1: $i \leftarrow$ Block ID;
2: $j \leftarrow$ Thread ID in all threads;
3: Calculates the Hamming distance between $P'$ and $T'_j$ as follows:
       $Len \leftarrow ham(P', T'_j)$;
       Where $T'_j = T'[j, ..., j+m-1]$ and $0 \le j \le BlockDim - 1$.
4: IF $Len > k$, $result[l] \leftarrow -1$;
5: ELSE $result[l] \leftarrow l$.
       Where $l = i \times BlockDim + j$.
```

Algorithm 6 : OPT-Block-Thread Parallel Algorithm

```
Require: A pattern $P$, a text $T$ and $k$, $0 \le k \le m$.
Ensure: A result vector $result[]$.
       If $ham(P, T_i) > k$, $result[i] = -1$;
       Else, $result[i] = i$.
       where $T_i = T[i, ..., i+m-1]$ and $0 \le i \le n-m$.
1: Copy the text $T$ and the pattern $P$ from the main memory of the host to the global memory of the device.
2: Read the pattern $P$ of the global memory to $P'$ of the shared memory of Block $i$ in parallel using $BlockDim$ threads in Block $i$, for all $0 \le i \le \lceil (n-m+1)/BlockDim \rceil - 1$.
3: Read substring $T_i = T[i \times BlockDim, ..., i \times BlockDim + BlockDim - 1 + m - 1]$ of the global memory to $T' = T'[0, ..., BlockDim - 1 + m - 1]$ of the shared memory of Block $i$ in parallel by using $BlockDim$ threads in Block $i$, for all $0 \le i \le \lceil (n-m+1)/BlockDim \rceil - 1$.
4: CALL OPT-Block-Thread-Parallel-Kernel($P'$, $T'$, $k$, $result[]$);
5: Copy the result vector $result[]$ from the global memory of the device to the main memory of the host.
```

T' in parallel. Thread 0 calculates the Hamming distance between P' and $T'_0 = T'[0, ..., 4]$, the result $ham(P', T'_0) = 2$, and $2 \le k$, so $result[0] = 0$. According to this method, we can calculate all the elements in $result[]$, i.e., $result[1] = -1, result[2] = -1, result[3] = -1, result[4] = 4, result[5] = -1, result[6] = -1, result[7] = 7$.

V. Experimental Results

The experiment environment of our work is described as follows. The video card is NVIDIA GeForce GTX260, whose computing ability is 1.3, and it has 27 stream multiprocessors, i.e., "compute-core". The host is a dual-core E7500 processor with 2.93 GHz dominant frequency. The operation system is Windows XP. We use actual

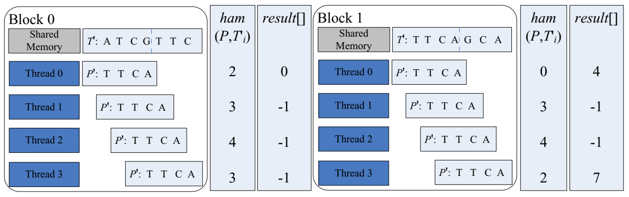


Figure 11. An example of the OPT-block-thread parallel algorithm.

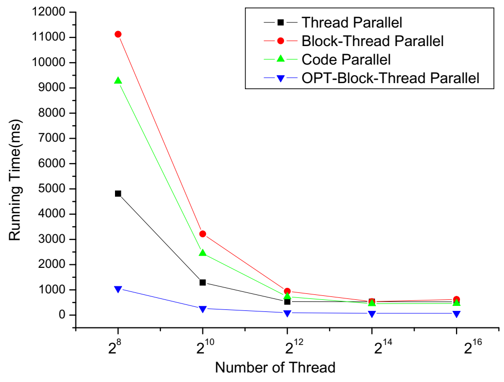


Figure 12. Execution times of the GPU algorithms versus the number of threads.

DNA sequences as experimental data, i.e., B.anthracis and Yersinia_pestis($http://huref.jcvi.org/$).

In addition to implement the algorithms developed in this paper, we also parallelize and implement the code parallel algorithm on CUDA, which is considered to be the best one on CPU [5]. It is noticed that although the algorithm is much better than other approximate string matching methods in theory, our experimental results show that the performance of this method is not satisfactory. The reason is that the method needs to call the kernel function over and over again and to exchange data between the main memory and the video memory again and again, causing large amount of extra cost.

Fig. 12 shows the execution times of the GPU algorithms versus the number of threads. It is observed that when the number of threads is increased to certain amount, the execution time of an algorithm will not change any more. The reason is that the number of threads which can run at the same time is limited, which is related to a specific GPU. When the number of threads is increased to the maximum number of threads that can operate at the same time on a GPU, more threads will not increase the execution speed of an algorithm.

Fig. 13 demonstrates the execution times of the GPU algorithms versus the pattern size, where the text string is the DNA sequence of B.anthracis. It is seen that the performance of the OPT-block-thread parallel algorithm is much better

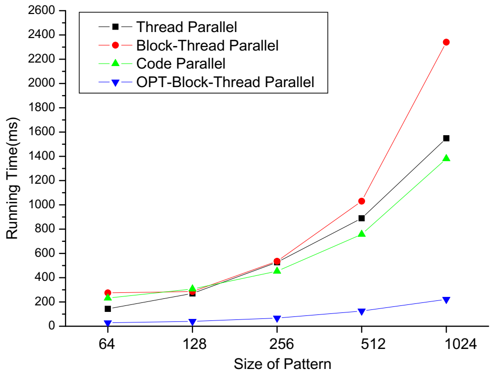


Figure 13. Execution times of the GPU algorithms versus the pattern size.

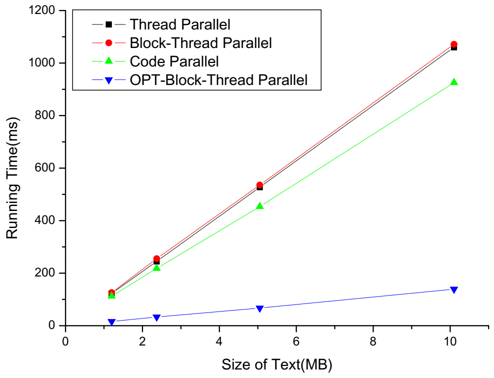


Figure 14. Execution times of the GPU algorithms versus the text size.

than others. The Block-thread parallel algorithm is not ideal, it is because that in the algorithm, every thread does too little work, and the cost to setup a thread spend much time, so it is not worth setting a thread which takes too little workload in parallel algorithm. It is clear that the optimization performed in the algorithm by using the shared memory reduces the time of frequent data access, which is the main factor that affects the performance of the algorithm.

Fig. 14 displays the execution times of the GPU algorithms versus the text size, where the pattern size is 256 characters. It can be seen that the execution times of all algorithms increase linearly as the text size increases. The two algorithms based on thread level parallelism and block-thread parallelism have nearly the same execution time, while the OPT-block-thread parallel algorithm is much better than others.

Fig. 15 shows the speedups of the GPU algorithms compared to the simple CPU algorithm, where the text strings are the DNA sequences of Yersinia_pestis. It is noticed that in our experimental environment, the speedup of the OPT-block-thread parallel algorithm can achieve speedup of 40–80. A longer pattern yields greater speedup. All other algorithms on the GPU can also get speedups of almost 10.

VI. Conclusion

For the problem of approximate string matching with k mismatches, this paper presents three parallel algorithms on GPU, and implements parallel computation for calculation of Hamming distances. The algorithms all take advantage of the parallel capability of GPU. Moreover, workload can be balanced in the course of computation. In the OPT-block-thread parallel algorithm, we use the shared memory to reduce the data access time of threads to the global memory, by using the memory model of GPU under the structure of CUDA, and make the algorithm to achieve the optimal performance. We use real DNA sequences as experimental data in this paper. Our results show that the algorithms have achieved favorable speedup.

Acknowledgment

This work is supported by Programs Foundation of Ministry of Education of China for New Century Excellent Talents in University(NCET-11-0955), Program for Group of Science and Technology Innovation of Heilongjiang Educational Committee under grant No.2011PYTD002, the Science and Technology Research of Heilongjiang Educational Committee under grant No.12511395, Programs Foundation of Heilongjiang Educational Committee for New Century Excellent Talents in University, the Science and Technology Innovation Research Project of Harbin for Young Scholar under grant No.2008RFQXG107 and No. 2011RFXXG014, the National Natural Science Foundation of China for Young Scholar under grant No.60803015, 61070193, Heilongjiang Province Founds for Distinguished Young Scientists under Grant No.JC201104, Heilongjiang Province Science and Technique Foundation under Grant No.GC09A109.

References

[1] Abrahamson, K., Generalized string matching, *SIAM Journal on Computing*, vol. 16, pp. 1039-1051, 1987.

[2] Landau, G. M. and U. Vishkin, Efficient string matching in the presence of errors, *26th IEEE Symposium on Foundations of Computer Science*, pp. 126-136, 1985.

[3] Galil, Z. and R. Giancarlo, Improved string matching with k mismatches, *SIGACT News*, vol. 17, pp. 52-54, 1986.

[4] Cormen, T. H., C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, *MIT Press*, 1990.

[5] Amir, A., M. Lewenstein, and E. Porat, Faster algorithms for string matching with k-mismatches, *Algorithms*, vol. 50, pp. 257-275, 2004.

[6] Kaplan, H., E. Porat, and N. Shafrir, Finding the position of the k-mismatch and approximate tandem repeats, manuscript, 2006.

[7] Gabriele, A., F. Mignosi, A. Restivo, and M. Sciortino, Approximate string matching: indexing and the k-mismatch problem, manuscript, 2005.

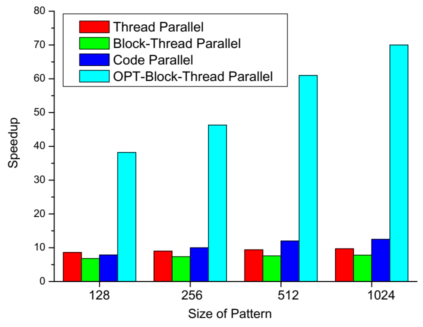
[8] Linhart, C. and R. Shamir, Matching with don't-cares and a small number of mismatches, *Information Processing Letters*, vol. 109, no. 5, pp. 273-277, 2009.

[9] Clifford, R. and E. Porat, A filtering algorithm for k-mismatch with don‘t cares, *Information Processing Letters*, vol. 110, no. 22, pp. 1021-1025, 2010.

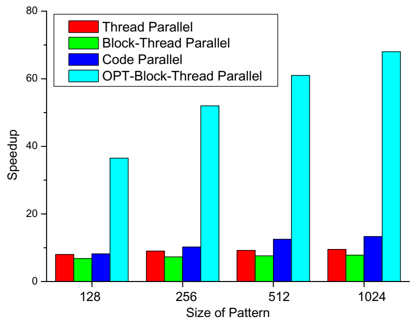
[10] Zhong, C. and G.-L. Chen, Parallel algorithms for approximate string matching on PRAM and LARPBS, *Journal of software*, vol. 15, no. 2, pp. 159-169, 2004.

[11] Kouzinopoulos, C. S. and K. G. Margaritis, String matching on a multicore GPU using CUDA, *13th Panhellenic Conference on Informatics*, 2009.

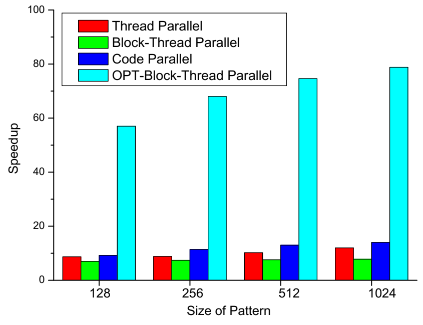
[12] Mark Harris, Parallel Prefix Sum(Scan) with CUDA, $http://developer.download.nvidia.com/$, 2007.



(a) text NC010157



(b) text NC010158



(c) text NC010159

Figure 15. Speedups of the GPU algorithms for the texts of various DNA sequences.