Implementation of String Match Algorithm BMH on GPU Using CUDA

Junrui Zhou*, Hong An, Xiaomei Li, Min Xu, Wei Zhou

Advanced Computer System Architecture research group, University of Science and Technology of China, Huangshan road, 230027 Hefei, China

Abstract

String match algorithm is widely used in the area of data mining. In this paper, we present an approach for elevating the performance of this algorithm via GPU (Graphic Processing Unit). With the rapid development of Graphics Processing Unit to many-core multiprocessors, it shows great potential in many applications and high performance computing. Especially, the heterogeneous architecture CPU+GPU shows enthusiastic capacity to accelerate parallel applications. Till now, some research has been done for parallel implementation of string match algorithm on GPU. But there are some constraints for acceleration of this algorithm: firstly, if-branch exists in the algorithm which can cripple the performance of the parallel implementation; secondly, improper memory access may bring heavy latency generated by bank-conflict. This paper presents the optimization of global memory access, shared memory access and the elimination of if-branch for elevating the performance of string match algorithm BMH. Meanwhile, we explore the interesting characteristics of the parallel BMH string match algorithm implemented on GPU using CUDA. We found that the performance elevation is affected by the characteristic of pattern, which also proves the potential of GPU.

© 2011 Published by Elsevier Ltd. Selection and/or peer-review under responsibility of Singapore Institute of Electronics

Keywords: BMH, string match, parallel computing, GPU, CUDA

1. Introduction

The Boyer-Moore-Horspool algorithm was chosen since it involves sequential accesses to the global memory, which can cut down the overhead of memory access as well as this algorithm is more effective

* Corresponding author. Tel.:+86 (0)551 360 0839.
E-mail address: junrui09@mail.ustc.edu.cn.
than some other string match algorithm [1].

The NVIDIA GPU behaves perfectly on computation-intensive application [10]. The minimal granularity of execution in GPU is threads in a half-warp, which are grouped into a number of blocks that in turn are grouped into grids. All the blocks that belong to a grid can be executed simultaneously and those threads are automatically managed by a hardware thread manager. The number of blocks and the number of threads per block can be specified by the programmer within the limit of the valid number constrained by the GPU.

To exploit the performance of applications implemented on GPU, how to use the memory on GPU and transform the structure of the algorithm should be firstly taken into account.

In this paper, the performance of the Boyer-Moore-Horspool string matching algorithm is evaluated using CUDA with attention on how the GPU utilization and the memory usage affect the performance.

This paper is organized as follows: Section 2 presents some related work. Section 3 gives a brief overview of the Boyer-Moore-Horspool string matching algorithm and the efficient optimization for memory access and program structure. Section 4 shows and analyses the experimental results. Section 5 concludes the paper and presents future works.

2. Related work

2.1. BMH serial algorithm

Table 1. BMH serial algorithm

| skip[1..256]=N; |
|---|---|
| for i to N | skip[P[i]]=N-i; |
| while i<=N | |
| k=i; | |
| for j=M to 1 | |
| if P[j]!=T[k] | break; |
| else | k--; j--; |
| if j=-1 | |
| match; | i++; |
| else | |
| mismatch; | |
| i+=skip[T[i]]; |

The string matching problem can be defined as: let \( \Sigma \) be an alphabet, given a text array \( T[N] \) and a pattern array \( P[M] \) where \( M \) is the length of the pattern and \( N \) is the length of the text, report all locations \( i \) in \( T \) where is an occurrence of \( P \), \( T[i + k]=P[k] \) for \( M \leq N \). Table 1 illustrates serial BMH string match algorithm. The Boyer-Moore-Horspool algorithm [1] uses a window of size \( M \) to search the text from right to left and an array (known as the bad-character shift) of the rightmost character of the window to skip character positions when a mismatch occurs.
2.2. GPU-Based String Matching

Since its release, CUDA (Compute Unified Device Architecture) has been used across many areas of computing including applied mathematics [3], sequence alignment [4], astrophysical simulations [5] and image processing [8], and so on. Using GPU to perform string matching computations has also been studied: the Cg programming language is used in [2] to offload to the GPU an Intrusion Detection System that uses a simplified version of the Knuth-Morris-Pratt algorithm, reporting a marginal improvement; the Aho-Corasick, Knuth-Morris-Pratt and Boyer-Moore algorithms were implemented in [9] using CUDA to perform Intrusion Detection with a reported increase of their performance by up to an order of magnitude; a speedup of up to 35x was achieved in [7] when performing string matching on sets of genome and chromosome sequences. A detailed survey of many scientific applications that use the GPGPU model can be found in [6].

In the field of network intrusion detection (NID), Giorgos Vasiliadis et al. used the GPU as a pattern match engine and presented a Snort based prototype system Gnort [13]. Gnort ported the Aho-Corasick algorithm [11] to run on GPU. The algorithm iterates through all the bytes of the input stream and moves the current state to the next correct state using a state machine that has been previously constructed during initialization phase.

Then in computational biology, Michael C. Schatz and Cole Trapnell presented a GPU-based exact string-matching program named Cmatch [12]. The program’s string-matching kernel executes parallelized searching of string matching algorithms using sophisticated data structures called suffix trees. As a result, with using in CUDA, Cmatch achieves a speedup as much as 35x on NVIDIA GTX 8800 over the equivalent serial CPU-bound version.

3. Implementation of BMH algorithm on GPU using CUDA

3.1. Store Strategy

![Fig. 1. Each stage of the store strategy for the text to be matched](image)

Figure 1 shows the store scheme for the text to be matched. In the first step of the execution of the algorithm on the GPU, the text is loaded to the global memory from host memory; then each block transfer segment of the text to its shared memory by every thread inside the block. Meanwhile, the pattern and skip arrays are transferred to constant Memory inside GPU to reduce the access latency.

3.2. Kernel of BMH algorithm on GPU

The input and output of this kernel are data to be processed and match results respectively.
Each segment of text named B_text is transferred into shared memory in each block and is in turn broken into some pieces named T_text in equivalent length which is assigned to each thread within one block.

To find all the locations where an occurrence happens, we redundantly copy the chars at the end of each substring with a length of M-1 and append them to the beginning of the Block_text processed by the next block, as shown in figure 2 where substring represent a segment of text. This redundant copy is adapted to Block_text when it is broken into Thread_text processed by each thread. Supposing the amount of block is Block_num and the size of shared memory is SM_size, the relation between Block_num and SM_size is as (1).

\[ SM_size = \frac{N}{B_{num}} + (M - 1) \]  

(1)

In the kernel each thread processes Thread_text concurrently and the length of Thread_text is called T_size. Supposing the number of threads in each block is named as B_size, the relation between S_size and T_size is as (2).

\[ T_{size} = \frac{SM_{size}}{B_{size}} + (M - 1) \]  

(2)

There are two main steps in the kernel: Firstly, each part of the text is copied from global memory to shared memory; secondly, each thread executes the BMH match algorithm in parallelism and find all locations where an occurrence of pattern happens.

3.3. Bank-conflict free solution

According to [13], bank-conflict which means at least two threads access the same bank of shared memory may be generated by the access interval of each adjacent thread in a half-warp. The size of half-warp in CUDA is fixed to 16. As depicted in figure 3, ninth thread access bank 1 of share memory which is accessed by first thread in the same half-warp simultaneously. Under this situation, a bank conflict occurs at bank 1. As we know, the bank conflict brings the latency of memory access. So the choice of a suitable access interval for adjacent threads is desirable. In fact, the access interval is equivalent to T_size which presents the length of the text access by each thread.
To avoid bank-conflict problem generated by the access of B_text stored in Shared memory, we use the conflict detection function presented in table 2 to select a proper value of T_size. The way to detect bank-conflict is to find out whether the bank location in shared memory is accessed by at least two threads at the same time within one half-warp. If that situation is not found, the T_size will not create the bank-conflict.

Table 2. The way used to detect bank conflict

<table>
<thead>
<tr>
<th>Conflict detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>int size=size of half-warp;</td>
</tr>
<tr>
<td>int c_size=sizeofchar;</td>
</tr>
<tr>
<td>int *map=new int[size];</td>
</tr>
<tr>
<td>for x=0 to T_size</td>
</tr>
<tr>
<td>for y=0 to size</td>
</tr>
<tr>
<td>loc= ((c_size<em>T_size+y+x</em>c_size)/bank_size)%size;</td>
</tr>
<tr>
<td>map[loc]++;</td>
</tr>
<tr>
<td>if map[loc] is more than 1</td>
</tr>
<tr>
<td>return bank conflict;</td>
</tr>
<tr>
<td>return bank conflict free;</td>
</tr>
</tbody>
</table>

Besides, through the two formula (1) and (2) listed before, we can conclude that T_size is inversely proportional to the total amount of threads used to process the text. Since the concurrency depends on the number of the threads, the performance has an indirect relation with T_size too. The parallelism increases with the growth of the amount of threads, however, more redundant copy is needed, which makes more global memory access. So T_size plays an irreplaceable role in the balance between concurrency and overhead of global memory access.
3.4. Global memory access optimization

Table 3. Different global memory access used in data copy from global memory to shared memory

<table>
<thead>
<tr>
<th>contiguous access</th>
<th>non-contiguous access</th>
</tr>
</thead>
<tbody>
<tr>
<td>beg=blockIdx.x<em>blockDim.x</em>(T_size-M+1)</td>
<td>beg=blockIdx.x<em>blockDim.x</em>(T_size-M+1)</td>
</tr>
<tr>
<td>for i=0 to T_size-M+1</td>
<td>for i=0 to T_size-M+1</td>
</tr>
<tr>
<td>sloc= Block_size*i+ThreadIdx.x;</td>
<td>sloc=blockIdx.x*(T_size-M+1)+i;</td>
</tr>
<tr>
<td>gloc=beg+i*block_size +threadIdx.x;</td>
<td>gloc=beg+threadIdx.x*(T_size-M+1)+i;</td>
</tr>
<tr>
<td>B_text[sloc]=text[gloc];</td>
<td>B_text[sloc]=text[gloc];</td>
</tr>
</tbody>
</table>

In the first step of BMH kernel, all characters in the text are copied from global memory to shared memory. In our implementation, we adopt both non-contiguous and contiguous global memory access to compare their bandwidth. And those two access model are listed in table 3. Each thread copies each character continuously at the same time in the contiguous access way; on the contrary, each thread copies each character with an interval equal to T_size in the non-contiguous access way.

In the two access model, contiguous access copies adjacent characters to B_text by threads in a half-warp simultaneously while non-contiguous access copies characters with an interval to B_text by threads in a half-warp simultaneously.

In the GPU device, the global memory space is not cached, so the right access pattern should be adopted to get maximum memory bandwidth [14]. Global memory bandwidth is used most efficiently when the simultaneous memory accessed by threads in a half-warp (during the execution of a single read or write instruction) can be coalesced into a single memory transaction. Just only under the situation of the words accessed by all the threads in a half-warp lie in the same segment of size equal to 32 bytes because only char type words are accessed from global memory. According to the way that the global memory access is processed by the GPU device, the contiguous access can be coalesced into a single memory transaction. Therefore, the contiguous access way is a better choice.

3.5. Elimination of if-branch in kernel

As we know, the mechanism of GPU processing if-branch is to execute each thread of one half-warp one by one serially. No doubt that manner cripples the concurrency of the kernel.

The branch elimination can be achieved by converting the branch into numerical calculation.

4. Experiment and results

4.1. Experimental Methodology

The BMH algorithm is realized on GPU using CUDA version 3.0. The experiments were executed on an Intel Xeon CPU with a 2.40GHz clock speed and 2GB of memory. The Graphics Processing Unit used was the NVIDIA GTX 275. To decrease random variation, the time results were average of 1000 runs.

The data set is a randomly generated text which is composed of abundant characters in the range of Unicode, and its size reaches to 1 MB.

Practical running time is the total time the algorithm needs to find all occurrences of a pattern in a specified text including the time used by the kernel execution and the data transfer time between the host and the GPU. All data was transferred in one batch to the global memory of the GPU and the pattern is transferred into the constant memory.

Besides, to get the speedup of the algorithm under different frequency of pattern in the specified text, we adjusted the frequency of the pattern in the text.
4.2. Results

Table 4. The four different task partitions under the situation of that n and Block_num are equal to 1536000 and 160 respectively

<table>
<thead>
<tr>
<th>partition</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block_size</td>
<td>480</td>
<td>240</td>
<td>160</td>
<td>120</td>
</tr>
</tbody>
</table>

Different task partitions used in our experiments are depicted in table 4 where B_size represents the number of thread in each block and the number of block used in the kernel is fixed. The B_size decreases gradually from partition 1 to partition 4. In other words, the number of thread decreases gradually from partition 1 to partition 4.

![Graph showing execution time of each kernel with different pattern length under each task partition](image)

Fig. 4. The execution time of each kernel with different pattern length under each task partition

Figure 4 shows running time of each kernel under different task partitions where m represents the length of pattern to be matched in the text. The performance of all kernels with different pattern length decreases as B_size decreases, which is due to that the total number of thread cut down when the B_size decreases as the Block_num is fixed. As we know, threads can process the kernel in parallelism, which means more threads usually elevate the performance. The time used to execute the kernel with m equal to 80 shrinks heavily from partition 4 to partition 1, which implies that a suitable task partition should be chosen to execute the kernel.

The BMH algorithm needs to determine whether the alphabet in the text is the same as the alphabet in pattern more frequently when the appearance of pattern in the text increases. So the execution time of serial BMH algorithm increases along with the frequency of appearance of pattern in the text. However, we can have an assumption that the addition of appearance of pattern affects the performance of parallel BMH algorithm much less since the influence is amortized by abundant threads.

To prove the assumption, we adjust the amount of the appearance of the pattern in the same text automatically via a program which replaces original string in text with the pattern at a different interval. Though this augment via interval make the appearance of pattern in text distributed evenly and a bit of ideal, it is very representative. Then, we run the parallel algorithm on each augmented text with the best task partition and the execution time is normalized to the time used by serial algorithm executed on each text. We adopt 1536000 and 20 as the length of text and pattern respectively with the best task partition 1 depicted in table 4. The speedup of parallel version over serial version is acquired for five different frequency of the appearance of pattern at a range from 1 percent to 5 percent.
From figure 5, we can see that the speedup of the parallel version increases linearly as the frequency of appearance of pattern in the text grows. This behavior of acceleration indicates the potential of GPU accelerator.

Theoretically, the performance of parallel version should be much more significant than the peak speedup archived in our experiment. To prompt the performance we augment the kernel through the replacement of global memory access model and the elimination of if-branch. The effect can be seen from figure 6, in which each optimized kernel performs better than original kernel.

The data transfer between GPU accelerator and host becomes a sever bottleneck as the size of processed text increases. We use device to represent GPU and kernel to represent inner of GPU. As depicted in figure 7, the percentage of data transfer between device and host increases from about thirty percent to fifty percent with the growth of size of text from 15360 to 1536000. That is due to the lower bandwidth between device and host than it of data transfer inside device. To get rid of this disadvantage, transferring data from host to device in stream might be the right solution.

5. Conclusion

In this paper, parallel implementation of Boyer-Moore-Horspool algorithm is presented using the CUDA toolkit. Comparing serial implementation with the parallel implementations is in terms of running
time. It is shown that the parallel implementation of the algorithms is at least 40 times faster than the serial implementation, especially when there is a vary occurrence of pattern in text. In order to achieve peak performance using a GPU, it is discussed that the hardware must be as fully utilized as possible, such as the shared memory should be used to take advantage of its very low latency as well as the bank conflict inside shared memory is should be avoided via the choice of proper access interval for the adjacent threads in the same half-warp. Besides, the global memory access should be coalesced to cut down the latency.

Further optimization of the parallel implementation of the algorithms could be considered to make better use of the GPU capability, such as loop unrolling in addition to smarter use of the shared memory and global memory.

At present, the choice of task partition is implemented by hand. To determine which partition is better, we should try a vary value of B_size and Block_num and compare the performance with each other partition, which is a waste of time. Therefore exploring the best partition automatically is desirable through a heuristic algorithm taking these factors of text length, pattern length and the occurrence frequency of pattern into account.

Acknowledgment

This work is supported financially by the National Basic Research Program of China under contract 2011CB302501, the National Natural Science Foundation of China grants 60633040 and 60970023, the National Hi-tech Research and Development Program of China under contracts 2009AA01Z106, the National Science & Technology Major Projects 2009ZX01036-001-002.

Reference