A GPGPU Implementation of Approximate String Matching with Regular Expression Operators and Comparison with Its FPGA Implementation

Yuichiro Utan, Masato Inagi, Shin'ichi Wakabayashi, and Shinobu Nagayama
Graduate School of Information Sciences, Hiroshima City University, Hiroshima, Japan

Abstract— *In this paper, we propose an efficient GPGPU implementation of an algorithm for approximate string matching with regular expression operators, originally implemented on an FPGA, and compare the GPGPU, FPGA and CPU implementations by experiments. Approximate string matching with regular expression operators is used in various applications, such as full text database search and DNA sequence analysis. To efficiently handle a long text in the matching, a hardware algorithm for FPGA implementation has been proposed. However, due to the limitation of FPGAs' capacity, it cannot handle long patterns. In contrast, our proposed GPGPU implementation is able to handle long patterns efficiently, utilizing the scalability of GPGPU programming. Experimental results showed that the GPU implementation is more than 18 times as fast as the CPU one when the pattern length is greater than 3200, while the FPGA one could not handle such a long pattern.*

Keywords: approximate string matching, regular expression, GPGPU, FPGA, CUDA

1. Introduction

Approximate string matching [1] is the problem to find substrings in a given string (*text*) which are similar to another given string, called a *pattern*. The degree of similarity between two strings, called *edit distance*, is obtained by deriving the *edit distance matrix*. Approximate string matching is one of the major problems in information science, and used in keyword search in databases, DNA sequence analysis in bioinformatics, and network intrusion detection, *etc*. Since the problem size is exponentially increasing, some efficient algorithms using special hardware [2] or a graphics processing unit (GPU) [3] have been proposed.

Although approximate string matching is more flexible than exact string matching, both kinds of matching deal with only simple patterns which consist of only alphabet characters. Such simple descriptions lead unacceptably long patterns in some applications in which the target substrings vary under certain rules (*e.g.*, network intrusion detection). Thus, matching that can deal with more efficiently described patterns is being required. One of the methods for efficiently describing patterns is regular expression. In this paper, we call a variant of approximate string matching in which patterns can include some regular expression operators, *approximate regular expression matching*. A systolic hardware algorithm for approximate regular expression matching has already been proposed and implemented on an FPGA in [4]. However, in this algorithm, the acceptable pattern length is limited by the amount of hardware resources (*i.e.*, the FPGA's capacity), and thus only short patterns can be handled.

Recently, parallel processing methods using GPUs are attracting attention. GPUs are originally application specific processors for graphics processing. However, since GPUs have highly parallel architectures to render pixel images in real-time, general-purpose computation on GPUs (GPGPU) [5] has been actively being studied to utilize GPUs' high performance. As a development environment for GPGPU, compute unified device architecture (CUDA) is provided by NVIDIA Corporation to facilitate the utilization of GPUs for general-purpose computing. GPGPU is based on homogeneous multithreading, and threads more than hardware resources (*e.g.*, ALUs) are automatically scheduled and allocated to the resources. Thus, GPGPU programming is scalable in terms of the number of threads.

Utilizing the high parallelism of GPUs, a GPGPU implementation of approximate string matching has been proposed in [3]. This method enhances the degree of data parallelism by handling multiple texts in parallel. Thus, it cannot fully utilize a GPU's parallelism when performing matching with a single text and a long pattern.

In this paper, we propose an efficient GPGPU implementation of approximate regular expression matching for long patterns. Our method is based on the FPGA implementation [4]. Comparing to the FPGA implementation, the main advantages of the GPGPU implementation are:

1) our method can handle much longer patterns, and
2) it requires no special hardware, like FPGAs.

The main differences of our proposed method from [3] are:

1) it can handle some regular expression operators, and
2) it is suitable to high-speed matching with a single text.

The method proposed in [3] divides the edit distance matrix into parallelogram regions. In the method, when only a single text is given, up to 32 elements in a region are calculated in parallel, and the regions are sequentially handled. Our

method divides the matrix into parallelogram regions in the same way. In our method, however, up to 32 elements in a region are calculated in parallel, and up to 16 regions are calculated in parallel, considering the data dependencies among regions. (The degrees of parallelism depend on the GPU.) This makes our method more efficient for matching with a single text than [3].

In addition, we evaluate the GPGPU implementation comparing to the FPGA implementation by experiments. The experimental results showed that the FPGA and GPU implementations were 8.3 and 2.9 times as fast as a CPU implementation when the pattern length is 320, respectively. Furthermore, the GPU implementation was more than 18 times as fast as the CPU one when the pattern length is greater than 3200, while the FPGA one could not handle such a long pattern.

The rest of this paper is organized as follows. In Section 2, the definitions of approximate string matching and approximate regular expression matching are described, and GPGPU is explained. Section 3 shortly describes the existing FPGA implementation of approximate regular expression matching [4]. Section 4 presents our GPGPU implementation of approximate regular expression matching. In Section 5, we compare the GPGPU, FPGA and CPU implementations by experiments. Finally, conclusions are given in Section 6.

2. Preliminaries

2.1 Approximate String Matching

Here, we define the approximate string matching problem and explain an algorithm for the problem. Given two strings P (pattern) and T (text), and a non-zero integer k (threshold), the approximate string matching problem is to find a substring of T whose edit distance [6] from P is less than or equal to k. Now, let us consider transforming a string S_1 to another string S_2 by iteratively applying single-character deletions, insertions and substitutions. When the costs of deletions, insertions and substitutions are given, the edit distance between S_1 and S_2 is the minimum total cost required to transform S_1 to S_2. Thus, the calculation of edit distance is essential to the approximate string matching problem.

Next, we explain how to calculate the edit distance. The edit distance between S_1 and S_2 is calculated as $D(m, n)$ defined in the following by using dynamic programming (DP), where $m = |S_1|$ and $n = |S_2|$.

$$D(i,j) = \min\{ \begin{array}{l} D(i-1,j) + \text{del}, \\ D(i,j-1) + \text{ins}, \\ D(i-1,j-1) + s(i,j) \end{array} \}, \quad (1)$$

where

$$s(i,j) = \begin{cases} \text{sub}(S_1[i], S_2[j]) & 1 \le i \le m, 1 \le j \le n \\ \infty & \text{otherwise,} \end{cases}$$

$S_1[i]$ and $S_2[j]$ are the i-th character of S_1 and the j-th character of S_2, respectively, $D(0,0) = 0$, and $D(i,j) = \infty$ if $i < 0$ or $j < 0$. ins and del in the formula are constants and denote the insertion and deletion costs, respectively. $\text{sub}(a,b)$ is the substitution cost of two characters a and b. sub is represented as an $\alpha \times \alpha$ two-dimensional array, where α is the number of alphabet characters. For discussion, let D be an $(m+1) \times (n+1)$ two-dimensional array such that $\text{D}[i,j] = D(i,j)$. D is called the edit distance matrix.

2.2 Approximate Regular Expression Matching

Here, we explain how to introduce regular expression operators and other operators into approximate string matching. Hereinafter, "a string P matches a string S" means that the edit distance between P and S is zero. The definitions of the main target operators are as follows.

1) *Single-character don't care* (*SCDC*) (?)
   The pattern "?" matches any single character.
2) *Variable-length don't care* (*VLDC*) ($?^*$)
   The pattern "$?^*$" matches any string, including the zero-length string "ε", where ε is the empty character.
3) *Negation* ($\bar{p}$)
   A pattern "$\bar{p}$" matches any single character other than p.
4) *Empty character matching* ($p@$)
   A pattern "$p@$" matches p and the empty character ε.
5) *Character-by-character matching* (*CC matching*) ($[p_1p_2\cdots p_l]$)
   A pattern $P =$ "$[p_1p_2\cdots p_l]$" matches only the string "$p_1p_2\cdots p_l$". If P and S have different lengths, the edit distance between P and S is ∞. Otherwise, the edit distance between P and S is the same as that between "$p_1p_2\cdots p_l$" and S.
6) *Exact matching* ($\langle[p_1p_2\cdots p_l]\rangle$)
   A pattern $P =$ "$\langle[p_1p_2\cdots p_l]\rangle$" matches only the string "$p_1p_2\cdots p_l$". If S is not "$p_1p_2\cdots p_l$", the edit distance between P and S is ∞.
7) *Kleene operator* (p^*)
   A pattern "p^*" matches any strings that do not include any character other than p.

The other target operators are shown in [4].

Due to space limitations, we here explain only the DP formulation of exact matching. Assume $P = \langle[p_1p_2\cdots p_l]\rangle$. Let $\text{ins}(i)$ be the insertion cost of a character between p_i and p_{i+1}. Also, let $\text{sub}(p,t)$ be the substitution cost from a pattern character p to a text character t. Exact matching is realized by setting the deletion, insertion and substitution costs as defined in Expressions (2), (3) and (4), respectively. Note that although insertions right after p_i ($1 \le i < l$) are not allowed, insertion right after p_l is allowed if some pattern characters follow the exact match (*e.g.*, "$\langle[abc]\rangle e$" matches "$abcde$"). Thus, the constant insertion cost $\text{ins}(= k)$ in the original DP formulation is replaced by the function $\text{ins}(i)$.

$$\text{del} = \infty \quad (2)$$

Table 1: Definitions of del, ins, and sub for each operator

operator	del	ins	sub
w/o operator p	del	ins	$\mathrm{sub}(p,t)$
1. SCDC ?	del	ins	0
2. VLDC ?*	0	0	0
3. Negation $\bar{p}$	del	ins	$\mathrm{sub}(\bar{p},t)$
4. Empty character $p@$	0	ins	$\mathrm{sub}(p,t)$
5. CC matching $[p_1p_2\cdots p_l]$ $1 \le i < l$ $i = l$	 ∞ ∞	 ∞ ins	 $\mathrm{sub}(p,t)$ $\mathrm{sub}(p,t)$
6. Exact matching $\langle[p_1p_2\cdots p_l]\rangle$ $1 \le i < l,\ p = t$ $1 \le i < l,\ p \neq t$ $i = l, p = t$ $i = l, p \neq t$	 ∞ ∞ ∞ ∞	 ∞ ∞ ins ins	 0 ∞ 0 ∞
7. Kleene operator p^*	0	$\min\{\mathrm{ins}, \mathrm{sub}(p,t)\}$	$\mathrm{sub}(p,t)$

$$\mathrm{ins}(i) = \begin{cases} \infty & 1 \le i < l \\ k & i = l \end{cases} \tag{3}$$

$$\mathrm{sub}(p,t) = \begin{cases} 0 & p = t \\ \infty & p \neq t \end{cases} \tag{4}$$

The other operators can be realized by similarly replacing the deletion cost del, insertion cost ins and substitution cost sub. The definitions of del, ins and sub for each operation are shown in Table 1.

2.3 GPU

A graphics processing unit (GPU) is an application specific processor for graphics processing and one of the main components of PCs. Utilizing GPUs for general-purpose computing is called GPGPU (general-purpose computation on GPUs) [5]. In recent years, GPUs without video outputs were developed for general-purpose computing by some semiconductor companies (*e.g.*, Tesla C2070 by NVIDIA Corporation and FireStream 9350 by Advanced Micro Devices, Inc.).

2.3.1 GPU Architecture

Fig.1 illustrates an architecture of NVIDIA's GPUs, called Fermi. It is composed of two LSIs, a GPU itself and a memory chip, called device memory. A GPU has up to 16 streaming multi-processors (SMs), each of which corresponds to a core in a multicore CPU. An SM has 32 streaming processors (SPs), each of which corresponds to an arithmetic logic unit in a CPU core. An SP has only limited functions such as arithmetic operations, and SPs execute instructions decoded by an SM in a SIMD fashion. That is, all the SPs in an SM simultaneously execute the same instruction. In addition, an SM has a memory shared by all

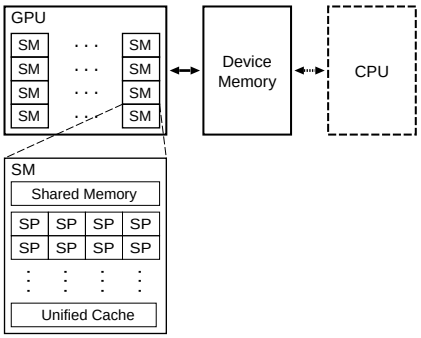

Figure 1: Architecture of NVIDIA Tesla series GPU

the SPs in the SM and some memories to cache data from and to the device memory.

GPUs have a hierarchical memory architecture. The memory architecture of the Fermi GPUs is as follows:

1) Global memory
   Global memory is a high-capacity memory realized by the device memory. It is accessible from any SPs in the GPU. The size of the memory is up to several giga-bytes. On the other hand, its latency is high and it requires 400 to 600 clock cycles to access. Accesses from SPs to the memory are cached in the SM to which the SP belongs. It is the only memory readable/writable from both the GPU and CPU.
2) Shared memory
   Each SM has a 64KB on-chip memory, and 16KB or 48KB of the memory is used as its shared memory. The rest of the on-chip memory is used as the L1 cache of global memory. All the SPs in an SM are quickly accessible to the memory.
3) Register
   Each SP has its own registers. Registers are the memory most quickly accessible from SPs. Registers in an SP is not accessible from the other SPs. Each SM has 8K to 32K registers, depending on the GPU.
4) Constant and texture memories
   Constant and texture memories are read-only memories realized by the device memory. They are accessible from any SPs in the GPU. Accesses from an SP to the memories are cached in the SM to which the SP belongs. They are writable from the CPU. In addition, texture memory has some additional functions, such as address space normalization to [0,1] and data interpolation between adjacent data points. Since the proposed method does not use those memories, we omit the detailed explanation of them.

2.3.2 CUDA

CUDA is a programming environment for developing and executing general-purpose applications on NVIDA's GPUs.

It makes multi-thread applications run on GPUs efficiently. An extended C/C++ is used as the programming language in CUDA.

In CUDA, concepts to manage threads, called *grid* and *thread block*, are introduced. A group of threads is called a thread block. The maximum number of threads in a thread block is 512. A group of thread block is called a grid. Each thread block in a grid is managed by adding a two-dimensional ID. The maximum number in each dimension of a thread block ID is 65,535.

Each thread executes a code, called a *kernel*. Each thread has a unique ID, by which threads handle different data, executing the same code. Threads in the same thread block shares data in the high-speed shared memory. On the other hand, a thread cannot access to shared memories in different thread blocks. Thus, codes need to be written so that there are as few data dependencies between thread blocks as possible. In addition, although threads in different thread blocks cannot be synchronized during a kernel execution, threads in the same thread block can be synchronized using a command and keep data consistency.

3. FPGA Implementation of Approximate String Matching with Regular Expression Operators

In this section, a hardware algorithm for approximate regular expression matching proposed in [4] is explained. Our proposed method is based on the same idea and adjusted to GPUs.

3.1 Architecture

The architecture for the hardware algorithm is shown in Fig. 2. Let $m = |P|$ and $n = |T|$. Then, it is a one-dimensional array of $m+1$ units, called *cells*, each of which compares a pattern character and a text character. That is, each cell calculates the elements in a row of the edit distance matrix D, shown in Fig. 3. Note that each dimension of D is expanded by one element in order to calculate the edit distances between the pattern and substrings of the text (please refer to [4] for more detail). The $m+1$ cells calculate the elements on a diagonal line in parallel and the calculation proceeds from the top-left corner to the bottom-right corner as shown in Fig. 3. Thus, its calculation time is $O(m+n)$. The resultant edit distances are output from the right-most cell and input to a comparator. The comparator compares the user-defined threshold k and an edit distance. If the edit distance is less than or equal to k, the comparator outputs a match signal. Since each cell handles one character in a pattern, the length of patterns is limited to the number of cells.

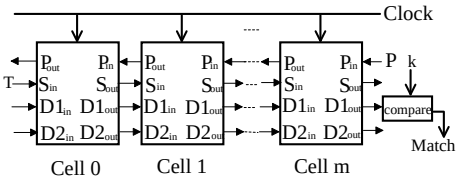

Figure 2: Architecture of Hardware Engine

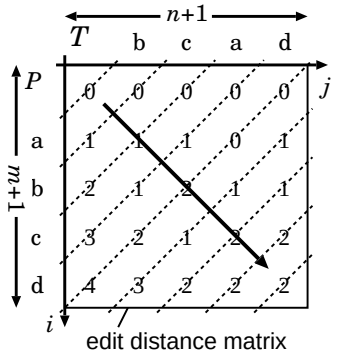

Figure 3: Order of Calculation in D

3.2 Basic Structure and Behavior of Cell

Fig. 4 shows the structure of a cell. C_p and C_t are registers and store one character in the pattern and one character in the text, respectively. DM is a memory to store the table of substitution costs. DM is a two-dimensional array of substitution cost from alphabet α_1 to α_2, called a distance matrix. That is, DM$[\alpha_1, \alpha_2]$ stores the value of sub(α_1, α_2). The distance matrix DM is set before starting matching. (In our experiment, the substitution costs of any two characters are set to 1 and DM is omitted.) $D1$ and $D2$ are registers to temporarily store elements of D. $D1$ stores the newest element the cell calculated, and $D2$ stores the second newest element.

Next, we explain the behavior of the cell. A cell i handles one character of the pattern, p_i, and calculates $D[i, *]$. When the cell i calculates $D[i, j]$ at a clock cycle T_k, $D[i, j-1]$, $D[i-1, j]$ and $D[i-1, j-1]$ are necessary. $D[i, j-1]$ was calculated at the cell i at the clock cycle T_{k-1} and currently stored in $D1$. $D[i-1, j]$ was also calculated at the cell $i-1$ at the clock cycle T_{k-1} and currently stored in $D1$ of the cell $i-1$. $D[i-1, j-1]$ was calculated at the cell $i-1$ at the clock cycle T_{k-2} and currently stored in $D2$ of the cell $i-1$. Thus, all the elements necessary to calculate $D[i, j]$ are stored in the cells i and $i-1$.

4. GPGPU Implementation of Approximate Regular Expression Matching

In this paper, we propose an efficient GPGPU implementation method of the hardware algorithm for approximate

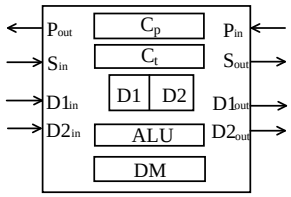

Figure 4: Structure of Cell

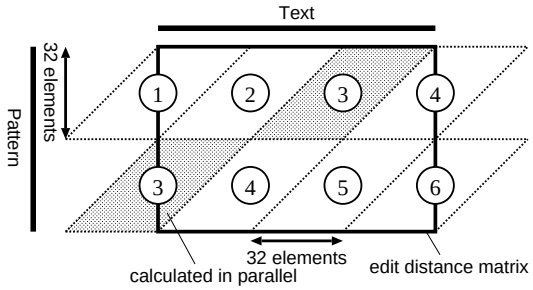

Figure 5: Parallel Calculation on GPU

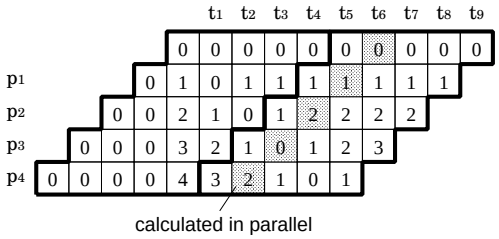

Figure 6: Calculation in Parallelogram Region

regular expression matching [4]. Although GPUs are processors specific to graphics processing, some studies have been conducted to utilize their highly parallel architecture for string matching (*e.g.*, [3]).

In [4], the hardware algorithm is implemented on an FPGA. However, since the length of patterns is limited to the number of cells and the number of cells is limited by the capacity of the FPGA, long patterns cannot be handled by the FPGA implementation. For example, the one implemented in the experiments in [4] can handle no more than 250 pattern characters. In contrast, since in a GPU an SM can handle multiple thread blocks in a time-division manner (*i.e.*, an SP handles multiple threads), a GPU can handle patterns whose length is greater than the number of SPs by allocating the function of each cell to a thread. This fact makes approximate regular expression matching applicable to the applications with long patterns (*e.g.*, analysis of DNA sequences).

In the following, we first show the overview of our method for approximate string matching (without regular expression) on GPUs. Then, we explain the effective memory access method for our matching method. Finally, we introduce regular expression operators to the method.

4.1 Division of Edit Distance Matrix D

In this paper, we propose an efficient approximate regular expression matching method that can handle long patterns utilizing a GPU. In our method, we divide the edit distance matrix D into multiple parts and effectively dispatch them to SMs.

First, we divide the edit distance matrix into the parallelogram regions so that the length of each side of the regions is 32, as shown in Fig. 5. This is because each SM handles 32 threads as one executable unit, called a *warp*. Then, we dispatch the calculations in parallelogram regions to SMs in a GPU. Since SPs in different SMs cannot be synchronized, the parallelogram regions are calculated in the order shown in Fig. 5 to maintain data dependencies. The parallelogram regions with the same number are calculated by multiple SMs in parallel. The calculations of the parallelogram regions with each number are started by calling a kernel and thus synchronized. The elements on a line parallel to the right and left sides of a parallelogram are calculated in parallel by the SPs in a SM. This division realizes an effective use of SMs and SPs in a GPU for the calculation of the edit distance matrix with a long pattern.

Fig. 6 shows the calculation in each parallelogram region. The elements in a region have the same data dependencies as that of Fig. 3. Thus, we allocate the function of a cell to a thread to calculate the elements on a line parallel to the right and left sides of the parallelogram region in parallel.

In summary, in our proposed method, the edit distance matrix is divided into parallelogram regions, and the regions are calculated in the order shown in Fig. 5 in parallel. In each region, the elements on a line parallel to the right and left sides of the parallelogram region are allocated to SPs and calculated in parallel.

4.2 Calculation of Edit Distance on Shared Memory

Here, we describe how to calculate the edit distance using shared memories. The calculation of the edit distance is easily implemented by placing whole the edit distance matrix D to the global memory. However, the cost to access to the global memory is very high and the latency is 400 to 600 clock cycles. Therefore, we utilize shared memories to calculate the matrix.

To calculate the value of elements in a parallelogram region, the calculation results of other regions are necessary. Since different regions are handled by different SMs, it is necessary for SMs to access the global memory to communicate each other. On the other hand, since SMs do not need to communicate each other when calculating inner

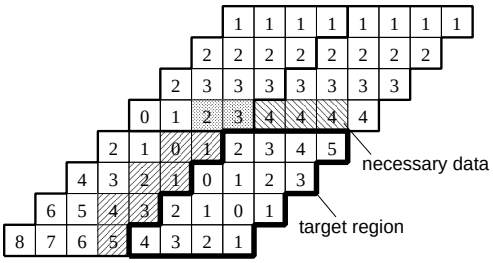

Figure 7: Data Dependencies among Parallelogram Regions

elements of regions, shared memories can be used to quickly calculate the elements. To calculate the elements on the i-th line parallel to the right and left sides of the parallelogram region, only the elements on the $(i-1)$-th and $(i-2)$-th lines need to be stored in the shared memory of the corresponding SM.

The data dependencies among parallelogram regions are shown in Fig. 7. The calculation in the parallelogram region enclosed by a heavy line requires only the shaded elements. Thus, in our method, only those elements are stored in the global memory. In other words, only the elements on the last two lines parallel to the right and left sides of a parallelogram region and those in the bottom row in the region are stored in the global memory.

Fig. 8 shows the pseudo code of the kernel. In the code, *r_D* is a register to temporarily store an element of the edit distance matrix, $D[i, j]$. *r_left*, *r_top*, and *r_diag* are registers to temporarily store the elements $D[i-1, j]$, $D[i, j-1]$, and $D[i-1, j-1]$, respectively. The register *r_left* is used to send the value of $D[i-1, j]$ for the calculation of $D[i+1, j]$ without using shared memories. The registers *r_top* and *r_diag* are used to shorten the *then* and *else* statements in the *if-then-else* statement. This is because a GPU executes both of *then* and *else* statements to execute threads in a SIMD fashion. Without *r_top* and *r_diag*, both of the *then* and *else* statements need to include similar codes (corresponding to the 18th line, which becomes more complicated when regular expression operators are introduced), and it degrades the performance. *idx* is the ID number of the thread in the grid, *tid* is the ID number of the thread block, and *b_id* is the ID number of the thread block. *text*, *top*, and *down* are arrays located in the shared memory to store text characters, the elements on the region, and the elements in the bottom row in the region, respectively.

In the 1st to 7th lines of the code, the data needed by the thread block are read from the global memory to the shared memory. Note that the data are read in parallel by using all the thread in the thread block. In the 8th to 24th lines, the elements in a row are calculated. Note that different threads in a thread block handle different rows, and the elements in the rows are calculated in parallel. In the 9th line, a text character is read to the register *t*. DM in the 10th line is a

```
01.  text[tid] = T[tid+(a-1)*SIZE];
02.  text[SIZE+tid] = T[tid+(a)*SIZE];
03.  top1[tid+1] = D[(b_id)*(SIZE+n+1)+tid+(a+1)*SIZE];
04.  s1[tid] = D[(idx+1)*(SIZE+n+1)+a*SIZE];
05.  s2[tid] = D[(idx+1)*(SIZE+n+1)+a*SIZE-1];
06.  r_E = D[(idx+1)*(SIZE+n+1)+a*SIZE];
07.  p = P[idx];
08.  for(i=0; i<SIZE; i++){
09.      t = text[tid];
10.      sub = DM[p*128+t];
11.      if(tid==0){
12.          r_top = top[i+1];
13.          r_diag = top[i];
14.      }else{
15.         r_top = s1[tid-1];
16.         r_diag = s2[tid-1];
17.      }
18.      r_D=min(r_top+del, r_left+ins, r_diag+sub);
19.      s1[tid] = r_D;
20.      s2[tid] = r_top;
21.      r_left = r_D;
22.      down[i] = r_D;
23.      __syncthreads();
24.  }
25.  D[(idx+1)*(SIZE+n+1)+SIZE+a*SIZE] = r_D;
26.  D[(idx)*(SIZE+n+1)+SIZE-1+a*SIZE] = r_top;
27.  D[(b_id*SIZE)*(SIZE+n+1)+tid+(a*SIZE)] = down[tid];
```

Figure 8: Pseudo Code of Kernel

two-dimensional array to store the substitution costs. In this line, the substitution cost of the pattern character and text character is obtained. In the 11th to 17th lines, $D[i-1, j]$ and $D[i-1, j-1]$ are read from the shared memory to registers. In the 18th line, $D[i, j]$ is calculated. In the 19th to 22nd lines, $D[i, j]$ and $D[i, j-1]$ in the shared memory are updated. Note that only the newest and second newest lines of elements are stored in the shared memory. In the 23rd line, all the threads in the thread block are synchronized (because each SP handles multiple threads). In the 25th to 27th lines, the date necessary to calculate the next regions are sent from the shared memory to the global memory.

4.3 Introduction of Regular Expression Operators

As shown in Section 2, our target regular expression operators can be realized by replacing del, ins and sub in the DP formula of the edit distance calculation. To input patterns with the regular expression operators to the kernel, we introduce an array of characters *OP* ($|OP| = |P|$) for representing operators to the corresponding pattern characters, and an array of integers *e* ($|e| = |P|$) for representing the first and last characters of the target substring of an operator, as input data to the kernel. In the kernel, *OP* and *e* are copied from the global memory to the shared memory like as pattern *P*.

5. Comparison of GPGPU and FPGA Implementations

To compare FPGA, GPGPU and CPU implementations, we conducted some experiments. For those implementations,

we used a PC equipped with an Intel Core i7 950 CPU (3.06GHz), 24GB main memory, CentOS, an NVIDIA Tesla C2070 (1.15GHz) GPU with 4GB device memory, and an FPGA board with a Xilinx Virtex-4 FPGA. The GPU and the FPGA board are connected to the PC with PCIe 2.0 x16 and conventional PCI buses, respectively. The FPGA implementation is written in Verilog HDL and mapped using Xilinx ISE 13.1.

In our GPGPU implementation, one thread block consists of 32 threads. Thus, each thread block uses 1060bytes of the shared memory. Each thread uses 42 registers. Since in the Tesla C2070 each SM has a 48KB shared memory, each SM can handle up to 45 thread blocks ($45 \times 1060 < 48$K). Therefore, our GPGPU implementation can handle long patterns whose length is less than or equal to 20,160 = $14 \times 45 \times 32$ = (the number of SMs) $\times$ (the maximum number of thread blocks per SM) $\times$ (the number of threads in one thread block). In contrast, since only 250 cells can be implemented on the target FPGA, the FPGA implementation can handle only patterns whose length does not exceed 250. The maximum clock frequency was 140MHz.

Table 2 shows the execution times of the GPGPU and CPU implementations when $|T| = 3,200,000$ and $|P| = 320$, and that of the FPGA implementation when $|T| = 3,200,000$ and $|P| = 250$. Note that the execution times include data transfer time from the main memory. As a result, the FPGA and GPGPU implementations were 8.3 and 2.9 times as fast as the CPU implementation, respectively.

Table 3 and Fig. 9 show the results in the cases of long patterns. We found that 1) the execution time of the CPU implementation is proportional to the pattern length, 2) that of the GPGPU implementation is stepwise in terms of the pattern length (there is a gap between $|P| = 3200$ and $|P| = 4800$). In addition, the GPGPU implementation when $|P| \geq 3200$ was more than 18 times as fast as the CPU implementation.

These results indicate that the FPGA implementation is the fastest and suitable to the cases of short patterns, and the GPGPU implementation is scalable and suitable to the cases of long patterns, such as analysis of DNA sequences.

Table 2: Execution Time when $|T| = 3,200,000$

Method	$\|P\|$	Time
CPU	320	23.60 [s]
GPGPU	320	8.23 [s]
FPGA	250	2.89 [s]

Table 3: Execution Time when $|T| = 320,000$ [ms]

$\|P\|$	320	640	1280	3200	4800	6400
CPU	2399	4803	9612	24020	36030	47960
GPU	1079	1102	1131	1212	1922	2031

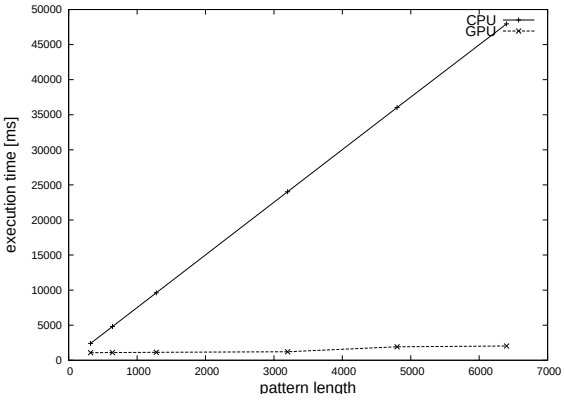

Figure 9: Execution Time of CPU and GPU Implementations

6. Conclusions

In this paper, we proposed an efficient GPU-based method for approximate regular expression matching with long patterns. Experimental results showed that 1) our proposed method and an FPGA-based method [4] are 2.9 and 8.3 times as fast as a CPU implementation, respectively, when the length of patterns is 320; 2) our method is more than 18 times as fast as the CPU implementation when the length of patterns is more than 3200. Our future work includes further improvement of memory access efficiency in our GPU-based method.

References

[1] J. Aoe (Ed.), *Computer Algorithms: String Pattern Matching Strategies*, IEEE Computer Society Press, 1994.

[2] S. Mikami, Y. Kawanaka, S. Wakabayashi, and S. Nagayama, "Efficient FPGA-based hardware algorithms for approximate string matching," in *Proc. the 23rd International Technology Conference on Circuits/Systems, Computers and Communications (ITC-CSCC 2008)*, July 2008, pp.201–204.

[3] K. Dohi, K. Benkridt, C. Ling, T. Hamada, and Y. Shibata, "Highly Efficient Mapping of the Smith-Waterman algorithm on CUDA-compatible GPUs," in *Proc. the 21st IEEE International Conference on Application-specific System Architecture and Processors (ASAP 2010)*, July 2010, pp. 29–36.

[4] Y. Utan, S. Wakabayashi, and S. Nagayama, "An FPGA-Based Text Search Engine for Approximate Regular Expression Matching," in *Proc. the 2010 International Conference on Field-Programmable Technology (FPT'10)*, Dec. 2010, pp. 69–74.

[5] E. Kandrot and J. Sanders, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, July 2010.

[6] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no.1, pp. 168–178, 1974.