Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 937694, 20 pages
http://dx.doi.org/10.1155/2015/937694

Research Article

Cache Locality-Centric Parallel String Matching on Many-Core Accelerator Chips

Nhat-Phuong Tran,[1] Myungho Lee,[1] and Dong Hoon Choi[2]

[1] *Department of Computer Science and Engineering, Myongji University, 116 Myongji Ro, Cheo-In Gu, Yong In, Kyungki Do 449-728, Republic of Korea*
[2] *Korea Institute of Science and Technology Information (KISTI), 245 Dae Hak Ro, Yu Seong Gu, Daejeon 305-806, Republic of Korea*

Correspondence should be addressed to Myungho Lee; myunghol@mju.ac.kr

Received 2 April 2015; Revised 6 August 2015; Accepted 7 September 2015

Academic Editor: Bronis R. de Supinski

Copyright © 2015 Nhat-Phuong Tran et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Aho-Corasick (AC) algorithm is a multiple patterns string matching algorithm commonly used in computer and network security and bioinformatics, among many others. In order to meet the highly demanding computational requirements imposed on these applications, achieving high performance for the AC algorithm is crucial. In this paper, we present a high performance parallelization of the AC on the many-core accelerator chips such as the Graphic Processing Unit (GPU) from Nvidia and the Intel Xeon Phi. Our parallelization approach significantly improves the cache locality of the AC by partitioning a given set of string patterns into multiple smaller sets of patterns in a space-efficient way. Using the multiple pattern sets, intensive pattern matching operations are concurrently conducted with respect to the whole input text data. Compared with the previous approaches where the input data is partitioned amongst multiple threads instead of partitioning the pattern set, our approach significantly improves the performance. Experimental results show that our approach leads up to 2.73 times speedup on the Nvidia K20 GPU and 2.00 times speedup on the Intel Xeon Phi compared with the previous approach. Our parallel implementation delivers up to 693 Gbps throughput performance on the K20.

1. Introduction

Recently, many-core accelerator chips such as the Graphic Processing Units (GPUs) from Nvidia and AMD and Intel's Many Integrated Core (MIC) architectures, among others, are becoming increasingly popular. The influence of these chips is rapidly growing in the High Performance Computing (HPC) server market and in the Top 500 list, in particular. They have a large number of cores and multiple threads per core, levels of cache hierarchies, large amounts (>5 GB) of the on-board memory, and >1 Tflops peak performance for the double precision arithmetic per chip. They are mostly utilized as coprocessors and execute parallel program kernels commanded by the host CPU with respect to the input data provided from the host memory to the on-board device memory. Using the many-core accelerators, a number of innovative performance improvements have been reported for HPC applications and many more are still to come.

String matching is an important algorithm in computer and network security [1–6], bioinformatics [7, 8], and so forth. Among many string matching algorithms, Aho-Corasick (AC) [9] is a multiple patterns string matching algorithm which can simultaneously match a number of patterns for a given finite set of strings (or dictionary). A Deterministic Finite Automata (DFA) is first constructed for the given set of pattern strings. Then the pattern matching operations are conducted with respect to the input text data while referencing the DFA. The input data is accessed sequentially; thus, the access pattern is quite predictable. However, the access to the DFA is irregular as there are a lot of jumps from one state to another state of the DFA when processing the input characters sequentially for possible matches. As the number of pattern strings increases, for example, up to several dozens of thousands of virus pattern strings in the computer virus scan [1], the number of states in the DFA increases accordingly. The large number of states in the DFA data structure with irregular access leads to the poor data locality and high cache misses. Therefore, in order to speed up the pattern matching and meet the performance requirements imposed on the AC, optimizing the cache locality is crucial.

In this paper, we develop a high performance parallelization for the AC string matching algorithm which significantly improves the cache locality for the irregular DFA access on the many-core accelerator chips such as the Nvidia Tesla K20 GPU and the Intel Xeon Phi. Previous research to parallelize the AC [4–6] partitions the input data amongst multiple threads and conducts intensive pattern matching operations in parallel while referencing the single DFA. This approach, however, leads to a large number of cache misses for the DFA access with the increase in the number of pattern strings. Our parallelization approach, instead, partitions the given set of pattern strings into multiple sets of a smaller number of pattern strings in a space-efficient way. Thus, multiple small DFAs are constructed instead of the single large DFA of the previous approach. Using the multiple small DFAs, intensive pattern matching operations are concurrently conducted with respect to the common input text string. This leads to significantly smaller cache footprints in each core's cache for referencing the partitioned DFAs which have irregular access patterns. Thus, it results in lower cache miss ratios and impressive performance improvements. Experimental results on the Nvidia Tesla K20 GPU based on the Kepler GK110 architecture show that our approach leads up to 2.73 times speedup compared with the previous input data partitioning approach. The throughput performance reaches up to 693 Gbps. Compared with single CPU core (out of 6-core 2.0 Ghz Intel Xeon E5-2650), we obtained a speedup in the range of 127~311. The speedup over the parallelized CPU version using 6 threads is in the range of 86~183. Experimental results on the Intel Xeon Phi with 61 x-86 cores also show up to 2.00 times speedup compared with the previous input data partitioning approach.

The rest of the paper is organized as follows. Section 2 describes the architectures of the latest many-core accelerator chips such as the Nvidia Tesla K20 GPU and the Intel Xeon Phi. Section 3 introduces the AC algorithm. Section 4 describes our parallelization approach which improves the cache locality for accessing the DFA. Section 5 shows the experimental results on the Nvidia Tesla K20 and on the Intel Xeon Phi. Section 6 explains the previous related research on parallelizing the AC algorithm. Section 7 wraps up the paper with conclusions.

2. Latest Many-Core Accelerator Chip Architectures

Recently, many-core accelerator chips are becoming increasingly popular for the HPC applications. Representative chips are the Nvidia Tesla K20 based on the Kepler GK110 architecture and the Intel Xeon Phi based on the Many Integrated Core (MIC) architecture. In the following subsections, we describe these architectures.

2.1. Nvidia Tesla K20 GPU. The latest GPU architecture is characterized by a large number of uniform fine-grain programmable cores or thread processors which have replaced separate processing units for shader, vertex, and pixel in the earlier GPUs. Also, the clock rate of the latest GPU has ramped up significantly. These have drastically improved the floating point performance of the GPUs, far exceeding that of the latest CPUs. The fine-grain cores (or thread processors) are distributed in multiple streaming multiprocessors (SMX) (or thread blocks) (see Figure 1). Software threads are divided into a number of thread groups (called WARPs) each of which consists of 32 threads. Threads in the same WARP are scheduled and executed together on the thread processors in the same SMX in the SIMD (Single Instruction Multiple Data) mode. Each thread executes the same instruction directed by the common Instruction Unit on its own data streaming from the device memory to the on-chip cache memories and registers. When a running WARP encounters a cache miss, for example, the context is switched to a new WARP while the cache miss is serviced for the next few hundred cycles. Thus, the GPU executes in a multithreaded fashion as well.

The GPU is built around a sophisticated memory hierarchy as shown in Figure 1. There are registers and local memories belonging to each thread processor or core. The local memory is an area in the off-chip device memory. Shared memory, level-1 (L1) cache, and read-only data cache are integrated in a thread block of the GPU. The shared memory is a fast (as fast as registers) programmer-managed memory. Level-2 (L2) cache is integrated on the GPU chip and used amongst all the thread blocks. Global memory is an area in the off-chip device memory accessed from all the thread blocks, through which the GPU can communicate with the host CPU. Data in the global memory get cached directly in the shared memory by the programmer or they can be cached through the L2 and L1 caches automatically as they get accessed. There are constant memory and texture memory regions in the device memory also. Data in these regions is read-only. They can be cached in the L2 cache and the read-only data cache. On Nvidia Tesla K20, the read-only data from the global memory can be loaded through the same cache used by the texture pipeline via a standard pointer without the need to bind to a texture beforehand. This read-only cache is used automatically by the compiler as long as certain conditions are met. The *__restrict__* qualifier should be used when a variable is declared to help the compiler detect the conditions [10].

In order to efficiently utilize the latest advanced GPU architectures, programming environments such as CUDA [10] from NVidia, OpenCL [11] from Khronos Group, and OpenACC [12] from a subgroup of OpenMP Architecture Review Board (ARB) have been developed. Using these environments, users can have a more direct control over the large number of GPU cores and its sophisticated memory hierarchy. The flexible architecture and the programming environments have led to a number of innovative performance improvements in many application areas and many more are still to come.

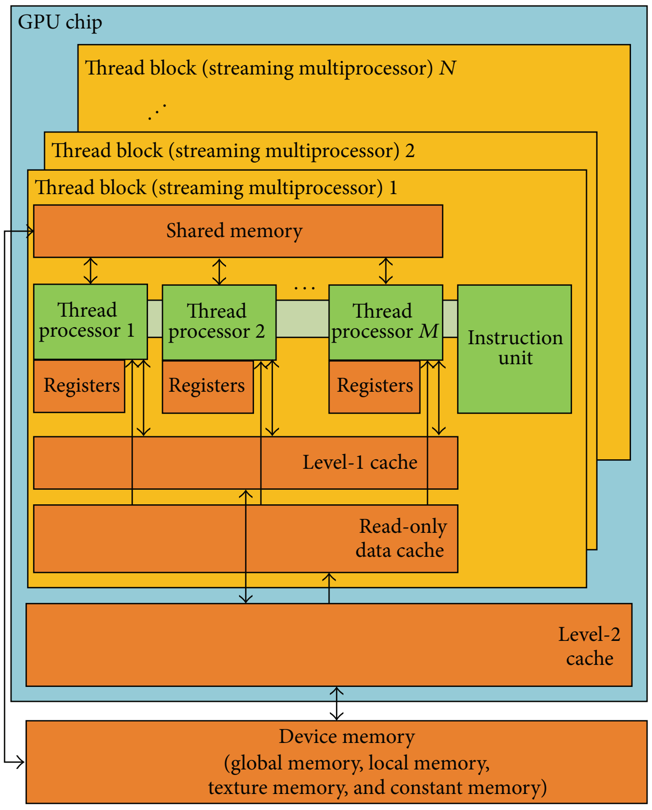

GPU chip
Thread block (streaming multiprocessor) N
Thread block (streaming multiprocessor) 2
Thread block (streaming multiprocessor) 1
Shared memory
Thread processor 1
Thread processor 2
Thread processor M
Instruction unit
Registers
Registers
Registers
Level-1 cache
Read-only data cache
Level-2 cache
Device memory (global memory, local memory, texture memory, and constant memory)

FIGURE 1: Architecture of a latest GPU (Nvidia Tesla K20).

2.2. Intel Xeon Phi. The Intel Xeon Phi (codenamed Knights Corner) is based on the Intel Many Integrated Core (MIC) architecture which combines multiple x86 cores on a single chip. This chip can run in either the native mode where an application runs directly on it or in the offload mode where the application runs on the host side and only the selected regions (compute-intensive regions) are offloaded to the Xeon Phi. For the offload mode, the Xeon Phi is connected to a host Intel Xeon processor through a PCI-Express bus.

In this paper, we use the Xeon Phi 5120D for our parallel implementation of the AC:

(i) This coprocessor has 60 in-order compute cores supporting 64-bit x86 instructions. These cores are connected by a high performance bidirectional ring interconnect (see Figure 2). It also has one service core, thus a total of 61 cores on the chip.

(ii) Each core is clocked at 1053 Mhz and offers the four-way multithreading (hyperthreading), 512-bit wide SIMD vectors which corresponds to eight double precision or sixteen single precision floating point numbers.

(iii) Each core has a 32 KB L1 data cache, a 32 KB L1 instruction cache, and a 512 KB unified L2 cache. Thus, 60 cores have a combined 30 MB L2 cache. L2 cache is fully coherent using the hardware support.

(iv) The Xeon Phi chip has 16 memory channels delivering up to 5 GT/s (Gigatransfer per second) transfer speed. The total size of the on-board system memory is 8 GB.

Programmers can use the same programming languages and models on the Xeon Phi as the Intel Xeon Processor. It can run applications written in Fortran, C/C++, and so forth and parallel models such as OpenMP, MPI, Pthreads, Intel Clik Plus, and Intel Thread Building Block [13].

3. Aho-Corasick (AC) String Matching Algorithm

The Aho-Corasick (AC) is a multiple patterns string matching algorithm which can simultaneously match multiple patterns for a given finite set of strings (or dictionary). The AC algorithm consists of two phases. In the first phase, a pattern matching machine called the AC automaton (machine) is constructed from a given finite set of pattern strings. In the second phase, the constructed AC machine is used to find the locations of the string patterns in the given input text [9].

Once the AC automaton is constructed, it invokes three functions in performing the pattern matching in its second phase: a goto function g, a failure function f, and an output function *output*. Figure 3 shows these functions for a given set of patterns {“he”, “she”, “his”, “hers”} [9]:

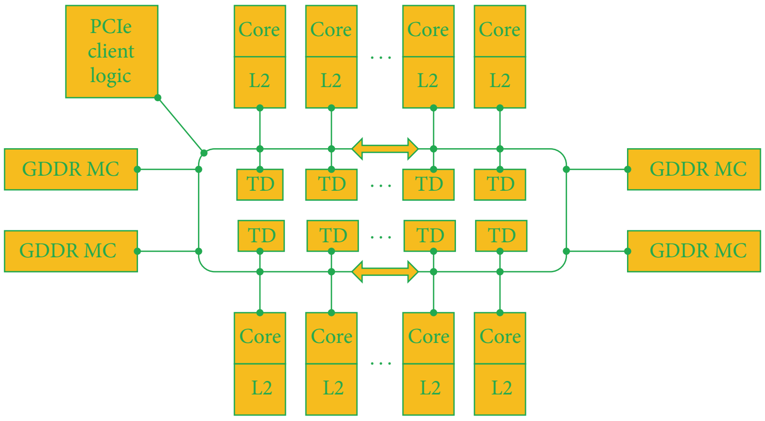
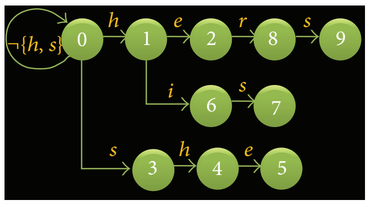
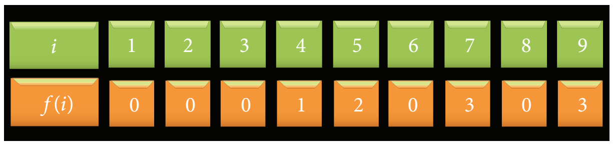


FIGURE 2: Architecture of Intel Xeon Phi.

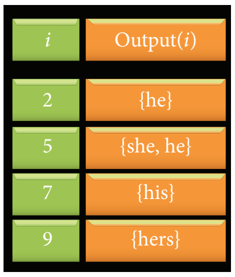


(a) Goto function: g



i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function: f



i	Output(i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) Output function: *output*

FIGURE 3: Functions used in AC algorithm [9].

(i) The directed graph in Figure 3(a) represents the goto function g. The g function maps a pair consisting of a state and an input symbol into a state or a message *fail*. For example, the edge labeled h from state 0 to 1 indicates that $g(0, \text{‘}h\text{’}) = 1$. $(\neg(\text{‘}h\text{’}, \text{‘}s\text{’})$ denotes all input symbols other than ‘h’, ‘s’). The absence of an arrow indicates failure. The AC machine has the property that $g(0, \sigma) \neq \textit{fail}$ for all input symbols σ.

(ii) The failure function f (Figure 3(b)) maps a state into another state. It is consulted whenever the goto function reports a "fail."

(iii) The output function *output* (Figure 3(c)) maps a set of keywords to output at the designated states.

The AC algorithm can be implemented as Nondeterministic Finite Automata (NFA) or Deterministic Finite Automata (DFA). In this paper, we implement the AC algorithm as a DFA which represents all of the possible states of the machine along with the information of the acceptable state transitions of the system [14]. The DFA consists of a finite set of states S and a next move function δ such that, for each state s and input symbol a, $\delta(s, a)$ is a state in S. Thus, the next move function δ is used in place of both the goto function and the failure function introduced in Figure 3. The output function is also incorporated in the DFA. Figure 4 shows the AC machine for a set of patterns {he, she, his, and hers}, where the three functions g, f, and *output* are integrated in a DFA. Starting from the initial state, the AC machine accepts an input character and moves from the current state to the next correct state.

Pseudocode 1 shows how the AC machine works as a DFA. In this pseudocode, n characters of the input text string are read sequentially while executing the for-loop. The next move function δ gets the new state from the current state and the character i. At the new state, the algorithm checks if there exists any match (if (*output*(state) != empty). If so, the output function is executed to print out the matched patterns. In this code, we only use two functions: the next move function δ and the output function. The failure function is removed while converting the NFA to the DFA. Assume that we have the pattern set {he, she, his, and hers} and the input text string "ushers." The DFA works in the following manner:

(i) Since $\delta(0, \text{‘}u\text{’}) = 0$, the AC machine feeds back to state 0.

(ii) Since $\delta(0, \text{‘}s\text{’}) = 3$, $\delta(3, \text{‘}h\text{’}) = 4$, and $\delta(4, \text{‘}e\text{’}) = 5$, the AC machine emits output(5) = {he, she}.

(iii) Since $\delta(5, \text{‘}r\text{’}) = 8$ and $\delta(8, \text{‘}s\text{’}) = 9$, the AC machine emits output(9) = {hers}.

The complexity of the AC algorithm is $O(n + m + z)$, where n is the sum of the length of the patterns, m is the length of the input text, and z is the number of the pattern occurrences in the input text. The construction of

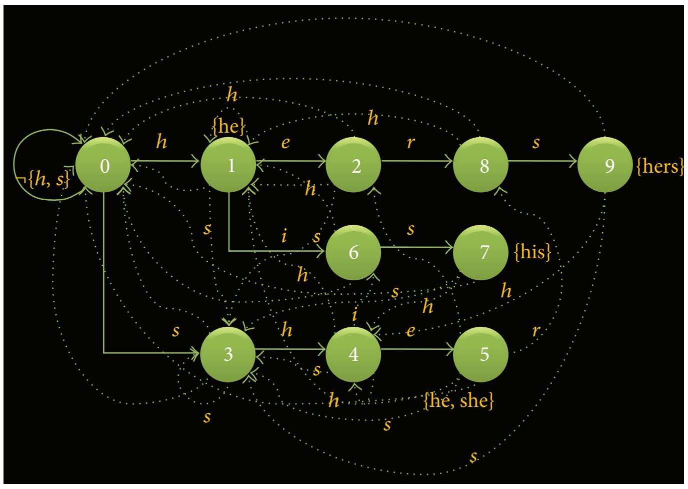


FIGURE 4: AC machine implemented as a DFA for a set of patterns {he, she, his, and hers}.

```
/* input: input text $x$, $n$ = length of input text
output: locations at which keywords occur in $x$ */
procedure DFA_AC(char *$x$, int $n$)
begin
        int state = 0;
        for (int $i = 0$; $i < n$; $i$++)
        begin
                state = $\delta$(state, $x[i]$);
                if (output(state) != empty)
                begin
                        print $i$
                        print output(state)
                end
        end
end
```

PSEUDOCODE 1: Pseudocode of the AC machine implemented as DFA.

the automaton takes $O(n)$. The pattern matching operations based on the automaton take $O(m + z)$. When the set of patterns is known in advance and does not change at the run time such as in the computer virus scan, the construction of the automaton can be conducted once off-line and the automaton is used multiple times for the pattern matches in the second phase. In this case, the complexity is $O(m + z)$.

4. Cache Locality-Centric Parallelization

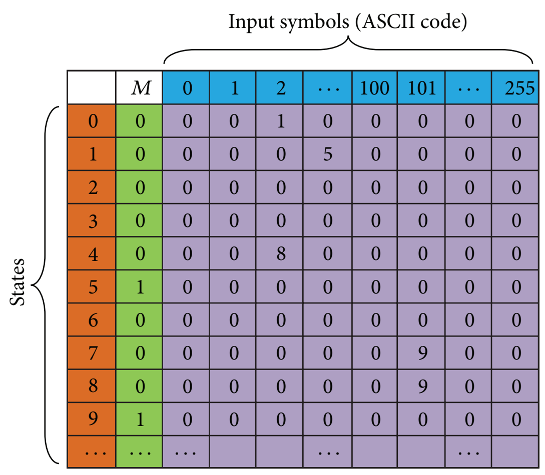
In this section, we explain our parallelization approach. We first describe preliminary steps. Then, we describe our approach which partitions the DFA into multiple smaller pieces for improving the cache locality.

4.1. Preliminaries

4.1.1. Execution Scenario of AC Algorithm. As explained in Section 3, the AC pattern matching DFA for a given finite set of strings (or dictionary) is constructed in the first phase. We assume that the constructed DFA is fixed in the second phase of the AC where the DFA is used to find the locations of the string patterns in the given input text. A good example case is the antivirus software where a virus database (or DFA) is constructed from a given set of several dozens of thousands of known viruses [1] in the first phase. Then, intensive pattern matching operations are conducted to detect the viruses in the hard disk image using the virus database in the second phase. The latter phase is time-consuming and is repeated multiple times using the same virus database before the user updates it. Therefore, in this paper, we assume that the AC DFA is constructed once on single CPU core of the host processor in the first phase and a parallel string matching is conducted on the GPU using the DFA in the second phase where our parallelization is conducted.

4.1.2. Constructing DFA. The AC pattern matching DFA is constructed in the first phase as a 2-dimensional matrix called the State Transition Table (STT) (see Figure 5). The rows of the STT represent the states in the DFA and the columns represent the input characters. Thus, for a given state i and an input character j, an entry STT$[i][j]$ denotes the corresponding next state or the failure state. Suppose that we have 256 input characters (mapped to 256 characters of the extended ASCII table), and then the STT needs 257 columns: 256 columns for characters and 1 column indicating if the current state is a matched state, where the output function is executed to print positions and patterns found in the input data at the state.

4.1.3. Data Placements. Once the STT is constructed by the host CPU and stored in the host memory, we copy it to


Input symbols (ASCII code)
States

	M	0	1	2	⋯	100	101	⋯	255
0	0	0	0	1	0	0	0	0	0
1	0	0	0	0	5	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	8	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	9	0	0
8	0	0	0	0	0	0	9	0	0
9	1	0	0	0	0	0	0	0	0
⋯	⋯	⋯			⋯			⋯	

Figure 5: State Transition Table (STT).

the device memory of the GPU along with the input data. When copying these data, we need to carefully decide where in the device memory of the GPU (global memory, constant memory, or texture memory) we need to store them. A large amount of data access is generated for both the input data and the STT data while the pattern matching operations are conducted. The input data is accessed sequentially from the beginning. On the other hand, the STT shows different access patterns. Starting from the initial state of the DFA, a character of the input data is looked up for the current state to find the next state. This state transition information is not known at compile-time; thus, it leads to close to random data access patterns for the STT.

Considering the above data access characteristics, we place the input data in the global memory so that it can be automatically loaded into the L1 cache (through the on-chip L2 cache) or explicitly loaded into the shared memory of the GPU by the programmer to minimize the access latencies. We attach the STT data in the texture memory so that the actively used part during the random access can be automatically cached in the L2 and the read-only caches of the GPU. This separates the access paths of the input and the STT so that they do not directly interfere with each other. Thus, it minimizes the memory access delays and uses the available memory bandwidths more efficiently. Figure 6 shows the resulting data access generated from the multiple threads assigned to the multiple thread processors (or fine-grain cores) of the thread blocks on the GPU when the pattern matching operations are performed based on our data placement scheme.

4.1.4. Other Considerations for Efficient Parallelization. With the input data and the STT data stored in the global memory and the texture memory, respectively, we consider the following for an efficient parallelization and high performance for the pattern matching operations:

(i) As stated above in Section 4.1.3, each input data chunk is assigned to a thread on the thread block. When the pattern matching operations are conducted, we span each thread's input chunk by $X - 1$ characters, where X is the maximum pattern length in the set of pattern strings. By doing this, we avoid to miss a pattern match when a pattern string lies on the boundary of the two adjacent data chunks for two different threads.

(ii) A software thread block running on the same hardware thread block (at the unit of WARP) generates a number of access to the input data and to the STT. As the GPU executes in the multithreaded fashion, the long global memory latencies for accessing the input data or the STT data for a WARP can be masked off or hidden by the pattern matching operations of other WARPs belonging to the same thread block or other thread blocks.

(iii) The global memory access overheads for loading the input data can be further reduced by efficiently utilizing the on-chip shared memory. We first divide the input data into a number of blocks. Each data block is assigned to each thread block. All threads in a block cooperate to load the corresponding data block from the global memory to the shared memory before the pattern matching operations are performed. The pattern matching operations for a block of threads are executed in a multithreaded fashion at the unit of WARP with the input data loads and the STT data loads from the global memory for some other WARPs. In order to use the shared memory in an optimal way, we need to carefully decide the number of thread blocks and the number of threads per each thread block. We will explain this in more detail in Section 4.3.

(iv) While loading the data into the shared memory, an important performance consideration is to coalesce the global memory access. In our parallel implementation, we let multiple threads of a block cooperate to load one chunk of data after another to fully load a data block for the thread block. We will describe our global memory access coalescing technique in more detail in Section 4.3.

(v) When the input data chunk is loaded from the global memory to the shared memory, we need a careful store scheme to avoid or minimize the shared memory bank conflicts when the stored data get accessed by multiple threads simultaneously which are also spread over multiple banks. We use a store scheme where the input data loaded from the global memory is divided up into a small number of bytes and stored to different banks to avoid any bank conflicts. We will describe our scheme in more detail in Section 4.3.

4.2. Parallelization Based on DFA Partitioning. Once the input data and the STT data are placed in the global memory and the texture memory, intensive pattern matching operations are conducted in the second phase of the AC while referencing these data. In the previous researches [4–6], they

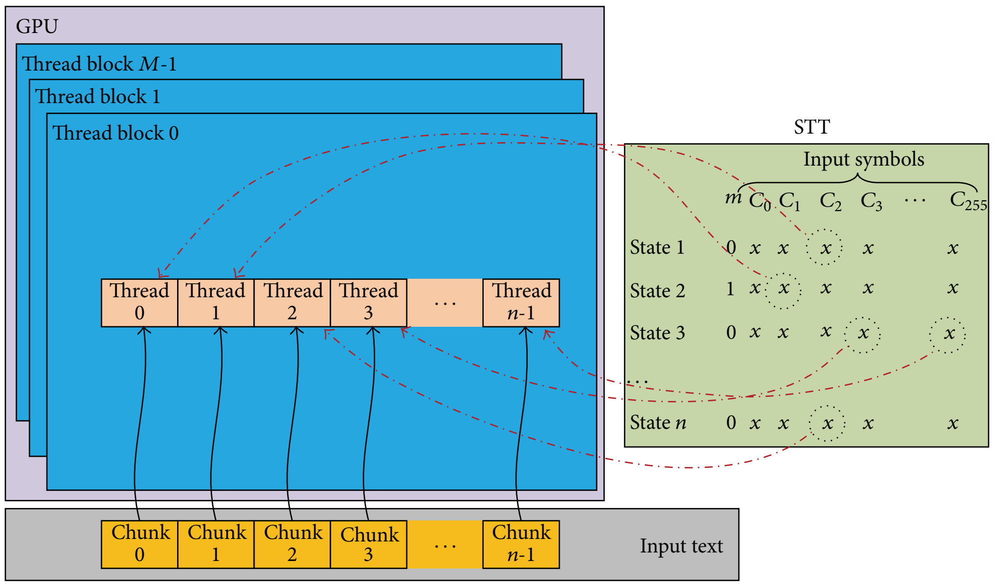


Figure 6: Data access patterns in the parallel AC algorithm on GPU.

parallelized the second phase by partitioning the input data into multiple pieces and assigning each piece to different processor cores or threads. Then, each core or thread conducts pattern matching operations in parallel while referencing the single large STT. This approach, however, incurs large cache miss overheads for accessing the STT as the STT access is quite irregular. Furthermore, as the number of pattern strings increases, the size of the STT increases accordingly. Thus, a large STT randomly accessed in parallel by multiple cores or threads leads to the poor cache locality and the low performance.

In order to significantly reduce the overheads associated with the irregular STT access with the high cache misses, we partition the given set of patterns into multiple small pattern sets. Then, for each small pattern set, we construct a corresponding DFA in the first phase of the AC which are represented as multiple small STTs (see Figure 7(b)). In the second phase, the whole input data is loaded by multiple cores or threads using the same STT for the pattern matching. Figure 8 compares our parallelization approach (Figure 8(b)) with the previous approach (Figure 8(a)). Previous approach [4–6] uses the partitioned input data amongst multiple cores or threads and the common large STT. Our approach uses the partitioned small STTs (STT1, . . . , STT4) from multiple cores or threads and the common whole input data. Thus, our approach significantly reduces the cache footprints for referencing the STT for each core or thread. Our approach, on the other hand, loads the whole input data for each core or thread. Since the input data is accessed sequentially, it can be efficiently loaded from the global memory into the on-chip shared memories of the GPU by the programmer. Thus, our approach leads to better cache hit ratio and overall performance as we will show later in Section 5 (Experimental Results).

When we partition the given set of patterns into multiple small pattern sets, we use an algorithm consisting of two parts:

(i) Part 1 (Algorithm 1) distributes the pattern strings with the same starting characters into one STT. It distributes the patterns in a round-robin way from the patterns sets with the largest number of occurrences to the least number of occurrences. In lines 5~6, we count the number of patterns whose starting character is c_j. This step forms the X set containing 256 elements corresponding to the 256 characters in the C set. Then, the X set and the C set are arranged in descending order (lines 7 and 8). Through this arrangement, the characters with the larger number of occurrences are placed towards the first position of the X set. From lines 9~13, the algorithm calculates the position of sets which the patterns are assigned to. These positions are calculated using the round-robin distribution. While part 1 helps distribute approximately the same number of patterns amongst STTs, there could be some variances in the resulting STT sizes.

(ii) Part 2 (Algorithm 2) balances the number of patterns in each STT by redistributing some patterns among STTs. A number of patterns are moved from the STT with the largest number of patterns to the STT with the least number of patterns. The nested while loops (lines 5~11) are used to make this transition. When entering the inner while loop, we check whether the length difference between two sets ($S_{p\max}$, $S_{p\min}$) with the maximum length and the minimum length is larger than a *threshold* value $((l_{p\max}/l_{p\min}) - 1.0 >$ *threshold*$)$. If so, we move one pattern from $S_{p\max}$

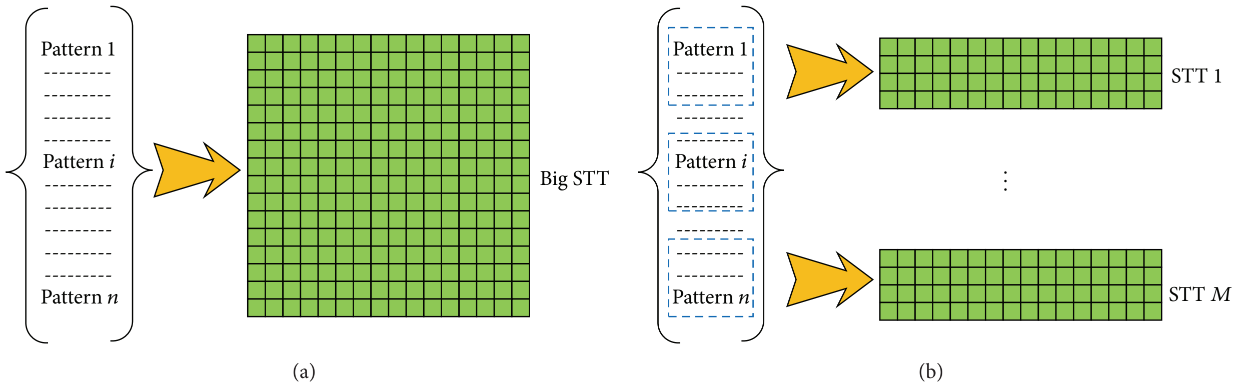


Figure 7: (a) Generating one large STT in previous approaches. (b) Generating multiple small STTs in our approach.

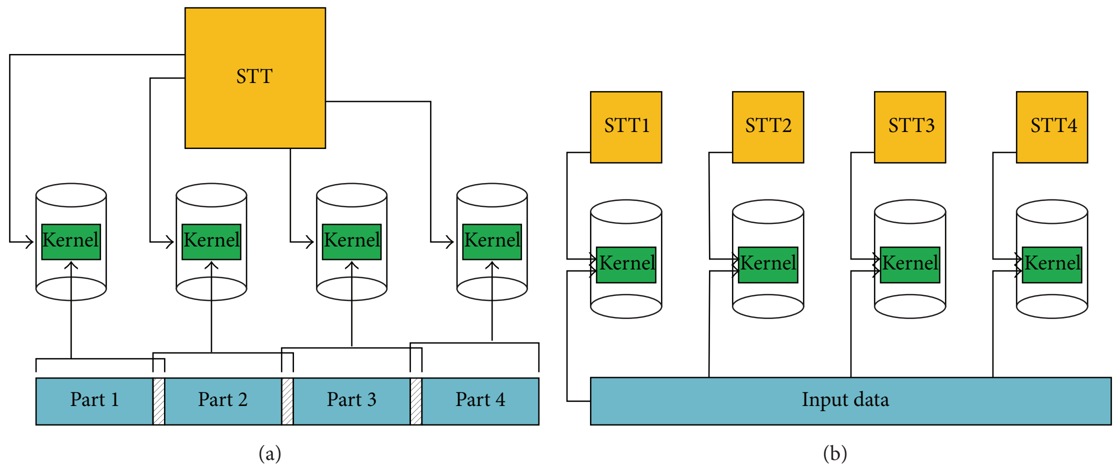


Figure 8: Comparison of our approach with previous approach: (a) previous approach using single large STT and input data partitioning; (b) our approach using multiple small STTs and no input data partitioning.

```
     Input: $n > 0, m > 0, P$, where,
        $n$: number of pattern strings
        $m$: number of STTs we want to generate ($m$: a multiple of 2)
        $P = \{p_1, \ldots, p_n\}$, a set of pattern strings
     Output: $S$, where,
        $S = \{s_1, \ldots, s_m\}$ and $s_k$ includes pattern strings belong to STT$_k$
(1)  Given $C = \{c_1, \ldots, c_{256}\}$, a set of 256 characters in ASCII table
(2)  Let
(3)     (i) $x_i$ be the number of pattern strings in $P$ which starts with character $c_i$
(4)     (ii) $x_i \in X$ and $X = \{x_1, \ldots, x_{256}\}$
(5)  foreach $c_j \in C$ do
(6)     Calculate $x_i$
(7)  Sort $X$ in the descending order
(8)  Arrange position of $c_j \in C$ based on the order of $x_i \in X$
(9)  foreach $p_i \in P$ which starts with character $c_j \in C$ do
(10)    if $j \div m = 0$ then
(11)       Assign $p_i$ to $s_k$, where $k = j \bmod m$
(12)    else
(13)       Assign $p_i$ to $s_k$, where $k = \max((j \bmod m), (m - (j \bmod m) - 1))$
```

Algorithm 1: Generating multiple small STTs.

```
     Input: $S$, where,
         $S = \{s_1, \ldots, s_m\}$ and $s_i$ includes pattern strings belong to $\mathrm{STT}_i$
     Output: $S$, where,
         $S = \{s_1, \ldots, s_m\}$ and $s_i$ has balanced length of patterns
(1)  Let
(2)      (i) $l_i$ be the total length of patterns in $\mathrm{STT}_i$, where $0 \le i \le m$
(3)      (ii) $p$max and $p$min be the positions of has-maximum-length $s$ and has-minimum-length $s$, respectively
(4)      (iii) threshold be the limit on the difference between the length of two STTs
(5)      do
(6)          while $(((l_{p\max}/l_{p\min}) - 1.0) > threshold)$ do
(7)              Move one pattern at the last position from $S_{p\max}$ to $S_{p\min}$
(8)              Update $l_{p\max}$ and $l_{p\min}$
(9)          Update the total length of patterns in sets of $S$
(10)         Update the $p$max and $p$min
(11) while $(((l_{p\max}/l_{p\min}) - 1.0) > threshold)$;
```

Algorithm 2: Balancing length of patterns in STTs.

to $S_{p\min}$ and update $l_{p\max}$ and $l_{p\min}$. We repeat this step until $((l_{p\max}/l_{p\min}) - 1.0)$ is equal to or less than the *threshold*. After the inner *while* loop exits, the positions and the total length of patterns of all sets are updated (lines 9~10). The *threshold* value is used again in the outer *while* loop to check the size difference of STTs. If the size difference between $S_{p\max}$ and $S_{p\min}$ is larger than the *threshold*, we enter the inner loop to rebalance the $S_{p\max}$ and $S_{p\min}$. The *threshold* value is chosen after conducting extensive experiments. We set the threshold = 15% for the total number of patterns smaller than 20,000 and 10% for the number of patterns larger than 20,000, respectively. Thus, using our algorithm, all the resulting STTs differ in size no larger than 15% for <20,000 patterns and 10% for >20,000 patterns.

An optimal distribution would generate the same number of patterns in all STTs and the sizes of all STTs get equal. Also the combined sum of all STTs gets minimized (as small as or close to the size of the original single large STT generated in the previous approach). Using our algorithm, the combined size of the multiple DFAs generated closely matches the size of the single large DFA which we will show in Section 5.4. Thus, our approach constructs multiple DFAs in a space-efficient way. Furthermore, it takes less time to construct multiple small DFAs compared with the time building one large STT using the previous approach which we will also show in Section 5.4.

Our DFA partitioning algorithm is time efficient because of the linear complexity of both Algorithms 1 and 2. Let us assume the following:

(1) n: the number of patterns;

(2) m: the number of parts (DFAs or STTs) we want to partition.

Algorithm 1 consists of four tasks:

(i) calculate values in X (lines 5~6),

(ii) sort values in set X (line 7),

(iii) sort values in C (line 8),

(iv) assign patterns to sets (lines 9~13).

We can calculate the execution times of the tasks as follows:

(1) (ii) and (iii) are always constants $O(1)$ (In fact their time complexities are $256 \times \log(256) = O(1)$.)

(2) (i) and (iv) = $256 \times n$, thus $O(n)$.

Thus, the complexity of Algorithm 1 is $O(n)$.

After Algorithm 1 is executed, we divide the number of patterns into m parts. Thus,

(1) each part has n/m patterns in the best case,

or

(2) one set is close to n patterns and the others are almost empty for the worst case.

Algorithm 2 has two loops. In the worst case, the inner loop executes $n/2$ times, because one STT is close to n patterns and the other STT is almost empty. In the outer loop, we need to update the length of sets and select two sets with the maximum length and the minimum length for the next step. This process selects two sets from m sets. Thus, the execution time is proportional to $m \times (m-1)$. Total of execution time = $m \times (m-1) \times (n/2)$. Thus, the complexity of Algorithm 2 is $O(n)$, because m is a constant (number of STTs).

4.3. Further Performance Optimizations. Besides the DFA partitioning based parallelization approach described above, we apply further performance optimization techniques to our GPU implementation. They are mostly taken from our earlier work [15].

(i) The input data is stored in a sequential fashion in the global memory. While loading a data block, we let multiple threads generate memory access which fall within the 128-bytes range so that these access get combined into one request and sent to the memory [10]. This saves the memory bandwidth a lot and improves the performance [15].

(ii) After the input data gets loaded from the global memory, each thread accesses a chunk of the input stored in the shared memory. When multiple threads attempt to read their own data chunk simultaneously which are spread over multiple banks of the shared memory, it will result in a lot of bank conflicts. We use a store scheme through which a chunk of data loaded from the global memory gets divided up into 4-bytes units and stored in the shared memory at the addresses which are mapped to the consecutive shared memory banks in a diagonal way. This store scheme avoids any bank conflicts and results in a conflict-free load from the shared memory banks [15].

(iii) The GPU is executing in multithreaded fashion. Having multiple threads available for the simultaneous execution can theoretically tolerate the off-chip memory (global memory, texture memory, etc.) access latencies which take a few hundred cycles. The bandwidth to the off-chip memory, however, has a limit. If there are too many concurrent access to the off-chip memory, it can lead to congestion in the memory access paths and further lengthen the latencies. Furthermore, the increased number of threads leads to the increased cache misses [16]. Therefore, finding an optimal number of threads to effectively hide the off-chip memory latencies while efficiently utilizing the large number of thread processor cores and the memory bandwidth is crucial for obtaining high performance. We attempt to find and schedule an optimal number of parallel threads onto the hardware thread blocks and the thread processor cores by almost exhaustively searching various input chunk sizes to be assigned to each thread [15].

We will show the performance benefits of the above optimization techniques besides our multiple STTs based approach later in Section 5.2.2.

5. Experimental Results

In this section, we first explain our implementation details for the DFA partitioning based parallel AC algorithm. Then, we present the experimental results on the Tesla K20 GPU and the Intel Xeon Phi. In order to prove the space efficiency of our approach, we also compare the size of the single STT (previous approach) versus the sum of multiple smaller STTs (our approach). We also show the cost comparisons of building multiple STTs in our approach compared with one large STT in the previous approach. Thus, we prove the time efficiency of our approach in building the STT also.

5.1. Implementation Details. Our experiments were conducted on a system incorporating the host Intel Xeon multicore processor (6-core 2.0 Ghz Intel Xeon E5-2650) with 20 MB level-3 cache, the Nvidia Tesla K20 GPU with 5 GB device memory, and the Intel Xeon Phi with 61 x-86 cores with 8 GB on-board memory. We also used the Centos 5.5 Linux. In the following subsections, we describe the methodology to generate the test input data and the pattern data. We also explain details about our parallel implementations.

Table 1: Input types.

Input	Contents	Size
Random text input	Random text input taken from magazines such as TIME and BBC	20~500 MB
Dictionary input		
Dict_Input_S	All pattern strings in the dictionary are listed once in the **Dict_Input_S** after shuffling	1~512 KB
Dict_Input_L	**Dict_Input_S** is replicated and concatenated to form **Dict_Input_L**	20~500 MB

5.1.1. Test Data Generation. In order to generate the random input data sets and the reference pattern data sets used in our experiments, we first collected 50 GB of the data from a variety of English magazines such as TIME and BBC, among many others. Then, we extracted the random input data and the pattern data from the collected data. We used the input data sizes in the range of 20 MB~500 MB. The number of patterns used is in the range of 100~50,000. We also generated a special input data, **Dict_Input**. There are two kinds of **Dict_Input**: (1) **Dict_Input_S**, where the contents of the input data are generated directly from all pattern strings in the dictionary. Thus, the **Dict_Input_S** has a small size. For example, when the number of patterns in the dictionary is 100 (50,000) and the average pattern length is 10 characters, the size of **Dict_Input_S** is around 1 KB (512 KB). (2) In **Dict_Input_L**, the contents of the input are generated by copying and concatenating all patterns in the dictionary to make the input size large. Information about the input data is summarized in Table 1. The characteristics of the pattern sets are given in Table 2.

5.1.2. Parallel Implementations. For the implementation on the Tesla K20 GPU, we used the shared memory to load the input text data. We also show the implementation without using the shared memory in order to quantify the benefit of using the shared memory. We describe both implementations below:

(i) P-1: the global memory only (or no shared memory) implementation (see Figure 9) copies the input text data into the global memory. Then, the actively used portion of the input data is cached into the on-chip caches (L2 and L1 caches) automatically, but it is not cached in the shared memory explicitly. The STT data is attached to the texture memory and the actively used portion of the STT data is cached in the L2 cache and the read-only data cache. In this implementation, the L2 cache is used by both the input data and the STT data. Thus, the performance effects of our cache locality-centric parallelization approach are more distinguished as the effective L2

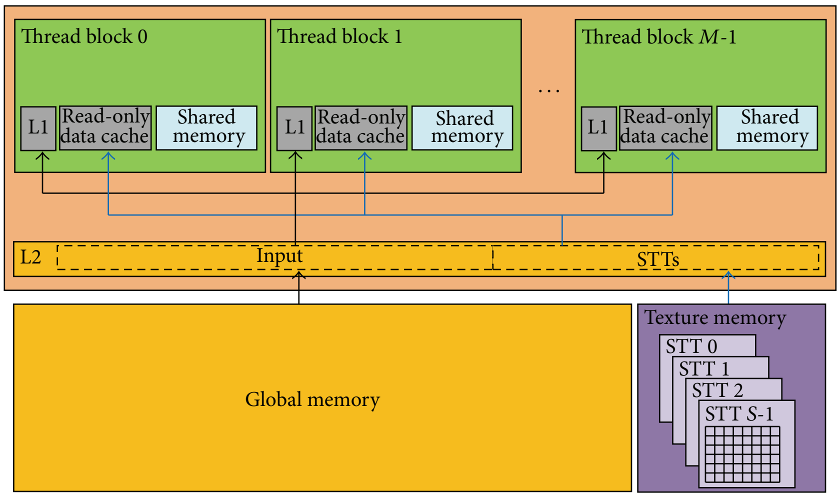


Figure 9: Illustration of global memory only (P-1) implementation.

Table 2: Characteristics of pattern sets.

Number of patterns	Avg. length	1 STT		4 STTs		8 STTs	
		Number of states	Size (KB)	Number of states/STT	Size (KB)	Number of states/STT	Size (KB)
100	7.58 bytes	573	575.5	148	148.6	80	80.3
						78	78.3
				146	146.6	76	76.3
						75	75.3
				142	142.6	74	74.3
						72	72.3
				142	142.6	72	72.3
						70	70.3
				Sum: **578**	Sum: **580.25**	Sum: **597**	Sum: **599.33**
5000	8.28 bytes	8606	8640.0	2177	2185.5	1104	1108.3
						1088	1092.3
				2167	2175.5	1086	1090.2
						1082	1086.2
				2145	2153.4	1076	1080.2
						1077	1081.2
				2139	2147.4	1082	1086.2
						1069	1073.2
				Sum: **8628**	Sum: **8661.7**	Sum: **8664**	Sum: **8697.84**
50000	8.58 bytes	26369	26471.2	6612	6637.8	3412	3425.3
						3406	3419.3
				6610	6635.8	3366	3379.1
						3298	3310.9
				6608	6633.8	3286	3298.8
						3282	3294.8
				6603	6628.8	3278	3290.8
						3237	3249.6
				Sum: **26433**	Sum: **26536.25**	Sum: **26565**	Sum: **26668.77**

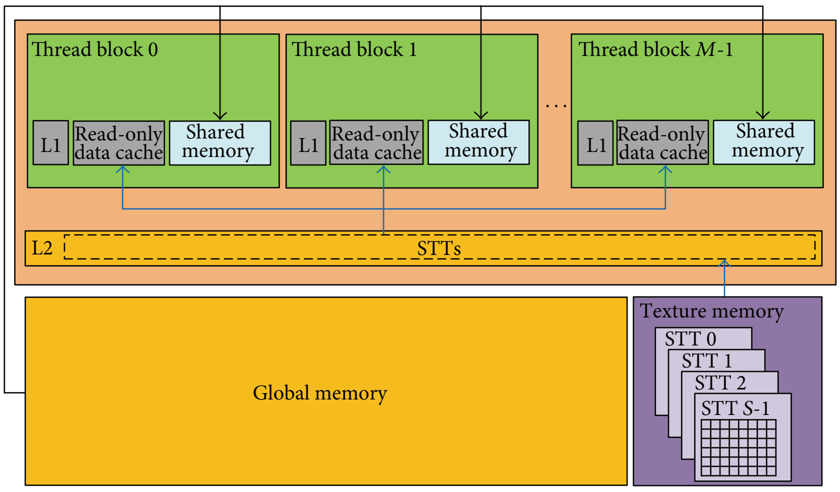


FIGURE 10: Illustration of shared memory (P-2) implementation.

cache sizes used by the input data and the STT data get reduced.

(ii) P-2: the shared memory implementation (see Figure 10) loads the input data from the global memory into the on-chip shared memories explicitly by the programmer. The STT data is placed in the texture memory and the actively used portion is loaded into the L2 cache first and then in the read-only data cache. Thus, the L2 cache is used by the STT data only. In this implementation, the input data caching is more efficient compared with the P-1 implementation which relies on the automatic caching. The STT data caching also becomes more efficient because the L2 cache is now dedicated to the STT data.

In order to implement the multiple STTs based string matching of the AC algorithm, we use the CUDA stream feature. Unlike the stream feature in the Fermi architecture where only 16-way concurrency is supported and the streams are multiplexed into a single queue, the Kepler K20 allows 32-way concurrency and one work queue per each stream. This leads to the concurrency at the full-stream level and no inner-stream dependency [17]. Thus, we create multiple CUDA streams equal to the number of STTs (4 or 8 streams in our case) and assign each stream to each pattern matching task where a smaller STT is referenced for possible matches with the whole input data. This makes sure that the pattern matching tasks can be performed concurrently using multiple STTs. In order to store the STTs in the texture memory, we use a new feature called the texture objects (or bindless textures since they do not require the manual binding/unbinding) from the Kepler architecture (with CUDA 5.0 or later). The number of texture objects created is equal to the number of STTs. We only pass these texture objects to the kernel for use.

Pseudocode 2 shows the pseudocode of our implementation on the K20 GPU. First, the texture objects are created to bind to the STTs (lines 2~7). Next, we create a number of streams (lines 9~14). Then, the streams cooperate to copy the input data to the device memory. (Each stream copies one data segment (lines 16~19).) After the input data is copied, each stream performs the pattern matching task using its input data and the STT data (lines 21~23). In the end, the results are copied back to the host side, and then the streams are destroyed (lines 25~31).

5.2. Performance Comparisons on K20 GPU.

We show performance results of our approach compared with previous approaches. In all experiments conducted, we show the time in conducting pattern matching operations only because the second phase of the AC algorithm was parallelized.

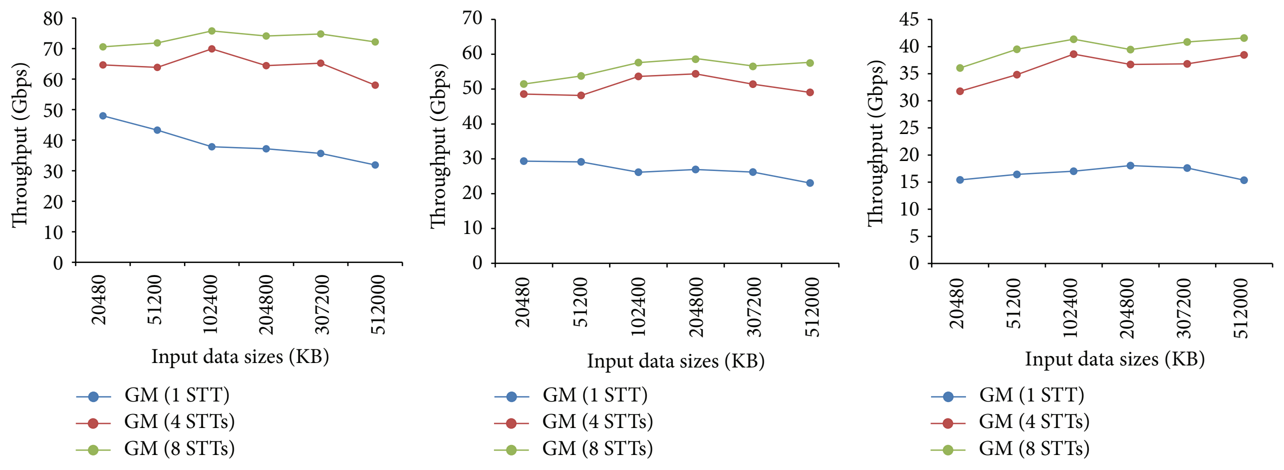
5.2.1. Performance Benefit of Our Approach over the Previous Approach.

Figure 11 shows the throughput performance of the P-1 (global memory only) implementation for a range of input data sizes (20~500 MB) and for a range of the numbers of patterns (100, 5000, and 50000) measured in Gbps. For performance comparisons, we also implemented the previous approach where single large STT and the partitioned input text data pieces are used for conducting the parallel pattern matches. (The graph marked with 1 STT shows the performance for the previous approach.)

(i) Our new approach (using 4, 8 STTs) outperforms the previous approach where single large STT is used. As the input data size increases, the performance of our approach improves steadily up to 100 MB and then starts to saturate. In P-1 approach, the L2 cache is shared by both the input data and the STT; therefore, as the input data size increases, the pressure at the L2 cache increases accordingly and leads to performance saturations. However, the performance gap between our approach and the previous approach gets widened. (We will show later that. In the P-2 experimental results, the performance saturation with the increase in the input data size disappears as the L2 cache is used by the STTs only.)

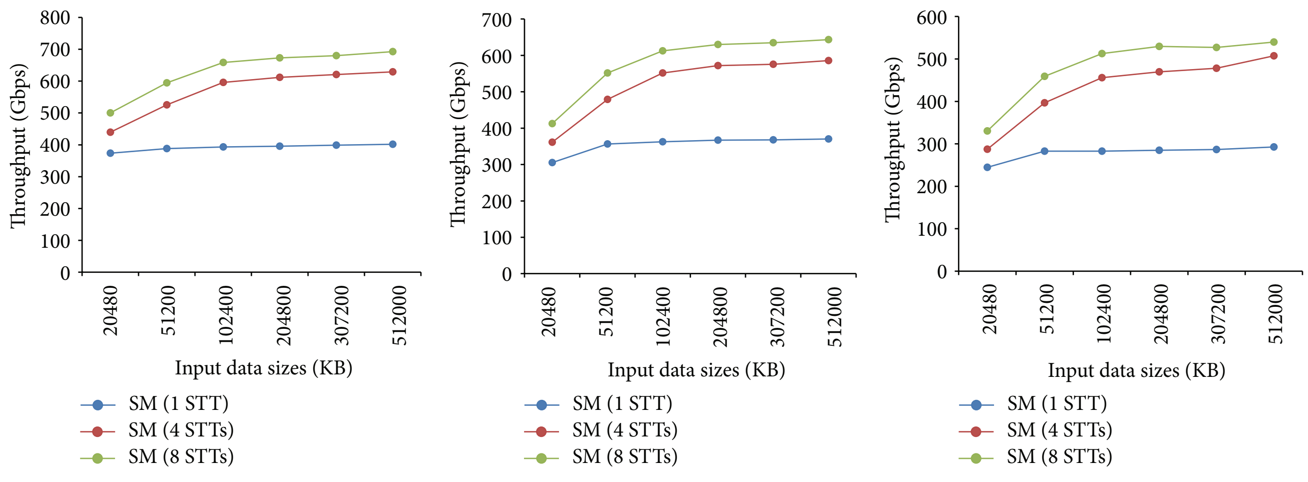
```
...
cudaArray *cuda_arrays = (cudaArray**) malloc (num_of_stts * sizeof(cudaArray*))
cudaTextureObject_t* textObj = (cudaTextureObject_t*) malloc (num_of_stts * sizeof
  (cudaTextureObject_t));
for (int i = 0; i < num_of_stts; i++) {
  cuda_arrays[i] = generate_cuda_array (get_dfa_matrix(i), get_dfa_width(i),
    get_dfa_height(i));
  textObj[i] = generate_texture_objects (cuda_arrays[i]);
}
...
cudaStream_t *streams =
(cudaStream_t*) malloc (nstreams * sizeof(cudaStream_t));
/* create multiple CUDA streams */
for (int i = 0; i < nstreams; i++) {
  cudaStreamCreate (&(streams[i]));
}
//copy data to GPU memory, each stream copies one segment
for (int i = 0; i < nstreams; i++) {
  long in_offset = i * input_len/nstreams;
  cudaMemcpyAsync (d_input + in_offset, h_input + in_offset, input_len * sizeof
    (char)/nstreams, cudaMemcpyHostToDevice, streams[i]);
}
/* each stream processes input data with each dfa (texObj [i]) */
for (int i = 0; i < nstreams; i++) {
  matching <<<blocks, threads, sm_size, streams[i]>>>(texObj[i], d_input,
  input_len, pattern_max_len, d_output, output_len);
}
/* copy results back to host CPU, each stream copies one segment */
for (int i = 0; i < nstreams; i++) {
  long out_offset = i * output_len/nstreams;
  cudaMemcpyAsync (h_output + out_offset, d_output + out_offset, output_len * sizeof
    (int)/nstreams, cudaMemcpyDeviceToHost);
}
for (int i = 0; i < nstreams; i++) {
  cudaStreamDestroy (streams[i]);
}
...
```

Pseudocode 2: Pseudocode using multiple CUDA streams to implement our approach.



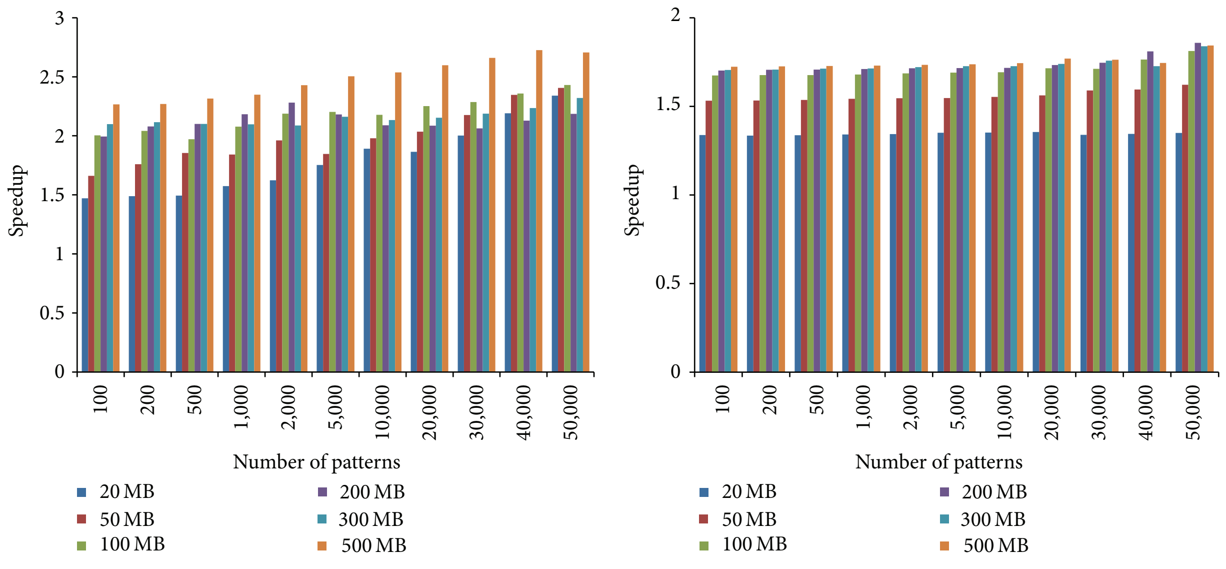
(a) 100 patterns (b) 5,000 patterns (c) 50,000 patterns

Figure 11: Throughput (Gbps) comparisons using P-1 implementation.



(a) 100 patterns (b) 5,000 patterns (c) 50,000 patterns

FIGURE 12: Throughput (Gbps) comparisons using P-2 implementation.



(a) (b)

FIGURE 13: Speedup comparisons: (a) using global memory only (P-1) implementation and (b) using shared memory (P-2) implementation.

(ii) When the number of patterns increases, the throughput performance gets lower in all the cases because the cache misses increase with the increase in the number of patterns. The larger number of patterns affects the performance of the previous approach more. Thus, the performance gap is widened.

Figure 12 shows the throughput performance of the P-2 (using the shared memory) implementation for a range of input data sizes (50~500 MB) and for a range of the numbers of patterns (100, 5000, and 50000) measured in Gbps.

(i) As in the P-1 results, the new approach (using 4, 8 STTs) outperforms the previous approach (1 STT). As the input data size increases, the performance gap gets larger. Our approach improves the performance further with the increase in the input data sizes up to 500 MB.

(ii) With the increase in the number of patterns, the throughput performance gets lower in all the cases. However, the performance gap between our approach and the single STT approach gets larger as in the P-1 implementation.

(iii) The best performance for the P-1 implementation is 75.8 Gbps and for the P-2 it is 692.7 Gbps. Thus, the shared memory implementation gives up to ~9.14 times better performance than the P-1 implementation.

Figure 13 shows the speedup of our approach over the previous approach.

(i) Figure 13(a) shows that, using the P-1 implementation, our approach gives the speedup in the range of 1.47~2.73 for the data sizes ranging from 20 MB to 500 MB

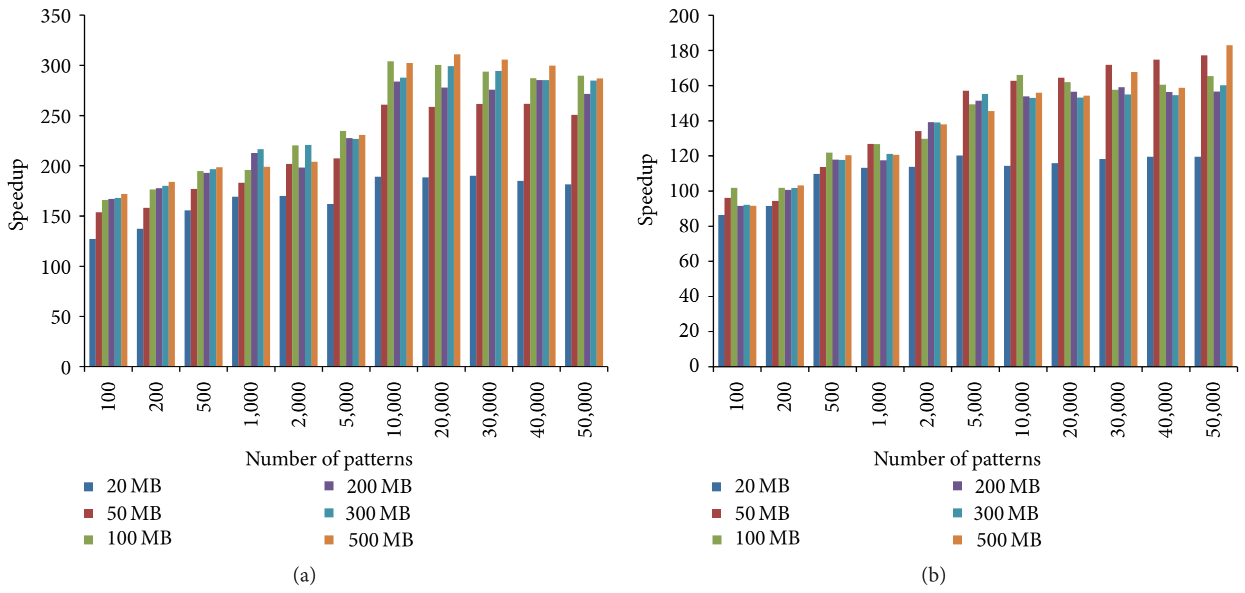


FIGURE 14: Speedup of our approach using the shared memory (P-2) implementation (a) over single CPU run (b) over 6-thread parallel version.

and the number of patterns ranging from 100 to 50,000. As the input data size increases, the speedup improves also. The number of patterns has direct performance impacts for all the input sizes up to 500 MB.

(ii) Figure 13(b) shows the speedup of our approach using the shared memory (P-2) implementation. Our approach results in the speedup of 1.34~1.86 for the data sizes ranging from 20 MB to 500 MB and the number of patterns ranging from 100 to 50,000. As the data size increases, the speedup increase shows up to 200 MB. Beyond 200 MB, the speedup increase saturates. As the number of patterns increases, the speedup increases accordingly.

(iii) The overall speedup for P-2 is lower, however, than the speedup for P-1. In the P-2 using the shared memory, the L2 cache is dedicated to the STT access. Thus, it can capture larger portions of the working set for the large single STT of the previous approach used in the P-2 implementation. When the number of patterns increases to 50,000, however, we see a sudden increase in the speedup. The working set of the large single STT used in the previous approach starts to overflow the L2 cache. This shows that the effectiveness of our approach is the larger for the larger number of patterns.

Figure 14(a) shows the speedup results of the P-2 implementation over single CPU core (out of 6-core 2.0 Ghz Intel Xeon E5-2650) run. The speedup ranges within 127~311. The speedup of P-2 implementation over the 6-thread parallel version ranges within 86~183 as shown in Figure 14(b).

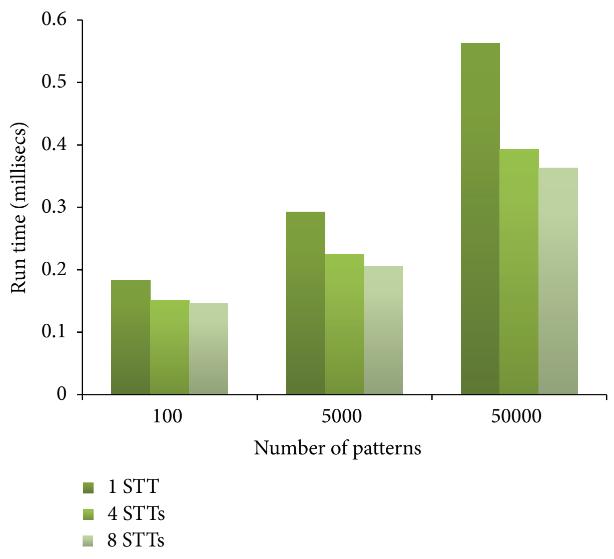
Figures 15(a) and 15(b) present the run times of using 1 STT (previous approach) and 4, 8 STTs (our approach) as we use the **Dict_Input_S** for both the P-1 and P-2 implementations. The results show that the run time of our approach (multiple STTs) is smaller than the previous approach (single STT) for both implementations P-1 and P-2. For P-1 implementation, the performance of using 4 STTs is better than 1 STT by 18.15%, 23.41%, and 30.14% for 100, 5,000, and 50,000 patterns. The performance of 8 STTs is better than 1 STT by 20%, 29.9%, and 35.4% for 100, 5,000, and 50,000 patterns. Figure 16 shows the throughput performance compared with the previous approach as we use **Dict_Input_S** for the P-1 and the P-2 implementations, respectively.

Also, Figures 17 and 18 show the throughput performance as we use **Dict_Input_L** for the P-1 implementation and the P-2 implementation, respectively. As shown in both figures, our approach outperforms the previous approach. In fact, the performance improvements of our approach show the similar trends that we observed when using the random input data.

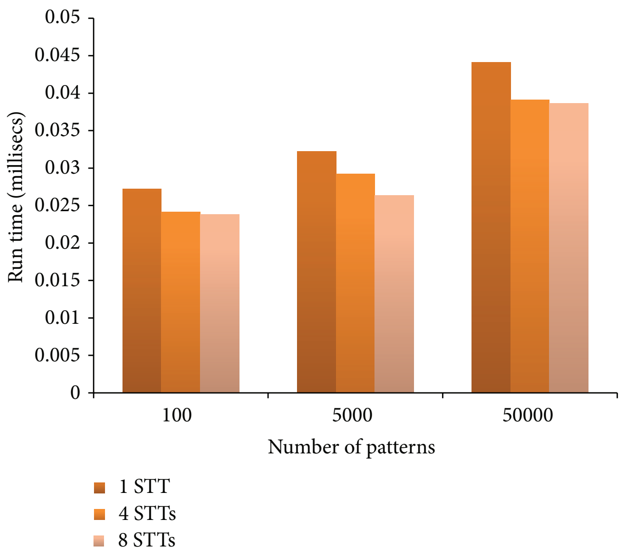
5.2.2. Effectiveness of Further Performance Optimization Techniques. As mentioned in Section 4.3, further performance optimization techniques are also applied in our implementation besides the STT partitioning technique.

(i) Figure 19 shows the speedup of the P-2 implementation with and without the shared memory bank conflicts. As shown, the bank conflicts affect the performance for the P-2 implementation. Avoiding the bank conflicts, the performance improves by 1.72x~4.48x for the number of patterns in the range of 100~50000.

(ii) In order to maximize the performance benefits of the multithreading capability of the GPU, we attempt to find the best number of threads/block for a given data size. For this, we conducted extensive performance tests. For the P-1 implementation, we changed

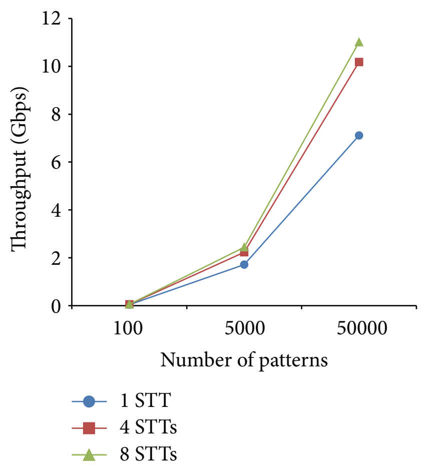


(a) P-1 implementation

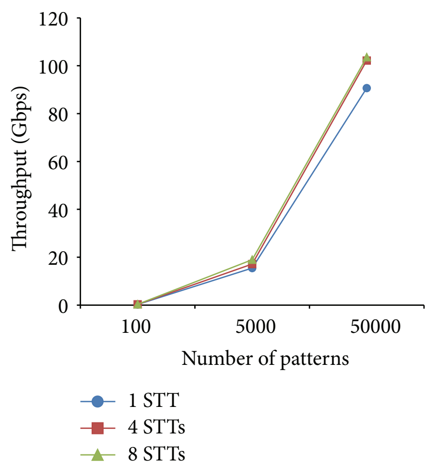


(b) P-2 implementation

FIGURE 15: Run time comparisons for **Dict_Input_S**: (a) P-1 implementation; (b) P-2 implementation.

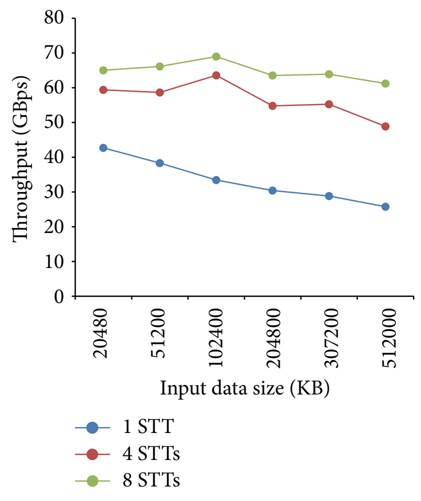


(a) P-1 implementation

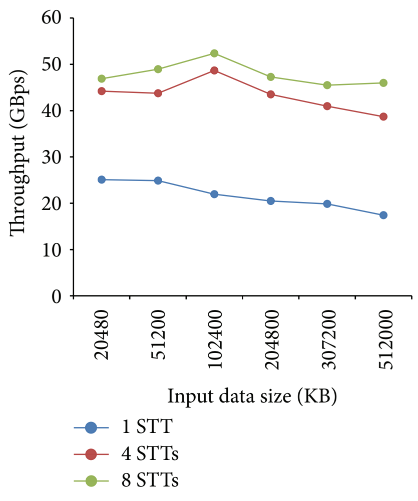


(b) P-2 implementation

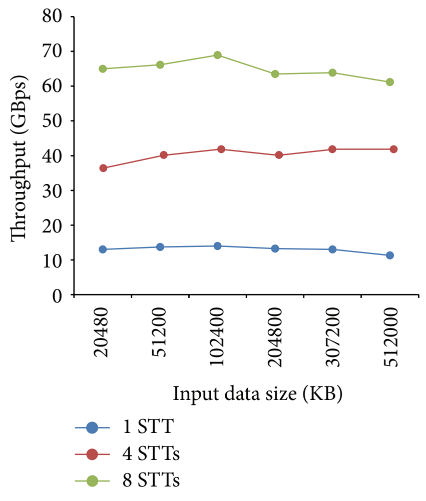
FIGURE 16: Throughput comparisons for **Dict_Input_S** using P-1 and P-2 implementations.



(a) 100 patterns



(b) 5,000 patterns



(c) 50,000 patterns

FIGURE 17: Throughput comparisons for **Dict_Input_L** using P-1 implementation.

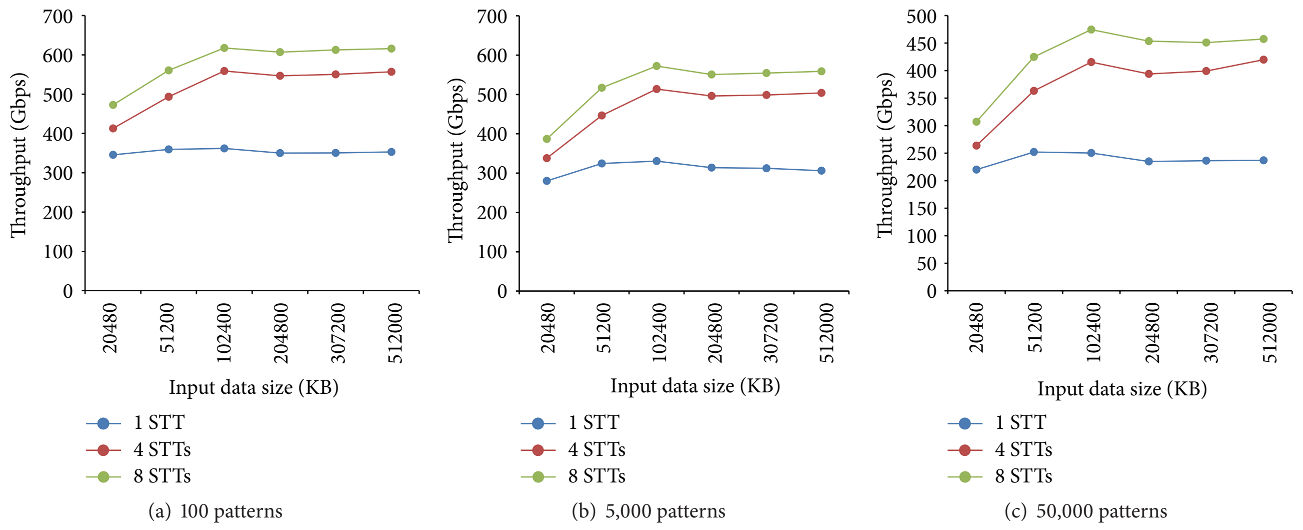


FIGURE 18: Throughput comparisons for **Dict_Input_L** using P-2 implementation.

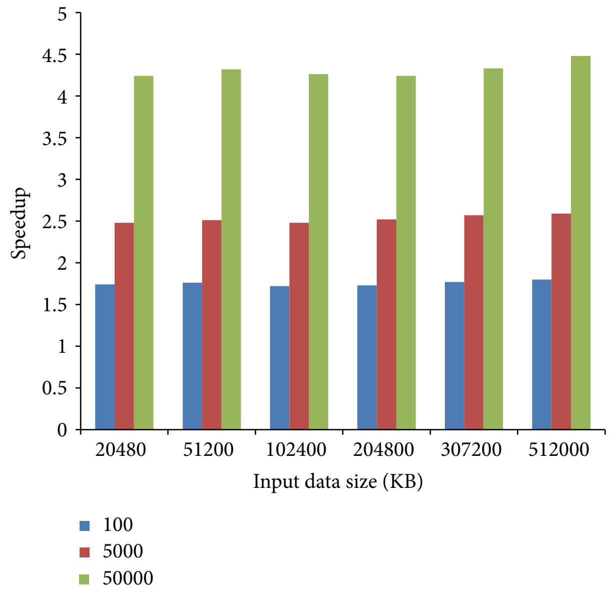
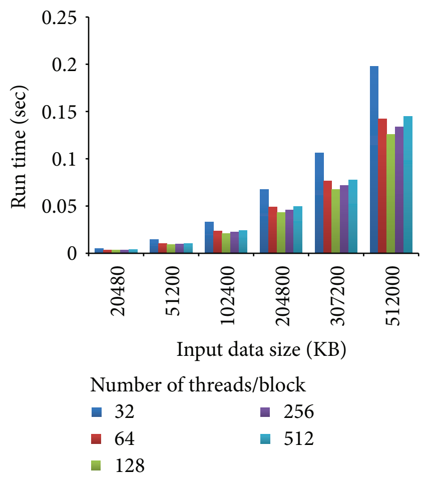


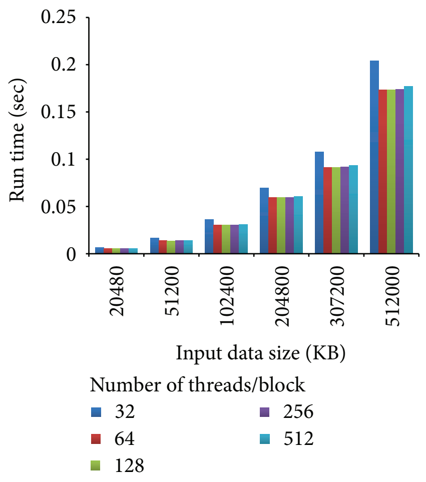
FIGURE 19: Speedup of avoiding bank conflicts using P-2 implementation.

the number of threads/block while keeping the same data size. Figure 20 presents the run time of the P-1 implementation with different numbers of threads/block. The results show that 128 threads/block gives the best performance for all the numbers of patterns (100, 5000, and 50000). For the P-2 implementation, the number of threads/block depends on the size of the shared memory. Thus, we need to carefully decide the size of shared memory. (The physical shared memory size is set to 48 KB in our experiments. However, we set a logical shared memory size for a block of threads smaller than 48 KB considering that the multiple blocks will execute in the multithreaded fashion on the same hardware thread block.) Through experiments, we observed that setting the shared memory size as 8 KB gives the best performance. Figure 21 presents the results for the P-2 implementation. We chose the number of threads/block in the range of 32~512. With 100 and 5000 patterns, 256 threads/block gives the best performance (Figures 21(a) and 21(b)) while 512 threads/block gives the best performance for 50000 patterns (Figure 21(c)). Thus, we use these numbers.

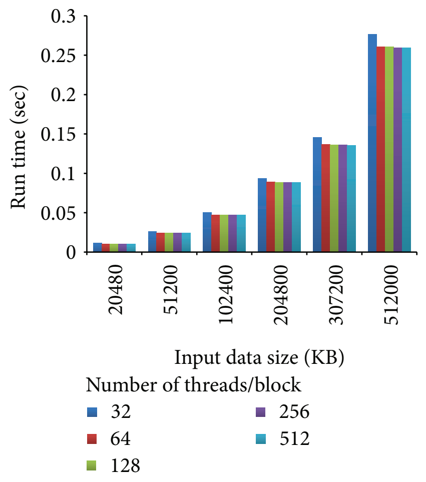
5.3. Performance Comparisons on Xeon Phi. For the implementation on the Xeon Phi, we first construct the STT(s) on the host Intel CPU. Since the memory hierarchy of the Xeon Phi is not as sophisticated as the GPU's memory hierarchy, both the input data and the STT data are copied directly to the on-board memory of the Xeon Phi. As explained earlier in Section 2, the Xeon Phi has two working modes: native mode and offload mode. We use the offload mode in our experiments. In the offload mode, a program running on the host can optionally launch or "offload" portions of the code to the Xeon Phi coprocessor. The programmer can identify which lines or portions of the code should be offloaded and can invoke the OpenMP threads. While conducting the AC algorithm, we offloaded the pattern matching procedure to the Xeon Phi coprocessor to take advantage of the coprocessor's multithreading capability. The input data and the STT data are used from the beginning and not changed during the program execution. Thus, their memories are allocated and copied to the coprocessor only one time at the offload stage. In addition, they are shared among multiple running threads by using *shared* clause. A large number of threads were created to process pattern matching tasks where each thread processes one chunk of input text. We used the dynamic scheduling to balance the workload among threads. In order to distribute the threads as evenly as possible across the entire system, the scatter affinity was applied.



(a) 100 patterns

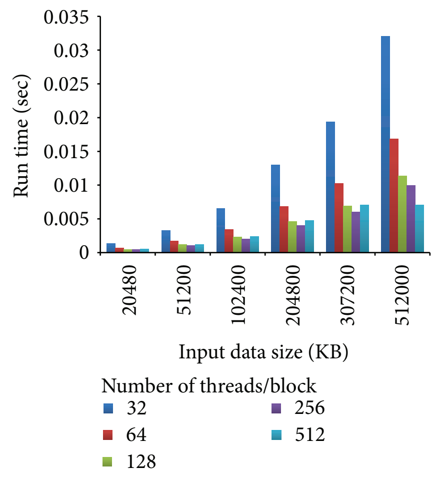


(b) 5,000 patterns

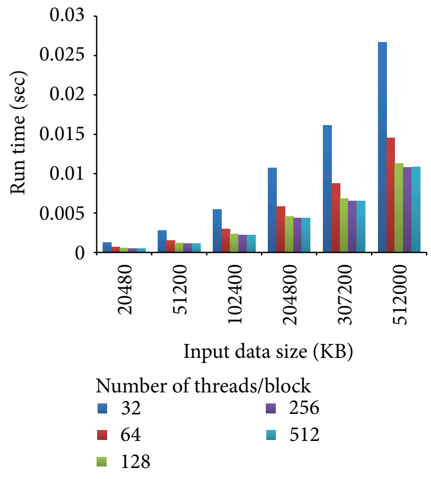


(c) 50,000 patterns

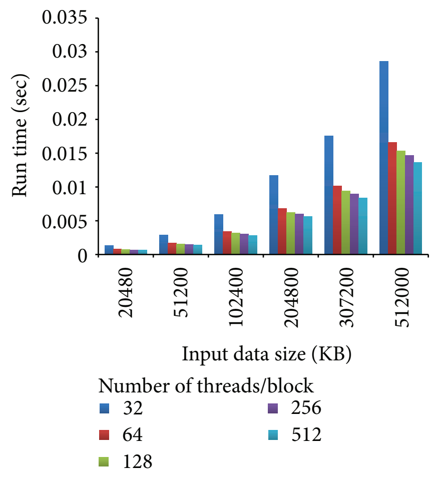
FIGURE 20: Run time of P-1 with different numbers of threads/block.



(a) 100 patterns



(b) 5,000 patterns



(c) 50,000 patterns

FIGURE 21: Run time of P-2 using shared memory with different numbers of threads/block.

Figure 22 shows the speedup of our approach using 4 STTs over the previous approach on the Intel Xeon Phi. The speedup ranges within 1.60~2.00. The speedup increases with the increase in the input data sizes. The Xeon Phi results confirm the benefit of our approach to reduce the working set size of individual STT which has irregular access patterns. Therefore, the partitioning of the STT significantly reduces the number of cache misses and leads to the improved performance. The Xeon Phi supports up to 4-way multithreading; thus, we can exercise up to 240 threads for the experiments considering that there are 60 compute cores. However, the best performance was obtained when we used 2- or 3-way multithreading per core.

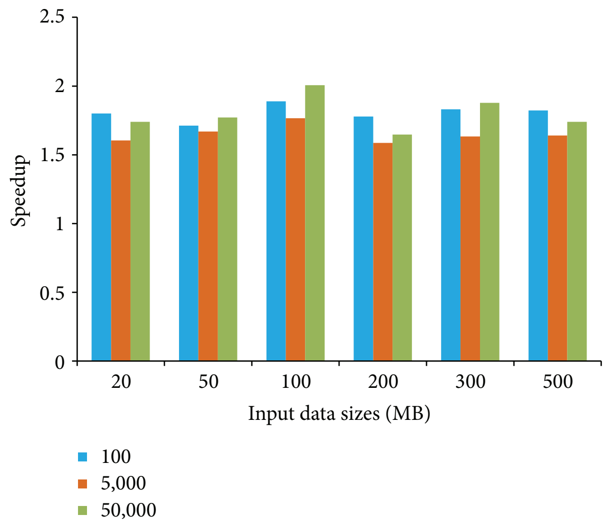


FIGURE 22: Speedup of our approach on the Intel Xeon Phi.

5.4. STT Size and Building Cost Comparisons. In order to evaluate the space efficiency of our approach, we measured the size of the single large STT in the previous approach and the sum of the multiple small STTs generated using our

Table 3: Size comparison of single STT and 4 STTs.

Number of patterns	100	5000	50,000
Size of single STT	576 KB	8,640 KB	26,472 KB
Combined size of 4 STTs	580 KB	8,662 KB	26,537 KB

Table 4: The building time of the single STT, 4 STTs, and 8 STTs in second.

Number of patterns	Single STT	4 STTs	8 STTs
100	0.2249	0.1195	0.099
200	0.3883	0.2209	0.1828
500	0.8046	0.4383	0.3716
1000	1.2972	0.6755	0.5953
2000	1.9574	1.0011	0.8747
5000	3.5452	1.9299	1.6624
10000	5.0627	2.786	2.7159
20000	7.443	4.0041	3.5579
30000	9.3607	5.1498	4.7219
40000	11.1045	6.1751	5.2814
50000	12.4503	6.798	5.6173

approach. Table 3 lists the size of single STT and the combined size of 4 STTs generated with different numbers of patterns. For 100, 5000, and 50000 patterns, the combined size of 4 STTs is only 0.88%, 0.24%, and 0.25% larger than the size of the single STT, respectively. Thus, our approach is space-efficient.

We also measured the time to build the single STT in the previous approach and the multiple STTs in our approach. The run time comparisons are shown in Table 4. In fact, the STT building cost decreases as the number of STTs increases. For example, when the number of patterns is 50000, the cost of building 8 STTs is 2.22 times faster than building single STT. Therefore, our approach is more time efficient in building the DFAs (or STTs) than the previous approach.

6. Previous Research

The AC pattern matching algorithm has been previously applied in various application areas. In fact, network and computer security and bioinformatics are two major areas where the AC algorithm is intensively applied.

In the area of network intrusion detection, Yang and Prasanna [18] proposed a head-body finite automata (HBFA) approach which implements the string pattern matching based on the AC algorithm. The HBFA implementation matches the dictionary up to a predefined prefix length using the Head-DFA. This reduces the run time memory by >20x and the performance scales up to 27x on a 32-core Intel many-core chip. Giorgos Vasiliadis et al. [4] presented an intrusion detection system based on the Snort open-source NIDS called Gnort. In parallelizing the pattern matching on the GPU, they relied on partitioning the input data amongst the thread blocks for the parallel AC string matching instead of partitioning the set of string patterns as in our approach. They did not use the shared memory in loading the input data. Instead, they replied on the automatic caching at the L1 cache of the GPU. Smith et al. [19] implemented a regular expression matching algorithm on the GPU based on the (extended) Deterministic Finite Automata. Jacob and Brodley [20] also proposed a solution to offload the signature matching computations to the GPU. They used the Knuth-Morris-Pratt (KMP) single string matching algorithm instead of the AC algorithm.

In the area of bioinformatics, Tumeo and Villa [8] presented an efficient implementation of the AC algorithm for accelerating DNA analysis on heterogeneous GPU clusters. Zha and Sahni [5] proposed a parallel AC algorithm on a GPU. Like in [4], they partitioned the input data amongst the thread blocks for the parallel string matching. They used the shared memory for loading the input data; however, they did not consider avoiding the shared memory bank conflicts.

Other previous researches had ported multistring matching applications to the IBM Cell Broadband Engine (BE). Scarpazza et al. [21, 22] ported the AC-opt version to the Cell BE. Zha et al. [23] proposed a technique to compress AC automaton to be used on the Cell BE. The compressed AC automaton, however, leads to indirect access in deriving the next state for a given state and a character which affects the performance. Villa et al. [24] presented a software based parallel implementation of the AC algorithm on a 128-processor multithreaded shared memory Cray XMT. They utilized the particular features of XMT multithreaded architecture and algorithmic strategies to minimize the number of memory references and reduce the memory contention in order to archive high performance and scalability. They also extended this work by characterizing the performance of the AC algorithm on various shared memory and distributed memory architectures in [6].

7. Conclusion

In this paper, we proposed a high performance parallelization of the AC algorithm which significantly improves the cache locality for the irregular DFA access on the many-core accelerator chips including the Nvidia GPU and the Intel Xeon Phi. Our parallelization approach partitions the given set of string patterns to generate multiple sets of a small number of patterns. Then, we constructed multiple small DFAs instead of constructing single large DFA in a space-efficient way. Using multiple small DFAs, intensive pattern matching operations are concurrently conducted with respect to the whole input text string. This significantly reduces the size of the cache footprints for the STT data on each core's cache and thus leads to significantly improved cache performance. Experimental results on the Nvidia Tesla K20 GPU show that our approach delivers up to 2.73 times speedup compared with the previous approach using single large DFA and up to 692.7 Gbps throughput performance. Compared with single CPU performance, we obtained a speedup in the range of 127~311. The speedup over the 6 OpenMP threads parallel version running on 6 CPU cores is in the range of 86~183. Experimental results on the Intel Xeon Phi with 61 x-86 cores also show up to 2.00 times speedup compared with the previous approach.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors would like to extend their gratitude to the anonymous reviewers for their valuable feedback about their papers. Their comments have greatly helped the authors in improving the quality and the presentation of their paper. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (Grant no. 2012R1A1A2042267). This work was supported by the Institute for Information & Communications Technology Promotion (IITP) Grant funded by the Korean government (MSIP) (no. B0101-15-0104, The Development of Supercomputing System for the Genome Analysis).

References

[1] Clam Antivirus, http://www.clamav.net.

[2] T.-H. Lee, "Generalized aho-corasick algorithm for signature based anti-virus applications," in *Proceedings of the 16th International Conference on Computer Communications and Networks (ICCCN '07)*, pp. 792–797, Honolulu, Hawaii, USA, August 2007.

[3] A. Tumeo, O. Villa, and D. Sciuto, "Efficient pattern matching on GPUs for intrusion detection systems," in *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF '10)*, pp. 87–88, ACM, Ischia, Italy, May 2010.

[4] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: high performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15–17, 2008. Proceedings*, vol. 5230 of *Lecture Notes in Computer Science*, pp. 116–134, Springer, Berlin, Germany, 2008.

[5] X. Zha and S. Sahni, "GPU-to-GPU and host-to-host multipattern string matching on a GPU," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1156–1169, 2013.

[6] A. Tumeo, O. Villa, and D. G. Chavarria-Miranda, "Aho-corasick string matching on shared and distributed-memory parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 3, pp. 436–443, 2012.

[7] M. C. Schatz and C. Trapnell, *Fast Exact String Matching on the GPU*, Center for Bioinformatics and Computational Biology, 2007.

[8] A. Tumeo and O. Villa, "Accelerating DNA analysis applications on GPU clusters," in *Proceedings of the 8th IEEE Symposium on Application Specific Processors (SASP '10)*, pp. 71–76, IEEE, Anaheim, Calif, USA, June 2010.

[9] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the Association for Computing Machinery*, vol. 18, pp. 333–340, 1975.

[10] NVIDIA, CUDA Toolkit Documentation, http://docs.nvidia.com/cuda/index.html.

[11] OpenCL, https://www.khronos.org/opencl/.

[12] OpenACC, March 2012, http://www.openacc.org/.

[13] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, Morgan Kaufmann, Walthman, Mass, USA, 2013.

[14] M. Norton, "Optimizing Pattern Matching for Intrusion Detection," 2004, http://pdf.aminer.org/000/309/890/optimizing_pattern_matching.pdf.

[15] N. P. Tran, M. Lee, S. Hong, and J. W. Bae, "Performance optimization of Aho-Corasick algorithm on a GPU," in *Proceedings of the 11th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA '13)*, Melbourne, Australia, July 2013.

[16] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken, "Analysis of multithreaded architectures for parallel computing," in *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '90)*, pp. 169–178, ACM, Crete, Greece, July 1990.

[17] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 white paper," http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[18] Y.-H. E. Yang and V. K. Prasanna, "Robust and scalable string pattern matching for deep packet inspection on multicore processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 11, pp. 2283–2292, 2013.

[19] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan, "Evaluating GPUs for network packet signature matching," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS '09)*, pp. 175–184, usa, April 2009.

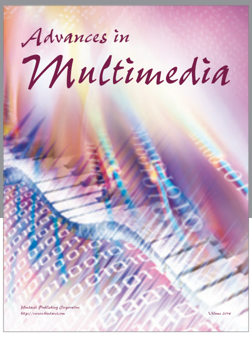
[20] N. Jacob and C. Brodley, "Offloading IDS computation to the GPU," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*, pp. 371–380, Miami Beach, Fla, USA, December 2006.

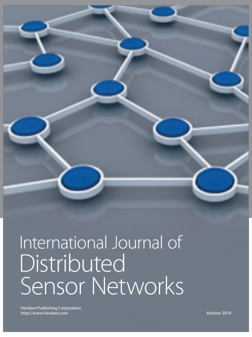
[21] D. Scarpazza, O. Villa, and F. Petrini, "Peak-performance DFA-based string matching on the cell processor," in *Proceedings of the International Workshop on System Management Techniques, Processes, and Services*, pp. 1–8, Long Beach, Calif, USA, March 2007.

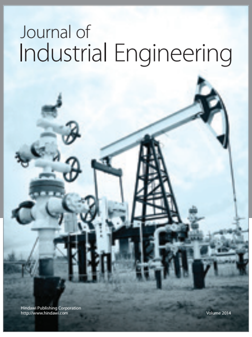
[22] O. Villa, D. P. Scarpazza, and F. Petrini, "Accelerating real-time string searching with multicore processors," *IEEE Computer Society*, vol. 41, no. 4, pp. 42–50, 2008.

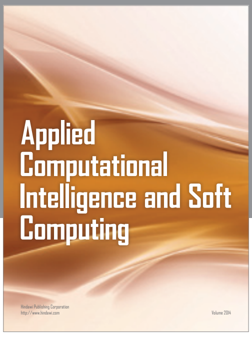
[23] X. Zha, D. P. Scarpazza, and S. Sahni, "Highly compressed multi-pattern string matching on the cell broadband engine," in *Proceedings of the 16th IEEE Symposium on Computers and Communications (ISCC '11)*, pp. 257–264, Kerkyra, Greece, July 2011.

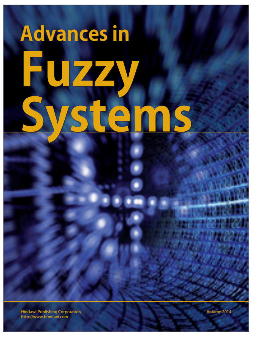
[24] O. Villa, C.-M. Daniel, and K. Maschhoff, "Input-independent, scalable and fast string matching on the cray XMT," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, pp. 1–12, Rome, Italy, May 2009.

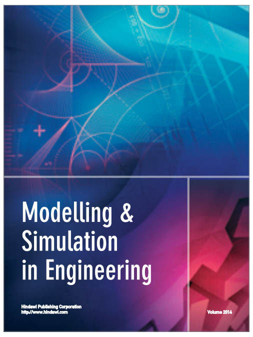












Hindawi

Submit your manuscripts at
http://www.hindawi.com