2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

High Throughput Parallel Implementation of Aho-Corasick Algorithm on a GPU

Nhat-Phuong Tran†, Myungho Lee†*, Sugwon Hong†, Jaeyoung Choi¶

†Department of Computer Science and Engineering, Myongji University
38-2 San Namdong, Cheo-In GuYong In, Kyung Ki Do, Korea 449-728

¶School of Computer Science and Engineering, Soongsil University
369 Sangdo Ro, Dongjak gu, Seoul, Korea 156-743

Abstract— **Pattern matching is an important operation in various applications such as computer and network security, bioinformatics, image processing, among many others. Aho-Corasick (AC) algorithm is a multiple patterns matching algorithm commonly used for such applications. In order to meet the highly demanding performance requirements imposed on these applications, achieving high performance for AC algorithm is crucial. In this paper, we present a high performance parallel implementation of AC algorithm on a Graphic Processing Unit (GPU) which efficiently utilizes the high degree of on-chip parallelism and the memory hierarchy of the GPU so that the aggregate performance (or throughput) of the GPU can be maximized. For this purpose our approach carefully places and caches the input text data and the reference pattern data used for pattern matching in the on-chip shared memories and the texture caches of the GPU. Furthermore, it efficiently schedules the off-chip global memory loads and the shared memory stores in order to minimize the overheads in loading the input data to the shared memories and also to minimize the shared memory bank conflicts. The proposed approach leads to a significant cut-down of the effective memory access latencies and leads to impressive performance improvements. Experimental results on Nvidia GeForce GTX 285 GPU show that our approach delivers up to 127Gbps throughput performance and up to 222-times speedup compared with a serial version running on 2.2Ghz Core2Duo Intel processor.**

Keywords- Aho-Corasick algorithm; GPU; parallelization; shared-memory bank conflict;

I. INTRODUCTION

Aho-Corasick (AC) algorithm [1] is a multiple patterns matching algorithm which can simultaneously match a number of patterns for a given finite set of strings (or dictionary). The AC algorithm is commonly used in various pattern matching applications such as network intrusion detection [15], [16], genome/protein matching for bio-sequence analysis [11], [14], image processing, among many others. In network intrusion detection, for example, intensive pattern matching operations are performed for a deep packet inspection using the AC algorithm. In order to speed up the pattern matching and meet the real-time performance requirement imposed on these applications, achieving high performance for AC algorithm is crucial.

Recently, the Graphic Processing Unit (GPU) is becoming increasingly popular for various applications. The latest GPU architectures have taken separate processing units like the shader, vertex, and pixel units from earlier designs and incorporated them into multiple uniform programmable processing cores. With the new architecture, the number of uniform fine-grain cores has drastically increased and huge floating-point performance improvements are made possible [5], [6]. Furthermore, the shared memory and the texture/constant caches are built on chip to effectively reduce the memory access latencies, whereas previous GPU's memory hierarchy was designed mainly for maximizing the memory bandwidths. User friendly programming environments have been also developed recently such as CUDA [5], [6] from NVidia, OpenCL [8] from Khronos Group, OpenACC [7] from a subgroup of OpenMP Architecture Review Board (ARB). Using these environments, one can have more direct control over the GPU cores and the memory hierarchy. Thus it has made a lot of innovative performance improvements in many application areas possible and many more are still to come.

In this paper, we develop a high performance parallel implementation of the AC algorithm for a GPU. In order to maximize the throughput performance of a GPU, we design parallelization approaches which efficiently utilize the high degree of on-chip parallelism and the memory hierarchy of the GPU. For this purpose, we carefully place the input text data and the reference data used for the pattern matching. The input text data whose size is at least a few hundred mega-bytes large can be fully or partially placed in the global memory of the Graphic-DRAM (GDRAM) or the device memory. However, they are too large to fit in on-chip memories (shared memory, texture/constant caches). Therefore, in order to efficiently load the input data to the shared memory, we carefully arrange the global memory accesses to coalesce them so that the number of global memory loads can be minimized for a block of data assigned to a block of threads. Furthermore, in order to minimize the shared memory bank conflicts for the loads used in the pattern matching, we carefully arranges the stores of the data returned from the global memory loads. For the reference pattern data with which the input text data is compared for possible matches, we organized them in a 2-dimensional matrix called the State Transition Table (STT) and place them in the texture memory. The actively used part of the STT is cached in the on-chip texture cache from the texture memory.

* Corresponding author: myunghol@mju.ac.kr

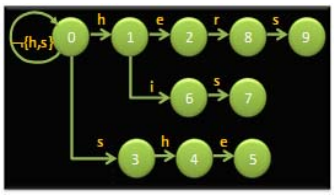
978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.116

The approach significantly cuts down the average memory access latencies to load both the input data and the reference data, and leads to impressive performance improvements for the AC algorithm. By implementing the proposed approach on a GPU (Nvidia GeForce GTX 285) using CUDA, we have observed an impressive throughput performance (up to 127 Gbps) and significant performance improvements (up to 222-times speedup) compared with the performance on a general-purpose multi-core processor (2.2Ghz 4-core Intel processor).

The rest of the paper is organized as follows: Sections II gives an overview of AC algorithm. Section III shows the architecture of the latest GPU and its parallel programming model. Section IV explains our parallelization approaches to maximize the throughput performance. Section V shows the experimental results on Nvidia GeForce GTX 285 GPU, and on a multi-core Intel processor for comparison with the GPU performance. Section VI wraps up the paper with conclusions.

II. AHO-CORASICK (AC) ALGORITHM

The Aho-Corasick (AC) algorithm is a multiple patterns matching algorithm which can match multiple patterns simultaneously for a given finite set of strings (or dictionary). The AC algorithm can be implemented as Non-deterministic Finite Automata (NFA) or Deterministic Finite Automata (DFA). The AC consists of two phases. In the first phase, a pattern matching machine called AC automaton (machine) is constructed from a finite set of patterns. In the second phase, the input text data is applied to the constructed AC machine in order to find the locations that the patterns appear [1].



(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 1. Functions used in AC algorithm [1]

AC automaton invokes three functions: a goto function *g*, a failure function *f*, and an output function *output*. Figure 1 shows these functions for a set of patterns {"he", "she", "his", "hers"} [1]:

- The directed graph in Figure 1(a) represents the goto function *g*. (¬('h', 's') denotes all input symbols other than 'h', 's'.) 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, 'h')=1. The absence of an arrow indicates *fail*. The AC machine has the property that g(0, σ) ≠ *fail* for all input symbol σ.
- The failure function *f* maps a state into another state. It is consulted whenever the goto function reports a "fail".
- The output function *output* maps a set of keywords to output at the designated states.

Assume that we have a text string "ushers". AC machine works in the following manner:

- Starting with state 0, the machine loops back to state 0 since g(0,'u')=0. For the same reason, the machine enters states 3, 4, 5 sequentially while processing the string 's', 'h', 'e' (g(0,'s')=3, g(3,'h')=4, g(4,'e')=5) and emits output, indicating that it has found the keywords "she" and "he" at the end of position in the text string.
- After then the machine advances to the next input symbol ('r'). Since g(5,'r')=fail, the machine enters state 2 because f(5)=2 (see Figure 1b). Then, since g(2,'r')=8, g(8,'s')=9 the AC machine enters state 9 and emits output "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 = δ(state, x[i]);
                if (output(state) != empty)
                begin
                        print i
                        print output(state)
                end
        end
end
```

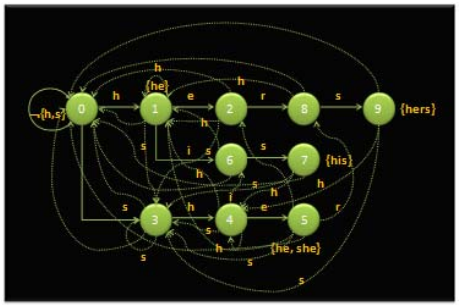
Figure 2. Pseudocode of the AC machine implemented as DFA

In this paper, we implement AC algorithm as a DFA. The AC machine implemented as a DFA represents all of possible states of the machine along with information of the acceptable state transitions of the system [4]. 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 goto function and failure function introduced in Figure 1. The output function is also incorporated in the DFA. Figure 2 shows how the AC machine works as a DFA.

Figure 3 shows the AC machine for a set of patterns {he, she, his, hers} implemented as 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. Assume that we have a text string "ushers". The AC machine implemented as a DFA works in the following manner:

- Since δ(0,'u')=0, the AC machine enters state 0.
- Since δ(0,'s')=3, δ(3,'h')=4, and δ(4,'e')=5, the AC machine emits output(5)={he, she}.
- Since δ(5,'r')=8 and δ(8,'s')=9, the AC machine emits output(9)={hers}.

The AC machine implemented as a DFA processes the input text with complexity O(n), where n is the length of the input text.



(Thin line: fail transition)

Figure 3. AC machine implemented as a DFA for patterns (he, she, his, hers)

III. Overview of GPU Architecture and Programming

GPU has become widespread and these days it is commonly incorporated in almost all computing platforms including desktop PC's, high performance computing servers, and even in mobile devices such as smart phones. Recent GPUs have shown impressive performance for floating-point operations, far exceeding that of the latest CPUs and the performance gap is widening. This is made possible with the drastic increase in the number of fine-grain cores by replacing the separate Shader, Vertex, and Pixel units with uniform multiple programmable processing units [5], [6]. In order to efficiently utilize the advanced flexible hardware design, more user friendly programming environments have been recently developed such as CUDA from Nvidia [5], [6], OpenCL from Khronos Group [8], OpenACC [7] from a subgroup of OpenMP Architecture Review Board (ARB). Using those environments, programmers can have more direct control over the GPU cores and the memory hierarchy. The flexible GPU hardware and user friendly software environments have led to a number of innovative performance improvements in many application areas and many more improvements are still to come.

In the experiments conducted in the paper, we use Nvidia GPU and CUDA. For executing CUDA programs, a hierarchy of memories is used on the Nvidia's GPU. These are registers and local memories belonging to each thread processor, a shared memory used in a thread block, and global memory accessed from all the thread blocks [5], [6]:

- Global memory is an area in the off-chip DDR Graphic DRAM (G-DRAM or commonly called as the device memory of which the size ranges from 256MB to 6GB). Through the global memory GPU can communicate with the host CPU.
- Shared memory sits within each thread block and shared amongst the threads running on multiple thread processors. Its typical size is 8~16KB. The access time closely matches with the register access time, thus it is a very fast memory.
- In the high-end Nvidia GPU such as the Tesla based on Fermi architecture, there is a level-1 (L1) data cache per thread block of which the size is 48KB. (Or the user can freely set the size of L1 data cache and the shared memory out of 64KB on-chip memory embedded on a thread block.)
- Registers are used for temporarily storing data for GPU computations in each thread processor, similar to CPU registers.
- Also each thread has its own local memory to load and store the data needed for the computations. The local memory is an area in G-DRAM. Thus it is a slow memory. When a thread processor runs out of registers, the local memory is used for the register spill/refill for each thread.
- Besides the above memories, there are constant memory and texture memory in the G-DRAM. Data in constant memory and texture memory can be cached as read-only data on chip in the constant cache and the texture cache respectively.

In CUDA programs, data needed for computations on GPU is transferred from the host memory to the global memory in the G-DRAM, distributed to the shared memories, texture memories, and constant memories by the programmer, then used by thread blocks and thread processors. Then multiple threads assigned to each thread block executes in the SIMD (Single Instruction Multiple Data) mode by having the same instruction managed by the Instruction Unit on different sections of data streaming from the global memory to the on-chip memories (shared memory, registers, etc.) on the same thread block (see Fig. 4) [6]. When a running thread encounters a cache miss, the context is switched to a new thread while the cache miss is serviced for the next few hundred cycles. Thus the GPU is executing in a multithreaded fashion.

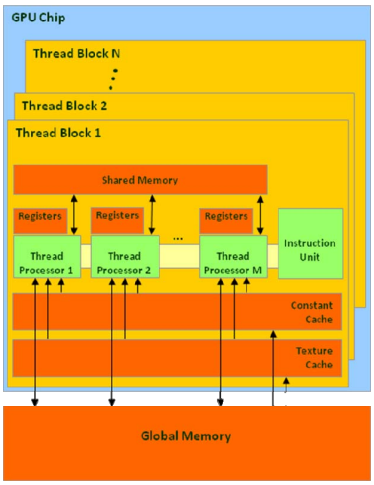


Figure 4. General architecture of a GPU

IV. PARALLELIZING THE AC ALGORITHM ON A GPU

A GPU provides many promising architectural characteristics in performing pattern matching operations. Compared with a general-purpose multi-core processor widely used these days, a GPU has a much larger number of cores (although simple) where massive parallel pattern matching operations can be performed at the same time in SIMD mode. Besides, a GPU's multithreaded execution can help further the throughput performance for the pattern matching operations. A GPU also provides much higher memory bandwidths than a multi-core processor. Thus it can feed input data and the reference pattern data at much higher rates for possible matches. On the other hand, a GPU has a complicated memory hierarchy whose efficient use dominates the application's performance and is mostly under the programmer's control. Therefore, achieving high performance for pattern matching algorithm such as AC on a GPU involves sophisticated parallelization techniques. In this section, we first introduce previous researches on parallelizing the AC algorithm for various applications. Then we introduce our high throughput parallelization approach on the GPU.

A. Previous Research

The AC pattern matching algorithm has been previously applied in various applications. In fact, network and computer security, and bioinformatics are two major areas where the AC algorithm is intensively applied. For example, it is used for a deep packet inspection in the network intrusion detection. It is used in anti-virus software to protect computers from viruses. It is also used in bio-sequence analysis for genome/protein matching. These security and bioinformatics applications are computationally demanding and require high-speed parallel processing. Recently, as GPUs are becoming increasingly popular, researchers are exploiting GPU's high degree of parallelism for speeding up the AC algorithm.

In the area of network intrusion detection, Smith et al. [13] implemented a regular expression matching on the GPU based on the (extended) deterministic finite automata. Their implementation of the signature matching on Nvidia G80 GPU achieved 6~9 times speedup compared with a serial version on Pentium 4 system. Jacob, et al. [2] also proposed a solution to off-load the signature matching computations to the GPU. Nevertheless, they used the Knuth-Morris-Pratt (KMP) single matching algorithm instead of the AC algorithm. Giorgos Vasiliadis et al. [16] presented an intrusion detection system based on the Snort open-source NIDS called Gnort. In order to parallelize the pattern matching on the GPU, they proposed two approaches to process packets: assigning a single packet to each thread or assigning a packet to a thread block. It outperformed conventional methods using single CPU core by a factor of two. Cheng-Hung Lin et al. [3] proposed a modification of the AC algorithm, called Parallel Failureless AC Algorithm (PFAC) which can remove all failure transitions in a conventional AC machine. PFAC creates a large number of threads for pattern matching by assigning each byte of an input stream to a thread to identify any pattern matching at the thread starting byte. They tested their proposed methods on the Nvidia GeForce GTX480 GPU with 192MB input data and ~2,000 patterns.

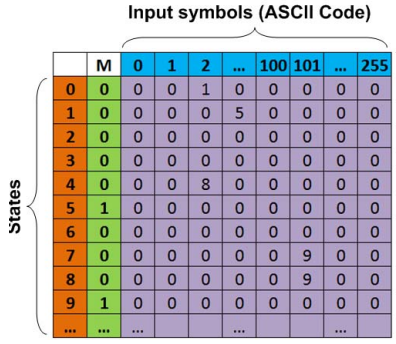
In the area of bioinformatics, Antonino Tumeo and Oreste Villa [14] presented an efficient implementation of the AC algorithm for accelerating DNA analysis on a heterogeneous GPU clusters. They partition the DNA sequence into multiple chunks and assign each chunk to a single CUDA thread. The performance results on single GPU is 5.47~12.35 faster than the best multi-processor solution, using 8 MPI processes. Xinyan Zha and Sartaj Sahni [18] proposed a parallel AC algorithm on a GPU. They addressed the problems on a GPU such as improving the utilization of available bandwidth in reading data from the device memory and writing results back to the device memory. The experimental results on Nvidia GT200 showed a speedup of 8.5~9.5 compared with a single CPU core and a speedup of 2.4~3.2 compared with the best multithreaded implementation on a multicore processor. They conducted experiments with a large input string size, but with a rather small number of patterns (33).

A few other researchers have attempted to port multi-string matching applications to different platforms. For instance, Scarpazze et. al. [9], [10] ported the AC-opt version to the IBM Cell Broadband Engine (BE) which achieved 2.5~10 Gbps throughput. In addition, Zha et al. [19] proposed a technique to compress AC automaton to be used on the Cell BE of which the sustained throughput ranges 0.9~2.35 Gbps.

B. Our New Approach

1) Construction of DFA for AC Automaton

In order to implement the AC pattern matching machine, we construct a DFA for a given finite set of strings (or dictionary). This is the phase 1 of the AC algorithm explained in Section II. The DFA makes exactly one state transition given an input character. In order to implement the DFA, we construct a 2-dimensional matrix (called State Transition Table: STT) to store the automaton structure. The rows represent 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 ASCII table), then the STT needs 257 columns (256 columns for characters and 1 column indicating if the current state is a matched state where output function is executed). Figure 5 illustrates the STT structure for AC automaton.



	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)

In the AC pattern matching algorithm where the pattern matching is performed for a given finite set of strings (or dictionary), the STT does not change at run-time once it is constructed. (In other kind of pattern matching algorithms such as the regular expression matching, the reference pattern data may change at run-time. Thus the STT structure may also need to change at run-time accordingly.) Therefore we construct the STT on single CPU core, then we copy it to the GPU side device memory for pattern matching operations against the input text data (phase 2 of the AC algorithm).

2) Efficient Data Placements

Once the STT is constructed and stored in the host memory, we copy it to the device memory. Besides the STT, the input text data is also copied to the device memory for the pattern matching operations. When copying these data to the device memory, we need to carefully consider their memory placements in order to efficiently utilize the GPU's memory hierarchy and the available bandwidths:

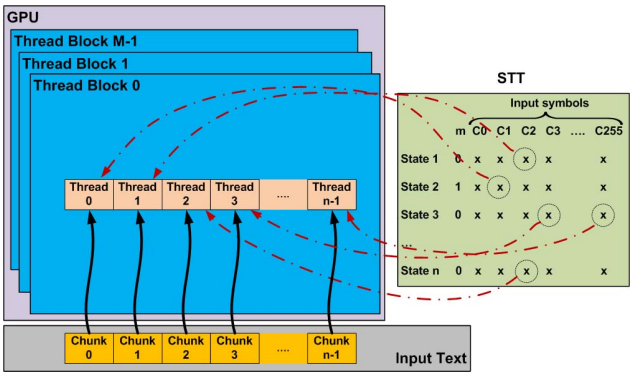


Figure 6. Data access patterns in AC algorithm using GPU

- The input data accesses are generated in parallel from the multiple thread processors on the GPU. The STT is also accessed by all GPU threads randomly for the concurrent pattern matching operations (see Figure 6). Separating the access paths of the input text data and the reference pattern data so that they do not directly interfere with each other reduces the memory access delays and improves performance. Furthermore, it can use the available memory bandwidths more efficiently.
- We place the input text data in the global memory by copying them from the host memory to the global memory region of the device memory. Additionally, an efficient use of the on-chip shared memory to cache the input text data from the global memory can speed up the pattern matching operations.
- The STT is placed in the texture memory which is a read-only memory space in the device memory. The actively used part of the STT is cached in the on-chip texture cache. The texture cache is optimized for 2-dimensional spatial local data [5] suitable for the 2-dimensional STT structure. With the texture cache, the effective texture memory latencies for the random accesses to the STT can be reduced while pattern matching operations are performed.

3) High Throughput Parallelization Strategies

Based on the above data placements, we develop efficient parallelization strategies which maximize the throughput performance of the AC algorithm. We first introduce an approach where we use the global memory only to store the input text data. Then we introduce the second approach to cache the input text data from the global memory to the on-chip shared memory. The STT is stored in the texture memory in both approaches.

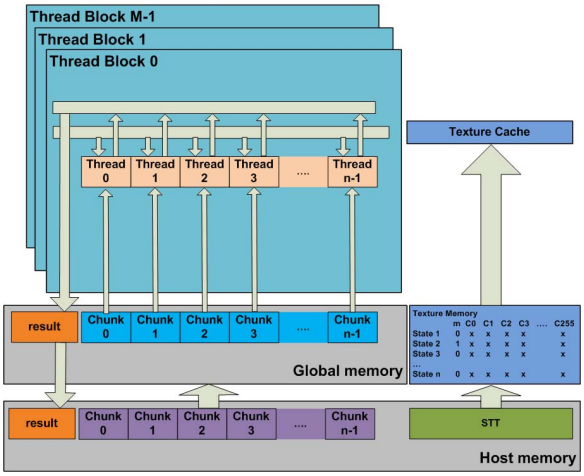


Figure 7. Illustration of global memory only approach

Global Memory Only Parallelization Approach

In this approach, we parallelize AC pattern matching algorithm while the input text data is stored in the global memory (see Figure 7):

- We divide the input text in the global memory into many chunks and assign one chunk to each thread processor (*N*-chunks to a thread block, where *N* is the number of thread processors in each thread block). As the GPU executes in the multithreaded fashion, the long global memory access latencies can be partially or fully masked off or hidden.
- Each thread applies a pattern matching on the assigned input chunk. This assignment incurs a problem when the assigned patterns are overlapped between the previous chunk and the following chunk. In order to solve this problem, we span each thread by adding X characters after the chunk that it is assigned, where X is the maximum pattern length in the finite set of patterns.
- The actively used part of the STT gets cached in the on-chip texture cache. As the number of patterns increases, the STT size also increases. Thus the number of texture cache misses also increases. The GPU's multithreaded execution can help hide the latency to the texture memory which is as large as the global memory access latency.

Shared Memory Parallelization Approach

In this approach using the shared memory, we intend to reduce the overheads associated with the global memory accesses for the input data.

- We first divide the input text data into a number of blocks. All threads in a thread block cooperate to load a block of the input data from the global memory to the shared memory (see Figure 8). Then each thread in a thread block performs the pattern matching with respect to its assigned chunk in the block.

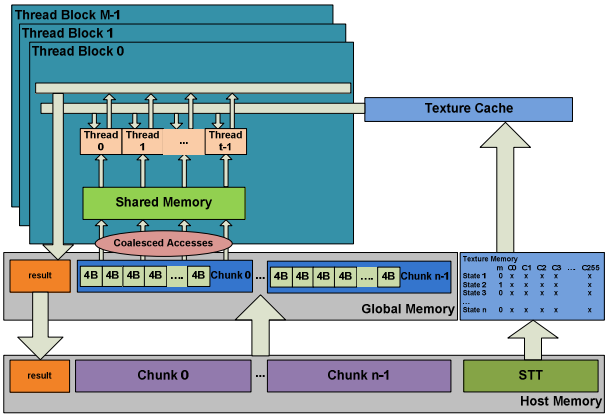


Figure 8. Illustration of shared memory approach

- While loading a data block, an important performance consideration is to coalesce the global memory accesses. Multiple global memory loads whose addresses fall within 128-bytes range are combined into one request to be sent to the global memory. This saves the memory bandwidth a lot and leads to improved performance.

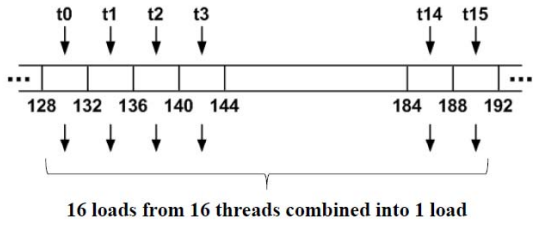


Figure 9. Coalesced accesses: 16 threads cooperate to load 64 bytes together

- The input text is stored in the global memory in a sequential fashion. If each thread slides on its own assigned chunk of data in a data block and naively loads data sequentially from the global memory to the shared memory, the total latency to load data for each thread will be high. Using coalesced load, we let threads of a thread block cooperate to read together and each thread read four bytes (32-bit word) at one time (see Figure 9). Thus multiple threads cooperate to load one chunk of data after another to fully load a block of data for the thread block.
- For example, let's assume the size of the data block assigned to a thread block is 1024 bytes and there are 16 threads per thread block (see Figure 10). Because each thread reads one 4-byte word at one time, 16 threads of a block cooperate to load 64 (=4*16) bytes data at one time. Therefore, we need 1024 / 64 = 16 coalesced loads from the global memory to fully load the 1024 bytes block of data to the shared memory.

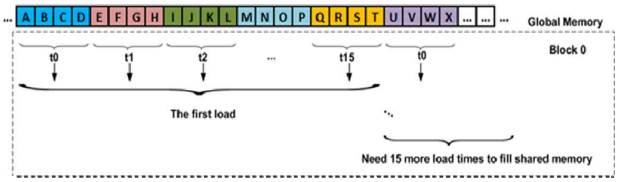


Figure 10. Loading 1024-bytes data from the global memory to the shared memory in 16 steps using memory coalescing (assuming shared memory size=1024 bytes, number of threads per block = 16, each thread reads 4 bytes at one time)

The shared memory where the data returns by the coalesced loads from the global memory is divided into multiple memory banks in order to maximize the number of simultaneous accesses from the GPU cores:

- Each bank has a bandwidth of 32-bits per clock cycle. Successive 32-bit words are assigned to successive banks.
- If there are multiple simultaneous accesses to the same bank, they result in bank conflicts. Conflicting accesses to the same bank are serialized and result in lowered performance.
- In AC algorithm each thread accesses a chunk of data for the pattern matching. If each chunk of data is sequentially stored to the shared memory, it will be spread out over multiple banks. When multiple threads attempt to read their own chunk of data simultaneously, it will result in a lot of bank conflicts.

In order to avoiding the bank conflicts, we need to carefully store the data fetched from the global memory to the shared memory:

- Figure 11 shows a store scheme through which a chunk of data cooperatively loaded by multiple threads gets divided up into 4-bytes units. Then those 4-bytes units are stored in the shared memory at the addresses which are mapped to consecutive shared memory banks in a diagonal way.
- This store scheme avoids any bank conflict because those 4-byte units are stored to different banks. Furthermore, this scheme results in a conflict-free load from the shared memory banks because the loads from multiple threads are now directed to different banks (see Figure 12).

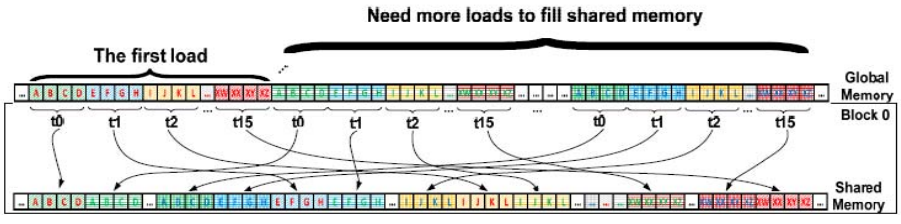


Figure 11. Data store scheme to avoid shared memory bank conflicts

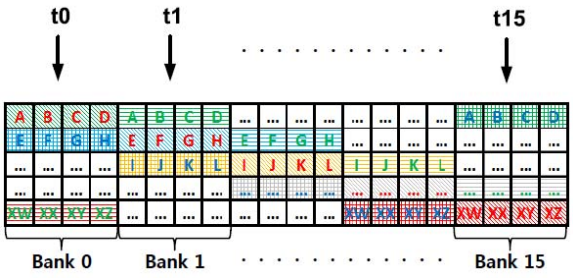


Figure 12. Data distribution to 16 memory banks using our store scheme

Besides the bank conflicts in the shared memory, other performance related considerations are as follows:

- Out of 16KB shared memory on our GPU (Nvidia GTX 285), we use 8~12KB for the input text data out of 16KB shared memory space. The remaining 4~8KB space in the shared memory is reserved for other works.
- Synchronization of data accesses is performed to make sure that all threads transfer data from the global memory to the shared memory before threads process other pattern matching tasks on its own data chunk.

V. EXPERIMENTAL RESULTS

We implemented the AC algorithm in three ways: serial implementation using single CPU core, two parallel implementations on a GPU using the global memory only approach and using the shared memory approach. Our experiments are conducted on a system including Intel multi-core processor (2.2Ghz Intel Core2Duo 4) with 2GB of main memory, Nvidia GeForce GTX 285 GPU with 240 thread processors (or cores) organized in 8 streaming multiprocessors (or thread blocks), operating at 1.48 GHz with 1GB device memory. The OS is Centos 5.5.

We used input data sizes in the range of 50KB - 200MB and the numbers of patterns in the range of 100 – 20,000. In order to generate random input data sets and the reference pattern data sets, we first collected 50GB of data from a variety of magazines such as TIME, BBC, among many others. Then we extracted input data and pattern data from the collected data. In all experiments conducted, we ignored the time spent in the construction phase of STT which run on single CPU core and the time to copy the input text data and the STT to the GPU device memory. This is fair because the STT construction and data copy are performed only once for a given finite set of strings, whereas the pattern matching operations are performed a large number of times.

A. Run Time Comparisons

Figures 13, 14, and 15 show the run times of the three approaches (serial, global memory only, and shared memory) for a range of input data sizes and for a range of the numbers of patterns. Comparing these results, we have observed the followings:

- The run times increase as the data size increases and as the number of patterns increases, in general.
- As the data size increases, however, the run time increase for the shared memory approach slows down with the increase in the number of patterns. For example, for 100MB and 200MB data, the run time differences for the different numbers of patterns are rather smaller compared with the single CPU core results and the global memory only results.

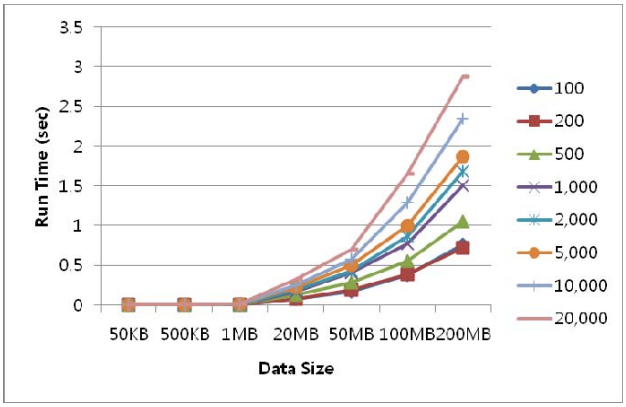


Figure 13. Run times for the serial approach using different input data sizes and different numbers of patterns

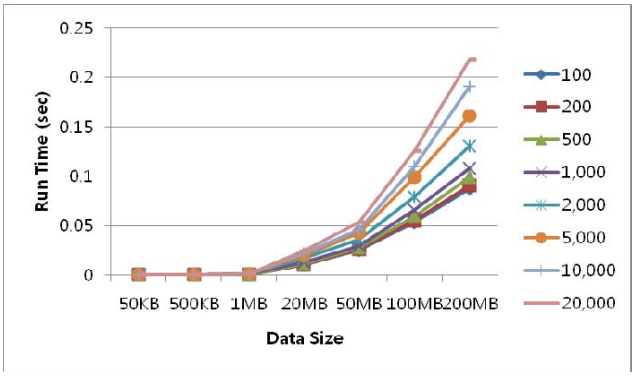


Figure 14. Run times for the global memory only approach using different input data sizes and different numbers of patterns

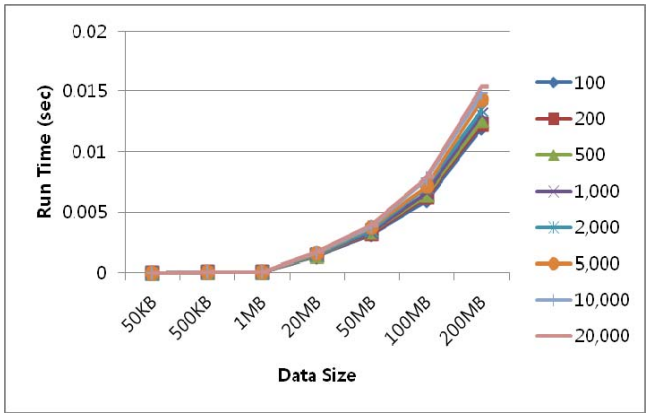


Figure 15. Run times for the shared memory approach using different input data sizes and different numbers of patterns

B. *Throughput Performance Comparisons*

Figures 16, 17, and 18 show the throughput performance of the three approaches for a range of input data sizes and for a range of the numbers of patterns, measured in Gbps.

- For a fixed number of patterns, the throughput increases with the data size increase in general. The throughput decreases with the increase of the number of patterns for all data sizes. For the shared memory approach, however, the throughput decrease is much smaller with the increase in the number of patterns. The maximum throughput reaches up to 127Gbps for the 200MB input data when the number of patterns is 100.

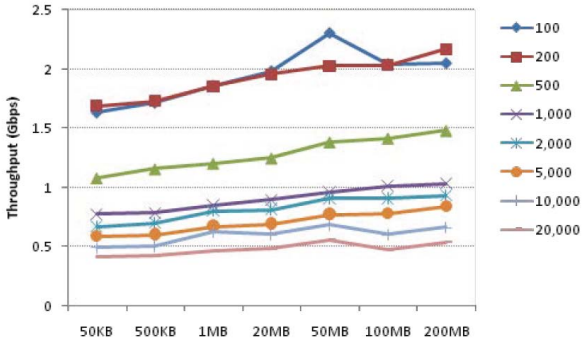


Figure 16. Throughput (measured in Gbps) for the serial approach using different input data sizes and different numbers of patterns

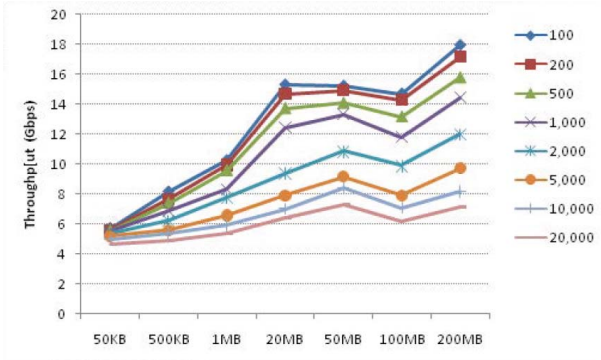


Figure 17. Throughput (measured in Gbps) for the global memory only approach using different input data sizes and different numbers of patterns

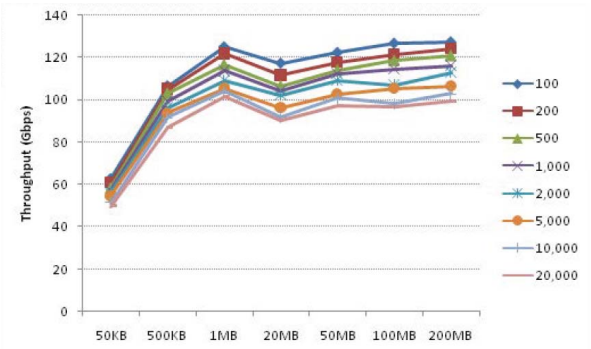
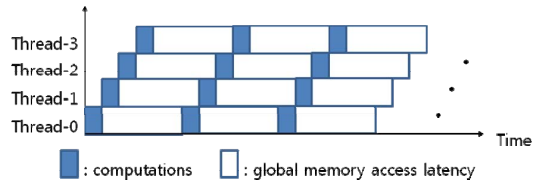
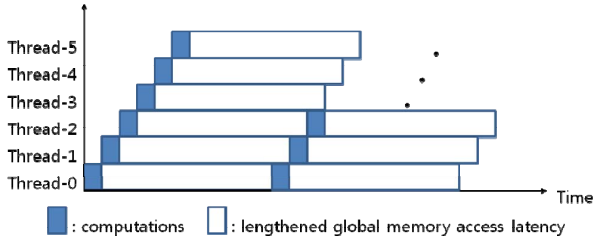


Figure 18. Throughput (measured in Gbps) for the shared memory approach using different input data sizes and different numbers of patterns

- As observed above, for a fixed input size, the absolute throughput performance decreases with the increase in the number of patterns. As the number of patterns increases, the texture cache misses where the STT is stored increases. The increased number of texture cache misses incurs more context switches in the multithreaded execution on the GPU.
- In the multithreaded execution, an ideal case is that the global memory access latency for the shared memory miss or the texture cache miss is effectively covered by multiple threads' useful computations (see Figure 19 (a)).
- In the global memory only approach, however, not only the texture cache misses incurs the context switch but also when the executing threads accesses the global memory. Thus the number of context switches is larger than the shared memory approach and results in a higher degree of multithreading in play. Also the average useful computation cycle for each thread is shorter as the input data is not cached in the shared memory.
- In our analyses of the throughput performance, the shared memory approach is close to the Figure 19 (a) case, whereas the global memory only approach is rather close to Figure 19 (b) case. Therefore, the shared memory approach resiliently tolerates the overheads incurred with the increased number of texture cache misses when the reference pattern data size increases.



(a) Memory access latencies effectively hidden with multithreading



(b) Performance saturation due to excessive multithreading

Figure 19. Performance effects of multithreading

C. *Speedup Comparisons*

Figures 20, 21 and 22 show the speedups of the global memory approach and the shared memory approach compared with the serial approach, and the speedup of the shared memory approach over the global memory only approach:

- For the global memory only approach, the speedup ranges 3.3 – 13.2.

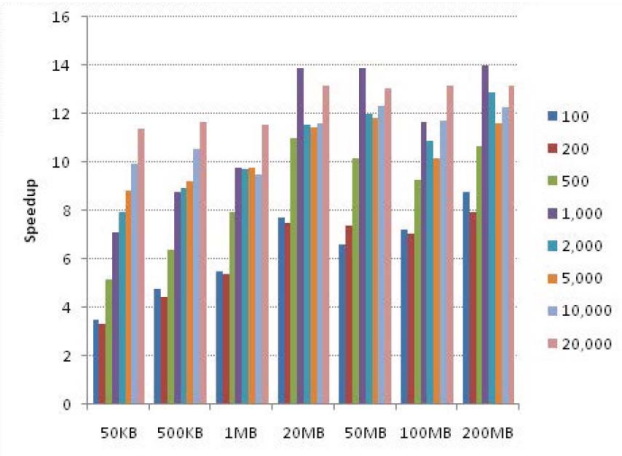


Figure 20. Speedup of the global memory only approach compared with the serial approach

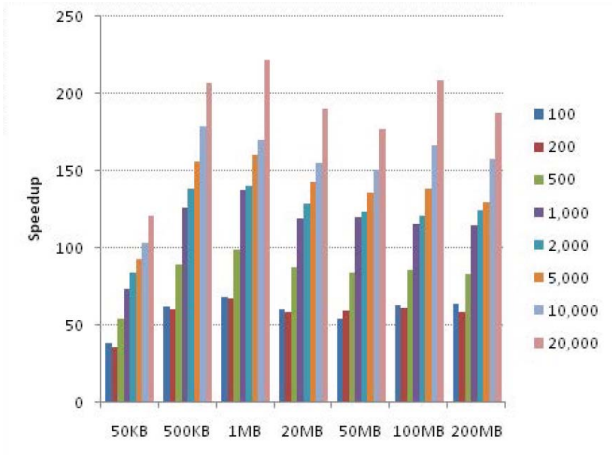


Figure 21. Speedup of the shared memory approach over the serial approach

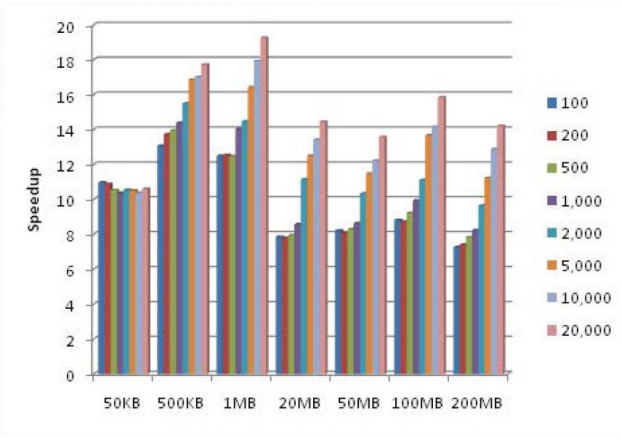


Figure 22. Speedup of the shared memory approach over the global memory only approach

- For the shared memory approach, the speedup ranges 36.1 – 222.0. The maximum speedup achieved is 222.0 using 100MB input data and 20,000 patterns.
- The shared memory approach compared with the global memory only approach shows 7.3 – 19.3 times speedup. Thus the benefit of the shared memory is large.

D. *Effects of Avoiding Shared Memory Bank Conflicts*

As explained in Section IV, avoiding the shared memory bank conflicts improves the performance. Figure 23 shows the effects:

- We compared the naïve shared memory writes for the data returned from the global memory loads, using memory access coalescing only, and our store scheme to avoid bank conflicts.
- It shows that our scheme performs 1.5~5.3 times faster than the naïve approach.
- The speedup of our scheme is larger as the numbers of patterns increases. A larger number of patterns incur more context switches, which in turn increases the accesses to the shared memory as more threads are running on the same hardware thread block. Therefore, the chances of the shared memory bank conflicts increases and the benefit of our scheme gets larger.

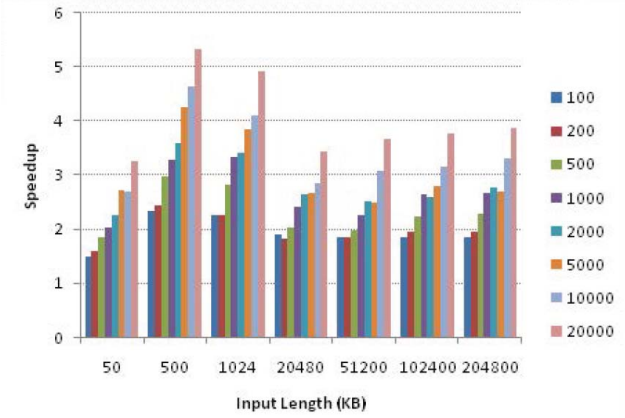


Figure 23. Speedup of our shared memory bank conflict avoiding scheme compared with the memory access coalescing only approach

VI. CONCLUSION

In this paper, we proposed a high throughput parallelization for the AC algorithm on a GPU. The proposed approach parallelizes the AC by efficiently placing and caching both the input text data and the reference pattern data in the on-chip shared memories and the texture caches. In loading the input data from the global memory to the shared memory, we carefully arrange the global memory loads and the shared memory stores so that the number of global memory accesses and the shared memory bank conflicts can be minimized. This significantly reduces the effective memory access latencies. Furthermore, it resiliently tolerates the overheads incurred with the increased number of texture cache

misses when the reference pattern data size increases. Experimental results on a 4-core, 2.2Ghz Intel processor and Nvidia GeForce GTX 285 GPU using CUDA shows that the new approach delivers impressive throughput performance of up to 127 Gbps and achieves up to 222-times speedup compared with a sequential run on single CPU core.

ACKNOWLEDGMENT

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: 2012-042269).

REFERENCES

[1] A.V. Aho and M.J. Corasick, "Efficient string matching: An aid to A.V. Aho and M.J. Corasick, "Efficient string matching: An aid to bibliographic search", Communications of the ACM, vol. 20, Session 10, Oct. 1977, pp. 761–772.

[2] N. Jacob, C. Brodley, "Offloading IDS Computation to the GPU", The 22nd Annual Computer Security Applications Conference, 2006

[3] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, Jyuo-Min Shyu, "Accelerating String Matching Using Multi-Threaded Algorithm on GPU", GLOBECOM 2010, 2010 IEEE Global Telecommunications Conference , vol., no., pp.1-5, 6-10 Dec. 2010.

[4] Marc Norton, "Optimizing Pattern Matching for Intrusion Detection", http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf, July 2004

[5] NVIDIA, "CUDA Best Practices Guide: NVIDIA CUDA C Programming Best Practices Guide – CUDA Toolkit 4.0", May, 2011.

[6] NVIDIA, "NVidia gtx280", http://kr.nvidia.com/object/geforce_family_kr.html

[7] OpenACC, http://www.openacc-standard.org , March, 2012

[8] OpenCL, http://www.khronos.org/opencl/

[9] D. Scarpazza, O. Villa, F. Petrini, "Peak-Performance DFA-based String Matching on the Cell Processor", International Workshop on System Management Techniques, Processes, and Services, 2007

[10] D. Scarpazza, O. Villa, F. Petrini, "Accelerating Real-Time String Searching with Multicore Processors", IEEE Computer Society, 2008

[11] Michael C. Schatz and Cole Trapnell, "Fast Exact String Matching on the GPU", Center for Bioinformatics and Computational Biology, 2007.

[12] Soumya Sen, "Performance Charaterization and Improvement of Snort as an IDS", http://www.princeton.edu/~soumyas/bell_labs_report_snort.pdf, August 2006

[13] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, C. Estan, Evaluating GPUs for Network Packet Signature Matching", Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on , vol., no., pp.175-184, 26-28 April 2009

[14] A. Tumeo, O. Villa, "Accelerating DNA analysis applications on GPU clusters", Application Specific Processors (SASP), 2010 IEEE 8th Symposium on , vol., no., pp.71-76, 13-14 June 2010

[15] Tumeo, A., Villa, O., "Efficient Pattern Matching on GPUs for Intrusion Detection Systems", in Proceedings of the 7th ACM international conference on computing frontiers, 2010

[16] Giorgos Vasiliadis, Spiros Antonatos, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors" , RAID, 2008, pp. 116-134.

[17] V. Volkov and J.W. Demmel,"Benchmarking GPUs to Tune Dense Linear Algebra", Proceedings of the ACM/IEEE SuperComputing 08 (SC 08), pp. Art. 31:1-11, Nov 2008

[18] X. Zha, S. Sahni, "Multipattern string matching on a GPU", Computers and Communications (ISCC), 2011 IEEE Symposium on, vol., no., pp.277-282, June 28 2011-July 1 2011

[19] X. Zha, D. Scarpazza, and S. Sahni, "Highly Compressed Multi-pattern String Matching on the Cell Broadband Engine", Computers and Communications (ISCC), 2011 IEEE Symposium on , vol., no., pp.257-264, June 28 2011-July 1 2011