2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications

Performance Optimization of Aho-Corasick Algorithm on a GPU

Nhat-Phuong Tran†, Myungho Lee†*, Sugwon Hong†, Jongwoo Bae‡

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

***Abstract*—Aho-Corasick (AC) algorithm is a multiple patterns matching algorithm commonly used for applications such as computer and network security, bioinformatics, image processing, among others. These applications are computationally demanding, thus optimizing performance for AC algorithm is crucial. In this paper, we present a performance optimization strategy for the AC algorithm on a Graphic Processing Unit (GPU). Our strategy efficiently utilizes the high degree of the on-chip parallelism and the complicated memory hierarchy of the GPU so that the aggregate performance (or throughput) for the AC algorithm can be optimized. The strategy significantly cuts down the effective memory access latencies and efficiently utilizes the memory bandwidth. Also, it maximizes the effects of the multithreading capability of the GPU through optimal thread scheduling. Experimental results on Nvidia GeForce GTX 285 GPU show that our approach delivers up to 127 Gbps throughput performance and 222-times speedup compared with a serial version running on single core of 2.2Ghz Core2Duo Intel processor.**

Keywords—Aho-Corasick algorithm; GPU; shared memory bank conflict; thread scheduling; multithreading

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 [16], [17], genome/protein matching for bio-sequence analysis [12], [15], image processing, among many others. In order to speed up the pattern matching operations and meet the real-time performance requirement imposed on these applications, optimizing performance for the AC algorithm is crucial.

Recently, the Graphic Processing Unit (GPU) is becoming increasingly popular for various applications. The architecture of the GPU has gone through a number of innovative design changes in the last decade which have drastically increased the peak floating-point throughput performance (flops) [5], [6]. In addition to the architectural innovations, user friendly programming environments have been recently developed such as CUDA [5], [6] from Nvidia, OpenCL [8] from Khronos Group, OpenACC [7] from a subgroup of OpenMP Architecture Review Board (ARB). The advanced GPU architecture and the flexible programming environments have made possible innovative performance improvements in many application areas and many more are still to come.

Applications greatly benefitted in performance by the GPU typically have intensive floating-point operations executing in parallel. Although the AC algorithm is not floating-point intensive, we attempt to deploy the GPU for achieving high performance for the AC algorithm. A GPU provides many promising architectural characteristics in performing pattern matching operations in the AC algorithm. Compared with a general-purpose multi-core processor, a GPU has a much larger number of (fine-grain) cores where massive parallel pattern matching operations can be performed in parallel. Besides, a GPU's multithreaded execution can further the throughput performance for the parallel pattern matching operations. A GPU also provides much higher memory bandwidths than a multi-core processor. Thus it can feed the 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. The efficient use of the GPU memory hierarchy is mostly under the programmer's control. Therefore, optimizing performance for the AC algorithm on a GPU involves sophisticated parallelization techniques.

In this paper, we present a performance optimization strategy for the AC algorithm on a GPU. Our strategy efficiently utilizes the high degree of the on-chip parallelism and the complicated memory hierarchy of the GPU so that the throughput performance can be optimized. First, our approach carefully places and caches the input text data and the reference pattern data used for the pattern matching operations. For the input data, we attempt to coalesce a number of loads issued to the off-chip memory into single load operation in order to save the memory bandwidths. The loaded data is then carefully placed in the user-managed on-chip shared memory so that the shared memory bank conflicts are avoided when multiple threads access the shared memory at the same time. For the pattern data, we place them in the texture memory so that the actively used part of the pattern data can be cache in the on-chip texture cache. As a result, the effective memory access latency is significantly reduced and the memory bandwidth is efficiently utilized. Second, our strategy schedules an optimal number of parallel threads onto the hardware thread blocks and the processing cores inside the thread block. The thread scheduling efficiently utilizes the large number of processing cores and the available memory bandwidths while the GPU is executed in the multithreaded fashion. Our strategy leads to impressive performance results. Experimental results on Nvidia GeForce GTX 285 GPU show that our approach delivers up to 127 Gbps throughput.

* Corresponding author: myunghol@mju.ac.kr

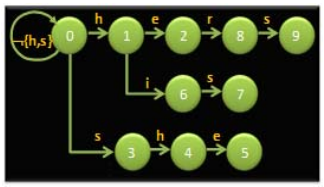
978-0-7695-5022-0/13 $26.00 © 2013 IEEE
DOI 10.1109/TrustCom.2013.138

Furthermore, compared with a serial execution on single core of 2.2Ghz Core2Duo Intel processor, it results in 222-times speedup.

The rest of the paper is organized as follows: Sections II introduces the AC algorithm. Section III describes the architecture of the latest GPU and its programming model. Section IV introduces previous research on paralleling the AC algorithm. Section V explains our strategy to optimize the performance of the AC algorithm on a GPU. Section VI shows the experimental results on Nvidia GeForce GTX 285 GPU. Section VII 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 a Non-deterministic Finite Automata (NFA) or a Deterministic Finite Automata (DFA). The AC consists of two phases: 1) first, a pattern matching machine called the AC automaton (machine) is constructed from a finite set of patterns; 2) second, 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]

In order to illustrate the AC algorithm in [1], we show an example. For a given set of patterns {"he", "she", "his", "hers"}, we first construct an AC automaton as shown in Figure 1. The AC automaton invokes three functions: a goto function *g*, a failure function *f*, and an output function *output*:

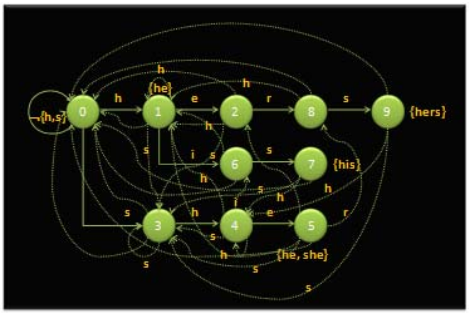
- The directed graph in Figure 1(a) represents the goto function *g*. (¬('h', 's') denotes all input symbols other than 'h', 's'.) The *g* function maps a pair consisting of a state and an input symbol into a state or a message *fail*. For example, the edge labeled h from state 0 to 1 indicates that *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". The pattern matching phase of the AC algorithm (the second phase of the algorithm in [1]) 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 the 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 1(b)). Then, since g(2,'r')=8, g(8,'s')=9 the AC machine enters state 9 and emits output "hers".

In this paper, we implement the AC algorithm as a DFA. The DFA consists of a finite set of states S and a next move function δ such that for each state *s* and an input symbol *a*, $\delta(s,a)$ is a state in S [4]. Thus, the next move function δ is used in place of both the goto function and the failure function illustrated in Figure 1. The output function is also incorporated in the DFA. Figure 2 shows the AC machine for a set of patterns {he, she, his, hers} where those three functions 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. Assume that we have a text string "ushers". The 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}.



(Thin line: fail transition)

Figure 2. AC machine implemented as a DFA for patterns (he, she, his, hers)

Figure 3 illustrates how the DFA for the AC algorithm is used for the pattern matching. As the pseudocode shows, the DFA processes the input text with length *n* in *O(n)*.

```
/*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 3. Pseudo code of the pattern matching for the AC machine implemented as DFA

III. Overview of GPU Architecture and Programming

The Graphic Processing Unit (GPU) was introduced in the late 1990s as a co-processor for accelerating the simulation and the visualization of 3D images commonly used in applications such as game programs. Since then the GPU has become widespread and these days it is commonly incorporated in many computing platforms including desktop PCs, high performance computing servers, and even in mobile devices such as smart phones. In the latest GPU, the clock rate has ramped up significantly compared with the earlier GPUs. Furthermore, the latest GPU architecture incorporates a large number of uniform programmable processing cores on a chip which execute in parallel. In order to efficiently feed data to the large number of cores, sizable on-chip memories are incorporated on the GPU chip. The increase in the clock rate and the new multi-core architecture design have made possible the impressive floating-point throughput performance (flops) of the GPU, far exceeding that of the latest CPUs. User friendly programming environments such as CUDA from Nvidia [5], [6], OpenCL from Khronos Group [8], and OpenACC [7] from a subgroup of OpenMP Architecture Review Board (ARB) have been recently developed for the efficient utilization of the advanced flexible architecture of the latest GPUs. The flexible GPU architecture and the user friendly software development environments have led to a number of innovative performance improvements in many applications and many more improvements are still to come [16, 21].

In the experiments conducted in the paper, we use Nvidia's GPU and CUDA. For executing CUDA programs, a hierarchy of memories is used on the Nvidia's GPU. They are registers and local memories belonging to each thread, a shared memory used in a thread block and shared by threads belonging to the block, and the global memory accessed from all the thread blocks [5, 6] (see Figure 4):

- Global memory is an area in the off-chip device memory. (The typical size of the device memory ranges from 256MB to 6GB.) Through the global memory, the GPU can communicate with the host CPU.
- Shared memory sits within each thread block and shared amongst the threads running on the multiple thread processors. The management of the shared memory is under the programmer's control. The typical size of the shared memory is 16KB. The access time closely matches with the register access time, thus it is a very fast memory.
- On a high-end Nvidia GPU such as the Tesla, there is a level-1 (L1) data cache per each thread block. Unlike the shared memory, L1 data cache is a hardware-managed cache. The typical size of the L1 data cache is 48KB. Or the user can freely set the size of L1 data cache and the shared memory out of the 64KB combined total size of the on-chip memory embedded on a thread block. In Nvidia GTX 285 which we use for the experiments, there is no L1 data cache. Thus we only use the shared memory.
- Registers are used for temporarily storing the data used for computations for each thread, similar to CPU registers.
- Also each thread has its own local memory area in the device memory to load and store the data needed for the computations. For example, when there are register spills/refills during the computations. Since the local memory is in the device memory, it is also a slow memory.
- Besides the above memories, there are constant memory and texture memory in the device memory. Data in the constant/texture memory are read-only. They can be cached in the on-chip constant cache and the texture cache respectively.

In CUDA programs, data needed for computations on the GPU is transferred from the host memory to the global memory, optionally placed in the shared memory by the programmer, and used by thread blocks and thread processors through the registers. The multiple threads assigned to each thread block executes in the SIMD mode by having the same instruction managed by the Instruction Unit on different portions of data. When a running thread encounters a cache miss, for example, the context is switched to a new thread while the cache miss is serviced for the next 200 cycles or more. Thus the GPU is executing in a multithreaded fashion.

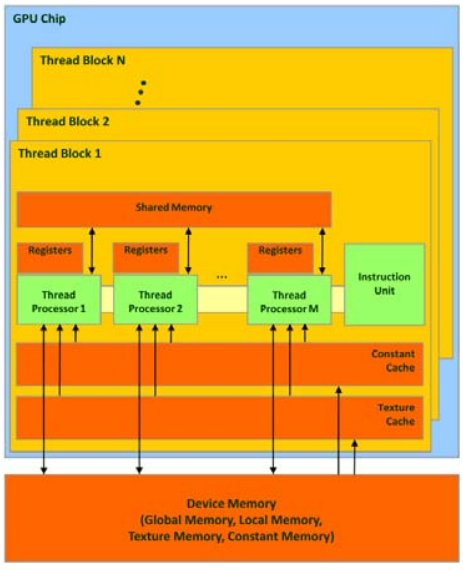


Figure 4. General architecture of a GPU

IV. PREVIOUS RESEARCH

The AC pattern matching algorithm has been previously applied in various applications. In fact, network/computer security, and bioinformatics are two major areas where the AC algorithm is intensively applied. For example, the AC is used for a deep packet inspection in the network intrusion detection. It is used in the 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 the GPU's high degree of parallelism for speeding up the AC algorithm.

In the area of network intrusion detection, Smith et al. [14] 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 the Pentium 4 based system. Jacob, et al. [2] proposed a solution to off-load signature matching computations to the GPU using the Knuth-Morris-Pratt (KMP) single matching algorithm. Giorgos Vasiliadis et al. [17] presented an intrusion detection system based on the Snort open-source NIDS called the 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. Their approach 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 [15] presented an implementation of the AC algorithm for accelerating the 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 multiprocessor solution, using 8 MPI processes. Xinyan Zha and Sartaj Sahni [19] proposed a parallel AC algorithm on a GPU. They addressed the problems on a GPU such as improving the utilization of the 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).

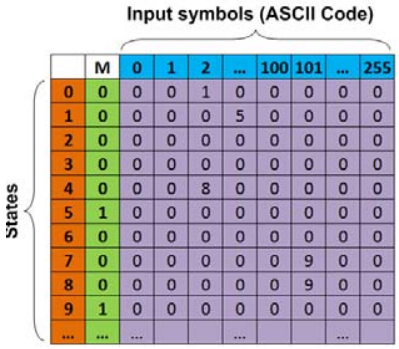
Other researchers also have attempted to port multi-string matching applications to different platforms. Scarpazze et. al. [10], [11] ported the AC-opt version to the IBM Cell Broadband Engine (BE). In addition, Zha et al. [20] proposed a technique to compress AC automaton to for the Cell BE.

V. OUR PERFORMANCE OPTIMIZATION STRATEGY

The AC algorithm introduced in Section II proceeds in two steps: 1) construction of the AC pattern matching machine; 2) conducting the pattern matching operations using the machine. In typical pattern matching applications using the AC algorithm, the first phase is performed once and the second phase is performed multiple times. For example, in the computer virus scan using the AC, the virus database (pattern data in our explanation) is updated once in every several days during which the same database is used for multiple virus scans. However, in other kinds of pattern matching algorithms such as the Regular Expression (RE) matching, the reference pattern data changes at the run-time. Thus, the entire phases of the RE matching needs to be incorporated in the parallel framework. In this paper, we perform the first phase of the AC sequentially using single CPU core. Then we perform the second phase on the GPU in parallel. Our parallelization and performance optimization strategy is focused on the second phase. In the following subsections, we introduce our parallelization strategy incorporating the efficient data placement, the memory access coalescing, avoiding the shared memory bank conflicts, and maximizing the effects of multithreading.

A. *Efficient Data Placement*

For a given finite set of pattern strings (or dictionary), we first construct the AC pattern matching machine using single CPU core. The machine is represented as a pattern data structure and stored in the host memory. We copy it to the device memory along with the input text data for the parallel pattern matching operations on the GPU. 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.



	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) for the AC automaton

The AC pattern matching machine constructed as a DFA in the first phase of the AC algorithm makes a transition from one state to another for a given input character. A 2-dimensional matrix (called State Transition Table: STT) is constructed to store the DFA. The rows of the STT 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 the AC automaton.

The constructed STT is copied to the device memory along with the input text data. When copying these data, we need to carefully decide where in the device memory (global memory, local memory, constant memory, texture memory) we need to store them. 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.

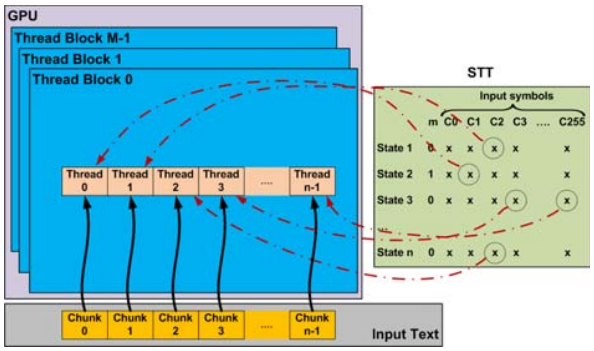


Figure 6. Data access patterns in AC algorithm using GPU

Based on the above, we do the following data placements:

- We copy the input text data from the host memory and place it in the global memory region of the device memory. For this, we use the zero-copy method of CUDA. The zero-copy allows direct accesses to the host memory from the thread processing cores on the GPU. The data is implicitly copied to the global memory. (The performace of the zero-copy is superior to the explicit data copy to the global memory as we will show in Section VI.) Additionally, we cache the input data in the on-chip shared memory to reduce the memory latency and speed up the pattern matching operations.
- The STT is copied from the host memory and 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 data [5] suitable for the STT structure. With the texture cache, the effective texture memory latencies of the random accesses generated from pattern matching operations can be reduced.

B. *Memory Access Coalescing*

The input data is stored in a sequential fashion in the global memory (implicitly by the zero-copy method). We divide the input data into many chunks. Each chunk is assigned to each thread processor. Thus *N*-data chunks or a data block is assigned to a hardware thread block, where *N* is the number of thread processors in each thread block in Figure 4. While loading a data block, an important performance consideration is to coalesce the global memory accesses. Multiple global memory loads whose addresses fall within the 128-bytes range are combined into one request and sent to the memory. This saves the memory bandwidth a lot and improves the performance. 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 the coalesced load, we let 16 threads of a thread block cooperate to read 64-bytes together (see Figure 7) and each thread read four bytes (32-bit word) at one time. Thus multiple threads cooperate to load one chunk of data after another to fully load a block of data for the thread block.

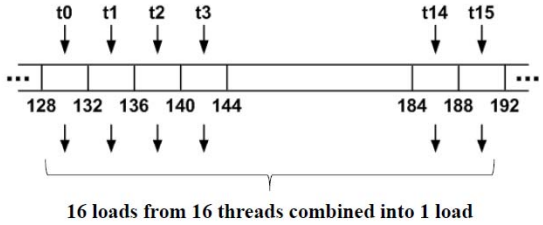


Figure 7. Coalesced accesses: 16 threads cooperate to load 64 bytes together

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 8). 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.

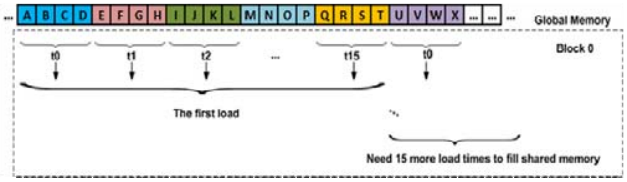


Figure 8. Loading 1024-bytes data from the global memory to the shared memory in 16 steps using memory coalescing (assuming shared memory size=1024 bytes, number of threads per block = 16, each thread reads 4 bytes at one time)

C. *Avoiding Shared Memory Bank Conflicts*

As the GPU executes in the multithreaded fashion, the long memory access latencies from the global memory can be partially or fully masked off or hidden. However, in order to minimize the effective latency, we need to cache them in the on-chip shared memory and optimize the use of the shared memory. The shared 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

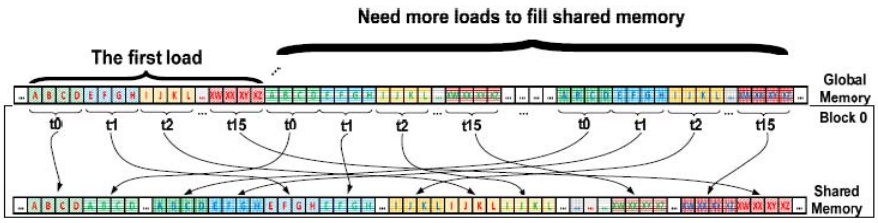


Figure 9. Data store scheme to avoid shared memory bank conflicts

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 the 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 which are also spread over multiple banks, it will result in a lot of bank conflicts.

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

- Figure 9 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 10).

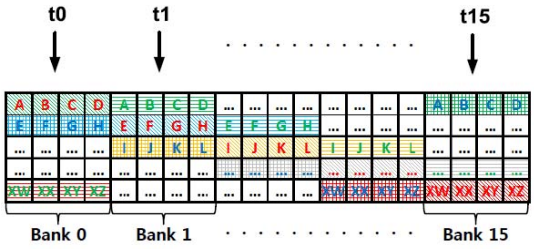
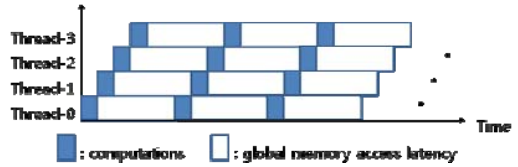


Figure 10. Data distribution to 16 memory banks using our store scheme

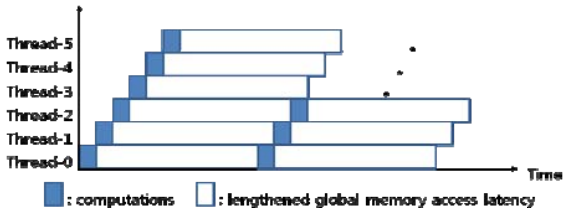
In previous research [17], they relied on the level-1 (L-1) data cache in their experimental GPU (Nvidia GeForce GTX 480) instead of using the shared memory. However, we believe that the efficient use of the shared memory leads to better performance than the L-1 cache. The shared memory is a programmer-controlled on-chip memory, thus an optimal data placement can lead to a significantly smaller number of cache misses and a smaller number of bank conflicts. In fact, the peak performance of the experimental GPU in [17] is at least 2-times better than our GPU (GTX 285). However, their performance result is not as good as ours. Thus our approach using the shared memory is a better idea. Previous research [19] use the shared memory, however, they did not address the issue of shared memory bank conflicts.

Besides the above shared memory related issues, other performance related considerations are as follows:

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



(a) Memory access latencies effectively hidden with multithreading



(b) Performance saturation due to excessive multithreading

Figure 11. Performance effects of multithreading

D. *Maximizing the Effects of Multithreading*

The GPU is executing in the multithreaded mode. 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 long time ($\geq$ 200 cycles). The bandwidth to the off-chip memory, however, has a limit. If there are too many threads accessing the off-chip memory concurrently, it can lead to congestions in the memory access paths and further lengthen the latencies [9]. Fig. 11 (a) depicts the case where an appropriate number of threads are used to effectively mask off the off-chip memory latencies by the multi-threaded execution. Fig. 11 (b) depicts the case where an excessive number of threads are generated which results in the lengthened off-chip memory access latencies due to the conflicts on the memory access paths by the generated threads. Therefore, finding an optimal number of threads to effectively hide the off-chip memory latencies while efficiently utilizing the large number of cores and the memory bandwidth is crucial for obtaining high performance. Our strategy finds and schedules an optimal number of parallel threads onto the hardware thread blocks and the processing cores inside the thread block by trying various input chunk sizes to be assigned to each thread.

VI. EXPERIMENTAL RESULTS

We implemented the AC algorithm in three ways: 1) serial implementation using single CPU core (S); 2) a parallel implementation on a GPU using the zero-copy method (P-1); 3) another parallel implementation using the shared memory (P-2). ((P-2) also uses the zero-copy method.) Our experiments are conducted on a system including the 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 the 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 the input data and the 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 as explained earlier in Section V.

A. *Run Time Comparisons*

Figures 12, 13, and 14 show the run times of the three approaches (S, P-1, P-2) for a range of input data sizes and a range of the numbers of patterns. Comparing these results, we have observed the followings:

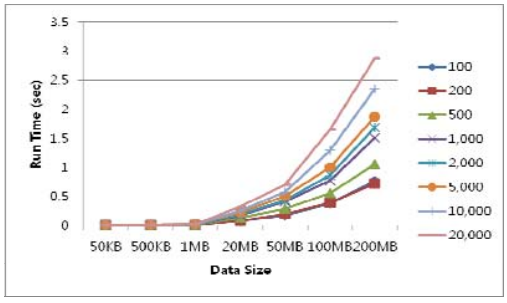


Figure 12. Run times for the serial (S) approach using different input data sizes and different numbers of patterns

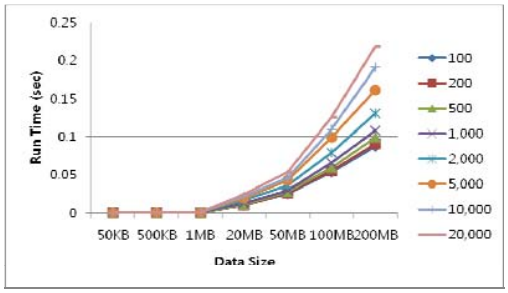


Figure 13. Run times for the P-1 approach using different input data sizes and different numbers of patterns

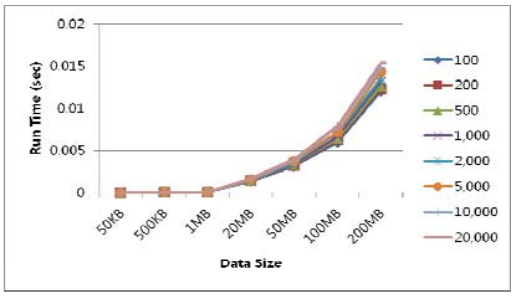


Figure 14. Run times for the P-2 approach using different input data sizes and different numbers of patterns

- 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 (P-2) 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 serial (S) results and the zero-copy only (P-1) results. Therefore, the shared memory approach (P-2) is resilient to the increase of the pattern data size.

The two parallel approaches (P-1, P-2) use the zero-copy method where the data in the host memory is directly accessed from the processing cores while they are implicitly copied to the global memory. Our experiments show that the zero-copy method performs significantly better than the explicit accesses to the global memory after the data is copied from the host memory. Figure 15 compares the data transfer times of the zero-copy and the explicit copy to the global memory.

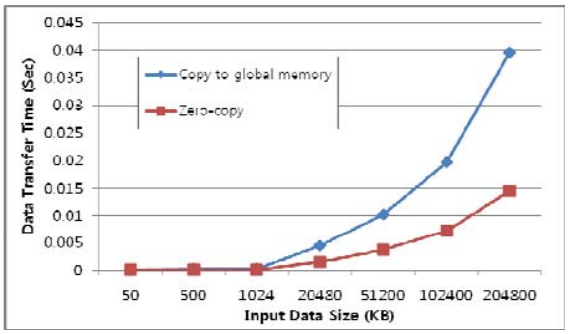


Figure 15. Data transfer time comparisons of the zero-copy method and the global memory approach

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 (P-2), however, the throughput decrease is much smaller with the increase in the number of patterns as we observed in the run time

comparisons. The maximum throughput reaches up to 127 Gbps for the 200MB input data when the number of patterns is 100.

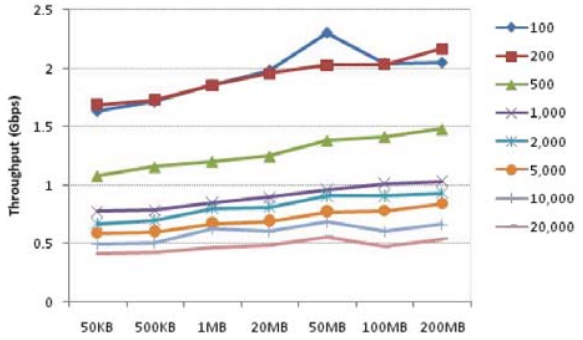


Figure 16. Throughput (measured in Gbps) for the serial (S) approach using different input data sizes and different numbers of patterns

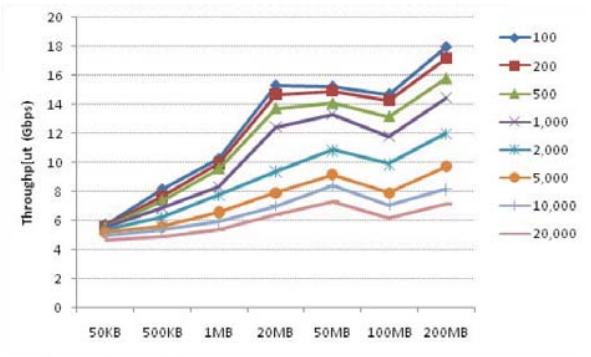


Figure 17. Throughput (measured in Gbps) for the P-1 approach using different input data sizes and different numbers of patterns

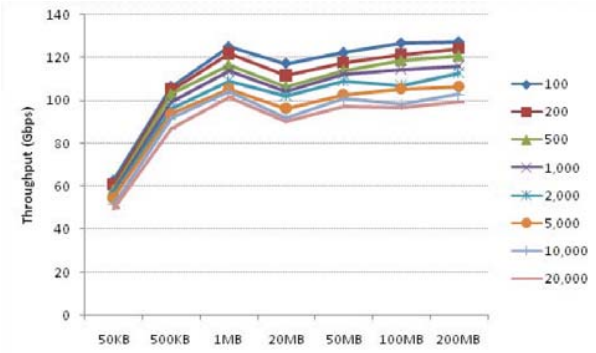


Figure 18. Throughput (measured in Gbps) for the P-2 approach using different input data sizes and different numbers of patterns

C. *Speedup Comparisons*

Figures 19, 20 and 21 show the speedups of the zero-copy only (P-1) approach and the shared memory (P-2) approach compared with the serial (S) approach, and the speedup of the P-2 approach over the P-1 approach:

- For the P-1 approach, the speedup ranges 3.3 – 13.2.
- For the P-2 approach, the speedup over the serial (S) version ranges 36.1 – 222.0. The maximum speedup achieved is 222.0 using 100MB input data and 20,000 patterns.
- The P-2 approach compared with the P-1 approach shows 7.3 – 19.3 times speedup. Thus the benefit of the shared memory is large.

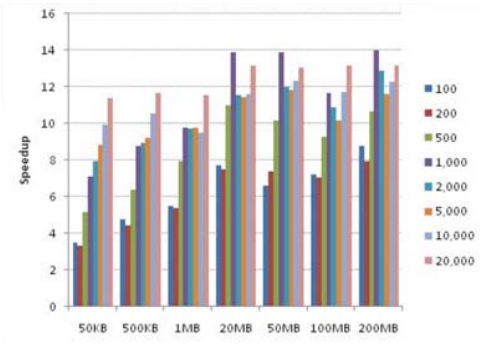


Figure 19. Speedup of the P-1 approach compared with the serial approach

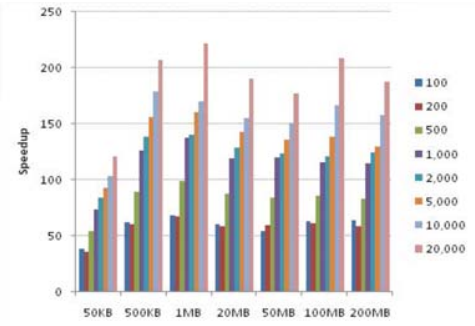


Figure 20. Speedup of the P-2 approach over the serial approach

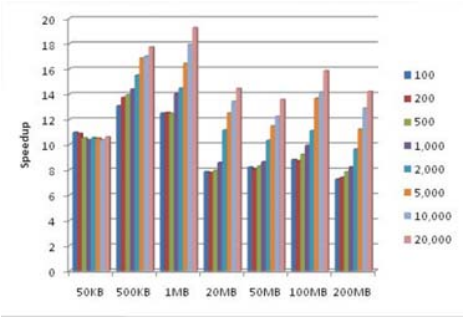


Figure 21. Speedup of the P-2 over the P-1 approach

D. *Effects of Avoiding Shared Memory Bank Conflicts*

As explained earlier in Section V, avoiding the shared memory bank conflicts improves the performance. Figure 22 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 the 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 conflict avoidance scheme gets larger.

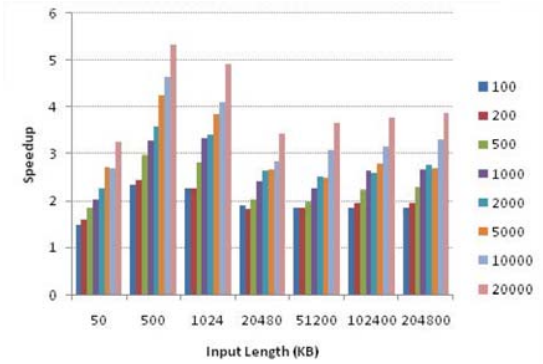


Figure 22. Speedup of our shared memory bank conflict avoiding scheme compared with the memory access coalescing only approach

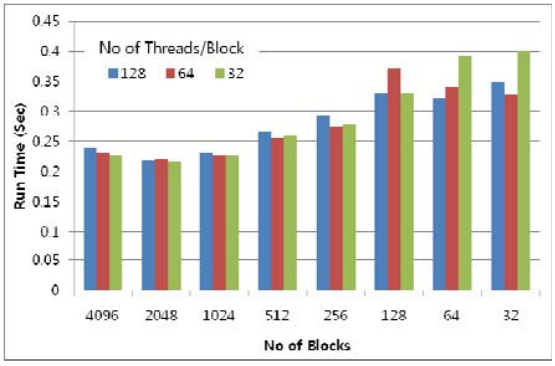


Figure 23. (No of blocks, No of thread/block) vs. run time for 200MB input data and 20,000 patterns using the P-1 approach

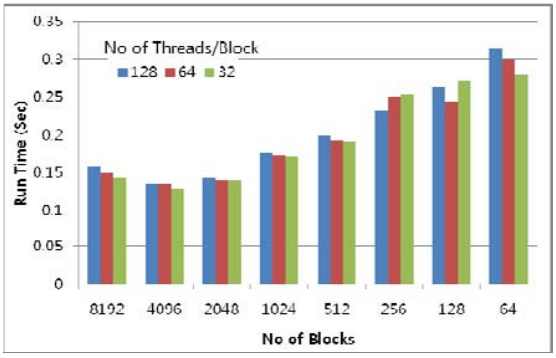


Figure 24. (No of blocks, No of thread/block) vs. run time for 200MB input data and 2,000 patterns using the P-1 approach

E. Finding and Sheduling Optimal Number of Blocks and Threads/Block

Finding an optimal number of blocks to be assigned to each thread block and the number of threads/block assigned to each thread processing core is crucial for obtaining high performance for the AC algorithm as explained in Section V. We've conducted extensive experiments to find optimal values:

- For the zero-copy only (P-1) approach, the global memory access latency and the texture memory latency can be hidden by the multithreading capability of the GPU. We've conducted a large number of performance tests to find the optimal number of blocks and the number of thread/block given a data size. Figure 23 shows the run times for 200MB data with 20,000 patterns using various numbers of blocks and threads/block. Assigning 2048 blocks and 32 threads/block gives the best performance. Figure 24 shows the run times when the number of patterns is reduced to 2,000. As in Figure 23, the best performance is obtained when 2048 blocks and 32-threads/block are used. In the same way, we find the best number of blocks and the number of threads/block for different data sizes and different pattern sizes.

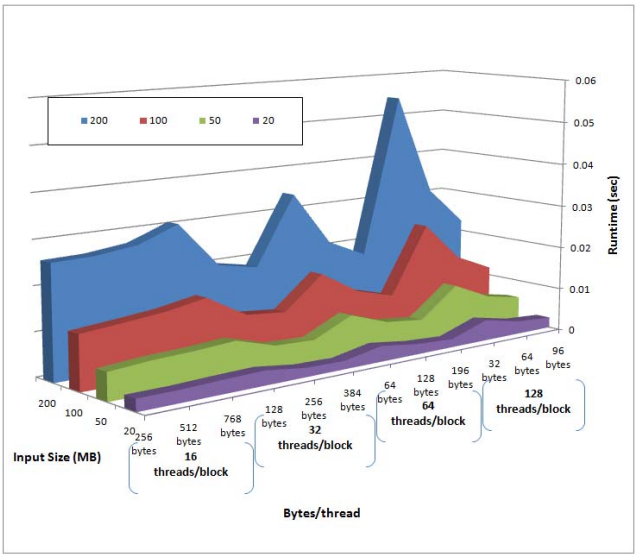


Figure 25. (No of threads/block, data size/thread) vs. run time for various input data sizes with 20,000 patterns

- Using the shared memory, the total input data size that we can keep in the shared memory is limited to 16KB (capacity of the shared memory). Therefore our optimization strategy focuses on finding the optimal number of threads/block and the data size per threads so that the total data size of the assigned number of threads does not exceed the 16KB capacity limit of the shared memory. When the data size per thread is small, we may accommodate multiple blocks without saving the contents of the shared memory to the global memory. Then the multithreading will come into play at the block level also.

Our optimization strategy uses up to 12KB for the input data of a thread block and uses the rest of the memory for some other work. Figure 25 shows the run times for the given number of threads and the data size per thread for the input data sizes in the rage of 20~200MB. The results show that when we assign 64-threads/block with 196-bytes per thread, we get the best run time. Since 64x196=12KB, we assign only one block to each thread block of the GPU. In the same way we find the best number of blocks and the best number of threads/block for different data sizes and different pattern sizes.

Our experiments to find an optimal number of blocks to be assigned to each thread block and the number of threads/block assigned to each thread processing core are by no means complete. However, to the best of our knowledge, no other previous research attempted similar experiments as ours before. We plan to extend our experimental framework to find an ultimate optimal scheduling method.

VII. CONCLUSION

In this paper, we proposed a performance optimization strategy for the AC algorithm on a GPU. The proposed strategy efficiently places and caches 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 host memory using the zero-copy method to the shared memory, we carefully arrange the host memory loads and the shared memory stores so that the number of the host memory accesses and the shared memory bank conflicts can be minimized. This significantly reduces the effective memory access latencies and efficiently utilizes the memory bandwidths. Furthermore, it maximizes the effects of the multithreading capability of the GPU by scheduling an optimal number of blocks and threads/block onto the hardware thread block and the thread processing cores on the block though extensive performance tests. Experimental results on a 4-core, 2.2Ghz Intel processor and Nvidia GeForce GTX 285 GPU using CUDA shows that our 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: 2010-0012059 and 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] R.H. Saavedra-Barrera, D.E. Culler, Thorsten von Eicken, "Analysis of multithreaded architectures for parallel computing", ACM Symposium on Parallel Algorithms and Architectures - SPAA , pp. 169-178, 1990

[10] 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

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

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

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

[14] 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

[15] 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

[16] 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

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

[18] 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

[19] 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

[20] 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