2012 IEEE 14th International Conference on High Performance Computing and Communications

Memory Efficient Parallelization for Aho-Corasick Algorithm on a GPU

Nhat-Phuong Tran, Myungho Lee*, Sugwon Hong, Minho Shin
Department of Computer Science and Engineering
Myongji University, 38-2 San Namdong, Cheo-In Gu
Yong In, Kyung Ki Do, Korea 449-728
myunghol@mju.ac.kr

***Abstract*—Pattern matching is a commonly used operation in many applications including image processing, computer and network security, bioinformatics, among many others. Aho-Corasick (AC) algorithm is one of the well-known pattern matching techniques and it is intensively used in computer and network security. In order to meet the real-time performance requirements imposed on these security applications, developing a high-speed parallelization technique is essential for the AC algorithm. In this paper, we present a new memory efficient parallelization technique which efficiently places and caches the input text data and the reference data in the on-chip shared memories and texture caches of the Graphic Processing Unit (GPU). Furthermore, the new approach efficiently schedules memory accesses in order to minimize the overhead in loading data to the on-chip shared memories. The approach cuts down the effective memory access latencies and leads to significant performance improvements. Experimental results on Nvidia GeForce 9500GT GPU shows up to 15-times speedup compared with a serial version on 2.2Ghz Core2Duo Intel processor, and 15Gbps throughput performance.**

Keywords-Aho-Corasick algorithm; computer security; GPU; parallelization

I. INTRODUCTION

Pattern matching is an important operation in various applications such as computer and network security, bioinformatics, among many others. In network intrusion detection, for example, intensive pattern matching is performed for a deep packet inspection [5, 9]. In bioinformatics, pattern matching is used for bio-sequence analysis for genome/protein matching [10, 12]. Among many pattern matching algorithms, Aho-Corasick (AC) algorithm [1] is commonly used for the above applications. The AC algorithm is a multiple patterns matching algorithm which can match multiple patterns simultaneously for a given finite set of strings (or dictionary). In order to speed up the pattern matching and meet the real-time performance requirement for these applications, efficient parallelization of AC algorithm is crucial.

Recently, the Graphic Processing Unit (GPU) is becoming increasingly popular in various applications. The GPU was originally introduced in the late 1990s as a co-processor for accelerating the simulation and visualization of 3D images. Earlier GPUs were designed more like Application Specific ICs (ASICs) with separate processing units for Shader, Vertex, Pixel. In the latest GPUs, however, those units are incorporated into multiple uniform programmable processing units or cores. With the new architecture, huge floating-point performance improvements are made possible [3, 4]. Furthermore, in order to utilize the flexible hardware design, user friendly programming environments have been recently developed such as CUDA from NVidia, OpenCL from Khronos Group, OpenACC from a subgroup of OpenMP Architecture Review Board (ARB). Using those environments along with the flexible GPU architecture has led to innovative performance improvements in many application areas, and many more are still to come [3, 14].

In this paper, we develop a memory-efficient parallelization technique in order to maximize the speedup of the AC algorithm on a GPU. For the efficient use of the memory, we carefully place the input string data and the reference data. For the input data whose size is at least a few hundred mega-bytes large, they can be totally or partly placed in the global memory of the Graphic-DRAM (GDRAM) or 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 memory access orders so that the number of global memory accesses can be minimized. The reference data with which the input string data is compared for a possible pattern match, we organized them in a 2-dimensional table called State Transition Table (STT) and place them in the texture memory so that the actively used part of the STT can be cached in the on-chip texture cache. 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 parallelization approach on a GPU (Nvidia 9500GT) using CUDA, we've observed a significant performance improvements (up to 15.72x speedup) compared with the performance on a general-purpose multi-core processor (2.2Ghz 4-core Intel processor). Furthermore, the new approach delivers up to 15Gbps throughput performance.

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 the programming model. Section IV explains our memory efficient parallelization

* Corresponding author: myunghol@mju.ac.kr

978-0-7695-4749-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HPCC.2012.65

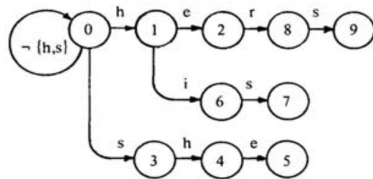
IEEE computer society

technique utilizing the on-chip caches in performing pattern matching operations. Section V shows the experimental results on Nvidia 9500GT GPU and a 4-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 parts. In the first part, a pattern matching machine called AC automaton (machine) is constructed from a finite set of patterns. In the second part, we apply the input text to the AC machine to find the locations that patterns occur [1]. AC automaton invokes three functions: a goto function *g*, a failure function *f*, and an output function *output*. Figure 1 shows the functions used by the AC machine [1] for a set of patterns {“he”, “she”, “his”, “hers”}:

- The directed graph in Figure 1(a) represents the goto function. (¬(‘h’, ‘s’) denotes all input symbols other than ‘h’, ‘s’.) The goto 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 maps a state into another state. It is consulted whenever the goto function reports a “fail”.
- The output function maps a set of keywords to output at the designated states [1].



(a)The *goto* function

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

(b)The *failure* function

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

(c)The *output* function

Figure 1. Functions used in AC algorithm

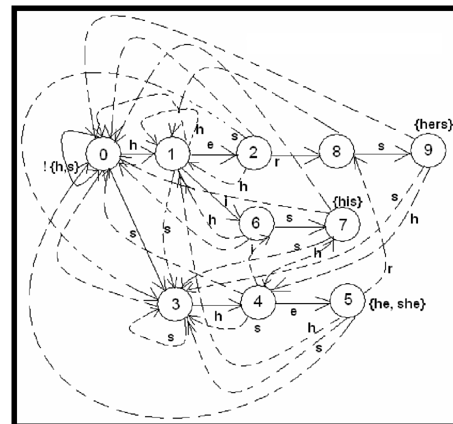
Assume that we have a text string “ushers”. AC machine works in the following manner:

- 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=f(5) (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. It helps the AC algorithm process the input text with complexity O(n), where n is the length of the input text. When the AC machine is implemented as a DFA, it represents all of possible states of the machine along with information of the acceptable state transitions of system [2]. 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 goto function and failure function introduced in Figure 1. The output function is also incorporated in the DFA. Figure 2 shows the AC machine implemented as a DFA.



(Dashed lines: fail transition)

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

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". (Note that this input string is different from the above input string used for a NFA). 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}.

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 visualization of 3D images commonly used in applications such as game programs. Since then GPU has become widespread and these days it is commonly incorporated in many computing platforms including desktop PC's, high performance computing servers, and even in mobile devices such as smart phones. The clock rate of the latest GPU has ramped up significantly compared with the earlier models. Furthermore recent GPUs have shown impressive performance for floating-point operations, far exceeding that of the latest CPUs and the performance gap is widening.

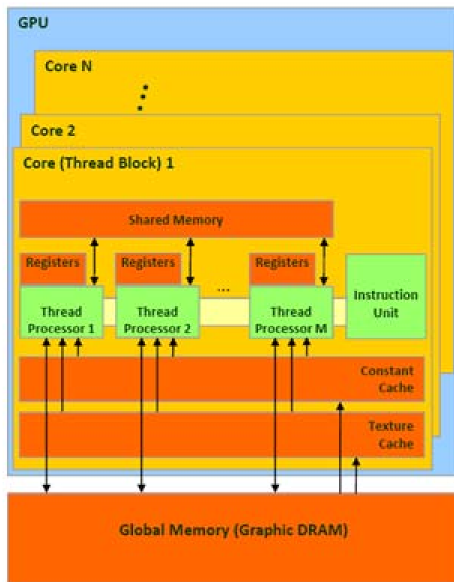


Figure 4. General architecture of a GPU

In the architecture of earlier GPUs, there were separate processing units for Shader, Vertex, Pixel. In the latest GPUs, those units are incorporated into multiple programmable processing units or thread processors which can be compared with a CPU core (see thread block or core in Fig. 4) [4]. (NVidia defines a thread processor as a core. However, we regard a thread block as a "core", because there is a shared "Instruction Unit" for all thread processors on a thread block. Thus, in our humble opinion, it is more suitable to call a thread block as a core.) The recent design is suitable for SIMD (Single Instruction Multiple Data) processing by having multiple threads assigned to each thread block executing the same instructions 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. In order to utilize the advanced flexible hardware design, more user friendly programming environments are recently developed. CUDA from NVidia, OpenCL from Khronos Group are good examples of such software environments [3]. Using those environments, programmers can have more direct control over the GPU pipeline and memory hierarchy, whereas in the old GPUs they relied on specific graphics API's. The flexible GPU hardware and user friendly software have led to a number of innovative performance improvements in many application areas and more improvements are still to come [3, 14].

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. They are registers and local memories belonging to each thread, a shared memory used in a thread block, and global memory accessed from all the thread blocks [3]:

- 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 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 which is 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 total size on-chip memory embedded on a thread block.)
- Registers are used for temporarily storing data used 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 also an area in G-DRAM. Thus it is also a slow memory.
- 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.

IV. PARALLELIZING AC ALGORITHM ON GPU

In this section, we first introduce previous researches on parallelizing Aho-Corasick algorithm. Then we introduce our memory efficient parallelization approach.

A. *Previous Researches*

The Aho-Corasick pattern matching algorithm has been previously applied in various application areas. In fact, network and computer security, and bioinformatics are two major fields where the AC algorithm is applied intensively. For example, it is used for a deep packet inspection used for 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 Graphics Processing Units (GPUs) are becoming increasingly popular, researchers are exploiting GPUs parallel processing capabilities to speed up the AC algorithm.

In the area of network intrusion detection, Giorgos Vasiliadis et al. presented an intrusion detection system based on the Snort open-source NIDS named Gnort [5]. In order to parallelize the pattern matching on GPU, authors proposed two approaches to process packets. First, each thread is assigned a single packet. Second, each thread block is assigned a single packet. The Gnort outperformed conventional methods using CPUs. In [9], Cheng-Hung Lin et al. proposed a new algorithm, called Parallel Failureless AC Algorithm (PFAC) which can remove all failure transitions in conventional AC state machine. PFAC allocates each byte of an input stream for a GPU thread to identify any pattern matching at the thread starting location.

In the area of bioinformatics, Antonino Tumeo and Oreste Villa presented an efficient implementation of the AC algorithm which is used to accelerate DNA analysis applications on a heterogeneous cluster [10]. In order to parallelize DNA analysis, authors partition the DNA sequence into multiple chunks and assign each chunk to a single CUDA thread. In [12], Xinyan Zha and Sartaj Sahni attempted to accelerate the AC algorithm by using GPU. The experimental results on NVIDIA Tesla GT200 shows that the speedup achieved was between 8.5 and 9.5 compared with a single thread CPU implementation and between 2.4 and 3.2 compared with the best multithreaded implementation. However, in the paper, authors assumed that the target string resides in the device memory and the results are to be left in the device memory. Moreover, the pattern data structure is precomputed and stored in the GPU device memory.

B. *Our New Approach*

1) Preparing Data for Pattern Matching on GPU

In order to implement the AC pattern matching machine, we use a DFA. This DFA makes exactly one state transition given each input character. Thus, we can use a matrix (called State Transition Table (STT)) to store the automaton structure. The rows represent states and the columns represent the input characters. We construct AC automaton on CPU and transfer it to the GPU side so that GPU can use it to find out possible pattern matching against the input text data. 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 indicates if the current state is a matched state). Figure 5 illustrates the STT which stores information of AC automaton.

Matching? / Input symbols / States

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

Figure 5. State Transition Table (STT)

STT is a common table which is accessed by all GPU threads randomly for the pattern matching. Once constructed on the CPU and copied onto GPU's device memory, it is used for read only. Thus, we use texture memory to store STT. Texture memory is a read-only memory space in the device memory and the data in the texture memory can be cached in the on-chip texture cache. Also, the texture cache is optimized for 2-dimensional spatial local data [3] which is suitable for 2-dimensional STT structure.

In order to implement pattern matching on GPU, the input text data and the reference data represented as the STT need to be transferred from the host memory to the device memory. Data transfer between the host memory and the device memory is a critical part when we parallelize applications on GPU in general, because the data transfer takes a lot of computation cycles. In order to overcome the data transfer overhead, we use the zero-copy feature of CUDA which enables GPU threads to directly access the host memory [3]. Thus, the data transfer can be performed asynchronously with the computations. In our implementation of AC algorithm, transferring the input data from the host memory to the device memory is executed only one time. Thus, with a large input data, this zero-copy feature makes the data transfer relatively faster compared to the computation time.

2) Memory Efficient Parallelizations

As the input text data and the reference data are ready in the GPU device memory (global memory and texture memory respectively), we need efficient parallelization strategies to speed up the pattern matching operations. We attempt to cache the reference data (STT) in the texture cache as much as possible to minimize the latencies for the random accesses to the STT while pattern matching operations are performed. The input text data is basically placed in the global memory. The accesses to the input data are generated from the multiple thread processors on the GPU. As the GPU executes in the

multithreaded fashion, the long global memory access latencies can be masked off or hidden by allowing multiple pattern matching operations on the same thread block. However, if we can figure out an efficient way to utilize the shared memory to cache the input text data, it will further speed up the pattern matching executions. In the followings, we introduce two step implementations of the AC pattern matching algorithm. In the first step, we use the global memory only to store the input text data on the GPU. Then we introduce the second step to cache the input text data from the global memory to the on-chip shared memory.

Global Memory Only Approach

- In this approach, we parallelize AC pattern matching algorithm in a straightforward way by storing the input text data in the global memory. We divide the input text into many chunks and assign each chunk to each thread processor on the same thread block. Multiple instruction streams share a thread block for multithreaded execution. Thus, the latencies for accessing the input text data can be partly or totally masked off by the multithreading effects.
- The intensive global memory accesses generated from each thread in performing parallel pattern matching can make significant performance degradations, although multithreaded execution can help hide the latencies.
- The reference data, STT, is bound to the texture memory at the initial phase. As the data placed in the texture memory get cache in the on-chip texture cache as they get referenced, we can exploit the temporal locality for the STT.
- Each thread accesses its own chunk directly from global memory and applies the pattern matching on the assigned 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 set of patterns.

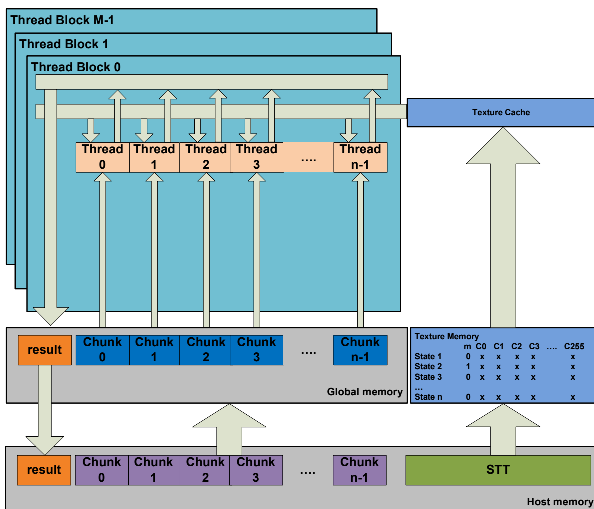


Figure 6. Illustration of global memory only approach

Shared Memory Approach

- In this approach using shared memory also, we first divide the input text into a number of chunks. Each chunk is processed by threads in each thread block. All threads in a block cooperate to load data from the global memory to the shared memory before applying the pattern matching (see Figure 7).

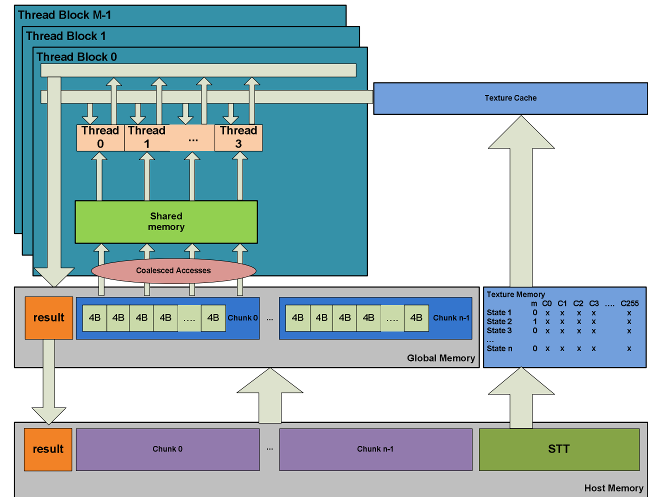


Figure 7. Illustration of shared memory approach

- While loading data, the most important performance consideration is to coalesce global memory accesses. Coalescing access is a coordinated read by a half-warp. On the NVIDIA 9500GT where the compute capability of the device is 1.1, the coalesced accesses require that the *k*-th thread in a half warp accesses the *k*-th word in a segment aligned to 16 times the size of the elements being accessed [3] (See Figure 8).

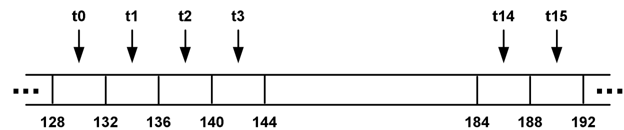


Figure 8. Coalesced accesses: read integer

- The input text is buffered in a sequential fashion in the memory. Each character contains one byte. If each thread slides on its own data and loads naively each character from the global memory to the shared memory, each thread reads a long sequence of data sequentially, Thus the above coalescing access requirement, the *k*-th thread in a half warp must access the *k*-th word, cannot be satisfied. Thus, the total latency to lead data needed for each thread will be high. In order to solve this problem, threads of a block must cooperate to read as a half warp and each thread read four bytes - one word of integer - at one time.
- Data needed to be loaded into the shared memory is longer than the size of the data loaded by threads of a block at one time. Thus, threads of a block need to load data multiple

times. For example, we assume the size of shared memory as 1024 bytes and 16 threads per block (see Figure 9). Because each thread read one four-byte word at one time, 16 threads of block need 1024 / (4*16) = 16 loading times from global memory to fill shared memory up.

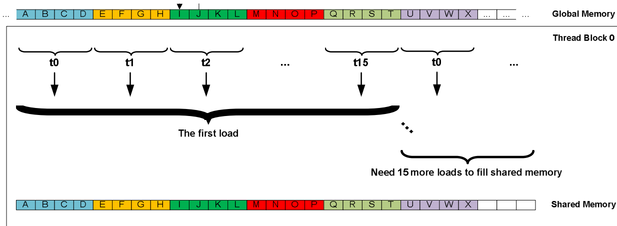


Figure 9. Loading data from global memory to shared memory (Shared memory size=1024 bytes, number of threads per block = 16, each thread reads 4 bytes at one time)

- In our approach, we initialize 32 threads for each block, the size of a warp on Nvidia 9500GT. Out of the 16KB shared memory space, we set the size for the input text as 8KB. Thus, each thread of a thread block needs to load 64 times (8192 / 4*32) to fill the allocated space in the shared memory up. The remaining 8KB space in the shared memory is used for other works.
- Synchronization of data will be done to make sure that all threads transferred data from the global memory to the shared memory successfully before threads process other works. After transferring data to the shared memory, each thread will apply AC algorithm on its own chunk as mentioned in global memory approach.

V. EXPERIMENTAL RESULTS

We implemented the parallel AC algorithm using CUDA on Nvidia GeForce 9500GT GPU which contains 32 streams processors (or thread processors) organized in 4 multiprocessors (or thread blocks), operating at 1.35 GHz with 256MB device memory. Our experimental system includes Intel multicore processor (2.2Ghz Intel Core2Duo 4) with 2GB of main memory. The OS is Centos 5.5 Linux.

We conducted the following three experiments:

- Serial version of AC algorithm: We implemented AC algorithm serially on Intel Core2Duo processor. The construction phase and the matching phase were both executed on single CPU core.
- Global memory version: We parallelized the AC algorithm from the serial version to the GPU version using CUDA. However, the construction of STT part still runs on a CPU core. After constructing the STT, it is moved to the texture memory on the GPU side. Matching phase is executed on the GPU. Threads access the input text data directly from the global memory.
- Shared memory version: Each block processes a chunk of data. Threads of a block coordinate to load data into the shared memory before applying the pattern matching on the shared memory.

In order to measure the performance of each version, we conducted experiments using different input lengths and different number of patterns. The numbers of patterns used are 100, 200, 500, and 1000. The input lengths used are 1MB, 10MB, 40MB, 100MB, 200MB. In all experiments we conducted, we ignored the time spent in the construction phase of STT (in serial version also) which run on single CPU core. In our opinion, this is fair because the STT construction is performed only once for a given finite set of strings, whereas the pattern matching process is performed a large number of times.

Through our experiments, we've obtained the run times of three approaches: serial, global memory, and shared memory. Comparing these results, we've observed the followings:

- Figure 10 shows the results of three approaches when the number of patterns is fixed at 1,000. The run times of all the three versions increase linearly with the increased data sizes.

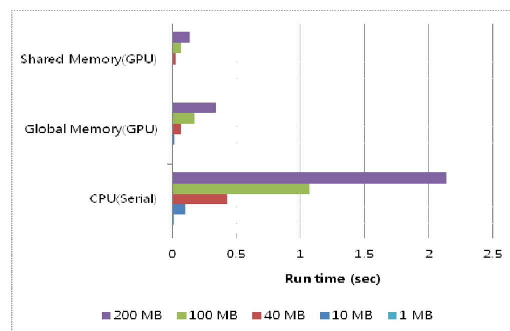


Figure 10. Run time comparison of three approaches using different input string sizes (number of patterns is fixed: 1,000)

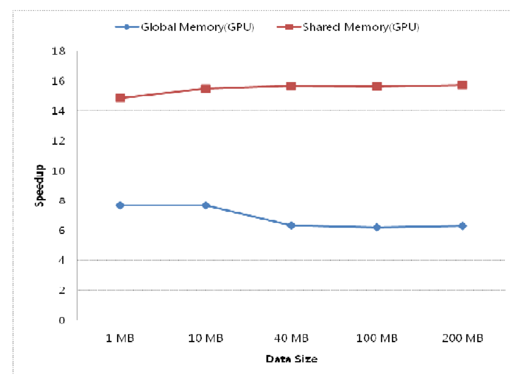


Figure 11. Speedups of global memory and shared memory versions compared with serial version using 1,000 patterns

- As shown in Figure 11, the maximum speedup achieved is 15.72x using 200MB input string data and 1000 patterns. For global memory approach, when a thread needs to match a pattern, data is fetched from global memory to registers. This takes a long time, because the access latency for the global memory takes hundreds of clock cycles while the access latency for the shared memory takes just a few clock cycles. Thus, the global memory version runs faster about 4.71 – 5.76 times only compared with the serial version, whereas the shared memory version runs faster by 14.86 – 15.72 times.

- Figure 12 shows that, with the input string size of 200MB, the run times increase with the number of patterns in the serial version. The run times of the other two parallel versions on the GPU change insignificantly with the number of patterns. This is especially true for the global memory version. This is because the multiple thread processors can handle multiple pattern matchings on the GPU, whereas, in the serial version on CPU, single CPU has to handle multiple patterns.

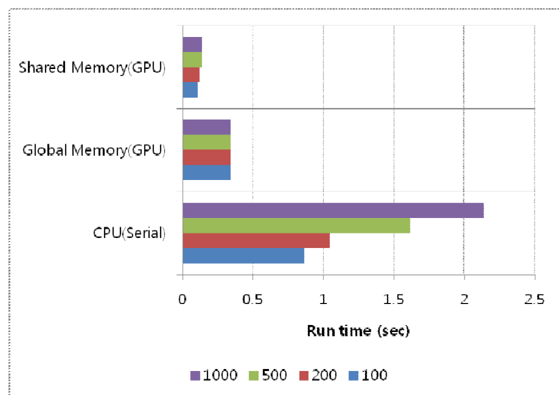


Figure 12. Run time comparison of three approaches using different numbers of patterns (Input string size is fixed: 200MB)

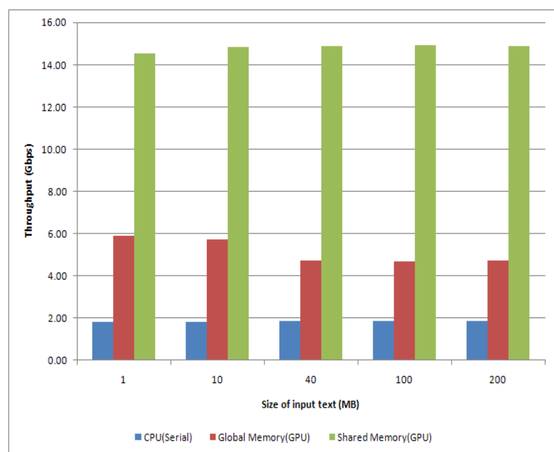


Figure 13. Throughput of three versions with number of patterns = 100

- Figure 13 shows the throughput measured in Gbps for the three versions with the number of patterns=100. While the serial version's throughput is below 2Gbps, the shared memory version's throughput reaches up to 15Gbps.

VI. Conclusion

In this paper, we proposed a memory efficient parallelization approach for AC algorithm. The proposed approach parallelizes the AC by efficiently placing and caching both the input text string data and the reference data organized as a 2-dimensioanl table (STT) in the on-chip shared memories and texture caches. This significantly reduces the average memory access latencies and leads to impressive performance improvements. Experimental results on a 4-core, 2.2Ghz Intel processor with 2GB DRAM running Centos5.5 OS and Nvidia 9500GT GPU using CUDA shows that the new approach achieves up to 15.72x speedup compared with the sequential version run on a single core of Intel Core2Duo processor. Furthermore, the speedup is larger with the input data size gets larger as parallel pattern matching performed using the multiple thread processors on the GPU becomes more effective as the data size gets larger. The resulting throughput performance also reaches up to 15Gbps on the Nvidia 9500GT GPU.

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).

References

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

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

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

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

[6] B. He, N. Govindaraju, Q. Luo, B. Smith, "Efficient Gather and Scatter Operations on Graphics Processors", Proceedings of the SuperComputing 07, pp. 175-186, Nov 2007.

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

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

[9] 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.

[10] L. Spracklen and S. Abraham, "Chip MultiThreading: Opportunities and Challenges", 11th International Symposium on High-Performance Computer Architecture (HPCA-11), pp 248-252, 2005.

[11] Tumeo, A., Villa, O., "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

[12] Xinyan Zha, Sahni, S., "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

[13] 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

[14] 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