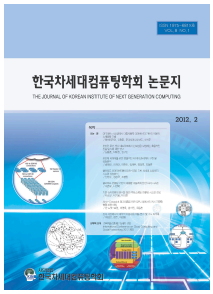


한국차세대컴퓨팅학회 논문지
Vol.8 No.1

ISSN : 1975-681X(Print)

Cache Conscious Parallel Pattern Matching for Aho-Corasick Algorithm on a GPU

Nhat-Phuong Tran, Myungho Lee, Sugwon Hong, Dong Hoon Choi

To cite this article : Nhat-Phuong Tran, Myungho Lee, Sugwon Hong, Dong Hoon Choi (2012) Cache Conscious Parallel Pattern Matching for Aho-Corasick Algorithm on a GPU, 한국차세대컴퓨팅학회 논문지, 8:1, 64-75

Aho-Corasick 알고리즘을 위한 GPU 상에서의 캐시 지향형 병렬 패턴 매칭

Cache Conscious Parallel Pattern Matching for Aho-Corasick Algorithm on a GPU

[1] 쟌 느앗-프엉, [1]* 이명호, [1] 홍석원, [2] 최동훈
Nhat-Phuong Tran, Myungho Lee, Sugwon Hong, Dong Hoon Choi

[1] (449-728) 경기도 용인시 처인구 남동 산 38-2 명지대학교 컴퓨터공학과
[2] (305-806) 대전시 유성구 대학로 245 한국과학기술정보연구원(KISTI)
myunghol@mju.ac.kr

요 약

패턴매칭 연산은 네트워크 보안, 바이오 인포매틱스와 같은 분야에서 광범위하게 사용되는 중요한 연산이다. 많은 패턴 매칭 알고리즘 들 중에서 Aho-Corasick (AC) 알고리즘이 위와 같은 분야에서 집중적으로 활용되고 있어, AC 알고리즘의 실행을 가속화하고 실시간 실행을 위한 성능 상의 요구사항을 만족시키기 위하여 효율적인 병렬화 기법을 개발하는 것이 필수적이다. 본 논문에서는 AC 알고리즘의 실행에 활용되는 입력 텍스트 데이터와 비교의 대상이 되는 2-차원 배열로 구성된 레퍼런스 데이터를 모두 GPU 상의 on-chip 메모리 (또는 캐시)에 적재하여 병렬 패턴 매칭을 실행하는 기법을 개발한다. 이러한 새로운 접근법의 개발에 있어 데이터를 on-chip의 shared memory에 적재할 때 필요한 메모리 접근들을 효율적으로 스케쥴링 함으로써 데이터 적재에 드는 오버헤드를 크게 감소시킨다. 따라서 새 접근법은 데이터 적재에 드는 메모리 대기시간을 크게 줄이고, 이에 따라 큰 폭의 성능 향상을 얻게 된다. NVidia 9500 GT GPU를 활용한 실험결과, 본 논문의 접근법을 활용한 병렬실행 결과 순차실행 결과(Intel Core2Duo 범용 마이크로프로세서를 활용한)와 비교하여15배까지 성능을 향상시킬 수 있었다.

Abstract

Pattern matching is a common and important operation in many applications including network security, bioinformatics, etc. Among many pattern matching algorithms, Aho-Corasick (AC) algorithm is intensively used in these applications. In order to speed up and meet the real-time performance requirement for AC algorithm, developing an efficient parallelization technique is essential. In this paper, we develop a new parallelization approach to cache both the input text data and the reference data organized as a 2-dimensional table in the on-chip memories (or caches) on the Graphic Processing Unit (GPU). The new approach also schedules memory accesses carefully to minimize the overhead in loading data to the on-chip shared memory. The approach significantly cuts down the memory latency to load the data and leads to impressive performance improvement. Experimental results on NVidia GT9500

* 교신저자

논문투고일자 : 2012. 01. 30 심사일자 : 2012. 01. 30 게재확정일자 : 2012. 02. 20

GPU shows up to 15x speedup compared with a serial version on 2.2Ghz Core2Duo Intel processor.
키워드: 패턴 매칭, Aho-Corasick 알고리즘, GPU, 병렬화
Keyword: pattern matching, Aho-Corasick algorithm, GPU, parallelization

1. 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, 8]. In computational biology, pattern matching is used for biosequence analysis for genome/protein matching [9, 10]. Among many pattern matching algorithms, Aho-Corasick (AC) algorithm [1] is commonly used for the above applications. The ACalgorithm 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, 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, 12, 13].

In this paper, we develop a new parallelization technique to speed up the AC algorithm which needs real-time processing. In the new parallelization approach, we attempt to cache both the input text data and the reference data organized as a 2-dimensional table in the on-chip memories (or caches) on the GPU. Furthermore, in order to efficiently load the input text data to the shared memory, we carefully arrange the memory access orders so that the number of global memory accesses can be minimized. The approach significantly cuts down the memory latency to load the data and leads to impressive performance improvement for the AC algorithm. By implementing the proposed parallelization approach on a GPU (NVidia GT9500) using CUDA, we've observed a significant performance improvements (up to 15x speedup) compared with the performance on a general- purpose multi-core processor (2.2Ghz 4-core Intel processor).

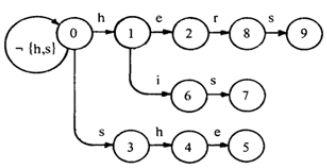
The rest of the paper is organized as follows: Sections 2 gives an overview of AC algorithm. Section 3 shows the architecture of the latest GPU and the programming model. Section 4 explains our parallelization technique efficiently utilizing the on-chip caches in performing pattern matching operations. Section 5 shows the experimental results on NVidia GT9500 GPU and a 4-core Intel processor for comparison with GPU performance. Section 6 wraps up the paper with conclusion.

2. 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). AC algorithm is used commonly used in computer and network security and bioinformatics. When AC is

implemented as a NFA, it is also applicable to search engines for a better word prediction and pattern matching in the text boxes. 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*. Fig. 1 shows the functions used by the AC machine [1] for a set of patterns {"he", "she", "his", "hers"}:

- The directed graph in Fig. 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

Fig. 1 Example of a figure caption.

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

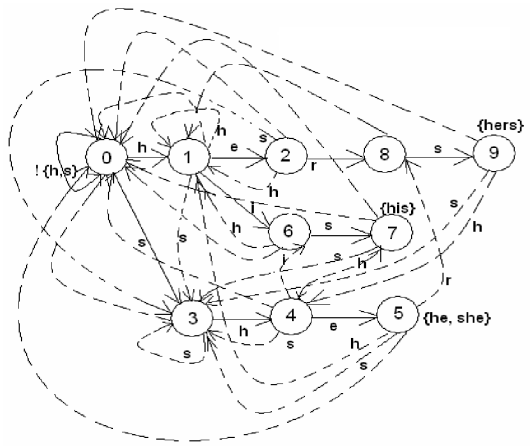
- Starting with state 0, the machine loops back to state 0 since g(0,'u')=0. For the same reason, the machine enters states 3, 4, 5 sequentially while processing the string 's', 'h', 'e' (g(0,'s')=3, g(3,'h')=4, g(4,'e')=5) and emits output, indicating that it has found the keywords "she" and "he" at 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".

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, δ(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 Fig. 1. The output function is also incorporated in the DFA. Fig. 2 shows the AC machine implemented as a DFA.

```
/*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
```

Fig. 2 Pseudocode of the AC machine implemented as DFA

Fig. 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:



(Dashed lines: fail transition)

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

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

3. 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.

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, 12].

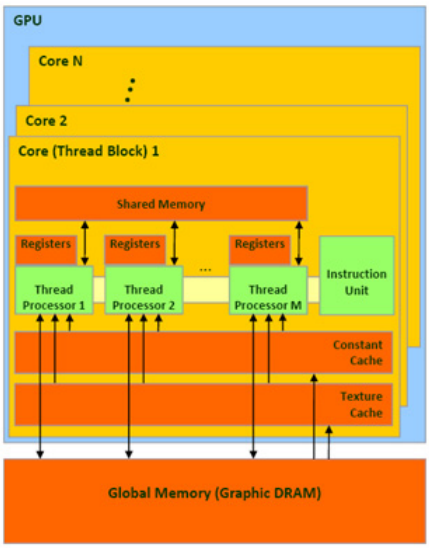


Fig. 4 General architecture of a GPU

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.

4. Parallelizing AC Algorithm on GPU

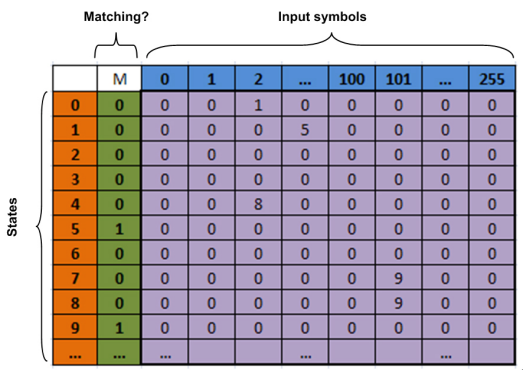
4.1 Previous Researches

The Aho-Corasick pattern matching algorithm has been applied in many areas. In fact, network intrusion detection and computational biology 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 and virus scan. It is also used in biosequence analysis for genome/protein matching. Furthermore, as Graphics Processing Units (GPUs) are becoming increasingly popular, researchers are exploiting GPUs to speed up the AC algorithm also.

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 [8], 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 computational biology, 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 [9]. In order to parallelize DNA analysis, authors partition the DNA sequence in multiple chunks and assign each chunk to a single CUDA thread. In [10], 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.



	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
...	...	...			...			...	

Fig. 5 State Transition Table (STT)

4.2 Our Approaches

(1) Storing and Transferring Data to GPU

In order to implement pattern matching on GPU, the data from the host memory need to be transferred to the device memory. Data which needs to be transferred are the input text and the STT which is constructed on CPU for storing state transition information in the AC pattern matching machine. 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 this paper, 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.

In this paper, we use a DFA to implement the AC pattern matching machine. This DFA makes exactly one state transition on 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 matching patterns. 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). Fig. 5 illustrates the STT which stores information of AC automaton.

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, using texture memory to store STT is a better choice compared with

the shared memory. 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-D STT structure.

(2) Parallelization Methods

First we implemented AC pattern matching algorithm using the global memory only to store the input text data on the GPU. Then we used the shared memory to cache the input text from the global memory.

Global Memory Only Approach

In this approach, we parallelize AC pattern matching algorithm in a straightforward way. Fig. 6 presents the idea of this approach. We divide the input text into many chunks. Each chunk is assigned to each thread. The input text is transferred to the global memory on the device from the CPU side host memory using zero-copy feature which was mentioned above. The STT is bound to the texture memory at the initial phase. Each thread accesses its own chunk directly from global memory and applies the pattern matching on the assigned chunk. Although this assignment helps balance the workloads among threads, the intensive global memory accesses generated from each thread in performing parallel pattern matching can make significant performance degradations. Also, the 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.

Shared Memory Approach

In the approach using shared memory also, we first divide the input text into a number of chunks. Each chunk is processed by threads in each block. All threads in a block cooperate to load data from the global memory to the shared memory before applying the pattern matching (see Fig. 7). 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 Fig. 8).

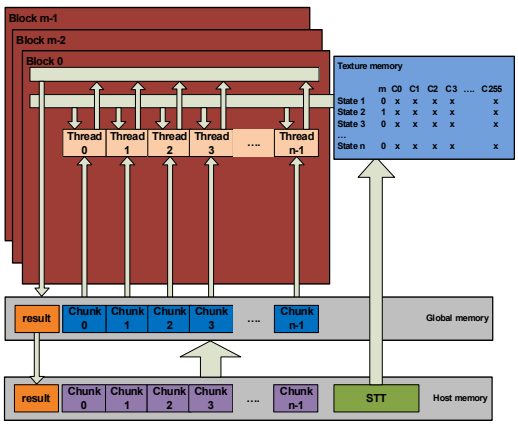


Fig. 6 Illustration of global memory only approach

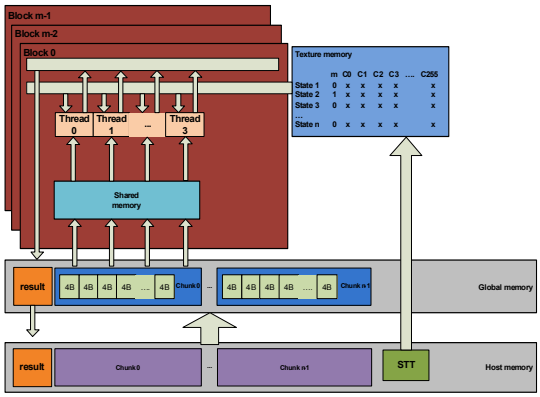


Fig. 7 Illustration of shared memory approach

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 data sequence and the above coalescing access

requirement, the k-th thread in a half warp must access the k-th word, cannot be satisfied. Thus, the latency 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. 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 (see Fig. 9).

Fig. 8 Coalesced accesses: read integer

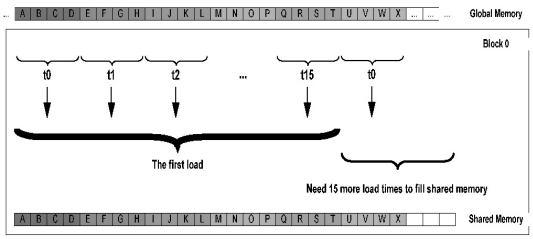


Fig. 9 Loading data from global memory to shared memory
(Shared memory size=1024 bytes, number of threads per block = 16, each thread reads 4 bytes at one time)

5. Experimental Results

We implemented the parallel AC algorithm using CUDA. In our experiments to evaluate the performance of our approaches, we used NVIDIA GeForce 9500GT GPU which contains 32 streams processors organized in 4 multiprocessors (or thread blocks), operating at 1.35 GHz with 256MB device memory. Also in our experimental system, the multicore processor is 2.2 Ghz 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 a 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. Table 1 and 2 list the number of patterns and the input lengths used for our experiments. For the test input data, we generated a synthetic trace of strings composing a dictionary. Since the dictionary consists of a fixed set of strings, the histogram of strings in the generated synthetic trace shows a uniform distribution. In all experiments we conducted, we ignore 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. Table 3 (see page 73) shows the runtimes of three versions: serial, global memory, and shared memory. As indicated in the last column of Table 1, the maximum speedup achieved is

15.72x using 200MB data and 1000 patterns. Fig. 10 shows that, with the input string size of 100MB, the runtimes increase linearly with the number of patterns in the serial version. The runtimes of the other two versions run on GPU change insignificantly with the number of patterns. This is especially true for the global memory version. Fig. 11 shows the results of three approaches when the number of patterns is fixed (200). The runtimes of all the three versions increase linearly with the increased data sizes.

Table 1 Number of patterns used in our experiments

Number of patterns
100 patterns
200 patterns
500 patterns
1000 patterns

Table 2 Different input lengths used in our experiments

Size(MB)
1 MB
10 MB
40 MB
100 MB
200 MB

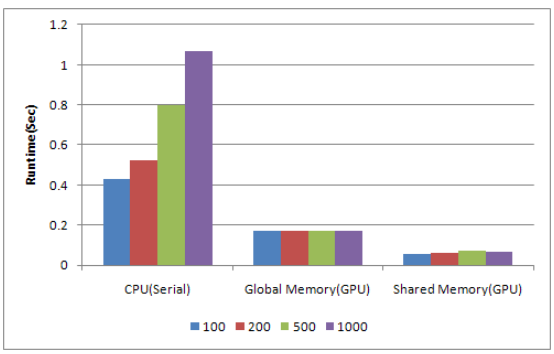


Fig. 10 Runtime comparison of three approaches using different numbers of patterns (Input string size is fixed: 100MB)

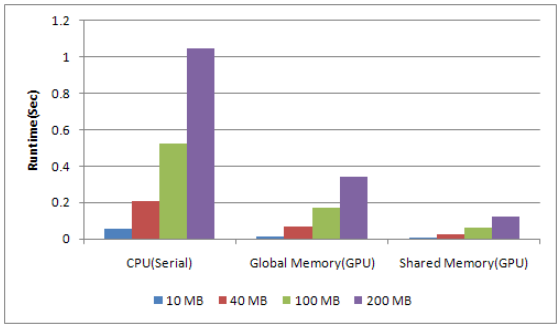


Fig. 11 Runtime comparison of three approaches using different input string sizes (Number of patterns is fixed: 200)

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 2.5－7.6 times only compared with the serial version, whereas the shared memory version runs faster by 8－15.7 times. The Fig. 12 shows the speedups of the global memory and the shared memory versions compared with the serial version when the input string size is fixed at 200MB. The speedup increases when the number of patterns increases. Fig. 13 shows the throughput measured in Gbps for the three versions with the number of patterns=100. While the serial version's throughput is just below 2Gbps, the shared memory version's throughput is up to 16Gbps. The general performance trend is that with a larger number of patterns, the speedup is larger. This is because, with a small number of patterns, the computing cores are not fully utilized, thus the performance gap between the serial execution and parallel execution is also limited. When the number of patterns is large, it can fully utilize the cores on the GPU. Thus leads to a larger speedups.

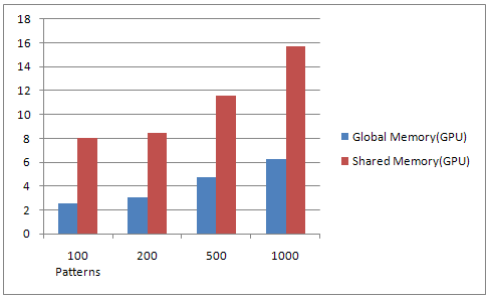


Fig. 12 Speedups of global memory and shared memory versions compared with serial version using different numbers of patterns (Input string size is fixed: 200MB)

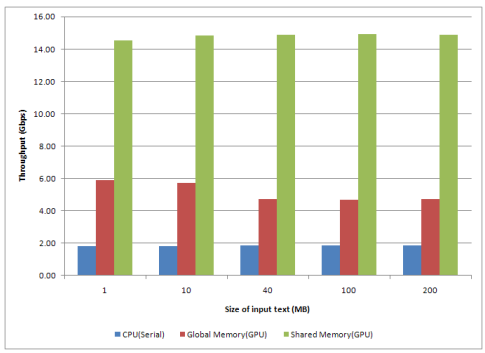


Fig. 13 Throughput of three versions with number of patterns = 100

6. Conclusion

In this paper, we proposed a new parallelization approach for AC algorithm. The proposed approach parallelizes the AC by efficiently caching both the input text string and the reference data (STT) in the on-chip caches. This reduces the memory access latencies significantly and leads to impressive performance improvements. Experimental results on a 4-core, 2.2Ghz Intel processor with 2GB DRAM and NVidia GT9500 GPU, running Centos5.5 OS, and using CUDA shows that the new approach achieves up to 15x speedup compared with the sequential version run on a single core of Intel Core2Duo processor. Furthermore, the speedup is larger with the larger input data size. More experiments are planned for furthering the improvement using both the multiple CPU cores besides the GPU.

Table 3 Runtimes of three different approaches and speedups of the new approaches

Number of patterns	Size(MB)	Serial		CUDA 1(Global Memory)		Speedup 1	CUDA 2(Shared Memory)		Speedup 2
		Run time	Throughput(Gbps)	Run time	Throughput(Gbps)	CUDA 1/ Serial	Run time	Throughput(Gbps)	CUDA 2/ Serial
100	1	0.0044	1.82	0.00136	5.88	3.24	0.00055	14.55	8.00
	10	0.044	1.82	0.01403	5.70	3.14	0.0054	14.81	8.15
	40	0.1723	1.86	0.0677	4.73	2.55	0.0215	14.88	8.01
	100	0.4305	1.86	0.171	4.68	2.52	0.0536	14.93	8.03
	200	0.8614	1.86	0.3398	4.71	2.54	0.1076	14.87	8.01
200	1	0.0052	1.54	0.0014	5.71	3.71	0.00062	12.90	8.39
	10	0.0523	1.53	0.0138	5.80	3.79	0.00614	13.03	8.52
	40	0.2089	1.53	0.0685	4.67	3.05	0.0245	13.06	8.53
	100	0.5237	1.53	0.1724	4.64	3.04	0.0612	13.07	8.56
	200	1.0482	1.53	0.34	4.71	3.08	0.1235	12.96	8.49
500	1	0.008	1.00	0.0014	5.71	5.71	0.0007	11.43	11.43
	10	0.0801	1.00	0.0137	5.84	5.85	0.007	11.43	11.44
	40	0.3218	0.99	0.0671	4.77	4.80	0.02787	11.48	11.55
	100	0.8	1.00	0.1725	4.64	4.64	0.06953	11.51	11.51
	200	1.6122	0.99	0.3398	4.71	4.74	0.1393	11.49	11.57
1000	1	0.0107	0.75	0.0014	5.71	7.64	0.00072	11.11	14.86
	10	0.1063	0.75	0.0139	5.76	7.65	0.00686	11.66	15.50
	40	0.4274	0.75	0.0677	4.73	6.31	0.02727	11.73	15.67
	100	1.0659	0.75	0.1725	4.64	6.18	0.06814	11.74	15.64
	200	2.1347	0.75	0.3396	4.71	6.29	0.1358	11.78	15.72

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"

[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] Soumya Sen, "Performance Charaterization and Improvement of Snort as an IDS"

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

[8] 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

[9] 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

[10] 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

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

[12] 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.

[13] J. Jeon, S. Hong, J. Bae, M. Lee, "Financial Derivatives Modeling Using GPU", Journal of KINGPC, 2009/03.

저자소개

◆ Nhat-Phuong Tran

- B.S., Information Technology from Natural Science University, Vietnam in 2004
- M.S., degree in Computer Science and Engineering, Myongji University, Republic of Korea, 2012
- Currently a Ph.D student in the Dept of Computer Science and Engineering, Myongji University
- Research interests are computer network and high performance computing

◆ Myungho Lee, Ph.D

- B.S., Computer Science and Statistic, Seoul National University, Korea
- M.S., Computer Science, University of Southern California, USA
- Ph.D., Computer Engineering, University of Southern California
- Staff Engineer, Scalable Systems Group, Sun Microsystems, Inc, Sunnyvale, California, USA.
- Currently an Associate Professor in the Dept of Computer Science and Engineering, Myongji University
- Research interests are high performance computing architecture, compiler, applications

◆ Sugwon Hong, Ph.D

- B.S., Physics, Seoul National University
- M.S., Ph.D., Computer Science, North Carolina State University
- Researcher, Korea Institute of Science and Technology (KIST), Energy Economics Institute (KEEI), SK Energy Ltd., and

Electronic and Telecommunication Research Institute (ETRI), Korea
• Currently a professor in the Dept. of Computer Science and Engineering, Myongji University since 1995.
• Major research fields are network protocol and architecture, network security

◆ **Dong Hoon Choi, Ph.D**

• B.S., Computer Science and Statistics, Seoul National University, Korea
• M.S., Computer Science, Korea Advanced Institute of Science and Technology (KAIST)
• Ph.D., Computer Science, Northwestern University, USA.
• Currently a researcher at Korea Institute of Science and Technology Information (KISTI)
• Primary research areas are cloud computing, virtualization. database systems, bioinformatics.