16

Graphics Processor-based High Performance Pattern Matching Mechanism for Network Intrusion Detection

Nen-Fu Huang, Yen-Ming Chu and Hsien-Wen Hsu
National Tsing Hua University
Taiwan

1. Introduction

As high-speed networking technology has progressed, the current network environment comprises many applications. However, many users still feel uncertain about these network applications due to security issues. Intrusion detection and prevention systems (IDS/IPS) are designed to detect and identify diverse threats over the network, such as worms, virus, spyware, and malicious codes, by performing deep packet inspection on packet payloads. Deep packet inspection is used to perform various processing operations in the entire packet, including the header and payload. Therefore, searching keywords in each traffic stream forms a bottleneck. That is, string matching is always an important issue as well as significant challenge in high speed network processing. For instance, Snort (Roesch, 1999), the most famous and popular open source IDS, takes over 2,500 patterns as signatures and takes more than 80% of CPU time for pattern matching. Thus, IDS need an efficient pattern matching algorithm or other mechanisms to speed up this key operation. Otherwise, an under-performing system not only becomes the network bottleneck but also misses some critical attacks.

Pattern matching algorithms have been studied for a long time, such algorithms include the Boyer Moore algorithm which solves single-pattern matching problem (Boyer & Moore, 1977) and the Aho-Corasick (AC) (Aho & Corasick, 1975) and Wu-Manber (Wu & Manber, 1994) algorithms, which solve multi-pattern string-matching problems. Research in this field has recently become popular again owing to the requirements for processing packets, especially for deep packet inspection applications. Various new concepts and algorithms have been proposed and implemented, such as Bitmap AC (Tuck et al., 2004), parallel bloom-filter (Dharmapurikar et al., 2004), reconfigure silicon hardware (Moscola et al., 2003) and TCAM-based mechanism (Yu et al., 2004).

Implementations of IDS can be categorized into hardware-based approaches and software-based approaches. The design concept for data structures and algorithms are usually different for these two implementations. The hardware approach is often used for network-based IDS, which is usually placed in the entrance of a local area network (LAN) and is responsible for scanning suspicious packets through it. Most of them store the famous Snort signatures, which are the collection of the characteristic of many network attacks, in the database to perform pattern matching. In order to process packets quickly and flexibly, parallel processing is the main architecture employed for network processing. The network processor (NP) is the most representative of these implementations. However, traditional network processors still suffer from poor performance and high cost when perform deep packet inspection, even though applying network processors for pattern matching has been proposed (Liu et al., 2004). Hence, many network security chip vendors have released special-purpose silicon products for accelerating the work of pattern matching. Nevertheless, such solutions are always expensive because of insufficient sales volume in the market.

On the other hand, software-based solutions, such as anti-virus software, personal firewalls, are very popular, especially in personal computers and servers. According to the reports, the security software market in Asia/Pacific (excluding Japan) is expect to grow up to over $US1100 millions in 2007 (IDC, 2006, Asia) and the market in Japan will also reach $US1927 million in 2010 (IDC, 2006, Japan), respectively. In terms of software, pattern matching is still necessary to detect network intrusion or to scan suspicious files. Form example, some famous network security software, such as Norton anti-virus, Trend-Micro pc-cillin, and Kaspersky anti-virus, have implemented the intrusion detection component in it. That is, host-based IDS becomes more and more common nowadays. However, the task of pattern matching slows down the system performance significantly because there is no additional hardware for accelerating. The problem is more crucial for servers, which often have to handle hundreds to thousands of connections simultaneously.

This study has found that graphics processors could constitute a solution for end hosts to perform pattern matching efficiently. With the parallel nature of graphics processors, the performance of pattern matching is greatly improved, even outperforms some previous hardware solutions. The personal computer has now become a standard consumer electronic device, particularly because of its ability to play PC/TV games, which increasingly require 3D processing. Players now demand for real-time, smooth and vivid frame transition, leading to the rapid development of graphics related technologies. Graphics processors are capable of increasingly powerful computation, even surpassing that of general processors in floating point computation. Developers of games or multimedia can design their own features by programming the graphics processor. This feature also catches the eye of developers of software other than games or graphics. Non-graphics applications using the programming power of graphics processors are called *General-Purpose Computations on Graphics Processor Units* (GPGPU).

This study proposes a novel approach and architecture to speed up pattern matching by using the GPUs. GPU is also capable of processing network traffic of multiple sessions in parallel. The contributions of this study can be summarized as follows:

- **Generic:** The proposed architecture is generic, and can be integrated with other systems accelerating pattern matching, such as network security system or content-intuitive systems.
- **Economics**: The GPUs are commodity and cost-effective. For example, the solution using NVIDIA GeForce 6800GT (NVIDIA GeForce 6800, 2005) costs 1/10 of other silicon solutions with the same performance.
- **Effective Utilization**: In general, the graphics processing sub-system is often idle in a PC. The computation power of GPU is not always fully utilized even when running games and other GPU-consuming applications. Hence, using a GPU to reduce the system load when performing pattern matching computations, such as virus scans or intrusion prevention, or using a GPU as a co-processor, could improve the performance of systems or applications.

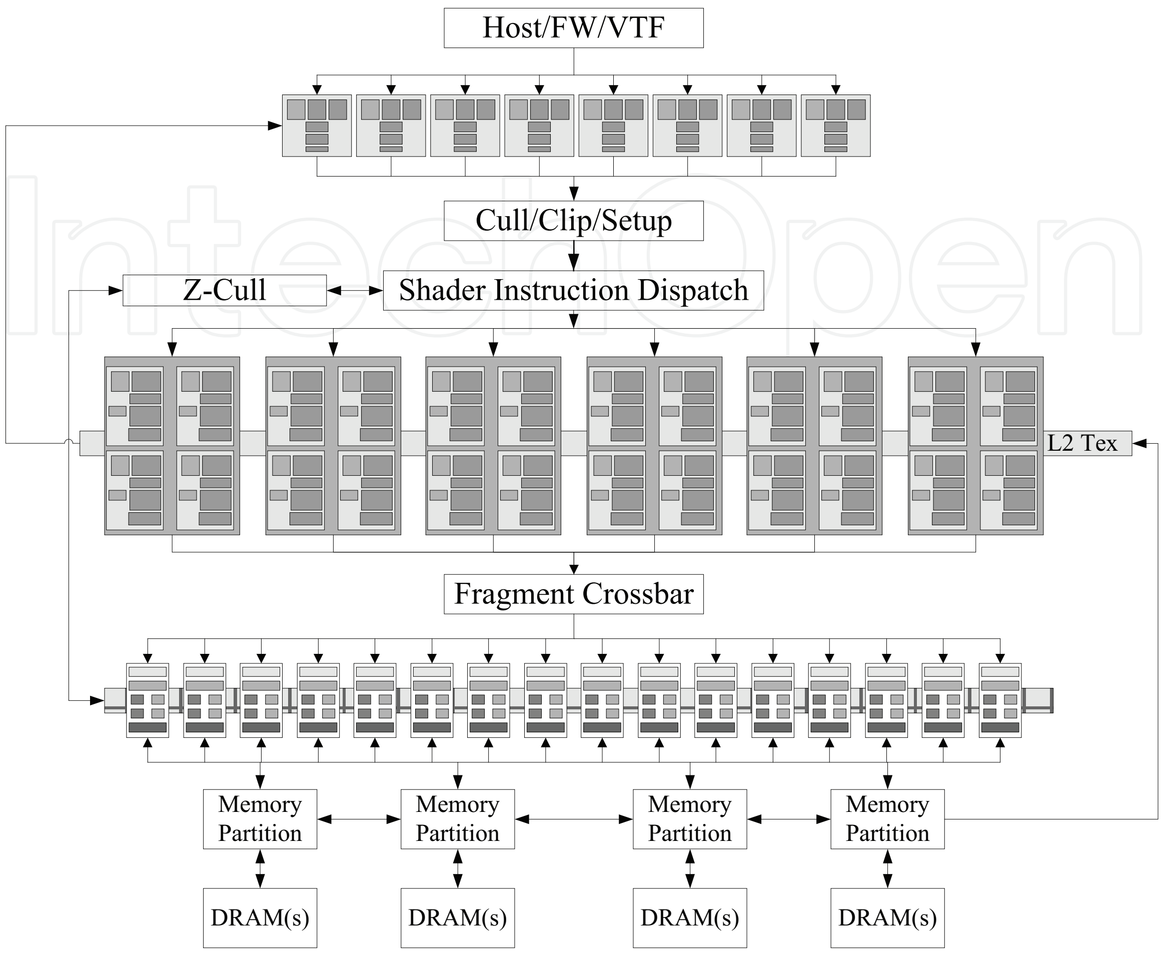

Fig. 1. NVIDIA GeForce 7900 GTX architecture

- **High Performance**: This study demonstrates that the proposed concept and mechanism work well. Experimental results indicate that the performance of the proposed mechanism (programming the GPUs by OpenGL Application Program Interface (Wright & Sweet, 2000)) is almost thrice that of a system using only a general processor (CPU). Moreover, considering a system environment designed for GPUs, the proposed system could reach 6.4Gbps throughput, and costs under $400 overall.

The rest of this chapter is organized as follows. Section 2 provides related research concerning pattern matching and GPU architecture. The architecture and operation of the proposed GPU-based pattern matching mechanism are presented in Section 3. Section 4 focuses on the performance evaluation, and provides a complete performance analysis of the proposed system. Finally, conclusions and future work are given in Section 5.

2. Related works

The proposed scheme integrates the pattern matching algorithms and the computing power of commodity GPUs. The AC algorithm is a simple, but efficient string matching algorithm to locate all occurrences of a finite set of keywords in a text string. It constructs a finite state automaton first by preprocessing a set of keywords and then applies that automaton to the input string. AC algorithm has best performance in worst case. In the following section, the GPU hardware architecture is described in detail since the rendering pipeline in GPUs is the key function in the proposed scheme. Works using GPUs for non-graphics applications are introduced in the end of this section.

2.1 GPU architecture

This section introduces the architecture and features of current GPUs. Due to market demand and the growth in semiconductor technology, the two major GPU suppliers, namely ATI (ATI Tech.) and NVIDIA (NVIDIA Corp.), are continuously improving the computing power of GPUs. The two currently most powerful GPUs, ATI X1900 and NVIDIA GeForce 7900 GTX, have hardware optimized for floating-point arithmetic. GPUs perform better than CPUs in many data parallel computations because of the strongly parallel nature of GPUs.

Fig. 1 depicts the architecture of NVIDIA's last flagship GPU, the GeForce7900 GTX, which has eight vertex shaders, 24 fragment shaders, and a core clock of 650 MHz. It uses GDDR3 memory with a bandwidth of 51.2 Gigabytes per second and a work clock of 800 MHz. Unlike CPUs, today's GPUs have higher off-chip memory clock rates than core clock rates. This feature can prevent the memory stall caused by heavy demand for memory access in shaders. For this study, the most important components in GPUs are the programmable vertex shaders and fragment shaders. A GPU program usually performs better with more vertex and fragment shaders.

In computer graphics, the procedure that transforms the vertices, colors, and coordinates information into the 2D or 3D images on the screen is called rendering pipeline (Fig. 2). The pipeline has three major portions. The vertex shaders are at the front, followed by the rasterizer, with the fragment shaders at the back of the pipeline. The input of vertex shaders comprises geometric information, including vertices and colors. The coordinates of vertices are transformed to the positions rendered on screen according to the default or user-defined coordinate matrix. The vertex shaders then perform the lighting computation for each vertex, and determine their colors. The rasterizer, which is a non-programmable fixed function in GPUs, produces every triangle of a polygon based on the processed vertices and the connectivity between them (Triangle Setup), and colors every triangle linearly (Digital Differential Analyzer). The fragment shaders process every pixel outputted by the rasterizer, and generate real pixels on the screen. Those pixels that have not been processed by fragment shaders are also called potential pixels. The first and most important job of the fragment shaders is texture mapping, which map textures polygon faces. The fragment shaders then perform alpha, stencil, and depth test to determine whether to render or discard pixels. Finally, the GPU blends test results with the pixels that have been rendered to target. The GPU writes results to the rendering target and draw them on screen at the end of the rendering pipeline. The rendering target is generally a frame buffer or texture memory. A rendering pass is the procedure in which a collection of data passes through the rendering pipeline and outputs to the rendering target.

Since the float-point computing power of GPUs grows much faster than the off-chip memory bandwidth (Pharr & Fernando, 2005) and the on-chip cache is small, accessing data from off-chip memory is rather expensive. Hence, GPUs require high arithmetic intensity.

The number of logic operations must be maximized, whereas the amount of communications between GPUs and memory need to be minimized.

To build high performance GPU program, besides avoiding heavily access to off-chip memory, the data structure needs to be designed carefully to exploit the cache. The CPU was originally designed for general-purpose applications, and must function in different conditions, so has a higher control capacity than data-path capacity, making the CPU appropriate for sequential tasks. Conversely, a GPU is special-purpose hardware designed for computer graphics and has less control logic hardware than a CPU. However, GPUs are optimized for parallel computing. Algorithms should be designed to consider the parallel nature of a modern GPU to optimize a program's performance.

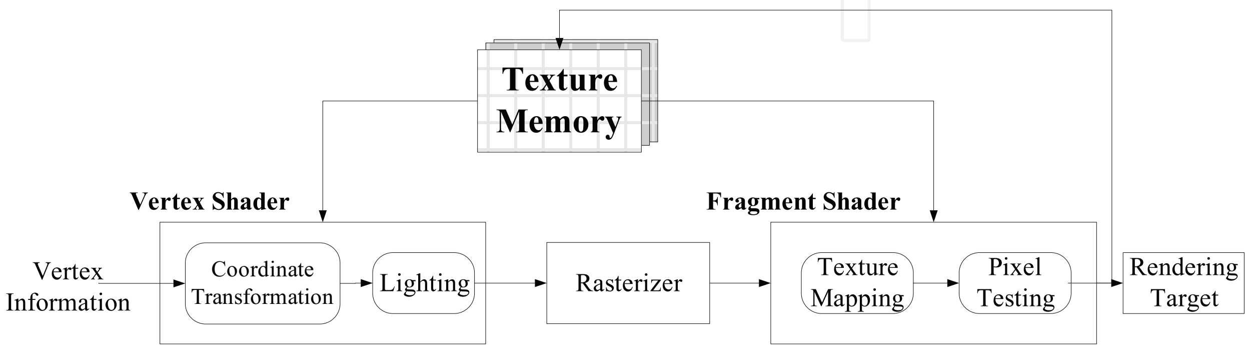

Fig. 2. Typical 3D pipeline

2.2 GPGPU

GPGPU signifies General-Purpose computation on GPUs (GPGPU, Online). As discussed in the previous section, commodity graphics processing units are becoming increasingly powerful. Researchers have developed various algorithms and systems based on GPUs to improve the performance of CPU-oriented programs (Trancoso & Charalambous, 2005). Cook (Cook et al., 2005) is the first to apply GPU to cryptography. They demonstrated the feasibility of utilizing GPUs for cryptographic processing to offload symmetric key encryption from CPU, and proved that block cipher algorithms are unsuitable for application with GPUs. Moreover, various GPU-based pattern-matching applications are popular (Vasiliadis et al., 2008) (Smith, 2009) (Goyal, 2008).

A number of works have been studied to process the stream data in a SIMD fashion. For instance, Lai applied Imagine Processor to Bloom Filter (Kapasi et al., 2002). However, these are neither implemented on a commodity GPU, nor analyzed from the viewpoint of computer graphics.

Recently, Advanced Micro Devices (AMD) and ATI Technologies, which jointed together on October 25, 2006, has released the first GPGPU product named AMD stream processor (AMD Stream Processor, Online) and announced at the Supercomputing 2006 (Supercomputing06, 2006) trade show. The AMD stream processor makes use of AMD's new technology called *CLOSE TO METAL* (CTM) to provide users a more powerful interface to develop applications based on GPUs. The parallel processing power of it is used for a wide range of applications, such as financial analysis, seismic migration analysis, and life sciences research, providing organizations and researchers now have the ability to process incredible amounts of information in significantly less time.

3. Packet inspection scheme using GPUs

The proposed procedure is a parallel byte streams pattern matching scheme which is suitable for any automaton-based string matching algorithm. AC is a well known representative of automaton-based algorithm. The AC algorithm is a sequential task model, in which one task can process only one byte stream. This study combines several sequential AC tasks to form one complex task that can process multiple byte streams, each is independent of the others. With the task parallelism of GPU, the fragment shaders can match multiple byte streams with patterns simultaneously, enabling independent pattern matching tasks to be executed at the same time. When several fragment shaders want to access data in the GPU memory, GPU can provide simultaneous memory access for decreasing the latency of data access and achieve data parallelism because of its multiple memory controllers. In theory, increasing the number of fragment shaders running in GPU raises the number of byte streams that are processed in parallel, thus improving performance. Therefore, the GPU must be provided as many byte streams as possible in order to prevent any fragment shader in the GPU from being idle.
The proposed approach can be applied in host-based IDS since GPU is almost available in every PC nowadays. It's particularly suitable for servers, which are very common to serve many connections simultaneously. The proposed scheme is expected to perform better in the network environment if the number of concurrent sessions exceeds some threshold. The later performance analysis uses the Defcon9 (Defcon 9, 2001) as network packet for input byte streams, and uses Snort's patterns as our patterns (Roesch, 1999). With the combination of the CPU and the GPU (GPU is the CPU's co-processor), the feasibility of the proposed approach is analyzed. Using real network traffic and real IDS patterns in our experiment also demonstrates that the proposed approach is very suitable for working in a high-speed network environment with multiple connection sessions.
The proposed approach is divided into three parts, namely *Data Flow*, *Data Structure*, and *Control Flow*. *Data Flow* transforms the original finite state automaton constructed with predefined patterns into another form that can be run in the GPU. That is, *Data Flow* must modify the layout of data in the CPU to fit that in the GPU. The major task of *Data Flow* is to execute the state transitions of multiple byte streams in GPUs. *Data Structure* is responsible for changing the data format between the CPU and GPU. Notably, casting operations may be necessary during processing due to the different data formats in GPU. Since casting operations decrease the throughput, an appropriate data structure must be chosen. *Control Flow* is responsible for the communication setup between the CPU and GPU. Additionally, *Control Flow* administers the overall operations of the proposed approach, and performs the program flow. The *Control Flow* constitutes the framework bottleneck, which is discussed later.

3.1 Model framework

Three data structures have to be maintained in the GPU texture memory. *Automata texture* is used to store the finite state automaton; *Text texture* is used to store multiple input streams, and *State texture* is employed to store the current states of input streams in the finite state automaton (Fig. 3). The dimensions of the *Text texture* and *State texture* are determined from the number of input streams in the system. For instance, if the number of input streams is 256, then the dimensions of *Text texture* and *State texture* can be configured as 16×16. The dimensions of rendering targets are set to 16×16, and the rasterizer in the GPU maps one pixel in the textures to one fragment. Therefore, the fragment shaders can process one single pixel in the textures (called a *texel*) at once. If one GPU has 16 fragment shaders, and 256 pixels need to be processed, then each fragment shader should handle 16 pixels within one rendering pass.

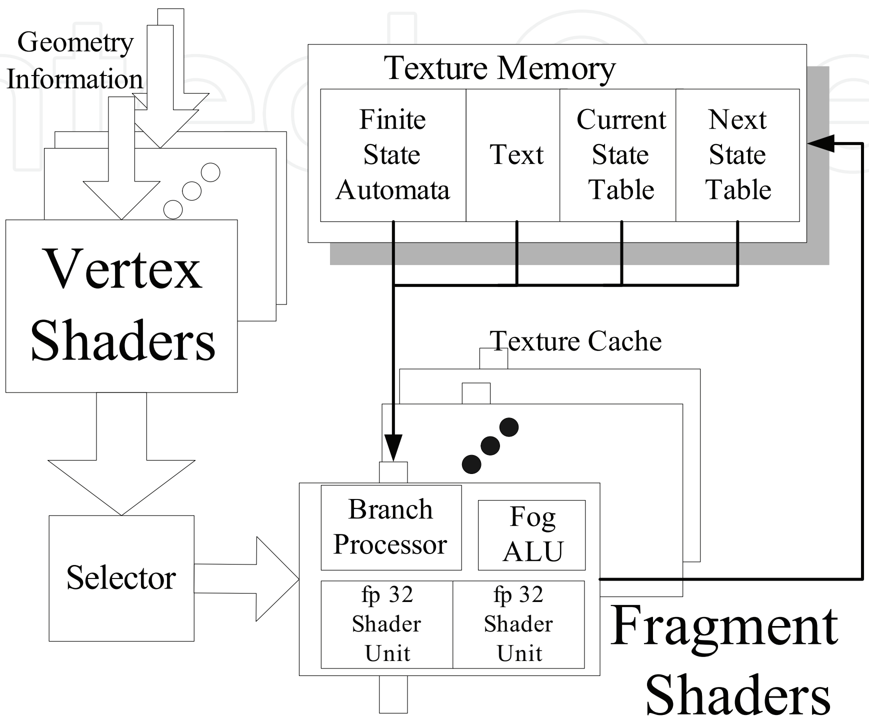

Fig. 3. The block diagram of pattern matching engine

Fig. 3 shows the flowchart of the proposed approach, which is described as follows:
Step 1. Construct the pre-defined patterns into finite state automata. Through some preprocessing like basic data format and address translation, the finite state automata are transformed into *Automata texture*, which are then downloaded from system memory to GPU texture memory via a high speed memory bus. The *Automata texture* cannot be modified once the operation is finished. If new patterns are to be added into the finite state automaton, then one new finite state automaton must be re-constructed.
Step 2. Define and allocate *State texture* and *Text texture*. The texture memory cannot be read and written in the meanwhile. Therefore, two state textures, *Current State texture* and *Next State texture*, are defined to read the current state and write the next state.
Step 3. Program the fragment shaders in the GPU by shading language (Rost, 2009). All required data are stored in the texture memory. The geometry information in Fig. 3 is utilized only to map *Text* and *Current State textures* to the rendering target. All fragment shaders execute the following instructions:

1. Obtain an input symbol from *Text texture*,
2. Obtain the current state from *Current State texture*, and
3. Apply the input symbol and current state as one index to obtain the next state from *Automata texture*.
4. Write the next state into *Next State texture* for use in the next rendering pass.

Step 4. Read in and transform the streams into *Text texture*, then download *Text texture* from system memory to texture memory in the GPU. The fragment shaders process every texel in the *Text texture* based on the pre-defined shader instructions in Step 3. Modern commodity GPUs always have multiple fragments shaders. This study assumes that the performance of the proposed approach improves with increasing number of fragment shaders. The number of concurrent input streams (denoted as S) does not have to be the same as the number of fragment shaders (denoted as FS). However, if $S < FS$, then some fragment shaders remain idle. If $S >= FS$, then the hardware driver automatically dispatches input streams to the idle fragment shaders. The fragment shaders return the matching results to the system memory from the texture memory after each rendering pass.

3.2 Data flow

Programming on a GPU is very different from programming on a CPU. It is not trivial to write a general-purpose application on GPUs due to few supported libraries and limited usage of branch/logic operations. Additionally, it is not convenient to access memory like programming in C language. Besides temporary variables, the memory allowed to read and write is texture memory. However, texture memory cannot be read and written arbitrarily. Vertex shaders and fragment shaders can read from texture memory, but cannot write to it directly. The results must be written into texture memory at the end of the rendering pipeline.

The pre-constructed finite state automaton is converted into the format suitable for GPU memory, through which fragment shaders can access data. The proposed approach utilizes pixels in *Automata texture* to represent each state in the finite state automaton (one pixel can carry 1~4 states, as discussed in the next section). GPUs operate according to the stream-kernel principle, in which a stream is a data collection of the same data type and needing similar operations, and a kernel comprises functions that operate on every element in the stream. Streams can provide the GPU parallel data, and each element in different streams is independent of the kernel. In *Data Flow*, taking the input symbols that the CPU transfers to the GPU steadily as streams and the multiple fragment shaders in the GPU as the computation kernel. Each input symbol processed by different fragment shaders is independent. *Data Flow* is described as follows.

Construction Phase

In the *Construction Phase*, the pre-constructed finite state automaton is represented by a deterministic finite automaton (DFA) table. Assuming that the size of symbol space is n, each state in the finite state automaton will be expanded to n next states. For example, for the ASCII-based 8-bit codes, the symbol space n = 256 (0x00-0xff). If the finite state automaton contains k states, then one DFA table with dimensions $n \times k$ must be maintained in memory. Considering the patterns in the Snort project as example, the compiled finite state automaton contains about 22000 states. Hence, one *Automata texture* with $n \times 22000$ pixels should be maintained. Since the dimension of a texture is limited to 4096×4096, the finite state automaton may need to be transformed, in case its dimensions exceed the limit of the fitting layout (the actual layout in GPU memory is shown later). Of course, there is no need to maintain one big 4096×4096 texture for every finite state automaton. The dimension can be adjusted according to the size of the finite state automaton. For example, one finite state automaton contains 1024 states, then its DFA needs to be mapped to one texture with

$1024 \times n$ pixels, so one texture (square) with dimensions $(1024 \times n)^{1/2} \times (1024 \times n)^{1/2}$ is adequate. The memory structure in the GPU memory is two-dimensional, but that in the CPU is one-dimensional. The following procedure is used for translation between the two different dimensions.

```
Procedure DataAddressing ($T$, $H$, $W$, $A$, $K$, $S$)
Input: Deterministic Finite Automaton table: $T$, the height of DFA table: $H$, the width of DFA table: $W$, the dimension of element space: $A$, the amount of states: $K$, and the set of states: $S$
Output: One two-dimensional texture in GPU memory: $GT$, transformed from the one-dimensional DFA table.
Load DFA table $T$;
Initialize: $t \leftarrow \emptyset$, $z \leftarrow \emptyset$, $offset \leftarrow \emptyset$;
For each $S[t]$ do
        $r \leftarrow$ the remainder of $t$ dividing by $H$;
        $p \leftarrow$ the quotient of $W$ dividing by $A$;
        $q \leftarrow$ the quotient of $t$ dividing by $W$;
        offset $\leftarrow$ multiply($A$, sum(multiply($r$, $p$), $q$));
        While $z < A$ do
                If $T[t, z] \neq$ NULL then
                        Load data from $T$ at index ($t$, $z$) to $GT$ at index (sum($offset$, $z$), $z$);
                Else
                        Continue;
                Increase $z$ by 1;
        End
End
Return;
```

Search Phase

The *Search Phase* utilizes the programmable fragment shaders in GPUs. Numerous shading languages like GLSL, HLSL, Cg, Brook (Buck et al., 2004), and assembly language can be used to accomplish such purpose. This study applies GLSL and OpenGL (Wright & Sweet, 2000) to program fragment shaders. Fragment shaders follow the SIMD programming model, which implies the same instructions are executed simultaneously on different data. The following search procedure is invoked by fragment shaders to perform state transitions. The input symbol is first obtained from the *Text texture* in GPU memory. The current state information is then obtained from *Current State texture* in the same way. The next state can be fetched by taking the input symbol and current state as an index of the *Automata texture. Render-to-texture*, which rendered computation results to texture rather than frame buffer, is then adopted to write the next states into the *Next State texture* for the next rendering pass. Additionally, the next states are transferred to the system memory for post-processing of pattern matching by CPUs. Although three memory lookup operations are executed in the above procedure, the speed of off-chip memory access inside GPUs is very fast, even up to 51 GB/s. The latency of these three memory lookups is assumed to be low.

```
Procedure Search (AT, IT, CT, H, W, A, P);
Input: The DFA texture: AT, the input symbol texture: IT, the current state texture: CT,
the height of the rendering target: H, the width of the rendering: W, and the dimension of
element space: A, the corresponding position in rendering target: P
Output: The next state information in DFA: NS;
Initialize: s ← ∅, c ← ∅, x ← ∅, y ← ∅;
Fetch an input symbol from IT at P, and store in s;
Fetch current state from CT at P, and store in c;
Round up and down s for accuracy;
If (s = NULL) then
        Return;
Else
        r ← the remainder of c dividing by H;
        r'← the quotient of c dividing by H;
        x ← sum(multiply(r, A), s);
        y ← r';
        NS ← Lookup(AT, x, y); /* Lookup next state from AT */
        If (NS is an accepted state) then
                Set accepted flag;
                Return True;
        Else
                Return True;
Return False;
```

3.3 Data structure

This section introduces how to represent the DFA table in GPU, and discusses memory optimization issues. The DFA table and other data needed are stored in the GPU texture memory. Accordingly, all data in system memory must be transformed into the texture data format, namely pixels. Since data are stored in pixels, the chosen pixel format significantly affects the size of GPU memory required. Additionally, the processing time of the *Search Phase* varies according to the pixel format. Common OpenGL pixel formats include color index, grayscale, RGB, and RGBA. RGBA and grayscale pixel format are considered here as examples.

RGBA Pixel Format

Assuming that each pixel contains four 32-bit components: Red(R), Green(G), Blue(B), and Alpha(A). The state information of a finite state automaton is stored as R values in pixels (Fig. 4), so that one state is represented as one pixel. Although modern GPUs support 16-bit floating-point format, its precision is far lower than that of 32–bit floating-point format. For instance, if the NVIDIA fp16 float is used, the element 3079 cannot be addressed, since the closest representable numbers of fp16 float are 3078 and 3080. This inaccuracy seriously affects computations. The fp16 float is not used in this study. Supposing that the space of input symbols ranges between 0x00 and 0xff, the Snort patterns have over than 22,000 states, each then can be expanded to 256 next states. Therefore, about 5 millions pixels are needed to store the entire DFA table. The limit of the texture dimension in GPUs is 4096×4096, meaning that one single texture contains at most 2^{24} pixels (16 millions). Therefore, storing entire DFA table of the Snort rules into one big texture has no problem. Additionally, modern GPU technology performs all computations with floating-point arithmetic. Floating-point related operations in modern GPU hardware are improving. Therefore, floating-point arithmetic gives the best performance for finite state automaton data format and related computations.

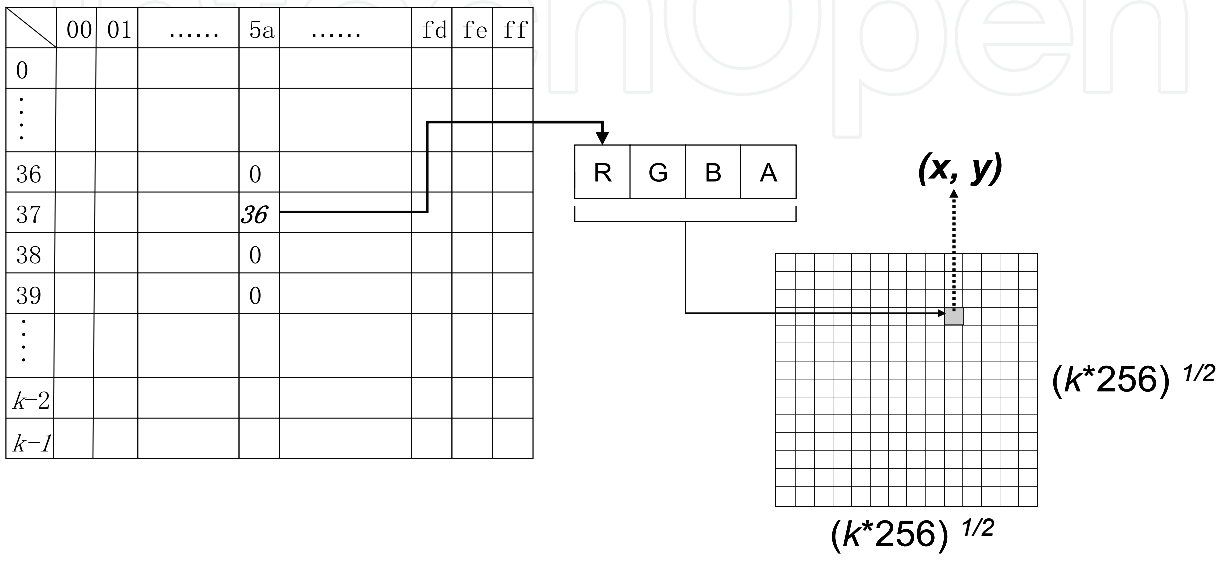

Fig. 4. Illustration of data packing

In Fig.4, the left side is a DFA table and the right side is a texture (square) with dimensions $(k \times 256)^{1/2} \times (k \times 256)^{1/2}$. The space of input symbols ranges between 0x00 and 0xff. The number of states is given as k. The dotted line indicates the mapping from state 37 to the R component of a pixel while reading in input symbol "5a". The coordinate (x, y) is derived from the mentioned Search Procedure.
In the above example, the state information of DFA table is stored in the R component of pixels, while the other three components, G, B, and A, are wasted. To fully utilize the components of pixels, the original patterns can be split into four groups, such that the number of states in each group are almost equal. The same component in different pixels comprises one finite state automaton. That is, the four finite state automata are traversed by fragment shaders concurrently. In this way, the four finite state automata, each with at most 65536 states, can theoretically be packed into one single big texture. Naturally, fragment shaders must execute extra instructions to perform all pattern matching operations. The processing time would be slower than in the case that only R component is used.

Grayscale Pixel Format

The grayscale texture can be adopted to represent one DFA table rather than applying all components, namely R, G, B, and A. Pixels in a grayscale texture have only a grayscale component. By assigning each grayscale component one 16-bit unsigned integer, 2^{16} states can be expressed at most. However, the grayscale value is clamped as a float-point number between 0.0 and 1.0 in OpenGL. Hence, extra computations are required to restore the state information to the original integer representation.

3.4 Control flow

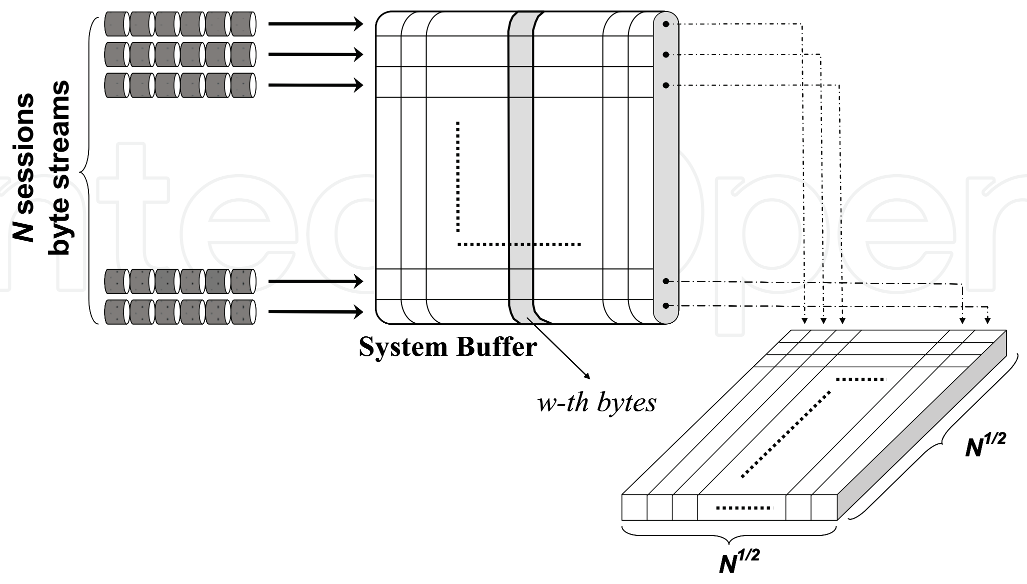

Fig. 5. Illustration for packing *N* byte-streams into the Text texture ($N^{1/2} \times N^{1/2}$)

In *Control Flow*, OpenGL is used to control the flow of the proposed approach. The implementation follows the general computer graphics programming model, except that the data are not intended to be displayed on the screen. The focus is the communications between the CPU and GPU. Beyond that, the *Control Flow* is also responsible for pre-processing of byte streams in CPU and post-processing of pattern matching. For illustration, considering the input phase, as shown in Fig. 5, *N* input byte streams need to be processed. First, all input streams are read into a buffer, and the *w*-th byte of each input stream is copied and placed into the *Text texture* in row-major order before starting the *w*-th rendering pass. This texture is then transferred to the GPU memory. The rendering pipeline can then be initialized and started. CPU has to trigger these operations every time before starting the rendering pipeline. As for the output phase, the results, which are rendered at the end of the rendering pipeline, are then transferred from GPU memory to system memory. The results are the accepted states in finite state automata. CPUs need to do computations with the accepted states for getting the corresponding matched patterns. We can use matched patterns or other related information as the matching results, and *Automata texture* is the only structure we have to modify.

4. Evaluation and analysis

The previous section demonstrated how finite state automata on GPUs function. The fingerprint of finite state automata is shown in the following section. Next, an attempt is made to demonstrate the performance of the proposed data flow with commodity GPUs by using the performance profiling tool released by NVIDIA for measuring unbiased throughput. Various pixel formats are also analyzed and compared with respect to the performance of the fragment shaders. *Data Flow* is then integrated with *Control Flow* to compare performance of the proposed approach with that of other AC algorithm-based implementations.

4.1 Memory fingerprint

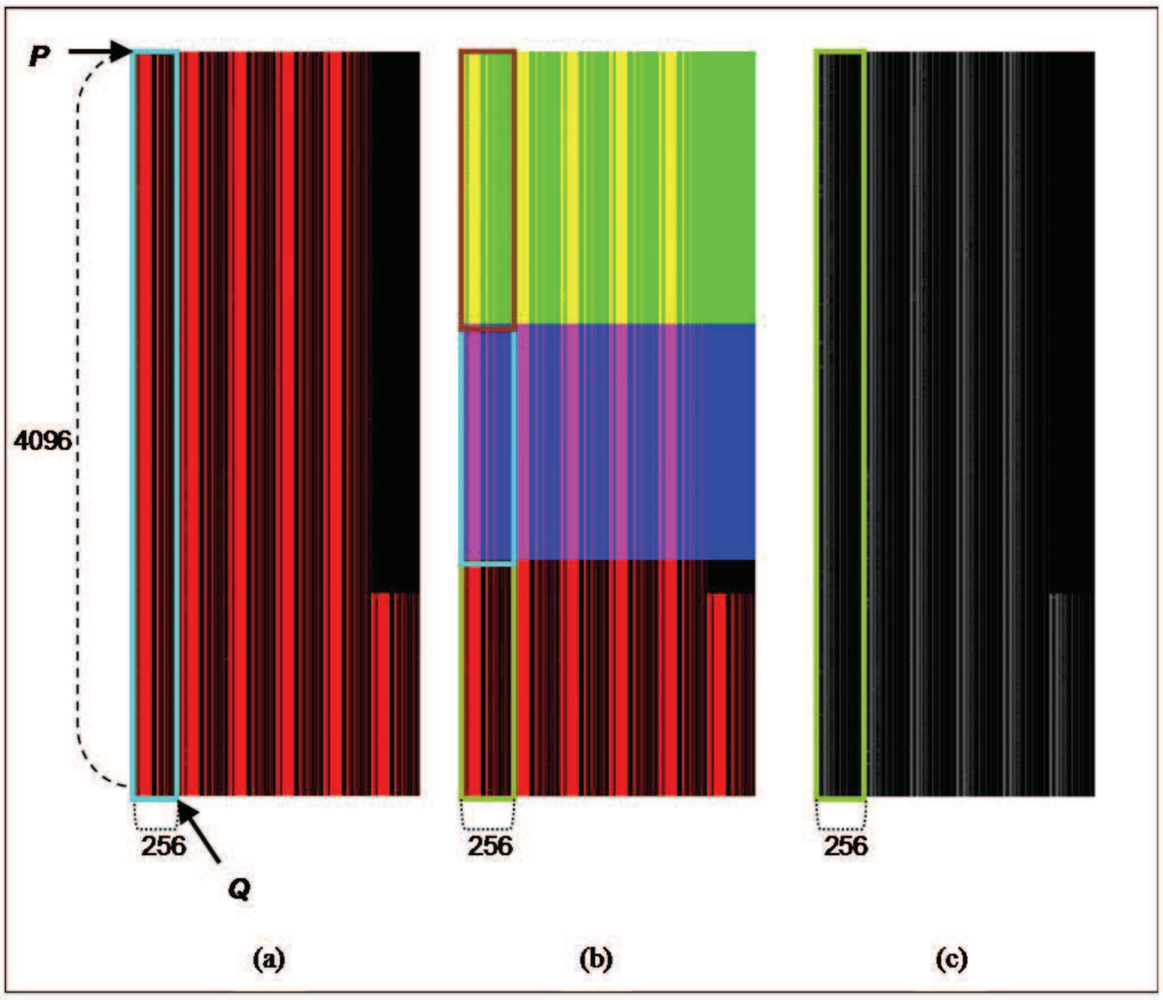

Fig. 6. The fingerprint of finite state automata

The images in Fig. 6 show the layouts of finite state automata (for Snort patterns), which are located in GPU memory. The lower left corner of these images is the origin pixel, and it represents the next state which current state is 0 and input symbol is 0. From the origin to the point *P* are the pixels with current states ranged from 0 to 4095 in order and input symbol 0. Similarly, from the origin to the point *Q* are the pixels with input symbols from 0 to 255 and current state 0.This rectangle with a width of 256 pixels and a height of 4096 pixels could contain all next states with current state from 0 to 4095 (Fig. 6(a)). In the same way, we could divide Fig. 6(a) into six rectangles, and it can carry $4096 \times 6 \times 256$ next states. Fig. 6(a) means the layout of finite state automaton located GPU memory which only utilizes *R* value of one pixel. Pixels with non-zero next state are red while pixels with zero next state are black.

As for the case that all *R*, *G*, *B*, and *A* components are used, pixels are black if four next states are all zero (Fig. 6(b)). When only the *R* component is utilized, the pixels are red. Moreover, when only the *G* component is utilized, the pixels are green. Similarly, the pixels are blue only if *B* component is utilized. Furthermore, the pixels are yellow if both *R* and *G* components are employed. Fig. 6(c) represents the layout in which grayscale pixel format is applied.

GPU / Pixel Format		Memory Size (Mbytes)		GeForce 6200 (NV44)	GeForce 6600 GT (NV43-GT)	GeForce 6800 Ultra (NV40)	GeForce 7800 GT (G70)
		64K states	Snort 2.4 (Apr.06)	4 fragment shaders	8 fragment shaders	16 fragment shaders	24 fragment shaders
Strategy	1	256	84.36	933.36	2666.64	4266.64	6400.00
	2	64	21.09	622.24	1777.76	2844.48	4266.64
	3	32	10.54	700.00	2000.00	3200.00	5485.68

Unit: Mbps

Table 1. Memory required and the throughput with different GPUs

4.2 Performance analysis

NVShaderPerf (NVIDIA ShaderPerf 2, 2008) is a command line utility to report shader performance metrics. It can produce the scheduling information based on instructions executed in shaders. In order to evaluate the proposed data flow, the pixel processing throughput is treated as the performance metric. In this section, the instructions executed in pixel shaders are different according to the pixel formats adopted in the three strategies. Although the purposes of these strategies are the same, the test result of these strategies varies.

To compare with other related works impartially, the *Control Flow* was separated from this experiment. This is because *Control Flow* involves interaction between the operating system platform and device driver, which is not the focus of the proposed approach. The details are provided in the next section.

Strategy 1 in TABLE 1 applies only the *R* component of pixels to represent the state information. Although the other three components, *G*, *B*, and *A*, are not used in Strategy 1, the operation is succinct. Strategy 1 obtains the optimum throughput among all. Assuming the space of input symbols ranges between 0x00 and 0xff, Strategy 1's deterministic finite state automaton has a maximum of 2^{24} next states since the texture dimension is limited to 4096 ×4096. Each component of *RGBA* is 32-bit, such that Strategy 1 would consume 256MB GPU memory (4Bytes × 4 × 4096 × 4096) if the texture dimension is set to 4096 × 4096. On the contrary, Strategy 2 applies all components to represent states of four finite state automata. The GPU pixel utilization of Strategy 2 is 4 times efficient of Strategy 1; i.e., Strategy 2 requires only 1/4 of the memory of Strategy 1. However, the amount of memory access in Strategy 2 is also 4 times Strategy 1. Strategy 3 adopts the grayscale component of pixels, and every component is 16-bit. The DFA with 2^{16} states requires 32MB of GPU memory in Strategy 3. All computations inside the GPU are based on 32-bit floating-point numbers. If 16-bit grayscale components are utilized for finite state automata, then shaders must perform extra computations for accuracy. These computations have significant overheads. Hence, Strategy 1 had the best throughput, and Strategy 3 was slightly worse than Strategy 1. The proposed approach is flexible, since it seeks a tradeoff between the number of patterns and throughput. Similar approaches are common in commercial Graphics Processor products.

Fig. 7 shows the performance comparison between the proposed approach and other famous related proposals. The algorithm proposed by Tuck (Tuck et al., 2004) applies an accessible embedded memory of 1024 bits, which is implemented in on-chip. The design cost can be assumed far more than the proposed approach using GPUs located in PCs originally. Even though Tuck's ASIC design performs better than the proposed system, the superiority of GPUs can be discovered. TABLE I also shows that the number of fragment shaders raises with each new generation of GPUs. Therefore, the proposed approach with the new GPU architecture performs better than that with older GPU architecture. The GF7800-1 approach in Fig. 7 even outperforms other implementations which used specific hardware (Cho et al., 2002), (Tuck et al., 2004), (Bos & Huang, 2005), (Song et al., 2005).

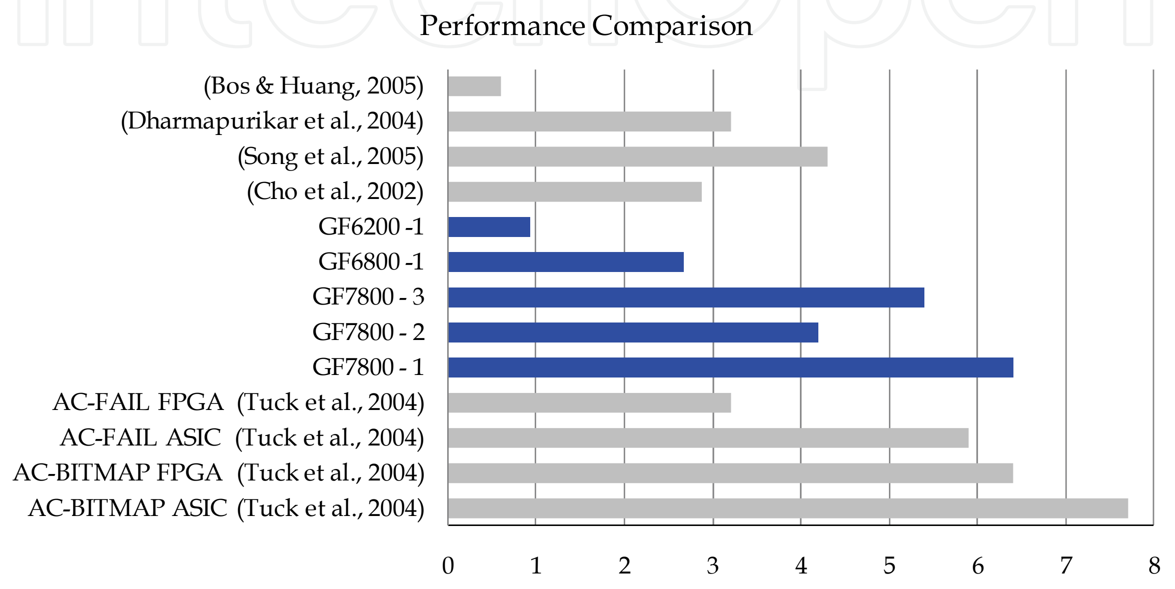

Fig. 7. The performance comparison between the proposed approach and other famous related proposal

4.3 System overhead

Fig. 7 indicates that the proposed solution is potentially as good as other FPGA or ASIC solutions. However, our proposed method is software-based since the GPU can be programmed through shader languages and almost every PC has GPU installed in it. It is considerable that the performance of a software-based solution is almost as good as that of hardware-based solutions. In this section, the proposed approach is integrated with commodity graphics cards in home PCs and designed to cooperate with other software in the operating system. The experimental environment was configured as follows:

Processor: AMD Sempron 2500+
Operating system: Windows XP Service Pack 2
System main memory: 512 DDR memory
Graphics card: NVIDIA GeForce 7600 GT with 12 fragment shaders
Graphics API: OpenGL

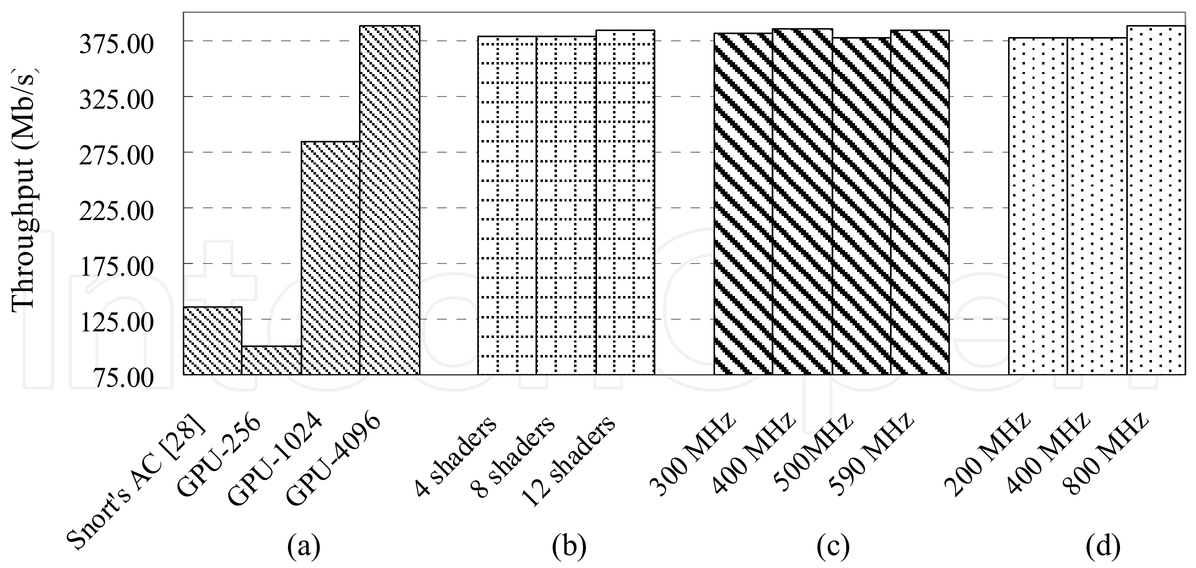

Fig. 8. (a) Performance of AC algorithm and the proposed approach. This experiment compared the performance of AC algorithm and the proposed approach with different number of sessions. (Number of sessions: GPU-256: 256, GPU-1024: 1024, GPU-4096: 4096); (b) Performance of the proposed approach with different number of fragment shaders; (c) Performance of the proposed approach at different GPU core clock rates; (d) Performance of the proposed approach at different GPU memory clock rates

The Snort 2.4 patterns were also taken as the keyword patterns, as in other related work. The Defcon9 trace is taken as the testing data for pattern matching. As shown in Fig. 8(a), the software performance between the proposed approach with various amount of sessions and the Snort's AC implementation (Norton, 2004) is compared. Multiple-session and single-session solutions were compared since traffic from a large network is an aggregation of many sessions. It shows that the proposed approach performs better than AC algorithm when processing 1024 or 4096 sessions, as well as the number of sessions raises. The AC algorithm is a sequential task model which naturally performs worse than a parallel task model. Thus it is always slower than the proposed approach once the benefit of parallel processing is greater than the overhead of using OpenGL API. From another perspective, this feature is advantageous for pattern-matching intuitive software, such as anti-virus software or intrusion detection systems on PCs. For instance, anti-virus software can scan multiple files simultaneously. Therefore, The GPU resources of PCs can be fully exploited.

Fig. 8(b) shows the performance of the proposed approach with various numbers of fragment shaders. The throughput rises with increasing numbers of fragment shaders. However, using 12 fragment shaders did not produce a significant performance improvement, which is not consistent with previous assumptions. The gDEBugger (Graphic Remedy gDEBugger, 2005) profile indicates that the proposed approach is CPU-bound. The number of fragment shaders has minor effect on the entire architecture, since a GPU can finish its job and return results to CPU within 20 microseconds with either 4 or 12 fragment shaders. The experimental results also imply that the proposed approach can reach the peak performance when using a GPU-based system.

Fig. 8(c) demonstrates that the core clock rate has a slight impact on performance. The difference between the maximum and minimum throughput was less than 10 Mbps. The performance was assumed to be better at a higher clock rate, but Fig. 8(c) shows the opposite results. The performance dropped at high clock rates because our implementation determined the throughput using traffic divided by processing time. The difference in processing time with various clock rates was less than 0.5 seconds. The proposed implementation was possibly interrupted or preempted by other operating system (Microsoft Windows) application threads, influencing our testing processing time and producing a non-reasonable result. Fig. 8(d) shows the testing result after adjusting the clock rate of the GPU memory, demonstrating that the performance of the proposed approach is directly proportional to the clock rate of GPU memory.

Fig. 8(b)–(d) show that the proposed approach performs well while the degree of parallelism is above a threshold, and the processing power of GPUs is never the bottleneck of the overall system. The GPU remains idle at most processing time period. The interaction between applications, OpenGL, OS, and device driver slows down the proposed system. Moreover, the data from CPU to GPU in every rendering pass is below 10KB. The proposed approach does not benefit from the high-speed PCI-E bus. Even though the proposed approach can perform better than other implementations in software, it has particularly strong potential when GPU and high-bandwidth bus are fully exploited.

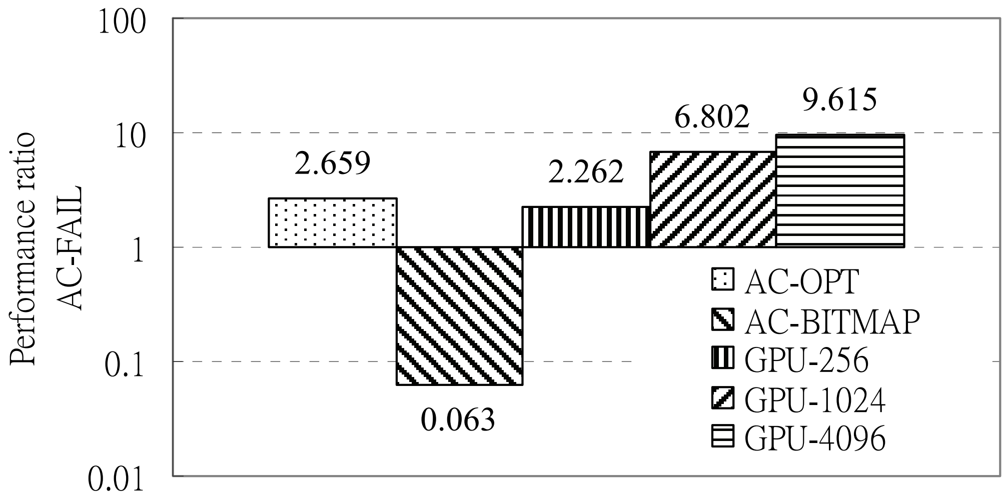

Fig. 9. Performance ratio of various AC implementations to AC-FAIL

Fig. 9 compares the software performance of the proposed approach and Tuck's algorithm (Tuck et al., 2004) with the performance of Tuck's AC-FAIL approach as the baseline. The proposed approach had a performance which is 9.615 times that of AC-FAIL when 4096 sessions were processed simultaneously, and even outperformed the optimized software approach in Tuck's algorithm. Although the GPU was used as the pattern matching accelerator, the proposed approach is still classified as a pure software application, since it utilizes no specific hardware. A graphics card is included in every modern personal computer, and therefore includes no extra cost. The proposed approach utilizes existing system resources.

5. Conclusion

This chapter discusses pattern matching algorithm and research about GPU, and presents a novel scheme that employs the GPU as an accelerator for multi-session deep packet inspection. This study is the first to consider the application of a GPU for this purpose. Several parameters are analyzed, such as the pixel formats employed by finite state automata and the number of shaders. The measurement and analysis show that the proposed scheme is better than other explored approaches, and the idea of pattern matching on the GPU is feasible. High-performing IDS within a host is possible to be made by applying this new concept for network processing, especially useful for servers to provide reliable services for multiple connections concurrently.

There are still many details that could be further improved in the future. First, GPUs are originally designed for graphics processing, so they have very good performance in matrix operations and have great power of floating point computation. For example, it can do multiply-add operations within one instruction. Problems such as how to store data in floating point format compactly or how to take the advantages of some special instructions designed for GPUs, need to be investigated deeply. As mentioned above, the proposed approach is appropriate for most automaton-based works, and the parallel nature of GPUs is particularly fitting for implementing various FSM-based algorithms for speedup (Tan et al., 2006). A suitable pattern matching algorithm may be found by further utilizing these virtues to exploit the power of GPUs. Additionally, GPU applications need high arithmetic intensity for peak performance. Therefore, rather than automaton-based pattern matching algorithms, hash-based algorithms, such as Wu-Manber or Bloom Filter, could also be utilized in GPUs. Second, GPGPU research usually focus on how to program fragment shaders regardless of programming vertex shader since it has limited operations on texture memory before, but the trend seems to change recently. Because some vertex shaders in GPUs can store data into the texture memory now, people try to use vertex shader to increase performance. All GPU-based algorithms might be improved by combining these two powerful processors, for instance, preprocessing in vertex shaders and taking other operations in fragment shaders. Third, as we mentioned in section II, AMD has released a stream processor, which is aimed at the GPGPU applications. With the new thin hardware interface called *CLOSE TO METAL* (CTM) proposed by AMD, programmers now could directly control the graphics kernel instead of using 3D application programming interfaces (APIs) which are originally designed for 3D rendering, such as OpenGL, DirectX, or with the compilation of GLSL, Cg and other shading languages. Therefore, the CTM can improve at least 8x throughput compare to 3D APIs. Thus, the system overhead generated by OpenGL APIs in our proposed method, has great chance to be solved in the near future. Finally, the main focus of future research will be NPGPU, a high-speed network packet processing application based on GPUs, like longest prefix matching, packet classification and network intrusion detection. With the rapid development of GPUs recently, as its programmability and flexibility increased, more friendly GPGPU development platform released, and the parallel nature of GPUs is consistent with the characteristic of simultaneous network connections, we ambitiously expect the diverse network applications based on GPUs will explode.

6. References

Aho, A. & Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, Vol. 18, No 6, (Jun. 1975), pp. 333–340, ISSN-0001-0782.

AMD Stream Processor™. [Online]. Available: *http://ati.amd.com/products/streamprocessor/index.html*

ATI Technologies Inc. , [Online]. Available: *http://www.ati.com/*

Bos, H. & Huang, K. (2005). Towards software-based signature detection for intrusion prevention on the network card, *Proceedings of Eighth International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, pp. 102-123, Seattle, Washington, Sep. 2005.

Boyer, R. S. & Moore, J. S. (1977). A fast string searching algorithm, *Communications of the ACM,* Vol. 20, No. 10, (Oct. 1977), pp. 761–772, ISSN-0001-0782.

Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Fatahalian, K.; Houston, M. & Hanrahan, P. (2004). Brook for GPUs: Stream computing on graphics hardware, *ACM Transactions on Graphics (SIGGRAPH)*, Vol. 23, No. 3, (Aug. 2004), pp. 777–786, ISSN- 0730-0301.

Cho, Y. H.; Navab, S. & Mangione-Smith, W. (2002). Specialized hardware for deep network packet filtering, *Lecture Notes In Computer Science*, Vol. 2438, (Sep. 2002), pp. 452-461, ISBN-3-540-44108-5.

Cook, D. L.; Ioannidis, J.; Keromytis, A. D. & Luck, J. (2005). CryptoGraphics: Secret key cryptography using graphics cards, *Proceedings of the RSA Conference, Cryptographer's Track (CT-RSA 2005)*, pp. 334–350, San Francisco, Feb. 2005.

Defcon 9 (2001). [Online]. Available: *http://www.defcon.org*

Dharmapurikar, S.; Krishnamurthy, P.; Sproull, T. & Lockwood, J. (2004). Deep packet inspection using parallel Bloom filters, *IEEE Micro*, Vol. 24, No. 1, (Aug. 2004), pp. 52–61. ISBN- 0-7695-2012-x.

Goyal, N.; Ormont, J.; Smith, R.; Sankaralingam, K. & Estan, C. (2008). Signature matching in network processing using SIMD/GPU architectures. Technical Report TR1628, University of Wisconsin at Madison, 2008.

GPGPU: General-Purpose Computation on GPUs. [Online]. Available: *http://www.gpgpu.org*

Graphic Remedy gDEBugger (2005). [Online]. Available: *http://www.gremedy.com*

IDC 2006 research report. (2006). Asia/Pacific Semiannual Security Software Tracker. [Online]. Available: *http://www.idc.com/getdoc.jsp?containerId=IDC_P5559*

IDC 2006 research report. (2006). Japan Security Software 2006-2010 Forecast and 2005 Vendor Shares. [Online]. Available:
http://www.gii.co.jp/english/id44743-vend-share.html

Liu, R.T.; Huang, N.F.; Chen, C.H. & Kao, C.N. (2004). A Fast String Matching Algorithm for Network Processor-based Intrusion Detection Systems, *ACM Transactions on Embedded Computer Systems*, Vol. 3, No. 3, (Aug. 2004), pp. 614 – 633, ISSN-1539-9087.

Kapasi, U. & Dally, W. J. et al. (2002) The Imagine stream processor, *Proceedings of* the *IEEE International Conference on Computer Design*, pp. 282–288, ISBN- 0-7695-1700-5, Freiburg, Germany, Sep. 2002, IEEE.

Moscola, J.; Lockwood, J.; Loui, R. P. & Pachos, M. (2003). Implementation of a content-scanning module for an internet firewall, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 31–38, ISBN-0-7695-1979-2, Napa, CA, April 2003, IEEE.

Norton, M. (2004). Optimizing Pattern Matching for Intrusion Detection, Jul. 2004. [Online]. Available: *http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf*

NVIDIA Corporation Inc. , [Online]. Available: *http://www.nvidia.com/*

NVIDIA GeForce 6800, (2005). [Online]. Available: *http://www.nvidia.com/page/geforce_6800.html*

NVIDIA ShaderPerf 2 (2008). NVIDIA developer tools: NVShaderPerf, [Online]. Available: *http://developer.NVIDIA.com/object/nvshaderperf_home.html*

Pharr, M. & Fernando, R. (2005). *GPU Gems 2*, Addison Wesley Publishing Company, ISBN-978-0321335593, New York.

Roesch, M. (1999). Snort: Lightweight intrusion detection for networks, *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, November 1999. [Online]. Available: *http://www.snort.org/*

Rost, R. J. et al. (2009) *OpenGL(R) shading language*, 3rd edition, Addison Wesley Professional, ISBN-978-0321637635.

Smith, R.; Goyal, N.; Ormont, J.; Sankaralingam, K. & Estan, C. (2009). Evaluating GPUs for network packet signature matching. *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, pp. 175–184, ISBN-978-1-4244-4184-6, Boston, MA, April 2009, IEEE.

Song, T.; Zhang, W.; Tang, Z. & Wang, D. (2005). Alphabet based selected character decoding for area efficient pattern matching architecture on FPGAs, *Proceedings of the Second International Conference on Embedded Software and Systems (ICESS'05)*, pp. 276–283, ISBN-0-7695-2512-1, Xian, China, Dec. 2005, IEEE.

Supercomputing06. (2006). The international Conference for High Performance Computing, Networking, Storage and Analysis. [Online] Available: *http://sc06.supercomputing.org/*

Tan,L.; Brotherton, B. & Sherwood, T. (2006). Bit-split string-matching engines for intrusion detection and prevention, *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 3 No. 1, pp. 3-34, ISSN-1544-3566.

Trancoso, P. & Charalambous, M. (2005). Exploring graphics processor performance for general purpose applications, *Proceedings of the 8th Euromicro Conference on Digital System Design (DSD'05)*, pp. 306–313, ISBN-0-7695-2433-8, Porto, Portugal, August 2005, IEEE.

Tuck, N.; Sherwood, T.; Calder, B. & Varghese, G. (2004). Deterministic memory-efficient string matching algorithms for intrusion detection, In *Proceedings of the IEEE Infocom Conference*, pp. 333–340, ISBN-0-7803-8355-9, Hong Kong, March 2004, IEEE.

Vasiliadis, G.; Antonatos, S.; Polychronakis, M.; Markatos, E. P. & Ioannidis, S. (2008). Gnort: High performance network intrusion detection using graphics processors, *Lecture Notes In Computer Science*, Vol. 5230, (Sep. 2008), pp. 116-134, ISBN-978-3-540-87402-7.

Wright, R. S. & Sweet, M. (2007). *OpenGL SuperBible, 4th edition,* Addison Wesley, ISBN-978-0321498823, New York.

Wu, S. & Manber, U. (1994). A fast algorithm for multi-pattern searching, Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.

Yu, F.; Katz, R. H. & Lakshman, T. V. (2004). Gigabit rate packet pattern-matching using TCAM, *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP 2004)*, pp.174–183, ISBN 0-7695-2161-4, Berlin, Germany, Oct. 2004, IEEE.