I. J. Computer Network and Information Security, **2015, 10, 70-77**
Published Online September 2015 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijcnis.2015.10.08

CUDA based Rabin-Karp Pattern Matching for Deep Packet Inspection on a Multicore GPU

Jyotsna Sharma and Maninder Singh
Computer Science & Engineering Department Thapar University Patiala, INDIA
Email: {jyotsana.sharma,msingh}@thapar.edu

***Abstract*—**This paper presents a study of the improvement in efficiency of the Rabin-Karp pattern-matching algorithm based Deep Packet Inspection. NVIDIA GPU is programmed with the NVIDIA's general purpose parallel computing architecture, CUDA, that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. The proposed CUDA based implementation on a multicore GPU outperforms the Intel quadcore processor and runs upto 14 times faster by executing the algorithm in parallel to search for the pattern from the text. The speedup may not sound exorbitant but nonetheless is significant, keeping in view that the experiments have been conducted on real data and not synthetic data and, optimal performance even with the huge increase in traffic was the main expectation, not just an improvement in speed with few test cases.

***Index Terms*—**CUDA, Deep Packet Inspection, Intrusion Detection, GPGPU, Network Forensics, Rabin Karp Pattern Matching

I. Introduction

The Graphics Processing Unit (GPU) which was primarily associated with processing graphics is rapidly evolving towards a more flexible architecture which has encouraged research and development in several computationally demanding non graphic applications where its capability can be utilized for computations that can leverage parallel execution [1]. General-Purpose Computing on GPU (GPGPU), also known as GPU Computing exploits the capabilities of a GPU using APIs such as OpenCL and the Compute Unified Device Architecture (CUDA) [2]. The opportunity is that we can implement any algorithm, not only graphics, but the challenge is to obtain efficiency and high performance in the field where they replace or support the traditional CPU computing.

A good candidate for such GPGPU is Deep Packet Inspection (DPI), the technology, where the appliance has the mechanism to look within the application payload of the traffic by inspecting every byte of every packet, and detect intrusions which are more difficult to detect as compared to the simple network attacks [3]. The computational and storage demand for such inspection and analysis is quite high. An NIDS spends 75% of the overall processing time of each packet in pattern matching [4]. Under high load conditions, high processing abilities are needed to process the captured traffic. A vast majority of earlier research focuses on specialized hardware for the the compute intensive Deep Packet Inspection [5]. Application Specific Integrated Circuits (ASICs) [6], Field Programmable Group Arrays(FPGAs) [7] and network processor units(NPUs) [8] provide for fast discrimination of content within packets while also allowing for data classification. Engaging special hardware translates to higher costs as the data and processing requirements for even medium size networks soon increase exponentially. Some researchers have discussed an interesting entropy based technique for fine-grained traffic analysis [9] [10]. Jeyanthi et al. presented an enhanced approach to this behavior-based detection mechanism, the "Enhanced Entropy" approach, by deploying trust credits to detect and outwit attackers at an early stage. The approaches seem good but call for methods like bi-directional measurements for best performance they which impose additional computing overhead [11].

The task of pattern-matching for DPI can be performed at a much more reasonable cost as GPUs, being a necessary component of most computers these days, are now readily and easily available The task is split into parallel segments resulting in searching the string much faster than a CPU.

There are several pattern-matching algorithms for DPI, Aho-Corasick and Boyer-Moore being the most popular [12][13]. In this paper the Rabin Karp algorithm [30] has been selected from over these and several other popular algorithms because it involves sequential accesses to the memory in order to locate all the appearances of a pattern and is based on a compute-intensive rolling hash calculations. The algorithm has delivered good performance and efficiency when executed on a multicore GPU(in this study, the Nvidia GeForce 635M).

II. Concepts

A. GPGPU

A GPU, capable of running thousands of lightweight threads in parallel, is designed for intensive, highly parallel computation, exactly for graphic rendering purposes. Current-generation GPUs, designed to act as high performance stream-processors derive their performance from parallelism at both the data and

Copyright © 2015 MECS

instruction-level. In GPU's architecture much more transistors are devoted to data processing and less to data caching[14].

Traditionally, GPUs have been designed to perform very specific type of calculations on textures and primitive geometric objects, and it was very difficult to perform non-graphical operations on them. One of the first successful implementation of parallel processing on GPU using non-graphical data was registered in 2003 after the appearance of shaders [15]. Latest generations of graphics cards incorporate new generations of GPUs, which are designed to perform both graphical and non-graphical operations. One of the pioneers in this area is NVIDIA[16] which rules the market with its parallel computing platform and API model, CUDA (Compute Unified Device Architecture) [17], gradually taking over its major competitor, AMD. Others also have their own programmable interfaces for their technologies: the vendor neutral open-source, OpenCL which gives a major competition to CUDA, AMD ATI Stream, HMPP, RapidMind and PGI Accelerator [18].

B. NVIDIA GPUs

NVIDIA's GPU offered the opportunity to utilize the GPU for GPGPU(General Purpose computing on Graphics Processing Unit). In 2001, NVIDIA GeForce 3 exposed the application developer to the internal instruction set of the floating point vertex engine(VS & T/L stage). Later GPUs extended general programmability and floating-point capability to the pixel shader stages and exploited data independence. With the introduction of the GeForce 8 series, GPUs became a more generalized computing device. The unified shader architecture introduced in the series was advanced further in the Tesla microarchitecture backed GeForce 200 series which offered double precision support for use in GPGPU applications. The successor GPU microarchitectures viz. Fermi[19] for the GeForce 400 and GeForce 500 series , then Kepler [20] GeForce 600 and GeForce 700 and subsequently Maxwell [21] GeForce 800 and GeForce 900 series , have dramatically improved performance as well as energy efficiency over the previous ones . Very recently NVIDIA announced major architectural improvements such as unified memory so that the CPU and GPU can both access both main system memory and memory on the graphics card, in its yet to be released Pascal architecture[22]. The new GPU generation also boasts of a feature called NVLink which would allow data between the CPU and GPU to flow at 80 GB per second ,compared to the 16GB per second available presently.

C. CUDA

There are a variety of GPU programming models, the popular ones being the the open source OpenCL[23] , the BSD licensed BrookGPU[24] which is free and CUDA[25] ,the Nvidia's proprietary framework. CUDA(Compute Unified Device Architecture) is supported well by Nvidia hence it has become the most popular of all. It has features like highly optimized data transfers to and from the GPU and efficient management of the GPU shared memory. The parallel throughput architecture of CUDA can be leveraged by the software developers through parallel computing extensions to many popular high level languages , such as ,C, C++, and FORTRAN, CUDA accelerated compute libraries, and compiler directives. Support for Java, Python, Perl, Haskell, .NET, Ruby and other languages is also available.

The CUDA programming model is a C-like language where the CUDA threads execute on a the *device*(GPU) which operates as a coprocessor to the the *host*(CPU). The GPU and CPU program code exist in the same source file with the GPU kernel code indicated by the *_global_* qualifier in the function declaration. The host program launches the sequence of kernels. A kernel is organized as a hierarchy of threads. Threads are grouped into blocks, and blocks are grouped into a grid. Each thread has a unique local index in its block, and each block has a unique index in the grid, which can be used by the kernel to compute array subscripts.

Threads in a single block will be executed on a single multiprocessor, sharing the software data cache, and can synchronize and share data with threads in the same block; a warp will always be a subset of threads from a single block. Threads in different blocks may be assigned to different multiprocessors concurrently, to the same multiprocessor concurrently (using multithreading), or may be assigned to the same or different multiprocessors at different times, depending on how the blocks are scheduled dynamically. There is a physical limit on the size of a thread block for a GPU determined by its compute capability. In this work, for the compute capability 2.1 GPU, it is 1536 threads or 32 warps.

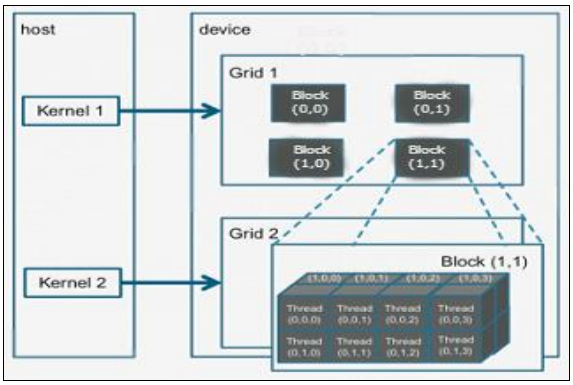

Fig. 1. CUDA Programming Model (Source: www.nvidia.com)

The CUDA programming model assumes that the host and the device maintain their own separate memory spaces. The CUDA program manages the device memory allocation and de-allocation as well as the data transfer between host and device memory[25]. CUDA devices provide access to several memory architectures, such as global memory, constant memory, texture memory, share memory and registers, with their certain performance characteristics and limitations. Fig.2 illustrates the memory architecture of CUDA device, specifically the Nvidia GT635M.

Copyright © 2015 MECS

Example of CUDA processing flow

- Copy data from host memory to device memory
- CPU instructs the process to GPU
- GPU execute parallel code in each core
- Copy the result from device memory to host memory

The appropriate use of the read-only constant and texture memory can improve performance and reduce memory traffic when reads have certain access patterns.

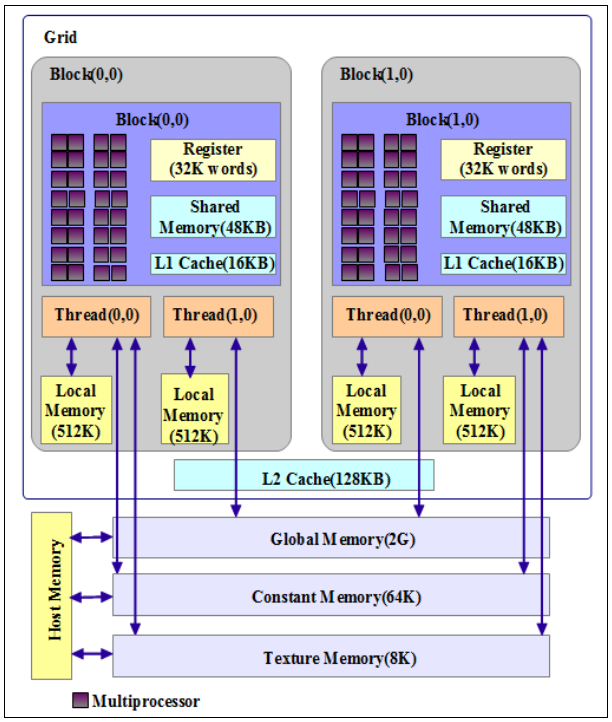

Fig. 2. Memory Architecture of NVIDIA GT635M

III. Pattern Matching Based DPI on GPU

A. Signature Based DPI

Signature-matching or Pattern-matching intrusion detection systems have a major I/O bound performance limitations caused by the overhead of reading packets from the network interface card. Packet reading becomes a bottleneck when the number of packets overwhelms the IDS host's internal packet buffers[38]. The solution to this problem in our approach is to capture the network traffic using an open source packet capturing software. The packet capture file is the input to the CUDA application, The CUDA program *rabinkarpGpu.cu* implements the rabin-karp pattern-matching algorithm on the GPU.

B. Pattern-Matching Algorithms

The literature review of related works suggests that there are two types of algorithms which have been proposed and utilized for pattern-matching, single pattern matching and multiple pattern-matching.

The most efficient algorithm for matching a single pattern against an input was proposed by Boyer and Moore[13]. The Boyer-Moore algorithm is based on skipping heuristics, therefore when the suffix of the *pattern* appears infrequently in the *Text* string, the execution time can be sub-linear. Naive or brute force is the most straightforward algorithm for string matching. It simply attempts to match the pattern in the target at successive positions from left to right by using a window of size m. In case of success in matching an element of the pattern, the next element is tested against the text until a mismatch or a complete match occurs. After each unsuccessful attempt, the window is shifted by exactly one position to the right, and the same procedure is repeated until the end of the text is reached. Knuth-Morris-Pratt [27] is similar to the Naive since it uses a window of size *m* to search for the occurrences of the pattern in the text but after a mismatch occurs it uses a precomputed array to shift several positions to the right.

Multiple-pattern matching scales much better than the single pattern matching. Aho-Corasick(AC) [12],Wu Manber (WM) [28] and AC-BM [29] are the classical multiple-pattern matching algorithms. The AC algorithm has good linear performance, making it suitable for searching a large set of signatures. The AC algorithm and its extensions are ideal for regular expression matching, but they are not optimal for fixed pattern matching like worm scanning because of its large number of states and frequent I/O operations. WM algorithm is a very efficient multi-pattern matching algorithm, which implements multi-pattern matching using bad character block transfer mechanism and search the pattern with the hash function. AC-BM combines the AC and BM algorithms. Instead of using the suffix of patterns as in AC, it uses the prefix. It uses the BM technique of bad character shift and the good prefix shift. We have examined another multiple-pattern matching algorithm, the Rabin-Karp algorithm[30]. The algorithm is as follows :

Algorithm 1: Rabin Karp Algorithm
(Serial Implementation for CPU)

```
 matches = {}
pattern_hash = hash(pattern)
 substring_hash = hash(s[0 : m])

for position from 0 to n - m - 1 do
 update substring_hash to hash(s[position :
      position + m])
 if substring_hash = pattern_hash then
   add position to matches

return matches
```

The key to Rabin-Karp is to incrementally update the hash as the potential match moves along the string to be searched. The hash function uses a prime *q*, whose value should be chosen such that 256q is no larger than a

computer word which makes the algorithm truly efficient. Exhibit 1 describes the checksum(hash) function as given in [30].

Exhibit 1: Checksum function

A binary string

$$X = x_1 x_2 \ldots x_n$$

can be regarded as a binary representation of the integer

$$H(X) = \sum_{i=1}^{N} x_i 2^{n-1} \quad (1)$$

For any integer q, the following is the hash function.

$$H_q(X) = H(X) \bmod q \quad (2)$$

Karp et al. devised a theorem for parallel pattern-matching. They suggested that the computation of the hashes for the substrings and their comparisons with the hash of the text string can be done in parallel using multiple processors in,constant time. This motivated us to experiment with the algorithm on a multicore GPU.

Theorem 12 [30]:
The string-matching problem for a pattern of length n and a text of length m ($n \pounds m$), where we find all matches, can be solved by m processors in time $O(\log m)$ with probability of error smaller than $0.697 / m^k$.

Given a text string *s* of length *n*, and a pattern of length m, the algorithm computes rolling hash(checksum) for both the pattern and consecutive substrings of length *m* of the searched string. Every time a substring's hash equals that of the pattern, a match is reported. Fig.3 illustrates the idea behind the Rabin-Karp algorithm. The rows with a red checkmark indicate the situations where the hash for the pattern and the text match. The algorithm can find any of a large number, say k, fixed length patterns in a text.

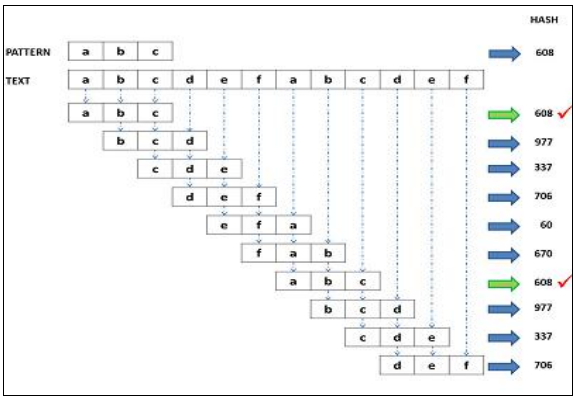

Fig. 3. Rabin-Karp algorithm

C. Pattern-matching with Rabin Karp on GPU

The CUDA framework has been used to develop and execute the algorithm on the GPU. The WinPcap library has been used for capturing the network traffic. The experiments that have been conducted on random as well as real traffic indicate that the proposed algorithm is upto 14X times faster than the Rabin Karp algorithm being executed on a CPU for the pattern search.

The host computer issues a *kernel* for the pattern-match to the GPU, which is executed on the device as several threads organized in thread blocks. One or more thread blocks, organized as warps are executed by each multiprocessor.

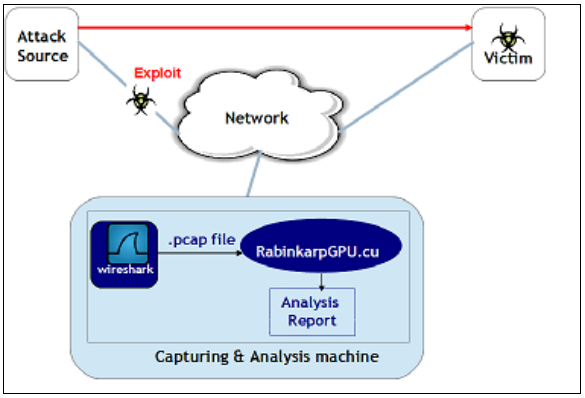

Fig. 4. Architecture of the proposed system

The implementation is listed in a step-by-step manner in Exhibit 2.

Exhibit 2: DPI on the GPU

Step 1:**Network Traffic Capture on the Host(CPU)** Tool: The open source protocol analyzer Wireshark [31] (known as Ethereal till 2006), or the classic old sniffer,Tcpdump [32], whose GUI is not as good as Wireshark but it requires fewer resources and also has fewer security holes. Both tools rely on the packet sniffing libraries, WinPcap and Libpcap[32].

Step 2:Transfer **of packets(*Text*) to the Device(GPU),** through a covert channel

Step 3: **Copy *patterns* from the host to the shared memory** (*cudaMemcpyAsync()* is used as it is non-blocking to the host,so control immediately returns to the host thread.)

Step 4: Considering each thread in a block responsible for one test pattern, **load a chunk of text into device memory,** with each thread doing one test pattern

Step 5: **Loop through the chunk of text looking for matches**

Step 6: Once pattern-matching completed with the loaded chunk, **load in the next chunk of text and repeat**

Step 7: When **end of file, store matches in a global array**

The pseudocode for the parallel implementation of the Rabin-Karp Algorithm for the GPU is as follows:

Pseudocode 1: Rabin Karp Algorithm for GPU (*rabinkarpGpu.cu*)

{P is the set of all fixed-string patterns
(string patterns of Snort V2.8)}
{T is the input string elements}
{q is the prime}

Phase 1: Data Loading (by the host)
1.Host Loads the patterns from the dictionary in *P*
2.Reads-in the PCAP file(s) for the Text and loads a packet at a time in T
3.Set the value of the prime number for computation
4.Host program calls the Rabin-Karp pattern-matching GPU Kernel with the parameters P,T and q

Phase 2:Pattern-matching (in the kernel)
1.GPU kernel calculates the hash(checksum) value of the pattern and the first substring of text(of length m)
2.Slide the pattern over text one by one
3.Checks the hash values of the substring (current window of text) and pattern
4.If the hash values match then only check for characters one by one,
If text and pattern match character by character then add position to matches
5.Calculate hash value for next substring and repeat from step 4
6. Return matches to the host

IV. Experimental Evaluation

A. Experiment Results

The proposed algorithm is executed on a commodity graphics card equipped with the programmable NVIDIA GeForce GT 635M having equipped with 144 CUDA cores and a Graphics Clock upto 675 MHz which means it has high parallel computation power to perform the task of multiple pattern-matching in parallel thereby delivering a significant speedup as compared to the regular scenario where the intrusion detection is performed on a CPU. The GPU uses a unified clock instead of a shader clock which leads to higher efficiency. Table 1. compares the execution time of the serial implementation on a quadcore Intel CPU with the parallel implementation of the Rabin-Karp pattern-matching algorithm on a multicore GPU with an without code optimization techniques. We observe that there is a good speedup gained with the optimized implementation on the GPU as compared to its serial implementation as shown in Fig.5. The average speedup observed for the optimized code GPU implementation is 12X. The best case speedup observed is 14X. The filesize of the .pcap file, the packet capture file is a significant factor. The performance continues to demonstrate a good speedup but for very large sizes, it begins to plateau, because of latency and memory throughput. Fig. 6 shows the GPU performance results for different packet capture file sizes.

Table 1. Execution-Time Comparison(CPU and GPU)

Filesize (Packet capture file)	Quadcore CPU	Nvidia GPU (UnOptimized)	Nvidia GPU (Optimized)
85MB	197.6ms	75.8ms	21.2ms
124MB	298.4ms	98.0ms	32.4ms
238MB	413.3ms	121.0ms	39.2ms
434MB	563.8ms	167.5ms	43.9ms
528MB	899.9ms	234.5ms	61.4ms
776MB	984.9ms	266.0ms	70.3ms
876MB	1197ms	354.6ms	84.3ms

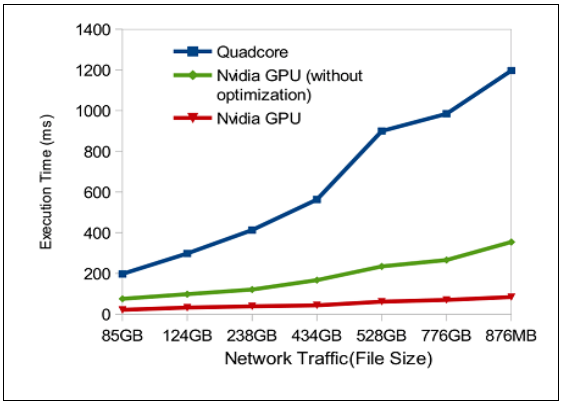

Fig. 5. Performance Results on CPU and GPU(NVIDIA)

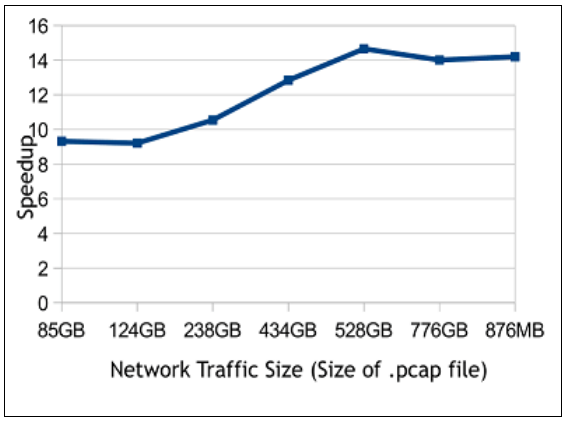

Fig. 6. Speed Improvement on GPU(Nvidia)

B. Code Optimization

The on-chip shared memory available for each multiprocessor is a great feature for writing a well optimized CUDA code. The shared memory is a small high bandwidth memory which is private to the threads in a block. The shared memory latency is 100x lower than the global memory latency. Storing frequently reused data to the shared memory can deliver substantial performance improvements [33], therefore the pattern and the hash table are stored in the shared memory to deliver the maximum performance. The data transfers between

Copyright © 2015 MECS

the CPU and GPU are done using the asynchronous variants for copying data (***cudaMemcpyAsync()***), which immediately return control to the host.

V. Related Work

A lot of research efforts for the efficient use of GPU for improving its performance for the pattern-matching, with a specific focus on decreasing the high processing time, have been studied in this work. Smith et al. examined the viability of SIMD-based architectures for signature-matching and presented a detailed architectural analysis [33]. Several researchers have experimented with various algorithms and techniques to leverage the abilities of the GPU in the field of intrusion detection [34][35]. Cascarano et al. addressed the state space issues of the traditional DFA based approaches and presented a NFA(non-deterministic automata) based regular expression engine for large and complex rule sets [36]. Vasiliadis et al. ported the open source IDS, Snort to a GPU, Gnort which has since been extended and improvised by several researchers to gain higher speed improvements [37]. Jacob et. al. also offloaded the packet-processing task of SNORT to the GPU; PixelSnort was programmed with the complex high level shading language Cg [38]. Tumeo et al. implemented the Aho-Corasick pattern-matching algorithm on a GPU [39].

Chung et al. proposed a GPGPU based parallel packet classification method to decrease the huge computing time to filter large number of packets[40]. Efforts in this research work aim to address the performance issues studied in the related endeavours.

One of the most common CUDA-optimization strategies is Memory Coalescing [41], which maximizes the global memory bandwidth usage ,by reducing the number of bus transactions, threads with adjacent global indexes in a block are forced to request contiguous data(packets) from global memory. A perfectly coalesced pattern greatly improves throughput. GPUS have multilevel set associative caches, which gives rise to the need for careful performance analysis due to the set associativity. Gusev and Ristov have notable contributions in this direction [42]. Several researchers including Fatahalian et al., Sim et al. and Ristov et al., to name a few, have highlighted the fact that GPU Caches affect application performance in a significant manner [43][44][45]. Mittal S. presented the classification of techniques for managing and leveraging the GPU cache [46].

An unintended application of GPUs is by hackers for accelerated password cracking. Olufun et al. in their work for developing a security model for WLANS, demonstrated the use of a GPU based encryption breaking tool, pyrit which cracked a dictionary file much faster than a CPU based tool [47].

VI. Conclusion & Future Work

In this paper the CUDA Toolkit is used to implement the parallel implementation of the Rabin-Karp pattern-matching algorithm. Both the serial and the parallel implementations of the algorithm were compared in terms of the execution time for varying network traffic sizes.

The parallel implementation on a multicore GPU of the pattern-matching algorithm achieved a speedup of upto 14X. The performance begins to show a plateau effect when the traffic size increases beyond a limit as the GPU is limited with its memory, nonetheless the system delivers optimal performance at increasing traffic. Peak performance has been achieved by removing dependencies on the global memory and improving the memory coalescing. The Nvidia Profiler [48] and the CUDA Occupancy Calculator[49] have been greatly helpful in profiling the application and discovering the bottlenecks. The satisfactory speedup gains motivates us to improve the code further by even more optimization of the code in the area of memory management. The important considerations in the work have been :

1. Selection of an appropriate candidate for the desired results on the GPU is very important. Not every serial code implemented on the GPU gives better performance.
2. Conversion of a serial code to the parallel code will not result in the desired improvement. Implementing proper CUDA code optimization results in efficient use of the GPU.

Most of the tools used in the research have been open-source except for CUDA. Developing and testing the application in OpenCL are being considered for future work so the entire work becomes open-source.

To ensure higher speedups even with highly increasing traffic size, future work efforts will be to utilize the power of grid computing and test the system on a GPU-grid.

The results in this paper indicate that definitely DPI performed on a GPU yields excellent performance especially when the underlying algorithm is carefully selected and tuned.

Acknowledgment

We would like to express our gratitude to the reviewers whose advice contributed to major improvements in the paper. Sincere acknowledgment of gratitude is for our young children in our respective families for being the source of joy and relaxation needed after hours of sitting with the laptop and our respective spouses for being highly patient and supportive.

References

[1] Owens, John D., Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. Proc. IEEE, 2008. vol. 96, no. 5: p. 879 -899.

[2] "CUDA Parallel Computing Platform". [Online].Available: http://www.nvidia.in/object/cuda_home_new.html. [Accessed 27 September 2013].

[3] AbuHmed, Tamer, Abedelaziz Mohaisen, and DaeHun Nyang. Deep packet inspection for intrusion detection systems: A survey. Magazine of Korea Telecommunication Society, November 2007. vol. 24, No. 11: p. 25-36.

[4] Cabrera, Joao BD, Jaykumar Gosar, Wenke Lee, and Raman K. Mehra. On the statistical distribution of processing times in network intrusion detection. In 43rd IEEE Conference on Decision and Control, December 2004. vol. 1: p. 75–80.

[5] Rafiq, ANM Ehtesham, M. Watheq El-Kharashi, and Fayez Gebali. A fast string search algorithm for deep packet classification. Computer Communications, June 2004. 27(15): p. 1524–1538.

[6] Tan, Lin, and Timothy Sherwood. Architectures for bit-split string scanning in intrusion detection. IEEE Micro 1 , 2006: p. 110-117.

[7] Dharmapurikar, Sarang, and John W. Lockwood. Fast and scalable pattern matching for network intrusion detection systems. Selected Areas in Communications, IEEE Journal on, 206. 24, no. 10: p. 1781-1792.

[8] Piyachon, Piti, and Yan Luo. Efficient memory utilization on network processors for deep packet inspection. Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, 2006: p.71-80. ACM.

[9] Wagner, Arno, and Bernhard Plattner. Entropy based worm and anomaly detection in fast IP networks. In Enabling Technologies: Infrastructure for Collaborative Enterprise, 2005. 14th IEEE International Workshops on, 2005.: p. 172-177. IEEE.

[10] Liu, Ting, Zhiwen Wang, Haijun Wang, and Ke Lu. An Entropy-based Method for Attack Detection in Large Scale Network. International Journal of Computers Communications & Control, 2014. 7, no. 3: p. 509-517.

[11] Jeyanthi, N., N. Ch SN Iyengar, PC Mogan Kumar, and A. Kannammal. An enhanced entropy approach to detect and prevent DDoS in cloud environment. International Journal of Communication Networks and Information Security (IJCNIS) , 2013. 5, no. 2: .

[12] Aho, Alfred V., and Margaret J. Corasick. Effcient string matching: An aid to bibliographic search. Communications of the ACM, 1975. 18(6): p. 333–340.

[13] Boyer, Robert S., and J. Strother Moore. A fast string searching algorithm. Communication of ACM, 1977. 20(10): p. 762-772.

[14] Zhang, Wu, Zhangxin Chen, Craig C. Douglas, and Weiqin Tong, eds. High Performance Computing and Applications: Second International Conference, HPCA 2009, Shanghai, China, Revised Selected Papers. 2010. Vol. 5938. Springer.

[15] "iXBT Labs - Computer Hardware In Detail", [Online]. Available: http://ixbtlabs.com/articles3/video/cuda-1-p1.html. [Accessed 28 September 2013].

[16] "NVIDIA", http://www.nvidia.in/page/home.html

[17] "NVIDIA CUDA Zone", https://developer.nvidia.com/cuda-zone.

[18] Ghorpade, Jayshree, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. Gpgpu processing in cuda architecture, 2012. arXiv preprint arXiv:1202.4347.

[19] "NVIDIA's Next Generation CUDA Compute Architecture: Fermi Architecture", [Online]. Available: http://www.nvidia.in/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, [Accessed 28 July 2013].

[20] "Whitepaper: NVIDIA GeForce GTX 680", [Online]. Available: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf, [Accessed 29 September 2013].

[21] "Whitepaper:NVIDIA GeForce GTX 750 Ti", [Online]. Available: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, [Accessed 29 September 2013].

[22] "NVIDIA Updates GPU Roadmap;Announces Pascal", http://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal/

[23] Stone, John E., David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, 2010. 12, no. 1-3: p. 66-73.

[24] Buck, I., T. Foley, D. Horn, J. Sugerman, P. Hanrahan, M. Houston, and K. Fatahalian. BrookGPU, 2003. http://graphics.stanford.edu/projects/brookgpu/

[25] NVIDIA: NVIDIA CUDA compute unified device architecture programming guide, 2007. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA CUDA Programming Guide 1.0.pdf

[26] Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. Queue, 2008. 6(2): p. 40-53.

[27] D. E. Knuth, J. MoKnuth, Donald E., James H. Morris, Jr, and Vaughan R. Pratt. Fast pattern matching in strings. SIAM journal on computing, 1977. 6(2): p. 323-350.

[28] Wu, Sun, and Udi Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, 1994.

[29] Coit, C. J., Staniford, S., & McAlerney, J. (2001). Towards faster string matching for intrusion detection or exceeding the speed of snort. In DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01, Proceedings 2001. Vol. 1: p. 367-373. IEEE.

[30] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. IBM J.Res. Dev., 1987. 31(2): p. 249–260. ISSN 0018-8646.

[31] "Wireshark". [Online]. Available: http://www.wireshark.org/.

[32] "Tcpdump, Libpcap and Winpcap". [Online]. Available: http://www.tcpdump.org/.

[33] Smith, Randy, Neelam Goyal, Justin Ormont, Karthikeyan Sankaralingam, and Cristian Estan. Evaluating GPUs for network packet signature matching. InPerformance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, 2009: p. 175-184. IEEE.

[34] Huang, Nen-Fu, Hsien-Wei Hung, Sheng-Hung Lai, Yen-Ming Chu, and Wen-Yen Tsai. A gpu-based multiple-pattern matching algorithm for network intrusion detection systems. Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008, 22nd International Conference on, 2008: p. 62-67. IEEE.

[35] Lin, Cheng-Hung, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. Accelerating pattern matching using a novel parallel algorithm on gpus. Computers, IEEE Transactions on, 2013. 62, no. 10: p. 1906-1916.

[36] Cascarano, Niccolo, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. iNFAnt: NFA pattern matching on GPGPU devices. ACM SIGCOMM Computer Communication Review 40, 2010. no. 5 : p. 20-26.

[37] Vasiliadis, Giorgos, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In Recent Advances

in Intrusion Detection, Springer Berlin Heidelberg, 2008: p. 116-134.
[38] Nigel Jacob and Carla E. Brodley, Offloading IDS Computation to the GPU. ACSAC, Dec. 2006: p. 371-380.
[39] Tumeo, Antonino, Oreste Villa, and Donatella Sciuto. Efficient pattern matching on GPUs for intrusion detection systems. In Proceedings of the 7th ACM international conference on Computing frontiers, 2010: p. 87-88. ACM.
[40] Hung, Che-Lun, Yaw-Ling Lin, Kuan-Ching Li, Hsiao-Hsi Wang, and Shih-Wei Guo. Efficient GPGPU-based parallel packet classification. In Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on, 2011: p. 1367-1374. IEEE.
[41] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, Understanding the impact of CUDA tuning techniques for Fermi. International Conference on High Performance Computing and Simulation (HPCS), IEEE, 2011: p. 631-639.
[42] Gusev, Marjan, and Sasko Ristov. Performance Gains and Drawbacks using Set Associative Cache. Journal of Next Generation Information Technology, 2012. 3, no. 3.
[43] Fatahalian, Kayvon, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS conference on Graphics hardware, 2004: p. 133-137. ACM, 2004.
[44] Sim, Jaewoong, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In ACM SIGPLAN Notices, 2012. vol. 47, no. 8: p. 11-22. ACM.
[45] Ristov, Sasko, Marjan Gusev, Leonid Djinevski, and Sime Arsenovski. Performance impact of reconfigurable L1 cache on GPU devices. Computer Science and Information Systems (FedCSIS), 2013, Federated Conference on, 2013: p. 507-510. IEEE.
[46] Mittal, Sparsh. A Survey Of Techniques for Managing and Leveraging Caches in GPUs. Journal of Circuits, Systems, and Computers, 2014. 23, no. 08: p. 1430002.
[47] Olufon, Tope, Carlene EA Campbell, Stephen Hole, Kapilan Radhakrishnan, and Arya Sedigh. Mitigating External Threats in Wireless Local Area Networks. International Journal of Communication Networks and Information Security (IJCNIS) , 2014. 6, no. 3.
[48] "NVIDIA Profiler", [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler, [Accessed 25 February 2014].
[49] "CUDA Occupancy Calculator", [Online]. Available: http://developer.download.nvidia.com/.compute/cuda/CUDA_Occupancy_calculator.xls, [Accessed 23 March 2014].

Authors' Profiles

Jyotsna Sharma is a research scholar at the CSED,Thapar University. She has focused her research on DPI based Forensic Analysis of Network Traffic using Grid Infrastructure. She is an M.Phil. in Computer Science and also a Graduate Member of The Institution of Engineers(India). She is a Certified Ethical Hacker(C|EH) from the EC-Council. She has several research articles to her credit and has also contributed a chapter to the 'Handbook of Research on Grid Technologies and Utility Computing, an IGI Global Publication, and is currently authoring a book on 'Web Engineering'. She received the Suman Sharma National Award from the Institution of Engineers (India) for academic distinction in the computer engineering discipline. She won the 2009 Google Global Community Scholarship for GHC2009. She has several years experience as an Assistant Professor and a Software Developer.

Dr. Maninder Singh is an Associate Professor at the Computer Science and Engineering Department, Thapar University, Patiala and also heads the Centre of Information and Technology Management (CITM). He received his Bachelor's Degree from Pune University, Master's Degree, with honours in Software Engineering from Thapar Institute of Engineering & Technology, and holds his Doctoral Degree with specialization in Network Security from Thapar University. His research interest includes Network Security, Grid Computing, Secure coding and is a strong torchbearer for Open Source Community. He has many research publications in reputed journals and conferences. He is on the Roll-of-honour @ EC-Council USA, being certified as Ethical Hacker (C|EH), Security Analyst (ECSA) and Licensed Penetration Tester (LPT).

Dr. Singh has successfully completed many consultancy projects (network auditing and penetration testing) for renowned national bank(s) and corporate and also architected Thapar University's network presence. In 2003 his vision for developing an Open Source Based network security toolkit was published by a leading national newspaper. Linux For You magazine from India declared him a 'Tux Hero' in 2004. He is a Senior Member of IEEE, Senior Member of ACM and Life Member of Computer Society of India. He has been volunteering his services for Network Security community as a reviewer and project judge for IEEE design contests. Recently Dr. Singh was aired on "Centre Stage" @ Headlines Today, national channel.

How to cite this paper: Jyotsna Sharma, Maninder Singh,"CUDA based Rabin-Karp Pattern Matching for Deep Packet Inspection on a Multicore GPU", IJCNIS, vol.7, no.10, pp. 70-77, 2015.DOI: 10.5815/ijcnis.2015.10.08

Copyright © 2015 MECS