GLoP: Enabling Massively Parallel Incident Response Through GPU Log Processing

Xavier J. A. Bellekens
Department of Electronic and Electrical Engineering
University of Strathclyde
Glasgow, G1 1XW, UK
xavier.bellekens@strath.ac.uk

Christos Tachtatzis
Department of Electronic and Electrical Engineering
University of Strathclyde
Glasgow, G1 1XW, UK
christos.tachtatzis@strath.ac.uk

Robert C. Atkinson
Department of Electronic and Electrical Engineering
University of Strathclyde
Glasgow, G1 1XW, UK
robert.atkinson@strath.ac.uk

Craig Renfrew
Agilent Technologies UK
5 Lochside Avenue
Edinburgh, EH12 9DJ, GB
craig_renfrew@agilent.com

Tony Kirkham
Agilent Technologies UK
5 Lochside Avenue
Edinburgh, EH12 9DJ, GB
tony_kirkham@agilent.com

ABSTRACT

Large industrial systems that combine services and applications, have become targets for cyber criminals and are challenging from the security, monitoring and auditing perspectives. Security log analysis is a key step for uncovering anomalies, detecting intrusion, and enabling incident response. The constant increase of link speeds, threats and users, produce large volumes of log data and become increasingly difficult to analyse on a Central Processing Unit (CPU). This paper presents a massively parallel **G**raphics Processing Unit (GPU) **Lo**g **P**rocessing (GLoP) library and can also be used for Deep Packet Inspection (DPI), using a prefix matching technique, harvesting the full power of off-the-shelf technologies. GLoP implements two different algorithm using different GPU memory and is compared against CPU counterpart implementations. The library can be used for processing nodes with single or multiple GPUs as well as GPU cloud farms. The results show throughput of 20 Gbps and demonstrate that modern GPUs can be utilised to increase the operational speed of large scale log processing scenarios, saving precious time before and after an intrusion has occurred.

Categories and Subject Descriptors

D.4.6 [**Security and protection**]: [Information flow controls]; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

Security, GPU, CUDA, Pattern Matching

SIN '14 Sep 09 - 11 2014, Glasgow, Scotland UK

http://dx.doi.org/10.1145/2659651.2659700

1. INTRODUCTION

Incident Response and threat detection play a key role in industrial system security. With the increased dependence of users on information gathering, e-commerce, social networking, and the Internet of Things (IoT), the amount of data to analyse before and after an intrusion has grown exponentially. This large data volume, in combination with the expanding number of malicious exploits from attackers, make it challenging for system administrators and incident response teams to analyse the logs. Existing frameworks and tools utilise modern search algorithms, however, most of these run sequentially on CPUs.

Research in malware detection [1], [2], Intrusion Detection Systems (IDS) [3], [4], and pattern matching [5], [6] demonstrate significant speed increases utilising parallelised hardware architectures such as Field Programmable Gate Arrays (FPGA) and Graphical Processing Units (GPU). These approaches effectively scale processing vertically and maximise speed from a single device. Cloud based research [7], [8], attempts to parallelise log-processing horizontally, by distributing workload to multiple nodes with services such as the Amazon EC2, GPU G2 instances. These two approaches, are by no means competing but rather complementary. Combining GPU processing in the cloud has the potential of massive speed increases.

This paper addresses the analysis of large scale log processing in a fast and cost-effective fashion using a single off-the-shelf GPU. The performance increase is not limited to a single GPU and can be utilised to enhance both multi-GPU nodes as well as GPU Cloud farms.

The typical use case scenario of the library is illustrated in Figure 1. Network equipment record events using the syslog protocol [9] on a server. The incident response team would periodically push the syslog traces for the last period, to a GPU processing server along with rules describing malicious activity. The GPU server will search through the logs and identify patterns that match any of the malicious rules generating alerts containing the number of suspected malicious activities and their location in the logs.

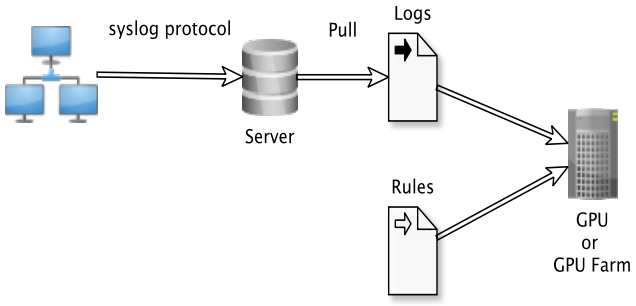

Figure 1: Typical use case scenario

The contributions of the paper are threefold. It proposes and implements a generic log processing library, called GLoP (**G**PU **Lo**g **P**rocessing). The library utilises the single-pattern matching algorithm Knut-Morris-Pratt (KMP) and the multi-pattern algorithm of Failureless Tries. Both algorithms are implemented for CPU and GPU and the performance of the implementations is quantified, for various number of search patterns. The library truncates the patterns to 8 characters reducing thread divergance and memory requirments on the GPU; a technique known as prefix or partial matching. Finally, GLoP contains two GPU implementations of Parallel Failureless Aho-Corasick algorithm that use global or texture memory. It is shown, that the implementation using the texture memory achieves double throughput compared to the implementation that uses global memory.

The remainder of this paper is organised as follows: Section 2 presents the related work, Section 3 introduces the GPU architecture and programming model. Sections 4 and 5 describe the Aho-Corasick algorithm and optimisation for its implementation in GLoP. The experimental environment and methodology are presented in Section 6. Section 7 describes and discusses the results obtained from multiple algorithm variants within GLoP. Finally, Section 8 presents the conclusions and future work.

2. RELATED WORK

Large scale log processing research has extensively studied the use of large scale data mining and big data scenarios using distributed frameworks analysis. *Shu et al.* have proposed a lightweight framework based on the Amazon Cloud Environment (EC2 and S3), using multiple nodes to speed up the log analysis processing, and harvesting the results using a map reduce implementation [7]. *Yang et al.* demonstrated that by using Hadoop MapReduce, it was possible to decrease the processing time of log files by 89% for intrusion detection purposes [8]. *Marty et al.* proposed a theoretical logging framework dedicated to cloud infrastructures and software as a service (SaaS) running on a third party public cloud service [10].

Cheng et al. described a fast filter virus detection engine running on GPUs based on eigenvalues with good performances [1]. Our previous work [11] has shown that a massively parallel pattern matching algorithm based on the Knuth-Morris-Pratt algorithm [12] can achieve a 29 fold increase in processing speed over CPU counterparts. From the output of these works, it is clear that the processing capabilities of off-the-shelf hardware have a great potential not only to increase the speed on a single stand-alone processing server but also on GPU cloud deployments [13].

String searching algorithms can be classified to single-pattern and multi-pattern matching. Single pattern matching algorithms search the complete string for a single pattern sequentially. The naive approach to search for a single pattern is to iteratively walk through the text string and every time there is a mismatch or a complete match, the algorithm rewinds back. Optimised algorithms such as the Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) avoid rewinding by introducing failure and backtracking tables respectively [14].

On the other hand multi-pattern matching algorithms search simultaneously for multiple patterns in the text string. The most common multi-pattern algorithm is the Aho-Corasick (AC) [15], [16] which has been implemented in a variety of hardware architectures such as FPGAs [4][17] and GPUs [18].

For cases where the cross-correlation between pattern prefixes is low, a two staged searching approach can be utilised to improve parsing speeds. In these scenarios, the patterns are truncated to create a set of prefixes. The first searching stage filters the text locations where the prefixes match. Subsequently the identified locations are passed to a secondary process where the patterns beyond the prefix are searched. This completes the searching for the full pattern. Such techniques are known as prefix or partial pattern matching [19, 20]. In this work, only the first searching stage is considered as the prefix cross-correlation is low and patterns share the same prefix for 0.0001% of the time [21]. Therefore the additional overhead to complete the full pattern search is negligible.

3. CUDA PROGRAMMING MODEL

The Compute Unified Device Architecture (CUDA) framework offers the possibility to researchers to use GPUs for General Purpose Graphics Processing. The framework allows researchers to access hardware features using an extension of the C99/C++ language from the host.

The present implementation of the library is implemented for the Tesla graphics card. The device consists of different multiprocessors each one of them containing streaming processors (SP). Each SP executing thousands of threads. Threads running on the GPU are organised in thread blocks. Within each block, the threads are organised in warps, each warp contains the same number of threads (32 for the Tesla K20 GPU). Each wrap executes in a Single Instruction Multiple Thread (SIMT) fashion and the multiprocessor periodically switches between warps maximising the resources, and hiding latencies [22].

The GPU contains different types of memory, *Global, Texture, Constant, Shared* which need to be managed explicitly at compilation time, by the programmer. The global memory is the largest memory on the device, and requires the more clock cycles to be accessed. The texture and constant memories are cached, and require less clock cycles to be accessed. Finally the shared memory is shared between

threads within a block and requires even fewer clock cycles but is a very scarce resource [23].

4. AHO-CORASICK

The AC algorithm is a multi-pattern matching algorithm and is able to identify multiple patterns in the text string on a single pass. The AC algorithm consists of two phases: the first phase constructs a state machine from a set of patterns, while the second phase searches for the patterns in the text.

A typical state machine diagram constructed during the first phase of the algorithm is shown in Figure 2. The state machine shown, consists of two patterns $P = \{HIS, SHE\}$. The straight arrows represent the "regular" transitions which occur when the text is matching, while the dashed arrows represent the "failure" transitions, when a mismatch occurs but the prefix of another pattern is matched. This allows single threaded implementation of the AC algorithm to avoid rewinding and back-track matches of multiple patterns on a single pass.

For example, for the text $T =$ "$SHIS$" and the state machine of Figure 2, the algorithm will start with two consecutive regular transitions from the initial state \$ to S and from S to H when comparing the first two characters of the text T. On the third comparison, the character I from the text string T does not result to a regular transition and consequently the failure transition is activated followed by a regular transition from H to I. Finally, for the last comparison results to a regular transition from I to S.

The most common modification to adopt the AC algorithm in a multi-threaded environment is to split the text T in multiple chunks and assign one chunk per thread. However, this creates problems when the text contains patterns across two chunks. For example, when the state machine of Figure 2 is considered, for the text string shown in Figure 3, it is clear that the character sequence "HIS" exists on the boundary between the chunks assigned to *Thread 0* and *Thread 1* and neither of these threads will match it. That can be avoided if the text string is split in chunks that overlap each other by at least the length of the longest pattern minus 1. The implication of the minimum chunk overlap is that it limits the number of chunks especially for scenarios where the maximum pattern length is long. Consequently the number of threads used to process the input text are also limited and this could become a bottleneck considering the massive number of threads that can be launched in the GPU (tens of thousands) [24].

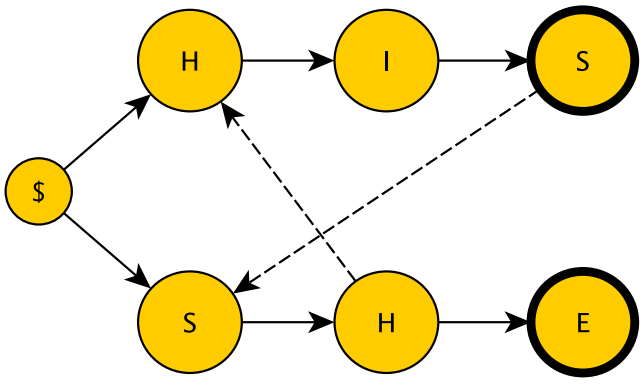

Figure 2: Aho-Corasick

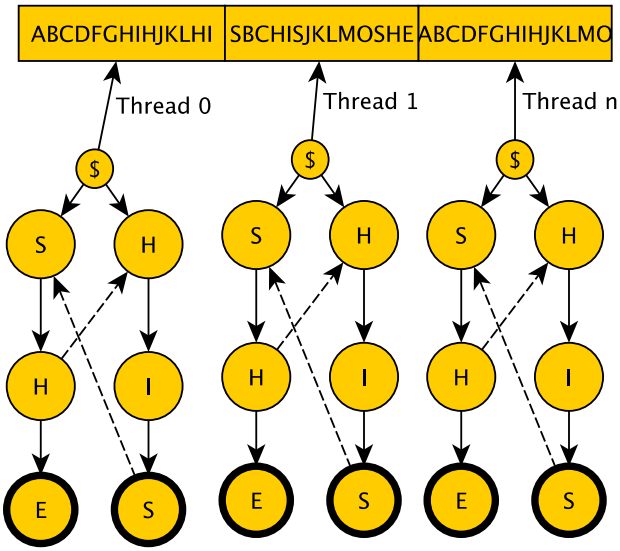

Figure 3: Aho-Corasick Boundary Problem

Another drawback of the AC algorithm in a multi-threaded environment is that the failure links create thread divergence. Thread divergence occurs when threads of the same wrap branch to different sections of the code. As described in Section 3, the GPU executes the wraps in a Single Instruction Multiple Thread manner and the GPU scheduler will allocate an execution slice to every branch. Threads on the same branch execute concurrently, while all the threads on a different branch are waiting for an execution slice. The scheduling within the wrap occurs in a Round Robin fashion. Thread divergence does not only consume more execution slices, but also increases memory access time as each branch makes memory transfers at different execution slices making access non-coalesced.

5. PARALLEL FAILURELESS TRIE AND PREFIX MATCHING

An alternative method to parallelise the AC algorithm that avoids the problem of patterns splitting over boundaries and reduces thread divergence is the Parallel Failureless Aho-Corasick (PFAC) [25]. With this method, "failure" transitions are removed from the state machine and one thread is instantiated for every character in the text string as shown in Figure 5. In the case of a match, the current thread will overlap the succeeding thread, allowing the state machine to change state. In the case of a mismatch, the current thread will terminate, releasing the resources and reducing divergence.

The failureless trie mechanism exploits effectively the large

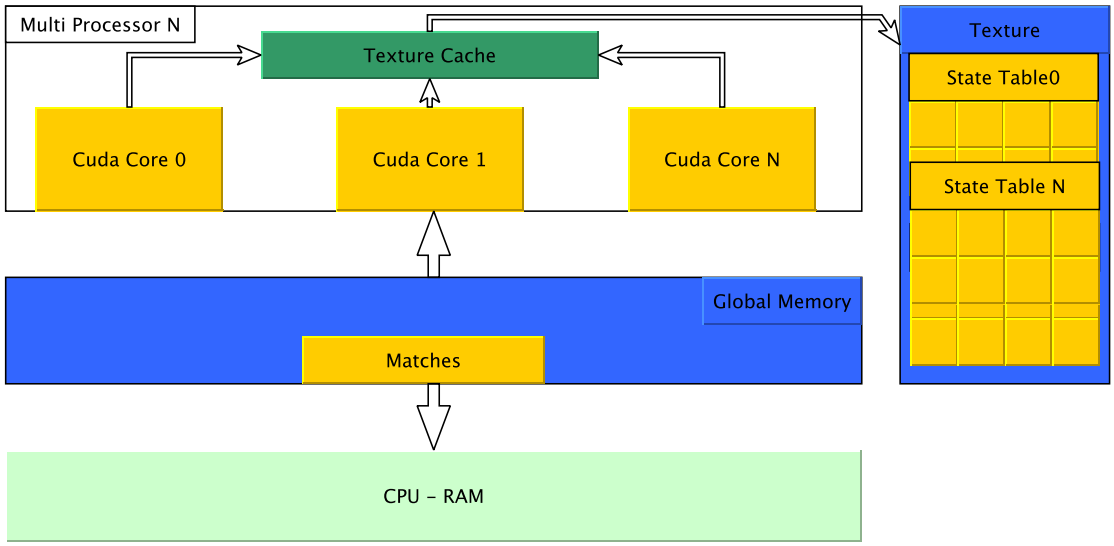

Figure 4: Texture Memory Architecture

number of threads that are available in massively parallelised off-the-shelve GPU devices, however further optimisation can be achieved.

Observe that the first memory transfer of text string to the threads occurs in a coalesced manner. Subsequently, each thread will transition to separate branches of the trie, leading to non-coalesced memory access and thread divergence. As the algorithm progresses deeper in the trie more threads will terminate due to mismatches leaving only a small number of threads continuing to match while the bulk of the threads terminate. This reduces the throughput significantly and can be avoided if the trie is truncated and all the threads terminate beyond a certain trie depth. The approach of truncating the trie is known as prefix or partial pattern matching [26, 21] and assumes that a secondary process takes over to complete the search; for example a CPU thread is lunched to complete the search.

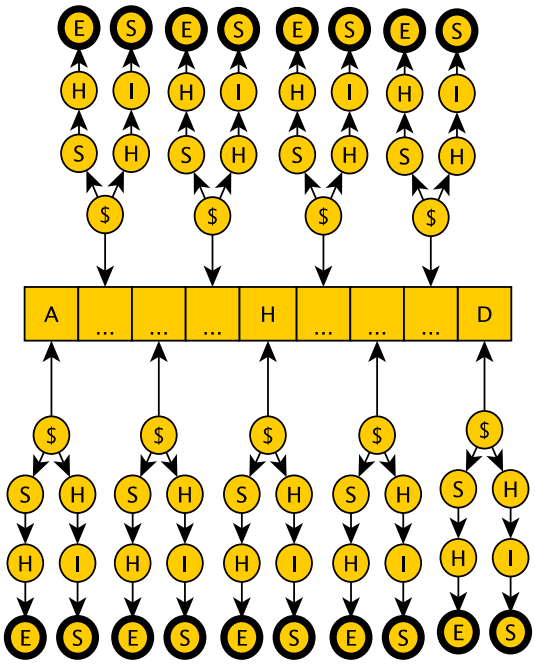

Figure 5: One Thread per Byte

In practise this approach is effective as the prefix length of a malicious activity pattern has a low probability being a false positive (i.e. the prefix exist in the text followed by a suffix that does not match that pattern). *Vasiliadis et al.* [21] has shown that a prefix length of 8 is adequate to achieve a false positive rate below 0.0001%. In this work this has been verified independently.

The prefix matching does not only reduce divergence, but also reduces memory footprint due to the reduced number of states in the trie. Memory reduction in itself may not be of significant benefit, but when texture memory is used the caching hit rate is increases leading to significantly reduced memory access times.

6. EXPERIMENTAL ENVIRONMENT

GLoP was implemented on a Nvidia Tesla K20m graphic card. The card consists of a single Printed Circuit Board (PCB), composed of 192 CUDA Cores distributed over 19 multiprocessors. The card is also composed 5 GB of global memory,and permits up to 26,624 active threads.

The base systems, is supplied with two Intel Xeon E5-2620 six core CPUs running at 2.0 GHz. The rig has a total of 64 GB ram. The server is running Ubuntu Server 11.10 (kernel version 3.0.0-12-server). The algorithm has been compiled with the optimisation brought by the latest version of CUDA 6.0, using C99.

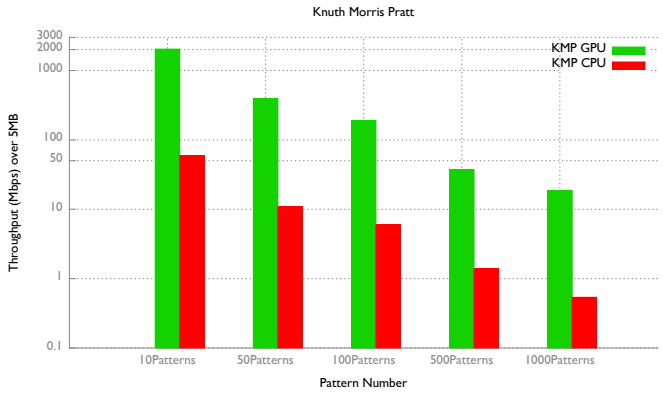

Figure 6: Throughput of the KMP Algorithm over 5MB chunk of log files

The GLoP library is executed using unified memory, allowing the users to access variables stored in global memory from the CPU and the GPU at any given time, and to facilitate the execution of streamed kernels. Each stream sends logs to GPU to be analysed, and matched against the failureless trie. During the launch of a kernel, the CPU is free to stream the next batch of logs to the GPU.

Two implementations of the algorithm are evaluated using global and texture memory. For the texture memory versions, the trie must be represented in memory using `int` types which are subsequently matched to ASCII values (e.g. "A"=65). The texture memory allows transactions to be cached, requiring fewer clock cycles to complete transfers compared to global memory accesses. The caching procedure is illustrated in Figure 4 where CUDA Cores (within a Streaming Multiprocessor) have access to texture memory via the cache, potentially speeding up significantly memory access.

During the evaluation, the algorithm is run 100 times and the results presented are the average of the total runs. This allows to mitigate the sources of jitter; such as background processes running competing for resources available. The patterns searched are also generated randomly against each log files.

The log files used during the trials have been automatically generated, using the Mersenne Twister (MT) uniform pseudo-random number generator [27]. Each synthetic log has an exact file size of 100 MB, and its uniqueness is ensured by computing the SHA256 hash.

In all comparison the throughput is calculated as follows:

- Let N be the size of the log data sent to the GPU.
- Let $Time_{gpu}$ be the time elapsed during with the algorithm is running.

$$\frac{8 * N}{Time_{gpu}} = Throughput \quad (1)$$

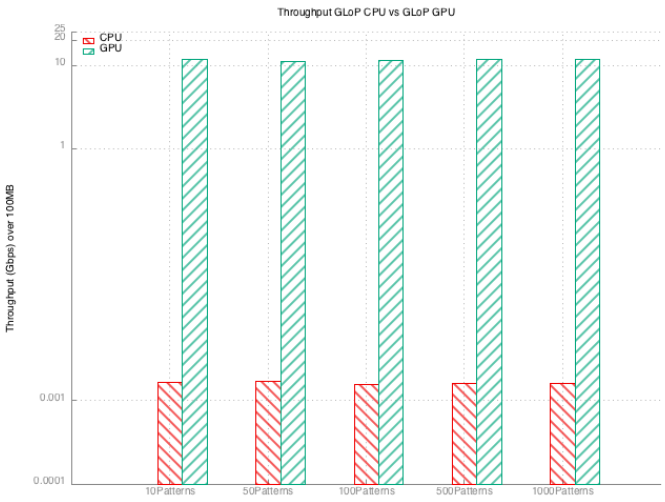

Figure 7: Throughput of the GLoP CPU-GPU Library over 100MB log files

7. EVALUATION

GLoP implements the multi-pattern failureless trie algorithm using global and texture memory as well as the single pattern algorithm Knuth-Morris-Pratt for comparison purposes. The KMP algorithm performance on CPU and GPU for varying number of patterns is shown in Figure 6. As the number of pattern increase, the throughput decreases linearly as the algorithm performs a single pattern search at time. The single threaded CPU version of the KMP algorithm achieves throughput of 50 Mbps when matching a log file of 5 MB against 10 patterns. On the CPU, the throughput is 10 times less when matching the same log file against 100 patterns. Despite the fact that the GPU version of the algorithm achieves throughput nearly 2,000 Mbps when 10 patterns at searched while the throughput decrease linearly to 15 Mbps for 1,000 patterns.

The KMP algorithm does not have sufficient throughput to cope with large scale log analysis, even with the boosted performance on the GPU. Its main shortcoming is that the

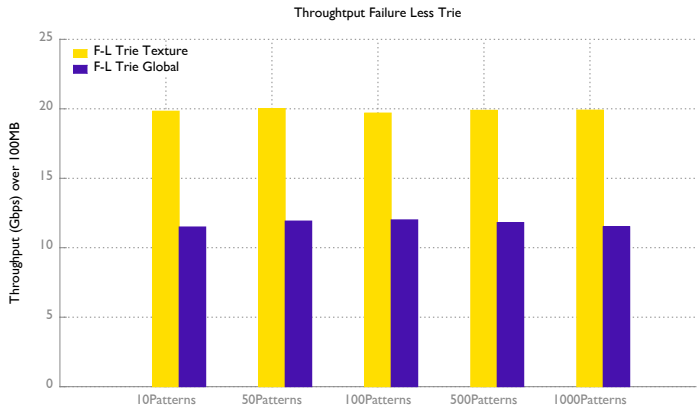

Figure 8: Throughput of the GLoP Library over 100MB log files

single pattern searching does not scale well for multiple patterns. GLoP shows that the multi-pattern algorithms such as the failureless trie are able to maintain a constant throughput independent of the number of patterns. This is demonstrated on Figure 7 where the performance of the Parallel Failureless Aho-Corasick algorithm is shown to sustain throughput for both the CPU and GPU. The performance of the failureless trie algorithm on the GPU when global memory is used is 11 Gbps for the 10 patterns and 100 MB file, while the corresponding CPU version achieves throughput of 1.7 Mbps.

Implementation of the failureless trie can be further improved when texture memory is used. The operation of the algorithm in this case is identical with the only different being that the state transition table being stored in texture memory. The performance comparison between the two algorithm using the global and texture memory respectively is shown in Figure 8. The texture implementation is shown to achieve double the throughput and this can be attributed to the caching capabilities which significantly reduce the cycles required to retrieve data from memory. The global memory implementation culminates at 11 Gbps for a 100 MB log file whereas the texture implementation reaches an overall throughput of 20 Gbps.

8. CONCLUSION AND FUTURE WORK

This paper presented GLoP, a massively parallel incident response engine offloading the large scale log processing, and pattern matching to the GPU allowing the CPU to concentrate on other tasks.

In this work GLoP has been evaluated against the single pattern matching algorithm Knuth-Morris-Pratt, and has demonstrated a throughput of 11 Gbps with the failure-less global memory implementation and an overall throughput of 20 Gbps for the failure-less texture memory implementation. Various synthetic log files have been used proving its efficacy, and in particular its ability to considerably speed up incident response processes, while remaining cost-effective. This work also highlighted the high performances demonstrated by the engine, as a single node, but also its possible and complementary use to cloud based log processing frameworks, such as Hadoop, to increase the processing power.

In future work, it is planned to reduce the memory footprint of the failure-less trie when using larger state machines, and improve the library by using Message Passing Interface (MPI) technique to allow the library to run on an HPC as well as investigating real-time operations.

Acknowledgment

The authors would like to thank Agilent Technologies for their insightful comments and feedback as well as their support.

9. REFERENCES

[1] Y. W. Cheng, "Fast virus signature matching based on the high performance computing of GPU," in *Communication Software and Networks, 2010. ICCSN '10. Second International Conference on*, pp. 513–515, Feb 2010.

[2] J. Harwayne-Gidansky, D. Stefan, and I. Dalal, "FPGA-based SoC for real-time network intrusion detection using counting bloom filters," in *Southeastcon, 2009. SOUTHEASTCON '09. IEEE*, pp. 452–458, March 2009.

[3] N. Jacob and C. Brodley, "Offloading ids computation to the GPU," in *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pp. 371–380, Dec 2006.

[4] I. Sourdis and D. Pnevmatikatos, "Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system," in *Field Programmable Logic and Application* (P. Cheung and G. Constantinides, eds.), vol. 2778 of *Lecture Notes in Computer Science*, pp. 880–889, Springer Berlin Heidelberg, 2003.

[5] M. Attig, S. Dharmapurikar, and J. Lockwood, "Implementation results of bloom filters for string matching," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pp. 322–323, April 2004.

[6] R. Takahashi and U. Inoue, "Parallel text matching using GPGPU," in *Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing (SNPD), 2012 13th ACIS International Conference on*, pp. 242–246, Aug 2012.

[7] X. Shu, J. Smiy, D. Daphne Yao, and H. Lin, "Massive distributed and parallel log analysis for organizational security," in *Globecom Workshops (GC Wkshps), 2013 IEEE*, pp. 194–199, IEEE, 2013.

[8] S.-F. Yang, W.-Y. Chen, and Y.-T. Wang, "ICAS: An inter-vm IDS log cloud analysis system," in *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pp. 285–289, Sept 2011.

[9] R. Gerhards, "The syslog protocol," march 2009. Request for Comments RFC 5424 (Proposed Standard), IETF.

[10] R. Marty, "Cloud application logging for forensics," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, (New York, NY, USA), pp. 178–184, ACM, 2011.

[11] X. Bellekens, I. Andonovic, R. Atkinson, C. Renfrew, and T. Kirkham, "Investigation of GPU-based pattern matching," in *The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013)*, 2013.

[12] D. Knuth, J. Morris, Jr., and V. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[13] M. Xin and H. Li, "An implementation of gpu accelerated mapreduce: Using hadoop with opencl for data- and compute-intensive jobs," in *Service Sciences (IJCSS), 2012 International Joint Conference on*, pp. 6–11, May 2012.

[14] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, pp. 762–772, Oct. 1977.

[15] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, pp. 333–340, June 1975.

[16] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, (Washington, DC, USA), pp. 112–122, IEEE Computer Society, 2005.

[17] V. Dimopoulos, I. Papaefstathiou, and D. Pnevmatikatos, "A memory-efficient reconfigurable aho-corasick FSM implementation for intrusion detection systems," in *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*, pp. 186–193, July 2007.

[18] X. Zha and S. Sahni, "GPU-to-GPU and Host-to-Host multipattern string matching on a GPU," *Computers, IEEE Transactions on*, vol. 62, pp. 1156–1169, June 2013.

[19] T. Jianlong, L. Xia, L. Yanbing, and L. Ping, "Speeding up pattern matching by optimal partial string extraction," in *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, pp. 1030–1035, April 2011.

[20] W.-S. Jung and T.-G. Kwon, "An independently partial pattern matching for content inspection at multi gigabit networks," in *Advanced Communication Technology (ICACT), 2010 The 12th International Conference on*, vol. 2, pp. 1574–1579, Feb 2010.

[21] G. Vasiliadis and S. Ioannidis, "Gravity: A massively parallel antivirus engine," in *Recent Advances in Intrusion Detection* (S. Jha, R. Sommer, and C. Kreibich, eds.), vol. 6307 of *Lecture Notes in Computer Science*, pp. 79–96, Springer Berlin Heidelberg, 2010.

[22] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2012.

[23] Nvidia, "Cuda C programming guide," 2013. http://docs.nvidia.com/cuda/.

[24] Q. Wang and V. Prasanna, "Multi-core architecture on FPGA for large dictionary string matching," in *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pp. 96–103, April 2009.

[25] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, "Accelerating pattern matching using a novel parallel algorithm on GPUs," *Computers, IEEE Transactions on*, vol. 62, pp. 1906–1916, Oct 2013.

[26] N. Yazdani and P. Min, "Prefix trees: new efficient data structures for matching strings of different lengths," in *Database Engineering and Applications, 2001 International Symposium on.*, pp. 76–85, 2001.

[27] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998.