Investigation of GPU-based Pattern Matching

X Bellekens, I Andonovic, RC Atkinson
Department of Electronic & Electrical Engineering
University of Strathclyde
Glasgow, G1 1XW, UK
Email: {xavier.bellekens, robert.atkinson}@strath.ac.uk

C Renfrew, T Kirkham
Agilent Technologies UK Limited
5 Lochside Avenue
City of Edinburgh, EH12 9DJ, GB
Email: {craig_renfrew, tony_kirkham}@agilent.com

***Abstract*—Graphics Processing Units (GPUs) have become the focus of much interest with the scientific community lately due to their highly parallel computing capabilities, and cost effectiveness. They have evolved from simple graphic rendering devices to extremely complex parallel processors, used in a plethora of scientific areas. This paper outlines experimental results of a comparison between GPUs and general purpose CPUs for exact pattern matching. Specifically, a comparison is conducted for the Knuth-Morris-Pratt algorithm using different string sizes, alphabet sizes and introduces different techniques such as loop unrolling, and shared memory using the Compute Unified Device Architecture framework. Empirical results demonstrate a 29 fold increase in processing speed where GPUs are used instead of CPUs.**

***Keywords*—CUDA, Deep Packet Inspection, Intrusion Detection Systems, Security, String-Matching.**

I. Introduction

Pattern Matching is widely used in a plethora of domains, such as gene sequencing [1], or with critical systems such as Forensic Examination [2], Intrusion Detection Systems (IDS), and Deep Packet Inspection (DPI) [3]. In each case the main principle is the same, matching up to thousand patterns against a specified input. The ever increasing volumes of data that systems must process and the increasing level of attacks gives rise to a requirement for ultra high speed execution of pattern matching, and that is the motivation for this research.

The parallel capabilities of GPUs have recently been used to improve existing algorithms in several scientific areas, such as mechanics [4], applied mathematics [5], physics [6], image processing [7] and, of course, to accelerate pattern matching. These applications are known as General Purpose Computations on Graphics Processor Units (GPGPUs). GPUs are an attractive solution due to their low cost (commercial off-the-shelf devices), and their enormous computational power (by virtue of their highly parallelised architecture).

Previous research into parallel computation has focussed on a hardware approach, e.g. Field Programmable Arrays (FPGAs) [8][9][10]. FPGA provide a solution and can offer significant performance improvements over CPUs at the expense of high development costs. GPUs also promise significant performance improvements but with lower development costs than an FPGA equivalent [11].

The idea of implementing pattern matching algorithms on GPUs is not entirely new. Both the Boyer-Moore-Horspool algorithm [12], and a modified version of Aho-Corasick (Parallel Failureless Aho-Corasick algorithm) [13] have been implemented on Nvidia hardware with good results. *Jacob et al* created an Intrusion Detection System called PixelSnort by using the Cg programming language and the Knuth-Morris-Pratt algorithm to improve the CPU version of the Snort Intrusion Detection System [14].

In this paper, the performance of the Knuth-Morris-Pratt (KMP) algorithm [15] is compared for CPU and GPU architectures. The performance of pattern matching algorithms is dependent on pattern lengths, the size of alphabet, etc; therefore a range of parameters has been examined in order to permit proper comparison.

The algorithm is executed on a commercial off-the-shelf workstation running the CUDA 5 framework. The KMP algorithm was chosen specifically because it accesses memory multiple times when searching for patterns. This is important since the algorithm requires the pattern and its failure table to be stored in memory and accessed during matches and mismatches. This facilitates the measurement of performance for different versions of the algorithm, and using different improvement techniques specific to CUDA and / or to GPUs.

The rest of this paper is organised as follows. Section II provides a background on pattern matching, the existing approaches and their limits. Section III details the experimental methodology used to evaluate the KMP algorithm. Section IV highlights the results obtained by the GPU version of the algorithm, and aims to demonstrate the potential of pattern matching and IDS on GPUs. Finally, Section V summarises the points raised in the previous sections and provides a research direction for future work.

II. Pattern Matching

Pattern matching is the process of verifying that a specific pattern is present in a text having the same or a larger length. The pattern and the text are usually represented as a one dimensional array. Let n denote the length of the text S, m the length of the pattern, and P the pattern. Furthermore, we assume at all time that the length of the pattern is smaller or equal to the length of the text; that is, $m \leq n$.

In order to give an idea of what pattern matching is, the brute force algorithm, and the KMP algorithm are described below.

The brute force algorithm is one of the simplest approaches that works as follows. It compares the first element of the pattern P with the first element of the text S and tries to match the successive position until the entire pattern is matched.

ISBN: 978-1-902560-27-4 © 2013 PGNet

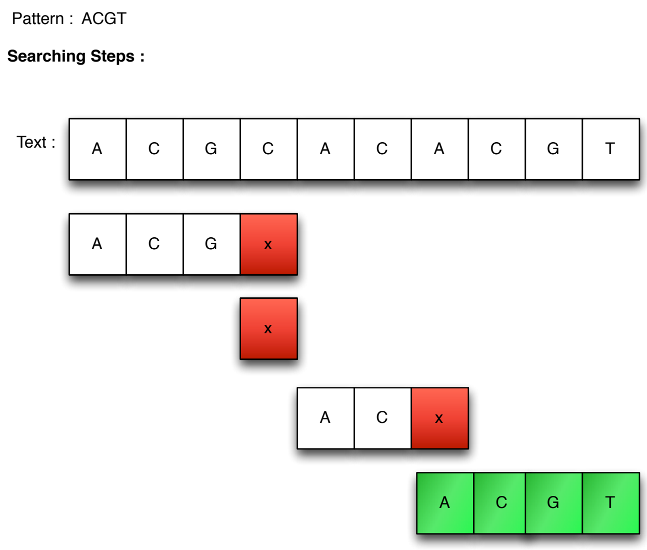

Fig. 1: Knuth-Morris-Pratt Steps

TABLE I: Failure Function

Position	**0**	**1**	**2**	**3**	**4**	**5**	**6**
P	A	C	A	A	C	T	C
Failure Table	0	0	1	1	2	0	0

Only after complete match or a mismatch does the algorithm move to the next position. This algorithm does not require any pre-processing, but it exhibits slow performance due to its computational inefficiency.

The KMP algorithm has been designed to reduce unnecessary comparisons, such as those performed in the brute force algorithm. The KMP uses a failure table to avoid comparing the text and the pattern at superfluous positions and skip the characters that have already been matched. This allows the algorithm to be significantly more efficient and hence faster than the brute force algorithm described previously.

The algorithm operates as follows. Firstly, it initialises a failure table for the pattern being searched. A failure function is built by analysing the pattern of interest and the repetition of its own first characters as shown in Table I. This enables the search function to calculate the number of characters that should be skipped during the searching phase. Figure 1 depicts how the algorithm works and how the shifting phase achieves to match patterns in a minimum amount of steps. The complexity of the algorithm is calculated as follows [15]:

$$O(m+n)$$

The algorithm can also be represented as a state machine, as illustrated in Figure 2. The state machine provides a better overview of how the algorithm behaves in relation to its associated failure table (skips are performed in the case of a mismatch).

Despite the greater efficiency of the KMP algorithm over a brute force approach, it does not scale well when matching

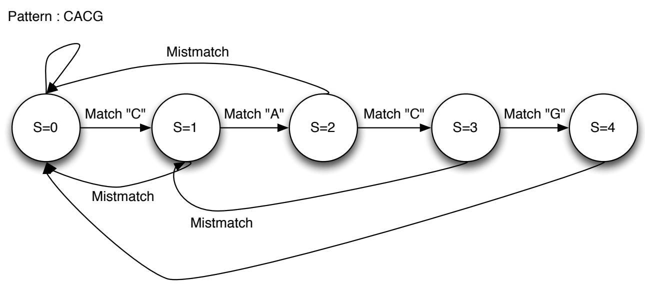

Fig. 2: Knuth-Morris-Pratt State Machine

multiple patterns: execution time increases linearly with number of patterns as a consequence of each pattern requiring one iteration through the text.

A. *Intrusion Detection Systems*

Intrusion Detection Systems are complex systems protecting critical infrastructures. Two types of IDS can be identified:

- Signature Detection
- Anomaly Detection

Signature detection, uses a pattern database to match the incoming and outgoing traffic [16] of a network, while the anomaly detection analyses the traffic behavior and flags all the anomalies [16][17]. This paper is focused on signature based systems, but these systems have flaws.

It is difficult and expensive to upgrade Intrusion Detection Systems, whereas the system presented in this paper uses off-the-shelve technologies, as well as public sources algorithms, and software allowing for a greater flexibility in the extensions and upgrades of the system; e.g., from more NICs to more powerful GPUs or to a new version of the CUDA framework. Current standard (un-parallelised) intrusion detection systems are not designed to support high bandwidth loads and an ever increasing number of patterns. Whereas off-the-shelve systems with GPUs offers the possibility to parallelise the resources intensive process of pattern matching and offload the CPUs, allowing them to focus on other important tasks.

III. Experimental Methodology

The experiments were performed on a high end Supermicro Superwork Station 7047GT-TPRF with two six-core Xeon processors (Intel E5-2620) running at 2.0Ghz, and supported by 64 GB of RAM. The machine also features an Nvidia Tesla K20m GPU (with 13 multiprocessors and 192 CUDA cores and 5Gb of DDR3 global memory); this permits up to 26624 active threads. Ubuntu Server 11.10 (kernel version 3.0.0-12-server) was the chosen operating system. This off-the-shelve high end hardware allows the achievement of outstanding performance with the algorithm running on the CPUs and on the GPUs.

In evaluating the execution time of algorithms some sources of error may be introduced: other processes may be running in the background and competing for resources, or lack of precision from the clock. In order to mitigate these

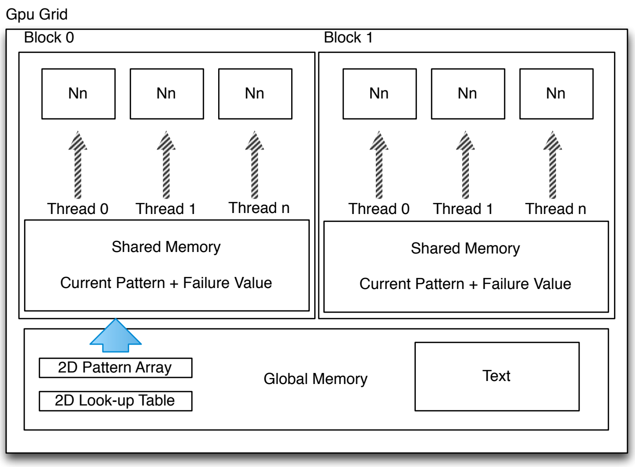

Fig. 3: GPU Grid

errors and also to provide confidence in the estimated execution times, each algorithm was averaged over 500 runs.

The KMP algorithm evaluated in this paper was implemented using C99 programming language and was compiled for the CPUs by *gcc 4.6.1* and for the GPUs using Nvidia's *nvcc* version 0.2.1221 without using any optimisation flags.

In order to permit fair comparison of pattern matching algorithms a number of well known evaluation cases are used. Let Σ denote the alphabet, and P_n be the number of patterns. The cases can thus be described as follows:

- searching genome sequence of the Yersinia Pestis bacteria (4.6 Mb) [18] which represents an alphabet of size $\Sigma = 4$;
- searching a password file leaked from the internet (4.0 Mb) which has an alphabet size of $\Sigma = 26$, i.e. the length of the English alphabet;

Although these evaluation cases are well known, they lack flexibility due to their fixed size alphabets; therefore, two additional evaluation cases were defined based on a 4.6 Mb file of uniformly randomised characters: one with an alphabet size $\Sigma = 10$ and one with size $\Sigma = 20$. The searched patterns were randomly chosen sequences with varying predefined length of $m = 20$, $m = 50$, $m = 100$ and $m = 500$ characters. A varying number T_n of threads was also used over a varying number of patterns $P_n = 10$, $P_n = 100$ and $P_n = 1000$ of $m = 20$ and compared with the CPUs.

To measure the improvement achieved by the GPU version of the *Knuth-Morris-Pratt* algorithm, the execution time was used. This metric is the total time needed for the algorithm to find all patterns searched for in the input text, and includes the transfer time between the host and the GPU (excluding the pre-processing time). The transmission and processing time was measured using standard C APIs and the timer function included in the CUDA toolkit allowing a precision of $1ns$. Only the performances of the searching phases were evaluated, allowing to transpose the algorithm later on to other systems and offload the pre-processing to other devices.

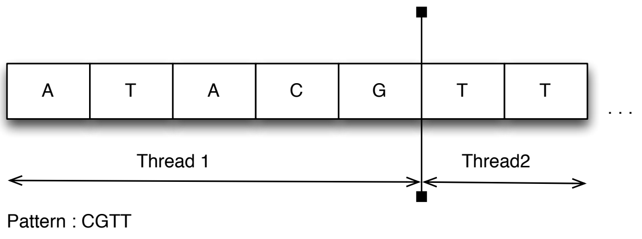

Fig. 4: Split Pattern over 2 Threads

The text, the pre-processed look-up table, and the patterns array were transferred in a single batched operation to the global memory to avoid multiple transfers between the host and the device. In order to permit multiple pattern searches, the algorithm was modified to support 2D arrays: each pattern requires one iteration through the input text. At each iteration one pattern and its corresponding look-up table were loaded into shared memory to allow the algorithm to access them in as few clock cycles as possible[19].

Each thread was then assigned to a specific part of the input text and processed a set of N_n of letters calculated as follows

$$\frac{N}{Total_{ThreadNumber}} + (M-1) = N_n$$

to avoid missing a pattern split over two parts of text as illustrated in Figure 4. Following the previous statements the complexity of the CPUs and GPUs based KMP can be evaluated as follows:

Sequential Version Let P_n be the number of patterns, m be the pattern length, and n be the text string length (e.g. *Yersinia Pestis*). The complexity is evaluated as follows :

$$P_n * O(m+n)$$

Parallelised Version Let B_n be the number of blocks and T_n the number of threads active in a block. The complexity of the GPUs version of the KMP algorithm can be calculated as follows :

$$P_n * O(m + \frac{n}{B_n * T_n})$$

The throughput performance of the algorithm can be calculated as follow :

$$\frac{8 * N}{Time_{gpu}} = Throughput$$

where $Time_{gpu}$ is the time elapsed during which the algorithm is running on the device, and $8 * N$ is the input length of the text in which the patterns are searched (bytes).

IV. Experimental Results

The experiments consisted of first modifying the KMP algorithm from a sequential version to a highly parallelised version, modifying its core and allowing it to search multiple patterns. Once modified the algorithm was then assessed by

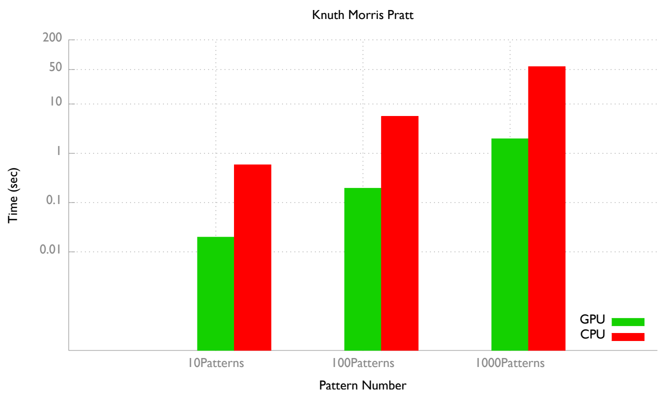

Fig. 5: Simulation Results

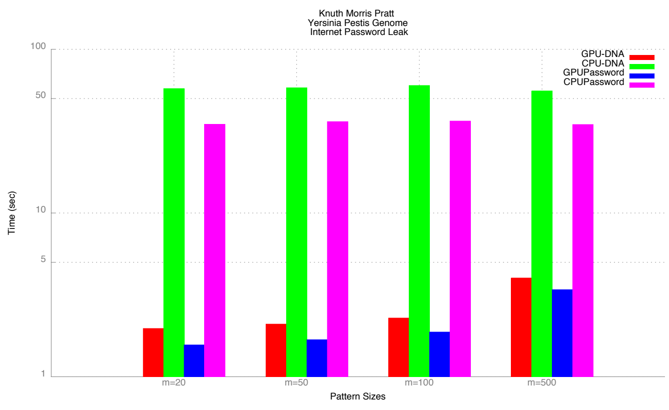

Fig. 7: Speedup over Multiple File Sizes and Σ Sizes.

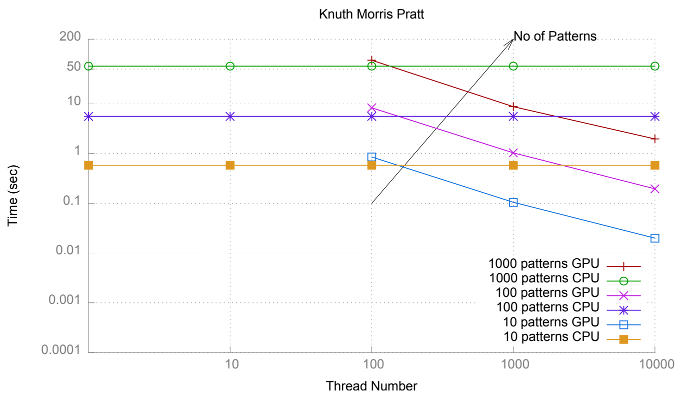

Fig. 6: Simulation Results over Multiple Pattern Sizes and Thread Number

measuring the overall execution time by searching for multiple patterns with varying pattern length and varying thread numbers. Multiple data files have also been used, with different (Σ) sizes.

The results indicate that GPUs have the potential to provide a performance improvement over CPUs by a factor of 29 for the KMP algorithm when using shared memory and loop unrolling. The loop unrolling permitted to optimise the overall kernel execution time by modifying the different *for loops* in the kernel. This also helped with the compiler's branch prediction while compiling the software. Also in accordance with the CUDA programming guide [19], the shared memory was used to store the patterns and the pre-processed look-up tables allowing access to each element with a minimum number at clock cycles and with a higher bandwidth than storing the patterns in global memory. This process is illustrated in Figure 3.

Figure 6 shows the linear improvement of the algorithm by modifying the number of patterns from $P_n = 10$ to $P_n = 1000$ and by modifying the number of threads from $T_n = 100$ to $T_n = 10000$ over the *Yersinia Pestis* genome file. The figure also demonstrates the importance of using a consequent amount of threads to significantly improve the performances over the serial version. It is also visible that, as the number of patterns increases, and the number of threads increases the improvements are linear. This occurs because the algorithm requires an iteration per pattern.

Figure 7 demonstrates the performance sensitivity of the KMP algorithm to different (Σ) and (m). Figure 7 reveals the algorithm's performance varies with pattern size from $m = 20$ to $m = 100$ but counter-intuitively, once the pattern size increases to $m = 500$ the performances are decreasing. This is a result of thread divergence, which occurs when multiple threads in the same warp (in this case a group of 32 threads) are executing different instructions simultaneously [18].

The loop unrolling technique allowed the multiple versions of the algorithm to gain an average improvement of 0.2 milliseconds over 500 hundred runs, using an alphabet of $m = 20$ and 10,000 threads. Although this performance improvement may seem insignificant, it might be important under a heavy network load or if a significant number of patterns need to be searched for.

Figure 5 illustrates GPUs and CPUs performance for various numbers of patterns (with fixed pattern size of $m = 20$ and 10,000 threads).

Table II shows that the algorithm performs sub-optimally compared to the algorithm *Parallel Failureless Aho-Corasick (PFAC)* proposed by *C-H. Lin et al.* [13] and the *Aho-Corasick* algorithm from *G. Vasiliadis et al.* [3] for Intrusion Detection Systems. The throughput achieved for a Alphabet $\Sigma = 4$, $T_n = 10000$ and $P_n = 1000$ nearly achieves 18.6 Mb/s and 20 Mb/s for an alphabet of $\Sigma = 26$. The performance of the KMP algorithm regarding the throughput is modest and therefore only appropriate for low bandwidth deep packet inspection systems or for digital forensic investigation tools. High speed network deep packet inspection will require further research.

V. Conclusion And Future Work

This paper presented a detailed discussion of Deep Packet Inspection and Intrusion Detection Systems on CPUs and GPUs. Their role and current limitations were also examined. In addition it was demonstrated that the current solutions are not adequate to exponentially growing network and provided a new approach for the protection of critical infrastructures.

TABLE II: Throughput Comparison

File	Threads	Pattern Number ($m = 20$)	Throughput (Gb/s) GPU	Throughput (Gb/s) CPU
DNA	10 000	1	19.827875	0.47744
DNA	10 000	10	1.98144	0.06703
DNA	10 000	100	0.18429	0.00618
DNA	10 000	1000	0.01824	0.000624
Password	10 000	1	18.5337	0.62786
Password	10 000	10	1.8932	0.08896
Password	10 000	100	0.19854	0.008957
Password	10 000	1000	0.019975	0.0007941

This paper presented a detailed parallel implementation of Knuth-Morris-Pratt and using loop unrolling, and shared memory using the CUDA 5 framework. The experimental results showed that the proposed algorithm improves data processing performance by a factor of 29 compared to the sequential version. The two different algorithms were also compared in terms of alphabet sizes, threads numbers and pattern sizes against time. This allowed an objective comparison of their performance [18] [3] [12]. It was also demonstrated that GPUs perform better as the number of threads is increased as this fully harnesses their processing power. Furthermore, large data batches are essential in avoiding transfer costs between the host and the device.

Future work will focus on the development of a more appropriate algorithm for a signature based DPI system using, dynamic parallelism, regular expressions, and state machines, allowing a greater flexibility and avoiding a linear time increase as shown in Figure 6. Different attack vectors will be analysed to allow the IDS to be as flexible as possible following the type of threat, as well as how to establish a strong and flexible way to transfer data from the Network Interface Card (NIC) operating in the kernel land to the user land where CUDA operates. Further research could also explore other interesting concepts such as cloud based DPI, Heuristic IDS and so forth but are for now out of the scope of this research.

Acknowledgment

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

References

[1] S. Manavski and G. Valle, “Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment,” *BMC Bioinformatics*, vol. 9, no. Suppl 2, p. S10, 2008. [Online]. Available: http://www.biomedcentral.com/1471-2105/9/S2/S10

[2] L. Marziale, G. G. R. Iii, and V. Roussev, “Massive threading: Using gpus to increase the performance of digital forensics tools.”

[3] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P, and S. Ioannidis, “Gnort: High performance network intrusion detection using graphics processors,” in *in Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, pp. 116–134.

[4] E. Elsen, P. LeGresley, and E. Darve, “Large calculation of the flow over a hypersonic vehicle using a {GPU},” *Journal of Computational Physics*, vol. 227, no. 24, pp. 10 148 – 10 161, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999108004476

[5] C. Vmel, S. Tomov, and J. Dongarra, “Divide and conquer on hybrid gpu-accelerated multicore systems,” *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. C70–C82, 2012. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/100806783

[6] E. Rustico, G. Bilotta, G. Gallo, A. Herault, and C. Del Negro, “Smoothed particle hydrodynamics simulations on multi-gpu systems,” in *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, 2012, pp. 384–391.

[7] H. Patel, “Gpu accelerated real time polarimetric image processing through the use of cuda,” in *Aerospace and Electronics Conference (NAECON), Proceedings of the IEEE 2010 National*, 2010, pp. 177–180.

[8] S. Pontarelli, C. Greco, E. Nobile, S. Teofili, and G. Bianchi, “Exploiting dynamic reconfiguration for fpga based network intrusion detection systems,” in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010, pp. 10–14.

[9] W. Jiang and M. Gokhale, “Real-time classification of multimedia traffic using fpga,” in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010, pp. 56–63.

[10] K. Benkrid, D. Crookes, J. Smith, and A. Benkrid, “High level programming for fpga based image and video processing using hardware skeletons,” in *Field-Programmable Custom Computing Machines, 2001. FCCM ’01. The 9th Annual IEEE Symposium on*, 2001, pp. 219–226.

[11] D. Hefenbrock, J. Oberg, N. T. N. Thanh, R. Kastner, and S. B. Baden, “Accelerating viola-jones face detection to fpga-level using gpus,” in *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 11–18. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2010.12

[12] C. Kouzinopoulos and K. Margaritis, “String matching on a multicore gpu using cuda,” in *Informatics, 2009. PCI ’09. 13th Panhellenic Conference on*, 2009, pp. 14–18.

[13] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, “Accelerating pattern matching using a novel parallel algorithm on gpus,” *IEEE Transactions on Computers*, vol. 99, no. PrePrints, p. 1, 2012.

[14] N. Jacob and C. Brodley, “Offloading ids computation to the gpu,” in *Computer Security Applications Conference, 2006. ACSAC ’06. 22nd Annual*, 2006, pp. 371–380.

[15] 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.

[16] H. Debar, M. Dacier, and A. Wespi, “Towards a taxonomy of intrusion-detection systems,” *Computer Networks*, vol. 31, no. 8, pp. 805 – 822, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128698000176

[17] R. Kemmerer and G. Vigna, “Intrusion detection: a brief history and overview,” *Computer*, vol. 35, no. 4, pp. 27–30, 2002.

[18] M. C. Schatz and C. Trapnell, “Fast exact string matching on the gpu.”

[19] Nvidia, “Cuda c programming guide,” 2012. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf