Trie Compression for GPU Accelerated Multi-Pattern Matching

Xavier Bellekens
Division of Computing and Mathematics
Abertay University
Dundee, Scotland
Email: x.bellekens@abertay.ac.uk

Amar Seeam
Department of Computer Science
Middlesex University
Mauritius Campus
Email: a.seeam@mdx.ac.uk

Christos Tachtatzis & Robert Atkinson
EEE Department
University of Strathclyde
Glasgow, Scotland
Email: name.surname@strath.ac.uk

Abstract—**Graphics Processing Units (GPU) allow for running massively parallel applications offloading the Central Processing Unit (CPU) from computationally intensive resources. However GPUs have a limited amount of memory. In this paper, a trie compression algorithm for massively parallel pattern matching is presented demonstrating 85% less space requirements than the original highly efficient parallel failure-less Aho-Corasick, whilst demonstrating over 22 Gbps throughput. The algorithm presented takes advantage of compressed row storage matrices as well as shared and texture memory on the GPU.**

***Keywords*–Pattern Matching Algorithm; Trie Compression; Searching; Data Compression; GPU**

I. Introduction

Pattern matching algorithms are used in a plethora of fields, ranging from bio-medical applications to cyber-security, the internet of things (IoT), DNA sequencing and anti-virus systems. The ever growing volume of data to be analysed, often in real time, demands high computational performance.

The massively parallel capabilities of Graphical Processor Units, have recently been exploited in numerous fields such as mathematics [1], physics [2], life sciences [3], computer science [4], networking [5], and astronomy [6] to increase the throughput of sequential algorithms and reduce the processing time.

With the increasing number of patterns to search for and the scarcity of memory on Graphics Processing Units data compression is important. The massively parallel capabilities allow for increasing the processing throughput and can benefit applications using string dictionaries [7], or application requiring large trees [8].

The remainder of this paper is organised as follows: Section II describes the GPU programming model, Section III provides background on multi-pattern matching algorithms while, Section IV discusses the failure-less Aho-Corasick algorithm used within this research. Section V highlights the design and implementation of the trie compression algorithms, while Section VI provides details on the environment. The results are highlighted in Section VII and the paper finishes with the Conclusion in Section VIII.

II. Background

A. GPU Programming Model

In this work, an Nvidia 1080 GPUs is used along with the Compute Unified Device Architecture (CUDA) programming model, allowing for a rich Software Development Kit (SDK). Using the CUDA SDK, researchers are able to communicate with GPUs using a variety of programming languages. The C language has been extended with primitives, libraries and compiler directives in order for software developers to be able to request and store data on GPUs for processing.

GPUs are composed of numerous Streaming Multiprocessors (SM) operating in a Single Instruction Multiple Thread (SIMT) fashion. SMs are themselves composed of numerous CUDA cores, also known as Streaming Processors (SP).

B. Multi-Pattern Matching

Pattern matching is the art of searching for a pattern P in a text T. Multi-pattern matching algorithms are used in a plethora of domains ranging from cyber-security to biology and engineering [9].

The Aho-Corasick algorithm is one of the most widely used algorithms [10][11]. The algorithm allows the matching of multiple patterns in a single pass over a text. This is achieved by using the failure links created during the construction phase of the algorithm.

The Aho-Corasick, however, presents a major drawback when parallelised, as some patterns may be split over two different chunks of T. Each thread is required to overlap the next chunk of data by the length of the longest pattern -1. This drawback was described in [12] and [13].

C. Parallel Failure-Less Aho-Corasick

Lin *et al.* presented an alternative method for multi-pattern matching on GPU in [14].

To overcome these problems, Lin *et al.* presented the failure-less Aho-Corasick algorithm. Each thread is assigned to a single letter in the text T. If a match is recorded, the thread continues the matching process until a mismatch. When a mismatch occurs the thread is terminated, releasing GPU

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-534-0

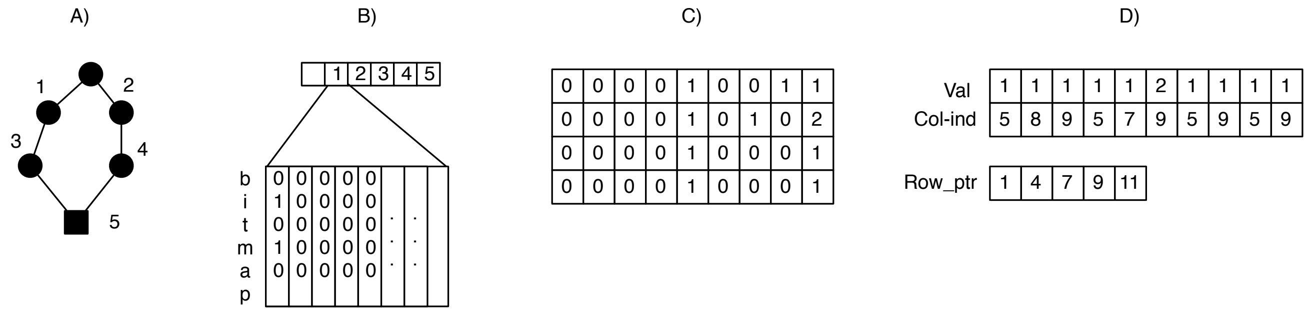

Figure 1. Memory layout transformation for a simplified transfer between the host and the device, allowing for better compression and improved throughput

resources. The algorithm also allows for coalesced memory access during the first memory transfer, and early thread termination.

III. Design and Implementation

The trie compression library presented within this section builds upon prior research presented in [15] and [16] and aims to further reduce the memory footprint of the highly-efficient parallel failure-less Aho-Corasick (HEPFAC) trie presented in [16], while improving upon the tradeoff between memory compression operation and the throughput offered by the massively parallel capabilities of GPUs.

The compressed trie presented in our prior research is created in six distinct steps. I) The trie is constructed in a breadth-first approach, level by level. II) The trie is stored in a row major ordered array. III) The trie is truncated at the appropriate level, as described in [15]. IV) Similar suffixes are merged together on the last three levels of the trie (This may vary based on the alphabet in use). V) The last nodes are merged together. VI) The row major ordering array is translated into a sparse matrix as described in [16].

In this manuscript, an additional step is added. The sparse matrix representing the trie is compressed using a Compressed Row Storage (CRS) algorithm, reducing furthermore the memory footprint of the bitmap array [17][18]. The compressed row storage also allows the array to be stored in texture memory, hence benefiting from cached data. The row pointer generated by the CRS algorithm is stored in shared memory, benefiting from both the cache and an on-chip location reducing the clock cycles when accessing data.

Figure 2 is a visual representation of steps I to V undertaken during the construction of the trie. As shown, the trie is truncated to an appropriate level. This technique was used by Vasiliadis *et al.* [19] and further studied by Bellekens *et al.* [16]. After truncation, similar suffixes within the trie are merged together and the leave nodes are replaced by a single end node.
Figure 3 shows the composition of the nodes in the trie. Each node is composed of a bitmap of 256 bits from the ASCII alphabet. The bitmap is modular and can be modified based on the trie requirements (e.g., for DNA storage). Each node also contains an offset value providing the location of the first child of the current node. Subsequent children can be located following the method described in [16].

Figure 1 depicts four different memory layouts in order to achieve better compression and increase the throughput on GPUs. Figure 1 (A) represents the trie created with a linked list. Figure 1 (B) represents the trie organised in a row major order, this allows the removal of pointers and simplifies the transition between the host and and the device memory. Figure 1 (C) represents the trie in a two dimensional array, allowing the trie to be stored in Texture memory on the GPU and annihilate the trade-off between the compression highlighted in Figure 1 A) and the throughput. Finally, Figure 1 (D) is improving upon the compression of our prior research while allowing the trie to be stored in texture memory and the row_ptr to be stored in shared memory.

The CRS compression of the non-symmetric sparse matrix A is achieved by creating three vectors. The val vector stores the values of the non-zero elements present within A, while the $col-ind$ store the indexes of the val. The storage savings for this approach is defined as $2nnz + n + 1$, where nnz represents the number of non-zero elements in the matrix and n the number of elements per side of the matrix. In the example provided in Figure 1 (C and D), the sparse matrix is

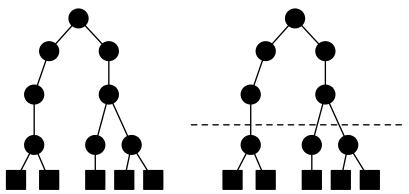

Figure 2. Trie Truncation and Node Compression

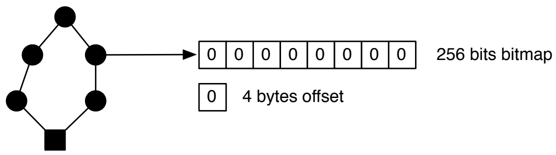

Figure 3. Bitmapped Nodes

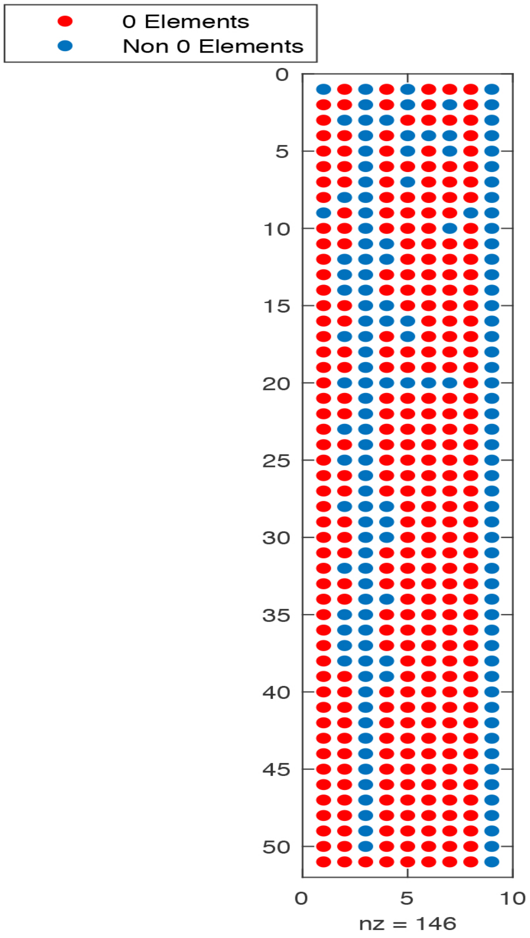

Figure 4. Sparse Matrix Representation of the a Compressed Trie Containing 10 Patterns

reduced from 36 to 24 elements.

The sparse matrix compression combined with the trie compression and the bitmap allows for storing large numbers of patters on GPUs allowing its use in big data applications, anti-virus and intrusion detection systems. Note that the CRS compression is pattern dependent, hence the compression will vary with the alphabet in use and the type of patterns being searched for.

IV. Experimental Environment

The Nvidia 1080 GPU is composed of 2560 CUDA cores divided into 20 SMs. The card also contains 8 GB of GDDR5X with a clock speed of 1733 MHz. Moreover the card possesses 96 KB shared memory and 48 KB of L1 cache, as well as 8 texture units and a GP104 PolyMorph engine used for vertex fetches for each streaming multiprocessors. The base system is composed of 2 Intel Xeon Processors with 20 cores, allowing up to 40 threads and has a total of 32GB of RAM. The server is running Ubuntu Server 14.04 with the latest version of CUDA 8.0 and C99.

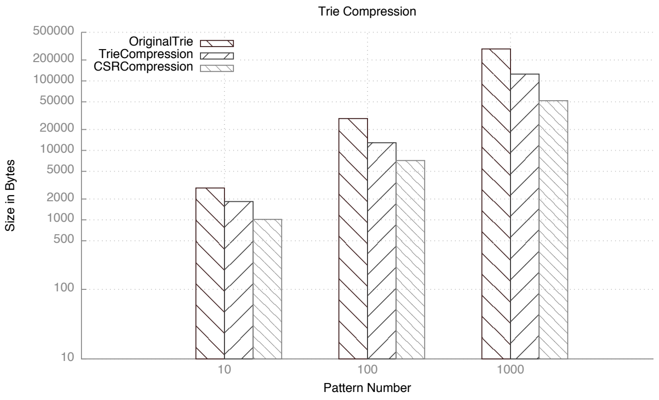

Figure 5. Comparison between the current state of the art and the CRS Trie

V. Results

The HEPFAC algorithm presented within this manuscript improves upon the state of the art compression and uses texture memory and shared memory to increase the matching throughput.

The evaluation of compression algorithm presented is made against the King James Bible. The patterns are chosen randomly within the text using the Mersenne Twister algorithm [20].

Figure 4 is a representation of the compressed trie stored in a 2D layout. The trie contains ten Patterns. The blue elements represent non-zero elements in the matrix while the red elements represent empty spaces within the matrix. The last column of the matrix only contains the offsets of each node.

Figure 5 demonstrates the compression achieved by the different compression steps aforementioned. The original trie required a total of 36 bytes for each nodes, 256 bits to represent the ASCII alphabet and four bytes for the offset. The trie compression, on the other hand requires 36 bytes for each node but reduces the size of the trie based on the

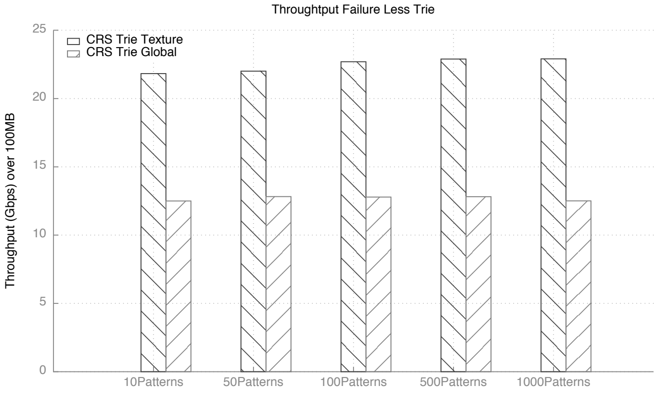

Figure 6. Throughput Comparison Between Global Memory and Texture Memory

ISBN: 978-1-61208-534-0

alphabet used (in this case to eight levels), then merges similar suffixes together and merges all final nodes in a single one. Finally, the CRS compression algorithm compresses the sparse matrix representation of the trie. This technique allows an 83% space reduction in comparison to the original trie and a 56% reduction in comparison to the trie compression algorithm presented in [15].

Figure 6 depicts the throughput obtained when storing the CRS trie in global memory and in Texture memory. Global Memory does not provide access to a cache and requires up to 600 clock cycles to access data. This inherently limits the throughput of the pattern matching algorithm to 12 Gbps. When the CRS trie is stored in texture memory and the row_ptr is stored in shared memory the algorithm demonstrate 22 Gbps throughput when matching a 1000 patterns within an alphabet $\Sigma = 256$.

VI. Conclusion

In this work a trie compression algorithm is presented. The trie compression scheme improves upon the state of the art and demonstrates 83% space reduction against the original trie compression and 56% reduction over the HEPFAC algorithm. Moreover, our approach also demonstrates over 22 Gbps throughput while matching a 1000 patterns. This work highlighted the algorithm on single GPU node, however, the algorithm can be adapted to cloud computing, or on FPGAs.

References

[1] A. Nasridinov, Y. Lee, and Y.-H. Park, “Decision tree construction on gpu: ubiquitous parallel computing approach,” Computing, vol. 96, no. 5, 2014, pp. 403–413.

[2] N. Nakasato, “Oct-tree method on gpu,” arXiv preprint arXiv:0909.0541, 2009.

[3] S. Memeti and S. Pllana, “Combinatorial optimization of work distribution on heterogeneous systems,” in 2016 45th International Conference on Parallel Processing Workshops (ICPPW), Aug 2016, pp. 151–160.

[4] D. Aparicio, P. Paredes, and P. Ribeiro, “A scalable parallel approach for subgraph census computation,” in European Conference on Parallel Processing. Springer International Publishing, 2014, pp. 194–205.

[5] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, “Compressing ip forwarding tables: towards entropy bounds and beyond,” in ACM SIGCOMM Computer Communication Review, vol. 43, no. 4. ACM, 2013, pp. 111–122.

[6] J. Bédorf, E. Gaburov, and S. P. Zwart, “Bonsai: A gpu tree-code,” arXiv preprint arXiv:1204.2280, 2012.

[7] M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro, “Practical compressed string dictionaries,” Information Systems, vol. 56, 2016, pp. 73–108.

[8] G. Navarro and A. O. Pereira, “Faster compressed suffix trees for repetitive collections,” Journal of Experimental Algorithmics (JEA), vol. 21, no. 1, 2016, pp. 1–8.

[9] T. T. Tran, M. Giraud, and J.-S. Varré, “Bit-parallel multiple pattern matching,” in International Conference on Parallel Processing and Applied Mathematics. Springer, 2011, pp. 292–301.

[10] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” Commun. ACM, vol. 18, no. 6, Jun. 1975, pp. 333–340. [Online]. Available: http://doi.acm.org/10.1145/360825.360855

[11] S. Vakili, J. Langlois, B. Boughzala, and Y. Savaria, “Memory-efficient string matching for intrusion detection systems using a high-precision pattern grouping algorithm,” in Proceedings of the 2016 symposium on architectures for networking and communications systems. ACM, 2016, pp. 37–42.

[12] 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.

[13] A. Tumeo, O. Villa, and D. Sciuto, “Efficient pattern matching on gpus for intrusion detection systems,” in Proceedings of the 7th ACM international conference on Computing frontiers. ACM, 2010, pp. 87–88.

[14] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, and J.-M. Shyu, “Accelerating string matching using multi-threaded algorithm on GPU,” in Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE, Dec 2010, pp. 1–5.

[15] X. J. A. Bellekens, C. Tachtatzis, R. C. Atkinson, C. Renfrew, and T. Kirkham, “A highly-efficient memory-compression scheme for gpu-accelerated intrusion detection systems,” in Proceedings of the 7th International Conference on Security of Information and Networks, ser. SIN ’14. New York, NY, USA: ACM, 2014, pp. 302:302–302:309. [Online]. Available: http://doi.acm.org/10.1145/2659651.2659723

[16] X. J. A. Bellekens, “High performance pattern matching and data remanence on graphics processing units,” Ph.D. dissertation, University of Strathclyde, 2016. [Online]. Available: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.698532

[17] K. Wu, E. J. Otoo, and A. Shoshani, “Optimizing bitmap indices with efficient compression,” ACM Transactions on Database Systems (TODS), vol. 31, no. 1, 2006, pp. 1–38.

[18] H. Wang, “Sparse array representations and some selected array operations on gpus,” Master’s thesis, School of Computer Science University of the Witwatersrand A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, 2014.

[19] G. Vasiliadis and S. Ioannidis, “Gravity: a massively parallel antivirus engine,” in International Workshop on Recent Advances in Intrusion Detection. Springer, 2010, pp. 79–96.

[20] M. Matsumoto and T. Nishimura, “Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator,” ACM Trans. Model. Comput. Simul., vol. 8, no. 1, Jan. 1998, pp. 3–30. [Online]. Available: http://doi.acm.org/10.1145/272991.272995