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

 ISBN: 978-1-61208-534-0

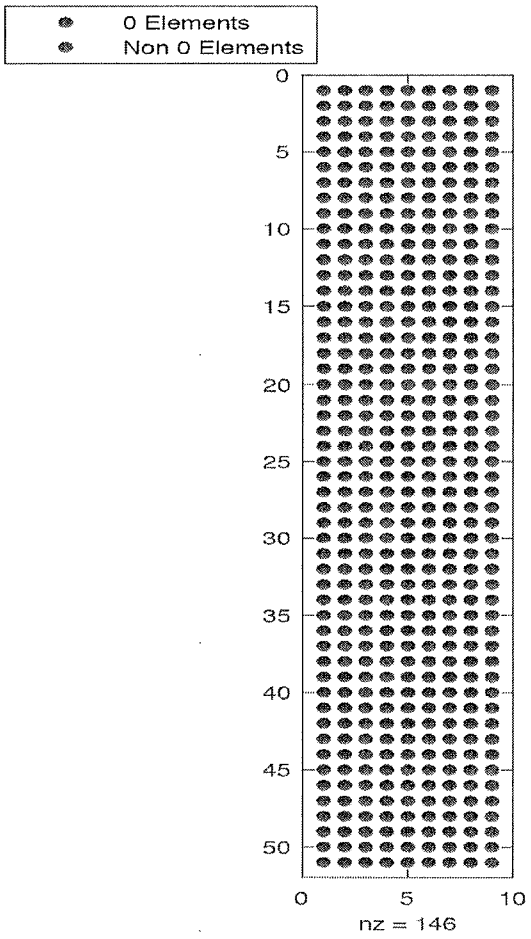

Figure 4. Sparse Matrix Representation of the a Compressed Trie Containing 10 Patterns

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.

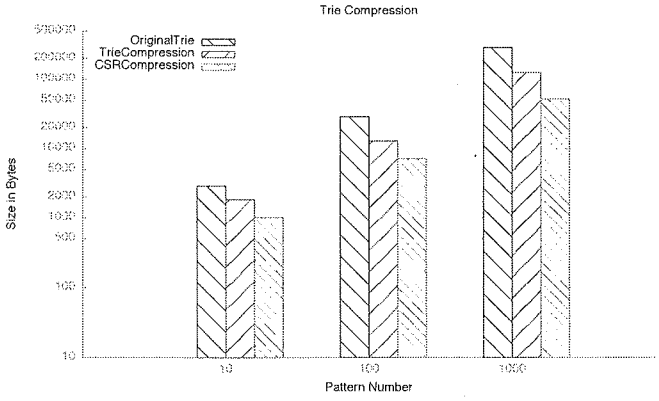

Figure 5. Comparison between the current state of the art and the CRS Trie

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

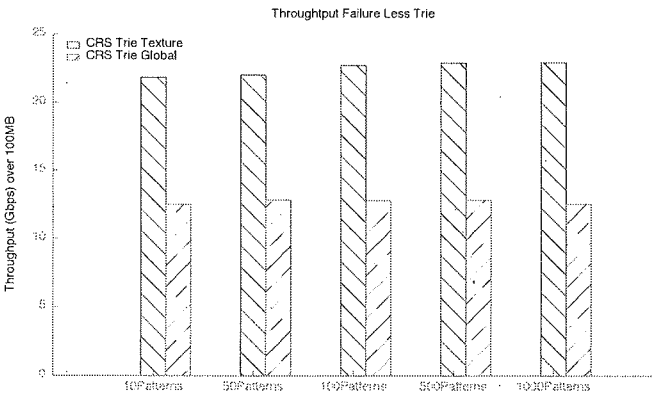

Figure 6. Throughput Comparison Between Global Memory and Texture Memory

 ISBN: 978-1-61208-534-0