String Matching on a multicore GPU using CUDA

Charalampos S. Kouzinopoulos and Konstantinos G. Margaritis
Parallel and Distributed Processing Laboratory
Department of Applied Informatics, University of Macedonia
156 Egnatia str., P.O. Box 1591, 54006 Thessaloniki, Greece
E-mail: {ckouz,kmarg}@uom.gr

***Abstract*—Graphics Processing Units (GPUs) have evolved over the past few years from dedicated graphics rendering devices to powerful parallel processors, outperforming traditional Central Processing Units (CPUs) in many areas of scientific computing. The use of GPUs as processing elements was very limited until recently, when the concept of General-Purpose computing on Graphics Processing Units (GPGPU) was introduced. GPGPU made possible to exploit the processing power and the memory bandwidth of the GPUs with the use of APIs that hide the GPU hardware from programmers. This paper presents experimental results on the parallel processing for some well known on-line string matching algorithms using one such GPU abstraction API, the Compute Unified Device Architecture (CUDA).**

***Keywords*-string matching, algorithms, CUDA, GPGPU, parallel and distributed computing, stream processing**

I. Introduction

Graphics Processing Units (GPUs) are graphics rendering devices introduced back in the 1980s to offload graphics-related processing from Central Processing Units (CPUs). Initially, GPUs were using two fixed-functioned processor classes to process data, vertex transform and lighting processors that allowed the programmer to alter per-vertex attributes and pixel-fragment processors that were used to calculate the colour of a fragment. Although this model worked perfectly for graphics processing, offered little to no flexibility for general purpose programming. The architecture of the GPUs evolved in 2001, when for the first time programmers were given the opportunity to program individual kernels in the graphics pipeline by using programmable vertex and fragment shaders [13]. Later GPUs (as the G80 series by NVIDIA) substituted the vertex and fragment processors with a unified set of generic processors that can be used as either vertex or fragment processors depending on the programming needs [15]. On each new generation, additional features are introduced that move the GPUs one step closer to wider use for general purpose computations. The use of a GPU instead of a CPU to perform general purpose computations is known as General Purpose computing on Graphics Processing Units (GPGPU).

To hide all the specific hardware details of the graphics cards from the programmers, a number of high level languages and APIs were introduced such as Cg [8], BrookGPU [20], Sh library [21] and Stream [22]. One such programming API for parallel computing using GPUs is the Compute Unified Device Architecture (CUDA) [19]. CUDA was released in 2007 enabling programmers to write code for execution on GPUs and to use the available thread processors of a GPU (also known as shaders in Graphics terminology) without the need to write threaded code.

Unlike CPUs that are optimized for use on sequential code, all commodity GPUs follow a streaming, data parallel programming model resembling SPMD (single-program multiple-data),. This model is structured around a graphics pipeline that consists of a number of computation stages that vertices have to pass to undergo transformation and lighting (including Vertex transformations, Rasterization, Fragmentation and Composition). These computation stages are connected together with a highly localized data flow, since the data that is produced by one stage is used in its entirety by the next one [13] and each stage is separately implemented in hardware, allowing highly efficient parallel computations. Key to the stream model is the notion that all data is represented as an ordered set of data of the same data type or in other words as a stream. A stream application consists of multiple functions (called kernels in CUDA) and each of these functions can be mapped on a different physical computation stage of the graphics pipeline.

GPUs have a parallel multi-core architecture, with each core containing thread processors that are capable of running hundreds of threads simultaneously. The threads are grouped into a number of blocks that in turn are grouped into grids. The threads in each block can be synchronized and exchange information using shared memory. All the blocks that belong to a grid can be executed in parallel while a hardware thread manager is responsible to manage threads automatically at runtime. The programmer can specify the number of blocks as well as the number of threads per block depending on the capabilities of each GPU with the CUDA compiler determining the optimal number of registers per thread to be used.

The idea of using a GPU to improve the performance of existing algorithms is not new. Since the release of CUDA, several researchers have used it across many areas of computing including applied mathematics [6], sequence alignment [9], astrophysical simulations [12] and image processing [17]. The use of GPUs to perform string matching computations has also been studied: the Cg programming language is used in [5] to offload to the GPU an Intrusion Detection System that uses a simplified version of the Knuth-Morris-Pratt on-line string matching algorithm, reporting a marginal improvement; the Aho-Corasick, Knuth-Morris-Pratt and Boyer-Moore algorithms were implemented in [18] using CUDA to perform Intrusion Detection on a set of synthetic payload trace and real network traffic with a reported increase of their performance by up to an order of magnitude; a speedup of up to 35x was achieved in [15] when executing the Cmatch algorithm in parallel to perform string matching on sets of genome and chromosome sequences. A detailed survey of many scientific applications that use the GPGPU model can be found in [14].

In this paper, the performance of the Naive, Knuth-Morris-Pratt, Boyer-Moore-Horspool and Quick-Search on-line exact string matching algorithms is evaluated when implemented using CUDA and executed on a state of the art Graphic Processor Unit with a focus on the way the level of the GPU utilization and the shared memory usage affects their performance. The Naive, Boyer-Moore-Horspool and Quick-Search algorithms, algorithms that to the best of our knowledge have never been implemented before for string matching using the CUDA API, used to locate all the appearances of a pattern on a set of reference DNA sequences and their running time when executed on the GPU was measured along that of the Knuth-Morris-Pratt algorithm. The algorithms were chosen as they involve sequential accesses to the memory in order to locate all the appearances of a pattern, an essential step to achieve peak performance when executed on a GPU as detailed in [2].

II. String matching

String matching is an important problem in text processing and is commonly used to locate the appearance of one dimensional arrays (the so-called pattern) in an array of equal or larger size (the so-called text). The string matching problem can be defined as: let Σ be an alphabet, given a text array T[n] and a pattern array P[m], report all locations [i] in T where there is an occurrence of P, i.e. T[i + k] = P[k] for m $\leq$ n.

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 [7] 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.

The Boyer-Moore-Horspool [4] algorithm uses a window of size m to search the text from right to left and a precomputed array (known as the bad-character shift) of the rightmost character of the window to skip character positions when a mismatch occurs.

Finally, the Quick-Search [16] algorithm performs character comparisons from left to right from the leftmost pattern character and in case of mismatch it computes the shift with the occurrence heuristic for the first text character after the last pattern character by the time of mismatch. A more detailed description of the algorithms can be found in [10].

III. Experimental Methodology

The experiments were executed locally on an Intel Xeon CPU with a 2.40GHz clock speed and 2 Gb of memory. The Graphics Processing Unit used was the NVIDIA GTX 280 with 1GB of GDDR3 global memory, 30 multiprocessors and 240 cores, enabling the use of a maximum number of 1024 active threads per multiprocessor for a total of 30720 active threads. The operating system used was a Scientific Linux and during the experiments only the typical background processes ran. To decrease random variation, the time results were averages of 100 runs.

The algorithms presented in the previous section were implemented using the ANSI C programming language and were compiled using Nvidia's nvcc compiler without any extra optimization flags. The implementation of each algorithm presented in this paper used 10 registers per thread to execute out of the 16384 registers per block that the GTX 280 provides. The number of registers per thread is a limiting factor on the number of concurrent threads the hardware is capable of executing [3]. The utilization of all 512 threads per block was achieved by ensuring that only a minimum amount of registers was used for each algorithm.

The data set and the experimental methodology was similar to the one used on [15]. It consisted of 3 reference sequences: the bacterial genomes of Yersinia pestis (4.6 Mb in length) and Bacillus anthracis (5.2 Mb in length) as well as a simulated BAC built of the first 200.000 characters of NCBI build 26 of Homo sapiens' chromosome 2. The query pattern was constructed of randomly chosen subsequences from each reference sequence with a length of 25, 50, 200 and 800 characters.

To calculate the speedup of the string matching algorithms, the practical running time was used as a measure. Practical running time is the total time in seconds an algorithm needs to find all occurrences of a pattern in a text including any preprocessing time and any transfer time between the host and the GPU and was measured using the timer functions of the cutil toolkit. All data was transferred in one batch to the global memory of the GPU and the pattern as well as any lookup tables the algorithms were using for the preprocessing were copied to the shared memory area of each multiprocessor.

IV. Experimental Results

Figure 1 presents the speedup achieved by the parallel execution of the Naive, Knuth-Morris-Pratt, Boyer-Moore-Horspool and Quick-Search string matching algorithms on a GPU for the Y.pestis, B. anthracis and BAC reference sequences and for different pattern sizes.

The parallel implementation of the Naive had an impressive performance increase comparing to the serial version when used on the Y. pestis and B. anthracis reference sequences, achieving a speedup of up to 24x for pattern sizes between m=25 and m=200. For m=800, the performance increase of the Naive algorithm ranged between 16x and 18x. On the same reference sequences, the Knuth-Morris-Pratt, Boyer-Moore-Horspool and Quick-Search algorithms achieved up to 18x, 20x and 19x speedup respectively when patterns of size m=25 and m=50 were used. Finally, for larger pattern sizes, the performance increase ranged between 5x and 12x for KMP, 2x and 15x for the Bmh and between 1.5x and 4.5x for the Qs algorithm. According to [15], the performance decrease when larger pattern sizes are used could be due to thread divergence, the serial execution of threads within the same warp that execute different instructions as a result of if and while statements in the kernel.

When used to locate all the positions in the BAC reference sequence where the pattern appears, the parallel implementation of the Naive algorithm had a similar performance increase as the one reported on the Y. pestis and B. anthracis reference sequences, ranging between 14x and 21x. The performance of the implementation of the Knuth-Morris-Pratt, Boyer-Moore-Horspool and Quick-Search algorithms was between 1.2x and 7.5x when patterns of size m=25 and m=50 were used and actually slower than the serial implementation for larger pattern sizes. The cost of initializing the kernel and the necessary structures for preprocessing a data set of not sufficient size together with the problem of thread divergence seems to produce inefficient code in some cases.

The CUDA programming guide states that the shared memory, a small high-bandwidth memory area that is private to the threads of each block and can be up to 150 times faster than the global memory, should be used in order to achieve better performance when executing parallel code. Storing frequently reused data to the shared memory can deliver substantial performance improvements [11]. The size of the shared memory is limited to 16KB per multiprocessor, and was used to fit the pattern as well as the lookup tables of the Knuth-Morris-Pratt, Boyer-Moore-Horspool and Quick-Search algorithms.

As can be seen on Figure 2, by using the per-block shared memory of the GPU comparing to using the global memory to store data, the parallel implementation of the algorithms was 2.5x to 24x faster depending on the algorithm. More specifically, the Boyer-Moore-Horspool benefited most from the use of the shared memory with a 24x speedup comparing to only using the global memory, the Quick-Search algorithm had a 13x speedup and the Knuth-Morris-Pratt had a 7x speed. Naive had a speedup of just 2.5x since is the only algorithm that is not using a lookup table.

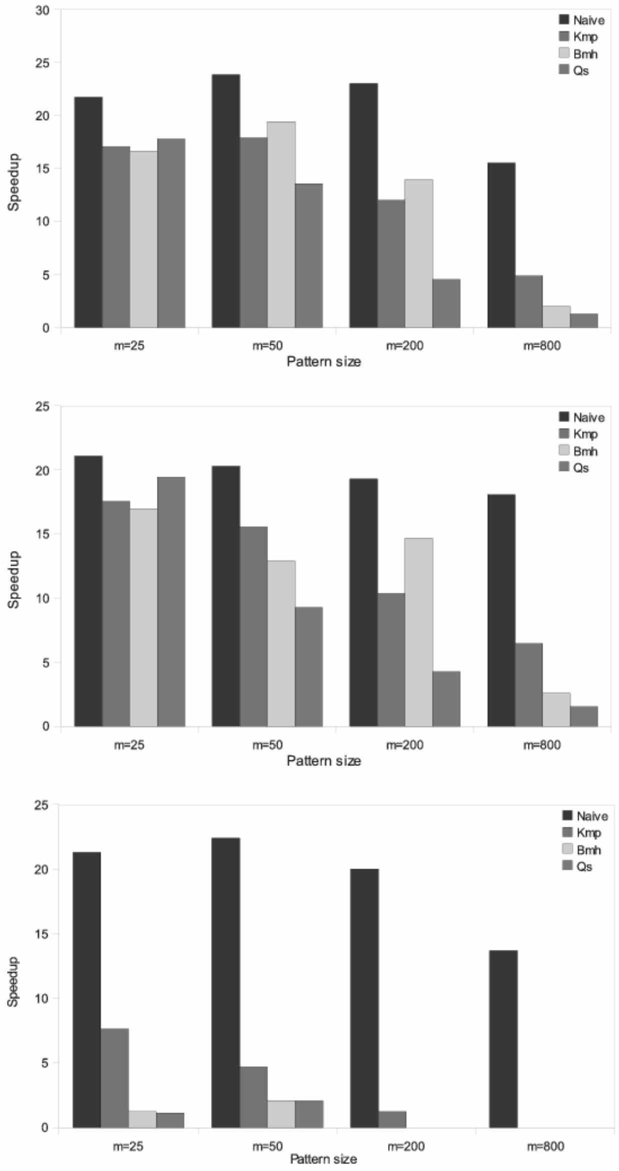

Figure 1. Speedup for the Y. pestis (a), B. anthracis (b) and BAC (c) reference sequences

Figure 3 depicts the way the performance of the Naive, Knuth-Morris-Pratt, Boyer-Moore-Horspool and Quick-Search algorithms is affected by the number of threads for the Y. pestis reference sequence and for a pattern size of

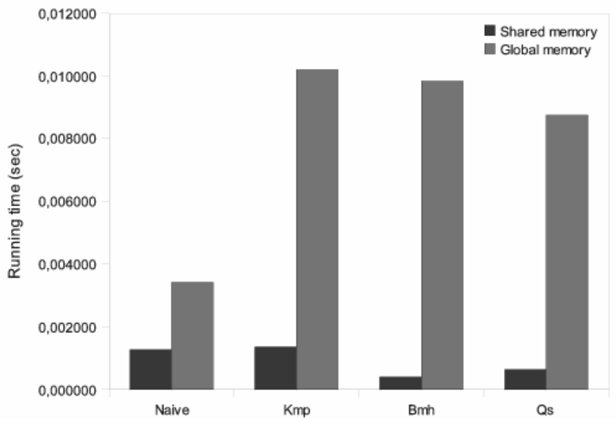

Figure 2. Speed improvement when using shared memory

m=25. The horizontal line depicts the performance of the algorithms when executed on the CPU while the curved line presents in a logarithmic scale the performance of the GPU implementation of the algorithms for a varied amount of threads.

The performance of a single thread processor is modest comparing to the performance of a CPU, even though the algorithms are designed to perform sequential accesses to the memory. The GPU starts to outperform the CPU when more than one block of 256 or more threads process the data allowing the hardware to switch between the blocks to hide the latency of the access to the global memory. From the same Figure it can also be concluded that peak performance can be achieved by using the maximum possible amount of threads in order to keep the GPU fully utilized.

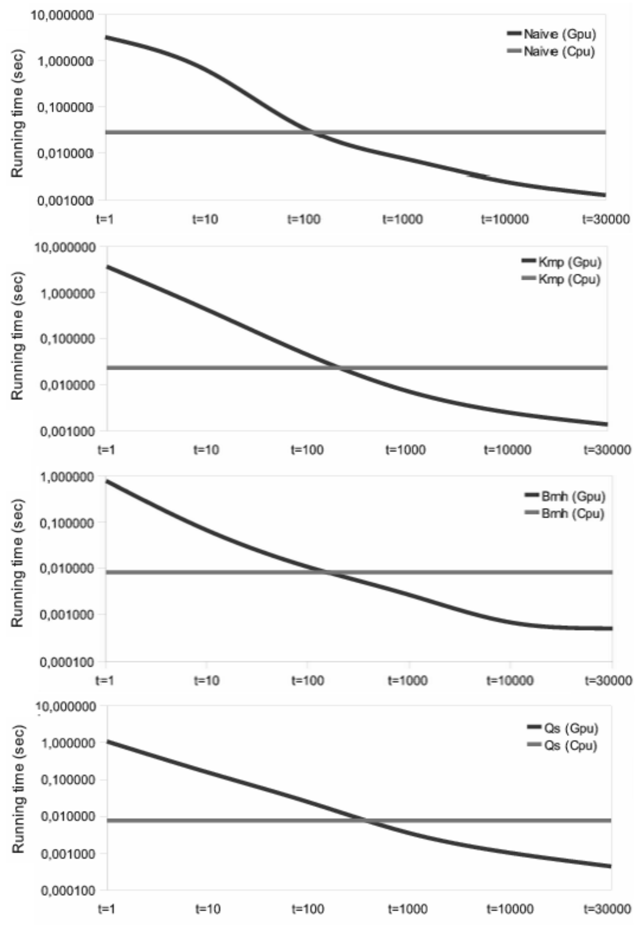

Figure 3. Running time of the Naive (a), KMP (b), Bmh (c) and Qs (d) algorithms for a varied amount of threads (t)

V. Conclusion

In this paper, parallel implementations were presented of the Naive, Knuth-Morris-Pratt, Boyer-Moore-Horspool and the Quick-Search on-line exact string matching algorithms using the CUDA toolkit. Both the serial and the parallel implementations were compared in terms of running time for different reference sequences, pattern sizes and number of threads. It was shown that the parallel implementation of the algorithms was up to 24x faster than the serial implementation, especially when larger text and smaller pattern sizes where used. The performance achieved is close to the one reported for similar string matching algorithms in [15] and [18]. In addition, it was discussed that in order to achieve peak performance on a GPU, the hardware must be as utilized as possible and the shared memory should be used to take advantage of its very low latency.

Future research in the area of string matching and GPGPU parallel processing could focus on the performance study of the parallel implementation of additional categories of string matching algorithms, including approximate and two dimensional string matching. Moreover, it would be interesting to examine the performance of the algorithms when executed on multiple GPUs and on hybrid mpi and cuda clusters. Finally, further optimization of the parallel implementation of the algorithms could be considered to make better use of the GPUs capabilities, including loop unrolling, matrix reordering and register blocking [1] in addition to smarter use of the shared memory.

References

[1] L. Buatois, G. Caumon and B. Lvy, *Concurrent Number Cruncher : An Efficient Sparse Linear Solver on the GPU*. In proceedings of the Third International Conference on High Performance Computing and Communications, 26-28, 2007.

[2] I. Buck, *Taking the Plunge into GPU Computing*. In GPU Gems 2. M. Pharr, Ed. Addison-Wesley, 509-519, 2005. ISBN: 0-321-33559-7

[3] T. R. Halfhill, *Parallel Processing with CUDA*. Microprocessor Journal, 2008.

[4] R. N. Horspool, *Practical fast searching in strings*. Software - Practice and Experience, 10(6): 501-506, 1980.

[5] N. Jacob and C. Brodley, *Offloading IDS Computation to the GPU*. Proceedings of the 22nd Annual Computer Security Applications Conference, 371-380, 2006.

[6] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith and J. Manferdelli, *High performance discrete Fourier transforms on graphics processors*. In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008.

[7] D. E. Knuth, J. H. Morris and V. R. Pratt, *Fast pattern matching in strings*. SIAM Journal on Computing. 6(1): 323-350, 1977.

[8] W. R. Mark, R. S. Glanville, K. Akeley and M. J. Kilgard, *Cg: A System for Programming Graphics Hardware in a C-like Language*. In Proceedings of the 2003 ACM SIGGRAPH, 896-907, 2003

[9] S. Manavski and G. Valle, *CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment*. BMC Bioinformatics, 9(2), 2008.

[10] P. D. Michailidis and K. G. Margaritis, *On-line String Matching Algorithms: Survey and Experimental Results*. International Journal of Computer Mathematics, 76(4): 411-434, 2001.

[11] J. Nickolls, I. Buck, M. Garland and K. Skadron, *Scalable parallel programming with CUDA*. ACM Queue, 6(2): 40-53, 2008

[12] L. Nyland, M. Harris and J. Prins *Fast N-Body Simulation with CUDA*. In GPU Gems 3. H. Nguyen, Ed. Addison-Wesley, 677-695, 2007. ISBN: 0-321-51526-9

[13] J. D. Owens, *Streaming Architectures and Technology Trends*. In GPU Gems 2. M. Pharr, Ed. Addison-Wesley, 457-470, 2005. ISBN: 0-321-33559-7

[14] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn and T. J. Purcell, *A survey of general-purpose computation on graphics hardware*. In Proceedings of the 2007 Computer Graphics Forum, 26(1): 80-113, 2007.

[15] M. C. Schatz and C. Trapnell, *Fast Exact String Matching on the GPU*.

[16] D. M. Sunday, *A very fast substring search algorithm*. Communications of the ACM, 33(8): 132-142, 1990.

[17] Maraike Schellmann, Jurgen Vording, Sergei Gorlatch and Dominik Meilander, *Cost-effective medical image reconstruction: from clusters to graphics processing units*. In Proceedings of the 2008 conference on Computing frontiers, 2008.

[18] G. Vasiliadis, S. Antonatos, M. Polychronakis, E.P. Markatos and S. Ioannidis, *Gnort: High Performance Network Intrusion Detection Using Graphics Processors*. Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection, 116 - 134, 2008.

[19] http://www.nvidia.com/object/cuda_home.html *CUDA Zone*.

[20] http://graphics.stanford.edu/projects/brookgpu *BrookGPU compiler and runtime implementation*.

[21] http://libsh.org/ *Sh library*.

[22] http://ati.amd.com/technology/streamcomputing/ *AMD Stream Computing*.