Neural, Parallel, and Scientific Computations 25 (2017) 165-180

PARALLEL TWO-DIMENSIONAL PATTERN MATCHING ALGORITHMS BASED ON GPU

CHARALAMPOS S. KOUZINOPOULOS, PANAGIOTIS D. MICHAILIDIS,
AND KONSTANTINOS G. MARGARITIS

CERN, Switzerland
Department of Balkan, Slavic and Oriental Studies, University of Macedonia,
156 Egnatia Str., 54636 Thessaloniki, Greece
Department of Applied Informatics, University of Macedonia,
156 Egnatia Str., 54636 Thessaloniki, Greece

ABSTRACT. The two-dimensional pattern matching problem is used to locate all the positions inside a two-dimensional text where a two-dimensional pattern of a smaller or equal size occurs. In this paper, we present a basic parallel implementation and different optimization techniques of two important two-dimensional pattern matching algorithms, Baker and Bird and Baeza-Yates and Regnier, on a Graphics Processing Unit (GPU) platform. The performance of the proposed parallel implementations is evaluated for the GPU architecture and for different problem parameters. Experimental results demonstrate that there is performance gain of the optimized GPU implementations over the unoptimized implementation. Additionally, it is shown that the parallel implementation of Baeza-Yates and Regnier has a significantly higher performance gain comparing to that of the Baker and Bird algorithm.

Key Words. Two-dimensional pattern matching, Parallel computing, GPU, CUDA.

1. Introduction

String matching is an important problem in text processing and is commonly used to locate one dimensional patterns (strings) on texts. Although strings are usually stored in linear files, their interpretation can be very heterogeneous. In many situations arising in digital data processing we encounter strings of symbols organized into multidimensional structures, such as two-dimensional. These structures are in fact very common, almost every computer user has encountered examples of such structures: images. The usage of multidimensional structures is widespread, with a very wide spectrum of applications ranging from computer vision to computational biology [12, 9, 2, 27, 21]. The two-dimensional pattern matching problem consists of taking two rectangular arrays, an $m_1 \times m_2$ pattern array and an $n_1 \times n_2$ text array and finding all occurrences of the pattern embedded as subarray in the text.

Received September 27, 2016

Several well-known algorithms have been presented by researchers for the exact two dimensional pattern matching problem such as the Naive, Karp and Rabin [13], Zhu and Takaoka [29], Baker and Bird [7, 8] and the Baeza-Yates and Regnier [6] algorithms. Furthermore, survey and experiments on these algorithms have been already reported [19]. Similar pattern matching algorithms that allow for approximation [5], rotation [3, 10] or scaling [4] have been considered.

Two-dimensional pattern matching algorithms are frequently used to process huge image databases. As the size of image databases rapidly increases over the years, it is expected that the algorithms should benefit when implemented in parallel on a graphic processing unit (GPU). In the research literature, parallel implementations on GPUs have been presented for exact pattern matching and multiple pattern matching algorithms [18, 20, 24, 28, 11, 25, 23]. Similar research efforts in parallelizing two-dimensional pattern matching algorithms have focussed on OpenMP and MPI programming paradigms [17]. This present work focuses on the parallelization of the Baker and Bird and the Baeza-Yates and Regnier two-dimensional pattern matching algorithms through their implementation on a GPU using the Compute Unified Device Architecture (CUDA) programming model. The performance of the parallel implementations is evaluated after applying different optimization techniques under various parameters such as the pattern and text size as well as the size of the alphabet. To the best of our knowledge, this is the first time that the proposed two-dimensional pattern matching algorithms are implemented in parallel on a GPU using different optimization techniques.

The rest of the paper is organized as follows: in Section 2, we provide a short description of the Baker and Bird and Baeza-Yates and Regnier two-dimensional pattern matching algorithms. In Section 3, we discuss the parallel implementations of two-dimensional pattern matching algorithms on the GPU architecture. In Section 4 we present the experimental results of the proposed implementations. Finally, in Section 5, we draw conclusions.

2. **Background**

In this section we provide a short description of the Baker and Bird and Baeza-Yates and Rengier algorithms, tested for parallelization on GPUs. These two-dimensional pattern matching algorithms were chosen specifically because they are practical and efficient in terms of searching time and are frequently encountered in other research studies. The Baker-Brid and Baeza-Yates and Regnier algorithms both convert the two-dimensional pattern matching problem into a one-dimensional pattern matching problem. More specifically, they consider the two-dimensional pattern array as a finite set of different patterns. They also use multiple pattern matching algorithms to locate all the occurrences of the pattern in the text.

2.1. **Baker and Bird algorithm.** The Baker and Bird algorithm like other pattern matching algorithms consists of two phases: the preprocessing phase and the searching phase. During the preprocessing phase of the algorithm, an Aho-Corasick automaton [1] is built for each row of the two-dimensional pattern array and an array is then preprocessed using the Knuth-Morris-Pratt algorithm [14]. During the preprocessing phase of the algorithm, an Aho-Corasick automaton [1] is built for each row of the two-dimensional pattern array and an array is then preprocessed using the Knuth-Morris-Pratt algorithm [14]. During the preprocessing phase of the algorithm, an Aho-Corasick automaton [1] is built for each row of the two-dimensional pattern array and an array is then preprocessed using the Knuth-Morris-Pratt algorithm [14]. The searching phase of the algorithm uses two steps: row-matching and column-matching. During the row-matching step, each row of the text is scanned using Aho-Corasick automaton. For every occurrence position obtained in the row-matching step, it must be checked if all rows of the pattern appear vertically. This is done in the column-matching step using the Knuth-Morris-Pratt algorithm. Detailed information on the preprocessing and the searching phase of the Baker and Bird algorithm can be found in [19].

2.2. **Baeza-Yates and Regnier algorithm.** The Baeza-Yates and Regnier algorithm is similar to the Baker and Bird algorithm. It uses the Aho-Corasick [1] algorithm to create an automaton for each row of the two-dimensional pattern array. Each row of the two-dimensional text is then scanned for the occurrences of the pattern. A significant difference though is that during the search phase only $\lfloor \frac{n}{m} \rfloor$ primary rows of the text are scanned, since they cover all possible positions where a pattern row may occur. If a match is found on a primary row, then m characters on each of the $m-1$ secondary rows are also scanned for the pattern using the Aho-Corasick algorithm, to determine if a complete match occurs. For further details and pseudocode about the preprocessing and searching phase of the above sequential algorithm can be found in [19].

3. GPU implementations using CUDA

This section presents data-parallel implementations of the Baker and Bird and the Baeza-Yates and Regnier two-dimensional pattern matching algorithms and different optimization techniques.

3.1. **Parallel Implementation.** To parallelize the Baker and Bird and Baeza-Yates and Regnier algorithms, different implementation approaches were considered as each yielded the best performance results.

The execution of Baker and Bird in parallel in an efficient manner is not trivial, due to the data dependencies introduced during the *column-matching* step of the

algorithm. In practice, these dependencies inhibit the parallelism level of the implementation. As discussed in section 2.1, the Knuth-Morris-Pratt algorithm is used to determine all the positions of the text where a complete match of the pattern occurs. This is achieved by maintaining a table a with a size of n. Then, for each position j, k of the text, the value of $a[k]$ depends on the previous value of $a[k]$, as calculated in position $j-1, k$ and affects the value of $a[k]$, as determined in position $j+1, k$. For this reason, the rows of the text should be partitioned in such a way that all characters of a single column are processed by the same thread.

To implement the Baker and Bird algorithm in parallel using the CUDA API, the following parallelization approach was used. Each thread received a single text character, starting from the first row of the text and moving to each subsequent row when all threads had completed their execution paths. Since each active thread was responsible for a different column of the text, table a can be substituted with a variable a. Consider the following example; assume that character σ, located at position j, k of the text, is assigned to a thread i of the device. If there is a transition between the initial state of the trie and a state q labeled by σ, the thread reads the character at position $j, k+1$ next and so on, until a mismatch or a terminal state of the Aho-Corasick algorithm. When a terminal state is encountered or in the case of a mismatch, the value of variable a is updated according to the methodology described in section 2.1. As no movement of the trie takes place, the *supply* function of the Aho-Corasick algorithm is no longer required and can be removed, similar to the PFAC variant of Aho-Corasick as presented in [20]. Note that the *supply* function is used to visit a previous state of the automaton when there is no transition from the current state to a child state.

Since there are no dependencies between characters of the text for the Baeza-Yates and Regnier algorithm, different rows of the text can be simultaneously processed by the threads of the GPU. Thus, it is expected that a greater speedup can be achieved, comparing to Baker and Bird. To implement Baeza-Yates and Regnier in parallel, the following parallelization approach was used. The first $numBlocks$ *primary* rows of the text were assigned to thread blocks $0, \ldots, numBlocks-1$. The n characters of each *primary* row were further divided into chunks, with each chunk having a size of $S_{chunk} = m$ characters and were subsequently assigned to threads $0, \ldots, threadId-1$ of each block. To ensure the correctness of the results, additional $m-1$ characters were used per chunk, for a total of approximately n redundant characters per *primary* row of the text. If $m \times (blockDim-1) < n$, then the next $blockDim$ chunks were assigned to the threads and so on, until the end of the row was reached. Likewise, if $m-1+(numBlocks-1) \times m < n$, then the next $numBlocks$ *primary* rows of the text were assigned to the thread blocks, until the end of the text was reached. When a potential match was found on a *primary* row, the $m-1$ *secondary* rows were also

scanned by the corresponding thread to determine if a complete match of the pattern occurs. The improvement for the *secondary* rows of the text when duplicate pattern rows exist as described in section 2.2 was omitted, as it is inefficient to allocate a copy of table b for each of the $numBlocks \times blockDim$ threads of the device. Therefore, in the worst case where $p^0 = \cdots = p^{m-1}$, $2m - 2$ *secondary* rows will be scanned.

The preprocessing phase of the Baker and Bird and the Baeza-Yates and Regnier algorithms was performed sequentially by the host CPU. The text and all following preprocessing tables were copied to the global memory and therefore were accessible by all threads of the device. *State_transition* is a two-dimensional array where each row corresponds to one of the $m \times m + 1$ states of the Aho-Corasick trie, each column to a different character of the alphabet Σ while the cells of the array represent the *next* state. To ensure that alignment requirements are met on each row, *state_transition* was allocated as pitched linear device memory using the *cudaMallocPitch()* function. *State_final* is a one-dimensional array, where each column corresponds to a different state of the trie while the cells indicate whether that state is final or not. Finally, an array *out* with a size of $\lceil \frac{n}{m} \rceil$ integers was used to determine the number of matches per thread. Baker and Bird also utilized the *next* one-dimensional array of the Knuth-Morris-Pratt algorithm that consists of $m + 1$ integers. Baeza-Yates and Regnier used the *state_supply* array for the *supply* function of the Aho-Corasick algorithm. The $Id()$ function of the algorithm that is used to access the indices assigned to the terminal states of the trie was represented by an *id* array of size m.

3.2. **Optimization Techniques.** Although the presented parallel implementations can be over an order of magnitude faster than the corresponding sequential, this section discusses a number of limitations.

The parallel implementation of the Baker and Bird algorithm has a reduced efficiency due to its data dependencies. Note that the GTX 970 GPU that is used for the experiments of this paper has 13 SMs and each SM supports up to 2048 resident threads for a maximum hardware support of 26624 resident threads. Based on the way the parallelism of the Baker and Bird is exposed though, only a maximum of $n - m + 1$ threads can be active at any time. This results to the underutilization of the hardware resources and a reduced occupancy of the SMs, even when a text with a size of $n = 10,000$ was used. It is worth noting though that as discussed in [26], maximizing occupancy does not always result in a better performance. For the chosen implementation approach of the Baeza-Yates and Regnier algorithm on the other hand, a total of $\lfloor \frac{n}{m} \rfloor$ threads can be simultaneously active at each of the $\lfloor \frac{n}{m} \rfloor$ *primary* rows of the text.

3.2.1. *Coalescing Memory Accesses.* The performance of the algorithm implementations depends largely on the access pattern of the threads to the global memory of the

device. For the implementation of the Baker and Bird algorithm, the threads access consecutive characters of the text. To expand from the example of section 3.1, assume that thread i is the first thread of a warp[1] and that the warp scheduler assigns to it the character σ at position j, k of the text. Accesses to global memory for compute capability 5.2 GPUs by all threads of a warp are coalesced into a single memory transaction when the requested words are within the same memory segment. The segment size is 32 bytes when 1-byte words are accessed, 64 bytes for 2-byte words and 128 bytes for words of $4, 8$ and 16 bytes. If the text resides in a correctly aligned area of the global memory, then threads $i, \ldots, i+31$ will access text characters $j, k \ldots, k+31$ inside the same memory segment. Then, all 32 accesses will be coalesced into a single memory access. Since the maximum memory throughput of the global memory is 128 bytes per transaction, the access pattern of the threads results in the utilization of the $\frac{1}{4}$ of the available bandwidth.

For the implementation of the Baeza-Yates and Regnier algorithm, the memory accesses of the threads to the global memory for characters of a *primary* row of the text are *strided* with a stride of size m. Since 1-byte words are accessed, the memory segment size is 32 bytes. Then, when patterns with a size $m = 4$ are used, accesses to global memory are coalesced into groups of 4, for patterns with a size $m = 8$ the accesses are coalesced into groups of 2 while for patterns with a size $m \geq 16$, accesses to global memory are serialized. To work around the coalescing requirements of the global memory and increase the utilization of the memory bandwidth, it is important to change the memory access pattern by reading words from the same memory segment and subsequently store them in the shared memory of the device. This involves the partition of the *primary* rows of the text into $\frac{n}{S_{memsize}}$ chunks and the collective read of $S_{memsize}$ characters from the global into the shared memory by all *blockDim* threads of a thread block. Then, for each 16 successive characters from the same segment, only a single memory transaction will be used. This technique results in the improvement of the global memory bandwidth utilization by a factor of 16. The threads can subsequently access the characters stored in shared memory in any order with a very low latency. Using the shared memory to increase the utilization of the memory bandwidth has two disadvantages. First, a total of $\frac{n}{S_{memsize}} \times (m-1)$ redundant characters are used per *primary* row that introduce significantly more work overhead when compared to the basic data-parallel implementation strategy. Second, using the shared memory effectively reduces the occupancy of the SMs.

3.2.2. *Texture Binding.* The preprocessing arrays of the algorithms are relatively small in size while at the same time they are frequently accessed by the threads.

[1]Since threads are grouped in a deterministic way, and a warp of a compute capability 5.2 GPU consists of 32 threads, it holds that $i \bmod 32 = 0$

The performance of their parallel implementation should then benefit when the relevant arrays are bound to the texture memory of the device. The texture reference was bound to the device memory using *cudaBindTexture()* for one-dimensional arrays and *cudaBindTexture2D()* for two-dimensional arrays allocated as pitched linear device memory. The textures were then accessed in-kernel using the *tex1Dfetch()* and *tex2D()* functions. Arrays accessed via textures not only take advantage of the texture caches to minimize the memory latency when cache hits occur but also bypass the coalescing requirements of the global memory. In the case of the Baker and Bird algorithm implementation, the pattern is accessed directly during the *column-matching* step to perform character-by-character verification against the text. To improve the performance of the implementation, the pattern array can be either bound to the texture memory or it can be copied during the start of the search phase directly to the shared memory of the device by all threads of each thread block. As binding would increase the pressure on the texture caches, the pattern array was copied to the shared memory for the optimized implementation of Baker and Bird. Recall also that the first dimension of the *state_transition* array corresponds to the $m \times m + 1$ states of the Aho-Corasick trie and that the maximum size for two-dimensional texture references is $65,536 \times 32,768$ *texels*. Then, for pattern sizes of $m = 256$ or more, $m \times m + 1 > 65,536$ and therefore the *state_transition* array cannot be bound to the texture memory of the device.

Table 1 lists the register usage per thread for the basic and optimized versions of the algorithm implementations as reported by the NVCC compiler using the *–ptxas-options=-v* flag.

TABLE 1. Register usage per thread

Algorithm	Basic implementation	Optimized implementation
Baker and Bird	25	22
Baeza-Yates and Regnier	30	31

4. **Experimental results**

The performance of the proposed parallel implementations of the algorithms was evaluated by comparing the running time of the GPU implementations. The experiments were executed on an Intel Xeon CPU with a $2.40GHz$ clock speed and $2GB$ of memory which was used as a host and a GTX 970 GPU which was used as the device with compute capability 5.2. This device has $4GB$ of GDDR5 global memory, $1253MHz$ Graphics clock rate and $3.5GHz$ memory clock rate and consists of 13 SMs. Each SM has 128 SPs for a total of 1664 SPs, $96KB$ of on-chip shared memory (with a maximum shared memory size per thread block of $48KB$), and a $64KB$ 32-bit

register file. Each thread block can have a maximum of 1024 threads while each SM supports up to 2048 active threads. "-funroll-loops" optimization flags. The parallel GPU implementations of the algorithms were compiled using the NVCC 5.0 compiler of the CUDA API with the "-O2" optimization flag.

For the Baker and Bird algorithm implementation, 13 thread blocks with 128 threads per block where used for $n = 1000$ and 13 thread blocks with 768 threads per block where used for $n = 10000$. For the Baeza-Yates and Regnier algorithm implementation, 60 thread blocks with 512 threads per block where used.

The parameters that affect the performance of two-dimensional pattern matching algorithms are the size n^2 of the text, the size m^2 of the pattern and the size $|\Sigma|$ of the alphabet used. The experiments were repeated 100 times.

The data set used for the experiments was a superset of the sets used in [6, 22, 29]. It consisted of a randomly generated text with a size of $n = 1,000$ and $n = 10,000$ with three alphabets of size $2,256$ and 1024 to simulate bitmaps with different color depths. The pattern had a size of $m = 4, 8, 16, 32, 64, 128$ and 256 characters.

The performance of the parallel implementations is evaluated and obtained time results are compared to that yielded by sequential implementations. Each implementation stage also incorporates the optimizations of the previous stages. The first stage of the implementation was unoptimized. In this first stage, the input data and the preprocessing arrays of the algorithms were stored in the global memory of the device. The second stage of the implementation involved the binding of the preprocessing arrays to the texture memory of the device. The third stage, the pattern array was collectively copied by all the threads of a thread block to the shared memory of the device. This stage only applied to the implementation of the Baker and Bird algorithm. Finally, the fourth stage of the implementation involved coalescing read accesses to the global memory and storing the retrieved data to shared memory of the device. As already discussed, this optimization was only required for the implementation of the Baeza-Yates and Regnier algorithm.

Figures 1-2 and Figures 3-4 depict the running times of the presented implementations of the Baker and Bird and the Baeza-Yates and Regnier algorithms for different types of data on a GTX970 GPU. Recall from the previous section that the optimizations were valid only for patterns with a size $m \leq 128$, since the maximum size for two-dimensional texture references is $65,536 \times 32,768$ *texels*. As can generally be seen, the running time of the implementations descreased with each optimization made to the code, although the algorithms were affected in different ways.

As discussed in section 3, the occupancy of each SM of the GPU for the parallel implementation of the Baker and Bird algorithm depends on $n-m+1$. When a text of a size $n = 1000$ and $n = 1,000$ were used, the performance of the unoptimized parallel

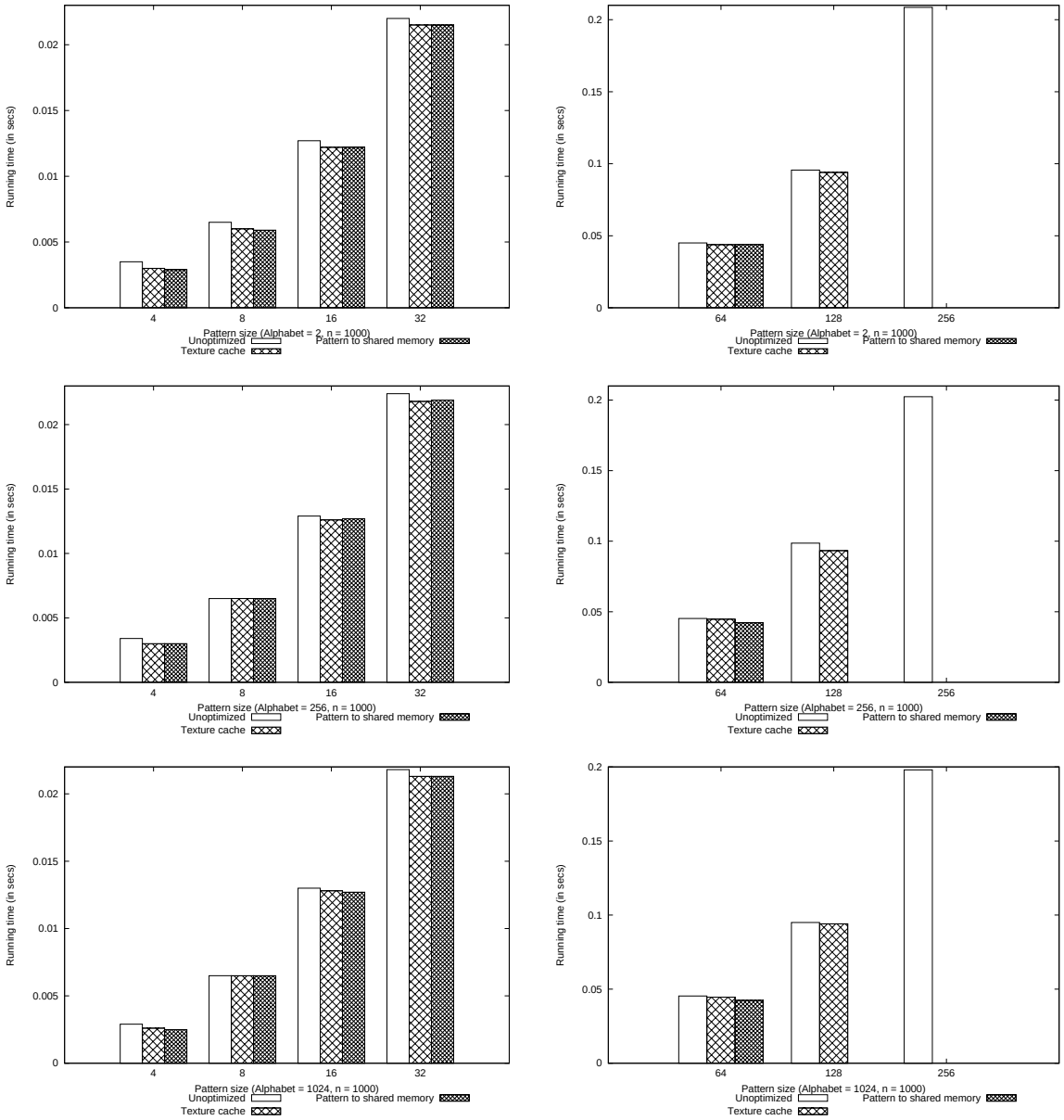

FIGURE 1. Running times of different optimization techniques for the parallel implementation of the Baker and Bird algorithm for $n = 1000$

implementation of the Baker-Bird algorithm decreased in the size of the pattern due to the decreased occupancy of the device. The performance of the algorithm implementation increased when the preprocessing arrays were bound to the texture memory of the device. The performance gain due to that specific optimization stage was more evident when data with a large alphabet size were used. The increased efficiency was caused mainly due to the frequent uncoalesced accesses of the threads to the global memory for the data stored in the preprocessing arrays.

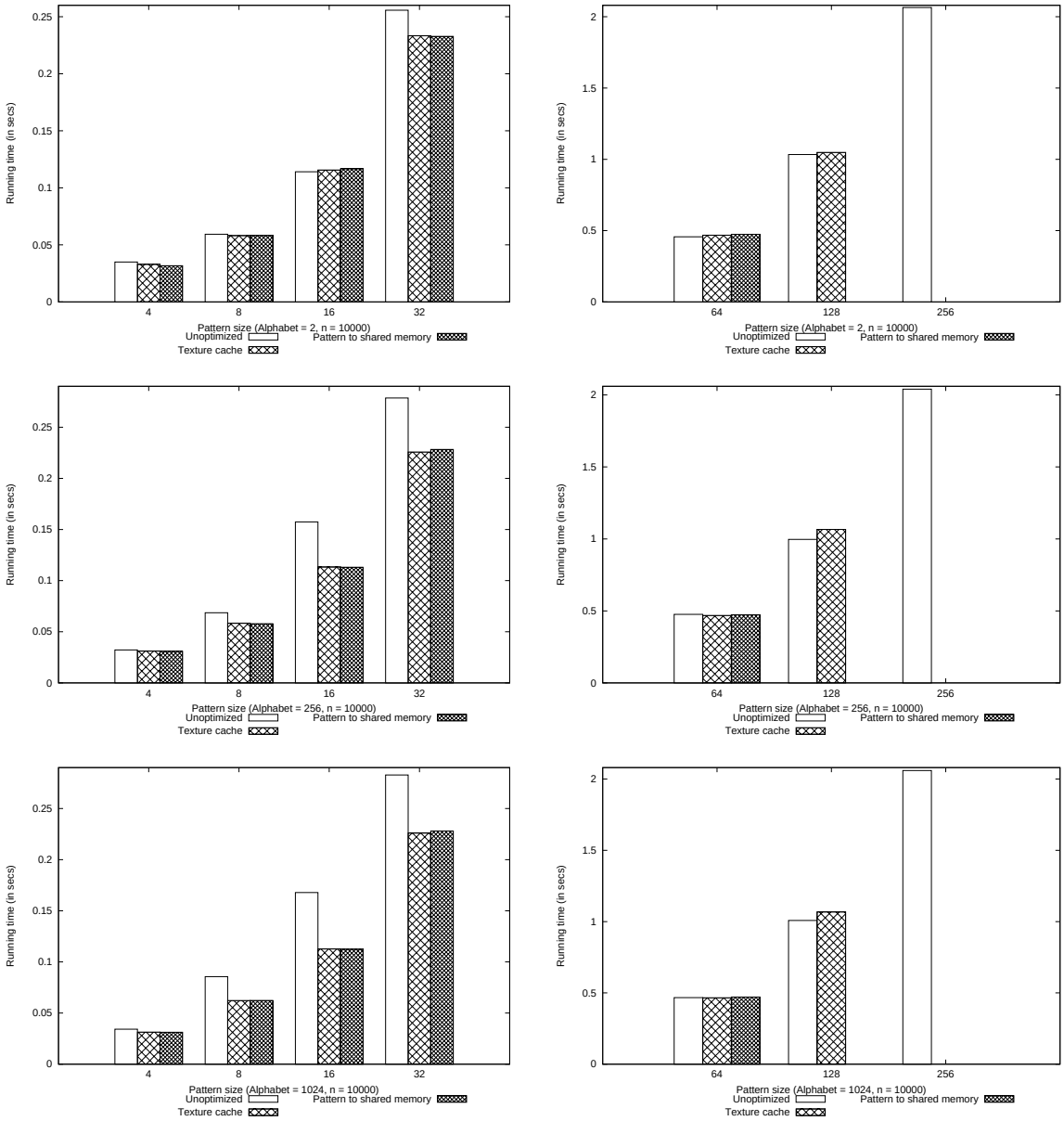

FIGURE 2. Running times of different optimization techniques for the parallel implementation of the Baker and Bird algorithm for $n = 10000$

When the pattern was collectively copied to the shared memory of the GPU by all threads of a thread block, the performance gain increased slightly comparing to the second optimization stage, especially when larger pattern sizes were used. In practice, the initial cost to copy the pattern array from the global to the shared memory of the device was substantial while the thread accesses to the pattern during the *column-matching* step of the Baker and Bird algorithm were not frequent enough to justify it. The performance gain of the final, optimized kernel ranged between 1

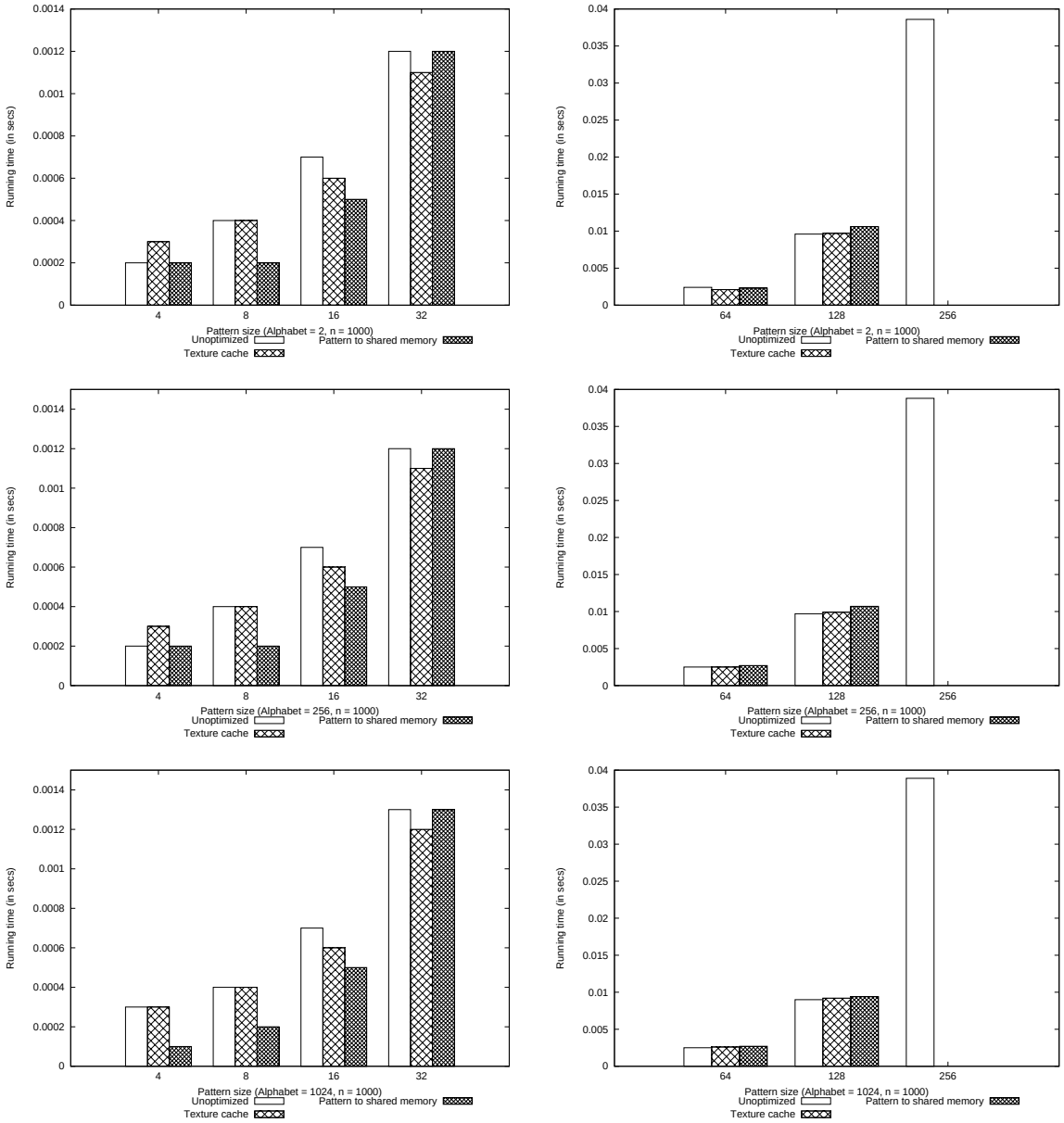

FIGURE 3. Running times of different optimization techniques for the parallel implementation of the Baeza-Yates and Regnier algorithm for $n = 1000$

and 1.20 times faster than the unoptimized kernel for $n = 1,000$ and between 0.96 and 1.50 for $n = 10,000$.

The parallel implementation of Baeza-Yates and Regnier had a significantly higher performance gain comparing to Baker and Bird. This can be attributed to the fact that the level of the algorithm's parallelism was improved due to the absence of data dependencies. Similar to the implementation of the Baker and Bird algorithm, the running time of Baeza-Yates and Regnier was mainly affected by the size n of

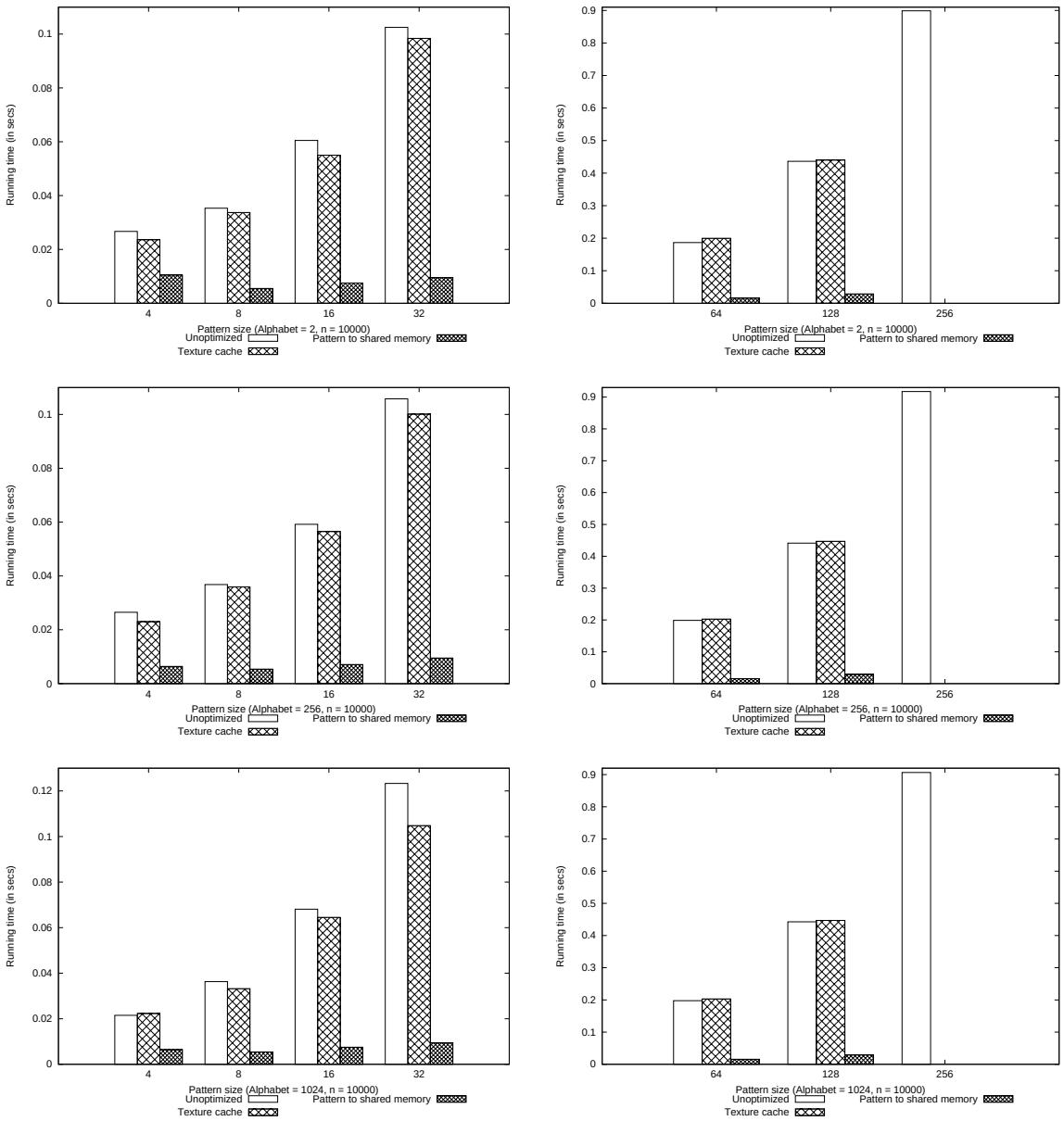

FIGURE 4. Running times of different optimization techniques for the parallel implementation of the Baeza-Yates and Regnier algorithm for $n = 10000$

the text and the size m of the pattern and to a lesser degree by the size $|\Sigma|$ of the alphabet.

For most types of data, the performance of the parallel implementation of the Baeza-Yates and Regnier algorithm decreased in the size of the pattern. When the second implementation stage was used, the performance gain of the parallel implementations improved especially when a text with a size of $n = 10,000$ was used. Finally, when the shared memory was used to work around the coalescing requirements of the

device, the performance gain of the parallel implementation of the Baeza-Yates and Regnier algorithm increased over the second implementation stage. The performance of the final, optimized kernel ranged between 1 and 2 times faster than the unoptimized implementation for $n = 1,000$ and between 2 and 15 times when a text with a size $n = 10,000$ was used.

5. **Conclusions**

In this paper, we presented details on the design and the implementation of the Baker and Bird and Baeza-Yates and Regnier algorithms on a GPU architecture using the CUDA API and evaluated their performance after applying different optimization techniques. The performance evaluation was based on different problem parameters. The optimization techniques used to further increase the performance of the implementations included the following; binding frequently used arrays to the texture memory of the device; copying the pattern to the shared memory of the device; coalescing read accesses to the global memory and storing the retrieved data to shared memory. It was concluded that the performance gain of the final optimized implemementation of the Baker and Bird algorithm was up to 1.5 times faster than the unoptimized implementation. It was also discussed that the final and optimized implementation of the Baeza-Yates and Regnier algorithm was up to 15 times faster than the unoptimized implementation.

Based upon the fact that in [6] it was suggested that the use of a multiple pattern matching algorithm based on Boyer-Moore instead of Aho-Corasick should result in the improvement of the searching phase of the Baeza-Yates and Regnier algorithm, efficient variants of the Baker and Bird and the Baeza-Yates and Regnier were introduced in [16, 15]. These variants have nearly all the characteristics that make them suitable for parallel execution on GPUs and their performance was further improved with their implementation on Graphics Processing Units. Finally, it would be interesting to examine the performance of the parallel implementations presented in this paper, especially for data sets with larger text and pattern sizes and for additional types of data, including photo archives and satellite imagery.

REFERENCES

[1] A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] J. Alwidian, H. Abu-Mansour, and M. Ali. Efficient algorithm for two dimensional pattern matching problem (non-square pattern). 2012.

[3] A. Amir, O. Kapah, and D. Tsur. Faster Two-dimensional Pattern Matching with Rotations. *Theoretical Computer Science*, 368(3):196–204, 2006. Combinatorial Pattern Matching.

[4] A. Amir, G. Landau, and U. Vishkin. Efficient Pattern Matching with Scaling. *Journal of Algorithms*, 13(1):2–32, 1992.

[5] R. Baeza-Yates and C. Perleberg. Fast and Practical Approximate String Matching. In *Combinatorial Pattern Matching*, pages 185–192. Springer, 1992.

[6] R. Baeza-Yates and M. Regnier. Fast Two Dimensional Pattern Matching. *Information Processing Letters*, 45(1):51–57, 1993.

[7] T. Baker. A Technique for Extending Rapid Exact-Match String Matching to Arrays of More than One Dimension. *SIAM Journal on Computing*, 7(4):533–541, 1978.

[8] R. Bird. Two Dimensional Pattern Matching. *Information Processing Letters*, 6(5):168–170, 1977.

[9] R. Cole, C. Hazay, M. Lewenstein, and D. Tsur. Two-dimensional parameterized matching. *ACM Transactions on Algorithms*, 11(2), 2014.

[10] K. Fredriksson, G. Navarro, and E. Ukkonen. Optimal Exact and Fast Approximate Two Dimensional Pattern Matching Allowing Rotations. *Combinatorial Pattern Matching*, 2373:235–248, 2002.

[11] L. Hu, Z. Wei, F. Wang, X. Zhang, and K. Zhao. An Efficient AC Algorithm with GPU. *Procedia Engineering*, 29:4249–4253, 2012.

[12] C. Hundt and F. Wendland. Efficient two-dimensional pattern matching with scaling and rotation and higher-order interpolation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7354 LNCS:124–137, 2012.

[13] R. Karp and M. Rabin. Efficient Randomized Pattern-matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[14] D. Knuth, J. Morris, and V. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[15] C. Kouzinopoulos. *Parallel and Distributed Implementations of Two-Dimensional and Multiple Pattern Matching Algorithms*. PhD thesis, Department of Applied Informatics, University of Macedonia, 2013.

[16] C. Kouzinopoulos and K. Margaritis. Improving the Efficiency of Exact Two Dimensional On-Line Pattern Matching Algorithms. In *Proceedings of the 12th Panhellenic Conference on Informatics*, pages 232–236, 2008.

[17] C. Kouzinopoulos and K. Margaritis. Parallel Implementation of Exact Two Dimensional Pattern Matching Algorithms using MPI and OpenMP. In *Proceedings of the 9th Hellenic European Research on Computer Mathematics and its Applications Conference*, 2009.

[18] C. Kouzinopoulos and K. Margaritis. String Matching on a Multicore GPU using CUDA. In *Proceedings of the 13th Panhellenic Conference on Informatics*, pages 14–18, 2009.

[19] C. Kouzinopoulos and K. Margaritis. Exact OnLine Two-Dimensional Pattern Matching Using Multiple Pattern Matching Algorithms. *Journal of Experimental Algorithmics*, 18, 2013.

[20] C. Lin, S. Tsai, C. Liu, S. Chang, and J. Shyu. Accelerating String Matching using Multi-Threaded Algorithm on GPU. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–5, 2010.

[21] T. Polcar and B. Melichar. A Two-dimensional Online Tessellation Automata Approach to Two-dimensional Pattern Matching. In *Proceedings of the Eindhoven FASTAR Days*, 2004. invited talk.

[22] J. Tarhio. A sublinear algorithm for two-dimensional string matching. *Pattern Recognition Letters*, 17(8):833 – 838, 1996.

[23] T. Tran, M. Giraud, and J.-S. Varr. Bit-parallel multiple pattern matching. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waniewski, editors, *Parallel Processing and Applied Mathematics*, volume 7204 of *Lecture Notes in Computer Science*, pages 292–301. Springer Berlin Heidelberg, 2012.

[24] A. Tumeo, S. Secchi, and O. Villa. Experiences with String Matching on the Fermi Architecture. *Architecture of Computing Systems-ARCS 2011*, pages 26–37, 2011.

[25] L. Vespa and N. Weng. SWM: Simplified Wu-Manber for GPU-based Deep Packet Inspection. In *Proceedings of the 2012 International Conference on Security and Management*, 2012.

[26] V. Volkov. Better Performance at Lower Occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.

[27] J. Zdarek. *Two-dimensional Pattern Matching using Automata Approach*. PhD thesis, Czech Technical University, 2010.

[28] X. Zha and S. Sahni. Multipattern String Matching on a GPU. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 277–282. IEEE, 2011.

[29] R. Zhu and T. Takaoka. A Technique for Two-dimensional Pattern Matching. *Communications of the ACM*, 32(9):1110–1120, September 1989.