Efficient Pattern Matching on CPU-GPU Heterogeneous Systems

Victoria Sanz[1,2(✉)], Adrián Pousa[1], Marcelo Naiouf[1], and Armando De Giusti[1,3]

[1] III-LIDI, School of Computer Sciences, National University of La Plata, La Plata, Argentina
{vsanz,apousa,mnaiouf,degiusti}@lidi.info.unlp.edu.ar
[2] CIC, Buenos Aires, Argentina
[3] CONICET, Buenos Aires, Argentina

Abstract. Pattern matching algorithms are used in several areas such as network security, bioinformatics and text mining, where the volume of data is growing rapidly. In order to provide real-time response for large inputs, high-performance computing should be considered. In this paper, we present a novel hybrid pattern matching algorithm that efficiently exploits the computing power of a heterogeneous system composed of multicore processors and multiple graphics processing units (GPUs). We evaluate the performance of our algorithm on a machine with 36 CPU cores and 2 GPUs and study its behaviour as the data size and the number of processing resources increase. Finally, we compare the performance of our proposal with that of two other algorithms that use only the CPU cores and only the GPUs of the system respectively. The results reveal that our proposal outperforms the other approaches for data sets of considerable size.

Keywords: Pattern matching · CPU-GPU computing · CPU-GPU heterogeneous systems · Hybrid programming · Aho-Corasick

1 Introduction

Pattern matching algorithms locate some or all occurrences of a finite number of patterns (pattern set or dictionary) in a text (data set). These algorithms are key components of DNA analysis applications [1], antivirus [2], intrusion detection systems [3,4], among others. In this context, the Aho-Corasick (AC) algorithm [5] is widely used because it efficiently processes the text in linear time.

The ever-increasing amount of data to be processed, sometimes in real time, led several authors to investigate the acceleration of AC on emerging parallel architectures. In particular, researchers have proposed different approaches to parallelize AC on shared-memory architectures, distributed-memory architectures (clusters), GPUs and multiple GPUs [6–10].

S. Wen et al. (Eds.): ICA3PP 2019, LNCS 11944, pp. 391–403, 2020.
https://doi.org/10.1007/978-3-030-38991-8_26

Although modern computers include multiple CPU cores and at least one GPU, little work has been done to accelerate pattern matching on such systems. In [11] the authors present a hybrid parallelization of AC on CPU-GPU heterogeneous systems. Briefly, the algorithm consists of generating the data structures needed for pattern matching on the CPU cores and performing the matching process on the GPU. Similarly, in [12] the authors propose a hybrid CPU-GPU pattern-matching algorithm that uses the CPU to filter incoming data and the GPU to complete the matching process. The filter phase detects blocks of data suspected of containing patterns, thus it reduces the GPU workload and data transfers. In summary, previous work has focused on using only one type of processing unit (PU) for the matching process, i.e. CPUs or GPUs, leading to underutilization of system resources. To our best knowledge, no work has focused on using both PUs in a collaborative way to accelerate the matching process of pattern matching algorithms.

In this paper, we present a novel hybrid pattern matching algorithm that efficiently exploits the computing power of a heterogeneous system composed of multicore processors and multiple GPUs. Also we address the problem of load balancing among the processing resources of the system. We evaluate the performance of our algorithm on a machine with 36 CPU cores and 2 GPUs and study its behaviour as the data size and the number of processing resources increase. Finally, we compare the performance of our proposal with that of two other algorithms that use only the CPU cores and only the GPUs of the system respectively. The results reveal that our proposal outperforms the other approaches for data sets of considerable size.

This paper extends the work in [13] by presenting (1) a generalization of our algorithm to a wider range of heterogeneous systems, (2) an analysis of its behaviour as the problem size and the number of processing resources increase, and (3) a detailed comparison with previous approaches.

The rest of the paper is organized as follows. Section 2 introduces the AC algorithm. Section 3 summarizes two approaches to parallelize AC. Section 4 introduces a hybrid OpenMP-CUDA programming model and a workload distribution strategy, which are specific to CPU–GPU heterogeneous computing. Section 5 describes our parallel algorithm for pattern matching on CPU-GPU heterogeneous systems. Section 6 shows our experimental results. Finally, Sect. 7 presents the main conclusions and some ideas for future research.

2 The Aho-Corasick Algorithm

The AC algorithm [5] has been widely used since it is able to locate all occurrences of user-specified patterns in a single pass of the text. The algorithm consists of two steps: the first is to construct a finite state pattern matching machine; the second is to process the text using the state machine constructed in the previous step. The pattern matching machine has valid and failure transitions. The former are used to detect all user-specified patterns. The latter are used to backtrack the state machine, specifically to the state that represents the

longest proper suffix, in order to recognize patterns starting at any location of the text. Certain states are designated as "output states" which indicate that a set of patterns has been found. The AC machine works as follows: given a current state and an input character, it tries to follow a valid transition; if such a transition does not exist, it jumps to the state pointed by the failure transition and processes the same character until it causes a valid transition. The machine emits the corresponding patterns whenever an output state is found. Figure 1 shows the state machine for the pattern set {he, she, his, hers}. Solid lines represent valid transitions and dotted lines represent failure transitions.

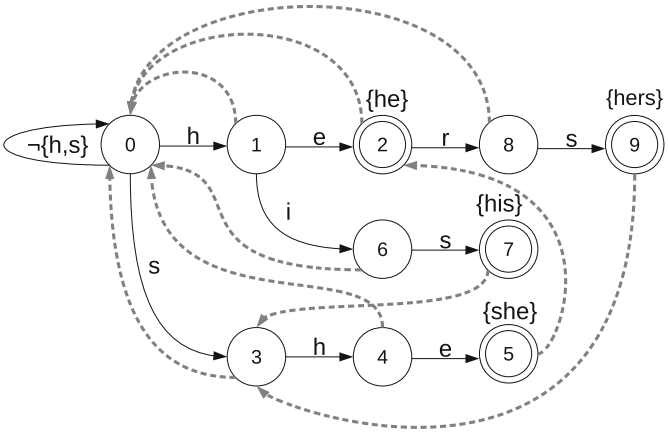


Fig. 1. Aho-Corasick state machine for the pattern set {he, she, his, hers}

3 Previous Approaches to Parallelize Aho-Corasick

The most straightforward way to parallelize AC [6] is based on dividing the input text into segments and making each processor responsible for a particular segment (i.e., each processor performs AC on its segment). All processors use the same state machine. The disadvantage of this strategy is that patterns can cross the boundary of two adjacent segments. This problem is known as the "boundary detection" problem. In order to detect these patterns, each processor has to compute an additional chunk known as "overlapping area", whose size is equal to the length of the longest pattern in the dictionary minus 1. However, this additional computation is an overhead that increases as the text is divided into more segments of smaller size. Figure 2 illustrates this strategy.

Another approach is the Parallel Failureless Aho-Corasick algorithm (PFAC) [7] that efficiently exploits the parallelism of AC and therefore is suitable for GPUs. PFAC assigns each character of the text to a particular thread. Each thread is responsible for identifying the pattern beginning at its assigned position and terminates immediately when it detects that such a pattern does not exist (i.e., when it cannot follow a valid transition). Note that PFAC does not use

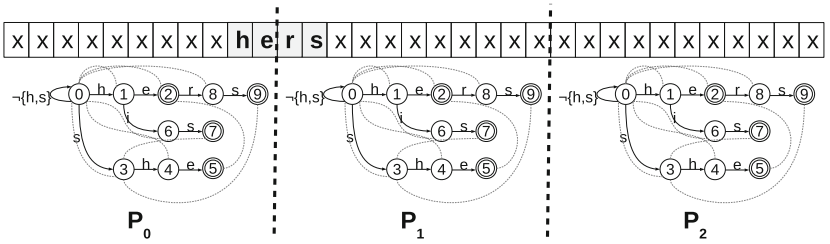


Fig. 2. Parallel AC

failure transitions and therefore they can be removed from the state machine. Figures 3 and 4 give an example of the PFAC state machine and the PFAC algorithm, respectively. Algorithm 1.1 shows the pseudocode of PFAC (code executed by each thread).

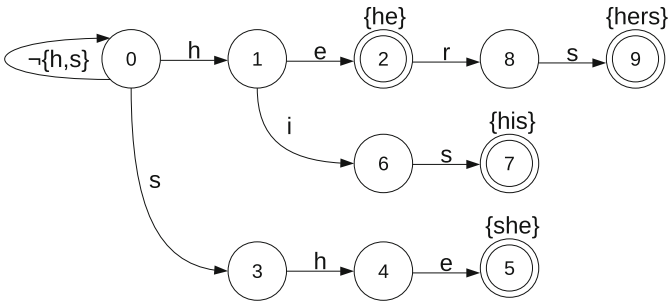


Fig. 3. PFAC state machine for the pattern set {he, she, his, hers}

Algorithm 1.1. Pseudocode of PFAC

```
pos = start
state = initial state
while ( pos < text size ){
    if (there is no transition for the current state and input character)
        break
    state = next state for the current state and input character
    if (state is an output state)
        register the pattern located at the position "start"
    pos = pos + 1
}
```

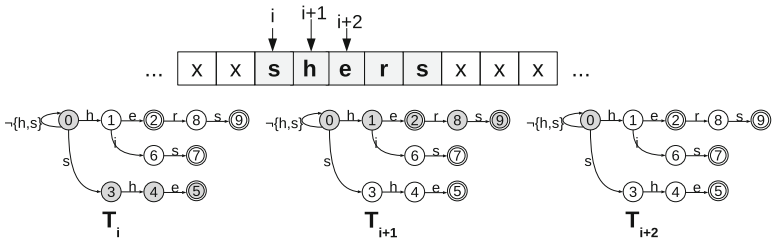


Fig. 4. Example of PFAC

4 A General CPU-GPU Computing Model

One of the reasons that motivate CPU-GPU heterogeneous computing is to improve the utilization of the processing units (PUs). The use of both types of PUs in a collaborative way may improve the performance of the application. In this section we introduce a hybrid OpenMP-CUDA programming model and a workload distribution strategy, which are specific to CPU-GPU heterogeneous computing.

4.1 Hybrid OpenMP-CUDA Programming Model

Assuming that the system is composed of N CPU cores and M GPUs, the proposed model creates two sets of threads. The first set has M threads, each one runs on a dedicated CPU core and controls one GPU, which involves the following steps: allocating memory on the GPU to store input and output data, transferring the data from CPU to GPU, calling the kernel function, transferring the results from GPU to CPU and freeing memory on the GPU. On the other hand, the second set has $N - M$ threads, which concurrently perform the corresponding calculations on the remaining CPU cores. Figure 5 depicts this hybrid programming model.

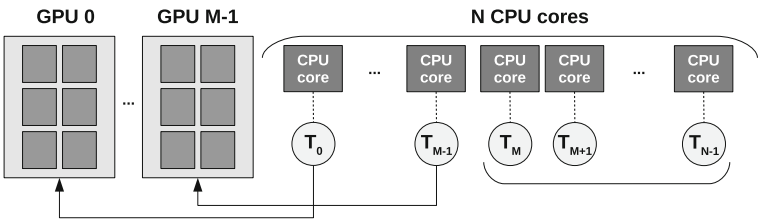


Fig. 5. Hybrid OpenMP-CUDA programming model

4.2 Workload Distribution Strategy

A workload distribution is optimal when the PUs complete their respective work within the same amount of time. In order to distribute the work among the PUs, we use a simple static workload distribution (i.e., the amount of work to be assigned to each PU is determined before program execution) based on the relative performance of PUs [14].

Specifically, we estimate the CPU and GPU(s) execution time in the collaborative implementation as $T'_{cpu} = T_{cpu} \cdot R$ and $T'_{gpu} = T_{gpu} \cdot (1 - R)$, respectively, where R is the proportion of work assigned to the CPU cores, T_{cpu} represents the execution time of the OpenMP algorithm on the available CPU cores and T_{gpu} is the execution time of the single-GPU or multi-GPU algorithm using CUDA, as appropriate. Clearly, the execution time of the collaborative implementation reaches its minimum when $T'_{cpu} = T'_{gpu}$, i.e. $T_{cpu} \cdot R = T_{gpu} \cdot (1 - R)$. From this equation we obtain $R = \frac{T_{gpu}}{T_{cpu}+T_{gpu}}$.

In our scenario, R has to be recalculated when the input data vary or the configuration of the system changes. According to R, the workload assigned to the CPU cores is $D_{cpu} = R \cdot D_{size}$ and the workload assigned to the GPU(s) is $D_{gpu} = D_{size} - D_{cpu}$, where D_{size} is the length of the text string.

Although it is impractical to run both OpenMP and CUDA applications in order to obtain R, we plan to use this first approach as a baseline to derive an estimation model for R.

5 Pattern Matching on CPU-GPU Heterogeneous Systems

Our implementation is based on the PFAC algorithm and uses the hybrid programming model proposed in Sect. 4.1.

Our algorithm generates the state machine on the CPU sequentially. The state machine is represented by a State Transition Table (STT) that has a row for each state and a column for each ASCII character (256). Each entry of the STT contains the next state information. Once generated, the STT is copied to the texture memory of the GPU(s) since this table is accessed in an irregular manner and, in this way, the access latency is reduced.

The algorithm distributes the workload (input text) between the CPU and the GPU(s) according to the strategy proposed in Sect. 4.2. Thus, the text is divided into two segments and the "boundary detection" problem appears. The first segment is assigned to the CPU and the second one to the GPU(s) (Fig. 6). In this way, the CPU has to compute an additional chunk (overlapping area) already residing in main memory and thus we reduce the amount of data to be transferred to the GPU(s).

When the heterogeneous system has several identical GPUs, the workload (segment) is distributed equally among them, taking into account the overlapping area. Each thread in charge of managing one GPU copies its segment into the global memory. Note that large segments may exceed the global memory

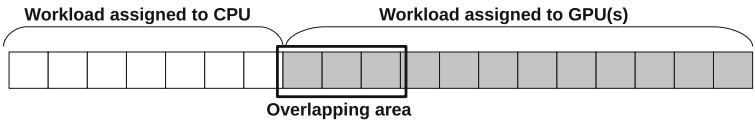


Fig. 6. Workload distribution between the CPU and the GPU(s)

capacity. In that case, the thread subdivides the segment into smaller segments and then it transfers and processes them one by one. It should be noted that each sub-segment must be transferred with the corresponding overlapping area. The implementation details of the PFAC algorithm on GPU (PFAC kernel) can be found in [7]. In summary, the kernel is launched with 256 threads per block. Each thread block handles 1024 positions of the input segment (i.e., each thread processes 4 positions). Each thread block loads the corresponding data into shared memory. Then, threads read input bytes from the shared memory in order to perform their work.

The threads that operate on the CPU cores distribute the workload (segment) equally among them via the OpenMP 'for' work-sharing directive.

6 Experimental Results

Our experimental platform is a machine composed of two Intel Xeon E5-2695 v4 processors and 128 GB RAM. Each processor has eighteen 2.10 GHz cores, thus the machine has thirty-six cores in total. Hyper-Threading and Turbo Boost were disabled. The machine is equipped with two Nvidia GeForce GTX 960; each one is composed of 1024 cores and 2 GB GDDR5 memory. Each CUDA core operates at 1127 MHz.

Test scenarios were generated by combining three English texts of different sizes with four English dictionaries with different number of patterns. All the texts were extracted from the British National Corpus [15]: text 1 is a 4-million-word sample (21 MB); text 2 is a 50-million-word sample (268 MB); text 3 is a 100-million-word sample (544 MB). The dictionaries include frequently used words: dictionary 1 with 3000 words; dictionary 2 with 100000 words; dictionary 3 with 178690 words; dictionary 4 with 263533 words.

To evaluate the effectiveness of our proposal, we compared the sequential version of PFAC (PFAC_SEQ) with the following parallel implementations:

- PFAC_CPU: implementation of PFAC on a multicore CPU using OpenMP.
- PFAC_GPU: implementation of PFAC on GPU using CUDA and executed with 256 threads per block.
- PFAC_MultiGPU: implementation of PFAC on multiple GPUs using CUDA and OpenMP, which is used only for managing the GPUs.
- PFAC_CPU-GPU: hybrid OpenMP-CUDA implementation of PFAC for heterogeneous systems composed of multicore processors and 1 GPU.
- PFAC_CPU-MultiGPU: hybrid OpenMP-CUDA implementation of PFAC for heterogeneous systems composed of multicore processors and multiple GPUs.

It should be noted that PFAC_SEQ, PFAC_CPU and PFAC_GPU are the original implementations provided by Lin et al. [7]. We developed the remaining versions described above, which are based on the aforementioned original implementations.

Our experiments focus on the matching step since it is the most significant part of pattern matching algorithms. For each test scenario, we ran each implementation 100 times and averaged the execution time. PFAC_CPU, PFAC_CPU-GPU and PFAC_CPU-MultiGPU were executed with the following system configurations: 6, 12, 18, 24, 30 and 36 threads/CPU cores. We considered the data transfer time (host-to-device and device-to-host, aka H2D and D2H) when evaluating the algorithms that use GPU(s), since it represents a significant portion of the total execution time [13] (i.e. it is not negligible).

First, we calculated the load balance of each run for the algorithms that use several processing resources (PFAC_CPU, PFAC_MultiGPU, PFAC_CPU-GPU and PFAC_CPU-MultiGPU). Load balance [16] can be defined as the ratio between the average time to finish all of the parallel tasks and the maximum time to finish any of the parallel tasks ($\frac{T_{avg}}{T_{max}}$). A load balance value near 1 means a better distribution of load.

In PFAC_CPU, each OpenMP thread represents a parallel task. On the other hand, in PFAC_MultiGPU, the work done by each GPU is a parallel task. Considering all tests for each algorithm, both achieve an average load balance of 0.99.

Table 1. Values of R used by PFAC_CPU-GPU and PFAC_CPU-MultiGPU

		No. of CPU cores					
		6	12	18	24	30	36
PFAC_CPU-GPU	Text 1	0.30	0.38	0.43	0.40	0.38	0.37
	Text 2	0.36	0.49	0.58	0.61	0.62	0.62
	Text 3	0.37	0.51	0.59	0.64	0.67	0.68
PFAC_CPU-MultiGPU	Text 1	0.21	0.28	0.33	0.30	0.28	0.28
	Text 2	0.24	0.35	0.43	0.47	0.48	0.48
	Text 3	0.24	0.36	0.44	0.49	0.52	0.54

PFAC_CPU-GPU and PFAC_CPU-MultiGPU consist of two parallel tasks: one is performed by the CPU cores and the other by the GPU(s). We distributed the input text among the PUs according to the strategy proposed in Sect. 4.2. Table 1 shows the value of R used by both algorithms, for each text and system configuration. Similarly, Table 2 presents the load balance achieved. The results reveal that our workload distribution strategy provides a good load-balance, which ranges between 0.75 and 0.96 for PFAC_CPU-GPU, and between 0.77 and 0.93 for PFAC_CPU-MultiGPU. Additionally, both algorithms follow a similar trend: the load balance improves as the size of the text increases.

Table 2. Load balance achieved by PFAC_CPU-GPU and PFAC_CPU-MultiGPU

		No. of CPU cores					
		6	12	18	24	30	36
PFAC_CPU-GPU	Text 1	0.82	0.86	0.84	0.80	0.77	0.75
	Text 2	0.89	0.92	0.95	0.91	0.91	0.92
	Text 3	0.90	0.94	0.96	0.95	0.94	0.94
PFAC_CPU-MultiGPU	Text 1	0.77	0.86	0.88	0.84	0.80	0.79
	Text 2	0.79	0.88	0.91	0.91	0.89	0.89
	Text 3	0.81	0.89	0.93	0.93	0.92	0.91

Next, we evaluated the performance (Speedup[1]) of the mentioned algorithms. For each algorithm and system configuration, the average speedup for each text is shown. This is because the speedup does not vary significantly with the dictionary.

Figure 7 illustrates the average speedup of PFAC_GPU and PFAC_MultiGPU, for different texts. In both cases, the speedup increases when going from Text 1 to Text 2, but then it plateaus. This is mainly due to the fact that the data transfer time (H2D and D2H) increases with the size of the text. In particular, for large texts, this overhead has a greater impact on performance. Note that the performance of PFAC_MultiGPU is less affected by data transfers, compared to PFAC_GPU. This is because PFAC_MultiGPU distributes the load equally among the GPUs of the system. Thus, independent small data transfers occur in parallel. Also, it can be observed that PFAC_MultiGPU achieves higher performance than PFAC_GPU.

Figure 8 shows the average speedup of PFAC_CPU, PFAC_CPU-GPU and PFAC_CPU-MultiGPU, for different texts and system configurations (the number of threads/CPU cores is indicated between parentheses). In each case, it can be seen that the system configuration that provides the best performance depends on the text. Moreover, in some cases different system configurations give the same performance for a given text. For this reason, fewer resources should be used to achieve an acceptable speedup with a low energy consumption. The analysis of energy consumption is out of the scope of this paper and is the subject of future work. Additionally, note that, for a fixed number of processing resources, the speedup of PFAC_CPU, PFAC_CPU-GPU and PFAC_CPU-MultiGPU tends to increase with the size of the text. Therefore, we conclude that these algorithms behave well as the workload increases.

Figure 9 compares the performance of the algorithms. In the case of PFAC_CPU, PFAC_CPU-GPU and PFAC_CPU-MultiGPU, we selected the system configuration that provides the best performance for each text. For example:

[1] Speedup is defined as $\frac{T_s}{T_p}$, where T_s is the execution time of the sequential algorithm and T_p is the execution time of the parallel algorithm.

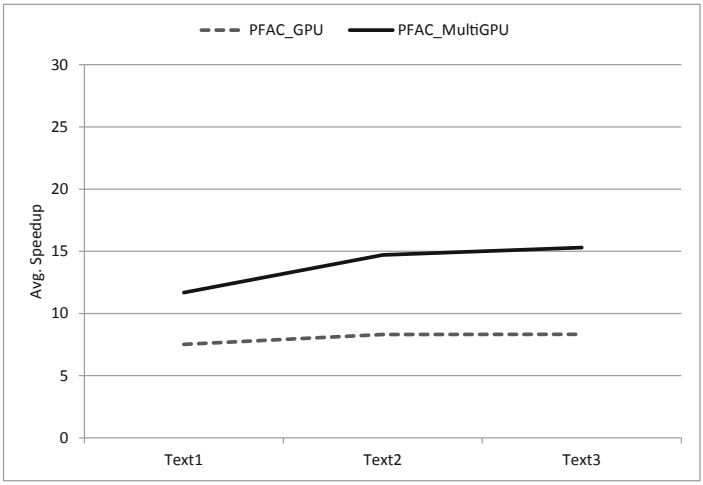


Fig. 7. Average speedup of PFAC_GPU and PFAC_MultiGPU

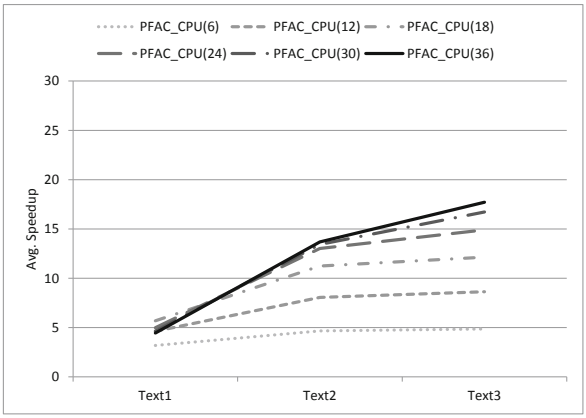
for PFAC_CPU and Text 1, we show the average speedup achieved with 18 CPU cores, whereas for Texts 2 and 3 we show the average speedup achieved with 36 CPU cores.

As it can be observed, PFAC_MultiGPU achieves the best performance for Text 1, followed by PFAC_CPU-MultiGPU (18 CPU cores + 2 GPUs), PFAC_CPU-GPU (18 CPU cores + 1 GPU), PFAC_GPU and PFAC_CPU (18 CPU cores). For this text, the algorithms achieve an average speedup of 11.69, 11.48, 8.14, 7.51, and 5.69 respectively.

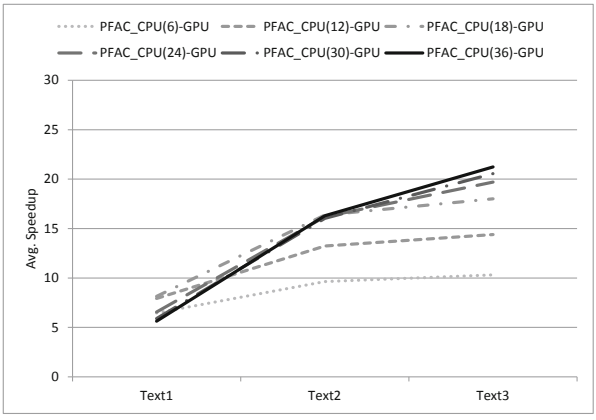
Furthermore, PFAC_CPU-MultiGPU (24 CPU cores + 2 GPUs) achieves the best result for Text 2, with an average speedup of 20.85, followed by PFAC_CPU-GPU (18 CPU cores + 1 GPU) with 16.37, PFAC_MultiGPU with 14.70, PFAC_CPU (36 CPU cores) with 13.69 and PFAC_GPU with 8.31.

Finally, for Text 3, PFAC_CPU-MultiGPU achieves the best average speedup (25.41) by using all available resources, followed by PFAC_CPU-GPU (36 CPU cores + 1 GPU) with 21.23, PFAC_CPU (36 CPU cores) with 17.71, PFAC_MultiGPU with 15.31 and PFAC_GPU with 8.33.

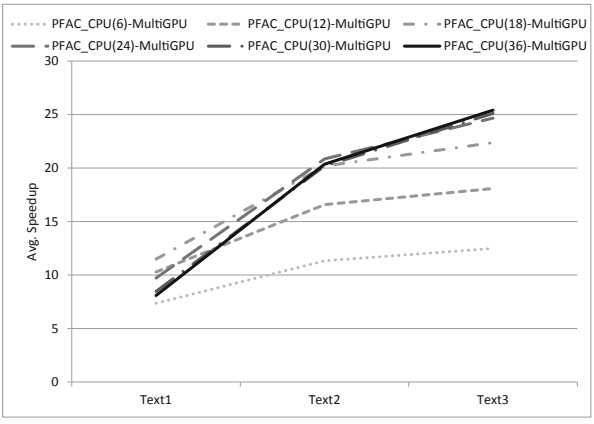
Note that for Texts 2 and 3, the algorithms that use the CPU cores outperform those that use only 1 or 2 GPUs. In general, this is due to the impact of H2D/D2H transfers on PFAC_GPU and PFAC_MultiGPU. The former algorithm transfers the entire data between CPU and GPU, whereas the latter transfers an equal amount of data to each GPU in parallel. On the other hand, both PFAC_CPU-GPU and PFAC_CPU-MultiGPU are less affected by data transfers since they transfer a smaller portion of data, according to R.



(a)



(b)



(c)

Fig. 8. Average speedup of (a) PFAC_CPU, (b) PFAC_CPU-GPU and (c) PFAC_CPU-MultiGPU

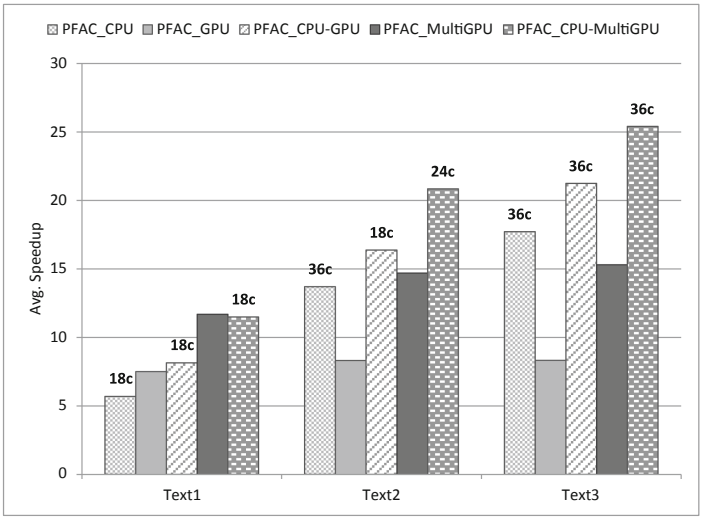


Fig. 9. Performance comparison of parallel matching algorithms

7 Conclusions and Future Work

In this paper we presented a novel pattern matching algorithm that efficiently exploits the computing power of a heterogeneous system composed of multicore processors and multiple GPUs. Our proposal is based on the Parallel Failureless Aho-Corasick algorithm for GPU (PFAC_GPU).

We evaluated the performance of our algorithm (PFAC_CPU-MultiGPU) on a machine with 36 CPU cores and 2 GPUs, and compared it with that of: PFAC_GPU; PFAC_MultiGPU, a version of PFAC that runs on multiple GPUs; PFAC_CPU, a version of PFAC for multicore CPUs; PFAC_CPU-GPU, a hybrid version of PFAC that uses multiple CPU cores and a single GPU.

The results showed that PFAC_CPU-MultiGPU outperforms the other algorithms, for texts of considerable size. In particular, it reaches an average speedup of 25.41 for the largest problem considered. However, PFAC_MultiGPU achieves the best performance for small texts. Furthermore, for a fixed number of processing resources, the speedup of PFAC_CPU-MultiGPU tends to increase with the size of the text. Therefore, we conclude that it behaves well as the workload increases.

As for future work, we plan to extend the experimental work to evaluate the PFAC algorithm on other parallel architectures such as Xeon Phi and CPU-GPU clusters. Also we plan to construct a model based load-balancing strategy.

References

1. Tumeo, A., Villa, O.: Accelerating DNA analysis applications on GPU clusters. In: IEEE 8th Symposium on Application Specific Processors (SASP), pp. 71–76. IEEE Computer Society, Washington D.C. (2010)
2. Clamav. http://www.clamav.net
3. Norton, M.: Optimizing pattern matching for intrusion detection. Sourcefire Inc., White Paper. https://www.snort.org/documents/optimization-of-pattern-matches-for-ids
4. Tumeo, A., et al.: Efficient pattern matching on GPUs for intrusion detection systems. In: Proceedings of the 7th ACM International Conference on Computing Frontiers, pp. 87–88. ACM, New York (2010)
5. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
6. Tumeo, A., et al.: Aho-Corasick string matching on shared and distributed-memory parallel architectures. IEEE Trans. Parallel Distrib. Syst. **23**(3), 436–443 (2012)
7. Lin, C.H., et al.: Accelerating pattern matching using a novel parallel algorithm on GPUs. IEEE Trans. Comput. **62**(10), 1906–1916 (2013)
8. Arudchutha, S., et al.: String matching with multicore CPUs: performing better with the Aho-Corasick algorithm. In: Proceedings of the IEEE 8th International Conference on Industrial and Information Systems, pp. 231–236. IEEE Computer Society, Washington D.C. (2013)
9. Herath, D., et al.: Accelerating string matching for bio-computing applications on multi-core CPUs. In: Proceedings of the IEEE 7th International Conference on Industrial and Information Systems (ICIIS), pp. 1–6. IEEE Computer Society, Washington D.C. (2012)
10. Lin, C.H., et al.: A novel hierarchical parallelism for accelerating NIDS using GPUs. In: Proceedings of the 2018 IEEE International Conference on Applied System Invention (ICASI), pp. 578–581. IEEE (2018)
11. Soroushnia, S., et al.: Heterogeneous parallelization of Aho-Corasick algorithm. In: Proceedings of the IEEE 7th International Conference on Industrial and Information Systems (ICIIS), pp. 1–6. IEEE Computer Society, Washington D.C. (2012)
12. Lee, C.L., et al.: A hybrid CPU/GPU pattern-matching algorithm for deep packet inspection. PLoS One **10**(10), 1–22 (2015)
13. Sanz, V., Pousa, A., Naiouf, M., De Giusti, A.: Accelerating pattern matching with CPU-GPU collaborative computing. In: Vaidya, J., Li, J. (eds.) ICA3PP 2018. LNCS, vol. 11334, pp. 310–322. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-05051-1_22
14. Wan, L., et al.: Efficient CPU-GPU cooperative computing for solving the subset-sum problem. Concurr. Comput. Pract. Exp. **28**(2), 185–186 (2016)
15. The British National Corpus, version 3 (BNC XML Edition). Distributed by Bodleian Libraries, University of Oxford, on behalf of the BNC Consortium (2007). http://www.natcorp.ox.ac.uk/
16. Rahman, R.: Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers. Apress, Berkeley (2013)