Accelerating Pattern Matching on Intel Xeon Phi Processors

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. In order to provide real-time response for large inputs, high-performance systems should be used. However, this requires adapting the algorithm to the underlying architecture. Intel Xeon Phi processors have attracted attention in recent years because they offer massive parallelism, good programmability and portability. In this paper, we present a pattern matching algorithm that exploits the full computational power of Intel Xeon Phi processors by using both SIMD and thread parallelism. We evaluate our algorithm on a Xeon Phi 7230 Knights Landing processor and measure its performance as the data size and the number of threads increase. The results reveal that both parallelism methods provide performance gains. Also they indicate that our algorithm is up to 63x faster than its serial counterpart and behaves well as the workload is increased.

Keywords: Pattern matching · Intel Xeon Phi Knights Landing processor · SIMD instructions · Thread-level parallelism · 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 a single pass.

Nowadays, the amount of data that need to be processed grows very rapidly. This has 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, multiple GPUs and CPU-GPU heterogeneous systems [6–14].

M. Qiu (Ed.): ICA3PP 2020, LNCS 12452, pp. 262–274, 2020.
https://doi.org/10.1007/978-3-030-60245-1_18

In recent years, Intel Xeon Phi systems have attracted attention of researchers because of their massive parallelism. These systems support multiple programming languages and tools for parallel programming, thus they provide better programmability and portability than a system using GPUs. There are two generations of Xeon Phi: Knights Corner (KNC) and Knights Landing (KNL) [15,16]. The former has limitations similar to those of GPUs: it is a PCIe-connected coprocessor with limited memory. The latter is also available as a standalone processor, therefore it does not have the limitations of the previous model. Furthermore, it is binary compatible with prior Intel processors.

In general, a Xeon Phi is composed of multiple cores. Each core has 1 or 2 vector processing units (VPUs), depending on the model. To take full advantage of its computational power, applications must exploit thread-level and SIMD parallelism. This is a challenge for developers since the code must be changed in order to launch multiple parallel tasks and expose vectorization opportunities.

So far, little work has been done to accelerate Aho-Corasick on Xeon Phi. In [17] the authors present a parallel AC algorithm and test it on a Xeon Phi KNC coprocessor. Briefly, the algorithm consists of dividing the text among threads, then each thread processes its segment in a vectorized way. Similarly, in [18] the authors propose an AC algorithm for Xeon Phi coprocessors, which uses a strategy that increases cache locality during the pattern matching process. The strategy is based on partitioning the set of patterns into smaller subsets and executing several independent instances of the matching procedure on the entire text (i.e., one instance for each subset of patterns). However, this proposal does not take advantage of the VPUs of the coprocessor.

In this paper, we present an AC algorithm that exploits the full computational power of Intel Xeon Phi processors by using both SIMD and thread parallelism. We evaluate our algorithm on a Xeon Phi 7230 KNL processor and show the performance gain when using SIMD instructions and threads respectively. Furthermore, we measure the performance of our algorithm as the data size and the number of threads increase, and study its behaviour. The results reveal that both parallelism methods improve performance. Also they indicate that our algorithm is up to 63x faster than its serial counterpart and behaves well as the workload is increased.

Our work differs from previous studies in two ways. First, our proposed algorithm is based on the Parallel Failureless Aho-Corasick (PFAC) algorithm [7], which efficiently exploits the parallelism of AC. PFAC is suitable for many-core architectures and it was first implemented for GPUs and multicore systems; in the last case vectorization techniques were not applied. Second, in contrast to previous works [17,18], we evaluate the performance and scalability of our proposed algorithm on a Xeon Phi KNL processor.

The rest of the paper is organized as follows. Section 2 provides some background information on pattern matching algorithms and Xeon Phi processors. Section 3 describes our parallel algorithm for pattern matching on Xeon Phi processors. Section 4 shows our experimental results. Finally, Sect. 5 presents the main conclusions and future research.

2 Background

This section describes the AC and PFAC algorithms, the Intel Xeon Phi processor and how to program it.

2.1 The Aho-Corasick Algorithm

The Aho-Corasick (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 AC state machine for the pattern set {he, she, his, hers}. Solid lines represent valid transitions and dotted lines represent failure transitions.

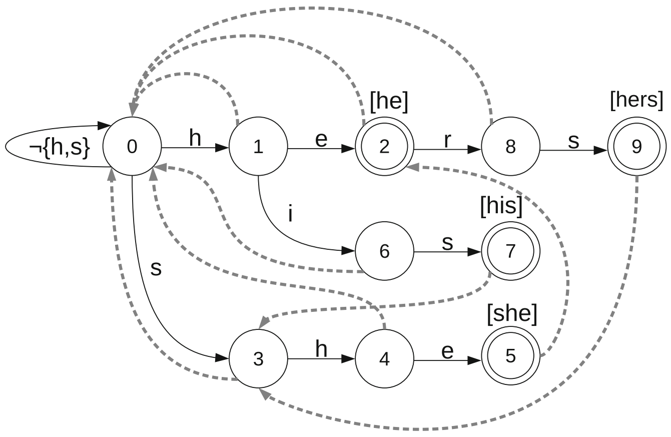

Fig. 1. AC state machine for the pattern set {he, she, his, hers}

2.2 The Parallel Failureless Aho-Corasick Algorithm

The Parallel Failureless Aho-Corasick (PFAC) algorithm [7] efficiently exploits the parallelism of AC and therefore is suitable for many-core architectures. PFAC assigns each position of the text to a particular thread. For each assigned position *start* (task), the thread is responsible for identifying the pattern beginning at that position. For that, the thread reads the text and traverses the state machine, starting from the *initial state*, and terminates immediately when it cannot follow a valid transition; at that point, the thread registers the longest match found. Note that all threads use the same state machine. Since PFAC does not use failure transitions, they can be removed from the state machine. Figures 2 and 3 give an example of the Failureless-AC state machine and the parallelization strategy of PFAC, respectively. Algorithm 1 shows the PFAC code executed by each thread for each assigned task.

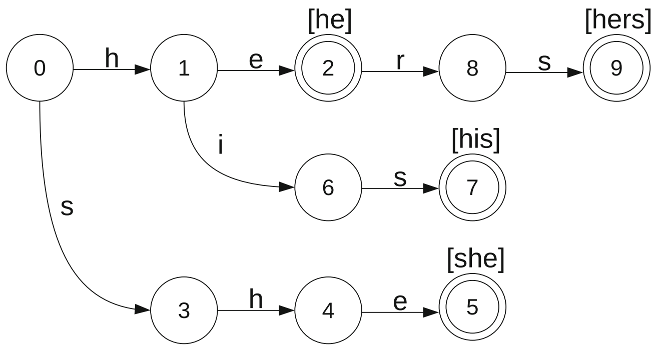

Fig. 2. Failureless-AC state machine for the pattern set {he, she, his, hers}

2.3 Intel Xeon Phi Knights Landing Processor

The Intel Xeon Phi KNL processor [16,19] is composed of many tiles that are interconnected by a cache-coherent, 2D mesh interconnect. Each tile consists of two cores, two vector-processing units (VPUs) per core, and a 1MB L2 cache shared between the two cores. Each core supports 4 hardware threads. Figure 4 illustrates this architecture.

Regarding the memory hierarchy, the KNL processor has two types of memory: (1) MCDRAM, a high-bandwidth memory integrated on-package, and (2) DDR, a high-capacity memory that is external to the package. The MCDRAM can be used as a last-level cache for the DDR (cache mode), as addressable memory (flat mode), or as a combination of the last two modes (hybrid mode) - i.e. using a portion of the MCDRAM as cache and the rest as standard memory. When using the flat or hybrid mode, the access to MCDRAM as memory requires programmer intervention.

Furthermore, the KNL processor can be configured in different modes (cluster modes) in order to optimize on-chip memory traffic.

The KNL processor supports all legacy x86 instructions, therefore it is binary-compatible with prior Intel processors, and incorporates AVX-512 instructions.

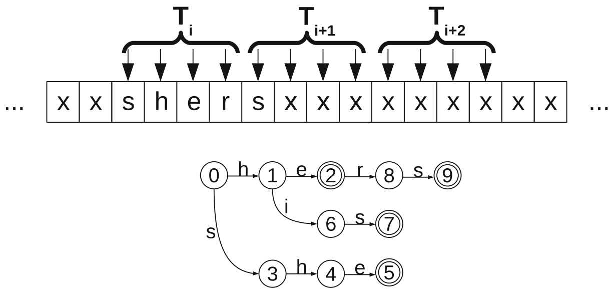

Fig. 3. Parallelization strategy of PFAC

Algorithm 1. PFAC code, executed by each thread for each assigned task

```
     {$start$ :: initial position of the task in the text}
     {$pos$ :: current position in the text}
     {$initial\ state$ :: initial state of the state machine}
     {$state$ :: current state of the state machine}
 1:  pos = start
 2:  state = initial state
 3:  while pos < text size do
 4:     if there is no transition for the current state and input character then
 5:        break
 6:     end if
 7:     state = next state for the current state and input character
 8:     if state is an output state then
 9:        store the pattern found at the position start in output[start]
10:     end if
11:     pos = pos + 1
12:  end while
```

The efficient use of cores and VPUs is critical to obtain high performance. To that end, the programmer must divide the work into parallel tasks or threads, and organize their code to expose vectorization opportunities. Finally, vectorization can be achieved manually, by writing assembly code or using vector intrinsics, or automatically, relying on compiler optimizations.

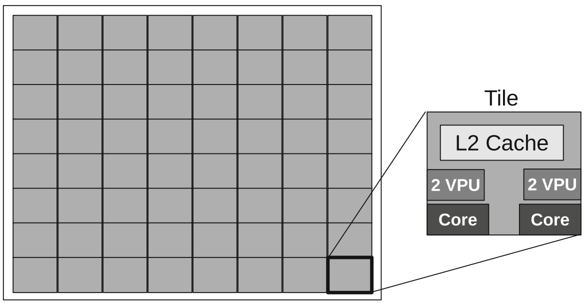

Fig. 4. Intel Xeon Phi KNL processor

3 Pattern Matching on Xeon Phi Processors

This section describes our strategy for parallelizing pattern matching on Xeon Phi processors and some implementation details.

3.1 Parallelization Strategy

Figure 5 depicts the parallelization strategy of PFAC_VEC, our pattern matching algorithm for Xeon Phi based on PFAC. Recall that the strategy followed by PFAC divides the input text into as many tasks as the text size; each task involves identifying the pattern starting at a certain text position; all tasks are equally distributed among threads and each thread follows a sequential control flow to process its tasks. In contrast, the strategy of PFAC_VEC splits the input text into blocks of tasks; each block is composed of a fixed number of consecutive text positions or tasks (this number is referred to as *block size*); all blocks are equally distributed among threads; then each thread processes each assigned block in a vectorized way, i.e. all its tasks are simultaneously solved.

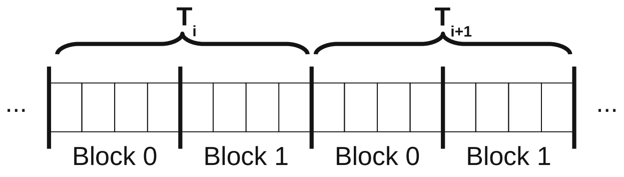

Fig. 5. Parallelization strategy of PFAC_VEC

The distribution of blocks among threads is done through the OpenMP *for* work-sharing construct. On the other hand, vectorization will be performed automatically by the compiler. However, the code executed by each thread to process a block of tasks must meet some requirements to be automatically vectorized [19]. Thus, it requires programmer intervention. The next section presents the implementation details of that code.

3.2 Implementation Details

Algorithm 2 shows the code executed by each thread to process a block of tasks in a vectorized way, taking advantage of the VPUs of the core. Note that all tasks of a block (*block size* in total) are solved simultaneously by applying the same operation to multiple data items. Remember that each task involves identifying the pattern that starts at a certain block position.

The thread first creates vectors of length *block size* to store the current position in the text, the current state of the state machine and the pattern found so far, for each task. The initialization of these vectors is vectorizable (Algorithm 2, Lines 1–5).

Then, the thread processes the tasks until all are finished (Algorithm 2, Lines 6–15). In each iteration of the outer loop, a state transition is carried out for each task. Specifically, the operations of reading (Algorithm 2, Line 8), determining the next state (Algorithm 2, Line 9), verifying if the reached state is an output state (Algorithm 2, Lines 10–12), and incrementing the current position to point to the next character (Algorithm 2, Line 13) are performed in a vectorized way.

Finally, the patterns found in this block are recorded in the output vector in a vectorized manner (Algorithm 2, Lines 16–18).

Algorithm 2. PFAC_VEC code, executed by each thread for each assigned block

```
    {start :: initial position of the block in the text}
    {initial state :: initial state of the state machine}
    {pos :: vector containing the current position in the text for each task}
    {state :: vector containing the current state of the state machine for each task}
    {result :: vector containing the longest match found for each task}
    {For each task, set pos, state and result to their initial values}
 1: for i = 0 to block size - 1 do {Vectorizable loop}
 2:    pos[i] = start + i
 3:    state[i] = initial state
 4:    result[i] = 0
 5: end for
    {Process all tasks simultaneously}
 6: while there is an unfinished task in the block do
       {Perform one more transition for each task}
 7:    for i = 0 to block size - 1 do {Vectorizable loop}
 8:       inputchar[i] = text[ pos[i] ]
 9:       state[i] = next state for state[i] and inputchar[i]
10:       if state[i] is an output state then
11:          store the pattern found at the position "start+i" in result[i]
12:       end if
13:       pos[i] = pos[i] + 1
14:    end for
15: end while
16: for i = 0 to block size - 1 do {Vectorizable loop}
17:    output[start + i ] = result[i]
18: end for
```

In order to guarantee the correct execution of the algorithm, lines 8 and 9 must not fail or have any effect when the current position in the text of task i exceeds the size of the text (pos[i] $\geq$ text size) and when there is not a valid transition for the current state and character of task i, respectively.

To solve the first problem (Line 8), an additional number of special characters was appended to the input text. This number is equal to the length of the longest pattern in the dictionary and represents the maximum amount of characters that can be processed when solving a task.

To solve the second problem (Line 9), a *dead state* was added to the state machine, i.e. a nonaccepting state that goes to itself on every possible input symbol. Also, additional transitions were added to this state from each other state q, on all input symbols for which q has no other transition. Figure 6 shows the Failureless-AC state machine for the pattern set {he, she, his, hers}, which includes the *dead state*.

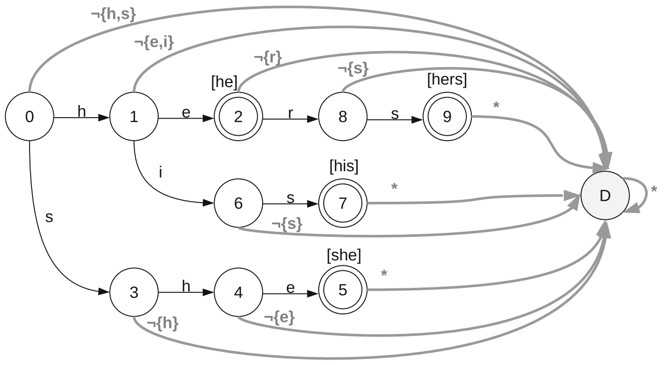

Fig. 6. Modified Failureless-AC state machine for the pattern set {he, she, his, hers}

Now, it can be said that the exit condition of the while loop in Line 6 involves determining if there is a task that has not reached the *dead state* yet. This operation is done by iterating over the vector of current states (the loop is vectorizable).

The characteristics of the proposed algorithm enable the Intel compiler to vectorize the code automatically. In particular, it reports an estimated potential speedup of 10.5x, 10x and 21.3x for the loops in Lines 1, 7, and 16 respectively. Furthermore, it shows a potential speedup of 15.3x for the exit condition of the while loop in Line 6.

4 Experimental Results

Our experimental platform is a machine with an Intel Xeon Phi 7230 (KNL) processor and 128 GB DDR4 RAM. This processor has 64 1.30 GHz cores and 16 GB MCDRAM. Each core supports 4 hardware threads, thus the processor supports 256 threads in total. The processor is configured in flat mode. We use the numactl command to place all data in MCDRAM.

Test scenarios were generated by combining four English texts of different sizes with four English dictionaries with different number of patterns. All the texts were extracted from the British National Corpus [20]: 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); text 4 is a 200-million-word sample

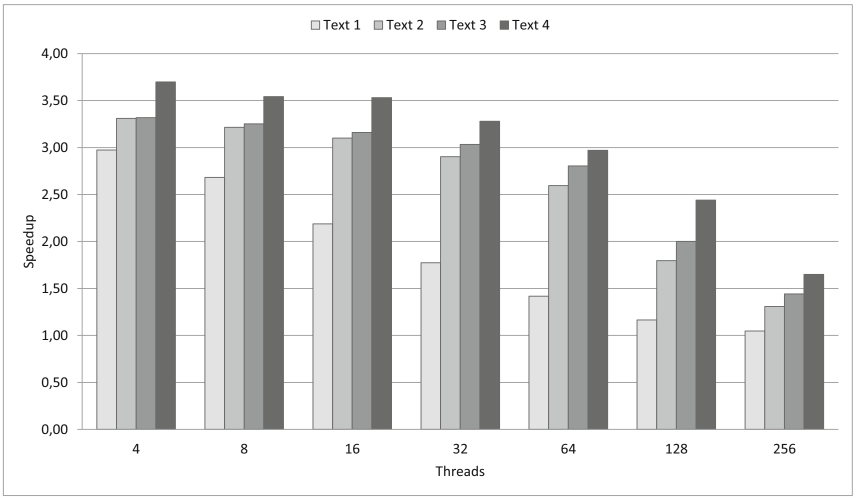

Fig. 7. Average speedup of PFAC_VEC over PFAC

(1090 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.

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.

First, we ran the single-threaded non-vectorized Failureless Aho-Corasick algorithm (SFAC), provided by Lin et al. [7], and the single-threaded vectorized version (SFAC_VEC), presented in this paper. In the last case, different values of *block size* (64, 128, 256) were used. For all test scenarios, the value of *block size* that produces the best result is 128. Using that value, SFAC_VEC is up to 3.6x faster than SFAC, and provides an average acceleration of 3.5 for each text. These results demonstrate that our vectorization technique provides performance gains.

Next, we ran the multi-threaded non-vectorized code (PFAC), provided by Lin et al. [7], and the multi-threaded vectorized code (PFAC_VEC), presented here, for different number of threads (4, 8, 16, 32, 64, 128, 256) and affinity settings [21] (none or default, scatter, compact, balanced). The value of *block size* used by PFAC_VEC is 128.

Regarding the affinity settings, the compact affinity gives the worst results for all tests. The other affinities perform similarly for different combinations of test scenarios and number of threads, except for the largest text considered and few threads, for which the none affinity provides the best execution times. Therefore, from now on, the none affinity will be used.

Afterwards, we evaluated the performance (Speedup[1]) of PFAC_VEC over PFAC and SFAC_VEC, respectively. For each text and number of threads, the average speedup is shown. This is because the speedup does not vary significantly with the dictionary.

Figure 7 illustrates the average speedup of PFAC_VEC over PFAC, for different texts and number of threads. It can be seen that the speedup ranges between 1.05 and 3.70. Higher speedup values are obtained when using fewer threads, and as this parameter increases the speedup decreases. However, for a fixed number of threads, the speedup tends to increase with the size of the text. These results reveal that our vectorization technique is also effective to reduce parallel execution times.

Figure 8 shows the average speedup of PFAC_VEC over SFAC_VEC, for different texts and number of threads. Note that for a fixed text, the speedup first increases with the number of threads and then decreases. However, for a fixed number of threads the speedup always increases with the size of the text. From these results we conclude that PFAC_VEC behaves well as the workload is increased.

Figure 9 shows the best performance (average speedup) of PFAC_VEC over SFAC_VEC for each text, and the number of threads that provides this value. As it can be observed, the number of threads that provides the best performance depends on the text. PFAC_VEC achieves an average speedup of 8.53 for text 1, 38.49 for text 2, 49.03 for text 3 and 62.88 for text 4, using 32, 64, 128 and 128 threads respectively.

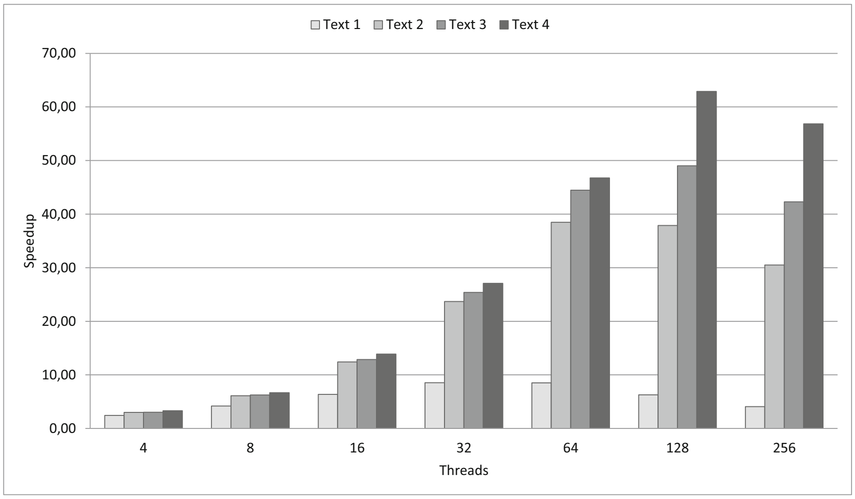

Fig. 8. Average speedup of PFAC_VEC over SFAC_VEC

[1] The Speedup of B over A is defined as $\frac{T_A}{T_B}$, where T_A is the execution time of Algorithm A and T_B is the execution time of Algorithm B.

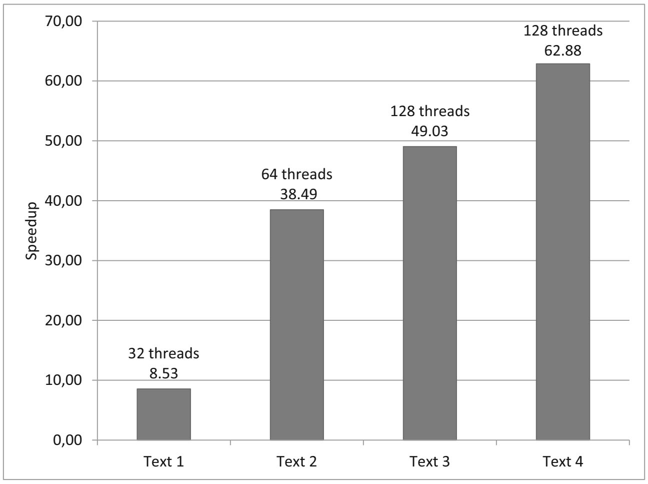

Fig. 9. Best performance (speedup) of PFAC_VEC for each text

5 Conclusions and Future Work

In this paper we presented a novel pattern matching algorithm that efficiently exploits the full computational power of Intel Xeon Phi processors by using both SIMD and thread parallelism. Our proposal is based on the Parallel Failureless Aho-Corasick (PFAC) algorithm.

In summary, our algorithm distributes blocks of tasks among threads. Then, each thread uses SIMD instructions for processing each assigned block (all its tasks are simultaneously solved).

We ran our algorithm (PFAC_VEC) on a Xeon Phi 7230 (KNL) processor and compared its performance with that of the multi-threaded non-vectorized (PFAC) and the single-threaded vectorized (SFAC_VEC) counterparts, respectively.

Experimental results showed that PFAC_VEC outperforms PFAC, indicating that SIMD parallelism is effective to reduce parallel execution times. Furthermore, they reveal that PFAC_VEC is up to 63x faster than SFAC_VEC, demonstrating that thread parallelism is also effective to accelerate pattern matching. Finally, we showed that PFAC_VEC behaves well as the workload increases.

As for future work, we plan to compare the results presented here with those of PFAC for GPU, PFAC for multi-GPU and PFAC for CPU-GPU heterogeneous systems. Also, we plan to apply our proposal to solve more practical problems.

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. White Paper. Sourcefire Inc. 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. Sanz, V., Pousa, A., Naiouf, M., De Giusti, A.: Efficient pattern matching on CPU-GPU heterogeneous systems. In: Wen, S., Zomaya, A., Yang, L.T. (eds.) ICA3PP 2019. LNCS, vol. 11944, pp. 391–403. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-38991-8_26
15. Chrysos, G.: Intel Xeon Phi X100 family coprocessor - the architecture. The first Intel Many Integrated Core (Intel MIC) architecture product. White Paper. Intel Corporation (2012). https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner
16. Sodani, A., et al.: Knights landing: second-generation Intel Xeon Phi product. IEEE Micro **36**(2), 34–46 (2016)
17. Memeti, S., Pllana, S.: Accelerating DNA sequence analysis using Intel(R) Xeon Phi(TM). In: Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA, pp. 222–227. IEEE (2015)
18. Tran, N., et al.: Cache locality-centric parallel string matching on many-core accelerator chips. Sci. Program. **2015**(1), 937694:1–937694:20 (2015)
19. Jeffers, J., et al.: Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco (2016)

20. 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/
21. Vladimirov, A., et al.: Parallel Programming and Optimization with Intel Xeon Phi Coprocessors. Handbook on the Development and Optimization of Parallel Applications for Intel Xeon Processors and Intel Xeon Phi Coprocessors. Colfax International, Sunnyvale (2015)