High Performance Pattern Matching on Heterogeneous Platform

Shima Soroushnia[1,*], Masoud Daneshtalab[1], Juha Plosila[1], Tapio Pahikkala[1] and Pasi Liljeberg[1]

[1]Department of Information Technology, University of Turku, Finland
{shisor, masdan, juplos, aatapa, pakrli}@utu.fi

Summary

Pattern discovery is one of the fundamental tasks in bioinformatics and pattern recognition is a powerful technique for searching sequence patterns in the biological sequence databases. Fast and high performance algorithms are highly demanded in many applications in bioinformatics and computational molecular biology since the significant increase in the number of DNA and protein sequences expand the need for raising the performance of pattern matching algorithms. For this purpose, heterogeneous architectures can be a good choice due to their potential for high performance and energy efficiency. In this paper we present an efficient implementation of Aho-Corasick (AC) which is a well known exact pattern matching algorithm with linear complexity, and Parallel Failureless Aho-Corasick (PFAC) algorithm which is the massively parallelized version of AC algorithm without failure transitions, on a heterogeneous CPU/GPU architecture. We progressively redesigned the algorithms and data structures to fit on the GPU architecture. Our results on different protein sequence data sets show that the new implementation runs 15 times faster compared to the original implementation of the PFAC algorithm.

1 Introduction

In the last decade, the exponential growth of biological information, the need to understand the complex interactions that determine the biological processes and the diversity and interconnectedness of organisms at the molecular level have driven the need for high performance computational biology[1]. Heterogeneous architectures that provide different processing elements with different characteristics on the same machine are becoming mainstream in high performance computing platforms and seem able to cope with these requirements. Such architectures integrate different multicore CPUs with manycore accelerators (GPGPUs). When running an application, some parts can execute on CPU while the other parts of the same application run on the GPU side to benefit from the both processing elements features. Aho-Corasick[2] is one of the widely used pattern matching algorithms in computational biology applications due to its linear complexity and ability to match multiple patterns at the same time. The algorithm works by building a state machine out of patterns and search through the data using the state machine. It can find all the occurrences of the patterns inside the data in a single pass of data and that is the main reason for the linear complexity. We developed a high performance parallel implementation of the AC algorithm for a heterogeneous computing environment. For

*To whom correspondence should be addressed.

this purpose, we use the message passing interface (OpenMP) for parallelizing the building of state machine on shared memory multicore CPU architectures and CUDA for fine grain parallelization of searching on the GPU. To increase the performance of Aho-Corasick algorithm, PFAC[3] which is the highly parallelized version of Aho-Corasick on GPGPUs has been introduced. In PFAC algorithm, the state machine is built without any failure transition and it is placed on the global memory. For searching through the data, a thread will be generated for each single character in the dataset. The total number of threads is equal to the length of text in dataset. Each thread needs to check the automaton in global memory and processes its character to see if it can find a valid transition, otherwise the thread will be terminated. In this paper, we attempt to speed up the PFAC algorithm by rearranging the data placements in the GPU memories. We also used the heterogeneous programming concept for parallelizing the building of the state machine using OpenMP.

2 Related works

Many research papers on accelerating pattern matching using GPUs have been published recently as they are becoming increasingly popular for various applications. Different implementations to the Aho-Corasick algorithm have been proposed to increase the efficiency and throughput of multiple pattern matching problems. Tumeo et al. [4] has introduced a parallel version of the AC algorithm for network based applications that process TCP/IP packets. They assigned a single TCP/IP packet to each CUDA thread, loading it in the shared memory, and then divide the packet in further chunks assigned to each thread of the thread block. Another GPU parallelization of AC algorithm has introduced in [5]. They tested their implementation assuming the case when all data reside initially in the GPU memory and the results are to be left in this memory. In this implementation, AC DFA (Deterministic Finite Automaton) is stored in the texture memory. Threads cooperatively read all the data needed to process a block, store this data in shared memory, and finally read and process the data from the shared memory. They extended their idea in [6] and tested the AC and Boyer-Moore, which is another exact pattern matching algorithm that performs the character comparison in reverse order to the desired keywords [7], in two cases. In the GPU-to-GPU case, they considered several refinements to their base GPU implementation and measured the performance gain from each refinement. For the host-to-host case, they analyzed different strategies to reduce the communication cost between the host and the GPU. Tran et al. [8] has developed a memory efficient parallelization of the AC algorithm by placing the state transition table on texture memory and loading the data chunks from global memory to the shared memory. They gained 15 times speedup comparing to the serial execution of the algorithm. Gnort [9] is another GPU implementation of the AC algorithm for Network Intrusion Detection Systems (NIDS). Each packet is copied to the shared memory of the Multiprocessor and stream processors search different parts of the packet concurrently. Another implementation based on Memory Page Size is introduced by Peng et al. [10]. They split the large scale text into TextSize/PageSize size-fixed pages. All the data are transferred to the GPU memory and different threads on GPU process different pages using the automaton which is bound to texture memory. As an experiment of parallelizing the AC algorithm on heterogeneous CPU-GPU platform, Tumeo et al. [1] has implemented the algorithm on GPU clusters for the use of DNA analysis applications. They developed a master/slave implementa-

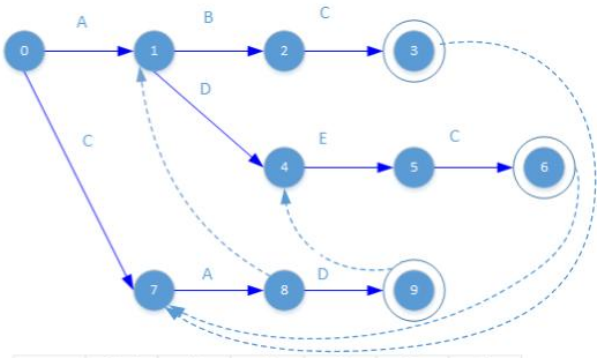
tion, where the master MPI process distributes the work to the various nodes and GPUs, and the slaves perform the effective computation. The GPU parallelization is based on partitioning the input text in multiple chunks and assigning each chunk to a single CUDA thread. To catch the patterns that go over the border of a single chunk, the chunks can be overlapped for the size of the longest pattern in the dictionary. The state transition table is bound to the texture memory. Also Tumeo et al. [10] have extended their previous work [4] on heterogeneous architecture by developing a master/slave scheduler, where the master MPI process distributes the work, and the slaves perform the effective computation. PFAC (parallel failure less Aho-Corasick) [11] is a highly parallelized implementation of the AC algorithm without any failure transition. In the PFAC algorithm, each character in the text is assigned to a thread and each thread is only responsible for finding the patterns starting at its own starting position. The state transition table is placed in the texture memory and data is preloaded to the shared memory. The PFAC algorithm is then used for implementing serial and parallel Bayesian spam filtering [12]. In this work, we have modified the AC algorithm by rearranging the placement of data in different GPU memories comparing to PFAC. We also include heterogeneous programming concept by using OpenMP interface for building the state machine on a multi-core CPU.

3 Aho-Corasick algorithm

A multi-string matching problem is defined as having a set of patterns $P_i = \{P_1, \ldots, P_n\}$ and we would like to detect all the occurrences of any of the patterns in P in a text stream T. The Aho-Corasick solution for this problem is to build a Finite State Automaton with the failure transitions, named pattern matching machine, and run this machine on the text to find any existence occurrence of the patterns inside the text. The algorithm has two main steps. First step is building the automaton out of the keywords we need to search for and second step is traversing the automaton for each character we are reading from the input file to find the matches. The automaton can be built as a keyword trie in which the nodes represent the state and the edges between nodes are labeled by characters that cause the transitions between nodes. These edges are categorized in two types. The forward transition edges and the backtrack edges that are the output of the failure function. Given a state S that can be reached by traversing the pattern matching machine using the forward transitions to find the pattern P, the failure function output for the S is the state that can be reached by longest suffix of P. That mean considering the cases that the pattern we reached, is a prefix for another pattern in pattern matching machine.

The trie is an efficient implementation of a dictionary of strings. It is also easy to implement a state machine as a lookup table where the rows represent states and the columns represent input character, so that each element of matrix shows the next state for the corresponding alphabet in the current state. Having the information about the current state and next input character, the machine can determine whether the input character causes a failure transition. If not, then it makes a transition to the state corresponding to the character.

The complexity of constructing a pattern matching machine and searching the text is linear to the total length of given patterns and the length of a text, respectively. Figure 1 shows an example of how the algorithm works. The solid arrows in the state machine represent the forward transitions and the dotted arrows represent the failure transitions.



	Final	A	B	C	D	E
0	0	1	0	7	0	0
1	0	0	2	0	4	0
2	0	0	0	3	0	0
3	1	7	7	7	7	7
4	0	0	0	0	0	5
5	0	0	0	6	0	0
6	1	7	7	7	7	7
7	0	8	0	0	0	0
8	0	1	1	1	9	1
9	1	4	4	4	4	4

Figure 1: Aho-Corasick State machine and transition table for finding key-words: ABC, ADEC, CAD.

In the example, the keyword set which we need to construct the automaton for is {ABC, ADEC, CAD}. We start with the root node and insert each pattern after another. For each pattern Pi, we need to start from the root and follow the path labeled by the characters of Pi. If the path ends before we check all the characters of Pi, we have to add new nodes for the rest of the characters. The last character of Pi should be marked as a final node to detect matches. It can be seen that the time complexity of building the state machine is linear to the sum of the length of all the keywords. The state machine then will be mapped to a lookup table. In each state which is represented as a row, we can find the suitable transition by checking the column which is matched the character we are facing. After building the state machine, the next phase is to pass the state machine through the input text. We start from the root and for each character in the text we follow the path in the state machine. If we reach to a final node, we will report a match and otherwise the search will be terminated. For the search action the complexity will be to the length of the text.

4 Implementation

Most of the previous parallelizations of Aho-Corasick algorithm are based on the method that input stream is divided to the chunks and each chunk is assigned to a separate thread to search through. An example can be seen in Fig. 2. It illustrates how the data parallelization method can speed-up the application.

In theory, finding the pattern in parallel with four threads will be four times faster than doing it sequentially since the four threads are doing the same search on different chunks of data simultaneously instead of doing the whole action with a single thread. However, this implementation

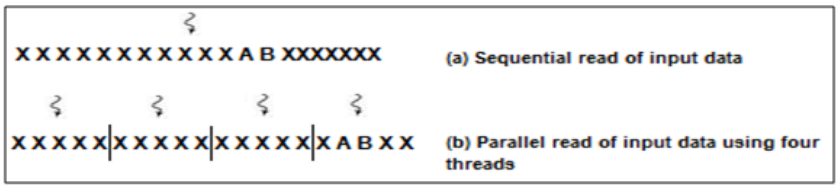


Figure 2: Sequential vs. parallel search.

will face problem when patterns are occurred in the boundary of data as shown in Fig.3. In this case, none of the threads searching the data chunks which include a part of the keyword can detect the keyword and the search will miss all the keywords occurring in the boundaries.

Figure 3: Boundary problem.

In this case, the only way to solve this problem is to assign overlapped data chunks to the threads and the size of this overlap should be equal to the size of the longest keyword. Fig.4 shows how data overlapping can solve the boundary problem. But this approach will lead to significant performance reduction, specifically in GPU implementation that memory access is the main issue and has considerable effect on the overall throughput of the system.

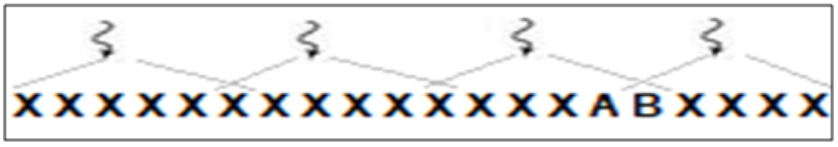


Figure 4: Threads with overlapped data to search through.

To overcome this problem, the PFAC algorithm is introduced [11]. In this algorithm, every character in the input stream is assigned to a thread which is only responsible to find the keywords beginning at its own starting position. If a thread does not have any valid transition for the character, the thread will be terminated immediately. But in case of having a valid transition, thread will continue reading the next character.

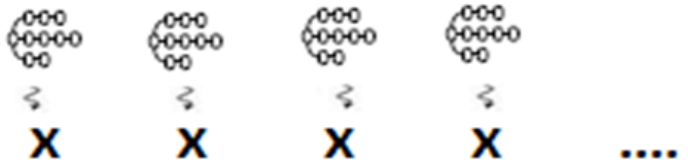


Figure 5: PFAC algorithm. Generating a thread for each character in the input stream.

One of the most important features of PFAC algorithm is that it removes the failure transition from the Aho-Corasick state machine by assigning a thread for each character. Each thread will only find its own keyword that starts in its own starting position, so there is no need to have failure transition anymore. Also the boundary problem that is introduced in the chunk based approach is somehow solved here as we are not assigning the data chunks to the threads. Fig. 6 shows how PFAC algorithm detects keywords with its failureless state machine.

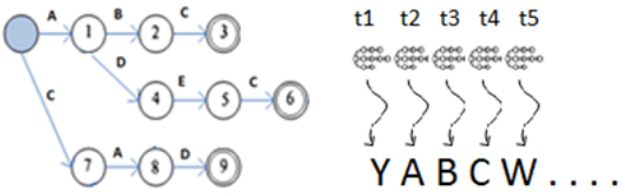


Figure 6: PFAC Finding the patterns with failureless state machine.

In the above example, assume that we are searching for the keyword ABC in the given text. A thread is generated for each character in the text and each thread starts from the root and searches for the keywords starting with its own starting position. First thread which reads Y as its starting character will be terminated in the first stage as there is no valid transition for Y in the state machine. Second thread which starts with A, will go to state 1 as its next transition and reads the next character which is B. B then causes the thread to move to the second state and by reading next character which is C, thread will report a match while it is in third state. Next character for t2 is W which causes the termination of the thread as there is no valid transition from state 3 for W. Next thread which is t3, can not find any valid transition for its starting character and will be terminated in the first stage. The forth thread will move to state 7 after reading its first character which is C, but after reading its second character which is W, it will be terminated since there is no valid transition for W from state 7. The implementation of the PFAC algorithm is in a way that the state transition table is placed in the texture memory, which is a part of global memory with an internal 6KB cache in each streaming processor. Then the input data will be first transferred to the GPUs shared memory and then searching through the data will be started. Although PFAC eliminates the boundary problem between threads, there are still boundary problems while prefetching data to the GPUs shared memories and patterns that are occur in the boundary of data chunks that are going to be loaded to the shared memory will not be detected without use of overlapping method again [13].

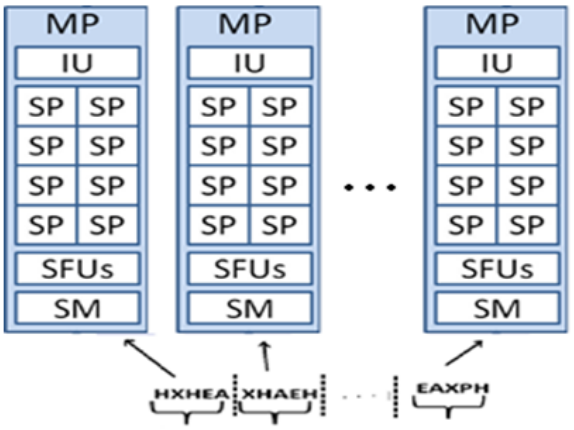


Figure 7: Boundary problem in PFAC algorithm.

An example is shown in Fig. 7. Pattern AX in the boundary of first two segments will not be found without overlapping the data chunks. Another issue about the PFAC algorithm is the memory accesses. Considering the fact that each thread needs to check the next transition for its input character from transition table which is placed in the texture memory, and also very limited size of texture cache, for most of the input characters we need to have two memory accesses. One will happen when reading the character and the other one when checking the transition table for that character. This problem will become more serious when we have large set of patterns to search for, which leads to have a larger transition table and more global memory accesses. Our solution for the aforementioned problem is to store the transition table in the shared memory of each streaming processor. Threads then start reading the input file as it is explained in the PFAC algorithm. For limiting the global memory access to once per thread, we use the fact that threads inside each streaming processor can communicate with each other via the shared memory. We allocate a small part of the shared memory for communication between the threads in each streaming processor, so that whenever a thread finds its next transition by checking the transition table from the shared memory, it will be assigned the value of next thread for continuing its search. This trend will be continued till each thread finds the keyword starting at the threads starting position or the termination of the thread due to not having the next transition. An example of this method is illustrated in the Fig.8. Assume that we are searching for pattern AB, all the threads will read their value from the global memory and start searching for the next transition by checking the transition table in the shared memory. As the thread with the value A will have a next transition and needs B to be its new value, it can have the value of its next thread instead of reading the value from the global memory.

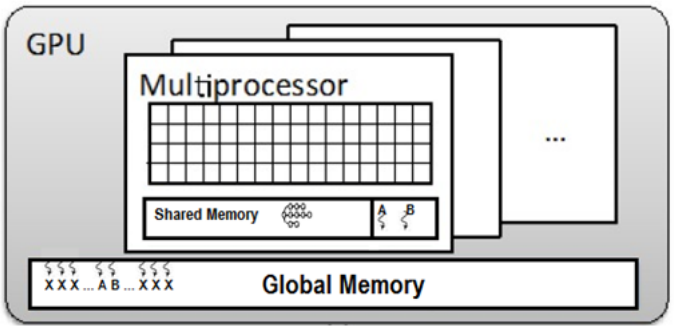


Figure 8: Storing the transition table of State machine in shared memory.

This implementation leads to reduction of memory accesses to once per character. Also, as the threads access to contiguous global memory locations, memory accesses will be coalesced, that means the device coalesces global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth. Hence we can use the maximum bandwidth for fetching data from the global memory and reducing the global memory latency as much as possible.

5 Experimental results

In this section we will discuss the performance of our heterogeneous implementation with respect to the original implementation of PFAC algorithm. The result for the modified PFAC algorithm without including the OpenMP part is also presented here to have a better overview of our modification results. Table 1 shows the comparison between the execution time of different protein datasets on different implementation of the algorithm. We also test the sequential version of the algorithm with a single thread on CPU. That means a single thread builds the state machine and starts to search the input data character by character. In sequential version, the state machine should include the failure function. The memory efficiency of GPU implementation is much more than sequential version due to removing the failure transitions. We selected a few sets of protein databases for different animals to assess the presented method. Execution time and system throughput for different implementations on NVIDIA Tesla C2075 and Intel Xeon Processor E5-2620 are shown below.

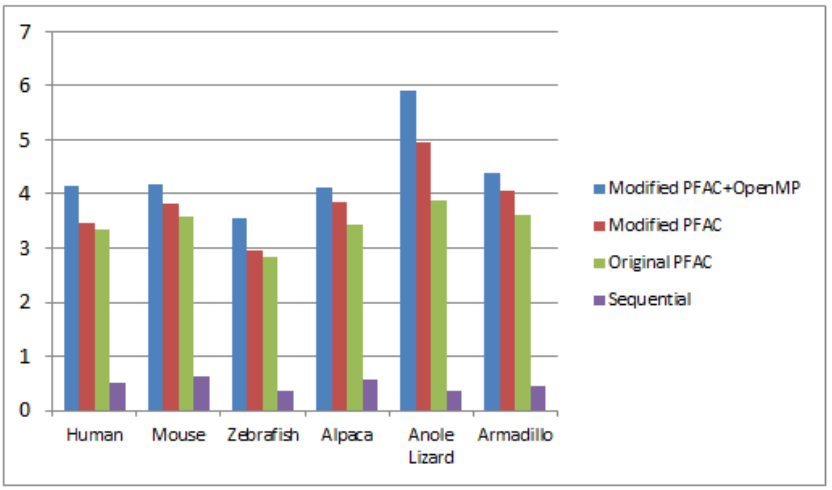


Figure 9: System throughput (Gbps).

Table 1: Execution time for each implementation (DB1: Human (26.635MB), DB2: Mouse (28.057MB), DB3: ZebraFish (22.59MB), DB4: Alpaca (28.914MB), DB5: Anole Lizard (59.714MB), DB6: Armadillo (31.04MB))

	Modified PFAC + OpenMP	Modified PFAC	Original PFAC	Sequential
DB1	6.23	7.08	7.94	53.06
DB2	6.42	6.9	7.83	55.02
DB3	6.05	7.02	7.95	60.04
DB4	6.91	7.12	8.43	49.65
DB5	9.36	11.88	15.42	107.05
DB6	6.7	7.32	8.62	67.32

Results show the average of 15% speedup for our implementation compared to to the original PFAC implementation. Figure 9 shows the system throughput calculated using the formula $inputsize/(T(Total))$ in Gigabits per second (Gbps). Increasing the size of the data set leads to have higher throughput.

As can be seen from the figure, the modified PFAC has higher throughput as the result of reducing the memory accesses by placing the transition table on the shared memory. The effect of the memory access reduction will be more visible while searching for the longer strings. Longer strings make larger transition table and so need more memory access.

6 Conclusion

In this paper, we have proposed a heterogeneous parallelization of AC algorithm. We modified the PFAC algorithm by rearranging the data placement in the GPU memories and using

OpenMp for parallelizing the building of the state machine. We compared our implementation with the original PFAC algorithm. The experimental results show the average of 15% speedup.

References

[1] A. Tumeo and O. Villa. Accelerating DNA analysis applications on GPU clusters. In *IEEE 8th Symposium on Application Specific Processors (SASP)*, pages 71–76. 2010.

[2] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, 1975.

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

[4] A. Tumeo, O. Villa and D. Sciuto. Efficient Pattern Matching on GPUs for Intrusion Detection Systems. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 87–88. ACM, New York, NY, USA, 2010.

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

[6] X. Zha and S. Sahni. GPU-to-GPU and Host-to-Host Multipattern String Matching On A GPU. *IEEE Transactions on Computers*, 62:1156–1169, 2012.

[7] J. S. M. Robert S. Boyer. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[8] N.-P. Tran, M. Lee, S. Hong and M. Shin. Memory Efficient Parallelization for Aho-Corasick Algorithm on a GPU. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 432–438. 2012.

[9] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In R. Lippmann, E. Kirda and A. Trachtenberg (editors), *Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*, pages 116–134. Springer Berlin Heidelberg, 2008.

[10] J. Peng, H. Chen and S. Shi. The GPU-based String Matching System in Advanced AC Algorithm. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1158–1163. 2010.

[11] C.-H. Lin, C.-H. Liu, L.-S. Chien and S.-C. Chang. Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs. *Computers, IEEE Transactions on*, 62(10):1906–1916, 2013.

[12] S. Haseeb, M. Motwani and A. Saxena. Serial and Parallel Bayesian Spam Filtering using Aho-Corasick and PFAC. *International Journal of Computer Applications*, 74(17):9–14, 2013.

[13] C. Agarwal, A. Rasool and N. Khare. PFAC Implementation Issues and their Solutions on GPGPUs using OpenCL. *International Journal of Computer Applications*, 72(7):52–58, 2013.