PAPER

Accelerating Weeder: A DNA Motif Search Tool using the Micron Automata Processor and FPGA

Qiong WANG[†a)], *Member*, Mohamed EL-HADEDY[††b)], Kevin SKADRON[†††c)], *and* Ke WANG[†††d)], *Nonmembers*

SUMMARY Motif searching, i.e., identifying meaningful patterns from biological data, has been studied extensively due to its importance in the biomedical sciences. In this work, we seek to improve the performance of Weeder, a widely-used tool for automatic *de novo* motif searching. Weeder consists of several functions, among which we find that the function namely oligo_scan, which handles the pattern matching, is the bottleneck especially when dealing with large datasets. Motivated by this observation, we adopt the Micron Automata Processor (AP) to accelerate the pattern-matching stage of Weeder. The AP is a massively-parallel, non-von-Neumann semiconductor architecture that is purpose-built for symbolic pattern matching. Relying on the fact that AP is capable of performing matching for thousands of patterns in parallel, we develop an AP-accelerated Weeder implementation in this work. In particular, we describe how to map Weeder's pattern matching to the AP chip and use the high-end FPGA on the AP board to postprocess the output from AP. Our experiment shows that the AP-accelerated Weeder achieves 751x speedup on pattern matching, compared to a single-threaded CPU implementation.

key words: *Automata processor, weeder, motif search, FPGA*

1. Introduction

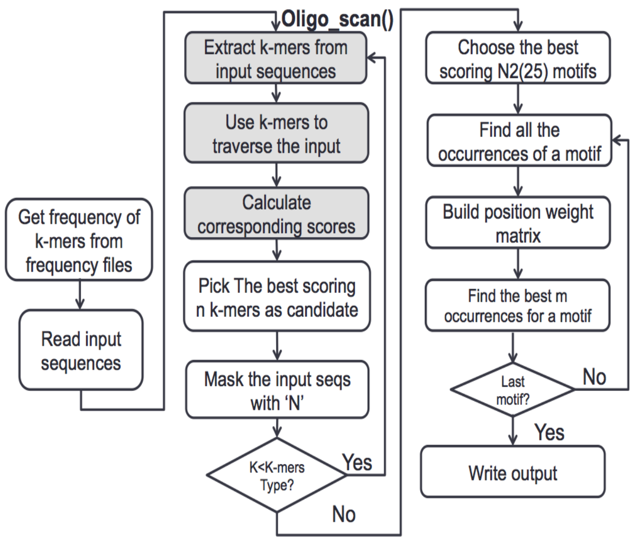
Unraveling the complex mechanisms that regulate gene expression is an important problem as we are entering the era of large-scale genome sequencing. Transcription-factors (TF) are the specific proteins that bind to DNA to control the gene expressions by activating or inhibiting the transcription machinery. The short segments of the DNA which transcription-factors bind to are called transcription-factor binding sites (TFBS). They usually range in size from 8-10 to 16-20bp. Functionally related DNA sequences generally share some common sequence elements such as binding sites. Therefore, accurately extracting consensus motifs for DNA-binding sites may greatly help scientists predict genes and genes functions. However, it is difficult to predict TFBS instances for a given TF due to the fact that the sites recognized by the factor are similar but variable. In particular, they usually differ in one or more nucleotides from each other.

Motif Searching. Roughly speaking, motif searching in biological sequences could be considered as the problem of finding short similar, but not necessarily identical, sequence elements shared by a set of nucleotide or protein sequences with a common biological function. To be more precise, motif searching is to find a substring of length k that occurs in a set of input sequences with up to d mismatches. In computational genomics, k-mers refer to all the possible subsequences (of length k) from a read obtained through DNA Sequencing. For example, consider three input sequences ACGTATCA, GAACATAT, and CACGTCAG. Suppose k=6, d=1. The 6-mer ACGTAT is one of the motifs of the given 3 input sequences. It appears at the first position in the first sequence with no mismatches, at the third position in the second sequence with one mismatch, and at the second position in the third sequence with one mismatch.

Over the past few years, numerous algorithms have been implemented and applied to motif search. Most of the motif-searching tools can be categorized into two major groups based on the combinatorial approach used in their design: 1) word-based (string-based) methods that mostly rely on exhaustive enumeration, i.e., counting and comparing oligonucleotide frequencies and 2) probabilistic sequence models where the model parameters are estimated using maximum-likelihood principle or Bayesian inference. Some examples of profile-based algorithms are MEME and Gipps Sampling [3], [4], [6], [7], [11]. Planted Motif Search (PMS), also known as (k, d) motif search, is a method for identifying all substrings of length k which appear in all the input sequences with at most d mismatches. Note that these solve different problems. MEME determines how well a motif conserved in a set of input sequences, while PMS is a theoretical study targeting to find the largest k and d in a shortest time. Indranil et. al proposed an exact algorithm for solving the PMS problem using the AP [10].

Weeder. In this work, we target Weeder [9], a de novo motif-finding software package. It is one of the widely-used tools for de novo motif discovery [7], [12]. It is a consensus-based method that enumerates exhaustively all the oligonucleotides up to user-defined length and calculates their occurrence frequency in a set of input sequences. Each motif is evaluated according to the number of sequences where it appears and how well conserved it is in each sequence, with respect to values calculated by comparing to a species-specific background model, built from the oligonucleotide distribution of all promoter (or intergenic) regions available for different species. The default lengths considered are 6, 8, and 10, with at most 1, 2, and 3 substitutions in the motif occurrences,

[†]College of Computer, National University of Defense Technology, China
[††]Coordinated Science Lab, University of Illinois Urbana-Champaign, USA
[†††]Department of Computer Science, University of Virginia, USA
a) E-mail: wangqiong@nudt.edu.cn
b) E-mail: hadedy@alumni.ntnu.no
c) E-mail: skadron@virginia.edu
d) E-mail: kw5na@virginia.edu
DOI: 10.1587/transinf.E0.D.1



Oligo_scan()
Extract k-mers from input sequences
Use k-mers to traverse the input
Calculate corresponding scores
Pick The best scoring n k-mers as candidate
Mask the input seqs with 'N'
K<K-mers Type?
Yes
No
Get frequency of k-mers from frequency files
Read input sequences
Choose the best scoring N2(25) motifs
Find all the occurrences of a motif
Build position weight matrix
Find the best m occurrences for a motif
Last motif?
No
Yes
Write output

Fig. 1 Flowchart of Weeder

respectively. Users could change k and d according to their purpose, but they also need to provide the corresponding oligo frequency files. The candidates are evaluated through a statistic model (Equation 1 in Section 4.2) and the best-scoring ones are collected to get the candidates for next run. Finally, best instances of each motif are computed.

Weeder narrows down the candidates to prune the search space. As shown in Fig. 1, Weeder first analyzes the species-specific background frequency files, which are needed in the evaluation stage of the motifs. Then it takes every k-mer (6-mers in the beginning) of the input sequences as a motif candidate, to traverse the input sequences sequentially. Each time it finishes a match between the k-mer and an input sequence, it calculates a score according to the number of times it appears, how well it is conserved, and the values computed from frequency files. Then it picks the best-scoring k-mers, based on which it generates a group of (k+2)-mers as the candidates for the next run. After it finishes the matching stage for the longest k-mers (10-mers in default), Weeder chooses the best-scoring motifs, which are more likely to represent conserved TFBSs. For these motifs, a Position-Weight Matrix (PWM) is built to represent the frequency of four possible nucleotides appearing in each position of them. Using a PWM, the most likely location of the motif within each sequence can be calculated.

Motivation of This Work. Since Weeder relies on exhaustive search, it needs to take all the substrings of length k as candidates and use them to traverse all the input sequences sequentially. Oligo_scan is the pattern matching function through which k-mers are traversed for each input sequence. This becomes prohibitively time-consuming as the number of input sequences becomes large. In other words, the function Oligo_scan is the bottleneck of Weeder when dealing with large datasets. For example, we use the dataset from DREAM Challenges [1] to evaluate the time cost of Oligo_scan. The length of each input sequence is

Table 1 Time for Weeder (s)

Number of input sequences	Total time	Time of Oligo_scan
5000	488	306
8300	1145	705
16000	3718	2226
166666	95050	37919

60 base pairs. We run the test on an Intel Core i7-3770 CPU with 8GB memory. Table 1 shows the running time of oligo_scan, from which we can see that when the dataset becomes larger, the time increases sharply. Accelerating the pattern-matching stage therefore plays a significant role in improving the performance of Weeder.

Our Contributions. In this work, we evaluate the ability of the Micron Automata Processor (AP) [2] to accelerate the pattern matching stage of Weeder. The AP is a hardware implementation of non-deterministic finite automata (NFA), with the first-generation boards supporting concurrent matching against approximately 1.5 million states. This allows testing of thousands of patterns in parallel, which makes it an ideal processor for massively-parallel pattern matching in Weeder. Relying on this fact, we develop an AP-accelerated Weeder implementation. In addition to the AP chips, we use the high-end FPGA on the AP board to postprocess the data output from AP. Our experiment shows that the AP-accelerated Weeder pattern matching achieves 751x speedup when compared to a single-threaded CPU implementation.

Organization. The rest of this paper is organized as follows. Section 2 briefly introduces the Micron Automata Processor (AP). Section 3 describes our proposed AP-accelerated Weeder. The performance of our solution is evaluated in Section 4. And finally in section 5, some conclusions are given and our future work is listed.

2. Micron Automata Processor

Micron has announced and provided preliminary software for a new accelerator, the Automata Processor (AP) [2], which is a highly-parallel, reconfigurable, non-Von Neumann architecture designed for execution of Non-Deterministic Finite Automata (NFA). It is purpose-built to address the challenges associated with symbolic pattern matching, which arises in regular-expression processing [17], pattern mining [14], text mining [13], and various other forms of data analytics [16].

The architecture of the first-generation AP, i.e., D480, is depicted in Figure 2. Below we briefly introduce its main components and working principle. More details about the AP could be found in the work [5].

2.1 The Automata Processor Elements

The AP chip consists of three fundamental types of functional elements: State Transition Elements (STE), Counters, and Boolean elements. Counters and Boolean elements are

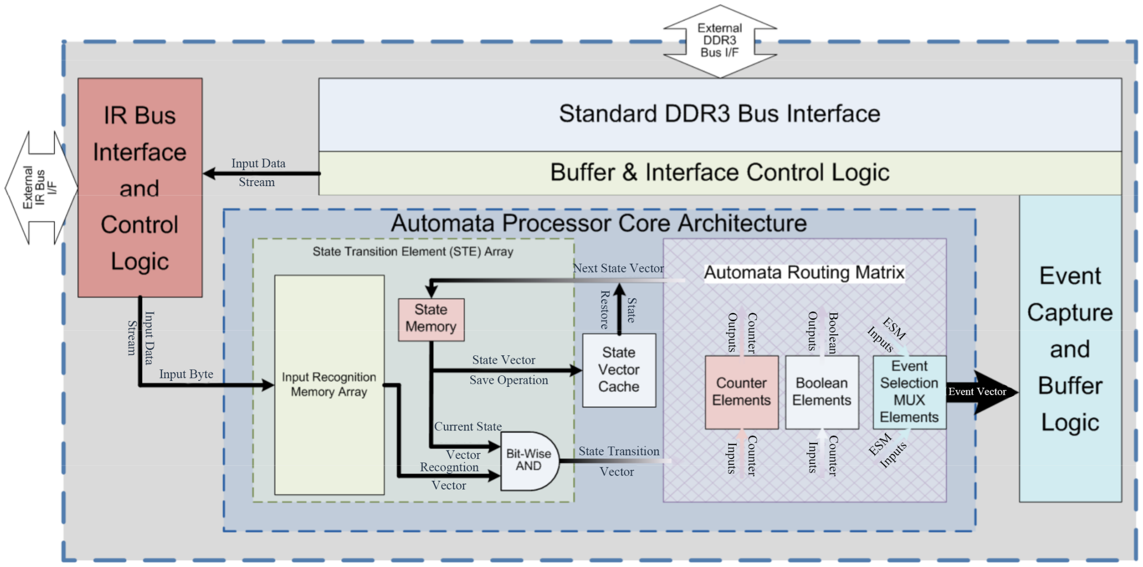


Fig. 2 Architecture of an AP chip [8]

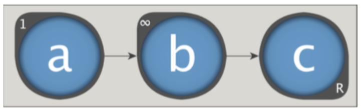


Fig. 3 An automaton consisting of three STEs

used with STEs to extend computational capabilities beyond NFAs.

In our work, we only use the STEs, which are the most important elements of the AP, because they represent the fundamental state and transition aspects of NFA execution.

STEs are based on the homogeneous NFA model, in which the matching operation is denoted in the state, and a match activates all successor states. Each STE can be programmed to match on any subset of the 8-bit ASCII character set. By default, all STEs are initialized in an inactive state, except those marked as *start* states (marked with "1" in the AP workbench) or those configured to always be active (marked with "∞ "). STEs in principle can have an arbitrary number of successor states, including itself, all of which are activated when that STE matches; however, hardware limitations impose some limits on the fan-out and fan-in for STEs. Figure 3 shows an very simple example automaton consisting of three STEs to accept "abc" at the very beginning or "bc" anywhere in the input stream.

2.2 Programing

The Automata Network Markup Language (ANML), an XML-based language for describing the composition of automata networks, contains elements that represent automata processing resources. Using ANML, a programmer can explicitly describe how these automata processing resources are connected together to create an automata network by configuring the elements and the connections, and providing input and allowing the automata network to compute. Micron also provides a graphical interface called the AP Workbench, which is for automaton design and debugging. It allows users to develop small automata rapidly and gives a direct understanding of the implementation of the ANML files. However, it is not suitable for large automata designs. AP SDK enables programming and operation of the AP hardware under Windows and Linux system. It provides API interfaces for C/C++, Python, and Java to develop Automata network.

2.3 Input and Output

The AP takes input streams of 8-bit symbols. The output is generated by the reporting elements (marked with "R"). If an element is configured as a reporting element, it will generate a one-bit signal once it is activated. All the reporting results will be buffered in the output event memory, which can be read by the FPGA on the AP board through a DDR interface.

It is important to note that an input stream is sent to all participating AP chips in parallel, so on every clock single, every STE on every AP chip sees a the same, next input symbol. The board can process four input streams in parallel, with each input stream sent to a different rank.

2.4 AP Chip and Board

An AP board consists of 4 ranks, each of which has 8 AP chips. Micron's current-generation AP-D480 chip is built on 50nm DRAM technology running at an input symbol rate of 133 MHz. Each column in the DRAM arrays represents an STE. The AP achieves its massive parallelism by using the input symbol to activate all the corresponding rows of the DRAM, thus reading out the response of all the STEs to that input symbol.

Each AP board is capable of processing up to 4 separate data streams concurrently. The D480 chip has two half cores and each half-core has 96 blocks. Each block has 256 STEs, 4 counters, and 12 Boolean elements. In total, one D480 chip has 49,152 STEs, 2,304 Boolean elements, and 768 counter elements. Since the AP takes the input 8-bit symbols each cycle, this time is called symbol cycle. The symbol cycle is 7.5 ns. The board is a PCI-Express board with a x8 interface. In addition to the AP ranks, the board contains an FPGA that acts as the PCI host, implements the interfaces to the AP ranks, and provides spare capacity that can be used for user functionality. A programming interface has not yet been made available for programmers to access the FPGA, so our work uses a separate FPGA to model the performance that can be achieved with the AP and on-board FPGA working in concert.

3. AP-accelerated Weeder

In this section, we describe how to accelerate Weeder using the AP.

3.1 Flowchart

The term k-mer refers to the motif of length k. The default k considered are 6, 8, and 10, with d= 1, 2, and 3 respectively in Weeder 2.0. As shown in Fig. 4, the AP-accelerated Weeder consists of four major parts:

- Preprocess (T1) : Except for the first round, whose candidate k-mers are the subsequences of the input sequences, the candidate motifs are generated based on the previous best-scoring motifs. Other data needed to calculate the scores are also preprocessed in this stage.
- Symbol replacement (T2): In some cases, the number of candidate motifs exceeds the capacity of the AP board, requiring multiple passes of matching. If the connection among the STEs stays the same, AP chips do not need to be recompiled. The symbol-matching rules of the STEs merely need to be updated with new candidate k-mers. The symbol replacement mechanism provides a fast way with maximum 0.05s each time.
- Pattern matching (T3): Streaming in the input sequences and matching the k-mers on AP chip. The input sequences are connected as a long sequence, where a symbol is inserted between them as the separator. The automata on the AP chips will compare the input sequences with the candidate k-mer and produce a report event every time it finds a substring of the input sequence whose Hamming distance from the k-mer is less than the maximum mismatch. AP records the position of each match in the input sequences and the corresponding reporting STE ID.
- Postprocessing (T4): After the AP finishes the pattern matching, it gets the number of occurrences and the corresponding mismatches. Our FPGA circuit then reads the output results and calculates the scores for k-mers in parallel.

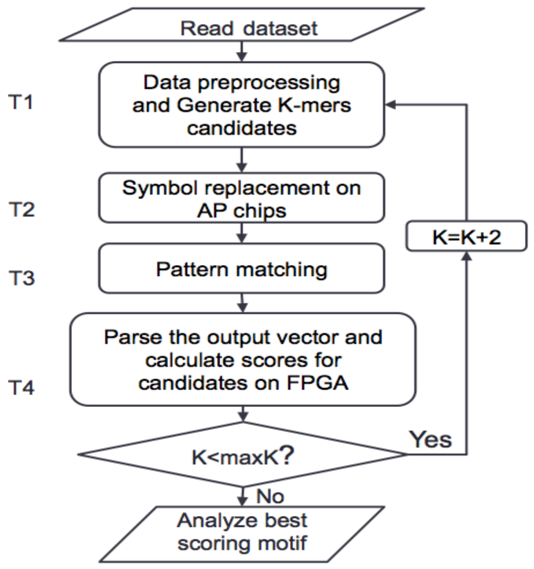

Read dataset
T1
Data preprocessing and Generate K-mers candidates
T2
Symbol replacement on AP chips
T3
Pattern matching
T4
Parse the output vector and calculate scores for candidates on FPGA
K<maxK?
Yes
K=K+2
No
Analyze best scoring motif

Fig. 4 Flowchart of AP-accelerated Weeder

3.2 Automaton Design

The Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. The pattern-matching module implemented on the AP uses the Hamming distance automata constructions described in [10]. Fig. 5 shows an automaton that is for finding the occurrences of ACGTAT in the input sequences with maximum 1 mismatch (here d=1). The last STEs in each row are the reporting elements, which will report every time it finds a match. The last STE in the first row reports when it finds an exact match, and the last STEs of the second and third rows find a match with 1 substitution. This is a basic automaton structure for one k-mer. The automaton, which accepts strings of length k with d mismatches, has $2d + 1$ rows of STEs and arranged in k columns requiring $(2d+1)k-d^2$ STEs in total. At the beginning of the pattern-matching stage, for the candidate k-mers, the AP will be compiled to have as many of these Hamming automata as possible, all of which have the same structure but differ in the specific k-mer pattern they match. They will be traversed by the input sequences in parallel. In the subsequent runs, the AP does symbol replacement for new candidates of the same length.

3.3 Postprocessing on the FPGA

After the AP finishes the pattern matching, the results are read by the FPGA to do the postprocessing. The output of AP contains two parts: the offset arrays which record the

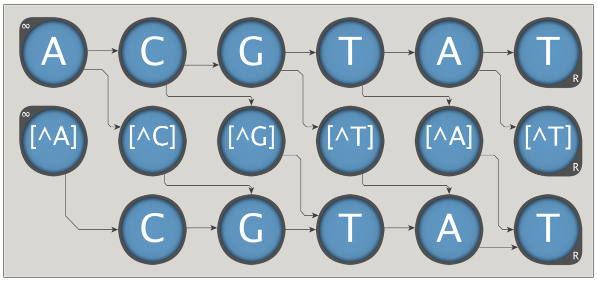


Fig. 5 An automaton to accept ACGTAT with maximum 1 mismatch

offset of the input stream whenever there is a report among all the report elements, and the output vector array in which every bit represents a report element's state (the bit will be "1" if a report element reports in that cycle).

In the experiment, we program with VHDL to design the postprocessing circuit, and synthesize in Xilinx's Vivado. The details of the design are shown in Fig. 6. It consists of several components as below:

- Score Calculation (Score_calc)

Weeder adopts a consensus-based enumeration method to search the motifs. It ranks all the possible motifs according to statistical measures of significance[15]. The equation to compute a score associated with pattern p is defined as:

$$\mathsf{Score}(p) = \sum_{i=1}^{n} I(p,i)) \ln \frac{\mathsf{Obs}(p,i,b_i)}{f(p,i,b_i) * \mathsf{length}(i)}, \tag{1}$$

where *I(p,i)*=1 if p has an occurrence in the i-th input sequence (with at most d mismatches) but equals zero otherwise. b_i denotes the number of mismatches of the best occurrences, which means if there are several occurrences, the one with the least substitutions is the best. Obs*(p,i,b_i)* represents the number of best occurrences of p in the i-th input sequence with b_i mismatches. We remark that the computation of *f(p,i,b_i)*, i.e., the expected frequency value, is not implemented in our design, since it is computed beforehand and stored in the frequency RAM in the preprocessing stage. n is the number of input sequences and length(i) is the length of the ith input sequence.

Score_calc reads and analyzes the vectors from the buffer pool to decide the value of Obs*(p,i,b_i)* and *b_i*. Then the scores are computed using the Vivado floating-point IP integrator. If it receives a finish signal from the central controller and it finishes the last calculation, it sets work_done to high to mark the end of the workload.

- Central Controller

The Central Controller is the most important part of the design. It employs a state machine that is in charge of the communication between the components. The Central Controller reads the offset and pushes the corresponding output vector to the accumulator at the same time. It compares the offset with the length range to decide whether this output event happens within the same input sequence or not. If not, it enables the calculation component to compute the score of the patterns for the current input sequence. Calc_ena signal is to signal the accumulator to start read from the output vector RAM line by line and accumulate the values to get the number of occurrences and number of mismatches in each input sequences for the patterns. Write_ena and read_ena signals are mutually exclusive signals which decide the read or write access to the buffer pool. If the buffers in the Score_calc component finish the reading, they set read_finishes to high. Once the read_finish is high, and Central Controller is in the state which waits for this signal, it signals write_ena to push the data from the accumulator to the buffer pool.

- Output Vector and Offset RAM

As described in section 4.2, the automaton for each pattern shares a similar design on AP. For each pattern of length k with d substitutions, it has $2d+1$ rows, thus $2d+1$ reporting elements where last STE in each row is a reporting element. Therefore, for 6-mers (at most 1 mismatch), 8-mers (2 mismatches) and 10-mers (3 mismatches), it has 3,5,7 reporting elements in the automaton respectively. Suppose the number of k-mers is a, the number of reporting elements of automaton for each k-mer is r, then the length of the output vector is a*r. The offset RAM has arrays of 64 bits, representing the offset in the input stream where it finds a report. Each line maps to the output vector correspondingly.

- Accumulator

The Accumulator is the component designed to accumulate the values from the output vector array. Every bit in a single output vector represents the state of one report element in a single cycle. For 6-mers with 1 mismatch, there are 3*a bits in an output vector where a is the number of 6-mers. Each cycle, the accumulator reads a line from the output vector RAM. But the start of the accumulation depends on the signal calc_ena from the central controller. Since the score calculation component include a multiplication, division, ln(x) and accumulation, it takes more cycles than the accumulator. However, the accumulation time can be ignored as it is overlapped with the computation time. As illustrated in Fig. 7, one can note that the postprocessing time is

$$T = (a + b \times n) \times t, \tag{2}$$

where a denotes the number of cycles needed for the accumulator to accumulate the output vectors produced by one input sequence, b denotes the number of cycles needed for equation calculation, n is the number of input sequences which have report events, and t is the clock time of the circuit.

- Buffer Pool

The buffer pool stores the number of occurrences for each pattern within an input sequence. The size of the pool is the number of bits in the output vector. It is written by the accumulator and read by the score_calc components. It is a key element in the pipeline of accumulator for score_calc.

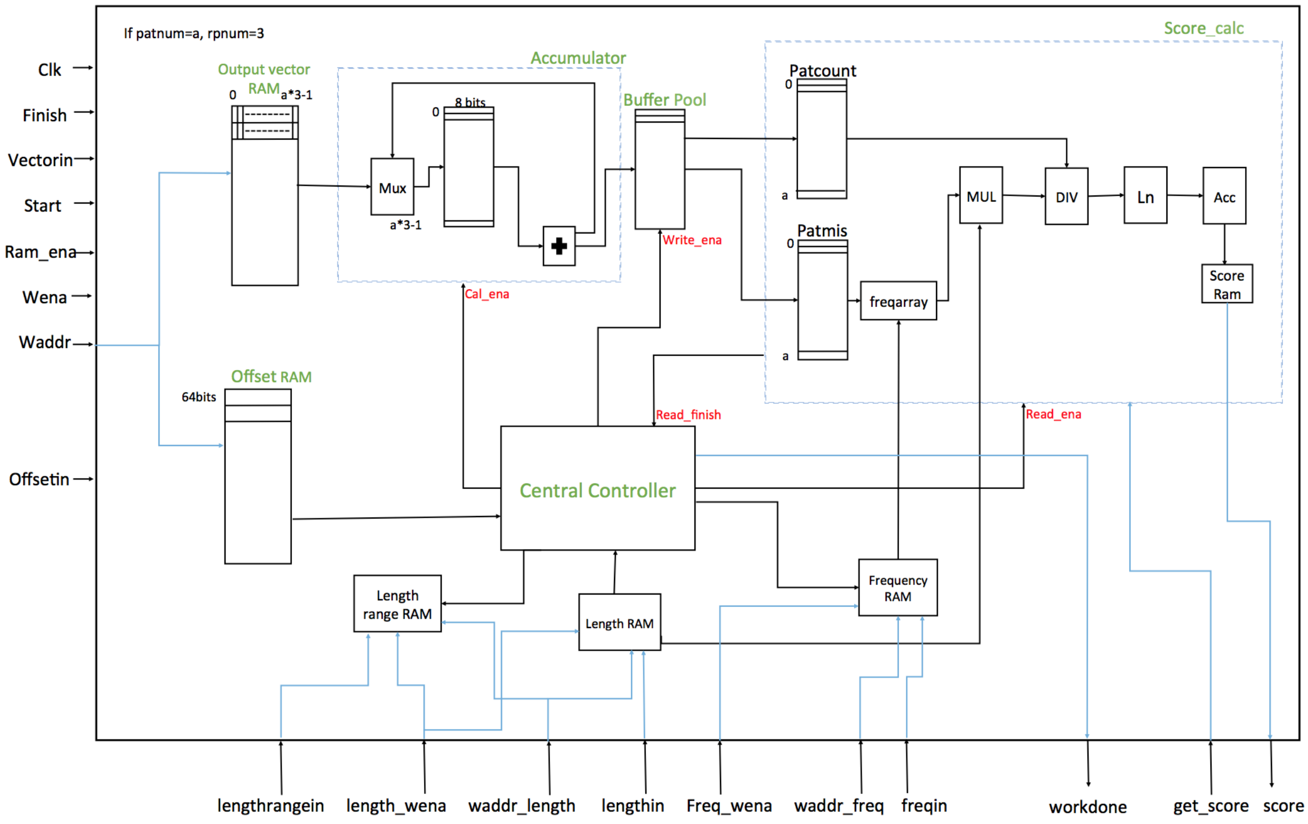


Fig. 6 Postprocessing on the FPGA

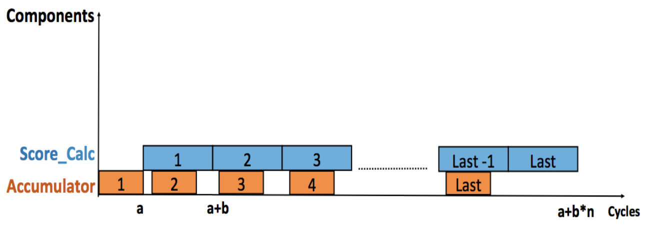


Fig. 7 Pipeline of accumulation and calculation

- Other RAMs

The values of length, length range and frequency RAMs are calculated and loaded in the preprocessing stage. The value of length range is used as boundary and compared to decide whether an output event happens in which input sequence. The data in frequency RAM is calculated based on the frequency files. The length RAM stores the length of each input sequences.

4. Tests and Results

In the experiment, we use the same dataset [1] as described in section 1. The preprocessing stage is running on an Intel Core i7-3770 CPU with 8GB memory. It is remarkable that since the AP hardware is not yet available, our work uses a separated FPGA to model the performance that can be achieved with the AP and a desirable on-board FPGA.

Table 2 FPGA Resource Utilization

Resource	LUT	FF	BRAM
Used	1045689	1742001	808
Available	1221600	2443200	1292
Utilization	85.6%	71.3%	62.5%

Table 3 Speedup of Oligo_scan for different numbers of input sequences

Number of input sequences	Original time	AP-accelerated time	speedup
5000	306	32.5	9
8300	705	38.9	18
16000	2226	40.9	54
166666	37919	50.45	751

Precisely, in our experiment, the postprocessing time is estimated using Vivado simulation and XC7V2000T-2 FPGA board. The FPGA resource utilization is shown in Table 2. We note that the current generation of AP, AP-D480, has an Altera FPGA containing only 270,000 LEs/101,620 ALMs. Such an FPGA is not large enough to hold the postprocessing circuit designed in our work. However, our work is mainly intended to show the speedup potential achievable with an AP approach, and the achieved experiment results indicate that for the motif searching, realizing the full potential is contingent on a larger FPGA.

Since commercial hardware is not yet available, the AP time is modeled by simply computing the length of the in-

Table 4 Time for AP-accelerated Weeder(s) of 166666 input sequences

k-mers	Preprocess (s)			Symbol Replacement			Pattern Matching (s)	Postprocess (s)	Total Time (s)
	Generate motifs	Generate ANML Files	Get length of input sequences	Number of *k*-mers	Times of Replacement	Total time (s)			
6	6.80536	1.0015	0.068	4095	1	0.05	0.08	0.11	8.11
8	5.88941	5.9986	0	55825	5	0.25	0.38	0.55	13.07
10	8.92706	9.9987	0	245903	44	2.2	3.3	4.84	29.27
Total	21.62183	16.9988	0.068	305823	50	2.5	3.76	5.5	50.45

put stream times the 7.5 ns clock period. Table 3 shows the speedup of Oligo_scan for different number of input sequences. The speedup grows as the numbers of input sequences increases. To provide more details about the performance of our proposed system, we choose the result of 166666 input sequences for analysis. This is a typical use case.

The total running time for function oligo_scan in CPU for 166666 input sequences is 37919s, as shown in Table 1. The time of AP-accelerated Weeder is calculated for different *k*-mers (*k*=6, 8, 10) in different stages separately. It is shown in Table 4, while Fig. 8 shows the percentage for each stage.

- After AP acceleration, the preprocessing stage takes the most time (76%). It includes several parts, including generating candidate motifs either from the input sequences or from the result of the last run, converting the automaton to ANML files, and computing the length of each input sequence. At the same time, the automata of the patterns are compiled on the AP chips; the data from frequency files of the candidate motifs are extracted from the frequency files which are used to fill the frequency RAMs; and the data needed for the postprocessing are stored in the other RAMs. It is worth noting that this stage could be accelerated using CPU multi-core parallelism.
- The time for the replacement stage depends on the number of candidates for the *k*-mer of different lengths and the number of passes needed. Since the symbol replacement mechanism provides a fast way with maximum 0.05s each time, we use 0.05s as an upper limit for each time.
- The time for pattern matching is determined by the length of the input stream and the number of replacement passes, since the input sequences need to be streamed in the AP for several times. Taking advantage of the pattern matching in parallel on AP, even though the input sequences are streamed in 44 times for 10-mers, the AP's ability to check many candidate patterns in parallel still yields a net speedup.
- Postprocessing is the second most expensive task in all (21.8%). As shown in Fig. 7, a is the number of cycles needed for analyzing vectors from a single input sequence while b is the number of cycles needed for Equation 1 calculation. We use an array of 500 output vectors as input to run the simulation. On average, it takes 44 cycles for the Accumulator of the FPGA processing, from attaining the output vector to finishing accumulation, and 86 cycles for the Score_calc, from reading data to producing the result. Thus, a=44 while b=86 in Equation 2. And the clock period of the circuit is 7.5ns, according to our synthesis result. Since every input sequence has at least a report event based on the test result, we have n=166666. Therefore, postprocessing time for each output result from AP is $(44+166666 \times 86) \times 7.5\text{ns} \approx 0.11\text{s}$.

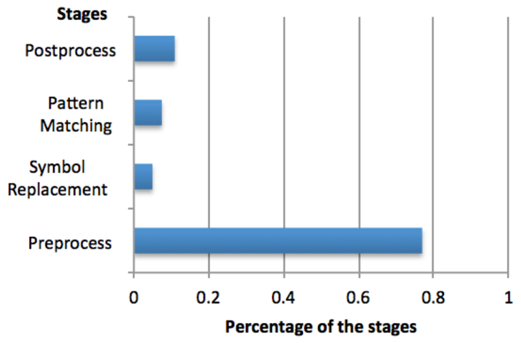


Fig. 8 Percentage of different stages for AP-accelerated Weeder

As shown in Table 4, the time for accelerated oligo_scan is 50.45s in total, and hence the speedup is 751x, as compared to 37919s (given in Table 1), which is the original running time of oligo_scan in a single-threaded CPU implementation.

5. Conclusions and Future Work

In this paper, we presented an AP-accelerated Weeder solution that accelerates the main bottleneck: a pattern-matching function called oligo_scan on Micron's Automata processor that tests how well candidate motifs are conserved in the input. Taking advantage of massively-parallel pattern matching, the AP shows up to 751x speedup for oligo_scan.

In the future work, we plan to further optimize the resource utilization of FPGA-based post-processing proposed in this work. Moreover, apart from oligo_scan, we found that another function, matrix::scan_z , also plays a significant role in Weeder. Particularly, it traverses the top-scoring k-mers calculated by oligo_scan to build the weight matrix. This could also be potentially accelerated by the AP via our proposed approach here.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (61472431, 61272143 and 61272144), grants from Micron Technology, and C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] Dream challenge. *http://dreamchallenges.org/*.

[2] Micron's automata processor. *https://www.micronautomata.com/*.

[3] Timothy L. Bailey and Charles Elkan. Fitting a mixture model by expectation maximization to discover motifs in biopolymer. In *Proceedings of the 2nd International Conference on Intelligent Systems for Molecular Biology*, pages 28–36, 1994.

[4] Timothy L Bailey and Charles Elkan. Unsupervised learning of multiple motifs in biopolymers using expectation maximization. *Machine learning*, 21(1-2):51–80, 1995.

[5] Paul Dlugosch, Dean Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[6] Charles E Lawrence, Stephen F Altschul, Mark S Boguski, Jun S Liu, Andrew F Neuwald, and John C Wootton. Detecting subtle sequence signals: a gibbs sampling strategy for multiple alignment. *science*, 262(5131):208–214, 1993.

[7] Andrei Lihu and Stefan Holban. A review of ensemble methods for *de novo* motif discovery in chip-seq data. *Briefings in Bioinformatics*, 16(6):964–973, 2015.

[8] Harold B Noyes. Micron's Automata Processor architecture: Reconfigurable and massively parallel automata processing. In *Proceedings of the Fifth International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2014.

[9] Giulio Pavesi, Paolo Mereghetti, Giancarlo Mauri, and Graziano Pesole. Weeder web: discovery of transcription factor binding sites in a set of sequences from co-regulated genes. *Nucleic Acids Research*, 32(Web-Server-Issue):199–203, 2004.

[10] Indranil Roy and Srinivas Aluru. Finding motifs in biological sequences using the Micron Automata Processor. In *Proceedings of 28th IEEE International Symposium on Parallel and Distributed Processing*, pages 415–424. IEEE, 2014.

[11] Marie-France Sagot. Spelling approximate repeated or common motifs using a suffix tree. In *Proceedings of LATIN'98: Theoretical Informatics*, pages 374–390. Springer, 1998.

[12] Martin Tompa, Nan Li, Timothy L Bailey, George M Church, Bart De Moor, Eleazar Eskin, Alexander V Favorov, Martin C Frith, Yutao Fu, W James Kent, et al. Assessing computational tools for the discovery of transcription factor binding sites. *Nature biotechnology*, 23(1):137–144, 2005.

[13] Ke Wang, Yanjun Qi, Jeffrey J. Fox, Mircea R. Stan, and Kevin Skadron. Association rule mining with the Micron Automata Processor. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 689–699, 2015.

[14] Ke Wang, Elaheh Sadredini, and Kevin Skadron. Sequential pattern mining with the Micron Automata Processor. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 135–144, 2016.

[15] Federico Zambelli, Graziano Pesole, and Giulio Pavesi. Using weeder, pscan, and pscanchip for the discovery of enriched transcription factor binding site motifs in nucleotide sequences. *Current Protocols in Bioinformatics*, pages 2–11, 2014.

[16] Keira Zhou, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. Brill tagging on the Micron Automata Processor. In *Proceedings of the 9th IEEE International Conference on Semantic Computing*, pages 236–239, 2015.

[17] Keira Zhou, Jack Wadden, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. Regular expression acceleration on the Micron Automata Processor: Brill tagging as a case study. In *Proceedings of the IEEE International Conference on Big Data*, pages 355–360, 2015.

Qiong Wang received her BS degree in computer science from National University of Defense Technology, China in 2011. Now, She is now a PhD candidate at National University of Defense Technology. Her current research interests include power management and application acceleration on heterogeneous architectures.

Mohamed El-Hadedy received the M.Sc degree in Electronics and Communication from Mansoura University, Mansoura, Egypt in 2006. He earned a PhD degree in Electrical and Computer Engineering from the Telematics Department at the Norwegian University of Science and Technology, Trondheim, Norway in 2012. He worked as a Senior Design Engineer at Atmel AS, Norway. After that, he joined University of Virginia as a Research Associate. Currently, he is a Research Scientist at the University of Illinois at Urbana-Champaign. His main research interests include Cryptography, Computer Architecture Design, Signal Processing, Image Processing, FPGA, Reconfigurable devices, Robotics, Big-data accelerators, Coherent Accelerators, and Genome Accelerators. He has two patents that are pending and is writing another one. He is a Member of IEEE.

Kevin Skadron is the Harry Douglas Forsyth Professor and department chair of Computer Science at the University of Virginia, where he has been on the faculty since 1999, when he received his Ph.D. in Computer Science from Princeton University. He is a Fellow of the IEEE and the ACM, and a recipient of the ACM SIGARCH Maurice Wilkes Award. Skadron's research interests include design and application of accelerators and heterogeneous architectures, including solutions to power, thermal, reliability, and programming challenges. To support research in these areas, he and his colleagues have developed a variety of modeling and benchmarking tools, including the ANMLZoo benchmark suite for automata processing, the Rodinia benchmark suite for heterogeneous computing, contributions to the SPEC ACCEL suite, and the HotSpot, VoltSpot, and ArchFP modeling tools.

Ke Wang is a senior scientist in the Department of Computer Science at the University of Virginia. He received his PhD from Tsinghua University in 2007. After receiving his PhD, he worked as a postdoctoral research associate in the Department of Computer Science and Technology at the Tsinghua University. He joined the University of Virginia in 2012. His research interests include applications and design of hardware accelerators, and more generally the design and optimization of heterogeneous computer architectures.