# A Hybrid Architecture for the Approximate String Matching on an FPGA

Takuma Wada, Shunji Funasaka, Koji Nakano and Yasuaki Ito
*Department of Information Engineering, Hiroshima University*
*Kagamiyama 1-4-1, Higashi-Hiroshima, Hiroshima, 739-8527 Japan*
*Email: {takuma, funasaka, nakano, yasuaki}@cs.hiroshima-u.ac.jp*

***Abstract*—The main contribution of this paper is to present a very efficient FPGA implementation, which performs the Approximate String Matching (ASM) for a pattern string and a text string of length $m$ and $n$, respectively. It is well known that the ASM can be done in $O(mn)$ time by the dynamic programming technique. Myers has presented a sophisticated sequential algorithm called bit-vector algorithm, which performs the ASM in $O(n)$ time using $m$-bit addition and bitwise operations. Hoffmann *et al.* have implemented the bit-vector algorithm in the FPGA and evaluated the performance. However, the performance of the bit-vector circuit implemented in the FPGA is degraded for large $m$ due to a long critical path of length proportional to $m$. We will present a circuit with $O(1)$-length critical path that performs the ASM with very high clock frequency and throughput. Also, to reduce the hardware usage, we present a hybrid circuit of the bit-vector and our ASM circuits. The experimental results show that, our hybrid circuit for the ASM is 20 times more efficient than the bit-vector circuit in terms of the performance per circuit resource. To see the potentiality of the ASM computation on the FPGA, we evaluated the performances of the ASM on the latest FPGA, GPU, and CPU. Our hybrid circuit implemented in Xilinx Virtex UltraScale+ XCVU9P FPGA is more than 58 times and 1400 times faster than parallel ASM computation on NVIDIA TITAN X GPU and Core i7-6700K CPU, respectively. Thus, the FPGA is promising as an accelerator of the ASM.**

## I. Introduction

*An Field Programmable Gate Array (FPGA)* is a programmable logic device designed to be configured by customers or designers after manufacturing. Since an FPGA chip maintains relative lower price and programmable features, it is widely used in those fields which need to update architecture or functions frequently such as communication and education. The most common architecture of recent FPGAs is an array of Configurable Logic Blocks (CLBs) [1], block RAMs [2], DSP slices [3], and programmable routing channels connecting them [4]. Although the architecture of the latest FPGAs is targeted for high performance digital signal processing [3], [5], it can be used for other applications and general purpose computing [6]. The main purpose of this section is to present logic circuits for the FPGA that performs the approximate string matching. Our logic circuits are implemented using CLBs in the FPGA, which consists of lookup tables (LUTs), flip-flops (FFs), multiplexers, and carry-chains.

Suppose that two strings $X$ (*pattern*) and $Y$ (*text*) of length $m$ and $n$ ($m \leq n$), respectively, are given. *The Approximate String Matching (ASM)* is a task to find a substring in $Y$ most similar to $X$. The similarity of two strings is measured by the edit distance of them, which is the number of three operations, insertion, deletion, and replacement of characters necessary to change one string into the other. The ASM has a lot of applications in the areas of signal processing, bio-informatics, natural language processing, among others. Thus, many sequential, parallel hardware algorithms for the ASM and the related computations have been presented [7], [8], [9], [10], [11], [12], [13]. It is well known that the ASM can be computed in $O(mn)$ time [14] using the dynamic programming technique, which computes a matrix $d$ of size $(m+1)\times(n+1)$. Figure 1 shows the values of matrix $d$ for $X = ababa$ and $Y = aaabbba$. Each element $d[i][j]$ of matrix $d$ is the edit distance of the prefix of $X$ of length $i$ and a substring of $Y$ ending at position $j$. Since the value of $d[i][j]$ can be computed from $d[i-1][j-1]$, $d[i-1][j]$, and $d[i][j-1]$, the ASM can be done by a column-major or a row-major computation of the matrix. Since each element $d[i][j]$ can be computed in $O(1)$ time, this dynamic programming algorithm runs in $O(mn)$ time. Also, it is easy to parallelize this sequential algorithm to run in $O(n)$ time using $m$ processors [11]. Each processor is assigned to a row of the matrix and compute elements of the assigned row from left to right.

$$
\begin{array}{cc|c|c|c|c|c|c|c|c|}
 & & \multicolumn{8}{c}{Y} \\
 & & & a & a & a & b & b & b & a \\
\cline{3-10}
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\cline{3-10}
 & a & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\cline{3-10}
 & b & 2 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
\cline{3-10}
X & a & 3 & 2 & 1 & 1 & 1 & 1 & 2 & 1 \\
\cline{3-10}
 & b & 4 & 3 & 2 & 2 & 1 & 1 & 1 & 2 \\
\cline{3-10}
 & a & 5 & 4 & 3 & 2 & 2 & 2 & 2 & 1 \\
\cline{3-10}
\end{array}
$$

Figure 1. The values of matrix $d$ computed by the dynamic programming algorithm for the ASM

Myers [15] has presented *the bit-vector algorithm*, which accelerates the ASM using additions and bitwise operations for $m$-bit words. The bit-vector algorithm computes the differences of neighboring elements of the matrix. More

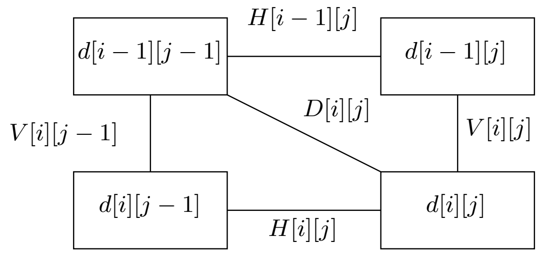

Figure 2. The relations of $d$, $D$, $V$ and $H$.

specifically, difference values $V[i][j] = d[i][j] - d[i-1][j]$, $H[i][j] = d[i][j] - d[i][j-1]$, and $D[i][j] = d[i][j] - d[i-1][j-1]$ of the matrix are computed for each position $(i, j)$ as shown in Figure 2. Since the values of $V[i][j]$ and $H[i][j]$ are either $-1$, 0, or 1, they can be encoded in 2 bits each. Also, the value of $D[i][j]$ is 0 or 1, it can be represented using only 1 bit. Using such bit encoding, the differences for all elements in a column are computed from those in the left column by a sequence of $m$-bit addition and bitwise operations [15]. Since commonly used computing devices such as CPUs and GPUs support 32/64-bit addition and bitwise operations, the bit-vector algorithm works very efficiently for pattern strings of length 32/64. Also, if $m$ is larger than 64, the bit-vector algorithm can be implemented by simulating $m$-bit addition and bitwise operations by iterative 32/64-bit operations in an obvious way. Thus, very roughly speaking, the speed up factor of the bit-vector algorithm can be 32/64 over the $O(mn)$-time dynamic programming algorithm, since 32/64 elements in a column of the matrix can be computed in $O(1)$ operations. Hoffmann *et al.* [16] have implemented the bit-vector algorithm on the FPGA. We call it *bit-vector circuit*. Their implementation simply uses an adder and logic gates to simulates the bit-vector algorithm as it is. However, the bit-vector circuit uses an $m$-bit adder and so the performance is degraded for large $m$ due to long carry chain of the adder.

The main contribution of this paper is to present logic circuits for the ASM and to implement them on an FPGA. We first present a small logic circuit $S$, which computes difference values $V[i][j]$, $H[i][j]$, and $D[i][j]$ from $V[i][j-1]$ and $H[i-1][j]$. We design circuit $\mathcal{S}$ with $m$ logic circuits of $S$, which computes these values for $m$ elements of the matrix in one clock cycle and the ASM can be done in $n + m - 1$ clock cycles. Since the critical path length of circuit $\mathcal{S}$ is $O(1)$, it runs in very high clock frequency and the throughput is quite large. However, circuit $\mathcal{S}$ needs more flip-flops to store the intermediate values than the bit-vector circuit shown in [16]. Thus, we present a hybrid circuit $\mathcal{H}$ to compute the difference values of the matrix. Hybrid circuit $\mathcal{H}$ has a design parameter which determines the size of circuit modules. Since that with extreme values of the design parameter correspond to circuit $\mathcal{S}$ and the bit-vector circuit, it is a hybrid of them. We can select the best design parameter that maximizes the performance of $\mathcal{H}$. We measure the performance of FPGA implementations for the ASM by "elements per CLB·sec", which is the number of elements in the matrix computed per one CLB in a second. The implementation results show that hybrid circuit $\mathcal{H}$ with best parameter for $m = 1024$ runs in 775MHz using 675 CLBs on Xilinx Virtex UltraScale+ XCVU9P-L2FLGA2104E FPGA. If $n = 1024^2$ (= 1M), the ASM can be done in 1.35ms. Also, since 1024 elements computed in every clock cycle, the performance is $1.18 \times 10^9$ elements per CLB·sec. Since the XCVU9P FPGA has 147,780 CLBs, we can expect that 175 hybrid circuits can be embedded in 80% of all CLBs. Note that if we use all CLBs, the clock performance is drastically degraded due to the overhead of detour channel routing in the FPGA. Thus, it makes sense to assume that we use 80% of CLBs in the FPGA. If this is the case the total performance of the FPGA with 175 hybrid circuits is about $140 \times 10^{12}$ elements per FPGA·sec.

To see the advantage of FPGA over the CPU and GPU, we have implemented the bit-vector algorithm on NVIDIA TITAN X GPU and Core i7-6700K(4GHz). Our GPU implementation is essentially the same as that shown in [17], and reproduces the experimental results on the latest GPU. The experimental results show that the performance of the GPU is $2.38 \times 10^{12}$ elements per GPU·sec. Thus, the performance of our hybrid circuit implemented in XCVU9P FPGA is 58 times better than the best GPU implementation. We have also evaluated the performance of a sequential bit-vector algorithm [15] on Core i7-6700K. The performance using a single thread is about $0.0125 \times 10^{12}$ elements per CPU·sec. Hence, even if 8 hyper threads in 4 cores work perfectly in parallel, the performance can not be larger than $0.100 \times 10^{12}$. Thus, the performance of our hybrid circuit on the FPGA is more than 1400 times better than Core i7-6700K even if 8 hyper threads works completely in parallel.

This paper is organized as follows. We first define the Edit Distance (EM) and the Approximate String Matching (ASM) and review sequential and parallel algorithms for the ASM in Section II. Section III shows the difference computation technique, which computes the difference values of elements in the matrix of the ASM. We then go on to show logic circuits to compute the difference values in Section IV. Section V shows the experimental results on FPGA, GPU, and CPU. Section VI concludes our work.

## II. Approximate string matching and edit distance

The main purpose of this section is to review Approximate String Matching (ASM) and the Edit Distance (ED). Please see [14], [18] for the details.

As a preliminary, we first define the Edit Distance (ED) of two strings. Suppose that source string $X = x_1x_2 \cdots x_m$ of length $m$ and destination string $Y = y_1y_2 \cdots y_n$ of length $n$ are given. Without loss of generality, we can assume that $m \leq n$. We want to change $X$ into $Y$ using the following three operations:

- insertion of a character,
- deletion of a character, and
- replacement of a character.

For example, $X = ababa$ can be changed into $Y = aaabbb$ in five operations as follows: $ababa \overset{delete}{\rightarrow} aaba \overset{delete}{\rightarrow} aaa \overset{insert}{\rightarrow} aaab \overset{insert}{\rightarrow} aaabb \overset{insert}{\rightarrow} aaabbb$. Alternatively, $X$ can be changed into $Y$ in three operations as follows: $ababa \overset{replace}{\rightarrow} aaaba \overset{replace}{\rightarrow} aaabb \overset{insert}{\rightarrow} aaabbb$. *The ED of two strings* is the minimum number of operations to change one string to the other. For example, the ED of $X$ and $Y$ above is three, because there exists a sequence of three operations to change $X$ into $Y$, and there exists no sequence of less than three operations to do the same thing. For later reference, let $\mathrm{ED}(X,Y)$ denote the ED of $X$ and $Y$.

The approximate string matching, a more flexible version of the edit distance, is a task to compute the value of $\mathrm{ASM}(X,Y)$ defined as follows:

$$\mathrm{ASM}(X,Y) = \min\{\mathrm{ED}(X,Y') \mid Y' \text{ is a substring of } Y\}$$

Clearly, $\mathrm{ASM}(X,Y)$ is small if $Y$ has a substring similar to $X$. It should be clear that $\mathrm{ASM}(X,Y)$ is always less than or equal to $m$, and $\mathrm{ED}(X,Y)$ takes a value between $n-m$ and $n$. For example, if $X$ and $Y$ share no character, then $\mathrm{ED}(X,Y) = n$ and $\mathrm{ASM}(X,Y) = m$. Also, if the prefix of $Y$ is $X$ then $\mathrm{ED}(X,Y) = n-m$ and $\mathrm{ASM}(X,Y) = 0$.

We use a matrix $d$ of size $(m+1) \times (n+1)$ to compute the ASM. Each $d[i][j]$ $(0 \leq i \leq m, 0 \leq j \leq n)$ is used to store the following value:

$$\min_{1 \leq j' \leq j} \mathrm{ED}(x_1x_2 \cdots x_i, y_{j'}y_{j'+1} \cdots y_j).$$

Note that $x_1x_2 \cdots x_i$ is a null string (i.e. string of length 0) if $i = 0$. After all values of $d$ is computed, we can compute the value of $\mathrm{ASM}(X,Y)$ by the following formula:

$$\mathrm{ASM}(X,Y) \quad = \quad \min_{0 \leq j \leq n} d[m][j]$$

Thus, if we have all values $d[m][j]$ $(1 \leq j \leq n)$, we can compute $\mathrm{ASM}(X,Y)$ very easily. Hence, in this paper we focus on the computation of these values.

Let us show how we compute all values of $d$. Suppose that $d[i-1][j-1]$, $d[i-1][j]$, and $d[i][j-1]$ are already computed. Let "$x_i \neq y_j$" denote the binary value such that it is 1 if $x_i \neq y_j$ and 0 if $x_i = y_j$. The value of $d[i][j]$ can be computed by the following recursive formulas:

$$\begin{aligned}
d[i][j] &= 0 \quad \text{if } i = 0 && (1)\\
&= i \quad \text{if } j = 0 && (2)\\
&= \min(d[i][j-1]+1, && (3)\\
&\qquad d[i-1][j]+1, && (4)\\
&\qquad d[i-1][j-1]+(x_i \neq y_j)) && (5)\\
&\qquad \text{if } i > 0 \text{ and } j > 0.
\end{aligned}$$

Using this formula, all values of matrix $d$ can be computed as follows:

**[Algorithm ASM]**
```
for $j \leftarrow 1$ to $n$ do $d[0][j] \leftarrow 0$
for $i \leftarrow 0$ to $m$ do $d[i][0] \leftarrow i$
for $i \leftarrow 1$ to $m$ do
 for $j \leftarrow 1$ to $n$ do
   $d[i][j] \leftarrow \min(d[i][j-1]+1, d[i-1][j]+1,$
     $d[i-1][j-1]+(x_i \neq y_j))$;
 output($d[m][j]$);
```

Figure 1 shows the values of $d$ for two strings $X = ababa$ and $Y = aaabbba$.

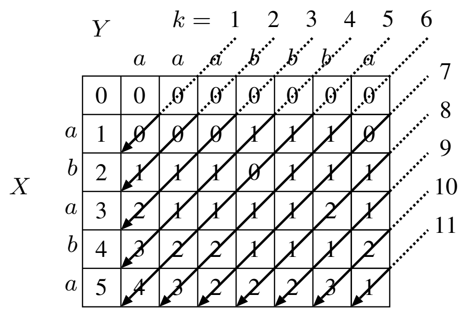

Figure 3. The values of matrix $d$ computed for each $k$

Next, we will show a parallel algorithm for the ASM. The key idea is to compute the values of the matrix $d$ from the top-left corner to the bottom-right corner [11], [12]. The details of the parallel algorithm is spelled out as follows:

**[Parallel ASM algorithm]**
```
for $j \leftarrow 1$ to $n$ do in parallel $d[0][j] \leftarrow 0$
for $i \leftarrow 0$ to $m$ do in parallel $d[i][0] \leftarrow i$
for $k \leftarrow 1$ to $n+m-1$ do
 for $i \leftarrow 1$ to $m$ do in parallel
   $j \leftarrow k-i+1$
   if $1 \leq j \leq n$ then
     $d[i][j] \leftarrow \min(d[i][j-1]+1, d[i-1][j]+1,$
       $d[i-1][j-1]+(x[i] \neq y[j]))$
 if($j \geq 1$) output ($d[m][j]$);
```

In the third for-loop, for each $k$ $(1 \leq k \leq n+m-1)$, the values $d[1][k], d[2][k-1], \ldots, d[m][k-m]$ are computed and stored. Figure 3 illustrates the computation performed

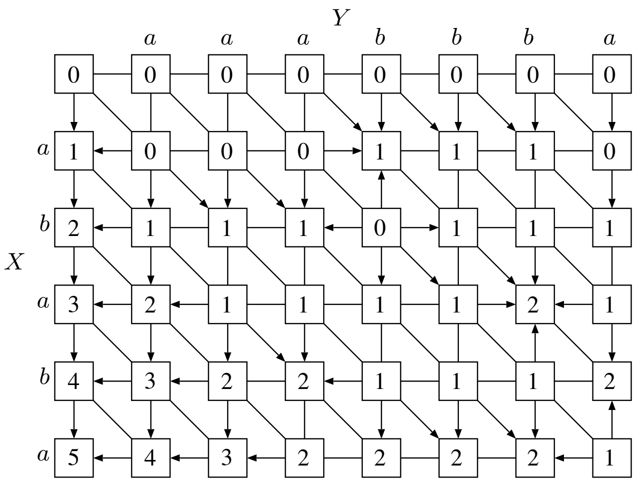

Figure 4. The difference values of matrix $d$

in each $k$ for the same matrix $d$ of Figure 1. It should be clear that this parallel algorithm correctly computes the ASM.

## III. Difference computation technique for the ASM

We will show *the difference computation technique*, which computes the difference values of elements in the matrix of the ASM. This technique has been invented by Myers and used by the bit-vector algorithm [15]

Let $d$ be a matrix obtained by the dynamic programming algorithm for the ASM. We use three matrices $V$, $H$, and $D$ such that each of them is the difference of adjacent values of $d$ as follows:

$$\begin{aligned} V[i][j] &= d[i][j] - d[i-1][j] \\ H[i][j] &= d[i][j] - d[i][j-1] \\ D[i][j] &= d[i][j] - d[i-1][j-1] \end{aligned}$$

Figure 2 illustrates the relations of these values. Also, Figure 4 shows the values of $V$, $H$, and $D$ for the same input strings as Figure 1. In Figure 4, each segment with no arrowhead represents 0 and each arrow shows the direction of increment of 1.

We will show the relations of $d$, $D$, $V$ and $H$. We omit "for all $i$ and $j$" to avoid iteration of the same redundant phrase. Since $d[i][j]$ is the minimum of $d[i][j-1]+1$, $d[i-1][j]+1$, and $d[i-1][j-1]+(x_i \neq y_j)$, we have $d[i][j] \leq d[i][j-1]+1$, $d[i][j] \leq d[i-1][j]+1$, and $d[i][j] \leq d[i-1][j-1]+1$. Thus, we have $V[i][j] \leq 1$, $H[i][j] \leq 1$, and $D[i][j] \leq 1$. We will prove $V[i][j] \geq -1$, $H[i][j] \geq -1$, and $D[i][j] \geq 0$ by induction. More specifically, we assume that $V[i][j-1] \geq -1$ and $H[i-1][j] \geq -1$ hold and prove that $V[i][j] \geq -1$, $V[i][j] \geq -1$, and $D[i][j] \geq 0$ are satisfied. If $D[i][j] \leq -1$ then $d[i][j] = d[i-1][j-1] + (x_i \neq y_j)$ is not satisfied. Hence, $d[i][j] = d[i-1][j]+1$ and/or $d[i][j] = d[i][j-1]+1$ must be satisfied. If $d[i][j] = d[i-1][j]+1$ holds, then $H[i][j] = 1$. Thus, $D[i][j] = V[i][j-1] + H[i][j] \geq 0$ is a contradiction. Hence, $D[i][j] \geq 0$ holds. Further, from $V[i][j-1] + H[i][j] = D[i][j] \geq 0$ and $V[i][j-1] \leq 1$, we have $H[i][j] \geq -1$. Similarly, we can prove $V[i][j] \geq -1$. Therefore, we have,

*Lemma 1:* For all $i$ and $j$ ($1 \leq i \leq m$ and $1 \leq j \leq n$), we have $-1 \leq H[i][j] \leq 1$, $-1 \leq V[i][j] \leq 1$, and $0 \leq D[i][j] \leq 1$.

Actually, in Figure 4, we can see segments and arrows of both directions for horizontal and vertical directions corresponding to $H$ and $V$. On the other hand, there is no diagonal arrows pointing the upper left, because the values of $D[i][j]$ cannot be $-1$.

From the definition of $H$, if we have all values of $H[m][1], H[m][2], \ldots, H[m][n-1]$, we can compute the value of $d[m][i]$ by the following recursive formula:

$$\begin{aligned} d[m][0] &= m \\ d[m][i] &= d[m][i-1] + H[m][i] \quad (1 \leq i \leq n). \end{aligned}$$

In other words, $d[m][i]$ can be computed by the prefix-sums of $m, H[m][1], H[m][2], \ldots, H[m][n]$. Thus, we focus on designing circuits that output $H[m][1], H[m][2], \ldots, H[m][n]$ one by one. A simple accumulator, which increments or decrements the stored value by 1, can compute the prefix-sums for these outputs in an obvious way.

We will show important properties to determine the values of elements in $V$, $H$, and $D$. Since $d[i][0] = i$ for all $i$ ($1 \leq i \leq m$) and $d[0][j] = 0$ for all $j$ ($1 \leq j \leq n$), $V[i][0] = 1$ and $H[0][j] = 0$. Also, since $d[i][j]$ is the minimum of $d[i][j-1]+1$, $d[i-1][j]+1$, and $d[i-1][j-1]+(x_i \neq y_j)$, $D[i][j]$ takes 1 if and only if all of the following three conditions are satisfied.

- $x_i \neq y_j$,
- $V[i][j-1] \geq 0$ (i.e. $d[i][j-1] \geq d[i-1][j-1]$), and
- $H[i-1][j] \geq 0$ (i.e. $d[i-1][j] \geq d[i-1][j-1]$).

In other words, if at least one of the three is not satisfied, then $D[i][j] = 0$. If $x_i = y_j$, then $d[i][j] \leq d[i-1][j-1]$. Thus, from Lemma 1, $D[i][j] = 0$. If $V[i][j-1] = -1$, then $d[i][j-1]+1 = d[i-1][j-1]$. Hence, $d[i][j] = d[i-1][j-1]$ and so $D[i][j] = 0$. Similarly, if $H[i-1][j] = -1$ then $D[i][j] = 0$. Thus, we have the following formula: $D[i][j] = (x_i \neq y_j) \wedge (V[i][j-1] \geq 0) \wedge (H[i-1][j] \geq 0)$. Also, from the definition of $V$, $H$, and $D$, we have $D[i][j] = H[i][j] + V[i][j-1] = V[i][j] + H[i-1][j]$. We can also confirm these relations from Figure 2. We summarize these properties as follows:

*Lemma 2:* We have
(1) $V[i][0] = 1$ for all $i$ ($1 \leq i \leq m$), and
(2) $H[0][j] = 0$ for all $j$ ($1 \leq j \leq n$).
Also, for all $i$ and $j$ ($1 \leq i \leq m$, $1 \leq j \leq n$), we have
(3) $D[i][j] = (x_i \neq y_j) \wedge (V[i][j-1] \geq 0) \wedge (H[i-1][j] \geq 0)$,

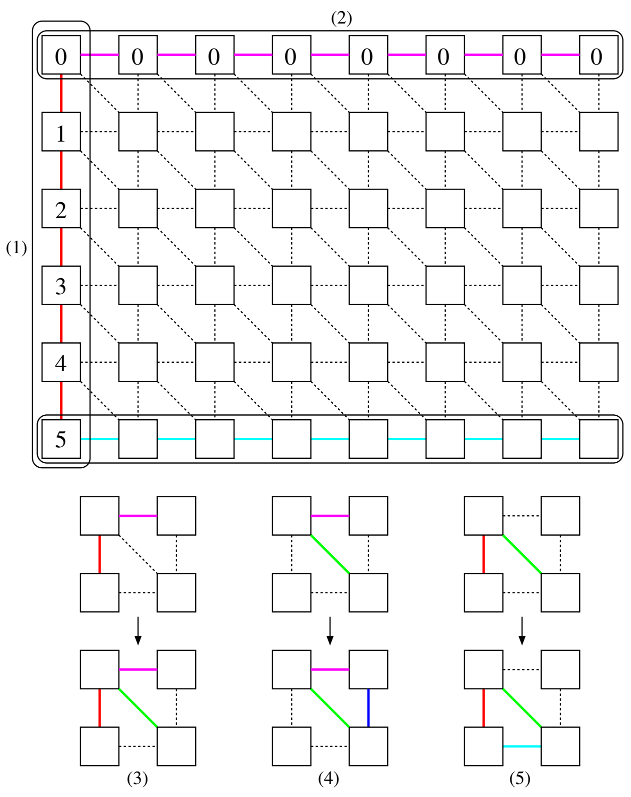

Figure 5. Illustrating (1), (2), (3), (4), and (5) in Lemma 2

(4) $V[i][j] = D[i][j] - H[i-1][j]$, and
(5) $H[i][j] = D[i][j] - V[i][j-1]$.

From Lemma 2, we can design algorithms for the ASM, which compute the values of $H[m][j]$ for all $j$ $(1 \leq j \leq n)$. They correspond to light blue segments in Figure 5. From (1) and (2) in Lemma 2, the values of $V$ and $H$ correspond to red and magenta segments in Figure 5 are initially given. Figure 5 also illustrates the computation of $D[i][j]$, $V[i][j]$ and $H[i][j]$ correspond to (3), (4), and (5) in Lemma 2, respectively. Thus, we can think that the ASM is a problem of painting segments in Figure 5, such that all segments in the bottom row are painted by repeatedly applying painting rules (3), (4), and (5).

We will use two operations that we call *operation 3-4-5* and *operation 5-3-4* as illustrated in Figure 6. Operation 3-4-5 uses painting rules (3), (4), and (5) for $V[i][j-1]$ and $H[i-1][j]$ in turn and two segments corresponding to $V[i][j]$ and $H[i][j]$ are painted. In operation 5-3-4, from three painted segments of $V[i-1][j-1]$, $V[i][j-1]$, and $D[i-1][j]$, two segments $V[i][j]$ and $D[i][j]$ are painted.

The idea of Myers's bit-vector algorithm [15] is based on operation 5-3-4. The values of $V[1][j-1], V[2][j-1], \ldots, V[m][j-1]$ are stored in two $m$ bits word. A certain number of addition and bitwise operations for words

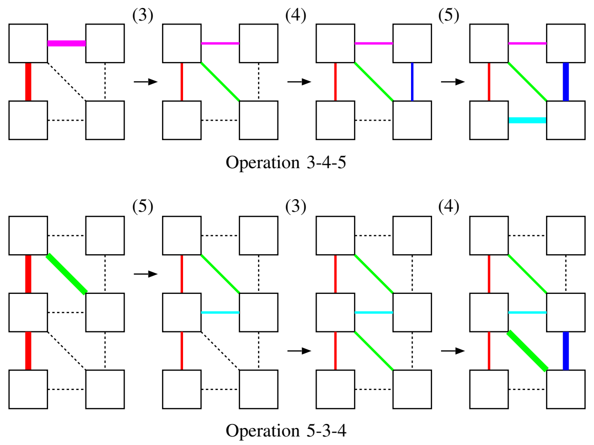

Figure 6. Illustrating operations 3-4-5 and 5-3-4

of $m+1$ bits are performed to compute the values of $V[1][j], V[2][j], \ldots, V[m][j]$. Since conventional processors supports addition and bitwise operations for 32/64-bit words, Myers's bit-vector algorithm is efficient for $m \leq 64$. Since addition and bitwise operations for long words can be simulated by iteration of those for 32/64-bit word, it also work for very large $m$. An FPGA implementation that simulates addition and bitwise operations for long words used in Myers's bit-vector algorithm have been presented in [16].

## IV. Logic circuits for the ASM

This section first shows logic circuits based on operations 3-4-5 and 5-3-4. After that, we will present a hybrid circuit of them.

### A. ASM circuit by operation 3-4-5

Figure 7 illustrates a combinational logic circuit $S$ implementing operation 3-4-5 in Figure 6, which computes $V[i][j]$ and $H[i][j]$ from $V[i][j-1]$ and $H[i-1][j]$. In boxes of the figure, combinational circuits to evaluate (3), (4), and (5) in Lemma 2 are implemented. Since each element of $V$ and $H$

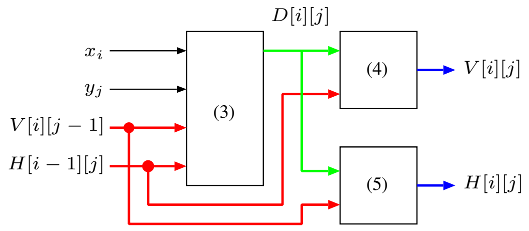

Figure 7. Combinational logic circuit $S$ for operation 3-4-5

takes $-1$, 0, or 1, two bits are necessary. We can use 2-bit words 01, 00, and 10 to represent the values of $-1$, 0, and 1 of each element, respectively.

We use circuit $S$ to design a combinational logic circuit $\mathcal{S}$ that simulates Parallel ASM algorithm illustrated in Figure 3. It computes $V[1][k], H[1][k], V[2][k-1], H[2][k-1], \ldots, V[m][k-m+1], H[m][k-m+1]$ from $V[1][k-1], V[2][k-2], \ldots, V[m][k-m]$ and $H[0][k], H[1][k-1], \ldots, H[m-1][k-m+1]$. Figure 8 illustrates circuit $\mathcal{S}$, which simply has $m$ circuits $S$. The figure also shows how the values of $V$ and $H$ are computed. Clearly, the computation performed for each $k$ by Parallel ASM algorithm is simulated by $\mathcal{S}$ in 1 clock cycle. Thus, $\mathcal{S}$ runs $m+n-1$ clock cycles for the ASM. Also, the critical path length of circuit $\mathcal{S}$ is $O(1)$ and $2 \cdot 2 \cdot m = 4m$ flip-flops are necessary to store the values of $V$ and $H$. Thus, we have,

*Theorem 3:* Circuit $\mathcal{S}$ with $4m$ flip-flops of depth $O(1)$ can perform the ASM in $m+n-1$ clock cycles.

## B. ASM circuit by operation 5-3-4

Figure 9 illustrates a combinational logic circuit $T$ implementing operation 5-3-4 in Figure 6, which computes $V[i][j]$ and $D[i][j]$ from $V[i-1][j-1]$, $V[i][j-1]$, and $D[i-1][j]$.

We can design a circuit $\mathcal{T}$ for the ASM using $m$ circuits $T$ as illustrated in Figure 10. They computes $V[1][k], V[2][k], \ldots, V[m][k]$ from $V[1][k-1], V[2][k-1], \ldots, V[m][k-1]$. Note that, to compute $V[1][k]$, a circuit $T/(5)$, in which a circuit for (5) is omitted in circuit $T$, is used. Also, a circuit to compute (5) in Lemma 2 is used to output $H[m][k]$, which is necessary to compute $d[m][k]$. Thus, we can say that a circuit $\mathcal{T}$ uses $m$ circuits $T$. Circuit $\mathcal{T}$ works for the computation for each column, it runs $n$ clock cycles to complete the ASM. Also, it needs $2m$ flip-flops to store the outputs $V[1][k], V[2][k], \ldots, V[m][k]$, which is the inputs of $\mathcal{T}$ in the following clock cycle. Note that the critical path length of circuit $\mathcal{T}$ is $O(m)$ since it has input/output chain corresponding to $V[i][k]$s. Thus, we have,

*Theorem 4:* Circuit $\mathcal{T}$ with $2m$ flip-flops of depth $O(m)$ can perform the ASM in $n$ clock cycles.

## C. Hybrid ASM circuit

We will show a hybrid ASM circuit, a combination of circuits $\mathcal{S}$ and $\mathcal{T}$. Recall that circuit $\mathcal{T}$ has $m$ circuits of $T$. A hybrid circuit $\mathcal{H}$ has $\frac{m}{r}$ circuits $\mathcal{T}$ with $r$ circuits of $T$, where $r$ is a parameter of the circuit. If $r = m$ then a hybrid circuit $\mathcal{H}$ is equivalent to circuit $\mathcal{T}$. Also, $\mathcal{H}$ is $\mathcal{S}$ when $r = 1$, because a circuit for (5) is attached to $T/(5)$ in $\mathcal{T}$ is circuit $S$. Thus, it is a hybrid of $\mathcal{S}$ and $\mathcal{T}$. Figure 11 illustrates a hybrid circuit $\mathcal{H}$, which has 3 circuits $\mathcal{T}$ with 3 circuits of $T$ each.

We will show that how hybrid circuit $\mathcal{H}$ works. Similarly to circuits $\mathcal{S}$ and $\mathcal{T}$, it computes the values of $V$ from left to right. Clearly, $\frac{m}{r}+n-1$ clock cycles are necessary to complete the ASM. Also, each $\mathcal{T}$ in $\mathcal{H}$ needs $2(r+1)$ flip-flops to store the necessary values of $V$ and $H$. Thus, $\mathcal{H}$ uses $\frac{m}{r} \cdot 2(r+1) = 2m(1+\frac{1}{r})$ flip-flops. The depth of the circuit of $\mathcal{H}$ is that of $\mathcal{T}$, which is equal to $O(r)$. Thus, we have,

*Theorem 5:* Circuit $\mathcal{H}$ with $2m(1+\frac{1}{r})$ flip-flops of depth $O(r)$ can perform the ASM in $\frac{m}{r}+n-1$ clock cycles.

## D. Bit-vector algorithm

This section briefly explains bit-vector algorithm [15] and the FPGA implementation [16]. Please see [15], [16] for the details. The idea of the bit-vector algorithm is to simulate circuit $\mathcal{T}$ by addition and bitwise operations as follows. The values of $V[1][j-1], V[2][j-1], \ldots, V[m][j-1]$ are stored in two $m$ bits word. A certain number of addition and bitwise operations for $m$-bit words are performed to compute the values of $V[1][j], V[2][j], \ldots, V[m][j]$. Since commonly used processors supports addition and bitwise operations for 32/64-bit words, the bit-vector algorithm is efficient if $m = 32$ or 64. Since addition and bitwise operations for long words can be simulated by iteration of those for 32/64-bit words, it also work for very large $m$.

We show how circuit $\mathcal{T}$ in Figure 10 is simulated. Let $V^+$ and $V^-$ be $m$-bit words such that the $i$-th bits of them are 1 if $V[i][j-1] = 1$ and $-1$, respectively. The $i$-th bits of them are 0 if $V[i][j-1] = 0$. Similarly, we use two $m$-bit words $H^+$ and $H^-$ to store $H[i][j]$s. Also, let $D^0$ be an $m$-bit word such that each $i$-th bit is 1 if $D[i][j] = 0$. Further, let $M$ be an $m$-bit word such that each $i$-th bit is 1 if $x_i = y_j$. After executing the following computation, $V^+$ and $V^-$ are updated by the values of $V[i][j]$s:

```
[Bit-vector algorithm]
    $D^0 \leftarrow (((M \& V^+) + V^+ \oplus V^+)|M|V^-;$
    $H^+ \leftarrow V^-|\tilde{}(D^0|V^+);$
    $H^- \leftarrow D^0 \& V^+;$
    $V^+ \leftarrow (H^+ << 1)|\tilde{}(D^0|H^+ << 1);$
    $V^- \leftarrow D^0 \& (H^+ << 1);$
```

The key of the bit-vector algorithm is addition "+" of $m$ bits to compute $D^0$. The carry chain of this addition simulates cascade connection of circuits $T$ in circuit $\mathcal{T}$ illustrated in Figure 10. We can design circuits that simulates the bit-vector algorithm using $m$-bit adder and arrays of logic gates for bitwise operations in an obvious way. We call it *bit-vector circuit*. Further, we can design a hybrid circuit by replacing each circuit $\mathcal{T}$ by the bit-vector circuit.

Essentially, the bit-vector algorithm is designed so that circuit $\mathcal{T}$ is simulated by additions and bitwise operations. So, intuitively, it makes no sense to simulate the bit-vector algorithm by a logic circuit, because a circuit simulation algorithm is simulated by a circuit. It is expected that such iterative simulation results increase of hardware resource usage and degrade the clock performance. However, as we will show in Section V, the hardware usage of both modules are not so different. Also, for large $m$, the clock performance of the bit-vector circuit is better. This is because an adder

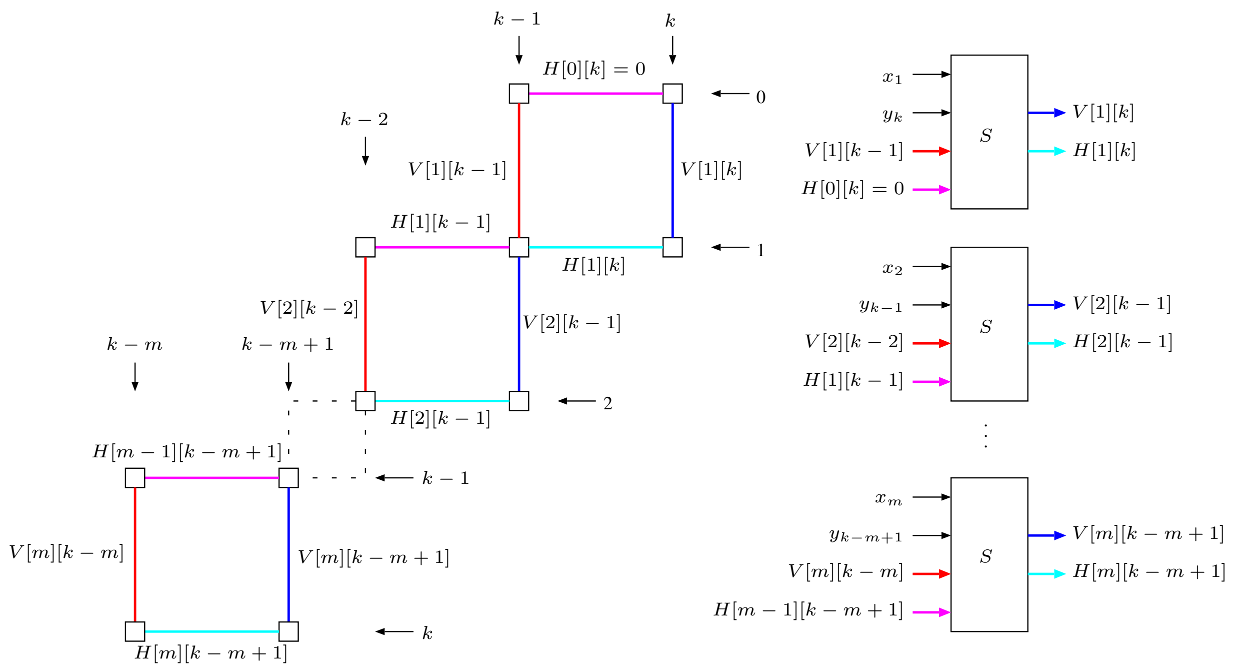

Figure 8. A combinational logic circuit $\mathcal{S}$ using $m$ circuits of $S$

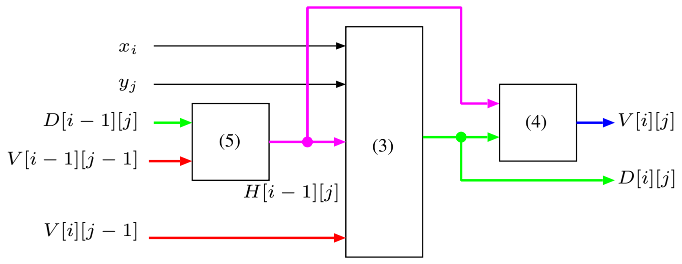

Figure 9. A combinational logic circuit $T$ for operation 5-3-4

used in the bit-vector circuit can be implemented in embedded fact carry chain logic in the FPGA very efficiently. The experimental results show that the performance of both circuits is almost the same for small $m$.

## V. Experimental results

We have implemented circuit $\mathcal{S}$, circuit $\mathcal{T}$, our hybrid circuit $\mathcal{H}$, and the bit-vector circuit [16] on Xilinx Virtex UltraScale+ XCVU9P-L2FLGA2104E FPGA with 147,780 CLBs, and evaluated the performance. This FPGA is a popular high-end FPGA, which is used in Virtex UltraScale FPGA VCU118 evaluation board [19] as well as Amazon cloud computing service AWS EC2 F1 instance. The logic synthesis performed using Vivado Design Suite. For reference, we have used NVIDIA TITAN X and Core i7-6700K (4.00GHz) for evaluating the performance of the bit-vector algorithms on a GPU and a CPU. Also, since the main application of the ASM is analysis of DNA sequences with 4 bases A, T, C, and G [20], we assume that pattern and text are strings of 2-bit characters to encode 4 bases.

Table I shows the performance of our hybrid circuit $\mathcal{H}$ using circuits $\mathcal{T}$ of various sizes for $m = 1024$. It shows clock frequency in MHz, the number of LUTs, FFs, and CLBs. The performance is evaluated for various size of $\mathcal{T}$ from 1 to 1024 for $m = 1024$. Note that, each CLB of Xilinx Virtex UltraScale+ has 8 LUTs and 16 FFs [1]. For example, when the size of $\mathcal{T}$ is 8, in 675 used CLBs with 5400 LUTs and 10800 FFs totally, only 3708 LUTs and 2687 FFs are used. If the size of $\mathcal{T}$ is 1 then our hybrid circuit is equivalent to circuit $\mathcal{S}$. Also, a single $\mathcal{T}$ is used when the size of $\mathcal{T}$ is 1024. We can see that a hybrid circuit with larger $\mathcal{T}$ uses fewer CLBs, because it uses fewer flip-flops to store $H[i][j]$. Also, since a larger $\mathcal{T}$ has a longer critical path for $V[i][j]$, the clock performance decreases. We can see that the hardware resource usage and the clock frequency in these tables follow Theorem 5. The tables also show the number of elements computed by a CLB in a second. For example, 128 circuits $\mathcal{T}$ of size 8 uses 675 CLBs and works in 775MHz. Let us consider that a set of values of $V[i][j]$, $H[i][j]$, $D[i][j]$ is *an element*. We can think that the task of the ASM is to compute all $m \times n$ elements of the matrix, and hybrid circuits in Table I compute 1024 elements in one clock cycle. Hence, if the circuit runs in 775MHz, it can computes $775 \times 10^6 \times 1024 = 793.6 \times 10^9$ elements per second. Thus, in the table, the value of "elements per CLB·sec" for $\mathcal{T}$ of size 8 is $\frac{793.6\times10^9}{675} = 1.18 \times 10^9$. This value is maximized when

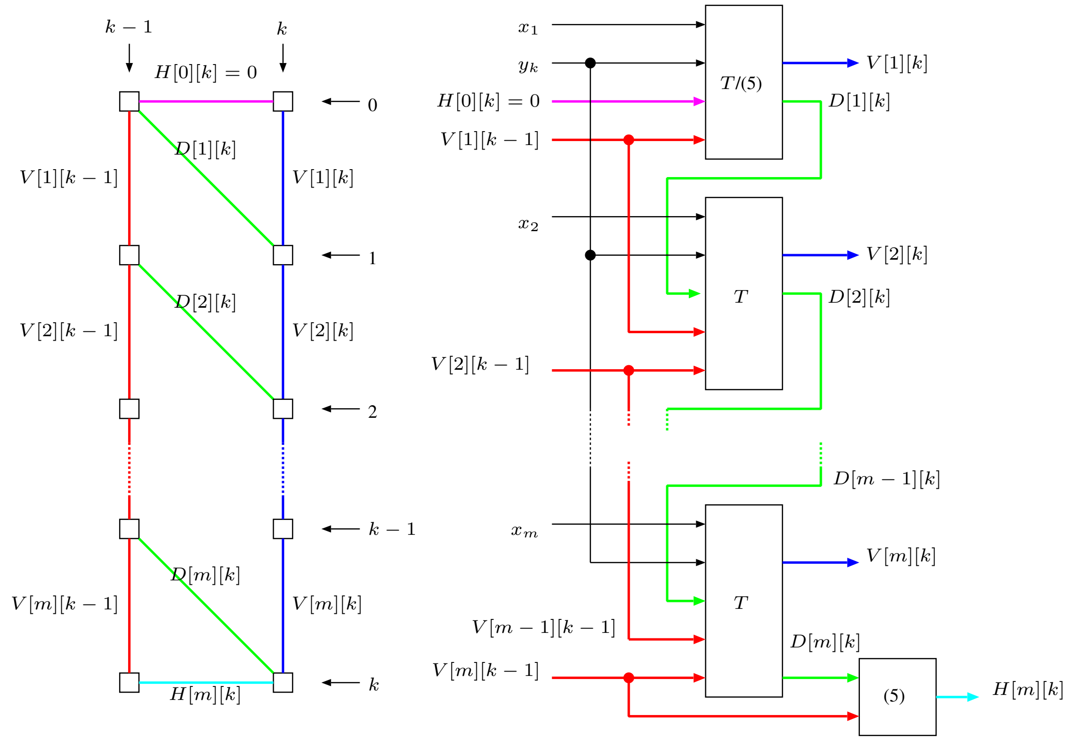

Figure 10. A combinational logic circuit $\mathcal{T}$ using $m$ circuits of $T$

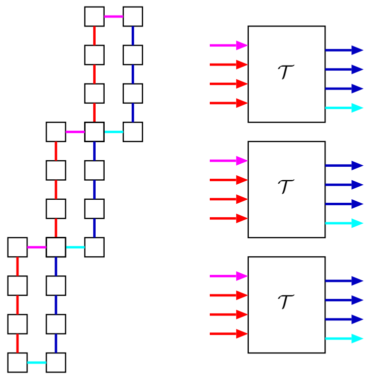

Figure 11. A hybrid circuit $\mathcal{H}$ using circuits of $\mathcal{T}$

Table I
THE PERFORMANCE OF OUR HYBRID CIRCUIT $\mathcal{H}$ USING CIRCUITS $\mathcal{T}$ FOR $m = 1024$

| Size of $\mathcal{T}$ | clock (MHz) | LUTs | FFs | CLBs | $10^9$ elements per CLB·sec |
|---|---|---|---|---|---|
| 1 | 775 | 3865 | 7169 | 1619 | 0.490 |
| 2 | 775 | 3070 | 4607 | 938 | 0.846 |
| 4 | 775 | 2815 | 3327 | 800 | 0.992 |
| 8 | 775 | 3708 | 2687 | 675 | 1.18 |
| 16 | 465 | 3272 | 2366 | 535 | 0.890 |
| 32 | 408 | 4279 | 2207 | 643 | 0.650 |
| 64 | 213 | 5047 | 2126 | 879 | 0.248 |
| 128 | 94 | 4689 | 2086 | 791 | 0.122 |
| 256 | 77 | 3768 | 2105 | 592 | 0.133 |
| 512 | 38 | 3165 | 2057 | 523 | 0.0744 |
| 1024 | 19 | 3121 | 2051 | 514 | 0.0379 |

the hybrid circuit is configured as 128 circuits $\mathcal{T}$ of size 8 each. Since XCVU9P FPGA has 147,780 CLBs, we can expect that the total performance of XCVU9P using 80% of CLBs is $1.18 \times 10^9 \times 147780 \times 0.8 = 139 \times 10^{12}$ elements per FPGA·sec. Note that if almost all of CLBs are used, the clock frequency is decreased due to detour connection routing and layout overheads. Thus, we assume that only 80% of CLBs not to overestimate the FPGA performance for the ASM. If this is the case, $\frac{0.8 \cdot 147780}{675} = 175$ hybrid circuits are implemented in the FPGA works in parallel. Since XCVU9P has 832 I/O pins, we can implement 175 hybrid circuits with 2 input pins to provide 2-bit characters text strings. Hence, it is feasible to embed 175 hybrid circuits in XCVU9P.

Table II shows the performance of our hybrid circuit using circuits $\mathcal{T}$ for $m = 4096$. The maximum performance is obtained when we use 128 circuits $\mathcal{T}$ of size 8 each, and the performance is $1.14 \times 10^9$ elements. Hence the performance evaluated by the number of elements computed per CLB·sec. is almost the same as that for $m = 1024$. Actually, circuits $\mathcal{T}$ of size 8 are used for both $m = 1024$ and 4096, the clock frequency is the same and the number of CLBs are proportional to the number of circuits $\mathcal{T}$. Hence, we can

Table II
THE PERFORMANCE OF OUR HYBRID CIRCUIT $\mathcal{H}$ USING CIRCUITS $\mathcal{T}$ FOR $m = 4096$

| Size of $\mathcal{T}$ | clock (MHz) | LUTs | FFs | CLBs | $10^9$ elements per CLB·sec |
|---|---|---|---|---|---|
| 1 | 775 | 15395 | 28673 | 6360 | 0.499 |
| 2 | 775 | 12287 | 18431 | 3584 | 0.886 |
| 4 | 775 | 11263 | 13311 | 2877 | 1.10 |
| 8 | 775 | 14842 | 10751 | 2792 | 1.14 |
| 16 | 431 | 13141 | 9471 | 2305 | 0.766 |
| 32 | 374 | 17206 | 8831 | 2724 | 0.562 |
| 64 | 190 | 20545 | 8511 | 3418 | 0.228 |
| 128 | 80 | 18561 | 8351 | 3150 | 0.104 |
| 256 | 69 | 15267 | 8477 | 2339 | 0.121 |
| 512 | 33 | 12670 | 8231 | 1997 | 0.0677 |
| 4096 | 3 | 12843 | 8195 | 1967 | 0.00625 |

Table III
THE PERFORMANCE OF OUR HYBRID CIRCUIT $\mathcal{H}$ USING THE BIT-VECTOR CIRCUITS FOR $m = 1024$

| Size of bit-vector | clock (MHz) | LUTs | FFs | CLBs | $10^9$ elements per CLB·sec |
|---|---|---|---|---|---|
| 1 | 775 | 3865 | 7169 | 1619 | 0.490 |
| 2 | 775 | 4093 | 5120 | 1123 | 0.707 |
| 4 | 775 | 3497 | 3584 | 787 | 1.01 |
| 8 | 775 | 4231 | 2816 | 913 | 0.87 |
| 16 | 732 | 4164 | 2432 | 687 | 1.09 |
| 32 | 669 | 4132 | 2240 | 661 | 1.04 |
| 64 | 596 | 4095 | 2144 | 634 | 0.963 |
| 128 | 494 | 4101 | 2096 | 614 | 0.824 |
| 256 | 370 | 4100 | 2072 | 612 | 0.619 |
| 512 | 243 | 4052 | 2066 | 606 | 0.411 |
| 1024 | 150 | 4095 | 2051 | 626 | 0.245 |

expect that the performance evaluated by the number of elements computed per CLB·sec. are almost the same for larger $m$.

Tables III and IV show the performance of circuits our hybrid circuits $\mathcal{H}$ using bit-vector circuits. The reader should compare them with Tables I and II, which show the performance of our hybrid circuits $\mathcal{H}$ using circuits $\mathcal{T}$. The peak performance of elements per CLB·sec is obtained when the size is 16 for $m = 1024$ and 4 for $m = 4096$. Their peak performances are little smaller than the hybrid circuit with circuits $\mathcal{T}$. We can think that hybrid circuits with one bit-vector circuit reproduces the previously published result in [16], which simply implements the bit-vector algorithm on the FPGA. Since the clock frequency is very low due to a long carry chain of an adder, the performance is much lower than our hybrid circuit. For example, when $m = 4096$, the performance of the previously published implementation shown in [16] is only $0.0555 \times 10^9$ elements per CLB·sec, while the performance of our hybrid circuit is $1.14 \times 10^9$. Thus, our hybrid circuit is more than 20 times efficient.

We have implemented the bit-vector algorithm in NVIDIA TITAN X GPU. Our implementation is essentially the same as that shown in [17] and reproduces their results on a latest GPU. Also, the sequential bit-vector algorithm is executed on Core i7-6700K CPU. Although it has multiple cores and

Table IV
THE PERFORMANCE OF OUR HYBRID CIRCUIT $\mathcal{H}$ USING THE BIT-VECTOR CIRCUITS FOR $m = 4096$

| Size of $\mathcal{T}$ | clock (MHz) | LUTs | FFs | CLBs | $10^9$ elements per CLB·sec |
|---|---|---|---|---|---|
| 1 | 775 | 15395 | 28673 | 6360 | 0.499 |
| 2 | 775 | 16363 | 20480 | 4419 | 0.718 |
| 4 | 775 | 14052 | 14336 | 3106 | 1.02 |
| 8 | 759 | 16910 | 11264 | 3468 | 0.896 |
| 16 | 695 | 16671 | 9728 | 2964 | 0.960 |
| 32 | 655 | 16527 | 8960 | 2719 | 0.987 |
| 64 | 579 | 16438 | 8580 | 2494 | 0.951 |
| 128 | 466 | 16413 | 8384 | 2445 | 0.781 |
| 256 | 319 | 16373 | 8288 | 2442 | 0.535 |
| 512 | 214 | 16057 | 8258 | 2415 | 0.363 |
| 4096 | 38 | 16383 | 8195 | 2802 | 0.0555 |

multiple threads can run at the same time, we use one thread to execute the sequential bit-vector algorithm.

Tables V and VI show the performance for $m = 1024$ and 4096, respectively. The ASM is computed for text strings of length $n = 1024^2$. Also, the running time is evaluated for multiple input instances from 2048 to 32K pairs of pattern/text. To fully utilize the GPU resources, we should invoke as many threads as possible. Thus, we perform the ASM computation for multiple input instances. Thus, the ASM computation of 32K instances for $m = 4096$ and $n = 1024^2$ computes $32\text{K}{\cdot}4096 \cdot 1024^2 = 2^{47} = 1.40 \times 10^{14}$ elements. The GPU implementation runs 59.09 seconds, we can say that the performance is $\frac{1.40\times10^{14}}{59.09} = 2.38 \times 10^{12}$ elements per GPU·sec. Also, the CPU performance is about $0.0125 \times 10^{12}$ elements per CPU·sec. Note that, the CPU implementation just repeats the same bit-vector operation, the running time is proportional to the number of elements, and so the performance in terms of elements per CPU·sec is almost the same.

We first compare the performance of the FPGA and the GPU in terms of the ASM. From Table I, only one hybrid circuit $\mathcal{H}$ with 675 CLBs on the FPGA can perform 32K instances of the ASM with $m = 1024$ and $n = 1024^2$ in $\frac{32\text{K}\cdot(1024+1024^2-1)}{775\text{M}} = 44.4$ seconds from Theorem 5. From Table V, the GPU implementation takes 14.77 seconds for the same task. Thus, the performance of one hybrid circuit $\mathcal{H}$ on 675 CLBs is 33.5% of the GPU. Since 175 hybrid circuits $\mathcal{H}$ can be embedded in a XCVU9P FPGA using 80% of all CLBs, we can say that a XCVU9P FPGA is more than 58 times faster than an NVIDIA TITAN X GPU.

Next, let us compare the performances of the FPGA and the CPU. From Table I, only one hybrid circuit $\mathcal{H}$ with 675 CLBs on the FPGA can perform 32K instances of the ASM with $m = 1024$ and $n = 1024^2$ in $\frac{32\text{K}\cdot(1024+1024^2-1)}{775\text{M}} =$ 44.4 seconds. From Table V, the CPU takes 2870 seconds. Thus, quite surprisingly, only 675 CLBs in the FPGA can be more than 64 times faster than the CPU. Since Core i7-6700K used for evaluation can execute 8 threads on 4 cores at the same time, a fully program cannot be more than 8

Table V
THE PERFORMANCE OF THE BIT-VECTOR ALGORITHM ON NVIDIA TITAN X FOR $m = 1024$ AND $n = 1024^2$

| #instances | 2048 | 4096 | 8192 | 16K | 32K |
|---|---|---|---|---|---|
| GPU Time (sec) | 1.241 | 2.048 | 3.830 | 7.447 | 14.77 |
| $10^{12}$ ele./GPU·sec | 1.77 | 2.15 | 2.30 | 2.36 | 2.38 |
| CPU Time (sec) | 177.8 | 358.6 | 714.8 | 1446 | 2870 |
| $10^{12}$ ele./CPU·sec | 0.0124 | 0.0123 | 0.0123 | 0.0122 | 0.0123 |

Table VI
THE PERFORMANCE OF THE BIT-VECTOR ALGORITHM ON NVIDIA TITAN X FOR $m = 4096$

| #instances | 2048 | 4096 | 8192 | 16K | 32K |
|---|---|---|---|---|---|
| GPU Time (sec) | 3.828 | 7.466 | 14.57 | 29.29 | 59.09 |
| $10^{12}$ ele./GPU·sec | 2.30 | 2.36 | 2.41 | 2.40 | 2.38 |
| CPU Time (sec) | 675.5 | 1361 | 2714 | 5761 | 11100 |
| $10^{12}$ ele./CPU·sec | 0.0130 | 0.0129 | 0.0130 | 0.0122 | 0.0127 |

times faster. Since we can implement 175 hybrid circuits $\mathcal{H}$ using 80% CLBs in a XCVU9P FPGA, we can say that XCVU9P is more than 1400 times faster than Core i7-6700K for the ASM.

## VI. CONCLUSION

The main contribution of this paper is to present a hybrid circuit for the Approximate String Matching (ASM) on the FPGA. Our hybrid circuit in the FPGA is more than 20 times efficient than the previously published implementation based on the bit-vector algorithm. Also, our hybrid circuit implemented in a Xilinx Virtex UltraScale+ XCVU9P FPGA is more than 58 times faster than NVIDIA TITAN X GPU and more than 1400 times faster than Core i7-6700K (all 4 cores are used). Thus, the FPGA is promising as an accelerator of the ASM computation.

## ACKNOWLEDGMENTS

The authors would like to thank Mr. Ryota Kadoto, who performed earlier stage of this work as his thesis.

## REFERENCES

[1] Xilinx Inc., "Ultrascale architecture configurable logic block: User guide," Feb 2017.

[2] ——, "Ultrascale architecture memory resources," May 2017.

[3] ——, "Ultrascale architecture DSP slice," June 2017.

[4] ——, "Ultrascale architecture and product data sheet: Overview," Feb. 2017.

[5] R. Woods, *FPGA-based Implementation of Signal Processing Systems*. Wiley, 2008.

[6] Y. Ito, K. Nakano, and S. Bo, "The parallel FDFM processor core approach for CRT-based RSA decryption," *International Journal of Networking and Computing*, vol. 2, no. 1, pp. 79–96, Jan. 2012.

[7] E. Ukkonen, "Algorithms for approximate string matching," *Information and Control*, vol. 64, no. 1–3, pp. 100–118, January–March 1985.

[8] T. V. Court and M. C. Herbordt, "Families of FPGA-based accelerators for approximate string matching," *Microprocess Microsyst.*, vol. 31, no. 2, pp. 135–145, Mar. 2007.

[9] Y. Liu, L. Guo, J. Li, M. Ren, and K. Li, "Parallel algorithms for approximate string matching with k mismatches on CUDA," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*. IEEE CS Press, May 2012, pp. 2414–2422.

[10] Y. Utan, M. Inagi, S. Wakabayashi, and S. Nagayama, "A GPGPU implementation of approximate string matching with regular expression operators and comparison with its FPGA implementation," in *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications*, July 2012.

[11] D. Man, K. Nakano, and Y. Ito, "An optimal implementation of the approximate string matching on the hierarchical memory machine, with performance evaluation on the GPU," *IEICE Trans. on Information and Systems*, vol. E97-D, no. 12, pp. 3063–3071, Dec. 2014.

[12] L. Saad, J. Bordim, K. Nakano, and Y. Ito, "A fast approximate string matching algorithm on GPU," in *Proc. International Symposium on Computing and Networking*, Dec. 2015, pp. 188–192.

[13] Y. Mitani, F. Ino, and K. Hagihara, "Parallelizing exact and approximate string matching via inclusive scan on a GPU," *IEEE Trans on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1989–2002, July 2017.

[14] P. H. Sellers, "The theory and computation of evolutionary distances: Pattern recognition," *Journal of Algorithms*, vol. 1, no. 4, pp. 359–373, December 1980.

[15] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM*, vol. 46, no. 3, pp. 395–415, May 1999.

[16] J. Hoffmann, D. Zeckzer, and M. Bogdan, "Using FPGAs to accelerate Myers bit-vector algorithm," in *Proc. of Mediterranean Conference on Medical and Biological Engineering and Computing*, Sept. 2016, pp. 535–541.

[17] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "Thread-cooperative, bit-parallel computation of levenshtein distance on GPU," in *Proc. of International Conference on Supercomputing*, 2014, pp. 103–112.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[19] Xilinx Inc., "VCU118 evaluation board user guide," Dec 2016.

[20] N. Patil, D. Toshniwal, and K. Garg, "Approximate pattern matching for DNA sequence data," in *Proc. of International Conference on Advances in Communication, Network, and Computing (CCIS, volume 142)*, 2011, pp. 212–218.