Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm

Nauman Ahmed*, Vlad-Mihai Sima†, Ernst Houtgast†, Koen Bertels* and Zaid Al-Ars*

*Computer Engineering Lab, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
†Bluebee, Molengraaffsingel 12–14, 2629 JD Delft, The Netherlands
E-mail: *{n.ahmed, k.l.m.bertels, z.al-ars}@tudelft.nl, †{vlad.sima, ernst.houtgast}@bluebee.com

***Abstract*—The fast decrease in cost of DNA sequencing has resulted in an enormous growth in available genome data, and hence led to an increasing demand for fast DNA analysis algorithms used for diagnostics of genetic disorders, such as cancer. One of the most computationally intensive steps in the analysis is represented by the DNA read alignment. In this paper, we present an accelerated version of BWA-MEM, one of the most popular read alignment algorithms, by implementing a heterogeneous hardware/software optimized version on the Convey HC2ex platform. A challenging factor of the BWA-MEM algorithm is the fact that it consists of not one, but three computationally intensive kernels: SMEM generation, suffix array lookup and local Smith-Waterman. Obtaining substantial speedup is hence contingent on accelerating all of these three kernels at once. The paper shows an architecture containing two hardware-accelerated kernels and one kernel optimized in software. The two hardware kernels of suffix array lookup and local Smith-Waterman are able to reach speedups of 2.8x and 5.7x, respectively. The software optimization of the SMEM generation kernel is able to achieve a speedup of 1.7x. This enables a total application acceleration of 2.6x compared to the original software version.**

I. Introduction

With the emergence of low cost high throughput next-generation DNA sequencing methods, it is now possible to diagnose many genetic disorders, e.g. cancer, with very high resolution. In DNA sequencing, DNA is broken down into small fragments which are then sequenced using sequencing machines that produce hundreds of millions of short DNA reads. These short reads are then processed to identify the differences between the DNA under test and a reference genome. Processing all these millions of short reads is a very time consuming process. Therefore, acceleration and optimization of the analysis time is needed to make DNA diagnostics feasible for a large population.

Read alignment is a core step in DNA analysis, which is the process of comparing two DNA strings to identify the amount of similarity between them. In read alignment, a short read (with a typical length of 100 bases) is aligned against a huge reference genome of for example nearly 3 billion bases for the human genome. The Smith-Waterman algorithm [1], used as a standard for sequence alignment, has $O(n \cdot m)$ complexity to align two sequences of length n and m. However, using Smith-Waterman for the alignment process is too time consuming for any practical purposes [2]. For this reason, many new read aligners have emerged in the past few years.

With the continued improvements in sequencing technologies, sequencing read lengths continue to increase gradually. BWA-MEM has been designed to perform efficient and accurate alignment of longer reads, making it one of the most popular alignment algorithms available [3]. However, BWA-MEM is one of the most computationally intensive algorithms needed for DNA analysis. In this work, we discuss the acceleration of BWA-MEM using both hardware and software techniques. This is the first acceleration of BWA-MEM reported in the literature. In this work, we accelerate BWA-MEM on the Convey HC2ex platform, consisting of an 8-core Intel Xeon based host processor connected to a co-processor with four Xilinx Virtex-6 FPGAs.

No work has been reported in the literature about the acceleration of BWA-MEM. Numerous works are published on the acceleration of other read alignment algorithms. In [4] is shown to be accelerated on Convey HC2ex platform. BWA-ALN also has an accelerated version on Convey HC1 [5]. Both algorithms are different from BWA-MEM and are not suitable for longer reads. The seed generation stage of the CUSHAW2 [6] is similar to BWA-MEM. It computes maximal exact matches instead of super maximal exact matches and uses a different kind of index for which it requires two passes to find all the seeds as compared to only one pass in BWA-MEM. CUSHAW2 has a GPU accelerated version called CUSHAW2-GPU [7].

The paper proposes to split BWA-MEM in different execution stages, that are then optimally mapped to CPU or FPGA. The algorithm has a lot of DRAM accesses causing large memory waits. We circumvent this problem in software by improving the algorithm to reduce the number of DRAM accesses. We also address this problem in hardware by coalescing the memory accesses. The compute intensive parts of the application are accelerated on the FPGA by exploiting the available parallelism. BWA-MEM is segmented in such a way that enables the parallel execution of host and coprocessor to maximize throughput. With the help of these hardware and software optimizations we are able to achieve a 2.6x speedup for the whole application. A large part of the application is

Acknowledgement—This work is supported by the Faculty Development Program of the University of Engineering and Technology Lahore, Pakistan

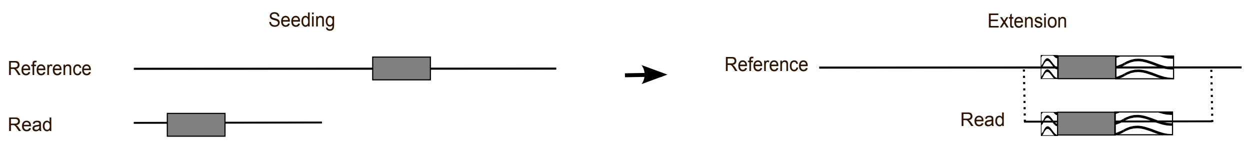

Fig. 1. A seed is first found (rectangular box) and then extended (shaded box).

TABLE I
PERCENTAGE EXECUTION TIME OF BWA-MEM STAGES

Execution stage	% Execution time	Bound
SMEM generation	31.78 - 38.88%	Memory
Suffix array lookup	6.65 - 14.15%	Memory
Seed extension	34.57 - 42.87%	Computation
Output	7.42 - 11.16%	Memory
Miscellaneous	1.33 - 6.04%	–

written in Xilinx Vivado HLS to reduce the design time.

The rest of the paper is organized as follows. Section II summarizes the BWA-MEM algorithm and the profiling results. Section III describes the implementation details of the software optimizations of the SMEM generation. Section IV and V present the hardware acceleration of the suffix array lookup and the local Smith-Waterman stages, respectively. Results are presented in Section VI. Conclusions and future work is discussed in Section VII.

II. BWA-MEM ALGORITHM

Most of the current read aligners (including BWA-MEM) are based on the observation that two DNA sequences of the same species are likely to contain short highly matched subsequences. Such kind of aligners follow a *seed-and-extend* strategy, which consists of two steps (1) seeding and (2) extension. The seeding step is to first locate the regions within the reference genome where a subsequence of the short read is highly matched. This subsequence is known as a *seed*. A seed could be an exact match or an inexact match with certain allowed number of differences. After seeding the remaining read is aligned to the reference genome around the seed in the extension step using Smith-Waterman or a Smith-Waterman like algorithm. BWA-MEM only finds exact matches while seeding.

A. Execution stages

In BWA-MEM, before starting the read alignment, an index of the reference genome is created. This is a one time step and hence not on the critical execution path. In our discussion we assume that an index is already present. The different execution stages of BWA-MEM read alignment algorithm are described below. The first two stages belong to seeding.

SMEM generation: BWA-MEM first computes the so-called *super-maximal exact matches* (SMEMs). An SMEM is a subsequence of the read that is exactly matching in the reference DNA and cannot be further extended in either directions. Moreover, it must not be contained in another match.

Suffix array lookup: The suffix array lookup stage is responsible for locating the actual starting position of the SMEM in the reference genome. An SMEM with its known starting position(s) in the reference genome forms seed(s) in the reference.

Seed extension: Seeds are subsequences of the read that are exactly matching in the reference genome. As shown in Figure 1, to align the whole read against the reference genome, these seeds are extended in both directions. This extension is performed using a dynamic programming algorithm based on Smith-Waterman.

Output: The read alignment information is written to a file in the SAM (sequence alignment/map) format [8].

B. Profiling results

We have profiled the BWA-MEM software version 0.7.8 to find the percentage execution times of different stages using `gprof`. Table I shows the division of total execution time among the different execution stages, taking a number of input data sets representing different read lengths into consideration. These datasets are acquired from GCAT (Genome Comparison and Analytic Testing) and aligned against the UCSC hg19 human reference genome [9]. The results show that there is no dominant execution stage which means that it is essential to accelerate all the stages to achieve a good overall speedup. The table also shows that the most of the stages of the application are memory bound.

In the following sections, we show the software optimization of the SMEM generation step and the hardware acceleration of the suffix array lookup and seed extension. Compared with the software all these stages can execute in parallel, which allows us to hide some of the computation time by pipelining the queries.

III. OPTIMIZATION OF SMEM GENERATION

A. Theoretical background

This is the first execution stage of BWA-MEM. As described above, the task is to find SMEMs. BWA-MEM uses the *FMD-index* of the reference genome to find the SMEMs. Let T and Y be two sequences of symbols. In DNA analysis, T is the reference DNA while Y is a subsequence of the read, and the symbols are drawn from an alphabet set Σ consisting of only four symbols i.e. $\Sigma = \{a, t, c, g\}$. The FMD-index is a set of data structures based on the Burrows-Wheeler transform of $T \oplus \overline{T}$ where $\oplus$ is the concatenation operator. $\overline{T}$ is Watson-Crick reverse complement of T. It is formed by first reversing the string T and then replacing symbol a with t and vice versa, and replacing g with c and vice versa. The FMD-index can locate all occurrences of Y in the reference genome T in time proportional to the length of Y. Let SA be the *lexicographically sorted* suffix array of $T \oplus \overline{T}$, where $SA(i)$ represents the ith suffix. Then the *suffix array interval* of Y is defined as $[I_l(Y), I_u(Y)]$, where

$$\begin{aligned} I_l(Y) &= \min\{i : Y \text{ is the prefix of } SA(i)\} \\ I_u(Y) &= \max\{i : Y \text{ is the prefix of } SA(i)\} \end{aligned}$$

$I_l(Y)$ and $I_u(Y)$ are known as the lower and upper limit of the interval, respectively. In other words, the suffix array interval is the set of all those indices of the sorted suffix array in which Y is the prefix. As the suffix array is lexicographically sorted, all these indices will occur together and we only need to know the first and the last index, i.e. $I_l(Y)$ and $I_u(Y)$, respectively. If Y is not present in the reference genome then $[I_l(Y), I_u(Y)]$ is an empty set and if Y is an empty string then $[I_l(Y), I_u(Y)]$ = $[1, 2|T|]$. The size of the interval can be defined as

$$I_s(Y) = I_u(Y) - I_l(Y) + 1 \quad (1)$$

If Y is not present in the reference genome $I_s(Y)$ is less than or equal to zero. In [10], it is shown that FMD-index can be used to find the *bi-interval* of a given DNA string Y. The bi-interval is defined as $[I_l(Y), I_l(\overline{Y}), I_s(Y)]$. $\overline{Y}$ is Watson-Crick reverse complement of Y. Once the bi-interval of Y is known, the suffix array interval can be found using equation 1. These suffix array intervals are used in the next execution stage (the suffix array lookup stage) to find the exact starting position of Y in the reference genome.

B. SMEM computation in BWA-MEM

An SMEM satisfies three conditions 1) it is an exact match, 2) the match cannot be extended in either directions, and 3) it is not contained in any other match. Algorithm 1 shows the method used in BWA-MEM to find all the SMEMs that include the base at position i_0 of the read P. The complete SMEM computation algorithm can be found in the literature [10].

The algorithm starts from the base at i_0 and first calculates the bi-interval of the string $P[i_0]$ and stores it in Temp array. This calculation is performed by calling the FMDINDEX function. The first `while` loop it moves in the forward direction and adds the next base to the previous string $P[i_0]$, and calculates the bi-interval of the string $P[i_0]P[i_0+1]$. The $forward$ parameter passed to the FMDINDEX indicates the direction of adding the base to the previous string. If this string exists in the reference genome T and the bi-interval is not the same as for $P[i_0]$, its bi-interval is saved in Temp. In this way the algorithm keeps on adding the bases to the previous string and storing the resulting bi-interval in Temp until the string does not exist in the reference genome, i.e. $s \leq 0$. Thus the Temp array contains the bi-intervals of a set of overlapping strings, all starting from i_0. In the second `while` loop the algorithm picks these strings one by one and enlarges them in the reverse direction by adding bases that are behind i_0. For each string its keeps on adding the bases in the backward direction until the resulting string no more exists in the reference genome. The bi-interval of the last hit string is then added to SMEMs array if the length of the string is at least a minimum required. Hence, SMEMs keeps the bi-intervals of the SMEMs found.

One limitation of this algorithm is the accesses to the FMD-index. Every time a base is added in either the forward or backward direction, the FMD-index is accessed. FMD-index is a large data structure having a default total size of 1.5 GB. These accesses to the FMD-index are not local and the difference between the memory addresses of consecutive accesses is huge as studied in [11]. This causes a large amount of data cache and data TLB misses. Most of the accesses to FMD-index end up in the DRAM. Algorithm 1 has a large execution time because it is mostly waiting for the memory request to complete. In our work we speed up the algorithm by reducing the number of FMD-index accesses.

Our improvement is based upon two observations. 1) The algorithm first computes an SMEM and then checks whether its length is greater than the minimum required. In this way, it backward enlarges even those strings which cannot have the total final length greater than the minimum required. 2) All the computed SMEMs have to include the base at position i_0 which means that they overlap with each other and share a common substring. If the bi-interval of this common substring is already known then it can be enlarged to find the SMEMs with less number of FMD-index accesses.

Algorithm 1: SMEM computation

```
Input: String $P$, start position $i_0$, length of reference genome $|T|$, and minimum
       required SMEM length $min\_smem\_len$
Output: Set of bi-intervals $[k, l, s]$ of the SMEMs covering the base at $i_0$
Function CompSMEM($P, i_0, |T|, min\_smem\_len$) begin
    Initialize $[k, l, s]$ as $[1, 1, 2|T|]$
    Initialize Temp, Fwd_len and SMEMs as empty arrays
    $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i_0], forward$)
    Append $[k, l, s]$ to Temp
    $i \leftarrow i_0 + 1$
    $x \leftarrow 1$
    Fwd_len[x] $\leftarrow 1$
    $x \leftarrow x + 1$
    while $i \leq |P|$ and $s > 0$ do
        $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i], forward$)
        if $s > 0$ and $[k, l, s] \neq$ Temp$[x - 1]$ then
            Append $[k, l, s]$ to Temp
            Fwd_len[x] $\leftarrow i - i_0 + 1$
            $x \leftarrow x + 1$
        $i \leftarrow i + 1$
    $x \leftarrow 1$
    while $x \leq$ |Temp| do
        $[k, l, s] \leftarrow$ Temp$[x]$
        for $i \leftarrow i_0 - 1$ to $1$ do
            $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i], backward$)
            if $s \leq 0$ then
                $back\_len \leftarrow (i_0 - i - 1)$
                $smem\_len \leftarrow$ Fwd_len$[x] + back\_len$
                if $smem\_len \geq min\_smem\_len$ then
                    Append $[k, l, s]$ to SMEMs
                break
        $x \leftarrow x + 1$
    return SMEMs
```

Algorithm 2: Optimized SMEM computation

```
Input: String $P$, start position $i_0$, length of reference genome $|T|$, minimum
       required SMEM length $min\_smem\_len$, a parameter to turn on-off the
       optimization $max\_fwd\_distance$
Output: Set of bi-intervals $[k, l, s]$ of the SMEMs covering the base at $i_0$
1  Function
   CompSMEMOpt($P, i_0, |T|, min\_smem\_len, max\_fwd\_distance$)
   begin
2      Initialize $[k, l, s]$ as $[1, 1, 2|T|]$
3      Initialize Temp, Fwd_len, Back_intv and SMEMs as empty arrays
4      $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i_0], forward$)
5      Append $[k, l, s]$ to Temp
6      $i \leftarrow i_0 + 1$
7      $x \leftarrow 1$
8      Fwd_len[x] $\leftarrow 1$
9      $x \leftarrow x + 1$
10     while $i \leq |P|$ and $s > 0$ do
11         $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i], forward$)
12         if $s > 0$ and $[k, l, s] \neq Temp[x - 1]$ then
13             Append $[k, l, s]$ to Temp
14             Fwd_len[i] $\leftarrow i - i_0 + 1$
15             $x \leftarrow x + 1$
16         $i \leftarrow i + 1$
       ┌──────────────────────────────────────────────────────────
17     │ $x \leftarrow 1$
18     │ $start \leftarrow i_0$
19     │ $stop \leftarrow i_0$
20     │ while $x \leq$ |Temp| do
21     │     $[k, l, s] \leftarrow$ Temp[$x$]
22     │     if Back_intv is empty or $stop - start \geq max\_fwd\_dist$
       │     then
23     │         ($[k, l, s], back\_len$, Back_intv) $\leftarrow$
       │         BackEnlarge($[k, l, s], P, i_0$)
24     │         $start \leftarrow i_0 +$ Fwd_len[$x$]
25     │         $stop \leftarrow i_0 +$ Fwd_len[$x + 1$]
26     │     else
27     │         $stop \leftarrow$ Fwd_len[$x$]
28     │         ($[k, l, s], back\_len$) $\leftarrow$
       │         FwdEnlarge($[k, l, s], P, start, stop$, Back_intv)
29     │     $smem\_len \leftarrow$ Fwd_len[$x$] $+ back\_len$
30     │     if $smem\_len \geq min\_smem\_len$ then
31     │         Append $[k, l, s]$ to SMEMs
32     │     $x \leftarrow x + 1$
33     │     $max\_len \leftarrow$ Fwd_len[$x$] $+ back\_len$
34     │     while $max\_len < min\_smem\_len$ do
35     │         $x \leftarrow x + 1$
36     │         $max\_len \leftarrow$ Fwd_len[$x$] $+ back\_len$
       └──────────────────────────────────────────────────────────
37     return SMEMs
```

Algorithm 3: Backward enlargement

```
Input: The bi-interval $[k, l, s]$, string $P$, start postion $i_0$, array to store
       intermediate intervals Back_intv
Output: Bi-interval $[k, l, s]$, length of backward string $back\_len$
1  Function BackEnlarge($[k, l, s], P, i_0$) begin
2      Initialize Back_intv as empty array
3      $back\_len \leftarrow 0$
4      for $i \leftarrow i_0 - 1$ to 1 do
5          $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i], backward$)
6          if $s > 0$ then
7              Append $[k, l, s]$ to Back_intv
8              $back\_len \leftarrow back\_len + 1$
9          else
10             return ($[k, l, s], back\_len$, Back_intv)
```

Algorithm 4: Forward enlargement

```
Input: The bi-interval $[k, l, s]$, string $P$, start and stop index for enlargement
       $start$ $stop$ respectively, array of intermediate intervals Back_intv
Output: Bi-interval $[k, l, s]$, length of backward string $back\_len$
1  Function FwdEnlarge($[k, l, s], P, start, stop$, Back_intv) begin
2      $back\_len \leftarrow$ |Back_intv|
3      for $x \leftarrow$ |Back_intv| to 1 do
4          for $i \leftarrow start$ to $stop$ do
5              $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i], forward$)
6              if $s < 0$ then
7                  break
8              else
9                  if $i = stop$ then
10                     return ($[k, l, s], x$)
```

C. Our optimization

Algorithm 2 to 4 present our optimized implementation of the SMEM generation. We have improved the algorithm of the second `while` loop shown in a box in Algorithm 2. Calculation of the Temp array is same as in the original algorithm. Based on the observations discussed in the previous section we have made two improvements in the original algorithm.

1) If we are certain that after the backward enlargement of a bi-interval, the total length of the resulting exact match cannot be larger than the minimum required, we skip the backward enlargement of the bi-interval. This will avoid wasteful accesses to the FMD-index. This is implemented in lines 33 to 36 in Algorithm 2.

2) The SMEMs have to include the base at position i_0. All the strings covering the base at position i_0 can be visualized as a concatenation of two strings. One that contains the symbols at read positions greater or equal to i_0 and the other containing the symbols at read positions less than i_0. We may call them as *forward string* and *backward string* respectively. The forward string of an SMEM is found in the first `while` loop of Algorithm 1 or 2 (both are the same). Therefore, before starting the second loop we know the forward strings of all possible SMEMs. If the SMEMs corresponding to the bi-intervals in SMEMs array are numbered as SMEM$_i$, SMEM$_{i+1}$ ... SMEM$_{i+r}$ ..., where SMEM$_{i+1}$ is computed after SMEM$_i$, then

$$\text{SMEM}_i = P[i_0 - m_i] \ldots P[i_0 - 1]P[i_0]P[i_0 + 1] \ldots P[i_0 + n_i]$$

$$\text{SMEM}_{i+r} = P[i_0 - m_{i+r}] \ldots P[i_0 - 1]P[i_0]P[i_0 + 1] \ldots P[i_0 + n_{i+r}]$$

Now we will show that how our optimized method calculates SMEM$_{i+r}$ using SMEM$_i$. The first `while` loop mandates that $n_{i+r} > n_i$. Due to this and the reason that both SMEMs must not contain each other (condition 3 for an SMEM), $m_{i+r} < m_i$. In our technique, we compute SMEM$_i$ using the original method but while doing that we store the bi-intervals of all the intermediate strings formed during the backward enlargement in the Back_intv array. All these intermediate strings have the same forward string, but their

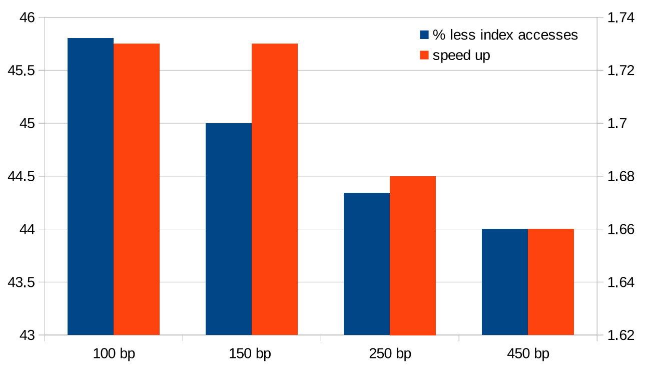

Fig. 2. Percentage reduction in FMD-index accesses (left Y-axis) and the corresponding speedup in SMEM generation (right Y-axis).

backward strings are extended by only one symbol. Hence, while computing SMEM$_i$ we keep track of the bi-intervals of all the strings: from $P[i_0-1]P[i_0]P[i_0+1]\dots P[i_0+n_i]$ to $P[i_0-m_i]\dots P[i_0-1]P[i_0]P[i_0+1]\dots P[i_0+n_i]$. As $m_{i+r} < m_i$, one of these strings is always $P[i_0-m_{i+r}]\dots P[i_0-1]P[i_0]P[i_0+1]\dots P[i_0+n_i]$ which is a substring of SMEM$_{i+r}$ and contains the full backward string of SMEM$_{i+r}$ but partial forward string ($n_{i+r} > n_i$). In our optimized version, this substring is enlarged in the forward direction by $(n_{i+r}-n_i)$ bases to find SMEM$_{i+r}$. To compute SMEM$_{i+r}$ using this method, total number of accesses to the FMD-index are $n_{i+r}+[(m_i-m_{i+r})\cdot(n_{i+r}-n_i)]$ as compared to $n_{i+r}+m_{i+r}$ in the original algorithm. The first term for both these methods is the same and it is observed that on the average, for small r, the second term of our technique is smaller than the second term of the original method, i.e. for small r, $(m_i-m_{i+r})\cdot(n_{i+r}-n_i) < m_{i+r}$. In our implementation we apply our optimization if $n_{i+r}-n_i \leq max\_fwd\_dist$ (default value 3), otherwise the computation is completed using the original method.

The results of our optimization are shown in Figure 2. The read lengths are given in number of base pairs (bps), which is the same as number of bases. The plot shows that for different read lengths the reduction in FMD-index accesses is 44-45% and the corresponding speedup varies from 1.66-1.73x. Our optimization only relies on algorithmic improvement and has limited overhead as we only need one extra array i.e. Back_intv. The maximum possible length of this array is equal to the read length.

IV. Suffix Array Lookup Acceleration

The suffix array lookup is taking around 15% of the computation time of the whole application. The objective is to retrieve the index in the reference string of one suffix string. In order to reduce the memory required for this, partial suffix and occurrence arrays are stored, and the inverse compressed suffix array (inverse CSA) is used [12]. The inverse CSA is given by:

$$\psi^{-1}(i) = C(B[i]) + O(B[i], i)$$

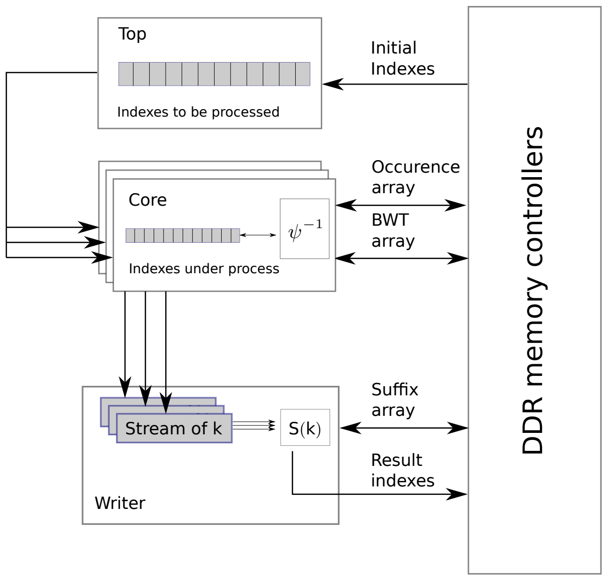

Fig. 3. Suffix array lookup architecture

where B is the BWT string, C is the count array and O is the occurrence array. The above equation is applied until the result is a factor of 32 and then the suffix can be computed using:

$$S(k) = S((\psi^{-1})^{(j)}(k)) + j$$

This implies that a series of memory reads are done to the B and O arrays. Due to their interdependency, these reads can not be prefetched by a general purpose cache. The result is that when running this function on the CPU, most of the time will be spent waiting for memory.

The hardware implementation can avoid waiting for the memory by using two main ideas: batching the input data and pipelining the memory reads. The batching will increase the number of memory reads that can be performed, while the pipelining will hide the memory latency. We implemented an architecture where indexes are sent to a number of hardware cores, and each hardware core processes a number of reads in a pipeline fashion. For the Convey platform, we implement a pipeline of 32 indexes, with 5 cores per FPGA, which gives a total of 640 indexes processed at the same time. This architecture can be see in Figure 3. All the represented blocks (Top, Core, Writer) will execute in parallel. Streams are used to avoid the need of synchronization. The number of core blocks can be adjusted depending on the FPGA area available.

V. Seed Extension Acceleration

The purpose of the seed extension kernel is to extend the length of an exact match while allowing for small differences, such as mismatches between the read and reference, or skipping symbols on either the read or reference. To obtain the extension, a method is used that is similar to the well-known Smith-Waterman algorithm [1], which is a dynamic programming method guaranteed to find the optimum alignment between two sequences for a given scoring system. A *similarity matrix* is filled that computes the best score out of all combinations of matches, mismatches and gaps.

A natural way to map dynamic programming algorithms onto reconfigurable hardware is as a linear systolic array. Many implementations that map the Smith-Waterman algorithm onto a systolic array have been proposed, amongst others [13], [14] and [15]. A systolic array consists of processing elements (PEs) that operate in parallel. In our case, we use such an array to take advantage of the available parallelism that exists while filling the similarity matrix, by processing the cells on the anti-diagonal in parallel. We map one read symbol to one PE. Hence, the length of the PE-array determines the maximum length of a read that can be processed. Each cycle, a PE processes one cell of the matrix and passes the resulting values to the next element.

A. Key differences

The seed extension kernel used in BWA-MEM is similar to the Smith-Waterman algorithm. However, since the purpose is to extend a seed, and not to find an optimal alignment between two sequences, three key differences arise:

1. Non-zero initial values: For an extension, the match between sequences will always start from their respective first symbols. Hence, unlike normal Smith-Waterman alignment, the initial values of the dynamic programming matrix are non-zero, but depend on the alignment score of the seed found by the SMEM generation function.

2. Additional output generation: Typically, a Smith-Waterman implementation generates a local and global alignment scores, which are the highest score in the matrix and the highest score that spans the entire read, respectively. The seed extension also returns the exact location inside the similarity matrix where these scores have been found. Furthermore, a *maximum offset* is calculated that indicates the distance from the diagonal at which a maximum score has been found. Therefore, the systolic array implementation passes additional values between the PEs compared to a regular Smith-Waterman systolic array implementation

3. Partial similarity matrix calculation: To optimize for execution speed, the software BWA-MEM uses a heuristic to only calculate those cells that are expected to contribute to the final score. Profiling reveals that in practice, only about 42% of all cells are calculated. This heuristic is not needed for our implementation, as it is able to perform all calculations on the anti-diagonal in parallel, which may also result in higher quality alignments.

B. Implementation details

Before deciding upon the final hardware design of the seed extension kernel, a number of ideas and designs alternatives have been considered, varying in acceleration potential, FPGA-resource consumption, suitability for certain data sets, and complexity.

A read has to travel through the entire systolic array, regardless of its actual length. To minimize latency, ideally a read would be processed by a PE-array matching its exact length. However, in practice this is not achievable, since it requires having a PE-array exactly matching each possible read length, which is impractical given the available resources on the FPGA. Therefore, we implemented an array with multiple exit points. This ensures that shorter reads do not have to travel through the entire array, reducing latency and increasing utilization and performance.

The FPGA-accelerated seed extension kernel is 1.5 times faster when comparing one module against one Xeon core. Our design contains three identical modules per FPGA, which is fast enough to completely hide the execution of this kernel by overlapping it with SMEM generation. The Seed Extension kernel implementation and design alternatives are described in more detail in [16].

VI. Results

The machine on which the implementation was tested is a Convey HC2ex. It consists of a host computer with 2 Intel Xeon CPU E5-2643 processors running at 3.30GHz (in total 8 cores) and 64 GB RAM memory. The co-processor is represented by 4 Xilinx Virtex-6 LX760 FPGA with a 64 GB memory attached.

The data set used to test the performance of the alignment algorithm is obtained from the GCAT framework [17]. The read length is 150 base pairs using pair ended reads.

The tests were run with the complete input and output in a RAM disk. A RAM disk is a virtual disk built in the DDR memory. This way, we eliminate any possible I/O limitations from the test. We note that there are options fast enough to feed and process the data output by the algorithm, either network or high performance disks, but we decided to choose the RAM disk solution to simplify as much as possible the methodology. Each test was run multiple times and the best result was selected. We measured the execution time of the original software, the hardware version without SMEM generation improvements and the version with all improvements. The results are presented in Table II.

We can notice, that due to Amdahl's law, accelerating Smith-Waterman and suffix array lookup, resulted only in a moderate speedup of 1.9x. Improving further the SMEM generation had a big impact, driving the total speedup to 2.6x

We also performed a more complete test of the behavior of the optimizations for different numbers of processors. The results are shown in Figure 4.

To show the possibilities of our design we analyzed the relation between FPGA number, core number and speedup obtained, considering the currently implemented hardware kernels. We assumed a 20% overhead for each FGPA for the memory controllers. The FPGAs and CPU cores are the same used for the implementation detailed above. The results are presented in Figure 5. We can see that to obtain a 2.5x speedup, for the current system we need to use only 2 FPGAs. For 3 FPGAs, we can get 2.5x speedup for up to 16 cores. For 4 FPGAs we can get the maximum speedup up until 20 CPU cores. It is worth mentioning that we focused our

TABLE II
EXECUTION TIMES (IN SECONDS) AND SPEEDUPS OF ACCELERATED BWA-MEM

Data set	SW exec	HW exec	Speedup vs SW	HW exec + SMEM opt	Speedup vs SW
gcat_set_041	534.00	280.00	1.91	208.00	2.57
gcat_set_042	530.50	279.00	1.90	208.00	2.55

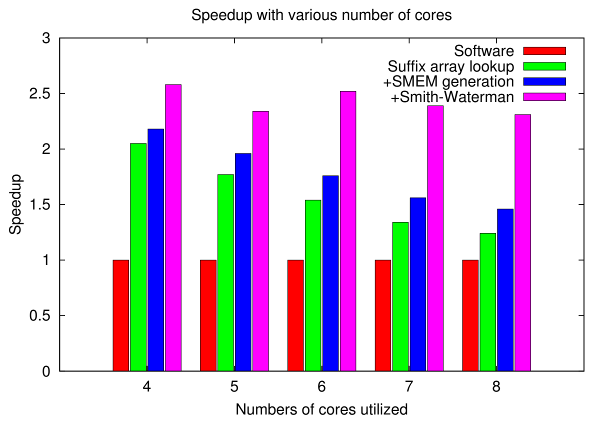

Fig. 4. Speedup for different numbers of CPU cores on the Convey HC2ex machine

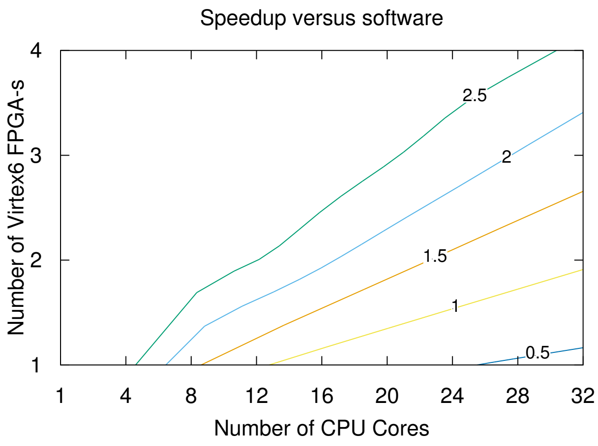

Fig. 5. Speedup in function of number of FPGAs and CPU cores.

implementation on the 8 core processor system, and we do not exclude further optimizations that would allow the area to be utilized better.

VII. Conclusion and Future Work

This paper presented a software optimized and hardware accelerated implementation of the well-known BWA-MEM DNA read mapping algorithm. The implementation focused on the three highest computationally expensive execution stages of the algorithm: SMEM generation, suffix array lookup and local Smith-Waterman. A system architecture was proposed to achieve a high acceleration for these components, containing two hardware-accelerated kernels and one kernel optimized in software. The two hardware kernels of suffix array lookup and local Smith-Waterman are able to reach speedups of 2.8x and 5.7x, respectively. The software optimization of the SMEM generation kernel is able to achieve a speedup of 1.7x. This enables a total application acceleration of 2.6x compared to the original software version. Analysis shows that the implementation is bottlenecked by the software part, which indicates that further acceleration of BWA-MEM functions in hardware could achieve higher performance. This will be the focus of our future work.

References

[1] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.
[2] T. W. Lam *et al.*, "Compressed indexing and local alignment of dna," *Bioinformatics*, vol. 24, no. 6, pp. 791–797, Mar. 2008.
[3] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv [q-bio.GN]*, May 2013. [Online]. Available: http://arxiv.org/abs/1303.3997
[4] E. Fernandez *et al.*, "FHAST: FPGA-based acceleration of Bowtie in hardware," *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 2015.
[5] "Hybrid Core Computing and Bioinformatics Applications," http://www.hpcsociety.org/Resources/Documents/121212Kirby-CONVEY-SHPCP_121212.pdf.
[6] Y. Liu and B. Schmidt, "Long read alignment based on maximal exact match seeds," *Bioinformatics*, vol. 28, no. 18, pp. i318–i324, 2012.
[7] ——, "Cushaw2-gpu: Empowering faster gapped short-read alignment using gpu computing," *Design Test, IEEE*, vol. 31, no. 1, pp. 31–39, Feb 2014.
[8] H. Li *et al.*, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.
[9] "Genome Comparison and Analytic Testing," http://www.bioplanet.com/gcat.
[10] H. Li, "Exploring single-sample SNP and indel calling with whole-genome de novo assembly," *Bioinformatics*, vol. 28, no. 14, pp. 1838–1844, Jul 2012.
[11] J. Zhang *et al.*, "Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures," in *CCGrid*, Delft, Netherlands, May 2013.
[12] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.
[13] T. Oliver, B. Schmidt, and D. Maskell, "Hyper customized processors for bio-sequence database scanning on FPGAs," in *Proceedings of the ACM/SIGDA*. ACM, 2005, pp. 229–237.
[14] C. W. Yu *et al.*, "A Smith-Waterman systolic cell," in *New Algorithms, Architectures and Applications for Reconfigurable Computing*. Springer, 2005, pp. 291–300.
[15] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform," in *Proceedings of the HPRCTA'07*. ACM, 2007, pp. 39–48.
[16] E. Houtgast, V. Sima, K. Bertels, and Z. Al-Ars, "An FPGA-Based Systolic Array to Accelerate the BWA-MEM Genomic Mapping Algorithm," in *SAMOS XV*. IEEE, 2015.
[17] G. Highnam *et al.*, "An analytical framework for optimizing variant discovery from personal genomes," *Nature communications*, vol. 6, 2015.