Maximizing Systolic Array Efficiency to Accelerate the PairHMM Forward Algorithm

Johan Peltenburg, Shanshan Ren, Zaid Al-Ars, Computer Engineering Laboratory, TU Delft
E-mail: {j.w.peltenburg, s.ren, z.al-ars}@tudelft.nl

***Abstract*—In the analysis of next-generation DNA sequencing data, Hidden Markov Models (HMMs) are used to perform variant calling between DNA sequences and a reference genome. The PairHMM model is solved by the Forward Algorithm, for which the performance and power efficiency can be increased tremendously using systolic arrays (SAs) in FPGAs. We model the performance characteristics of such SAs, and propose a novel architecture that allows the computational units to continuously perform useful work on the input data. The implementation achieves up to 90% of the theoretical throughput for a real dataset. The implementation of the proposed architecture achieves more than 2.5x throughput over the state-of-the-art on a similar contemporary platform.**

***Keywords*—High-Throughput Sequencing, GATK, HaplotypeCaller, PairHMM, Systolic Array, FPGA**

I. Introduction

Next-generation DNA sequencing methods allow cost-effective sampling of DNA [1]. This data is used e.g. to understand and treat human diseases. The analysis of the huge amounts of data resulting from such samples is still a computational challenge today. Hidden Markov Models (HMM) are used during analysis to find pairwise alignments of DNA sequences. More specifically PairHMMs [2] can be used to calculate the probability that two sequences are related, which is called the overall alignment probability. In this work, we consider the alignment probability of a read to a haplotype.

Because of the computational complexity and the data volume, PairHMM calculations in genome analysis pipelines (such as Genome Analysis ToolKit or GATK [3]) take a long time to complete on conventional machines. However, the PairHMM Forward Algorithm, which is also used in the software implementation of the GATK HaplotypeCaller, is an algorithm exhibiting a long datapath. Such algorithms are often good candidates for FPGA implementation. An FPGA accelerator is often able to achieve a high throughput and high power-efficiency. In other research, it has been shown that FPGAs can be suitable candidates to implement the algorithm using Systolic Arrays (SAs). However, a drawback of some architectures is that the computational resources are sometimes under-utilized due to control issues or data padding.

In this work, we attempt to optimize SA utilization, allowing for near continuous processing on all the computational elements of the SA. Our future aim is to implement many small but efficient SAs instead of implementing one large but inefficient SA. Our contributions are as follows:

- We provide a model to calculate the utilization of an SA.
- We analyze architectural alternatives allowing continuous processing of the PairHMM Forward Algorithm.
- We implement one such architecture that is more than 2.5x faster than the state-of-the-art FPGA implementation and 10x faster than a state-of-the-art CPU.

II. Background

A. PairHMM Forward Algorithm

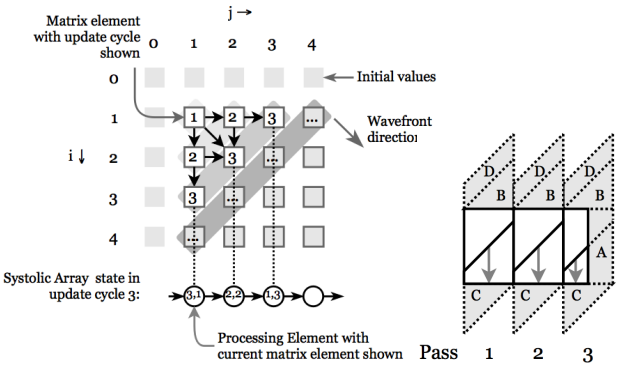
Algorithm 1 PairHMM Forward Algorithm used in the GATK HaplotypeCaller

$M \leftarrow I \leftarrow D \leftarrow 0_{X+1,Y+1}$
$D_{0,0\ldots Y} \leftarrow C_{init}$
for $i \leftarrow 1, X$ **do**
  for $j \leftarrow 1, Y$ **do**
    $M_{i,j} \leftarrow \alpha_{i,j} \cdot (\beta_i \cdot M_{i-1,j-1} + \gamma_i \cdot I_{i-1,j-1} + \gamma_i \cdot D_{i-1,j-1})$
    $I_{i,j} \leftarrow \delta_i \cdot M_{i-1,j} + \epsilon_i \cdot I_{i-1,j}$
    $D_{i,j} \leftarrow \eta_i \cdot M_{i,j-1} + \zeta_i \cdot D_{i,j-1}$
return $\sum_{j=0}^{Y} M_{X,j} + I_{X,j}$

The PairHMM Forward Algorithm as implemented in the HaplotypeCaller is seen in Algorithm 1. M, I and D are the matrices for match, insertion and deletion probabilities. $\alpha_{i,j}$ is the emission probability: for each position in the read i it can have two different values, depending on the bases of the read and haplotype at position i and j. β, γ, δ, ϵ, η and ζ are transmission probabilities that only depend on the read position i. In the software implementation, all probabilities are floating-point values. We define X and Y as the length of the read and haplotype, respectively.

When updating some cell (i, j) of the matrices M, I and D, a dependency exists on the values of cells $(i-1, j-1)$, $(i-1, j)$ and $(i, j-1)$. Thus, only matrix cells laying on the anti-diagonals of the matrix can be updated in parallel. Therefore, Algorithm 1 is commonly implemented in hardware using a one-dimensional systolic array (SA) consisting of a number of processing elements (PEs). Each PE implements the inner loop in the algorithm, updating one cell in each of the matrices M, I, and D. During every update cycle, the SA updates the cells on the anti-diagonal of the matrices (sometimes called a 'wavefront'). A simplified diagram of such an SA can be seen in Fig. 1a. As the anti-diagonal grows, the amount of exploitable parallelism grows as well.

When the length of the haplotype (or read) is larger than the number of elements in the SA, the SA can compute the matrices by making multiple vertical (or horizontal) *passes* through the matrix, processing only a subset of columns (or rows) and wrapping back to the top (or side) of the matrix after completion of a pass. This can be seen in Fig. 1b. The values in the last column (or row) in the pass are often stored



(a) One pass detail (b) Multiple passes and potential overhead

Fig. 1: An example of how an SA can solve a PairHMM using the Forward Algorithm (Algorithm 1).

in a FIFO buffer. Whenever a pass is shorter than the amount of PEs in the SA, padded data is inserted (Fig. 1b case A).

B. Related work

Earlier research discussed using SAs to solve similar HMM-based algorithms in the field of computational biology [4] [5]. These proposed SA designs introduce overhead when model parameters must be reconfigured between subsequent passes or workloads. Subsequent research such as [6] and [7] show more advanced SA designs, deploying double buffering of model parameters of alternating passes and workloads, allowing for near continuous processing.

More recent work implements the same PairHMM Forward Algorithm as this work in FPGA on the Convey Computer platform, showing higher throughput than single threads of the host processor [8]. However, the architecture introduces overhead when switching between passes, as parameters are shifted into the PEs. In [9], which we consider as the current state-of-the-art FPGA implementation, PEs are partially internally pipelined, achieving a high throughput. This design uses the CAPI interface of the IBM POWER8 platform, which we will also use in this work.

In this paper, we introduce a new architecture that is able to continuously perform *useful* calculations in the PEs of the SA. Once the first input data pair is loaded, our design wastes virtually no cycles due to memory latency or parameter reconfiguration. Thus, the design is able to achieve extremely close to the maximum theoretical performance of a fixed-size SA.

III. Performance model

We define the length of the read and the haplotype as X and Y. The total amount of cell updates required to process the Forward Algorithm is $X \times Y$. A useful measure of performance for the Forward Algorithm is the throughput in number of cell updates per second (CUP/s). In this paper, we will only count *effective* cell updates, which are cell updates that contribute to the final result (i.e. not on padded data).

The throughput of an SA design is affected by the average utilization of the PEs. We observe that while processing the Forward Algorithm with an SA, under-utilization of the PEs may be introduced in several cases (also shown in Fig. 1b):

(A) When data is padded if a pass is not as wide as the SA.
(B) If the PEs in the SA may only work on one pass at a time, under-utilization of the PEs occurs at the start of a pass.
(C) Same as B, but at the bottom of a pass.
(D) When switching between passes, to update the model (α, β, etc.) in the PEs.
(E) When the height of the matrix is shorter than the number of PEs, and more than one pass is required, the read must be padded. Otherwise, the feedback FIFO will not contain any data yet for the first PE to work on in the next pass. (Not shown in Fig. 1b).

We consider an SA of fixed size, thus the overhead introduced in case A and E is inevitable. However, we aim to eliminate the other causes of overhead.

A. Fixed-size systolic array performance

Consider the processing of the Forward Algorithm in an SA where; W is the width of the matrix, H is the height of the matrix and E is the number of PEs in the SA. Also, assume one cell update per clock cycle. In the ideal case, if we would process a large amount of pairs (thereby ignoring initial and final latency), that are of similar size, and if the input data is available at any time at the inputs of the PEs, the average utilization of the whole SA for one pair is given by:

$$\text{Avg. utilization} = \frac{WH}{E\lceil \frac{W}{E} \rceil \cdot \max(E, H)} \tag{1}$$

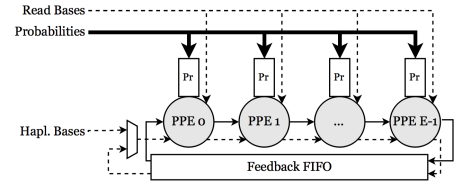
Eq. 1 takes the number of cells in the original matrices and divides this by the number of cells in the padded matrices. This gives the ratio of effective cell updates verses all cell updates (including padding). In the case of such a workload, we may obtain the average number of *effective* cell updates U_{avg} per clock cycle by multiplying the average utilization by the number of PEs in the SA:

$$U_{avg}(W, H, E) = \frac{WH}{\lceil \frac{W}{E} \rceil \cdot \max(E, H)} \tag{2}$$

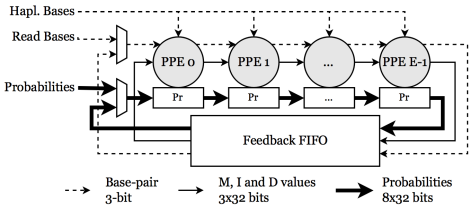
Thus, cells padded to the bottom of the matrix (in each pass, only when $H < E$) and cells padded to the right of the matrix in the final pass are also taken into account.

If the height of the matrix is equal or larger than the number of PEs (i.e. $H \geq E$) and the width of the matrix is an integer multiple of the number of PEs (i.e., $W = nE, n \in \mathbb{Z}_{>0}$), all PEs perform useful work in every pass. In this case, maximum throughput is achieved ($U = E$). This also shows an SA of length $E = 1$ is always maximally efficient (i.e. an SA of this size needs no padding, since passes are of width 1).

Modern FPGAs contain enough computational fabric to implement a large number of PEs. However, the number of SAs cannot be as high, since it quickly becomes bounded by the available memory and interconnect. For example, the FPGA used for this work offers enough resources to implement 112



(a) Architecture HS: haplotype data is streamed in horizontally, read data is streamed in vertically.



(b) Architecture RS: read data is streamed in horizontally, haplotype data is streamed in vertically

Fig. 2: Two SA architectures.

PEs, but the FPGA lacks resources to implement 112 SAs in parallel, requiring 112 controllers, input buffers, feedback FIFOs and other items in the data and control paths. A more feasible combination would be to have, e.g. 7 SAs of 16 PEs each. This work focuses on implementing an architecture for a single SA, that achieves as close to the maximum performance of Eq. 2 as possible.

IV. Alternative architectures

A. Alternative architectures

To achieve the maximum performance, the matrix can be mapped onto the SA in two ways. In one, (HS in Fig. 2a), the data that depends on the haplotype position (haplotype bases) is streamed-in at the head of the SA. The data that depends on the read position (probabilities and read bases) is fed vertically into the PEs. In this approach, the matrix is mapped to have the read on the horizontal axis, and the haplotype on the vertical axis of the matrices. The other approach (RS, Fig. 2b) has horizontal and vertical data streams swapped.

All data that is fed *horizontally* can be streamed from input FIFOs into the head of the SA. When reuse of this data is required in a new pass, the feedback FIFO will provide this data and intermediate values that were streamed out of the SA after processing the last column of the previous pass. All data that is fed *vertically* can be distributed to the respective PEs using a bus connected to registers (or RAM).

Although architectures similar to HS are often used (with the exception of [6]), we argue the use of RS. The reason to select RS is related to the sizes of the read and haplotype, X and Y. The haplotype is at least as long as the read, but often much longer. Consider again Eq. 2. When the ratio between fully utilized passes and underutilized passes is high (i.e. when Y is large) the efficiency is also high, since a relatively larger number of passes will have full SA utilization.

Internally, the PEs are pipelined, such that the critical path in the circuit is reduced, allowing higher clock frequencies for the whole SA. The throughput of the SA is directly proportional to its clock frequency.

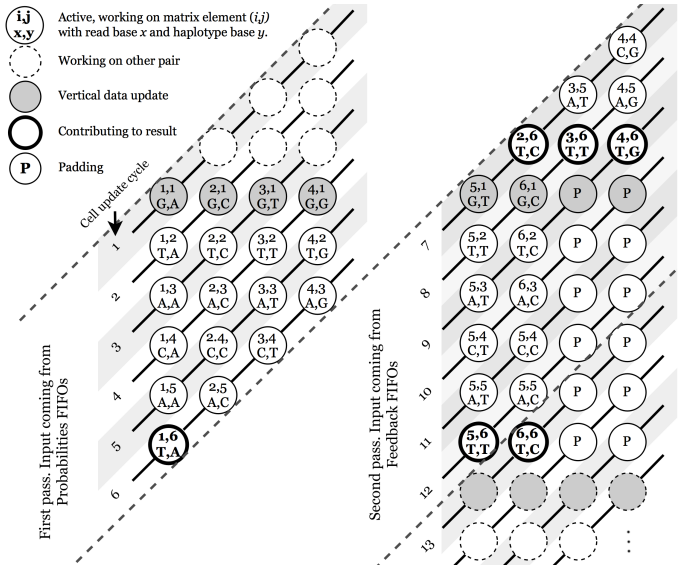


Fig. 3: Example of processing a pair for which the read length $X = 6$, the haplotype length $Y = 6$ and the number of PEs $E = 4$.

B. Maximizing utilization

To achieve maximum utilization, overhead from the cases B, D and C described in Section III must be prevented. This can be done by observing that, during one cell update cycle, the vertical data of *at most* one PE needs to be updated, i.e. at most one PE in the SA will enter a new pass in each cell update cycle. Therefore, a bus connected to the vertical data registers needs to transfer the vertical data of only one PE per cycle.

In this way, any data that is still in the SA from a previous pass or pair does not have to be completely streamed out, allowing cell updates between passes and pairs to take place within the SA (solving case B and C). Furthermore, when the vertical data bus is able to transfer all required data in one cycle, overhead caused by updating model parameters in the PEs can be avoided (solving case D)

An example of continuous processing on the RS architecture is given for the following case: The number of PEs, $E = 4$, the length of the read $X = 6$, the length of the haplotype: $Y = 6$, the read is 'GTACAT' and the haplotype is 'ACTGTC'.

As shown in Fig. 3, on each anti-diagonal, the state of the complete SA is depicted during one cell update cycle, and superimposed over the matrix cells of a pass. For each cell update cycle, the vertical data of *at most* one PE must be updated. Similarly, the output of at most one PE holds data contributing to the final result. Therefore, the M and I output of each PE are logically OR-ed with each other and sent to an accumulator. This implements the last line of the procedure in Algorithm 1. By setting the haplotype and read base to a value called "Padding" (denoted by 'P' in the figure), the PEs output will be invalidated.

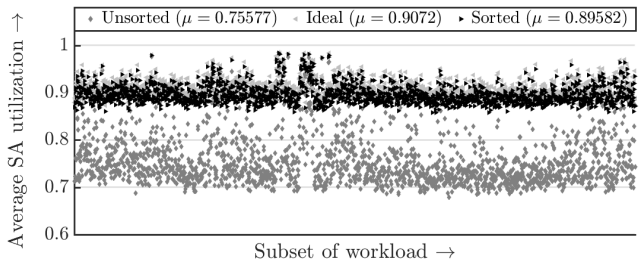


Fig. 4: Effect of sorting on the efficiency of the SA, with E=16.

C. Control mechanism

Since PEs are internally pipelined (Section IV-A), to allow multiple PairHMMs to run in each of the pipeline slots, one could use BRAM and allocate a specific region for each of the N pairs that is active in an N stage pipeline. However, such a control mechanism is complex, since it must track all SA control signals, as well as RAM addresses, for each of the N pipeline slots independently. At the side of the memory interface, it must keep track of N pointers, data counters, and other control information.

The control mechanism can be extremely simplified by allowing the smallest unit of processing to be *batches* of N pairs. By implementing FIFOs for the input data, the host can prepare a batch of N pairs to be processed, ordering the batch in memory in such a way that the accelerator itself does not have to deal with ordering at all. The accelerator keeps track of control signals of only one batch instead of keeping track of all control signals for each of the N pairs.

Although simplifying control complexity, batches have a minor drawback in terms of performance; if the pairs contained in the batch are of completely different sizes, smaller pairs require a lot of padding, in turn decreasing SA efficiency.

Consider the processing of N pairs in a batch, where the n-th pair has read length X_n and haplotype length Y_n. The total amount of work required in cell updates U_{req} to process the batch is given by:

$$U_{req} = \sum_{n=0}^{N-1} X_n Y_n \tag{3}$$

When the amount of work done on a batch U_{batch} is determined by the largest read and haplotype, it can be calculated (containing overhead due to padding) using Eq. 2 as follows:

$$U_{batch} = N \cdot U_{avg}(\max_n Y_n, \max_n X_n, E) \tag{4}$$

Dividing Eq. 3 by Eq. 4 gives the efficiency per batch.

When the read and haplotype lengths are different, the SA has low efficiency due to abundant padding. A large portion of this drawback can be mitigated by sorting the pairs by number of passes required, then sorting each list of pairs with the same number of passes by read size. After sorting, the batches are created by the host and sent to the accelerator. When the workload is very large, sorting makes it likely that haplotypes and reads inside a batch share a similar number of passes and read size.

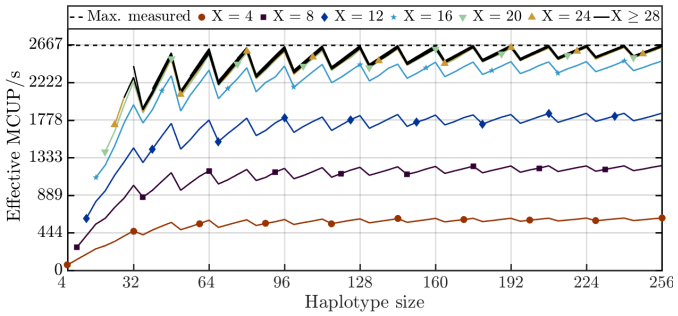


Fig. 5: Synthetic benchmark. PEs: $E = 16$. Workload size: 2^{14}. Step size: 4. Read size: X. Theoretical maximum throughput: 2667 MCUP/s. Max. measured: 2661 MCUP/s.

To reduce the sorting time, we sort only small subsets of the workload. For the whole genome sequencing dataset we used for this work (see Section VI), we split the workload into 1832 subsets of 2^{14} pairs and sort them. In Fig. 4, we compare it to the SA utilization when using unsorted subsets and the ideal utilization given by Eq. 2, in the case where we would not use batches, but are able to start working on pairs in independent pipeline slots. We find that using sorted batches almost achieves ideal performance.

V. Implementation

We implemented architecture RS using an AlphaData ADM-PCIE-7V3 FPGA accelerator card, for which a POWER8 CPU on an IBM Power System S824L (8247-42L) serves as a host. This system offers the Coherent Accelerator Processor Interface (CAPI) to the accelerator through IBMs Power Service Layer (PSL) interface. The memory interface at the host side is therefore similar to [9]. To abstract away the PSL interface, we use the CAPI Streaming Framework from [10].

The SA consists of E Pipelined Processing Elements (PPEs). Each PPE implements the inner loop of Algorithm 1 as a 16-stage pipeline. The maximum number of PPEs we could fit (using Vivado 2016.2) was 112. This bound is determined by the number of DSP blocks. The DSP blocks are used by the floating-point units in the PPEs. The FPGA allows 3600 DSP blocks to be used, but the PSL is distributed as a pre-routed design and prevents the use of a quarter of the DSP blocks. In this work, we implement the SA using $E = 16$ and $E = 32$.

VI. Experimental results

To measure the performance for different sizes, we generate workloads of increasing read (X) and haplotype (Y) size, where $Y \geq X$, in steps of 4. Each workload contains 2^{14} pairs. The performance for each workload is shown in Fig. 5. Our SA runs at 166.7 MHz, thus the maximum theoretical throughput is $E \cdot f$ in cell updates per second (CUP/s).

Padding in the horizontal direction (when $X < E$), deteriorates the throughput, as the utilization of the SA is very low. When there is no padding in the horizontal direction, the throughput quickly grows towards the maximum theoretical throughput. Also, the effect of having haplotype sizes of integer multiples of the number of PEs is clearly visible.

TABLE I: FPGA post-routing power estimate and area

Part	LUTs	Registers	RAM36	DSP	Power(W)
Available: 7VX690	433200	866400	1470	3600	
16 PEs + interfaces	119937	140397	473	378	11.212
16 PEs, this work only	47346	60525	181	354	2.721
32 PEs + interfaces	163450	189085	473	730	13.213
32 PEs, this work only	90862	109213	181	706	4.585

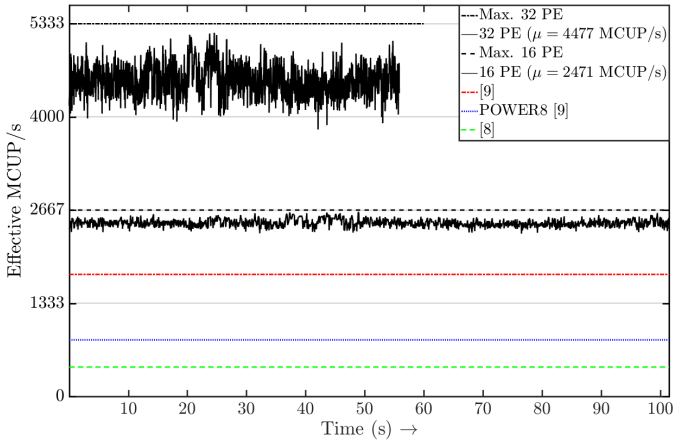


Fig. 6: SA throughput using a real dataset with $E = 16$ and $E = 32$. Subsets size 2^{14}

In this case, the performance nears the maximum theoretical throughput. The highest throughput measured was 99.76% of the maximum. The last bit of overhead is introduced by the memory latency at initialization and termination.

For a realistic benchmark, we use the same dataset as the work presented in [9] (whole human genome dataset G 15512.HCCI954.1 mapped to chromosome 10). The dataset contains over 30 million pairs. We split and sort the dataset in subsets of 2^{14} pairs. The results for sizes $E = 16$ and $E = 32$, the maximum theoretical throughput for each SA, the reported throughput of [9] and [8] and the reported maximum for the POWER8 host CPU are shown in Fig. 6. For $E = 32$, we achieve a throughput of 84% of the maximum performance; for $E = 16$, this is 93%. The lower throughput for $E = 32$ is caused by the large number of reads in the dataset of which the size is smaller than E, resulting in much variation. However, for the SA with $E = 16$, we observe that the utilization is higher, since padding occurs less. Although for $E = 32$, the SA is twice as long as for $E = 16$, the run-time is only 1.8x lower. Furthermore, with the same amount of processing elements, our architecture shows an average improvement of throughput of 2.5x over the state-of-the-art. With half the processing elements, our implementation achieves a 1.4x higher throughput.

In Table I the area statistics of the SA design with 16 and 32 PEs are shown after placing and routing. We show the logic available in the device, the logic utilization of our system (including interfaces) and for our design only. Moreover, the power estimation of Xilinx Vivado is included. From Table I and Fig. 6, we estimate the power efficiency to be $339 \cdot 10^6$ CUP/J.

VII. Conclusion

We analyzed the efficiency of systolic arrays that implement the PairHMM Forward Algorithm to find the overall alignment probability of a read to a haplotype. This paper shows architectures which can implement fixed-size SAs in such a way that the overhead is minimal. We implemented one of the architectures, where the data corresponding to the read position is streamed through the systolic array. This implementation achieves 99.76% of the theoretical maximum performance for a synthetic dataset, and around 90% for a real dataset, depending on the size of the systolic array and the read-haplotype pairs. A systolic array with 32 processing elements is able to calculate the overall alignment probabilities of a whole genome dataset mapped to chromosome 10 in under 60 seconds, while only using approximately one third of the FPGAs DSP resources.

In future work, we aim to implement several small SAs in parallel, such that each SA may achieve a high utilization, increasing the overall throughput.

Acknowledgment- This work was supported by the European Commission in the context of the ARTEMIS project ALMARVI (project #621439).

References

[1] J. Shendure and H. Ji, "Next-generation dna sequencing," *Nature biotechnology*, vol. 26, no. 10, pp. 1135–1145, 2008.

[2] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[3] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. Del Angel, M. A. Rivas, M. Hanna, *et al.*, "A framework for variation discovery and genotyping using next-generation dna sequencing data," *Nature genetics*, vol. 43, no. 5, pp. 491–498, 2011.

[4] A. C. Jacob, J. M. Lancaster, J. D. Buhler, and R. D. Chamberlain, "Preliminary results in accelerating profile HMM search on FPGAs," in *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, IEEE, 2007.

[5] K. Benkrid, P. Velentzas, and S. Kasap, "A high performance reconfigurable core for motif searching using profile HMM," in *Adaptive Hardware and Systems, 2008. AHS'08. NASA/ESA Conference on*, pp. 285–292, IEEE, 2008.

[6] Y. Sun, P. Li, G. Gu, Y. Wen, Y. Liu, and D. Liu, "Accelerating HMMer on FPGAs using systolic array based architecture," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–8, IEEE, 2009.

[7] M. N. M. Isa, K. Benkrid, and T. Clayton, "A novel efficient FPGA architecture for HMMER acceleration," in *2012 International Conference on Reconfigurable Computing and FPGAs*, pp. 1–6, IEEE, 2012.

[8] S. Ren, V.-M. Sima, and Z. Al-Ars, "FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis," in *IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 1465–1470, IEEE, 2015.

[9] M. Ito and M. Ohara, "A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for pair-HMM algorithm," in *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*, pp. 1–3, IEEE, 2016.

[10] M. Brobbel, "CAPI Streaming Framework." https://github.com/mbrobbel/capi-streaming-framework, 2016.