Hardware Acceleration of the Pair-HMM Algorithm for DNA Variant Calling

Sitao Huang[†], Gowthami Jayashri Manikandan[†], Anand Ramachandran[†],
Kyle Rupnow[‡], Wen-mei W. Hwu[†], Deming Chen[†]

[†]University of Illinois at Urbana-Champaign, USA
{shuang91, manknd n2, aramach4, w-hwu, dchen}@illinois.edu

[‡]Advanced Digital Sciences Center, Singapore
k.rupnow@adsc.com.sg

ABSTRACT

With the advent of several accurate and sophisticated statistical algorithms and pipelines for DNA sequence analysis, it is becoming increasingly possible to translate raw sequencing data into biologically meaningful information for further clinical analysis and processing. However, given the large volume of the data involved, even modestly complex algorithms would require a prohibitively long time to complete. Hence it is the need of the hour to explore non-conventional implementation platforms to accelerate genomics research. In this work, we present an FPGA-accelerated implementation of the Pair HMM forward algorithm, the performance bottleneck in the HaplotypeCaller, a critical function in the popular GATK variant calling tool. We introduce the PE ring structure which, thanks to the fine-grained parallelism allowed by the FPGA, can be built into various configurations striking a trade-off between instruction-level parallelism (ILP) and data parallelism. We investigate the resource utilization and performance of different configurations. Our solution can achieve a speed-up of up to 487× compared to the C++ baseline implementation on CPU and 1.56× compared to the best published hardware implementation.

Keywords

Hardware Acceleration; FPGA; forward algorithm; Pair-HMM; Computational Genomics; PE ring

1. INTRODUCTION

Bioinformatics is a fast-growing field, with increasing demand for high computational capabilities for several applications. Next Generation Sequencing (NGS) technologies and the increasing availability of genome data through public databases, have enabled us to develop comprehensive pipelines to sequence and process complex genomes. Translation and interpretation of the raw sequencing data to biologically meaningful information for further clinical analysis like disease prediction, drug performance evaluation etc., are of utmost importance now. Several algorithms have been developed to address problems like DNA sequence alignment [1, 2, 3, 4], error correction [5, 6, 7] and variant calling [8]. Many bioinformatics workflows suffer from very high computational times brought forth by the explosion in data available from low-cost, high-throughput sequencing. In response to this, several bioinformatics algorithms have been implemented on alternative computing platforms like FPGAs and GPUs to reduce their execution times. Such recent developments point to workflows being executed on heterogeneous computing platforms where instructions for the critical and computation intensive parts of an algorithm will be off-loaded for execution on an FPGA or a GPU to achieve significant speed-ups compared to a CPU-only implementation. Parallelization and acceleration of complex bioinformatics algorithms has become an area that is being widely explored [9, 10, 11, 12].

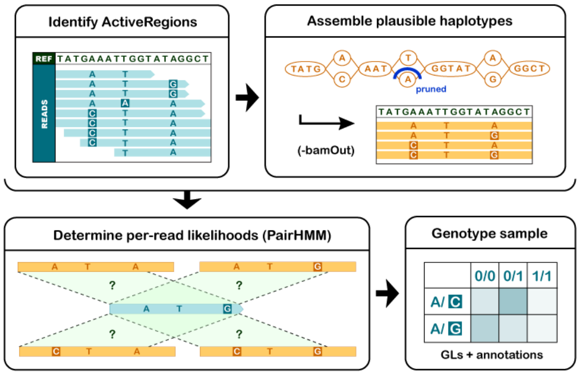

Figure 1: The major steps in HaplotypeCaller [13].

Our work deals with a specific bioinformatic analysis called variant calling. Variant calling identifies differences between a given subject's DNA and a standard reference DNA. The input data to the analysis are the standard reference DNA and the sequencing data of the individual in the form of alignment files. Alignment files are a specific type of representation of the sequencing data showing the sequenced reads and the most likely areas they are sequenced from with

FPGA '17, February 22-24, 2017, Monterey, CA, USA

© 2017 ACM. ISBN 978-1-4503-4354-1/17/02...$15.00

DOI: http://dx.doi.org/10.1145/3020078.3021749

respect to the reference sequence. These associations are determined by minimizing the edit-distance between a read sequence and regions in the reference sequence.[1, 2] Edit-distance minimizations may be considered valid because the error rates involved in the sequencing technologies of interest are small and localized, and we are primarily interested in small variations from the reference.

GATK's HaplotypeCaller [14, 13] is one of the most popular variant calling tools available today. The tool does the analysis in two steps. In the first step, it identifies locations in the genome where the chances are high that a variation is present based on simple computations. These locations, called active sites, are processed further to confirm the initial assessment.

To determine if an active site in the sample is a variant or not, the tool assembles what it thinks are the probable haplotypes in the region surrounding the active site. The probable haplotypes in a location are the tool's initial guess regarding the different sequences of DNA present in that particular location. More than one sequence may be found at a given location in the genome because, for instance, the human genomic material is made up of pairs of chromosomes, and two chromosomes in a pair can be different locally, while being very similar globally. Once the probable haplotypes are constructed, the tool assesses which of the candidate haplotypes are most likely present in the individual's genome by calculating the likelihood of alignment of each haplotype to the read sequences in that region in the input alignment file. If the established haplotypes are different from the reference sequence, the tool determines that there is a variation at that location. The general flow of the HaplotypeCaller is summarized in Figure 1.

GATK's HaplotypeCaller assumes that haplotypes and read sequences follow a pair hidden Markov model (Pair-HMM). Pair-HMM [15] is a popular statistical model to study pairwise alignment probabilities of two sequences. We can infer several aspects of the alignment using various inference algorithms of the Pair-HMM model such as, optimal sequence alignment (Viterbi algorithm) and the overall alignment probability (forward algorithm). The forward algorithm of the Pair-HMM gives the statistical measure of the similarity between two sequences. It computes the overall alignment probability by summing the likelihood of all possible alignments between the two sequences. The forward algorithm for the Pair-HMM model are used in several applications like gene prediction, functional similarity analysis between two protein sequences and variant calling [16, 17, 18]. Specifically, it is used by GATK's HaplotypeCaller to measure similarity between reads and probable haplotypes.

Table 1 shows the runtime results of each stage of the HaplotypeCaller for Chromosome 20 from sample NA12878 (Whole Genome Sequence data) as published by Mauricio et al. of Broad Institute [19]. This shows that the major bottle-neck is the forward algorithm computations of the Pair-HMM. In the Pair-HMM stage, every candidate haplotype is compared to each input read to compute the likelihood score using the forward algorithm. The number of computations involved is of the order $N \times M \times R \times H$, where N is the number of input reads, M is the number of candidate haplotypes, R is the length of the input reads and H refers to the length of the candidate haplotype.

In this work, our objective is to achieve a speed-up of the Pair-HMM's forward algorithm on an FPGA-based computing platform to minimize the bottleneck of computing flows similar to GATK's HaplotypeCaller that utilize such a Pair-HMM model to compute the overall alignment probability.

Table 1: Profiling results for HaplotypeCaller run on Chromosome 20 of NA12878 Whole Genome Sequence (WGS) sample [19].

Stage	**Time**	**Runtime**
Assembly	2,598s	13%
Pair-HMM (forward algorithm)	14,225s	70%
Traversal + Genotyping	3,379s	17%

Pair-HMM is a type of more complicated dynamic programming algorithm, compared to many other programming algorithm. The propagation operator in Pair-HMM is complicated combination of floating-point additions and multiplications, rather than simple `min` or `max` operators as in many other dynamic programming algorithms. And, there are three matrices that each has data dependencies on the other two. Besides, Pair-HMM requires all the values to be at least single-precision floating-point numbers. Normalizing to fix point domain will lead to either overflow or underflow problem. Performing floating-point operations efficiently usually a hard problem for FPGAs.

Traditionally, in FPGA-accelerated solutions to the Pair-HMM, systolic arrays have been commonly used. However, the systolic array structure lacks the flexibility of handling variable input lengths and hence lacks design scalability and configurability. In this work, we propose a Processing Element (PE) ring structure to compute the forward algorithm. To the best of our knowledge, this is the first PE ring structure based implementation of the Pair-HMM forward algorithm. The PE ring structure can be configured for inputs of varied lengths without any changes to the hardware. In the following sections, we will demonstrate how our PE ring structure exhibits significant advantage in terms of performance and flexibility.

The contributions in this work may be summarized as follows:

- We propose a ring-based hardware implementation of the Pair-HMM's forward algorithm, which can support flexible lengths for input read sequences. Our implementation can achieve significant speed-ups of up to 487× compared to the C++ baseline implementation on CPU, and 1.56× compared to the published best hardware implementation.

- Several optimization techniques for improving PE ring's performance are proposed and discussed, the PE ring structure and the optimization may be readily extended to accelerate other applications that are based on similar dynamic programming algorithms.

- Based on both the above implementation and optimization techniques, we present experimental results to illustrate the trade-offs and other design considerations in using the PE ring structure to accelerate dynamic programming algorithms.

- Our work provides an example of how to effectively accelerate complicated floating-point dynamic programming calculation with FPGA.

The rest of the paper is organized as follows: Section 2 provides an overview of the prior implementations of bioinformatics algorithms that are similar to the Pair-HMM; Section 3 introduces the fundamentals of the forward algorithm of the Pair-HMM; Section 4 presents the details of our ring-based implementation of the forward algorithm; Section 5 illustrates the experimental results and presents a comparison of the performance of the ring-based implementation to other implementations of the Pair-HMM algorithm; Section 6 concludes the paper.

2. RELATED WORK

There are many illustrative examples of the speed optimizations offered by FPGA accelerators for bioinformatics algorithms. FPGA acceleration of Error Correction in NGS reads has been achieved in [12]. The work gets a significant 35× speed-up compared to the software implementation of its base error correction algorithm presented in [5].

Dynamic Programming algorithms have been explored and implemented on varies types of hardware platforms, such as GPU, FPGA and ASICs; For the FPGA platform, most of the work use systolic arrays structures, e.g. the work in [20]. An FPGA accelerated version of the Smith-Waterman algorithm that identifies the best local alignment between two DNA sequences is presented by Isaac TS Li et al. [11]. The work achieved a 160× acceleration compared to the baseline software version. In [21], ring-based structure is proposed to accelerate the dynamic time warping algorithm, which is a type of dynamic programming algorithm whose propagation operator is `min` operator. In that problem, the data could be processed in the fixed point number domain. Ring-based structure is also adopted to accelerate dynamic programming algorithm in this work. However, the dynamic programming algorithm this work accelerate, Pair-HMM, is more complicated. Pair-HMM's propagation operator is a complicated combination of floating-point additions and multiplications, and there are three matrices involved at the same time. This is the reason why Pair-HMM is hard to accelerate using FPGA. There are some other work accelerates dynamic programming algorithm with novel circuit design. In [22], the accumulated score/penalty in dynamic programming problem is represented by the latency of a path in combinational circuit, which corresponding to the dynamic programming search path. This novel design achieve significant speedup compared to the computation based methods. However, this latency based design could only be applied to the those dynamic programming problems whose propagation operator is `max` or `min` operator.

An FPGA implementation of the Pair-HMM stage of HaplotypeCaller on Convey computers is reported in [19]. It achieves 13× speedup compared to the Java implementation of the Pair-HMM algorithm on CPU. The Convey machine contains four high-end FPGAs. In our work, we target a single FPGA chip. However, we leverage most of the optimization techniques feasible through the ring-based RTL-design modifications and achieve higher performance and flexibility of the Pair-HMM's forward algorithm. We achieve a faster and efficient implementation of the Pair-HMM's forward algorithm that can be used in similar flows that utilize it for other applications.

A recent work from Altera [23] accelerates the Pair-HMM algorithm with Altera OpenCL SDK and FPGAs. Their work achieves significant speedup compared to software. The hardware structure in this work is the systolic array. Processing elements are placed in a grid structure. Grid structures are very common in FPGA acceleration designs. However, using the grid structure to process dynamic programming matrices introduces additional overhead of having to store intermediate results back to memory in every step of computation. Our PE ring implementation eliminates this overhead. In the PE ring, the output of one PE is delivered to the neighbor PE and consumed immediately. The whole PE ring produces at most one intermediate result that needs to be stored every cycle. Besides, the PE ring is very amenable to trade offs through restructuring which can reduce the number of idle PEs when boundary conditions (starting or ending of processing a single sequence pair) happen. This can be thought of as a trade-off between Instruction Level Parallelism (ILP) and data-parallelism. In the case of using a single PE ring, the execution goes over one set of dynamic-programming matrices with identical dimensions, while when using multiple smaller rings, we concentrate in parallel on many, possibly differently sized, sets of dynamic programming matrices which may not be amenable to simultaneous processing by a single PE ring. We will discuss more about such techniques later.4.3.

3. FORWARD ALGORITHM

When used as a generative model, the Pair-HMM maybe thought of as emitting a pair of aligned sequences X and Y. To model an alignment based on edit-distance, as opposed to Hamming distance, the model has match, insert and delete states. The Pair-HMM allows us to draw inferences about the alignment quality of a pair of sequences under the assumption that the sequence pair was emitted by itself [Figure 2]. In the HaplotypeCaller, the overall alignment quality between a candidate haplotype and an input read is computed using the Pair-HMM.

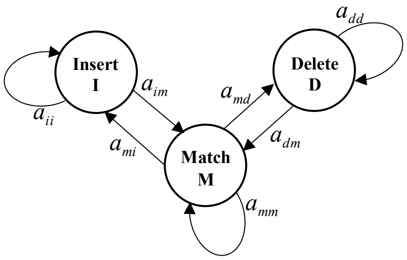

Figure 2: Insert, delete, and match states of Pair-HMM.

Table 2: An example of hidden-state sequence generation using Pair-HMM.[16]

Seq X:	A	G	G	T	A	-
Seq Y:	-	-	G	T	A	A
Hidden state Sequence:	I	I	M	M	M	D

Figure 2 shows the state transitions in a Pair-HMM model. The hidden states are represented as M, D and I. When in state M, the pair-HMM emits symbols from both sequences, implying that the symbols may align to each other. When in state I, it emits one symbol from sequence X, and a blank symbol - meaning no symbol from sequence Y, indicating an insertion in sequence X. Similarly D state represents a deletion in sequence X (or an insertion in sequence Y). The edge-weights between the states represent transition probabilities. Table 2 illustrates how a particular alignment between two sequences may be represented using a Pair-HMM state sequence. The probability of the particular alignment is the product of the corresponding state transition probabilities. To find the overall alignment probability, we need to find the sum of the probabilities of all such alignments between the two sequences. However, if we follow a brute-force approach to evaluate each possible sequence alignment, it can be computationally expensive as there can be a large number of possible alignments between two sequences. Instead, the forward algorithm is used to efficiently compute the overall alignment probability.

The forward algorithm is essentially a dynamic programming approach, which uses three matrices: f^M, f^I and f^D. i and j correspond to the position indices in the sequences X and Y. $f^k(i,j)$ is the *forward variable* that represents the combined probability of all alignments up to positions (i,j) that end in state k. The forward algorithm may be summarized as:

Initialization:

$$\begin{cases} f^M(0,0) = 1 \\ f^X(0,0) = f^D(0,0) = 0 \\ f^M(i,0) = f^I(i,0) = 0 \\ f^M(0,j) = f^D(0,j) = 0 \end{cases} \tag{1}$$

Recursion:

$$\begin{cases} f^M(i,j) = Prior \cdot (a_{mm} f^M(i-1,j-1) \\ \qquad + a_{im} f^I(i-1,j-1) + a_{dm} f^D(i-1,j-1)) \\ f^I(i,j) = a_{mi} f^M(i-1,j) + a_{ii} f^I(i-1,j) \\ f^D(i,j) = a_{md} f^M(i,j-1) + a_{dd} f^D(i,j-1) \end{cases} \tag{2}$$

Termination:

$$Result = f^M(N_h,N_r) + f^I(N_h,N_r) + f^D(N_h,N_r) \tag{3}$$

where N_h and N_r are the lengths of the haplotype (seq. X) and input read (seq. Y) respectively.

Many quantities in the forward algorithm recursion need elaboration. These quantities are evaluated from additional information available in the input dataset pertaining to the quality of a read sequence and various characteristics related to its alignment at each position in the sequence. Specifically, four different quality score values are used in GATK's formulation of the Pair-HMM, which we base our implementation on. Three of them assign penalties to gaps in the alignment, namely the insertion gap open penalty (or base insertion quality Q_i), the deletion gap open penalty (or base deletion quality Q_d) and the gap continuation penalty (Q_g). In addition, there is also data that indicates the level of confidence in the correctness of each symbol in each read sequence, which we represent as Q_{base}. In the forward algorithm recursion presented here, a_{ij} represents a transition probability from state i to state j. For example a_{mm} represents a transition from the match state to the match state. The *Prior* value represents the probability of emitting an aligned pair of symbols. The emission and transition probabilities are computed in the Pair-HMM implementation for the HaplotypeCaller as follows [24]:

$$Prior = \begin{cases} 1 - Q_{base}; & \text{if the bases match} \\ Q_{base}; & \text{if the bases don't match} \end{cases} \tag{4}$$

$$\begin{aligned} a_{mm} &= 1 - (Q_i + Q_d) && \text{– match continuation} \\ a_{im} &= 1 - Q_g && \text{– insertion to match} \\ a_{dm} &= 1 - Q_g && \text{– deletion to match} \\ a_{mi} &= Q_i && \text{– match to insertion} \\ a_{ii} &= Q_g && \text{– insertion continuation} \\ a_{md} &= Q_d && \text{– match to deletion} \\ a_{dd} &= Q_g && \text{– deletion continuation} \end{aligned} \tag{5}$$

This set of prior probabilities and transition probabilities need to be computed for each input read, as they differ for each position in the read.

4. DESIGN AND IMPLEMENTATION

The flow for Pair-HMM computations in GATK's HaplotypeCaller is as follows: The Pair-HMM input datasets are parsed to get the read and haplotype bases, read base qualities and other gap penalty scores. The prior and transition probability matrices for each read-haplotype pair are computed using these quality scores. The input matrices are fed to the ring-based forward algorithm computation block on the FPGA to get the overall alignment likelihood. To parse and pre-process the read-sequences and the haplotypes, and obtain the various quality scores in the right form, we use the reference software implementation of the Pair-HMM stage of the HaplotypeCaller[19]. The original order of computations of the forward algorithm's Dynamic Programming(DP) matrix is row-wise (or column-wise). These computations cannot be parallelized as they are due to the data dependencies between the successive computations according to equation 2. Modifying the array access patterns will help us leverage the parallel nature of the computations and design a Processing Element (PE) ring structure to maximize performance. Figure 3 shows the data dependencies within the forward algorithm matrix and the computing order in our implementation. PEs are placed along the diagonal and all diagonal values are computed in parallel and stored in internal registers. These register values are used to calculate the values of the next diagonal of the matrix. The forward algorithm involves computations for 3 such DP matrices f^M, f^D and f^I.

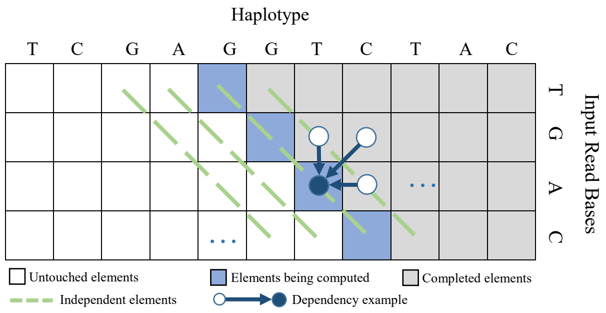

Figure 3: Diagonal matrix access pattern: computations along the diagonal can be parallelized as there are no dependencies among them

The forward algorithm essentially requires a series of arithmetic operations on probability values. Software implementations can comfortably work with floating-point numbers

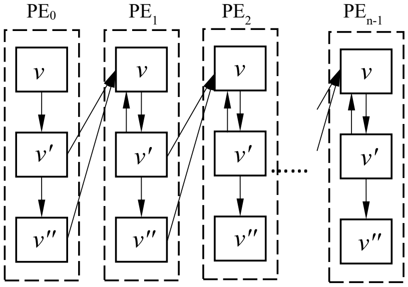

Figure 4: The internal shift registers and their data dependencies among PEs

to represent data and execute computations. For FPGA-based acceleration, it is common practice to normalize all the floating-point numbers to be within a range of fixed point numbers, and use the FPGA to process those fixed point numbers. However, based on our experiments, the values that turn up in the forward algorithm's computations cover a large range of values, tens of orders of magnitude larger than what fixed point numbers can represent. Working with the fixed point domain will lead to overflow or underflow issues for this algorithm, or a very large loss of precision. Thus, our design needs to use floating-point numbers and computations to obtain the requisite accuracy and correctness of computation.

Generally, the hardware design for a dynamic programming algorithm consists of a series of PEs, similar to what was described in Figure 3. In our design, a fixed number of processing elements are arranged in a ring-based structure. Ring-based organization has been proposed and used to accelerate other dynamic programming algorithms[21]. However, given that the forward algorithm requires two different types of critical operations (both addition and multiplication) and given the fact that we need to use floating point operations for accurate execution of the algorithm, the forward algorithm poses a new set of challenges to tackle. According to equations 2, there are 7 multiply operations and 4 add operations for Pair-HMM calculation in each PE. The critical path consists of 2 adders and 2 multipliers. All the operations are implemented using Altera's floating-point IPs, and are fully pipelined. However this adds many cycles of latency; floating-point multiplication requires 5 cycles while floating-point addition requires 7 cycles (at an operating frequency of 200MHz). Thus, the overall latency of the critical path is 24 cycles (on Stratix V). Without careful optimizations, the initiation interval of each PE will be too large to provide useful acceleration. We will discuss the techniques we use to improve the performance of these arithmetic operations in section 4.3.

4.1 PE Array

Figure 6 shows a simplified organization of the PE array, the core component of the PE ring structure. The figure also illustrates how various PEs in the design process the DP matrix elements. Our design consists of n basic PEs, where n is the optimal number of read bases to be processed. As shown in Figure 6, during any step of processing, the processing elements (PEs) are "placed" (processing) along a diagonal of the matrix. One may notice that there are no dependencies along such a diagonal (Figure 3), and all the PEs can compute at the same time, given the results of computation of the previous diagonal. After each computation, the ring moves along the horizontal direction of the matrix to place itself on the next diagonal parallel to the current one. Though there are no dependencies among the PEs within a diagonal, there are dependencies among the neighboring PEs across consecutive diagonals. There are several data buses between the neighboring PEs to share intermediate results to satisfy these dependencies.

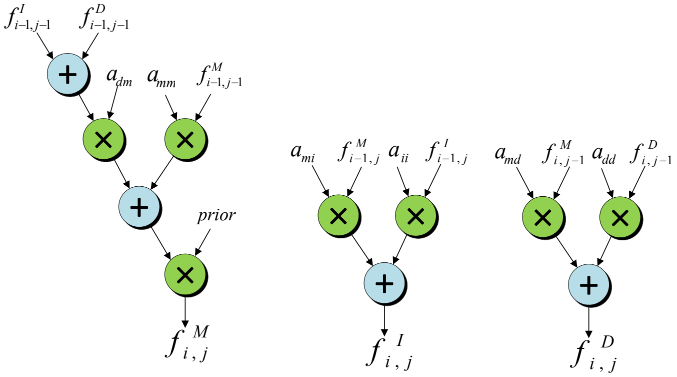

Figure 5: Arithmetic operations inside PEs

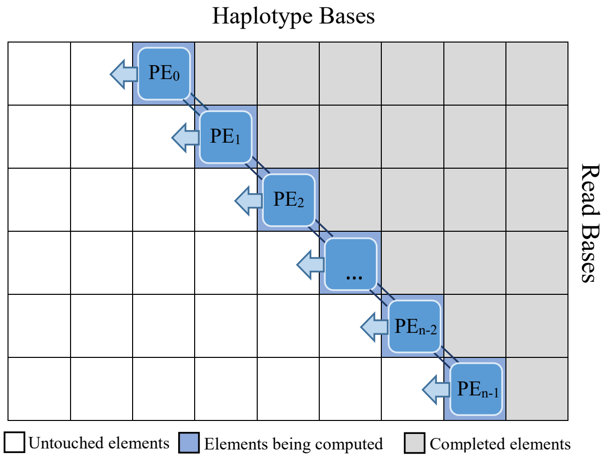

Figure 6: PE array processing the corresponding matrix elements

Let $v(PE_i)$ be the temporary calculation result produced by the i-th PE in the current clock cycle, and $v'(PE_i)$ be the result produced by the i-th PE at the last clock cycle, and $v''(PE_i)$ be the result produced by the i-th PE at the clock cycle before the last clock cycle. Then, based on the data dependency relationships among the matrix elements of the forward algorithm, we get the following formula.

$$v(PE_i) = f(v'(PE_i), v'(PE_{i-1}), v''(PE_{i-1})) \qquad (6)$$

where f is a function that summarizes the forward algorithm recursion (equation (2)). In our design, $v(PE_i)$, $v'(PE_i)$, and $v''(PE_i)$ are stored in three sets of registers. The value of $v'(PE_i)$ and $v''(PE_i)$ can be stored for computations through shift registers. Figure 4 shows the internal registers in the PEs and the data dependencies among those PEs.

Moreover, equation (6) is a general equation that can be used to describe many dependency patterns. Thus, a similar hardware structure can be used to accelerate other dynamic programming algorithms with dependencies that look like equation 6.

4.2 PE Ring

The PE array structure can be extended to a PE ring-based organization by connecting the first PE and the last PE with an internal buffer. The ring-based structure provides extra flexibility and scalability for the design. Figure 7 demonstrates how we use the PE ring structure to calculate the values of the forward algorithm matrix.

With the PE ring structure, the first n rows of the forward-algorithm matrix are calculated first, where n is the number of instantiated PEs (different from the input read length and the haplotype length). After the first PE reaches the last element in the first row, it can continue to process the $(n+1)^{th}$ row in the matrix. In the PE ring structure, the first PE, PE_0, and the last PE, PE_{n-1}, are connected to an internal data buffer. This internal buffer is used to store the temporary results produced by the last PE, PE_{n-1}, so that the first PE can use this data when it processes the new row in the matrix. This way, the design can handle computations for different sizes of the dynamic programming matrix irrespective of the number of PEs instantiated in the design.

In our PE ring implementation, only one of the PEs, the PE that computes the first element of the first row of the forward algorithm matrix, (labeled the "first PE") takes in the haplotype bases and quality scores, and the inputs to all the other PEs come from their neighboring PEs. This saves a lot of data transfer among a large number of PEs, and data storage.

PE ring has several advantages over the other organizations, e.g. the systolic array and the PE array. First of all, with a ring-based structure, we are able to process matrices with more rows than the number of instantiated PEs. This cannot be done with a PE array. The second advantage of PE ring is that there are at most two intermediate results produced per PE which are consumed within two cycles by itself or by neighboring PEs (equation 6). This reduces a lot of data transfer and data storage overhead in some other organizations, e.g. systolic arrays. In the case that a $m \times n$ 2-dimensional systolic array is used to compute the dynamic programming algorithm, after each step of calculation, n or m of intermediate data items are produced and need to be stored in the local on-chip buffer or external off-chip memory until the entire matrix computation is complete.

4.3 Optimizations

We adopt several techniques to further improve the hardware described in Sections 4.1 and 4.2.

4.3.1 Shorten critical paths in arithmetic operations

Figure 5 shows the arithmetic operations inside each PE. The naive implementation of Figure 5 has 24 cycles of latency (based on Altera's Stratix V floating-point IPs), which is a big drawback from the point of view of overall latency. If we analyze the critical path in Figure 5 carefully, we will find that the operands of the first adder and first 2 multipliers in the critical path ($f^I_{i-1,j-1}$, $f^D_{i-1,j-1}$, and $f^M_{i-1,j-1}$) have been ready 2 rounds earlier than the current round.

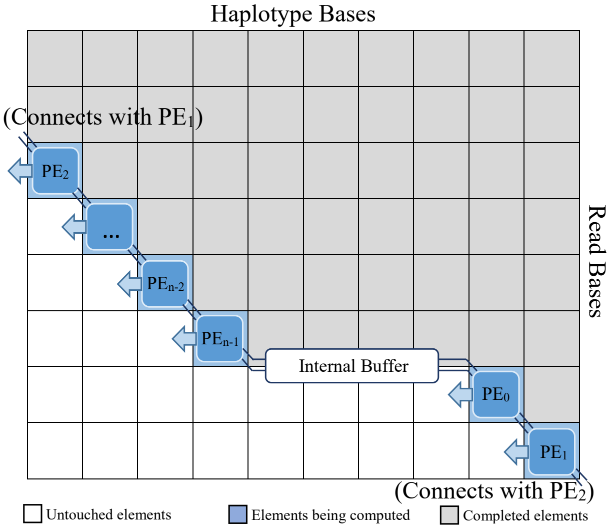

Figure 7: PE ring processing the corresponding matrix elements.

This means that we would be able to start the computation of the critical path earlier and re-distribute the operations among PEs. Based on this idea, we move the first adder and the first two multipliers to the previous PE, and start computation as soon as the operands are ready. After adopting this scheme, the arithmetic operations within each PE are shown in Figure 8. Intermediate computation results $t^a_{i,j}$ and $t^b_{i,j}$ are sent to the next PE in the PE ring. Note that, compared to Figure 5, the operands' indices of first adder and first two multipliers (f^M, f^I, and f^D) have changed, because they are used to compute t^a and t^b in advance.

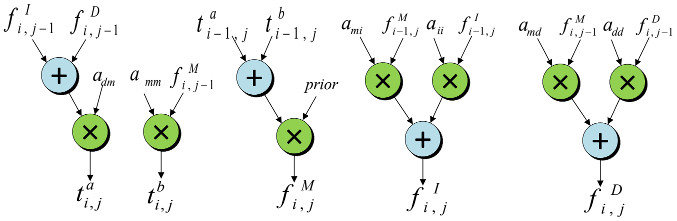

Figure 8: Optimized arithmetic operations inside PEs

With this optimization, the critical path of arithmetic operations is shortened to have only one adder and one multiplier with 12 cycles of latency (on Stratix V), while the overall number of adders and multipliers remains the same.

4.3.2 Pipelining and resource sharing

Note that all the adders and multipliers in the PEs are fully pipelined and the critical path has the latency of 12 cycles (on Stratix V). During this 12-cycle period, we could initiate the computation of other matrices corresponding to another read-haplotype pair. Since the computations of DP matrices are independent from each other, the PEs could compute 12 matrices at the same time.

Figure 9 shows an example of computing multiple matrices at the same time with full pipelining. In the different

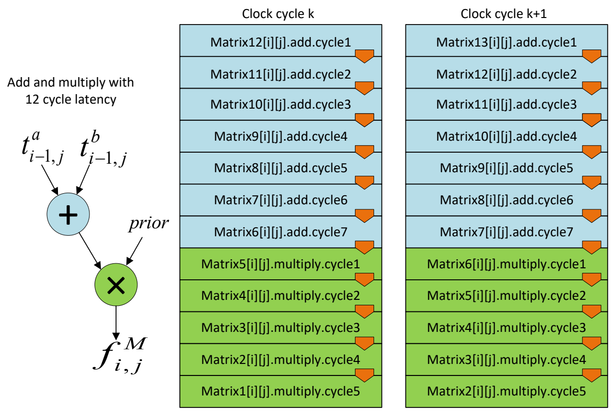
Clock cycle k
Clock cycle k+1
Add and multiply with 12 cycle latency
$t^a_{i-1,j}$
$t^b_{i-1,j}$
+
prior
×
$f^M_{i,j}$
Matrix12[i][j].add.cycle1
Matrix11[i][j].add.cycle2
Matrix10[i][j].add.cycle3
Matrix9[i][j].add.cycle4
Matrix8[i][j].add.cycle5
Matrix7[i][j].add.cycle6
Matrix6[i][j].add.cycle7
Matrix5[i][j].multiply.cycle1
Matrix4[i][j].multiply.cycle2
Matrix3[i][j].multiply.cycle3
Matrix2[i][j].multiply.cycle4
Matrix1[i][j].multiply.cycle5
Matrix13[i][j].add.cycle1
Matrix12[i][j].add.cycle2
Matrix11[i][j].add.cycle3
Matrix10[i][j].add.cycle4
Matrix9[i][j].add.cycle5
Matrix8[i][j].add.cycle6
Matrix7[i][j].add.cycle7
Matrix6[i][j].multiply.cycle1
Matrix5[i][j].multiply.cycle2
Matrix4[i][j].multiply.cycle3
Matrix3[i][j].multiply.cycle4
Matrix2[i][j].multiply.cycle5

Figure 9: Example of pipelining inside i^{th} PE

pipeline stages of the arithmetic operators, operands from different matrices are used to do the computation.

4.3.3 Tuning PE ring size and number of PE rings

Another property that can be used to tune the design is the size of the PE rings, i.e., the number of PEs in a single PE ring. If the number of PEs in the PE ring is small, we can place a larger number of PE rings inside the FPGA chip.

Having smaller, but many PE rings could have some potential benefits. First of all, with multiple PE rings, multiple matrices can be processed at the same time. It is to be noted that this parallelism is different from that described in section 4.3.2. The parallelism utilized with multiple PE rings is more coarse-grained. Second, a smaller PE ring will potentially have lesser number of idle PEs during the first and the last few cycles of processing a matrix.

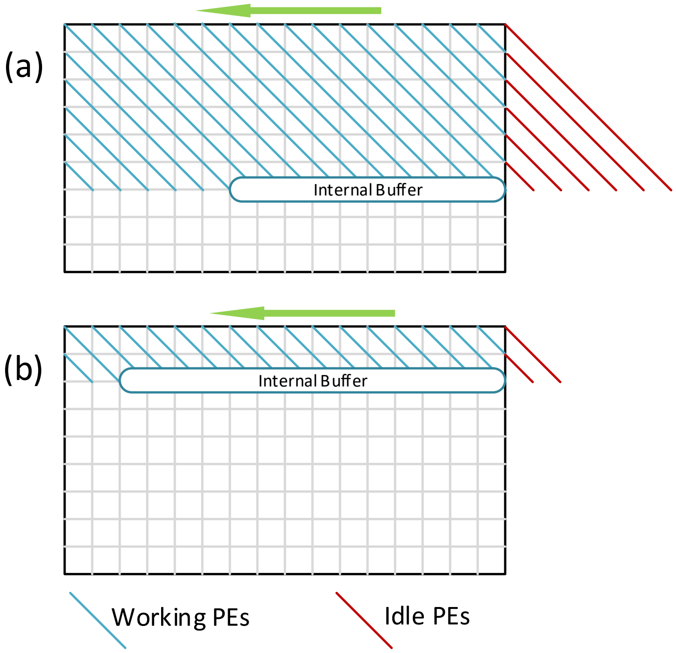

Figure 10: Idle PEs at the beginning of processing and depth of internal buffer when using (a) long PE ring; (b) shorter PE ring

Note that there are idle PEs for a small number of cycles at the beginning of matrix computations or at the last few rounds (when the number of remaining rows is smaller than the number of PEs in the ring). This also means that there are no idle PEs when processing the middle rows of the DP matrix since the PEs that finish the computation of a row will move to uncomputed rows below.

Figure 10 illustrates the idle PEs when the PE ring is processing the DP matrix during the first few cycles. The parallel diagonal lines represent the positions of PE ring at consecutive cycles. The PE ring starts from the top right corner, and moves to the left. Comparing (a) and (b) in Figure 10, we can see that if the PE ring is shorter, there will be less idle PEs during processing. Considering the fact that while using shorter PE rings, we can place more PE rings on the FPGA; we can also compare the idle $\#PE \times \#cycle$ product for the whole FPGA design. Figure 11 shows one intuitive way to do the comparison. In the figure, the area of triangles represents the idle $\#PE \times \#cycle$ product. Assuming that the maximum total number of PEs that can be placed into an FPGA chip is a constant, i.e. if $\#PE \times \#ring$ is a constant, then we can see that the idle $\#PE \times \#cycle$ product of a single longer PE ring is a single big triangle, while the sum of idle $\#PE \times \#cycle$ product of the shorter PE rings are several smaller triangles. Comparing the areas of two set of triangles, we could see that the configuration of shorter PE rings has less idle PEs in general.

We could also count the number of idle PEs directly. Let M be the total number of PEs that can be placed in the FPGA, i.e. the length of a PE ring when we try to deploy a single large PE ring on thee FPGA. Let $M = kN$, where N is the length of smaller PE rings, and there are k PE rings. Then, for the single PE ring case, the number of idle PEs are

$$\sum_{i=1}^{M-1} i = \frac{1}{2}M(M-1). \tag{7}$$

For the smaller PE ring case, the number of idle PEs are

$$k\sum_{i=1}^{N-1} i = \frac{k}{2}N(N-1) = \frac{1}{2}M(N-1). \tag{8}$$

Thus a design with multiple smaller PE rings only has $\frac{N-1}{M-1}$ of the idle PEs of the design with a single long PE ring.

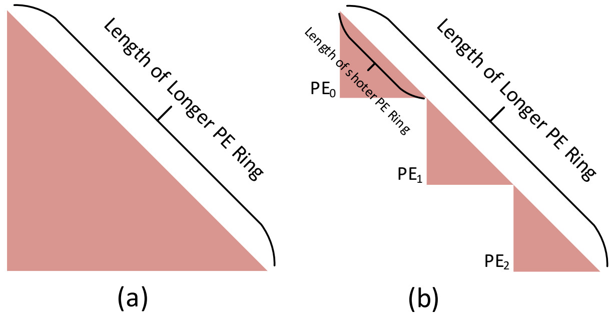

Figure 11: Comparing #idle_PEs × #idle_cycles when using (a) one single longer PE ring; (b) multiple shorter PE rings

Then, consider idle PEs while processing the last few rows of the DP matrix. Figure 12 illustrates this case. This case of idle PEs happens when the number of matrix's rows could not be divided exactly by the number of PEs. This is common because the number of matrix's rows, which is the read sequence's length could be an arbitrary positive integer. If we assume the distribution of read sequence's length is uniform, then any number of idle PEs could happen with the same probability, giving a mean value of half of the PE ring length. This means the number of idle PEs is proportional to the length of PE rings. If the total number of PEs is a constant, then the total numbers of idle PEs for different configurations are similar. Based on this analysis, using shorter PE rings would not reduce the number of idle PEs during the computation of the final rows of the DP matrix. Figure 12(b) shows only the processing steps of one of the n short PE rings.

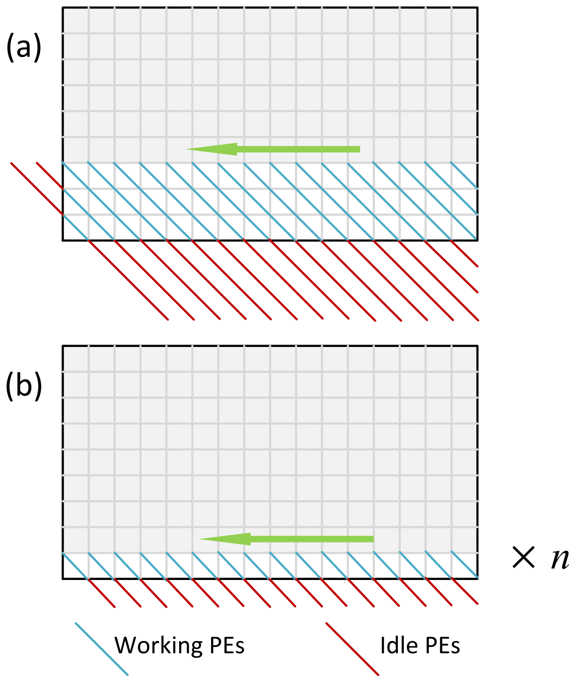

Figure 12: Idle PEs at the last stage of processing when using (a) one single long PE ring; (b) n short PE rings (showing only one of n PE rings in the figure)

Even though using shorter PE rings could reduce the number of idle PEs at the beginning of processing, using shorter PE rings could also have some disadvantages. While using a shorter PE ring, the internal buffer needs to be larger. This is because there will be more intermediate results produced before the next iteration on columns starts. That will significantly increases memory block utilization. This point is also illustrated in Figure 10. Besides, too many small PE rings will lead to potential memory port contention, because each PE ring will need to fetch data in every clock cycle.

4.3.4 *Floating-point operator implementation*

All the floating-point addition and multiplication operators are built with Altera's floating-point IPs. Those arithmetic IPs could be instantiated and mapped to either DSP blocks or logic elements in FPGA. Both two mappings give operators with the exact same functionality. However, the amount of available logic resource and DSP resource could be different on a target FPGA, thus the ratio of IPs mapped to logic elements and that mapped to the DSPs should be carefully tuned to achieve maximum resource utilization in the FPGA chip. In our implementation, we calculate the maximum number of operators that can be implemented using logic elements and with DSPs. Based on this calculation, we figure out the best ratio of number of IPs mapped to logic and of those mapped to DSP. In our final design for Stratix V FPGA, we utilize 83% of the logic elements and 75% of DSP blocks. Generally, when considering the mapping of IPs, given a fixed target frequency, the latency in terms of cycles could be less important than resource utilization. This is because when all the operators are fully pipelined, we may be able to almost completely hide the effects of latency by feeding in new data to occupy the different pipeline stages of the operators, as shown in Figure 9.

This type of tuning of the IP implementation becomes even more important if there are multiple mapping choices, multiple latency options and multiple frequency choices. In our case, there aren't many options, thus manual tuning works good enough. If the number of options grows, this could become an interesting design space exploration problem.

5. EXPERIMENTAL RESULTS

5.1 Test Data and Target FPGAs

The test data comes from a Whole Genome Sequence (WGS) dataset available at [24] that represents Pair-HMM inputs generated by HaplotypeCaller from GATK version 2.7. The benchmark consists of individual datasets each having different haplotype sizes and input read lengths. Each dataset contains testcases consisting of read and haplotype pairs of lengths varying from 10 to 302 bases.

To fully analyze the performance and resource utilization of our implementation, we synthesize our design targeting two FPGAs, Altera's Stratix V FPGA (5SGXEA7N2F45C2), which is the FPGA on Terasic's DE5-Net experiment board, and the new Arria 10 FPGA (10AX115H1F34E1SG). Altera's Stratix V is built with 28nm process technology, and it is one of Altera's high-end FPGAs. Altera's new Arria 10 FPGA uses 20nm process technology, and it has more logic and DSP resources. Besides, the Arria 10 FPGA has hard floating-point elements, which makes floating-point computations more efficient. We collect the performance data of our design by running simulations of our design.

5.2 Performance

5.2.1 *Compared with Other Implementations*

We compare our performance data with that of other representative implementations [19], including CPU, GPU, multi-cores, and FPGAs. The comparison shows that our implementation is faster than other reported implementations.

Table 3 compares the performance of our implementation versus a few others. The performance data for CPU and GPU platforms are reported in [19]. The dataset used is the "`10s`" dataset available along with the original Java implementation in GATK. We present our results with this

Table 3: Performance comparison across various implementations

Platform	Runtime(ms)	Speedup
Java on CPU	10800	1×
C++ Baseline	1267	9×
Intel Xeon AVX Single Core	138	78×
NVidia K40 GPU	70	154×
Intel Xeon 24 Cores	15	720×
Altera OpenCL (Stratix V)	8.3	1301×
PE Ring (Stratix V)	**5.3**	**2038×**
Altera OpenCL (Arria 10)	2.8	3857×
PE Ring (Arria 10)	**2.6**	**4154×**

dataset because this allows us to compare to other implementations. We also run the experiments using larger datasets, and significant speedup is also achieved for larger datasets. The execution time on other datasets is listed in Figure 13.

Our performance data in the table is based on 8 PEs/ring × 8 rings configuration on the Stratix V FPGA, and 8 PEs/ring × 16 rings configuration on the Arria 10 FPGA. Our performance numbers are based on the overall FPGA frequency of 200MHz. Our FPGA synthesis targets have the same number of logic elements and DSP blocks as that of Altera's OpenCL implementation [23].

Prior to our work, the state-of-the-art implementation has been Altera's OpenCL implementation of the Pair-HMM on FPGA. These performance results come from Altera's whitepaper [23]. As shown in the table, on Stratix V, our PE ring design could achieve 1.56× further speedup compared with Altera's implementation. On Arria 10, our design is 7.7% faster than Altera's implementation. The speedup from our Arria 10 implementation is smaller because our implementation contains 128 PEs while Altera's implementation has 208 PEs. In comparison to other platforms, our proposed PE ring design for Arria 10 could achieve 5.77× speedup over Intel Xeon 24 core AVX implementation, 26.92× speedup over K40 GPU implementation, and more than 4000× speedup over the original Java implementation.

5.2.2 *Impact of PE Ring size*

Based on our synthesis results, Altera Stratix V FPGA can accommodate a single PE ring consisting of 64 PEs or multiple rings of shorter lengths. For example, 8 PE rings each consisting of 8 PEs can be put into the Stratix V FPGA. We synthesize designs with various lengths and numbers of PE rings, and run simulations to get performance data.

Figure 13 shows the normalized execution time on three datasets when the PE ring sizes are varied keeping the total number of PEs constant. The figure legend "$m \times n$" stands for the configuration of m PEs/ring × n rings. As discussed in section 4.3.3, using multiple smaller PE rings could reduce the total number of idle PEs during the processing. Figure 13 supports this observation.

Note that we do not further reduce the size of the PE rings below 8 PEs/ring. There are two reasons. First of all, too many small PE rings will lead to potential memory port contention, because each PE ring needs to read input data every clock cycle. Second, multiple smaller PE rings require more and larger internal buffers, which significantly increases memory block utilization.

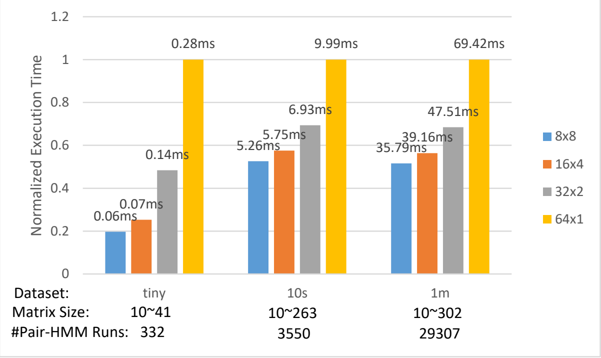

Figure 13: Normalized execution time on three datasets when using different sizes of PE rings (Stratix V). "$m \times n$" stands for the configuration of m PEs/ring × n rings.

5.2.3 *Implementation on Stratix V and Arria 10*

To explore performance trends with more PEs and hard floating-point IP blocks, we synthesize our design targeting Altera's Arria 10 FPGA as well. Arria 10 FPGA has hard floating-point elements that shorten the latency of floating operations. On Arria 10, the latencies of single precision floating-point addition and multiplication are both only 3 cycles when using hard floating-point elements. Besides, the amount of logic resources available on the Arria 10 FPGA is more than what is available on Stratix V. The Arria 10 synthesis target has 427,200 Adaptive Logic Modules (ALM), while the Stratix V target has 234,720 ALMs. Also, Arria 10 has 1,518 DSP blocks, while Stratix V has 256 DSP blocks. Arria 10 FPGA is able to accommodate 128 PEs, and the total latency of arithmetic operations in each PE is only 6 cycles, while the Stratix V FPGA can accommodate 64 PEs, and the total latency of arithmetic operations in each PE is 12 cycles.

Table 4: Synthesis Results for both target FPGAs

FPGA	#PEs	Fmax	Logic	DSP
Stratix V	64	200.16 MHz	83%	75%
Arria 10	128	230.73 MHz	4%	93%

Table 4 shows the maximum number of PEs, maximum frequency, and resource utilization for the two implementations. We observed that our design on Arria 10 is able to achieve a higher frequency. However, the implementation on Arria 10 utilizes almost all the DSP resources while utilizing a very small percentage of logic elements. This is because the synthesis tool (Altera Quartus Prime) maps all the floating-point IPs for Arria 10 to DSP blocks. If those floating-point operators could be mapped to the logic elements, the Arria 10 FPGA will be able to fit in much more PEs, and thus gain further speedup. In the Stratix V case, the resource utilization is more balanced, as a fraction of the operations are also mapped to the logic elements.

6. CONCLUSION

Pair-HMM forward algorithm computation is a major bottleneck in several DNA sequence analysis flows. Essentially, the Pair-HMM's forward algorithm is a complicated floating-point number based dynamic programming algorithm with a high computational complexity. The forward algorithm involves the computation of three matrices while respecting data dependencies among the matrix elements, and it involves a series of floating-point add and multiply operations. In this work, we propose an efficient and flexible ring-based hardware implementation of the Pair-HMM forward algorithm, as well as several optimization techniques to further boost the performance of the PE ring structure. Our ring-based design achieves a significant speed-up of up to 487× compared to the C++ baseline implementation on CPU, and up to 1.56× further speedup compared to the published best hardware implementation. In our design, the ring structure exhibits its unique advantages of flexibility allowing trade-offs between coarse and fine-grained parallelism, and reduced data transfers between the hardware kernel and memory components. We also analyze at depth, the details of how dynamic programming calculations implemented on hardware could benefit from varying PE ring size. The proposed design could be configured as multiple shorter PE rings, which has less idle PEs during computation. This configuration could be adjusted accordingly based on the resources available on the specific FPGA.

Acknowledgment

This work is partially supported by IBM Faculty Award and C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

7. REFERENCES

[1] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[2] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[3] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[4] Heng Li, Jue Ruan, and Richard Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, 2008.

[5] Yun Heo, Xiao-Long Wu, Deming Chen, Jian Ma, and Wen-Mei Hwu. Bless: bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, page btu030, 2014.

[6] Wei-Chun Kao, Andrew H Chan, and Yun S Song. Echo: a reference-free short-read error correction algorithm. *Genome research*, 21(7):1181–1192, 2011.

[7] Lucian Ilie, Farideh Fazayeli, and Silvana Ilie. Hitec: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302, 2011.

[8] Qingguo Wang, Peilin Jia, Fei Li, Haiquan Chen, Hongbin Ji, Donald Hucks, Kimberly Brown Dahlman, William Pao, and Zhongming Zhao. Detecting somatic point mutations in cancer genome sequencing data: a comparison of mutation callers. *Genome medicine*, 5(10):1, 2013.

[9] Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC bioinformatics*, 8(1):474, 2007.

[10] Chi-Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, et al. Soap3: ultra-fast gpu-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012.

[11] Isaac TS Li, Warren Shum, and Kevin Truong. 160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (fpga). *BMC bioinformatics*, 8(1):1, 2007.

[12] Anand Ramachandran, Yun Heo, Wen-mei Hwu, Jian Ma, and Deming Chen. Fpga accelerated dna error correction. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1371–1376. EDA Consortium, 2015.

[13] Broad Institute. Haplotypecaller overview. https://www.broadinstitute.org/gatk/guide/article?id=4148.

[14] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[15] Richard Durbin, Sean R Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[16] Byung-Jun Yoon. Hidden markov models and their applications in biological sequence analysis. *Current genomics*, 10(6):402–415, 2009.

[17] Chuong B Do, Mahathi SP Mahabhashyam, Michael Brudno, and Serafim Batzoglou. Probcons: Probabilistic consistency-based multiple sequence alignment. *Genome research*, 15(2):330–340, 2005.

[18] William H Majoros, Mihaela Pertea, and Steven L Salzberg. Efficient implementation of a generalized pair hidden markov model for comparative gene finding. *Bioinformatics*, 21(9):1782–1788, 2005.

[19] Broad Institute. Accelerating variant calling, 2013.

[20] Sean O Settle. High-performance dynamic programming on fpgas with opencl. In *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*, pages 1–6, 2013.

[21] Zilong Wang, Sitao Huang, Lanjun Wang, Hao Li, Yu Wang, and Huazhong Yang. Accelerating subsequence similarity search based on dynamic time warping distance with fpga. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 53–62. ACM, 2013.

[22] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. Race logic: A hardware acceleration for dynamic programming algorithms. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 517–528. IEEE, 2014.

[23] Altera. Accelerating genomics research with opencl and fpgas, 2016.

[24] Pair-hmm test data. https://github.com/MauricioCarneiro/PairHMM/tree/master/test_data.