Acceleration of the Pair-HMM forward algorithm on FPGA with cloud integration for GATK

Rick Wertenbroek and Yann Thoma
HEIG-VD School of Business and Engineering Vaud
HES-SO University of Applied Sciences Western Switzerland
CH-1400 Yverdon-les-Bains, Switzerland
Email: rick.wertenbroek@heig-vd.ch, yann.thoma@heig-vd.ch

***Abstract*—The Pair-HMM forward-algorithm is an essential algorithm found in many genomic related analyses. The high number of floating point operations in the algorithm makes it one of the main contributors to the compute time of analysis pipelines. To speed-up computations we propose an FPGA based hardware accelerator for the Amazon AWS F1 Cloud platform. The accelerator is open source and has been tested within the popular Genomic Analysis Toolkit (GATK) pipeline. The accelerator achieved up to 15× speed-up against the software implementation when used in-pipeline. The accelerator has also been tested in the experimental Spark (distributed) version of the GATK HaplotypeCaller tool. An in-depth analysis of the compute time contributions allowed to point out the main bottlenecks for accelerators in the GATK pipeline, resulting in a hybrid CPU-FPGA solution to best exploit both resources.**

I. Introduction

The human genome project was completed in 2003, achieving its goal to cover over 90% of the human genome with more than 99.99% accuracy. This project came at a cost of about $2.7 billion in 1991 dollars [1]. Sequencing has since seen an extreme decrease in cost, decreasing faster than Moore's Law in the last ten years [2] [3]. This decrease in cost has led to the generation of immense amounts of data. Sequencing data in itself is not the end, the real intention is drawing conclusions and answering relevant questions. To this end the data requires to be processed and the associated costs may now exceed the costs of sequencing itself [4]. In order to keep pace with the data generation computing must also scale [5].

To answer ever more complex questions faster requires ever better computational resources. Researchers in bioinformatics have traditionally used computers, clusters or even supercomputers. The trend of complementing these computing resources with graphic processing units (GPUs), Field Programmable Gate Arrays (FPGAs), and other accelerators (e.g., tensor processing units or manycore processors) allowed to decrease processing time and achieve better performance per watt. FPGAs have been shown to be effective candidates to address these factors [6]. This is the case not only in genomics but in many other big data related fields. Microsoft has integrated FPGAs in its data-centers to reduce the power consumption of their search engine Bing, as well as to provide new services in their cloud platform [7]. Amazon, IBM, and Alibaba have been integrating FPGAs in their cloud services as well. FPGAs allowed Edico Genome, now part of Illumina Inc., to achieve the fastest-ever analysis of a thousand genomes [8].

Integration of FPGAs in cloud computing allows researchers to develop new solutions without having to buy exotic hardware and setting up a complex compute cluster. The cloud framework also provides a normalized environment to share solutions with other researchers. The problem with FPGA accelerators in the past was that they were developed for a very specific platform and would be difficult to port to other systems or impossible in cases were the source code is not available. This brings the problem that researchers cannot use or even replicate the results of others unless they have access to the code and acquire specific, often very expensive, hardware. The cloud computing framework alleviates this problem by giving access to the same FPGA platform to anyone and makes it possible to share a design with other researchers either as an encrypted binary or as source code and will eventually lead to better research. In the same way it has been experienced in software development with the open-source movement.

It is with this in mind that we present an open-source FPGA accelerator for the Pair-HMM forward algorithm (FA) with cloud integration on the Amazon EC2 platform for one of the most popular genomic analysis pipelines, GATK [9].

A. Background

One of the main processing pipelines in genomic data analysis is germline variant discovery and analysis. Germline variants are used in disease gene discovery research as well as clinical genetic testing. The three main steps of the variant discovery pipeline are : pre-processing, variant discovery itself, and callset refinement. Pre-processing consists mainly of mapping the raw unmapped sequencing reads to a reference. This task is common in many analysis pipelines and can be done once and serve as a basis for different subsequent analyses. The second step, variant discovery, is the heart of the pipeline and mainly consists of calling the variants and will be detailed below. The final step, callset refinement, is mainly filtering and annotating the variants.

The variant discovery is often the most time consuming task of this pipeline (over 50% of total time) [10]. Especially if we consider the first step done since this is common to other analyses. To discover variants with GATK a tool called the HaplotypeCaller (HC) is used. The HC is capable of calling single nucleotide polymorhpisms (SNPs) and indels simultaneously. It does so by reconstructing active regions, i.e., regions with signs of variations, via local de-novo assembly. This allows the HC to be more accurate than other solutions. The reconstructed regions, assembled haplotypes (haploid genotypes), are then given likelihoods given the sequencing reads data. This is done by using the Pair-HMM forward algorithm. Finally for each potentially variant site, the HC assigns the most likely genotype to each sample [11].

B. Pair-HMM forward algorithm

Given a Pair-HMM model of two sequences, the forward algorithm allows to compute the probability that a given pair of sequences are related by any alignment [12]. This makes it possible to estimate the likelihoods of the assembled potential haplotypes given the sequencing reads. To compute the probabilities of the alignments an affine gap model is used with the finite state machine of Fig. 1 A.

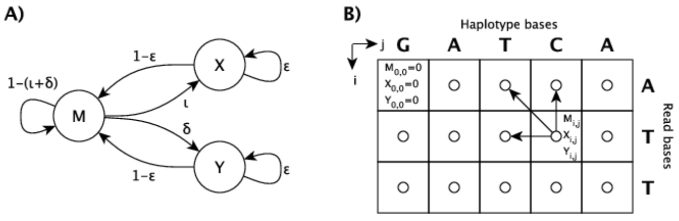

Fig. 1. A) Finite state machine model for affine gap alignment. M : bases match, X : Insertion of a base, Y : deletion of a base. B) Probability matrix for the Pair-HMM FA with dependencies shown.

The probabilities for the finite state machine model for affine gap alignment, $\delta, \iota, \varepsilon$, are given by the read deletion, insertion and gap continuation quality scores (d, i, g) in phred scale on a base per base basis, with the following equations :

$$\delta = 10^{-Q_d/10}, \iota = 10^{-Q_i/10}, \varepsilon = 10^{-Q_g/10} \tag{1}$$

The equations characterizing the Pair-HMM forward algorithm corresponding to the matrix of Fig. 1 B are :

Initial conditions :

$$\begin{cases} M_{-1,j} = 0, & X_{-1,j} = 0, \quad Y_{-1,j} = IC, \\ M_{i,-1} = 0, & X_{j,-1} = 0, \quad Y_{j,-1} = 0 \quad \forall i, j \\ M_{0,0} = 1, & X_{0,0} = 0, \quad Y_{0,0} = 0 \end{cases} \tag{2}$$

Recursion :

$i = 0, ..., |read| - 1, j = 0, ..., |hap| - 1$ except $(i = 0, j = 0)$

$$\begin{cases} M_{i,j} = prior_j \cdot ((1 - (\iota + \delta)) \cdot M_{i-1,j-1} + \\ \qquad (1 - \varepsilon) \cdot (X_{i-1,j-1} + Y_{i-1,j-1})) \\ X_{i,j} = \varepsilon \cdot X_{i-1,j} + \iota \cdot M_{i-1,j} \\ Y_{i,j} = \varepsilon \cdot Y_{i,j-1} + \iota \cdot M_{i,j-1} \end{cases} \tag{3}$$

$$prior_j = \begin{cases} 1 - (10^{-Q_b/10}) & \text{read and hap bases match} \\ (10^{-Q_b/10})/3 & \text{read and hap bases don't match} \end{cases}$$

Termination :

$$L = log_{10}(\sum_{j=0}^{|hap|-1} M_{|read|-1,j} + X_{|read|-1,j}) - log_{10}(IC) \tag{4}$$

Where L is the read likelihood for a given haplotype for all paths ending in the M (match) and I (insertion) states, ignoring all paths that end in deletions. IC is an initial condition, a high constant to prevent underflow, e.g., 2^{2010} in the Java implementation. Q_b is the base quality (estimated base call accuracy) of the given read base in phred scale. The higher L is for a given read-haplotype pair the more likely the two sequences are related.

II. State of the art

The high cost of the forward algorithm, due to the high amount of floating point operations, led researchers to develop accelerated implementations of the algorithm.

A. Software acceleration

The original Java version was compared to a C++ version of the algorithm in [10] and single instruction multiple data (SIMD) versions were experimented with. Collaboration between Intel and the Broad Institute led to the development of the Intel Genomic Kernel Library (GKL) which contains accelerated versions of algorithms used in genomics including the Pair-HMM FA.

1) SIMD implementations: The current SIMD implementations were developed by Intel and are integrated in GATK through the GKL. The GKL is open source and can be found at https://github.com/Intel-HLS/GKL. They provide single and double precision implementations of the algorithm using AVX and AVX-512 SIMD instructions.

2) Multi-threaded implementations: Intel also provided a multi-threaded implementation allowing parallel computation of the forward algorithm. This version is implemented through the OpenMP framework and is also available in the GKL.

3) Distributed Version: The GATK developers have been creating distributed versions of their tools using the Spark framework. This allows to distribute the computing tasks on multiple compute nodes, which can be local cores or a distributed compute cluster. The use of the Map-Reduce paradigm and distributed datasets makes it possible to scale the computation over several nodes. The Spark versions of the tools, e.g., HaplotypeCallerSpark, deliver better performance, but are still in development and don't necessarily produce the same results as the non Spark versions of the tools.

B. Hardware acceleration

Several hardware accelerators have been developed to accelerate the computation of the pair-HMM forward algorithm on FPGA as well as on GPU. The only hardware accelerator that is currently integrated in GATK is the FPGA accelerator by Intel [13]. The FPGA version is not officially supported (marked as "experimental") but can be used with either a Nallatech 385a or Inspur F10A FPGA card [14] (both Intel Arria 10 GX based) . To the best of our knowledge this is the only accelerator publicly available. However, no source code is given. Other FPGA implementations were explored in [15], [16], [17], [18], [19], and [20]. GPU solutions include [21] and [22]. Solutions that provide a common benchmark will serve as references for the results section.

C. Integration discussion

While the software accelerated versions are well integrated in GATK and already provide a significant speed-up, the hardware accelerators are not. Most have only been benchmarked outside of the GATK, or any other, pipeline. This is important because the benchmark will not reflect the real impact of the accelerator in a typical workflow and may mislead users. Albeit the speed-up numbers seem impressive (up to $4000\times$), typical workflows will not necessarily scale as much. Intel reported an overall speed-up of the pipeline of $1.2\times$ when comparing their FPGA solution to the AVX implementation [13]. The fastest GPU solution also achieved an in-pipeline speed-up of $1.2\times$ over the AVX solution [22].

The limiting factor for the speed-up of the FA is how GATK implements the pipeline. Computations for the FA are generated sequentially one region at a time (batch). A batch is comprised of the possible haplotypes and the sequencing reads (amount relative to coverage). The only parallelism possible given this constraint is at the batch level. The small size of a batch makes it hard to fully exploit the accelerator given the overhead costs of transferring data to the accelerator. This is further explored in section V. When software architecture is not created with accelerators in mind the addition of an accelerator can result in loss of performance or mitigated results. This was explored in [23] were FPGA accelerators that could achieve a speed-up of $120\times$ were actually slowing down the pipeline by three orders of magnitude when integrated. Finally once the integration was finely tuned and optimized speed-ups of 2.6x were achieved. This gives us a warning that even if an accelerator is extremely fast it may under-perform in a real software unless integration is done carefully.

III. Contributions

The contributions of this work are the following :
1) An open-source hardware accelerator for the Pair-HMM FA of the GATK HaplotypeCaller tool. 2) Integration within the GATK pipeline without modification to GATK itself. 3) Support for cloud based FPGA as a Service (FaaS) in the Amazon compute cloud. 4) The possibility to configure the parallelism levels of the accelerator with multiple levels of granularity. 5) Support for both the sequential and distributed (Spark) versions of the HaplotypeCaller. 6) "in-pipeline" benchmarks followed by a discussion and ideas for future integration of hardware accelerators in genomic pipelines.

IV. Implementation

The FPGA bitstreams, source code (SystemVerilog), and documentation can be found in the following git repository :
https://github.com/rick-heig/PHMM-F1

A. Hardware accelerator overview

A hierarchic diagram of the accelerator can be seen in Fig. 2. The PC will transfer read and haplotype data to the DDR4 through DMA over PCIe. The PC will also give job instructions to the FPGA over a 32-bit AXI bus mapped over PCIe. The FPGA has access to the DDR4 through a 512-bit AXI bus in order to retrieve the data for the computations and will generate an interrupt to notify the PC once all the computations are done.

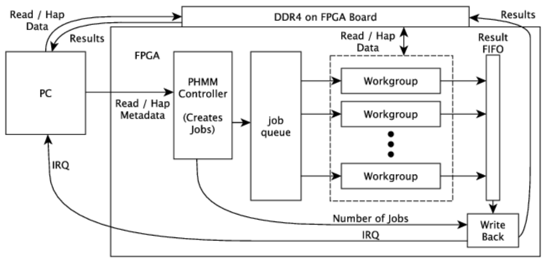

Fig. 2. Overview of the FPGA accelerator.

1) Pair-HMM Controller: For a given batch of pair-HMM computations this module will generate jobs for the workgroups to compute. The jobs consist of meta-data comprised of an ID, the read and haplotype lengths as well as their position in the DDR4 memory, and the initial conditions for the computation. The jobs go in a job queue to be processed by workgroups.

2) Workgroups: The workgroups are the entities that will take jobs (PHMM FA computations) and compute the results. The number of workgroups is generic and can be parametrized at the FPGA image generation giving us a first level of granularity for parallelization.

Workgroups, as can be seen in Fig. 3, consist of several workers and a controller. The controller has the responsibility of taking jobs in the queue and assign them to idle workers. The controller will also get the sequence data (bases and probability values) from the DDR4 and stream it to the assigned worker. Once the worker has all the required data it is started and the controller can assign a job to another worker. The workers are linked to a compute engine for the FA floating point operations (cell updates). When a worker finishes a job the result is written to a "round-robin result propagator", which takes the results from the workers and writes them to the output FIFO. The workgroup has a 512-bit AXI-Full bus to the DDR4 while the data buses from controller to workers are 32-bit AXI-Stream.

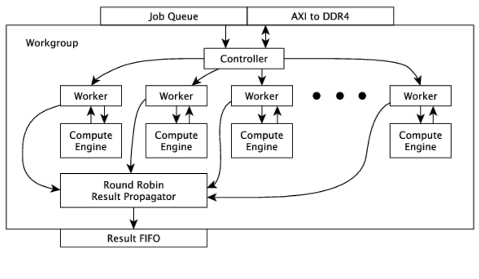

Fig. 3. Workgroup - Controller and workers with their compute engine

The number of workers in a workgroup is also generic, giving us a second level of granularity for parallelization.

3) Workers: The workers compute the result of the FA for a given read-haplotype pair. They do so by computing insertion, deletion, and match probability scores for all cells of the read-haplotype pair matrix. The dependencies restrict the order in which the operations are done. The workers compute the matrix cell by cell following a "zig-zag" pattern in order to fulfill the dependencies as can be seen in Fig. 4.

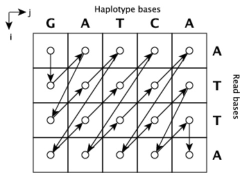

Fig. 4. Computation order of the forward algorithm in worker.

For each cell a compute request is sent down to the compute engine pipeline. In order to realize this, two "crawler" processes were created that travel through the matrix following the computation order shown above. One crawler is responsible to issue compute requests checking that the dependencies are fulfilled (results came back from the compute pipeline). The other crawler is responsible to get and store the results. Each crawler advances to the next cell when it has accomplished its task for their current cell as can be seen in Fig. 5. Once results have fulfilled their dependencies and are no longer needed they are discarded, requiring to keep at most one diagonal of results at any given time.

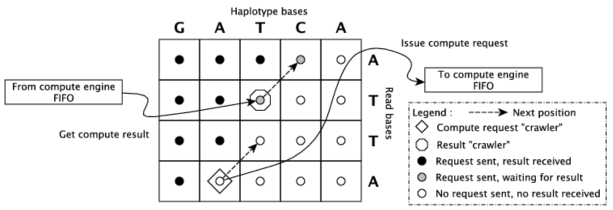

Fig. 5. Worker inner process generating compute requests.

The crawler processes can execute their task every clock cycle if the dependencies are met.

4) Compute Engine: The compute engine is responsible for computing the probabilities of a cell in the forward algorithm, i.e., compute the values $M_{i,j}, X_{i,j}, Y_{i,j}$ in Eq. 3. A compute request to the compute engine is comprised of all the values needed to compute $M_{i,j}, X_{i,j}, Y_{i,j}$.

The diagonal dependency computation (upate of $M_{i,j}$) is split between the cell to the top and the current cell in order to reduce the depth of the compute pipeline, therefore reducing the latency. Splitting the compute pipeline this way has also been done in [18] and [20]. The lower the latency of the pipeline the better the performance as will be explored in the next subsection. The compute engine is roughly equivalent to what is referred to as processing engine (PE) in other implementations, they both compute the $M_{i,j}, X_{i,j}, Y_{i,j}$ in Eq. 3 but may differ in implementation.

5) Compute Engine pipeline against parallel Processing Engines: Several other solutions [15], [17], [18], [20] use parallel arrays of PEs to compute partial diagonals of the matrix. While this solution may seem faster, this is not necessarily the case, and has some other drawbacks as well. Fig. 6 is used to illustrate this.

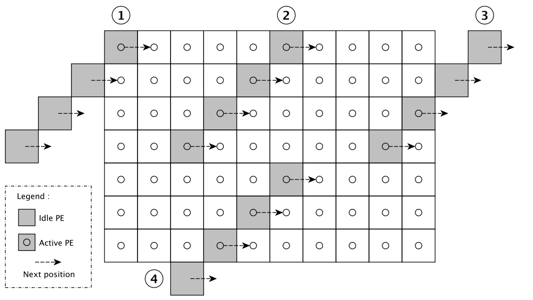

Fig. 6. PE array computation of PHMM FA matrix.

With a parallel array of PEs, partial diagonals of length equal to the number of PEs are computed one after each other. Each cell on this diagonal is computed in parallel. Fig. 6 shows some situations with an array of 4 PEs.

Situation ② is the typical case, 4 cells will be computed and the PE array will advance to the next 4 cells thereafter. While these four computations are done in parallel, they require a number of cycles equal to the pipeline depth (latency) to finish. The next diagonal cannot be sent down the pipeline before the dependencies are met. Therefore, the PE pipelines only get one computation from this matrix and remain empty for the rest of their depth. This problem is usually solved by sharing the PE array between multiple instances of the algorithm, typically as many as the pipeline depth. While this solves the problem of the pipelines being underutilized, it also requires providing data from several instances of the algorithm to the PE array, data consisting of multiple floating point (32-bit) values. This can lead to difficulties routing the architecture inside an FPGA.

Situations ①, ③, ④ show the PE array being underutilized because of the geometry of the matrix. ① cannot be avoided because of the initial dependencies. The first result requires to be computed in order for the two next computations to be started, and so on, until the N^{th} diagonal, where N is the number of PEs in the array. ③ can be avoided with some clever buffering of results as is done in [18]. ④ can be mitigated, as is done in [20], by joining matrices that share a common haplotype, and have the idle PEs overlap over on the next matrix.

The number of cycles required to calculate a whole matrix can be computed with Eq. 5. Where $\#PE$ is the number of PEs and N is the PE pipeline depth (latency).

$$cycles = (|hap| + 2 \cdot (\#PE - 1)) \cdot \lceil |read| / \#PE \rceil \cdot N \quad (5)$$

Eq. 5 shows us that the number of cycles will be divided by the number of PEs, but also that the number of cycles will grow relative to the pipeline's latency. In contrast, using a single pipelined compute engine (a single PE) can actually achieve higher performance while being much simpler to map to an FPGA. The traversal of the matrix is not done by partial diagonals of length #PE but full diagonal by full diagonal as can be seen in Fig. 4. Because there are no dependencies between elements on the same diagonal, computations can be issued each clock cycle. The problem of dependencies for the first diagonals remain, i.e., it is necessary to wait for the first result to come out of the pipeline before issuing the two computations of the second diagonal. However once the diagonal length is greater than the pipeline depth, a computation can be launched every clock cycle. A graph of the pipeline usage over time can be seen in Fig. 7. It shows a pipeline of depth 5 and its usage over time t. ① For the first 5 cycles the pipeline has only one computation going trough it (first diagonal with 1 element). Then when the result comes out, ② the second diagonal can be computed and its two elements can be put into the pipeline one cycle after each other since they both only depend on the first element. The pipeline usage (duty cycle) continues to grow until the size of the diagonal is greater than or equal to the pipeline depth. At this point the pipeline will provide results every clock cycle and therefore subsequent computations can be put in the pipeline every clock cycle. Note that the time where the pipeline is full ●●● is not necessarily a multiple of the pipeline depth. Fig. 7 also shows that the cost due to latency is constant.

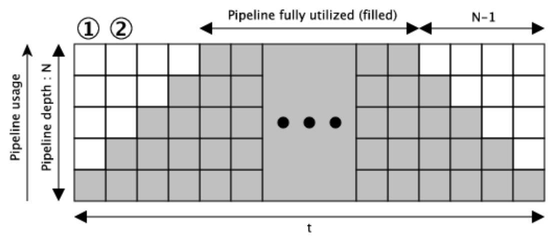

Fig. 7. Compute engine pipeline usage (duty cycle) over time

The total number of cycles required to calculate a matrix is given by Eq. 6. (Assuming that the length of the sequences $|read|$ and $|hap| \geq N$, where N is the depth of the pipeline).

$$cycles = |read| \cdot |hap| + \cancel{2} \cdot \frac{(N-1) \cdot N}{\cancel{2}} \tag{6}$$

The number of cycles is the product of the read and haplotype lengths plus the cycles spent waiting for the pipeline in the $N-1$ first and last diagonals (white cells in Fig. 7).

While the impact of the pipeline depth (latency) grows with the length of the sequences in Eq. 5, it is a constant number of cycles for Eq. 6. This relation remains true even if optimization to reduce the number of idle PEs are in place. This and the fact that the PE array is unused by the current matrix for $N-1$ cycles (cycles in which the array is possibly used by other matrices) are the main differences between the two solutions. Therefore if the latency is bigger than the number of PEs a parallel solution can actually be counterproductive. Having the impact of the latency be a constant relative to the matrix size also makes it less of a problem to use high latency pipelines and therefore may allow for higher frequencies of operation. Note : The pipeline depth is typically a dozen cycles or more (19 in our implementation, but is parameterizable). This is why it was chosen to use a single compute engine per worker.

6) Write-Back: The write-back module reads results from the result FIFO and writes them back in the DDR4 memory. The results are stored in an array of 32-bit single precision floating point values. The position of the results is given by an ID. The ID follows the formula $read_{index} \cdot \#haplotypes + haplotype_{index}$. This value is the same index than where the results are stored in software and therefore the results can be transfered directly by DMA from the DDR4 to the software.

B. Resource utilization

Resource utilization is shown in Table I and shows the utilization for an accelerator with 8 workgroups of 12 workers. The first column shows the total usage including the DDR4, PCIe, and shell logic. The second column shows the accelerator itself. The main limiting resource are block RAMs. The required quantity of BRAMs could be reduced by encoding bases with 4 bits instead of 8 and phred quality scores on 6. Lowering the resource usage would allow more workers but this does not scale linearly because of shared resources such as the DDR4 memory (see result sections).

TABLE I
AMAZON F1 - XILINX XCVU9P - RESOURCE UTILIZATION

Resources	Full FPGA 8w12w	Accelerator 8w12w
LUT	631,232 / 1,181,768 (53.41%)	452,375 (38.25%)
LUTRAM	41,888 / 591,840 (7.08%)	29,946 (5.06%)
FF	939,695 / 2,363,536 (39.76%)	733,782 (31.05%)
BRAM	1,725 / 2,160 (79.86%)	1,526.5 (70.65%)
DSP	2,516 / 6,840 (36.78)	2,513 (36.78%)

C. Software

The FPGA is utilized through a modified version of the Intel GKL. The GATK pipeline is left untouched but requires to be recompiled with the new version of the GKL. GATK was left as-is in order to avoid adding any bugs and introduce unnecessary complexity or dependencies.

1) Precision: As the computations are done using single precision floating point values, it is possible for the values to become too small and the computation lose any meaning. This is detected in software and a double precision computation will be launched for the specific read-haplotype pair that failed. This problem is common to all single precision implementations and is therefore already taken care of in the GKL. Note that if for any reason a computation would fail on the accelerator (e.g., failure to communicate with the card) the software will run the failed computation with the next best software solution available.

2) Calling accuracy: The results of the accelerator were compared to the AVX versions for correctness and will consequently produce the same results (they both implement the same algorithm). Therefore the variants called by the GATK HaplotypeCaller will be the same regardless of the implementation (AVX, OpenMP, or FPGA).

D. Scalability

Multiple accelerators can be instantiated on a single FPGA board to be shared between multiple runs of the GATK pipeline or multiple threads of a Spark run as can be seen in Fig. 8.

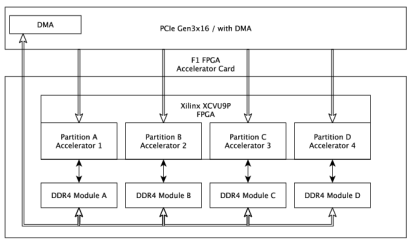

Fig. 8. Multiple accelerators on the same FPGA board.

Sharing accelerators between multiple instances of GATK was accomplished with system wide POSIX named semaphores. The semaphores reflect the number of accelerators available on a system and handle mutual exclusion. Effectively creating a "shared accelerator pool". When a computation of the forward algorithm is launched it will try to acquire an accelerator from the pool, if an accelerator is available it will be used, if there are none, the computation will be done using the next best software method available (e.g., AVX or OpenMP). This makes for a hybrid solution that allows better use of the accelerators. It also allows for integration of FPGA enabled nodes with standard nodes (without an FPGA) in the same Spark framework. This solution allows the system to scale seamlessly in distributed Spark setups composed of heterogeneous compute nodes.

As will be discussed in the results section, it is more interesting to do some computations (small batches) on the CPU due to the overhead of transferring data to the accelerators. A hybrid solution (CPU+FPGA) is also more desirable since it takes advantage of all available resources.

V. Results

A. Benchmarks

1) Datasets: The chosen dataset for the benchmarks comes from the NA12878 sample of the 17 member CEPH pedigree 1463 from the Illumina Platinum Genomes [24]. The sequencing data as well as the validated phased variants are publicly available and therefore make this a perfect benchmarking dataset. The benchmarks were run on the exome sequencing data of the first three chromosomes of NA12878[1].

2) Benchmarks results: Most benchmarks of the existing accelerators rely on a small dataset that was used in [10] when evaluating C++ over Java. The so-called "10s" dataset, because it took ≈10s to run on the Java implementation at that time. Although the dataset has been widely used for benchmarks it does not reflect the size of a typical dataset. Another problem is that every benchmark is done in their own way e.g., in how data is provided to the accelerator. Some benchmarks are also only simulated and not run on real hardware. The problem is that there is no standardized way of running the benchmarks.

[1]ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/working/20101201_cg_NA12878/

This is why the benchmarks presented here are done "in-pipeline" and the reported executions times are those given by the HaplotypeCaller tool in GATK. The tool provides the total execution time as well as three internal measures. The time spent in setup for JNI call (time to call C/C++ from Java) which is extremely small. The time spent in PairHMM computeLogLikelihoods() which is the application of the FA and likelihoods computations, and finally the time spent in the Smith-Waterman algorithm. Using the time spent in the PairHMM section for our benchmarks allows for a standard method of measuring the impact of the accelerator in the GATK pipeline. (The Spark version only gives total time).

Table II reports the time results from running the HaplotypeCaller on the three first chromosomes of NA12878. All runs were done on an Amazon F1 instance. The sequencing reads are from the full exome aligned to NCBI36 using BWA. Instructions for replicating the benchmarks as well as running extra benchmarks are available at :

https://github.com/rick-heig/PHMM-F1

TABLE II
Computation times for Pair-HMM and the HaplotypeCaller

	chr1	**chr2**	**chr3**
Java (reference)	PHMM : 1,252s 1x Total : 31.70m 1x	531.2s 1x 17.31m 1x	314.8s 1x 12.02m 1x
AVX	PHMM : 432.5s 2.3x Total : 18.14m 1.8x	184.9s 2.8x 11.77m 1.5x	111.5s 2.8x 8.70m 1.4x
OpenMP 4t	PHMM : 121.5s 10.3x Total : 13.33m 2.4x	54.02s 9.8x 9.77m 1.8x	33.5s 9.4x 7.45m 1.6x
OpenMP 8t	PHMM : 101.8s 12.3x Total : 13.14m 2.4x	48.63s 10.9x 9.91m 1.8x	31.1s 10.1x 7.75m 1.6x
Spark 4t + OMP 4t	PHMM : Not Available Total : 6.33m 5.0x	N/A 4.35m 4.0x	N/A 3.63m 3.3x
OpenMP 4t FPGA 24w	PHMM : 160.8s 7.8x Total : 13.89m 2.3x	46.33s 11.5x 9.57m 1.8x	23.2s 13.6x 7.31m 1.6x
OpenMP 4t FPGA 96w	PHMM : 132.8s 9.4x Total : 13.30m 2.4x	45.0s 11.8x 9.47m 1.8x	21.0s 15x 7.31m 1.6x
Spark 4t + FPGA 4x24w	PHMM : Not Available Total : 5.87m **5.4x**	N/A 4.08m **4.2x**	N/A 3.26m **3.7x**

B. In-depth analysis

1) Pair-HMM compute time in the HaplotypeCaller: With the original Java implementation of the Pair-HMM FA, its contribution to the total HC compute time was significant. In our benchmarks it took over 65% for chr1, 50% for chr2, and 43% for chr3. With AVX acceleration this was reduced to 39%, 26%, and 24%. With the OpenMP (4 threads) implementation it was further reduced to 15%, 9%, and 7%. With our hybrid FPGA (96w) solution the contributions are 16%, 7%, and 4%. While this is dataset dependent, the compute time contribution of the Pair-HMM FA in the HC tool is now less than 20%, meaning that in order to achieve further speed improvements of the HC tool it may be better to work on the other parts. E.g., local de-novo assembly. The contribution of the Smith-Waterman algorithm in HC however was negligible when it is AVX accelerated, less than 1% on all runs.

2) DMA Overhead: Benchmarks to compute the DMA overhead alone were run to show the cost of transferring data at the batch level by GATK. Since GATK generates a small batch and waits for the results before issuing another batch there is no way to cancel the transfer overhead by grouping all the data in a single transfer without modifying GATK itself. From Fig. 9 we can see that under 5,000,000 cell computations the cost of transferring the data to the accelerator alone is bigger or equal than the cost of computing the result in software. Therefore it is counterproductive to solely rely on the accelerator. The solution taken was to precompute the number of cell computations, which is the sum of the matrix sizes, and use the accelerator only above a threshold.

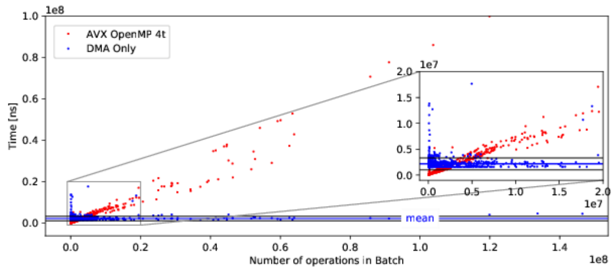

Fig. 9. DMA overhead compared to running jobs with OpenMP 4t

While this overhead is specific to this implementation and the FPGA platform used (Amazon F1), it is probable that other implementations will suffer similarly. This is especially problematic for runs that consist of numerous small batches.

3) Compute time relative to number of operations: Fig. 10 shows the compute time of the FA relative to the number of operations (total cell updates in all matrices of batch). It also shows the threshold at which the FPGA enabled. The threshold was set at 10,000,000 operations to avoid the DMA overhead. While the FPGA solutions can be much faster than the OpenMP solution the cumulative time is quite similar because of the job distribution as can be seen in Fig. 11. The FPGA outperforms the OpenMP implementation on bigger batches by quite a margin, however the sheer number of small batches that are not run on the FPGA limits the potential gains.

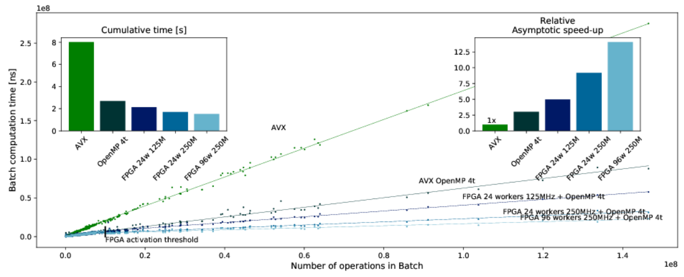

Fig. 10. Computation time relative to number of cell updates in Batch. Cumulative (total) time of all the Batches in region 10,000,000 to 10,200,000 on chromosome 20 of NA12878. And asymptotic relative speed-ups. Lines are the least square regressions given the data points.

4) Comparison with other solutions: The common benchmark to most accelerators is the 10s dataset. It is used to compute the number of giga cell updates per second (GCUPS) capabilities of the solution. This measure is equal to the number of operations required to compute the Pair-HMM FA over the whole dataset divided by the runtime. Table III

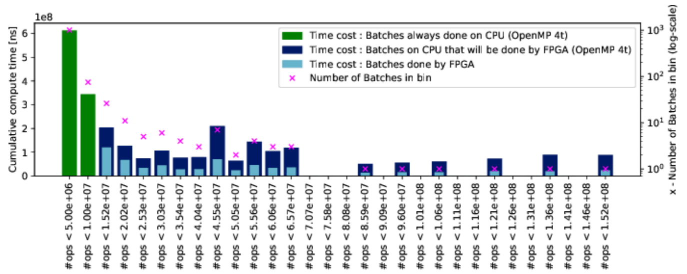

Fig. 11. Cumulative computation time of Batches binned by number of cell updates (ops) in region 10,000,000 to 10,200,000 on chromosome 20 of NA12878. And number of batches in bin ×.

reports the different GCUPS values. Our computation kernel achieves close to 24 GCUPS, 96 compute engines running at 250 MHz, the full system however performs around 7 GCUPS on this dataset due to the data transfer overhead. The GCUPS performance is dependent on the dataset, this relation has been studied in [20]. Results from [20] were used for the Intel values since Intel changed their dataset in [13]. Finally, only [13] and [22] gave overall "in-pipeline" measures which both gave 1.2× improvements on the execution time compared to the AVX solution. Our solution achieved a similar overall speed-up of 1.3× on chr1, and 1.2× on both chr2 and chr3.

TABLE III
GCUPS PERFORMANCE COMPARISON WITH OTHER ACCELERATORS

Technology	Hardware	#PE	GCUPS
Java [10]	CPU	-	0.0058
C++ [10]	CPU	-	0.049
Intel AVX (1 core) [13] [20]	Intel Xeon	-	0.45
Nvidia GPU [10]	Nvidia K40	-	0.89
Intel AVX 24 core [13] [20]	Intel Xeon	-	4.16
Nvidia GPU [22]	Nvidia K40	-	4.87
RS [17]	XC7VX690T	32	5.32
FPGA 96w w/ DMA (ours)	XCVU9P	96	7.04
Intel OpenCL [13] [20]	Stratix V	64	7.52
PE Ring (simulation) [18]	Stratix V	64	11.77
PE Chunks [20]	XCKU060	64	12.48
Intel OpenCL [13] [20]	Arria 10	208	22.28
FPGA 96w kernel (ours)	XCVU9P	96	23.54
PE Ring (simulation) [18]	Arria 10	128	23.99

VI. Discussion and Future Work

The premise for hardware acceleration of the Pair-HMM FA was that it was the major contributor to the compute time in the HC tool. However with multi-threaded software accelerated versions the contribution is now less than 20% meaning that hardware accelerators can only improve the tool overall compute time to that extent. Although small contributions are still valuable, especially with big workloads, in order to further improve the HC other sections of the tool need to be revised. The impact can be seen with the parallel Spark implementations of the tool, resulting in overall speed-ups of the HC tool of 2.9×, 2.8×, and 2.4× on chr1-3 over AVX and 3×, 2.9×, and 2.7× when FPGA accelerated.

While the FPGA accelerator can improve the time of Pair-HMM FA and achieve speed-ups of 3×, 4×, 5× on chr1-3 over the AVX solution, the smaller contribution relative to the full tool mitigates the overall gains on the HaplotypeCaller tool.

The accelerator itself could be improved by bypassing the external DDR4 memory completely. Having the CPU streaming the batches to the workers directly instead of generating the jobs in the FPGA and having the workgroups query the data from the DDR4. This the main bottleneck since all workers compete to access the DDR4, the latency from the DDR4 also adds up to the computation time.

VII. Conclusions

We proposed a hybrid FPGA - CPU solution for the Pair-HMM FA that is up to $15\times$ faster than the original Java implementation and up to $5\times$ faster than the AVX accelerated version. Our solution was integrated in the GATK pipeline. We not only managed to speed-up computations but also provided means to share the accelerator between instances of the pipeline or in a Spark distributed setup. While FPGA solutions can achieve extremely high speed-ups against CPU solutions when comparing the compute kernels alone, data locality is the real problem. Depending on the size of the batches and given the current design of the GATK pipeline it is not possible to have them perform at peak performance. Therefore, a hybrid solution, using the accelerator only when a significant gain is possible seems the best option. This solution also benefits from the fact that CPU and FPGA can be used in tandem, exploiting both strengths.

The accelerator was implemented for Amazon F1 cloud instances in order to provide an accelerator which can be used without requiring to buy specific equipment. In order to achieve better performance increments a rework of the GATK pipeline is required. Sequential processing of small batches makes it difficult to take advantage of hardware co-processors. The development of the Spark parallel implementations of the GATK tools could be an opportunity to integrate co-processors in a more optimal way. E.g., by applying the pre-processing of the data in parallel (map) then generate a batch with all required data for computation (reduce) and send it to a hardware accelerator in one go. Retrieve all results and post-process them in parallel again (map).

Acknowledgments

The authors would like to thank HEIG-VD for the funding of the Power Efficient Hardware Acceleration of Genomic Algorithms (PEHAGA) program which made this project possible, as well as Ioannis Xenarios from UNIL for the fruitful discussions.

References

[1] The human genome project completion: Frequently asked questions. Accessed: March 2018. [Online]. Available: https://www.genome.gov/11006943/human-genome-project-completion-frequently-asked-questions/

[2] E. R. Mardis, "A decade's perspective on DNA sequencing technology," *Nature*, vol. 470, pp. 198 EP –, Feb 2011, perspective.

[3] DNA Sequencing Costs: Data - Data from the NHGRI Genome Sequencing Program (GSP). Accessed: March 2018. [Online]. Available: https://www.genome.gov/sequencingcostsdata/

[4] A. Sboner, X. J. Mu, D. Greenbaum, R. K. Auerbach, and M. B. Gerstein, "The real cost of sequencing: higher than you think!" *Genome Biology*, vol. 12, no. 8, p. 125, Aug 2011.

[5] P. Muir *et al.*, "The real cost of sequencing: scaling computation to keep pace with data generation," *Genome Biology*, vol. 17, no. 1, p. 53, Mar 2016.

[6] H.-C. Ng, S. Liu, and W. Luk, "Reconfigurable acceleration of genetic sequence alignment: A survey of two decades of efforts," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–8.

[7] Inside the Microsoft FPGA-based configurable cloud. Accessed: December 2017. [Online]. Available: https://azure.microsoft.com/en-us/resources/videos/build-2017-inside-the-microsoft-fpga-based-configurable-cloud/

[8] Children's Hospital of Philadelphia and Edico Genome achieve fastest-ever analysis of 1,000 genomes. Accessed: February 2018. [Online]. Available: http://edicogenome.com/childrens-hospital-philadelphia-edico-genome-set-guinness-world-records/

[9] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, "The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, 2010.

[10] M. Carneiro, "Accelerating variant calling," in *Broad Institute, Intel Genomic Sequencing Pipeline Workshop, Powerpoint Presentation, Mount Sinai*, 2013.

[11] Gatk haplotypecaller documentation. Accessed: July 2019. [Online]. Available: https://software.broadinstitute.org/gatk/documentation/tooldocs/current/org_broadinstitute_hellbender_tools_walkers_haplotypecaller_HaplotypeCaller.php

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

[13] C. Rauer *et al.*, "Accelerating Genomics Research with OpenCL and FPGAs."

[14] Testing FPGA implementation of HaplotypeCaller (PairHMM). Accessed: July 2019. [Online]. Available: https://gatkforums.broadinstitute.org/gatk/discussion/10501/testing-fpga-implementation-of-haplotypecaller-pairhmm

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

[16] 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)*, April 2016, pp. 1–3.

[17] J. Peltenburg, S. Ren, and Z. Al-Ars, "Maximizing systolic array efficiency to accelerate the PairHMM forward algorithm," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2016, pp. 758–762.

[18] S. Huang *et al.*, "Hardware Acceleration of the Pair-HMM Algorithm for DNA Variant Calling," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 275–284.

[19] S. S. Banerjee, M. el-Hadedy, C. Y. Tan, Z. T. Kalbarczyk, S. Lumetta, and R. K. Iyer, "On accelerating pair-HMM computations in programmable hardware," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–8.

[20] D. Sampietro, C. Crippa, L. Di Tucci, E. Del Sozzo, and M. D. Santambrogio, "FPGA-based PairHMM Forward Algorithm for DNA Variant Calling," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–8.

[21] S. Ren, K. Bertel, and Z. Al-Ars, "Exploration of alternative GPU implementations of the pair-HMMs forward algorithm," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, Dec 2016, pp. 902–909.

[22] S. Ren, K. Bertels, and Z. Al-Ars, "Efficient Acceleration of the Pair-HMMs Forward Algorithm for GATK HaplotypeCaller on Graphics Processing Units," *Evolutionary Bioinformatics*, vol. 14, 2018.

[23] Y.-T. Chen *et al.*, "When Spark Meets FPGAs: A Case Study for Next-Generation DNA Sequencing Acceleration," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[24] M. A. Eberle *et al.*, "A reference dataset of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree," *bioRxiv*, 2016.