On Accelerating Pair-HMM Computations in Programmable Hardware

Subho S. Banerjee, Mohamed el-Hadedy, Ching Y. Tan,
Zbigniew T. Kalbarczyk, Steve Lumetta and Ravishankar K. Iyer
University of Illinois at Urbana-Champaign
Urbana, IL - 61801, USA.

***Abstract*—This paper explores hardware acceleration to significantly improve the runtime of computing the forward algorithm on Pair-HMM models, a crucial step in analyzing mutations in sequenced genomes. We describe 1) the design and evaluation of a novel accelerator architecture that can efficiently process real sequence data without performing wasteful work; and 2) aggressive memoization techniques that can significantly reduce the number of invocations of, and the amount of data transferred to the accelerator. We describe our demonstration of the design on a Xilinx Virtex 7 FPGA in an IBM Power8 system. Our design achieves a 14.85× higher throughput than an 8-core CPU baseline (that uses SIMD and multi-threading) and a 147.49× improvement in throughput per unit of energy expended on the NA12878 sample.**

***Keywords*—Coprocessors, Reconfigurable architectures, Bioinformatics, Genomics.**

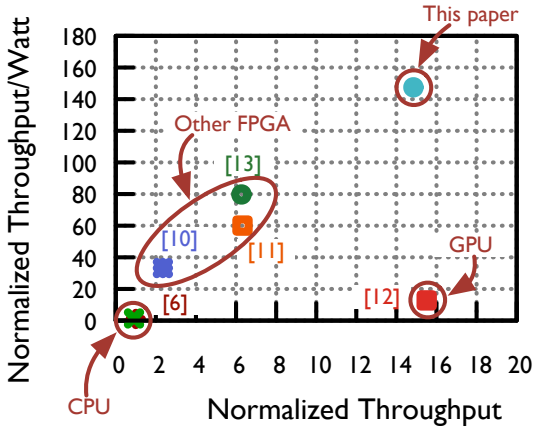

Figure 1. Comparing this paper with related work based on throughput (MCUP/s) and throughput-per-watt (MCUP/s/Watt). Values are normalized to our CPU baseline running on a Power8 CPU.

I. Introduction

An important computation (and a critical bottleneck) in medical sequence analysis pertaining to the analysis of variants (mutations) in sequenced genomic data is the *forward algorithm* [1] (FA) on *Pair-Hidden Markov Models* [2] (PHMMs) (see Section II). The FA algorithm, which is generally viewed as one of the best ways to compute the statistical similarity between two sequences, is widely used in genomic data analysis workflows for gene prediction, functional similarity analysis between protein sequences, multiple sequence alignment, phylogeny, and germline- and somatic-variant calling [3]. This paper addresses the problem of accelerating the GATK HaplotypeCaller [4], a popular and trusted variant calling and genotyping tool[1] that incorporates a PHMM model (described in Section II-B) and is widely used in clinical settings (e.g, in the diagnosis of critical diseases like cancer). The FA constitutes the most computationally complex phase of this application, accounting for nearly 70-80% of the runtime while processing human clinical datasets.

The FA algorithm is inherently parallelizable at two levels: 1) at the level of the algorithm, i.e., *intra-task* parallelism through the anti-diagonal recurrence pattern in a single FA execution; 2) at the level of the data, i.e., *inter-task* parallelism by computing the independent instances of FA in parallel throughout the course of analysis. All of the prior efforts to address this problem [6]–[13] use some form of a systolic array (SA) architecture. These architectures look to optimize only for intra-task parallelism; they underutilize on-chip resources and waste energy when input sizes are not multiples of the number of processing elements.[2] As a consequence, such designs cannot efficiently handle realistic data where input sizes (i.e., the lengths of the query DNA fragments) can vary significantly (see Section III-A). A common thread of research in this area has been to utilize control algorithms and data placement strategies to overcome these shortcomings, thereby leading to increased algorithmic complexity (for CPUs and GPUs) and larger on-chip areas for FPGAs [6], [11]–[13].

This paper proposes (in Section III) the design of an accelerator for the FA algorithm that overcomes the aforementioned shortcomings. Unlike previous approaches, we spent our entire resource budget optimizing for inter-task parallelism (thereby exploiting the embarrassingly parallel nature of the problem). Intra-task parallelism is addressed by deep pipelining to maximize temporal sharing (reuse) of computational resources. We demonstrate that this design maximizes overall throughput by optimally using parallelism, and minimzes control related hazards and stalls. Our accelerator produces a speedup of 14.85× over an 8-core Power8 processor executing the baseline software implementation, and is 147.49× better in speedup-per-unit-energy terms. Figure 1 demonstrates this speedup compared to the related work available on this problem (explained further in Section VI). In our initial design implementation on an IBM Power8 system and using an FPGA attached over the CAPI [14] interface, we observe that the key performance-limiting factors are 1) latency overhead in accelerator invocation through the software stack, and 2) redundant computation done

[1]It is a part of the Broad Institute Best Practices Workflow [5] for variant-calling.

[2]For example, a 32 processing element systolic array performs 1568 (56 × 28) unnecessary computations when processing two sequences of lengths 100, and 200.

across multiple unrelated instances of the FA. To alleviate these performance bottlenecks, we propose (in Section IV) two additional algorithmic optimizations that span the hardware-software interface. These techniques prune the inputs of the FA algorithm and memoize its output 1) to reduce the number of invocations of the FA kernel, and 2) to reduce the size of the sequences being compared. The result is an effective reduction in the amount of data that has to be transferred from the host to the accelerator to complete a batch of FA computations; this contributes a further $2.8\times$ improvement in performance.

The main contributions of this paper are as follows:

1) Identifies the performance-limiting issue with today's CPU/GPU and systolic-array-based FA accelerators.
2) Presents the architecture of an ensemble of processing elements that maximize inter-task parallelism and uses aggressive pipelining to address intra-task parallelism, thereby overcoming the inefficiencies of the SA architectures.
3) Evaluates the proposed design on an FPGA, and couple it with a coherent interface to the CPUs memory, allowing work-sharing between the CPU and accelerator.
4) Presents two pruning strategies to memoize results to reduce the input data-set size as well as the number of invocations of the FA accelerators.
5) Demonstrates that integration of the accelerator and pruning techniques into the GATK HaplotypeCaller can accelerate it by $3.287\times$ (close to the Amdahl's law limit) over the baseline CPU implementation.
6) Evaluates the potential impact of several emerging high-bandwidth memory technologies to alleviate the host-accelerator bandwidth limitations in PCIe/CAPI.

II. Background

A. Pair-HMM Model

PHMM models are instances of Bayesian multinets that allow for a probabilistic interpretation of the alignment problem [2]. An alignment models the relationship (homology) between two sequences via a series of mutations (M), insertions (I), and deletions (D) of nucleotides. The FA algorithm of the PHMM allows the computation of statistical similarity by considering all alignments between two sequences and computing the overall alignment probability by summing over them. These divergent sets of alignments are caused by evolutionary mutations or sequencing errors. Specifically, given two sequences S_1 and S_2 of lengths n and m respectively, the FA algorithm defines the computation of three probabilities, $f_M(i,j)$, $f_I(i,j)$, and $f_D(i,j)$. $f_k(i,j)$ corresponds to the combined probability of all alignments for substrings $S_1[0:i]$ and $S_2[0:j]$ that end in state $k \in \{M, I, D\}$. The FA algorithm can be recursively defined as follows:

$$\left.\begin{array}{rcl} f_M(i,j) & = & P(a_{mm} f_M(i-1,j-1) + \\ & & a_{im} f_I(i-1,j-1) + \\ & & a_{dm} f_D(i-1,j-1)) \\ f_I(i,j) & = & a_{mi} f_M(i-1,j) + a_{ii} f_I(i-1,j) \\ f_D(i,j) & = & a_{md} f_M(i,j-1) + a_{dd} f_D(i,j-1) \end{array}\right\} \quad (1)$$

The parameters a_{mm}, a_{im}, a_{dm}, a_{mi}, a_{ii}, a_{md}, a_{dd} and P are derived from the values of base-quality scores, map-quality scores, and the values $S_1[i]$ and $S_2[j]$. They represent a statistical model that jointly describes 1) dependence between adjacent nucleotides, 2) dependence between hidden and observed sequences that describes a multi-nucleotide mutation model, a point mutations model, and 3) sequencing and alignment errors using an affine gap score model [2]. Patcher et al. [1] describes the rationale behind using these quality metrics in the PHMM model to set the a_* parameters. Finally, the overall similarity metric between the sequences is the sum of the probabilities across the states M, I, and D when comparing the entire strings S_1 and S_2, i.e., $f_M(n,m) + f_I(n,m) + f_D(n,m)$. Hence, each FA needs to compute the recursion stated in Equation 1 $n \times m$ times. That corresponds to a total computational-time complexity of $O(nm)$ and a total space complexity of $O(\max(n,m))$.

Algorithm 1 Algorithmic skeleton of the GATK HaplotypeCaller. The functions `EnumerateHaplotypes`, `Align`, and `Genotype` are described in [15].

```
1:  alignment ← Aligned set of reads in an active region
2:  reference ← Reference genome for an organism in an active region
3:  n ← Number of ploids in the organism
4:  haplotypes ← EnumerateHaplotypes(alignment.reads)
5:  for h ∈ haplotypes do
6:      for r ∈ reads do
7:          score[h, r] ← PairHMM(h, r)
8:      end for
9:  end for
10: best_haplotypes ← Find n-best haplotypes
11: new_align ← ∅
12: for r ∈ alignment.reads do
13:     new_align ← argmax_{h∈bh} Align(r, h)
14: end for
15: variants ← ∅
16: for haplotype ∈ best_haplotypes do
17:     for loc ∈ haplotype do                ▷ Every position in the haplotype
18:         variants ← variants ∪ Genotype(new_align.atLocus(loc))
19:     end for
20: end for
21: return variants
```

Table I. Distribution of runtime between the phases of Algorithm 1 in the GATK HaplotypeCaller for the NA12878 sample executing on the hardware configuration described in Section V.

Stage	Absolute Time (hr)	Percentage Time	Line Number
Assembly	2.87	13.8	1–4
PHMM FA	14.78	71.1	5–9
Realign + Misc	3.13	15.1	10–21

B. Germline Variant Calling

In this work, we accelerate the germline variant-calling and genotyping tool called the GATK HaplotypeCaller (or GATK), which statistically infers differences between sequenced genomes and *reference genomes*, where reference genomes represent "average" genome for a population of the candidate species. The base algorithm of variant calling and genotyping is straightforward: input sequence fragments are aligned to a reference sequence, and at every position the number of mismatches are counted. However, this process is complicated by the fact that data from sequencing machines are inherently noisy (from sequencing errors), alignments are often incorrect (from mapping errors) and polyploidy of an organism (i.e., there are many copies of a genome per individual). The PHMM model is applied in this context to model the aforementioned errors (e.g., sequencing errors, alignment errors, or mutations) statistically and to assign sequenced DNA fragments to their corresponding ploids. GATK computes the variants in a sequenced genome by filtering

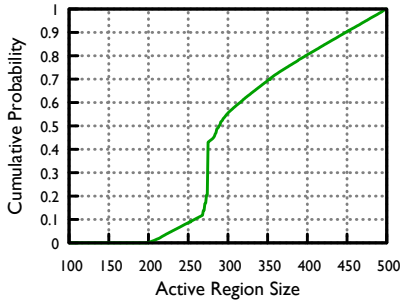

Figure 2. Distribution of active region (haplotype) sizes in a sample of the NA12878 dataset.

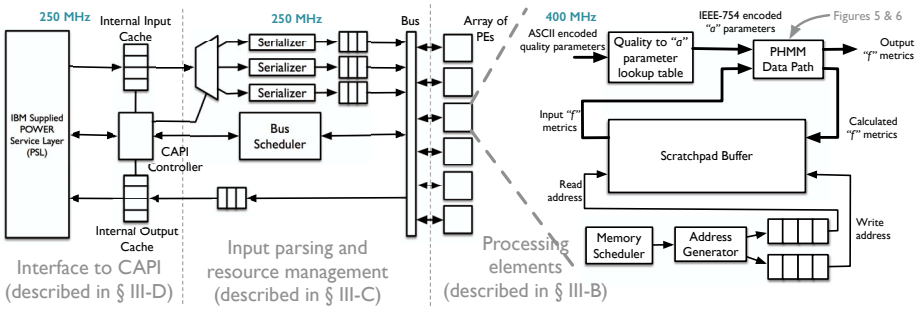

Figure 3. Architecture of the data-path and the control-path of the proposed accelerator comprising the 1) host-accelerator interface over CAPI, and 2) input parsing and load balancing.

the genome into *active regions* that might contain possible mutations. Algorithm 1 is then applied in parallel to all active regions in order to reconstruct haplotypes (using DeBruijn graphs [16]) for the ploids for each active region. The FA algorithm is then used to compute the probability that a sampled sequence fragment originated from a certain haplotype. These probabilities are used to weight each haplotype to find a candidate set that might best represent a ploid. Finally, the reads are realigned to their best haplotype. A count of the number of mismatches to the haplotype (instead of the reference) is then used to determine the presence of a variant and its genotype [17]. GATK uses single-precision floating-point numbers to compute Equation 1. In the case of an underflow, double precision floating-point numbers are used to recompute the result. The FA algorithm constitutes the bulk (nearly 70%, see Table I) of the runtime of GATK on the popular NA12878 sample from the GIAB consortium [18], and as such is a good candidate for acceleration. Banerjee et al. [15] show that GATK and transitively the FA computation also forms a significant portion of the runtime of sequencing data-analytics workflows on modern compute infrastructures (e.g., clouds and supercomputers).

III. Accelerator

A. Design Philosophy

Based on our analysis of the algorithms and input datasets, we offer a set of insights that drove our design philosophy:

Insight 1. *Diversity in input size.* Haplotypes generated by the HaplotypeCaller show great diversity in size (see Figure 2). Traditional SA-based architectures for accelerating the FA algorithm are often not able to handle this diversity effectively because of the cycles wasted when the size of the recursion lattice is not divisible by the number of PEs (processing elements) in the SA, resulting in holes in the processor's pipeline. Traditional SA based architectures deal with these issues by using complicated control mechanisms [11]–[13] that improve pipeline utilization. CPU- and GPU-based architectures that exploit SIMD instructions [6], [7] also experience this issue, albeit to a lesser degree.

Insight 2. *Exploiting inter-task parallelism.* Systolic array architectures exploit anti-diagonal parallelism to minimize latency for a single task. However, given that the FA algorithm itself demonstrates several orders of magnitude greater parallelism between tasks than within tasks, we believe that it is more prudent to exploit inter-task parallelism. Such an approach also addresses the problem of input diversity, as we can exploit the data-parallel nature of the problem without data dependencies between PEs. As we show in our results (Sec. V), the increased data set size needed for inter-task parallelism does not limit our implementation.

Insight 3. *Why not GPUs, and why FPGAs?* Insights 1 & 2 discourages the use of GPUs because of its programming model of lock-step parallelism. Previous efforts of using GPUs in the context of this problem have successfully extracted intra-task parallelism at the warp-level. However, input diversity leads to control divergence when inter-task parallelism is exploited. In contrast, FPGAs provide the flexibility to build a processing pipeline that is tailored to the computation at hand and its input characteristics. We explore the difference in performance between GPUs and our method in Section VI. We demonstrate that the our design is unmatched in the performance-energy trade-off space, however we concede that the GPU represents a different (and in some cases a preferable) point on the performance–developer-productivity spectrum.

Figure 3 illustrates the overall design and implementation of the accelerator. The accelerator is organized as an array of independent processing elements (PEs; see Section III-B), which asynchronously pull inputs from the CPU's memory space over a cache-coherent CAPI bus (see Section III-D). Reads and writes to the host processor's memory space are made in a streaming fashion to overlap computation of haplotype-read pairs on the host with the accelerator.

B. Processing Element (PE)

Figure 3 also shows the design of a single PE. The nucleotide inputs to the PE are encoded as 4-bit unsigned numbers. These correspond to nucleotides `A`, `C`, `G`, `T`, and the ambiguous nucleotides `N`, `-`, `R`, `Y`, `K`, `M`, `S` and `W`. The remaining symbols can be used to accommodate FA on protein sequences. The quality-score inputs are in their standard ASCII encoding [2]. These are used to compute the FA algorithm parameters. Each PE has: 1) a table-lookup based function to compute floating-point parameter (probability) values from input quality scores; 2) a data-path consisting of single precision floating-point adders and multiplier; 3) a scratchpad buffer to store intermediate values of the f_k matrices (where $k \in \{M, I, D\}$); and 4) scheduling circuitry that generates a valid sequence of read-write addresses for the scratchpad buffer. Using table-lookup allows us to use 8-bit encoding of quality scores as opposed to their 32-bit floating point encoding when transferring inputs to the accelerator, thereby reducing IO bandwidth requirements. Each PE is fed from a BRAM bank

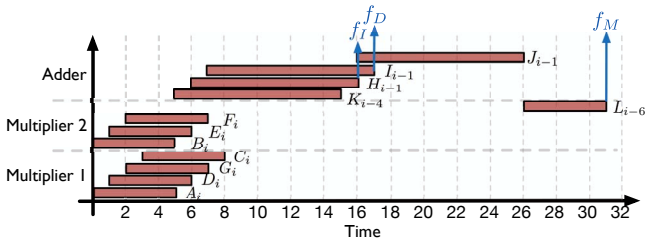

Figure 4. Schedule of the adders and mulipliers on the datapath illustrated in Figure 5.

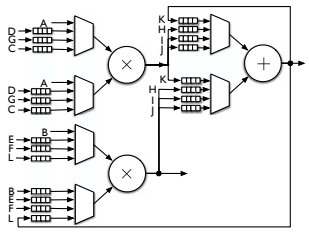

Figure 5. Circuit diagram for the PE data-path shown in Figure 3.

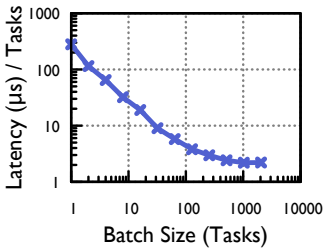

Figure 6. Quantifying the acclerator call and data-transfer overhead.

that stores the values of each of its inputs. We now briefly describe the design of the data-path and the scheduler.

1) Data-Path: The data-path of the PE has to implement the recurrence relations in Equation 1. Each step of this recurrence has 8 multiplication operations and 4 addition operations. Our design finds the optimal trade-off point between latency/throughput and resource utilization on the chip to implement this computation. We use a Xilinx intellectual property (IP) that provides a 5-cycle latency and 1-op/cycle throughput for the float-multiplier and a 10-cycle latency and 1-op/cycle throughput for the float-adder. Both the adders and multipliers run at 400 MHz. Our design utilizes two multipliers and one adder for a 32-cycle latency and 0.25-recursion-step/cycle throughput schedule. Figure 4 demonstrates the utilization of the two multipliers and adders in the 32-cycle period to compute one step of the recurrence. The circuit representation corresponding to this schedule is shown in Figure 5. The inputs of the circuit elements, labeled A through L are shown in both Figures 4 and 5, where the subscripts represent the step of the recurrence currently being computed. For example, A_i and L_{i-6} (in Figure 4) represents that the i^{th} recurrence step for A and the $i-6^{th}$ recurrence step for L is computed in the same 32-cycle window. We achieve synchronization in this scheme by using shift registers attached to the muxed inputs of the adder and multipliers as shown in Figure 5. The lengths of these registers can be derived from Figure 4. The outputs of this data-path is fed back into the inputs of the following stages via the scratchpad buffer (implemented as a BRAM block) shown in Figure 3.

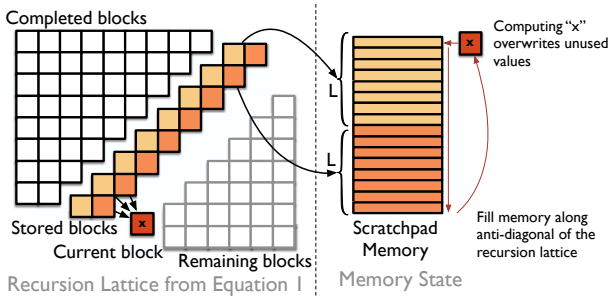

Figure 7. Exploiting anti-diagonal *wavefront* pattern of Equation 1 to optimize memory size to $L-2$, where $L = \min(n, m)$

2) Memory Scheduler: To minimize the size of the scratch-pad buffer (in Figure3) we compute steps of the recurrence (Equation 1) in an anti-diagonal fashion, as shown in Figure 7. In the figure, we illustrate the antidiagonal pattern by dividing the entire recurrence lattice into four parts as follows:

1) *Completed Blocks.* Blocks for which the value of the recurrence has been computed and is no longer required.
2) *Stored Blocks.* Blocks for which the value of the recurrence has been computed and these values are required in the subsequent steps of the computation.
3) *Current Block.* Block whose inputs have been produced and can start computation.
4) *Remaining Blocks.* Blocks whose inputs have not yet been generated.

Here each block refers to the three tuple $(f_M(i,j), f_I(i,j), f_D(i,j))$. We observe that that limiting the maximum size of the buffer to $2L$, where L is the size of the largest anti-diagonal, is sufficient to compute the FA algorithm. Figure 7 demonstrates that once this buffer is full, simply starting over at the begining only overwrites data that is no longer required for the computation. Our current implementation supports matrices up to $L = 512$. This limit is sufficient to accommodate the largest haplotype (500 bases) generated by GATK, as well as reads from the popular Illumina HiSeq sequencing platforms (150–250 bases long). The scheduler generates a pattern of read and write addresses into the memory for the aforementioned data-path. The scheduler deals with the upper and lower triangles in the recurrence lattice (e.g., $i + j \leq 8$ where $8 \times {}^{1}/_{0.25}$ is the latency of the pipeline), when a PE's pipeline cannot be kept full because of the dependencies between the inputs and outputs of the recurrence.

C. On-Chip Bus-Scheduing and Load-Balancing

On the accelerator side, we multiplex inputs from the CPU (1024 bit cache lines) among the array of processors by parsing the input stream through a *"Serializer"* and storing the FA executions in FIFOs to be fed to idle processors over a bus (see Figure 3). We observed that using cache line aligned inputs significantly reduces the complexity of parsing the input stream on the accelerator side, though this incurs an overhead on the CPU side. Our experimental system (described in Section V) had 1TB of RAM attached to the CPUs and hence this is an acceptable trade-off. This design decision can be revisited for other machine configurations. It is to be noted that this does not affect performance of the accelerator, merely the complexity of the "Serializer." We use a straightforward arbitration mechanism for the bus described in Figure 3. In the case of ties, the bus scheduler arbitrates inputs in a round-robin fashion. Outputs are handled in a similar fashion.

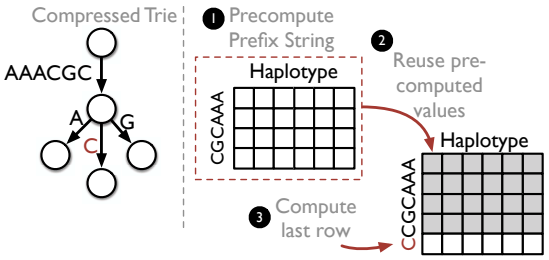

Figure 8. Exploiting common subsequences between reads in an active region to memoize the computation of the FA

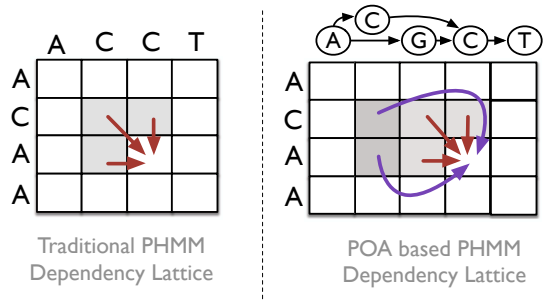

Figure 9. Dependencies between cells when calculating FA on a de-Bruijn graph

D. Host-Accelerator Interface

Communication between the host and accelerator is implemented using the CAPI interface [14] with an IBM Power8 CPU. The CAPI interface allows an accelerator (a PCIe-attached FPGA) coherent access to the virtual address space of a process running on the host CPU. Our accelerators pull data directly from three circular buffers in the processor's memory space. These correspond to the reads, haplotypes, and quality scores. The CPU generates new tasks (new instances of the FA algorithm to be computed) by executing Algorithm 1 and enqueues inputs to the accelerator into the respective circular buffers. The accelerator and other threads on the CPU then consume these inputs from the buffers. CAPI is beneficial in this instance, as we can make use of the cache coherency between the CPUs and FPGA to easily implement mutual exclusion.

IV. Algorithmic Optimizations

In building the accelerator as described in Section III, we noticed that *batch size* (number of tasks streamed to the accelerator at a time) has a particularly dominant effect in the performance. Figure 6 illustrates this loss in performance as a function of batch size. We attribute this behavior to the software overheads (e.g., system calls, IRQ handlers) that initiate the accelerator. Our observations can be explained by the fact that simply batching tasks amortizes this cost over several individual accelerator invocations. To further improve 1) batching efficiency (latency amortization), 2) host-accelerator data transfer bottlenecks (PCIe limitations), and 3) reuse of precomputed results (across multiple tasks), we have developed two algorithmic methods for pruning inputs and memoizing outputs of the FA algorithm when it is used in conjunction with the HaplotypeCaller. It is to be noted that both the optimizations reduce the throughput of the FPGA accelerator, but improve end-to-end performance by reducing the number of computations that have to be performed.

A. Common Prefixes in Reads

We observe that when Algorithm 1 is invoked on active regions of high-coverage, high-quality datasets, a large number of reads share a common prefix. This is an artifact of the alignment process, of having similar reads start close to each other, and of repeats in the reference genome. According to the formulation of the FA algorithm, for the same haplotype, reads with common prefixes produce exactly same results under Equation 1. We exploit this observation by reusing values of the recurrence relation for common prefixes. This is done by constructing a compressed trie [19] of the reads in an active region, and computing the longest common prefix (LCP) in the trie. The accelerated FA algorithm is then computed on the LCP and the answer is memoized. All substrings of the LCP in the trie can then use the lattice values computed in the LCP as a starting point for further computation. In fact, this optimization can be reapplied once the LCP is removed from the trie. Though straightforward, this heuristic significantly reduces the amount of computation required in the HaplotypeCaller (by $20-30\%$ for human clinical datasets). For example, consider the read sequences `AAACGCA`, `AAACGCC` and `AAACGCG`; they share a common prefix `AAACGC`. Figure 8 illustrates a compressed trie consisting of these reads as well as the use of the LCP to memoize the result of the FA algorithm. We see that the LCP can be reused across the three reads and the computation can be reduced to 1) FA on the LCP 2) Computing 3 rows corresponding to `A`, `C` and `G`. This optimization is carried out on the host-side (CPU side). The CPU schedules the LCP lattices on to the FPGA accelerator, and the remaining computations are carried out on the CPU using the SIMD (AVX or AltiVec instructions). This optimization significantly reduces: 1) number of invocations of the accelerator, 2) the amount of data that has to be moved to the accelerator, and 3) the amount of redundant computation that is performed on the accelerator.

B. FA on De Bruijn Graphs

In addition to the reads that share common prefixes, the haplotypes generated by the assembly process in Algorithm 1 also share large common subsequences. The De-Bruijn graph [16] produced as a result of the assembly on an active region encodes these common subsequences in a graphical format much like the compressed trie in the previous section. Computing the FA of a read and the De-Bruijn graph potentially allows us to reuse these values as well as reduce the number of invocations of the FA algorithm from the number of reads $\times$ the number of haplotypes to just the number of reads (when executing independent of the optimization in Section IV-A). The FA algorithm has to be modified to allow computing similarity between a De Bruijn graph and a read. Given that the graph represents a partially ordered set of strings (haplotypes), we first compute a topological sort of the graph to convert it into a total order. Then, we follow the same FA algorithm as described in Equation 1, with one major difference: dependencies between lattice elements now incorporate the De-Bruijn graph. Figure 9 gives an example of conflating graph dependencies with FA dependencies. As the quantities being added in Equation 1 represent probabilities, and the branches on the De-Bruijn graph represent mutually exclusive subsequences of haplotypes,

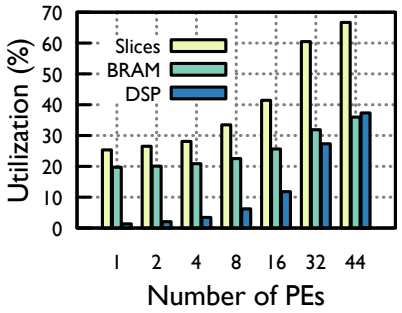

Figure 10. Resource utilzation on the FPGA as a function of the number of PEs.

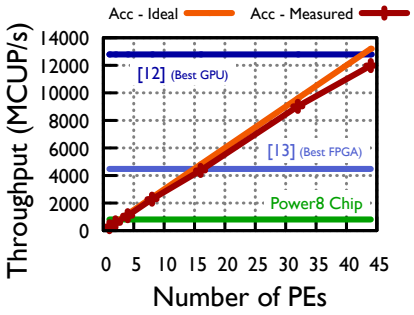

Figure 11. Mean throuhgput of the accelerator as a function of the number of PEs.

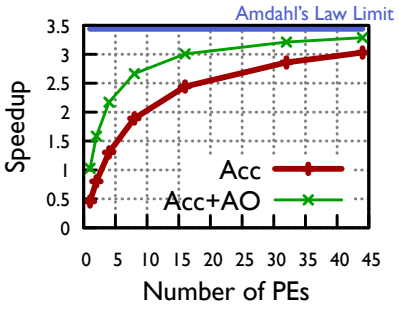

Figure 12. Mean end-to-end speedup of the HaplotypeCaller when applying the accelerator (ACC) and algorithmic optimizations (AO).

this transformation produces the correct answer. This is a generalization of Lee et al.'s POA algorithm [20] to PHMM models. The software controller is augmented with the ability to dispatch subsequences of haplotypes to the FA accelerator, instead of traversing the De-Bruijn graph and dispatching entire haplotypes. The final reduction (addition), of the various topologically sorted subsequences is computed on the CPU. This reduction corresponds to the additional dependencies shown in Figure 9.

V. Experimental Results

The accelerator is implemented in mixed-language HDL. We used IPs from Xilinx to implement the single-precision floating point adder and multipler and BRAM blocks. We implemented the accelerator on an IBM Power8 S824L system with an Alpha-Data ADM-PCIE-7V3 board (that uses a Xilinx Virtex 7 XC7VX690T FPGA). All the IO interfaces were clocked at 250 MHz, and the PEs were clocked at 400MHz. All measurements (baseline CPU as well as FPGA based) were done on this system. Our input data-set for this section was derived from sample G15512.HCCI954.1 (same as used in [10], [13]) and the hg38 reference human genome. We verified the correctness of our FPGA implementation by comparing it to the CPU-only version of the GATK HaplotypeCaller (`v3.6` - Provides SIMD optimized and multi-threaded implementations of the algorithm). We use the C++ implementation of the FA on the Power8 CPU as a baseline [21]. This has been optimized using AltiVec SIMD instructions and multi-threading. Section VI describes a comparison of this implementation to one [6] that uses AVX256 SIMD for x86 processors.

A. Resource Utilization

We observed near-linear scaling of the utilization of on-chip resources for the accelerator with the number of PEs (see Figure 10). Even though the figure shows a high utilization of logic slices on the FPGA, the actual numbers of LUTs and FFs are much lower. For example, in the 32 PE case, even though we used 60.47% of the slices, we used only 46.85% of LUTs and 26.64% of FFs on the FPGA. Figure 13 and Table II report the power consumption of the accelerator. These reports were generated by the Vivado Design Suite.

B. Performance of the Accelerator

Compared with a C++ implementation optimized by IBM for their 8-core Power8 architecture, the proposed accelerator

Table II. Comparing the power and performance of the proposed design to that of a Power8 CPU

	Throughput (MCUP/s)	Power (W)		
		Static	Dynamic	Total
Power8 Core	100.8	–	–	–
Power8 Chip	806.6	–	–	190
Single PE on FPGA	296.1	0.5	4.686	8.437
44 PEs on FPGA	11983.6	2.448	13.485	19.139

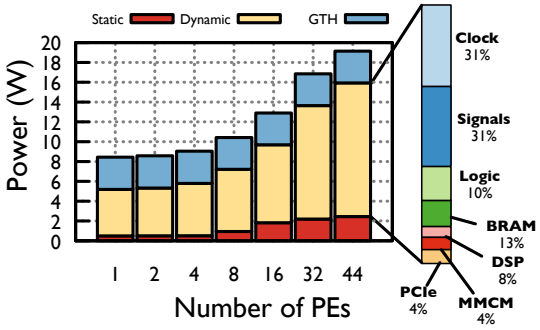

Figure 13. Power consumption of the accelerator as the number of PEs is increased.

increases aggregate throughput by $14.85\times$ (i.e., $^{11983.6}/_{806.6}$) in terms of throughput. We quantify throughput using the popular MCUP/s measure. A MCUP or mega cell update represents the computation of 10^6 steps of the recursion in Equation 1 (traditionally each recursion step is called a cell). Further, adding the algorithmic optimizations from Section IV we observe a $41.59\times$ (*i.e.,* 14.85×2.8) improvement in performance. Table II demonstrates that the proposed accelerator significantly outperforms the Power8 CPU in terms of power and performance. A single PE outperforms a Power8 core and a 44-PE accelerator outperforms an 8-core Power8 processor by $147.49\times$ (i.e., $^{11983.6}/_{806.6} \times {}^{190}/_{19.139}$) in terms of performance per unit energy (i.e., MCUP/Joule). Figure 11 describes the performance scaling of the accelerator with the number of PEs. At 44-PEs we observe some non-linearities (the difference between ideal and measured performance) because of insufficient off-chip bandwidth and limitations with the round-robin bus scheduling strategy in Section III-C.

C. Integration into GATK

Use of the proposed accelerator and algorithmic optimizations inside the GATK HaplotypeCaller demonstrated a

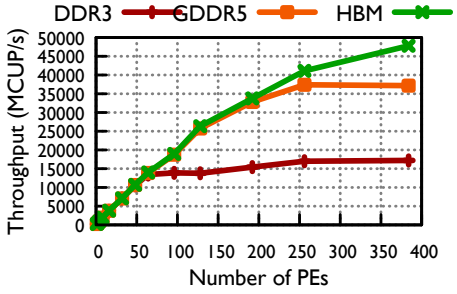

Figure 14. Simulated throughput when replacing the CAPI-based interface with a DRAM interface.

Table III. LIST OF RELATED HIGH PERFORMANCE IMPLEMENTATIONS OF PHMM. WE USE THE BEST RESULT PRESENTED IN THE RESPECTIVE PAPER TO COMPARE PERFORMANCE.

Paper	HLS	SA	Platform	Speedup	Speedup / Power
C++ Baseline [21]	–	–	Power8 CPU	1×	1×
[6], [7]	–	–	Intel Xeon CPU	0.91×	1.33×
[8]	✓	✓	Convey HC2	NA[a]	NA[a]
[9]	✓	✓	Stratix V D8	NA[a]	NA[a]
[10]	✗	✓	Power8 & Xilinx KU060	2.35×	33.24×
[11]	✗/✓[b]	✓	Arria 10	6.32×	60.04× (TDP = 20W)
[12]	–	–	Power8 & NVIDIA K40	15.51×	12.80× (TDP = 235W)
[13]	✗	✓	Power8 & Xilinx XC7VX	6.27×	79.74×
This Paper	✗	✗	Power8 & Xilinx XC7VX	14.85×	147.49×

[a] Insufficient data in cited paper to make comparison.
[b] Paper presents both HLS and HDL. We use best performance HDL implementation for comparison.

maximum acceleration of 3.287× in runtime when using 44 PEs (see Figure 12). The algorithmic optimizations presented in Section IV account for approximately 2.8× reduction in runtime of the FA algorithm. Figure 12 shows this improvement as it applies to end-to-end GATK application. It is to be noted that the optimizations from Section IV are input-dependent and can produce varying results for other datasets. Furthermore, Figure 12 demonstrates the diminishing returns from adding more processors in GATK because of Amdahl's law (3.44× asymptotic limit from Table I). After using the accelerator and optimizations presented in this paper, the `Align` function in Algorithm 1 dominated the runtime.

D. High-Bandwidth Memory Interfaces

With the industry trend of increasing the FPGA area in each successive generation, the number of PEs that fit into an FPGA will also grow. However, simple scaling of the number of processors leads to sub-linear performance scaling, as performance is limited by off-chip bandwidth for the FPGA through the PCIe/CAPI interconnect. To test the scalability of our accelerator, we replaced the CAPI interface with that of a simulated memory controller through the trace-driven simulation framework called Ramulator [22]. We observe that changing effective bandwidth can lead to significant non-linearities in scaling behavior (see Figure 14). For example, using HBM (which is already commercially available on flagship Xilinx Ultrascale+ FPGAs) leads to near-linear scaling of performance up to 256 PEs, after which non-linear scaling is observed. This performance scaling, though significant in terms of the FA algorithm, has almost no impact in the GATK HaplotypeCaller because of the diminished returns from Amdahl's law (as seen in Figure 12).

VI. RELATED WORK

In this section, we briefly describe related work (see Table III) that has accelerated the FA algorithm on a variety of processors (e.g., CPU, GPU, and FPGA devices). The accelerators in Table III are generally based on SA architectures, which suffer from the shortcoming of inefficient use of hardware resources for handling varied input sizes of real sequence data. Figure 1 is a graphical representation of Table III. Some of the related work does not clearly mention throughput and power measurements. In all such cases the Intel AVX implementation of the FA algorithm [6], [7] is used to normalize performance stated in the respective papers to that of our baseline. In the situation that power measurements are not provided in cited papers, we assume publicly stated TDP for the device used in the implementation. Our results demonstrate that the design proposed in this paper outperforms all the previous designs in terms of both the PHMM micro-benchmark, and transitively to the GATK HaplotypeCaller on representative datasets. It is to be noted that our performance results in Table III do not contain the algorithmic optimizations (from Section IV) that further improves performance by 2.8× (i.e., we compare *only* the design of the accelerator). The GPU implementation in [12] demonstrates marginally higher absolute performance (i.e., $1.04\times = {}^{15.51}/_{14.85}$), but our design represents a significant improvement (i.e., $11.5\times = {}^{147.49}/_{12.80}$) in performance-per-energy consumed terms. Furthermore, the K40 GPU has `x16` PCIe connections compared to our `x8`, and on-board GDDR5 memory, use of which will also benefit our design (see Figure 14).

Similarity to Smith-Waterman and other Levenshtein distance (LD) algorithms. The PHMM computation is an generalization of edit-distance formulation proposed by Levenshtein in [23] to probabilistic gap penalties [1]. Several LD variants have been accelerated in ASICs and FPGAs (e.g., [24]–[26]). However the key point of difference between the LD and the PHMM computations is in the use of floating point math which produces a significantly more complicated data-path. Furthermore, LD computations have been shown to be memory-bound for large lattices, whereas PHMM computations are significantly more compute intensive [19].

VII. CONCLUSION

This paper presented an approach to accelerating the computation of the FA algorithm in hardware. Our key insight of using input data characteristics to inform on architectural design patterns allows us to outperform traditional architectures in terms of both energy per operation and runtime performance. The use of this accelerator in genomic data analysis represents a significant acceleration in terms of time spent in computation. The proliferation of sequencing platforms and the resulting explosion in genomic data [27] will make this accelerator even more important in the future. Though most of the techniques presented in this paper are applicable only to the FA algorithm and transitively only in bioinformatics applications, the general design philosophy of using input data characteristics, in addition to the algorithmic definition, for design specialization of accelerators can be broadly applied across a large number of domains.

ACKNOWLEDGMENT

This material is based upon work partially supported by an IBM Faculty Award and the National Science Foundation under Grant No. CNS 13-37732. We thank Jenny Applequist for her help in preparing the manuscript.

REFERENCES

[1] L. Pachter and B. Sturmfels, *Algebraic Statistics for Computational biology*. Cambridge University Press, 2005., vol. 13.

[2] R. Durbin *et al.*, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[3] B.-J. Yoon, "Hidden Markov Models and their Applications in Biological Sequence Analysis," *CG*, vol. 10, no. 6, pp. 402–415, Sep 2009.

[4] A. McKenna *et al.*, "The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, Jul 2010.

[5] M. A. DePristo *et al.*, "A framework for variation discovery and genotyping using next-generation DNA sequencing data," *Nature Genetics*, vol. 43, no. 5, pp. 491–498, Apr 2011.

[6] M. Carneiro, "Accelerating variant calling," https://hpc.mssm.edu/files/Carneiro_workshop.pdf, 2013, accessed: 2017-04-01.

[7] Intel, "Replication Recipe for HaplotypeCaller Tool Runtimes," http://www.intel.com/content/dam/www/public/us/en/documents/pdf/10380-2%20Recipe-GATK_021615.pdf, 2016, accessed: 2017-04-01.

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

[9] C. Rauer and N. J. Finamore, "Accelerating Genomics Research with OpenCL and FPGAs," https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01262-accelerating-genomics-research-with-opencl-and-fpgas.pdf, 2016, accessed: 2017-04-01.

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

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

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

[13] J. Peltenburg *et al.*, "Maximizing systolic array efficiency to accelerate the PairHMM Forward Algorithm," *2016 IEEE Int. Conf. Bioinformatics and Biomedicine (BIBM)*, vol. 00, pp. 758–762, 2016.

[14] J. Stuecheli *et al.*, "CAPI: A Coherent Accelerator Processor Interface," *IBM J. Research and Develop.*, vol. 59, no. 1, pp. 7:1–7:7, Jan 2015.

[15] S. S. Banerjee *et al.*, "Efficient and Scalable Workflows for Genomic Analyses," in *Proc. ACM Int. Workshop on Data-Intensive Distributed Computing*, ser. DIDC '16. New York, NY, USA: ACM, 2016, pp. 27–36.

[16] N. G. de Bruijn, "A Combinatorial Problem," *Koninklijke Nederlandsche Akademie Van Wetenschappen*, vol. 49, no. 6, pp. 758–764, Jun 1946.

[17] H. Li, "A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data," *Bioinformatics*, vol. 27, no. 21, pp. 2987–2993, 2011.

[18] J. M. Zook *et al.*, "Extensive sequencing of seven human genomes to characterize benchmark reference materials," *Scientific Data*, vol. 3, p. 160025, Jun 2016.

[19] A. Konopka and M. Crabbe, *Compact Handbook of Computational Biology*. CRC Press, 2004.

[20] C. Lee, C. Grasso, and M. F. Sharlow, "Multiple sequence alignment using partial order graphs," *Bioinformatics*, vol. 18, no. 3, pp. 452–464, Mar 2002.

[21] G. Van Der Aura, "Speed up HaplotypeCaller on IBM POWER8 systems," http://gatkforums.broadinstitute.org/gatk/discussion/4833/speed-up-haplotypecaller-on-ibm-power8-systems, accessed: 2017-03-31.

[22] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Comput. Architecture Letters*, vol. 15, no. 1, pp. 45–49, Jan 2016.

[23] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," Tech. Rep. 8, 1966.

[24] R. J. Lipton and D. Lopresti, "A systolic array for rapid string comparison," in *Proceedings of the Chapel Hill Conference on VLSI*, 1985, pp. 363–376.

[25] A. Madhavan, T. Sherwood, and D. Strukov, "Race logic: A hardware acceleration for dynamic programming algorithms," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 517–528, Jun. 2014.

[26] A. Sirasao *et al.*, "Fpga based opencl acceleration of genome sequencing software," in *Poster presented at Supercomputing 2015*, Austin, TX, Nov 2015. [Online]. Available: http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post269.html

[27] Z. D. Stephens *et al.*, "Big Data: Astronomical or Genomical?" *PLOS Biology*, vol. 13, no. 7, pp. 1–11, 07 2015.