Hardware Acceleration of the Pair Hidden Markov Model Forward Algorithm

John Campbell, Sam Hall, Joseph Nwabueze, Scott Smith
{jbca,samjhall,jnnwa,ssscrazy}@umich.edu
University of Michigan
Ann Arbor, Michigan

ABSTRACT

The Pair-Hidden Markov Model Forward Algorithm takes up the bulk of the computational workload in variant calling applications and is usually bottlenecked by single or multithreaded CPU performance. Several dedicated hardware accelerator architectures have been proposed to exploit the parallelism inherent in the Forward Algorithm. We present one such accelerator that incorporates optimizations to the critical computation path and the overall hardware utilization to achieve performance superior to that of a software implementation.

KEYWORDS

Variant Calling, DNA sequencing, Pair HMM, Forward Algorithm

ACM Reference Format:
John Campbell, Sam Hall, Joseph Nwabueze, Scott Smith. 2020. Hardware Acceleration of the Pair Hidden Markov Model Forward Algorithm. In *Proceedings of ACM Conference (University of Michigan EECS 570).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Bioinformatics is a growing field with a very active area of research. The field of Bioinformatics has many problems requiring complex computations and long run times. One such task is variant calling, in which potential variants of a reference DNA sequence are identified.

One of the most commonly used tools in Bioinformatics is the Genome Analysis Toolkit (GATK) [5]. GATK offers a tool specifically designed for variant calling called HaplotypeCaller. HaplotypeCaller uses Pair Hidden Markov Models to identify related DNA sequences. The GATK tool compares experimental reads to known haplotype reference strands and determines the overall alignment probability between them. To calculate the overall alignment probability, GATK utilizes the Forward Algorithm (FA) for Pair Hidden Markov Models [6].

University of Michigan EECS 570, April 2020, Ann Arbor, MI, USA

Solving the FA is very computationally intensive. According to [4], the vast majority of the computation time of variant calling is spent doing FA calculations. Therefore, it is worth investigating the acceleration of the FA in order to accelerate the process of variant calling. Several groups have proposed hardware accelerators for this algorithm [1, 4, 9]. Others have proposed GPU accelerators [7].

This paper discusses the design of a hardware accelerator for the Pair HMM FA. It is largely based on the work in [4]. The goal of this project was to replicate most of their work and perform analysis of the speedups and hardware utilization improvements enabled by some of their optimizations. We also compare the hardware accelerator to a variety of C++ CPU implementations.

The remainder of this paper is laid out as follows: Section 2 gives a more in-depth background into variant calling, Pair HMMs, and the FA; Section 3 discusses the design of the hardware accelerator; Section 4 discusses the various C++ implementations of the FA as well as our simulation and testing methodology; Section 5 presents and analyzes the results of our experiments; Section 6 explores some of the related work that has been done to accelerate variant calling; Section 7 concludes the report; and Section 8 briefly discusses the contributions made by each of our team members.

2 BACKGROUND

Variant calling is the process by which genomic variations, such as polymorphisms, insertions, or deletions, are identified between a reference read sequence and an experimental read. This can help establish evolutionary relationships between sequences and can aid in understanding differences in susceptibility to diseases, for example. The relationship between two read sequences can be modeled using a Pair Hidden Markov Model [3]. Pair HMMs use statistical inferences to estimate the similarity between two sequences. The Pair HMM FA generates the overall probability of alignment between the reference and experimental reads by examining the probability of specific alignments. The execution of this FA takes up the bulk of the compute time in modern variant calling tools such as GATK's HaplotypeCaller [4]. The recursive form of the algorithm traverses three N x M matrices where N is the length of the reference read and M

Initialization:

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

Recursion:

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

Termination:

$$\begin{cases} Result = f^M(N_h, N_r) + f^I(N_h, N_r) + f^D(N_h, N_r) \end{cases}$$

Figure 1: Recurrence relation for the Pair HMM Forward Algorithm [4]

is the length of the experimental read. Dynamic programming (DP) (in the form of memoization) is used to make this algorithm tractable for longer sequences [9].

The recurrence relation is described in Figure 1. f^M, f^I, and f^D represent the probability matrices on which the algorithm operates. The a_x terms are transition probabilities that are defined by the user at runtime according to the data set on which they are working [3]. *Prior* is either $1 - Q_s$ (where Q_s is the Phred quality score associated with that specific base in the experimental read) or Q_s depending on if bases i and j in the reference and experimental sequences match or not.

In order to compute the final output probability, each matrix must be sequentially populated with values according to the described equations. The proportion of the total workload that the Forward Algorithm comprises, combined with the matrix-based nature of the algorithm itself, thus make it a clear target for hardware accelerators. Specifically, DP can make the FA exhibit a high degree of parallelism, with multiple threads computing different matrices or different parts of each matrix.

It can be seen that each element in each FA matrix is dependent on three neighboring elements. This places a constraint on accelerator architectures as different threads must synchronize on dependent data. Systolic array-based hardware accelerators have been proposed that leverage multiple simultaneously executing processing elements (PEs) to traverse the f^M, f^I, f^D matrices far faster than single-threaded CPU implementations [8]. These PEs are connected through a direct bus or a memory unit in order to quickly share information and satisfy the data dependencies. By shifting the PEs across each matrix in lock step, dependencies can be handled while maintaining a high degree of parallelism.

3 DESIGN

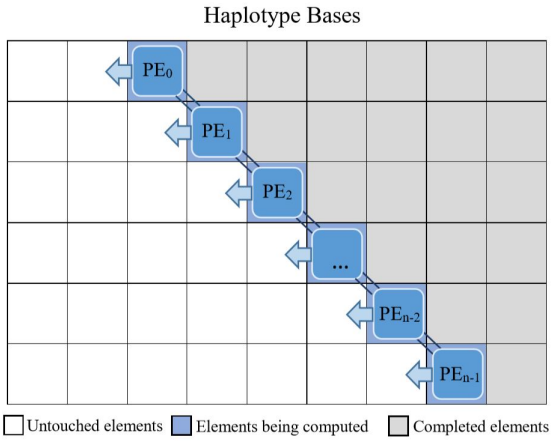

Figure 2: Systolic array corresponding to matrix indices [4]

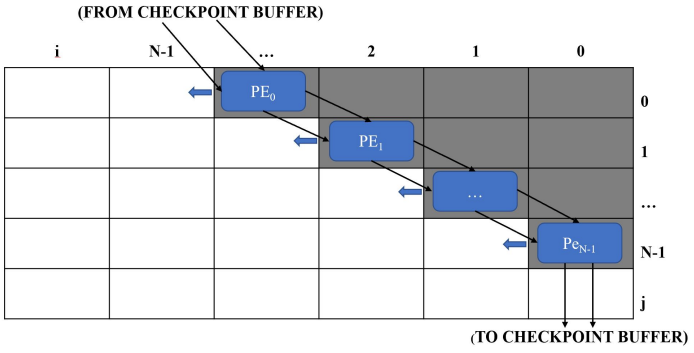

Figure 3: Systolic array showing checkpoint buffer connections

Systolic Array Baseline

The baseline Verilog design is based on the systolic array architecture described in [4, 8]. This architecture consists of an array of PEs, where each PE calculates one index in the DP matrix at a time. Due to the dependencies between indices in the same row/column, the PEs make concurrent calculations across the anti-diagonal of the matrix, forwarding their two previous results ($i-1$ and $i-2$) to the PE in the next row. The reason for this forwarding is made clear by 1. Each PE needs data from indices $(i, j-1)$ and $(i-1, j-1)$. These results will have been computed by the previous PE in the array. Once all active PEs have completed their calculations,

the PEs advance in lockstep to the next $(i+1)$ index in their assigned row. With n PEs, this architecture computes rows $[0, n-1]$ in the F_M, F_I and F_D matrices. If the matrix consists of m rows and k columns, where $m \leq n$, the computation can be completed in one pass. In this case, the final output from the PE calculating the mth row is summed to obtain the final result from Figure 1. However, if $m > n$, the array must make multiple passes, each computing n rows of the matrices until the mth row is computed.

Each PE only forwards its previous two results to the next PE, therefore extra logic must implemented to enable multiple passes through a matrix. To this end, we configured the final PE such that during a given pass it must store the entire row it is computing to a checkpoint buffer. This checkpoint buffer will then be read by the first PE in the next pass.

In order to highlight why this is necessary, consider two input strings are both 100 bases long. In this situation, the PEs need to compute results for a 100x100 matrix. If there are only 10 PEs, the PE array must make 10 passes in order to compute all 100 rows. During the first pass, the 10th (last) PE must store the entire row it computes to the checkpoint (as shown in Figure 3). This is because this row needs to be read by the first PE during the next pass in order to calculate the 11th row. This is due to the $(i, j-1)$ data dependencies.

Critical Path Reduction Optimization

Figure 1 illustrates all of the computations that are made by each PE for each entry in the array. The critical path inside the PE consists of two floating point multiplications and two floating point additions to produce the F_M result as shown in Figure 4. In order to perform this arithmetic, we modified some open source IEEE 754 double precision floating point units (FPUs) from [2]. These FPUs were originally designed for area efficiency. The FPUs originally required dozens of cycles to finish their computations and synthesized with a clock frequency of only 50MHz. We spent considerable effort modifying the FPUs. The modified FPU synthesized with a clock frequency of 100MHz and required fewer clock cycles to complete an addition operation. We verified the accuracy and latency of our modified FPUs using 100,000 input test cases. The new shorter FPU delays result in a critical path of 48 cycles on average.

We were able to further reduce the critical path in half using the critical path reduction described in [4]. This optimization recognizes that the inputs required to compute the critical path (F_m) were available during the previous computation. Using this information, it is possible to compute the first half of the critical path:

$$a_{dm} * (F_i(i-1, j-1) + F_d(i-1, j-1))$$
and,
$$a_{mm} * F_m(i-1, j-1)$$

during the previous computation, therefore reducing the critical path to a single floating point multiplication and addition. This results in a reduction of the critical path latency from an average of 48 cycles to 24 cycles. Since the latency required to compute F_M is now equivalent to the latency to compute F_I and F_D, any further pre-computation would not reduce the critical path latency.

PE Ring Optimization

The PE ring optimization is based on the observation that when the baseline systolic array implementation requires multiple passes to process the memo (when the number of PEs < rows in the memo), no PE is permitted to begin subsequent passes until all PEs finish the current pass. Since the PEs compute across the anti-diagonal, enforcing this restriction results in the final PE finishing the current pass $(n-1) * k$ cycles after the first PE (where n is the number of PEs and k is the number of cycles per computation). This restriction reduces the PE utilization, and therefore the throughput of the baseline implementation since it causes PEs to stall unnecessarily. These stalls are unnecessary because once the first PE finishes the current pass, the data necessary to start computation on the next pass (the first index in the checkpoint buffer written to by the last PE) is available to be read.

The PE ring optimization elides these unnecessary stalls, allowing a PE that has finished the current pass to begin computation on the next pass before all PEs have finished the current pass, as shown in Figure 5. This is permissible as long as the PEs continue to advance in lockstep, since all data dependencies will still be honored. The PE ring allows for greater throughput, since it enables the maximum number of PEs to compute at any given time.

4 METHODOLOGY

We compared two versions of the hardware accelerator to various C++ implementations. The first version of the hardware accelerator is denoted the baseline architecture. The

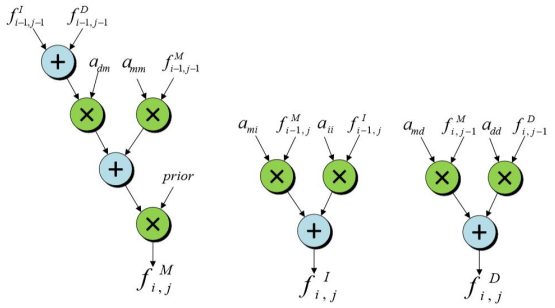

Figure 4: Arithmetic operations inside PEs [4]

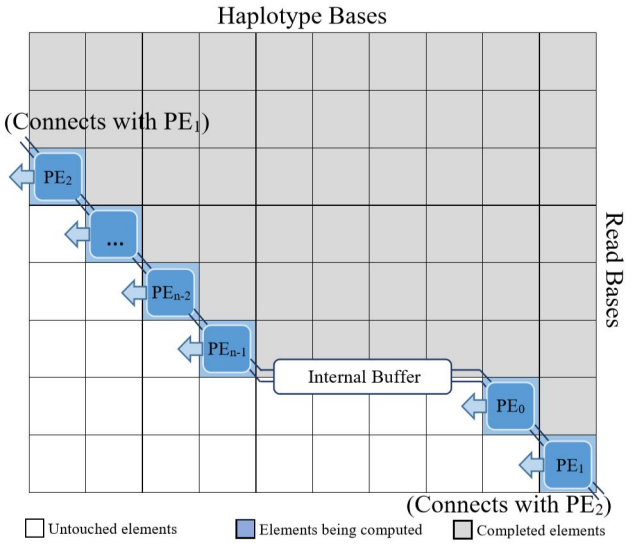

Figure 5: The PE Ring Optimization [4]

baseline architecture does not incorporate the ring buffer or tree-height-reduction critical path optimizations. The second version of the hardware accelerator incorporates both optimizations and is dubbed the optimized architecture. The remainder of this section discusses the various C++ implementations we attempted and provides some analysis into their merits and pitfalls.

C++ Implementation

The GATK library's FA is completely single threaded in its implementation and written in Java, and so in this section we will take a look at trying to speed up the latency and throughput of software variant calling using C++ and multi-threading.

The FA used in variant calling software at its core is the Smith-Waterman string matching algorithm. This is a DP algorithm that uses computations from the last 2 most recent calculations to compute the current cell. It follows the recurrence relation from Figure 1. For a DP memo, this implies that we need three two-dimensional memos - one dimension for each of the two sequences being matched and one memo for each action (insertion, deletion, or match). The data dependencies for a single cell are the cells cells one column before (west) and one row before (north), as well as the cell one column and one row before (northwest).

Multiple Thread Versions

For multi-threading, we took two approaches. The first was to see if it was possible to accelerate the single read performance of the algorithm, and the second was to see if we could accelerate the performance of multiple reads. All multi-thread versions were written using the standard POSIX Thread interface.

Multi Thread Single Read

With multiple threads we can try and compute a diagonal of independent cells in parallel threads. Our main idea is create a set of threads that align to the matrix anti-diagonal and compute multiple cells at once in a lock-step fashion. To synchronize the execution of said threads, we use pthread_barriers to. Otherwise, the threads will communicate with shared data structures. The general format is drawn in Figure 6.

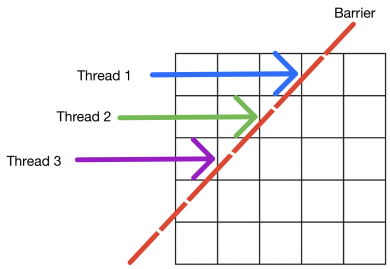

Figure 6: Threads processing though memo in lock step

The simplest form of parallelization is to create the same $i * j$ memo from the single thread implementation and create i threads that are implicitly indexed in a diagonal (similar to the figure's barrier) and move together in lock-step for each cell. Each thread is assigned a row from $[0, i)$ and a starting index equal to $0 - row\_index$. This design is very similar to the PE design we implemented in Verilog. It can also be parameterized in a similar way with a variable number of threads grouped together to pass over the memo multiple times. The main drawback of this method is that there are many cycles where none of the threads will be doing work at the front and end of each horizontal pass. They will instead just immediately check in to the barrier and wait. Another further optimization to this method is to use only two memos as we did before in the single thread. These two memos hold values from the last two diagonals computed. It is also possible to implement a structure similar to the ring optimization but we decided not to do so for performance reasons discussed later on.

Multi Thread Multiple Reads

To try and increase the throughput, we tried to organize multiple threads working on different reads at the same time. This is generally done by having each thread select a subset of the input set of experimental reads and compute their FA against the same reference read. Again, the batches can be grouped amongst some threads to try and balance utilization.

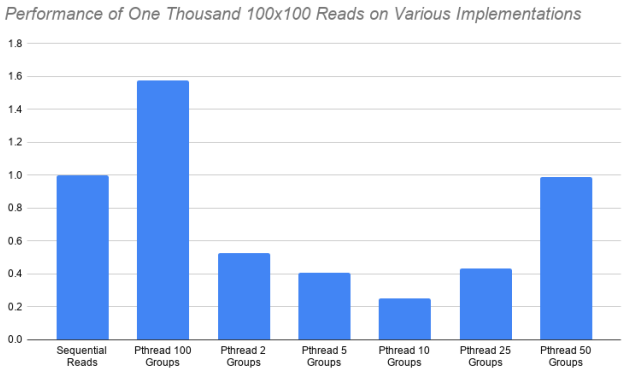

Figure 7: C++ Throughput Optimization Results

C++ Outcomes

The relative performances for our C++ throughput optimizations are found in Figure 7. The C++ programs were tested using a Ryzen 5 3600 at 3.6 GHz (6 cores, 12 threads). Unfortunately, our research in reducing latency with multithreading showed that the overhead from thread synchronization heavily dominated the benefit of computing cells in parallel. Single thread latency was consistently better than the multithreaded, PE style implementation by several orders of magnitude. However, the throughput optimizations proved more fruitful. Grouping the reads into thread batches proved more successful. For one thousand 100x100 reads, we can see that grouping the threads in a size that matches the resource count to avoid context switches and other synchronizations which were likely the cause of slowdowns in the latency optimizations. We'd expect that in GPUs there may be latency improvements as there are more efficient large scale synchronization capabilities. We'd also expect that CPU threads following a single thread form of execution could be further accelerated using vectorized units in a similar fashion to the PE ring. However, there would be significant work with aligning the data to be calculated in the diagonal fashion to get a lock-step-like execution.

5 RESULTS

The optimized version of the hardware accelerator was able to demonstrate improvements in execution time over the CPU implementations and the baseline architecture. The optimized version incorporates both the PE ring and tree-height-reduction optimizations. The topmost plot in Figure 8 shows how the execution time changes with the number of PEs. The optimized version begins to outperform the CPU baseline with just over 20 PEs on long string lengths. The accelerator was able to outperform the CPU implementation without pipelining multiple reads at once. Pipelining multiple reads through the accelerator would only further increase the performance gap and allow for better utilization of the FPU resources. In the future, it would be ideal to finish the pipelining in order to explore this further.

The ring optimization allowed us to gain significant improvements in PE utilization over the basic systolic array implementation. This resulted in a corresponding decrease in overall execution time as the systolic array was able to maintain 100% PE utilization continuously between the initial "fill" period and the ending PE "drain." It is important to note that we noticed non-monotonicity in the graph of PE utilization versus the number of PEs. The utilization of the ring-optimized accelerator is primarily a function of read length and the number of PEs but is also secondarily dependent on the ratio of the two. In general, utilization will locally decrease between PE counts that divide the read length and will locally increase when the number of PEs divides the read length. This can be understood by noting that the fact that, on the last pass when the number of PEs does not divide the read length, PEs may sit idle while the last few rows of the matrices are computed, lowering overall utilization.

When comparing the optimized accelerator to the baseline Verilog implementation, it is clear that the critical compute path and ring optimizations make a significant difference in total execution time. As expected critical path reduction provided a 2x speedup over the non-optimized version across all of our test cases. The remainder of the speedup came from implementing the ring optimization. Gains from this change allowed the accelerator to beat the CPU implementation in total execution time with more than 25 PEs connected.

Area Estimation

In order to obtain area estimates, we synthesized our design using the standard cell library that was available to us. However, the synthesis tools provide a unit-less value for the estimated area consumption (due to standard cell placement and routing). Therefore we believe that the number is either normalized to the square of the minimum feature size in the process technology, or the size of some other standard cell. Since a single inverter synthesizes to approximately 33 area units, we believe this value is normalized based on the process technology's minimum feature size. However, this library does not provide documentation regarding the process technology that the standard cells simulate, and due to some of the build documentation dating back to 2003, we estimate that the performance and overhead of the provided standard cells are based on a process technology somewhere between 90nm-180nm. This results in an area overhead of between $0.47mm^2$ and $1.7mm^2$ per processing element. Therefore, we conservatively estimate that on a $100mm^2$ die, we can fit 50 PEs along with the control logic.

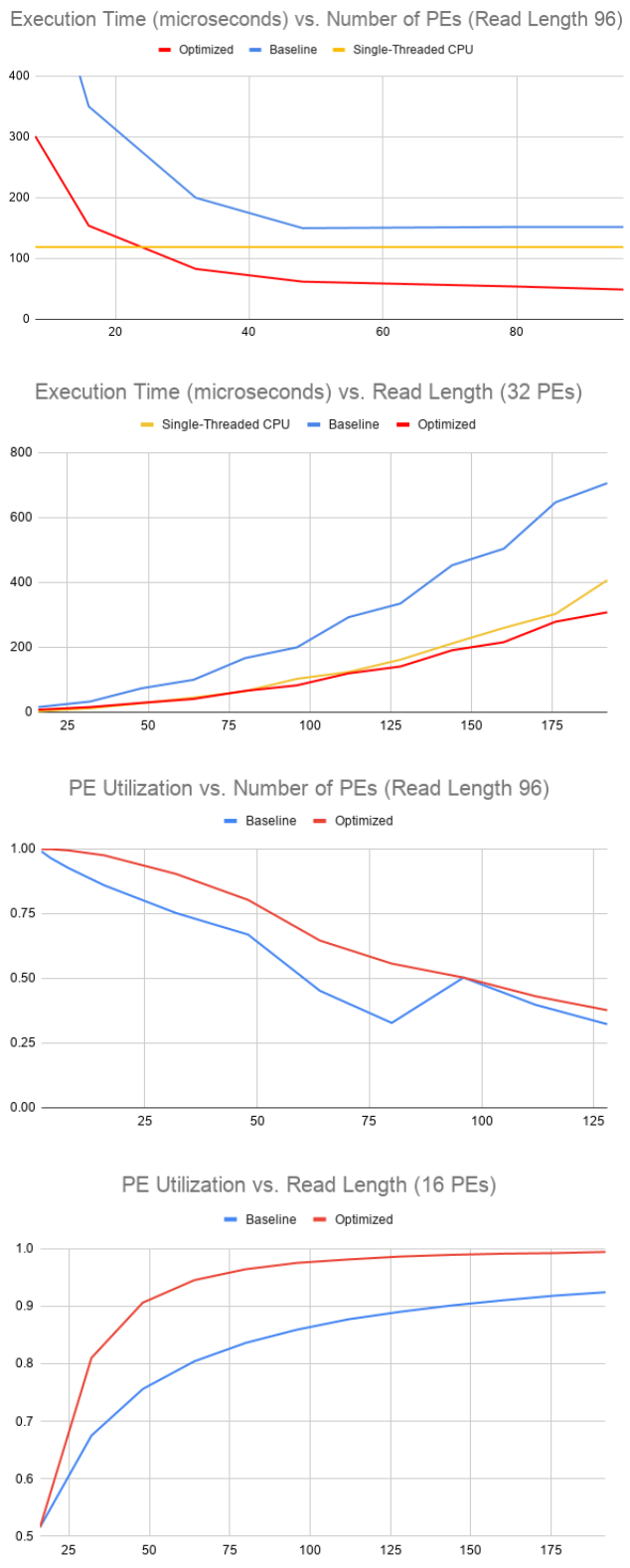

Figure 8: Performance Results for the Baseline and Optimized Architectures

6 RELATED WORK

There has been a significant amount of research done related to the acceleration of Bioinformatics algorithms. This project focused on hardware acceleration of the Pair HMM Forward Algorithm. The work in this paper aimed to replicate much of the work in [4], but also shared many similarities with [8, 9]. GPU acceleration has also been explored in [7]. A non-cooperative PE structure which removes dependencies between the PEs has been proposed by [10].

7 CONCLUSION

The Pair HMM FA is computationally intensive and significantly contributes to the latency of variant calling. It performs slowly on CPU implementations, even multithreaded CPU implementations. Hardware acceleration can lead to significant speedups over software implementations. Even with our two decade old synthesis tools, we were able to show speedup over the software implementation for single reads. In addition, pipelining the system would have further increased the performance of our system for processing multiple reads.

In the future, additional effort should be spent on fully pipelining the implementation. In addition, investigating better synthesis tools and floating point units would also help improve performance and give a better representation of how well the design would perform in hardware.

Exploring GPU accelerations for Pair HMM would also be interesting to analyze. Comparing GPU, CPU, and specialized hardware implementations would provide insight into the advantages and disadvantages of each approach. Engineers and scientists could then use this information to make decisions about how to best perform variant calling for their particular use cases.

8 CONTRIBUTIONS

- John Campbell helped design and debug the baseline systolic array RTL implementation. He also designed the control logic for the PE ring optimization.
- Sam Hall worked on the C++ implementations of the Pair HMM algorithm and developed test case generating tools for debugging the hardware and software. He also worked on the control logic for the PE ring optimization.
- Joseph Nwabueze worked on C++ Implementations of various multi-threaded versions of the Pair HMM Algorithm. He also helped with implementation of pipelined Floating Point Units.
- Scott Smith worked on configuring the open source floating point units to work with the project. He also helped design the design and debug the processing elements.

REFERENCES

[1] Subho S. Banerjee, Mohamed el Hadedy, Ching Y. Tan, Zbigniew T. Kalbarczyk, Steve Lumetta, and Ravishankar K. Iyer. 2017. On accelerating pair-HMM computations in programmable hardware. In *2 2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

[2] Jon Dawson. 2019. Synthesiseable IEEE 754 Floating Point Library in Verilog. https://github.com/dawsonjon/fpu

[3] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. 2002. *Biological Sequence Analysis*. THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE, Cambridge, United Kingdom, Chapter 3–4, 46–99.

[4] Sitao Huang. 2017. *Hardware Acceleration of the Pair HMM Algorithm for DNA Variant Calling*. Master's thesis. University of Illinois Urbana-Champaign, Urbana, Illinois.

[5] Broad Institute. 2020. Genome Analysis Toolkit. Retrieved May 21, 2020 from https://gatk.broadinstitute.org/hc/en-us

[6] Broad Institute. 2020. HaplotypeCaller in a nutshell. Retrieved May 21, 2020 from https://gatk.broadinstitute.org/hc/en-us/articles/360035531412-HaplotypeCaller-in-a-nutshell

[7] Enliang Li. 2019. *Exploration of GPU acceleration for pair-HMM algorithm and its application in the DNA alignment problem*. Master's thesis. University of Illinois Urbana-Champaign, Urbana, Illinois.

[8] Shanshan Ren, Vlad-Mihai Sima, and Zaid Al-Ars. 2015. FPGA Acceleration of the Pair-HMMs Forward Algorithm for DNA Sequence Analysis. In *2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE.

[9] Davide Sampietro, Chiara Crippa, Lorenzo Di Tucci, Emanuele Del Sozzo, and Marco D. Santambrogio. 2018. FPGA-based PairHMM Forward Algorithm for DNA Variant Calling. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, Milan, 1–8.

[10] Pengfei Wang, Yuanwu Lei, and Yong Dou. 2019. Pair-HMM Accelerator Based on Non-cooperative Structure. *IEICE Electronics Express* 16, 15 (2019), 1–6. https://doi.org/10.1587/elex.16.20190402