A Scalable Hardware Accelerator for Mobile DNA Sequencing

Karim Hammad, *Member, IEEE*, Zhongpan Wu, *Member, IEEE*, Ebrahim Ghafar-Zadeh, *Senior Member, IEEE*, and Sebastian Magierowski, *Member, IEEE*

***Abstract*— DNA sequencers are being miniaturized and increasingly targeted toward mobile applications. However, the intense bioinformatic computing needs of sequencers present a challenge for remote use with limited energy supply. This article presents a step toward realizing a low-power and high-speed bioinformatic engine, a hardware-accelerated basecaller, for mobile sequencing applications. The design is targeted for nanopore-based sequencers and is architected to easily scale to the complexity of this sensor. In addition to accelerating the CPU in real time with a custom field-programmable gate array (FPGA) through a high-speed serial link, the proposed framework envisages the challenging memory requirement of high-order nanopore sensors. The framework proposes a memory management scheme, which provisions the memory requirement problem in three dimensions: the basecalling speed, the circuit's area, and power consumption. The implementation results demonstrate a 142× basecalling speed improvement over a 12-core CPU-only reference, as well as significant speedup compared with other existing solutions. Also, an energy efficiency improvement of three orders of magnitude is measured.**

***Index Terms*— Field-programmable gate array (FPGA) acceleration, hidden Markov model (HMM) basecalling, memory design, mobile DNA sequencing, nanopore, peripheral component interconnect express (PCIe), reusable integration framework for FPGA accelerator (RIFFA).**

I. Introduction

DNA sequencing is an important experimental procedure in genomics. Over the decades, advances in this procedure have greatly influenced the cost and scale of genome analysis. This influence has accelerated substantially since 2003, the year that the Human Genome Project [1] was declared complete. Recent studies [2] showed that the cost to sequence a human genome over time is a stark record of just how profound this improvement has been; since 2001, sequencing costs have plummeted by eight orders of magnitude, an average doubling in "performance" every 8.5 months. This impressive progress is the result of numerous improvements made by industrial and academic sectors to improve the DNA sequencing "pipeline" (the set of steps comprising a genome-measurement procedure).

Among the most exciting recent developments in the field is its expansion toward mobile contexts. In particular, palm-sized DNA measurement machines have emerged [3] that weigh less than 100 g. These machines can be injected with "short" and specially prepared DNA samples ("strands"), in a continuous manner and, in response, continually output electronic time-series representations of these input molecules. These output signals are drawn from a parallel array of DNA sensors. From a sensor array spread across an area of roughly 1 cm^2, ideally, the equivalent of one human genome can be measured in about 3.5 h [4], [5]. The emerging nature of this technology suggests that even greater throughput may be achieved as these devices mature. Such technology promises to greatly expand the reach of biomolecular analysis beyond specialized research laboratories.

One barrier to large-scale deployment of such miniature DNA measurement machines is computing. Although a vast amount of data can be drawn from them, converting the electronic outputs to their text-equivalent (i.e. the DNA "base" molecule labels `A`, `C`, `G`, `T`) requires intense bioinformatic computation. Presently, this is accomplished outside the device itself, with off-the-shelf computing elements (CPU and GPU). The power needs of such computers limit the opportunity for sequencing in mobile settings. Hence, this article is motivated by developing a low-power accelerator for such power-sensitive sequencers.

In this article, we describe a hardware-accelerated bioinformatics compute engine of promise for use in mobile sequencing. Our focus is on the "basecalling" phase of the sequencing pipeline, the first major bioinformatic step in the process. Basecalling is a detection step that classifies physical measurements (e.g., such as the aforementioned electronic signals) of DNA strands into their text equivalent. A DNA measurement machine may have thousands or millions of such sensors working in parallel. Billions of DNA strands, each composed of dozens to thousands of bases, are randomly distributed across such arrays. The bioinformatic steps that follow basecalling (outside the scope of this article) correct and assemble the basecalled "reads" into contiguous segments of the test subject's genome, stretching tens of thousands or billions of bases.

Manuscript received May 28, 2020; revised August 29, 2020 and November 13, 2020; accepted November 29, 2020. Date of publication January 5, 2021; date of current version January 28, 2021. This work was supported in part by the Canadian Food Inspection Agency (CFIA), in part by CMC Microsystems, in part by the Natural Sciences and Engineering Research Council (NSERC), and in part by the Southern Ontario Smart Computing Innovation Platform (SOSCIP). *(Corresponding author: Karim Hammad.)*

Karim Hammad is with the Department of Electrical Engineering and Computer Science, York University, Toronto, ON M3J 1P3, Canada, and also with the Arab Academy for Science, Technology and Maritime Transport, Cairo, Egypt (e-mail: khammad@eecs.yorku.ca; khammad@aast.edu).

Zhongpan Wu, Ebrahim Ghafar-Zadeh, and Sebastian Magierowski are with the Department of Electrical Engineering and Computer Science, York University, Toronto, ON M3J 1P3, Canada (e-mail: zhongpan@eecs.yorku.ca; egz@eecs.yorku.ca; magiero@eecs.yorku.ca).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2020.3044527.

Digital Object Identifier 10.1109/TVLSI.2020.3044527

The probabilistic nature of the basecalling problem for nanopore signals (i.e., arising from the additive noise accumulated by the thermal variation of the nanopore sensor and its subsequent signal conditioning circuits) was addressed in the literature by two approaches. First, the hidden Markov model (HMM) has demonstrated a great potential in addressing the stochastic characteristic of the nanopore signal [6]. The authors have studied a small-scale 3-mer solid-state nanopore sensor model and a 64-states HMM basecaller to decode the sensor's electrical signal. The studied HMM method has shown a significant potential for an accurate modeling of the studied nanopore without reflecting upon the computational challenges. Hence, our recent work reported in [7] has offered a field-programmable gate array (FPGA) acceleration framework for the CPU implementation of the HMM approach reported in [6]. The speedup factor for the proposed FPGA-accelerated basecaller in [7] compared to the CPU-only basecaller was 172× with three orders of magnitude improvement in energy efficiency. Another prime example reported in the literature for the application of the HMM approach is the basecaller, denoted as Nanocall, presented in [8]. Nanocall has been fully implemented on a 4-core CPU in C++ employing a 4096-states HMM basecalling scheme (i.e., 6-mer nanopore model). Nanocall has demonstrated a maximum basecalling speed of 6.7 Kbases/s.

Second, the deep learning neural network (NN) approach has been noted to be widely addressing the nanopore basecalling problem. In [9], various NN-based basecallers known as Albacore, Guppy, and Scrappie were comparatively evaluated. These 5-mer basecallers have demonstrated different performances in terms of speed based on the underlying recurrent NN (RNN) architecture. Guppy was reported to be the fastest with 1.5 Mb/s due to hosting an NVIDIA GTX 1080 GPU as an accelerator, whereas Albacore and Scrappie are only CPU-based. Despite the significant potential of the RNN basecallers presented in this study, the computational complexity of the HMM is still noted to be lower than RNN. This essentially raises a sp between both approaches when considering hardware accelerators in a mobile sequencing context. It should be noted that the NN class of the basecaller is out of this article's scope.

In this article, we present a design architecture that partitions an HMM-based basecalling algorithm across a commodity CPU and an accompanying FPGA-based accelerator. Importantly, the design can be scaled with the complexity of the DNA sensor. The design is tuned for nanopore-based sequencing machines, currently the only portable DNA measurement devices available [5], [10]. Within this architecture, we use an efficient memory design to maximize the basecaller's speed/power ratio. The design is realized and its performance is experimentally assessed. To our knowledge, this is the first such basecaller design to be reported in the open literature. Preceding work [7], [11] has begun to address this issue, but for a DNA sensor model that is roughly two orders of magnitude less complex. On the other hand, to the best of our knowledge, the hardware acceleration challenges for the more realistic 4096-states HMM basecaller have not been reported yet in the open literature. Hence, we put this problem as the subject of this article.

The rest of this article is organized as follows. Section II presents the nanopore sequencing methodology and its challenges. The algorithm used by our proposed basecaller and its architectural transformation to match our scalable basecalling acceleration engine design is discussed in Section III. The proposed FPGA architecture for the proposed acceleration engine is described in Section IV. The implementation results are presented in Section V. Section VI concludes this article.

II. Mobile Sequencing Technology

DNA sequencing based on nanopore sensors [5] is an excellent example of the potential to extend molecular measurement to mobile contexts. Although the process by which it is realized is complex, the nanopore sensor concept is simple; it is an opening (or pore) located in an otherwise impermeable support membrane. For DNA sequencing purposes, the diameter of such sensors is about 2 nm, just enough for a single DNA strand to thread through (translocate) at-a-time. The nanopore is surrounded by a complex electrical and fluidic apparatus. Thus, as a strand translocates a step-wise electronic signal is ultimately generated (see Fig. 1). This signal is indicative of the DNA's base molecule make-up. In existing nanopore-based sequencers, hundreds of such measurements can be generated simultaneously from nanopore sensors arrayed across areas as small as 1 cm^2.

Ideally, only four distinct levels, each indicative of one of the four base types would be apparent in the signal. However, the interaction between the DNA molecule and the nanopore is complicated. For example, multiple DNA bases interact with the nanopore at a single time instant essentially "blurring" the measurement. Also, the noise induced by the sensor and its associated apparatus further distorts the measurement. Thus, to basecall any one strand's signal, a sophisticated statistical detector is needed. Many possibilities are available to implement this, and in this work, we focus on sequence detectors employing the hidden Markov model (HMM) formalism.

In the literature, only a few studies have been noted to consider the hardware acceleration problem in the DNA sequencing context in general [12], [13] and in basecalling nanopore signals in particular [9]. It is worth noting that the GPU-accelerated nanopore basecaller reported in [9] is based on an NN solution that is out of the scope of this article. This further supports the motivation of this work.

III. DNA Basecalling Using HMM

HMM-based detectors consider the history of measurement in classifying any particular input. This makes them suitable for situations such as nanopore basecallers where some subset of k bases (a k-mer) is contributing to the measured signal at any one time. By tracking a history of such measurements, an HMM detector can better distinguish which of the DNA's four possible bases was actually responsible for the signal at any one measurement instant. HMM algorithms are used across a variety of applications in bioinformatics with more

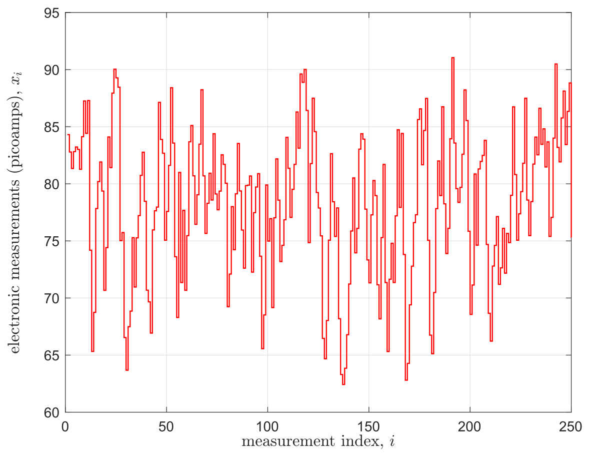

Fig. 1. Example of a nanopore signal output.

details available in [14] and [15]. Its application specifically to nanopore basecalling is given in [8], an approach we employ here as well.

A. Emission and Transitions Models

For our application, the hidden states of the HMM model describe the possible k-mers present inside the nanopore sensor at a given measurement time instant. HMM-based prediction of the pore state is achieved with the help of emission and transition models.

The emission model considers each new measurement x_i (e.g., each new level as shown in Fig. 1, indexed across measurements of unique levels) and computes the likelihood of this measurement given its model of the sensor. A sensor model consists of a set of expected measurement levels, $\{\mu_j\}$, and a corresponding set of expected standard deviations from these levels, $\{\sigma_j\}$, one set for each possible pore state j. For models of sensors with a resolution of k bases (roughly, nanopores that produce signals in response to k bases at-a-time), $j \in \{0, 1, \ldots, 4^k - 1\}$ one for each possible k-mer combination in the nanopore. Assuming that x_i conforms to a Gaussian distribution, this likelihood, for each possible model parameter, can be expressed as

$$\epsilon_j(x_i) = \left(2\pi\sigma_j^2\right)^{-\frac{1}{2}} \exp\left[(x_i - \mu_j)^2/2\sigma_j^2\right]. \tag{1}$$

The number of these computations (i.e., the number of HMM "states"), and hence the complexity of the emission model, scales linearly with j and hence exponentially with the resolution of the detector.

The HMM's transition model governs which sequence of k-mer observations are possible from measurement to measurement. In particular, it specifies the possible ways that a certain pore state j at measurement i can make a transition to another state at the ensuing measurement $i+1$. Naturally, not all transitions are possible. Ideally, only four transitions would be expected from any measurement i to the next measurement at $i+1$ since only one of four bases can enter the pore at any particular time and thus result in only four different possible measurement values. In practice, however, a more complex transition scheme must be accounted for. An example of this is shown in Fig. 2 for one state in a 6-mer HMM. The example considers the transition for the state j associated with the AAGATC 6-mer, but the same general transition pattern would be present for all 4096 states in such an HMM.

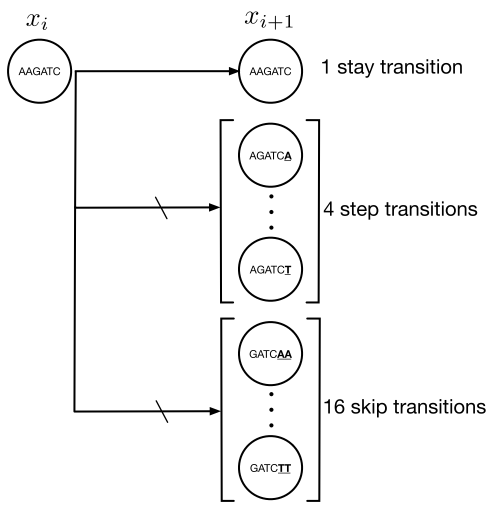

Fig. 2. Example of 21 transitions model for a 6-mer nanopore.

As shown in Fig. 2, rather than just making one of four possible step transitions, 17 other transition possibilities are included, and thus, a total of 21 possible transitions are accounted from present measurement states at i to the following measurement states at $i+1$. These include one stay transition that accounts for the possibility that a previous measurement is mistaken for a new one and 16 skip transitions that account for the possibility that a base was not registered and hence that a new measurement actually contains contributions from the last two bases to enter the nanopore. More discussions on these mechanisms are available in [7] and [8]. The probability of a transition from one state to the next is denoted with $\tau(\text{state at } i, \text{state at } i+1)$. As with μ_j and σ_j, these are nanopore model parameters.

B. Basic Basecalling Algorithm

The HMM basecaller employs a dynamic programming strategy to execute its text labeling. As with the Viterbi algorithm [16] upon which it is based, the basecaller can be viewed as searching for an optimum path across a trellis of $M = 4^k$ states ($j \in \{0, \ldots, M-1\}$) spanning N measurements (x_0 to x_{N-1}). The details of the HMM detection method are described in the pseudocode provided in Algorithm 1.

The basecaller is composed of two main parts. First, the HMM trellis (which utilizes the emission and transition models of the sensor) is used to compute the posterior probabilities (as explained below) for the M states, a computation repeated over N measurements. Second, the traceback procedure, guided by the posterior values, then goes backward over the HMM trellis to find the most likely sequence of states (i.e., $\boldsymbol{\pi}^* = (\pi_i^*)_{i=0}^{N-1}$) and, hence, the most likely sequence of bases (i.e., $\hat{\boldsymbol{a}} = (\hat{a}_i)_{i=0}^{N-1}$) associated with the measurements.

In more detail, first, the Initialize block (lines 1–2) is invoked once at beginning for the first event x_0 of the input sequence. The M emission probabilities for x_0, $\epsilon_j(x_0)$, are

Algorithm 1 HMM Basecalling—CPU-Only Version

```
1: Initialize:
2: $\alpha_0(j) \leftarrow \epsilon_j(x_0)\ \forall\ j \in \{0, \ldots, 4095\}$
3: Iterate:
4: for $i \leftarrow 1, N-1$ do
5:   for $j \leftarrow 0, 4095$ do
6:     $\alpha_i(j) \leftarrow \epsilon_j(x_i) \max_{\nu \in \boldsymbol{\omega}(j)}[\alpha_{i-1}(\nu)\tau(\nu, j)]$
7:     $\beta_i(j) \leftarrow \arg\max_{\nu \in \boldsymbol{\omega}(j)}[\alpha_{i-1}(\nu)\tau(\nu, j)]$
8:   end for
9: end for
10: EndState:
11: $\pi^*_{N-1} \leftarrow \arg\max_j[\alpha_{N-1}(j)]$
12: $\hat{a}_{N-1} \leftarrow \pi^*_{N-1}\ \&\ 6$
13: Traceback:
14: for $i \leftarrow N-1$ to $1$ do
15:   $\pi^*_{i-1} \leftarrow \beta_i(\pi^*_i)$
16:   $\hat{a}_{i-1} \leftarrow \pi^*_{i-1}\ \&\ 6$
17: end for
```

sequentially calculated in a loop based on (1), and the result is stored as the posteriors of x_0 and $\alpha_0(j)$.

Second, the Iterate block constructs the HMM trellis. Constructing the trellis is carried out by two nested loops: the outer loop (lines 4 and 9) traverses the N input events, whereas the inner loop (lines 5–8) traverses the M states for each input event x_i. In other words, for each new event x_i, the Iterate block calculates the M posterior probabilities $\alpha_i(j)$ in line 6. In addition, line 7 computes M $\beta_i(j)$ values, and these are pointers to the trellis state at i most likely to have followed those at $i-1$. The most probable transition is selected from the 21 possible transitions, $\boldsymbol{\omega}(j)$, as shown in Fig. 2. It should be noted that the posteriors calculation in line 6 implies calculating the states' emission probabilities, $\epsilon_j(x_i)$, as in line 2.

Third, the EndState block (lines 10–12) determines the end-state index, π^*_{N-1}, and the end-state bases, $\hat{a}_{N-1}$. The end-state index is essential for the Traceback procedure (i.e., explained in the following paragraph) and acts as the starting point for the optimal path of the nanopore states.

Fourth, the Traceback block (lines 13–17) is the HMM decoder's second computational portion that utilizes the pointers calculated by the HMM trellis in finding the most probable sequence of states, $\boldsymbol{\pi}^*$, over which the nanopore has experienced. Knowing the states sequence then directly helps determining the resulting DNA bases $\hat{\boldsymbol{a}}$ leading to it and, hence, finalizing the basecalling task.

C. Unrolled Basecalling Algorithm

The posterior calculation in Algorithm 1 (lines 6–7) is especially suitable for unrolling, ideally across all M states. For hardware accelerators, this could be directly met with M parallel compute blocks. However, given the complexity of existing nanopore sensors in DNA sequencing applications, such one-step unrolling is not practically feasible. In contemporary cases, nanopore models depend on as many as $M = 4096$ states, the 6-mer case. The complexity of each state calculation coupled with power constraints precludes most current technologies from economically dealing with this in a mobile context.

Algorithm 2 HMM Basecalling–FPGA-Accelerated Version

```
1: Initialize on FPGA:
2: for $s \leftarrow 0, 63$ do
3:   $\alpha^s_0(r) \leftarrow \epsilon^s_r(x_0)\ \forall\ r \in \{0, \ldots, 63\}$
4: end for
5: Iterate on FPGA:
6: for $i \leftarrow 1, N-1$ do
7:   for $s \leftarrow 0, 63$ do
8:     for $r \leftarrow 0, 63$ do
9:       $\alpha^s_i(r) \leftarrow \epsilon^s_r(x_i) \max_{\nu \in \boldsymbol{\omega}^s(r)}[\alpha_{i-1}(\nu)\tau(\nu, r)]$
10:      $\beta^s_i(r) \leftarrow \arg\max_{\nu \in \boldsymbol{\omega}^s(r)}[\alpha_{i-1}(\nu)\tau(\nu, r)]$
11:    end for
12:  end for
13: end for
14: EndState on FPGA:
15: $\pi^*_{N-1} \leftarrow \arg\max_r[\alpha^{s=0}_{N-1}(r)]$
16: for $s \leftarrow 1, 63$ do
17:   for $r \leftarrow 0, 63$ do
18:     if $\max_r[\alpha^s_{N-1}(r)] > \max_r[\alpha^{s-1}_{N-1}(r)]$ then
19:       $\pi^*_{N-1} \leftarrow \arg\max_r[\alpha^s_{N-1}(r)]$
20:     end if
21:   end for
22: end for
23: $\hat{a}_{N-1} \leftarrow \pi^*_{N-1}\ \&\ 6$
24: Traceback on CPU:
25: for $i \leftarrow N-1$ to $1$ do
26:   $\pi^*_{i-1} \leftarrow \beta_i(\pi^*_i)$
27:   $\hat{a}_{i-1} \leftarrow \pi^*_{i-1}\ \&\ 6$
28: end for
```

Therefore, we propose another version of Algorithm 1, which could be efficiently accelerated using an accompanying hardware device such as an FPGA. The modified FPGA-accelerated algorithm is shown in Algorithm 2.

The essence of Algorithm 2 is breaking-up the inner loop (lines 6–7) of Algorithm 1, and essentially the initialization loop in line 2, into smaller size loops. In particular, the 4096 loop interactions are chopped to 64 iterations each of which loops 64 times. In other words, the 4096 states are divided over 64 time *segments*. Each segment s is responsible for computing 64 "adjacent" states, r (i.e., segment 0 contains states 0–63, segment 1 contains states 64–127, …, segment 63 contains states 4032–4095) in a parallel fashion. In summary, each global state index j in Algorithm 1 is mapped to a local state index r, which belongs to specific segment index s in Algorithm 2 using the following formula:

$$j = s \times 64 + r \tag{2}$$

where $s, r \in \{0, \ldots, 63\}$ and $j \in \{0, \ldots, 4095\}$.

The motivation of the loop segmentation is to take advantage of constructing a parallel structure composed of 64 slices to compute the 64 posteriors of a single segment of states at a time on a hardware accelerator such as an FPGA. This potentially overcomes the logic resource limitation of target hardware and the power consumption challenge they would face if intended for a 4096-state design.

The 64-state parallel structure is time-multiplexed over 64 cycles to finalize the posterior computations of the 4096 states for one measurement x_i. Thus, the loops of the *Initialize*, *Iterate*, and *EndState* blocks in Algorithm 1 are divided over 64 segments and sent to the FPGA for accelerated computing. This can be seen in lines 2–4, 7–12, and 16–22 in Algorithm 2, respectively. On the other hand, the loop of the *Traceback* block loop remains unchanged as it is executed on the CPU side. It is worth mentioning that reading/writing the 4096 posteriors from/to the memory on the FPGA accelerator throughout the 64 segments of each event follows a certain memory management scheme that we propose and describe in the next section. That scheme speeds up the computations of lines 9 and 10 in Algorithm 2.

IV. Basecalling Accelerator Architecture

In this section, we present an FPGA basecalling accelerator architecture for decoding 6-mer nanopore signals. The design is guided by the modified HMM basecaller outlined in Algorithm 2. The proposed architecture addresses the tradeoff between parallelizing the significant computational requirement for a 6-mer nanopore signal and the power requirement. To strike that balance, our proposed architecture comes up with a custom memory management scheme on the FPGA side of the basecalling system. One unique aspect, elaborated in the following discussions, of our architecture is that it can seamlessly scale down to deal with 5- and 4-mer nanopore signals if needed.

A. Proposed Basecalling Acceleration Engine

The block diagram of the proposed FPGA architecture for the basecalling acceleration engine is shown in Fig. 3. The double-stroke arrows denote multiple signal channels (the number of channels is indicated above the arrow); the single-stroke arrows denote only one signal. The engine is designed to execute the *Initialize*, *Iterate*, and *EndState* blocks of Algorithm 2. The remaining function, *Traceback*, is left for execution on the CPU itself. Besides being a suitable workload for the CPU, this also helps avoid overloading the FPGA's limited resources which we reserved for the compute-intensive HMM trellis processing kernels.

As mentioned in Section III, the calculations of both the *Initialize* and *Iterate* blocks for the 4096 states are partially parallelized. Also, as commonly done in stochastic detection, we replace the linear probability terms used in lines 3, 9, and 10 of Algorithm 2 with the negative log-likelihood representations. This allows the probability chain rule to be computed using addition rather than multiplication. Consequently, the max[$\cdot$] operations in lines 9, 10, 15, 18, and 19 in Algorithm 2 are replaced with min[$\cdot$], and (1) reduces to

$$-\ln \epsilon_r^s(x_i) = \ln\left(\sigma_r^s\right) + \left(x_i - \mu_r^s\right)^2 \tag{3}$$

where $\epsilon_r^s(x_i)$ is the emission probability of state $r \in \{0, \dots, 63\}$ of segment $s \in \{0, \dots, 63\}$ when observing x_i; μ_r^s and σ_r^s are the level mean and level standard deviation model parameters for state r of segment s, respectively. Thus, the FPGA block for a single-state emission calculation is comprised of a two-input subtractor, a two-input multiplier, and a two-input adder.

Similarly, the logarithmic posterior and pointer calculations in lines 9 and 10, respectively, of Algorithm 2 are computed

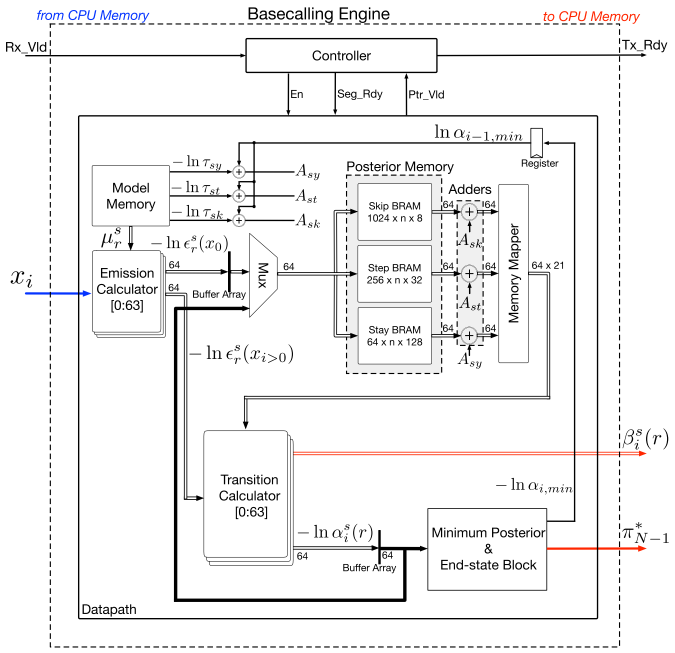

Fig. 3. FPGA architecture of the proposed basecalling acceleration engine.

for each state r of segment s as follows:

$$\ln \alpha_i^s(r) \leftarrow \ln \epsilon_r^s(x_i) + \min_{\nu \in \boldsymbol{\omega}^s(r)} \left[\ln \alpha_{i-1}(\nu) + \ln \tau(\nu, r)\right] \tag{4}$$

$$\beta_i^s(r) \leftarrow \arg\min_{\nu \in \boldsymbol{\omega}^s(r)} \left[\ln \alpha_{i-1}(\nu) + \ln \tau(\nu, r)\right]. \tag{5}$$

The function of each building block for the proposed engine architecture in Fig. 3 is summarized as follows:

1) The *Controller* block maintains the data flow configuration and timing for all other components of the engine. In particular, it signals to enable permissions for each block to indicate the start and end of operation cycles, and it receives valid status signals indicating elapsed calculations for event pointers and the end-state pointer. Another major function for the controller is to ensure that the engine's data path is time-multiplexed 64 times for each input event. The number of segments handled by the *Controller* can be seamlessly programmed, thus allowing the engine to scale across 3 to virtually any nanopore model without the need for any structural changes. For each new input and each segment, the controller also manages the engine's external communication with the CPU.
2) The *Model Memory* is a memory block that stores the HMM's transition and emission model parameters.
3) The *Emission Calculator* block calculates the negative logarithm of the 64 emission probabilities $-\ln \epsilon_r^s(x_i)$ in parallel for each segment.
4) The *Mux* block passes the posterior of x_0 (from *Emission Calculator*) at initialization and then the posterior for $x_{i>0}$ (from *Transition Calculator* for all ensuing calculations). In each case, the *Mux* output is sent to the *Transition Calculator* via the *Posterior Memory* as explained in the following section.

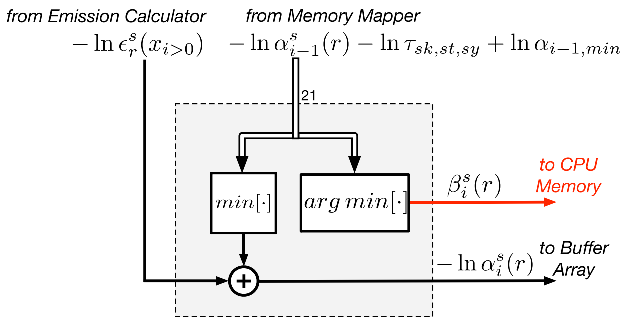

Fig. 4. Architecture of a single slice from the transition calculator.

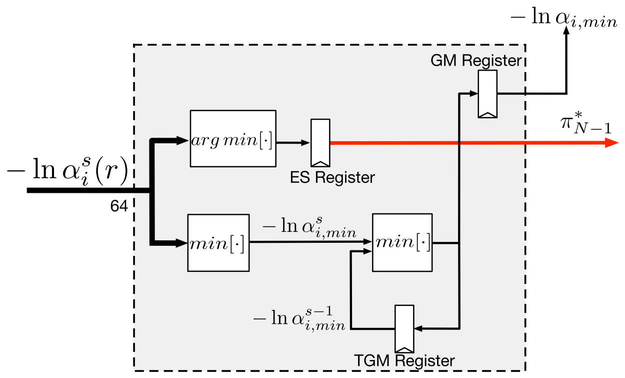

Fig. 5. Architecture of the minimum posterior and end-state block.

5) The *Posterior Memory* is the engine's main memory block that is used for sequentially storing the 4096 posteriors of input x_i across its 64 segments and retrieving the stored posteriors again for the following input x_{i+1} according to a custom memory control and addressing scheme. For this purpose, the memory is composed of three-block random access memory (BRAM) subblocks to address the three transitions model explained in Fig. 2. The details of the BRAM subblocks sizes, arrangement and read/write addressing, and enable schemes are explained in the following section.
6) The *Adders* layer after the *Posterior Memory* is responsible for two functions. First, adding the transition probabilities to the previous event posteriors which corresponds to the addition operation in the arguments of the $\min[\cdot]$ and $\arg\min[\cdot]$ in (4) and (5), respectively. Second, subtracting the global minimum posterior probability (among the 4096 states) of the previous event from the BRAM output to avoid the numerical overflow possibly accrued over a long run of inputs.
7) The *Memory Mapper* block links each state with the 21 possible states that may transition between inputs $i-1$ and i, as shown in Fig. 2. More specifically, it is a crossbar switch that allows any state r in segment s to draw preceding posteriors α_{i-1} from the set $\boldsymbol{\omega}^s(r)$ of its possible 21 preceding states. The delivery of appropriate values through this crossbar is ultimately guaranteed by the BRAM design as explained in the following subsection. In other words, the memory mapper has 64 output lines, each of which carries 21 numbers (i.e., previous event posteriors). Each line goes to a single-state slice within the *Transition Calculator* block, which corresponds to the line's carried posteriors according to the transition model shown in Fig. 2. It is worth mentioning that the increased data width of the memory mapper's output compared to its input pertains to the repetitive characteristic of the transition model. In particular, each of the 16 consecutive HMM states has the same 16 skip transitions, whereas each of the four consecutive states has the same four-step transitions. These repetitive transition patterns cause multiplying the 64 posteriors loaded from the skip and step BRAMs (within the *Posterior Memory* block) by 16 and 4, respectively, at the output of the memory mapper.
8) The *Transition Calculator* block is used to simultaneously compute the 64 posteriors, $\ln \alpha_i^s(r)$, in (4) and pointers, $\beta_i^s(r)$ in (5), across 64 parallel "slices." As usual for our design, these 64 computations are done one segment at-a-time for models that consist of more than 64 states. The structure of each slice is shown in Fig. 4. The slice is composed of a $\min[\cdot]$ function, $\arg\min[\cdot]$ function, and two-input adder. The $\min[\cdot]$ and $\arg\min[\cdot]$ functions are those adopted in (4) and (5), respectively. The adder finalizes the calculation of state r posterior probability in segment s, $\ln \alpha_i^s(r)$, by adding the state's emission probability, $\ln \epsilon_r^s(x_i)$, according to (4).
9) The *Minimum Posterior and End-state Block* has two functions, as shown in Fig. 5. The first is to find the global minimum posterior probability, $-\ln \alpha_{i,\min}$, for each x_i (i.e., stored in the *GM* (global minimum) *Register*) and use it in normalizing the BRAM readout for the following x_{i+1} as discussed earlier. The global minimum posterior probability for each event is searched among the 64 segments using two $\min[\cdot]$ functions. The first $\min[\cdot]$ function finds the local minimum posterior of each segment (among the 64 posteriors at the *Transition Calculator's* output port). The second $\min[\cdot]$ function keeps a running track of the global minimum and stores it in the *TGM* (temporary global minimum) *Register*. The second function of the block is to find the end-state index (i.e., π_{N-1}^*) for the last event as in line 19 of Algorithm 2. The global end-state index for x_{N-1} is found using the $\arg\min[\cdot]$ function and placed in the *ES* (End state) *Register* upon finding the global minimum posterior for x_{N-1}.

The basecalling engine starts off by receiving x_0 from the CPU's main memory and enters the *Initialize* phase of its computation. Over 64 segments, it computes the 4096 emissions of x_0, $-\ln \epsilon_r^s(x_0)$. These results are stored, for each segment s, in the *Posterior Memory* using the 2-to-1 *Mux*, as shown in Fig. 3. This operation (as well as other operations) is controlled by the engine's *Controller* block. In particular, at this phase, the *Controller* only enables the *Emission Calculator* block while disabling all other blocks of the engine. It also controls the read/write mechanisms for the three BRAMs during each segment as explained in the following subsection.

For subsequent inputs x_i, the engine works as follows. For each segment s, the *Emission Calculator* and the *Memory Mapper* feed the *Transition Calculator*. The first provides the logarithm of the current segment's emission probabilities, whereas the second provides the appropriate preceding event (i.e., x_{i-1}) posterior probabilities that correspond to the current segment states. The preceding posteriors come from the *Memory Mapper* after being added to the transition probabilities and normalized using the minimum posterior ($\alpha_{i-1,min}$) of the previous event x_{i-1}. The *Transition Calculator* then calculates the current segment posteriors and pointers according to (4) and (5), respectively. The segment pointers $\beta_i^s(r)$ are sent to the CPU through the peripheral component interconnect express (PCIe) interface (as discussed in the following subsection), whereas the segment posteriors are stored in the three BRAMs and sent to the normalization stage. The whole process is repeated in the same manner for all states' segments until reaching the last segment, $s = 63$, of x_i.

Similar to the sequential calculation of pointers and posteriors across the event's segments, the normalization stage keeps updating the *TGM Register* with the segment's local minimum posterior, $-\ln \alpha_{i,\min}^s$, every new segment. This is done until finding the global minimum posterior (among the 4096 states), $-\ln \alpha_{i,\min}$, at the last segment (i.e., $s = 63$). The global minimum posterior is then stored in the *GM Register* to be used for normalizing the BRAMs readout in the following event x_{i+1}. Once the engine reaches the last event ($i = N-1$), the end state index (π_{N-1}^*) is found (by the arg min[$\cdot$] function) concurrently while searching for the global minimum posterior among the succession of segments. The end state index is sent to the CPU's main memory with the last segment pointers of x_{N-1} (i.e., equivalent to line 19 of Algorithm 2). The stream-line communication between the CPU and FPGA is discussed at the end of this section.

B. Engine's Memory Design

The main purpose of the *Posterior Memory* is to efficiently store the 4096 posteriors of each trellis state at input i and successively retrieve them for update at input $i+1$. As introduced above, this is accomplished for 64 states—one segment—at-a-time. The reading and writing needed to accomplish this are managed by the engine's *Controller* block.

The *Posterior Memory* is composed of three BRAM blocks: one for storing/retrieving the posteriors of the *Skip* states, one for the *Step* states, and one for the *Stay* states of each segment. In particular, when processing the posteriors of x_i, the *Skip* BRAM block first retrieves the 64 skip posteriors of x_{i-1} that correspond to each state's segment of x_i. Meanwhile, the *Skip* BRAM block stores the calculated 64 posteriors of each segment for event x_i. The BRAM block is specially arranged to ensure that both the reading and writing processes are executed in one clock cycle. The same concept applies to the *Step* and *Stay* BRAM blocks.

The arrangement of the *Skip* BRAM block is shown in Fig. 6. The number provided inside each memory location corresponds to the state index of the posterior probability that should be stored in that location (e.g., 0 corresponds to

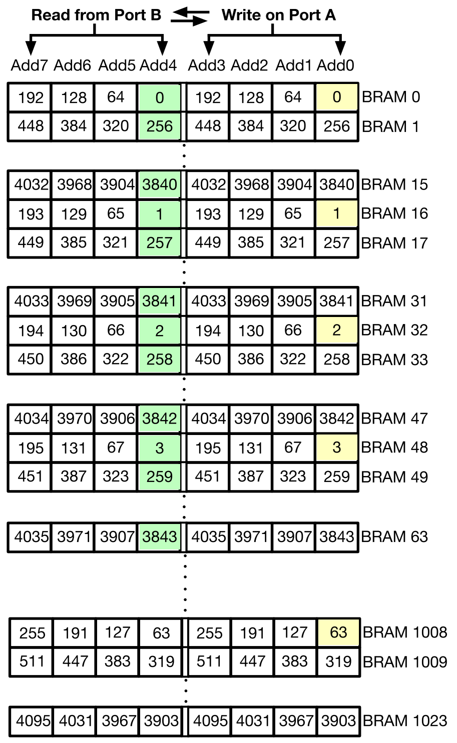
Read from Port B
Write on Port A
Add7 Add6 Add5 Add4 Add3 Add2 Add1 Add0
192 128 64 0 192 128 64 0 BRAM 0
448 384 320 256 448 384 320 256 BRAM 1
4032 3968 3904 3840 4032 3968 3904 3840 BRAM 15
193 129 65 1 193 129 65 1 BRAM 16
449 385 321 257 449 385 321 257 BRAM 17
4033 3969 3905 3841 4033 3969 3905 3841 BRAM 31
194 130 66 2 194 130 66 2 BRAM 32
450 386 322 258 450 386 322 258 BRAM 33
4034 3970 3906 3842 4034 3970 3906 3842 BRAM 47
195 131 67 3 195 131 67 3 BRAM 48
451 387 323 259 451 387 323 259 BRAM 49
4035 3971 3907 3843 4035 3971 3907 3843 BRAM 63
255 191 127 63 255 191 127 63 BRAM 1008
511 447 383 319 511 447 383 319 BRAM 1009
4095 4031 3967 3903 4095 4031 3967 3903 BRAM 1023

Fig. 6. Arrangement of the dual-port Skip BRAM block for the 4096-states HMM.

$\alpha_i(j = 0)$, 256 corresponds to $\alpha_i(j = 256)$, and so on). The memory block has two independent ports (i.e., port A for writing and port B for reading). The block contains 1024 dual-port BRAMs, each of which has a width of n and depth of 8. The width is discussed in the next section with respect to the design's accuracy and the FPGA power consumption budget. The eight locations of each BRAM are used interchangeably for reading and writing with each new event. In particular, for x_i, the locations with the address range 0–3 are used for writing x_i data, whereas the address range 4–7 is used for reading x_{i-1} data. For the following x_{i+1}, the reading address range becomes 0–3, whereas the writing address becomes 4–7. The writing and reading addresses keep alternating in such a ping-pong fashion [17] with the succession of events to avoid memory collisions (i.e., reading and writing from/to the same memory location).

The arrangement of the 64 posteriors of each segment inside the BRAM block follows the indexing scheme highlighted in Fig. 6. In particular, the index set of the enabled *Skip* BRAMs for reading the previous event's 64 *Skip*-posteriors corresponding to segment s, where $s \in \{0, 1, 2, 3, \ldots, 63\}$, of the current event is

$$\phi_{sk}^{rd}(s) = \{(s\%16) * 64 : 63 + (s\%16) * 64\}. \tag{6}$$

The *read* address of the selected *Skip* BRAMs in each segment s is

$$\psi_{sk}^{rd}(s) = \begin{cases} \lfloor s/16 \rfloor + 4, & i\%2 = 0 \\ \lfloor s/16 \rfloor, & i\%2 = 1. \end{cases} \tag{7}$$

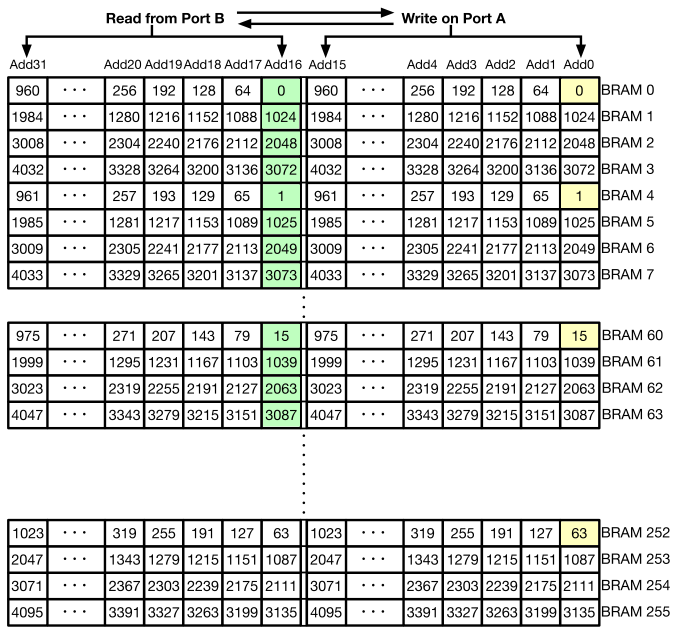
Read from Port B
Write on Port A
Add31 Add20 Add19 Add18 Add17 Add16 Add15 Add4 Add3 Add2 Add1 Add0
960 ··· 256 192 128 64 0 960 ··· 256 192 128 64 0 BRAM 0
1984 ··· 1280 1216 1152 1088 1024 1984 ··· 1280 1216 1152 1088 1024 BRAM 1
3008 ··· 2304 2240 2176 2112 2048 3008 ··· 2304 2240 2176 2112 2048 BRAM 2
4032 ··· 3328 3264 3200 3136 3072 4032 ··· 3328 3264 3200 3136 3072 BRAM 3
961 ··· 257 193 129 65 1 961 ··· 257 193 129 65 1 BRAM 4
1985 ··· 1281 1217 1153 1089 1025 1985 ··· 1281 1217 1153 1089 1025 BRAM 5
3009 ··· 2305 2241 2177 2113 2049 3009 ··· 2305 2241 2177 2113 2049 BRAM 6
4033 ··· 3329 3265 3201 3137 3073 4033 ··· 3329 3265 3201 3137 3073 BRAM 7
975 ··· 271 207 143 79 15 975 ··· 271 207 143 79 15 BRAM 60
1999 ··· 1295 1231 1167 1103 1039 1999 ··· 1295 1231 1167 1103 1039 BRAM 61
3023 ··· 2319 2255 2191 2127 2063 3023 ··· 2319 2255 2191 2127 2063 BRAM 62
4047 ··· 3343 3279 3215 3151 3087 4047 ··· 3343 3279 3215 3151 3087 BRAM 63
1023 ··· 319 255 191 127 63 1023 ··· 319 255 191 127 63 BRAM 252
2047 ··· 1343 1279 1215 1151 1087 2047 ··· 1343 1279 1215 1151 1087 BRAM 253
3071 ··· 2367 2303 2239 2175 2111 3071 ··· 2367 2303 2239 2175 2111 BRAM 254
4095 ··· 3391 3327 3263 3199 3135 4095 ··· 3391 3327 3263 3199 3135 BRAM 255

Fig. 7. Arrangement of the dual-port step BRAM block for the 4096-states HMM.

On the other hand, the index set of the enabled *Skip* BRAMs for writing the 64 posteriors of each segment s for the current event is

$$\phi_{sk}^{wr}(s) = \{\lfloor s/4 \rfloor : 16 : 1008 + \lfloor s/4 \rfloor\}. \tag{8}$$

The write address of the selected *Skip* BRAMs in each segment s is

$$\psi_{sk}^{wr}(s) = \begin{cases} s\%4, & i\%2 = 0 \\ s\%4 + 4, & i\%2 = 1. \end{cases} \tag{9}$$

For instance, consider any even input index (i.e., $i = 2, 4, 6, \ldots$). For segment 0, the *Skip* BRAM block readout comes from address 4 of BRAMs 0–63 and the *Transition Calculator* output posteriors (with indices 0–63) are written to address 0 of BRAMs 0, 16, 32, 48, ..., 1008. For segment 1, the readout locations are from address 4 of BRAMs 64–127, whereas the write locations are address 1 of BRAMs 0, 16, 32, 48, ..., 1008. It should be noted that for x_0, the BRAMs are used only during the writing cycle according to (8) and (9) since no posteriors for a preceding event are stored in the BRAMs to be read. The same case applies for the *Step* and *Stay* BRAMs, which is explained in the following.

The arrangement of the *Step* BRAM block is shown in Fig. 7. The block contains 256 dual-port BRAMs each of which has a width of n and a depth of 32. Similar to *Skip* BRAMs, the 32 locations of the 256 *Step* BRAMs are half-swapped across the succession of x_i for reading and writing the previous and current inputs' posteriors, respectively. The index set of the enabled *Step* BRAMs for reading the previous inputs's 64 *Step*-posteriors corresponding to segment s of the current input is

$$\phi_{st}^{rd}(s) = \{(s\%4) * 64 : 63 + (s\%4) * 64\}. \tag{10}$$

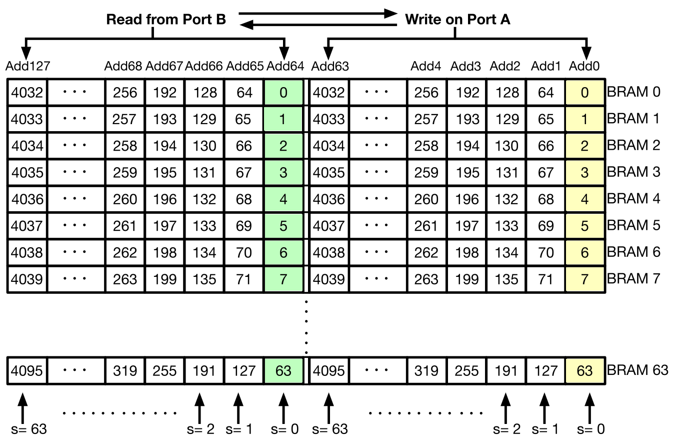
Read from Port B
Write on Port A
Add127 Add68 Add67 Add66 Add65 Add64 Add63 Add4 Add3 Add2 Add1 Add0
4032 ··· 256 192 128 64 0 4032 ··· 256 192 128 64 0 BRAM 0
4033 ··· 257 193 129 65 1 4033 ··· 257 193 129 65 1 BRAM 1
4034 ··· 258 194 130 66 2 4034 ··· 258 194 130 66 2 BRAM 2
4035 ··· 259 195 131 67 3 4035 ··· 259 195 131 67 3 BRAM 3
4036 ··· 260 196 132 68 4 4036 ··· 260 196 132 68 4 BRAM 4
4037 ··· 261 197 133 69 5 4037 ··· 261 197 133 69 5 BRAM 5
4038 ··· 262 198 134 70 6 4038 ··· 262 198 134 70 6 BRAM 6
4039 ··· 263 199 135 71 7 4039 ··· 263 199 135 71 7 BRAM 7
4095 ··· 319 255 191 127 63 4095 ··· 319 255 191 127 63 BRAM 63
s= 63 s= 2 s= 1 s= 0 s= 63 s= 2 s= 1 s= 0

Fig. 8. Arrangement of the dual-port Stay BRAM block for the 4096-states HMM.

The read address of the selected *Step* BRAMs in each segment s is

$$\psi_{st}^{rd}(s) = \begin{cases} \lfloor s/4 \rfloor + 16, & i\%2 = 0 \\ \lfloor s/4 \rfloor, & i\%2 = 1. \end{cases} \tag{11}$$

On the other hand, the index set of the enabled *Step* BRAMs for writing the 64 posteriors of each segment s for the current input is

$$\phi_{st}^{wr}(s) = \{\lfloor s/16 \rfloor : 4 : 252 + \lfloor s/16 \rfloor\}. \tag{12}$$

The write address of the selected *Step* BRAMs in each segment s is

$$\psi_{st}^{wr}(s) = \begin{cases} s\%16, & i\%2 = 0 \\ s\%16 + 16, & i\%2 = 1. \end{cases} \tag{13}$$

The arrangement of the *Stay* BRAM block is shown in Fig. 8. The block contains 64 dual-port BRAMs, each of which has a width of n and a depth of 128. Similar to the *Skip* and *Step* BRAM blocks explained previously, the 128 locations are split into two 64 deep memory sections for simultaneous reading and writing operations for each input's segment and used in an alternating fashion across inputs. The *Stay* BRAMs indexing and the read and write memory addressing in each event segment are simple compared to that for the *Skip* and *Step* BRAM blocks. The index set of the enabled *Stay* BRAMs for reading the previous event's 64 *Stay*-posteriors corresponding to segment s of the current event is

$$\phi_{sy}^{rd}(s) = \{0 : 63\}. \tag{14}$$

The read address of the selected *Stay* BRAMs in each segment s is

$$\psi_{sy}^{rd}(s) = \begin{cases} s + 64, & i\%2 = 0 \\ s, & i\%2 = 1 \end{cases}. \tag{15}$$

On the other hand, the index set of the enabled *Stay* BRAMs for writing the 64 posteriors of each segment s for the current input is

$$\phi_{sy}^{wr}(s) = \{0 : 63\}. \tag{16}$$

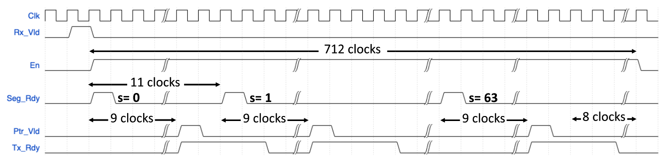

Fig. 9. Timing diagram for the proposed acceleration engine while processing single event.

The write address of the selected *Stay* BRAMs in each segment s is

$$\psi_{sy}^{wr}(s) = \begin{cases} s, & i\%2 = 0 \\ s + 64, & i\%2 = 1 \end{cases}. \tag{17}$$

C. CPU-Accelerator Communication

This section focuses on the physical and logical links between the proposed accelerated basecaller explained in Algorithm 2 and the proposed FPGA acceleration engine in Fig. 3. In particular, it describes the physical interface and the communication protocol utilized by the CPU to offload the computations of the *Initialize* (lines 1–4), *Iterate* (lines 5–13), and *EndState* (lines 14–23) blocks of Algorithm 2 to our FPGA-engine (i.e., described in Fig. 3) in a real-time streaming fashion.

Our acceleration engine includes the Reusable Integration Framework for FPGA Accelerators (RIFFA) IP [18]. RIFFA is an open-source communication architecture, which allows real-time exchanging of data between the user's FPGA IP core(s) and the CPU's main memory via PCIe. To establish its logical channel, RIFFA has a collection of software libraries on the CPU-side and IP cores on the FPGA-side. For each segment's set of 64 pointers, $\beta_i^s(r)$, and the *end-state* index of the last event, π_{N-1}^*, computed by the accelerator, the RIFFA IP's Tx module generates the PCIe packets, which carries the calculated data. The flow of these packets to/from the FPGA is controlled using two sets of handshaking signals [18] one of which controls the data flow in the CPU-to-FPGA direction and the other for the FPGA-to-CPU direction. The hardware interfacing between our acceleration engine and the handshaking protocol is carried out by our engine's *Controller* block as explained in the following paragraph.

Taking a closer look at the processing cycle of a single event x_i sent from the CPU, the picture is shown in Fig. 9. To avoid confusing the reader, it should be noted that the `Clk` waveform shown in Fig. 9 does not represent the exact number of cycles spared by our accelerator to fit the column width. Instead, the exact number of clock cycles for most waveforms has been explicitly indicated in the figure. Each segment requires 11 clock cycles to calculate the corresponding 64 pointers and 64 posteriors and to store the calculated posteriors in the *Posterior Memory*. In particular, nine clock cycles are consumed to calculate the pointers, one clock cycle to calculate the posteriors, and one clock cycle to store the posteriors in the three BRAM subblocks explained in the previous section.

With the arrival of a new 128-bit PCIe packet (which contains four measured events) from the CPU, the RIFFA driver signals `Rx_Vld` to our BC engine. The engine's *Controller* arranges processing the four encapsulated events (each of which is 32 bits) within the arrived packet in sequence of one event at a time. In other words, the *Controller* globally enables the engine's data path block shown in Fig. 3 using `En` for the time required by each event (i.e., 712 clock cycles) in response to `Rx_Vld`. For simplicity, the detailed signaling scheme of each individual block within the engine's data path is not included in this discussion.

During the processing cycle of each event, the *Controller* organizes the 64-segment calculations using the `Seg_Rdy` pulse. The calculation of the 64 pointers of each segment is then announced by the data path block using the `Ptr_Vld` signal. Upon receiving `Ptr_Vld`, the *Controller* asserts the `Tx_Rdy` signal for the Tx channel to allow sending the segment's calculated 64 pointers to the CPU in four consecutive clock cycles. It is worth noting that the eight clock cycles shown in Fig. 9 at the end of the event's processing cycle are needed for carrying out the minimum posterior calculation locally for the last segment (i.e., $s = 63$) and globally across the 64 segments. It could also be inferred from the timing waveform in Fig. 9 that the same eight clock cycles (for the segments from $s = 0$ to $s = 62$) for a specific segment are being overlapped with the 11 clock cycles for the following segment. The overlapped minimum posterior calculation for a certain segment with the core processing cycles of the following segment is intended to reduce the event's processing latency and, hence, the overall basecaller's speed and power efficiency.

D. Accelerator's Scalability

The previous sections have focused on the extreme 4096-states HMM case of our accelerator's architecture, which addresses the 6-mer nanopore signal. However, this does not undermine the scalability of our proposed accelerator architecture in Fig. 3 to implement HMM-basecaller for lower resolution nanopore signals. In particular, if the required number of HMM states is reduced by $4\times$ to 1024-states (i.e., the 5-mer nanopore case), the accelerator's hardware architecture is adapted as follows.

1) The accelerator's datapath runs 16 segments (i.e., 1024/64) for each input event instead of 64 segments in case of 4096-states HMM. The time-mulitplexing period of the accelerator's datapath is seamlessly modified by its *Controller* block by setting the `En` signal for 184 clock cycles corresponding to 16 segments (i.e., $16 \times 11 + 8$), as previously shown in Fig. 9 for the 64 segments case.
2) The storage of the accelerator's *Model Memory* block is modified by the 1024-HMM model parameters.
3) The three BRAM blocks constituting the accelerator's *Posterior Memory* block are downsized (compared to the 4096-states case) to accommodate 1024 states' posteriors in each event. In particular, the depth for three memory blocks is reduced by a factor of 4 while keeping the number of BRAMs within each, as shown Figs. 6–8. Hence, the depth of the *Skip* BRAM block is set to 2

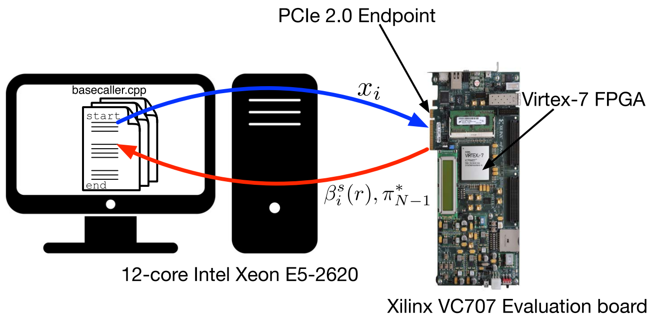
PCIe 2.0 Endpoint
basecaller.cpp
start
end
x_i
Virtex-7 FPGA
$\beta_i^s(r), \pi^*_{N-1}$
12-core Intel Xeon E5-2620
Xilinx VC707 Evaluation board

Fig. 10. Proposed basecaller hardware test setup.

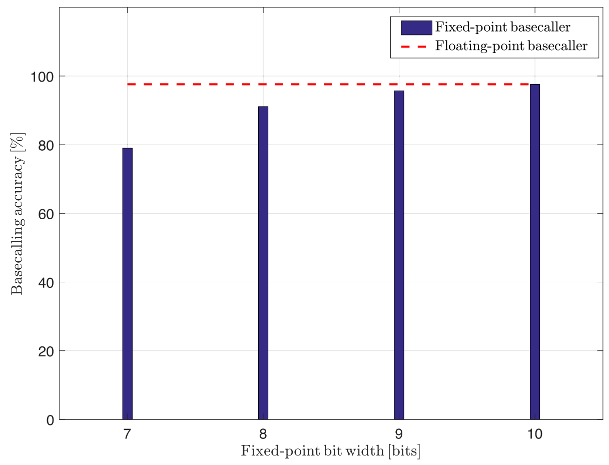

Fig. 11. Simulated fixed-point basecaller accuracy for different bit widths.

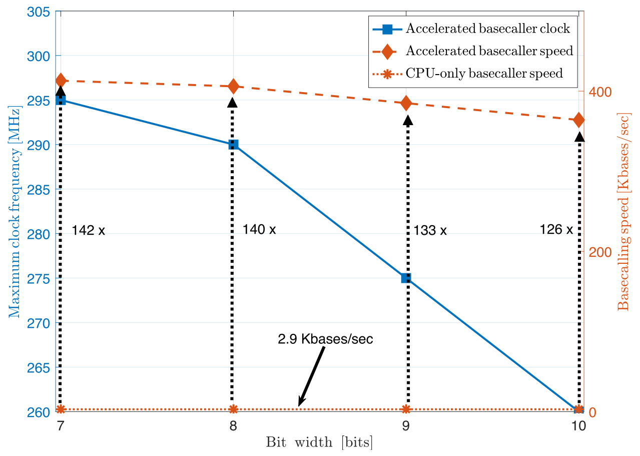

Fig. 12. Accelerator's maximum clock frequency and the measured basecalling speed for different bit widths.

(i.e., one address location for the write port and one address location for the read port), whereas the depth for the *Step* and *Stay* BRAM blocks is set to 8 and 32, respectively.

4) Equations (6)–(17), which govern the reading and writing operation for the three BRAM blocks, are modified to match the aforementioned new depths within the segment range $s \in \{0, 1, 2, 3, \ldots, 15\}$.

V. Experimental Evaluation

A. System Setup

In this section, an implementation of the proposed basecalling acceleration engine in Fig. 3 is evaluated. The implementation is realized on a Virtex-7 FPGA device (i.e., XC7VX485T-2FFG1761C) hosted on a Xilinx VC707 evaluation board, as shown in Fig. 10. The FPGA device contains 75 900 slices, each of which has four look-up tables (LUTs) and eight flip-flops (FFs). The engine accelerates a 12-core Intel Xeon E5-2620 v3 clocked at 2.4 GHz with 32 GB of RAM. A sketch of this setup is shown in Fig. 10.

B. Test Results For 4096-States HMM Accelerator

The acceleration engine studied in Fig. 3 is implemented in a fixed-point format on the target FPGA device while provisioning the extreme computational case for 6-mer nanopore signal. The simulation results shown in Fig. 11 demonstrate how the choice of fixed-point resolution compares to a floating-point implementation. With 10-bits resolution, the fixed-point calculation is able to attain a basecalling accuracy of 97.6%, the percentage of bases that are called correctly; this is on par with floating-point computations. Predictably, lower resolution basecaller accuracy suffers, 95.7%, 91.1%, and 79% accuracies are achieved at 9, 8, and 7 bits, respectively. Nonetheless, for certain applications, these may still prove as useful approximations and we continue to consider these in our hardware evaluations below. It should be noted that, for simplicity, the simulated basecalling accuracy shown in Fig. 11 is based on an emulated input sequencing data based on the 6-mer nanopore model reported in [8].

The results in Fig. 12 illustrate the maximum attainable clock frequency and basecalling speed measured for the CPU-only and CPU-FPGA-accelerated basecaller. The results emphasize the substantial effect of the FPGA design's bit width on the maximum achievable clock frequency and, hence, the basecalling speed of our acceleration engine. First, looking at the 10-bits case that matches the floating-point accuracy, our accelerated basecaller achieved a basecalling speed that is 126× faster than the CPU-only basecaller. In particular, with a 10-bits resolution, the maximum attainable clock frequency for our accelerator was recorded at 260 MHz resulting in a basecalling speed of 364 Kbases/sec compared to the 2.9 Kbases/s in the CPU case.

On the other hand, reducing the fixed-point bit width from 10 to 7 bits increases the maximum clock frequency of our FPGA accelerator by 13.5%. In other words, the maximum possible clock frequency of our FPGA accelerator is measured at 295 MHz in case of the 7-bits fixed-point design. This leads to a measured basecalling speed of 413 Kbases/s that is 142× faster than CPU-only basecaller.

The speed improvement for our accelerated basecaller over the CPU-only basecaller pertains to the full parallel computing of operations in the *Initialize* and *Iterate* blocks of Algorithm 2 realized by our FPGA design. In the *Initialize*, our accelerator calculates 64 posterior values (i.e., essentially emission probabilities) in a single clock. In the *Iterate* block, the parallel computing takes place in a hierarchical manner that is 64 posteriors (corresponding to a single states' segment) are calculated in a single clock (i.e., unrolling the states loop in lines 8–11). For each of the 64 posteriors, our accelerator executes 21 transition additions as in (4) and (5) in a single clock cycle. Also, the normalization subtraction of the previous event posteriors is carried out in a single clock cycle during each segment calculation by the three adder circuits following the *Posterior Memory* in Fig. 3. Moreover, the end state calculation, in the *EndState* block, is executed in an overlapped manner across the 64 segments of the last event compared to the CPU bubble sorting 4096 numbers to find their minimum.

It is worth mentioning that the C-implementation of the CPU-only basecaller does not exploit specific optimizations (e.g., multicore architectures, multithreading, cache structures, and SIMD extensions). Such kind of optimization techniques is important for developers focusing on improvements for their particular platforms. We rather implemented the code in the manner reflected in the pseudocode presented in Algorithm 1 and relied on the compiler to achieve an appropriately mapped code. In particular, we used the gcc compiler (ver. 4.8.5) while specifying both -O2 and -O3 as the optimization settings both of which, separately, result in the same performance. Our goal was to provide a simple and standard benchmark for our proposed hardware accelerator that could be widely adopted on commodity CPUs for performance evaluation.

In a mobile sequencing context, the processing speed is not the sole objective for the designer (and the user as well). The power consumption margin is another important limiting factor that determines the practical feasibility of the adopted computing platform. This pertains to the fact that mobile DNA sequencers are essentially battery-enabled devices and, thus, long lifetime for these devices while being used in the field requires low power consumption levels. The results reported in Fig. 13 shows the dynamic range of the average power consumed by our accelerator for the four fixed-point designs we are studying. The power consumption spans a range from 2.5 to 6.1 W over the clock frequency range from 50 to 295 MHz. The results also confirm that the effect of the bit width on the accelerator's power consumption is dominating that of the clock frequency. This can be seen for the 8-bits design at 290 MHz compared to the 7-bits design at 295 MHz and the 9-bits design at 275 MHz compared to the 7-bits design at 295 MHz. On the other hand, the CPU's measured average power for running Algorithm 1, over the same event sequence as our accelerator, is 25.16 W (i.e., after subtracting the CPU's idle power of 61.99 W). Thus, our FPGA accelerator (i.e., the 8-bits accelerator clocked at 290 MHz) consumes at least 4$\times$ less power than the CPU for executing the exact same calculations while being faster by 140$\times$.

It should be noted that the measurements reported in Figs. 12 and 13 for both the CPU-only and the proposed accelerated basecallers did not account for the traceback stage in Algorithms 1 and 2, respectively. In particular, the traceback stage execution time on the CPU for a file with events' sequence length of 7000 is measured at 10 μs compared to

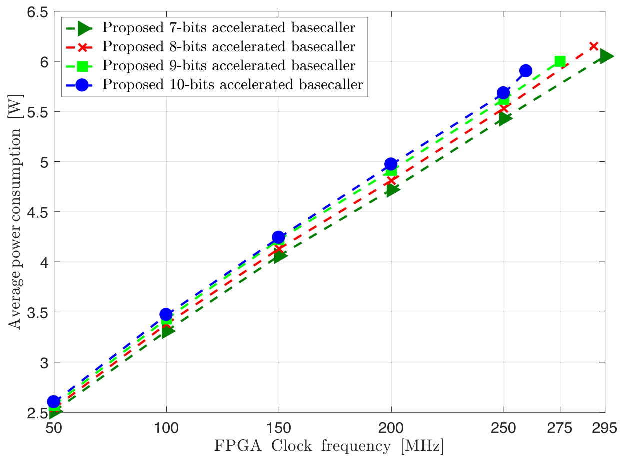

Fig. 13. Measured average power consumption.

17-ms execution time on the FPGA (i.e., for the *Initialize*, *Iterate*, and *EndState* stages). Also, the CPU power consumed for executing the traceback stage is measured at only 0.5 W. Hence, the traceback stage adds marginal latency and power for the basecaller's overall computational budget and can be safely removed from the analysis. This stems from the fact that the core compute-intensive tasks of the basecaller exist in its trellis calculations. From another perspective, the power measurements of our proposed basecaller reported in Fig. 13 does not consider the CPU's idle power since this power is totally independent of our basecaller's operation (i.e., base power consumed by the desktop computer upon power it up and running the OS). Also, due to RIFFA's streamline communication, the CPU does not spare any waiting idle times during the data exchange with our FPGA accelerator while consuming only 0.3 W on top of its idle power (i.e., 61.99 W), which only presents 5% of the FPGA's 6 W power budget. It is also important to note that the basecalling speed results reported in Fig. 12 have completely considered the data movement latency in both directions (i.e., CPU-to-FPGA and FPGA-to-CPU data-paths). In essence, the timer function placed in the C-code on the CPU side starts its operation at the beginning of sending the CPU data and stops after the CPU successfully receives all the FPGA results. Hence, the reported basecalling speed is inclusive and considers the CPU sending latency, FPGA processing latency, and the CPU receiving latency. In sum, focusing our analysis on the accelerator's power and latency budgets does not undermine the proposed collaborative CPU/FPGA framework.

Combining both the speed and power performances to evaluate the energy efficiency (i.e., the bases per joule metric) provides more insightful means to compare our proposed accelerated basecaller with the CPU-only basecaller. As such, the energy efficiencies for both basecallers are calculated in Fig. 14 using the speed and power results previously reported in Figs. 12 and 13, respectively. The results show that our basecaller is more energy efficient than the CPU-only basecaller by two orders of magnitude. These results confirm that the increase in the basecalling speed for our basecaller is outpacing the increase in its power budget with increasing

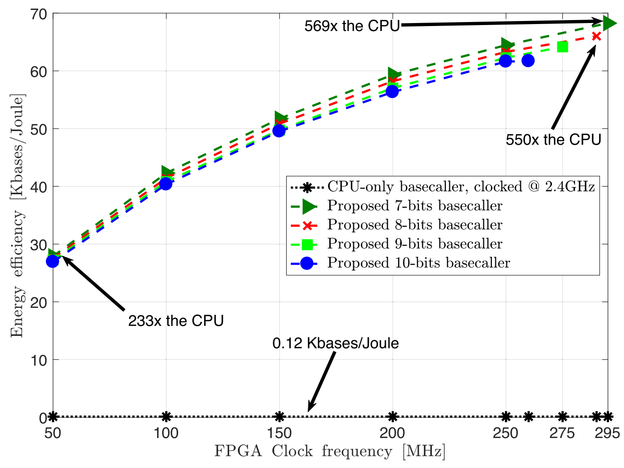

Fig. 14. Energy efficiency.

TABLE I
PROPOSED ACCELERATOR'S RESOURCE UTILIZATION ON VIRTEX-7 XC7VX485T-2FFG1761C DEVICE

Bit width	LUT	FF	DSP	BRAM
7-bits	19%	9%	2.3%	74%
8-bits	21%	10%	2.3%	74%
9-bits	23%	11%	2.3%	74%
10-bits	25%	12%	4.6%	74%

the clock frequency. Thus, it is generally favorable for our basecaller to run at faster clock frequencies for a specific sequencing volume of data since it leads to less computational time and energy, and longer battery lifetime for the sequencing device.

To summarize, the bit-width analysis of our basecaller's design emphasizes an essential tradeoff between its accuracy and energy efficiency, that is, cutting the basecaller's bit width from 10 to 7 bits results in dropping the accuracy by 19% while boosting the basecaller's speed by 13.5% (i.e., from 364 Kbases/s at 260 MHz to 413 Kbases/s at 295 MHz, respectively) and reducing the power consumption by 6.3% (i.e., from 5.9 to 5.53 W at 260 MHz).

The resource utilization for each of the four fixed-point configurations of our proposed accelerator on the target Virtex-7 device is reported in Table I. The utilization of LUTs and FFs steadily increases with the bit width due to the requirement of more combinational logic elements and wider registers to handle wider data bus. However, the DSPs (i.e., DSP48 blocks) is doubled only in case of the 10-bits accelerator due to the multiplication operation of the *Emission Calculator* block, which essentially doubles the data width. In other words, each of the 64 slices of the *Emission Calculator* block consumes only one DSP block in case of the 7-, 8-, and 9-bits designs, whereas two DSP blocks are allocated for each slice in case of the 10-bits design.

The utilization of the BRAM blocks is invariant with the bit width of our accelerator. This stems from the flexibility of the Virtex-7 architecture, which allows allocating either a complete 36-Kb BRAM block or a partial 18-Kb subblock according to the required memory width and depth. Our accelerator only utilizes 18-Kb BRAMs to build the *Model Memory* (i.e., 64 BRAMs for storing the emission parameters) and the *Posterior Memory* (i.e., 64 *Stay* BRAMs, 256 *Step* BRAMs, and 1024 *Skip* BRAMs). For the maximum required depth in case of the *Stay* BRAM (i.e., 128), a single 18-Kb BRAM can accommodate a bit width of up to 36 bits. Therefore, all of our fixed-point configurations in Table I are similarly realized by 18-Kb BRAM primitives, and consequently, the number of BRAMs is the same.

As explained in the previous section, the engine's memory is structured to ensure fitting the model data (i.e., the posteriors for the 4096 HMM states) in the on-chip limited storage resources of our target FPGA device while maintaining the fastest possible speed for reading and writing data (i.e., one clock cycle for each). The BRAM resources needed scale linearly with the number of slices adopted by the *Emission* and *Transition* calculator blocks. As apparent from our utilization levels, doubling the number of slices (which essentially leads to doubling the basecalling speed) would exceed our FPGA's available BRAM resources. Hence, a clear tradeoff is presented to the system design in terms of the power and circuit area/cost from one side and the speed from another side.

C. Accelerator's Scalability Use-Case

To support the scalability of our proposed accelerator architecture (i.e., shown in Fig. 3) as explained in Section IV-D, we present in this section a specific use case, which scales down the 4096-states HMM implementation case studied above to a 1024-states HMM accelerator. The purpose is to demonstrate the adaptability of our proposed acceleration architecture and to show its performance for a nanopore model with lower computational requirements.

The accelerator's hardware design for the 4096-states HMM engine (i.e., evaluated in the previous subsection) has been adapted to implement a 1024-states HMM engine instead according to the steps discussed in Section IV-D. In addition, the CPU-only version of the 1024-states HMM basecaller has been developed to ensure that the CPU comparison is still in place as the case with the previous results. As expected from the discussion provided in Section IV-D, the measured power consumption for the 1024-states HMM accelerator stayed at the same level compared to that of the 4096-states accelerator, as previously shown in Fig. 13. This pertains to the fact that downsizing the depth of the internal BRAM blocks of the accelerator's *Posterior Memory* and *Model Memory* blocks in the 1024-states HMM case does not lower the number of the physical BRAM blocks utilized on our FPGA device (i.e., the resources utilization shown in Table I does not change). However, the measured basecalling speed of the 1024-states HMM accelerator is approximately 4× faster than that for the 4096-states HMM due to the reduced number of states' segments for each input event by the same approximate factor (i.e., the event calculation latency inside the 1024-states accelerator is 184 clock cycles compared to the 712 cycles for the 4096-states accelerator shown in Fig. 9). As a result, the 1024-states HMM accelerator demonstrated a

TABLE II
PERFORMANCE COMPARISON OF THE PROPOSED HMM-BASECALLER WITH EXISTING HMM IMPLEMENTATIONS

Reference	Hardware	Application	HMM States	Speed	Peak Performance	Power
Our CPU Basecaller	**CPU**	**Basecalling**	**4096**	**2.9 Kb/s**	**0.6 GOPS**	**25.2W**
Our CPU Basecaller	**CPU**	**Basecalling**	**1024**	**13.1 Kb/s**	**0.63 GOPS**	**25.2W**
Our Accelerated Basecaller	**Xilinx Virtex-7**	**Basecalling**	**4096**	**364 Kb/s**	**83 GOPS**	**5.9W**
Our Accelerated Basecaller	**Xilinx Virtex-7**	**Basecalling**	**1024**	**1410 Kb/s**	**80 GOPS**	**5.9W**
Accelerated Variant Caller [12]	Xilinx KU-3	Variant calling	3	1579 Kb/s	33 GOPS	-
Accelerated Basecaller [7]	Xilinx Virtex-7	Basecalling	64	12.5 Mb/s	39.2 GOPS	4.9W

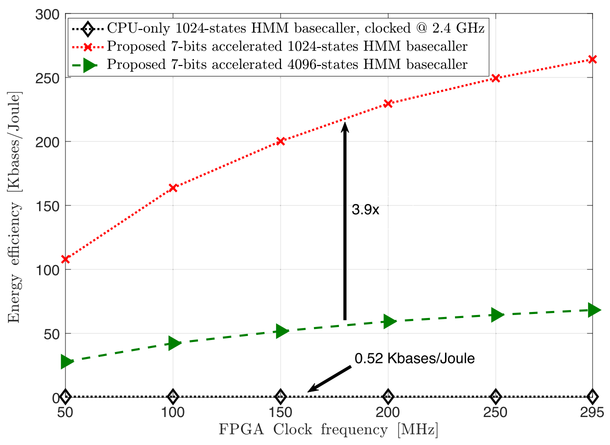

Fig. 15. Energy-efficiency comparison for different HMM complexities.

boost in the energy-efficiency by a factor of 4 compared to the 4096-states accelerator as shown in Fig. 15 for the 7-bits implementation. On the other hand, the CPU-only version of the 1024-states basecaller has consumed the same incremental power on the CPU (i.e., 25.16 W) while running at a speed of 13.14 Kbases/s compared to 2.9 Kbases/s for the 4096-states basecaller, as reported in Fig. 12. In sum, reducing the HMM computational complexity for our proposed hardware accelerator results in improving the energy-efficiency performance by a similar factor.

D. Comparison With Existing Works

The results reported in Table II compare our HMM accelerator, its bases/second output speed, its operations/second (OPS) peak performance, and its power consumption to other HMM implementations. The numbers reported for both versions of our accelerator (i.e., 4096-states and 1024-states HMM) are based on the highest accuracy 10-bits design, which is clocked at 260 MHz. The OPS calculation in Table II is based on the number of operations carried out by the basecaller for each input event. As discussed in the previous subsection, the core operations of the basecaller for each input event exist is its trellis algorithm (i.e., the traceback algorithm performs single memory load operation per event). Besides its use in nanopore basecalling, we include an HMM implementation in a DNA sequencing-related context, that is, variant calling. David *et al.* [8] reported a 6-mer HMM nanopore basecaller on a 4-core 3.4-GHz 12-GB CPU, which achieved a maximum basecalling speed of 6.7 Kbases/s and peak performance of 1.8 GOPS. Our previous work in [7] has studied the hardware acceleration problem for the HMM basecaller that suits a 3-mer solid-state nanopore sensor model proposed in [6]. Hence, that accelerator handles the HMM problem on a scale that is 64× smaller than that of our proposed 4096-states HMM accelerator in this work can address. However, as shown in Table II, the speed of our proposed 4096-states HMM accelerator is only 34× slower than the 64-states HMM accelerator proposed in [7]. This is due to the relatively faster clock frequency and shorter processing latency for each 64-states segment computation of our proposed accelerator compared to that in [7]. In [12], a PCIe-mediated FPGA DNA variant call accelerator capable of calling 1.579 Mbases/s is reported. This variant caller is based on the PairHMM Forward Algorithm [15], a variant of the approach adopted in our algorithm. Although about 4.3× faster than our 4096-states design's basecalling speed, our implementation requires about $10^4\times$ more transition calculations per data point. It is worth mentioning that, in a related context to the variant caller reported in [12], other studies in [19] and [20] have developed FPGA-accelerated computing architectures for aligning DNA sequences. However, we could not add these studies to Table II due to the fact that these studies considered different algorithms other than HMM.

Finally, we believe that the flexibility presented for the proposed scalable acceleration architecture allows it to accommodate computing requirements for future complex nanopore models, that is, partitioning the accelerator's operations across multiple FPGA devices with multiple independent PCIe channels (i.e., supported by RIFFA) will be achievable, albeit, an ASIC realization for the accelerator in such a case would be more power and cost-effective solution.

VI. CONCLUSION

In this article, we have developed a scalable real-time FPGA acceleration framework for basecalling nanopore-derived DNA measurements of varying k-mer resolution. To investigate a challenging case in terms of the hardware design and performance, the presented implementation has specifically addressed the 6-mer (i.e., 4096-states) nanopore signal, a complexity on-par with existing nanopore-based sequencers.

We introduced the architecture of the proposed basecalling acceleration engine. The engine accelerates the basecaller by realizing the compute-intensive trellis-traversal functions of a sequence detection algorithm based on dynamic programming (the execution of the traceback kernel is deferred to the CPU). To enable the implementation of complex basecallers within resource-constrained FPGA hardware, the proposed architecture utilizes a time-multiplexed version of the basecaller that processed the trellis over a sequence of 64 time segments. To enable such a time-multiplexed approach, our design employs an efficient memory structure based on the FPGA BRAM resources. The proposed architecture also accommodates real-time streaming of data between our FPGA acceleration engine and the CPU's main memory (over PCIe).

We experimentally evaluated the effectiveness of our accelerated basecaller and compared its performance to the classical CPU-based implementation. The implementation results demonstrated the superiority of our 28-nm FPGA-accelerated basecaller relative to a 12-core Intel Xeon CPU-basecaller in terms of both speed and power. In particular, our basecaller outperformed the CPU-only basecaller's speed by a factor of 142× when clocking the FPGA at 295 MHz while consuming an average of only 24% of the CPU's average power.

Based on its physical performance and its relatively modest resource needs, the proposed framework offers a potential solution for emerging low-power mobile sequencing devices and provides a useful benchmark for future embedded solutions.

REFERENCES

[1] L. Gannett, "The human genome project," *The Stanford Encyclopedia Philosophy*, E. N. Zalta, Ed. Stanford, CA, USA: Stanford Univ., Metaphysics Research Lab, 2019.

[2] (Oct. 2019). *National Human Genome Research Institute*. [Online]. Available: https://www.genome.gov/sequencingcosts

[3] P. M. Ashton *et al.*, "MinION nanopore sequencing identifies the position and structure of a bacterial antibiotic resistance island," *Nature Biotechnol.*, vol. 33, no. 3, pp. 296–300, Dec. 2014.

[4] *Oxford Nanopore Technologies*. Accessed: Jan. 2020. [Online]. Available: https://nanoporetech.com/products/minion

[5] C. Brown and J. Clarke, "Nanopore development at Oxford nanopore," *Nature Biotechnol.*, vol. 34, no. 8, pp. 810–811, Aug. 2016.

[6] W. Timp, J. Comer, and A. Aksimentiev, "DNA base-calling from a nanopore using a viterbi algorithm," *Biophys. J.*, vol. 102, no. 10, pp. L37-L39, May 2012.

[7] Z. Wu, K. Hammad, E. Ghafar-Zadeh, and S. Magierowski, "FPGA-accelerated 3rd generation DNA sequencing," *IEEE Trans. Biomed. Circuits Syst.*, vol. 14, no. 1, pp. 65–74, Feb. 2020.

[8] M. David, L. J. Dursi, D. Yao, P. C. Boutros, and J. T. Simpson, "Nanocall: An open source basecaller for Oxford nanopore sequencing data," *Bioinformatics*, vol. 33, no. 1, pp. 49–55, Jan. 2017.

[9] R. Wick, L. Judd, and K. Holt, "Performance of neural network basecalling tools for Oxford nanopore sequencing," *Genome Biol.*, vol. 20, no. 1, p. 129, 2019.

[10] S. Ko, L. Sassoubre, and J. Zola, "Applications and challenges of real-time mobile DNA analysis," in *Proc. 19th Int. Workshop Mobile Comput. Syst. Appl.*, 2018, pp. 1–6.

[11] Z. Wu, K. Hammad, R. Mittmann, S. Magierowski, E. Ghafar-Zadeh, and X. Zhong, "FPGA-based DNA basecalling hardware acceleration," in *Proc. IEEE 61st Int. Midwest Symp. Circuits Syst.*, Aug. 2018, pp. 1098–1101.

[12] D. Sampietro, C. Crippa, L. Di Tucci, E. Del Sozzo, and M. D. Santambrogio, "FPGA-based PairHMM forward algorithm for DNA variant calling," in *Proc. IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors*, Jul. 2018, pp. 1–8.

[13] Y. Li and Y. Lu, "BLASTP-ACC: Parallel architecture and hardware accelerator design for BLAST-based protein sequence alignment," *IEEE Trans. Biomed. Circuits Syst.*, vol. 13, no. 6, pp. 1771–1782, Oct. 2019.

[14] O. Ibe, *Markov Processes for Stochastic Modeling*. Amsterdam, The Netherlands: Elsevier, 2013.

[15] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[16] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inf. Theory*, vol. IT-13, no. 2, pp. 260–269, Apr. 1967.

[17] O. Teig, "Ping-Pong scheme uses semaphores to pass dual-port-memory privileges," *EDN*, vol. 41, no. 12, p. 4, 1996.

[18] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, p. 22, Sep. 2015.

[19] K. Benkrid, Y. Liu, and A. Benkrid, "A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 4, pp. 561–570, Apr. 2009.

[20] N. Sebastiao, N. Roma, and P. Flores, "Integrated hardware architecture for efficient computation of the *n*-best bio-sequence local alignments in embedded platforms," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 7, pp. 1262–1275, 2012.

Karim Hammad (Member, IEEE) received the B.Sc. and M.Sc. degrees in electronics and communications engineering from the Arab Academy for Science, Technology and Maritime Transport (AASTMT), Cairo, Egypt, in 2005 and 2009, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Western Ontario, London, ON, Canada, in 2016.

In 2018, he joined the Department of Electrical Engineering and Computer Science, Lassonde School of Engineering, York University, Toronto, ON, Canada, as a Post-Doctoral Visitor. He is currently an Assistant Professor with the Department of Electronics and Communications Engineering, AASTMT, Cairo. His research interests include wireless networks cross-layer design and digital circuit design.

Zhongpan Wu (Member, IEEE) received the B.E. degree in software engineering from DaLain Jiaotong University, China, in 2015, and the M.Sc. degree in electrical engineering and computer science from York University, Toronto, ON, Canada, in 2019, where he is currently pursuing the Ph.D. degree.

He was employed as a Research Assistant with York University, from May 2016 to 2017. His research interests include ASIC and FPGA accelerators design, computer architecture, and machine learning.

Ebrahim Ghafar-Zadeh (Senior Member, IEEE) received the B.Sc. degree from the K.N. Toosi University of Technology, Tehran, Iran, in 1994, the M.Sc. degree from the University of Tehran, Tehran, Iran, in 1996, and the Ph.D. degree from Ecole Polytechnique, Montreal, QC, Canada, in 2008, all in electrical engineering.

He continued his research, as a Post-Doctoral Fellow with the Department of Electrical and Computer Engineering, McGill University, Montreal, and the Department of Bioengineering, University of California at Berkeley, Berkeley, CA, USA. In 2013, he joined the Department of Electrical Engineering and Computer Sciences, York University, Toronto, ON, Canada, where he currently serves as an Associate Professor and the Director of biologically inspired sensors and Actuators (BioSA) Laboratory. His research focuses on BioSA for cell and molecular analysis. He has authored or coauthored more than 125 articles, in various BioSA topics in high-quality journals and international conferences.

Sebastian Magierowski (Member, IEEE) received the Ph.D. degree in electrical engineering from the University of Toronto, Toronto, ON, Canada, in 2004.

From 2004 to 2012, he served on the Faculty of the Department of Electrical and Computer Engineering, University of Calgary, Calgary, Italy. In 2012, he joined the Department of Electrical Engineering and Computer Science, Lassonde School of Engineering, York University, Toronto, where he is currently an Associate Professor. As part of his industrial experience (Nortel Networks, PMC-Sierra, Protolinx Corp.) he has worked on CMOS device modeling, high-speed mixed-signal IC design, and data networks. His research interests include analog/digital CMOS circuit design, communication systems, biomedical instrumentation, and signal processing for biomolecular sensing and analysis.