BMC Bioinformatics

RESEARCH Open Access

FPGA-based accelerator for adaptive banded event alignment in nanopore sequencing data analysis

Yilin Feng[1*†], Zheyu Li[1†], Gulsum Gudukbay Akbulut[1], Vijaykrishnan Narayanan[1], Mahmut Taylan Kandemir[1] and Chita R. Das[1]

†Y. Feng and Z. Li contributed equally to this work.

*Correspondence: ypf5071@psu.edu

[1] Department of Computer Science and Engineering, The Pennsylvania State University, 201 Old Main, University Park, PA 16802, USA

Abstract

Background: Adaptive Banded Event Alignment (ABEA) stands as a critical algorithmic component in sequence polishing and DNA methylation detection, employing dynamic programming to align raw Nanopore signal with reference reads. Motivated by the observation that, compared to CPUs and GPUs, cutting-edge FPGAs demonstrate—in certain cases—superior performance at a reduced cost and energy consumption, this paper presents an efficient FPGA-based accelerator for ABEA, leveraging the inherent high parallelism and sequential access pattern within ABEA.

Result: Our proposed FPGA-based ABEA accelerator significantly enhances ABEA performance compared to the original CPU-based implementation in *Nanopolish* as well as the state-of-art acceleration on GPU and FPGA platforms. Specifically, targeting Xilinx VU9P, our accelerator achieves an average throughput speedup of 10.05× over the CPU-only implementation, an average 1.81× speedup over the state-of-art GPU acceleration with only 7.2% of the energy, and a speedup of 10.11× compared to an existing FPGA accelerator.

Conclusion: Our work demonstrates that intensive genome analysis can benefit significantly from cutting-edge FPGAs, offering improvements in *both* performance and energy consumption.

Keywords: Nanopore sequencing, Genome polishing, Signal-level alignment, FPGAs, Hardware acceleration

Introduction

DNA sequencing is the process of determining the precise order of nucleotides (A, T, C, G) within a DNA molecule. Among the emerging third-generation sequencing technologies, Nanopore sequencing stands out for its capabilities to produce long sequence reads, ranging from thousands to tens of thousands of base pairs at a high speed. This sequencing technology is valuable for a wide range of applications in genomics, epigenetics, and transcriptomics.

The nanopore sequencing detects individual DNA molecules as they pass through a biological nanopore and generates continuous electrical signals or current traces. Changes in these signals are measured to identify nucleotide bases [36] via a 'basecalling' step. Long reads generated by nanopore sequencing facilitate the assembly of complex genomes, the characterization of structural variations, and the analysis of repetitive regions, which are challenging for short-read sequencing technologies.

However, nanopore sequencing technologies also come with their specific challenges. For example, compared with second-generation sequencing, nanopore sequencing technologies exhibit higher error rates (up to 8%), due to probabilistic methods employed in converting electrical signals to nucleotide strings [3, 26]. Consequently, a subsequent 'polishing' process is necessary to improve the accuracy and quality of nanopore sequencing data. This polishing process aligns the raw signals to a biological reference sequence to correct sequencing errors and detect the modified nucleotides. Given the long length of the reads, these downstream processes for analyzing nanopore sequencing data are computationally intensive and require specialized optimization and acceleration strategies.

Adaptive Banded Event Alignment (ABEA), introduced by *Nanopolish* [31], is a key algorithmic component of the genome polishing workflow, aligning the raw signals to a reference sequence via a dynamic programming (DP) strategy. Our profiling results show that ABEA takes approximately 70% of the total CPU processing time as implemented in *Nanopolish*. Prior works have attempted to accelerate the ABEA algorithm on various hardware architectures. For example, in CPUs, *Nanopolish* [31] and *Multi-Nanopolish* [9] launch multiple threads to deal with batches of reads. In comparison, *f5c* [8] optimizes ABEA on heterogeneous CPU-GPU architectures, processing most of the reads on GPU and the remaining longer reads on CPU.[1] *f5c* can perform 3–5× faster than the CPU-only implementation of ABEA in *Nanopolish*. Samarasinghe et al. [28] use OpenCL to accelerate ABEA targeting FPGAs with energy considerations. Their work consumes only 43% of the energy required by the GPU-based acceleration in *f5c* with a performance penalty of 4× slower execution.

Recent work also utilizes deep learning to correct errors in nanopore sequencing. HERRO [34] is proposed as a deep learning-based framework for correcting Simplex nanopore reads, achieving a 100-fold improvement in accuracy. However, in the context of telomere-to-telomere assemblies, HERRO targets nanopore reads longer than 10,000 base pairs, instead of all reads with varying length. Additionally, the training and inference of HERRO take up to several hours on 4 Nvidia A100 GPUs, making it quite time-consuming and highly energy-inefficient. In comparison, ABEA achieves comparable accuracy (>99%) for assembly with higher throughput and less energy consumption [10].

On the other hand, some other works skip the computation-intensive basecalling step and analyze the raw signal directly to meet real-time processing needs. Such works primarily involve subsequence dynamic time warping (sDTW) [6, 27, 29] to align raw signal to reference genomes and eject uninterested reads directly. sDTW has a similar dynamic programming pattern to ABEA and also works for raw signal

[1] Specifically, the reads whose average events per base exceed 5 are processed on CPU.

analysis. However, while sDTW mainly works as a pre-filter to reject untargeted reads in real time, ABEA enhances the accuracy of the assembly after basecalling and read mapping. Unlike ABEA, which does not have size limitations, sDTW is impractical for reference genomes of gigabase size, such as the human genome, due to unacceptable latency. Additionally, ABEA delivers higher alignment accuracy based on precise event-level alignment and adaptive bands to reduce noise.

Compared with CPUs and GPUs, modern FPGAs offer—in certain cases – high performance at a lower cost and with reduced energy consumption [4] (the details of the FPGA used in this study are given in Supplementary Materials: Section 2). Additionally, FPGAs exhibit the inherent advantage of flexibility, allowing the programmers to *configure* memory and power usage depending on the specific tasks required by the application. However, the previous FPGA accelerator for ABEA [28] exhibits *lack* of parallelism in cell calculation in DP table, and also does *not* optimize memory access patterns to make full use of available memory bandwidth based on the specific characteristics of ABEA.

Motivated by the observations above, this paper develops, implements, and evaluates an efficient FPGA-based accelerator for ABEA using Xilinx Vivado High-Level Synthesis (HLS) tool-set [18]. Despite the massive parallel calculation involved in the DP table of ABEA, deploying ABEA on an FPGA presents several critical challenges. Firstly, significant challenges arise from the 'data dependencies' among multiple bands (rows in the DP table) and cells, rendering the parallelization of ABEA non-trivial. Additionally, the varying processing times of different lengths of reads lead to load imbalance, degrading the overall performance substantially. Thus, leveraging the hardware architecture of FPGA, this paper addresses these challenges and makes the following main contributions:

- It proposes an FPGA-based accelerator for ABEA, with the goal of exploiting *both* intrinsic high parallelism and sequential data access pattern characteristics exhibited by ABEA.
- The proposed ABEA accelerator allows multiple reads to perform alignment concurrently in a pipelined fashion, benefiting from the data independence across the bands. This new design enables us to achieve the throughput of one band per cycle for each alignment pipeline.
- Our accelerator integrates two specialized types of Compute Units (CUs): the *Pipeline CU* and the *Ultra Long CU*. The Pipeline CU is optimized for processing buckets of regular reads in a fine-grained pipelining approach. On the other hand, the Ultra Long CU is designed to handle the alignment of ultra-long reads individually, ensuring a balanced load and an efficient alignment process.
- We perform a thorough empirical evaluation of ABEA under different types of compute engines. Our evaluations indicate that the proposed FPGA-based ABEA accelerator demonstrates a remarkable average throughput improvement of 10.05× compared to a classical CPU-only implementation [31]. Further, in comparison to a state-of-the-art GPU-based acceleration [8], it achieves an average speedup of 1.81× while consuming only 7.2% of the energy. Additionally, it achieves a throughput speedup of 10.11× compared to an existing FPGA accelerator [28].

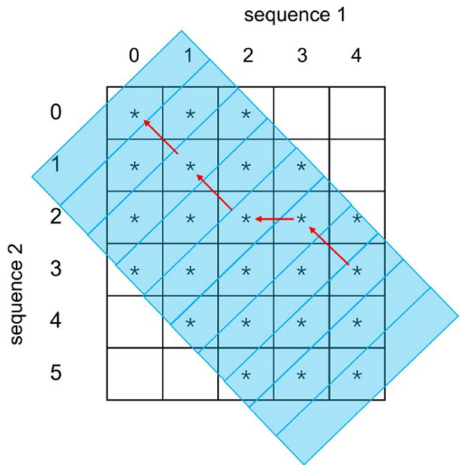

Fig. 1 Alignment with a fixed bandwidth set to 3. Only cells within bands are calculated, marked with '*'. Backtracking traces back to find the optimal alignment as the red arrows show

Background

Adaptive banded event alignment (ABEA)

As a vital component of the genome polishing and methylation detection workflow (genome sequencing and DNA methylation workflow are detailed in Supplementary Materials: Section 1), the ABEA algorithm accounts for approximately 70% of the total CPU processing time in *Nanopolish*, based on our profiling results. Given the substantial volumes of data produced by the nanopore sequencing technology, there is a pressing need for efficient solutions to accelerate ABEA and enhance the performance of the widely-used nanopore sequencing applications.

The ABEA algorithm performs signal-level alignment between events and k-mers to obtain the true annotation. Specifically, events identified from raw signals are *aligned* to a generic k-mer model signal, which encapsulates the frequency distribution of all possible k-mers within a sequence. The generic k-mer model is constructed using the base-called read and a pore-model table provided by Oxford Nanopore Technologies (ONT). The pore-model table contains comprehensive information on all potential k-mers, including their corresponding mean signal value and standard deviation. Consequently, the k-mer model signal is generated by retrieving the appropriate entry in the pore-model table for each k-mer in the read. By aligning events to k-mers, the ABEA algorithm can determine which event corresponds to a specific k-mer in the reference sequence (read mapping results to a reference genome) [30].

Sequence alignment typically utilizes the dynamic programming (DP) approach to compute the alignment score at each pair of positions within a DP table [23, 33]. The process consists of three steps: *initialization*, *score calculation*, and *backtracking*. Due to the quadratic time and memory space complexity of traditional DP alignment, banded alignment techniques are utilized to restrict the search space. In banded alignment, bands of fixed width, centered around the main diagonal, are defined as the DP table. During the alignment process, only the cells within these bands are calculated. As depicted in Fig. 1, only the cells intersecting with these bands are computed, and these cells are marked with '*'. To ensure that the alignment falls within the bands, an appropriate bandwidth

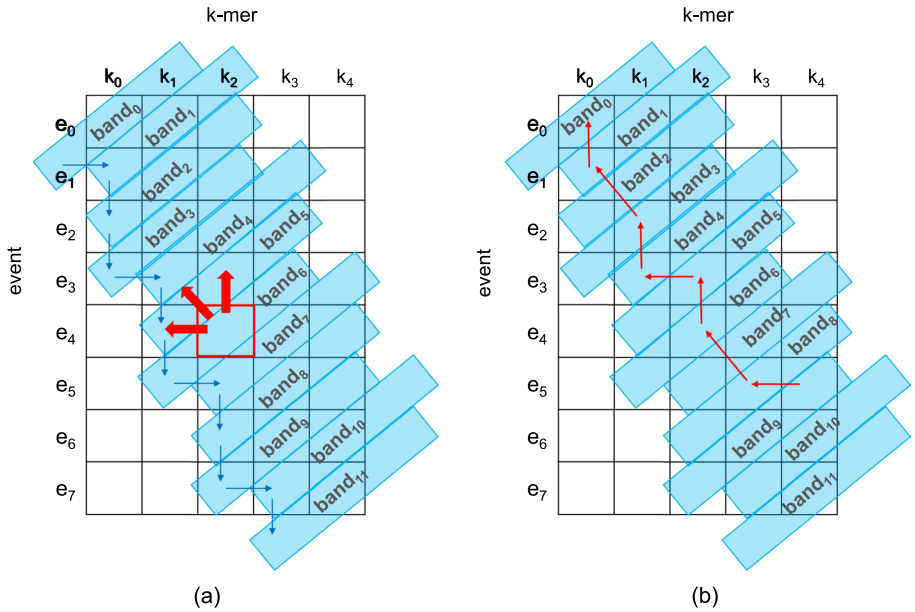

Fig. 2 Alignment with adaptive bands **a** Score calculation step. Bands with a width set at 3 are dynamically adjusted by moving downward or to the right, as depicted by blue arrows. Bands are computed from the top-left to the bottom-right along the left diagonal. Each score calculation depends on three previously calculated scores from the left, up, and previous diagonal cells, as bold red arrows show. **b** Backtracking step. The optimal alignment is found by tracing back from the maximum score along the recorded directional trace flags, as indicated by red arrows

needs to be set. In the backtracking step, the alignment traces back to find the optimal alignment, as indicated by the red arrows shown in the figure.

Given the substantial length of reads with a high error rate produced by nanopore sequencing, alignments frequently deviate significantly from the diagonal. This deviation requires a wider bandwidth for alignment, posing substantial computational challenges. To address this issue, ABEA utilizes an adaptive band scheme instead of static bands when aligning events to k-mers. In this adaptive approach, bands are dynamically adjusted by moving either downward or to the right, determined by the Suzuki–Kasahara heuristic rule [35], as depicted by the blue arrows in Fig. 2a. This dynamic adaptation ensures that the alignment result falls within these bands, facilitating efficient alignment while maintaining accuracy.

Figure 2 provides a simple example illustrating ABEA with a bandwidth set at 3. Figure 2a depicts the score calculation step, featuring 12 bands aligning 8 events to 5 k-mers of reads. These 12 bands are computed sequentially along the left diagonal, from top-left to bottom-right. Each score calculation depends on three previously calculated scores from the left, up, and previous diagonal cells. For example, as shown by the red arrows in Fig. 2a, cell(4,2) depends on cell(4,1), cell(3,2), and cell(3,1). In this case, the calculation of the n-th band is dependent on the (n-1)-th band and (n−2)-th band. Following the score computation, the backtracking step starts from the maximum cell score at the last k-mer, tracing back along recorded directional flags. The resulting backtracking path represents the 'optimal' alignment, as depicted by the red arrows in Fig. 2b.

Algorithm 1 provides the pseudo-code for ABEA. The input data includes k-mers of the read, events, and pore-model table. The resulting alignment is represented as a list

of event and k-mer pairs. Line 1 declares three intermediate arrays: *band_score*, *trace_table* and *band_lower_left*. The *band_score*, *trace_table* store the scores and trace-back direction flags in bands. The *band_lower_left* array holds the coordinates of the first cell (event index, k-mer index) for each band. Lines 2–4 initiate scores and trace flags of the first two bands. Lines 5–6 initiate the starting cell's coordinates for the first two bands. For example, *band_lower_left[0]={1,−1}* and *band_lower_left[1]={1,0}* in Fig. 2. Next, the for-loop between lines 7–24 iterates over the remaining bands. The direction of band movement is determined in line 8 and the starting cell's coordinates are updated in lines 9–13. If the movement direction is to the right, the k-mer index increases by one while the event index remains unchanged for the starting cell's coordinates. Conversely, if the movement direction is downward, the event index increases by one while the k-mer index remains unchanged. Subsequently, the index bounds of each band are decided in line 14, ensuring that only cells within the boundary of events or k-mers are considered. The inner loop from lines 15 to 23 calculates scores within the index bound. Firstly, three neighboring scores are obtained. Then, the value of *lp_emission* is calculated, representing the log probability value that signifies the likelihood of a specific event corresponding to a particular k-mer. *lp_emission* is used along with some heuristically-determined constants (*lp_skip*, *lp_stay*, and *lp_step*) to compute scores from the diagonal, upward, and leftward directions. The maximum score is then determined, and both the maximum score and the direction from which it is obtained are stored in the *band_score* and *trace_table* arrays, respectively. After iterating over all bands, ABEA traces back from the maximum score at the last k-mer to obtain the alignment, represented as (event index, k-mer index) pairs along the direction flags recorded in the *trace table*.

Algorithm 1 ABEA algorithm

Data: *k-mers*, *events*, *model*
Result: *aligned_pairs*
Declare arrays: band_score[n_bands][B], trace_table[n_bands][B], band_lower_left[n_bands];
`/* n_bands is the total number of bands and B is the bandwidth */`
band_score[0][*] = -∞, band_score[1][*] = -∞;
trace_table[0][*] = 0, trace_table[1][*] = 0 ;
band_score[0][0] = 0;
band_lower_left[0] = $\{band_0.e_i, band_0.k_i\}$;
band_lower_left[1] = $\{band_1.e_i, band_1.k_i\}$;
for *band_idx = 2 to n_bands* **do**
  right = Suzuki_Kasahara(band_score[band_idx-1]);
  if *right==true* **then**
    band_lower_left[band_idx] = move_right(band_lower_left[band_idx-1]);
    `/* k-mer index increases by 1 and event index keeps the same as the previous band */`
  else
    band_lower_left[band_idx] = move_down(band_lower_left[band_idx-1]);
    `/* event index increases by 1 and k-mer index keeps the same as the previous band */`
  end
  min_offset, max_offset = get_index_bound(band_lower_left[band_idx]);
  for *offset = min_offset to max_offset* **do**
    diag, up, left = get_scores(band_score[band_idx-1], band_score[band_idx-2]);
    lp_emission = log_probability_match(k_mers, events, model);
    score_d = diag + lp_step + lp_emission;
    score_u = up + lp_stay + lp_emission;
    score_l= left + lp_skip;
    band_score[band_idx][offset] = MAX(score_d, score_u, score_l) ;
    trace_table = the direction of the max score ;
  end
end
aligned_pairs = backtracking(band_score, trace_table) ;

Related works

Nanopore sequencing data analysis is usually performed by aligning genomic reads to the reference genome. Based on the input types, the alignments can work on base space or signal space. The conventional workflow translates raw signals generated from Nanopore sequencers to strings of nucleotide bases with basecallers [24, 38], and then maps these strings to reference genome to get some mapping locations for downstream analysis [5, 13]. Due to the high latency, high cost and low throughput of the basecalling step, some methods analyze raw signals in real-time for selective sequencing [6, 27].

For the conventional nanopore data analysis workflow, basecalling primarily relies on deep neural networks (DNN) for high accuracy, such as Guppy [39] and Dorado [37]. Due to the computational intensity of these methods, some hardware-specific designs

were proposed for acceleration. RUBICALL [32] is the first hardware-optimized mixed-precision basecaller evaluated on a GPU and spatial vector computing system that outperforms the state-of-the-art basellers. GenPIP [16] is an in-memory genome analysis accelerator by integrating basecalling and read mapping tightly. DeepNano-coral [25] is an energy-efficient basecaller developed for Edge TPU, consuming only 10w of power. After basecalling, traditional read mappers, such as minimap2 [13], follow *Seed-Filter-Align* paradigm to align basecalled reads to a genome reference, identifying similarities between sequences. Pre-alignment filtering based on specific architecture [1, 2, 11] is also employed to eliminate highly dissimilar pairs and accelerate read mapping. *Additionally, since errors are often introduced by nanopore basecalling from noisy raw signals, especially in long reads, various methods have been proposed for error correction*. ABEA [31] aligns raw signals to the mapping results to recover the lost information in basecalling. HERRO [34] corrects Simplex nanopore reads longer than 10,000 base pairs using deep learning.

Due to the computational intensity of basecalling in conventional analysis workflows, signal space-based analysis methods for selective sequencing without basecalling are proposed to satisfy the real-time processing requirements. During selective sequencing, classification is performed to eject non-target reads, saving significant time and cost. Subsequent dynamic time warping (sDTW) [15] is commonly used by existing methods to filter non-target reads by aligning signals to reference genomes. Due the quadratic complexity of sDTW, several hardware-specific acceleration methods were proposed. SquiggleFilter [6] is a hardware/software co-designed filter on an edge device for virus detection based on a custom sDTW, which adaptively adjusts the length of read prefix to improve accuracy. DTWax [27] adapts SquiggleFilter's underlying sDTW algorithm to suit GPU via floating point operations and Fused-Multiply-Add operations for high throughput. HARU [29] targets low-cost and resource-constrained system-on-chip devices and accelerates sDTW with on-chip FPGA using fine-grained parallelism. Additionally, indexing methods [41], hash tables [7] and Seed-Filter-Align mappers [14] are also be employed for direct raw signals analysis.

While most analysis workflows utilize conventional CPU, GPU implementations, there is a growing interest in accelerating these workflows using hardware implementations such as FPGAs. Many alignment workflows share a common computational pattern - filling a 2D Dynamic Programming(DP) score matrix to compute the edit distance, which can be efficiently mapped to a systolic style architecture consisting of multiple Processing Elements(PE) on an FPGA. Early works, such as CELL [40], maps the Smith-Waterman algorithm using the systolic approach on FPGA, accelerating the distance computation stage between two genome strings. Another approach [17] also employs systolic-style architecture on FPGA, performing both the forward distance computation stage and traceback mapping stage. Gandafl [12] incorporats a dataflow architecture to perform an end-to-end genome mapping flow, leveraging optimizations like reads interleaving and software co-design to further enhance the efficiency and reduce host-FPGA communication overhead. However, Gandafl only addresses short-read mapping without considering variable lengths. While these previous works have shown promising results, they mostly focus on genomic reads in string representations at the base level. Additionally, they only address limited workloads (fixed-size or short reads). Furthermore, none

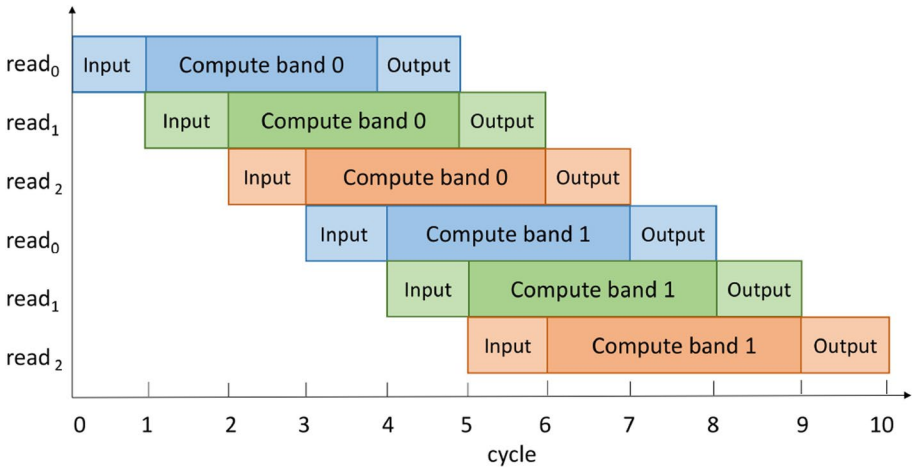

Fig. 3 Timeline of inter-read alignment in full pipeline. Each input and output takes 1 cycle and computation takes 3 cycles. At least 3 reads enable a full pipeline

of them optimize the heuristic move-up/move-down behavior of the ABEA algorithm. One recent work [28] attempts to accelerate the signal-space ABEA on FPGA. However, it lacks parallelism among cell calculations and multiple reads alignment, significantly limiting its speedup. In contrast, our accelerator integrates two types of CUs (one for regular reads and the other for ultra-long reads) and employs an 'inter-read pipeline' to enhance the parallelism across multiple reads, addressing the limitations of previous approaches.

Methodology

Challenges in hardware optimization for ABEA

The core of ABEA lies in computing and storing the 'band score table' and 'trace table', both updated from the top-left to the bottom-right. It is to be noted that *no* data dependency exists in alignments among different reads or score calculations among multiple cells of a band. Given this high degree of parallelism in score calculations of ABEA, FPGA is positioned as a promising device for its acceleration. However, the parallelization of ABEA for FPGA deployment poses at least two challenges. Firstly, there exist 'data dependencies' across different bands, with the movement direction of the current band determined by the previous band using the Suzuki-Kasahara rule. Additionally, each score calculation relies on scores from the left, diagonal, and up cells, which are derived from the previous two bands. Secondly, the varying lengths of reads lead to significant differences in processing times. Unfortunately, this load imbalance degrades the performance, especially as the longest read dominates the total processing time.

To address these challenges, we propose the following solutions. Firstly, we employ *two-level parallelism* for 'inter-read alignment' to address the data dependency issues. This approach allows for parallel score calculations and pipelined processing of multiple reads. Specifically, launching a sufficient number of new reads establishes a 'full pipeline', resolving data dependence between consecutive bands within a read. Figure 3 depicts the timeline of pipelined inter-read alignment, where input and output operations take 1 cycle separately and computation takes 3 cycles. In this scenario, launching at least 3

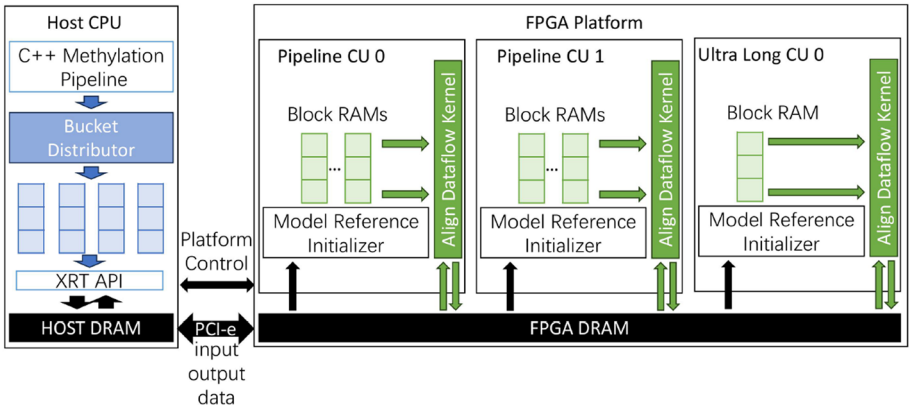

Fig. 4 High-level architecture diagram of the proposed accelerator. C++ Methylation is running on Host CPU and bucket distributor partitions reads into different buckets based on lengths. Once a bucket is full, Xilinx Runtime Library (XRT) will be invoked for transferring data from Host DRAM to FPGA DRAM and launching the specific CU

reads working in a full pipeline adequately addresses the band dependency. For instance, when band 1 of read 0 begins calculation on the 4th cycle, its preceding band would have been already prepared. Our accelerator requires 8 cycles for one band calculation; hence, a full pipeline would need at least 8 reads. Moreover, based on the observation that each band only depends on the previous two bands, we retain only the *three successive bands* for each iteration, thus reducing space complexity significantly. Finally, a 'data partitioning' approach based on the number of bands is utilized to achieve load balance.

High-level accelerator architecture

Figure 4 shows the high-level diagram of the proposed accelerator. It is to be noted that, multiple Compute Units (CUs) can be implemented on the FPGA platform, and each CU can be called and controlled independently. in our design, we consider two types of CUs, namely, *Pipeline CU* and *Ultra Long CU*. The Pipeline CU is optimized for throughput, which allows fined-grained inter-read pipelining. In comparison, the Ultra Long CU is optimized for latency and low hardware utilization, which only accepts one read at a time. This CU targets scenarios where achieving inter-read pipelining becomes challenging, especially in the case of ultra-long reads. The forthcoming subsections will delve into the specifics of these design decisions. The quantity of implementable CUs depends on the specifications of the FPGA. Consequently, there exists a tradeoff in determining the priority between these two types of CUs. Table 1 lists the essential parameters used in our architecture design. All arithmetic operations are performed in standard *fixed-point* representations. We have validated the results against a CPU-based C++ implementation, to ensure enough bits are reserved for all numerical values.

Initially, the C++ Methylation pipeline is running on the Host CPU. A bucket distributor partitions the incoming reads into different buckets based on their 'length' (number of bands). Thus, reads with similar lengths are assigned to the same bucket. The details of our bucket scheduling are described in Supplementary Materials: Section 4. Next, whenever a bucket has collected enough reads for a batch transfer, a Xilinx Runtime Library

Table 1 Architecture parameters, descriptions, and the implemented values in our work

Parameters	Description	Values
B	Bandwidth; number of entries in each ABEA band	100
N	Number of reads to enable full pipeline	8
P	Number of bands processed in post analysis for backtracking in parallel	2
K	Read vector size, number of consecutive read characters in one vector access	64
J	Event vector size; number of consecutive event floats in one vector access	16
M	Number of bits to represent integer part in standard fixed point representation	30
F	Number of bits to represent the fractional part in standard fixed point representation	10
Port width	Width of FPGA DRAM interface	64 bytes

[1] **B**=100 is typically used in previous studies [8] and we adopted the same configuration here. **N** is determined by the Vitis tool and discussed in Sect. 3.4.5. **P** is chosen to match the **Port Width** of FPGA DRAM channel. **K** and **J** are determined experimentally and also discussed in Sect. 3.4.1. **M** and **F** are determined experimentally and we also report their impact on accuracy in Sect. 4.2. **Port Width** is an FPGA device parameter and cannot be changed

(XRT) API is invoked to initiate a transfer from Host DRAM to FPGA DRAM through the PCIe interface. Once the input data are available in FPGA DRAM, the launch of the specific CU and output data transfer are also handled by XRT APIs.

Model reference initializer

The ABEA algorithm requires a pore-model table when aligning, which is essentially a look-up table between text representation (ACTG) and electrical signals, as discussed in the previous section. The pore-model table is universal and valid for the entire dataset, thus only one initialization is required at the very beginning. As the FPGA side of Fig. 4 shows, each CU is accompanied by a Model Reference Initializer. For the Pipeline CU, the pore-model table is maintained as N copies in Block RAMs, allowing independent access for multiple reads. For the Ultra Long CU, only one copy is required since there is no inter-read pipelining.

Dataflow kernel

Figure 5 shows components of the Dataflow Kernel in the Pipeline CU and the details of each component are provided in the following subsections. The kernel is designed based on a streaming dataflow architecture, where inputs and outputs are all connected via stream FIFOs. This allows all stages and sub-functions to run concurrently based on the availability of input data. More importantly, any external DRAM access latency is hidden by the stream FIFO buffer.

Data loading stage

As depicted on the left side of Fig. 5, this stage is responsible for accessing input data, specifically reads and events, for downstream stages. Note that external DRAM access is very expensive and can easily reach hundreds of cycles without careful optimization. In terms of the overall performance, the difference can be 1.5×−2× in our unit experiments. To optimize the external DRAM bandwidth efficiency, we aggregate multiple consecutive reads/events into a single vector access. While larger vector sizes do consume more hardware to interface with DRAM channels, we enlarge the specific vector sizes (K and J) through unit experiments until no noticeable benefits are observed.

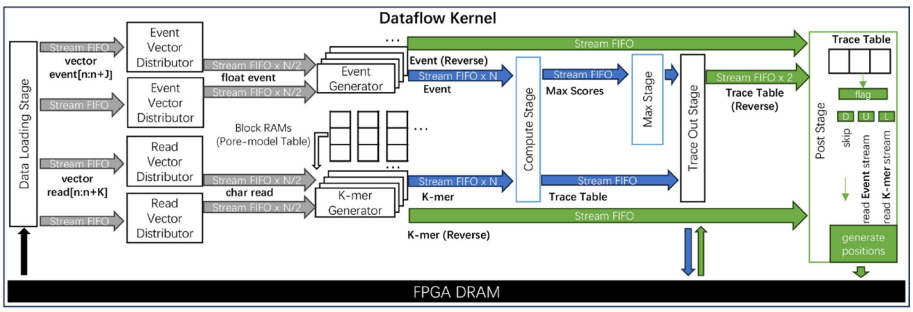

Fig. 5 Details of Dataflow Kernel in the Pipeline CU. The Data Loading Stage loads reads and events from DRAM using vector access. Events/reads are decomposed into individual events as `float` type or bases as `char` type by the Event Vector Distributor and the Read Vector Distributor, respectively. The `float` event is converted into a fixed-point representation and sent to the Compute Stage or the Post Stage by Event Generator. the Kmer Generator converts consecutive bases into k-mers. The Compute Stage performs alignment and generates score and trace tables. The max score collected by the Max Stage and the reverse trace table produced by the Trace Out Stage are transferred to the Post Stage for performing backtracking. All inputs and outputs are connected via stream FIFOs, allowing all stages and sub-functions to run concurrently

Event vector distributor & read vector distributor

Once data are loaded, the vectors containing the consecutive reads/events are received and then decomposed into individual events (represented as `float` types) or read bases (represented as `char` types) by the Distributors. Each read alignment requires its own event FIFO and read FIFO, due to its unique access pattern. As a result, a total of N stream FIFOs (buffering `float` event) and a total of N stream FIFOs (buffering `char` read) are implemented. Each distributor is responsible for distributing the decomposed vectors into N/2 stream FIFOs–emitting one `float` event or one `char` read to one stream FIFO every cycle. Thus, two instantiations of the Event Vector Distributor and the Read Vector Distributor are implemented to ensure that we have no I/O bottleneck for downstream stages.

Event generator

As shown in the middle of Fig. 5, the Event Generator receives the decomposed events from the Event Vector Distributor and then converts the `float` event into a standard fixed-point representation, allowing efficient computation in the following stages. Depending on the current kernel status, the Event Generator sends the converted events to either the Compute Stage or the Post Stage. Similarly, as discussed above, N independent instantiations of the Event Generator are implemented to support inter-read pipelining.

Kmer generator

We utilize a k-mer size of 6 in this work as 6-mer affects the current of ONT nanopore signal. As the input data streams in, a shift register converts every 6 consecutive bases represented in text (ACTG) into a k-mer index. Then, the k-mer index is used to retrieve the corresponding k-mer represented in the electrical signals using the Model Reference. As discussed above, N independent instantiations are required.

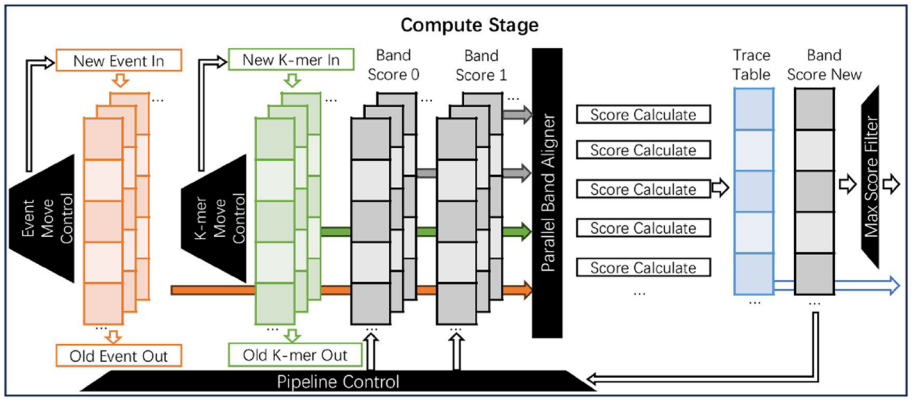

Fig. 6 Details of the Compute Stage in the Dataflow Kernel. N instantiations of the most recent B events/k-mers and the previous two bands' scores are required. Events/k-mers are stored in a shift register structure with B entries. As the band moves right or down, the Event Move Control/Kmer Move Control shifts in one new event/k-mer and pushes out the oldest ones. Whenever a new band score is produced, the previous two band scores get updated. The Pipeline Control guides the Parallel Band Aligner to select from the next ready inputs and perform alignment in each circle, achieving inter-read pipelining. B parallel score calculations are achieved for intra-read parallelism. One band throughput is achieved if fully pipelined

Compute stage

The Compute Stage receives k-mers/events from previous stages and performs the alignment. The details of the Compute Stage are shown in Fig. 6. The score computation within one band requires the most recent B events, B k-mers, and the previous two band scores, as reflected on lines 16–21 of Algorithm 1. As shown in Fig. 6, the events (orange blocks) and k-mers (green blocks) are stored in separate shift register structures, each with B entries. As the band moves right or down, the Event Move Control/K-mer Move Control shifts in one new event/k-mer and places it at the top of the shift register. The oldest event/k-mer automatically gets pushed out. This shift behavior naturally addresses the Suzuki Kasahara rule described in lines 8–13 Algorithm 1. Two previous band scores are also stored and updated whenever a new band score is produced. Every event, k-mer, and band scores have N instantiations which keep track of N reads in the Compute Stage. At each cycle, the Pipeline Control guides the Parallel Band Aligner to select from the next ready inputs and perform alignment, achieving inter-read pipelining. Intra-read parallelism is achieved by performing B parallel score calculations, as captured in lines 15–23 Algorithm 1. The filtered scores and trace tables are sent to the downstream stages. It can be observed that the Parallel Band Aligner requires 6 cycles of latency to produce the result. Also, adding the Pipeline Control increases the latency from 6 cycles to 8 cycles when implemented in Vitis HLS 2021.2. Thus, an N value of 8 is required to enable a *full pipeline*, with 1 cycle initiation interval. When fully pipelined, the Compute Stage can align and produce one band of the score/trace table every cycle.

Max stage

As shown in the middle right of Fig. 5, the Max Stage is responsible for collecting all max scores for all N reads, as well as forwarding them to the downstream stages for backtracking.

Trace out stage

The aligned trace tables produced by the Compute Stage are streamed into the Trace Out Stage at one band/cycle rate. It is impossible to store all trace tables on local Block RAMs due to the unpredictable length; so, all trace tables are required to be written back into FPGA DRAM. Each entry in the trace table is a 2-bit flag. To avoid the underutilization of the DRAM bandwidth, all B entries within one band arriving at the Trace Output Stage are encoded into a single customized vector with 256 bits. Similar to the Data Loading Stage, an external DRAM access is not initiated until 32 bands are accumulated at the Trace Out Stage. Once all N reads finish the alignment, the Output Stage starts to read the trace table back in reverse order (i.e., from the last band to the first band) and sends it to the Post Stage for post-analysis. Similarly, the trace tables are transferred through burst accesses and then reordered. Considering a maximum DRAM bandwidth of 64 bytes (512 bits) per cycle, two bands are streamed out every cycle for post-analysis.

Post stage

The last stage on the right side of Fig. 5 is responsible for backtracking to find the optimal alignment—line 25 in Algorithm 1. In our implementation, P reads perform alignment in a fully pipelined fashion at the Post Stage. Unlike the alignment, the Post Stage does not include complicated score calculation steps and can be pipelined and parallelized easily using `pragma unroll`.[2] Two bands are fetched from streams at every cycle in reverse order, and processed in parallel. The aligned positions are sent to the output streams. Details of the Post Stage algorithm are given in Supplementary Material: Section 3.

Ultra-long reads alignment

Due to the varying lengths, in the worst case when there is only one remaining alignment in the pipeline, the throughput decreases to 1 band in every N cycle from 1 band in every 1 cycle. To address this potential problem, we *divide* the alignments into different 'buckets' according to their number of bands so that the alignments with a similar number of bands can be processed in a *pipelined fashion*. Besides, despite the ultra-long reads constituting under 5% of the overall dataset, their alignment on CPU was identified as a bottleneck in previous work [8], slowing the overall running time by up to 2×. If we were to follow the same approach as in the previous work [8], aligning these ultra-long reads on the CPU alone, it would disproportionately dominate our accelerator's running time too. These alignments take much longer execution time, and it is difficult to construct a bucket with a similar length due to large variances in lengths. As a result, they are processed one by one instead of in a pipelined fashion with other alignments.

[2] This directive instructs the compiler to unroll a given loop by a factor of n so that the loop's body is replicated to create n copies, reducing the number of iterations by a factor of 1/n.

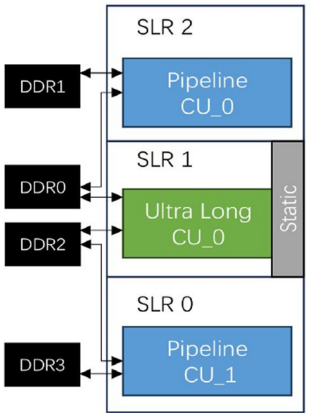

Fig. 7 Device map of Xilinx VU9P FPGA. Each Pipeline CU fits in a single Super Logic Region (SLR) and an Ultra Long CU fits in the smaller SLR1 with a reserved Static Region. Ultra Long CU shares one DRAM channel with each Pipeline CU

Table 2 FPGA SLR resource utilization for each CU type after full placement

CU type	BRAM	DSP	FF	LUT
Pipeline CU	0.37k(50%)	0.66k(25%)	174k(25%)	275k(70%)
Ultra Long CU	0.26k(35%)	0.66k(25%)	125k(18%)	228K(58%)

We designed a specific CU, namely, the Ultra Long CU that is optimized for single-read alignment. The Ultra Long CU removes all pipeline-related logic and memory structures in the Compute Stage, reducing the latency from 8 to 6 cycles. Similarly, only one instantiation of all other stages is required, which also leads to lower hardware resource utilization. We evaluate the effectiveness of our Ultra Long CU in the following section.

Implementation and results

FPGA implementation

As shown in the previous section, our FPGA accelerator consists of multiple independent CUs, instead of a single monolithic CU. This design strategy is motivated by two key factors. First, it allows us to *prioritize* resources for specific types of CUs, optimizing their performance. And, second, implementing a large monolithic CU on FPGA often presents various challenges, including routing congestion and timing violations.

In this work, we target Xilinx VU9P offered by Amazon AWS F1 instance. As shown in Fig. 7, three Super Logic Regions (SLR) are available in Xilinx VU9P. We fit one Pipeline CU into a single SLR region, avoiding cross-SLR routing. For SLR1, around 30% of resources are reserved for device control (Static Region), which makes it suitable for fitting into a smaller 'Ultra Long CU'. It is worth noting that each Pipeline CU has access to two independent DRAM channels while the Ultra Long CU shares one DRAM Channel with each Pipeline CU. Clearly, more CUs can be implemented if a larger FPGA is available, such as Xilinx U250.

The resource utilization of the proposed accelerator deployed on Xilinx VU9P is detailed in Table 2 with the frequency set to 100 MHz. Our implementation comprises 2 Pipeline CUs and 1 Ultra Long CU.

Table 3 Relevant statistics on our datasets

Datasets	D_1	D_2	D_3	D_4	D_5	$D_{example}$
No. of reads	668016	451020	270189	117140	38335	19275
No. of ultra-long reads	4631	6417	9321	19	58	103
Total bases (Mbases)	3203	3620	2730	696	190	158

Experimental design

In our evaluations, we compare the performance of our ABEA accelerator against different prior works on CPU, GPU, and FPGA platforms in terms of i) throughput, ii) power consumption, and iii) execution time. The CPU-only multi-threaded implementation in *Nanopolish* [31] is tested on an AMD EPYC processor with 12 CPU threads. *f5c* [8] is the acceleration of ABEA on heterogeneous CPU-GPU architectures and is tested on an NVIDIA GeForce RTX 3070 GPU with 8 G GDDR6X RAM and an Nvidia Tesla V100 GPU with 16GB HBM, respectively. The evaluation results of the acceleration [28] using OpenCL framework on Altera Stratix V FPGA with 4GB RAM, are obtained directly from their works. Additionally, we analyze the breakdown of execution time across different stages, evaluate the effect of bucket size and bucket range on the performance, and validate the efficiency of the designed Ultra Long CU.

The 'Nanopore WGS Consortium' sequencing dataset [10] is used for our experiments. This dataset is available in the project under accession PRJEB23027 from European Nucleotide Archive (ENA).[3] Reads of the human genome (cell line NA12878) from 53 individual flow cells are publicly available under the project. We selected reads from five individual flow cells under accession ERR2184700, ERR2184710, ERR2184719, ERR2184733 and ERR2184734, and labeled them as D_1, D_2, D_3, D_4 and D_5, respectively. Additionally, $D_{example}$ is a small subset of the NA12878 WGS Consortium data. Note that these datasets were also used in recent studies (e.g., [8, 28, 31]). The relevant characteristics of these datasets are listed in Table 3, including the total number of reads, number of ultra-long reads, and total read bases. Ultra-long reads are defined as the number of events that exceed 120,000 or read lengths that exceed 60,000. We perform alignment for ultra-long reads on the Ultra Long CU, instead of the Pipeline CU. It is to be emphasized that our accelerator does *not* perform any alignment on the CPU, unlike the conventional GPU-based methods (e.g., *f5c*), which rely on the CPU to perform the ultra-long reads.

We have verified the accuracy of our ABEA accelerator by comparing the aligned pairs generated by *Nanopolish* with those produced by our accelerator. The observed differences in positions of aligned pairs, attributed to floating-point arithmetic across different architectures, were found to be less than 0.03%.

In the following sections, we first demonstrate the performance gain achieved due to the architectural difference in comparison with GPU/CPU in Sects. 4.3 and 4.5. Next, we highlight the effectiveness of our bucketing strategy in leveraging multiple CUs on the FPGA

[3] https://www.ebi.ac.uk/ena/browser/home.

accelerator in Sect. 4.4. Finally, we present the performance improvements across various datasets and average throughput/power consumption in real-world workloads.

Quantitative comparison between GPU and FPGA

In this section, we analyze the performance benefit caused by pure hardware architectural difference, that is, excluding all other effects such as software scheduling and length variances. We created an artificial dataset with 128 identical reads of 82k bases. This allows GPU and FPGA to generate the maximum achievable throughput.

Our FPGA accelerator has two Pipeline CUs running at 100Mhz, as shown in Fig. 7. At maximum pipeline utilization(i.e. zero pipeline stall caused by any reasons such as DRAM latency or length variances), each Pipeline CU can achieve 1 band/cycle score calculation rate and 2 bands/cycle post backtracing rate as discussed in Sects. 3.4.5 and 3.4.8. Thus, the theoretical computation time on the FPGA accelerator with 2 Pipeline CUs on the artificial dataset should be $(128/2 * 82k + 128/2 * 82k/2)/100Mhz = 78ms$. We ran the artificial dataset on the real FPGA instance and measured 85ms, which is in line with the theoretical computation time. Several millisecond differences could be attributed to device driver overhead.

Similarly, we ran the GPU-based *f5c* acceleration on the artificial dataset using an NVIDIA GeForce RTX 3070 GPU and measured a total running time of three CUDA kernel 266ms (*align-core*, *align-core* and *align-post* kernels), making it 3.1× slower than the FPGA accelerator(85ms). A key to GPU acceleration is achieving high warp issue efficiency, allowing for high warp parallelism to hide latency. However, *f5c* experiences significant warp stall due to branch divergence (e.g., determining if a cell is the first one of the band), leading to an average stall of 7 cycles per warp in their *align-core* kernel, which is responsible for band scores calculation.

Effect of bucket size and bucket range

While our proposed FPGA accelerator has demonstrated strong performance on the artificial dataset (achieving 3.1× speedup over the GPU), effective scheduling is crucial to extend this performance to the real-word datasets consisting of reads with various lengths. To maximize the utilization of the FPGA accelerator, we implemented a bucket scheduler that essentially groups reads with similar lengths into the same bucket. The Total Cycles spent on Pipeline CUs majorly can be divided into two parts: 1) **Effective Cycles**, which compute useful band scores, and 2) **Wasted Cycles**, caused by length variations in the pipeline where no useful band scores are computed. Here we define pipeline utilization as:

$$\mathbf{PipelineUtilization} = \mathbf{EffectiveCycles}/\mathbf{TotalCycles}$$

Pipeline utilization statistics for varying bucket ranges on D_1 are shown in Table 4, with bucket size set to 128. The bucket range is the length range of reads in one bucket, where the length of one read is the sum of its read length and the number of events. The maximum utilization is achieved with a bucket range of 10K. This observation can be explained by (1) if the bucket range is too large, pipeline utilization decreases due to greater variation in length among reads, thus shorter reads in the pipeline have to wait for longer reads to finish. (2) if the bucket range is too small, it would be difficult to collect enough reads within the specified range, leading to many small fragmented batches

Table 4 Pipeline Utilization

Range	5K	10K	15K	20K	25K
Waste Cycles Due to Fragmentation (%)	0.09	0.03	0.01	0.01	0.01
Pipeline Utilization (%)	0.69	0.70	0.67	0.63	0.59

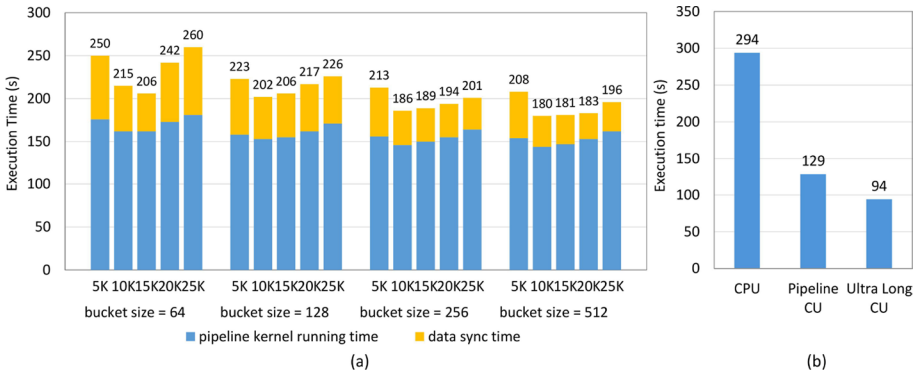

Fig. 8 **a** The effect of bucket size and bucket range on Pipeline CU running time on D_1. Bucket Size is the number of reads in one bucket, and Bucket Range is the range of the sum of read length and number of events per bucket. **b** Ultra-long reads ABEA execution time in different methods: multi-threads on CPU; Pipeline CU with one ultra-long read in each pipeline; Ultra Long CU

(< N reads), which cannot satisfy fully pipelined execution. As shown in Table 4, with a bucket range of 5K, a significant amount of waste cycles are attributed to small fragmented batches.

The Pipeline CU running time with varying bucket range and bucket size collected from the actual FPGA instance is shown in Fig. 8a. The result is in line with the utilization statistics, as a bucket range of 10K yields the best performance in most cases. It is worth noting that for a bucket range of 5K, fragmented batches cause more performance penalties than the utilization statistics indicated. This could be attributed to driver overhead, as too many inefficient small batches and frequent data synchronizations with the FPGA accelerator introduce latency. Regarding bucket size, ideally, large sizes are more advantageous since every CU call and DRAM transfer through the XRT API incur a latency penalty. Therefore, batching as many reads as possible is ideal. However, the benefits plateau beyond a certain threshold. In our case, we observe no significant improvement once the bucket size reaches 256.

However, the GPU-based *f5c* groups reads into batches based on the order of reading data from the disk for thread parallelism processing, without considering the variation in read lengths. The load imbalance workload across warps and blocks leads to low achieved occupancy. Specifically, both *align-pre* kernel and *align-post* kernel achieve around 20% out of their theoretical occupancy 100% and 67%, respectively.

Efficiency of ultra long CU

One out of the three SLRs on the FPGA is dedicated to the Ultra Long CU, as shown in Fig. 7. This design choice is motivated by two primary factors. First, ultra-long reads

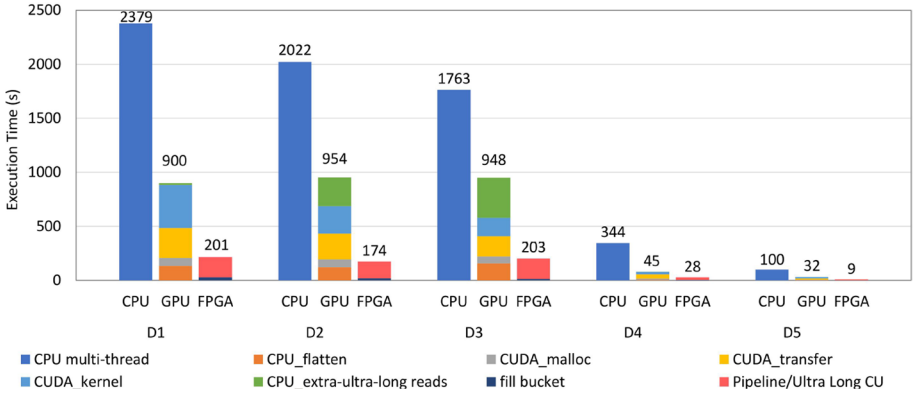

Fig. 9 ABEA execution time comparison among CPU, CPU/GPU-3070 and VU9P FPGA

processed on the CPU disproportionately dominate the total running time on large datasets, making acceleration critical, as discussed in Sect. 3.5. Second, SLR 1, being smaller, is not well-suited for a Pipeline CU, however, it can comfortably accommodate a smaller Ultra Long CU without pipeline control, as discussed in Sect. 4.1

The efficiency of the Ultra Long CU is verified here by performance comparison among the CPU, the Pipeline CU, and the Ultra Long CU. Firstly, we launch multiple threads to perform ABEA for ultra-long reads on the CPU. Secondly, we launch one Pipeline CU to perform ABEA for ultra-long reads, where there is only one valid ultra-long read in each pipeline. Lastly, the Ultra Long CU is used for ultra-long reads as in our accelerator. As Fig. 8b clearly shows, the Ultra Long CU performs better than the other two methods for ultra-long reads. Compared to the CPU implementation, the Ultra Long CU enhances the parallelism through cell score calculation. While one read in the Pipeline CU computes one band every 8 cycles, the Ultra Long CU achieves a faster computation rate of one band every 6 cycles by removing the pipeline.

We show the benefit of incorporating both types of CUs when running on various datasets in the following section.

Execution time comparison

Figure 9 shows the ABEA execution time comparison across CPU, CPU/GPU-3070, and VU9P FPGA. The proposed accelerator on VU9P achieves an average 11.1× speedup over the CPU implementation across the five datasets tested. It is noteworthy that our accelerator can achieve over 11× speedup with the CPU on all datasets, except D_3. In D_3, the high percentage of ultra-long reads causes the Ultra Long CU to dominate the total execution latency. Due to resource limitations, handling only one ultra-long read at a time restricts the achievable parallelism compared to the Pipeline CU. Notably, our accelerator displays significantly higher effectiveness on large datasets with more than 1 G bases, specifically D_1, D_2, and D_3, showcasing over a 4.5× speedup compared to CPU/GPU-3070.

Figure 9 also provides a detailed breakdown of ABEA execution. For the CPU/GPU-3070 implementation of *f5c*, the execution time is broken down into stages: flattening data structures on CPU, CUDA memory allocation, data transfer between CPU and

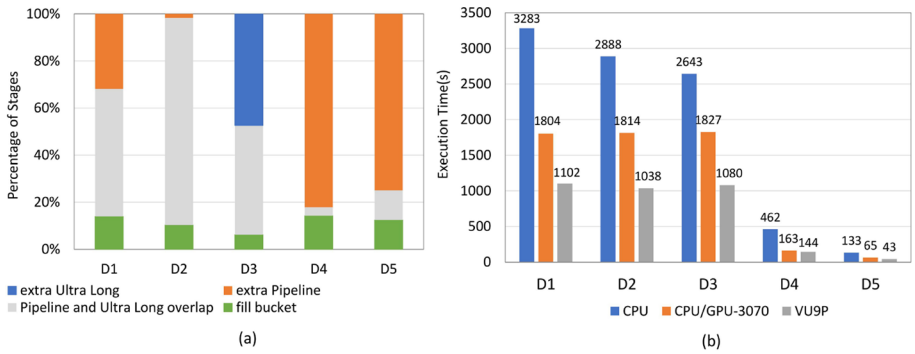

Fig. 10 **a** Percentage of running time for various stages in our ABEA accelerator. All CUs operate independently and are invoked by different threads in an overlapped fashion. The period during which all CUs operate concurrently is recorded as Pipeline and Ultra Long overlap. Any CU activity occurring beyond this overlap period is also tracked, with the additional running times labeled as extra Pipeline or extra Ultra Long. **b** Methylation Detection workflow end-to-end execution time comparison among CPU, CPU/GPU-3070 and VU9P FPGA

GPU, CUDA kernel and additional ultra-long reads processing on CPU. For the proposed FPGA accelerator, the execution time is broken down into filling buckets on CPU and Pipeline/Ultra Long CU(including data synchronization and CU execution which overlap with each other). These breakdowns demonstrate that the proposed FPGA accelerator's speedup is achieved through faster data preparation, improved throughput for read alignments, and the elimination of extra CPU processing. Starting with the data preparation, the CPU/GPU-3070 implementation incurs significant overhead due to the expensive *CPU_flatten* required to compute the index. By contrast, our proposed FPGA accelerator leverages a dataflow architecture that only requires lightweight *fill_bucket*. Then, in terms of accelerator performance, FPGA's *Pipeline/Ultra Long CU* time is significantly faster than GPU running time(*CUDA_malloc*, *CUDA_transfer* and *CUDA_kernel*). The higher throughput of our CUs has been verified by the unit test in Sect. 4.3. Furthermore, unlike the GPU implementation, our FPGA accelerator does not require any CPU extra processing time for ultra-long reads, as this task is entirely handled by the Ultra Long CU on the FPGA.

Breakdown of execution time into different stages

The breakdown of execution time into various stages in CU execution is depicted in Fig. 10a, encompassing the filling buckets stage on the CPU, overlap stage of both Pipeline CU and Ultra Long CU, and extra kernel stage of either the Pipeline CU or the Ultra Long CU. it can be seen from these results that, across most datasets, ABEA is dominated by the Pipeline CU, except for D_3, where a large percentage of ultra-long reads causes the domination of the Ultra Long CU.

Throughput and power consumption comparison

We compare the throughput on individual datasets $D_{example}$, D_2 and D_3, as well as the average throughput and power consumption of our approach against other state-of-the-art approaches on CPU, GPU, and FPGA platforms. $D_{example}$ is a small subset

Table 5 Compute throughput (Mbases/s) on separate datasets, and average throughput (Mbases/s) / power consumption (kbases/watt): comparison among ABEA on CPU-only implementation, GPU/CPU acceleration, and FPGA-based acceleration

Platform	$D_{example}$	D_2	D_3	Avg Thrpt	Avg Pwr Cons
VU9P	21.9	20.8	13.4	16.9	771.54
CPU	2.03	1.79	1.54	1.68	24.07
CPU/GPU-3070	7.67	3.79	2.87	3.38	55.78
DE5-net (projected)	1.67	–	–	1.67	135.1
CPU/GPU-V100	9.26	10.43	8.19	9.33	46.53

[1] **CPU**: AMD EPYC with 12 CPU threads

[2] **CPU/GPU-3070**: AMD EPYC + NVIDIA GeForce RTX 3070 GPU with 8 G HBM

[3] **DE5-net**: Intel E5-1630V3 + Altera Stratix V FPGA with 4GB RAM. Performance is projected by assuming 4 FPGA devices are available to match the resource of a single VU9P FPGA. Power(28nm) is scaled to match the same technology node(16nm) as VU9P FPGA

[4] **CPU/GPU-V100**: Intel Xeon Silver4114 + Nvidia Tesla V100 GPU with 16GB HBM

containing 158 Mbases. Detailed platform configurations are listed in the notes of Table 5. Statistics are obtained by running multiple datasets of various sizes across these platforms. For DE5-net [28], only the result of the small $D_{example}$ dataset is reported. Since the VU9P FPGA has approximately 4× the logic resource of Altera Stratix FPGA [20, 21], we project the performance by scaling the best-reported implementation from the DE5-net work by 4×. Additionally, we re-scale the power consumption of the DE5-net work to compensate for technology node differences(28nm to 16nm) using industry reports [22]. For the CPU/GPU-V100 [8], results from two large datasets (3620 Mbases and 2730 Mbases, namely labeled as D_2 and D_3 in our experimental design) and $D_{example}$ are reported. For power evaluations, we collect the results directly from the AWS-provided 'FPGA-describe-image' commands, an average power of 14w and maximum power of 24w are reported by the commands. Since these commands do not contain power consumed by off-chip DRAM, we assume all 4 off-chip DRAM channels are active and collect the power numbers from Xilinx Power Estimator(2023.1.2) [19]. The CPU and GPU power figures are collected using the 'lm-sensor' commands and 'nvidia-smi' commands respectively. The proposed accelerator on VU9P achieves outstanding results, delivering a 10.05× speedup over the CPU and a 5× speedup over the CPU/GPU-3070 in terms of throughput (Mbases/s). Additionally, the energy efficiency is 32.1× better than the CPU and 13.83× better than the CPU/GPU-3070. The proposed accelerator also demonstrates a 10.11× throughput improvement over DE5-net equipped with the Altera Stratix V FPGA. Compared to the CPU/GPU-V100 setup equipped with the NVIDIA Tesla V100, our solution is 1.81× better in terms of throughput while only consuming 7.2% of the energy. Importantly, these results achieved on the VU9P utilize only 4 available DRAM channels with a theoretical aggregated memory bandwidth of 77GB/s, while the NVIDIA Tesla V100 utilizes 32 HBM channels with a theoretical aggregated memory bandwidth of 896GB/s.

Methylation workflow execution time comparison

The total end-to-end methylation detection workflow execution time (including disk I/O) comparison across CPU, GPU, and FPGA is shown in Fig. 10b. In our evaluation, we

launched 64 multiple I/O processes to optimize the I/O performance, ensuring the disk I/O is no longer a bottleneck. The comparison results demonstrate that the proposed accelerator achieves an average 2.9× speedup over the CPU implementation and an average 1.5× speedup over GPU acceleration. Notably, our accelerator performs significantly better on larger datasets (> 1 G bases), highlighting that ABEA introduces increasingly severe delays as the input size increases, a problem our accelerator effectively addresses with its high throughput.

Conclusions and further work

In this paper, we presented and experimentally evaluated an FPGA-based accelerator for ABEA. The key idea of the accelerator is to design two distinct CUs, namely, the Pipeline CU and the Ultra Long CU, to improve throughput and reduce energy. Implemented on the Xilinx VU9P platform, our accelerator achieves i) a remarkable 10.05× throughput improvement over a classical CPU-based implementation, ii) a 1.81× speedup over a state-of-art GPU acceleration while consuming only 7.2% the energy, and iii) a speedup of 10.11× compared to a previously-published FPGA accelerator. In this work, our optimization efforts have primarily focused on fine-grained pipelined inter-reads alignments for regular reads. Given the significant increase in ultra-long reads from third-generation sequencing technologies, our future work will mainly concentrate on improving the pipelining alignment without limiting read lengths, targeting specifically ultra-long reads.

Appendix A

Supplementary Information

Background of genome sequencing, FPGA programming and pseudocode of Post Stage are detailed in supplementary material.

Supplementary Information

The online version contains supplementary material available at https://doi.org/10.1186/s12859-024-06011-1.

Supplementary file 1.

Acknowledgements
We thank our colleague Sijie Lan at Pennsylvania State University for providing valuable writing suggestions for this manuscript.

Author Contributions
Y.F. and Z.L. conceived the study and designed the FPGA accelerator. Y.F, Z.L. and G.G.A. wrote and revised the manuscript. V.N., M.K. and C.D. supervised the study and revised the manuscript. All author(s) read and approved the manuscript.

Funding
This work is supported in part by NSF Grant 1931531, DOE Grant DE-SC0023186 and PRISM (Processing with Intelligent Storage and Memory).

Availability of data and materials
The HLS code, FPGA configuration file and raw test results are publicly available in the GitHub repository, https://github.com/fengyilin118/ABEA-HLS. The sequencing dataset used for the experiments, provided by the Nanopore WGS Consortium, is accessible under project accession PRJEB23027 in the European Nucleotide Archive (ENA) and can be downloaded from https://github.com/nanopore-wgs-consortium/NA12878.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interest
The authors declare that they have no conflict of interest.

Received: 9 June 2024 Accepted: 9 December 2024
Published online: 17 March 2025

References

1. Alser M, Hassan H, Kumar A, Mutlu O, Alkan C. Shouji: a fast and efficient pre-alignment filter for sequence alignment. Bioinformatics. 2019;35(21):4255–63.
2. Alser M, Shahroodi T, Gómez-Luna J, Alkan C, Mutlu O. Sneakysnake: a fast and accurate universal genome pre-alignment filter for CPUs, GPUs, and FPGAs. Bioinformatics. 2020;36(22–23):5282–90.
3. Chaisson MJ, Tesler G. Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. BMC Bioinformatics. 2012;13:1–18.
4. Cong J, Fang Z, Lo M, Wang H, Xu J, Zhang S. Understanding performance differences of FPGAs and gpus. In: 2018 IEEE 26th annual international symposium on field-programmable custom computing machines (FCCM), 2018; pp 93–96.
5. Dong J, Liu X, Sadasivan H, Sitaraman S, Narayanasamy S (2004) mm2-gb: Gpu accelerated minimap2 for long read DNA mapping. bioRxiv
6. Dunn T, Sadasivan H, Wadden J, Goliya K, Chen K-Y, Blaauw D, Das R, Narayanasamy S. Squigglefilter: An accelerator for portable virus detection. In MICRO-54: 54th annual IEEE/ACM international symposium on microarchitecture, 2021; p. 535-549.
7. Firtina C, Soysal M, Lindegger J, Mutlu O. Rawhash2: Mapping raw nanopore signals using hash-based seeding and adaptive quantization. Bioinformatics, btae 2024;478.
8. Gamaarachchi H, Lam CW, Jayatilaka G, Samarakoon H, Simpson JT, Smith MA, Parameswaran S. Gpu accelerated adaptive banded event alignment for rapid comparative nanopore signal analysis. BMC Bioinformatics. 2020;21(1):343.
9. Hu K, Huang N, Zou Y, Liao X, Wang J. MultiNanopolish: refined grouping method for reducing redundant calculations in Nanopolish. Bioinformatics. 2021;37(17):2757–60.
10. Jain M, Koren S, Miga KH, Quick J, Rand AC, Sasani TA, Tyson JR, Beggs AD, Dilthey AT, Fiddes IT, et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. Nat Biotechnol. 2018;36(4):338–45.
11. Kim JS, Senol Cali D, Xin H, Lee D, Ghose S, Alser M, Hassan H, Ergin O, Alkan C, Mutlu O. Grim-filter: fast seed location filtering in DNA read mapping using processing-in-memory technologies. BMC Genomics. 2018;19:23–40.
12. Koliogeorgi K, Xydis S, Gaydadjiev G, Soudris D. Gandafl: dataflow acceleration for short read alignment on NGS data. IEEE Trans Comput. 2022;71(11):3018–31.
13. Li H. Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics. 2018;34(18):3094–100, 05.
14. Lindegger J, Firtina C, Ghiasi NM, Sadrosadati M, Alser M, Mutlu O (2023) Rawalign: Accurate, fast, and scalable raw nanopore signal mapping via combining seeding and alignment. arXiv preprint arXiv:2310.05037.
15. Loose M, Malla S, Stout M. Real-time selective sequencing using nanopore technology. Nat Methods. 2016;13(9):751–4.
16. Mao H, Alser M, Sadrosadati M, Firtina C, Baranwal A, Cali D, Manglik A, Alserr N, Mutlu O. Genpip: In-memory acceleration of genome analysis via tight integration of basecalling and read mapping. In 2022 55th IEEE/ACM international symposium on microarchitecture (MICRO), IEEE Computer Society, 2022; p. 710–726.
17. Marmolejo-Tejada JM, Trujillo-Olaya V, Rentería-Mejía CP, Velasco-Medina J. Hardware implementation of the smith-waterman algorithm using a systolic architecture. In 2014 IEEE 5th Latin American symposium on circuits and systems, IEEE, 2014; p. 1–4.
18. AMD. Vivado Design Suite User Guide: High-Level Synthesis. https://docs.amd.com/v/u/en-US/ug902-vivado-high-level-synthesis, 2021.
19. AMD. Xilinx Power Estimator (XPE). https://www.xilinx.com/products/technology/power/xpe.html, 2023.
20. AMD. AMD Virtex UltraScale+ FPGAs - Resources. https://docs.amd.com/v/u/en-US/ultrascale-plus-fpga-product-selection-guide, 2024.
21. Intel. Stratix V GX FPGA Development Board Reference Manual. https://www.intel.com/content/www/us/en/content-details/654294/stratix-v-gx-fpga-development-board-reference-manual.html, 2024.
22. TSMC. Logic Technology. https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_16_12nm, 2024.
23. Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol. 1970;48(3):443–53.
24. Pagès-Gallego M, de Ridder J. Comprehensive benchmark and architectural analysis of deep learning models for nanopore sequencing basecalling. Genome Biol. 2023;24(1):71.
25. Perešíni P, Boža V, Brejová B, Vinař T. Nanopore base calling on the edge. Bioinformatics. 2021;37(24):4661–7.
26. Quick J, Quinlan AR, Loman NJ. A reference bacterial genome dataset generated on the minion™ portable single-molecule nanopore sequencer. Gigascience. 2014;3(1):2047-217X.

27. Sadasivan H, Stiffler D, Tirumala A, Israeli J, Narayanasamy S. Accelerated dynamic time warping on GPU for selective nanopore sequencing. J Biotechnol Biomed. 2024;7:137–48.
28. Samarasinghe S, Premathilaka P, Herath W, Gamaarachchi H, Ragel R. Energy efficient adaptive banded event alignment using opencl on FPGA. In 2021 10th international conference on information and automation for sustainability (ICIAfS), 2021; p. 369–374.
29. Shih PJ, Saadat H, Parameswaran S, Gamaarachchi H. Efficient real-time selective genome sequencing on resource-constrained devices. GigaScience. 2023;12:1–16.
30. Simpson J. Aligning nanopore events to a reference. http://simpsonlab.github.io/2015/04/08/eventalign/, 2015.
31. Simpson JT, Workman RE, Zuzarte PC, David M, Dursi LJ, Timp W. Detecting DNA cytosine methylation using nanopore sequencing. Nat Methods. 2017;14(4):407–10.
32. Singh G, Alser M, Denolf K, Firtina C, Khodamoradi A, Cavlak MB, Corporaal H, Mutlu O. Rubicon: a framework for designing efficient deep learning-based genomic basecallers. Genome Biol. 2024;25(1):49.
33. Smith T, Waterman M. Identification of common molecular subsequences. J Mol Biol. 1981;147(1):195–7.
34. Stanojević D, Lin D, Florez de Sessions P, Šikić M. Telomere-to-telomere phased genome assembly using error-corrected simplex nanopore reads. bioRxiv, 2024.
35. Suzuki H, Kasahara M. Introducing difference recurrence relations for faster semi-global alignment of long sequences. BMC Bioinformatics. 2018;19(1):33–47.
36. O. N. Technologies. Oxford Nanopore Technologies. https://nanoporetech.com, 2023.
37. O. N. Technologies. Dorado. https://github.com/nanoporetech/dorado, 2024.
38. O. N. Technologies. How basecalling works. https://nanoporetech.com/platform/technology/basecalling, 2024.
39. Wick RR, Judd LM, Holt KE. Performance of neural network basecalling tools for oxford nanopore sequencing. Genome Biol. 2019;20:1–10.
40. Yu CW, Kwong K, Lee K-H, Leong PHW. A smith-waterman systolic cell. In Field Programmable Logic and Application: 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003 Proceedings 13, Springer, 2003;p. 375–384.
41. Zhang H, Li H, Jain C, Cheng H, Au KF, Li H, Aluru S. Real-time mapping of nanopore raw signals. Bioinformatics. 2021;37:i477–83.

Publisher's Note