Energy Efficient Adaptive Banded Event Alignment using OpenCL on FPGAs

Suneth Samarasinghe
dept. of Computer Engineering
University of Peradeniya
Sri Lanka
imsuneth@gmail.com

Pubudu Prematilaka
dept. of Computer Engineering
University of Peradeniya
Sri Lanka
pubudu.prematilaka@eng.pdn.ac.lk

Wishma Herath
dept. of Computer Engineering
University of Peradeniya
Sri Lanka
wishmalherath@gmail.com

Hasindu Gamaarachchi
Kinghorn Centre for Clinical Genomics
Garvan Institute of Medical Research;
School of Computer Science and Engineering
University of New South Wales
Australia
hasindu@garvan.org.au

Roshan Ragel
dept. of Computer Engineering
University of Peradeniya
Sri Lanka
roshanr@eng.pdn.ac.lk

Abstract*—Nanopore sequencing is a third-generation sequencing technology that can read long DNA or RNA fragments in real-time. Nanopore sequencers measure the change in the electrical current when nucleotide bases translocate through a protein nanopore. These signal level changes are utilized in various nanopore data analysis workflows (such as identifying DNA methylation, polishing and variant detection) to obtain useful results from nanopore sequencing data. Adaptive Banded Event Alignment (ABEA) is a dynamic programming algorithm that is used as a key component in many nanopore data analysis workflows. Prior investigations have shown that ABEA consumes 70% of total CPU time in *Nanopolish*, a popular nanopore data analysis software package. Thus, optimizing the ABEA algorithm is vital for efficient nanopore data analysis. A previous study has proposed an accelerated version of ABEA on GPUs using CUDA that improves the execution time, at the cost of higher energy consumption. With the advancements of High-Level Synthesis (HLS) tools, Field Programmable Gate Arrays (FPGAs) are becoming more and more popular as accelerators that are energy efficient. In this work, we explore the use of the OpenCL for accelerating ABEA on FPGA with energy considerations. We propose a modified version of ABEA for FPGAs using OpenCL and apply various optimization techniques, leading to a few different implementations. We compare the performance of our implementations with other implementations on different hardware platforms in terms of execution time and energy consumption. We show that our best implementation archives an energy consumption of only 43% of the previous implementation of ABEA on GPU, however, with around 4x increase in execution time. We provide all our implementations as open-source software at https://github.com/imsuneth/abea-openclhttps://github.com/imsuneth/abea-opencl

I. Introduction

DNA can be defined as molecules that encode the genetic instructions of humans and almost all other organisms. DNA sequencing is the process of identifying the order of the four chemical building blocks called 'bases' that form the DNA molecule. These four bases are Adenine (A), Thymine (T), Cytosine (C) and Guanine (G). A rapid DNA sequencing technology is beneficial in different applications including the ability to act pre-emptively before disease development and commence treatment.

The latest sequencing technologies generate data in the order of terabytes. In particular, the MinION sequencer manufactured by Oxford Nanopore Technologies has the potential to generate $\sim$1 TB of raw signal data during a typical sequencing run. This higher data volume of sequences poses a crucial challenge in terms of the high computational power required to process data [1].

Nanopore sequencing is a third-generation DNA sequencing technology that does not require sample amplification and it offers several advantages over other sequencing technologies [2]. Some of the significant advantages are the ability to produce long-reads, portability and real-time analysis.

Nanopore sequencing measures ionic current variation as the DNA molecule passes through the nanoscale pore. This ionic current variation at a given time instance is indicative of the bases that are passing through the pore. 'Base-calling' is the process of converting the raw signal into the character representation of DNA bases (i.e., A, C, G, T) [1]. This base-calling process is prone to error. To minimise the impact of base-calling errors on the final results, the raw signal is used in a computational process called 'polishing' [1] [3].

One of the pivotal algorithms used for polishing is the Adaptive Banded Event Alignment (ABEA) algorithm that uses a dynamic programming strategy to align a raw signal to a biological reference sequence [4]. ABEA is compute-intensive and time-consuming (takes around 70% of total CPU time in the popular Nanopolish software package) [1]. Thus, accelerating this ABEA is highly beneficial for efficient

978-1-6654-4429-3/21/$31.00 ©2021 IEEE

Nanopore data analysis.

DNA sequencing is an emerging field that is rapidly evolving. Therefore, it is crucial to reduce the total hardware and software implementation time. Traditional Hardware Description Languages (HDLs) such as Verilog and VHDL have been a design bottleneck for FPGA accelerators over the past years [5]. Using an HLS tool like OpenCL massively reduces the design and development time. In the literature, the utilization of GPUs in this field with CUDA has improved the execution time of algorithms at the cost of higher energy consumption.

In this paper, we present a re-engineered version of ABEA to run on FPGAs using the OpenCL framework along with various optimisations and evaluate the performance in terms of execution time and energy consumption.

II. Background

A. OpenCL Framework

OpenCL platform has a host and one or more devices connected to the host. Each device may have multiple Compute Units (CUs) with multiple Processing Elements (PEs). OpenCL supports two types of kernels, namely NDRange kernels and Single-Work-Item (SWI) kernels. In an NDRange kernel, OpenCL generates a deep pipeline as a computing unit. All the work-items from all the work-groups execute on that pipeline. The compiler automatically performs work-group pipelining. An NDRange kernel has a similar thread hierarchy to CUDA. Each thread is called a work-item, and multiple work-items are grouped to form a work-group. In the SWI kernels, the entire kernel is run by a single work-item, and loop iterations are pipelined to achieve higher performance. The compiler attempts to achieve the least possible Initiation Interval (II) - the number of hardware clock cycles a pipeline must wait for before launching a successive loop iteration.

B. Adaptive Banded Event Alignment Algorithm (ABEA)

A Dynamic Programming (DP) strategy is used to determine the optimal alignment between two biological sequences. Popular algorithms that use DP are Needleman–Wunsch (NW) algorithm and the Smith-Waterman (SW) algorithm.

Typically, sequences that are aligned to each other are mostly similar. Thus, the alignment should lie close to the left diagonal of the DP table. Reducing the number of computations by only considering the cells around the left diagonal is called banded alignment. Long reads (length 100 to 1000 times longer than short reads) are noisier and the alignment path can deviate significantly from the left diagonal due to high errors and large indels. Hence, the band should have a large width which leads to high processing times when millions of reads are aligned.

To increase the processing speed, especially for long reads, a heuristic algorithm called Suzuki-Kasahara (SK) [6] was proposed in 2017. SK uses an adaptive band that allows a shorter band to accommodate such kind of alignment. A modified version of the SK algorithm is used in nanopolish for event-space alignment and it is called Adaptive Banded Event Alignment. Figure 1 is the pseudo-code of the ABEA algorithm. The ABEA algorithm can be divided into three main steps. Initialization of first two bands (line 2), filling the cells with score value for the rest of the bands (line 3 - 16), and finally, traceback step (line 17) which finds the best event-space alignment. Out of these three, the second step is highly compute-intensive. Refer to [1] for a detailed explanation of the ABEA algorithm.

```
1:  function ALIGN(ref,model,events)
2:      initialise_first_two_bands (score,trace,ll_idx)
3:      for i ← 2 to n_bands do
4:          i, dir ← suzuki_kasahara_rule (score[i-1])
5:          if dir == right then
6:              ll_idx[i] ← move_band_to_right (ll_idx[i - 1])
7:          else
8:              ll_idx[i] ← move_band_down (ll_idx[i - 1])
9:          end if
10:         min_j, max_j ← get_limits_in_band (ll_idx[i])
11:         for j ← min_j to max_j do
12:             s, d ← compute (score[i-1], score[i-2], ref,
    events, model)
13:             score[i,j] ← s
14:             trace[i,j] ← d
15:         end for
16:     end for
17:     alignment ← backtrack(score, trace, ll)
18: end function
```

Fig. 1. Adaptive Banded Event Alignment (ABEA)

III. Related Work

SW alignment algorithm has been heavily optimized in the HPC domain. In [7], Rucci et al. have presented SW implementation, which is capable of aligning DNA sequences of unrestricted size for DE5-net FPGA using OpenCL. In this work, the kernel is implemented using the task parallel programming model. They showed that using smaller data types for kernel implementation has increased the performance and reduced resource consumption. In [8], Sirasao et al. have presented FPGA and OpenCL based acceleration for the SW algorithm. They have benchmarked performance per watt on different hardware platforms, including CPUs, GPUs, and FPGAs. Also, it presents a performance tradeoff using OpenCL based programming environment. In [9], Abdul et al. have presented a novel approach to accelerate SW algorithm on FPGAs. They have divided the algorithm into sub-functions and parallelized them by making a group of base-pair alignments. Also, for faster alignment, a 4-bit representation is used for DNA bases.

The most recent work of accelerating the ABEA algorithm [1] is presented by Gamaarachchi et al. They have deployed an accelerated version of the algorithm on CPU-GPU heterogeneous system using CUDA. Their implementation is referred to as *f5c-gpu*. They have achieved 3-5x performance improvement on the CPU-GPU system compared to the original ABEA version of the nanopolish software package.

A major drawback of this implementation is it is limited to Nvidia GPUs due to CUDA API. Also, Modern FPGAs have increased performance with less power consumption. Therefore, we identified that the FPGA accelerator is a suitable choice to address GPUs' energy inefficiency. According to the best of our knowledge, the ABEA algorithm has no FPGA accelerated implementation in the literature.

IV. Methodology

We implement ABEA as both NDRange and SWI kernels. When the host program is executed, it loads the dataset into the host memory, programs the device with the kernel design, allocates buffers, sets kernel arguments and eventually launches the kernels. When the device finishes the execution, the host program reads the results back and stores them in the host memory.

A. NDRange Kernel Implementation

For our NDRange kernel implementation, we adapted the GPU approach taken in [1] and re-engineered it to evaluate the performance of the OpenCL implementation on FPGA. We partitioned the main kernel into three sub kernels, namely pre, core, and post. We tried to achieve the maximum benefit of hardware resources and optimal work-group configuration by splitting the kernel. The NDRange kernel programming model on FPGAs does not support thread-level parallelism like in GPUs. Adding multiple compute units can increase the work-group level parallelism by pipeline stage replication and pipeline widening. The limitation of hardware resources on DE5-net FPGA only allowed us to add a maximum of two compute units. Therefore, our NDRange kernel implementation did not provide the expected performance. Also, the algorithm had data dependencies that are not desired for NDRange kernels. To address these limitations, we decided to modify the kernel structure to the SWI kernel.

B. Single-Work-Item Kernel Implementation

The implementation of SWI kernels follows a typical C program written for the CPU and they are best suited for implementing deeply pipelined algorithms. Each loop-iteration is used as the unit of execution of a kernel. Therefore, multiple loop-iterations are computed in different pipeline stage in parallel.

In the ABEA algorithm, the first step (line 2 in Figure 1) initializes bands and trace arrays and initializes the first two bands. A Rank for each kmer in the sequence is determined by assigning a weight for each base and shifting according to the place of the base within a kmer and saves in an array called 'kmer_ranks' (using a for-loop) for later computations. This for-loop can be pipelined with an initiation interval of 1 since there are no data or memory dependency between two iterations.

The second step (line 3 to line 16 in Figure 1) calculates the rest of the bands (b2, b3,..) while moving the adaptive band according to the Suzuki Kasahara rule. Calculation of the current band depends on the previous two bands results. Therefore, the loop has to be serially executed.

```
1: function ALIGN_SWI_KERNEL(ref, model, events))
2:     for read ← 1 to n_reads do
3:         for i ← 0 to n_kmers do
4:             for j ← 0 to kmer_size do
5:                 rank ← accumulate_rank (sequence[i+j])
6:             end for
7:             kmer_ranks[i] ← rank
8:         end for
9:         for i  0 to n_bands do
10:            for j  0 to bandwidth do
11:                bands[i,j] ← -infinity
12:                trace[i,j] ← 0
13:            end for
14:        end for
15:        bands, trace ← initialize_first_two_bands
16:        for i ← 2 to n_bands do
17:            i dir ← suzuki_kasahara_rule(score[i-1])
18:            if dir == right then
19:                ll_idx[i] ← move_band_right(ll_idx[i - 1])
20:            else
21:                ll_idx[i] ← move_band_down(ll_idx[i - 1])
22:            end if
23:            min_j,max_j ← get_limits_in_band(ll_idx[i])
24:            for j ← min_j to max_j do
25:                s,dir←compute score and max_score_dir
26:                score[i,j] ← score
27:                trace[i,j] ← max_score_dir
28:            end for
29:        end for
30:    end for
31: end function
```

Fig. 2. SWI Implementation of ABEA

An inner loop always goes through the band and fills the cells within a band. This loop can be pipelined with a minimum initiation interval of 1 due to the absence of data or memory dependency between loop iterations.

The final traceback step (line 17 in Figure 1) consists of a loop with high data dependency between two loop iterations. This behaviour results in pipelines with an initiation interval of almost the latency of the pipeline stage. Therefore, it is equivalent to serial execution, which is more suitable for running on a CPU than an SWI kernel on FPGA.

According to the above observations, we built a heterogeneous system merging the first step (line 3 to line 15 in Figure 2) with the second step (line 16 - 29 in Figure 2) to build a deeply pipelined SWI kernel and the CPU performs the final traceback step. Note that the for-loop at line 4 and line 10 in Figure 2 has a constant trip count (KMER_SIZE). Therefore, we fully unrolled the loop for better performance. But, the trip count of the for-loop at line 24 changes based on outer loop iterations. Therefore, we partially unroll this for-loop with

a unroll factor of 4. The OpenCL offline compiler generates separate hardware to handle loop-carried dependencies. We instruct the compiler to avoid falsely detecting loop-carried dependencies for better hardware resource utilization.

Figure 3 shows a pipeline diagram including only the main for-loops in the kernel (For-loops at line 2, line 16, and line 24 in Figure 2). Computation related to a new read starts its execution in every clock cycle. A set of bands in a read executes in a serial manner due to unavoidable data dependencies. A new cell inside a band starts its execution in every clock cycle.

Since, the number of reads in a dataset is extremely large, if we allocate memory on a per-read basis, executing FPGA kernels per read introduces a significant data transfer and control overhead. Therefore, we process a batch of reads at a time to optimize the performance. We allow direct memory access (DMA) and allocate data structures that are 64-bit aligned on the host for efficient data transfer.

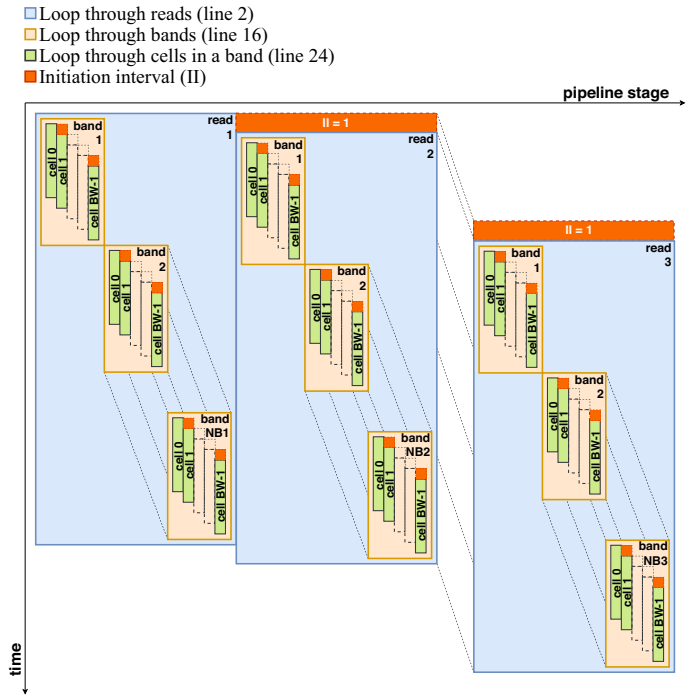


Fig. 3. Pipeline Diagram of SWI Kernel Implementation

V. Experiments and Results

A. Experimental setup

Table I shows specifications of hardware accelerators and the host PC used to obtain results.

B. Dataset

The experimental dataset (*chr_22*) is a subset of publicly available NA12878 (human genome) "Nanopore WGS Consortium" sequencing data [3]. The dataset consists of 19275 reads with 158.8M bases, and the mean read length is 7.7K bases.

TABLE I
Device Specifications

Platform	FPGA (DE5-net)	GPU
Host	Intel(R) Xeon(R) CPU E5-1630 v3 @3.70GHz 32 GB RAM	Intel(R) Xeon(R) CPU E5-1630 v3 @ 3.70GHz 32 GB RAM
Accelerator	Altera Stratix V 4 GB RAM	Tesla K40 12 GB RAM
Operating System	CentOS 7	Ubuntu 20.04
Compiler/ SDK	Intel FPGA OpenCL 18.0	CUDA 11.1

Detailed analysis of all the loops in SWI kernel as reported by the OpenCL offline compiler is shown in Table II. Apart from the three main for-loops mentioned above, other loops are fully unrolled when the lower and upper bounds are constant for each iteration of its outer loop. The rest of the loops are made to execute in a pipelined manner with an initiation interval of 1.

TABLE II
Loop Analysis of SWI Kernel Implementation (DE5-net)

Loop at	Pipelined	II	Bottleneck	Details
line 2	Yes	>=1	n/a	User-contrained II
line 3	Yes	∼ 1	n/a	II is an approximation
line 4	n/a	n/a	n/a	Fully unrolled
line 9	Yes	∼ 1	n/a	II is an approximation
line 10	n/a	n/a	n/a	Fully unrolled
line 16	No	n/a	n/a	Out-of-order inner loop
line 24	Yes	∼ 1	n/a	4X partially unrolled, II is an approximation

*Line numbers are stated respect to Figure 2

Table III shows the estimated resource utilisation by the SWI kernel in the design, global interconnect, and board interface compiled for DE5-net FPGA.

TABLE III
Estimated Resource Usage of SWI Kernel Implementation (DE5-net)

	ALUTs	FFs	RAMs	DSP Blocks
SWI kernel	230608	259136	1188	133
Global Interconnect	9860	22796	61	0
Board Interface	39076	51471	283	0
Total	279544 (60%)	333403 (36%)	1532 (60%)	133 (52%)
Available	469440	938880	2560	256

We select a set of implementations on different platforms and perform event alignment on *chr_22 dataset*. Then we compare the performance in terms of data transfer time, execution time and power consumption. The selected set of implementations are as follows.

cpu f5c-cpu [1]
cuda-k40 f5c-gpu forcing all of the reads to be computed on GPU (Tesla K40) [1]

TABLE IV
COMPARISON BETWEEN DIFFERENT IMPLEMENTATIONS

Implementation	Data (s)	Pre (s)	Core (s)	Post (s)	Ailgn (s)	Total (s)	Power (W)	Energy (J)	Rank
cpu	-	221.238			221.238	221.238	72.38	16013	5
cuda-k40	2.653	22.358	50.779	33.443	106.580	115.719	148.45	15822	4
ocl-k40	3.621	10.057	68.564	36.737	115.358	118.979	147.18	16978	6
nd-init	6.308	2.662	888.559	40.107	931.328	937.636	22.34	20806	7
nd-opt-1	7.581	2.932	1206.191	43.018	1252.141	1259.722	22.34	27973	8
swi-init	87.637	632.188			632.188	719.825	20.34	12859	2
swi-opt-1	82.299	126.049	240.015	20.717	386.781	469.080	18.12	14972	3
swi-opt-2	85.137	368.473		12.466	380.939	466.076	16.87	7118	1

ocl-k40 OpenCL NDRange implementation on GPU (Tesla K40)
nd-init Direct conversion from CUDA to OpenCL NDRange
nd-opt-1 nd-init with core kernel decomposition
swi-init Initial implementation as a OpenCL SWI kernel on FPGA
swi-opt-1 Band initialization and filling rest of the bands on FPGA, backtracking on CPU
swi-opt-2 swi-opt-1 with reduced initiation interval with loop unrolling, minimizing loop carried dependencies, and aligning kernel structs

Power consumption of different hardware platforms is calculated as explained below.

Our DE5-net board does not have an on-board power sensor. Therefore, we run Quartus early power estimator tool [10] on the placed-and-routed OpenCL kernel to estimate the power usage of the board. We assume that each memory module uses a maximum of 1.17 Watts based on the datasheet of a similar memory model [11]. Hence, 2.34 Watts is added to the resulting estimation value to account for the power consumption of the two memory modules.

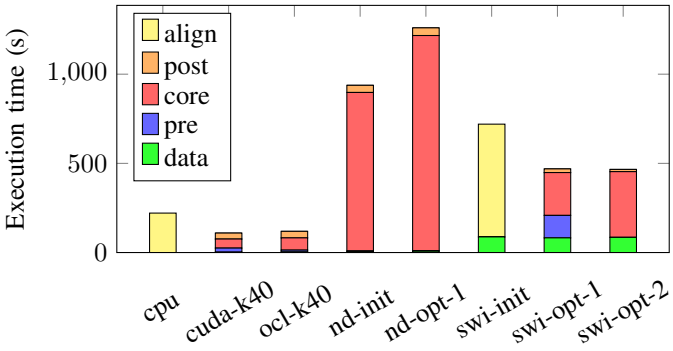
We used nvidia-smi tool [12] to measure power draw of the Nvidia GPU card with a sampling interval of 1ms. To measure the power consumption of CPU and RAM modules of the host computer, we use Intel Power Governor software utility library [13].

When an implementation uses both host and FPGA for event alignment calculations, we calculate the total energy consumed by both host and FPGA. We assume 72W as the maximum power consumption of host. 72W is the power consumption while executing the *cpu* implementation with all pre, core, and post computations. Hence, it is reasonable to take it as the max boundary.

Table IV shows the results we obtained for different implementation. Following is an explanation of each column of the table.

Data For implementations with both host and device, the summation of host-to-device and device-to-host data transfer time in seconds
Pre Pre kernel execution time in seconds
Core Core kernel execution time in seconds
Post Post kernel execution time
Align Execution time without data transfer time in seconds
Total Execution time with data transfer time in seconds
Power Power consumption of the hardware in Watts
Energy Energy consumption of the hardware for event-alignment in Joules (summation of $executiontime \times power$ for each hardware)
Rank Rank given to the implementation for less energy consumption

Figure 4 shows the execution time of each implementation and Figure 5 depicts the total energy consumption of each implementation for performing event alignment on *chr_22* dataset.



*align time is only for the implementations without kernels

Fig. 4. Execution Time on Different Implementations

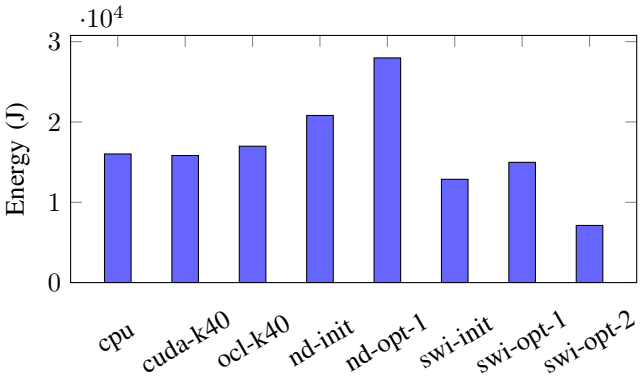


Fig. 5. Energy Consumption on Different Implementations

The observations in Table IV can be analyzed and justified as follows.

Even though NDRange kernels on FPGA have a lesser power consumption than GPU implementations, they reported a higher execution time. Therefore, they are ranked at 7 and 8 in terms of the energy consumption.

Usually, *f5c-gpu* allocates a set of very long reads (selected according to a heuristic) to be computed on the CPU and the rest of the reads on the GPU. It results in around 50 seconds of execution time on Tesla K40. But, here we force the *f5c-gpu* implementation to compute all the reads only on the GPU (*cuda-k40*). We observe that *cuda-k40* and *ocl-k40* perform almost at the same level.

Due to the lesser execution time of *cuda-k40*, it outperforms the energy advantage of *cpu* and gets ranks 4 and *ocl-k40* gets rank 6. As mentioned, since the CPU's power requirement is lesser than that of the GPU, based on the energy consumption, the *cpu* implementation gets rank 5. Among SWI implementations, kernels with suitable FPGA specific optimization techniques shows an improved the performance in execution time and power consumption which lead to less energy consumption. Hence, *swi-opt-2* implementation is in rank 1 and others get rank 2 and 3.

Among NDRange implementations on FPGA, decomposition of kernels into too many kernels results in poor execution time even though the power consumption (estimated for DE5-net) is the same.

Among FPGA implementations, all SWI kernels (*swi-**) perform significantly better than NDRange kernels on FPGA (*nd-**) in terms of both execution time and power consumption. The best SWI kernel is 2x faster and consumes only 34% of the energy compared to the best NDRange kernel.

Compared to the *cpu*, *cuda-k40* has around 2x execution time reduction and 2x power increase resulting almost the same energy consumption. Compared to the *cpu*, *swi-opt-2* has around 2x execution time increase and 4.5x power reduction resulting 2.2x reduction of energy consumption. As shown in Figure 4, in terms of execution time, GPU implementations (both *cuda-k40* and *ocl-k40*) perform better and 4x faster than *swi-opt-2* on DE5-net.

However, in terms of the energy required to perform ABEA on the same dataset, SWI kernel implementations on FPGA are in lead. *swi-opt-2* on DE5-net needs only 43% of the energy consumption of the GPU implementation on Tesla K40.

VI. Discussion

Throughout our work, we identified the potential and ease of using HLS over traditional methods for hardware programming. We used DE5-net FPGA with OpenCL 18.0 for experimenting and evaluating the results. It is a mid-range hardware compared to the state-of-the-art. The maximum predicted frequency we got for the kernels was around 250 MHz and it is even lesser at the execution. The kernel operating frequencies of FPGAs are significantly lower compared to CPUs and GPUs. Due to the absence of power sensors in the DE5-net board, we had to estimate based on the circuit elements and using Intel Quartus Early Power Estimator (which is stated to give a medium accuracy of the estimate). The true power consumption of kernels may differ due to many other reasons such as environmental conditions. Therefore, we believe that the advancement of FPGA hardware and optimized HLS tools can provide better results.

VII. Conclusion

In this paper, we presented several implementations of the ABEA algorithm using OpenCL to run on FPGA, that employ various optimisations. We evaluated the performance of the implementations in terms of runtime and energy consumption. Among our FPGA related implementations, the single work item kernel applied with suitable FPGA specific optimization techniques performed better than other FPGA implementations including the NDRange kernel. In terms of the total energy consumption, FPGA implementations of ABEA performed better than the implementation on GPU. Our optimized single work item FPGA implementation on DE5-net needed only 43% of the energy consumption of the GPU implementation running on a Tesla K40 GPU. However, our FPGA implementation was 4X times slower than the GPU version.

References

[1] H. Gamaarachchi, C. W. Lam, G. Jayatilaka, H. Samarakoon, J. T. Simpson, M. A. Smith, and S. Parameswaran, "GPU Accelerated Adaptive Banded Event Alignment for Rapid Comparative Nanopore Signal Analysis," *BMC Bioinformatics*, pp. 1–13, 2019.

[2] J. Clarke, H. C. Wu, L. Jayasinghe, A. Patel, S. Reid, and H. Bayley, "Continuous base identification for single-molecule nanopore DNA sequencing," *Nature Nanotechnology*, vol. 4, no. 4, pp. 265–270, 2009.

[3] M. Jain, S. Koren, K. H. Miga, J. Quick, A. C. Rand, T. A. Sasani, J. R. Tyson, A. D. Beggs, A. T. Dilthey, I. T. Fiddes, S. Malla, H. Marriott, T. Nieto, J. O'Grady, H. E. Olsen, B. S. Pedersen, A. Rhie, H. Richardson, A. R. Quinlan, T. P. Snutch, L. Tee, B. Paten, A. M. Phillippy, J. T. Simpson, N. J. Loman, and M. Loose, "Nanopore sequencing and assembly of a human genome with ultra-long reads," *Nature Biotechnology*, vol. 36, no. 4, pp. 338–345, 2018.

[4] J. T. Simpson, R. E. Workman, P. C. Zuzarte, M. David, L. J. Dursi, and W. Timp, "Detecting DNA cytosine methylation using nanopore sequencing," *Nature Methods*, vol. 14, no. 4, pp. 407–410, 2017.

[5] H. M. Waidyasooriya, M. Hariyama, and K. Uchiyama, *Design of FPGA-based computing systems with openCL*. 2017.

[6] H. Suzuki and M. Kasahara, "Introducing difference recurrence relations for faster semi-global alignment of long sequences," *BMC Bioinformatics*, vol. 19, no. Suppl 1, 2018.

[7] E. Rucci, C. Garcia, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matias, "Accelerating smith-waterman alignment of long DNA sequences with OpenCL on FPGA," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10209 LNCS, pp. 500–511, Springer Verlag, 2017.

[8] A. Sirasao, E. Delaye, R. Sunkavalli, and S. Neuendorffer, "Fpga based opencl acceleration of genome sequencing software," *System*, vol. 128, no. 8.7, p. 11, 2015.

[9] S. Abdul, M. Al, Z. A. Majid, and A. K. Halim, "High Speed DNA Sequencing Accelerator using FPGA Faculty of Electrical Engineering Universiti Teknologi MARA," pp. 8–11, 2008.

[10] Altera Corporation, "PowerPlay Early Power Estimator User Guide Subscribe Send Feedback," 2015.

[11] Kingston Technology, "Memory Module Specifications KVR16S11S6/2 2GB 1Rx16 256M x 64-Bit PC3-12800 CL11 204-Pin SODIMM."

[12] "NVIDIA System Management Interface." Accessed: Mar. 27, 2021. [Online]. Available: http://www.developer.nvidia.com/nvidia-system-management-interface.

[13] "Intel Power Governor." Accessed: Mar. 23, 2021. [Online]. Available: https://www.software.intel.com/content/www/us/en/develop/articles/intel-power-governor.html.