Accelerating Adaptive Banded Event Alignment Algorithm Using OpenCL

- Semester 8 Report -

Wishma Herath
Suneth Samarasinghe
Pubudu Prematilaka

Department of Computer Engineering
University of Peradeniya

Final Year Project (courses CO421 CO425) report draft submitted as a requirement of the degree of
B.Sc.Eng. in Computer Engineering

January 2021

Supervisors: Prof. Roshan Ragel (University of Peradeniya) and Dr. Hasindu Gamaarachchi (University of New South Wales)

I would like to dedicate this thesis to my loving parents and “teachers” . . .

Declaration

I/We hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is our own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgments.

Wishma Herath
Suneth Samarasinghe
Pubudu Prematthilaka
January 2021

Acknowledgements

We would like to thank the immense support and the guidance given by our project supervisors Dr. Hasindu Gamaarachchi and Prof. Roshan Ragel. Special gratitude goes to the Faculty for providing the necessary equipment and support for this project.

Abstract

Nanopore sequencing can be taken as one of the most prominent technologies being developed as a cheap and fast alternative to conventional sequencing methods, especially in the sequencing of polynucleotides in the form of DNA or RNA. Applications involved in direct contact with genotyping and point-of-care diagnostics require efficient bioinformatics algorithms to analyze the raw nanopore signal data.

To perform these requirements efficiently, the utilization of an optimized bioinformatics algorithm causes a significant change in the field of nanopore sequencing. Adaptive Banded Event Alignment(ABEA) is a commonly used bioinformatics algorithm. The original implementation of ABEA in the Nanopolish software package has already been parallelized and optimized for GPUs(named f5c). It performs efficiently on heterogeneous CPU-GPU architectures.

Even though the ABEA algorithm has been fine-tuned to exploit architectural features in GPUs, the hardware on such generic processors cannot be modified. It will be a tremendous achievement if customized hardware performs with the optimized ABEA algorithm to improve the overall system.

Currently, one of the growing trends in the FPGA domain is OpenCL. OpenCL allows writing programs in high-level languages such as C. Then, the programs can be converted by underlying layers to run on heterogeneous systems consisting of CPU, GPU, and FPGA.

In this work, We implemented a re-engineered version of ABEA which enables specifically to run on FPGAs with the usage of OpenCL. We tried to experimentally identify and adapt FPGA optimization techniques to achieve better performance. Eventually we were able to archive twice-better-power-consumption advantage by comparing with the previous implementation. For the Performance evaluation, we presented observations of how both Single-Work-Item kernel and NDRange kernel implementations are performed with respect to different datasets and read length ranges. With the use of OpenCL Profiler, the profiling operations on both memory objects and kernels are discussed in the Results and Analysis section.

Table of contents

List of figures

List of tables

Nomenclature

Acronyms / Abbreviations

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DNA	Deoxyribonucleic Acid
DP	Dynamic Programming
FIR	Finite impulse response
RNA	Ribonucleic Acid
RTL	Register Transfer Level
SW	Smith-Waterman
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

In the field of biology, it has been facing a rapid development in data from many types of research in the same way as other scientific directions. Nowadays, vast data volumes related to biomedical fields are generated in sequencing centers, analytical facilities, and some individual laboratories. Thus, obtaining the relevant information within a particular time is a challenging task faced by the scientific community.

Genomic medicine is a developing medical discipline that incorporates using genomic information. In applications like clinical diagnosis, rapid species identification, and advanced therapies, genomics plays a considerable contribution to the biomedical research field. Genomic medicine includes early diagnosis, more efficient disease prevention and management, and the reduction of medication side-effects dependent on gene signatures. Therefore, new methods of developing low-cost completed genome DNA sequences play a massive role in genomics' future.

Nowadays, the focus of genome projects has moved from data production to data analysis. The central challenge is how to analyze such an amount of data rapidly and accurately. The modern sequencing methods generate data such that traditional analysis tools are not able to cope with them. Therefore, DNA analysis algorithms have to be implemented on hardware accelerators to achieve the expectations efficiently and effectively.

1.1 Background

1.1.1 Nanopore Sequencing

DNA can be described as a molecule that encodes the genetic instruction of a life. In other words, it is the blueprint of life. Therefore, the technology that does the accurate and rapid DNA sequencing have deeper impacts on human diseases and personalized medicine. Existing Non-nanopore DNA sequencing technologies require a considerable amount of sample planning and complex algorithms for data processing[3].

As they have disadvantages such as poor throughput, high cost, and limited read length as a result of the development of three generations, DNA sequencing technology is now using single molecular nanopore technology. The key benefits of nanopores include label-free, ultra-long readings, high throughput, and low material requirements. Any of these dramatically simplifies the experimental process and can be used for DNA sequencing applications effectively. Moreover, nanopores as single-molecule sensing technologies have great potential uses for the study of ions, DNA, RNA, peptides, proteins, drugs, polymers, and macromolecules[4][5].

Nanopore technologies can be broadly classified into two categories as biological and solid-state. Although Biological nanopores have been widely used in single-molecule detection, disease diagnosis, and DNA sequencing, the included protein pores have a constant pore size, profile, and lack of stability[6]. But it has been shown that solid-state nanopores have many superior advantages over their biological counterparts, such as chemical, thermal, and mechanical stability, size adjustability, and integration[7].

However, real-time DNA sequencing is currently a major challenge[8]. However, the new generation (third generation) of sequencing technology will produce ultra-long DNA 'reading' from single molecules in real-time. For example, Oxford nanopore Technologies (ONT) produces a pocket-sized sequencing device called MinION, a relatively affordable and compact sequencing device capable of sequencing in remote areas with no network access even at the point of treatment[1].

While the current is sampled and digitized, ONT sequencing devices measure DNA strand passing through biological nanopores composed of recombinant proteins[9]. But, there can be some stochastic noise due to several factors such as homopolymers (same base repeating multiple times), contaminants in the sample, entanglements of long DNA strands, and depletion of ions in the measured signal[10].

Furthermore, the signal can be warped due to the variations of the DNA strand's movement speed through the pore. Then, the conversion from the raw signal to the character representation is done using artificial neural networks, generating a typical

accuracy >90% for single reads[11]. This conversion is called base-calling, and the base-callers are the software tools that perform the conversion[9].

The particular sequence is aligned to a reference sequence which consists of a previously generated consensus sequence once the nanopore read is base-called. Here the sequence alignment takes in global optimization algorithms to detect the most similar target and to compare the differences between sequences. Moreover, the error-rate of nanopore sequencing is relatively high when it is compared to biologically occurring variation in individual genomes. Therefore, the derived sequence alignments are distinct in nature from previous sequencing technologies.

After that, the 'polishing' downstream processing step is taken into place. It utilizes both the base-space alignment results and the raw signal. Additionally, in order to recover the lost biological information during base-calling, it reuses the raw signal. This polishing step is used to correct errors during base-calling or to detect modified nucleotide bases such as DNA methylation.

However, it has shown that identification of genetic variants can be increased up to an accuracy of more than 99% by using raw signal data by performing multiple overlapping reads[12][13]. Alternatively, since it reuses the raw signal data, the downstream analysis could correct for base-calling errors as well.

On the other hand, during the base-calling, some critical biological information is lost. Some baseline models cannot handle methylated data because either they are trained on unmethylated sequences, or they abstract non-canonical bases. Which means these molecules may be classified as unmethylated bases erroneously. Here, the process of identifying methylation is known as methylation calling.

1.1.2 Methylation Calling

Nanopore sequencing offers real-time analysis at the expense of a higher error rate. It is predominantly caused by the conversion process of the raw signal into DNA bases via probabilistic models, and it is referred to as base-calling. To overcome base-calling errors, the raw signal can be revisited a posteriori.

The raw signal can be re-examined a posteriori to resolve base-calling errors. Such polishing could be accurate by aligning the raw signal with a biological reference sequence for base-calling errors and thus detect raw signal idiosyncrasies by comparing observed signal rates with predicted levels at all associated positions[14].

As mentioned earlier, base calling causes essential biological information to be lost. Some base-calling models cannot handle methylated data either because they are trained in non-methylated sequences or because non-canonical bases are abstracted. Such molecules

can also be wrongly marked as non-methylated bases. The methylation calling means the process of identifying methylation[14].
Three steps are to be followed for a given read under methylation calling and steps are performed for each reading in the data set,

1. Event detection
2. Signal-space alignment
3. Hidden Markov Model (HMM) profiling

Event detection is the time series segmentation of the raw signal based on sudden signal level changes. Each of these segment is called *event* and it is denoted by mean ($\mu_{\overline{x}}$), standard deviation ($\sigma_{\overline{x}}$) and the duration of the raw signal samples ($n_{\overline{x}}$). Events are aligned to a generic k-mer model signal to obtain a better match between events and the raw signal. Nanopolish software package accomplishes this task using the Adaptive Banded Event Alignment (ABEA) algorithm. The alignment between the events and the k-mers in a reference genome can be subjected to Hidden Markov Model (HMM) profiling to identify if a given base is methylated or not[1].

k-mer	mean	sd
AAAAAA	μ_0	σ_0
AAAAAC	μ_1	σ_1
AAAAAG	μ_2	σ_2
AAAAAT	μ_3	σ_3
AAAACA	μ_4	σ_4
.	.	.
.	.	.
.	.	.
.	.	.
TTTTTT	μ_{4095}	σ_{4095}

Fig. 1.1 An Example of k-mer Model[1]

1.1.3 Adaptive Banded Event Alignment Algorithm (ABEA)

Dynamic programming (DP) is used to find out the optimal alignment between two biological sequences. Famous algorithms that use DP are Needleman–Wunsch (NW) algorithm and the Smith-Waterman (SW) algorithm. They are of quadratic time and space complexity. NW and SW have been utilized extensively to fine-tune the high-quality DNA sequences. Nonetheless, due to their prolonged time consumption, many heuristic changes have been made without losing efficiency to increase alignment speed.

Figure 1.2 depicts the graphical representation of DP table of the SW algorithm performing alignment between a target sequence (6 bases long t_0 to t_5) and query sequence (8 bases long q_0 to q_7). Initial values are set to zero. Then, S(i,j) – scores are computer per each sell based on a scoring scheme. Finally, trace back starting from highest score sell and ending from 0 score cell (red color path of arrows).

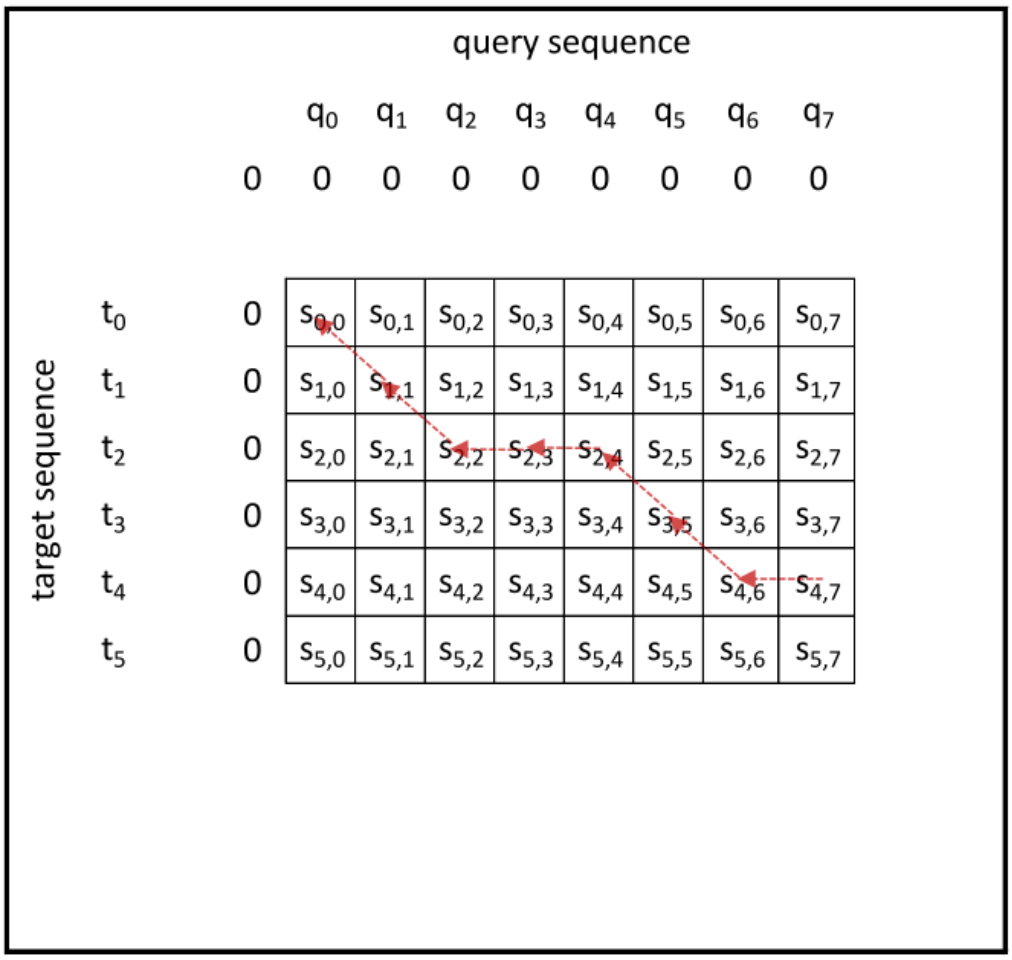


Fig. 1.2 Optimal Sequence Alignment[1]

A modern computer can quickly handle a single alignment. However, when the number of alignments that need to be processed increases, the computational complexity also increases. Therefore, to reduce the number of computations, banded alignment approaches were introduced. Nanopore gives out long reads that have 100 to 1000 times the length of a short read that are not appropriate for small static bands. The identification of long indels is a significant benefit of long readings. The strong demand for bandwidth leads to exceptionally high processing times when millions of readings are aligned.

Further, we can assume that sequences aligned to each other are essentially similar. Thus the alignment (red color path of arrows) should lie close to the left diagonal. Therefore, to reduce the number of computations, only the cells around the left diagonal are considered. This banded alignment approach is shown in Figure 1.3 with a band width of 4.

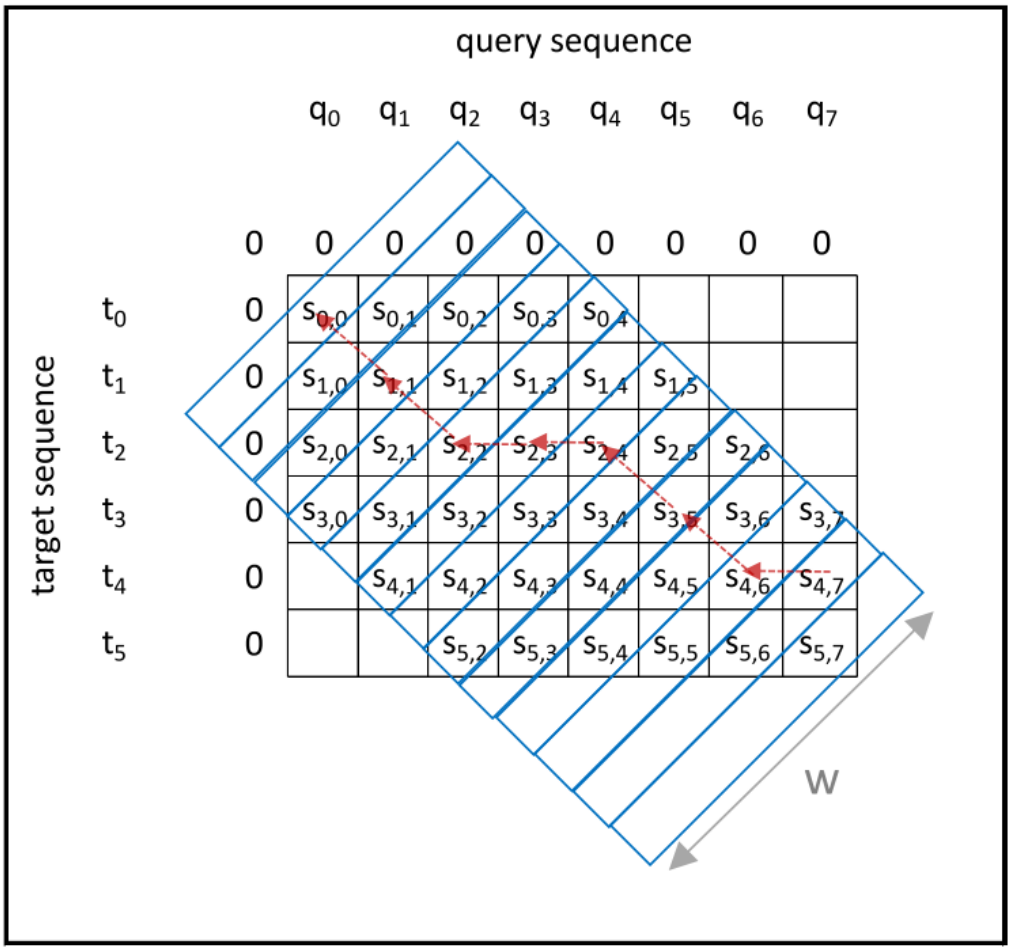


Fig. 1.3 Banded Sequence Alignment (band-width=4)[1]

The banded alignment approach is mostly suitable for short reads than long reads generated by Oxford Nanopore Technologies (ONT) and Pacific Biosciences (PacBio). Because both ONT and PacBio generated, reads have a length 100 to 1000 times bigger than short reads, and they are much noisier. Therefore, such small bands are not suitable because the alignment path can deviate significantly from the left diagonal due to high errors.

In 2017, a heuristic algorithm Suzuki-Kasahara (SK)[15] was implemented to increase the processing speed. SK uses an adaptive band scheme that allows a shorter band to accommodate such kind of alignment. However, the band is no longer enough for the whole alignment to lead to an unsatisfactory alignment. It is overcome with the use of an adaptive unit as shown in Figure 1.4.

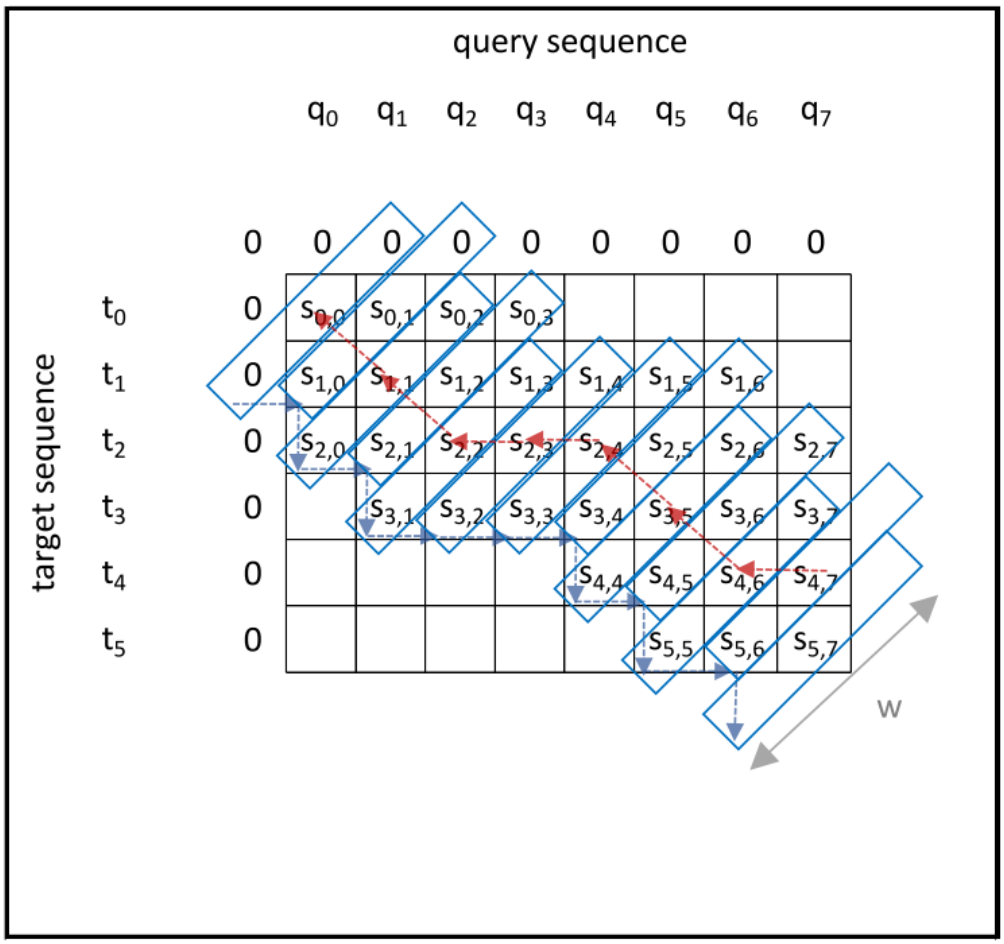


Fig. 1.4 Adaptive Banded Sequence Alignment[1]

Edited version of the SK algorithm is used in Nanopolish for event-space alignment and it is called ABEA 3.1.

1.1.4 OpenCL for FPGA Architecture and Programming

During the past couple of years, GPUs have been frequently utilized in supercomputers to accelerate various data processing types. However, the high-power usage of these devices remains a bottleneck in deploying large supercomputers. For this reason, Field-Programmable Gate Arrays (FPGA) are a promising alternative to GPUs specifically because of their relatively low power consumption.

The most common approach to achieve better performance is by assigning the computationally intensive task to hardware and exploiting the parallelism in the algorithm[16]. Field Programmable Gate Arrays (FPGAs) have proved an effective platform for the implementation of these algorithms. FPGAs are in-between general-purpose processors and ASICs on the spectrum of processing elements[17].

Historically, hardware developers used hardware description languages (HDL), also known as High-level programming environments like Verilog and VHDL, to program hardware at register transfer level (RTL)[18]. When the application gets large, this approach can be complicated and frustrating even with a proper implementation structure. The time to design, verify, and optimize (time-to-market) an application using RTL is significant and requires previous hardware design experience, which implies increased development cost.

This reason forced developers to come up with high-level synthesis (HLS) tools like Intel OpenCL (Open Computing Language) HLS and Xilinx Vivado HLS[19]. These tools provide the ability to write applications in high-level programming languages such as C/C++ and SystemC and then generate the RTL design of the program to support hardware like FPGAs. HLS reduces the time-to-market and increases developers' productivity by taking the overhead of deciding the microarchitectural detail of the FPGA design.

In[20], it has been shown that the OpenCL computing paradigm is a viable design approach for high-performance applications on FPGAs, and it is a framework for parallel programming and includes a language, API, libraries, and a runtime system to support software development. OpenCL is developed to allow parallel computation to accelerate, addressing a wide range of platforms[21]. The programs written in OpenCL can then be converted to RTL designs to support a wide variety of platforms. An OpenCL platform comprises one host and one or many devices, which are computed units that may consist of multiple processing elements (PEs).

OpenCL platform model[2], an abstract hardware model for devices (Figure 1.5). One platform has a Host and one or more devices connected to the host. Each Device may have multiple compute units with multiple processing elements.

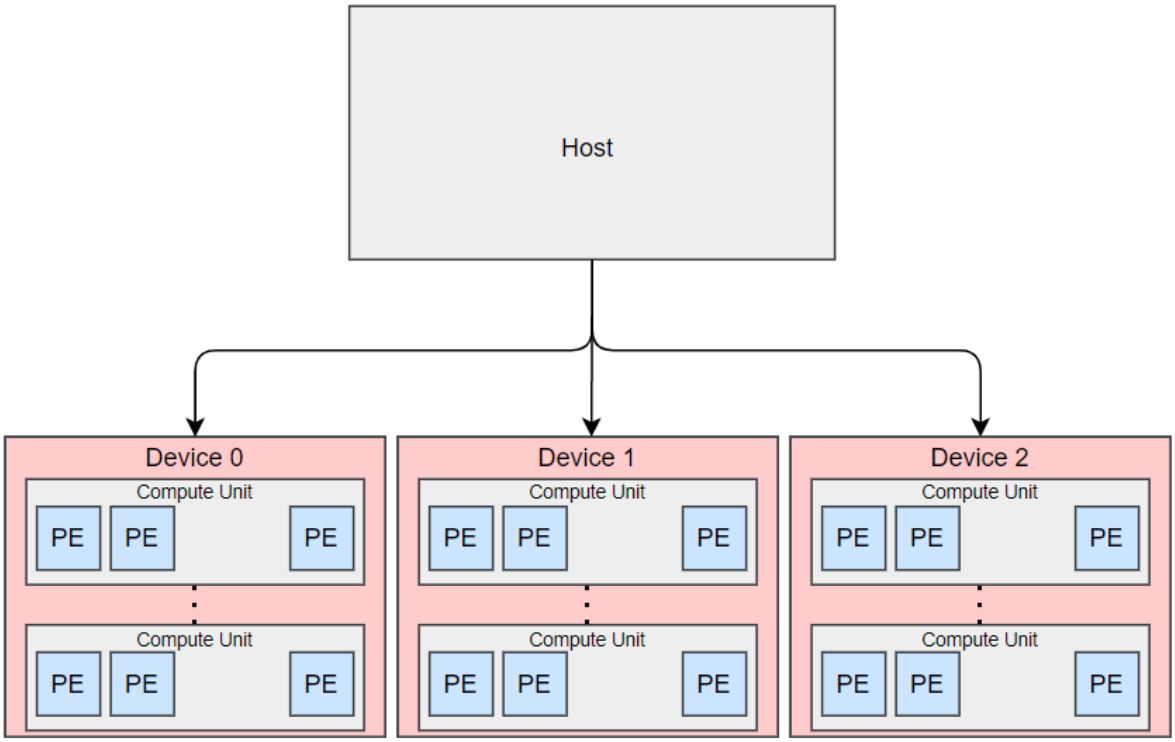


Fig. 1.5 OpenCL Platform Model

The host program performs the following tasks

- Allocate memory on the FPGA
- Transfer the input data from the host to the FPGA
- Execute the kernel
- Transfer the output results from the FPGA to the host
- Release the allocated memory

Figure 1.6 illustrates the schematic diagram of the Intel FPGA SDK for the OpenCL programming model.

The execution model (Figure 1.7) shows the communication mechanism between the host and devices in the context environment. The host submits work to devices and manages the workload in the context using the OpenCL API platform layer. The command queue is the communication media that the host uses to read, write, and execute.

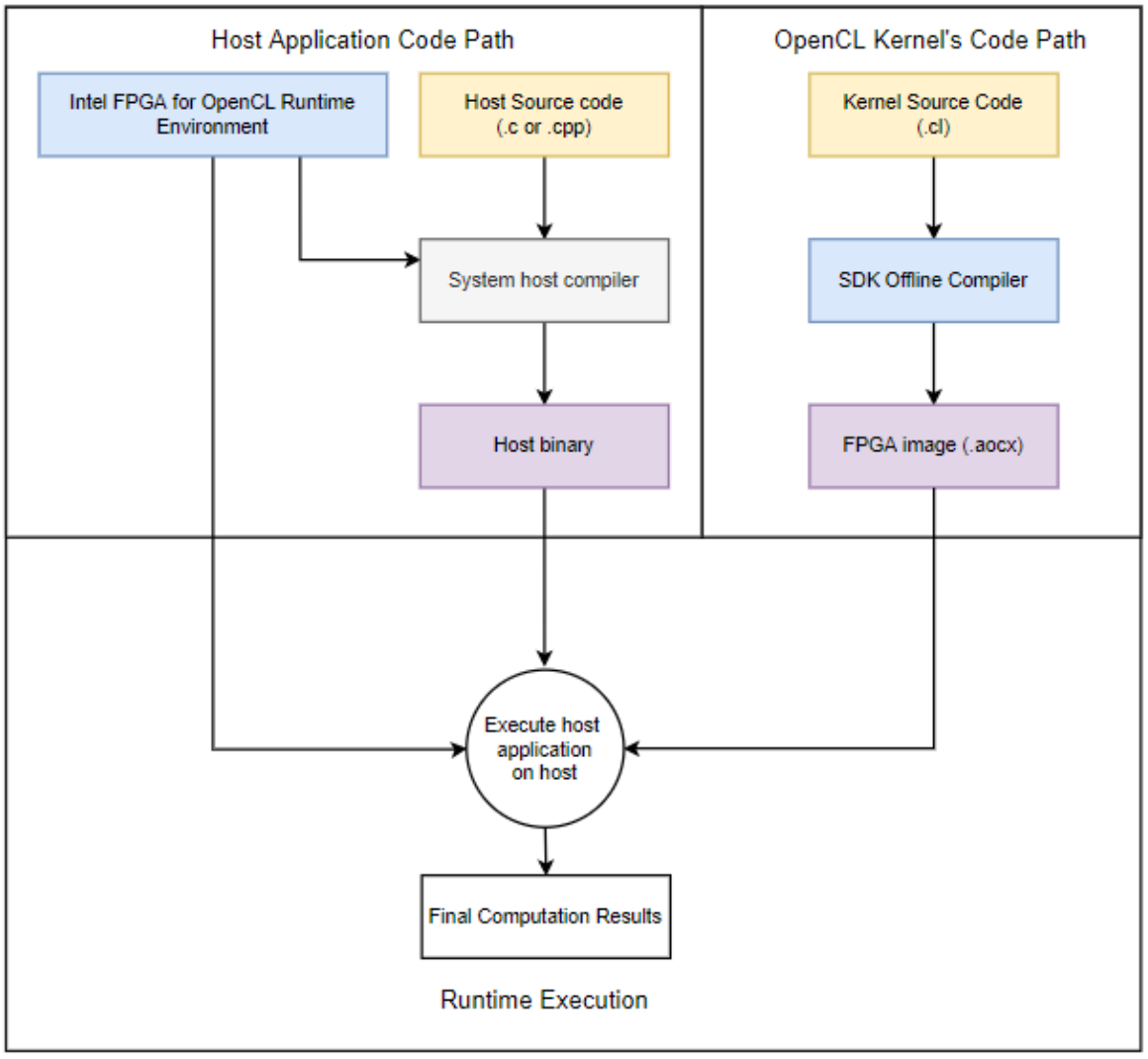


Fig. 1.6 Schematic diagram of the Intel FPGA SDK for OpenCL programming model

OpenCL for FPGA uses two types of kernels, namely 'Single work item' kernels and 'NDRange kernels'[2] (Figure 1.8). There is only one work item in a single work item kernel, while the NDRange kernel has multiple work items. The Single work item kernel shares data among multiple loop-iterations using a private memory, while NDRange kernels share data among multiple work-items by using local memory.

In[2], it is emphasized loop unrolling, optimizing floating-point operations, optimizing fixed-point operations, optimizing vector operations as common optimization techniques for both single work-item kernels and NDRange kernels.

The memory hierarchy of OpenCL is shown in Figure 1.9. The host memory is accessible only to the host. The global memory is accessible to both the host and the

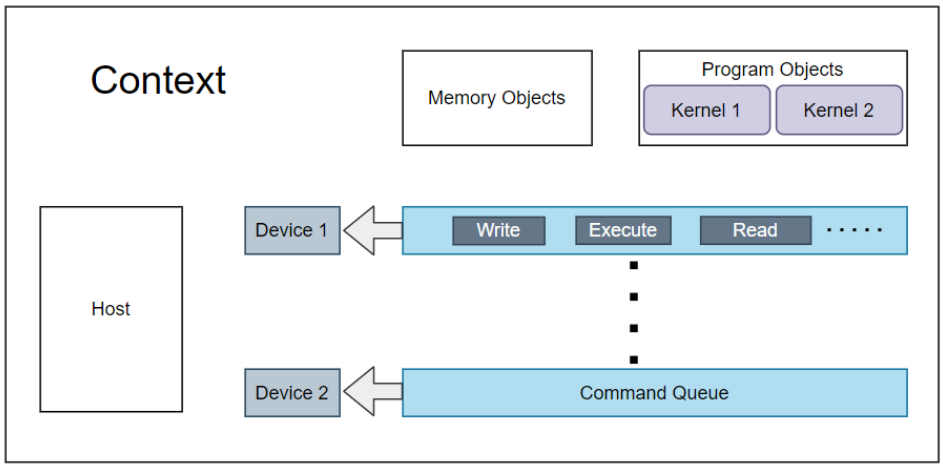


Fig. 1.7 OpenCL Execution Model

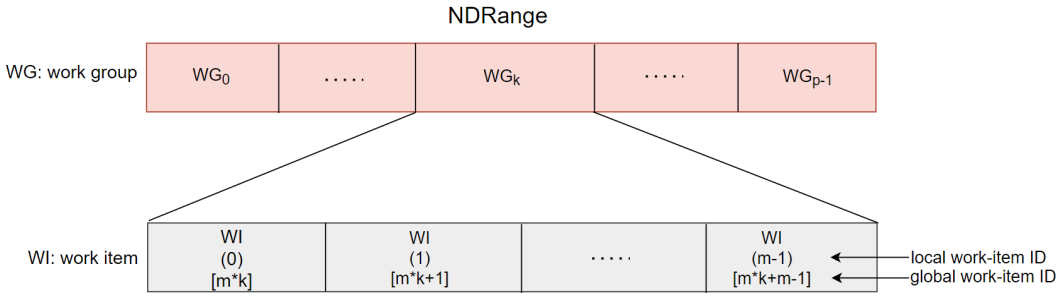


Fig. 1.8 OpenCL Kernel Programming Model[2]

device. Constant memory is read-only and only accessible to the device. Each workgroup has a local memory shared by each work item, and a work item has its own private memory.

1.2 Problem Statement

The dynamic programming algorithm ABEA is a time-consuming step in nanopolish software packages that come under nanopore DNA sequencing. It has been discovered that it consumes ≈70% of the total CPU time in execution[1]. The GPU implementation of ABEA has limited to NVIDIA GPUs due to CUDA API[1]. Therefore, we identified that it is essential to analyze the ABEA algorithm's run-time on different hardware platforms and investigate strategies to reduce the run-time.

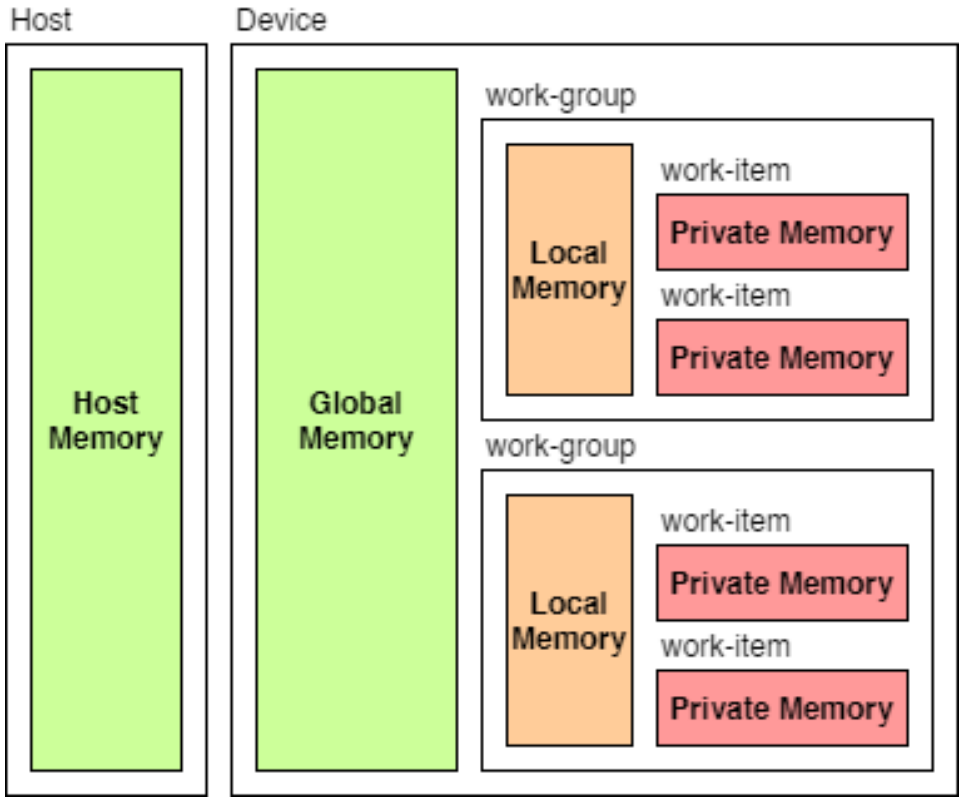


Fig. 1.9 OpenCL Memory Model

Moreover, GPU-oriented HPC systems consume high power consumption compared to FPGAs[22]. Therefore, we identified that it is necessary to deploy the ABEA algorithm to run on a low-power-consuming hardware platform.

1.3 Proposed Solution

We propose to address the limitations of the latest GPU version of ABEA. Using FPGA-based implementation, we hope to achieve better performance and power utilization. Using OpenCL, we develop a portable version of the algorithm, which we can deploy in FPGAs, GPUs, CPUs, and other processors and accelerators.

Chapter 2

Related Work

2.1 GPU Accelerated Adaptive Banded Event Alignment Algorithm

Previous research[1], which is done under the objective of accelerating ABEA, deployed an accelerated version of the algorithm on GPUs using CUDA. In this work, high read length variability was one of the critical problems solved by various memory optimizations and a heterogeneous computing approach that uses both CPU and GPU. They have achieved 3-5x performance improvement on the CPU-GPU system compared to the original CPU version of the nanopolish software package.

As of now, the complete methylation calling of a human genome can be performed in real-time while the nanopore sequencer is operating on an embedded system such as in an SoC equipped with an ARM processor and an NVIDIA GPU.

They have re-engineered the original Nanopolish methylation detection tool to efficiently utilize existing CPU resources, which they have referred to as f5c. According to its results, f5c powered by GPU accelerated ABEA can process the output from the rest of the pipeline on a single NVIDIA TX2 SoC, at a speed of (>600 Kbases per second) to carry on with the sequencing output as shown in Figure 2.1.

Also, they have shown that if the original Nanopolish was executed on the NVIDIA TX2 SoC, the processing speed is limited to $\approx$ 256 Kbases per second. Their work will not only reduce the associated costs of nanopore data processing and data transfer but will also improve the turnaround time of the final test outcome.

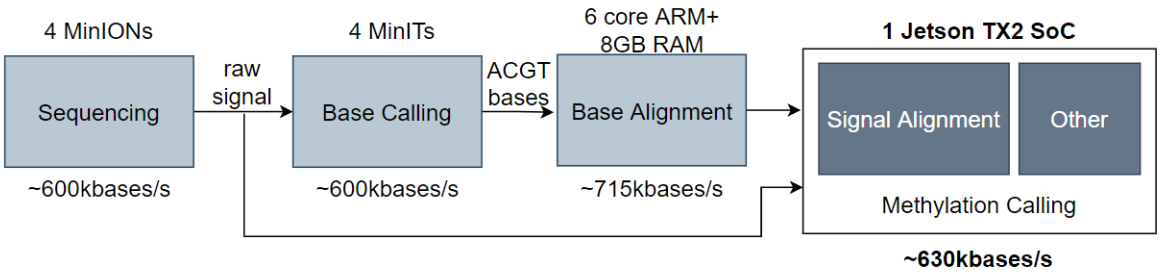


Fig. 2.1 Human Genome Processing on-the-fly[1]

2.2 Algorithms Accelerated on FPGAs

2.2.1 Smith-Waterman Algorithm Acceleration Using OpenCL

Smith-Waterman (SW)[23] algorithm is a widely used pairwise sequence alignment algorithm that finds the best possible aligned sub-segment in a pair of sequences. Accelerating SW is a great challenge in the field of high-performance computation.

In[24], Rucci et al. has presented SW implementation, which is capable of aligning DNA sequences of unrestricted size for Altera Stratix V using OpenCL. In this work, the kernel is implemented using the task parallel programming model. The alignment matrix is divided into vertical blocks. In a row-by-row manner, each block is computed from top to bottom and left to right. This approach supported by OpenCL has improved the data locality and has reduced the memory requirement for block execution. They showed that using smaller data types for kernel implementation has increased the performance and reduced resource consumption.

In[25], by Rucci et al. SW kernel, has exploited inter-task parallelism. They have utilized the SIMD (Single Instruction Multiple Data) vector capability available in the FPGA. Therefore, instead of using one sequence at a time, multiple sequences are aligned at a time. It is emphasized that the allocation of 64-byte host side buffers has improved the data transfer efficiency because Direct Memory Access (DMA) takes place to and from the FPGA.

In[26], Sirasao et al. has presented FPGA and OpenCL based acceleration to 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.

2.2.2 High-Performance Stencil Computation Using OpenCL

Stencil computations are one of the most important types of algorithms in High-Performance Computing (HPC) that are widely used applications in the fields of weather, wave, seismic, and fluid simulations, image processing, and convolutional neural networks.

In 2017, Waidyasooriya et al. proposed an FPGA platform using OpenCL for stencil computations[27] using iteration-parallel computation where multiple iterations are processed in parallel. With that, they proposed an optimization methodology to find the optimal architecture for a given application[27]. They have achieved higher processing speed relative to multicore CPU and GPU implementations and more than 60% of the peak performance given by FPGA.

In[28], Wang et al. proposed a new heterogeneous architecture design for stencil computations to improve performance with saved FPGA resources. Further, they developed a performance model to determine optimal stencil accelerator design parameters and proposed a framework to optimize and synthesize stencil computations onto FPGAs automatically. They achieved a 1.65X performance increment compared to state-of-the-art with fewer hardware resources.

In[29], Jia et al. have Optimized 1D convolution, 2D convolution, and 2D Jacobi iteration kernels for both Single-Task and NDRange modes. They were able to gain 7.1X and 3.5X speedup factors for the Sobel and Time-domain FIR filters than Altera design examples.

2.2.3 K-Nearest Neighbor Algorithm Using OpenCL

K-Nearest Neighbor Algorithm (KNN) is one of the most popular machines learning algorithms[30] and due to high computational complexity for large datasets, it has become popular in the field of high-performance computing.

In[31], Pu et al. have proposed a new solution to speed up the KNN algorithm on FPGA-based heterogeneous computing systems with OpenCL. They have introduced a specific bubble sorting algorithm based on FPGA's parallel pipeline structure to optimize the KNN algorithm. The GPU accelerated our KNN algorithm by 410 times the speed of the 4-threads CPU implementation, while FPGA achieved 148 times. When comparing the power consumption, CPU implementation could merely classify 0.015 query objects per Joule and GPU achieved 4.024, while FPGA 12.056. The energy-efficient ratio (EER) in FPGA is three times better than the GPU.

Two different implementations of the energy-efficient approach for the KNN algorithm are presented[32] by Muslim et al. Furthermore, they have compared the performance

between GPU and FPGA implementations of the same algorithm. In the first approach, both sorting and nearest neighbor identification are performed by the host. It uses only global memory and because the independent data usage algorithm is extremely parallelizable.

In the second approach, they have implemented two kernels to calculate distances and to find k-smallest distances, and return their indices at the end of execution. In this approach, FPGA implementation is the fastest, and still, it consumes lesser power and energy. It has performed seven times faster than the first approach.

2.2.4 Convolutional Neural Networks (CNN) Using OpenCL

It is challenging to apply CNNs for real-time applications with the requirement of low power consumption. Recent studies on accelerating CNNs on FPGAs, especially with high-level synthesis, have shown the advantage of reconfigurability and energy efficiency, and fast turn-around-time over GPUs.

In[33], Suda et al. proposed a systematic design space exploration methodology to maximize the throughput of an OpenCL based FPGA accelerator for a given on-chip memory, registers, computational resources, and external memory bandwidth. They implemented a CNN with fixed-point operations on FPGA using OpenCL and identified critical design variables that affect the throughput and execution times. Then it was modeled and validated as a function of those variables. They proposed and demonstrated a systematic way to minimize the total execution time of large-scale CNNs: AlexNet[34] and VGG on FPGAs.

In[35], Zhang et al. proposes an analytical performance model to perform in-depth, quantitative analysis on resource requirements and performance of CNN classifier kernels and available resources on modern FPGAs. Further, they propose a new kernel design to address the key performance bottleneck of chip memory bandwidth identified by applying the model to analyze VGG CNN to balance memory access between computation, on-chip, and off-chip optimally. They have verified the effectiveness of the proposed model and were able to achieve the highest performance, energy efficiency, and performance density relative to state-of-art OpenCL FPGA CNN implementations.

In[36], Wang et al. introduced and demonstrated PipeCNN, an efficient FPGA accelerator that is open for researchers to be implemented on a variety of FPGA platforms with reconfigurable performance and cost. It includes a set of OpenCL kernels (namely Convolutional kernel, Data mover kernel, and other kernels) integrated using Altera's OpenCL extension channels. Throughput optimization is done by data vectorization and parallelization of CUs. Optimizations of bandwidth are achieved by introducing a sliding-

window data buffering scheme. Fixed-point arithmetic is used instead of floating-point to reduce memory bandwidth requirements and hardware costs.

2.2.5 Molecular Dynamics Applications Using OpenCL

Molecular dynamic (MD) is the area of computer simulations to analyze the physical behavior of atoms and molecules in space. The simulation is driven by the numerical results given by relatively applying classical Newtonian dynamic equations to atoms or molecules.

In[37], they propose an OpenCL-based heterogeneous computing system with an FPGA accelerator. They have implemented the most time-consuming, non-bonded interaction computations using the FPGA accelerator. Since the atoms move with time, the number of atoms in a cell is not constant, making the loop boundaries data-dependent. So it is not suitable for OpenCL implementation. To get around this issue, they introduced a pipelined architecture replacing nested loops.

In[38], they tried to experiment and determine whether the OpenCL implementation is competitive with an HDL implementation of MD using several designs with pipelines:

- Single-level implementations in Verilog and OpenCL,
- A two-level Verilog implementation with the optimized arbiter,
- Several two-level OpenCL implementations with different arbitration and hand-shaking mechanisms, including one with an embedded Verilog module.

2.3 OpenCL Kernel Optimization on FPGAs

It is mandatory to consider best practices when deploying OpenCL kernels on FPGAs. Imperfect approaches can lead to an underutilization of the FPGA computing capabilities.

In[39], Shata et al. presented three-fold to optimize OpenCL kernels on FPGAs. They can be summarized as follows.

- Avoiding Global Atomic Operations: Imperfect memory hierarchy design of the implementation leads to memory stalls and insufficient memory bandwidth. As mentioned in [39], to reduce global atomic operations, they have to do within the same work-group. After finishing all the atomic operations locally, it has to fetch the results back to the global memory. This will reduce the number of global memory accesses by work-items.

- Specifying work-group size: Intel FPGA runtime and offline compiler enforce some constraints on the work-group size and the number of concurrently running work-groups. When the OpenCL kernel contains a barrier at compilation time, the maximum size of the work-group is set to 256. At runtime, the work-group size is set to 1 when the kernel contains a barrier, queries the local work-item id, or uses local memory. To get the best performance from such kernels, the work-group's size should be appropriately adjusted when enqueueing the kernel at the host side.
- Alignment of the Allocated Host Memory: It is recommended to align the host buffers that will be read/written by the FPGA device to at least 64 bytes. This allows direct memory access (DMA) transfer, and hence improving the data transfer efficiency.

According to Best Practices Guide[40], provided by Intel FPGA SDK for OpenCL, the following guidelines have to consider to optimize global memory access.

- The default OpenCL offline compiler configures global memory in a burst interleaved configuration. It prevents load imbalance by ensuring that memory accesses do not favor one external memory bank over another. However, OpenCL allows users to manually partition memory banks, depending on the application to achieve better performance.
- To minimize global memory accesses, it recommends first preload data from a group of computations from global memory to constant, local, or private memory. Then perform the kernel computations on the preloaded data and then write the results back to global memory.
- If the FPGA board offers heterogeneous global memory types, the user can deploy different memory accesses with varying efficiencies.

2.4 Takeaways from Related Work

Following are the key points in terms of optimizing FPGA-based accelerations using OpenCL.

- Larger pipelines lead to better performance but at the cost of higher resource consumption.
- The use of smaller data types for kernel code results in better performance and less resource consumption on FPGAs.

- Data level parallelism is important to achieve successful performance rates at the expense of a moderate increase in resource usage.
- When considering DNA sequencing algorithms, larger workloads benefit all kernels regardless of sequence similarity.
- When considering power efficiency, most of the FPGA accelerators are better than GPU-based implementations.
- The exploitation of OpenCL memory hierarchy such as the private memory offers considerable benefits, although constant memory usage hardly improves the performance.
- Data transfer time between CPU and FPGA is a performance bottleneck. This can be eliminated using unified memory space for CPU and FPGA.
- Since OpenCL allows multiple devices exploitation; the workload can be distributed among multiple FPGAs to achieve better performance.
- Unlike the existing HDL-based alternatives, the OpenCL paradigm facilitates portability.

Chapter 3

Design and Implementation

In the first phase of our project, we followed the procedure described in[1]. The ultimate goal of this approach is to optimize the ABEA algorithm. Following, describes the methodologies that we have used based on[1]. In the latter phase, we have fine-tuned the implementation based on FPGA specific optimization techniques.

In this system, the application starts out executing on the CPU, and then the CPU launches kernels on FPGA. The data transferred between host and the device is done using the PCIe bus to minimize the impact of communication.

The Pseudo code of the CPU implementation of ABEA algorithm in shows in Figure 3.1. The align function consists of four main steps.

1. Initialization of first two bands (b_0 and b_1)
2. Outer loop: iterates through rest of the bands from top-left to bottom-right of the DP table
3. Inner loop: iterates through each cell in current band(b_i)
4. Backtracking: find the actual alignment (event-ref pairs)

3.1 NDRange Kernel Implementation

When a massive problem is broken down into finite number of sub-problems, each individual component becomes simple in implementation and quick in execution. The individual threads are called work items and the complete amount of work to do is called the NDRange.

Input:
ref[]: the base-called read (1D char array)
events[]: event table containing $\{\mu_{\overline{x}}, \sigma_{\overline{x}}, n_{\overline{x}}\}$ *of each event—1D* {*float,float,float*} *array*
model: pore-model
Output:
alignment[]: alignment denoted by a list of {*event index,k-mer index*}*—1D* {*float,float,float*} *array*

```
1: function ALIGN(ref, model, events))
2:     initialise_first_two_bands(score, trace, ll_idx)
3:     for i ← 2 to n_bands do
4:         idir ← 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. 3.1 Adaptive Banded Event Alignment(ABEA)

NDRange kernel programming model does not support thread-level parallelism like in GPUs. The compiler will generate at least one compute unit for each kernel written during the kernel compilation process. The hardware generated by the compiler is in a deep pipeline, and in each clock cycle, it attempts to start a new execution of the kernel. A typical kernel will be several hundred clock cycles deep, and there can be hundreds of work items in-flight simultaneously at different stages of the pipeline. The compiler also performs work-group pipelining, and it allows multiple work-groups to be in a flight in the same compute unit. As work-items perform different operations simultaneously for different data, the behavior of NDRange kernel is similar to multiple-instructions multiple-data (MIMD) computation.

For our NDRange kernel implementation, we followed the GPU approach taken in [1] and convert it to evaluate the performance of OpenCL implementation on FPGA.

We broke 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 following section describes pre, core, and post kernel implementations.

3.1.1 Pre Kernel

The pre kernel task is to initialize the first two bands of the dynamic programming table and pre-compute the values in the *kcache* data structure. *kcache* is a novel data structure introduced by Gamaarachchi et al. in[1].

In this implementation, we used a 2D work-item plan and assign a work-group to each read. The work-group configuration is shown in Figure 3.3.

The pseudocode of pre kernel shows in Figure 3.2. Lines 2-3 initializes the work-item index to execute. Lines 5-8 initializes the first two bands of the dynamic programming table. Lines 10-11 initializes the index of the lower-left band and lines 13-16 initializes *kcache*.

```
1:  function ALIGN_PRE(..., model))
2:      j ← thread index along x
3:      i ← thread index along y
4:      (ref, score, trace, ll_idx, kcache) ← get_OpenCL_pointers(i, ...)
5:      if j < W then
6:          score[0, j], trace[0, j] ← −∞, 0
7:          score[1, j], trace[1, j] ← −∞, 0
8:      end if
9:      if j == 0 then
10:         ll_idx[0] ← {ei_0, ki_0}
11:         ll_idx[1] ← {ei_1, ki_1}
12:         score[0, si_0] ← 0
13:         for k = 0 to numkmers do
14:             kmer ← get_kmer_at(ref, k)
15:             kcache[k] =← get_entry_from_poremodel(kmer, model)
16:         end for
17:     end if
18: end function
```

Fig. 3.2 Pre Kernel Pseudo Code

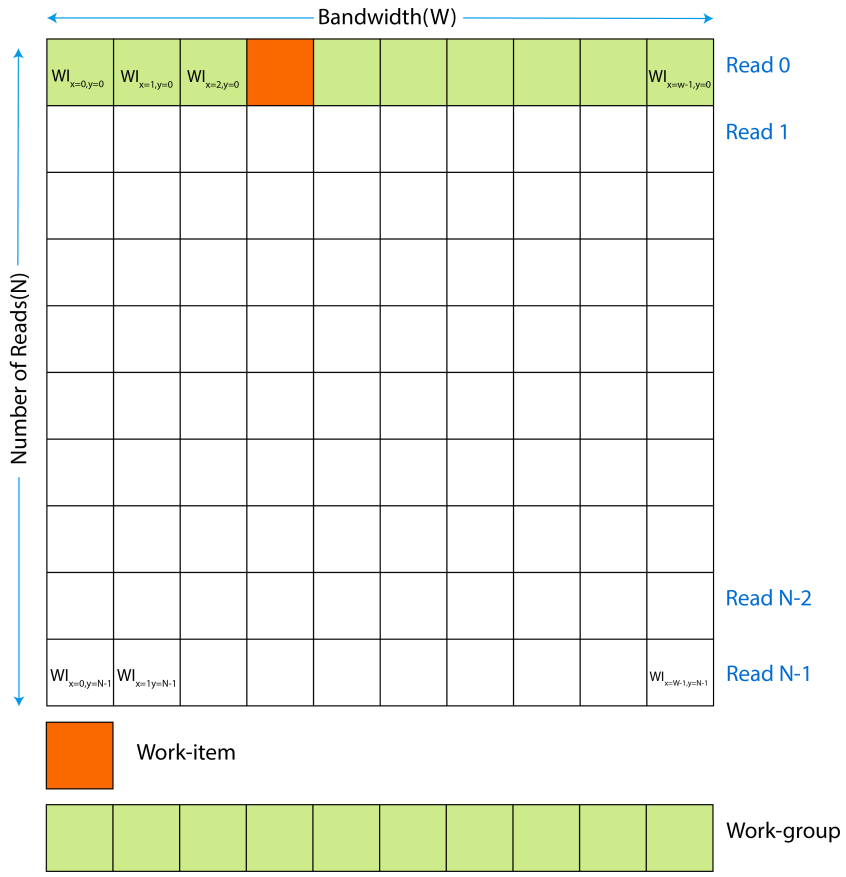


Fig. 3.3 Work-group configuration for Pre and Core kernels

3.1.2 Core Kernel

As we ported the CUDA implementation to OpenCL in NDrange implementation, we identified that the core kernel has the highest computational weights compared to the other two kernels. The core kernel is responsible for the dynamic programming table filling. The pseudo code of the core kernel shows in Figure 3.4.

After configuring work-groups as shown in Figure 3.3, ideally, reads should be pipelined because they are assigned to separate work-groups. Since there are data dependencies among bands, the compiler failed to pipeline reads. It has heavily impacted the overall performance since they are ultra-long reads($>$ 1 Mbases).

3.1.3 Post Kernel

The post kernel performs the final backtracking using intermediate results calculated from pre and core kernels. The GPU implementation has not achieved fine-grained parallelism for the post kernel since they have used one thread block for the read.

```
function ALIGN_CORE(...)
    j ← thread index along x , i ← thread index along y
    (events, score, trace, ll_idx, kcache) ← get_OpenCL_pointers(i, ...)
    n_bands ← n_events + read_len
    __shared__c_score[W], p_score[W], pp_score[W]
    __shared__c_ll_idx, p_ll_idx, pp_ll_idx
    if j < W then
        p_score[j], pp_score[j] ← score[1, j], score[0, j]
        p_ll_idx, pp_ll_idx ← ll[1], ll[0]
        __barrier()
        for i ← 2 to n_bands do
            if j==0 then
                dir ← suzuki_kasahara_rule(p_score)
                if dir == right then
                    c_ll_idx ← move_band_to_right(p_ll_idx)
                    ll[i] ← c_ll_idx
                else
                    c_ll_idx ← move_band_down(p_ll_idx)
                    ll[i] ← c_ll_idx
                end if
            end if
            __barrier()
            min_j, max_j ← get_limits_in_band(c_ll_idx)
            __barrier()
            if j ⩾ min_j AND j < max_j then
                s, d ← compute(p_score, pp_score, kcache, events, model)
                c_score[j] ← s
                trace[i, j] ← d
            end if
            __barrier()
            score[i, j] ← c_score[j]
            pp_score[j], p_score[j], c_score[j] ← p_score[j], c_score[j], −∞
            if j==0 then
                pp_ll_idx, p_ll_idx ← p_ll_idx, c_ll_idx
            end if
            __barrier()
        end for
    end if
end function
```

Fig. 3.4 Core-kernel Pseudo Code

In our implementation, the post kernel works as a single work item kernel. To get the best performance on a single work-item kernel, loop iterations have to pipeline. Since

there are several global memory accesses, pipeline stalls lead to poor performance on FPGA as well. Post kernel time can be neglected because it is very little compared to the core kernel.

3.2 Optimization Techniques for NDRange Kernel

3.2.1 Decomposition of the Algorithm into Multiple Kernels

All-in-one kernels tend to use a large number of registers. But, typical OpenCL devices only have a finite number of register file sizes. Therefore, with fewer concurrent warps, large kernels often result in poor overall performance compared to multiple kernels. Splitting kernel preferred for efficient work-group assignment too. Also, it will impact some of the optimization techniques, such as loop unrolling, since they consume a large hardware resources[40]. As a solution to this problem, we divided the algorithm into three kernels(Pre, Core, Post).

There are several work-item synchronizations associated with the core kernel. These barriers will break the pipeline, and it is a drawback in terms of work-group pipelining. To overcome this issue, we split the core kernel further into six more kernels at barrier points and investigated the run-time. We observed a massive number of kernel switches within the core kernel during the execution. It added an extra delay to the overall run-time.

3.2.2 Specifying Work-Group Size

Specifying work-group size will support the Intel FPGA SDK for OpenCL offline compiler to generate the best fitting kernel design on the board with hardware optimizations. In our NDRange kernel implementation, a two-dimensional work-item plan uses to execute work-groups parallel. Work-group width is 100 work-items which is sufficient for the ABEA algorithm, and work-group is assigned per read.

3.3 Single Work-item Kernel Implementation

The implementation of single work item kernels is very similar to a typical C program written for CPU. Single work item kernels contain loops. 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.

```
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 ← get_rank_and_accumulate(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_to_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, direction_of_which_the_max_score_came
26:                score[i,j] ← s
27:                trace[i,j] ← dir
28:            end for
29:        end for
30:    end for
31: end function
```

Fig. 3.5 Single Work Item Implementation of ABEA

Single Work Item Kernels are best suited for implementing deeply pipelined algorithms. The ABEA algorithm can be divided into three main steps. Initialization of first two bands, Filling the cells with score value for the rest of the bands, and finally traceback step which finds the best event-space alignment. Out of these three, the second step is highly compute-intensive.

The first step (line 2 - 15) initializes bands and trace arrays, initializes the first two bands and fills an array called ‘kmer_ranks’. This array is required in later computations.

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.

The number of hardware clock cycles a pipeline must wait before it can launch the successive loop iterations is called as the Initiation interval (II). This for-loop can be pipelined with an II of 1 since there are no data or memory dependency between two iterations.

The second step (line 16 - 29) 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 as explained in Section 1.1.3. Therefore, the loop (line 16) can not be pipelined due to data dependency between two loop iterations figure showing data dep. of bands. The inner loop (line 24) always goes through a band and fills the cells within a band. This loop can be pipelined with a minimum II of 1 due to the absence of data or memory dependency between loop iterations.

The final traceback step consists of a loop with high data dependency between two loop iterations. This behavior results in pipelines with an II 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 a single work item kernel on FPGA.

According to the above observations, we merged the first and second steps to build a deeply pipelined Single work item kernel. Then CPU performs the traceback step. Figure 3.6 shows a pipeline diagram including only the main for-loops in the kernel (For-loop at line 2, line 16, and line 24 in Figure 3.5). Computations related to a new read starts its execution in every clock cycle, set of bands in a read executes in a serial manner due to unavoidable data dependencies, and a new cell inside a band starts its execution in every clock cycle.

3.4 Optimization Techniques for Single Work Item Kernel

3.4.1 Loop Unrolling

Loop unrolling is replicating the body of the loop multiple times. It reduces (eliminates) the loop control overhead and loop test instructions to reduce (eliminate) the trip count of a loop. In general, loop unrolling leverages to process more data in a single clock cycle. Additionally, when there are no loop-carried dependencies the loop iterations can be executed in parallel improving the performance. But, loop unrolling increases the FPGA resource usage. OpenCL provides the capability of partially unrolling where we can specify the unrolling factor.

Fig. 3.6 Pipeline diagram of Single-work-item Implementation

Loops at line 4 and line 10 in Figure 3.5 can be fully unrolled simply putting OpenCL *pragma* "pragma unroll" before the loop. First loop in Listing 3.1 shows the code for fully unrolled loops at line 4. Note that the trip counts of the loop is constant (KMER_SIZE). We can confirm whether they have unrolled from the loop analysis Table 4.7.

When the trip count of a loop changes per out-loop iterations, the OpenCL offline compiler cannot generate a circuit fully unrolling the loop. We can partially unroll a loop using OpenCL *pragma* "pragma unroll <N>" where N is the unroll factor. The OpenCL offline compiler tries to unroll the loop at most N times. If we set N to 1, it keeps the

compiler away from automatically unrolling the loop. Second loop in Listing 3.1 shows the code for 4X partially unrolled loops at line 24. Note that the trip counts of the loop is not a constant (max_offset - min_offset).

```
#pragma unroll
for (int i = 0; i < KMER_SIZE; ++i) {
    //Loop body
}

#pragma unroll 4
for (int offset = min_offset; offset < max_offset; ++offset) {
    //Loop body
}
```

Listing 3.1 Fully and partially unrolled loops

3.4.2 Avoid Loop-carried Dependencies Due to Memory Accesses

We stated that Single-work-item kernels are deeply pipelined by the OpenCL offline compiler at the compilation time. To decide the pipeline structure, the compiler checks for loop-carried dependencies. When the kernel code becomes complex and massive due to the algorithms and data structures used, the compiler may falsely detect loop-carried dependencies. The OpenCL offline compiler generates separate hardware to handle load and store instructions that are dependent. These dependent instructions must be executed in order to preserve the correctness of the algorithm.

We can instruct the compiler to avoid falsely detecting loop-carried dependencies using the OpenCL *pragma* "pragma ivdep". The offline compiler does not generate the additional hardware for the loop below the *pragma* declaration. This might result in a reduction of logic utilization of the kernel. Further, it might reduce the II as well due to the removal of data dependency between successive loop iterations.

In our Single-work-item implementation, the OpenCL offline compiler pipelined the outer-most loop at line 2 in Figure 3.5 with a finite amount of II by default. The loop is there to iterate over the reads one by one. Since we allocate separate memory space for necessary data structures per read, we can guarantee that there are no loop-carried dependencies for this loop. Listing 3.2 shows how we instruct the compiler to avoid loop-carried dependency using the *pragma*.

3.4.3 Specifying a Loop Initiation Interval (II)

Initiation interval is the number of hardware clock cycles a pipeline must wait before it can launch the successive loop iterations. An optimally pipelined loop has a II of 1. Then, one loop iteration is launched every clock cycle. OpenCL enables the designer to specify the II for a particular loop. Declaration of OpenCL *pragma* "pragma ii <N>" instructs the offline compiler to achieve II of N at the compilation time. Setting a higher value for N will instruct the compiler to be less aggressive in optimizing II. Being less aggressive on loops which are short-running (ex: initializations) than the loops long-running (ex: compute intensive part of the algorithm) will allow to achieve a higher f_{max} for the overall kernel design.

Listing 3.2 shows how we instruct the compiler to optimize the outer most loop to achieve II of 1 using the *pragma*.

```
#pragma ii 1
#pragma ivdep
for (int read_i = 0; read_i < n_reads; read_i++) {
    //Loop body
}
```

Listing 3.2 Fully and partially unrolled loops

3.5 Optimization Techniques For Both NDRange and Single-work-item kernels

3.5.1 Processing as Batches of Reads

In the CPU implementation align function is called per each read. But, in OpenCL implementation, a batch of reads is processed at a time to optimize the performance by data transfer overhead between the host main memory and device memory and by allowing the maximum usage of device resources for parallelization.

3.5.2 Allocate Align Memory on Host Side

The host must transfer the necessary input data to the kernels running on the FPGA. Since we are processing batch of reads at a time we want to transfer huge chunks of data. This transfer can be made efficient by allowing direct memory access (DMA). To achieve this, we allocate aligned memory on the host side for the data structures necessary for

the kernels. According to the Intel FPGA OpenCL bets practices guide, the memory must be at least 64-byte aligned to allow DMA.

We can use *posix_mem_align* function from *stdlib.h* for the allocation of aligned memory on the host side. For windows system, this can be done using *_aligned_malloc* function from *malloc.h* header file. Listing 3.3 shows an example of allocation aligned memory for an integer array of size N.

```
//On Linux
#define AOCL_ALIGNMENT 64
#include <stdlib.h>

void *array;
posix_memalign((void **)&read_ptr_host, AOCL_ALIGNMENT, N *
sizeof(int)); //Allocate
free(ptr); //De-allocate


//On Windows
#define AOCL_ALIGNMENT 64
#include <malloc.h>
void *array;
_aligned_malloc (N * sizeof(int), AOCL_ALIGNMENT); //Allocate
_aligned_free(ptr); //De-allocate
```

Listing 3.3 Allocating aligned memory on Linux and Windows

To set up aligned memory allocations, add the following source code to your host program:

3.5.3 Aligned Structs in Kernels

When structs are present in kernels, properly aligned structs results in generating the most efficient hardware by the OpenCL offline compiler. If the designer does not specify the alignment for a struct, the compiler decides it based on the size of the struct and satisfying criteria of ISO C standard. To align a struct we can use OpenCL attribute "aligned(N)" N is the alignment and it must be given in bytes.

If the fields in a struct does not match the struct size specified, the OpenCL compiler adds padding to the fields. Sometimes this might affect the efficiency of the hardware. To prevent this, we can use OpenCL attribute "packed" at the declaration of the struct

along with the "aligned" attribute. A usage of both of them in our implementation is show in Listing 3.4. The total size of 4 fields is 20 bytes (8 + 4 + 4 + 4) and minimum field size is 4 bytes. Therefore, we packed and aligned the "event_t" struct to 32 bytes.

```
typedef struct {
    uint64_t start;
    float length;
    float mean;
    float stdv;
}
__attribute__((packed))
__attribute__((aligned(32))) event_t;
```

Listing 3.4 Aligned structs in kernels

3.6 Using Intel FPGA Emulator for debugging

Usually, the OpenCL offline compiler takes few hours to compile and generate the hardware even for a simple kernel code. In our case, the compilation time for a kernel was around 3 hours to complete on the workstation we used. The Intel FPGA SDK for OpenCL emulator can be used to assess the functionality of the kernels and the correctness of the results without generating the hardware files(.aocx) for the FPGA. It takes only few seconds to compile the kernel for the emulator which makes the debugging process fast. First two commands in Listing 3.5 shows how to compile a kernel and run it on the emulator. Later two commands are to show how to compile our NDRange implementation and run it for a dataset (ecoli is the folder including the dataset given as an argument for the host application).

```
# aoc -march=emulator <kernel_code>.cl -board=<board_name> -o <
    emulator_binary>.aocx
# CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=<number_of_devices> <
    host_application_filename>

# aoc -march=emulator align.cl -board=de5net_a7 -o align.aocx
# CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=1 ./host ecoli
```

Listing 3.5 Compile and run kernels on OpenCL FPGA Emulator

However the OpenCL FPGA emulator has limitations compared to the real FPGA hardware. The .aocx file generated for the emulator is not optimized and it does not implement the actual parallel executions such as piplelines and executes sequencially. Therefore, it is significantly slower than the real hardware. Because of that we used the small ecoli dataset and a dataset with only one read filtered out from the ecoli dataset for debugging using the emulator.

3.7 Compile and Run Kernels on FPGA

After the debugging phase of the kernel, it can be compiled and run on the real FPGA hardware. We can use "aoc" command to compile a kernel code (.cl), generate optimization reports (kernel_name/reports/report.html) with other optimization flags.

3.8 Profiling the Kernels (Intel Dynamic Profiler)

Adding *–profile* option at the program compilation will generate performance counters in the FPGA to acquire the memory access information at run-time. Once the kernel executes, detailed memory access will record to *profile.mon* file. Using Intel "Dynamic profiler" software, the file mentioned above can leverage to review memory access bottlenecks. Listing 3.6 shows how to compile the kernel with profiling enabled, run the host program, and open the Intel Dynamic Profiler. Figure 3.7 and 3.8 shows the source code view and kernel view with various measurements using the Intel Dynamic Profiler.

```
# aoc -report -board=de5net_a7 <kernel_code>.cl -o <
    compiled_binary>.aocx
# ./host
# aocl report <compiled_binary>.aocx profile.mon <kernel_code>.
    cl
```

Listing 3.6 Profile kernels using Inter Dynamic Profiler

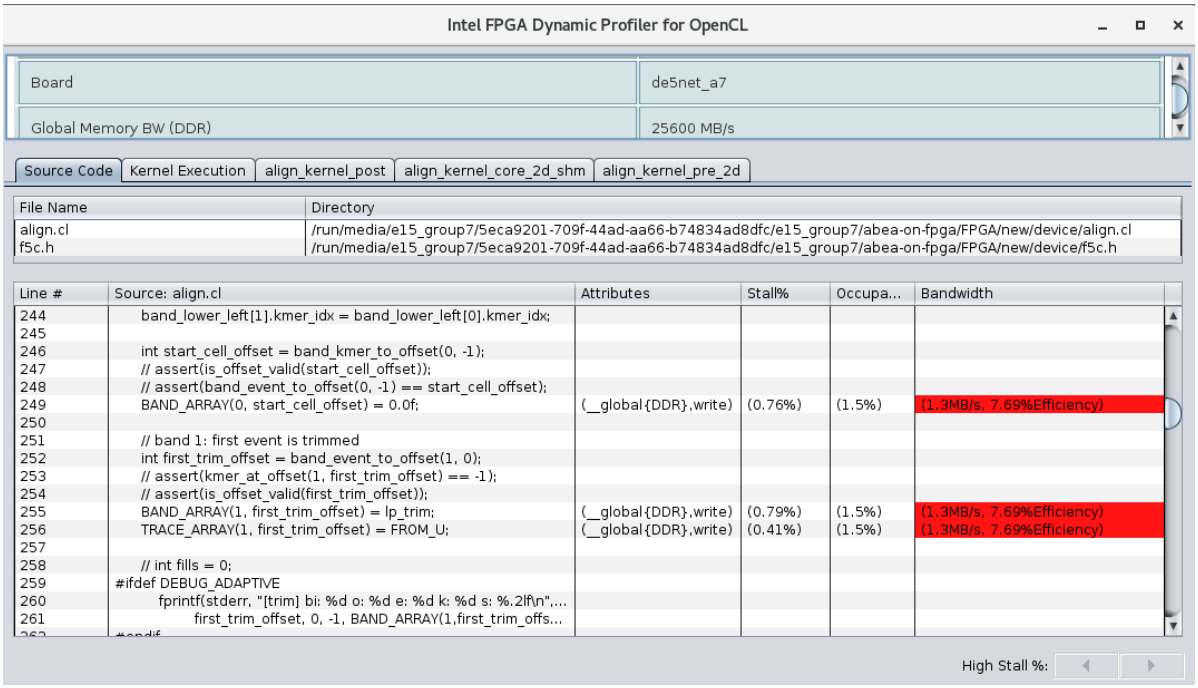
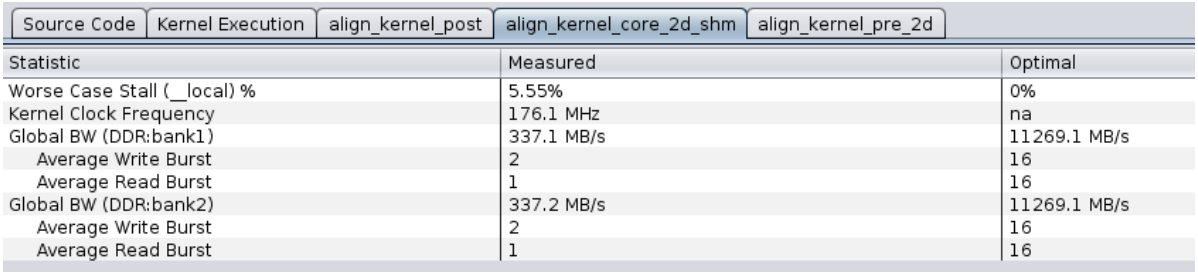


Fig. 3.7 Intel FPGA Dynamic Profiler for OpenCL - Source code



Statistic	Measured	Optimal
Worse Case Stall (__local) %	5.55%	0%
Kernel Clock Frequency	176.1 MHz	na
Global BW (DDR:bank1)	337.1 MB/s	11269.1 MB/s
Average Write Burst	2	16
Average Read Burst	1	16
Global BW (DDR:bank2)	337.2 MB/s	11269.1 MB/s
Average Write Burst	2	16
Average Read Burst	1	16

Fig. 3.8 Intel FPGA Dynamic Profiler for OpenCL - Kernel

Chapter 4

Results and Analysis

4.1 Experimental Setup

4.1.1 Isolation of ABEA Algorithm and Testbed Preparation

The isolated alignment algorithm is implemented in OpenCL and it was tested by comparing the output with dumped input and corresponding output of CPU implementation of the algorithm.

The process flow of the test-bed is illustrated in Figure 4.1.

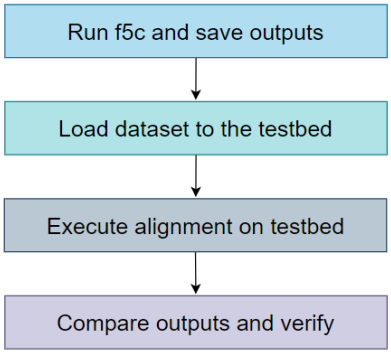


Fig. 4.1 Process Flow of the Testbed

After data dumping, it is used as the input to our implementation. The process flow is illustrated in Figure 4.2.

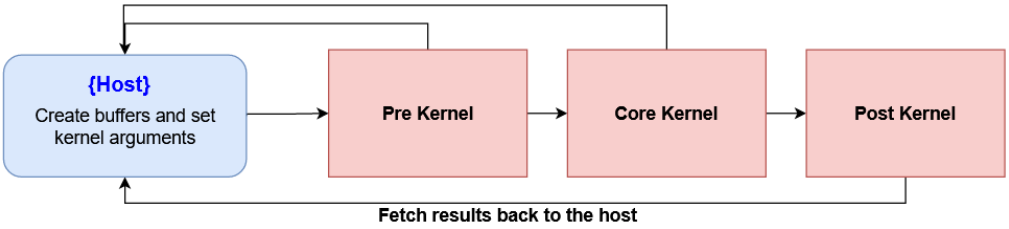


Fig. 4.2 Kernel Work Flow

4.1.2 Device Specifications

Table 4.1 shows specifications of hardware accelerators and the host PC used to obtain results.

Platform	FPGA	GPU	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	Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz 394 GB RAM
Accelerator	Intel Stratix V 4 GB RAM	Tesla K40 12 GB RAM	Tesla v100 16 GB RAM
Operating System	CentOS 7	Ubuntu 20.04.1 LTS	CentOS 7
Compiler	Intel FPGA OpenCL SDK 18.0	CUDA SDK 11.1	CUDA SDK 11.1

Table 4.1 Device Specifications

4.2 Dataset

The experimental data set is a subset of publicly available reads aligned to a 2kb region in the E. coli draft assembly (Table 4.2) and publicly available NA12878 (human genome) Nanopore WGS Consortium sequencing data as a large dataset (Table 4.3).

The datasets used for the experiments, their statistics (number of reads, total bases, mean read length and maximum read length) are listed in Table 4.4

Sample	E. coli str. K-12 substr. MG1655
Instrument	MinION sequencing R9.4 chemistry
Basecaller	Albacore v2.0.1
Region	“tig00000001:200000-202000”
Note	Ligation-mediated PCR amplification performed

Table 4.2 Details of the Small Dataset

Sample	Human cell line (NA12878)
Instrument	MinION sequencing R9.4 chemistry
Basecaller	Albacore v2.0.2
Region	“chr20:5,000,000-10,000,000”

Table 4.3 Details of the Large Dataset

Dataset	Number of reads	Number of bases (Mbases)	Mean read length (Kbases)	Max read length (Kbases)
ecoli	143	0.8	5.727	12.618
chr_22	19275	158.8	7.7	196

Table 4.4 Statistical Information of the Dataset

4.3 Results and Analysis

Here, we evaluate the current two versions of the algorithm - Single-work-item kernel and NDRange kernel implementations with CPU and GPU implementations. The reads are loaded to the device via the host one batch at a time. To obtain CPU execution time, we used the same host computer stated in section 3.2.1. Execution time for each kernel are measured using gettimeofday function from sys/time.h.

4.3.1 NDRange Kernel Observations

Our first implementation was NDRange kernel which was directly ported from the CUDA version. To identify the maximum batch size (number of bases per batch), we did the following experiment on DE5net FPGA. Using the chr_22 dataset, we tabulated the execution times while changing the maximum bases per batch (B) from 0.5M to onward.

As the results in Table 4.5, Kernel execution time slightly reduces with the increase of B, the reason is the better resource utilization of the FPGA. Therefore, in further evaluations of the NDRange kernel implementation on DE5net FPGA, we use the maximum bases per batch as 1.4Mbases. More than 1.4Mbases on DE5net FPGA results in out-of-resources error.

Dataset	B	Data(s)	Pre(s)	Core(s)	Post(s)	Total(s)	CPU(s)
small	2M	0.029	0.011	2.875	0.091	2.977	1.851
large	0.5M	6.327	3.851	908.470	69.528	981.849	275.305
large	1M	6.588	2.938	891.956	47.146	942.040	226.317
large	1.2M	6.478	2.747	884.471	43.280	930.498	220.810
large	1.3M	5.949	2.662	886.176	41.021	929.859	220.810
large	1.4M	6.365	2.662	888.559	40.107	931.328	246.243

Table 4.5 Kernel Execution Times for Different Maximum bases per batch

Considering the kernel execution times in Table 4.5, we can clearly see a lack of performance in the Core kernel. To understand the bottlenecks of the kernel code we used the Intel FPGA Dynamic Profiler for OpenCL. According to the profiler results, the Core kernel is suffering from stalls at load/store operation to the global memory of the device. Since there are multiple work items are executing simultaneously, these stalls significantly affect the execution time of the Core kernel. Further resulting poor allocation of the kernel clock frequency as shown in Figure 4.3.

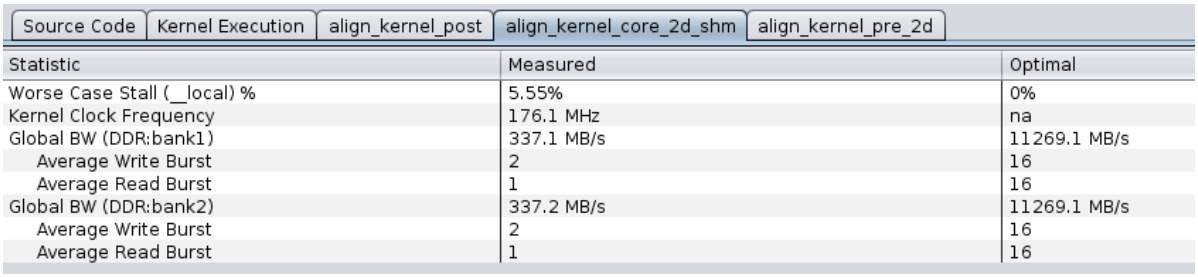


Fig. 4.3 Intel FPGA Dynamic Profiler for OpenCL - Core kernel

With the above observations such as poor kernel clock and stalls when accessing global memory, we tried minor optimization techniques that can be done by defining flags while compiling the kernels stated in Intel FPGA SDK for OpenCL Standard Edition: Best Practices Guide[40]. But, the results were almost the same as above.

Before changing the composition of kernels or inside the kernels, we suspected whether the existence of few very long reads in a batch causes the poor execution time.

Then we filtered the large dataset before feed into the 3 kernel implementation. 20 bins of reads were generated with the bin size of 10Kbases depending of the length of the reads. The distribution of number of reads per each bin is shown in Figure 4.4.

We can filter out very long reads which is the minority in the dataset and process them on the host while processing the rest of the reads which is the majority on the FPGA.

The evaluation results for those 20 bins are in Table 4.6.

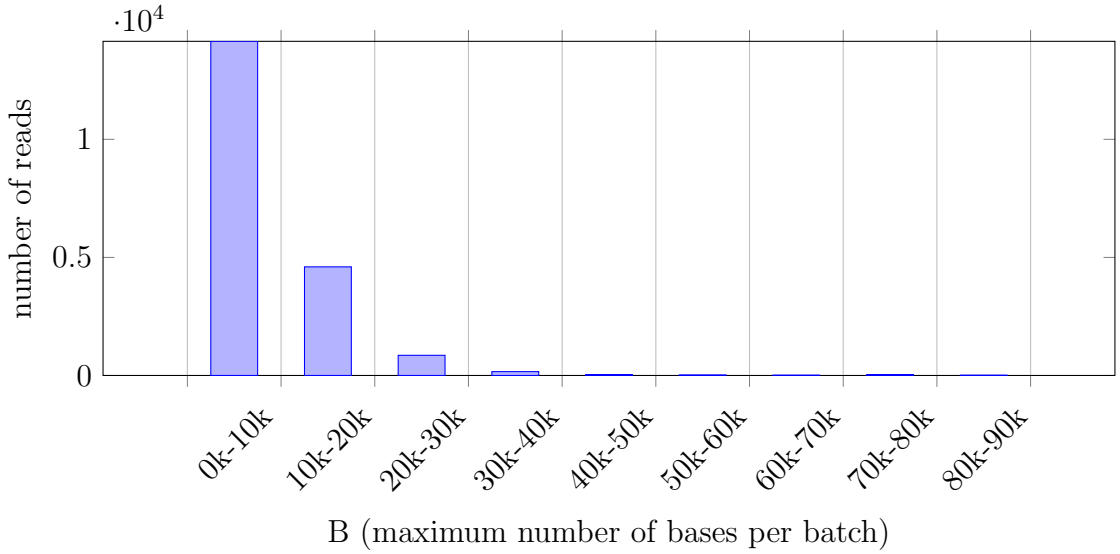


Fig. 4.4 Number of reads in each read length range

Range	Reads	Data(s)	Pre(s)	Core(s)	Post(s)	Kernels(s)	CPU(s)	Speedup
0k-10k	14150	2.326	0.729	299.041	13.507	313.277	97.362	0.31
10k-20k	4600	2.155	0.741	230.643	11.732	243.116	100.493	0.41
20k-30k	855	0.715	0.27	75.841	4.635	80.746	34.058	0.42
30k-40k	163	0.248	0.085	25.058	1.617	26.76	10.993	0.41
40k-50k	37	0.062	0.031	6.165	0.599	6.795	3.36	0.49
50k-60k	23	0.053	0.021	4.911	0.304	5.236	2.407	0.45
60k-70k	14	0.033	0.033	3.335	0.333	3.701	1.669	0.45
70k-80k	37	0.185	0.049	17.048	1.232	18.329	7.004	0.38
80k-90k	13	0.049	0.021	5.084	0.562	5.667	2.28	0.40
90k-100k	1	0.004	0.021	1.32	0.348	1.689	1.284	0.76

Table 4.6 Kernel Execution Times for Different ranges of read length

4.3.2 Discussion of NDRange Kernel Observations

As we stated in Section 3.1, the NDRange kernel programming model does not support thread-level parallelism like in GPUs. Instead, OpenCL will generate a compute unit that consists of a deep pipeline. Adding multiple compute units can increase the work-group level parallelism by pipeline stages replication and pipeline widening. But, the limitation of hardware resources on FPGA only allows us to add a maximum of two compute units.

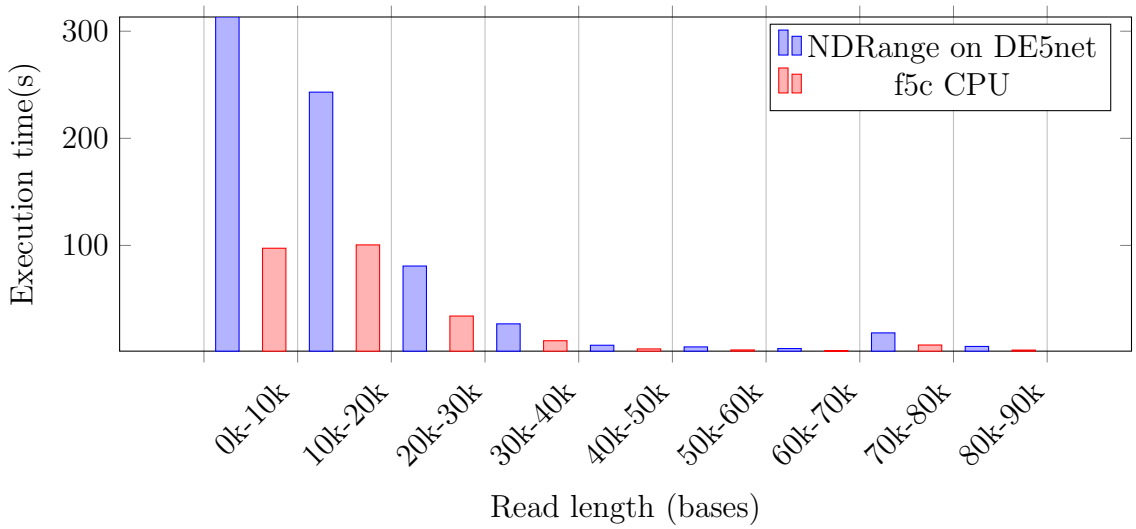


Fig. 4.5 Execution time of NDRange kernel on DE5net vs f5c on CPU

Typical GPU consists of many multiprocessors. Therefore, the NDRange version of f5c does not provide the desired performance on FPGA.

Further, NDRange kernels, work-items are considered as the unit of execution and executed in a pipeline manner. For single-work-item kernels, loop-iterations are considered as the unit of execution and executed in a pipeline manner.

If there are no data dependencies in the kernels, both approaches give similar results. But if there are data dependencies, Single-work-item kernels tend to perform better compared to NDRange kernels. Because, the work-items of NDRange kernel have to stall more often until the required data is available from the other work-items.

In ABEA algorithm, as we have stated before, score calculation of a cell in the band depends on three other score values which should be calculated before.

To address these limitations and the nature of the algorithm, we decided to change the kernel structure and build a Single-work-item kernel.

4.3.3 Single-work-item Kernel Observations

Detailed analysis of all the loops in Single-work-item kernel is shown in Table 4.7. Apart from the three of the 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. Rest of the loops are made to execute in a pipeline manner with an II of 1. All the optimization methods used in Single-work-item kernel implementation are discussed later under section 3.4 of this document.

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	Funny 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

Table 4.7 Loop Analysis of Single-work-item Implementation (DE5net)

Table 4.8 shows the estimated resources used by Single-work-item kernel in the design, all channels, global interconnect, constant cache, and board interface compiled for DE5net FPGA.

	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

Table 4.8 Estimated Resource Usage of Single-work-item Implementation (DE5net)

System viewer snapshot from the compilation report is given in Figure 4.6. It shows the interconnection of load-store units and different memory elements (Global, Local and Private memory) using control flow, memory lines. The final trace-back step executes within the host for better performance.

Comparison Between Different Implementations

Here we select a set of implementations on different platforms and compare the performance in terms of data transfer time, execution time and power consumption. Selected set of implementations are as follows.

Using each implementation, we perform event alignment on the same dataset (chr_22). Then we measure and compare the energy consumption of different hardware. 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[41] on the placed-and-routed OpenCL kernel to estimate the power usage of the board. We assume that each memory module uses a

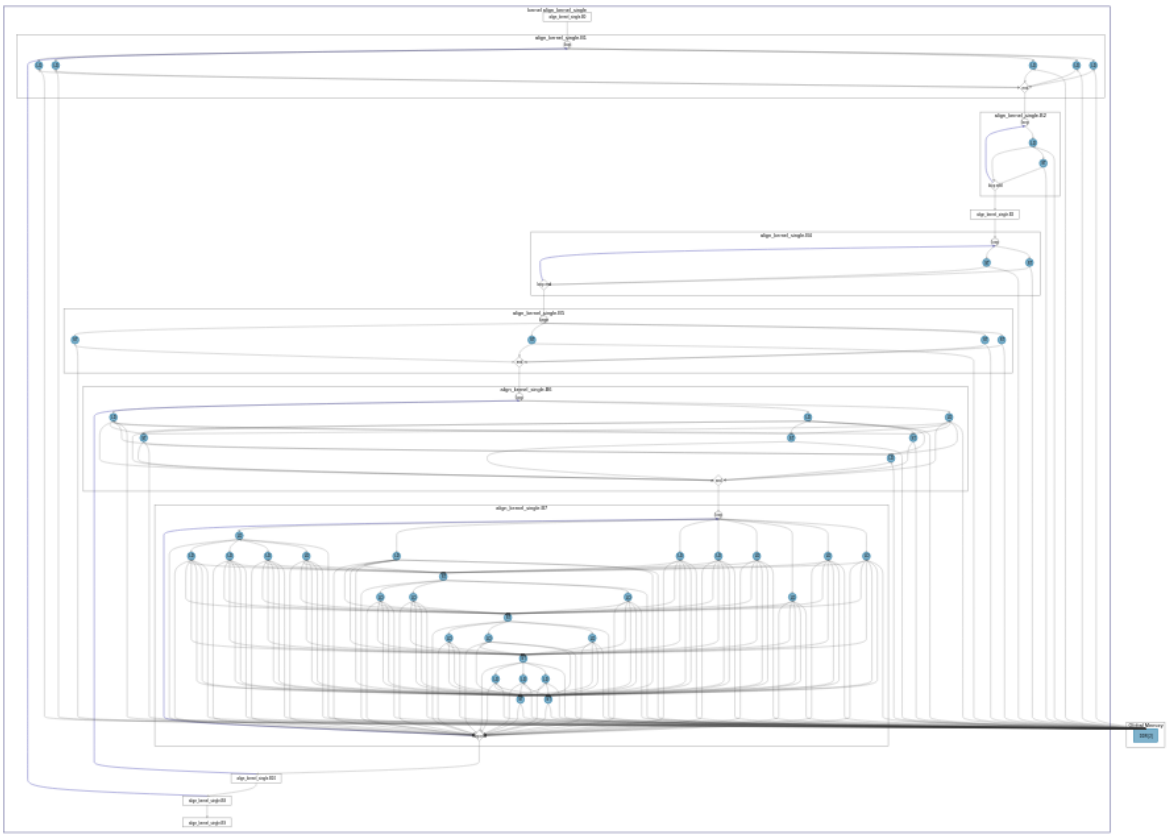


Fig. 4.6 System viewer of Single-work-item Implementation (DE5net)

cpu	f5c-cpu
cuda-k40	f5c-gpu forcing all of the reads to be computed on GPU (Tesla K40)
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 single-work-item kernel on FPGA.
swi-opt-1	Band initialization and filling rest of the bands on FPGA, post processing on CPU.
swi-opt-2	swi-opt-1 with reduced II with loop unrolling, minimizing loop carried dependencies, aligning kernel structs.

maximum of 1.17 Watts based on the datasheet of a similar memory model[42]. Hence, add 2.34 Watts to the resulting estimation value to account for the power consumption of the two memory modules.

We used nvidia-smi tool[43] to measure power draw of the Nvidia GPU cards 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[44].

When an implementation use both host and FPGA for event alignment calculations, we calculate the total energy consumed by both host and FPGA. We assume 72W as the power consumption of host which is the same we get while executing cpu implementation. Since, 72W corresponds to the power usage for all pre, core, and post computations, it is reasonable to take it as the max boundary.

Table 4.9 shows the results we obtained for different implementation. Following is an explanation of each column of the table.

data(s) - For implementations with both host and device, the summation of host-to-device and device-to-host data transfer time in seconds.
pre(s), core(s), post(s) - For implementations with separate kernels, execution time in seconds.
align(s) - Execution time without data transfer time in seconds.
total(s) - Execution time with data transfer time in seconds.
P(W) - Power consumption of the hardware in Watts.
E(J) - Energy consumption of the hardware for during the align(s) in Joules. (Summation of execution time*power for each hardware)
R - Rank given to the implementation for less energy consumption.

Figure 4.7 shows depicts the execution time of each implementation, Figure 4.8 depicts the power consumption of each implementation, and Figure 4.9 depicts the total energy consumption of each implementation for performing event alignment on chr_22 dataset.

Impl.	data(s)	pre(s)	core(s)	post(s)	ailgn(s)	total(s)	P(W)	E(J)	R
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	632.188	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

Table 4.9 Comparison between different implementations

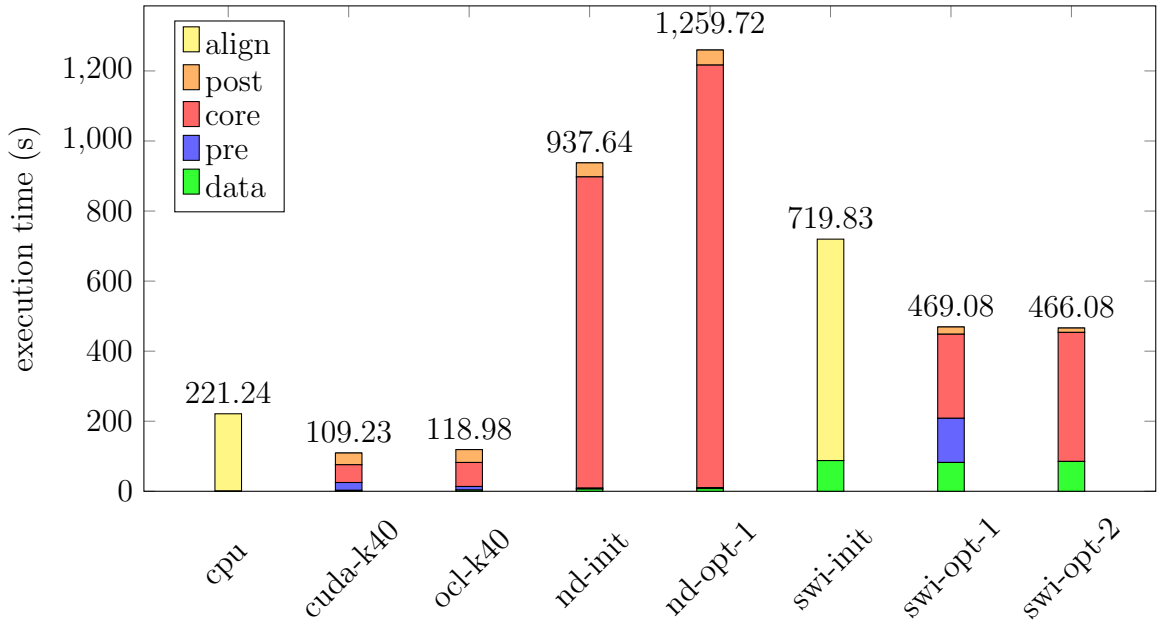


Fig. 4.7 Execution time of different implementations

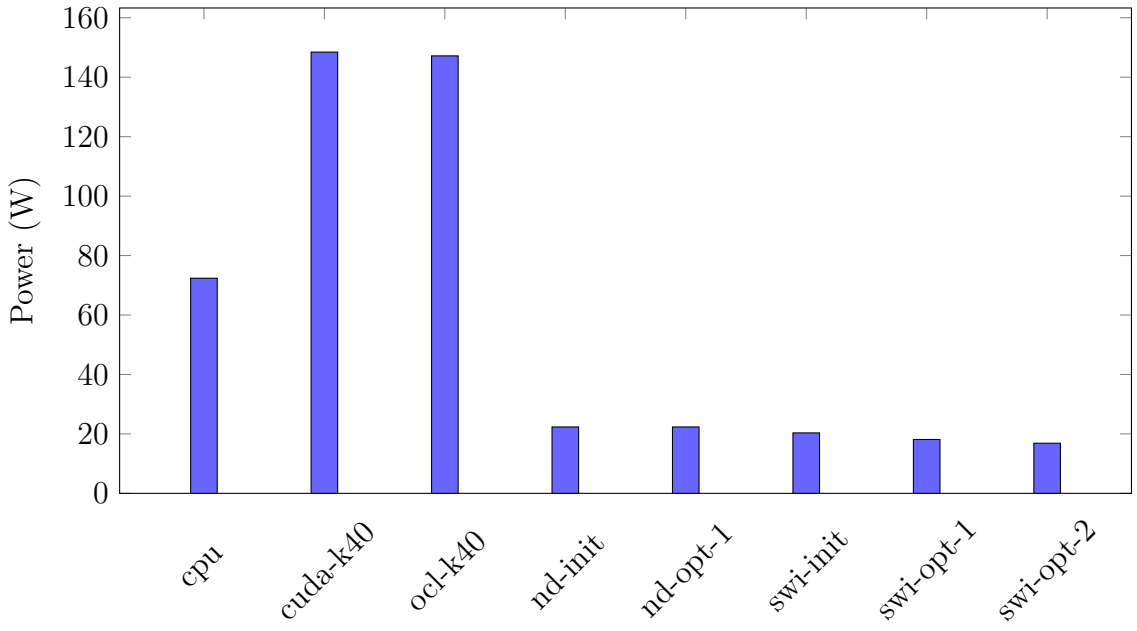


Fig. 4.8 Power consumption of different implementations

4.3.4 Discussion of Single-work-item Kernel Observations

The observations in Table 4.9 can be analyzed and justified as follows.

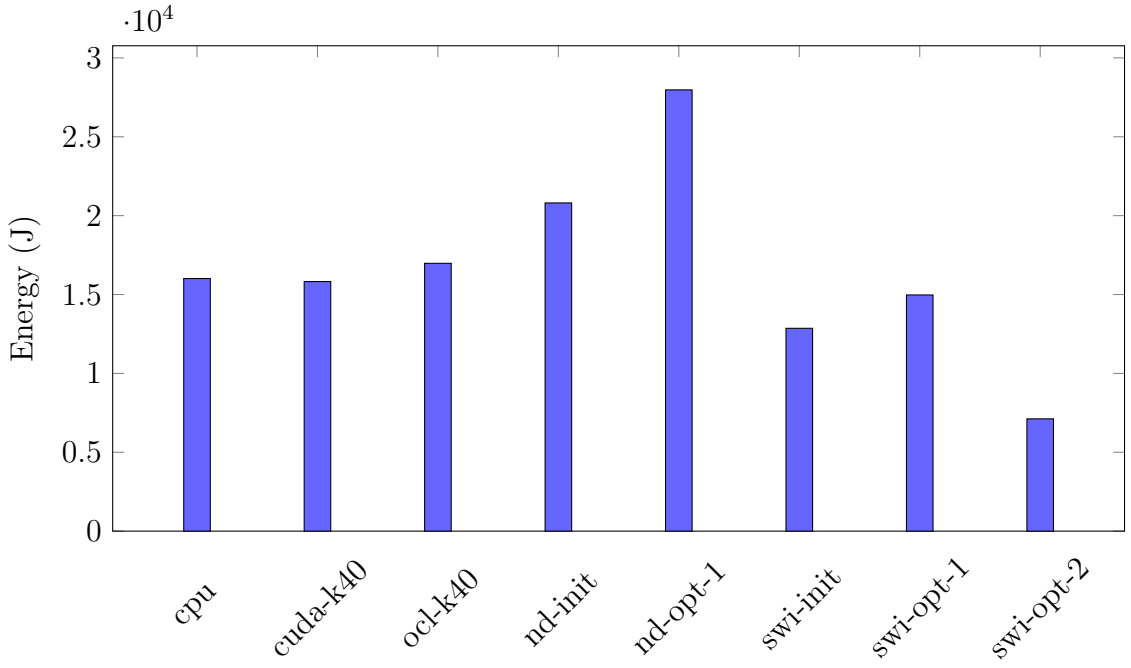


Fig. 4.9 Energy consumption of different implementations

Eventhough 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* allocate 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.

Unlike in FPGAs, in NVIDIA GPUs, Our NDrange OpenCL implementation executes in a similar programming model to CUDA, and it works as a SIMD (Single Instruction Multiple Data). When considering CUDA and OpenCL, there are minor differences. The reason for slight execution time degradation in the *ocl-k40* could be the kernel compilation during the execution time.

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 Single-work-item 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 eventhough the power consumption (estimated for DE5net) is the same.

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

As shown in Figure 4.7 In terms of execution time, GPU implementations (both *cuda-k40* and *ocl-k40*) perform better and 4x faster than *swi-opt-2* on DE5net.

However, as shown in Figure 4.9, in terms of the energy required to perform ABEA on the same dataset, Single-work-item implementations on FPGA are in lead. *swi-opt-2* on DE5net needs only 43% of the energy consumption of the GPU implementation on Tesla K40.

Chapter 5

Conclusion and Future Work

The Adaptive Banded Event Alignment algorithm is an improved version of DNA sequence alignment, which is extensively used in nanopore DNA sequencing. In the previous work, this algorithm has been parallelized and run efficiently on GPUs.

In our work, we introduce several implementations of the ABEA algorithm using OpenCL to run on FPGA. We evaluate the performance of the implementations in terms of runtime and energy consumption.

Among FPGA related implementations, Single-work-item kernel with suitable FPGA specific optimization techniques performs better than other FPGA implementations including NDRange kernel.

In terms of runtime, GPU implementations (both CUDA and OpenCL NDRange kernel) on Tesla K40 perform better and 4x faster than FPGA implementations on DE5net.

However, in terms of the energy consumption to perform ABEA on the same dataset FPGA implementations are in lead. FPGA implementation on DE5net needs only 43% of the energy consumption of the GPU implementation on Tesla K40.

Through out the work in this project, we identified the potential and ease of using HLS over traditional methods for hardware programming. We used DE5net FPGA with OpenCL 18.0 for experimenting and evaluation of 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 Hz and it is even lesser at the execution. The kernel operating frequencies of FPGAs are significantly low compared to CPUs and GPUs. The absence of power sensors in the DE5net board we had to estimate based on the circuit elements using Intel Quartus Early Power Estimator which they state gives a medium accuracy of the estimate. The true power consumption of kernels may differ due to many other reasons such as the environmental conditions.

Therefore, we believe that with the advancement of FPGA hardware and HLS tools with better optimizations methods can provide better results.

References

[1] H. Gamaarachchi, C. W. Lam, G. Jayatilaka, H. Samarakoon, and M. A. Smith, "GPU Accelerated Adaptive Banded Event Alignment for Rapid Comparative Nanopore Signal Analysis," *bioRxiv Bioinformatics*, pp. 1–37, 2019.

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

[3] Y. Feng, Y. Zhang, C. Ying, D. Wang, and C. Du, "Nanopore-based fourth-generation DNA sequencing technology," *Genomics, Proteomics and Bioinformatics*, vol. 13, no. 1, pp. 4–16, 2015.

[4] R. D. Maitra, J. Kim, and W. B. Dunbar, "Recent advances in nanopore sequencing," *Electrophoresis*, vol. 33, no. 23, pp. 3418–3428, 2012.

[5] M. Wanunu, "Nanopores: A journey towards DNA sequencing," *Physics of Life Reviews*, vol. 9, no. 2, pp. 125–158, 2012.

[6] D. W. Deamer and D. Branton, "Characterization of nucleic acids by nanopore analysis," *Accounts of Chemical Research*, vol. 35, no. 10, pp. 817–825, 2002.

[7] S. Cheley, L. Q. Gu, and H. Bayley, "Stochastic sensing of nanomolar inositol 1,4,5-trisphosphate with an engineered pore," *Chemistry and Biology*, vol. 9, no. 7, pp. 829–838, 2002.

[8] S. Liu, Q. Zhao, J. Xu, K. Yan, H. Peng, F. Yang, L. You, and D. Yu, "Fast and controllable fabrication of suspended graphene nanopore devices," *Nanotechnology*, vol. 23, no. 8, 2012.

[9] H. Lu, F. Giordano, and Z. Ning, *Oxford Nanopore MinION Sequencing and Genome Assembly*, vol. 14. The Authors, 2016.

[10] F. J. Rang, W. P. Kloosterman, and J. de Ridder, "From squiggle to basepair: Computational approaches for improving nanopore sequencing read accuracy," *Genome Biology*, vol. 19, no. 1, pp. 1–11, 2018.

[11] R. R. Wick, L. M. Judd, and K. E. Holt, "Performance of neural network basecalling tools for Oxford Nanopore sequencing," *Genome Biology*, vol. 20, no. 1, pp. 1–16, 2019.

[12] 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.

[13] N. J. Loman, J. Quick, and J. T. Simpson, "A complete bacterial genome assembled de novo using only nanopore sequencing data," *Nature Methods*, vol. 12, no. 8, pp. 733–735, 2015.

[14] R. D. Maitra, J. Kim, and W. B. Dunbar, "Recent advances in nanopore sequencing," *Electrophoresis*, vol. 33, no. 23, pp. 3418–3428, 2012.

[15] 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.

[16] S. Qasim, S. Abbasi, and A. Bandar, "Advanced FPGA Architectures for Efficient Implementation of Computation Intensive Algorithms: A State-of-the-Art Review," *MASAUM Journal of Computing*, vol. 1, no. 2, pp. 300–303, 2009.

[17] K. Nagarajan, B. Holland, A. D. George, K. C. Slatton, and H. Lam, "Accelerating machine-learning algorithms on FPGAs using pattern-based decomposition," *Journal of Signal Processing Systems*, vol. 62, no. 1, pp. 43–63, 2011.

[18] N. Street, "VERILOG HDL based FPGA design Gary Gannot and Michiel Ligthart Exemplar Logic , Inc .," 1994.

[19] C. Rust, F. Stappert, R. Künnemeyer, R. Kuennemeyer, J. Cong, and B. Liu, "From Timed Petri Nets to to Interrupt-Driven Embedded Control Software," *... -Aided Design of ...*, vol. 30, no. 4, pp. 473–491, 2003.

[20] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," *Proceedings - 22nd International Conference on Field Programmable Logic and Applications, FPL 2012*, no. March, pp. 531–534, 2012.

[21] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "OpenCL-based FPGA-platform for stencil computation and its optimization methodology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 1390–1402, may 2017.

[22] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient FPGA implementation of Open CL high-performance computing applications via high-level synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.

[23] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[24] 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.

[25] E. Rucci, C. Garcia, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matias, "SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences," *BMC Systems Biology*, vol. 12, no. Suppl 5, 2018.

[26] 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.

[27] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "OpenCL-based FPGA-platform for stencil computation and its optimization methodology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1390–1402, 2017.

[28] S. Wang and Y. Liang, "A Comprehensive Framework for Synthesizing Stencil Algorithms on FPGAs using OpenCL Model," *Proceedings - Design Automation Conference*, vol. Part 12828, no. c, 2017.

[29] Q. Jia and H. Zhou, "Tuning Stencil codes in OpenCL for FPGAs," *Proceedings of the 34th IEEE International Conference on Computer Design, ICCD 2016*, pp. 249–256, 2016.

[30] Z. Zhang, "Introduction to machine learning: K-nearest neighbors," *Annals of Translational Medicine*, vol. 4, no. 11, 2016.

[31] Y. Pu, J. Peng, L. Huang, and J. Chen, "An efficient KNN algorithm implemented on FPGA based heterogeneous computing system using OpenCL," *Proceedings - 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015*, no. July 2015, pp. 167–170, 2015.

[32] F. Muslim, A. Demian, L. Ma, L. Lavagno, and A. Qamar, "Energy-efficient FPGA Implementation of the k-Nearest Neighbors Algorithm Using OpenCL," *Position Papers of the 2016 Federated Conference on Computer Science and Information Systems*, vol. 9, no. October 2020, pp. 141–145, 2016.

[33] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J. S. Seo, and Y. Cao, "Throughput-optimized openCL-based FPGA accelerator for large-scale convolutional neural networks," *FPGA 2016 - Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 16–25, 2016.

[34] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pp. 1–14, 2015.

[35] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 25–34, 2017.

[36] D. Wang, J. An, and K. Xu, "PipeCNN: An OpenCL-Based FPGA Accelerator for Large-Scale Convolution Neuron Networks," 2016.

[37] H. M. Waidyasooriya, "OpenCL-Based Implementation of an FPGA Accelerator for Molecular Dynamics Simulation," vol. 3, no. 2, pp. 11–23, 2017.

[38] C. Yang, J. Sheng, R. Patel, A. Sanaullah, V. Sachdeva, and M. C. Herbordt, "OpenCL for HPC with FPGAs: Case study in molecular electrostatics," *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, 2017.

[39] K. Shata, M. K. Elteir, and A. A. EL-Zoghabi, "Optimized implementation of OpenCL kernels on FPGAs," *Journal of Systems Architecture*, vol. 97, no. November, pp. 491–505, 2019.

[40] B. P. Guide, "Intel® FPGA SDK for OpenCL™," pp. 1–122, 2016.

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

[42] M. O. Power, O. Temperature, S. Temperature, B.-d. D. D. Strobe, and A. Reset, "Memory Module Specifi cations CL11 Registered w / Parity 240-Pin DIMM MODULE DIMENSIONS :."

[43] "NVIDIA System Management Interface | NVIDIA Developer."

[44] Intel, "Intel® Power Governor."