Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU

Licheng Guo*, Jason Lau*, Zhenyuan Ruan, Peng Wei, and Jason Cong
Computer Science Department, University of California, Los Angeles
{lcguo, lau, zainryan, peng.wei.prc, cong}@cs.ucla.edu

***Abstract*—In genome sequencing, it is a crucial but time-consuming task to detect potential overlaps between any pair of the input reads, especially those that are ultra-long. The state-of-the-art overlapping tool Minimap2 outperforms other popular tools in speed and accuracy. It has a single computing hot-spot, chaining, that takes 70% of the time and needs to be accelerated.**

There are several crucial issues for hardware acceleration because of the nature of chaining. First, the original computation pattern is poorly parallelizable and a direct implementation will result in low utilization of parallel processing units. We propose a method to reorder the operation sequence that transforms the algorithm into a hardware-friendly form. Second, the large but variable sizes of input data make it hard to leverage task-level parallelism. Therefore, we customize a fine-grained task dispatching scheme which could keep parallel PEs busy while satisfying the on-chip memory restriction. Based on these optimizations, we map the algorithm to a fully pipelined streaming architecture on FPGA using HLS, which achieves significant performance improvement.

The principles of our acceleration design apply to both FPGA and GPU. Compared to the multi-threading CPU baseline, our GPU accelerator achieves 7× acceleration, while our FPGA accelerator achieves 28× acceleration. We further conduct an architecture study to quantitatively analyze the architectural reason for the performance difference. The summarized insights could serve as a guide on choosing the proper hardware acceleration platform.

I. Introduction

The recently developed third-generation sequencing brings new computation challenges. It produces ultra-long reads[1] in the orders of 10,000 base pairs that are much longer than previous sequencing technologies. Such long reads are crucial for genetic research [1]. For example, de novo assembly[2] from long reads could better preserve the information of a repetitive region, because the whole region might be captured in a read. To assemble these long reads back to their original genome, one crucial step, also the performance bottleneck, is to identify potential overlaps between any pair of reads. Current software tools for overlap detection take significant execution time and have not been well studied for hardware acceleration.

Minimap2 [2] is a state-of-the-art tool for pairwise overlapping. Its speed and accuracy far surpass other mainstream tools, including the ones that are specialized for a single type of alignment [2] [3] [4] [5] [6]. Like most tools, Minimap2 follows the *seed-chain-align* paradigm. However, its chaining algorithm is highly accurate even without the following fine-grained alignment step, and the alignment is not necessary in the context of detecting pairwise overlaps. The profiling results show that the chaining step costs about 70% of total time, which motivates us to accelerate that step.

It is challenging to accelerate the chaining step of Minimap2. The algorithm is a one-dimensional dynamic programming, where each input element is compared with N (64 in our implementation) previous elements to determine the best predecessor. It involves reduction operations in each iteration to find the maximal value among the numbers. However, the calculated maximal value will be immediately used in the subsequent computation. This dependency is a general problem for one-dimension dynamic programming. It will become critical paths in FPGA accelerator designs, or lead to increased instruction count and divergence on GPU.

To address this challenge, we propose to reorder the computation order of the algorithm. We serialize one reduction operation to N sequential comparisons, while in each cycle we process the comparisons from N different sliding windows in parallel. In this way, the previous critical path, which is a reduction tree, is converted to independent parallel comparisons. Based on the modified algorithm, we propose a fully-pipelined streaming architecture that effectively realizes the algorithm with initiation interval (II) to be 1.

Additionally, one difficulty comes from the large and irregular input sizes. For our dataset, the total input sizes for a chaining task of one read can vary from less than 1 MB to more than 7 MB, which makes on-chip memory inadequate and may cause idling issues. To solve this problem, we design a fine-grained task partitioning scheme, where we tile the input data and interleave them to ensure each PE can be fully utilized all the time.

We not only accelerate the task on an FPGA. but also explore the possibility of speeding it up on a GPU. First, we explore how to properly map different levels of parallelism to different hardware levels. Second, to efficiently leverage

* indicates co-first authors and equal contributions

[1] *read* is an inferred sequence of base pairs of the input DNA fragments

[2] *de novo assembly* is the process of reconstructing the genome from reads without a reference genome

Fig. 1. The diagram of overlap detection. For a given set of reads, first we extract features from each read and build a hash table. Those read that largely share the same features are identified as overlap candidates. Then for a pair of candidate reads, the common features are chained together.

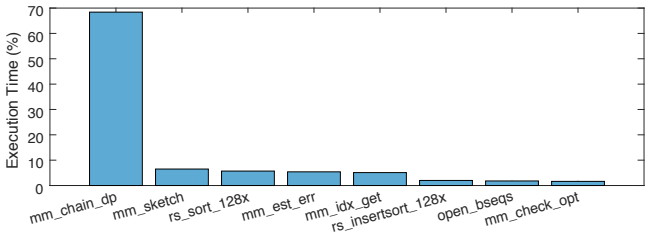

Fig. 2. Time breakdown of Minimap2 working as an overlapper by *gprof*, `mm_chain_dp` is the function that performs chaining

a large degree of data re-use in this algorithm, care must be taken to place data at proper memory levels. Besides, since the GPU architecture also suffers from the reduction operation to find the maximal value and its index, we similarly apply the reordering of computing sequence of the algorithm.

Experiment results show that both devices achieve great speed-ups over the well-optimized CPU baseline. Our FPGA design performs 4× better than GPU. We further dive into the architecture to quantitatively analyze which hardware is more suitable for this algorithm. For this application, we find out that even if we ideally optimize our kernel for NVIDIA Tesla P100 GPU, the theoretical performance will still be capped by the computing resource, and inferior to our FPGA accelerator design.

In summary, we make the following contributions:

- Modify the Minimap2 chaining algorithm to be hardware-friendly, and propose a fine-grained task dispatching method to ensure workload balancing.
- Through hardware-software co-design, we implement an FPGA accelerator using HLS on AWS F1 instance and a GPU kernel on NVIDIA Tesla P100 GPU.
- The FPGA accelerator achieves 27.8× acceleration over the original 14-thread software, 12.4× over our highly optimized software and 3.9× over our GPU design.
- We quantitatively analyze and compare the design and performance difference of FPGA and GPU, which could be a reference when choosing between FPGA and GPU.

II. Background

A. Pairwise Overlap Detection

The long read assembly process uses the Overlap-Layout-Consensus (OLC) computing paradigm [7] [8], which is different from previous short read assembly processes. In OLC assembly, the first step, also the efficiency bottleneck [9], is the detection of overlaps between any pair of reads to form an overlap graph. Fig. 1 shows the overall workflow of pairwise overlapping. Though similar to the read-to-reference alignment problem, read-to-read overlap detection is a distinct problem. It can benefit from specialized algorithms that perform efficiently and robustly on high error rate long reads. The general sequence mapping problem usually follows a *seed-chain-align* form, while for pairwise overlap detection, the last step (base-level alignment) might not be necessary if the chaining step provides required accuracy [2].

B. Overview of Minimap2

Minimap2 is a general-purpose alignment tool that can map different types of sequences against a large reference database. It follows a typical seed-chain-align procedure as most full-genome aligners [2].

In its first stage, seeding, it collects short seeds (called minimizers [10], a special type of k-mer) from the reference and query sequences and indexes them in a hash table. Based on the hash table, reads that share a large number of seeds are selected as overlap candidates.

Then in the second stage, chaining, the tool performs dynamic programming to compare the locations of each shared seed, and group together those who have a consistent distance relative to each other. The Minimap2 chaining algorithm is fast and accurate, even without the following alignment step. The chaining alone is reported to be more accurate than many famous long-read mappers [2]. The chaining step is the focus of this paper.

Finally, if the base-level alignment is requested, in the aligning stage Minimap2 applies dynamic programming to extend the matches of seed, and resolves the gaps between adjacent anchors in the chains with approximate matches. This step is not required for pairwise overlapping.

Combined with different parameters, these three steps can address a series of mapping problems. For example, it works on not only long reads but also short reads, and not only DNA but also RNA sequence. Besides a read overlapper, Minimap2 could replace any whole-genome aligner to determine the difference between two complete sequences, or a reference-based read aligner to assemble fragment reads based on known reference. In all these situations, the chaining step costs a non-trivial amount of time—especially when performing pairwise overlapping in de novo assembly where it does not invoke base-level aligning, and chaining takes about 70% of the execution time, as Fig. 2 shows. In other scenarios, such as reference-based assembly, chaining takes about the same amount of time with aligning (both about 30% of the total time).

C. Minimap2 Chaining Algorithm

We first clarify some common terminologies.

1) Anchor: A short match between two reads, which is represented by a 3-tuple (x, y, w) to encode a match between

interval $[x-w+1, x]$ on one read and interval $[y-w+1, y]$ on the other.

2) Chain: A list of anchors that are correlated in position. Together these anchors can represent an estimated overlap region.

3) Overlap: A global sequence match between two reads, and often occurs when local regions on each read originate from the same location within a larger sequence.

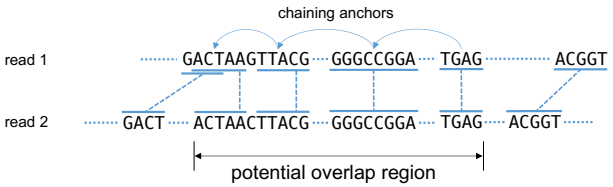

Fig. 3. Finding Estimated Overlap Region

For example in Fig. 3, there are 6 anchors between the two reads. Anchor #2, #3, #4 and #5 form a chain, representing a potential overlap region, while Anchor #1 and #6 might be filtered out due to inconsistency in position.

Minimap2 chains the anchors through dynamic programming. For each anchor, we compute its weight with each of the N previous anchors to determine the best predecessor. In practice, we can almost always find the optimal chain with N = 50; and even in the case where the heuristic fails, the result is close to optimal [2]. In the actual implementation, we choose N to be 64. Equation 1 shows the transition function of the dynamic programming algorithm [2].

$$score(i) = \max\{\max_{i>j\geq 1}\{score(j) + weight(j,i)\}, w_i\} \quad (1)$$

where

$$weight(j,i) = \alpha(j,i) - \beta(j,i)$$

$$\alpha(j,i) = \min\{y_i - y_j, x_i - x_j, w_i\}$$

$$\beta(j,i) = \gamma_c((y_i - y_j) - (x_i - x_j))$$

$$\gamma_c(l) = \begin{cases} 0.01 \times avg\_anchor\_w \times |l| + 0.5\log_2|l|, & l \neq 0 \\ 0, & l = 0 \end{cases}$$

We demonstrate some of the metrics in the transition function in Fig. 4. α value is the number of matching bases between the two anchors. β is the gap cost, reflecting the inconsistency in position between two anchors. For example, in Fig. 3 the β between anchors #5 and #6 will be significant due to the inconsistency in position.

Algorithm 1 shows the details of the chaining algorithm.

Fig. 5 shows an example in a real-life chaining scenario, where each line color represents an anchor. In this example, they form an apparent overlap.

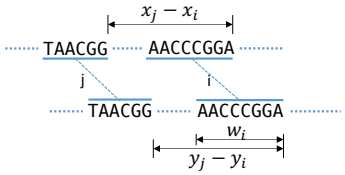

Fig. 4. Position relation between two anchors,

Algorithm 1 *Minimap2* Chaining Algorithm

Input: anchor[]: a list of anchors between a pair of reads
Output: predecessor[]: the predecessors and score[] the chaining scores.

```
1: for i = 1 to n do
2:     for j = i − N to i − 1 do
3:         α = Alpha(anchor[i], anchor[j])
4:         β = Beta(anchor[i], anchor[j])
5:         weight[j] ← score[j] + α − β
6:     end for
7:     p ← arg max_j {weight[j]}
8:     if weight[p] < anchor[i].w then
9:         score[i] = anchor[i].w
10:        predecessor[i] = −1
11:    else
12:        score[i] = weight[p]
13:        predecessor[i] = p
14:    end if
15: end for
```

III. FPGA Accelerator Design

In this section we first discuss different design choices and analyze how the dependency inside the algorithm incurs a very long critical path. Then we introduce a method for transforming the computing order to remove this limitation and achieve full pipelining with II = 1. Based on the adjusted algorithm, we present the streaming microarchitecture.

A. Reordering Computation Sequence

Although there are rich task-level and intra-task-level parallelism inside this dynamic programming algorithm, it cannot be directly mapped to hardware with high performance due to loop-carried dependency.

The upper half of Fig. 6 shows the pseudocode of the original software. For each anchor, the inner loop computes its weight with N previous anchors, which could be unrolled in hardware. Then, we need to obtain the maximum among the N weights to be the final chaining score for the current anchor. However, this value will be immediately used in the next outer-loop iteration. This process will be mapped to a reduction tree inside a feedback loop, as Fig. 7 shows. Therefore, the next outer-loop iteration cannot initiate until the `max` operation finishes. However, more than two cycles are required for the reduction tree to work at 250 MHz, which limits the initiation interval of the pipeline.

Specifically, the pseudocode `score[i]` calculated in iteration `i` (line 6) will be immediately used in iteration `i+1` (line 4). The recurrence of minimal II could be calculated as follows [11]:

$$RecMinII = \max_i \lceil Latency(c_i)/Distance(c_i) \rceil \quad (2)$$

where $Latency(c)$ is the sum of operation latencies along circuit c, and $Distance(c)$ is the sum of dependency dis-

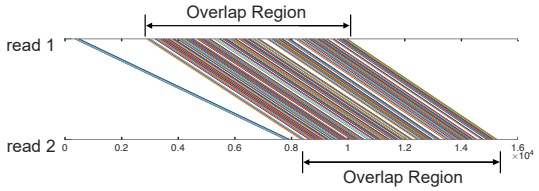

Fig. 5. Anchors between read pair forming overlap region.

```
1 //pseudo code for original software
2 for (i = 0; i < n; i++) { #pragma HLS pipeline
3   for (j = i-N; j < i; j++) { #pragma HLS unroll
4     temp[j] = Weight(anchor[i], anchor[j])+score[j];
5   }
6   score[i] = max{temp[:]}; // II bounded by "max"
7   predecessor[i] = argmax{temp[:]};
8 }
9
10 /*-------------------------------------------------------*/
11
12 //transform computing order
13 set score[:] = {-1}
14 for (i = 0; i < n; i++) { #pragma HLS pipeline
15   curr_score = score[i];
16   for (j = i+1; j < i+N; j++) { #pragma HLS unroll
17     temp = Weight(anchor[j], anchor[i]) + curr_score;
18     if (temp > score[j]) {
19       score[j] = temp; // dynamic update
20       predecessor[j] = i;
21 }}}
```

Fig. 6. Pseudocode of original and transformed algorithm.

tances along circuit c. In this case, assume in the best case $Latency(\texttt{score[i]}) = 2$ (latency for finding the maximum of 64 elements) and $Distance(\texttt{score[i]}) = 1$, since it will be used immediately next cycle. Therefore, the pipeline is bounded by an II=2.

The bottleneck of the `max` reduction operation could be removed by reordering the computation. The core idea is to separate the reduction of N elements into N individual comparing operations in N cycles. The latter half of Fig. 6 shows the idea. Instead of comparing the current anchor with N *previous* anchors and finding a maximum, we compare the current anchor with N *later* anchors and update the temporary value of each of them.

In this way, `score[i]` will be updated N times in outer-loop iteration `i-N` to `i-1`, which is equivalent to the original algorithm, except the reduction of N elements is replaced by N parallel comparing so that the II could easily achieve 1 under the timing constraints.

B. Fully Pipelined PE Design

Fig. 9 demonstrates the design based on the transformed algorithm. It consists of a task scheduler, a result collector and an array of PEs. The scheduler and collector use a double buffer technique to interact with DRAMs while overlapping computing time with data transfer time. Each PE handles the anchors from one read. The anchors are streamed into the PE with II = 1, and the resulting chaining scores are streamed out from the PE at the same rate.

From the modified algorithm, each anchor will be used $N + 1$ times, and the currently in-use anchors form a sliding

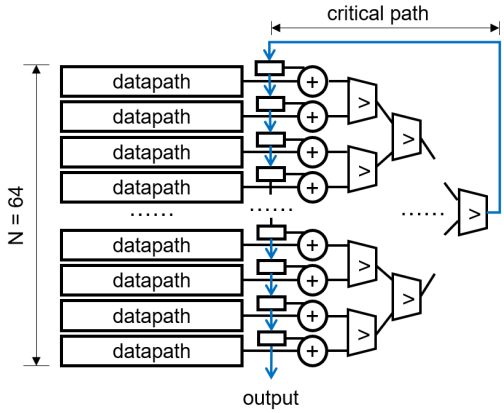

Fig. 7. Critical path of direct implementation of the original algorithm. The generated chaining scores from the reduction tree are looped back to be added up by other weights as line 4 shows.

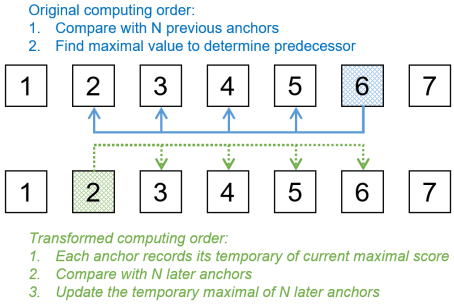

Fig. 8. Illustration of computation order transformation.

window of length $N + 1$. This data movement is mapped to a FIFO. Inside the PE, the input anchors will be streamed into a FIFO structure of depth $N + 1$. The anchor at the head of the FIFO will be compared with N later anchors to calculate the connection scores in parallel, corresponding to unrolling the inner loop in line 16 in Fig. 6. Each cycle the anchors shift forward one stage so that every anchor will be used to compare with N later anchors, corresponding to the outer loops in line 14. The parallel computing logic in the middle computes the weights as in the transition function Equation 1.

Additionally, there might be neighboring anchors that are unrelated, which need to be filtered out. Since different tasks are streamed into one PE without interval, we need to ensure that the first anchors of the later read pair do not engage with the last anchors from the previous pair. To do so, we assign a common tag to the input of a read pair, and two anchors with different tags will not be compared. This corresponds to the *filter* logic in the pipeline.

In the algorithm, each score is initialized to be -1, and each inner-loop iteration updates the value once. After N inner-loop iterations, we obtain one final score. This process (lines 18-20) is mapped to the right part of the PE, where the initial value -1 is streamed through N stages to obtain the final score value. One obvious question is why the value is shifted all around, instead of staying at the same place to be updated N times.

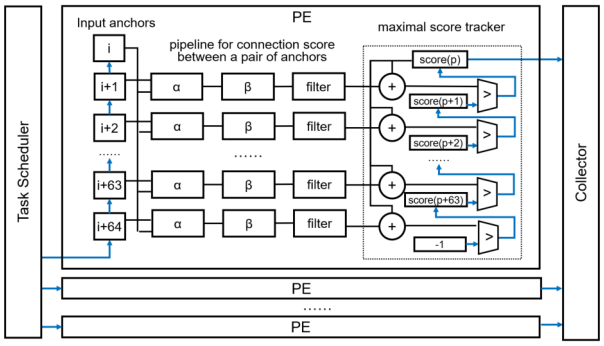

Fig. 9. Overview of a processing element array.

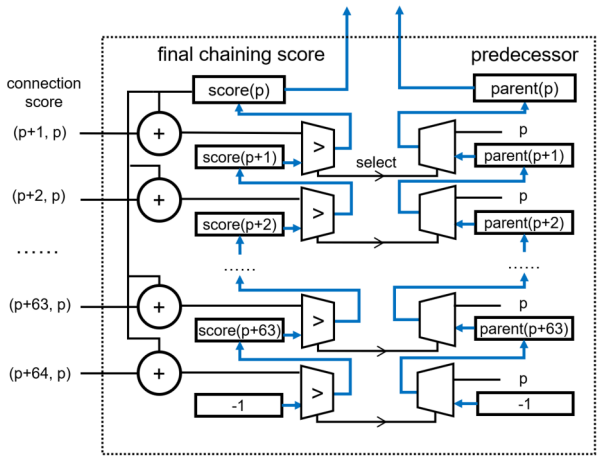

Fig. 10. Maximal score tracker.

The answer is that the inputs on the left hand are being shifted. The input anchors on the left and the output scores on the right are in one-to-one correspondence, so that the temporary score values on the right should shift in the same direction.

We record not only the maximal score but also the selected predecessor as the trace-back process. To do this, we update the tentative predecessor while updating the tentative maximal score. The predecessors are shifted in the same direction.

Therefore, in each cycle one new score can be obtained from a PE. Since the chaining process of each pair of reads is independent of each other, we can implement multiple PEs—each handling one data stream in parallel. A comparison of Fig. 7 and Fig. 10 will reveal the influence of the software modification on the hardware microarchitecture. The previous reduction tree is replaced by a stream of comparators, effectively breaking the previous critical path.

Our solution is based on the modified algorithm, but one may attempt to bypass the dependency limitation by pipelining the inner-loop instead of unrolling, then extracting more task-level parallelism. However, this approach is less efficient, as the memory port must dispatch data to a considerable number of individual pipelines, resulting in severe frequency degradation. Also, redundant pipeline controllers introduce large resource overheads.

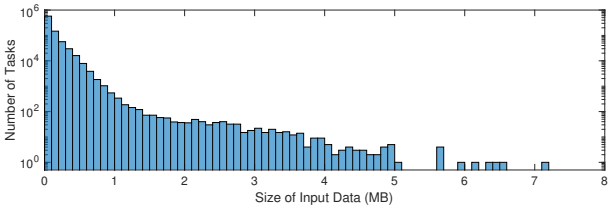

Fig. 11. Histogram of Input Data Size

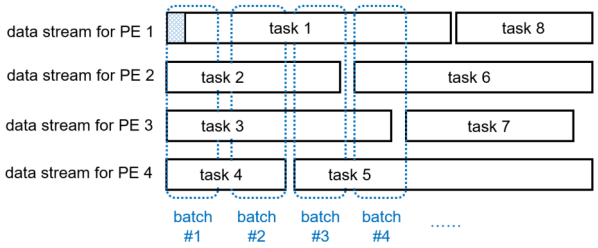

Fig. 12. Fine-grained task dispatching scheme

C. Fine-Grained Load Dispatching

When a pair of reads does not overlap, the algorithm has few anchors as input. If we take the chaining of a pair of reads as a task, tiny tasks with few anchors introduce overhead and hurt resource utilization. Minimap2 addresses this problem by concatenating anchors between one read to all others as a task, and introduces a tag to separate them in runtime. But a challenge remains: the input sizes from different tasks expose a high variance, as Fig. 11 shows. This distribution can easily lead to unbalanced workloads among PEs, which may decrease the PE utilization. To solve this problem, we propose a fine-grained task dispatching scheme to balance the load distribution, as Fig. 12 demonstrates.

In our design, a single task will be only assigned to a single PE; the distributor adaptively dispatches tasks to the idle PEs. The data of a task is partitioned into multiple fine-grained chunks. Rather than only dispatching data from a single task, the distributor dispatches a data batch which contains grouped chunks from different concurrent tasks. In this way, at any clock cycle, every PE is able to fetch input data from the newly transferred batch. This dispatching scheme helps us to achieve the optimal PE utilization.

IV. CPU Baseline and GPU Accelerator

In order to better evaluate the performance of our FPGA accelerator, we optimize the original software in the CPU architecture level, and further design a GPU accelerator for the same task with best efforts.

A. Multi-Core CPU Baseline

We optimize the CPU software of the chaining process in the following three main steps. First, we take advantage of processor affinity and bind a thread to a designated core. In contrast to the default thread scheduling scheme, tasks will not be moved to another idle core, and cache could be reused.

Second, in our multi-socket system, we allocate data of tasks in a neighbor NUMA node of its corresponding core to reduce memory access latency. Finally, we make use of the vector extensions in an Intel CPU to push the processing throughput further. With SIMD instructions, the weights between eight pairs of anchors could be computed concurrently, compared to one in the original code. The same computation order transformation method is used on our CPU code to avoid the expensive reduction operation.

B. GPU Accelerator

To implement the design on the GPU, we also need to explore how to map the different levels of parallelisms to different GPU components and consider where to put the data.

First, the relatively high degree of data reuse determines that we need to map intra-task level parallelism (the inner-loop) to GPU threads. For one task, the currently in-use data forms a sliding window of length $N + 1$, and each anchor, along with the computed score and predecessor number, will be accessed $N + 1$ times continuously. These data should be placed in the local storage for performance concerns, but will consume about 1 KB shared memory per task. For a modern NVIDIA Pascal GPU, a streaming multiprocessor (SM) could keep storage and contexts of 64 warps[3] concurrently to achieve a high compute resource occupancy, but has only 64 KB shared memory in total. Therefore, if we map the task-level parallelism to GPU threads, i.e., each thread handles the chaining of a pair of reads, then one warp (32 threads) will need more than half of the total local storage.

Second, the same computation order transform technique in Section III applies to GPU as well. When different GPU threads cooperate to handle the same task, they also need to perform a synchronized reduction to find the maximal of N numbers, which is similar to FPGA. This reduction operation not only incurs a large instruction overhead, but also suffers from a utilization problem since a large number of threads will be idle in the process. Section V-C gives a quantitative analysis of the influence of computation order transformation on GPU based on the profiling results.

In summary, for our GPU design, each SM handles the chaining of 32 pairs of reads in a interleaving way to hide the instruction latency. Within the task of a pair of reads, 64 threads handle the parallel weights computing. Additionally, in order to hide the time of data transfer between CPU and GPU, we use asynchronous memory operations and order them with CUDA streams.

Table I summarizes and compares the design choices for the three types of computing devices.

TABLE I
PARALLELISM MAPPING CHOICE FOR CPU, FPGA AND GPU

Device	Parallelism Mapping	Degree of Concurrency
FPGA	Task-level → PEs Intra-task → pipelined lanes	1 task per PE 64 inner loop iterations in parallel
GPU	Task-levelk → CUDA blocks Intra-task → CUDA threads	32 tasks per GPU SM 64 inner loop iterations in parallel
CPU	Task-level → CPU threads Intra-task → SIMD instructions	1 task per CPU core 8 inner loop iterations in parallel

V. EXPERIMENT EVALUATION

A. Experiment Setup

The FPGA design is implemented on an Amazon EC2 F1 instance, which includes a Xilinx UltraScale+ VU9P FPGA in 16 nm process. The UltraScale+ FPGA is connected to the host CPU by a PCIe x16 interface. We implement our GPU baseline on the NVIDIA Tesla P100 GPU, which is also fabricated in a 16 nm process and is connected to the host by a PCIe x16 interface. Additionally, we test the baseline multithreading CPU design on 14-core Xeon E5-2680 v4 CPU in a 14 nm process.

We use the public *Caenorhabditis Elegans 40× Sequence Coverage* [12] dataset obtained from PacBio sequencer to evaluate our hardware accelerator. The dataset is about 4.6GB in *FASTA* format.

B. Multi-Core CPU Baseline Evaluation

The Xeon E5-2680v4 CPU has 14 cores operating at a maximum frequency of 3.30 GHz. We record the performance improvements from each of our optimization steps.

The original version with a single thread takes 1420.0s to finish, while our optimized 14-thread software takes 101.9s, i.e., a 13.9× speedup, which is almost linear to the number of cores. As a comparison, the initial parallelized version takes 140.9s with only 10.1× speedup before adding CPU thread affinity and memory optimization.

With SIMD processing and the computation order transformation, the CPU execution time is further reduced to 63.7s. Although we process eight scores of anchors in parallel, we could not achieve 8× speedup. This is because a vector instruction could have a higher overhead than its scalar counterparts, and it requires all eight values to be computed to get the results. Moreover, it could not benefit from quickly filtering out some of the anchors early to avoid complex calculation.

C. GPU Performance Evaluation

The NVIDIA Tesla P100 GPU operates at 1.303 GHz frequency. The total computation time is 20.1s, which is a 7.1× speedup over the original 14-thread software, but still 3.9× longer than its FPGA counterpart. Profiling results show that the GPU resource is already well used by our CUDA kernel: it could achieve 100% occupancy, 78% computing resource utilization, and 67% shared memory usage.

The computation order transformation technique significantly boosts the GPU performance, making it 1.9× faster than direct implementation of the original algorithm. Experiments show that the original reduction operation alone generates 61 instructions, while the effective computation for weights only produces 49 instructions. Moreover, during the reduction, more than 80% of the instructions of all threads are waiting for other

[3]A warp is the basic unit of execution scheduling; it contains 32 threads that execute the same instruction. In the case of the NVIDIA Pascal GPU, 2 of the 64 warps will be issued at a time, in parallel.

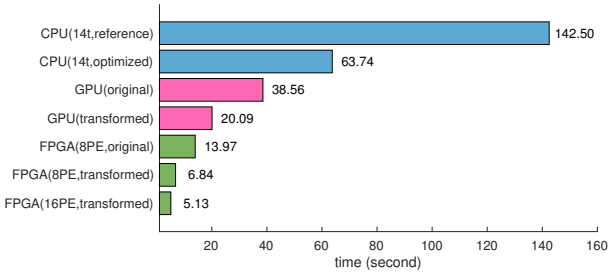

Fig. 13. Kernel execution time

threads to complete. Instead, after the computation order transformation, we do not have a centralized reduction operation anymore, which greatly improves the GPU performance. The equivalent update operation only costs five instructions with 100% thread utilization.

D. FPGA Performance Evaluation

We implement the design Using Vivado HLS. With 8 PEs the design could achieve 250 MHz and overall II of 1, and it could saturate the PCIe bandwidth between CPU and FPGA. The dataset is processed in only 6.84s. Table II shows the resource utilization of the design with 8 PEs. As shown, the current design only takes a small share of FPGA resource. With future generations of PCIe bus, we could further scale up the throughput. For example, with two PE arrays, the design achieves 200 MHz and the same task takes only 5.1s to finish. In comparison, the original algorithm can only achieve an II of 2 when running at 250 MHz, which severely limits its performance.

TABLE II
FPGA RESOURCE CONSUMPTION OF 8 PE VERSION

Name	BRAM_18K	DSP48E	FF	LUT
Total	300	520	304637	385038
Available	4320	6840	2364480	1182240
Utilization	6%	7%	12%	32%

Due to the restriction of the OpenCL model, host data must be first copied into FPGA DRAM, which incurs unnecessary copying time. Instead, the data should be directly streamed to the computing logic for optimal performance. To do so, we could either implement the design in RTL level or integrate our design into the recently developed ST-Accel framework [13] which supports host-kernel communication during kernel execution. This remains as future work.

Fig. 13 shows the performance difference between different computing devices. As a reference, the original software version with a single thread takes 1420.0s to execute, which is not listed in the figure. Our FPGA accelerator is 277× faster than the original single-thread software; 28× faster than the multi-thread software on 14-core CPU, 12× faster than our highly optimized software version and 4× faster than our optimized GPU implementation.

VI. QUANTITATIVE COMPARISON OF FPGA AND GPU

In this section we conduct a quantitative analysis on why the performance of FPGA is better than GPU using this algorithm. In many situations, although claimed to have higher energy efficiency [14][15][16], the acceleration rates by FPGA are not as good as those by GPU, while designing an FPGA accelerator takes a much longer time than GPU. To make things worse, FPGA is usually more expensive than GPU in the same generation. Therefore, it is critical to ascertain on what kind of applications can the performance of FPGA beat GPU. Here, we carry out an in-depth study for this application to better understand the tradeoff between FPGA and GPU.

To evaluate the performance, we compare how many chaining scores could be generated per second. Remember that obtaining one chaining score requires a comparison of one anchor with N previous anchors and finding the maximal weights.

A. Theoretical Performance Analysis

For FPGA, we achieve overall initiation interval of 1 for each PE, and have at most 16 PEs at 200 MHz. This means that in each cycle a PE can generate 1 score at 200 MHz, which equals 3600M scores per second. If we take the PCIe bandwidth limitation between CPU and FPGA into consideration, an array of 8 PEs working at 250 MHz would be able to saturate the bandwidth of PCIe x16, which could generate 2000M scores per second.

The estimation of GPU performance is more complicated than FPGA. Here, we first assume that the GPU computing resource has been perfectly utilized and model a theoretical best performance—which we will find is worse than our FPGA accelerator. Then we will discuss what factors will influence the actual GPU performance from our ideal estimation.

Since the GPU interleaves the instructions from different warps to hide latency, when there are enough warps to schedule, every cycle one instruction could complete in a SIMD processor. Considering the huge embarrassingly parallelism between tasks, we assume there are enough warps to hide the latency of all instructions to be 1 cycle. We then assume all 3584 CUDA cores[4] of the NVIDIA Tesla P100 GPU are fully utilized, so that every cycle we finish 3584 instructions summed up from all threads. The inner loop contents, i.e., computing the weights between two anchors, is compiled into 49 instructions so that obtaining a chaining score takes $49 \times 64 = 3136$ instructions summed up from all threads. The GPU runs at 1303 MHz. Therefore, the estimated best performance possible is $3584/3136 \times 1303 = 1489$M scores per second, far less than FPGA's actual achieved performance.

In reality, many factors could not reach the ideal that we assume here. For example, the instruction latency could not always be hidden with context switching between warps because of resource contention or warp synchronization. Control flow related code and other sequential parts are inevitable in addition to the computational instructions. Besides, in our case

[4]The maximum count of threads that could be executed concurrently.

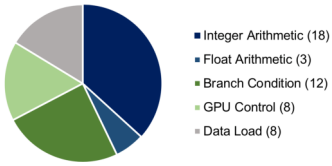

Fig. 14. Breakdown of GPU instructions of weight computation.

one anchor is compared with 64 previous anchors, so that this parallelism can be perfectly mapped to 2 CUDA warps, each consisting of 32 CUDA threads. However, if one anchor only needs to be compared with 48 previous anchors, then half of the threads in the second warp will idle.

B. Why Can FPGA Beat GPU?

One key parameter used in GPU performance analysis is the number of instructions for each CUDA thread, which directly determines the possible performance of GPU. We dive into the instructions, manually map the instructions back to their use to understand why GPU loses the race in this application.

Fig. 14 shows the approximate breakdown of the instructions for computing the weight between two anchors. As can been seen, only 21 instructions are for arithmetic operation, while 20 instructions are for GPU control and branch condition. This is because the algorithm to compute the weight includes much checking and filtering with conditions. For FPGA, these control logics will be integrated into stages in the customized pipeline, making the pipeline longer but having no impact on throughput; while for GPU, those control logics turn into instructions that increase the execution time. This find implies that those very complicated in control flow might be more suitable for FPGA.

Meanwhile, in this application most computations are of 17-bit integer type, which is a plus for FPGA. If large quantities of floating point operations are involved, FPGA might not have enough DSP or have routing problems when using too many DSPs. GPU has such a powerful floating point calculation capability that NVIDIA Tesla P100 GPU could perform 64-bit floating-point arithmetic with a performance of 5.3 TFLOP/s. This implies that those tasks of irregular-width integer type might be more suitable for FPGA.

Additionally, 8 data load related instructions are generated. Since each anchor consists of 4 numbers (x, y, w, tag), GPU first needs to fetch the data from shared memory and spend 4 instructions to unpack it to get the actual numbers. For FPGA, the inputs are kept in BRAM and can be used immediately with combinational logic. This implies that those applications with highly complex data structures might be more suitable for FPGA.

Also, one unique advantage of FPGA for streaming applications is the small latency between the input stream and output stream, which is crucial for real-time applications.

Considering the long design cycle for FPGA, performance estimation is necessary before one carries out the implementation. This situation again reveals the need for a tool to help estimate the performance difference between GPU and FPGA for a given computation pattern. This remains as future work.

VII. Related Work

The fact that the cost of genome sequencing drops faster than Moore's law motivates many FPGA acceleration studies for sequencing algorithms. The general sequence mapping problem can be summarized by the seed-chain-align procedure. A large quantity of work accelerated the base-level *alignment* step – especially the matrix-fill of the well-known Smith-Waterman algorithm and its variation [17][18][19][20][21][22][23][24][25][26]. Other studies [27][28][29][30][31][32] are devoted to the seeding phase or taking care of both phases. To the best of our knowledge, no previous work has accelerated the *chaining* step as we do.

Detection of overlap between read pairs is a particular form of sequence mapping problem that in many cases does not require a base-level alignment step. Turakhia et al. [33] propose and implement in hardware to concatenate all reads into a long string and align each individual read to the long concatenation using a reference-based alignment algorithm. Meng et al. [34] accelerated the pairwise overlapping of optical label-based reads. However, the algorithms and design challenges are all different. On the algorithm level, they deal with an alignment problem of two lists of labels using 2-D dynamic programming; for us, chaining one list of anchors is a 1-D dynamic programming problem. On the hardware design level, the dependency relationship in our algorithm seriously influences the pipeline initiation interval, so we need a special adaptation of the algorithm. The dependency relation in their situation will not, since the newly computed score will not be immediately used. Plus, tiling and fine-grained load dispatching are required in our case.

For other stages in the de novo assembly process, Varma et al. [35] proposed a method to pre-process the read data to reduce the overall assembly time; Ramachandran et al. [36] accelerated the error correction stage for short reads.

VIII. Conclusion

In this paper we accelerate the pairwise overlapping of very long reads. The algorithm has rich parallelism, but also has many components that impede parallel execution. We propose to transform the computation order of the algorithm to make it hardware-friendly, and design a task distributing method to ensure a balanced workload. Then we develop a fully pipelined streaming FPGA accelerator. We further implement a well-designed GPU kernel. Experiment evaluation shows that our FPGA accelerator achieves far better performances than other solutions. Then we perform a quantitative analysis of the performance difference between GPU and FPGA.

IX. Acknowledgement

This research is partially supported by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program and the contributions from the member companies under the Center for Domain-Specific Computing (CDSC) Industrial Partnership Program.

REFERENCES

[1] M. O. Pollard, D. Gurdasani, A. J. Mentzer, T. Porter, and M. S. Sandhu, "Long reads: their purpose and place," *Human molecular genetics*, 2018.

[2] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 1, p. 7, 2018.

[3] ——, "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences," *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, 2016.

[4] J. Chu, H. Mohamadi, R. L. Warren, C. Yang, and I. Birol, "Innovations and challenges in detecting long read overlaps: an evaluation of the state-of-the-art," *Bioinformatics*, vol. 33, no. 8, pp. 1261–1270, 2016.

[5] M. J. Chaisson and G. Tesler, "Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory," *BMC bioinformatics*, vol. 13, no. 1, p. 238, 2012.

[6] I. Sovic, M. Sikic, A. Wilm, S. N. Fenlon, S. Chen, and N. Nagarajan, "Fast and sensitive mapping of error-prone nanopore sequencing reads with GraphMap," *bioRxiv*, p. 020719, 2015.

[7] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature biotechnology*, vol. 33, no. 6, p. 623, 2015.

[8] G. Guidi, M. Ellis, D. Rokhsar, K. Yelick, and A. Buluç, "BELLA: Berkeley efficient long-read to long-read aligner and overlapper," *bioRxiv*, p. 464420, 2018.

[9] G. Myers, "Efficient local alignment discovery amongst noisy long reads," in *International Workshop on Algorithms in Bioinformatics*. Springer, 2014, pp. 52–67.

[10] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[11] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2013, pp. 211–218.

[12] PacificBiosciences, *Caenorhabditis Elegans 40x Coverage Dataset*, 2014. [Online]. Available: http://datasets.pacb.com.s3.amazonaws.com/2014/c_elegans/list.html

[13] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong, "ST-Accel: A high-level programming platform for streaming applications on FPGA," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 9–16.

[14] P. Zhou, H. Park, Z. Fang, J. Cong, and A. DeHon, "Energy efficiency of full pipelining: A case study for matrix multiplication," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 172–175.

[15] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *2010 International Conference on Field-Programmable Technology*. IEEE, 2010, pp. 94–101.

[16] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 47–56.

[17] E. Rucci, A. De Giusti, M. Naiouf, G. Botella, C. García, and M. Prieto-Matias, "Smith-Waterman algorithm on heterogeneous systems: A case study," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*. IEEE, 2014, pp. 323–330.

[18] Y. Yamaguchi, H. K. Tsoi, and W. Luk, "FPGA-based Smith-Waterman algorithm: analysis and novel design," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2011, pp. 181–192.

[19] Y.-T. Chen, J. Cong, J. Lei, and P. Wei, "A novel high-throughput acceleration engine for read alignment," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 199–202.

[20] P. Faes, B. Minnaert, M. Christiaens, E. Bonnet, Y. Saeys, D. Stroobandt, and Y. Van de Peer, "Scalable hardware accelerator for comparing DNA and protein sequences," in *Proceedings of the 1st international conference on Scalable information systems*. ACM, 2006, p. 33.

[21] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun, "A reconfigurable accelerator for Smith-Waterman algorithm," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 12, pp. 1077–1081, 2007.

[22] W. Tang, W. Wang, B. Duan, C. Zhang, G. Tan, P. Zhang, and N. Sun, "Accelerating millions of short reads mapping on a heterogeneous architecture with FPGA accelerator," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 184–187.

[23] Y. Yamaguchi, T. Maruyama, and A. Konagaya, "High speed homology search with FPGAs," in *Biocomputing 2002*. World Scientific, 2001, pp. 271–282.

[24] C. W. Yu, K. Kwong, K.-H. Lee, and P. H. W. Leong, "A Smith-Waterman systolic cell," in *New Algorithms, Architectures and Applications for Reconfigurable Computing*. Springer, 2005, pp. 291–300.

[25] P. Chen, C. Wang, X. Li, and X. Zhou, "Accelerating the next generation long read mapping with the FPGA-based system," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 11, no. 5, pp. 840–852, 2014.

[26] Z. Nawaz, M. Nadeem, H. van Someren, and K. Bertels, "A parallel FPGA design of the Smith-Waterman traceback," in *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE, 2010, pp. 454–459.

[27] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2012, pp. 161–168.

[28] E. Fernandez, W. Najjar, and S. Lonardi, "String matching in hardware using the FM-Index," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 218–225.

[29] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Reconfigurable acceleration of short read mapping," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2013, pp. 210–217.

[30] N. Ahmed, V.-M. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 240–246.

[31] M.-C. F. Chang, Y.-T. Chen, J. Cong, P.-T. Huang, C.-L. Kuo, and C. H. Yu, "The SMEM seeding acceleration for DNA sequence alignment," in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 2016, pp. 32–39.

[32] J. Cong, L. Guo, P.-T. Huang, P. Wei, and T. Yu, "SMEM++: A pipelined and time-multiplexed SMEM seeding accelerator for DNA sequencing," in *28th International Conference on Field-Programmable Logic and Applications (FPL 18)*, 2018.

[33] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 199–213.

[34] P. Meng, M. Jacobsen, M. Kimura, V. Dergachev, T. Anantharaman, M. Requa, and R. Kastner, "Hardware accelerated novel optical de novo assembly for large-scale genomes," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE, 2014, pp. 1–8.

[35] B. S. C. Varma, K. Paul, M. Balakrishnan, and D. Lavenier, "FAssem: FPGA based acceleration of de novo genome assembly," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE, 2013, pp. 173–176.

[36] A. Ramachandran, Y. Heo, W.-m. Hwu, J. Ma, and D. Chen, "FPGA accelerated DNA error correction," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 1371–1376.

[37] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of FPGAs and GPUs," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 93–96.

[38] J. Wang, X. Xie, and J. Cong, "Communication optimization on GPU: A case study of sequence alignment algorithms," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 72–81.