Hardware Accelerated Alignment Algorithm for Optical Labeled Genomes

PINGFAN MENG, MATTHEW JACOBSEN, MOTOKI KIMURA, University of California, San Diego
VLADIMIR DERGACHEV, THOMAS ANANTHARAMAN, MICHAEL REQUA, BioNano Genomics
RYAN KASTNER, University of California, San Diego

De novo assembly is a widely used methodology in bioinformatics. However, the conventional short-read based *de novo* assembly is incapable of reliably reconstructing the large-scale structures of human genomes. Recently, a novel optical label based technology has enabled reliable large-scale *de novo* assembly. Despite its advantage in large-scale genome analysis, this new technology requires a more computationally intensive alignment algorithm than its conventional counterpart. For example, the run-time of reconstructing a human genome is on the order of $10,000$ hours on a sequential CPU. Therefore, in order to practically apply this new technology in genome research, accelerated approaches are desirable. In this article, we present three different accelerated approaches, multi-core CPU, GPU and FPGA. Against the sequential software baseline, our multi-core CPU design achieved a $8.4\times$ speedup while the GPU and FPGA designs achieved $13.6\times$ and $115\times$ speedups respectively. We also discuss the details of the design space exploration of this new assembly algorithm on these three different devices. Finally, we compare these devices in performance, optimization techniques, prices and design efforts.

Categories and Subject Descriptors: B.5.1 [**REGISTER-TRANSFER-LEVEL IMPLEMENTATION**]: Design

General Terms: Design, Performance, Experimentation

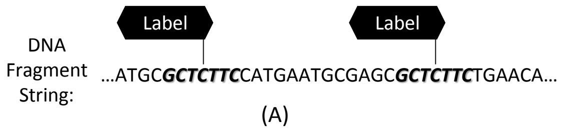
Additional Key Words and Phrases: Genome, Acceleration, *De Novo* Assembly, GPU, FPGA

ACM Reference Format:
Pingfan Meng, Matthew Jacobsen, Motoki Kimura, Vladimir Dergachev, Thomas Anantharaman, Michael Requa, and Ryan Kastner, 2015. Hardware Accelerated Alignment Algorithm for Optical Labeled Genomes *ACM Trans. Reconfig. Technol. Syst.* V, N, Article A (January YYYY), 22 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The ability to construct *de novo* assemblies is widely pursued for medical and research purposes. These *de novo* assemblies are especially invaluable in the studies of structural variations of genomes [Birol et al. 2009]. However, the conventional short-read technology based *de novo* assemblies provide structural information only on a micro-scale ($< 1,000$ bases per fragment). They are not capable of reconstructing the large-scale structures of human genomes [Hastie et al. 2013]. This is due to the fact that using the short-read based assembly leads to ambiguity when these large-scale ($> 100,000$ bases per fragment) genomes have frequent structural repetitions (typical medium to large genomes contain 40 - 85% repetitive sequences [Zuccolo et al. 2007]).

In recent years, research has shown that a novel optical label based technology is able to overcome this limitation of the short-read technology [Lam et al. 2012]. This novel technology fluorescently labels the DNA molecule strings at the locations where a specific nucleobase combination appears (e.g. label wherever the combination GCTCTTC appears, as demonstrated in Fig. 1(A)). Then the labeled DNA molecules are linearized by being passed through a nanochannel device. These linearized strings with labels are imaged by a CCD camera as demonstrated in Fig. 1(B). In the image field, on each string, the physical distances between every two labels are measured and collected. This process results in a uniquely identifiable sequence-specific pattern of labels to be used for *de novo* assembly. As opposed to the four letters, these



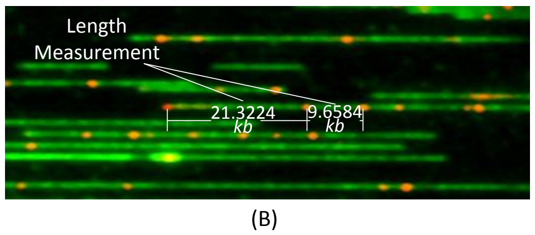


Fig. 1: Demonstration of the optical labeling process. (A) Fluorescent labels attached to "GCTCTTC". (B) The real image field of labeled DNA fragments from a microscopy CCD camera. The strings are linearized DNA fragments. The glowing dots are fluorescent labels. The numbers in kilo-bases(*kb*) are examples of physical distance measurement between labels.

arbitrary physical distances are unlikely to contain structural repetitions. Therefore, this optical method enables the reconstruction of the large-scale genomes for modern bioinformatic studies. In genomic studies, $N50$ is a widely used metric to measure the ability of a technology to assemble large-scale structures. Research results show that this novel optical assembly enhances the $N50$ by two orders of magnitude compared to the short-read assembly [Hastie et al. 2013].

The task of the *de novo* assembly is reconstructing the genome from a set of DNA fragments. The most computationally intensive part of this task is the algorithm that aligns every pair from the DNA fragment set. This pair-wise alignment algorithm for the optical assembly is fundamentally different from the short-read alignment. In the conventional short-read based process, as depicted in Fig. 2 (A), the alignment algorithm is applied on the strings with "A","C","G" or "T" DNA nucleobase letters. As opposed to the short-read letters, the new optical method aligns the locations of the fluorescent labels on the strings shown in Fig. 2 (B). Aligning these arbitrary numbers obtained from a human genome takes nearly $10,000$ hours on a sequential CPU. Moreover, research [Baday et al. 2012] has shown that the resolution of the optical label method can be further enhanced by adding multiple types (colors) of labels.

Therefore, accelerating this alignment algorithm is desired not only for the purpose of shortening the process time but also for enabling this optical based technology in genome studies that require high resolutions.

In this article, we present three accelerated approaches for the optical labeled DNA fragment alignment using multi-thread CPU, GPU and FPGA. These designs are compared against a single thread sequential CPU implementation. The major contributions are:

— The first attempt to accelerate the large-scale genome assembly on hardware.
— An end-to-end FPGA accelerated implementation.
— A GPU accelerated implementation.
— A comparison and design space exploration of the multi-core CPU, GPU and FPGA.

This article is the extension of our previous publication [Meng et al. 2014]. We particularly extend the methodology of the FPGA design space exploration and the hardware

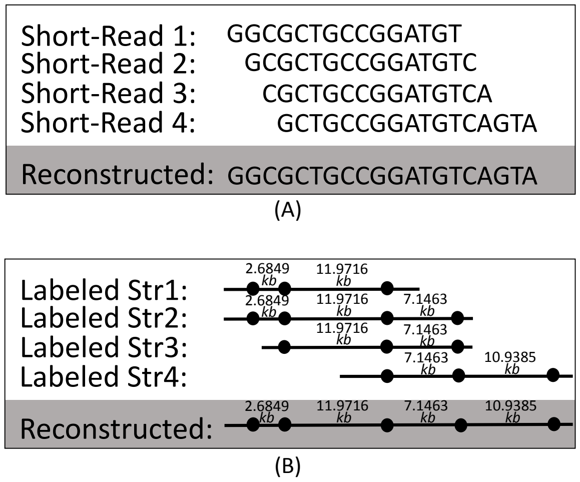

Short-Read 1: GGCGCTGCCGGATGT
Short-Read 2: GCGCTGCCGGATGTC
Short-Read 3: CGCTGCCGGATGTCA
Short-Read 4: GCTGCCGGATGTCAGTA
Reconstructed: GGCGCTGCCGGATGTCAGTA
(A)
Labeled Str1:
Labeled Str2:
Labeled Str3:
Labeled Str4:
Reconstructed:
2.6849 kb
11.9716 kb
7.1463 kb
10.9385 kb
(B)

Fig. 2: Comparison of the conventional and the novel *de novo* assembly methods. (A) Alignment process in the conventional short-read based method. (B) Alignment process in the novel optical label based method. Each dot represents a fluorescent label.

comparison. We also investigated the relationship between the accelerator performance and the number of alignments per host-device data transfer.

The rest of the paper is organized as follows. We discuss related work in Section 2. We describe the alignment algorithm in Section 3. This is followed by descriptions of the accelerated designs in Section 4. Experimental performance results are provided in Section 5. We compare the hardware accelerators in Section 6. We conclude in Section 7.

2. RELATED WORK

Multiple accelerated approaches for short-read assembly have been proposed in recent years. Olson *et al*. have proposed a multi-FPGA accelerated genome assembly for short-reads in [Olson et al. 2012]. They accelerated the alignment algorithm on the FPGAs for the reference guided genome assembly with $250\times$ and $31\times$ speedups reported against the software implementations BFAST and Bowtie respectively. Varma *et al*. have presented a FPGA accelerated *de novo* assembly for short-reads in [Varma et al. 2013]. They chose to accelerate a pre-processing algorithm on the FPGA to reduce the short-read data for the CPU assembly algorithm. They reported a $13\times$ speedup over the software. They also proposed an improved FPGA implementation exploiting the hard embedded blocks such as BRAMs and DSPs in [Varma et al. 2014]. Attempts have also been made to accelerate genome assembly on GPUs. Aji *et al*. have proposed a GPU accelerated approach for short-read assembly in [Aji et al. 2010]. They reported a $9.6\times$ speedup. Liu *et al*. proposed a GPU accelerated DNA assembly tool - SOAP3 [Liu et al. 2012] which achieves $20\times$ speedup over Bowtie.

Although these approaches have improved the performance of the short-read assembly significantly, they are limited to micro-scale genomes. There is still no high performance solution for large-scale genome structure analysis. Our implementations provide an accelerated solution for this large-scale genome task.

Our implementations are fundamentally different from these previous efforts because they employ the novel optical label based genome assembly. In this article, our accelerated designs differ from the previous short-read approaches in two ways: 1) the

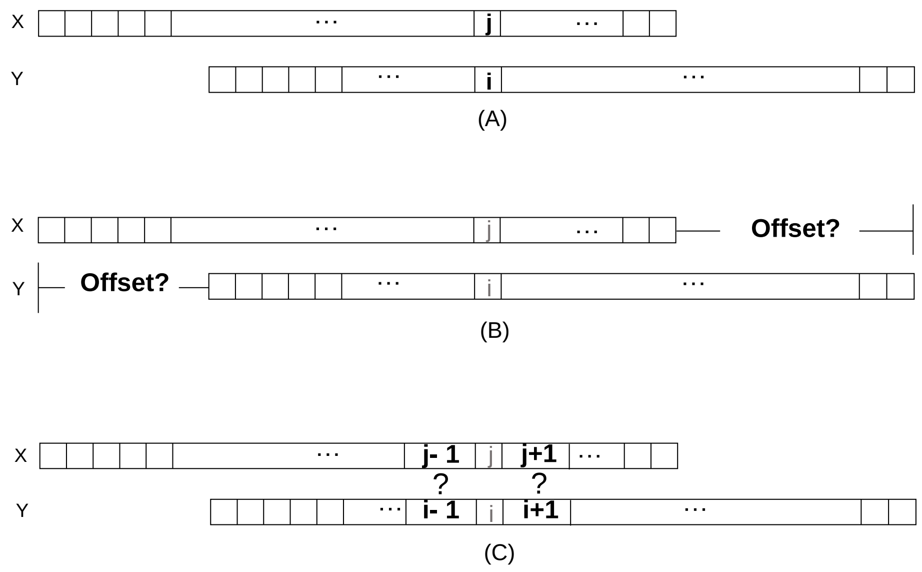


Fig. 3: Array alignment: assume X_j aligned to Y_i, (A) evaluate the similarity between X_j and Y_i; (B) evaluate the boundary offset penalty when X_j aligned to Y_i; (C) evaluate the similarities between X_j's neighbors and Y_i's neighbors.

data in the optical method requires more precision bits than conventional four letters (A,C,G,T) do; 2) the physical label locations require a different alignment algorithm [Valouev 2006] from the traditional Smith-Waterman. Most short-read methods employed the traditional Smith-Waterman algorithm which computes each score matrix element from its three immediately adjacent elements. The algorithm in our optical label based method computes each element from a 4×4 area as demonstrated in Fig. 5. These differences not only increase the computational intensity but also require a different hardware parallel strategy from the ones proposed in these previous short-read based works. To the best of our knowledge, our implementations are the first attempt to accelerate the large-scale genome assembly using GPUs and FPGAs.

3. ALGORITHM

Our goal is to align every pair of floating point number arrays which represent the physical distances of the optical labels on the DNA fragments. As shown in Fig. 3, for arrays X and Y, we decide whether the alignment (X_j aligned to Y_i) is valid based on three evaluations. Firstly, as shown in Fig. 3 (A), we need to evaluate the similarity between X_j and Y_i, which is intuitive. Secondly, as depicted in Fig. 3 (B), we need to evaluate the boundary offset penalty. When X_j is aligned to Y_i, the leftmost ends of X and Y may create an offset. Large offsets produce unwanted gaps in the DNA assembly. A valid alignment should have minimum offset. The third evaluation is to calculate the similarities between X_j's neighbors and Y_i's neighbors as demonstrated in 3 (C). The necessity of this evaluation is also intuitive. Even if X_j is very similar to Y_i, the alignment is not valid if the other elements of the two arrays are dissimilar.

These intuitive evaluations of the alignment are realized by a dynamic programming method specifically modified for the optical DNA analysis by Valouev [Valouev 2006]. The overall flow diagram of the algorithm is demonstrated in Fig. 4. The algorithm aligns two arrays X and Y of optical label positions by computing a likelihood

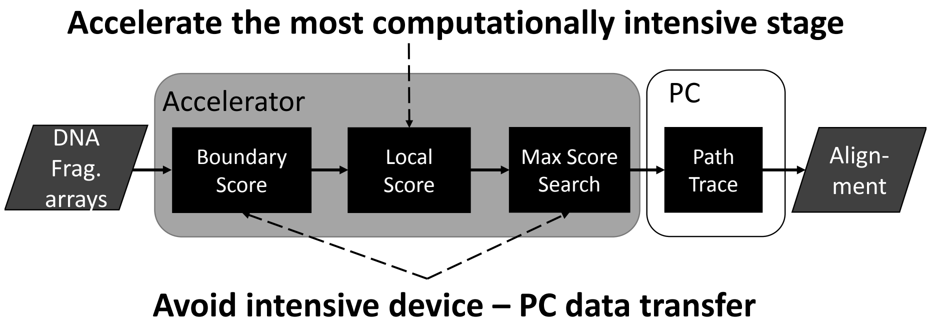

Accelerate the most computationally intensive stage
Accelerator
PC
DNA Frag. arrays
Boundary Score
Local Score
Max Score Search
Path Trace
Align-ment
Avoid intensive device – PC data transfer

Fig. 4: Overall algorithm and its hardware partitioning. We accelerate the local score stage due to its computational intensity. In our partitioning, we also assign the boundary score and max score search stages to the accelerator to avoid intensive device-PC data communication.

score matrix and finding the maximum within this matrix. Each score represents the likelihood of a possible alignment between X and Y. Assuming the sizes of the input arrays are M and N, the algorithm computes a $M \times N$ score matrix as depicted in Fig. 5. The computation of each element in the matrix requires local scores. The black square in the figure shows an example of a local score. Those elements near the edges, shown as the grey regions in the figure, also require boundary scores. Thus, the alignment algorithm consists of three steps: 1) compute the boundary scores as described in Algorithm 1; 2) compute the local scores as described in Algorithm 2; 3) find the best score and its correspondent (i, j) in the score matrix as shown in lines 10 - 12 of Algorithm 2. If the best score passes the threshold, then we find an alignment between X and Y with X_j aligned to Y_i using a trace-back operation. In our hardware accelerated approaches, we keep the trace-back operation on the host PC. We therefore only describe the best score computation in detail as follows.

The computation of the boundary scores is described in Algorithm 1. In the algorithm, to compute a boundary score element located at (i, j), we firstly compute its leftmost offset $Lx_{i,j}$ or $Ly_{i,j}$ as shown in lines 12 - 20. Then we compute an "end" likelihood and several mixed likelihoods as shown in lines 21 - 30. We choose the maximum among these likelihoods to be the boundary score for this position. This process is iterated, as shown in lines 4 and 11, to produce the boundary scores for the top 4 rows and the leftmost 4 columns of the score matrix. An identical boundary score algorithm is also applied on the rightmost offsets of the input arrays to fill the bottom 4 rows and the rightmost 4 columns of the score matrix. These boundary score locations are visualized in Fig. 5.

We compute a local score to represent the similarity between X_j and Y_i as well as the similarities between X_j's neighbors and Y_i's neighbors. In Algorithm 2, to compute each local score $score_{i,j}$, we generate 16 score candidates correspondent to its upper-left 4×4 neighbors (refer to Algorithm 2 lines 5 - 7). Each of the 16 candidates is computed by adding a local likelihood (this represents the similarity between X_j and Y_i) to its correspondent previous score (this represents the similarity between X_j's neighbors and Y_i's neighbors) from the 4×4 area (the shaded area in Fig.5). The score in $score_{i,j}$ is updated with the maximum among all these 4×4 candidates. This process is iterated $M \times N$ times to generate the complete score matrix as shown in lines 3 and 4. Then we find the highest score within the matrix (lines 10 - 12), which represents the best

Algorithm 1 The Boundary Score Algorithm

```
Input: Two arrays of optical label locations $X, Y$; Sizes of the input arrays $1 : M$, $1 : N$
 1: $Likelihood_{local}(x, y, m, n)$ local likelihood function
 2: $Likelihood_{end}(x, m, n)$ end likelihood function
 3: $Likelihood_{mix}(x, y, m, n)$= $Likelihood_{end}((x + y)/2, m, n) + Likelihood_{local}(x, y, 1, 1) - Likelihood_{local}((x + y)/2, (x + y)/2, 1, 1)$ mixed local and end likelihood function
 4: for $i$ = 1 to $N$ do
 5:     $Lx_{i,j} = 0$, $Ly_{i,j} = 0$
 6:     if $i \leq 4$ then
 7:         $j_{max} = M$
 8:     else
 9:         $j_{max} = 4$
10:     end if
11:     for $j$ = 1 to $j_{max}$ do
12:         if $X_j < Y_i$ then
13:             while $Y_i - Y_{Ly_{i,j}} > X_j$ do
14:                 $Ly_{i,j} + +$
15:             end while
16:         else
17:             while $X_j - X_{Lx_{i,j}} > Y_i$ do
18:                 $Lx_{i,j} + +$
19:             end while
20:         end if
21:         $score_{i,j} = Likelihood_{end}(min(X_j, Y_i), j+1-max(1, Lx_{i,j}), i+1-max(1, Ly_{i,j}))$
22:         if $X_j < Y_i$ then
23:             for $k$ = $Ly_{i,j}$ to $i - 1$ do
24:                 $score_{i,j} = max(score_{i,j}, Likelihood_{mix}(X_j, Y_i - Y_k, j, i - k))$
25:             end for
26:         else
27:             for $k$ = $Lx_{i,j}$ to $j - 1$ do
28:                 $score_{i,j} = max(score_{i,j}, Likelihood_{mix}(X_j - X_k, Y_i, j - k, i))$
29:             end for
30:         end if
31:     end for
32: end for
Output: Score matrix $score[1 : N][1 : M]$ filled with boundary scores in the top 4 rows and leftmost 4 columns
```

alignment for X and Y. This highest score is used in the post processes to complete the genome reconstruction.

The likelihood functions in Algorithm 1 and 2 are derived from an error model proposed in [Valouev 2006]. The functions $Likelihood_{local}(x, y, m, n)$ and $Likelihood_{end}(x, m, n)$ are computed as shown in Equations 3 and 4 respectively. The $Likelihood_{local}(x, y, m, n)$ function consists of two terms: the bias value B_{XY} (provided in Equation 1); the maximum between the penalty value (provided in Equation 2) and a constant $P_{OutlierPenalty}$. The values of the constants used in Equations 1 - 4 are empirically tuned to suit the optical experiment [Valouev 2006]. Changing these values does not influence the computing speed of the algorithm. Therefore, without the loss of generality, in our implementations, we tuned these constants to suit our experiment input data - a synthetic human genome. These constant values are listed in Table I.

Algorithm 2 The Dynamic Programming Score Algorithm

```
Input: Two arrays of optical label locations $X$, $Y$; Sizes of the input arrays $1 : M$,
    $1 : N$; Score matrix $score[1 : M][1 : N]$ with boundary scores filled
1:  $Likelihood_{local}(x, y, m, n)$ local likelihood score function
2:  $score_{best} = -\infty$
3:  for $i$ = 1 to $N$ do
4:      for $j$ = 1 to $M$ do
5:          for $g = max(1, i-4)$ to $i-1$ do
6:              for $h = max(1, j-4)$ to $j-1$ do
7:                  $score_{i,j} = max(score_{i,j}, A_{g,h} + Likelihood_{local}(x_j - x_h, y_i - y_g, j-h, i-g))$
8:              end for
9:          end for
10:         if $score_{i,j} > score_{best}$ then
11:             $score_{best} = score_{i,j}$, $j_{best} = j$, $i_{best} = i$
12:         end if
13:     end for
14: end for
Output: Best score $score_{best}$; The $X$ and $Y$ indices of the best score $j_{best}$ and $i_{best}$
```

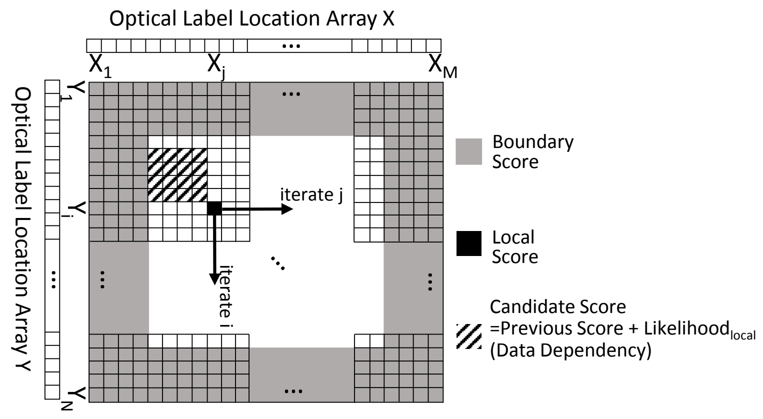

Optical Label Location Array X
X_1
X_j
X_M
Optical Label Location Array Y
Y_1
Y_i
Y_N
iterate j
iterate i
Boundary Score
Local Score
Candidate Score =Previous Score + Likelihood$_{local}$ (Data Dependency)

Fig. 5: Visualized pair-wise alignment process. The 2D array represents the likelihood score matrix. Each (i, j) element in the matrix is a likelihood score for aligning X_j with Y_i. The top, bottom, left and right grey regions represent the boundary score computations. The black square and the shaded area displays one iteration of the dynamic programming process. The computations for the black square have data dependencies to the shaded area. The arrows show that this computation is iterated to fill the entire matrix.

Let F represent the number of DNA fragments of an assembly process. Let M be the number of labels on the fragment. The algorithm requires $O(F^2N^2)$ times of $Likelihood_{local}$ operations to complete an assembly process. The DNA fragment pool typically has $100,000$ - $1,000,000$ arrays. A typical input array length (M or N) is 15 - 100 elements. Therefore, the number of $Likelihood_{local}$ operations, in a human genome assembly process, is on the order of 10^{15}. The total amount of computations requires more than 10,000 hours on a sequential CPU.

Each element of the input arrays represents a distance, which is on the order of thousands of bases, between two neighboring optical labels on the actual DNA frag-

Table I: Constant values in score functions

Constant	C	V	δ	B	$B^{'}$	P_{miss}	R	P	B_{end}	$B^{'}_{end}$
Value	3.6505	0.0449	0.0010	−0.0705	0.9144	1.5083	−0.0931	−8.1114	0.0226	0.3992

ment. The synthetic data used in our implementations is designed to simulate these properties of the real-world human genomes. Since the data ranges and array lengths are similar, the computation performance tested with this synthetic data reflects the performance with the real-world genomes.

Our focus is to accelerate this pair-wise algorithm which aligns the optical labeled DNA molecule fragments to construct the contigs in the assembly process. The scaffolding process, using optical labeled contigs, is not a computationally intensive operation which can usually be performed on a sequential computer in 10-30 minutes.

$$bias_{XY} = [max(0, x - \delta) + max(0, x - \delta)] * B + B^{'} \tag{1}$$

$$pen = C - \frac{(x-y)^2}{V * (x+y)} - P_{miss} * (m+n) - [max(0, x - \delta) + max(0, x - \delta)] * R \tag{2}$$

$$Likelihood_{local}(x, y, m, n) = bias_{XY} + max(pen, P) \tag{3}$$

$$Likelihood_{end}(x, m, n) = 2 * max(0, x - \delta) * B_{end} + B^{'}_{end} - P_{miss} * (m + n - 2) \tag{4}$$

4. ACCELERATED DESIGNS

We partitioned the algorithm by accelerating some parts on the hardware and keeping some parts on the PC. This partitioning strategy is depicted in Fig. 4. The most computationally intensive stage of the algorithm is the local score computation. Therefore, in our partitioning, we accelerated the local score computation on the hardware. The two stages, boundary score computation and maximum score search, are not as computationally intensive as the local score stage. However, these two stages have significantly intensive data communication with the local score stage. In order to avoid this communication bottleneck between the PC and the hardware accelerator, we also assigned these two stages on the accelerator. The path trace stage consists of control intensive operations. Therefore, we kept this stage on the PC.

We identified three levels of possible parallelism in the algorithm (from coarse-grained to fine-grained): 1) align multiple pairs in parallel; 2) compute multiple elements (rows or columns) in the score matrix in parallel; 3) compute the 16 likelihood scores for each score element in parallel. These three levels and their hierarchy are depicted in Fig. 6. Particular computation and data reuse patterns exist in each level of possible parallelism. These patterns create tradeoffs in hardware accelerated designs.

When using level 1 parallelism (processing multiple pairs in parallel), each pair is data independent. Therefore, data communications or synchronization between parallel processes do not exist. However, it requires more computing resource to manage multiple alignments concurrently as well as more storage resource for intermediate data (e.g. multiple score matrices). On the other hand, the other two levels of parallelism (levels 2 and 3) provide more opportunities to share or reuse the data between the parallel processes due to their finer granularity. However, these two fine-grained levels of parallelism may introduce performance challenges such as higher

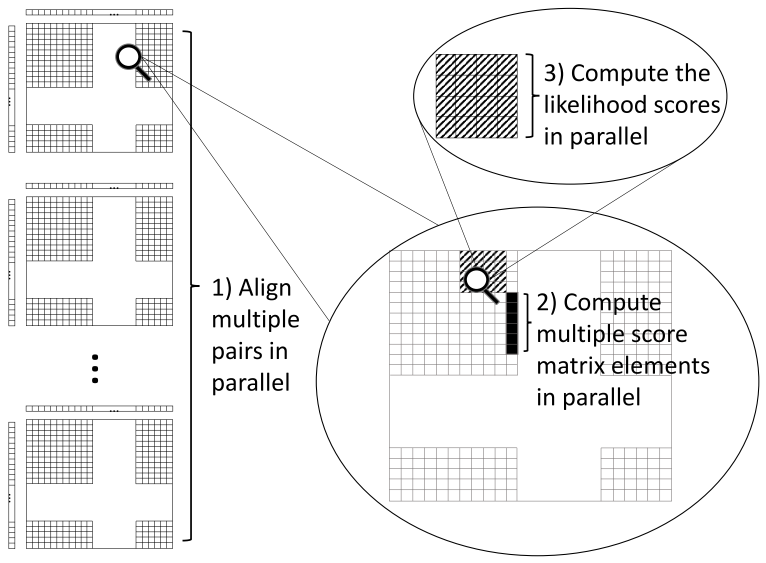

3) Compute the likelihood scores in parallel
1) Align multiple pairs in parallel
2) Compute multiple score matrix elements in parallel

Fig. 6: Possible parallelism in the algorithm.

synchronization overhead on processors and placement and route complexity on FPGAs. These complex architectural tradeoffs create design space exploration problems. We explored these design spaces to determine the proper level or combination of levels of parallelism to match the architectural features on the hardware. We also applied multiple optimization techniques on each design. We applied SIMD instructions and multi-thread techniques on the CPU design. For the GPU design, we tuned the CUDA code to tackle the data dependency caused by the local score computation. We also implemented a low level FPGA design due to the inefficient resource utilization provided by the state of the art high level synthesis tools. In the following sections, we describe the design space explorations and the optimal designs in detail.

4.1. Multi-core CPU

In the CPU design, we firstly improved the locality of the program by dictating the compiler to store the highly reused variables in the CPU registers. We then parallelized the algorithm by inserting OpenMP directives. The performance is highly correlated with the granularity of the iterations in the algorithm. We evaluated the fine-grained strategy which processes multiple rows and columns in parallel on the multiple CPU cores. The evaluation results indicated that it is expensive to synchronize and exchange fine-grained data among the cores. The multi-core CPU is more suitable for the coarse-grained parallelism. Therefore, we chose to align multiple pairs in parallel on the multi-core CPU.

We divided the total workload into several sets of alignment tasks and assigned each of the sets to a CPU core as demonstrated in Fig. 7. When one CPU core finishes its current alignment workload, it can start aligning another pair immediately without synchronizing with the other CPU cores. This setup does not create “dead” parallel processes or threads when the input array sizes change during the run-time. Therefore, all the CPU cores are completely occupied during this process. In addition, within each core, the process is in a sequential fashion which is suitable for control dominated operations such as the boundary score computation. We also forced functions

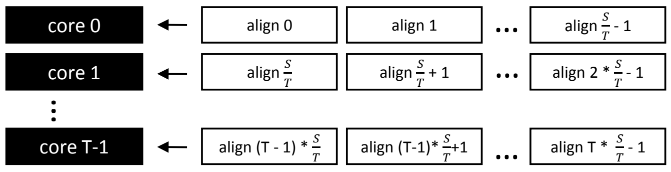

core 0
core 1
core T-1
align 0
align 1
align $\frac{S}{T}$ - 1
align $\frac{S}{T}$
align $\frac{S}{T}$ + 1
align 2 * $\frac{S}{T}$ - 1
align (T - 1) * $\frac{S}{T}$
align (T-1)* $\frac{S}{T}$+1
align T * $\frac{S}{T}$ - 1

Fig. 7: Multi-core CPU accelerated design. Assume there are S pairs of optical arrays to be aligned and the CPU has T cores.

$Likelihood_{local}(x, y, m, n)$ and $Likelihood_{end}(x, m, n)$ to be static and inlined in order to provide more optimization opportunities for the compiler.

The computations of the $Likelihood_{local}$ function provide us an opportunity to utilize the CPU SSE SIMD instructions. Therefore, we program the $Likelihood_{local}$ function to process 4 elements with a SIMD fashion using "__m128" type of its intrinsic operands.

4.2. GPU

The GPU design consists of three CUDA kernels, invoked from a C++ host code. The CUDA kernels accelerate the alignment algorithm to keep the intermediate data on the GPU during the process. The C++ host program only sends the input DNA arrays to the first kernel and receives the output maximum score from the third kernel.

There are multiple options for CUDA kernel design based on different levels of granularity. We firstly evaluated the coarse-grained only strategy on the GPU. The evaluation shows that coarse-grained parallelism is significantly bounded by a low GPU occupancy. Therefore, to fully utilize the GPU parallel computing power, we added fine-grained parallelism in our design. The GPU design computes multiple rows and columns in fine-grained parallel within each GPU thread-block. The design also utilizes multiple thread-blocks to align multiple pairs in coarse-grained parallel. Computing the 16 candidates in parallel is not efficient on the GPU since it requires a 16-element reduction process which creates idle threads frequently.

We partitioned the algorithm into three CUDA kernels 1) boundary score kernel; 2) dynamic programming kernel; 3) maximum score search kernel. We chose this kernel partitioning because these parallelized computations require GPU global synchronization after 1) and 2).

In the boundary score kernel design, we fully parallelized the computations due to the data independency. The GPU thread arrangement is: assigning the boundary score computation for each element (lines 12 - 30 in Algorithm 1) to one GPU thread; assigning the boundary score computations of each alignment to one GPU thread-block. With this design, we maximized the GPU parallel resource occupancy. Moreover, since this design assigns all the computations of an alignment to the same thread-block, we were able to store the intermediate data in the shared memory to minimize the memory access delay in the computations.

The pseudo code of the dynamic programming kernel is described in Listing 1. We parallelized the score element computations using $N \times 4$ threads in each thread-block. The candidate score computation for each matrix column requires 4 previous columns as described in line 7 of Algorithm 2. Parallelizing this part of the algorithm is a challenging task due to this data dependency. We overcame this issue by dynamically assigning the columns of the score matrix to 4 groups of threads. As described in Listing 1, we used $threadIdx.y$ to partition $N \times 4$ threads into 4 groups. They form a software pipeline. Each thread group is only responsible for a specific candidate computation (leftmost, 2nd left, 2nd right or rightmost). By increasing col_id, we stream the columns of the score matrix into this pipeline.

The example in Fig. 8 shows a snapshot of this software pipeline when the GPU is processing columns 8, 9, 10 and 11. These columns are assigned to the different stages (thread groups) of the pipeline: column 11 to group $threadIdx.y = 0$; column 10 to $threadIdx.y = 1$; column 9 to $threadIdx.y = 2$; column 8 to $threadIdx.y = 3$. The computations in the pipeline stages $threadIdx.y = 0 - 3$ are leftmost candidates, 2nd left candidates, 2nd right candidates and rightmost candidates respectively, as shown in the shaded blocks in Fig. 8. In the snapshot, these computations all require the data from column 7 which has already been computed in the previous col_id iteration (refer to the "for" loop in Listing 1). Once the computations in the snapshot are finished, the data in column 8 is then ready. With the data from column 8, the pipeline streams a new column (column 12 in the snapshot) by increasing the iteration index col_id. These 4 thread groups execute different instructions to implement the 4 stages of the pipeline. In order to fit this design on the GPU SIMD architecture, we ensured the threads of each GPU warp to execute the same instruction by extending N to a multiple of 32.

Once the dynamic programming kernel finishes computing the score matrix, the third kernel searches the matrix to find $score_{best}$, i_{best} and j_{best}. We implemented this maximum score search kernel using the reduction approach. We kept the reduction process of each alignment within one thread-block. Therefore, this process does not require the expensive global synchronization on the GPU. Then, we created multiple thread-blocks to concurrently process the reductions for multiple alignments. We also applied shared memory and efficient warp arrangement in the reduction.

4.3. FPGA

Similar to the GPU, the FPGA also accelerates the algorithm by processing the computations in a parallel fashion. The FPGA is a customizable architecture. There are usually two ways to implement the parallelism on the FPGA: 1) replicate a logic module multiple times to physically create multiple parallel data paths; 2) pipeline the architecture to process the multiple data concurrently in a streaming fashion. A high performance design requires a proper decision on which technique is used to implement each of the three levels of the algorithmic parallelism. Moreover, due to the FPGA resource constraints, a feasible design also requires the proper number of replications in each level of parallelism. There exists many possible settings of choices of parallel techniques and numbers of replications. In order to reduce the size of the design space, we firstly constructed a reasonable structure of the FPGA design based on heuristics. Fig. 9 depicts this FPGA structure. Due to the algorithmic data dependency, it is impossible to replicate parallel data paths for both row and column dimensions. Therefore, we chose to only replicate the row parallel data paths (level 2 parallelism)and pipeline the column dimension (level 2 parallelism). We replicated the likelihood score units (level 3 parallelism) to sustain the throughput of this full pipeline in the column dimension. We also replicated the entire alignment module multiple times (level 1 parallelism) to maximize the overall throughput. We then permutated the numbers of parallel paths in this structure to find the optimal setting.

Implementing multiple RTL designs to measure the performances of these permutations requires a significant amount of effort. Therefore, exploring the FPGA design space using RTL designs is inefficient in terms of the development complexity. Instead of manually implementing multiple RTL designs, we propose a method using Vivado HLS which enables rapid FPGA implementations to explore different parallel structures.

We restructured the original software C code, as described in Listing 2, into the format that represents the parallel hardware structure. We firstly constructed a function *lkh_score()* to implement the likelihood score computation in equation 3. To im-

Listing 1: Pseudo Code for Dynamic Programming GPU Kernel

```
//gridDim.x=number of alignments
//blockDim.x=N, blockDim.y=4
__global__ void par_4_col_kernel(/*input/output arguments */)
{
    int align_offset=M*N*blockIdx.x;
    //shared mem delecration
    //move input X, Y arrays from global memory to shared memory
    for (int col_id=0; col_id<M; col_id++)
    {
        if (threadIdx.y==0)
        {
            /*use feedback_score to compute the leftmost candidates and find the
                max for col_id+3*/
        }
        else if (threadIdx.y==1)
        {
            /*use feedback_score to compute the 2nd left candidates and find the
                max for col_id+2*/
        }
        else if (threadIdx.y==2)
        {
            /*use feedback_score to compute the 2nd right candidates and find the
                max for col_id+1*/
        }
        else if (threadIdx.y==3)
        {
            /*use feedback_score to compute the rightmost candidates and find the
                max for col_id*/
            //output the score for col_id
            //feedback_score[threadIdx.x]=score for col_id
        }
    __syncthreads();
    }
}
```

plement the full pipeline, we restructured the local score computation code into a function *pipeline_unit()* with 4 pipeline stages. Lines 35 - 39 describe a line buffer used to feed the input into this pipeline. We then call *pipeline_unit()* in the *alignment_module()* function. We replicated multiple instances of *lkh_score()*, *pipeline_unit()* and *alignment_module()* to generate parallel data paths in the three levels. Finally, we used function *top_module()* to wrap up these sub-modules. This new C code structure eases the scheduling task for the HLS tool to generate efficient architectures.

We permutated the numbers of the three levels of data path replications in the restructured C code by modifying the *limit* parameter in the *HLS ALLOCATION* directives. We evaluated 10 different settings by running the entire HLS design tool chain including the placement and route phase. Fig. 10 depicts the evaluations of these HLS designs. The experimental results indicate that design *H* achieves the highest throughput efficiency among all the evaluated designs. We then implemented design *H* in RTL to further improve the resource efficiency.

Our RTL FPGA design consists of two modules: 1) boundary score module and 2) dynamic programming and maximum score module. To achieve a high throughput, we fully pipelined the FPGA architecture to output a new likelihood score every clock cycle. The two modules are able to run concurrently in a streaming fashion.

The architecture of the boundary score module is described in Fig. 11. In this figure, we demonstrate the boundary score module by only showing an example computing the

Listing 2: Pseudo Code for local score element computation function in HLS C code

```
void lkh_score(/*argument declaration*/)
{
 //compute likelihood score as shown in equation ref{equ:s_function}
}

void pipeline_unit(/*argument declaration*/)
{
    #pragma HLS ALLOCATION instances=lkh_score limit=16
    //use HLS ALLOCATION directive to control the number of lkh_score replications

    /*declare buffering variables for the 4 stages (4 columns)*/

    /*call lkh_score() to compute likelihood scores for 0 - 3 columns*/

    //pipeline
    /*stage0: max for column 0*/
    /*stage1: max for column 1*/
    /*stage2: max for column 2*/
    /*stage3: max for column 3*/
}

void alignment_module(/*argument declaration, e.g. input: DATA_TYPE x*/)
{
    #pragma HLS ALLOCATION instances=pipeline_unit limit=5
    /*use HLS ALLOCATION directive to control the number of pipeline_unit
        replications*/

    /*declare x, y line buffers:
    e.g. DATA_TYPE x0,x1,x2,x3,x4;*/

    for(/*iterate row index*/)
    {
        for(/*iterate column index*/)
        {
            //update the x line buffer
            x4=x3;
            x3=x2;
            x2=x1;
            x1=x0;
            x0=x;
            /*call pipeline_unit(x0,x1,x2,x3,x4,...)*/
        }
    }
}

void top_module(/*argument declaration*/)
{
    #pragma HLS ALLOCATION instances=alignment_module limit=2
    /*use HLS ALLOCATION directive to control the number of alignment_module
        replications*/

    /*call alignment_module()*/
}
```

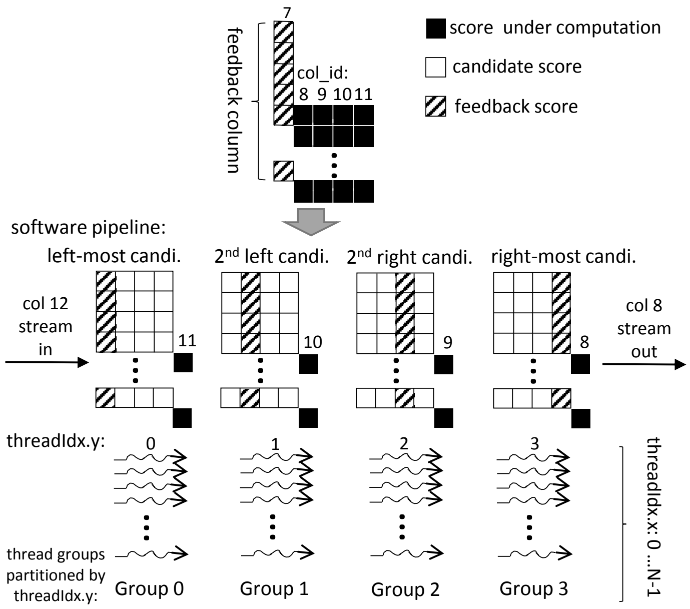


Fig. 8: Visualized GPU kernel for dynamic programming, assuming the score matrix size is $M \times N$. N rows $\times$ 4 columns of score elements are computed concurrently. For example, columns 8,9,10 and 11 are computed concurrently. At the given state in the example, column 11 is assigned to the threads whose $threadIdx.y = 0$. Then the leftmost candidates of column 11 are computed using the previously computed data in column 7. Similarly, columns 10,9 and 8 are assigned to $threadIdx.y = 1$, $threadIdx.y = 2$ and $threadIdx.y = 3$ respectively. N is set to a multiple of 32 to ensure each warp has the threads with the same $threadIdx.y$.

scores at the 4th top row. The other rows and columns are identical to this example. We replicated this architecture 16 times to process the top 4 rows, bottom 4 rows, left 4 columns and right 4 columns of boundary scores in parallel. This boundary score module is fully pipelined and consists of control logic (the black blocks in the figure), arithmetic units (the grey blocks), muxer and a shifting register for X. The control logic and arithmetic units correspond to Algorithm 1. The shifting register is for accessing $X_{Lx_{i,j}}$ and X_k as shown in lines 17 and 28.

The design of the dynamic programming module is described in Fig. 12. This architecture consists of 5 major pipeline stages as shown in Fig. 12. Stage 0 computes 16 (4×4) $Likelihood_{local}$ functions in parallel. These $Likelihood_{local}$ modules are fully pipelined. Stages 1 - 4 compute the maximums of the leftmost, second left, second right and rightmost columns of candidates, respectively.

We replicated the described architecture 5 times to process 5 rows of scores in parallel. After the last column of the current 5 rows, the next 5 rows will enter this architecture to continuously fill the pipeline. The output of all the rows are passed to a pipeline maximum module to find $score_{best}$, i_{best} and j_{best}. We chose to process 5 rows in parallel to match the throughput of the boundary score module. The two modules are thus able to run in a streaming fashion without idling.

As depicted in Fig. 12, the results of Stage 0 are delayed by the registers to feed Stages 2 - 4 at the correct cycles. Shown in the figure, the computation for $score[i][j+3]$ is at Stage 1; $score[i][j+2]$ is at Stage 2; $score[i][j+1]$ is at Stage 3; $score[i][j]$ is at Stage

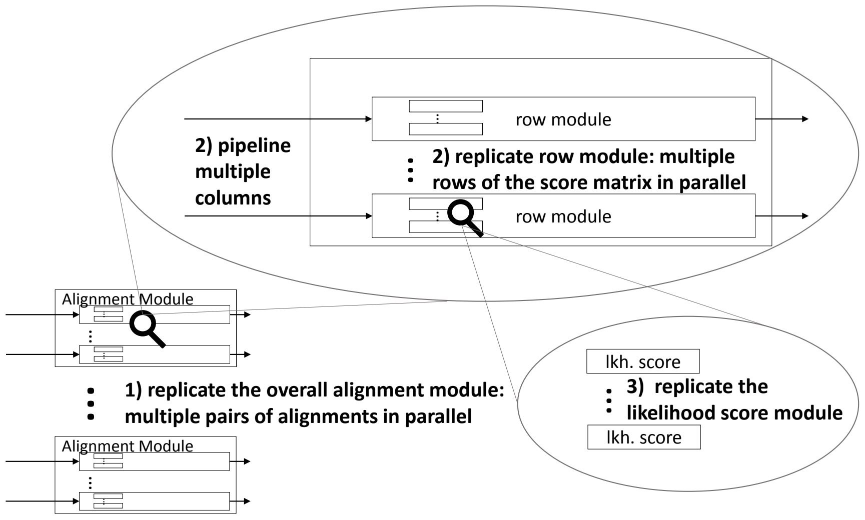

row module
2) pipeline multiple columns
2) replicate row module: multiple rows of the score matrix in parallel
row module
Alignment Module
1) replicate the overall alignment module: multiple pairs of alignments in parallel
Alignment Module
lkh. score
3) replicate the likelihood score module
lkh. score

Fig. 9: FPGA implementation of the three levels of parallelism. 1) replicate the overall architecture to process multiple alignments in parallel; 2) within each alignment score matrix, replicate the row modules to process multiple rows in parallel and pipeline multiple columns; 3) for each score element, replicate the likelihood score module to compute multiple likelihood scores in parallel.

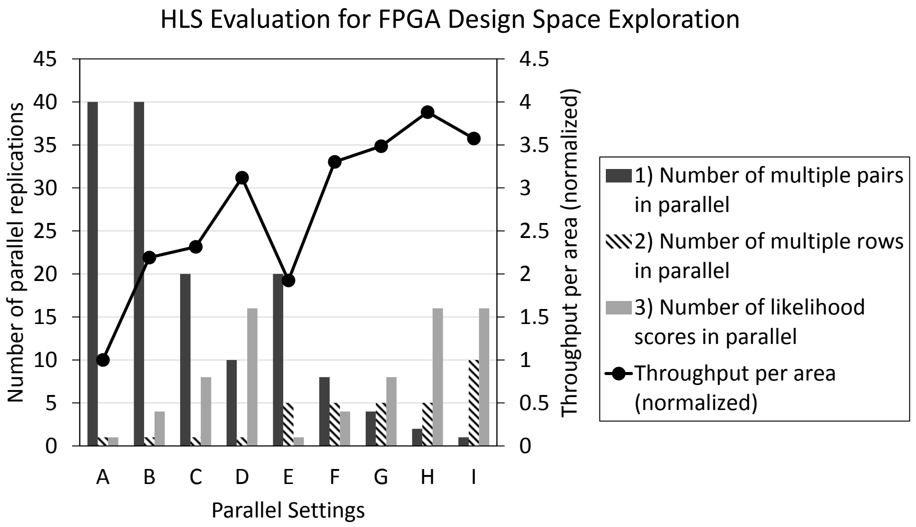


Fig. 10: FPGA design space exploration using HLS. Evaluations of 10 different designs on the Xilinx VC707 FPGA board (Design A - J). Throughput per area for each design is normalized to the lowest value.

4. These stages are all using $score[i-1:i+3][j-1]$. $score[i-1:i-4][j-1]$ are the scores created and stored in the BRAMs during the computation of the previous 5 rows. $score[i:i+3][j-1]$ are created from the previous cycle as a feedback loop. Therefore, in

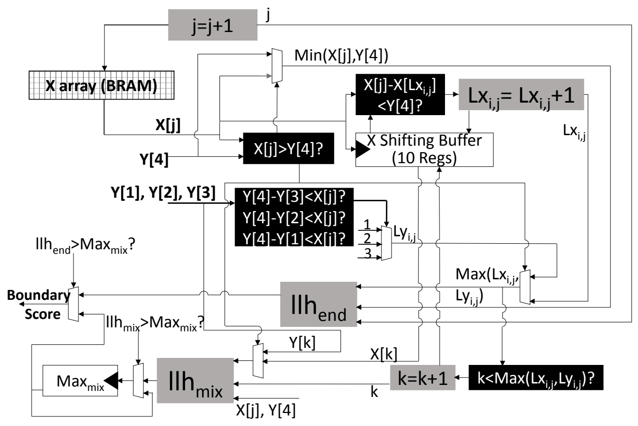


Fig. 11: FPGA boundary score module. The control logic corresponds to Algorithm 1. A shifting register storing 10 elements of X is used for accessing $X_{Lx_{i,j}}$ and X_k efficiently. llh_{end} and llh_{mix} represent the likelihood score modules for $Likelihood_{end}$ and $Likelihood_{mix}$.

order to keep the pipeline outputting new data every cycle with the constraint of this feedback loop, we designed a combinational logic to compute the 4 parallel additions and the "Max 5 to 1" operation within one clock cycle.

We used fixed point numbers and arithmetic in the FPGA design. Due to the data range, we used 26 bits for the scores and 18 bits for the input arrays, both with 10 decimal bits. These fixed point numbers can represent the optical labeled fragments up to $1,000,000$ bases. This range covers most human genome assembly applications. In the score functions, we implemented the divisions using lookup tables.

Since the RTL implementation is more resource efficient compared with the HLS implementation, we were able to replicate the overall alignment architecture 5 times to align 5 pairs of DNA molecules concurrently. In the HLS design, we were only able to replicate this architecture 2 times. We enhanced the resource efficiency by more than 200% using the RTL design.

5. RESULTS

The input data for the alignment algorithm in our experiment consists of 16642 DNA molecule fragments. Each fragment contains $5 - 182$ labels. The range of the distances between labels is $500 - 3.79 \times 10^5$ bases.

We tested the multi-core CPU design on a 3.1GHz Intel Xeon E5 CPU with 8 cores. The CPU design was compiled with O3 GCC optimizations. The GPU design was tested on a Nvidia Tesla K20 Kepler card. The FPGA design was implemented and tested on a Xilinx VC707 FPGA development board. The input data used in our experiments is a set of synthetic human genome sequences.

Fig. 13 presents the performance of our implementations. The baseline is a highly optimized C++ program without any parallelism. The average time for aligning two optical labeled molecules is $42.486\mu s$ in the baseline implementation. The run-times for the boundary score, dynamic programming and maximum score operations are $9.351\mu s$, $30.594\mu s$ and $2.541\mu s$ respectively.

The OpenMP parallelized C++ program consumes $5.04\mu s$ aligning a pair of molecules on an 8 core CPU with hyper-thread technology on each core. The performance of this multi-core implementation achieves a $8.4\times$ speedup which is proportional to the num-

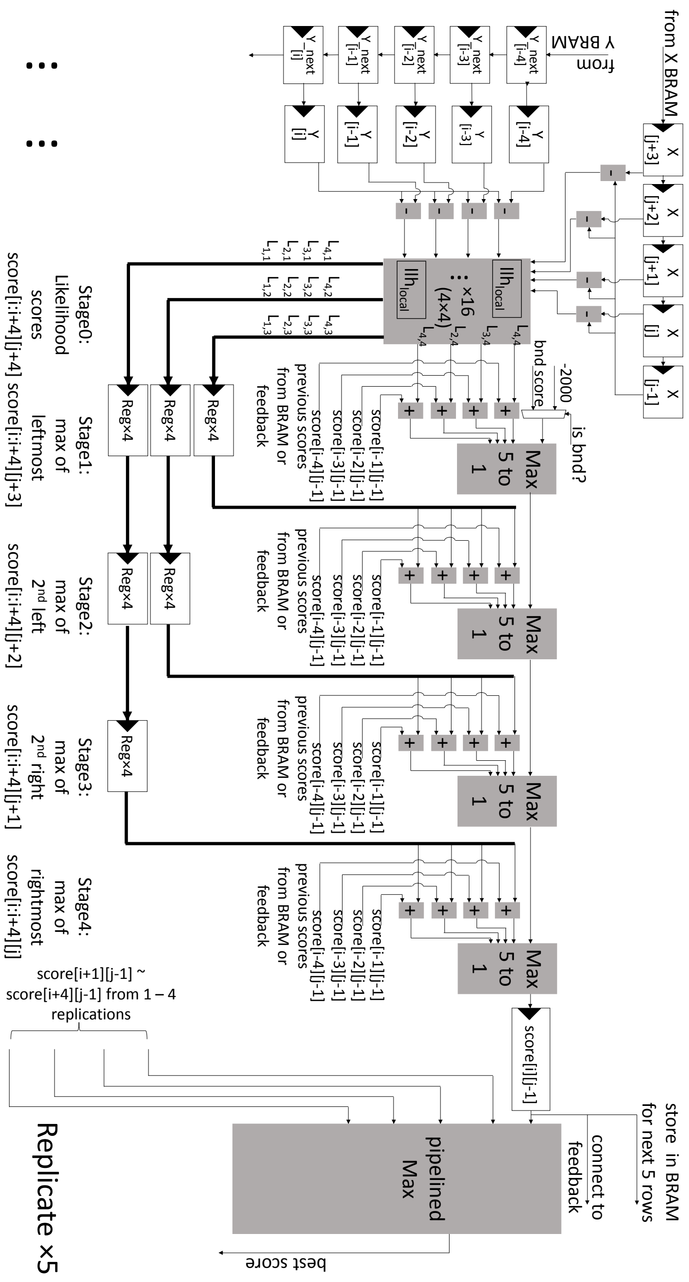


Fig. 12: FPGA dynamic programming module with three levels of concurrency. First, the 16 (4×4) $Likelihood_{local}$ functions are computed concurrently with 16 parallel llh_{local} modules. Second, this architecture is fully pipelined. $score[i][j+3]$, $score[i][j+2]$, $score[i][j+1]$ and $score[i][j]$ are processed concurrently in the different stages of the pipeline. Third, the architecture is replicated to process 5 rows in parallel.

ber of cores. The extra $0.4\times$ speedup is contributed by hyper-thread. The CPU SSE SIMD technique boosts the performance with a $11\times$ speedup against the baseline.

Our GPU implementation was written using the Nvidia CUDA 6.5 SDK. The GPU runs at a base frequency of 706 MHz and has 2496 CUDA cores. We varied the number of input alignments per host-device data transfer transaction from 10 to 10240 to investigate how the GPU design performs. As shown in Fig. 14 (B), the dynamic programming kernel converges to the minimal run-time after increasing the number of alignments per transaction to 2560. The boundary kernel and the max reduction kernel converge to their minimums when the number of alignments per transaction hits 640. The data copying operation from the device to host keeps speeding up with the increase of the number of alignments. This is due to the fact that the output array only contains very few data (each alignment only generates one max score, and two indices of the X and Y arrays) which never saturates the memory transaction bandwidth. However, the dynamic programming kernel dominates the overall run-time. Therefore, as a consequence, the overall performance saturates the max throughput after increasing the number of alignments to 2560.

The best performance of the GPU design is at $3.116\mu s$ per alignment with a $13.6\times$ speedup against the baseline. The run-time for the boundary score, dynamic programming and maximum score kernels are $0.940\mu s$, $1.484\mu s$ and $0.494\mu s$ respectively. The data transferring time between the host memory and GPU memory is $0.198\mu s$.

The FPGA design was built using Xilinx ISE 14.7 in Verilog. The FPGA design was implemented on a Xilinx Virtex 7 VC707 board receiving input and sending output using RIFFA [Jacobsen and Kastner 2013] (configured as a x8 Gen 2 PCIe connection to the PC). We also investigated how the FPGA design performs when changing the number of alignments per RIFFA transaction between the host CPU and the FPGA. We also varied the number of alignments per transaction from 10 to 10240. The simulated design generates the ideal throughput of the FPGA acceleration module. We measured the bandwidth of RIFFA by sending the data to the FPGA and receiving the same data back to the host without doing any computation on the FPGA. We only measured the host to device bandwidth since the transfer from the device to host contains very few data. As shown in Fig. 14 (A), the actual FPGA performance is significantly lower than the simulated ideal performance due to the RIFFA bandwidth limit when the number of alignments is between 10 and 640. After the number of alignments reaches 5120, the actual FPGA performance converges to its maximum which is slightly lower than the ideal performance due to the host software overhead.

Our FPGA experimental result shows the best throughput at 2.7 million pairs of molecules per second or equivalently $0.367\mu s$ per alignment. Thus, the FPGA implementation achieves a $115\times$ speedup against the baseline. Our FPGA implementation runs at a frequency of 125 MHz. Table II lists the resource utilization of the entire design including the PCIe communication logic. The design occupied 89% of the slices on the FPGA. In order to meet the timing constraint with such a high logic utilization, we used “SmartXplorer” to permutate multiple placement and route strategies in ISE. Since we used fixed-point number representation in the FPGA design, compared to the baseline floating-point design, we observed a 0.019% error which is negligible in real applications.

6. HARDWARE COMPARISON

Table III presents the summary of the comparison between the three hardware. We compare the performances and prices of the hardware accelerators.

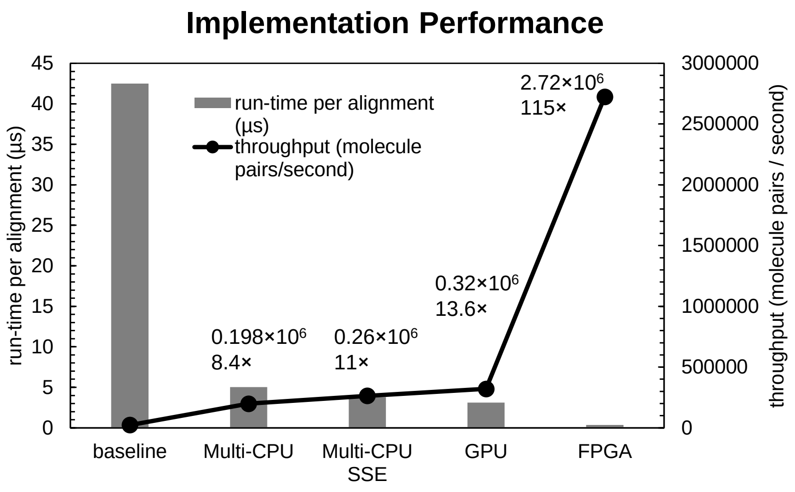


Fig. 13: Performance of the accelerated designs. Speed up factors against the single CPU baseline implementation. Throughput is defined as the number of DNA molecule pairs that a design is able to process in 1 sec.

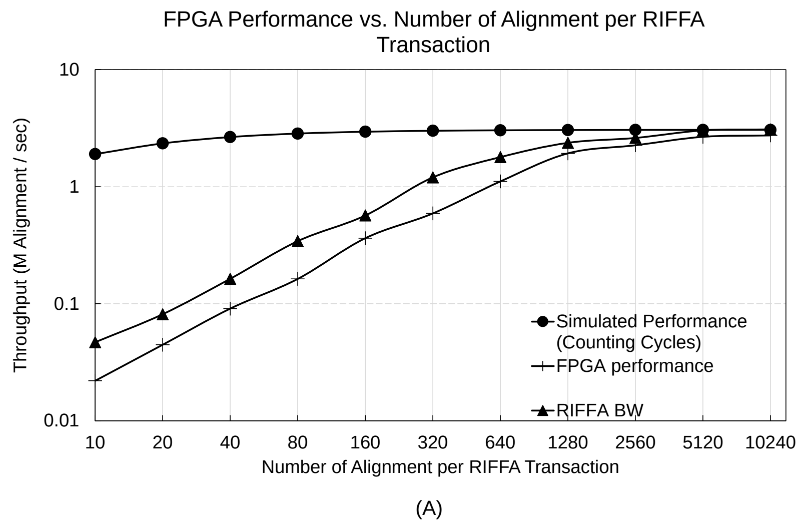
Table II: FPGA design resource utilization on VC707

Slice Reg.	Slice LUT.	BRAM	DSP48E
150412	251979	159	2280
24%	82%	15%	81%

6.1. Performance

Although the multi-core CPU has the highest operating frequency among the three hardware, it achieves the lowest speedup. This is due to the fact that the multi-core CPU has very limited parallel computing resources: 8 cores with hyper-thread. These cores are not closely coupled in the architecture. Frequently synchronizing these cores for fine-grained parallel computations becomes significantly expensive. Therefore, we were only able to utilize these cores to align multiple molecule pairs in a coarse-grained parallel fashion. The SSE SIMD extension provides a limited level of fine-grained parallelism. The CPU 128-bit SIMD extension does not provide dedicated SIMD units to achieve massive fine-grained parallelism.

The GPU, as opposed to the multi-core CPU, has a SIMD architecture that supports fine-grained parallelism. We therefore observed a higher speedup on the GPU. However, the control dominated boundary score computations introduce a significant amount of diverse instructions which harm the parallelism in the SIMD architecture. The GPU accelerates the dynamic programming algorithm by $20\times$ while it only accelerates the boundary score algorithm by $10\times$. Compared with the GPU, each CPU coarse-grained parallel core is processing each alignment in a sequential fashion which has more advantage in dealing with the control dominated instructions. Moreover, the array size differences create multiple *inactive* threads. With the CUDA profiler "nvvp", we observed that these inactive threads occupy more than 45% of the GPU computing resource due to the control branch diversities. Compared with the GPU design, the multi-core CPU and the FPGA suit this feature of the algorithm better. The multi-core CPU coarse-grained parallelism avoids inactive threads. Unlike the GPU threads



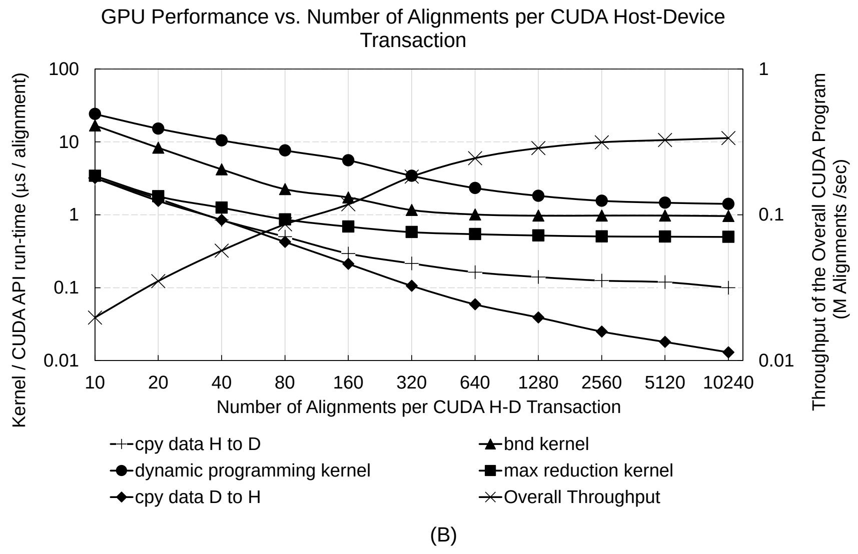


Fig. 14: Performance vs. number of alignments per host - accelerator device transaction. (A) FPGA performance vs. number of alignments per RIFFA send/receive transaction; the simulated performance is calculated by counting the cycles of the RTL design; the RIFFA bandwidth is measured in M alignments per second. (B) GPU performance (throughput) vs. number of alignments per CUDA H-D or D-H transaction; the run-time curve of each kernel and CUDA API is also plotted.

issued before the program starts and unchangeable during the run-time, the FPGA pipeline terminates and moves on to the next array when the current array finishes during the run-time. These unsuitable GPU features limit the performance for the alignment algorithm.

Table III: Hardware Comparison

Accelerator	Multi. Core CPU	GPU	FPGA
Parallel Architecture	Coarse-grain cores	Massively parallel threads	Replicated paths
Solution for data dependency	NA	Thread/Warp strategy	Manually design full pipeline & single-cycle logic
Customized Operator	NA	NA	Look up table division
Customized Bit-width	No	No	Yes
Frequency	3.1 GHz	706 MHz	125 MHz
Performance	$8.4\times$	$13.6\times$	$115\times$
DSE & Develop Effort	2 weeks	3 months	9 months
Price	\$2,000	\$3,200 + \$2,000	\$3,495 + \$2,000
Performance per \$ (aligns/sec/\$)	99.2	61.7	495.5

On the FPGA architecture, the customized logic avoids the diverse instruction issue in the boundary score algorithm. In the dynamic programming module, the pipeline on the FPGA is spatial. The data is transferred from one logic to the next logic using on-chip registers. In opposition, in the GPU design, we implemented a similar optimization using warps (different $threadIdx.y$) as shown in Listing 1 and Fig. 8. Each GPU warp represents a logic module on the FPGA. Unlike the spatial pipeline, the GPU warps are scheduled temporally. The data is not transferred spatially between warps. In contrast, the warps read or write the data on the shared memory. Although these warps are designed to be processed efficiently on the GPU, the FPGA spatial pipeline still outperforms the GPU warps without the overhead from scheduling and memory access. Moreover, the boundary score module stores its output in the low latency BRAM on the FPGA. The dynamic programming module can then access these boundary scores within one clock cycle. As opposed to the FPGA, the GPU dynamic programming kernel reads the boundary scores from the global memory with a higher latency. For these reasons, the FPGA implementation achieves the highest performance.

6.2. Hardware Prices

The prices of the acceleration hardware vary significantly depending on the complexities of the devices. The devices used in our implementations belong to the high-end category. The Xilinx VC707 FPGA evaluation board costs about $\$3,495$ [Xilinx 2014]. The Nvidia K20 GPU can be purchased for $\$3,200$. These high-end GPUs and FPGAs have comparable prices. The high-end CPU, Intel Xeon E5 has a relatively lower price which is roughly $\$2,000$. In our comparison, we added the CPU price to the cost of the GPU and FPGA accelerated systems since both of them used the CPU as a host to send and receive data. In our application, the performances per dollar are $99.2\ aligns/sec/\$$, $61.7\ aligns/sec/\$$ and $495.5\ aligns/sec/\$$ for the multi-core CPU, GPU and FPGA respectively.

7. CONCLUSION

In this article, we have addressed the necessity to accelerate the optical label based DNA assembly. We have presented three different accelerated approaches: a multi-core CPU implementation, a GPU implementation and a FPGA implementation. We have also presented the detailed design space explorations for these three approaches. The speedups over the sequential CPU baseline are $8.4\times$, $13.6\times$ and $115\times$ for the multi-core CPU, GPU and FPGA respectively. Using spatial pipelines, the FPGA design has been customized to suit the algorithm more efficiently than the other two hardware. The tradeoff to this performance efficiency on the FPGA is its significant design complexity in comparison with the approaches on the other two hardware.

REFERENCES

A.M. Aji, Liqing Zhang, and Wu chun Feng. 2010. GPU-RMAP: Accelerating Short-Read Mapping on Graphics Processors. In *Computational Science and Engineering (CSE), 2010 IEEE 13th International Conference on*. 168–175. DOI:http://dx.doi.org/10.1109/CSE.2010.29

Murat Baday, Aaron Cravens, Alex Hastie, HyeongJun Kim, Deren E Kudeki, Pui-Yan Kwok, Ming Xiao, and Paul R Selvin. 2012. Multicolor Super-resolution DNA Imaging for Genetic Analysis. *Nano letters* 12, 7 (2012), 3861–3866.

Inan Birol, Shaun D. Jackman, and others. 2009. *De novo* transcriptome assembly with ABySS. *Bioinformatics* 25, 21 (2009), 2872–2877. DOI:http://dx.doi.org/10.1093/bioinformatics/btp367

Alex R. Hastie, Lingli Dong, and others. 2013. Rapid Genome Mapping in Nanochannel Arrays for Highly Complete and Accurate *De Novo* Sequence Assembly of the Complex *Aegilops tauschii* Genome. *PLoS ONE* 8, 2 (02 2013), e55864. DOI:http://dx.doi.org/10.1371/journal.pone.0055864

M. Jacobsen and R. Kastner. 2013. RIFFA 2.0: A reusable integration framework for FPGA accelerators. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. 1–8. DOI:http://dx.doi.org/10.1109/FPL.2013.6645504

Ernest T Lam, Alex Hastie, and others. 2012. Genome mapping on nanochannel arrays for structural variation analysis and sequence assembly. *Nature Biotechnology* 30, 8 (2012), 771–776.

Chi-Man Liu, Thomas Wong, and others. 2012. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics* 28, 6 (2012), 878–879. DOI:http://dx.doi.org/10.1093/bioinformatics/bts061

Pingfan Meng, Matthew Jacobsen, Motoki Kimura, Vladimir Dergachev, Thomas Anantharaman, Michael Requa, and Ryan Kastner. 2014. Hardware accelerated novel optical de novo assembly for large-scale genomes. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. 1–8. DOI:http://dx.doi.org/10.1109/FPL.2014.6927499

C.B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W.L. Ruzzo. 2012. Hardware Acceleration of Short Read Mapping. In *FCCM '12*. 161–168. DOI:http://dx.doi.org/10.1109/FCCM.2012.36

Anton Valouev. 2006. *Shotgun Optical Mapping: A Comprehensive Statistical and Computational Analysis*. Ph.D. Dissertation. Los Angeles, CA, USA. Advisor(s) Waterman, Michael.

B.S.C. Varma, K. Paul, and M. Balakrishnan. 2014. Accelerating Genome Assembly Using Hard Embedded Blocks in FPGAs. In *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*. 306–311. DOI:http://dx.doi.org/10.1109/VLSID.2014.59

B. Sharat Chandra Varma, Kolin Paul, M. Balakrishnan, and Dominique Lavenier. 2013. FAssem: FPGA Based Acceleration of De Novo Genome Assembly. In *FCCM '13*. IEEE Computer Society, Washington, DC, USA, 173–176. DOI:http://dx.doi.org/10.1109/FCCM.2013.25

Xilinx. 2014. Xilinx Products. http://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html. (2014). [Online].

Andrea Zuccolo, Aswathy Sebastian, Jayson Talag, Yeisoo Yu, HyeRan Kim, Kristi Collura, Dave Kudrna, and Rod A Wing. 2007. Transposable element distribution, abundance and role in genome size variation in the genus Oryza. *BMC Evolutionary Biology* 7, 1 (2007), 152.