# Hardware Accelerated Novel Optical *De Novo* Assembly for Large-Scale Genomes

Pingfan Meng*†, Matthew Jacobsen*†, Motoki Kimura*‡, Vladimir Dergachev§, Thomas Anantharaman§, Michael Requa§ and Ryan Kastner*†
*Department of Computer Science and Engineering
University of California, San Diego, USA
Email: †{pmeng, mdjacobs, kastner}@cs.ucsd.edu ‡motoki.kimura.xr@renesas.com
§BioNano Genomics, USA
Email: {vdergachev, tanantharaman, mrequa}@bionanogenomics.com

***Abstract*—*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 paper, 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 reveal the insights of the design space exploration of this new assembly algorithm on these three different devices by comparing the results.**

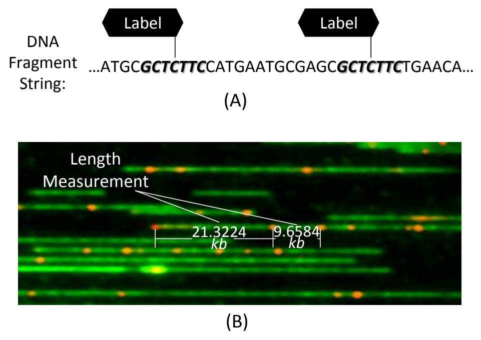


Fig. 1. Demonstration of the optical labeling process. (A) Fluorescent labels attached to "GCTCTTC". (B) The real image field of labeled DNA fragments from a microscopy CCD camera. The strings are linearized DNA fragments. The glowing dots are fluorescent labels. The numbers in kilo-bases(*kb*) are examples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

## I. 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 [1]. 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 [2]. 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 [3]).

In recent years, research has shown that a novel optical label based technology is able to overcome this limitation of the short-read technology [4]. 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 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 [2].

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

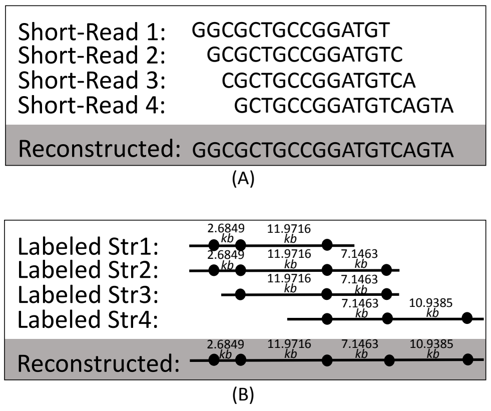

Short-Read 1: GGCGCTGCCGGATGT
Short-Read 2: GCGCTGCCGGATGTC
Short-Read 3: CGCTGCCGGATGTCA
Short-Read 4: GCTGCCGGATGTCAGTA
Reconstructed: GGCGCTGCCGGATGTCAGTA
(A)
Labeled Str1:
Labeled Str2:
Labeled Str3:
Labeled Str4:
Reconstructed:
2.6849 kb
11.9716 kb
7.1463 kb
10.9385 kb
(B)

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.

$10,000$ hours on a sequential CPU. Moreover, research [5] 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 paper, we present three acceleration 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 in 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.

The rest of the paper is organized as follows. We discuss related work in Section II. We describe the alignment algorithm in Section III. This is followed by descriptions of the accelerated designs in Section IV. Experimental performance results and comparison are provided in Section V. We conclude in Section VI.

## II. 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 [6]. 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 [7]. 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 [8]. 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 [9]. They reported a $9.6\times$ speedup. Liu *et al*. proposed a GPU accelerated DNA assembly tool - SOAP3 [10] 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 paper, our accelerated designs differ from the previous short-read approaches in two ways: 1) the 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 [11] 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 $4\times4$ area as demonstrated in Fig. 3. 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.

## III. Algorithm

The accelerated alignment algorithm is a Dynamic Programming method specifically modified for the optical DNA analysis, proposed by Valouev [11]. The algorithm aligns two arrays $X$ and $Y$ of optical label positions by computing a likelihood 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. 3. 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.

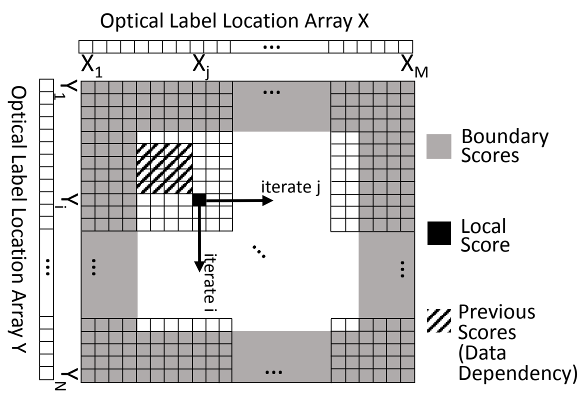

Optical Label Location Array X
$X_1$
$X_j$
$X_M$
Optical Label Location Array Y
$Y_1$
$Y_i$
$Y_N$
iterate j
iterate i
Boundary Scores
Local Score
Previous Scores (Data Dependency)

Fig. 3. 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.

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. Therefore, we compute a boundary score to represent this offset. 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. 3.

Besides the offsets on the ends, when evaluating the likelihood of the alignment between $X_j$ and $Y_i$, we also need to consider how well their neighboring elements align. We compute a local score to represent this. In Algorithm 2, to compute each local score $score_{i,j}$, we generate 16 score candidates correspondent to its upper-left $4\times4$ neighbors (refer to Algorithm 2 lines 5 - 7). Each of the 16 candidates is computed by adding a local likelihood to its correspondent previous score from the $4 \times 4$ area (the shaded area in Fig.3). The score in $score_{i,j}$ is updated with the maximum among all these $4 \times 4$ 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 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 [11]. 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:

**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
```

**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 $i_{best}$
```

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 [11]. 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.

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

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

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

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

## IV. Accelerated Designs

The algorithm consists of three levels of possible parallelism: 1) align multiple pairs in parallel; 2) compute multiple rows and columns of the score matrix in parallel; 3) compute the 16 score candidates for each score element in parallel. The input DNA fragment pool typically has $100,000 - 1,000,000$ arrays. A typical input array length is 15 - 100. In our accelerated designs, we mapped these computational patterns to the particular architectural features of each hardware. In the following sections, we describe these designs in detail.

### A. Multi-core CPU

In the CPU design, we 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. 4. When one CPU core finishes its current alignment workload, it can start aligning another pair immediately without synchronizing with the other CPU cores. Therefore, all the CPU cores are completely occupied during this process to maximize the throughput.

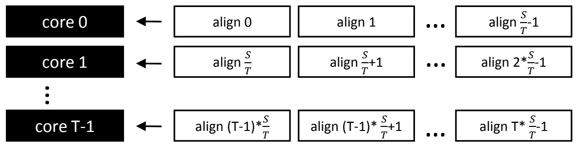


Fig. 4. Multi-core CPU accelerated design. Assume there are $S$ pairs of optical arrays to be aligned and the CPU has $T$ cores.

### B. GPU

The GPU design consists of three CUDA kernels, invoked from a C++ host code. The CUDA kernels accelerate the entire 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

TABLE I
CONSTANT VALUES FOR SCORE FUNCTIONS

| Constant | $C$ | $V$ | $\delta$ | $B$ | $B'$ | $P_{miss}$ | $R$ | $P$ | $B_{end}$ | $B'_{end}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Value | $3.6505$ | $0.0449$ | $0.0010$ | $-0.0705$ | $0.9144$ | $1.5083$ | $-0.0931$ | $-8.1114$ | $0.0226$ | $0.3992$ |

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. 5 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. 5. 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 (an efficient $M \times N$-element reduction kernel) searches the matrix to find $score_{best}$, $i_{best}$ and $j_{best}$.

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();
    }
}
```

## C. FPGA

The FPGA also accelerates the entire algorithm while a host PC program is responsible for sending the input and receiving the output. Our final implementation is in Register-transfer level (RTL). Our design space exploration was performed using High-Level Synthesis (HLS) .

The FPGA is a customizable architecture. There are two techniques to implement parallelism on the FPGA: 1) replicate a logic module to create actual parallel data paths; 2) pipeline the architecture to process the multiple data concurrently in a streaming fashion. To achieve a high performance, our FPGA design utilizes a combination of these two techniques. However, different combinations of these two techniques result in very different performances. In our development process, this required us to explore multiple designs to find the optimal combination.

RTL design requires a significant amount of effort. HLS effectively reduces this effort. Therefore, we evaluated the different designs by using Vivado HLS. The possible replicated parallel data paths and pipelines we explored are depicted in Fig. 6. Shown in Fig. 7, the evaluation indicates that the strategy with the highest throughput is the combination of replicating candidate modules, replicating the row module, pipelining the architecture to stream multiple columns and replicating the overall module. Besides the throughput, this strategy requires the least FPGA input/ouput bandwidth since it only inputs and outputs the data for two pairs of DNA fragments concurrently. Thus, we selected this strategy. However, the FPGA designs using HLS are resource inefficient compared to the ones using RTL design. We implemented the selected strategy in RTL to achieve a better 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. 8. In this figure, we demonstrate the boundary score module by only showing an example computing the 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

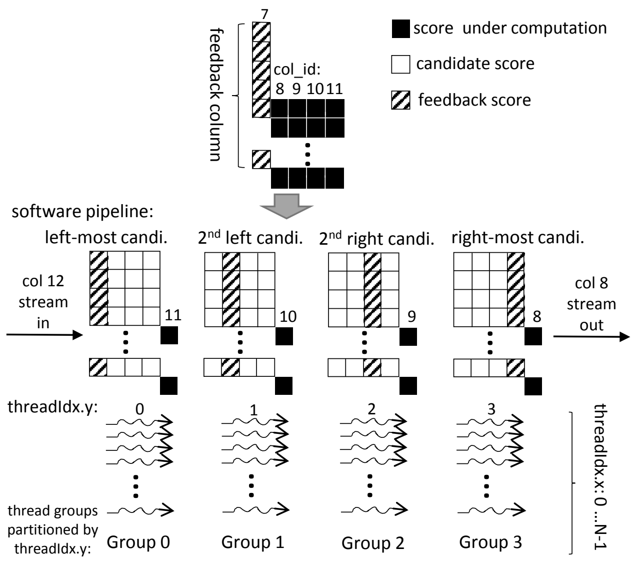
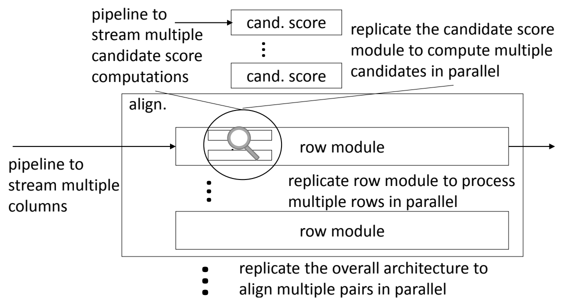


Fig. 5. Visualized GPU kernel for dynamic programming, assuming the score matrix size is $M \times N$. $N$ rows $\times$ 4 columns of score elements are computed concurrently. For example, columns 8,9,10 and 11 are computed concurrently. At the given state in the example, column 11 is assigned to the 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 $threadIdx.y$.



pipeline to stream multiple candidate score computations
cand. score
cand. score
replicate the candidate score module to compute multiple candidates in parallel
align.
row module
row module
pipeline to stream multiple columns
replicate row module to process multiple rows in parallel
replicate the overall architecture to align multiple pairs in parallel

Fig. 6. FPGA pipeline and parallel data paths hierarchy.

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. 9. This architecture consists of 5 major pipeline stages as shown in Fig. 9. Stage 0 computes 16 ($4 \times 4$) $Likelihood_{local}$ 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

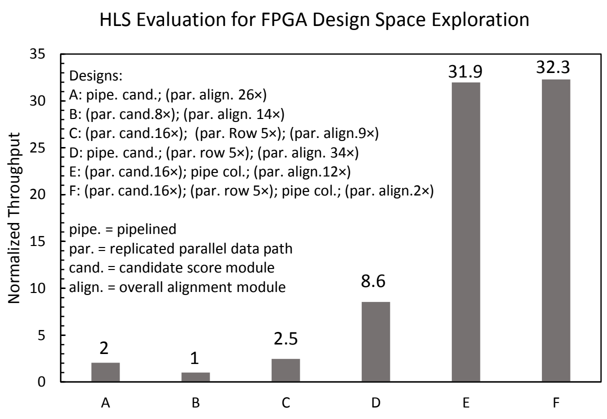


Fig. 7. FPGA design space exploration using HLS. 6 different designs that fit Xilinx VC707 FPGA (Design A - F). Throughput for each design is normalized to the slowest design.

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.

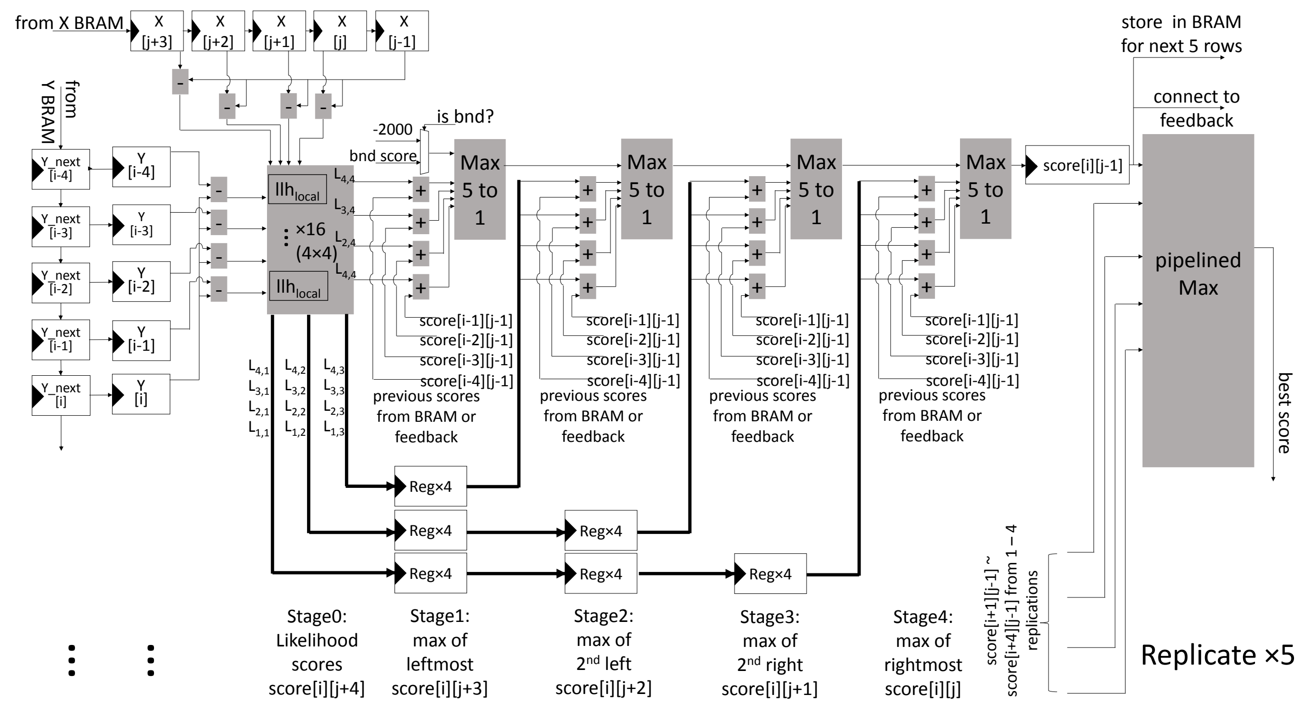
As depicted in Fig. 9, the results of Stage 0 are delayed by the registers to feed Stages 2 - 4 at the correct cycles. Shown in the figure, the computation for $score[i][j+3]$ is at Stage 1; $score[i][j+2]$ is at Stage 2; $score[i][j+1]$ is at Stage 3; $score[i][j]$ is at Stage 4. These stages are all 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 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. In the score functions, we implemented the divisions using lookup tables.

In order to fully utilize the Virtex 7 FPGA resource, we replicated the overall alignment architecture in the RTL design 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.

## V. Results and Comparison

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.



from X BRAM
from Y BRAM
llh_local
×16 (4×4)
Max 5 to 1
pipelined Max
store in BRAM for next 5 rows
connect to feedback
score[i][j-1]
best score
previous scores from BRAM or feedback
Reg×4
Stage0: Likelihood scores score[i][j+4]
Stage1: max of leftmost score[i][j+3]
Stage2: max of 2nd left score[i][j+2]
Stage3: max of 2nd right score[i][j+1]
Stage4: max of rightmost score[i][j]
score[i+1][j-1] ~ score[i+4][j-1] from 1 – 4 replications
Replicate ×5

Fig. 9. FPGA dynamic programming module with three levels of concurrency. First, the 16 ($4 \times 4$) $Likelihood_{local}$ 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.

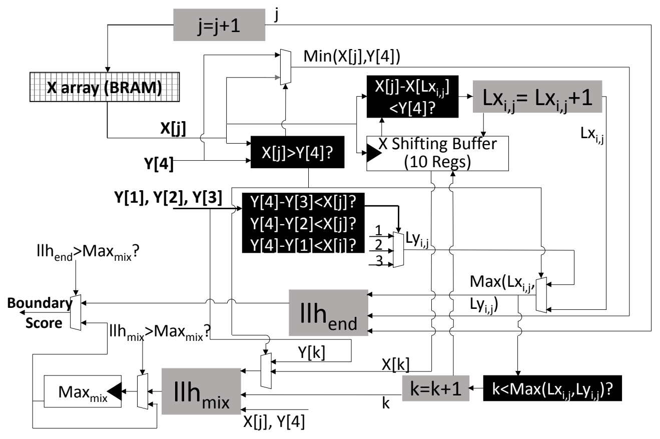


Fig. 8. FPGA boundary score module. The control logic corresponds to Algorithm 1. A shifting register storing 10 elements of $X$ is used for accessing $X_{Lx_{i,j}}$ and $X_k$ efficiently. $llh_{end}$ and $llh_{mix}$ represent the likelihood score modules for $Likelihood_{end}$ and $Likelihood_{mix}$.

## A. Experimental Results

Fig. 10 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 number of cores. The extra $0.4\times$ speedup is contributed by hyper-thread.

Our GPU implementation was written using the Nvidia CUDA 5.5 SDK. We manually tuned our CUDA program to achieve a 100% computing resource occupancy on the Nvidia K20 GPU. The GPU runs at a base frequency of 706 MHz and has 2496 CUDA cores. The 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 [12] (configured as a x8 Gen 2 PCIe connection to the PC). Our FPGA experimental result shows a 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. Based on the resource utilization result, we estimate that more replications are able to fit on a VC709 FPGA board to align 8 pairs in parallel and result in a $184\times$ speedup over

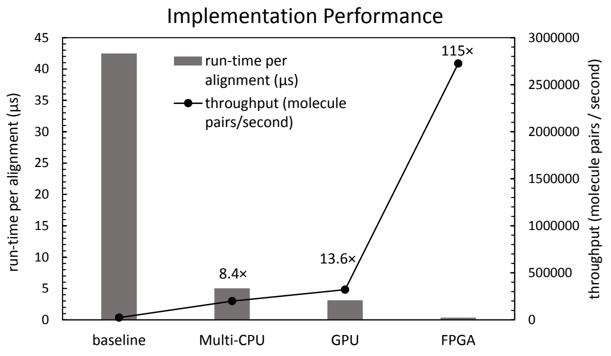


Fig. 10. Performance of the accelerated designs. Speed up factors against the single CPU baseline implementation. Throughput is defined as the number of DNA molecule pairs that a design is able to process in 1 sec.

TABLE II
FPGA DESIGN RESOURCE UTILIZATION ON VC707

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

the baseline. 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.

## B. Hardware Comparison

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 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 $10\times$.

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. 5. 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 high latency global memory. For these reasons, the FPGA implementation achieves the highest performance.

## VI. CONCLUSION

In this paper, 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. 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. We estimate the design is scalable to process more alignments in parallel on a larger FPGA device.

## REFERENCES

[1] I. Birol, S. D. Jackman *et al.*, "*De novo* transcriptome assembly with abyss," *Bioinformatics*, vol. 25, no. 21, pp. 2872–2877, 2009.

[2] A. R. Hastie, L. Dong *et al.*, "Rapid genome mapping in nanochannel arrays for highly complete and accurate *De Novo* sequence assembly of the complex *Aegilops tauschii* genome," *PLoS ONE*, vol. 8, no. 2, p. e55864, 02 2013.

[3] A. Zuccolo, A. Sebastian, J. Talag, Y. Yu, H. Kim, K. Collura, D. Kudrna, and R. A. Wing, "Transposable element distribution, abundance and role in genome size variation in the genus oryza," *BMC Evolutionary Biology*, vol. 7, no. 1, p. 152, 2007.

[4] E. T. Lam, A. Hastie *et al.*, "Genome mapping on nanochannel arrays for structural variation analysis and sequence assembly," *Nature Biotechnology*, vol. 30, no. 8, pp. 771–776, 2012.

[5] M. Baday, A. Cravens, A. Hastie, H. Kim, D. E. Kudeki, P.-Y. Kwok, M. Xiao, and P. R. Selvin, "Multicolor super-resolution dna imaging for genetic analysis," *Nano letters*, vol. 12, no. 7, pp. 3861–3866, 2012.

[6] C. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. Ruzzo, "Hardware acceleration of short read mapping," in *FCCM '12*, April 2012, pp. 161–168.

[7] B. S. C. Varma, K. Paul, M. Balakrishnan, and D. Lavenier, "Fassem: Fpga based acceleration of de novo genome assembly," in *FCCM '13*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 173–176.

[8] B. Varma, K. Paul, and M. Balakrishnan, "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*, Jan 2014, pp. 306–311.

[9] A. Aji, L. Zhang, and W. chun Feng, "Gpu-rmap: Accelerating short-read mapping on graphics processors," in *Computational Science and Engineering (CSE), 2010 IEEE 13th International Conference on*, Dec 2010, pp. 168–175.

[10] C.-M. Liu, T. Wong *et al.*, "Soap3: ultra-fast gpu-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 2012.

[11] A. Valouev, "Shotgun optical mapping: A comprehensive statistical and computational analysis," Ph.D. dissertation, Los Angeles, CA, USA, 2006.

[12] M. Jacobsen and R. Kastner, "Riffa 2.0: A reusable integration framework for fpga accelerators," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–8.