Accelerating Long Read Alignment on Three Processors

Zonghao Feng
HKUST
Kowloon, Hong Kong
zfengah@cse.ust.hk

Shuang Qiu
HKUST
Kowloon, Hong Kong
sqiuac@cse.ust.hk

Lipeng Wang
HKUST
Kowloon, Hong Kong
lwangay@cse.ust.hk

Qiong Luo
HKUST
Kowloon, Hong Kong
luo@cse.ust.hk

ABSTRACT

Sequence alignment is a fundamental task in bioinformatics, because many downstream applications rely on it. The recent emergence of the third-generation sequencing technology requires new sequence alignment algorithms that handle longer read lengths as well as more sequencing errors. Furthermore, the rapidly increasing volume of sequence data calls for efficient analysis solutions. To address this need, we propose to utilize commodity parallel processors to perform the long read alignment. Specifically, we propose manymap, an acceleration of the leading CPU-based long read aligner minimap2 on the CPU, the GPU, and the Intel Xeon Phi processor. We eliminate intra-loop data dependency in the base-level alignment step of the original minimap2 through redesigning memory layouts of dynamic programming (DP) matrices. This change facilitates the effective vectorization of the most time-consuming procedure in alignment. Additionally, we apply architecture-aware optimizations, such as utilizing high bandwidth memory on Xeon Phi and concurrent kernel execution on GPU. We evaluate our manymap in comparison with the extended minimap2 on a Xeon Gold 5115 CPU, a Tesla V100 GPU, and a Xeon Phi 7210 processor. Our results show that manymap outperforms minimap2 by up to 2.3 times on the overall execution time and 4.5 times on the base-level alignment step.

CCS CONCEPTS

• **Applied computing** → **Life and medical sciences**; • **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

sequence alignment, parallel processing

ACM Reference Format:
Zonghao Feng, Shuang Qiu, Lipeng Wang, and Qiong Luo. 2019. Accelerating Long Read Alignment on Three Processors. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337918

1 INTRODUCTION

Sequence alignment is the process of aligning biological sequences to identify their similar regions. Among various bioinformatics tasks, sequence alignment is essential in that it is a basic step in many downstream applications, such as genome assembly and association studies. The dynamic programming based alignment algorithms [1] can produce optimal results, but are very time-consuming. To deal with large volumes of sequencing data efficiently, heuristic alignment algorithms using the *seed-and-extend* strategy are proposed [10]. The idea is to locate matches of short subsequences, called *seeds*, then use optimal alignment algorithms to extend *seeds* to get complete alignment results.

While the time spent on sequencing has been greatly reduced over years, sequence analysis has become the bottleneck of the entire pipeline, which may take hours or even days. The gap between the quick growth of data and limited analysis capabilities is severe [18]. To fill this gap, modern hardware, e.g., graphics processing unit (GPU) and Intel Xeon Phi processor, are utilized to accelerate bioinformatics applications [15, 25].

Recently, with the invention of third generation sequencing technologies, substantially longer reads can be produced rapidly. Representatives include the Pacific Biosciences (PacBio) single molecular real-time (SMRT) sequencing and the Oxford Nanopore Sequencing. Compared with the short reads produced by second generation sequencing, third generation sequences are two orders of magnitude longer. However, the error rate also becomes higher, which poses great difficulties in sequence analysis [6].

Since most existing sequence analysis algorithms are inapplicable to the long and noisy reads, new algorithms targeted at third generation sequences are proposed. Minimap2 [9] is the leading algorithm for long read alignment. It is more than 30 times faster than some of its competitors, and its mapping quality is the highest.

To achieve efficient long read alignment on modern processors, we propose manymap, which is a fully parallelized long read aligner optimized for CPU, GPU and Intel Xeon Phi Knights Landing (KNL) processor. We build manymap on the basis of minimap2, as it is a state-of-the-art long read aligner on the CPU. The design of manymap aims at exploiting the full feature set of hardware to maximize the performance. Specifically, our contributions are:

- We profile the original minimap2 on CPU and KNL and identify that the base-level alignment step is the performance bottleneck. To accelerate this step, we propose a new memory layout of DP matrices, which eliminates data dependency and improves vectorization on three processors.
- On KNL, we optimize memory access by using the high bandwidth Multi-Channel DRAM (MCDRAM) and memory-mapped I/O, which reduces the indexing loading time by half. Additionally, we adopt an optimized thread affinity strategy to resolve thread resource contention and achieve effective pipelining.
- On GPU, by using concurrent kernel execution, we launch up to 128 CUDA streams with 512 threads each to maximize

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337918

Table 1: Overview of sequence alignment algorithms

	CPU	GPU	Xeon Phi
Optimal Alignment	Striped [4], SWIPE [21]	CUDASW++ 3.0 [15]	XSW [25], SWAPHI [14]
Short Read Alignment	BWA [10], GEM [17]	CUSHAW2-GPU [13]	MICA [16]
Long Read Alignment	BLASR [3], minimap2 [9]	Our work	

parallelism. We optimize memory access by utilizing shared memory and adopting a memory pool.
- We perform an extensive study of performance on three processors. On the base-level alignment step, manymap outperforms minimap2 by up to 4.5, 3.9, 3.2 times on CPU, GPU, and KNL, respectively. The overall execution performance is increased by 1.4 and 2.3 times on CPU and KNL, respectively.

2 BACKGROUND AND RELATED WORK

2.1 Sequence Alignment Algorithms

We briefly review the representative sequence alignment algorithms, as shown in Table 1. First of all, the most basic type is dynamic programming based alignment algorithms, which aligns sequences at base-level and produces optimal result. It is widely used for sequence database search, and also plays a key part in heuristic alignment algorithms. To accelerate dynamic programming, various vectorization algorithms are proposed. For example, SWIPE [21] uses an inter-sequence parallel approach, which has been accelerated on GPU [15] and Xeon Phi [25].

Short read alignment algorithms usually follow the *seed-and-extend* heuristic, since matches of *seeds*, i.e., short subsequences, are likely to be part of the complete alignment. They use hash tables or Burrow-Wheeler Transform [2] based FM-index [5] to query matches of *seeds*, then extend them with base-level alignment algorithms. There are many short read alignment accelerations, such as CUSHAW2-GPU [13] on GPU, and MICA [16] on Xeon Phi. They accelerate seed matching query with redesigned index structure. Additionally, they also benefit from the vectorization methods of base-level alignment.

With years of active research, short read alignment is no longer the bottleneck of data analysis [11]. However, the invention of third-generation sequencing technologies poses a new challenge of designing fast and accurate long read alignment algorithms that are capable of processing sequences with long read length and high error rate. Minimap2 [9] is the state-of-the-art long read alignment algorithm with both high efficiency and accuracy. Nevertheless, accelerations of long read alignment algorithms on modern hardware have not been adequately studied yet. Our work fills this gap by accelerating minimap2 on CPU and extending it to GPU and KNL.

2.2 Many-core Processors

GPUs are many-core processors that are widely used in general purpose computations. GPUs use the single instruction, multiple threads (SIMT) execution model, i.e., instructions are executed concurrently on groups of threads called *warps*. A single routine on the GPU, called *kernel*, can have multiple thread *blocks*. Moreover, multiple *kernels* can be executed on GPU concurrently. This multi-tier parallel model enables massive parallelism.

The second generation Intel Xeon Phi processor, code named Knights Landing (KNL), is another popular many-core processor with 64 cores in a single chip. It is a fully independent self-bootable processor, which does not rely on a host CPU and eliminates the data transfer cost. In addition, KNL has 2 vector processing units (VPU) per core, which support the AVX-512 SIMD instruction set. Additionally, KNL contains 16 GB on-board MCDRAM, which provides higher memory bandwidth than DDR RAM. Although existing CPU-based code can run on KNL without any modifications due to the x86 compatibility, performance gain is not promised without architecture-aware optimizations. Through tuning programs on KNL, some optimizations can also boost the performance on CPU, which is called the "dual-tuning" effect [7].

Compared with modern multi-core CPUs, many-core processors have a large number of low-performance cores and are specially designed for highly-parallel tasks. Therefore, to achieve the ideal performance on many-core processors, it is required to reduce serial execution and increase parallelism through techniques like vectorization.

3 THE MINIMAP2 LONG READ ALIGNMENT ALGORITHM

In this section, we introduce the minimap2 [9] long read alignment algorithm, which is the most popular one among all the long read aligners, and the basis of our work.

3.1 Workflow

The design of long read aligners shares much commonality with short read aligners. Similar to the *seed-and-extend* heuristic that most short read alignment algorithm uses, minimap2 is based on the *seed-chain-extend* heuristic.

Firstly, in the seeding step, minimap2 extracts short subsequences, called *seeds*, from the query sequence and the reference sequence, then checks if there are any matches of the seeds from the query to the reference. The choice of seeding strategy is of critical importance to the accuracy and sensitivity of the alignment. Minimap2 uses minimizers [20] as seeds to suit the new error profile of third-generation sequences. Additionally, minimap2 stores seeds in a hash table to perform lookups efficiently.

Secondly, for each query, the matched seeds, called *anchors*, are clustered in the chaining step. Based on their positions, sets of colinear anchors are located as *chains*. The chains are approximate overlaps between the query and the reference, and can be further extended to get complete alignment results.

Finally, in the aligning step, base-level alignments are performed to fill the gaps between anchors in chains. The base-level alignment compares two sequences base by base using optimal alignment methods. This step is the most compute-intensive part of the entire algorithm, and thus is the primary focus of our acceleration.

3.2 Base-level Alignment with Difference-based Formula

We describe the details of the base-level alignment step of minimap2. Minimap2 performs semi-global alignment, in which the beginnings of two sequences must be aligned but their ends are free of penalty. For simplicity, we use one-piece affine-gap penalty $q + k \cdot e$ in the following formulas, where q is the gap open cost and e is the gap extension cost.

Given the target sequence T and the query sequence Q, the dynamic programming formula is shown in Equation (1). $s(T_i, Q_j)$ is the substitution score between T_i and Q_j. $H_{i,j}$ is the score of aligning prefixes of T and Q ending at T_i and Q_j, and $E_{i,j}, F_{i,j}$ are scores with a gap ends at T or Q. [1]

$$\begin{cases} H_{ij} = \max\{H_{i-1,j-1} + s(T_i, Q_j), E_{ij}, F_{ij}\} \\ E_{i+1,j} = \max\{H_{ij} - q, E_{ij}\} - e \\ F_{i,j+1} = \max\{H_{ij} - q, F_{ij}\} - e \end{cases} \tag{1}$$

To accelerate the time-consuming optimal alignment process, SIMD vectorization is widely adopted. Since cells on the same anti-diagonal of the DP matrix have no data dependency on each other, they can be loaded into vectors and updated in parallel. However, for long reads, the alignment score is too big to fit into 8-bit or even 16-bit integers using traditional SIMD implementations. As a result, recalculations have to be performed with large units in vectors, which reduces the parallelism. To address this issue, minimap2 uses the difference-based formula [24] to achieve space-efficient vectorization. Instead of storing the complete score, it calculates and stores the differences between cells, which are small enough to fit into 8-bit integers. We define:

$$\begin{cases} u_{ij} \triangleq H_{ij} - H_{i-1,j} & v_{ij} \triangleq H_{ij} - H_{i,j-1} \\ x_{ij} \triangleq E_{i+1,j} - H_{ij} & y_{ij} \triangleq F_{i,j+1} - H_{ij} \end{cases}$$

Then, we can derive a difference-based formula from Equation (1):

$$\begin{cases} z_{ij} &= \max\{s(T_i, Q_j), x_{i-1,j} + v_{i-1,j}, y_{i,j-1} + u_{i,j-1}\} \\ u_{ij} &= z_{ij} - v_{i-1,j} \\ v_{ij} &= z_{ij} - u_{i,j-1} \\ x_{ij} &= \max\{0, x_{i-1,j} + v_{i-1,j} - z_{ij} + q\} - q - e \\ y_{ij} &= \max\{0, y_{i,j-1} + u_{i,j-1} - z_{ij} + q\} - q - e \end{cases} \tag{2}$$

For the ease of computation, minimap2 transforms the coordinate into $r \leftarrow i + j$ and $t \leftarrow i$. We then have Equation (3) for the optimal alignment using the difference-based formula.

$$\begin{cases} z_{rt} &= \max\{s(T_t, Q_{r-t}), x_{r-1,t-1} + v_{r-1,t-1}, y_{r-1,t} + u_{r-1,t}\} \\ u_{rt} &= z_{rt} - v_{r-1,t-1} \\ v_{rt} &= z_{rt} - u_{r-1,t} \\ x_{rt} &= \max\{0, x_{r-1,t-1} + v_{r-1,t-1} - z_{rt} + q\} - q - e \\ y_{rt} &= \max\{0, y_{r-1,t} + u_{r-1,t} - z_{rt} + q\} - q - e \end{cases} \tag{3}$$

4 LONG READ ALIGNMENT ON MODERN PROCESSORS

In this section, we introduce the design and implementation of our manymap long read aligner. We first analyze the profiling result of minimap2 to identify the performance bottleneck in Section 4.1 and give an overview of the optimized workflow in Section 4.2. Then we introduce the base-level alignment algorithm of manymap in Section 4.3 and architecture-aware optimizations on KNL and GPU in Sections 4.4 and 4.5.

4.1 Profiling

Given the x86 compatibility of the Knights Landing processor, the original minimap2 program can run on KNL without any code modification or recompilation. However, it is unclear whether there is any performance improvement, and if so, how much the improvement is and where it comes from.

Table 2: Performance breakdown of minimap2

	CPU		KNL	
	Time (s)	Percentage	Time (s)	Percentage
Load Index	4.71	3.89	28.74	1.60
Load Query	0.43	0.36	3.58	0.20
Seed & Chain	35.79	29.56	266.90	14.90
Align	79.22	65.42	1481.59	82.69
Output	0.93	0.77	9.85	0.61

We profiled the original minimap2 aligner on both CPU and KNL by aligning a PacBio simulated dataset to the human reference genome with one thread. The performance breakdown in Table 2 shows that the most time consuming step of minimap2 is base-level alignment, which takes 65.42% and 82.69% of the total time on CPU and KNL, respectively. It suggests that our focus should be on the acceleration of the base-level alignment algorithm.

Also, we observe that the single thread performance of KNL is very poor, due to KNL's low processor frequency. The overall running time on KNL is nearly 15 times slower than the CPU version. Therefore, it is of great importance to improve parallelism and reduce serial execution on manycore processors.

4.2 Overview

We illustrate the workflow of manymap in Figure 1. Firstly, in order to accelerate the most time-consuming base-level alignment step, we design a revised DP formula to resolve a data dependency issue. In addition, we achieve full vectorization by using SIMD and SIMT instruction level parallelism on KNL and GPU, respectively.

On the KNL processor, we take advantage of its hardware features. We utilize KNL's MCDRAM as addressable memory to achieve fast data transfer. We notice that the performance of minimap2 on KNL is dragged down by the I/O intensive components, such as loading index and writing results. Therefore, we adopt MCDRAM as addressable memory and memory-mapped I/O to reduce the data transfer cost. To better utilize the massively large number of threads, we adopt an optimized thread affinity scheduling strategy

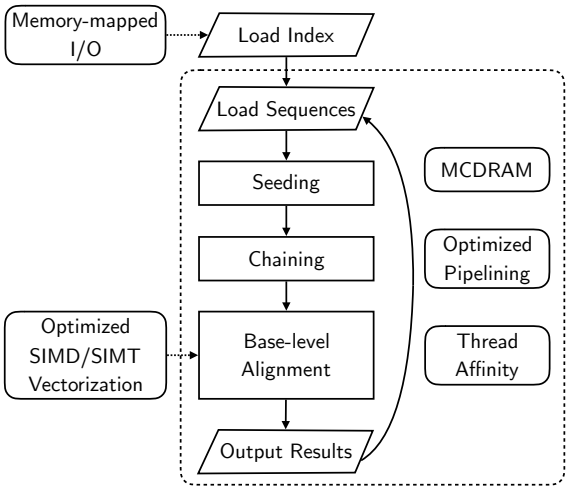


Figure 1: Workflow of manymap

to avoid thread contention. We also optimize the pipelining workflow, so that loading query sequences and outputting results fully overlaps computation.

On the GPU, we apply a multi-tier parallel model to increase parallelism and exploit the potential of manycore architecture. With the enhanced concurrent kernel execution capability of Turing GPUs, we are able to align up to 128 sequences simultaneously, while each kernel uses up to 512 threads. Additionally, we optimize memory performance through dynamically dispatch data to shared memory and global memory, and implement a memory pool to reduce data transfer cost.

4.3 Base-level Alignment Optimizations

4.3.1 Eliminate Intra-loop Dependency. The dynamic programming formula for base-level global alignment of minimap2 is shown in Equation (3). Minimap2 reorganized cells in the original DP matrix, so that cells on the same anti-diagonal are moved to the same column. This way, calculations within each column are independent. Moreover, cell values can be easily loaded in SIMD vectors since they are stored consecutively.

However, we observe that minimap2's vectorization approach has a dependency issue that limits its performance. Like most base-level alignment algorithms, minimap2 uses linear arrays to store DP matrices, i.e., only one column of cells are stored and iteratively updated. Though this approach can reduce memory usage, intra-loop data dependency is introduced, which results in extra computations.

We take the matrix x in Equation (3) as an example. To calculate values of cell (r, t), we need to read $x_{r-1,t-1}$ and write $x_{r,t}$. Suppose we store x in a linear array X, then we need to read $X[t-1]$ and write $X[t]$ at iteration r. However, we must read $X[t-1]$ before it was updated, or we will get $x_{r,t-1}$ rather than $x_{r-1,t-1}$.

One feasible solution to resolve the dependency issue is to use two arrays and swap them in each iteration, but it will double the space usage. Minimap2 resolves the dependency issue by using a temporary variable to keep the old value of $X[t]$. Nevertheless, it reduces the efficiency of vectorization. Our motivation is to completely remove the dependency.

We propose a new memory layout of DP matrices that can eliminate the data dependency between cells in the same column, without any extra cost. The formal definition of our algorithm is shown as follows. Recall that the alignment is performed between target sequence T and query sequence Q. We apply coordinate transformation $r \leftarrow r$, $t' \leftarrow t - r + |Q|$ to matrix v and x in Equation (3). This way, we can derive Equation (4). After the transformation, calculations on cell (r, t') depend on $v_{r-1,t'}$ and $x_{r-1,t'}$, which are in the same row t'. That is, we just need to read $X[t']$ first, then write $X[t']$ in place. Therefore, the intra-loop data dependency is eliminated.

$$\begin{cases} z_{rt} &= \max\{s(T_t, Q_{r-t}), x_{r-1,t'} + v_{r-1,t'}, y_{r-1,t} + u_{r-1,t}\} \\ u_{rt} &= z_{rt} - v_{r-1,t'} \\ v_{rt'} &= z_{rt} - u_{r-1,t} \\ x_{rt'} &= \max\{0, x_{r-1,t'} + v_{r-1,t'} - z_{rt} + q\} - q - e \\ y_{rt} &= \max\{0, y_{r-1,t} + u_{r-1,t} - z_{rt} + q\} - q - e \end{cases} \tag{4}$$

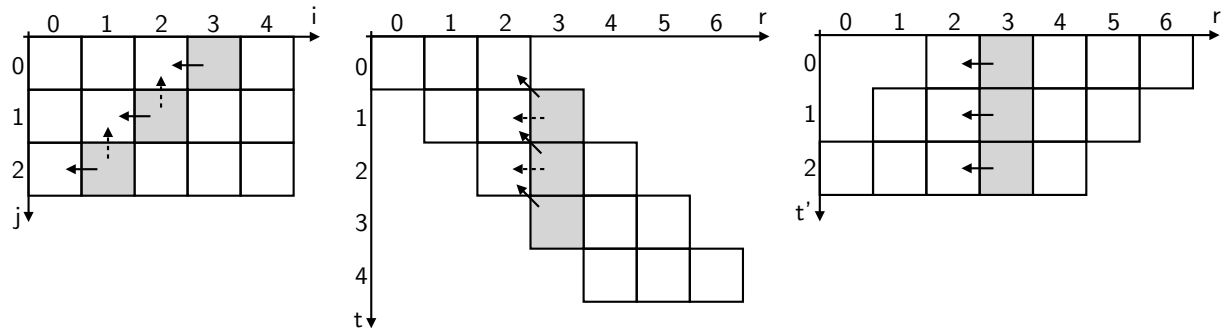
Algorithm 1 shows the implementation details of our DP formula using linear space arrays. To better understand our algorithm, we illustrate three memory layouts in Figure 2. The original DP matrix of global alignment using difference-based formula Equation (2) is shown in Figure 2a. The cells on the same anti-diagonal have no dependency between each other, since each cell only depends on the cell to its left ($v_{i-1,j}$ and $x_{i-1,j}$) or top ($u_{i,j-1}$ and $y_{i,j-1}$). Minimap2 changed the DP matrix to Figure 2b. The anti-diagonal marked grey in Figure 2a is placed in the same column. Now the dependency lies on the cell to its left ($u_{r-1,t}$ and $y_{r-1,t}$) or left-top ($v_{r-1,t-1}$ and $x_{r-1,t-1}$). In our algorithm, matrix u and y still use the layout in Figure 2b, while matrix v and x use memory layout in Figure 2c. This way, all the dependencies are on the left cells.

The space complexity of our new layout is $O(|Q|)$, while minimap2 uses $O(|T|)$ space. For semi-global alignments, the length of the query sequence $|Q|$ usually equals to or less than the length of the target sequence $|T|$. Therefore, the memory requirement of our algorithm remains the same with minimap2.

4.3.2 Vectorization. Vectorization is a common technique used in parallel processors. Vector processing can generally achieve considerable performance boost, because vector operations are applied to multiple data items simultaneously. In manymap, we manually vectorize the base-level alignment on CPU and KNL with SIMD instructions, and on GPU in a SIMT manner.

The original minimap2 implements the base-level alignment with SSE SIMD instructions on the CPU. The vector width is 128 bit, so it can update 16 8-bit integers in parallel. Due to the data dependency, the vectorization of minimap2 is not efficient, because it uses additional vector shift instructions to maintain temporary variables, as shown in Figure 3a.

With our dependency-free DP formula, the vectorization of base-level alignment can be greatly simplified. The complex vector load procedure can be reduced to a single load instruction, as shown in Figure 3b.



(a) original (Equation (2)) **(b) minimap2 (Equation (3))** **(c) manymap (Equation (4))**

Figure 2: Memory layouts of DP matrices. Solid arrows and dashed arrows denote dependencies of v, x and u, y, respectively.

Algorithm 1: Base-level alignment using revised DP formula

Input : target sequence T, query sequence Q, scoring matrix s, gap open cost q, gap extension cost e
Output: DP matrices U, V, X, Y

```
for r ← 0 to |T| + |Q| − 2 do
    st ← max{0, r − |Q| + 1};
    en ← min{|T| − 1, r};
    for t ← st to en do
        t' ← t − r + |Q|;
        z ← s(T[t], Q[r − t]);
        vt ← V[t'] ;                          ▷ get v_{r−1,t'}
        ut ← U[t] ;                           ▷ get u_{r−1,t}
        a ← X[t'] + vt ;                      ▷ a = x_{r−1,t'} + v_{r−1,t'}
        b ← Y[t] + ut ;                       ▷ b = y_{r−1,t} + u_{r−1,t}
        z ← max{z, a, b};
        U[t] ← z − vt ;                       ▷ u_{rt} = z_{rt} − v_{r−1,t'}
        V[t'] ← z − ut ;                      ▷ v_{rt'} = z_{rt} − u_{r−1,t}
        X[t'] ← max{0, a − z + q} − q − e;
        Y[t] ← max{0, b − z + q} − q − e;
    end
end
```

Furthermore, we extend our algorithm to AVX2 and AVX-512 SIMD instruction sets. On the CPU, we use AVX-512BW instructions, which can calculate 64 cells simultaneously. As for the KNL processor, though its vector width is also 512 bit, it can only operate 32-bit integers. To achieve high parallelism, we use AVX2 instructions instead, which can update 32 8-bit integers.

On the GPU, we vectorize base-level alignment using the SIMT model. We use a thread block as a vector, and each of the threads in the thread block calculates one data item. Therefore, we can update 512 cells in parallel when we use 512 threads in a thread block. We extend minimap2's base-level alignment algorithm to GPU, as shown in Figure 4a. Its performance is limited by branch divergence and thread synchronization cost. Our manymap on GPU can execute much more efficiently, since there is no branch divergence, as shown in Figure 4b.

4.4 Optimizations on Knights Landing

4.4.1 High Bandwidth MCDRAM. There are three memory modes available on KNL. They differ by the strategy of utilizing MCDRAM. In the cache mode, MCDRAM is completely assigned as cache between the processor and the DDR memory. In the flat mode, MCDRAM is used as program allocatable memory to increase the size of total available memory. In the hybrid mode, MCDRAM is partitioned as cache and memory under given portions [7].

We use the flat mode in manymap, which is more flexible than other two modes. We can control the preferred memory type between MCDRAM and DDR RAM using the `numactl` utility. We can determine whether to use the MCDRAM by checking the size of index. If the index can fit in the MCDRAM, we choose MCDRAM as the preferred memory type; otherwise, we use DDR RAM only. Since the capacity of MCDRAM is 16 GB, it can accommodate the index of human reference genome.

4.4.2 Memory-mapped I/O. Our profiling result shows that the I/O performance of the Knights Landing processor is very limited in single thread I/O operations, because the single core performance is low. The theoretical peak bandwidth can only be achieved under multi-thread I/O scenarios, which is not suitable for our application. The comparison between the time breakdown of minimap2 on CPU and KNL shows that the index loading time is nearly three times longer on KNL, which severely drags down the overall performance.

Since the index file structure is complex, the loading of index files requires dynamically allocating memory while parsing the list length of each entry in the index. The index loading procedure is thus unsuitable to parallelize. In order to improve the I/O performance, we use memory-mapped I/O to accelerate the loading of the index file and sequence files.

Memory mapping is a function supported by modern operating systems. It works by mapping files on the disk to the virtual address space in the memory, so that the program can read from the file as if it is accessing an array. It suits well for the task of reading large files, since it not only provides superior speed, but also supports on demand loading, which requires little memory space.

```
xt  = _mm_load_si128(&X[t]);  // xt  = x[r-1][t..t+15]
x1  = _mm_srli_si128(xt, 15);
xt  = _mm_slli_si128(xt, 1);
xt  = _mm_or_si128(xt, tmp);  // xt  = x[r-1][t-1..t+14]
tmp = x1;                     // tmp = x[r-1][t+15]
```

(a) minimap2

```
xt = _mm_loadu_si128(&X[t - r + qlen]);
```

(b) manymap

Figure 3: Comparison of SIMD implementations to load $X[t]$

We implement the most memory intensive component of manymap using memory-mapped I/O. At the beginning of the loading step, the index file is mapped to the memory, and we get a pointer to the memory address of the file. Since the original loading procedure is highly fragmented, its overhead is high. Memory-mapped I/O resolves the issue by performing consecutive file reads. With memory-mapped I/O, the index loading step of manymap is two times faster than that of minimap2 on KNL.

4.4.3 Thread Affinity. Next, we introduce the optimized thread affinity setting used in manymap. The KNL processor supports four hyper-threads per core, but the shared resources of each core are very limited. Therefore, it is very important to properly arrange the threads, or the performance might be affected by resource contention.

Let P denote the number of cores in the processor. The IDs of cores are $0, 1, \ldots, P-1$, and each core supports k threads. Given T threads with ID $0, 1, \ldots, T-1$, thread affinity determines the assignment of threads to cores. Note that T should not exceed the number of all the available threads, i.e., $T \leq kP$. For the KNL processor, we have $P = 64$, $k = 4$, and $T \leq 256$.

We implement three types of thread affinity using pthreads. The first two strategies are the same as OpenMP's affinity definitions, and the third one is optimized for KNL, as shown below.

(1) *compact*: The threads are assigned close to each other, i.e., thread i is assigned to core $\lfloor i/k \rfloor$. The *compact* affinity uses the least number of cores.
(2) *scatter*: The threads are assigned evenly to each core, i.e., thread i is assigned to core $i\%P$. It tends to use all the available cores.
(3) *optimized*: We propose an *optimized* thread affinity for KNL. It utilizes as many cores as possible, like *scatter* does, but always reserve one core for I/O tasks. This optimization avoids resource contention and has better I/O performance than *scatter*.

4.4.4 Pipeline Optimization. Minimap2 uses a pipeline to overlap parallel alignment and serial I/O operations. At the beginning, the query sequence file is separated into multiple batches. Then, two threads each executes a three-step procedure to align each batch. In the first step, one batch of reads is loaded into memory. In the second step, the multi-threaded read alignment (seeding, chaining and the base-level alignment) is performed. In the third step, the alignment result is formatted and output. The two threads in the pipeline alternatively process each batch, so that the alignment step of one thread can overlap the input step and the output step of the other thread, and the cost of I/O is thus hidden.

```
if (tid == 0) {
    xt = tmp;
    tmp = X[t + blockDim.x * gridDim.x - 1];
} else {
    xt = X[t - 1];
}
__syncthreads();
```

(a) minimap2

```
xt = X[t - r + qlen];
```

(b) manymap

Figure 4: Comparison of GPU implementations to load $X[t]$

While minimap2's pipeline works well on CPU, it is not suitable for KNL according to our empirical evidence. We observe that the I/O cost significantly increases on KNL due to low single thread performance. As a result, the computation step cannot fully hide the input and output step. To resolve this issue, we adopt two strategies. One is by reducing the I/O cost through memory access optimizations, as we have mentioned in Sections 4.4.1 and 4.4.2. The other is to redesign the pipeline by introducing an additional thread for I/O. This way, the input step and output step can also overlap.

In addition, we find that some long reads may take longer time than other reads to align, which may result in load imbalance. Therefore, for each read batch, we sort them by length to make sure that long reads are processed first, so that the waiting time between threads is reduced.

4.5 Optimizations on GPU

4.5.1 Parallel Model. The key to manycore processing on GPU is how to properly divide tasks to realize parallelism. To fully exploit the potential of GPU, the application needs to run with high parallelism. We deploy a two-level parallel model on GPU.

Firstly, multiple kernels are launched concurrently to perform base-level alignment. Each kernel corresponds to the alignment of one pair of sequences. The host CPU feeds alignment tasks to GPU asynchronously, so that the computation and data transfer can be overlapped. On GPU devices with compute capability 7.0 or higher, at most 128 kernels can be executed concurrently.

Secondly, we use multiple threads to calculate the DP matrices in a SIMT manner. Using our revised dynamic programming formula, warp divergences are eliminated, so that no costly thread synchronization within the thread block is required. Furthermore, we utilize loop unrolling to remove branch control instructions of the inner DP loop.

Instead of organizing threads into multiple thread blocks, we use one thread block with up to 512 threads for the alignment of one pair of sequences. The major motivation of this design is about concurrency. To use multiple thread blocks, we have to handle the synchronization between thread blocks. Two feasible solutions are to split kernel into multiple small kernels for implicit synchronization, or use cooperative groups to sync all the threads in a grid.

However, neither approach can achieve the concurrency as the single kernel version does.

4.5.2 Memory Access Optimizations. The memory requirement for each kernel depends on the length of sequences and whether we need the alignment score or path. If only the alignment score is needed, then the space complexity is linear to sequence length $O(|T| + |Q|)$ by using linear arrays to store DP matrices. Otherwise, if the complete alignment path is required, then we have to use a $O(|T| \cdot |Q|)$ matrix for backtracking. Since the sequence lengths are known before kernel launches, we can calculate the required memory size and dynamically assign shared memory if space allows. When the capacity of shared memory is exceeded when sequence is long, then let DP matrices reside in global memory. The memory access in the inner loop of DP is coalesced to increase global memory throughput.

We observe that the data exchange pattern between the CPU and GPU is that small batches of data are frequently transferred. To reduce the cost of memory allocation, we implement a memory pool that can be reused by kernels. Each CUDA stream can visit its own proportion in the memory pool by ID. Furthermore, we allocate pinned memory on the host side to achieve the highest bandwidth.

We note that the processing capability of GPU is bounded by the memory size. For long sequences, the memory usage is significantly high. For example, suppose we align two sequences of 32 thousands bp (base pairs) each, then 2 GB memory is required to calculate the alignment path. Insufficient memory space might result in reduced concurrency, so we fall back to CPU alignment in such cases. Nevertheless, in actual alignment, such demanding sequences rarely occur.

5 PERFORMANCE EVALUATION

In this section, we evaluate the performance of manymap. Firstly, we perform micro benchmarks on the base-level alignment step. We analyze how our revised DP formula improves the performance, as well as the impact of SIMD vector width and memory modes. Then, we perform macro benchmarks, which measure the overall performance of our aligner on simulated and real datasets. We show the scalability of manymap on KNL with respect to the number of threads, discuss the impact of thread affinity, and compare manymap with a set of long read aligners.

5.1 Experimental Setup

5.1.1 Hardware Configurations. Our experiments are conducted on two servers, namely gpu1 and knl1. Their hardware configurations are shown in Table 3. The gpu1 server is responsible for experiments of CPU and GPU, while experiments on KNL are run on knl1.

5.1.2 Micro Benchmarks. In micro benchmarks, we compare the base-level alignment algorithms of minimap2 and manymap on three processors, namely CPU, GPU, and KNL. For CPU and KNL, we run with all the hardware threads available, i.e., 40 threads on CPU and 256 threads on KNL. For GPU, we use 1 block with 512 threads per kernel, and dispatch tasks to 128 CUDA streams, i.e., up to 65,536 threads are launched concurrently. Since the original minimap2 aligner only has SSE implementation, we extend it to AVX2 and AVX-512, as well as GPU, to compare with manymap.

Table 3: Hardware configurations

Server	gpu1		knl1
Processor	CPU	GPU	Xeon Phi
Model	Xeon Gold 5115	Tesla V100	Xeon Phi 7210
# Cores	20	5120	64
Base Frequency (MHz)	2400	1245	1300
Max Frequency (MHz)	3200	1380	1500
Device Memory Type	-	HBM2	MCDRAM
Device Memory Size (GB)	-	16	16
Host Memory Size (GB)	256		96
Compiler	gcc 6.3	nvcc 10.0	gcc 6.3
OS	CentOS 7, Linux kernel 3.10		

The workload of micro benchmarks are sequences dumped from the function calls of minimap2 in the alignment process of PacBio SMRT reads to the human reference genome. We prepare 6 workloads of lengths from 1 thousand to 32 thousand bp. In addition, depending on the return value, we perform two types of alignment in micro benchmarks. The score-only alignment uses linear space and only returns the alignment score. The alignment with complete path requires quadratic space for backtracking.

The performance metric for micro benchmarks is GCUPS (giga cell update per second). GCUPS is calculated by $(|T| \cdot |Q|)/(t \cdot 10^9)$, where t is the runtime of aligning two sequences in length $|T|$ and $|Q|$. In our experiments, we set $|T| = |Q|$, and use the average per-pair alignment time as t.

5.1.3 Macro Benchmarks. We use both a simulated dataset and a real dataset for the performance evaluation, as shown in Table 4. The simulated dataset is generated from the human reference genome hg38 using PBSIM [19]. We sample the error profile from H. sapiens 50x Sequence Coverage data produced by PacBio SMRT sequencing, so that the simulated reads has exactly the same characteristics as the real reads. In our experiment, we align the simulated dataset to the hg38 reference genome with option `-ax map-pb`, which uses parameters tuned for PacBio reads.

Table 4: Datasets for Macro Benchmarks

Dataset	Simulated	Real
Platform	PacBio SMRT	Nanopore
Number of Reads	895,439	356,209
Average Length (bp)	5,567	3,957.8
Maximum Length (bp)	24,981	514,461
Total Bases	4,985,012,420	1,409,812,422
Read File Size (GB)	9.4	2.7
Index File Size (GB)	5.2	6.8

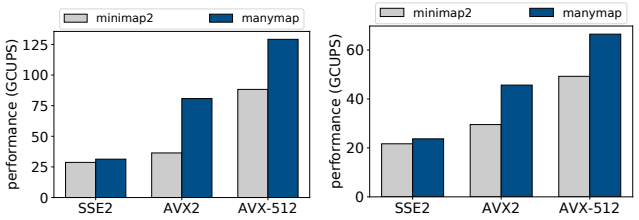
The real dataset is selected from the Oxford Nanopore human sequencing dataset [6], with flowcell id FAB23716. We observe the Nanopore dataset has different characteristics from PacBio's. The average length is less than PacBio, while the maximum length is much longer. We also align the real dataset to the hg38 reference genome, with parameter `-ax map-ont`.

In addition, we reproduce the experiment in the minimap2 paper for comparison with other aligners in terms of accuracy. We use

a simulated PacBio dataset with 33,088 reads, which is the same dataset as minimap2's experiment used.

5.2 Micro Benchmarks

5.2.1 SIMD instruction sets on CPU. We first compare the base-level alignment algorithm of minimap2 and manymap using three SIMD instruction sets on the CPU, as shown in Figure 5. We disable optimizations of manymap for the fairness of comparison.

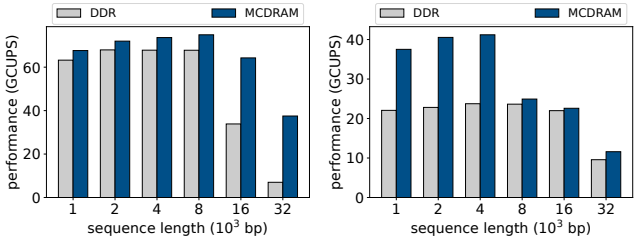


(a) Alignment with score only **(b) Alignment with complete path**

Figure 5: Comparison of SIMD instruction sets

Using the SSE2 instruction set, manymap is about 10 percent faster than minimap2 on both types of alignment. As for the AVX2 instruction set, manymap is 2.2 times and 1.6 times faster than minimap2, respectively. On the AVX-512 instruction set, manymap is 1.5 times faster. The largest performance gain is achieved on AVX2, because AVX2 uses more instructions to shift vectors than other two instruction sets.

5.2.2 Memory Modes on KNL. The choice of memory mode of KNL has considerable effect on the performance. We compare the performance of manymap on KNL between DDR RAM and MCDRAM, as shown in Figure 6.



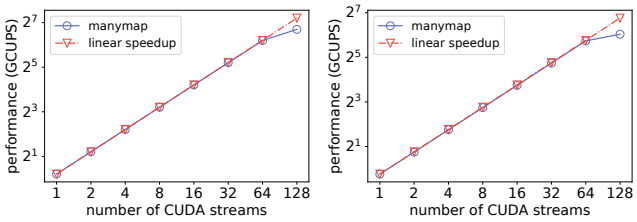
(a) Alignment with score only **(b) Alignment with complete path**

Figure 6: Comparison between memory modes

For the alignment with score only, the space complexity is low, so MCDRAM has no significant advantage over DDR RAM when the sequences are short. When the sequence length is longer than 16 thousand bp, the memory usage is more than 0.1 GB, and using MCDRAM brings up to 5 times speedup.

When we perform alignment with backtracking, the memory usage climbs up rapidly with respect to sequence length. Using MCDRAM is around 1.8 times faster when the memory usage is less than 16 GB, i.e., can fit in MCDRAM. When the memory usage exceeds the capacity of MCDRAM, e.g., aligning sequences in 8 thousand bp that requires 18 GB memory, the performance of MCDRAM and DDR RAM are comparable.

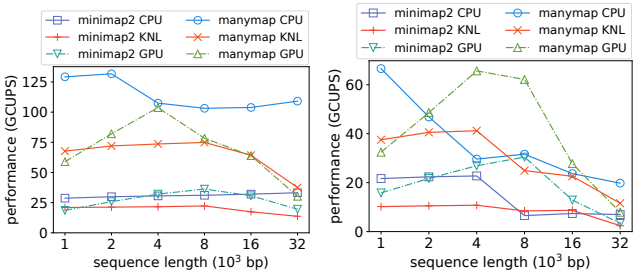
5.2.3 Concurrency on GPU. We evaluate the performance of varied number of CUDA streams on GPU. We use the 4 thousand bp workload, and the result is shown in Figure 7. The performance increases with a linear speedup from 1 to 64 streams on both types of alignment. With 128 streams, the maximum number of resident grids is reached, and the performance slightly increases. The overall speedup are 90 and 77.4, respectively. The result shows that concurrent stream execution is highly efficient.



(a) Alignment with score only **(b) Alignment with complete path**

Figure 7: Performance of varied number of CUDA streams

5.2.4 Comparison of processors. We evaluate the base-level alignment performance of minimap2 and manymap on three processors. Figure 8a shows the alignment performance when only the final score is returned. By using the revised DP formula, the CPU version of manymap is 3.3 to 4.5 times faster than minimap2 on CPU on all lengths. On KNL, manymap reports up to 3.4 times speedup over the direct port of minimap2 when the sequence length is 8 thousand. On sequences longer than that, the performance of KNL decreases with the length increase due to limited per-thread resource. The performance of manymap on the GPU fluctuates. The peak performance on GPU is achieved when the sequence length is 4 thousand, which is 3.2 times faster over minimap2. For longer sequences, the score matrix cannot fit in GPU's shared memory, so the performance is dragged down by global memory access.



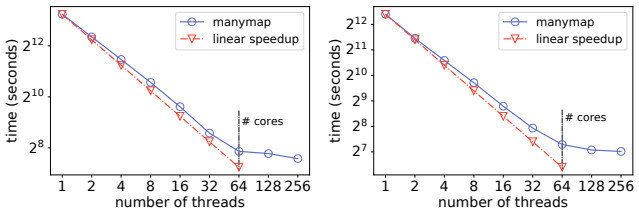
(a) Alignment with score only **(b) Alignment with complete path**

Figure 8: Performance of varied sequence length

Figure 8b shows the performance of alignment that returns the complete alignment path by backtracking. On CPU, manymap is 1.3 to 4.5 times faster than minimap2. Manymap is 3.9 times faster than minimap2 on KNL when the sequence length is 4 thousand. The performance on KNL decreases when the data cannot fit in the MCDRAM. As for GPU, we observe a similar curve as in Figure 8a. It achieves the best performance among all processors when the sequence length is between 2 to 16 thousand. However, the performance sharply decreases due to reduced concurrency. For example, aligning sequences in 32 thousand bp requires 2 GB memory, so only 8 kernels can run concurrently as the global memory size is limited.

5.3 Macro Benchmarks

5.3.1 Scalability on KNL. We evaluate the scalability of manymap with the number of threads varied on KNL. As shown in Figure 9, we align both datasets using threads varied from 1 to 256. Manymap scales well on physical cores of the Knights Landing processor. On the simulated dataset, the speedup is 50.55 with 64 threads, which achieves 79% parallel efficiency (speedup divided by number of threads).



(a) Simulated Dataset **(b) Real Dataset**

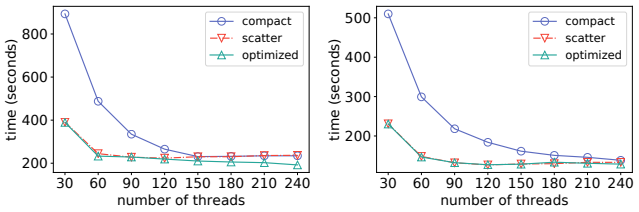
Figure 9: The runtime of manymap on KNL, compared with the linear speedup, shown in log-log scale

When the number of threads is more than the number of physical cores, the performance increase slows down. On the real dataset, manymap was only 21% faster using four threads per core than one thread per core. This is because the resource shared by hyper threads is very limited. On KNL, two physical cores in the same tile share 1MB L2 cache. As a consequence, cache misses will increase when hyper threading is enabled, because these threads have to compete for the limited resource.

5.3.2 Thread Affinity on KNL. We evaluate the performance of manymap using different thread affinity strategies, namely, *compact*, *scatter* and *optimized*. The result is shown in Figure 10.

When the number of threads is less than or equal to the number of cores, the *scatter* and *optimized* affinity show comparable performance, because they result in the same thread assignment. The *compact* affinity is nearly two times slower, because threads are assigned to the least number of physical cores, thus not all cores are utilized.

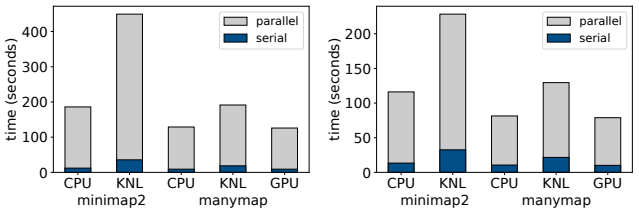
As the number of threads increases, the performance of the *compact* affinity gradually approaches *scatter*, since the core utilization rate is increased. On the simulated dataset, our *optimized* affinity outperforms *compact* and *scatter* by up to 22% when the number of threads is 150 or more, because it reserves one core for I/O tasks, so that the cost of I/O can be reduced and further improves pipelining. The real dataset is less affected by the thread affinity setting, because it involved less overlapped I/O and computations.



(a) Simulated Dataset **(b) Real Dataset**

Figure 10: Comparison of thread affinity strategies

5.3.3 Comparison with other aligners. The performance breakdown of minimap2 and manymap is shown in Figure 11. Overall, manymap achieves 1.4 and 2.3 times speedup on CPU and KNL, respectively. The serial execution time on both CPU and KNL has reduced, thanks to memory-mapped I/O. The GPU version only outperforms the CPU version of manymap by a small margin, because the maximum occupancy is not achieved.



(a) Simulated Dataset **(b) Real Dataset**

Figure 11: Performance breakdown of minimap2 and manymap

We also compare manymap with the minimap2 aligner and other long read aligners, including minialign [23], Kart [12], BLASR [3], NGMLR [22], and BWA-MEM [8]. Their runtime on KNL are measured by directly porting without modifications. We use the maximum number of hardware threads available on CPU and KNL, except minialign, Kart, and BWA-MEM, which use 64 threads on KNL, because they either do not support 256 threads or failed to execute.

Table 5 shows the performance, space usage, and accuracy of various aligners on the simulated PacBio dataset. The accuracy of alignment refers to the error rate of alignment, i.e., the number of wrong alignments divided by the total number of aligned reads. Manymap produces the same alignment result as minimap2, thus has the same accuracy.

We observe that minialign and Kart reports the shortest runtime on CPU and KNL, respectively. However, they both have a significant higher error rate than manymap. BLASR and NGLMR can produce accurate alignments, but the time cost is higher than

Table 5: Comparison of long read aligners

		manymap	minimap2	minialign	Kart	BLASR	NGMLR	BWA-MEM
	Error Rate (%)	0.378	0.378	0.973	4.1	0.559	0.808	1.158
	Index Size (MB)	5362	5362	4292	5174	11826	5076	5174
CPU	Time (seconds)	18.005	26.779	13.95	32.87	44.79	572.82	874.97
	RAM Usage (MB)	11263	8292	13021	11138	14837	8714	5913
KNL	Time (seconds)	36.934	75.32	64.225	34.65	490.88	1134.44	1213.14
	RAM Usage (MB)	11280	11416	23182	11288	14862	30966	8522
GPU	Time (seconds)	16.772	-	-	-	-	-	-

others. BWA-MEM reports the worst accuracy and longest runtime, since it is originally designed for short read alignment.

6 CONCLUSION

The emerging third-generation sequencing technologies are producing large-scale biological data in an unprecedented speed. In consequence, the need of high performance bioinformatics algorithms is in an increasing urgency.

In this paper, we accelerate long read alignment on the CPU, GPU and KNL processors. Specifically, we vectorize the base-level alignment step using SIMD and SIMT instructions. We apply various memory access optimizations specialized to each architecture. The evaluation result shows that our manymap scales well to the number of cores, and outperforms both the CPU and KNL version of the original minimap2 aligner.

Our experimental study shows that vector width, memory type, and thread affinity have substantial impact on performance. We find a high-end server CPU is still the most efficient platform for long read alignment tasks. Though KNL and GPU both outperform CPU on the base-level alignment step, the low single thread performance and occupancy issue limits their overall performance.

We summarize the following recommendations of accelerating applications on manycore processors. Firstly, in order to achieve the peak performance, the computation should be highly parallelized and vectorized. Secondly, due to the weak single core performance, serial operations could drag down the overall performance, even if they are fast on the CPU. Thirdly, depending on the memory requirement of the program, we should carefully arrange the memory to maximize throughput and avoid performance loss.

ACKNOWLEDGMENTS

This work was partly supported by grants 16206414 from the Hong Kong Research Grants Council and MRA11EG01 from Microsoft.

REFERENCES

[1] Stephen F Altschul and Bruce W Erickson. 1986. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology* 48, 5-6 (1986), 603–616.
[2] M. Burrows and D. J. Wheeler. 1994. *A block-sorting lossless data compression algorithm.* Technical Report.
[3] Mark J Chaisson and Glenn Tesler. 2012. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics* 13, 1 (2012), 238.
[4] Michael Farrar. 2007. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* 23, 2 (2007), 156–161.
[5] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581.
[6] Miten Jain, Sergey Koren, Karen H Miga, Josh Quick, Arthur C Rand, Thomas A Sasani, John R Tyson, Andrew D Beggs, Alexander T Dilthey, Ian T Fiddes, Sunir Malla, Hannah Marriott, Tom Nieto, Justin O'Grady, Hugh E Olsen, Brent S Pedersen, Arang Rhie, Hollian Richardson, Aaron R Quinlan, Terrance P Snutch, Louise Tee, Benedict Paten, Adam M Phillippy, Jared T Simpson, Nicholas J Loman, and Matthew Loose. 2018. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology* 36, 4 (2018), 338–345.
[7] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming Knights Landing Edition.* Morgan Kaufmann Publishers Inc., Boston, MA, USA.
[8] Heng Li. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. (2013). arXiv:1303.3997v2
[9] Heng Li. 2018. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (2018), 3094–3100.
[10] Heng Li and Richard Durbin. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* 25, 14 (2009), 1754–1760.
[11] Heng Li and Nils Homer. 2010. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics* 11, 5 (2010), 473–483.
[12] Hsin-Nan Lin and Wen-Lian Hsu. 2017. Kart: A divide-and-conquer algorithm for NGS read alignment. *Bioinformatics* 33, 15 (2017), 2281–2287.
[13] Yongchao Liu and Bertil Schmidt. 2014. CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing. *IEEE Design & Test* 31, 1 (2014), 31–39.
[14] Yongchao Liu and Bertil Schmidt. 2014. SWAPHI: Smith-waterman protein database search on Xeon Phi coprocessors. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 184–185.
[15] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. 2013. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics* 14, 1 (2013), 117.
[16] Ruibang Luo, Jeanno Cheung, Edward Wu, Heng Wang, Sze-Hang Chan, Wai-Chun Law, Guangzhu He, Chang Yu, Chi-Man Liu, Dazong Zhou, Yingrui Li, Ruiqiang Li, Jun Wang, Xiaoqian Zhu, Shaoliang Peng, and Tak-Wah Lam. 2015. MICA: A fast short-read aligner that takes full advantage of Many Integrated Core Architecture (MIC). *BMC Bioinformatics* 16, S7 (2015), S10.
[17] Santiago Marco-Sola, Michael Sammeth, Roderic Guigó, and Paolo Ribeca. 2012. The GEM mapper: fast, accurate and versatile alignment by filtration. *Nature Methods* 9, 12 (2012), 1185–1188.
[18] John D McPherson. 2009. Next-generation gap. *Nature Methods* 6, 11 (2009), S2–S5.
[19] Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. 2012. PBSIM: PacBio reads simulator - toward accurate genome assembly. *Bioinformatics* 29, 1 (2012), 119–121.
[20] Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. 2004. Reducing storage requirements for biological sequence comparison. *Bioinformatics* 20, 18 (2004), 3363–3369.
[21] Torbjørn Rognes. 2011. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics* 12, 1 (2011), 221.
[22] Fritz J Sedlazeck, Philipp Rescheneder, Moritz Smolka, Han Fang, Maria Nattestad, Arndt Haeseler, and Michael C Schatz. 2018. Accurate detection of complex structural variations using single-molecule sequencing. *Nature Methods* 15, 6 (2018), 461–468.
[23] Hajime Suzuki. 2018. minialign: fast and accurate alignment tool for PacBio and Nanopore long reads. https://github.com/ocxtal/minialign
[24] Hajime Suzuki and Masahiro Kasahara. 2018. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics* 19, 1 (2018), 45.
[25] Lipeng Wang, Yuandong Chan, Xiaohui Duan, Haidong Lan, Xiangxu Meng, and Weiguo Liu. 2014. XSW: Accelerating Biological Database Search on Xeon Phi. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 950–957.