GPU acceleration of Darwin read overlapper for de novo assembly of long DNA reads

Ahmed, Nauma; Qiu, Tong Dong; Bertels, Koen; Al-Ars, Zaid

DOI
10.1186/s12859-020-03685-1

Publication date
2020

Document Version
Final published version

Published in
BMC Bioinformatics

Ahmed *et al. BMC Bioinformatics* 2020, **21**(Suppl 13):388
https://doi.org/10.1186/s12859-020-03685-1

BMC Bioinformatics

SOFTWARE

Open Access

GPU acceleration of Darwin read overlapper for de novo assembly of long DNA reads

Nauman Ahmed[1,2*], Tong Dong Qiu[1], Koen Bertels[1] and Zaid Al-Ars[1]

From The 18th Asia Pacific Bioinformatics Conference
Seoul, Korea. 18-20 August 2020

*Correspondence:
n.ahmed@tudelft.nl
[1]Delft University of Technology, Delft, Netherlands
[2]University of Engineering and Technology Lahore, Lahore, Pakistan

Abstract

Background: In Overlap-Layout-Consensus (OLC) based de novo assembly, all reads must be compared with every other read to find overlaps. This makes the process rather slow and limits the practicality of using de novo assembly methods at a large scale in the field. Darwin is a fast and accurate read overlapper that can be used for de novo assembly of state-of-the-art third generation long DNA reads. Darwin is designed to be hardware-friendly and can be accelerated on specialized computer system hardware to achieve higher performance.

Results: This work accelerates Darwin on GPUs. Using real Pacbio data, our GPU implementation on Tesla K40 has shown a speedup of 109x vs 8 CPU threads of an Intel Xeon machine and 24x vs 64 threads of IBM Power8 machine. The GPU implementation supports both linear and affine gap, scoring model. The results show that the GPU implementation can achieve the same high speedup for different scoring schemes.

Conclusions: The GPU implementation proposed in this work shows significant improvement in performance compared to the CPU version, thereby making it accessible for utilization as a practical read overlapper in a DNA assembly pipeline. Furthermore, our GPU acceleration can also be used for performing fast Smith-Waterman alignment between long DNA reads. GPU hardware has become commonly available in the field today, making the proposed acceleration accessible to a larger public. The implementation is available at https://github.com/Tongdongq/darwin-gpu.

Keywords: Genomics, Read overlapper, De novo assembly, Long DNA reads, GPU acceleration

Background

DNA sequencing techniques used today produce short pieces of data (called reads) that represent parts of the sampled DNA, possibly containing some errors. The length and error rate of these reads depends on the sequencing technique used. DNA assembly tries to combine the reads into larger, more accurate DNA segments. For these DNA reads, graph-based assemblers are used for the assembly process, which comes in two flavors: Overlap-Layout-Consensus (OLC) and de Bruijn Graph (dBG).

The OLC assemblers [1] first find overlaps and build an overlap graph. Each node represents a read, and each edge represents an overlap between two reads. During the layout phase, the graph is analyzed to find paths, corresponding to segments of the original genome. The perfect graph contains one path that visits each node exactly once. This problem can be described as finding a Hamiltonian Path. A Hamiltonian Path includes all vertices of a graph exactly once. Examples of assemblers that use the OLC approach are Dazzler [2] and SGA [3].

In dBG based assemblers [4], each read is divided into *K-mers*. A *K-mer* is a substring of a read having a length *K*. Each *K-mer* represents a directed edge between two vertices, where the source vertex represents the first $K-1$ bases of the *K-mer*, and the destination vertex the last $K-1$ bases of the *K-mer*. When a particular $K-1$ vertex does not exist, it is created, otherwise, the existing one is reused. The weights of the edges indicate how many times a particular *K-mer* is encountered. The next step is to find an Eulerian Path, which is a path that includes all edges of a graph exactly once. Examples of dBG assemblers are Velvet [5], ABySS [6] and SOAPdenovo2 [7].

So called Next Generation Sequencing (NGS) techniques produce reads with lengths anywhere from 50 to 500 base pairs. They can be produced at high throughput but at the expense of a smaller read length. However, DNA can contain repeat regions, where a certain piece of DNA is repeated many times back-to-back, or a repeat could appear in many different places in the genome. Since these repeats can be longer than the produced short reads, this means the reads cannot be used to resolve these repeat regions. Third generation sequencing produces much longer reads, of up to 60K base pairs. Due to their length, these reads are more likely to contain a whole repeat region, which makes them suitable for accurately reconstructing the repeat regions. A major drawback of longer DNA reads is their higher error rate, ranging from 15-30%, depending on the exact sequencing technology. dBG based assembly is the more preferred approach for NGS reads, which are much shorter and have much lower error rates. However, de Bruijn Graphs are quite susceptible to sequencing errors, since one substituted base pair causes *K* incorrect *K-mers*. Pair this with an often-used values of *K* above 50 and third generation sequencing error rate of about 15%, it is clear that the graph will contain a lot of incorrect edges. Therefore, OLC based assemblers are more suitable for state-of-the-art third generation long DNA reads produced by Pacbio and Oxford Nanopore sequencers.

Darwin [8] is a read overlapper for the assembly of long DNA reads. Darwin is designed to be highly accurate, achieving a sensitivity of 99.89% and a precision of 88.30% for simulated Pacbio reads. This is higher than other commonly used read overlappers such as Daligner [2]. The ASIC (Application-Specific Integrated Circuit) implementation of Darwin is shown to be hundreds of times faster than other software based overlappers. However, ASIC implementation requires bulk volume production to be economically feasible. Moreover, DNA analysis using high-throughput DNA sequencing is

an evolving field, and any major improvement in the algorithm will require a new ASIC implementation which costs both time and money.

Heterogeneous systems with GPU accelerators have become easily accessible due to their widespread use. They have shown convincing speedups in many high performance computing applications. Therefore, GPU acceleration of various genome analysis algorithms has been the topic of many research works, like in [9, 10] and [11]. In this paper, we present a GPU accelerated version of Darwin. We identified the computational bottleneck in the Darwin software and replaced it with the GPU accelerated version. The accelerated implementation proposed in this paper is orders of magnitude faster than its software counterpart. The contributions of the paper are as follows:

- The paper shows the GPU implementation of the Darwin read overlapper used in the de novo assembly of long DNA reads.
- The paper shows that the GPU acceleration of Darwin is orders of magnitude faster than the multithreaded software version on both IBM Power8 and Intel Xeon machines using a real Pacbio dataset.
- The results in the paper show that the GPU implementation of Darwin can also be applied for accelerating Smith-Waterman alignment of long DNA reads.

Background

Smith-Waterman (SW) [12] algorithm finds local alignment between a pair of sequences. Smith-Waterman is exact, producing the optimal local alignment. It can be implemented using dynamic programming which computes a 2D matrix S. Let V and W be the two sequences to be aligned. Let $V_0, V1, \ldots, V_{|V|-1}$ and $W_0, W1, \ldots, W_{|W|-1}$ be the bases of V and W, respectively. $|V|$ and $|W|$ are the lengths of V and W. $S(i,-1) = S(-1,j) = 0$ for $i = 0, 1, 2 \ldots, |V|-1$ and $j = 0, 1, 2 \ldots, |W|-1$. The cells in the matrix are computed using the following recurrence relation:

$$S(i,j) = max \begin{cases} S(i-1,j) + gap \\ S(i,j-1) + gap \\ S(i-1,j-1) + subt\left(V_i, W_j\right) \\ 0 \end{cases} \tag{1}$$

$$m_{i,j} = \begin{cases} (i,j) & S(i,j) > m \\ m_{i,j} & S(i,j) \leq m \end{cases} \tag{2}$$

$$m = max \begin{cases} m \\ S(i,j) \end{cases} \tag{3}$$

$$D(i,j) = \begin{cases} 0 & S(i,j) = 0 \\ \uparrow & S(i,j) = S(i-1,j) \\ \leftarrow & S(i,j) = S(i,j-1) \\ \nwarrow & S(i,j) = S(i-1,j-1) + subt\left(V_i, W_j\right) \end{cases} \tag{4}$$

Here, S and D are the score and traceback matrices, respectively. *match*, *mismatch* and *gap* are numeric parameters. $subt\left(V_i, W_j\right)$ is equal to *match* if $V_i = W_j$, and is equal to *mismatch* otherwise. *gap* is the penalty for inserting a gap. m is the alignment score, which is initialized to zero, and $m_{i,j}$ is the corresponding position on V and W. The traceback matrix is required to compute the actual alignment. Traceback starts from the highest

scoring cell and follows the arrows in D until a zero or a boundary of the matrix is encountered. Equations 1 and 3 indicate that for computing the alignment score m there is no need to store the whole S matrix as all cells of the S matrix are computed using only the values of three other cells $S(i-1,j)$, $S(i,j-1)$ and $S(i-1,j-1)$. Hence, to compute the alignment score, storing only the values in the previous row and column are sufficient to compute m. The above equations are for calculating the alignment with a linear-gap scoring model. However, Darwin and our GPU implementation also support the more commonly used affine gap penalty model in which there are separate penalties for opening a gap (*gapo*) and extending a gap (*gape*).

A straightforward way of finding all overlaps is performing an alignment algorithm, such as Smith-Waterman, on every pair of reads. The number of alignments is quadratic with the number of reads, and the runtime of one alignment is quadratic with the lengths of the involved reads, making this method not feasible. Many heuristic algorithms have been developed to perform this alignment, for different lengths and error rates. Seed-and-extend is one heuristic, which dramatically reduces the amount of computation needed [13]. A seed is a *K-mer* made up of K consecutive bases of a read. Instead of performing Smith-Waterman on each read pair, only read pairs that have one or more common *K-mers* are aligned. A common *K-mer* between two or more reads is known as a "seed hit". Darwin also uses the seed-and-extend approach, which reduces the amount of computation needed, without compromising the output by much. Other algorithms, like BLAST [14], also use the seed-and-extend approach, but give sub-optimal alignments. Results in [8] show that Darwin provides optimal Smith-Waterman alignments between long DNA sequences with error rates up to 40%.

Darwin

Darwin is read overlapping algorithm for de novo assembly of third-generation long DNA reads. It is based on the seed-and-extend. It consists of a filter called D-SOFT (Diagonal-band Seed Overlapping based Filtration Technique), which finds seed hits, and GACT (Genome Alignment using Constant memory Traceback), which extends the seed hit by performing sequence alignment between the sequences on the left and right of the seed hit. Figure 1 shows the seed-and-extend technique employed in Darwin to find the overlap between *Read A* and *Read B*. To compute the overlap, a seed hit is extended on both sides by aligning *R_left* with *Q_left* and *R_right* with *Q_right*. This speedups the computation by avoiding the computation of a large number of dynamic programming matrix cells (grey cells in Fig. 1).The dynamic programming matrix computed to align *R_left* with *Q_left* is known as *left extension matrix*. Similarly, the dynamic programming matrix computed to align *R_right* with *Q_right* is known as *right extension matrix*.

D-SOFT

D-SOFT is the seeding stage of Darwin, also known as the *filtering* stage. Darwin uses minimizers [15] as seeds which are *K-mers* extracted from all the reads to be overlapped. The position of a seed is stored in a minimizer table which records the location of the seed in a read along with the identifier of the read. The window size w is the most important parameter for building a minimizer table and must be smaller than the seed length (K). To obtain seed hits, 'N' *K-mers* of a read are used as seeds that are located in other reads using the minimizer table. Two reads are considered for alignment in the extension phase

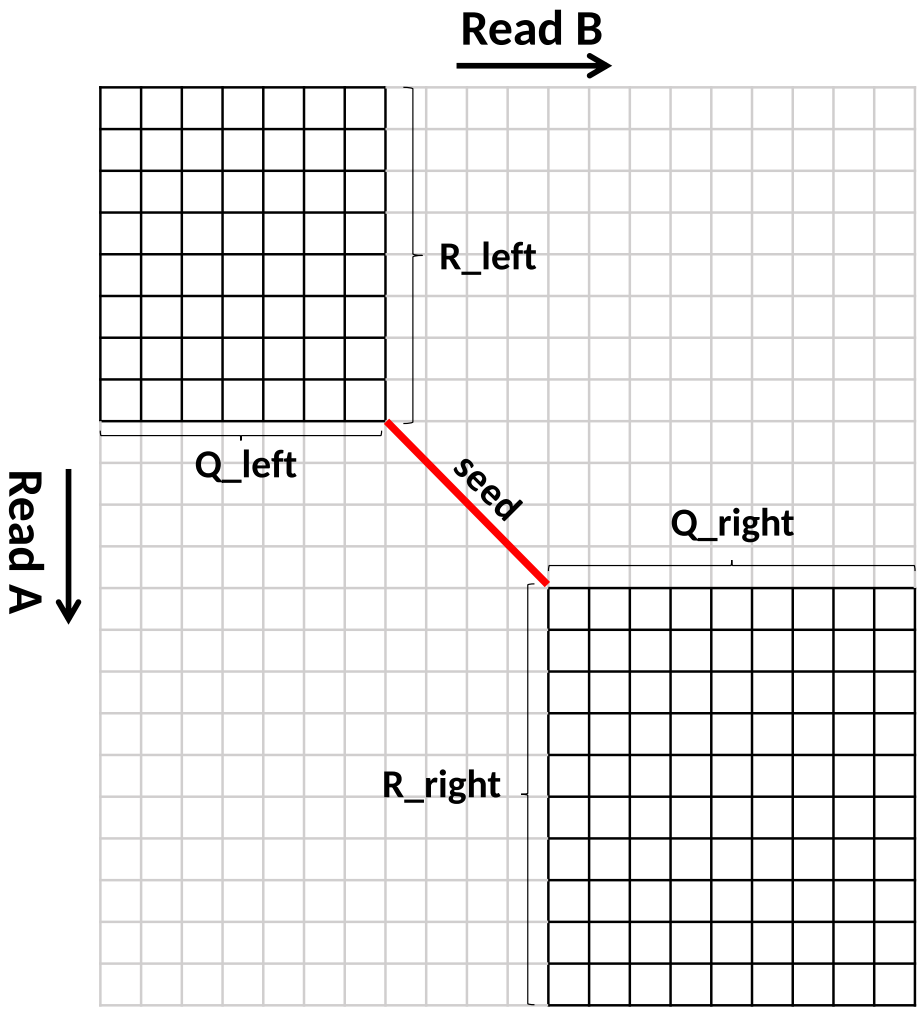

Fig. 1 The seed-and-extend method used in Darwin to find the overlap between *Read A* and *Read B*. *R_left* and *R_right* are the substrings of Read A. *Q_left* and *Q_right* are the substrings of Read B

if they have at least h unique bases in common. The pair of reads passing this filter are aligned using a modified Smith-Waterman algorithm described below.

GACT

The seed-and-extend approach to find overlap between two reads is much faster than performing the complete Smith-Waterman alignment algorithm. However, the reduced dynamic programming matrices could still be quite large. State-of-the-art third generation sequencers produce reads having lengths in megabases and the dynamic programming matrix in the seed extension may have around 1 Tera cells to compute. For example, if the seed hit lies at the beginning of the two reads, i.e. near the top left corner in Fig. 1, the right extension matrix is nearly as large as the full matrix.

Numerous efforts to accelerate Smith-Waterman have been made, both by using hardware like [16] and software [17]. But the memory required to store the traceback matrix D is still an issue. One can apply the Hirschberg's algorithm described in [18] to reduce

the RAM storage but at the cost of increase in computation time. Therefore, Darwin proposed the GACT algorithm for seed extension. It has two advantages: 1) All the cells of the right and left extension matrix are not computed reducing the computation time. 2) The traceback matrix is very small. GACT performs normal Smith-Waterman on a submatrix of the extension matrix, known as *tiles* of size TxT. After computing a tile it computes the next tile, which overlaps the previous tile with at least O cells on both axes. For reasonable values for T and O, GACT has shown to produce the same result as normal Smith-Waterman [8]. Figure 2 shows an example of computing the extension matrix with the GACT algorithm. In the example $T = 8$ and $O = 3$. The tiles are computed in the order $T1, T2, T3$ and $T4$. The example in Fig. 2 can be used to explain both the computation of left and right extension matrices. The only difference is $R = \overline{R\_right}$ and $Q = \overline{Q\_right}$ in case of right extension, where $\overline{R\_right}$ and $\overline{Q\_right}$ is the reverse of *R_right* and *Q_right* sequences, respectively. Listing 1, shows the algorithm for left extension. Positions *i_curr* and *j_curr* are produced by D-SOFT. The start and end position of the current tile are stored in (*i_start*, *j_start*) and (*i_curr*, *j_curr*), respectively. The traceback path of the whole left extension is kept in *tb_left*. The function *Align*() uses Smith-Waterman to compute traceback matrix *D* between subsequences *R_tile* and *Q_tile*. Once the traceback matrix is filled, traceback is performed starting from the bottom-right cell, except for the first tile, where traceback starts from the highest-scoring cell. The starting cells of the traceback are coloured yellow in Fig. 2. *Align*() returns the number of bases in *R* and *Q* aligned by this tile (*i_off*,*j_off*), the traceback arrows/pointers (*tb*) and the position of the highest-scoring cell (*i_max*,*j_max*). *Align*() also limits *i_off* and *j_off* to at most

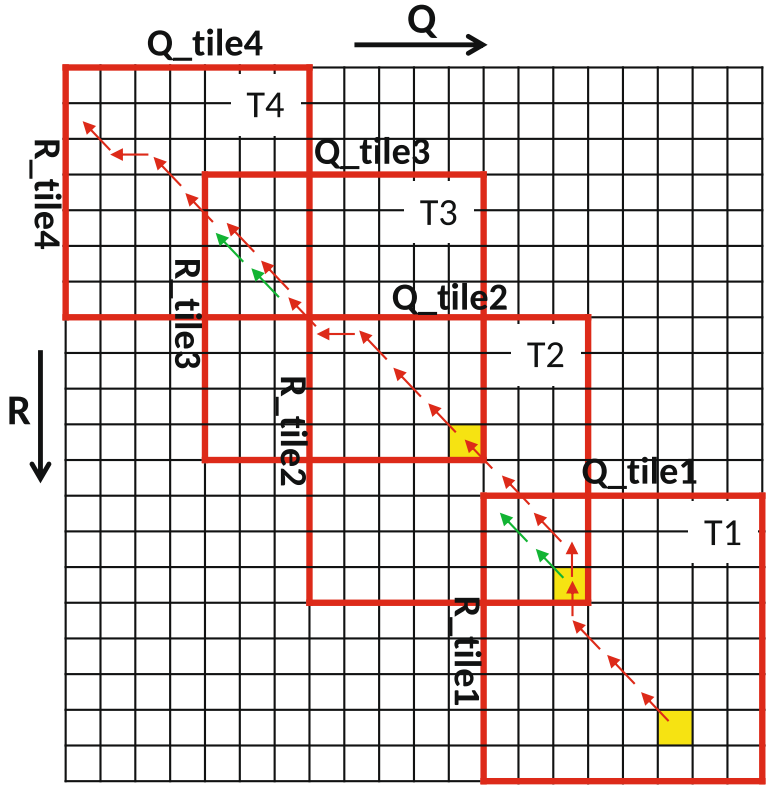

Fig. 2 An example of the GACT algorithm

Listing 1 GACT algorithm

```
tb_left = []
(i_curr, j_curr) = (i_seed, j_seed)
t = 1
 while (i_curr > 0 and j_curr > 0)
{
 (i_start, j_start) = (Max(0, i_curr − T), Max(0, j_curr − T
     ))
 (R_tile, Q_tile) = (R[i_start : i_curr], Q[i_start : i_curr
     ])
 (i_off, j_off, i_max, j_max, tb) = Align(R_tile, Q_tile, t,
      T–O)
 tb_left.Prepend(tb)
 if (t == 1){
 (i_curr, j_curr) = (i_max, j_max)
 t = 0
 }
 if (i_off == 0 and j_off == 0) {
 break;
 }
 else {
 (i_curr, j_curr) = (i_curr − i_off, j_curr − j_off)
 }
}
return (i_max, j_max, tb_left)
```

$T - O$ bases, to ensure the next tile overlaps by at least O bases on both R and Q. The green arrows shows the path taken by the traceback in a tile if there is no limit and the traceback is allowed to complete. The left extension finishes when it hits the end of R or Q, or when traceback cannot add any bases to the existing alignment. The memory needed for the traceback is $\mathcal{O}\left(T^2\right)$, which is constant since T is chosen upfront. The whole alignment of the extension is contained in *tb_left* and is equivalent to the path traced by the red arrows in Fig. 2. The alignment score of the extension can also be computed with the help of *tb_left* The right extension operates on the reverse of R and Q.

The performance of GACT is linear ($O(max\{|ReadA|, |ReadB|\} \cdot T)$) where $|ReadA|$ and $|ReadB|$ are the lengths of *Read A* and *Read B*, respectively. It is more suited for long reads than banded alignment [19] because banded alignment uses a static band around the main diagonal. GACT allows for flexible bands since the position of the new tile depends on the traceback path, this is useful for long reads that have high indel rates.

GPU processing

A GPU is a Graphics Processing Unit, which is a processor that is mainly used to perform video processing. GPUs contain many cores that allow them to perform parallelizable tasks very quickly. A GPGPU, or General Purpose GPU, can be programmed to perform tasks that are different from video processing. GPUs cannot operate on their own, they

must be guided by a CPU. The functions that run on a GPU are called *kernels* and are usually launched by a CPU.

CUDA is a parallel programming platform that allows people to use Nvidia GPUs for their applications. Developers can write kernels, launched from a CPU function. The GPU is referred to as *device* and the CPU as *host*. Kernels can be launched from the CPU with a certain number of *thread blocks* with each block containing many GPU threads. The number of blocks and the number of threads in a block are the kernel launch parameters. Each thread executes the kernel code, although they usually operate on different data.

On a hardware level, an NVIDIA GPU is divided into Streaming Multiprocessors (SM). Each SM contains several cores, or Streaming Processors (SP), these are the basic building blocks and perform the actual calculations. Each block is assigned to at most one SM. This block's threads are then executed as warps, with 32 threads per warp. Each SM has multiple warp schedulers, so multiple warps can run in parallel on an SM. All threads in a warp must execute the same instruction, if a thread is the only to take a branch, the other threads must wait until the branch is completed, this is called *thread divergence*.

GPUs have several different memory types and levels. It has its own DRAM known as the *global memory* and a cache shared by all SM's. Accesses to the global memory are also executed in parallel, this means that all threads try to read/write to the memory in parallel. If the addresses are next to each other, only one memory transaction is needed, since a transaction processes a whole memory line. This is known as coalescing. Non-coalesced memory accesses cause multiple memory transactions.

A general workflow using a GPU is:

1. Data is copied from the main memory to the GPU global memory.
2. The CPU launches the GPU kernel.
3. The GPU executes the kernel.
4. Results are copied from the GPU to the CPU memory.

Previous research

Multiple efforts to accelerate the DNA alignment algorithm on GPU have been made. MUMmerGPU [20] is one of the first GPU accelerated algorithms, it stores a suffix tree of the reference sequence on the GPU, and aligns it with queries. Its newest GPU implementation shows a 13x speedup over the CPU implementation. CUDAlign [21] accelerates the exact Smith-Waterman algorithm and allows an affine gap. The input sequence length is only restricted by the available global memory. It uses linear space and boasts a 702x and 19.5x speedup compared to 1 core and 64 cores, respectively. CUSHAW2-GPU [22] is an accelerated short read aligner. Other work has been done on accelerating BWA-MEM [23] and Protein database search [24].

Implementation

Profiling

We measured the runtime of various elements of the Darwin algorithm on the CPU. Two notable parts are D-SOFT (which consists of building the minimizer table, and finding the seeds) and aligning using GACT. Of those two, the *Align*() function in GACT (Listing 1) takes the most time, namely 99.9% for Pacbio reads. Therefore, we accelerated the *Align*() function on GPU. We selected a tile size T of 320 as it gives optimal Smith-Waterman

alignment scores [8]. With this setting, 63% of the tiles are exactly 320x320. The remaining tiles are smaller as they occur near the edges of *R*-*Q* matrix (Fig. 2). If some GPU threads have a smaller tile size, it will cause some divergence, because they will have to wait until the threads with larger tile sizes are finished.

Acceleration

It is possible to run the whole GACT kernel on the GPU, for both left and right extension. However, since it is not known how long the resulting alignment will be, and all GPU threads have to wait until all threads are done, this will cause lots of idle time. Instead, it is chosen to only have a single tile of size TxT aligned per GPU-thread per GPU-invocation as shown in Fig. 3. The *Align*() function for many different R, Q pairs are executed in parallel on the GPU. Figure 4 shows the flow graph of GPU accelerated Darwin. It has two GPU kernels shown by green processes in the flow graph. All other tasks are performed on the CPU. The CPU builds the minimizer table using all the reads for which the overlaps have to be computed. The accelerated algorithm processes a set of reads to exploit the massive parallelism of the GPU. The CPU first computes the seed hits for all the reads in the set using the D-SOFT algorithm. With the help of the seed hit location the sequences for the left and right extension matrices are determined. i.e. (*R_left*,*Q_left*) and (*R_right*,*Q_right*). One tile (*R_tile*, *Q_tile* pair) from each extension matrix is assigned to a GPU thread for alignment. All the tile alignments are computed in parallel on the GPU. There are enough seed hits, and hence sufficient extension matrices in the set of reads to fully utilize all the GPU resources. In the post-processing step, full alignment of the extension using *tb_left* (and *tb_right* for right extension) is constructed on the CPU. As described in the "Profiling" section all the tasks other than computing the alignment between *R_tile* and *Q_tile* takes a negligible amount of time on the CPU.

To reduce the GPU memory accesses, the alignment is preceded by a packing step as indicated in the flow graph of Fig. 4. The bases of both sequences are packed in a 4-bit format, where 8 bases are packed into a 32-bit integer. This packing is performed on the GPU and it is hundreds of times faster than packing bases on CPU [25]. To align *R_tile* with *Q_tile*, we extended the local alignment kernel of GASAL (GPU Accelerated Sequence Alignment Library) library [25]. The tile is subdivided into submatrices of size 8x8. Since there are 8 bases in one integer, only two global memory accesses are required to compute a single submatrix. The layout of a tile computed on the GPU is shown in Fig. 5. Each green box contains an 8x8 submatrix. The submatrices are computed in the order

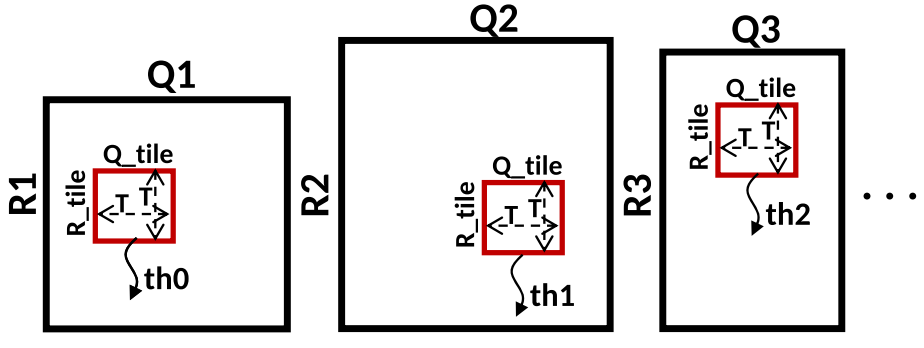

Fig. 3 GPU thread assignment for parallelization of the tile computation in GACT. *th* is a GPU thread

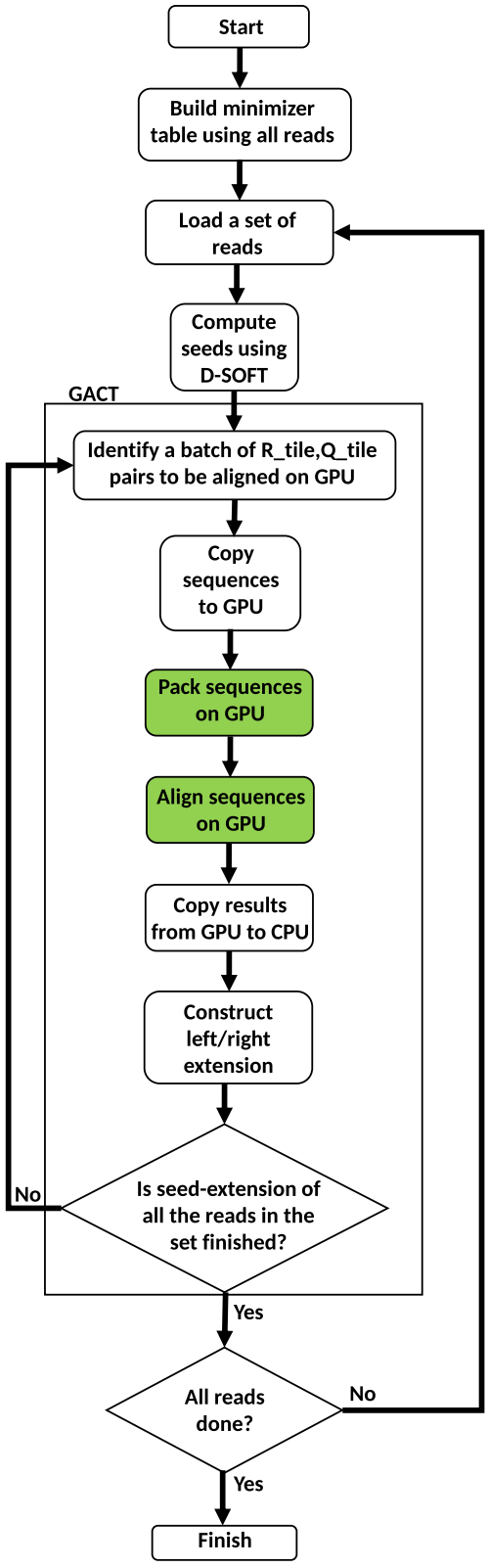

Fig. 4 The flow graph of GPU accelerated Darwin. The green boxes are GPU kernels

shown by their number. The arrows in the submatrix number 4 show the order of computation of the dynamic programming cells in a submatrix. The figure shows that the *Q_tile* sequence is read multiple times. Hence, packing the sequence with 4 bits per base helps to keep it in the cache for faster access. It is clear from Eqs. 1-4 that to compute a column of the submatrix only the cells in the left column and the row above it are required. The required column and row are colored blue in Fig. 5. The column has only 8 elements, and hence can be stored in GPU registers. Therefore, the total amount of memory required is $\mathcal{O}\left(T + T^2\right)$, where $\mathcal{O}(T)$ is required for computing maximum alignment score m and $\mathcal{O}\left(T^2\right)$ for storing the traceback matrix D. Algorithm 1 shows the GPU implementation of the *Align*() function. The pseudocode above Line 4 is for computing the position of maximum alignment score ($i_max, jmax$) and the traceback matrix D. Observe that the all the writes in D are coalesced to optimize the memory bandwidth and reduce the number of memory transactions. The pesudocode below Line 4 is for computing the traceback path *tb*. The GPU accelerated Darwin supports both linear as well as affine gap penalties. Algorithm 1 shows the alignment using the linear gap penalties. The algorithm with affine gap penalties has a similar layout and omitted here for brevity.

Algorithm 1: The GPU kernel for the *Align*() function of GACT (Listing 1) to compute a tile of size TxT

Input: Two sequences R_{tile}*_pack* and Q_{tile}*_pack* of length $T/8$ packed in 32 bit words where each word contains 4 bases (4 bits per base); A backtrack matrix D of size $(T \cdot T) \cdot n_{threads}$ allocated in the GPU memory, where $n_{threads}$ is the number of GPU threads; tile number t; maximum number of bases to align max_{off}; number of GPU threads $n_{threads}$, GPU thread ID tid, score for match $match$, mismatch penalty $mismatch$ and gap penalty gap are known by default

Output: Number of aligned bases in R_{tile} and Q_{tile} (i_{off}, j_{off}); position of the maximum alignment score (i_{max}, j_{max}) and the traceback arrows/pointers tb

1 **Function** AlignOnGPU(R_{tile}*_pack* , Q_{tile}*_pack*, D, t, max_{off}) **begin**
2 Initialize H array of length T containing zeros
3 $max\_sc = 0$
4 **for** $i = 0$ **to** $T/8 - 1$ **do**
5 Initialize h and p arrays of length of 9 containing zeros
6 $r = R_{tile}$*_pack*$[i]$ // r is a 32 bit word
7 $r_{idx} = i$
8 $q_{idx} = 0$
9 **for** $j = 0$ **to** $T/8$ **do**
10 $q = Q_{tile}$*_pack*$[i]$ // q is a 32 bit word
11 **for** $k = 1$ **to** 8 **do**
12 $qbase = (q >> (32 - (k \cdot 4)))\&15$ /* C++ like shift operation followed by AND to extract a base from a 32 bit word */
13 $h[0] = H[q_{idx}]$
14 **for** $m = 1$ **to** 8 **do**
15 $rbase = (r >> (32 - (m \cdot 4)))\&15$ /* C++ like shift operation followed by AND to extract a base from a 32 bit word */
16 **if** $qbase == rbase$ **then**
17 $tmp = p[m] + match$
18 **else**
19 $tmp = p[m] - mismatch$
20 $h[m] = \max\{tmp, p[m] - gap, h[m-1] - gap, 0\}$
21 **if** $h[m] == 0$ **then**
22 $D[(((r_{idx} + (m-1)) \cdot T) + q_{idx}) \cdot n_{threads}) + tid] = 0$
23 **else if** $h[m] == tmp$ **then**
24 $D[(((r_{idx} + (m-1)) \cdot T) + q_{idx}) \cdot n_{threads}) + tid] = \nwarrow$
25 **else if** $h[m] == p[m] - gap$ **then**
26 $D[(((r_{idx} + (m-1)) \cdot T) + q_{idx}) \cdot n_{threads}) + tid] = \leftarrow$
27 **else if** $h[m] == h[m-1] - gap$ **then**
28 $D[(((r_{idx} + (m-1)) \cdot T) + q_{idx}) \cdot n_{threads}) + tid] = \uparrow$
29 **if** $h[m] > max\_sc$ **then**
30 $max\_sc = h[m]$
31 $i_{max}, j_{max} = \{r_{idx} + (m-1), q_{idx}\}$
32 $p[m] = h[m-1]$
33 $H[q_{idx}] = h[m]$
34 $q_{idx} = q_{idx} + 1$
35 // the traceback starts below
36 **if** t==1 **then**
37 $(start_i, start_j) = (i_{max}, j_{max})$
38 **else**
39 $(start_i, start_j) = (T-1, T-1)$
40 **while** $D[((start_i \cdot T) + start_j) \cdot n_{threads}) + tid]! = 0$ **and** $i_{off} \leq max_{off}$ **and** $j_{off} \leq max_{off}$ **and** $start_i \geq 0$ **and** $start_j \geq 0$ **do**
41 tb.prepend($D[((start_i \cdot T) + start_j) \cdot n_{threads}) + tid]$)
42 **if** $D[((start_i \cdot T) + start_j) \cdot n_{threads}) + tid] == \nwarrow$ **then**
43 $start_i = start_i - 1$
44 $start_j = start_j - 1$
45 $i_{off} = i_{off} + 1$
46 $j_{off} = j_{off} + 1$
47 **else if** $D[((start_i \cdot T) + start_j) \cdot n_{threads}) + tid] == \uparrow$ **then**
48 $start_i = start_i - 1$
49 $i_{off} = i_{off} + 1$
50 **else if** $D[((start_i \cdot T) + start_j) \cdot n_{threads}) + tid] == \leftarrow$ **then**
51 $start_j = start_j - 1$
52 $j_{off} = j_{off} + 1$
53 **return** $\{(i_{off}, j_{off}), (i_{max}, j_{max}), tb\}$

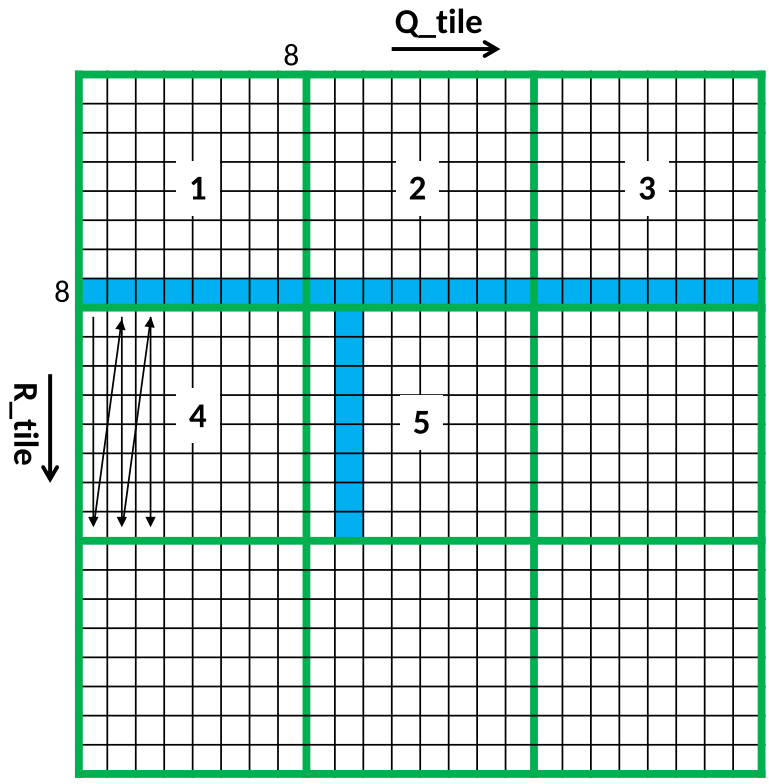

Fig. 5 Layout of the tile computed on the GPU. Each green box is a 8x8 submatrix

Sequence alignment of long DNA sequence is a performance bottleneck in genome analysis algorithms. The results of Darwin alignment are same as for normal Smith-Waterman for reasonable values of T and O. Hence, the Darwin algorithm can also be applied for Smith-Waterman alignment between two long DNA sequences and our GPU implementation of Darwin can be used to accelerate Smith-Waterman alignment (with traceback) for long DNA sequences.

Results

We compared our GPU acceleration with the hand-optimized CPU version of Darwin [26] (commit: 16bdb81). Tests are performed on both IBM as well as Intel machines. The IBM machine (S824L) has 2 sockets with each socket containing a 10-core POWER8 @ 3.42 GHz processor. Each core has 8-way Simultaneous Multithreading. Hence, there are 160 logical cores in total. The machine has 256 GB of RAM and a Tesla K40m. The CUDA version is 7.5, and the operating system is Ubuntu 3.19.0-28-generic. The GCC version is 4.9.2.

The Intel machine has 2 sockets with each socket containing a 6-core Xeon E5-2620 @ 2.4 GHz processor. Each core has 2-way Hyperthreading. Hence, there are 24 logical cores in total. The machine has 32 GB of RAM and a Tesla K40c. The CUDA version is 9.2, and the operating system is CentOS 7.5. The GCC version is 4.8.5. The K40c and K40m have the same performance, the only difference lies in their cooling method.

We use Pacbio 54x Human sequencing data [27]. The data has a size of 172 gigabytes containing 21,856,161 reads. Since the runtime of the experiments has a quadratic

relationship with the number of reads, we use first 50 megabytes (8566 reads) as the input dataset to finish the experiments in a reasonable time. Even with 50 megabytes input dataset, the CPU implementation takes more than 2 hours to run on 8 threads of the Intel machine (Fig. 8). The input dataset contains reads up to 33 kilo bases long with an average read length of 6 kilo bases. Darwin computes the overlaps between the reads in the input dataset. The settings for GPU and CPU implementation are as follows: $match, mismatch, gapo, gape = (1, -1, -1, -1)$, $N = 800$, $T = 320$, $O = 120$, $K = 14$, $h = 21$, $w = 1$.

Since our GPU implementation accelerates only the *Align*() function on the GPU, everything else is executed on the CPU with multiple threads. Each CPU thread launches a batch of *R_tile* and *Q_tile* sequences to be aligned on the GPU. Since all these CPU threads share a single GPU, it is necessary to investigate how the choice of numbers of CPU threads, number of GPU blocks and the number of threads in a block affect the performance. Figure 6 shows the total execution size for various settings of these factors. The figure shows that the fastest execution time is obtained with 8 CPU threads running with the GPU launch parameters of 32 blocks and 64 threads per block. We performed a similar analysis for the Intel machine and found that 8/32/64 is the best setting in the case of the Intel machine as well. Therefore, we used the 8/32/64 ((Number of CPU threads) / (number of blocks) / (number of threads per block)) setting for running the GPU implementation in the remainder of the experimental results.

Figure 7 shows the total execution time of the CPU implementation of Darwin and compares it with the total execution time of the GPU accelerated Darwin, for the IBM machine. Note that the *y*-axis of the figure represents a logarithmic scale due to the high speedup achieved by the GPU implementation. The CPU implementation is running with 64 threads, which gives nearly the fastest execution time. Two GPU times are reported: "GPU" and "GPU-coalesced". "GPU" is the time without coalescing the accesses to the traceback matrix *D*. The figure shows that the GPU acceleration without coalescing is 2.4x faster than the CPU implementation. Coalescing further accelerates the GPU implementation by 10x to achieve an overall speedup of 24x.

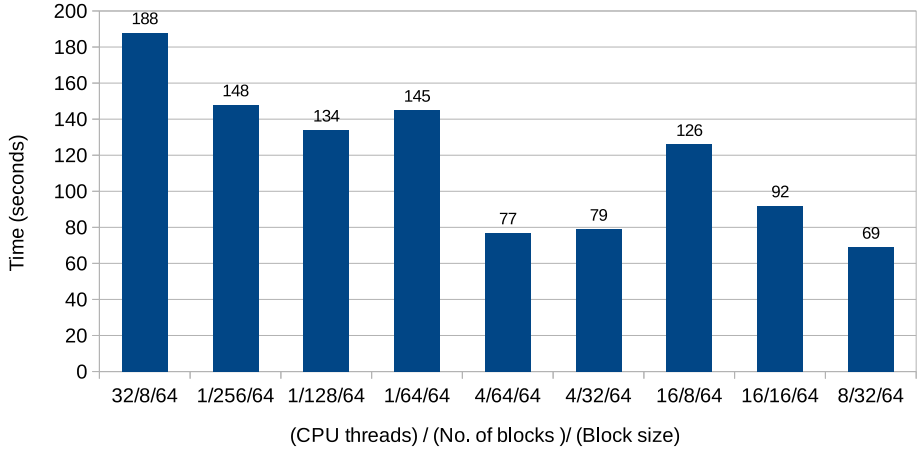

Fig. 6 Total execution time of GPU accelerated Darwin for different values of number of CPU threads, numbers of blocks and block size on an IBM machine

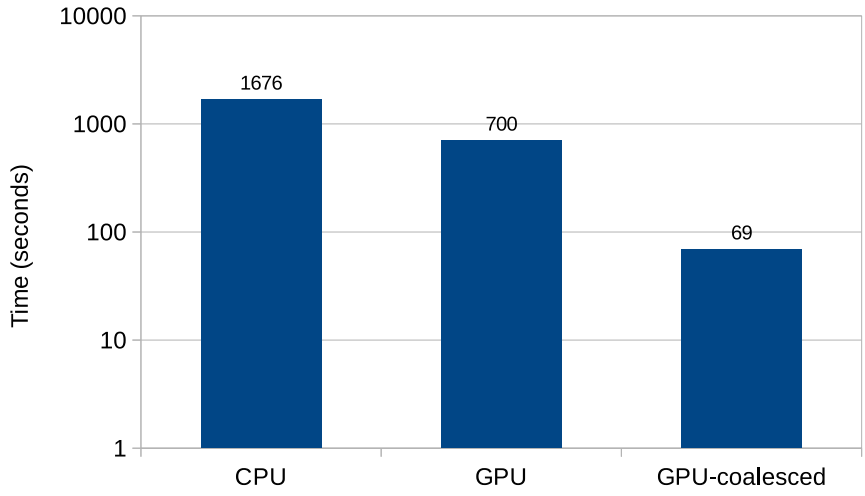

Fig. 7 Total execution time of the Darwin's CPU implementation and GPU accelerated Darwin, on the IBM machine. The CPU implementation is running with 64 CPU threads

Figure 8 show the comparison of total execution times on the Intel machine, again with a logarithmic scale on the *y*-axis. The CPU implementation is running with 8 threads, which gives nearly the fastest execution time. The non-coalesced GPU implementation achieves a speedup of 11.8x over the CPU. With coalesced memory accesses the speedup becomes 109x. Figures 7 and 8 indicate that coalescing helps to improve the speedup by around 10x. This happens due to efficient utilization of large GPU global memory bandwidth.

The above results were obtained with the linear-gap penalty model which is also the default setting in Darwin. However, Darwin and hence our GPU acceleration also support the affine gap model. Table 1 shows the total execution time of the CPU and GPU implementation of Darwin for various values of *match*, *mismatch*, *gapo* and *gape* on the

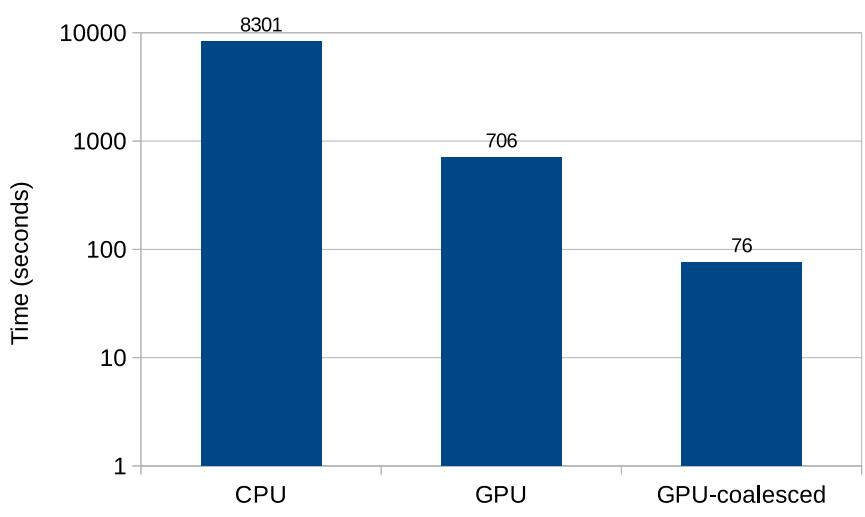

Fig. 8 Total execution time of Darwin's CPU implementation and GPU accelerated Darwin, on the Intel machine. The CPU implementation is running with 8 threads

Table 1 Runtimes and speedup for different scoring schemes on the IBM machine

	(2,-1,-2,-2)	(1,-3,-1,-1)	(5,-4,-10,-1)
CPU	31m15	21m28s	31m27s
GPU-coalesced	76.0	59.3	78.0
speedup	24.7	21.7	24.2

IBM machine. The CPU implementation is running with 64 threads. The table shows that the speedup nearly remains constant (20x-25x) regardless of the scoring scheme. Hence our GPU acceleration is equally effective for both linear and affine gap penalty scoring models.

Conclusions

Read overlapping is an important step in OLC based de novo assemblers. Darwin is a fast and accurate read overlapper for assembly of long DNA reads. It is based on the seed-and-extend paradigm. It has two stages: 1) D-SOFT, to compute the seeds and 2) GACT, to extend the seed hits on both sides to compute the overlap between two reads. The ASIC implementation of Darwin is shown to be hundreds of times faster than software based read overlappers. GPUs are cost-effective and easily accessible processing units that are used to accelerate many high performance applications. In this paper, we have shown a GPU implementation of Darwin which accelerates the Smith-Waterman alignment with traceback computation used in the GACT stage. We pack the sequences on the GPU and compute the Smith-Waterman alignment matrix by dividing the matrix into 8x8 submatrices. This helps to reduce the GPU memory accesses. To further reduce the memory transactions, writing to the traceback matrix is coalesced. We tested our implementation against the hand-optimized CPU implementation of Darwin. The results show that using the real Pacbio dataset, our GPU implementation is 24x faster than 64 IBM Power8 threads and 109x faster than 8 Intel Xeon threads, regardless of the scoring scheme (linear or affine gap). The GPU implementation can also be used to accelerate generic Smith-Waterman alignment of long DNA sequences. The implementation is available at https://github.com/Tongdongq/darwin-gpu.

Availability and requirements

Project name: darwin-gpu
Project home page: https://github.com/Tongdongq/darwin-gpu
Operating system(s): Linux
Programming language: C++, CUDA
Other requirements: CUDA toolkit version 8 or higher.
License: Apache 2.0
Any restrictions to use by non-academics: Not applicable

Abbreviations

ASIC: Application Specific Integrated Circuit; CPU: Central Processing Unit; CUDA:Compute Unified Device Architecture; DNA: Deoxyribonucleic Acid D-SOFT: Diagonal-band Seed Overlapping based Filtration Technique; GACT: Genome Alignment using Constant memory Traceback; GASAL: GPU Accelerated Sequence Alignment Library; GPU: Graphical Processing Unit; NGS: Next Generation Sequencing; SM: Streaming Multiprocessor; SP: Streaming Processor

Acknowledgments

We are thankful to Yatish Turakhia from Stanford University for his help during the research.

About this supplement
This article has been published as part of *BMC Bioinformatics Volume 21 Supplement 13, 2020: Selected articles from the 18th Asia Pacific Bioinformatics Conference (APBC 2020): bioinformatics*. The full contents of the supplement are available online at https://bmcbioinformatics.biomedcentral.com/articles/supplements/volume-21-supplement-13.

Authors' contributions
NA provided the initial GPU implementation, wrote the manuscript and performed some experiments. TDQ carried out the study, extended the GPU implementation to be integrated in Darwin, performed the experiments and wrote some parts of the manuscript. KB headed the research and revised the manuscript. ZA proposed the idea, supervised the research and helped in writing the manuscript. All author(s) read and approved the final manuscript.

Funding
This work is partly funded by the Faculty Development Program of the University of Engineering and Technology Lahore and by the FitOptiVis ECSEL Joint Undertaking project under grant number H2020 - ECSEL - 2017 - 2 - 783162. Publication costs are funded by the TUDelft library open access fund.

Availability of data and materials
Not applicable.

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Published: 17 September 2020

References

1. Kececioglu JD, Myers EW. Combinatorial algorithms for dna sequence assembly. Algorithmica. 1995;13(7):7–51.
2. Myers G, Tischler G, Cunial F, Pippel M. DAZZLER: Dresden Azzembler for Long Read DNA Projects. https://https://dazzlerblog.wordpress.com. Accessed 2 July 2019.
3. Simpson JT, Durbin R. Efficient de novo assembly of large genomes using compressed data structures. Genome Res. 2012;22(3):549–56.
4. Pevzner PA, Tang H, Waterman MS. An eulerian path approach to dna fragment assembly. Proc Natl Acad Sci U S A. 2001;98(17):9748–53.
5. Zerbino D, Birney E. Velvet: algorithms for de novo short read assembly using de bruijn graphs. Genome Res. 2008;18:074492.
6. Simpson JT, Wong K, Jackman SD, Schein JE. Abyss: a parallel assembler for short read sequence data. Genome Res. 2009;19:089532.
7. Luo R, Liu B, Xie Y, Li Z. Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. Gigascience. 2012;1(18):1–6.
8. Yatish Turakhia GB, Dally WJ. Darwin: genomics co-processor provides up to 15,000X acceleration on long read assembly. In: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '18. Williamsburg: ACM; 2018. p. 199–213.
9. Ahmed N, Lévy J, Ren S, Mushtaq H, Bertels K, Al-Ars Z. GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data. BMC Bioinformatics. 2019;20(1):520.
10. Ren S, Ahmed N, Bertels K, Al-Ars Z. GPU accelerated sequence alignment with traceback for GATK HaplotypeCaller. BMC Genomics. 2019;20(2):184.
11. Houtgast EJ, Sima V-M, Bertels K, Al-Ars Z. Hardware acceleration of bwa-mem genomic short read mapping for longer read lengths. Comput Biol Chem. 2018;75:54–64.
12. Smith TF, Waterman MS. Identification of common molecular subsequences. J Mol Biol. 1981;147(1):195–7.
13. Ahmed N, Bertels K, Al-Ars Z. A comparison of seed-and-extend techniques in modern dna read alignment algorithms. In: 2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). Piscataway: IEEE; 2016. p. 1421–8.
14. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. J Mol Biol. 1990;215(3):403–10.
15. Roberts M, Hayes W, Hunt BR, Mount SM. Reducing storage requirements for biological sequence comparison. Bioinformatics. 2004;20(18):3363–9.
16. Rucci E, Garcia C, Botella G, De Giusti A, Naiouf M, Prieto-Matias M. SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences. BMC Syst Biol. 2018;12(5):96.
17. Farrar M. Striped smith–waterman speeds database searches six times over other SIMD implementations. Bioinformatics. 2007;23(2):156–61.
18. Hirschberg DS. A Linear Space Algorithm for Computing Maximal Common Subsequences. Commun ACM. 1975;18(6):341–3.
19. Chao KM, Pearson WR, Miller W. Aligning two sequences within a specified diagonal band. Comput Appl Biosci CABIOS. 1992;8(5):481–7.
20. Trapnell C, Schatz MC. Optimizing data intensive gpgpu computations for dna sequence alignment. Parallel Comput. 2009;35(8):429–40.

21. de O Sandes EF, de Melo ACMA. Smith-waterman alignment of huge sequences with gpu in linear space. In: 2011 IEEE International Parallel Distributed Processing Symposium. Piscataway: IEEE; 2011. p. 1199–211. https://doi.org/10.1109/IPDPS.2011.114. https://ieeexplore.ieee.org/document/6012857/.
22. Liu Y, Schmidt B. CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing. Des Test IEEE. 2014;31(1):31–39.
23. Houtgast EJ, Sima VM, Bertels KLM, Al-Ars Z. An efficient gpu-accelerated implementation of genomic short read mapping with bwa-mem. In: Proc. International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies. Hong Kong, China: ACM; 2016.
24. Hasan L, Kentie MA, Al-Ars Z. Dopa: Gpu-based protein alignment using database and memory access optimizations. BMC Res Notes. 2011;4:1–11.
25. Ahmed N, Mushtaq H, Bertels KLM, Al-Ars Z. Gpu accelerated api for alignment of genomics sequencing data. In: Proc. IEEE International Conference on Bioinformatics and Biomedicine. Piscataway: IEEE; 2017. p. 510–515.
26. Turakhia Y. Darwin: A co-processor for long read alignment. https://github.com/yatisht/darwin. Accessed 5 Nov 2018.
27. Data release: 54x long-read coverage for PacBio-only de novo human genome assembly. 2014. https://www.pacb.com/blog/data-release-54x-long-read-coverage-for/. Accessed 2 July 2019.

Publisher's Note