The Journal of Supercomputing (2023) 79:8699–8728
https://doi.org/10.1007/s11227-022-05007-z

Irregular alignment of arbitrarily long DNA sequences on GPU

Esteban Perez-Wohlfeil[1,2] · Oswaldo Trelles[2] · Nicolás Guil[2]

Accepted: 4 December 2022 / Published online: 26 December 2022

Abstract

The use of Graphics Processing Units to accelerate computational applications is increasingly being adopted due to its affordability, flexibility and performance. However, achieving top performance comes at the price of restricted data-parallelism models. In the case of sequence alignment, most GPU-based approaches focus on accelerating the Smith-Waterman dynamic programming algorithm due to its regularity. Nevertheless, because of its quadratic complexity, it becomes impractical when comparing long sequences, and therefore heuristic methods are required to reduce the search space. We present GPUGECKO, a CUDA implementation for the sequential, seed-and-extend sequence-comparison algorithm, GECKO. Our proposal includes optimized kernels based on collective operations capable of producing arbitrarily long alignments while dealing with heterogeneous and unpredictable load. Contrary to other state-of-the-art methods, GPUGECKO employs a batching mechanism that prevents memory exhaustion by not requiring to fit all alignments at once into the device memory, therefore enabling to run massive comparisons exhaustively with improved sensitivity while also providing up to 6x average speedup w.r.t. the CUDA acceleration of BLASTN.

Keywords Gpu acceleration · Comparative genomics · Sequence comparison · CUDA

Esteban Perez-Wohlfeil
estebanpw@uma.es

Oswaldo Trelles
ots@ac.uma.es

Nicolás Guil
nguil@uma.es

1 Dynatrace Research, Linz, Austria

2 Computer Architecture Department, University of Málaga, Málaga, Spain

1 Introduction

The GPGPU computing model (General-purpose computing on graphics processing units) [1] is becoming more and more widely applied in nearly all research fields that require intensive computation [2]. For instance, GPU computing is becoming the standard for both recent and traditional disciplines, such as Machine Learning [3], Chemistry [4] or Physics [5], whose experiments are largely inconceivable without hardware accelerators [6]. However, the low cost and versatility of these devices come at a price: performance is directly subjected to how well the computing problem can be modelled using data parallelism. Such imposition can become a strong barrier towards the acceleration of certain applications. This is generally the case in the field of bioinformatics [7], where a large deal of the applications is irregular in nature (*e.g.* dependent on inherent patterns), and hence their parallelization in GPU architectures is typically not immediate nor does it yield significant speedup [8]. For instance, sequence comparison algorithms [9–11] are made of several independent processes which usually require computing hash values, dynamic programming tables, large I/O operations, sorting procedures, etc., all of which add up to the difficulty of adapting irregular applications to the data parallelism paradigm. Therefore, merely porting CPU-based algorithms into GPUs is not sufficient to achieve an adequate level of performance [12], but rather redesigning and adapting the algorithm is required.

Furthermore, besides the technical difficulties of transforming irregular CPU-based algorithms into regular GPU-based, the field of sequence comparison is accompanied by a tremendous explosion in data availability, with databases growing exponentially both in size [13] and length. In fact, the ability to exhaustively align these massive genomes results in numerous advantages, such as providing new insight into the evolution of species [14] or developing specialized treatments in precision medicine [15], among others. Nevertheless, the trend of increasingly more and larger genomes is creating a series of computational bottleneck in the field of sequence comparison that cannot be properly addressed by current software method, for which current software methods cannot cope.

This remains true even in the case of hardware accelerators, since the majority of GPU sequence-comparison implementations are oriented towards optimal alignment using the dynamic programming Needleman-Wunsch or Smith-Waterman algorithm [16]. These algorithms represent exceptional examples of multifold GPU acceleration even when data dependencies exist, achieving up to a 100x speedup compared to CPU-based versions. Examples of this category include CUDALIGN [17] or SW# [18], which reportedly can compute full-chromosome comparisons in several hours [19], despite the two-orders-of-magnitude speedup. While proposals such as CUDALIGN can also make use of further GPU devices to speed up the comparison even more, the approach can require a considerable amount of compute time if we consider a full-genome comparison, which requires a quadratic number of comparisons. This typically results in around 500 chromosome comparisons per pair of species. Moreover, and besides the difference in complexity, exact and heuristic alignments can be more or less adequate depending on the goals of the sequence comparison experiment. Therefore, the lack of a GPU-capable alternative to exact dynamic programming algorithms motivates the necessity of redesigning heuristic and seed-and-extend sequence comparison algorithms for GPU devices to cope with the enormous search spaces derived from all-vs-all chromosome comparisons.

Current heuristic methods typically employ either alignment-free [20] or seed-and-extend [21] strategies. While the first serve their purpose as extremely fast approximations that require nearly no computational resources (*e.g.* CHROMEISTER [22]), fine-grained alignments are often needed for in-depth genetic analyses, and these require the actual calculation of alignments which in turn consumes more computing time. Examples of seed-and-extend algorithms include MEGABLAST [23] and GECKO [24]. To the best of our knowledge, MEGABLAST is the only seed-and-extend algorithm that has been subject of a GPU implementation (namely GBLASTN [25]). However, its purpose is mostly focused on the comparison of several small query sequences with large databases, and therefore it is not prepared for chromosome-level comparisons. In this manuscript we present GPUGECKO, a CUDA [26] implementation for GPU devices of the seed-and-extend and ungapped sequence-comparison algorithm designed to be able to handle massive all-vs-all chromosome (or any type of nucleotide sequence) comparisons. Based on GECKO, our proposal describes how to overcome several challenges and limiting factors of the irregular sequence comparison algorithm when deployed on GPUs. These include:

1. An unknown and explosive number of alignment seeds, which results in unbalanced load.
2. The arbitrary length of extended alignments, which requires careful synchronization mechanisms to avoid unbalance, branch divergence and warp stalling.
3. The overlap of seeds and their extension, which can result in redundant computation.

The main contributions of this manuscript towards the overcoming of such limitations are two novel kernels that make use of parallel reductions to work cooperatively in a fine-grained fashion, which results in effectively dealing with irregularity without requiring expensive synchronization mechanisms.

Moreover, we show that GPUGECKO is not only faster, but also more sensitive in terms of the number of detected alignments than state-of-the-art methods since no masking of low complexity regions nor removal of highly repetitive seeds is performed. To the best of our knowledge, GPUGECKO is the only native seed-and-extend sequence comparison software capable of fully aligning whole mammalian chromosomes in under one minute on average on a GeForce 980 GTX device. The GPUGECKO software is free to use and can be downloaded and compiled from its GitHub repository [27].

Furthermore, even if the proposed strategies are described for the application domain of DNA sequence comparison, these mechanisms for GPU acceleration are also applicable in any scenario involving string comparison [28]. For instance, sequence alignment algorithms have already been used for plagiarism detection [29]. Thus, GPUGECKO can be adapted into using an extended alphabet to enable the comparison of massively sized books or any text source while benefiting from the hardware acceleration improvements.

2 Background

In this section, the internals of the original GECKO algorithm are introduced in order to facilitate the understanding of the different kernels proposed in the GPU implementation.

The original GECKO algorithm performs the comparison of two DNA strings employing secondary memory to offload the enormous size of the data structures generated by the processing of chromosome-sized sequences. This is an interesting feature, since it enables to run virtually any comparison that would otherwise result in memory exhaustion (eventually crashing) in most algorithms. However, this feature generates longer delays due to the input/output disk operations (*i.e.* reading and writing), which end up affecting performance negatively, especially when running multiple instances in parallel.

The GECKO workflow begins by first building a dictionary of overlapping words (*i.e.* a mapping between DNA strings of fixed size k and their positions, see Table 1 for related definitions) and saving it to secondary memory. This is repeated for both input sequences. Secondly, each dictionary is sorted by the numeric value of words. Afterwards, equal words from the query and reference sequences are matched together into tuples containing the value of the word itself and its position. These tuples are known as seeds since they are used as starting or seeding points of the posterior alignments (see “From words to alignments” in the Supplementary Material for more details). This part of the process is typically quadratic in the number of equal words, and thus responsible for most of the memory consumption, especially when sequences are closely related or extremely long. Lastly, the seeds need to be sorted in order to enable filtering and posterior extension.

From the perspective of parallelism, the original GECKO implementation[1] included a coarse-grained parallel strategy at the task-level, namely using three threads to compute the dictionaries (one for the query, reference and reverse-complement of the reference sequence) and two threads to compute the generation and matching of seeds. Additionally, a variable number of threads was used to perform sorting (of both words and seeds) in batches along with a final merge.

In the following subsections, more details regarding the algorithmic workflow of GECKO are included.

[1] Source code can be found at https://github.com/otorreno/gecko.

Table 1 Brief definition of common terms used in sequence alignment and particularly in seed-and-extend algorithms

Term	Definition
Sequence	String of nucleotides [A, C, G, T] and N as the unknown character
Query	Sequence for which it is wanted to know its similarity to another sequence
Reference	Sequence for which a query is being aligned to
Word	Substring of size k in position p in a sequence
Seed/hit	Tuple of two equal words which occur both in the query and reference
Alignment	Highly similar segment between two sequences
Sensitivity	The ability of a heuristic method to detect existing alignments
Identity	The sum of pairwise equal nucleotides in an alignment divided by its length
Coverage	The total aligned nucleotides divided by the length of the query sequence

2.1 Words and seeds generation

Each word w_i or substring of fixed size k in the sequence has to be indexed prior to the matching in order to enable a computationally efficient retrieval of equal words, resulting in the computation of $n - k + 1$ words. A collision-free indexation of DNA words can be computed as shown in Equation 1:

$$h_i = \sum_{j=0}^{k} |A|^k * v(s_{i+j}) \tag{1}$$

where h_i is the numerical value of word w_i and $v(x)$ maps each letter of the alphabet $A = \{A, C, G, T\}$ to an integer value $\{0, 1, 2, 3\}$. Such indexation is particularly useful for hash-table approaches (*i.e.* two equal words will map to the same position in the table). Nevertheless, other common indexation approaches include the use of probabilistic hash tables [30], tree-based structures [31] or even data-compression algorithms [32]. In the case of GECKO, the dictionaries need to be sorted to enable the matching between words of both sequences in $\mathcal{O}(n + m)$ time (being n, m the number of words in each dictionary), which results in the generation of seeds.

Note that the size k of the words plays a key role in the efficiency of seed-and-extend algorithms. In essence, it represents a trade-off between speed and sensitivity, with small k values detecting more seeds but being more sensitive to random matches, and vice versa.

The resulting number of seeds is unpredictable and depends mostly on the relatedness of the sequences being compared. For example, if the first $\approx$ 30 megabase pairs of the comparison between chromosome X of *Homo sapiens* and *Mus musculus* were split in $\approx$ 8,000 pieces of equal size (roughly 4,000 by 4,000 base pairs per split), most of the splits would contain less than 100 seeds, whereas some of them would contain almost 10,000,000 seeds.

In order to handle such extreme variation, most algorithms limit the number of seeds generated by high-frequency words (such as low-complexity regions [33]). While this approach effectively reduces the search space, it also implies a loss of sensitivity. With such limitations in mind, in this manuscript we propose an integral acceleration of the seed-and-extend algorithm capable of producing exhaustive and unfiltered results by leveraging the compute capabilities of modern GPU architectures.

2.2 Extension of seeds

The seed-extending module will be described before the filtering process since a general knowledge of the seed-extension is required beforehand. The seed-extending module of GECKO can be defined as the optimization problem of finding the pair of offsets p_1, p_2 that maximize the alignment score $s_{i,j}$ of a given seed, namely:

$$s_{i,j} = \arg\max_{p_1,p_2} \sum_{l=-p_1}^{p_2} g(q_{i+l}, r_{j+l}) \tag{2}$$

where i, j are the coordinates of the seed in both sequences and the function $g(q_i, r_j)$ returns 1 if $q_i = r_j$ and -1 otherwise. Note that the optimization function is subject to the boundaries of the sequences, namely $0 \leq p_1 < min(i,j)$ and $0 \leq p_2 < min(|q| - i, |r| - j)$. A heuristic stopping condition is added such that the scoring of the function cannot drop below zero or otherwise the iterative search may run until exhausting the sequence length while trying to find the optimal set of parameters [34]. The resulting seed extended by p_1 and p_2 is called a High-scoring Segment Pair (or HSP) with score $s_{i,j}$, and represents an ungapped alignment between two regions of the query and reference sequences.

2.3 Filtering of seeds

The filtering of seeds is an essential step in the GECKO workflow that greatly reduces the computation time of the extension step. It allows for a large number of the generated seeds to be filtered due to their close proximity to one another. As an example, consider two exactly equal DNA segments with composition $x_0, x_1, ..., x_k, ..., x_n$ surrounded by unequal segments. First, the number of words will be all of the overlapping substrings $S = \{s_{0:k}, s_{1:k+1}, ..., s_{n-k:n}\}$. Secondly, since the composition of both sequences is equal, there will be a seed for each of the words listed. However, given that each of these seeds belongs to the same segment, the alignment extension will result in the very same HSP. Thus, adjacent seeds can be filtered out up to a maximum distance of two word sizes or $2k$ base pairs (given an equal match/mismatch scoring system), since the score can only reach 0 after as many misaligned nucleotides have been matched as those contained in the original seed.

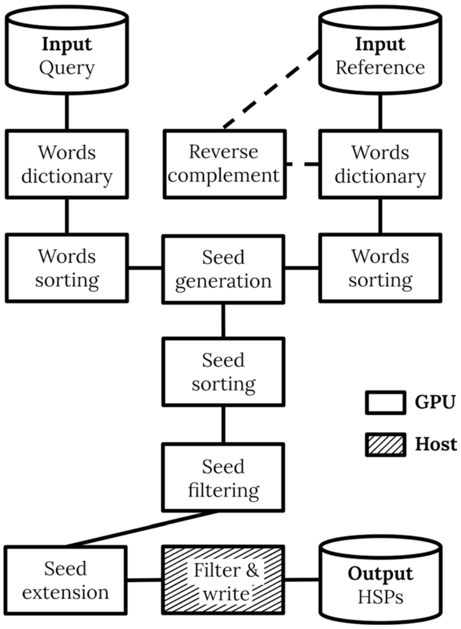

Fig. 1 Seed-and-extend sequence comparison algorithm in GPUGECKO. Note that the reverse complement step is run only once in a second execution of the workflow as a substitute of the input reference sequence. Blank steps represent GPU kernel programs, whereas striped ones represent host programs

3 Methods

The following sections describe the methodology of the proposed algorithm GPUGECKO. Section 3.1 briefly addresses the modules that comprise the seed-and-extend algorithm and analyses the computational load of each stage in the original GECKO algorithm. Section 3.2 describes how GPUGECKO employs a batching strategy to process sequence comparisons of virtually any size. Afterwards, the details, metrics and designs of each kernel are discussed in Sect. 3.3.

3.1 Algorithm description and workload analysis

In short, the seeded sequence comparison algorithm in GECKO can be divided into (1) creating a dictionary (index) of the overlapping words of equal size for each of the sequences and sorting them by their value, (2) linear matching of similar words between both dictionaries into seeds (hits) and (3) extension of seeds into aligned HSPs. A workflow diagram is shown in Fig. 1, featuring all stages of the GPU algorithm. Note that the query sequence needs to be compared against both the reference and the reverse complementary of the reference sequence. This is required in order to enable detection of a certain type of evolutionary event (*i.e.* inversions). Furthermore, all computation is performed within the GPU device from start to finish with the exception of the final filtering and writing of HSPs to disk.

Table 2 shows the amount of computing time spent at each of the stages in the original GECKO implementation. The depicted values were obtained from the sample dataset used throughout the rest of the manuscript, which is aimed at

Table 2 Analysis of the computing requirements per stage in the CPU version of GECKO

Processing step	μ time (%)	σ time (%)	Lower (%)	Upper (%)
Reverse complement	2.16	1.16	1.44	2.88
Words dictionary	27.83	14.37	18.92	36.73
Words sorting	27.08	13.66	18.61	35.55
Seed generation	10.22	7.02	5.87	14.57
Seed sorting	21.65	23.56	7.05	36.26
Seed filtering	1.63	1.32	0.81	2.45
Seed extension	9.44	6.61	5.34	13.54

From left to right, (1) processing stage, (2) average computing percentage, (3) standard deviation of the computing time and (4) upper and lower boundaries of computing percentage as given by the confidence interval at $\gamma = 95\%$

representing real-world sequence comparison scenarios (see the Sample dataset in Sect. 4.1). The lower and upper confidence intervals were calculated from the averaged and normalized runtimes.

It can be observed from Table 2 that the most time-consuming step is the generation of words, although closely followed by the sorting of both words and seeds and finally the seed generation and extension. Although the seed filtering and reverse complement stages only require a small fraction of the computing time (around 2%), these were also included in the GPU pipeline to achieve higher speedup, since otherwise (given Amdahl's law [35]) the theoretical speedup is limited. Moreover, when comparing mammalian chromosomes in CPU-based algorithms, the filtering of seeds can require around 100 seconds, whereas the extension of seeds can take around 10 minutes. Therefore, all stages were implemented as GPU kernels in order to enable the maximum theoretical speedup, even when only a reduced speedup could be achieved due to lesser computational requirements.

3.2 Subsequence batching

The seed-and-extend sequence comparison algorithm is irregular in the number of seeds and in the length of the alignments (HSPs). However, it is the generation of seeds which is particularly difficult, since its quadratic growth can quickly exhaust memory.

In the CPU implementation of GECKO, secondary storage was used to offload seeds (as well as words and HSPs) from the main memory. While this approach enables virtually any system to run massive comparisons, it also has a strong impact on performance both from the perspective of speed and more importantly, on other processes running in the system due to the massive amounts of data that need to be written to disk. In particular, a single execution is likely to overlap computation with I/O operations; however, given enough parallel instances, the latency of the I/O requests might cause long waiting times for running threads due to reading and writing of intermediate results.

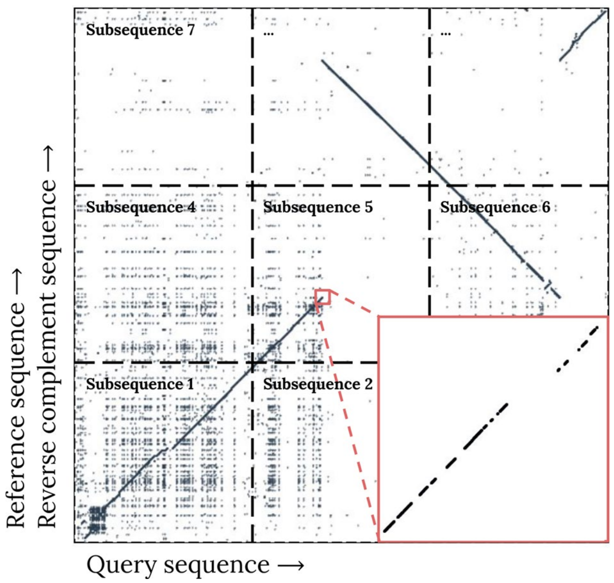

Fig. 2 Example of batched sequence comparison. The query sequence is represented in the x-axis, whereas the reference and reverse complement sequences are represented in the y-axis. Both sequences are split into subsequences and computed independently. Notice that the execution workflow is computed not only on the query and reference sequence, but also on the query and the reverse complement of the reference. The zoomed square shows how small HSPs comprise the main similarity signal. The sequences employed for this example are *Gallus gallus* (common chicken) chr. 18 and *Meleagris gallopavo* (common turkey) chr. 20

In the case of GPUGECKO, given that the number of seeds is unknown until actual computation, a batching procedure is proposed to split the sequences in subsequences on which to run the algorithm separately, eventually merging the resulting HSPs. The batching procedure is depicted in Fig. 2, where each subsequence is shown as an independent dotted square. Alignments are represented as black dots which contain sets of words and seeds along the *x*- and *y*-axis (query and reference sequence, respectively). This mechanism prevents the explosion in the number of seeds that occurs in other state-of-the-art algorithms (see Sect. 4.2) while also enabling to fully run the algorithm without requiring to offload data to secondary memory in the host when main memory is not sufficient.

In this line, the proposed implementation leverages the compute capabilities of GPUs by trading off storage space for time, therefore running every stage as a combination of host and device memory without requiring to offload data structures to secondary memory. In fact, only the initial loading of the input sequences into CPU memory and the final writing of alignments from CPU memory require disk operations.

However, using a batching procedure penalizes the performance of the execution with additional data transfers between the host and device even when pinned host-memory is used. Thus, batching implies multiple data transfers needing memory allocations and deallocations in each processing stage, as each stage has different memory requirements. In order to minimize the latter, a memory pool strategy is employed, *i.e.* all memory is allocated as a single large chunk at the initialization of the algorithm. This enables us to replace system allocations (which can be very slow due to memory fragmentation, context-switching, etc.) with pointers addressing variables throughout the large memory chunk. This allows to turn multiple memory allocations into simple arithmetic operations.

Another side effect of this procedure is that the set of words (and their sorting) belonging to a row or column are processed b times (being b the number of subsequences in the comparison). Notice that this does not occur for seeds nor HSPs, which are dependent on the words from both the query and the reference. Consequently, they are completely different at each subsequence since only the calculation of words of one of the sequences is repeated (depending on whether a query or reference stride is used). Although this is computationally redundant, the alternative of storing these words in host memory for its reutilization in next iterations does not yield significant speedup since costly I/O transfers through PCI bus are required. In addition, this solution also increases the requirements of pinned memory in the host.

Still, the size of the subsequences must be carefully chosen: if it is too small there will be a large number of data transfers and kernel launches with little work, which result in poor performance; on the contrary, larger sizes will risk spawning too many seeds at once that will not fit into the memory of the device. To overcome this limitation, we performed an empirical study regarding the balance between subsequence size, seed generation, input sequences and performance and included a trade-off parameter to control the subsequence size as a function of the memory of the device.

However, from a biological perspective, it is important to mention that dividing the original sequence into subsequences risks breaking HSPs into two parts, especially when these are close to the boundaries of the subsequence. A Monte Carlo [36] simulation was run to calculate the proportion of HSPs being affected by this mechanism in the sequence comparison between *Homo sapiens* chr X and *Mus musculus* chr X. In particular, the location and length of HSPs were modelled using a uniform and exponential distribution, respectively, each with the corresponding parameters as extracted from the sequence comparison (further details can be found in the Supplementary Material). The simulation resulted in an average of $1.14 * 10^{-3}\%$ HSPs being split in two as a consequence of the batching strategy. While this result cannot be generalized to all other sequence comparisons, it can be extrapolated that more distant sequences will potentially result in less split HSPs, whereas more closely related sequences will result in more. Nevertheless, a trade-off between accuracy and speed is already expected in the context of seed-and-extend algorithms as a result of utilizing exact words of size k as seeds for alignments.

3.3 Kernel implementation

In this section, each stage of the developed CUDA workflow is addressed and analysed in terms of performance metrics. In the case of kernels where sorting libraries were employed, a reasoned discussion is provided based on computational complexity. From an implementation perspective, two things should be noted:

1. The data structures and representation of words (up to $k = 32$), seeds and alignments within the device are:
    - Words, which are represented by tuples of 64- and 32-bit integers which encode the numerical hash value as calculated in Equation 1 and the position of the word in the sequence, respectively. Both are materialized as pairwise arrays.
    - Seeds, which are represented as a single 64-bit integer which encodes the two 32-bit positional values of the words that conform the seed (transformed into a main diagonal and a single positional value, see Sect. 3.3.4).
    - Alignments, which are coded as two 32-bit integers which act as offsets to the original seed value and are also implemented as arrays.
2. The execution configuration (particularly thread block size) can be changed depending on the CUDA device (see [37]) except for the seed extension kernel, where the number of block threads must match the size of the word k.

3.3.1 Words and reverse-complement kernel

The words kernel is in charge of computing the individual words that compose a subsequence, as well as assigning each of them a unique numerical value. A key-value pair is built for each possible overlapping word w_i in the subsequence, with the key corresponding to the numerical hash h_i and the value corresponding to the original position in the sequence (see Sect. 2.1). In particular, each thread in a thread block calculates the numerical value of an overlapping word w of size $k = 32$ (which corresponds to 32 nucleotides in a segment of 32 bytes) by adding the value of the nucleotide at position i times 4 to the power of i. This means that thread t_i in block b_j is in charge of calculating the 32-word starting at position $i + j * block_{size}$. Note that each word is consecutively overlapped by one byte and therefore shared memory is employed to cache $t_{threads} + k$ bytes per block prior to computing the hash. This is required or otherwise the one-byte-shift per thread will split transactions in two, thus reducing global load efficiency. The storing of both key and value per each word is fully coalesced as long as the number of threads per block is divisible by 4, since each thread will perform a consecutive global store instruction on aligned sections that will begin at a multiple of the number of threads per block.

The design of the words kernel is dictated by the way in which the input sequences are indexed. That is, whether the query and reference sequences are stored either as a char array or as a larger data type (*i.e.* 4 or 8-byte types). Using a 1-byte per nucleotide strategy results in balanced computation and memory bandwidth, whereas increasing bandwidth by fetching larger data types at once results in reduction of instructions issued per cycle due to stall by execution dependency.

Moreover, the encoding of multiple nucleotides into a single byte (*e.g.* two) is not considered since:

1. The actual nucleotide values of the words are only used twice throughout the pipeline, namely at the words kernel and at the seed extension kernel. Therefore the encoding and decoding would represent significant overhead for the limited use.
2. DNA sequences are typically stored as FASTA files which use the standard ASCII encoding of one byte per character. The changing of such encoding would have to be carried out somewhere, potentially at the host side in order to minimize PCI transfers.
3. Three bits would be required in order to represent the four nucleotides plus the letter `N` (*i.e.*, the unknown nucleotide), which removes nearly all advantages of a two-bit encoding.

Thus the best performance is achieved by (1) keeping the one-byte per nucleotide of the original encoding and (2) caching DNA segments into shared memory to prevent unaligned accesses from the overlapping indexing. This results in a coalescent kernel with 100% global load efficiency, 100% global store efficiency and almost full occupancy. Further improvement is achieved when looping over the current *k*-mer by replacing as many integer instructions (which create instruction dependencies) as possible with cached constants.

In a similar fashion, the reverse-complement kernel calculates the reverse-complementary strand of the reference sequence, which also needs to be compared against the query input. Nevertheless, as opposed to the words kernel, there is no "walking pattern" in the reverse complement kernel, since threads in a block access consecutive memory segments (and not overlapping ones). Therefore, the kernel will achieve 100% efficiency in both the load and the global storage without requiring shared memory caching. In combination with asynchronous streaming transfers and a pinned-memory layout, the reverse-complement kernel is able to modestly improve acceleration depending on the size of the sequences while freeing up the host CPU and also compensating for the host-to-device transfers.

3.3.2 Sorting words

Sorting is a key aspect in the original GECKO algorithm, as well as one of the most time-consuming steps. The sorting of keys is used twice per execution (for words and seeds) to enable both the generation of seeds and the filtering of these in a prior step to their extension. The fact that GECKO and GPUGECKO require massive sorts (particularly in the case of sorting seeds) represents a strong advantage since sorting procedures are one of the most widely researched GPU algorithms in existence. This implies that GPUGECKO can always be revisited and updated with faster sorting methods, thus improving performance.

In the present case, the sorting method to choose must be able to deal efficiently with (1) millions of elements at once and (2) long numbers. Regarding (1), our

Table 3 Sequences included in the sample dataset, sorted by search space (i.e., the product of the lengths)

Sequences	Query len.	Ref. len.	Search sp.	Seeds
Myco. hyop. 232, 7422	892,725	898,121	801×10^9	0.4×10^6
Esch. coli B, K12	4,478,925	4,558,627	$20,417 \times 10^9$	3.6×10^6
GALGA18, MELGA20	10,494,880	9,898,897	$103,887 \times 10^9$	0.6×10^6
ORYLA6, DANRE25	23,318,240	37,418,471	$872,532 \times 10^9$	71×10^6
SUSSC11, BOSTA12	79,119,489	90,363,562	$7,149,518 \times 10^9$	12×10^6
HOMSAX, MUSMUX	151,099,878	163,484,862	$24,702,542 \times 10^9$	730×10^6
HOMSA1, GORGO1	225,279,443	211,487,502	$47,643,786 \times 10^9$	$3,136 \times 10^6$

The number of seeds corresponds to the total number of seeds generated using a word size of $k = 32$. Note that the names of the mammalian genomes (*i.e.* from the third row on) have been abbreviated with the first three and two letters of the genus and species, respectively, and the number of chromosomes

scenario comprises large sequences (such as mammalian chromosomes) which will typically require sorting from 10^6 to 10^8 pairs (see Table 3 in Sect. 4.1) when using a subsequence batching procedure as described previously. Regarding (2), each element being sorted is either the numerical value of the word (in the range $[0, 2^{64})$) or the seed-diagonal value (in the range $[0, 2^{32})$). These characteristics are appropriate for the merge sort algorithm [38], since its complexity is only dependent on the number of items, particularly $\mathcal{O}(N * \log N)$, as opposed to radix sort [39], which is also dependent on the size of the keys ($\mathcal{O}(k * N)$ approximately, being k the size of the key). This implies that, since keys are 64-bit long in both sorting cases, more items than can be fitted in the memory of the device would be required in order for the complexity of radix sort to pay off.

Regarding implementation, the mergesort algorithm of the ModernGPU library [40] was used along with a custom memory allocator that reduces memory allocations. Such optimization was deemed necessary given the number of times that sorting is performed per each subsequence as a result of batching. Source code for the custom memory allocator is provided in the Supplementary Material.

3.3.3 Generation of seeds

Once words are sorted, seeds can be spawned by matching equal words. This procedure consists of sweeping both word collections (query and reference) while keeping two indices that are incremented depending on key equality. That is, the query index is incremented if the key is smaller than the reference key, and vice versa. Since these two dictionaries are sorted, the sweep is purely linear, *i.e.* each key is visited only once. Therefore, in order to make an efficient GPU implementation of the kernel, it is required to overcome challenges which are related to both the architecture (strict linear complexity of the CPU counter version) and the application domain (irregularity in the distributions). These include:

1. Load balancing. A mechanism must be considered to balance how many seeds are calculated per block, since this number is unpredictable and can vary quadratically between blocks.
2. Synchronization at the grid level. A lightweight mechanism is needed to coordinate the writing of seeds per block such that performance is unaffected (*e.g.* large, shared memory buffers would impact negatively on occupancy).
3. Synchronization at the block level. Threads belonging to a block also require orchestration to write data without interleaved gaps.
4. High overhead per little work. Each seed requires only a small amount of computation while still requiring data retrieval and storage.

The two main mechanisms involved in the seed-generating kernel, namely load balancing and synchronization at the thread and block level are depicted in Algorithm 1. Note that the abbreviations t_x, b_x and g_x are used for *threadIdx*$_x$, *blockIdx*$_x$ and *gridDim*$_x$, respectively.

Regarding load balancing, a simple function of the family $\frac{1}{x}$ is employed to distribute increasingly smaller sizes of work items (encoded as a range of sorted words to process) in order to prevent blocks from locking up entire streaming multiprocessors, while others are idling. This enables to achieve full and uniform utilization of the multiprocessors with virtually no overhead.

Regarding writing synchronization, two mechanisms are devised. The first one is aimed at dealing with the distribution of the average number of seeds, whereas the second one is devised to handle the distribution that accounts for a potentially quadratic number of seeds (see Sect. 2.1).

The first one is based on a static division of the memory space such that each block has a reserved memory space. While of limited size, this space enables blocks to write seeds without requiring inter-block communication. In fact, only intra-block synchronization (*i.e.* per thread) is required to write to this memory space. To do so, we take advantage of the fact that the word dictionaries are sorted: if thread t_i does not generate a seed, then we know that t_{i+j} does not generate a seed either. This means that after identifying the first matching thread t_s that does generate a seed, we can pinpoint the range of threads $t_s, t_{s+1}, ..., t_i$ that have generated seeds and therefore perform a consecutive write, which effectively translates not only in a coalesced write, but also on avoiding the insertion of gaps between writes (otherwise these gaps would quickly fill up the reserved space). From an implementation perspective, determining the first position t_s can be performed quickly by computing a shuffle-warp min reduction of the thread indices that produced seeds of the warp (see Line 10). In conjunction with another reduction (either as a sum of the number of seeds in the current warp, or as a max reduction of the thread indices), all threads are made aware of the positions where the writes will take place, thus avoiding the insertion of gaps between writes.

The second mechanism, which is concerned with the quadratic number of seeds, includes fewer but much larger and unowned memory spaces. Since these are not mapped to blocks, their access is controlled by an atomic counter such that when a block's dedicated memory space is exhausted, the atomic counter returns the next unowned memory space (see Line 14). Unless the kernel is governed by atomic operations, this does not represent a negative impact on performance, especially since these operations are only required rarely (in fact they occur less than a few hundred times per execution).

The workflow of the seed-generating kernel proceeds as follows:

1. Process the query dictionary into successive batches of increasingly smaller sizes. Each block is assigned a batch and begins by performing a binary search on the reference dictionary in order to find the first reference key which has a greater or equal value than that of the query batch.
2. The query keys (and positions) are cached into shared memory in warp-sized chunks which get coalesced into a single transaction (see Line 6). Cached query keys only require to be updated once the last query key is found to be smaller than the current reference key. The reference keys (and positions) are also cached into shared memory and updated in every matching iteration in chunks of 32 memory-aligned items.
3. Each block computes query keys one by one against consecutive warp-sized reference keys and writes the results to the corresponding memory space using shuffle min reduction. Depending on the outcome of the matching comparisons, the next query key is computed or the next block of keys is fetched.
4. Once each block has finished, seeds are compacted to remove the gaps corresponding to the in-between writing sections reserved to every block. Doing so decreases the number of elements to sort and facilitates the posterior filtering.

Algorithm 1 Pseudocode of the seed-generation kernel

```
1:  procedure GENERATE_SEEDS(Words $w_{xy}$, Array $m$, Atomic $lock$)
2:      $shared$ keys$_{xy}$[32]
3:      start$_x$, end$_x$ $\leftarrow$ ranges($b_x$, $g_x$)
4:      start$_y$ $\leftarrow$ bsearch(start$_x$, w$_y$)
5:      while start$_x$ $<$ end$_x$ do
6:          keys$_x$ $\leftarrow$ w$_x$[start$_x$ + $t_x$]
7:          do
8:              keys$_y$ $\leftarrow$ w$_y$[start$_y$]
9:              if keys$_x$[start$_x$] == keys$_y$[$t_x$] then $t_s \leftarrow t_x$, hit $\leftarrow$ 1 end if
10:             t$_s$ $\leftarrow$ shuffle_reduction(t$_s$, min())
11:             m[t$_s$ + $t_x$] $\leftarrow$ diagonal(keys$_x$, keys$_y$)
12:             hit $\leftarrow$ shuffle_reduction(hit, sum())
13:             accumulated $\leftarrow$ accumulated + hit
14:             if accumulated $>$ m$_{size}$ then
15:                 m $\leftarrow$ atomic_update(m, lock)
16:             end if
17:             start$_y$ $\leftarrow$ start$_y$ + 32
18:         while keys$_x$[start$_x$] $\leq$ keys$_y$[0]
19:         start$_x$ $\leftarrow$ start$_x$ + 1
20:     end while
21: end procedure
```

This procedure results in a kernel that achieves full coalescence and full shared memory efficiency. In fact, the limitation of the kernel is the stalling caused by execution dependency as a collateral effect of using the warp shuffle instructions in order to synchronize threads at the block level. Nevertheless, this approach is more efficient than using temporary shared memory to distribute indices, since arithmetic operations also need to be performed on these.

3.3.4 Sorting and filtering of seeds

In the case of words sorting, tuples of keys and values were sorted only by their numerical hash key. However, in the case of seeds, the sorting has to be done on both the diagonal value of the seed and its positional value in the query or reference sequence since doing otherwise prevents from filtering adjacent seeds and hence the removal of duplicates. State-of-the-art GPU sorting algorithms are not designed for multiple key sorting, but rather for one key and one value [41] or one key and multiple values [42]. In order to achieve the maximum performance, we propose to merge both the diagonal value and the positional value into a single 64-bit key. This approach has several advantages over a custom sorting implementation that handles multiple keys, particularly (1) the fastest state-of-the-art sorting algorithm can be used and revisited, as explained before, (2) the merging of the diagonal and positional value can be done in constant time when seeds are created and (3) no additional comparison overhead is added to the sorting algorithm, preventing performance degradation. In essence, the merging of diagonal and positional values into one key is possible because of the ternary relationship $d = x - y$, which produces a single 64-bit key. Moreover, this key combines the positional information of both query and reference while also being suitable to be sorted upon. Equation 3 shows the calculation of the merged key.

$$d_{merge} = d_{prev} * d_{\text{length}} + p_{\text{query}} \tag{3}$$

where d_{prev} is the original diagonal value calculated as $d_{prev} = p_{\text{query}} - p_{reference}$, d_{length} is the maximum length of diagonal calculated as $d_{\text{length}} = \max(q_{\text{length}}, r_{\text{length}})$, p_{query} and $p_{\text{reference}}$ are the positions of the seed in the query and reference sequences and $q_{\text{length}}, r_{\text{length}}$ are the lengths of the input sequences. Notice that Equation 3 separates the space into as many consecutive segments as the number of diagonals (which is $n_{\text{diagonals}} = q_{\text{length}} + r_{\text{length}}$), each of size d_{length} and thus fitting p_{query} such that order is conserved. Due to the key being 64 bits in size, the upper bound for the length of the input sequences (assuming $q_{\text{length}} = r_{\text{length}}$) is calculated as follows in Equation 4:

$$\begin{aligned} 2^{64} &= n_{\text{diagonals}} * d_{\text{length}} 2^{64} \\ &= (q_{\text{length}} + r_{\text{length}}) * \max q_{\text{length}}, r_{\text{length}} 2^{64} \\ &= 2 * q_{\text{length}} * q_{\text{length}} 2^{64} \\ &= 2 * q^2_{\text{length}} \sqrt{\frac{2^{64}}{2}} = q_{\text{length}} \end{aligned} \tag{4}$$

Which gives us an estimated upper bound of $q_{\text{length}} = 2,147,483,648 * \sqrt{2} \approx 3 * 10^9$. Given that there is an upper bound for the length of chromosomes [43] and that the estimated upper bound for the merged key is several times superior to the size of mammalian and plant chromosomes, this strategy can be used in any scenario. Moreover, since the sorting of seeds is applied to each subsequence from the batching procedure, it is rather unlikely that a GPU device has enough memory to store the seeds produced by splits of size 3 gigabase pairs at once. In short, the previous procedure enables to sort seeds (which require two keys to be sorted) at once without sacrificing functionality nor speedup.

After the sorting of seeds is completed, filtering needs to be performed. Only seeds whose positional value is sufficiently close (typically less than twice the size of the seed) to a preceding seed in the same diagonal can be filtered out. Such filtering can be seen as a stream compaction operation in which the seeds that do not satisfy the given condition of proximity and diagonal value are first marked for removal and then followed by a scan and scatter operation.

3.3.5 Seed-extension

Each seed can be extended into an arbitrary sized HSP. Thus, a GPU kernel running more than one seed per block easily results in branch divergence (for the size range of HSPs can vary several orders of magnitude, typically from 100 bp to over 40,000 bp), and even worse, in a thread block being held on a SM for as long as the largest HSP requires. Such behaviour derives in blocks locking a higher amount of resources and therefore in increasing latency and degrading occupancy and performance. In this line, GPUGECKO proposes a one-block-per-seed kernel where each block runs 32 threads in complete lock step, aligning contiguous 32-byte sections of DNA first in the forward direction and then in the reverse direction.

Algorithm 2 Pseudocode of the seed-extending kernel for the forward direction

```
1: procedure EXTEND_SEEDS(Seed $seed$, Sequence $s_1$, Sequence $s_2$)
2:     score $\leftarrow 32$
3:     while score $> 0$ do
4:         x $\leftarrow seed_x - 32 + threadIdx_x$
5:         y $\leftarrow seed_y - 32 + threadIdx_x$
6:         sim $\leftarrow s_{1,x} == s_{2,y}$
7:         for offset $= 16$; offset $> 0$; offset $\div 2$; do
8:             sim $\leftarrow sim +$ shuffle_exchange($sim$, $offset$)
9:         end for
10:        sim $\leftarrow$ shuffle_exchange($sim$, $0$);
11:        score $\leftarrow score + sim - (32 - sim)$
12:        seed $\leftarrow seed - 32$
13:    end while
14: end procedure
```

Algorithm 2 shows the pseudocode for the seed-extending kernel. Note that for illustration purposes, boundary checks for beginning and ending of the sequences are not shown. As can be seen, the kernel begins computation by fixing the score of the current seed at 32, which corresponds to the 32 matching nucleotides of the two equal words. Then, until the score drops below zero, the kernel keeps aligning consecutive segments of 32 nucleotides (or 32 bytes). This is done in parallel by all of the 32 threads of the block: each thread compares one nucleotide addressed by the query position $\text{seed}_x - 32 + \mathit{threadIdx}_x$ and the reference position $\text{seed}_y - 32 + \mathit{threadIdx}_x$ (note that when computing the opposite direction, the value 32 is added instead of subtracted). The values seed_x and seed_y correspond to the origin position of the seed in both sequences. The threadIdx_x value, which contains the id of the current thread, is added to the computation so that each thread calculates a consecutive nucleotide. Next, the outcome of the nucleotide comparison (per thread) is stored in variable *sim* either as a 1 or a 0. This enables to employ a sum and reduction approach where all threads participate in an iterative summation and a final exchange of the number of matched nucleotides per segment of 32 nucleotides (line 10). Last, the score and the current position of the alignment (in respect to the seed) are updated in order to fetch another segment of 32 nucleotides. Also notice that no branch divergence can occur in this scenario, and that sequence access by each thread is consecutive, hence resulting in almost fully coalesced access. Every warp's access to the sequence will require on average two transactions, as opposed to a fully coalesced access with only one. This is due to the memory alignment of the sequence, which is stored as a char array, and therefore seeds can start at any position in the sequence, sometimes spanning across two memory segments and hence requiring two transactions. Regarding the performance of the kernel, the most limiting factor is the number of arithmetic operations performed on every segment of 32 nucleotides (*i.e.* 32 bytes), which result in execution dependency being the most limiting factor. Therefore, further optimizations in the kernel should target computation (and particularly instruction-level parallelism) as opposed to maximizing memory bandwidth.

Still, besides the optimizations for the actual computation of the seed extension, an algorithmic optimization is also included in the kernel. A large deal of seeds might be spaced throughout the same diagonal at a distance which is larger than the maximum gap to filter them out (seeds closer than twice the length of a seed can be filtered without losing sensitivity). In a parallel execution, this can result in recomputing most of those seeds, since no synchronization mechanism is foreseen. Since block synchronization is a costly mechanism, we overcome this scenario by including a static partitioning of the seeds, *i.e.* each block will compute not only one but several adjacent seeds, enabling it to detect whether a particular seed overlaps with the following seed without requiring inter-block communication. In short, the addition of the partitioning mechanism along with a shared-memory buffering prior to storage in global memory can accelerate the seed-extension kernel multi-fold (up to 14x in the case of *Escherichia coli*) depending on the amount of closely spaced seeds, compared to the default full-recomputation approach. In theory, further improvement can be achieved by increasing the number of seeds per block; however, this also results in less occupancy in cases where no such "repetitive seeds" exist. Thus, the seeds per block were balanced experimentally among all comparisons and set to 32.

3.4 CUDA multi-process service

As discussed in Sects. 1 and 2, the seed-and-extend algorithm is affected mainly by three sources of irregularity: (1) huge input sequences, (2) unpredictable and quadratic generation of seeds and (3) arbitrary length of alignments. This implies that the performance of each kernel is bounded by different irregularity sources, including memory, computation, host to device transfers, memory allocation overhead, etc. Moreover, if the size of the available memory grows linearly, the number of words that can be processed per subsequence grows linearly as well. However, with a linear increase in the number of words, seeds grow quadratically, which translates into a higher imbalance between kernels and a much bigger risk of seed explosion and consequent memory exhaustion (particularly because the number of seeds is not known). Besides such behaviour, and in order to enable both low-end and high-end GPUs to run GPUGECKO, the complete algorithm is optimized to utilize between 4 and 6 GB of device RAM. Nevertheless, a mechanism is also considered to take advantage of higher-end GPUs with larger memories, such as *e.g.* the RTX 3090 with up to 24 GB of RAM. This approach consists of executing whole comparisons in splits using the CUDA Multi-Process Service (CUDA MPS), which enables to run each subsequence in parallel within the same GPU. This is possible since (1) each subsequence execution is limited to around 5 GB of GPU RAM and thus several subsequences can be launched; and (2) the overlap between kernels that are bounded by computation, memory or other factors enables to achieve speedups close to the number of subsequences (see Sect. 4.4). Scripts are also provided in the official repository to automate the process of running a multi-split comparison within a CUDA MPS instance.

4 Results and discussion

In this section, the proposed method is tested in terms of speedup, performance and quality of results on different hardware devices. First, a comparison against the sequential CPU version of GECKO is provided on a per-kernel basis. Second, a comparison between the multi-core version of GECKO, the GPU accelerated version of BLASTN [44], namely GBLASTN [25], and GPUGECKO[2] is shown as state-of-the-art comparison. This experiment is also complemented with a discussion on the decisions regarding the hardware implementation differences of GBLASTN and GPUGECKO. Third, a hardware comparison is featured including the last four microarchitecture generations of CUDA devices. Last, an experiment regarding all-vs-all chromosome comparison of mammalian species is included to serve as use case. All experiments were set up with a minimum of 80% alignment identity.

[2] Source code is available at https://github.com/estebanpw/cudagecko.

4.1 Infrastructure and datasets

The following servers were employed:

1. The 980 GTX configuration includes 2x Intel Xeon E5-2698 v3 processor at 2.3 GHz, 256 GB of RAM and 4x NVIDIA GeForce GTX 980 with 4 GB of GDDR5 RAM.
2. The 1080 Ti configuration includes 1x Intel Xeon E5-2609v4 at 1.7 GHz, 32 GB of RAM and 2x NVIDIA GeForce GTX 1080 Ti with 11 GB of GDDR5X RAM.
3. The 2080 Ti configuration includes 1x Intel Xeon E5-2630V4 at 2.2 GHz, 64 GB of RAM and 2x NVIDIA GeForce RTX 2080 Ti with 11 GB of GDDR6 RAM.
4. The 3090 RTX configuration includes 1x Intel Xeon E5-2609v4 at 1.7 GHz, 64 GB of RAM and 1x NVIDIA GeForce RTX 3090 with 24 GB of GDDR6X RAM.

Each test included in advance was executed 10 times and averaged. Unless indicated otherwise, all executions were performed in the 980 GTX configuration server.

The dataset employed in the comparisons contains seven pairs of sequences (totalling 14 sequences) from different species and of increasing size (in particular, bacteria and mammalian chromosomes). Table 3 shows the pairs of sequences along with the length, search space and number of seeds per comparison at $k = 32$. In order to enable reproducibility, reference numbers of each sequence were included in the Supplementary Material.

4.2 State-of-the-art comparison

In this section, the performance and quality of results of GPUGECKO are compared to both GECKO and the CUDA implementation of the well-known algorithm BLASTN (GBLASTN). GECKO was included to serve as baseline in regard to the GPU methods and also due to its competitiveness for small sequences, as it will be shown. Note that aligners based on dynamic programming (such as CUDALIGN) are not included in the comparison due to the unfair difference in runtime (for example, the optimal alignment of chromosome 22 of *Homo sapiens* and *Gorilla gorilla* can take over 26,000 seconds using 3 GPUs [17], whereas the heuristic seed-and-extend algorithms require less than a hundred).

Depending on the goal of the sequence comparison experiment, researchers might be interested in two scenarios: (1) an exhaustive comparison where no heuristic seed filtering is employed (besides the default provided by seed-and-extend methods) and (2) a heuristic comparison where each method applies its default filtering technique. In this section, experiments for both scenarios are performed. Note that filtering heuristics are usually based on partially discarding words that are very frequent in order to avoid seed explosion. However, this can result in less sensitivity, especially in segments containing many repetitions and DNA duplications. Last, the

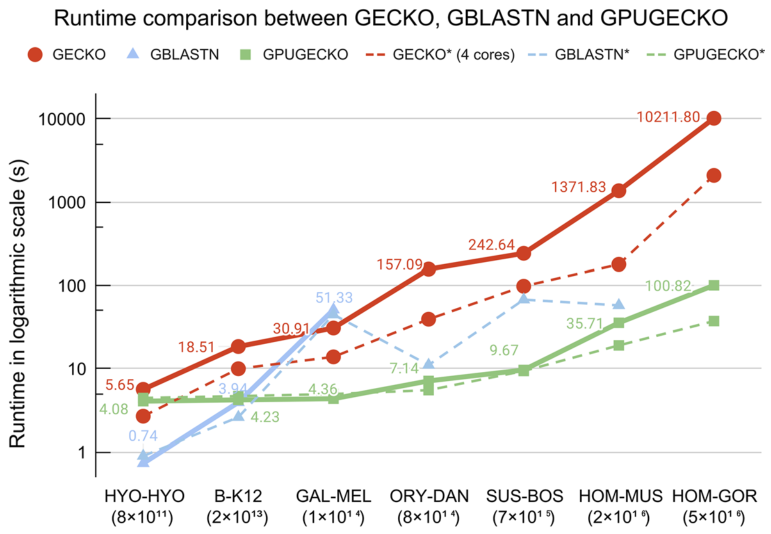

Fig. 3 Runtime comparison plot between GECKO, GBLASTN and GPUGECKO in both modes (exhaustive and seed-filtering). The solid lines represent the exhaustive executions (no filtering), whereas the dotted lines include the heuristic filtering of seeds and in the case of GECKO, multicore support. GECKO is shown in red with circle data points, GBLASTN is shown in blue with triangle data points, and GPUGECKO is shown in green with squared data points. The x-axis shows each comparison in the sample dataset along with the approximate search space in bp^2. The y-axis shows the runtime in seconds in $\log_{10}$ scale

standalone version of the CPU-based GECKO was executed single-threaded in the exhaustive comparison (as baseline) and with up to four cores in the heuristic comparison, which is the standard setting when running GECKO. Note that the multicore support follows the original task-parallelism approach as provided in the original implementation, which in average uses between three to four cores at maximum.

The experiments are described in terms of runtime, speedup and sensitivity of the alignments. The reported local alignments were restricted to a minimum of 80% of identity calculated as $n_{\text{idents}} \div \text{alignment}_{\text{length}}$, and the coverage was calculated as the percentage of unique base pairs from the query identified in the alignments to the reference ($n_{\text{idents}} \div \text{query}_{\text{length}}$), *i.e.* the proportion of the query found in the reference. It should be mentioned that this definition of coverage is not the one from genetic sequencing but rather from sequence alignment (see *e.g.* [23]).

Figure 3 shows the runtime for each of the methods (namely GECKO, GBLASTN and GPUGECKO) depending on whether they are run exhaustively (solid lines) or with heuristic seed-filtering techniques (dotted lines). The same runtimes along with coverage values are also shown in Tables 4 and 5.

On the one hand, in the exhaustive mode, GBLASTN is the fastest for the smaller sequences (particularly bacteria), but it is not able to process the larger comparisons where the number of generated seeds does not fit entirely into global memory (thus the solid line corresponding to GBLASTN is only present for the three smallest sequences). Notice that the nearly constant runtime of GPUGECKO in the smaller sequences is due to its worst-case memory allocation policy, which prevents

Table 4 Exhaustive sequence comparison between GECKO, GBLASTN and GPUGECKO

Exhaustive run	GECKO (sequential)		GBLASTN		GPUGECKO	
Comparison	T (s)	Cov. (%)	T (s)	Cov. (%)	T (s)	Cov. (%)
HYO-HYO (8×10^{11})	5.65	90.10	**0.73**	92.80	4.08	**94.93**
B-K12 (2×10^{13})	18.51	93.14	**3.94**	91.08	4.23	**94.66**
GAL-MEL ($1\times^{14}$)	30.91	49.85	51.32	58.53	**4.36**	**59.61**
ORY-DAN (8×10^{14})	157.09	0.23	DNF	DNF	**7.14**	**0.25**
SUS-BOS (7×10^{15})	242.64	5.59	DNF	DNF	**9.67**	**5.67**
HOM-MUS (2×10^{16})	1,371.83	0.99	DNF	DNF	**35.71**	**1.02**
HOM-GOR (5×10^{16})	10,211.80	66.31	DNF	DNF	**100.82**	**69.98**

The first column comprises the abbreviation of the sequence comparison. The following three column pairs represent the runtime (averaged 10 times) as well as the coverage reported as a percentage over the query sequence assuming a minimum of 80% similarity of the alignments for each of the methods. The abbreviation "DNF" stands for "Did Not Finish"

memory exhaustion from quadratic seed explosion (more details in Sect. 4.3) at the expense of an initialization penalty factor. On the other hand, as sequences become larger, GPUGECKO outperforms GBLASTN even when comparing its exhaustive mode against the heuristic mode of GBLASTN (dotted lines in Fig. 3, meaning that higher alignment coverage can be achieved while also lowering the runtime).

When comparing GECKO and GPUGECKO, it can be observed that GECKO obtains close runtimes to GPUGECKO for the smallest sequence comparison. But as longer sequences are compared, both the efficient exploitation of the massive parallelism available in the GPU and the drastic reduction of I/O transfers enable to obtain high speedups in favour of GPUGECKO. A per-kernel comparison between the sequential CPU and the accelerated GPU version is available in the Supplementary Material.

Table 5 Non-exhaustive sequence comparison employing seed-filtering techniques between GECKO, GBLASTN and GPUGECKO

Heuristic run	GECKO (4 cores)		GBLASTN		GPUGECKO	
Comparison	T (s)	Cov. (%)	T (s)	Cov. (%)	T (s)	Cov. (%)
HYO-HYO (8×10^{11})	2.69	90.10	**0.90**	92.22	3.59	**94.91**
B-K12 (2×10^{13})	10.01	93.14	**2.61**	91.06	3.75	**94.65**
GAL-MEL (1×10^{14})	13.90	49.82	44.79	58.23	**4.01**	**59.29**
ORY-DAN (8×10^{14})	39.44	0.21	11.09	0.15	**4.72**	**0.22**
SUS-BOS (7×10^{15})	98.18	5.48	67.78	**5.59**	**8.29**	5.00
HOM-MUS (2×10^{16})	179.10	0.96	57.88	0.87	**16.58**	**0.98**
HOM-GOR (5×10^{16})	2,098.03	66.29	DNF	DNF	**41.52**	**69.66**

The first column comprises the abbreviation of the sequence comparison. The following three column pairs represent the runtime (averaged 10 times) as well as the coverage reported as a percentage over the query sequence assuming a minimum of 80% similarity of the alignments for each of the methods. The abbreviation "DNF" stands for "Did Not Finish".

In terms of coverage (the higher, the better, see Table 4), GPUGECKO provides the most alignments at the same level of similarity (80% of identity), whereas GECKO and GBLASTN follow closely behind. The small differences observed are because GPUGECKO is prepared to compute extremely repeating seeds, whereas other methods have internal limits that aim to prevent memory exhaustion. Note that overall low coverage values do not imply a lack in sensitivity of seed-and-extend algorithms, but rather a combination of biological factors, such as sequence relatedness, evolutionary pressure, number of shared genes. The validity of seed-and-extend methods for sequence alignment has been studied numerous times (*e.g.* see [45–47]).

When heuristic seed filtering is enabled (dotted lines in Fig. 3 and Table 5), GPUGECKO follows the same speedup trend when compared to GECKO, ranging between 1 and 2 orders of magnitude (up to 49x, except for the first comparison). Moreover, GBLASTN is able to compute all but the last comparison (which is the most computationally intensive one due to the high similarity of the sequences) and is again the fastest on the set of small bacteria sequences but loses performance as the size of the sequences grows. In this line, GPUGECKO achieves between 2x and 11x compared to GBLASTN in the rest of the dataset, with an average of 6x speedup. The coverage values show that each method detects between 1% to 10% less alignments when compared to the exhaustive versions. As in the previous case, GPUGECKO achieves the highest coverage in all cases with the exception of the *Sus scrofa* and *Bos taurus* comparison, where GBLASTN detects the highest number of alignments.

4.3 Implementation differences between GBLASTN and GPUGECKO

In order to understand the differences in runtime between GBLASTN and GPUGECKO, it is mandatory to discuss their implementation. Besides implementation decisions, there is also the algorithmic difference regarding subsequence batching: while GBLASTN attempts to run a whole comparison at once (which is faster in essence but results in exceeding the limitations of physical memory for medium- to large-sized sequences), GPUGECKO splits it into batches to take advantage of the computing power of the GPU. Moreover, as stated previously, GPUGECKO always performs a worst-case allocation and initialization (and subsequent deallocation) of the memory structures in the device and the pinned-memory in the host. This results in preventing potential buffer overflows from the number of generated seeds, although at the expense of a constant initialization penalty which is not amortized until larger sequences are employed. Regarding implementation, the first difference is the number of kernels that are implemented in the GPU, since GPUGECKO runs integrally in the graphics device, whereas in the case of GBLASTN, only three kernels are executed within the GPU, namely the reference scan, the table lookup and the mini-extension. These kernels correspond to the matching of words from the query to the reference (equivalent to generating the seeds) and their extension. However, notice that the generation of words (their hashing) is performed within the host, as opposed to GPUGECKO. Also note that the gapped extension procedure in GBLASTN is not implemented in a GPU kernel, nor is it analysed here since GPUGECKO does not produce gapped alignments either.

Regarding the matching of words, GBLASTN scans the reference in segments such that each thread maps a region of the reference and queries the lookup table for matches. At a first step, the matches are not extended nor saved, and the only thing recorded is whether the reference word has a match or not. This effectively avoids branch divergence, as all threads do nearly the same. Afterwards, each match will be computed knowing that all threads will have to perform extension, thus reducing branching divergence. However, it comes at the price of performing lots of global atomic operations (to record the words), which is further reduced by writing in batches. In the case of GPUGECKO, the matching of words is split into two steps, namely (1) sorting words by their hash first and (2) matching them linearly. As opposed to GBLASTN, where words are inserted into a table and then consulted, in GPUGECKO all words are first sorted to enable matching in near-constant time. As explained before, this presents the advantage of employing the merge sort algorithm in the device, which has received lots of improvements over time.

As a last step, the extension in GBLASTN is performed by giving each thread a different seed. This approach comes with several problems (as explained in Sect. 3.3.5), particularly branch divergence and block locking (a block will last in the SM as long as one thread has not finished extending resulting in increased latency). In GPUGECKO, this is avoided by running one block of 32 threads per seed in lockstep, thus completely removing both problems.

4.4 GPU hardware comparison

In this section, a runtime comparison between cards corresponding to the last generations of CUDA microarchitectures is presented, including Maxwell (980 GTX), Pascal (1080 Ti), Turing (2080 Ti) and Ampere (3090 RTX). The difference in number of CUDA cores ranges from 2,048 (980 GTX) to 10,496 (3090 RTX), while the difference in memory size ranges between 4 GB and 24 GB (other hardware metrics such as the memory type or bus width are not mentioned in the sake of clarity). The sample dataset was run using the CUDA MPS instance. Regarding the number of subsequences and memory usage, (1) the 980 GTX was run with only one subsequence of 4 GB of memory, (2) the 1080 Ti and 2080 Ti were run using two subsequences of 5 GB of memory each and (3) the 3090 RTX was run using 5 subsequences of 4.5 GB of memory each.

Figure 4 shows the runtime for all comparisons in the sample dataset for each of the tested GPU cards. Almost no performance gain is observed in the smaller comparisons, which is expected since not enough work is available. This is the case up until the *Homo sapiens* Chr. X and *Mus musculus* Chr. X comparison, where the difference in runtime becomes more evident, especially between Pascal and Turing generations (almost 2x). Moreover, in the case of the largest comparison, namely *Homo sapiens* Chr. 1 vs *Gorilla gorilla* Chr. 1, GPUGECKO achieves an average speedup of 1.55x from one card generation to the next one,

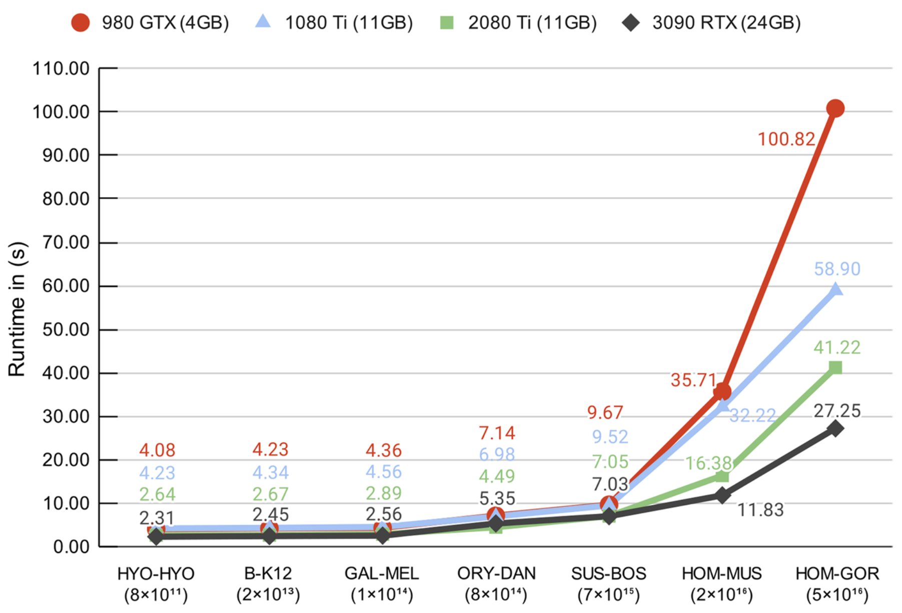

Fig. 4 Runtime comparison plot of GPUGECKO between different CUDA microarchitectures. The x-axis shows each comparison in the sample dataset along with the approximate search space. The y-axis shows the runtime in seconds

which is in fact in concordance with the average speedup reported across several device generations for a variety of algorithms [48]. Lastly, the overall achieved speedup between the 980 GTX and 3090 RTX is 3.69x, which is considerable given the differences in bandwidth and number of cores (4.17x and 5x higher, respectively).

4.5 All-versus-all chromosome comparison

In this section, we show how GPUGECKO can be used by researchers to perform large-scale research in comparative genomics. Using only one GeForce 980 GTX, we compared all chromosomes between Homo sapiens and Mus musculus, which accounts for 504 chromosome comparisons ranging in sizes from 47 megabase pairs to 242 megabase pairs and creating search spaces up to $4.5 * 10^{16}\ bp^2$. All executions were run without limiting the number of repetitive words nor seeds, thus exhaustively generating all alignments which contain at least one seed of size $k = 32$. The runtime for the complete experiment was 3 hours and 39 minutes, or an average of 26 seconds per pair of chromosomes comparison. A heatmap containing the coverage per comparison is available in the Supplementary Material. The same comparison is made in only 110 minutes (less than two hours, an average of 13 seconds per pair of chromosomes) using seed-skipping policies without sacrificing sensitivity significantly, see the

Supplementary Material for a comparison of coverage and runtime between the exhaustive and the seed-skipping mode.

5 Conclusions

In this manuscript, we have introduced GPUGECKO, a new CUDA-GPU algorithm for the irregular seed-and-extend sequence comparison method. On the one hand, from a computational and hardware acceleration standpoint, GPUGECKO overcomes several challenges that (to the best of our knowledge) have not been addressed before in any other GPU sequence comparison algorithm. These comprise:

1. Massive search space. GPUGECKO is able to divide the input search space into smaller comparisons to effectively cope with the irregular computing stages. We have shown how to implement this strategy and prevent its performance drawbacks (namely additional host-to-device transfer time penalties and increased work redundancy) by using custom memory pools and pinned memory, and providing a formula to balance the number of seeds per subsequence.
2. Complete GPU acceleration. While other approaches only deploy specific functions of the original algorithm as CUDA kernels, GPUGECKO features a complete GPU pipeline. This has resulted in a high speedup in regard to the sequential CPU version, and moreover, in a greater acceleration when compared to the GPU implementations of other software.
3. Overcoming the source of irregularity. We have proposed two new kernels that address the largest sources of irregularity, namely (1) the unpredictable and quadratic generation of seeds and (2) the extension of arbitrarily long alignments. These kernels are able to overcome both challenges by working in a cooperative fashion using parallel reductions while reducing branch divergence and block synchronization.

On the other hand, from a bioinformatics perspective, GPUGECKO is the first fully GPU-native algorithm capable of comparing huge DNA sequences (such as plant or mammalian chromosomes), typically requiring under a minute mark per chromosome comparison. We have shown that it is able to compare exhaustively any two chromosomes faster (up to 10x and an average of 6x in the exhaustive and non-exhaustive fashion compared to GBLASTN, respectively) while providing a higher degree of alignment coverage. When running with seed-skipping policies, GPUGECKO is also faster while still providing a higher level of alignment coverage in most of the cases. In the same line, GPUGECKO enables researchers to perform extremely large-scale sequence comparison experiments with full exhaustiveness, for instance being able to compare all chromosomes of two species (which accounts around 500 comparisons) in less than 4 hours using only one GPU. Previously, these experiments could only be carried out with dynamic-programming-based algorithms, which took up to 10 hours per each comparison and, more importantly, do not always represent the best approach to tackle a pairwise sequence comparison (such as in the case of global alignments which include long genome rearrangements).

Still, several lines of research remain open for future work in GPUGECKO. These include:

1. Multi-gpu computing support. Due to the subsequence batching procedure, each grid comparison can be run independently among any number of GPUs. While there is a huge potential for speedup, careful mechanisms must be devised to synchronize the fetching of subsequence data without resulting in PCI transfer stalls.
2. Perform gapped alignments. GPUGECKO can report alignments of a minimum of 32 bp, and thus a gap-including mechanism could help improve signal detection between closely spaced HSPs.
3. Automatic prediction of seeds. If the number of seeds is known prior to computation (or approximated) the factor used to separate memory spaces for words or seeds could be fine-tuned, resulting in larger speedups.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s11227-022-05007-z.

Author Contributions EPW conceived and designed the method, implemented the algorithm, performed the experiments, analysed and interpreted data and wrote the paper. OT analysed and interpreted data, contributed materials, analysis tools or data and wrote the paper. NG analysed and interpreted data, contributed materials, wrote the paper and supervised the project. All authors read and approved the final manuscript.

Funding Funding for open access publishing: Universidad Málaga/CBUA. This work has been partially supported by the European project ELIXIR-EXCELERATE (grant no. 676559), the Spanish national project Plataforma de Recursos Biomoleculares y Bioinformáticos (ISCIII-PT13.0001.0012 and ISCIII-PT17.0009.0022), the Fondo Europeo de Desarrollo Regional (UMA18-FEDERJA-156, UMA20-FEDERJA-059), the Junta de Andalucía (P18-FR-3130), the Instituto de Investigación Biomédica de Málaga IBIMA and the University of Málaga.

Data Availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

Declarations

Code availability All source code is available in the GitHub repository https://github.com/estebanpw/cudagecko.

Conflict of interest Not applicable.

Ethical approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

References

1. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell TJ: A survey of general-purpose computation on graphics hardware. In: Computer Graphics Forum, vol. 26, pp. 80–113 (2007). Wiley Online Library
2. Navarro CA, Hitschfeld-Kahler N, Mateu L (2014) A survey on parallel computing and its applications in data-parallel problems using gpu architectures. Commun Comput Phys 15(2):285–329
3. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al. (2016) Tensorflow: A system for large-scale machine learning. In: 12th symposium on operating systems design and implementation, vol 16, pp 265–283
4. Stone JE, Hardy DJ, Ufimtsev IS, Schulten K (2010) Gpu-accelerated molecular modeling coming of age. J Mol Gr Modell 29(2):116–125
5. Lu F, Song J, Cao X, Zhu X (2012) Cpu/gpu computing for long-wave radiation physics on large gpu clusters. Computers Geosci 41:47–55
6. Li Z, Wang Y, Zhi T, Chen T (2017) A survey of neural network accelerators. Front Computer Sci 11(5):746–761
7. Papangelopoulos N, Vlachakis D, Filntisi A, Fakourelis P, Papageorgiou L, Megalooikonomou V, Kossida S (2013) State-of-the-art gpgpu applications in bioinformatics. Int J Syst Biol Biomed Technol (IJSBBT) 2(4):24–48
8. Burtscher M, Nasre R, Pingali K (2012) A quantitative study of irregular programs on gpus. In: 2012 IEEE International Symposium on Workload Characterization (IISWC), pp. 141–151. IEEE
9. Hasan L, Al-Ars Z, Vassiliadis S (2007) Hardware acceleration of sequence alignment algorithms-an overview. In: 2007 International Conference on Design & Technology of Integrated Systems in Nanoscale Era, pp. 92–97 . IEEE
10. Li H, Homer N (2010) A survey of sequence alignment algorithms for next-generation sequencing. Briefings Bioinf 11(5):473–483
11. Aluru S, Jammula N (2013) A review of hardware acceleration for computational genomics. IEEE Des Test 31(1):19–30
12. Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, Satish N, Smelyanskiy M, Chennupaty S, Hammarlund P et al. (2010) Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, pp. 451–460
13. Benson DA, Cavanaugh M, Clark K, Karsch-Mizrachi I, Lipman DJ, Ostell J, Sayers EW (2016) Genbank. Nucleic Acids Res 45(D1):37–42
14. Koonin EV, Aravind L, Kondrashov AS (2000) The impact of comparative genomics on our understanding of evolution. Cell 101(6):573–576
15. Megquier K, Turner-Maier J, Swofford R, Kim J-H, Sarver AL, Wang C, Sakthikumar S, Johnson J, Koltookian M, Lewellen M et al. (2019) Comparative genomics reveals shared mutational landscape in canine hemangiosarcoma and human angiosarcoma. Mol Cancer Res 17(12):2410–2421
16. Fakirah M, Shehab MA, Jararweh Y, Al-Ayyoub M (2015) Accelerating needleman-wunsch global alignment algorithm with gpus. In: 2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA), pp. 1–5. IEEE
17. de Oliveira Sandes EF, Miranda G, Martorell X, Ayguade E, Teodoro G, Melo ACM (2016) Cudalign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in gpu clusters. IEEE Trans Parallel Distrib Syst 27(10):2838–2850

18. Korpar M, Šikić M (2013) Sw#-gpu-enabled exact alignments on genome scale. Bioinformatics 29(19):2494–2495
19. Pérez-Serrano J, Sandes E, de Melo ACMA, Ujaldón M (2018) Dna sequences alignment in multi-gpus: acceleration and energy payoff. BMC Bioinf 19(14):421
20. Vinga S (2014) Alignment-free methods in computational biology. Oxford University Press, Oxford
21. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. J Mol Biol 215(3):403–410
22. Pérez-Wohlfeil E, Diaz-del-Pino S, Trelles O (2019) Ultra-fast genome comparison for large-scale genomic experiments. Sci Rep 9(1):1–10
23. Chen Y, Ye W, Zhang Y, Xu Y (2015) High speed blastn: an accelerated megablast search tool. Nucleic Acids Res 43(16):7762–7768
24. Torreno O, Trelles O (2015) Breaking the computational barriers of pairwise genome comparison. BMC Bioinf 16(1):250
25. Zhao K, Chu X (2014) G-blastn: accelerating nucleotide alignment by graphics processors. Bioinformatics 30(10):1384–1391
26. Nvidia C (2011) Nvidia cuda c programming guide. Nvidia Corp 120(18):8
27. Official GPUGECKO repository. GitHub. Revision d473f03 on gpuhits branch. (2021)
28. Jokinen P, Tarhio J, Ukkonen E (1996) A comparison of approximate string matching algorithms. Softw Pract Exp 26(12):1439–1458
29. Horton R, Olsen M, Roe G et al. (2010) Something borrowed: sequence alignment and the identification of similar passages in large text collections
30. Melsted P, Pritchard JK (2011) Efficient counting of k-mers in dna sequences using a bloom filter. BMC Bioinf 12(1):1–7
31. Chockalingam SP, Pannu J, Hooshmand S, Thankachan SV, Aluru S (2020) An alignment-free heuristic for fast sequence comparisons with applications to phylogeny reconstruction. BMC Bioinf 21(6):1–12
32. Langmead B, Trapnell C, Pop M, Salzberg SL (2009) Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biol 10(3):1–10
33. Morgulis A, Gertz EM, Schäffer AA, Agarwala R (2006) A fast and symmetric dust implementation to mask low-complexity dna sequences. J Comput Biol 13(5):1028–1040
34. Pearson WR, Lipman DJ (1988) Improved tools for biological sequence comparison. Proc Nat Acad Sci 85(8):2444–2448
35. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, pp. 483–485
36. Mooney CZ (1997) Monte Carlo simulation, vol 116. Sage publications, London
37. Torres Y, Gonzalez-Escribano A, Llanos DR (2013) ubench: exposing the impact of cuda block geometry in terms of performance. J Supercomput 65(3):1150–1163
38. Cole R (1988) Parallel merge sort. SIAM J Comput 17(4):770–785
39. McIlroy PM, Bostic K, McIlroy MD (1993) Engineering radix sort. Comput Syst 6(1):5–27
40. Official ModernGPU repository. https://github.com/moderngpu/moderngpu. Revision 2b39855 on master branch
41. Arkhipov DI, Wu D, Li K, Regan AC (2017) Sorting with gpus: A survey. http://arxiv.org/abs/1709.02520
42. Bandyopadhyay S, Sahni S (2010) Grs–gpu radix sort for multifield records. In: 2010 International Conference on High Performance Computing, pp. 1–10 . IEEE
43. Schubert I, Oud J (1997) There is an upper limit of chromosome size for normal development of an organism. Cell 88(4):515–520
44. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. J Mol Biol 215(3):403–410
45. Kurtz S, Phillippy A, Delcher AL, Smoot M, Shumway M, Antonescu C, Salzberg SL (2004) Versatile and open software for comparing large genomes. Genome Biol 5(2):1–9
46. Krumsiek J, Arnold R, Rattei T (2007) Gepard: a rapid and sensitive tool for creating dotplots on genome scale. Bioinformatics 23(8):1026–1028

47. Ye J, McGinnis S, Madden TL (2006) Blast: improvements for better sequence analysis. Nucleic acids research 34(suppl_2), 6–9
48. Svedin M, Chien SW, Chikafa G, Jansson N, Podobas A (2021) Benchmarking the nvidia gpu lineage: From early k80 to modern a100 with asynchronous memory transfers. In: Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, pp. 1–6