J Sign Process Syst (2014) 77:131–149
DOI 10.1007/s11265-014-0883-2

Large-Scale Pairwise Alignments on GPU Clusters: Exploring the Implementation Space

Huan Truong · Da Li · Kittisak Sajjapongse · Gavin Conant · Michela Becchi

Received: 9 September 2013 / Revised: 25 January 2014 / Accepted: 14 March 2014 / Published online: 1 April 2014

Abstract Several problems in computational biology require the all-against-all pairwise comparisons of tens of thousands of individual biological sequences. Each such comparison can be performed with the well-known Needleman-Wunsch alignment algorithm. However, with the rapid growth of biological databases, performing all possible comparisons with this algorithm in serial becomes extremely time-consuming. The massive computational power of graphics processing units (GPUs) makes them an appealing choice for accelerating these computations. As such, CPU-GPU clusters can enable all-against-all comparisons on large datasets. In this work, we present four GPU implementations for large-scale pairwise sequence alignment: *TiledDScan-mNW*, *DScan-mNW*, *RScan-mNW* and *LazyRScan-mNW*. The proposed GPU kernels exhibit different parallelization patterns: we discuss how each parallelization strategy affects the memory accesses and the utilization of the underlying GPU hardware. We evaluate our implementations on a variety of low- and high-end GPUs with different compute capabilities. Our results show that all the proposed solutions outperform the existing open-source implementation from the Rodinia Benchmark Suite, and *LazyRScan-mNW* is the preferred solution for applications that require performing the traceback operation only on a subset of the considered sequence pairs (for example, the pairs whose alignment score exceeds a predefined threshold). Finally, we discuss the integration of the proposed GPU kernels into a hybrid MPI-CUDA framework for deployment on CPU-GPU clusters. In particular, our proposed distributed design targets both homogeneous and heterogeneous clusters with nodes that differ amongst themselves in their hardware configuration.

Keywords Heterogeneous system · Sequence alignment · GPU

H. Truong · G. Conant · M. Becchi (✉)
MU Informatics Institute, University of Missouri, Columbia, USA
e-mail: becchim@missouri.edu

H. Truong
e-mail: Huantruong@mail.missouri.edu

G. Conant
e-mail: conantg@missouri.edu

D. Li · K. Sajjapongse · M. Becchi
Department of Electrical and Computer Engineering, University of Missouri, Columbia, USA

D. Li
e-mail: da.li@mail.missouri.edu

K. Sajjapongse
e-mail: kittisak.sajjapongse@mail.missouri.edu

G. Conant
Division of Animal Sciences, University of Missouri, Columbia, USA

1 Introduction

The pairwise sequence alignment algorithms, both local and global [1, 2], are in many ways the core technology for the study of biological sequences. They have key roles in multiple sequence alignment [3], phylogenetics [4], and molecular evolution studies [5]. In addition, heuristic improvements to the basic dynamic programming approach are essential features of sequence database search programs such as FASTA [6] and BLAST [7, 8] and various forms of genome assembly algorithms [9, 10]. At some level, these later approaches are also related to more general approximate string matching algorithms [11]: the link being that both types of approach make assumptions about the maximum number of acceptable mismatches between the sequences under consideration. Acceleration of sequence searches is useful because, while the dynamic programming approach to alignment is only $O(n^2)$ in time complexity, biologists often wish to make millions or

even billions of such comparisons [12]. However, these heuristics depend on the assumption that the vast majority of the sequence pairs being compared have essentially no similarity and that, once this fact has been demonstrated for a sequence pair, the computation of the alignment itself is unnecessary.

Increasingly a second class of problem is becoming relevant. In this case, there is a requirement to compare very large numbers of sequences that are all evolutionarily related. As a result, it is not possible to omit the computation of any of the alignments, making approaches such as that of BLAST inappropriate. One example is the computation of very large multiple sequence alignments for analyses such as inference of the "tree of life" [13–15]. A similar problem motivates our work here, namely the analysis of complex microbial communities through the sequencing of a particular microbial gene, the 16S rDNA gene. Biologists have discovered that many microbes cannot be cultured under laboratory conditions but that it is possible to assess their presence through the direct sequencing of the DNA in an environment [16–20]. To compare microbial communities across environments, it is helpful to survey a single gene: the 16S gene is useful in this regard as it is essentially ubiquitous across prokaryotic life. However, the sequencing of the gene is only a first step: it is then necessary to compare the sequences generated to each other and to other known 16S sequences to assess the taxonomic diversity present in the sample. As there are hundreds of thousands of 16S sequences in sequence databases and tens of thousands of unique sequences among those [21], this analysis can be daunting.

The problem as stated is clearly highly parallel, and, as such, we sought to bring the massively parallel computing potential of GPUs to bear on it. General-purpose graphics processing units (GPGPUs) are advancements of hardware originally developed to accelerate complex graphical rendering for applications like 3D gaming. These devices can be programmed in several ways, including the CUDA framework proposed by Nvidia. The design of parallel kernels for GPU directly affects the utilization of the underlying hardware: specifically its compute cores and memory hierarchy. This usage, in turn, influences the performance achieved.

Our contributions can be summarized as follows.

- We propose four implementations of multiple pairwise alignments using the Needleman-Wunsch (NW) algorithm on GPU. Three of our parallel kernels (*TiledDScan-mNW*, *DScan-mNW* and *RScan-mNW*) are general purpose. Our forth implementation (*LazyRScan-mNW)* is optimized for problems that require performing the trace-back operation only on a subset of the sequence pairs in the initial dataset (for example, the pairs whose alignment score exceeds a predefined threshold).
- The methods considered differ in their computational patterns, their use of the available hardware parallelism, and their handling of the data dependences intrinsic in NW. Our analyses give insights into the architectural benefits and costs of using GPUs for bioinformatics, insights likely applicable to other domains.
- We integrate our general purpose GPU kernels with an MPI framework for deployment on CPU-GPU clusters. Our distributed solution targets both homogeneous and heterogeneous clusters, the latter having nodes with different hardware configurations and compute capabilities.
- We evaluate our framework on a dataset of about 25,000 unique 16S rDNA genes from the Ribosomal Database [21]. We use a variety of Nvidia GPUs: from the low-end Quadro 2000 to the high-end Tesla K20. We show that our optimizations are effective on all of the considered devices. Our experiments based on the general purpose *TiledDScan-mNW*, *DScan-mNW* and *RScan-mNW* kernels show a throughput in the order of 250 and 330 pairwise alignments/s on low- and high-end GPUs, respectively. In addition, we achieve a throughput of 1,015 pairwise alignments/s on a 6-node commodity cluster equipped with a low-end GPU on each node. Finally, our *LazyRScan-mNW* kernel allows throughputs up to about 4,000 and 6,000 pairwise alignments/s on low- and high-end GPUs, respectively.

Our GPU implementations can be freely downloaded at: http://nps.missouri.edu/nps_wiki/index.php/Code.

2 Related Work

In recent years GPUs and other accelerator devices have been widely used to accelerate a variety of scientific applications from many domains [22–24]. In particular, a number of biological applications, including BLAST [25], hidden Markov models [26, 27] and structure comparisons [28], have been ported to GPU or FPGA architectures. Most relevant to our work are several sequence alignment algorithms implemented on GPU [23, 29–31]. In Section 3, we will provide more background on one of these: the NW implementation in the Rodinia benchmark suite [23]. As an aside, we note that one can also exploit bit-level parallelism to modestly accelerate alignment algorithms [11]: unfortunately, such approaches require assumptions about the scoring matrix and gap costs used that would restrict the general applicability.

Among the alignment implementations, Liu et al., [32] present an optimized sequence database search tool based on the Smith-Waterman (SW) *local* alignment algorithm (in contrast to the NW *global* alignment problem considered here). Compared to other implementations [29, 33, 34], their tool provides better performance guarantees for protein database searches. Li et al., [35] offer a GPU acceleration of SW intended for a single comparison of two very long sequences;

we focus on accelerating many pairwise alignments of shorter sequences. We are interested in the NW problem, which rather than being used for database search is more commonly applied to situations where all possible pairwise alignments are required (e.g., alignments for phylogenetics or metagenomics as described above). In their first phase (computation of the alignment matrix), NW and SW share similar computation patterns, so optimization techniques can be reused between the two methods.

There are also distributed CPU-based implementations of NW: for example ClustalW-MPI [36] aligns multiple protein, RNA or DNA sequences in parallel using MPI. Biegert et al., [37] have introduced a more general MPI bioinformatics toolkit in the form of an interactive web service that supports searches, multiple alignments and structure prediction. Our tool differs from these in *combining* MPI and CUDA to allow deployment on CPU/GPU clusters where multiple GPUs may be employed simultaneously.

3 Background

3.1 Analysis of the Needleman-Wunsch Algorithm

The goal of the Needleman-Wunsch algorithm (NW) is to find the alignment of two strings (generally protein or DNA) that maximizes a cost function. That cost function consists of two parts. The first is a match and mismatch scoring matrix that gives the cost of aligning matching or mismatching sequence elements (hereafter $S(x_i,y_j)$). For DNA alignments, simple schemes such as rewarding matches (+4) and penalizing mismatches (−5) are often used. For protein alignments, it is more common to use an empirical scoring matrix [e.g., BLOSUM; 38]. The second part of the function is a cost for "gaps:" i.e., regions of one sequence not aligned against regions of the other. Here, we will apply a linear gap cost G. As input data, NW takes two sequences of length m and n. The optimal alignment is then computed within a 2-D matrix M of size $(m+1)\times(n+1)$. Note that this matrix can be virtual: there are linear space memory implementations of the algorithm (e.g. Hirschberg's algorithm [39]). Each element in M is then computed according to equation (1).

$$M(i,j) = \max \begin{cases} M(i-1,j-1) + S(x_i,y_j) \\ M(i-1,j) + G \\ M(i,j-1) + G \end{cases} \quad (1)$$

Here, $M(i,j)$ is the alignment score in the i^{th} row and j^{th} column of M. The first row and column of M are initialized as gaps of increasing length [1]: once this initialization is complete, the remaining positions can be computed given the values above them, to their left and to their left diagonal.

It is apparent from this description that the memory and computing requirements of a naïve implementation of the algorithm can be significant, as they scale as O(mn) (often spoken of as $O(n^2)$). For instance, in our experiments, we use database of roughly 25,000 unique 16S rDNA genes from the Ribosomal Database Project [21]. Performing all possible pairwise alignments involves roughly 300 million comparisons. Moreover, the computation itself is somewhat memory intensive: as equation (1) indicates, computing each new element in the alignment matrix requires three reads from memory and one write to store the new value. On the other hand, the computation is relatively trivial, requiring three additions and a comparison.

The NW algorithm can be broken into two phases: (1) the *computation of the alignment matrix M* (described above), and (2) the *trace-back* operation, which uses the alignment matrix to reconstruct the sequence alignment itself. Unless linear-space implementations of NW [39] are adopted, the trace-back is a linear-time operation accounting for a small fraction of the overall execution time. In Section 4, we focus on the computation of the alignment matrix.

3.2 Brief Introduction to Nvidia GPUs and CUDA

Nvidia GPUs comprise a set of Streaming Multiprocessors (SMs), where each SM in turn contains a set of simple in-order cores. These in-order cores execute instructions in a SIMD manner. GPUs have a heterogeneous memory organization consisting of high latency off-chip global memory, low latency read-only constant memory (which resides off-chip but is cached), low-latency on-chip read-write shared memory, and texture memory. GPUs adopting the Fermi and Kepler architecture, such as those used in this work, are also equipped with a two-level cache hierarchy. Judicious use of the memory hierarchy and of the available memory bandwidth is essential to achieve good performances. In particular, the utilization of the memory bandwidth can be optimized by performing regular access patterns to global memory. In this situation, distinct memory accesses are automatically *coalesced* into a single memory transaction, thus limiting the memory bandwidth used.

The advent of CUDA has greatly simplified the programmability of GPUs. With CUDA, the computation is organized in a hierarchical fashion, wherein *threads* are grouped into *thread-blocks*. Each thread-block is mapped onto a different SM, whereas different threads are mapped to simple cores and executed in SIMD units, called *warps*. The presence of control-flow divergence within warps can decrease the GPU utilization and badly affect the performance. Threads within the same block can communicate using shared memory, whereas threads within different thread-blocks are fully independent. Therefore, CUDA exposes to the programmer two degrees of parallelism: *fine-grained* parallelism within a

thread-block and *coarse-grained* parallelism across multiple thread-blocks.

3.3 Rodinia-NW: Needleman-Wunsch on GPU

The Rodinia benchmark suite [23] offers a GPU parallelization of NW (hereafter, *Rodinia-NW*), that we will use as baseline.

Rodinia-NW operates as follows. Since each element in the alignment matrix depends on its left-, upper- and left-upper-neighbors, a way to exploit parallelism is by processing the matrix in minor diagonal manner. Each minor diagonal depends on the previous one, thus leading to the need for iterating over minor diagonals. However, at each iteration, all the (independent) elements in the same minor diagonal line can be calculated simultaneously. If the matrix is laid out in global memory in row-major order, the involved memory access patterns are uncoalesced, potentially leading to performance degradation. Since each element in the alignment matrix is used for calculating three other elements, performance can be improved by leveraging shared memory and dividing the alignment matrix in square tiles (each of them fitting the shared memory capacity). *Rodinia-NW* performs tiling and exploits two levels of parallelism: (i) within each tile elements are processed in minor diagonal manner, and (ii) different tiles in the same minor diagonal line can also be processed concurrently by distinct thread-blocks. Threads within the same thread-block manipulate the data and store elements in shared memory temporarily. After the computation of a tile completes, all of the data are moved to global memory using coalesced accesses. For square alignment matrices and tiles of width N and T, respectively, *Rodinia-NW*'s parallel kernel is invoked $2\times\lceil N/T\rceil-1$ times (once for each minor diagonal of tiles). After carefully analyzing *Rodinia-NW*, we found the following limitations.

First, *Rodinia-NW* is designed for a single pairwise comparison. Applications such as those above require hundreds to thousands of comparisons. As such, they introduce a second exploitable level of parallelism, especially as each pairwise comparison is independent. Moreover, the sequences generally differ in length but *Rodinia-NW* only supports sequences of equal length, requiring padding to handle more general cases.

Second, *Rodinia-NW* requires three data transfers for each alignment, an approach that can be improved. Before kernel launch, the alignment matrix is initialized (with the gap information) on the CPU. Next, alignment matrix and score matrix are copied from CPU to GPU. The alignment matrix is processed on the GPU and then copied back to the CPU as a final step. We note that the two copies of the alignment matrix are *O(nm)* each. However, the first data transfer of the alignment matrix can be avoided by initializing its 1st row and 1st column directly on the GPU.

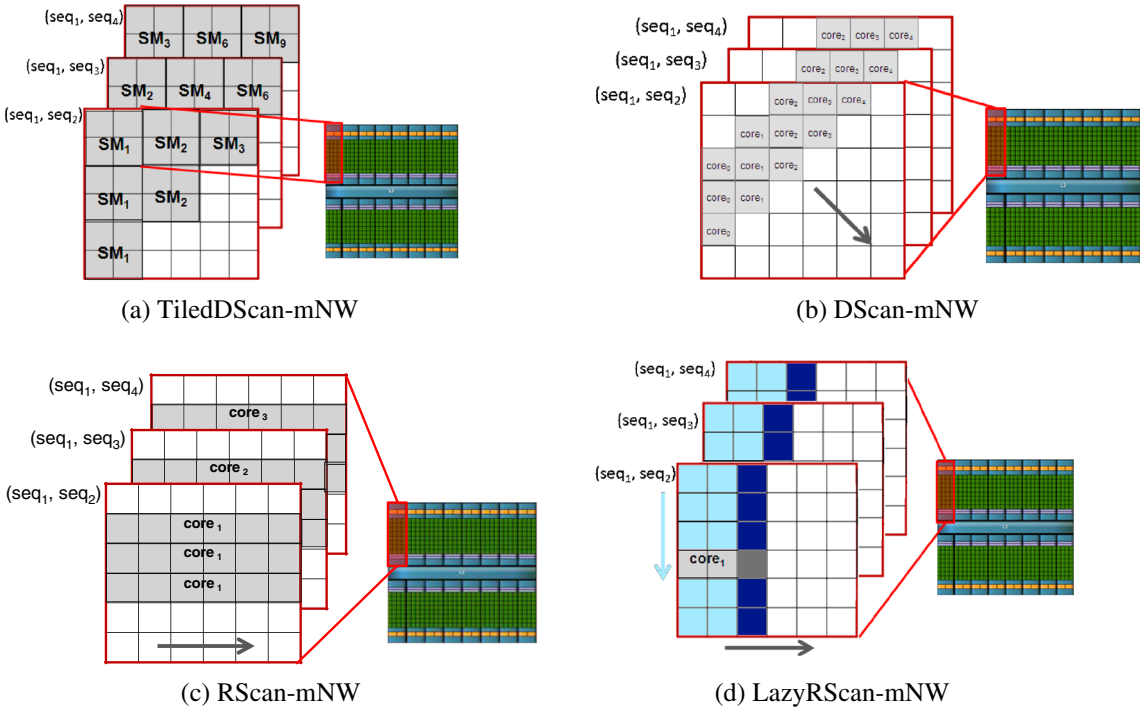
Finally, CUDA does not support global barrier synchronization among thread-blocks within a parallel kernel (an implicit global synchronization takes place at the end of each kernel execution). Since in *Rodinia-NW* each tile is mapped to a thread-block and tiles must be processed in diagonal strip manner, a global synchronization among thread-blocks operating on the same diagonal is required before proceeding to the next diagonal. This is accomplished by invoking multiple kernel launches from the host side. This approach has two limitations: (i) each kernel launch has an associated overhead (that depends on the GPU device), and (ii) the GPU is underutilized by kernel launches that process small numbers of tiles (i.e., those corresponding to the first and the last diagonals).

4 Design of GPU-Workers

In this Section we describe four alternative implementations of multiple pairwise alignments using NW on GPUs: *TiledDScan-mNW*, *DScan-mNW*, *RScan-mNW and LazyRScan-mNW*. All these implementations, exemplified in Figure 1, aim to overcome the limitations pointed out above. In Section 5, we describe the integration of these implementations in our distributed framework for large-scale sequence alignments.

4.1 TiledDScan-mNW: Multiple Aligments with Tiling

The first method (*TiledDScan-mNW*) is a direct extension of *Rodinia-NW* to multiple pairwise alignments. This approach still uses tiling and operates in diagonal strip manner, performing multiple kernel invocations to compute the alignment matrices. However, for each kernel invocation, multiple alignment matrices are concurrently processed using different thread-blocks (and SMs). This is illustrated in Fig. 1a, where we concurrently perform three pairwise comparisons: (seq_1, seq_2), (seq_1, seq_3) and (seq_1, seq_4). In the first iteration, the top-left tiles of the three matrices are processed in parallel by three thread-blocks, and thus mapped onto three streaming multiprocessors: SM_1, SM_2 and SM_3. In the second iteration, the tiles of the second minor diagonal of the three matrices are processed in parallel by six thread-blocks, and thus mapped onto streaming multiprocessors SM_1-SM_6. Note that, for m pairwise comparisons, the number of kernel invocations of *TiledDScan-mNW* is reduced by a factor m (as compared to *Rodinia-NW*); for each kernel call, the number of thread-blocks is increased by a factor m. This reduction has two advantages: (i) a limited kernel invocation overhead, and (ii) an improved GPU utilization. Execution configurations with a large number of threads allow not only exploiting all the SMs and cores available on the GPU, but also hiding the global memory access latencies (and NW is a memory-intensive application). Being an extension of *Rodinia-NW*, *TiledDScan-mNW* retains its advantages: regular computational patterns and coalesced memory access patterns when storing alignment data from shared memory to global memory.



(a) TiledDScan-mNW (b) DScan-mNW

(c) RScan-mNW (d) LazyRScan-mNW

Figure 1 Illustrations of our 4 GPU implementations of NW-based parallel pairwise alignments and of the mapping to the underlying GPU cores and SMs. In TiledDScan-mNW, the alignment matrices are tiled, and each tile is processed by a thread-block (and mapped onto a SM). In DScan-mNW, every alignment matrix is processed by a thread-block (and mapped onto a SM). In RScan-mNW and LazyRScan-mNW, every alignment matrix is processed by a thread (and mapped onto a core). However, in LazyRScan-mNW, every thread only writes to the global memory the last column of each slice (denoted by *darker blocks*).

4.2 DScan-mNW: Single-Kernel Diagonal Scan

TiledDScan-mNW still requires multiple kernel invocations to perform m pairwise alignments. Even if the parallelism within each kernel call is improved by a factor m compared with *Rodinia-NW*, some kernel invocations still exhibit limited parallelism (and limited opportunity to hide memory latencies). Our second implementation – *DScan-mNW* – performs a diagonal scan with a single kernel call. As illustrated in Fig. 1b, in this case each alignment matrix is assigned to a thread-block (and mapped onto a SM). No tiling is performed. The computation iterates over diagonals. For each diagonal, every element is processed by a thread (and mapped onto a core).

To limit the number of expensive accesses to global memory, the computation is fully performed in shared memory. The alignment matrix is stored in row-major order in global memory and in minor diagonal order in shared memory. According to equation (1), at each iteration three diagonal lines are required: the first two diagonal lines cache previous data and the third one contains the newly computed elements. Once computed, this third line can be copied from shared to global memory. At that point, the first diagonal line can be discarded and the shared memory reused for the next iteration. To summarize, the matrices are created in shared memory and moved to global memory diagonally. The main disadvantage of this approach is the uncoalesced memory accesses required to store diagonal data to global memory. We found that the latencies of such irregular access patterns can be effectively hidden by using large numbers of threads.

The computational pattern of our *DScan-mNW* is similar to the SW intra-task parallelization proposed by Liu et al. [32]. However, [32] avoids uncoalesced memory accesses by storing the alignment matrix in global memory in minor diagonal order. We found that, when using large thread-blocks to hide memory latencies (e.g., 512 threads/block), the overhead due to uncoalesced memory access patterns is reduced to 10 % and 7 % of the execution time on Fermi and Kepler GPUs, respectively (the exact percentage depends also on the clock-rate of the memory system). On the other hand, storing the alignment matrix in row-major facilitates the trace-back operation (which is not considered in [32]) in two ways: first, it avoids the need for complex index translation; second, the more regular data layout leads to better caching properties.

4.3 RScan-mNW: Row Scan Via Single CUDA Core

Our third method – *RScan-mNW* – uses a *fine-grained matrix-to-core mapping* and a *row-scan* approach. First, each alignment matrix is computed by a single GPU core. Second, to allow regular compute and memory access patterns, each

alignment matrix is computed row-wise (rather than diagonal-wise). This computational pattern is illustrated in Fig. 1c.

This method leverages shared memory in order to allow data reuse and minimize the global memory transactions. The parallel kernel iterates over the rows of the alignment matrices. For each iteration, only two rows per matrix must reside in shared memory: the previously computed one and the one containing newly computed elements. Only the left-most element of the new row must be loaded from global memory; for the rest, the computation happens solely in shared memory. Once the new row has been computed, it is copied from shared to global memory. The previously computed row can be discarded, and the new one can be cached for use in the next iteration. The kernel has two phases: computation and communication. In the computation phase, the threads within a thread-block operate fully independently: each thread computes the data corresponding to the row of an alignment matrix and stores them in shared memory. In the communication phase, threads belonging to the same thread-block cooperate to transfer row data from shared to global memory in a coalesced fashion (that is, each alignment matrix is transferred cooperatively by multiple threads). In case of very long sequences, rows are split into slices so as to fit into shared memory. The size k of these slices is configurable. Large slices require more shared memory, which in turn limits the number of active threads on each SM. Small slices (e.g. $k<32$ elements) lead to warp underutilization in the communication phase, which in turn can hurt the performance. The usage of shared memory is a major concern in the kernel configuration process. The per-block shared memory can be calculated using the following formula:

$$\text{shmem} = \text{BLOCK\_SIZE} \times k \times (1char + 2ints)$$

Each thread stores three sets of data: the sequence data and two sections of the alignment matrix. Each thread-block performs BLOCK_SIZE pairwise alignments using slices of size k. By setting the BLOCK_SIZE and k to 32, we use 12 KB of shared memory with no warp underutilization. With this setting, each SM can concurrently run up to four thread-blocks.

The advantages of this approach are twofold. First, the computational pattern is extremely regular: unlike diagonals, rows are all of the same size. Second, data transfers between shared and global memory are naturally coalesced. The main drawback to this approach is that the parallelism is limited by the GPU memory capacity. For example, if the sequences to be compared are of length 2,000 and the alignment matrices contain 4-byte integers, then each matrix will be of size 32 MB. To fully utilize the cores of typical GPUs (say 480 cores), we should allow 480 parallel pairwise comparisons, requiring a total of roughly 15GB of memory. This number considerably exceeds the 1-5GB of memory present on most GPUs. Therefore, on long sequences *RScan-mNW* will tend to underutilize the GPU resources. On the other hand, this approach is very promising for short sequences (e.g. <500). For long sequences, an alternative optimization would be to break the alignment matrices into smaller strips to reduce the memory footprint, and use dual-buffering to move previously computed strips to the CPU while computing new ones. Finally, we note that certain scoring schemes allow for linear memory NW algorithms of minimal complexity: under these limited and less-commonly used schemes, highly efficient parallelism could be achieved using *RScan-mNW.*

The computational pattern of our *RScan-mNW* is similar to the SW inter-task parallelization proposed by Liu et al. [32]. However, their proposal does not use shared memory in the kernel and adopts a different data layout in global memory. Specifically, to avoid uncoalesced global memory accesses, Liu et al. place data corresponding to different alignment matrices into continuous global memory space. For instance, the i^{th} element of global memory is from the i^{th} alignment matrix, while the $(i+1)^{th}$ element is from the $(i+1)^{th}$ alignment matrix. This memory layout leads to poor data locality during the trace-back phase. As mentioned above, trace-back is not considered in [32], but is a necessary operation in the problem we consider.

4.4 LazyRScan-mNW: Whole Matrix Calculation via Single CUDA Core with Lowered Global Memory Requirements

The three aforementioned methods share the same problem: the maximum number of alignments that can be computed in parallel is limited by the amount of global memory. As discussed, this limitation is particularly significant in the case of *RScan-mNW*, where each alignment is mapped onto a GPU core. In this case, the limited number of alignment matrices that can be accommodated in the available global memory leads to severe core underutilization. However, several applications require the actual sequence alignment (computed through the trace-back operation) only for a subset of the pairs in the dataset. For these applications, it is possible to record only the alignment score (rather than the whole alignment matrix). This score can then be used as a filter to avoid performing unnecessary trace-back operations on sequence pairs that are too dissimilar. Exactly such a requirement spawned our initial interest in this topic — namely computing all possible pairwise comparisons among a large number of sequences but retaining only those with a high level of similarity. In this case, the lack of the need to have the whole alignment matrix stored in global memory makes it possible to derive a method that can make full use of the computational power of the GPU cores.

With this relaxed requirement in mind, Lazy*RScan-mNW* optimizes our row-scan approach (*RScan-mNW*) so as to minimize global memory reads and writes and at the same time make better use of the shared memory available on the

streaming multi-processors. The new method is based on the following insight: much of the information in one slice of an alignment matrix can be discarded safely during the computation of that slice. Lazy*RScan-mNW* is extremely frugal about global memory usage, and avoids accessing global memory whenever possible. The computational pattern is illustrated in Fig. 1d and presented in the pseudo-code in Fig. 2.

In a manner similar to *RScan-mNW*, in order to effectively use shared memory even in the presence of long sequences, Lazy*RScan-mNW* divides the alignment matrix into vertical *slices* of k columns. However, when copying the scores into global memory, this method only retains the scores in the last column of each slice (and discards the other scores). For each matrix, the computation of the next slice resumes where the last one left off, by taking the stored rightmost scores of the last computation as its leftmost column's scores. Given the leftmost column's scores, each iteration of vertical slice can compute the rightmost column's scores by performing the standard NW computation. Thus, by carefully selecting k (see section 6.3), we reduce the number of write operations to global memory. Computing a slice of width k and height l requires $2\ l$ global memory operations (l reads and l writes). In addition, for each matrix, the global memory requirement is limited to l scores, and the shared memory requirement is linear in k. Limiting the shared memory utilization allows more thread-blocks to be executed in parallel, thus hiding global memory access latencies.

We further reduce the shared memory usage of each vertical slice by implementing an optimization suggested by Myers and Miller [40]. Specifically, instead of storing in shared memory two distinct rows (the current and the previously computed one), we store a hybrid row obtained by progressively overwriting the last computed row with the current one. This method requires recording an additional interim *diagonal look-back* value, representing the conflicted array element that would otherwise be overwritten by the next row's value. We store this additional value in a register. Figure 3 illustrates the optimization.

Lazy*RScan-mNW* inherits all the advantages of *RScan-mNW* as well as some additional benefits. First, each thread only requires a linear amount of global memory. Thus, the global memory required will be equal to *batch-size*$\times l$ *integers*, where l is the length of the longest sequence, and *batch-size* the number of pairwise alignments computed in parallel. Second, instead of reading and writing to global memory at every cell, the method reads and writes to global memory once every k cells. This allows the GPU scheduler to alternate between warps that are performing calculation and warps that are waiting on memory accesses, thus better hiding the latency between global memory accesses. Third, the amount of shared memory required to store the scores is halved using the optimization described above.

However, this method has a drawback: it is no longer trivial to restore the alignment matrix, making the trace-back operation challenging. *LazyRScan-mNW* is therefore more suitable to applications that require the trace-back to be calculated only for a subset of the sequence pairs (for example, those with a similarity score above a predefined threshold).

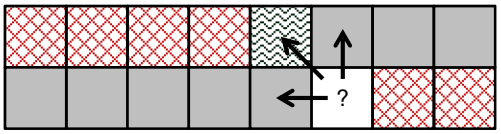


Figure 3 Storing in shared memory a hybrid row consisting of cells belonging to two contiguous rows allows saving half of the shared memory required for row calculation. As computation (*white cell*) progresses from left-to-right horizontally, obsolete cells (cross-hatched in red) are gradually evicted from shared memory. To complete the computation, only the solid grey cells as well as an interim diagonal look-back value (denoted with the zig-zag pattern) are required. The diagonal lookback value is stored in a register.

LazyRScan-mNW

```
1: Initialize matrix size (w, l)
2: for each slice size (k, l):
3:     calculate row scores for row 0
4:     for row ← 1 to l:
5:         load leftmost score for column 0 from global memory
6:         for column ← 1 to k:
7:             calculate value for(row, column) in shared memory
8:         save rightmost score for column l to global memory
9:     end for row
10:    advance slice by k
11: end for each slice
11: returns final matrix score
```

Figure 2 Pseudo-code version of LazyRScan-mNW.

4.5 Comparisons of Memory Usage Between Different Methods

Table 1 summarizes the global and per-block shared memory usage of the described methods. In the table, *batch-size* is the number of pairwise alignments computed in parallel, l is the length of the longest sequence, and k denotes the slice size, and is a configurable value smaller than l.

Table 1 Global and per-block shared memory requirements for different methods (the amount of storage needed for actual sequence data is not accounted for in global memory usage). A *batch* is a set of pairwise alignments processed in parallel on the GPU.

Method	Global memory	Per-block shared memory
Rodinia-NW	*batch-size*$\times\ 2\ l^2$ ints	$2\ k^2$ ints
TiledDScan-mNW	*batch-size* $\times\ l^2$ ints	k^2 ints $+2\ k$ chars
DScan-mNW		$3\ l$ ints $+2\ l$ chars
RScan-mNW		$(2\ k$ ints $+\ k$ chars$) \times$ *block-size*
LazyRScan-mNW	*batch-size* $\times\ l$ ints	$(k$ ints $+\ k$ chars$) \times$ *block-size*

5 Design of Our Distributed Framework

5.1 System Overview

Our framework follows a *producer-consumer* model and consists of two major components: a *Cluster-Level Dispatcher (CLD),* and a set of *Node-Level Dispatchers (NLDs).* The CLD operates at the cluster-level: it progressively distributes work to the NLDs on the compute nodes and then later aggregates the results back from them. At the node level, each NLD distributes work to that node's CPUs and GPUs. The CPU workers and GPU dispatcher on each node compute the NW alignments.

5.2 Cluster-Level Dispatcher

At the cluster level, the framework spawns a group of MPI processes consisting of a root CLD process and a set of NLDs, one per requested node. The CLD progressively distribute jobs to NLDs and tracks their progress. The CLD is flexible enough to distribute different amounts of work to different NLDs depending on relative node performance.

Figure 5 shows the CLD's architecture. It consists of two functional units. One generates work to be performed and tracks overall progress; the second distributes work to the NLDs. When the NLDs return results, the distribution unit passes them to the work-generating unit where a bookmarking mechanism tracks overall progress. The pseudo-code for CLD and NLD is shown in Fig. 4. In the pseudo-code, each work list consists of a set of pairwise alignments to be performed. As can be seen, the work is pulled from the NLDs and distributed by the CLD upon request.

Cluster-level Dispatcher (CLD)

```
1: Initialize
2: Distribute work lists to NLDs
3: while there exist pairwise alignments to be performed do
4:     if work request from NLD do
5:         send work list to NLD
6:         process returned results
7:     end if
5: end while
```

Node-level Dispatcher (NLD)

```
1: Query compute resources (CPUs and GPUs)
2: Receive work list from CLD
3: while no stop signal from CLD
4:     if work list is empty do
           request new work list from CLD
5:     end if
6:     if idle CPU available do
           assign a set of pairwise alignments to CPU
7:     end if
8:     if GPU returns alignment matrices do
           trace-back on CPU
9:     end if
10:    if idle GPU available do
           assign a set of pairwise alignments to GPU
           invoke alignment kernel
11:    end if
12: end while
```

Figure 4 Pseudo-code of CLD and NLD.

5.3 Node-Level Architecture

The NLDs consume data from the CLD and dispatch those data to CPU and GPU-level workers (Fig. 5). When a NLD is spawned on a node with c CPU cores, it generates c threads. Of these, c-2 perform standard NW alignments. One of the remaining two threads is responsible for alignment trace-back (which recovers the sequence alignment itself): experimental data suggest that one trace-back unit per node is sufficient. The final thread acts as the GPU-dispatcher: it is aware of the number and performance of the node's GPUs. As such, it asynchronously distributes work to different GPUs by invoking a GPU worker function on them. GPU workers perform parallel sequence alignments using one of the GPU implementations described in Section 4. The GPU implementation selection is made based on the performance of the target GPU. To simplify the implementation of the GPU workers, if the two sequences to be aligned have different lengths, the GPU dispatcher will pad the shorter one to the same length as longer one so that the alignment matrix is always square. Both the CPU workers and the GPU dispatcher continuously query the NDL for work and return results until there is no work left on the NLD. When the NLD itself is idle, it queries the CLD for further sequence pairs to align. The NLDs are independent of each other and communicate only with the CLD.

In our implementation, the trace-back operation is performed on CPU. Our decision is motivated by the following considerations. First, we experimentally verified that trace-back is a reasonably fast process, accounting for less than 0.2 % of the execution time. Second, trace-back is in nature a sequential operation, with no fine-grained data-structure parallelism: the only form of parallelism that can be exploited on GPU is essentially inter-alignment coarse-grained parallelism. The main reason for performing trace-back on GPU would be to avoid large data transfers from GPU to CPU. In our implementation, we hide such data transfers by performing dual

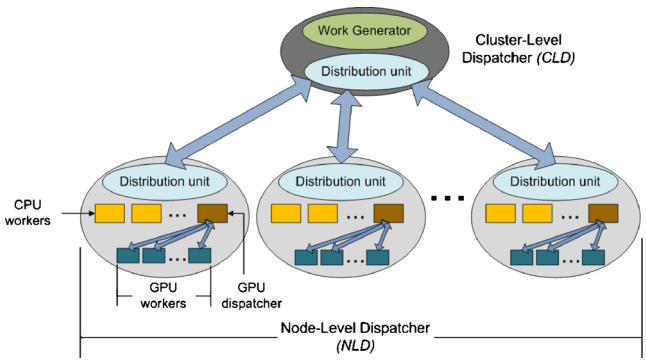


Figure 5 Design of our distributed framework for CPU-GPU clusters.

buffering and overlapping kernel execution with GPU-to-CPU data copy. A different choice has been made by Gao et al. [31], who generate a movement matrix in addition to the alignment matrix and use that movement matrix to perform trace-back in a straightforward manner. By keeping the computation of the alignment matrix separated from the trace-back, we allow our code to be reused by applications requiring the SW algorithm. We note that the computation of the alignment matrix in NW and SW is almost identical (except for the initialization and the handling of negative scores); the two algorithms differ mainly in the trace-back operation.

5.4 GPU-Dispatcher Optmizations

Pinned Memory Generally each GPGPU computation consists of three stages. First, the initial data are copied from the host to the device's global memory. Then, the CPU launches the parallel kernel, allowing the calculation of the results on the device. After this operation finishes, the data are copied back to the host memory. With pageable memory, the memory copy operations contribute a significant fraction of the execution time. This cost is partially due to the fact that pageable memory is allocated in user space. When a memory copy is invoked, the GPU driver will allocate some pinned memory in kernel space, perform a memory copy from user to kernel space, and then initiate a DMA transfer to the GPU. Using pinned memory directly avoids this user-to-kernel space copy, thus reducing the cost of memory transfers.

Dual-Buffering Dual-buffering is a common scheme for reducing overhead by overlapping communication and computation. It requires that a non-blocking communication mechanism be provided by the system. In our system, there are two main operations that are time-consuming but can be executed concurrently: kernel calls to calculate the alignment matrix, and the memory copy operation to transfer the calculated alignment matrix from device's global memory back to the host's memory. Dual-buffering is employed in order to ensure mutual exclusion in the access to the two alignment matrices, one of which is being calculated and one copied. The result is that the kernel always works on half of the available memory, leaving the other half – the buffer that is already fully calculated – free to be transferred back in a non-blocking manner. By using dual-buffering, we can further exploit the concurrency offered by the GPU by interleaving the two operations (Fig. 6-a).

To implement dual-buffering, we used asynchronous pinned memory operations and CUDA streams. As a result, the total execution time can be less than the sum of the time for kernel calls and for memory operations. Ideally, using dual-buffering completely hides the faster operation. However, this latency hiding is not free: the number of transfers and kernel calls is doubled. This doubling results in a slight increase in both kernel and memory transfer time due to function call overhead. We have measured this effect on the GTX460 GPU using a sequence length of 1536 and the *DScan-mNW* implementation. The results are shown in Fig. 6-b. As can be seen, if all memory transfer operations could be hidden using dual-buffering, then the time-savings compared to the non-dual-buffering case would be:

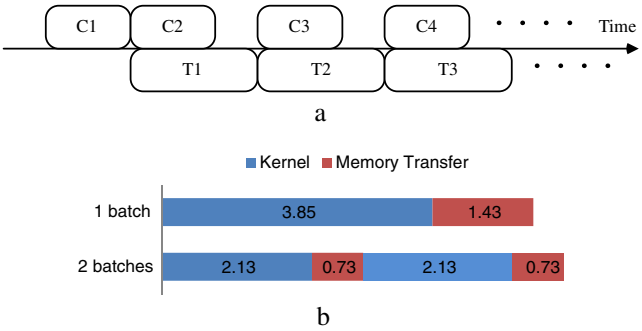


Figure 6 **a** Illustration of dual-buffering. **b** Measurement of kernel execution and memory transfer time (using pinned memory) for n_{align} sequence alignments on a GTX460 GPU. The number of alignments performed (n_{align}) is selected so as to use 70 % of the global memory. In the 1-batch case, we group the pairs in a single batch of size n_{align}. In the 2-batch case, we group them in two batches, each of size $n_{align}/2$.

$$(1–2.13 \times 2/(3.85 + 1.43)) \times 100\% = 19.31\%.$$

Global Memory Access Pattern Optimization Parallel threads in CUDA are grouped into thread-blocks, and the GPU coalesces the global accesses into as few transactions as possible. Misaligned memory operations cause delays in the computation as some threads have to wait for the misaligned reads to complete. This issue however, can potentially be minimized. In the *LazyRScan-mNW* implementation, global memory reads and writes on the same position on different matrices of consecutive threads are also consecutive, enabling the hardware to perform memory access operations in a coalesced fashion.

Caching Global Memory Reads into Shared Memory Global memory is needed to store the sequences, however reading the sequences from global memory is an expensive operation. In *LazyRScan-mNW*, each slice of sequence of width k has a relatively small number of bases that need to be repeatedly accessed and compared against the other sequence. Thus, we avoided reading the slice sequence from global memory by caching the whole slice up front in shared memory. Given that, in the inner loop (Fig. 2, line 6), each base of the slice is then repeatedly compared to one base from the other sequence, the base belonging to the other sequence could also be cached up front to reduce global memory read latency.

Device-to-Host Data-Level Optimization In the case of *LazyRScan-mNW*, it is possible to only transfer the final score instead of the whole alignment matrix, largely eliminating the device-to-host transfer latency. In order to allow this simplification, we derive a lower bound L on the potential alignment score with a given percent identity. As an aside, the percent identity statistic is a reasonable indicator of sequence relatedness and is commonly employed in the biological literature. Our bound guarantees that all pairs having identity score at least as high as our threshold level are realigned and the trace-back computed on the CPU. Given identity score target M_{ID} ($0<M_{ID}<1$) and two sequences of length m and n (where $m > n$), L is given by:

$$L = m \times M_{ID} \times S_{match} + 2 \times m \times S_{gap} \times (1 - M_{ID})$$

S_{match} is the alignment score for two matching bases and S_{gap} is the penalty for a gap (recall we are using linear gaps). For the required percentage identity is to be achieved, we must have at least $m \times M_{ID}$ positions that match. The worst case of an alignment with such a percent identity is then that all of the remaining positions are gaps (S_{gap}), giving us our value of L.

6 Experimental Evaluation

In this Section, we present two sets of experiments: (i) single GPU experiments, and (ii) cluster experiments. The former are meant to evaluate our GPU implementations of NW, the latter to evaluate our distributed framework.

6.1 Experimental Setup

Hardware Setup Single GPU experiments have been performed on a variety of low-end and high-end GPUs, listed in Table 2. Cluster level experiments have been performed on two cluster settings (a low-end and a high-end cluster), whose setups are summarized in Table 3.

Software Setup The CUDA 5.0 driver and runtime are installed in all the machines used. The OS in use on the high-end cluster is CentOS5.5/6 with g++4.1.2; the OS in use on the low-end cluster is Ubuntu 12.04 with g++ 4.6.3. We used MPICH2 (version 1.4.1p1) as the implementation of MPI. Each data point represents the average across 3 executions.

Dataset Our reference dataset consists of about 25,000 unique 16S rDNA genes from the Ribosomal Database [21]. The sequences are on average 1,536 bases long.

Table 2 Characteristics of the GPUs used in our evaluation.

GPU	Type	Values
Low-end GPUs	Quadro 2000	4 SM × 48 cores ~1 GB Global memory
	GTX 460	7 SM×48 cores ~1 GB Global memory
	GTX 480	15 SM×32 cores ~1.5 GB Global memory
High-end GPUs	Tesla C2050	14 SM×32 cores ~2.6 GB Global memory
	Tesla C2070/C2075	14 SM×32 cores ~5 GB Global memory
	Tesla K20	13 SM×192 cores ~4.7 GB Global memory

6.2 Performance on Single GPU for General use Cases

Our first set of experiments is meant to evaluate our GPU implementations and compare them with *Rodinia-NW*. In Section 3.3 we noted two limitations in *Rodinia-NW*: unnecessary memory transfers from CPU to GPU and inefficiencies in the computational kernel and its invocations. Below, we will show how we improve performance with respect to both limitations.

Memory Transfers As explained in Section 3.3, *Rodinia-NW* initializes the alignment matrix on CPU and copies it to GPU. Also, to simplify memory access during computation, it creates a temporary substitution score table of size m x n during CPU initialization. For problems of the size considered, data transfer consumes considerable amount of time. An obvious optimization is to move the initialization from CPU to GPU. In addition, by omitting the creation of the temporary substitution table, more alignment matrices can be accommodated on the GPU, thus allowing for increased parallelism. In Fig. 7 we show the effect of these optimizations on different GPUs. In all experiments, 64 pairwise alignments are performed. The *optimized* version initializes the alignment matrices on GPU and avoids the initial CPU-to-GPU data transfer. On top of this, the *optimized + pinned memory* version uses pinned memory. As can be seen, the proposed memory optimizations lead to a 5–10 % and a 20–25 % decrease in execution time on low-end and high-end GPUs, respectively. In addition, the combination of the memory optimization with the use of pinned memory leads to a decrease in execution time in excess of 30 % and 50 % on low-end and high-end GPUs, respectively.

Kernel Computation We now focus on the performance of our compute kernels. Our analysis has two goals: (i) evaluating the performance improvements over *Rodinia-NW*, and (ii) devising criteria for selecting the optimal GPU implementation

Table 3 Cluster setup.

Cluster	Nodes	CPUs	GPUs
Low-end	Node-1	1 × Intel Core 2 Quad Q9400, 2.66 GHz, 4 GB RAM	1 × Quadro 2000
	Node-2		
	Node-3		
	Node-4		
	Node-5	1 × Intel Core 2 Duo E8400, 3.0 GHz, 4 GB RAM	1 × Quadro 2000
	Node-6	1 × Intel Xeon E5-1603, 2.8 GHz, 4 GB RAM	1 × GTX 460
High-end	Node-1	2 × Intel Xeon E5620, 2.4 GHz, 48 GB RAM	2 × Tesla C2050
	Node-2		
	Node-3	2 × Intel Xeon E5-2620, 2.0 GHz, 64 GB RAM	Tesla C2050
			Tesla C2070
			Tesla C2075
	Node-4	2 × Intel Xeon E5620, 2.4 GHz, 48 GB RAM	4 × GTX 480

depending on the underlying GPU device. In Fig. 8 we show the relative speedup in kernel computation time of *DScan-mNW* and *TiledDScan-mNW* over *Rodinia-NW* (the speedup is computed as the ratio between the compute time of *Rodinia-NW* and that of our GPU implementations). We performed experiments on all available GPUs and varied the number of pairwise comparisons performed from 8 to 64. Given its fine-grained alignment-to-core mapping, on these datasets *RScan-mNW* underutilizes the GPUs and yields poor performance. This behavior, in general, holds when comparing long sequences on GPUs with 1-5GB device memory. Therefore, we focus on the other schemes.

Figure 8a reports the speedup of *TiledDScan-mNW* over *Rodinia-NW*. Note that *TiledDScan-mNW* performs fewer kernel calls (and therefore, has less kernel overhead) and involves more per-kernel computation (thus leading to increased parallelism). This difference is the source of the performance improvement achieved by *TiledDScan-mNW* over *Rodinia-NW*. Note that the speedup increases with the computational power of the GPU (from 1.2x on the Quadro2000 to 2x on the K20). In fact, the increased parallelism in the *TiledDScan-mNW* kernel can be better serviced by GPUs with more SMs and compute cores.

As can be seen in Fig. 8b, *DScan-mNW* also outperforms *Rodinia-NW* on all devices and datasets. Its performance is also generally better than that of *TiledDScan-mNW*, except on Tesla C207× cards. It is somewhat surprising that our approach does not show substantial speedup over *Rodinia-NW*

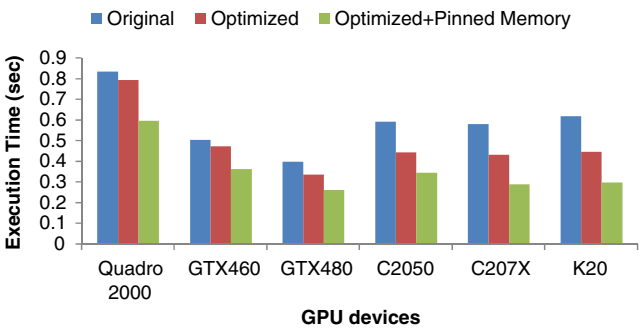


Figure 7 Evaluation of memory optimizations on Rodinia-NW: original implementation, optimized implementation with initialization of the alignment matrices on GPU, optimized implementation with pinned memory.

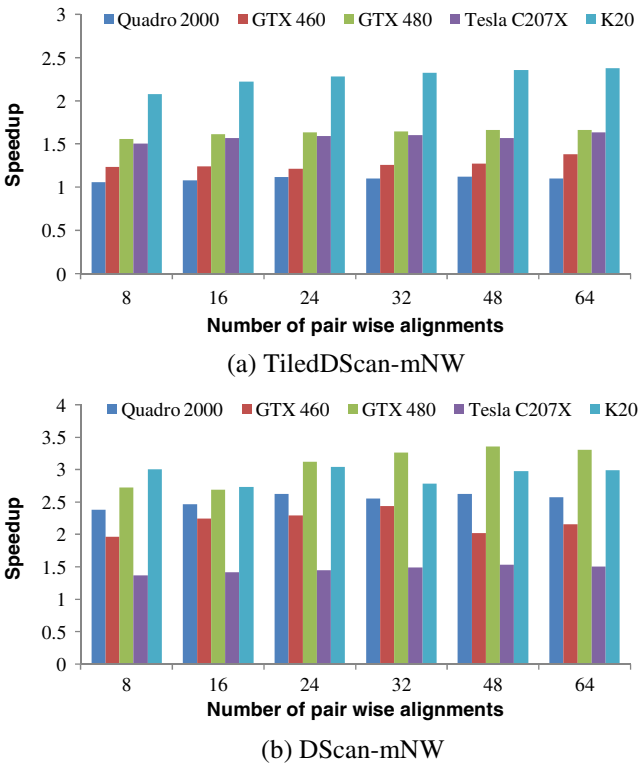


Figure 8 Kernel speedup over Rodinia-NW. **a** TiledDScan-mNW. **b** DScan-mNW.

on this device. It must be said that NW is an integer application, and Tesla GPUs are optimized for larger memory capacity (5GB vs. 1GB) and improved support for double precision floating-point operations, and have a reduced clock rate (1.15GHz vs. 1.4GHz in GTX 480 cards, for example). We believe that the high number of uncoalesced memory accesses performed by *DScan-mNW* may drive the poor performances on Tesla C207× cards, which have a slower memory clock.

Figure 9 reports the speedup of *RScan-mNW* over *Rodinia-NW* on sequences of different lengths. The number of pairwise comparisons performed in each experiment is reported on top of each bar (all experiments have been configured so as to use 70 % of the global memory). As mentioned in Section 4.3, in *RScan-mNW* each core computes an alignment matrix: in order to fully employ the computational resources of the GPU, *RScan-mNW* needs to perform a large number of parallel sequence alignments. However, the number of parallel pairwise alignments is limited by the available memory. For example, on K20 GPUs, sequence length of 1536 limits the number of parallel pairwise comparisons to 373, thus using only 373 of the 2496 cores on the device (e.g., ~15 % efficiency). On the other hand, sequences of length 512 allow 3356 parallel pairwise alignments, thus using all the available cores.

There is therefore pressure on the global memory capacity for long sequences, where the GPU global memory becomes the bottleneck, penalizing the performance of *RScan-mNW*. On the other hand, with shorter sequences, *RScan-mNW*'s performance improves: for 512-base sequences it gives a speedup of 5x over *Rodinia-NW*.

In general, Figs. 8 and 9 show that *DScan-mNW* and *TiledDScan-mNW* are preferable to *RScan-mNW* on the 1,536-base sequences in the 16S rDNA gene dataset. In addition, these results show that our methods overcome the inefficiencies of *Rodinia-NW* and suggest that *DScan-mNW* is preferable on all devices except Tesla C207×. On such cards, *TiledDScan-mNW* provides better performance. This finding

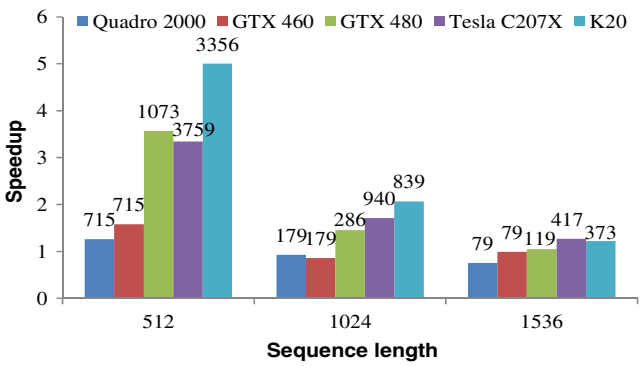


Figure 9 Kernel speedup of RScan-mNW over Rodinia-NW on sequences of various lengths. The numbers on the bars represent the number of pairwise alignments on the device for each experiment.

will be used to configure our GPU-workers. As next step, we want to determine how to size the amount of work that each GPU-worker should pull from the GPU-dispatcher to operate at full capacity. In fact, we want to fully employ the GPUs present in the system. The number of pairwise comparisons that can be performed concurrently on each GPU is limited by its memory capacity. We configured each GPU to operate with its global memory 75 % full. For the sequence lengths being considered, this leads to 79, 79, 119, 208, 417, and 372 parallel alignments on Quadro2000, GTX460, GTX480, Tesla C2050, Tesla C207× and K20 GPU, respectively.

Figure 10-a shows the speedup reported by *DScan-mNW* over an 8-threaded OpenMP implementation running on the 8-core CPU on Node-4 (see Table 3). The numbers over each bar represent the throughput in number of pairwise alignments/sec. For each GPU, we performed three experiments: one using unpinned memory, one using pinned memory, and one using dual-buffering. We first define an "optimal batch size" b_{SIZE} for a particular GPU to be the number of simultaneous alignments that can be performed given the device memory (as above). For the first two versions, we ran analyses consisting of a number of sequences equal to $3b_{SIZE}$ in order to effectively time the computation. For dual-buffering, only half of the GPU memory performs alignments at one time, so 6 batches of size $b_{SIZE}/2$ were timed. The

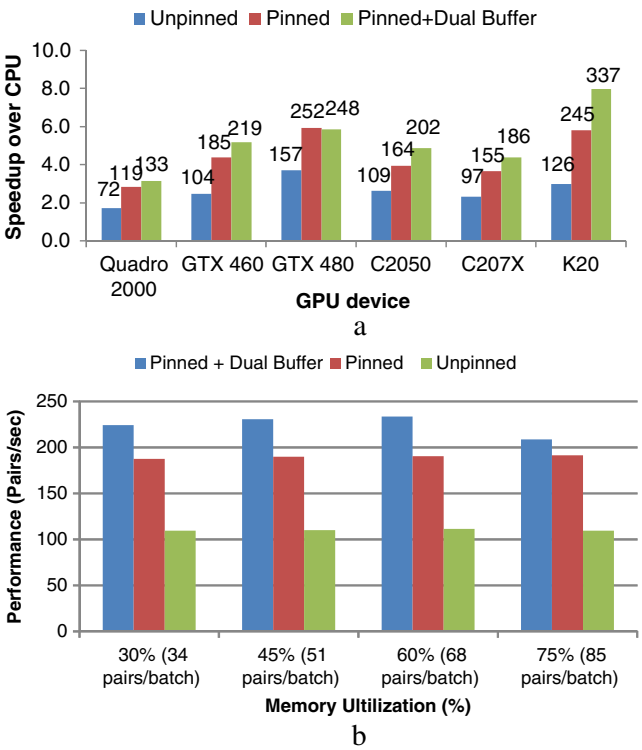


Figure 10 a Speedup reported by DScan-mNW over an 8-threaded CPU implementation running on the 8-core CPU on Node-4. The numbers on the bars represent the processing throughput (number of pairwise alignments/s). b Performance of DScan-mNW with different memory utilizations. The amount of memory used does not greatly alter performance.

performance was measured as the number of sequence pairs compared per second.

As can be observed from Fig. 10-a, switching to pinned memory offers a gain of roughly 1.6x, consistent with previous findings [41]. The application of dual-buffering along with pinned memory offers an additional average 1.2x speedup, with the exception of the GTX480 system, which does not show significant speedup. We speculate that the reason for this lack of improvement is that the GTX480 has a more restricted handling of CUDA streams (it allows concurrent memory transfer and kernel execution with a single copy engine) that does not allow the same level of overlapping of memory transfers and kernel computations possible on other devices (with 2 copy engines). In general, it can be observed that even inexpensive low-end GPUs (like the GTX460 and GTX480) offer throughput in the order of 200–250 pairwise comparison/s.

Figure 10-b shows the effect of varying the GPU's memory use on the performance of *DScan-mNW*. The data shown are based on using a GTX460 GPU with 1GB of memory and a sequence size of 1536. Recall that *DScan-mNW*'s memory usage is of the order of l^2**batch size*, meaning that selecting a memory footprint determines the batch size to be used. This fact would tend to imply that we expect performance to drop when less global memory is employed because the batch size may be smaller than the number of available cores. However, when we actually test the effects of changing the memory size, the results are somewhat surprising: performance is rather similar over a range of memory sizes. This observation is not specific to the GTX460: it is also seen with the Tesla C2075 (data not shown). As we explore further in Section 6.4, this behavior seems to be due to saturation of memory bandwidth resulting from this kernel's numerous uncoalesced memory accesses.

6.3 Performance Analysis of LazyRScan-mNW on Single GPU

As discussed in Section 4, thanks to its optimized global and shared memory usage, Lazy*RScan-mNW* can potentially offer some speedup over the other three GPU kernels.

The critical parameter affecting the performance of this method is the slice size k, that is, the number of columns calculated by a thread before writing a column of the alignment matrix to global memory. Importantly, this parameter can influence the performance of the kernel in two conflicting ways. On one hand, larger values of k help threads to avoid expensive global memory read and write operations. On the other hand, smaller values of k reduce shared memory use so that the GPU can schedule more threads to run simultaneously, hiding the global memory latencies by alternating between warps waiting on memory and on computation operations.

We explored how the performance varies with k. In particular, we considered kernel configurations that fully use the global and shared memory available on the GPU. To this end, we set the batch size (that is, the number of alignments performed in a single kernel calls) so to utilize 80 % of the global memory. We recall (Table 1) that *LazyRScan-mNW* stores in global memory l integer scores per alignment (l being the length of the longest sequence). Therefore, the batch size can be computed by dividing the amount of global memory used by $l \times 4$ bytes. In addition, in this method every thread performs a full alignment: thus, the kernel is invoked with a number of threads equal to the batch size. In our analysis, we started with thread-blocks of the size of a warp (32 threads), and performed several experiments by progressively doubling the thread-blocks' size until reaching 512 threads per block. Finally, we wanted to see if having multiple thread-blocks concurrently executing on one streaming multi-processor (SM) would improve performance. Thus, we varied the shared memory utilization between 25 % (12 KB), 50 % (24 KB), and 100 % (48 KB), respectively resulting in 4, 2 and 1 thread-block concurrently scheduled on the same SM. The parameter k can then be calculated from the formula on shared memory use in Table 1: namely, by dividing the amount of shared memory per SM by the block-size times 5 bytes (that is, the amount of storage required for 1 integer and 1 char). To simplify this operation, we provide to potential users a spreadsheet along with our open-source code to help determine k for each of the scenarios above.

Table 4 shows the performance achieved on a GTX 480 GPU when varying the shared memory use, the block size,

Table 4 Performance of LazyRScan-mNW with different shared memory, block size, and slice size settings. Bold values represent the best results reported for each shared memory configuration.

Concurrent blocks per SM	Block size	Slice size (k)	Performance (Pairs/sec)
4	32	75	777
	64	37	1943
	128	**18**	**3987**
	256	8	3691
	512	3	1317
2	32	152	742
	64	75	768
	128	37	1935
	256	**18**	**3924**
	512	8	3527
1	32	306	315
	64	152	740
	128	75	768
	256	37	1924
	512	**18**	**3840**

and the slice size k. As can be seen, the performance of *LazyRScan-mNW* is relatively consistent for fixed k when the shared memory and block size settings are changed. Specifically, the performance peaks at $k=18$. When we repeated this same experiment on different GPU devices, we again found the same value of k to be optimal (Fig. 11). These results can be explained as follows. In all configurations with $k=18$, 512 threads are scheduled concurrently on each SM (that is, 4 blocks×128 threads or 2 blocks×256 threads or 1 block×512 threads). This suggests that scheduling 16 warps per SM is enough to hide the memory latencies, and, at the same time, a value of $k=18$ is large enough to reduce the amount of expensive global memory operations. As the characteristics of k remain relatively constant among different GPU devices, there seems to be some generality to these results. We also notice that the peak performances of *LazyRScan-mNW* (from about 4000 pairwise alignments/s on low-end GPUs to about 6000 pairwise alignments/s on high-end GPUs) are far better than those reported by the other three GPU kernels. Interestingly, the Kepler K20 GPU does not offer substantial performance improvements over Tesla C20XX devices. This observation can be explained by the fact that the memory latencies are in all cases sufficiently hidden by context switching between 16 warps. The K20 GPU, however, offers more favorable performance for small values of k. In this case, the more frequent global memory operations are better hidden by the massive multi-threading of the Kepler architecture.

6.4 Detailed Analysis of Bottlenecks in Kernel Performance with the Nvidia Profiler

As mentioned in Section 6.2. B, the lack of performance improvements seen when the memory use is increased (Fig. 10-b) is interesting. We would like to understand why limiting the amount of global memory used did not reduce the performance of *DScan-mNW*.

To answer that question, we took a deeper look at potential bottlenecks for two of these implementations: *DScan-mNW* and *LazyRScan-mNW*. We selected these two because they are at opposite ends of the spectrum of memory use: *DScan-mNW* makes frequent access to global memory while *LazyRScan-mNW* does not intensively use global memory but relies heavily on shared memory. Using the Nvidia profiler *nvprof*, we produced execution profiles of both implementations on the GTX460 and visualized those profiles with the Visual Profiler.

Figure 12-a shows the proportion of time the GPU spends in compute and in memory operations, respectively, for *DScan-mNW* in the case of 75 % memory utilization. The device spends much more time in memory operations than in compute ones. When we investigated the same metrics in the case of 30 % global memory utilization, the results are very similar (data not shown). The program's memory accesses tend to induce stalls due to their execution dependence (Fig. 12-b), suggesting performance is being limited by memory access patterns. When we analyzed these memory accesses more closely, we found that cache memory was only being used lightly but that the global memory was being heavily accessed, as reported by the profiler. We conclude that this implementation suffers from a global memory bandwidth capability problem, such that the GPU-to-global memory bandwidth is saturated even when only part of the GPU memory is being used to store alignment matrices. Thus, the implementation is unable to efficiently employ the available cores due to the limitations of memory bandwidth. This hypothesis clarifies the results shown in Fig. 8-b, where GPUs with very different numbers of cores did not differ greatly in their performance.

Since *LazyRScan-mNW* does not use global memory to store alignment matrices, it can potentially avoid memory bandwidth limitations. And indeed, Figure 13 shows that the proportion of time spent in compute operations is much higher than that seen for *DScan-mNW*. Moreover, the majority of the memory accesses are now to fast shared memory rather than to (slower) global memory. However, this change does introduce a new limiting factor: even in cases where many more cores are available per streaming processor, we cannot efficiently employ them due to the fixed amount of shared memory.

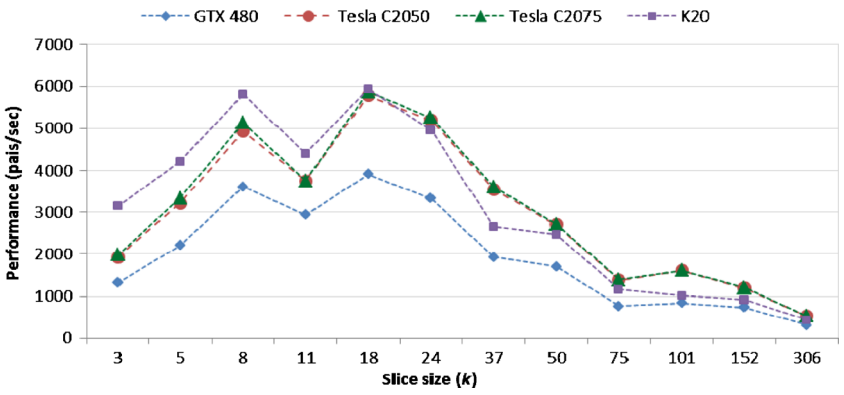
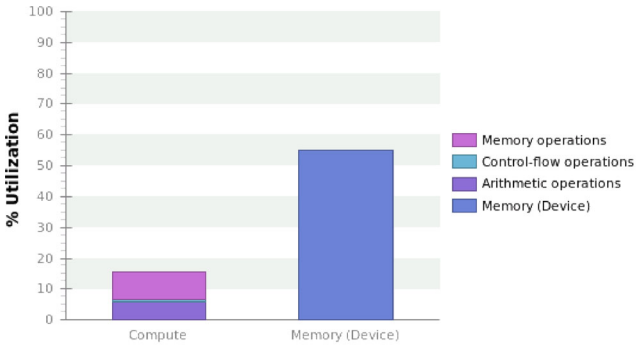
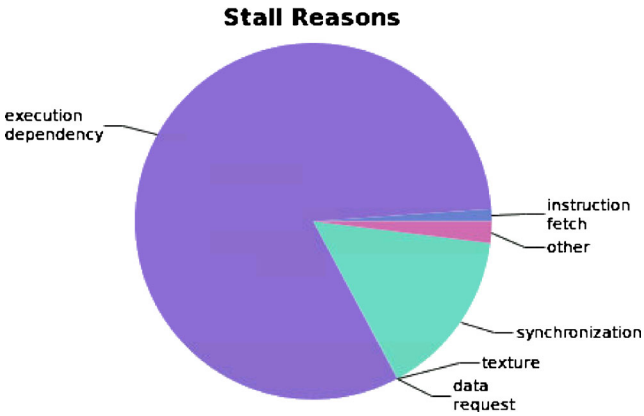


Figure 11 Effect of slice size k on performance of LazyRScan-mNW.



a. Compute and memory bounds for *DScan-mNW*.



b. Stall sources for *DScan-mNW*.

Figure 12 **a** Compute and memory bounds for *DScan-mNW*. **b**. Stall sources for *DScan-mNW*.

6.5 Performance Analysis on a Single Node, Using GPU Kernels as Filtering Methods

To evaluate our special case of computing all possible pairwise alignments but only retaining them if they meet a predefined percent identity threshold, we carried out performance tests on large-batch pairwise comparisons. These experiments differ from the ones presented in Section 6.3 in two ways. First, the dataset has a small number of sequences but leads to a large number of comparisons due to the large number of pairwise combinations. Second, after the alignment score is known, we need only perform the trace-back (on the CPU) to recover the actual alignment if a given sequence pair has an alignment score greater than the value of L above.

In the experiment, we created a FASTA file consisting of the first 200 sequences in our initial dataset (thus, making $200 \times 199/2 = 19900$ pairwise comparisons). The identity cut-off score is set to 97 % (a common value). Only sequence pairs that have identity ≥97 % are selected for performing higher-quality trace-back and alignment on the CPU. In our dataset, the number of such pairs is 6095. As such, with an ideal filter that takes no time to do computation, the expected speedup assuming that all pairs take the same time to align is $4950/6095 = 3.26x$.

Table 5 compares the performance gains obtained by using either *LazyRScan-mNW* or one of our traditional NW implementations (dual-buffering pinned-memory *DScan-mNW*) as a "filter" over the original method of performing all pairwise comparisons on the CPU. To ensure the fairness of the comparison, we have eliminated the matrix transfer operations from *DScan-mNW*. The experiment was done on Node-6 (Table 3). Note that this workstation is equipped with a single low-end GPU card (GeForce GTX 460).

Lazy*RScan-mNW* is the fastest implementation. As can be seen, both methods serve relatively well as a crude filter to avoid unnecessary alignments and offer a significant time saving even in the presence of a high-end CPU. However, the even greater performance improvement seen with *LazyRScan-mNW* is quite encouraging because it gives speedups that approach the maximum expected (3.26x, see above).

While *LazyRScan-mNW* is very applicable to our specific use case and provides significant speedup on a single node, it is also less flexible compared to the other three GPU methods, as the alignment is necessarily recomputed on the CPU (although we are currently studying mechanisms to efficiently perform trace-back on the GPU). Even with the trace-back not in place, the approach does still allow avoiding potentially expensive CPU operations or CPU/GPU communication and offers a considerable amount of time saving, proving to be a practical solution for important biological problems.

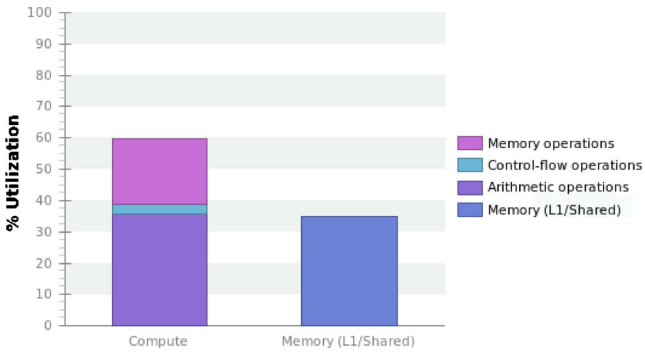


Figure 13 Compute and memory bounds for *LazyRScan-mNW*.

Table 5 Speedup on a single node as a result of using DScan-mNW and LazyRScan-mNW as a preliminary filter for sequence analysis.

Method	Action	Time (secs)	Overall (pairs/sec)
CPU Standalone	Alignment	2071.71	24.0
CPU + GPU DScan-mNW	Filtering	51.27	65.2
	Alignment	645.95	(2.71*x*)
CPU + GPU LazyRScan-mNW	Filtering	5.18	76.8
	Alignment	645.95	(3.2*x*)

6.6 Performance on CPU-GPU Clusters

In this section, we use the 16S rDNA gene dataset [21] to evaluate the performance and scalability of our distributed framework described in Section 5. Our evaluation consists of experiments on single node and on two clusters (see Table 3). In all experiments, we set the size of the work-lists sent by the CLD to the NLDs to 5,000 pairwise alignments; this figure allows full use of all available CPUs and GPUs. Load-balancing across nodes with different compute capabilities is automatically achieved by our pull-model: NLDs associated with slower nodes will request work-lists from the CLD at lower frequency. In these experiments we use our general-purpose GPU kernels and perform trace-back on CPU for all considered sequence pairs (that is, without filtering).

6.7 Experiments on Single Nodes

6.7.1 Heterogeneity

There are many levels of heterogeneity that we must consider in our framework. The lowest level occurs within a node. Taking one node in our high-end cluster as an example, *node-3* has 12 CPU cores and 3 different types of GPU: Tesla C2050, C2070 and C2075. These GPUs have the same number of CUDA cores with the same clock speed but different memory capacities. Our framework will read the GPUs' configuration in the initialization phase and set the appropriate parameters for launching the NW kernel and transferring data according to each GPU's memory capacity. In this way, the framework *load balances* between these GPUs. *Node-3* achieves 149 pairwise alignments/s (CPUs only), 359 pairs/s (GPUs only) and 487 pairs/s (CPUs+GPUs). Thus, the throughput when using both CPUs and GPUs increases roughly by a factor 3.3x and 1.4x compared to when using CPUs and GPUs alone, respectively. We conclude that our framework is able to handle the heterogeneity within nodes.

6.7.2 Vertical Scalability

Vertical scalability is another important metric for the system. It reflects the framework's ability to efficiently use added resources (e.g. RAM, CPUs, GPUs) to increase performance on a single node. Figure 14 shows the scalability of *node-4,* which has 8 CPUs and 4 GTX 480 GPUs. The x-axis is the number of GPUs while the y-axis shows processing speed (pairs/s). When we increase the number of GPUs from 1 to 3, the performance scales reasonably well from 313 pairs/s to 657 pairs/s (e.g., greater than 2x speedup). However, when the fourth GPU is employed, only a slight improvement is seen (36 extra pairs/s): a bottleneck has clearly been reached. One source of such bottleneck is the PCI-E controllers, each of which controls two PCI-E slots and hence two GPUs. Thus, when moving from one to two GPUs that share a controller, a performance gain of only 125 pairs/s is achieved. When the third GPU (with an independent controller) is added, performance increases by 219 pairs/s. Given that the fourth GPU competes in bandwidth with the third, we speculate that both the CPU process that dispatches work to these GPUs and their controllers may be becoming overloaded.

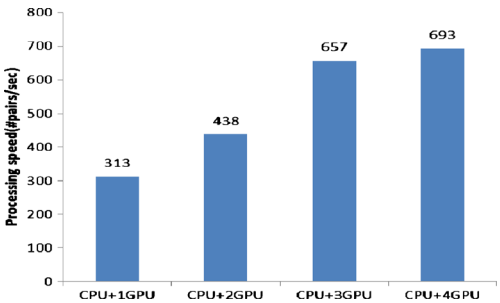


Figure 14 Vertical scalability on Node-4 from Table 2.

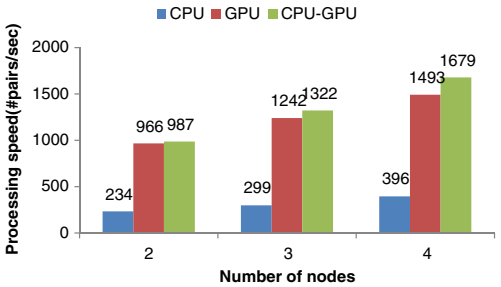


Figure 15 Performance on our high-end cluster.

6.8 Experiments on Clusters

The next level of heterogeneity in our cluster is the differences in CPU-GPU capacity between nodes. To explore this issue, we considered two clusters: (1) a high-end commodity cluster of four workstations with multiple GPUs per node and (2) a low-end cluster with six desktops and a single GPU per node.

6.8.1 High-end Cluster

Figure 15 shows the performance of the framework on the high-end cluster. The blue, red and green bars show the processing speed for CPU only, GPU only and CPU-GPU,

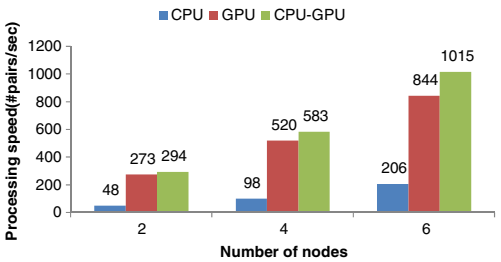


Figure 16 Performance on our low-end cluster.

respectively. When the framework only uses the available CPUs, the processing speed is between 200 and 400 pairs/s. When the GPUs are included, the processing speed jumps to 1000~1700 pairs/s (a 4 to 5-fold speedup). Surprisingly, the use of only the available GPUs has relatively little performance cost. Scalability is also reasonably good from one all the way to four nodes, with 250~350 pairs/s added per new node.

6.8.2 Low-end Cluster

Figure 16 presents similar results from the low-end cluster. Again, the addition of GPUs to CPUs in the framework yields a speedup of 5~6x over the CPU-only configuration. Interestingly, the high-end cluster is only 0.6x faster than the low-end one (although this limited difference is partly due to there being more total nodes on the low-end cluster). It is encouraging that even relatively modest hardware coupled with GPUs can dramatically improve the processing speed of sequence alignments.

7 Conclusion

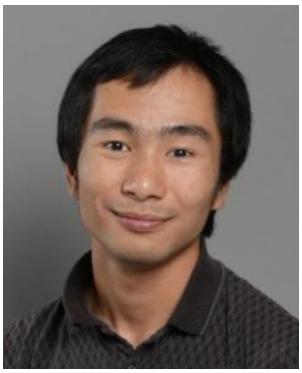
In this work, we have designed four implementations of multiple pairwise alignments using the Needleman-Wunsch algorithm on GPU. Three of our parallel kernels (*TiledDScan-mNW*, *DScan-mNW* and *RScan-mNW*) are general purpose. Our forth implementation (*LazyRScan-mNW)* is optimized for problems that require performing the trace-back operation only on a subset of the sequence pairs in the initial dataset (for example, the pairs whose alignment score exceeds a predefined threshold). We have highlighted how the different computational patterns affect the employment of the underlying hardware. We have integrated our general-purpose GPU kernels with an MPI framework for deployment on homogeneous and heterogeneous CPU-GPU clusters. We have evaluated our framework on a real-world dataset and on a variety of low-end and high-end Nvidia GPUs. Our experiments based on the general purpose *TiledDScan-mNW*, *DScan-mNW* and *RScan-mNW* kernels show a throughput in the order of 250 and 330 pairwise alignments/s on low- and high-end GPUs, respectively. In addition, we achieve a throughput of 1,015 pairwise alignments/s on a 6-node commodity cluster equipped with a low-end GPU on each node. Our *LazyRScan-mNW* kernel allows throughputs up to about 4,000 and 6,000 pairwise alignments/s on low- and high-end GPUs, respectively, and shows to be a very effective filtering method. Finally, we have performed an extensive experimental evaluation on the impact of the slice size on the performance of the Lazy*RScan-mNW* method on a variety of GPU devices with distinct compute capabilities (2.0, 2.1 and 3.5). Our results can be used in a production setup to tune the code to the underlying hardware.

Acknowledgments We thank the reviewers for their feedback. This work has been supported by NSF award CNS-1216756 and by equipment donations from Nvidia Corporation.

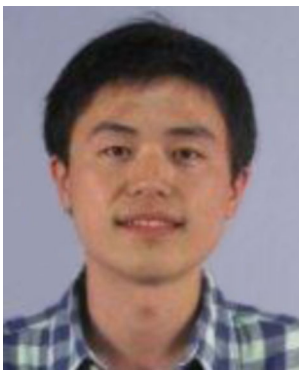
References

1. Needleman, S. B., & Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology, 48*, 443–453.
2. Smith, T. F., & Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology, 147*(1), 195–197.
3. Thompson, J. D., Higgins, D. G., & Gibson, T. J. (1994). CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice. *Nucleic Acids Research, 22*, 4673–4680.
4. Hillis, D. M., Moritz, C., & Mable, B. K. (1996). *Molecular systematics* (2nd ed.). Sunderland: Sinauer Associates.
5. Nei, M., & Gojobori, T. (1986). Simple methods for estimating the numbers of synonymous and nonsynonymous nucleotide substitutions. *Molecular Biology and Evolution, 3*(5), 418–426.
6. Pearson, W. R., & Lipman, D. J. (1988). Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America, 85*(8), 2444–2448.
7. Altschul, S. F., Gish, W., Miller, W., et al. (1990). Basic local alignment search tool. *Journal of Molecular Biology, 215*(3), 403–410.
8. Altschul, S. F., Madden, T. L., Schaffer, A. A., et al. (1997). Gapped blast and Psi-blast : a new-generation of protein database search programs. *Nucleic Acids Research, 25*(17), 3389–3402.
9. Li, H., Ruan, J., & Durbin, R. (2008). Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research, 18*(11), 1851–1858.
10. Langmead, B., Trapnell, C., Pop, M., et al. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology, 10*(3), R25.
11. Myers, G. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM), 46*(3), 395–415.
12. Benson, D. A., Cavanaugh, M., Clark, K., et al. (2013). GenBank. *Nucleic Acids Research, 41*(Database issue), D36–D42.
13. Meusemann, K., von Reumont, B. M., Simon, S., et al. (2010). A phylogenomic approach to resolve the arthropod tree of life. *Molecular Biology and Evolution, 27*(11), 2451–2464.
14. Pace, N. R. (2009). Mapping the tree of life: progress and prospects. *Microbiology and Molecular Biology Reviews, 73*(4), 565–576.
15. Parfrey, L. W., Grant, J., Tekle, Y. I., et al. (2010). Broadly sampled multigene analyses yield a well-resolved eukaryotic tree of life. *Systems Biology, 59*(5), 518–533.
16. Beja, O., Suzuki, M. T., Heidelberg, J. F., et al. (2002). Unsuspected diversity among marine aerobic anoxygenic phototrophs. *Nature, 415*(6872), 630–633.
17. Kim, M., Morrison, M., & Yu, Z. (2011). Status of the phylogenetic diversity census of ruminal microbiomes. *FEMS Microbiology Ecology, 76*(1), 49–63.
18. Tringe, S. G., & Rubin, E. M. (2005). Metagenomics: DNA sequencing of environmental samples. *Nature Reviews Genetics, 6*(11), 805–814.
19. Venter, J. C., Remington, K., Heidelberg, J. F., et al. (2004). Environmental genome shotgun sequencing of the sargasso Sea. *Science, 304*(5667), 66–74.

20. Whitford, M. F., Forster, R. J., Beard, C. E., et al. (1998). Phylogenetic analysis of rumen bacteria by comparative sequence analysis of cloned 16S rRNA genes. *Anaerobe, 4*(3), 153–163.
21. Cole, J. R., Wang, Q., Cardenas, E., et al. (2009). The ribosomal database project: improved alignments and new tools for rRNA analysis. *Nucleic Acids Research, 37*(Database issue), D141–D145.
22. Tarditi, D., Puri, S., & Oglesby, J. (2006). Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGARCH Comput. Archit. News, 34*(5), 325–335.
23. Che, S., Boyer, M., Meng, J., et al. (2009). "Rodinia: A benchmark suite for heterogeneous computing," in Proc. of IISWC, pp. 44–54.
24. "Nvidia Applications Catalog" http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf
25. Vouzis, P. D., & Sahinidis, N. V. (2010). GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics, 27*(2), 182–188.
26. Schatz, M. C., Trapnell, C., Delcher, A. L., et al. (2007). High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics, 8*, 474.
27. Walters, J. P., Meng, X., Chaudhary, V., et al. (2007). MPI-HMMER-boost: distributed FPGA acceleration. *The Journal of VLSI Signla Processing Systems for Signal, Image, and Video Technology, 48*(3), 6.
28. Pang, B., Zhao, N., Becchi, M., et al. (2012). Accelerating large-scale protein structure alignments with graphics processing units. *BMC Res Notes, 5*, 116.
29. Manavski, S. A., & Valle, G. (2008). CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics, 9*(Suppl 2), S10.
30. Liu, W., Schmidt, B., Voss, G., et al. (2007). Streaming algorithms for biological sequence alignment on GPUs. *IEEE Transactions on Parallel and Distributed Systems, 19*, 1270–1281.
31. Gao, Y., and Bakos, J. D. (2012). "GPU Acceleration of Pyrosequencing Noise Removal," in Proc. of SAAHPC, Argonne, IL USA, pp. 94–101.
32. Liu, Y., Maskell, D. L., & Schmidt, B., (2009). "CUDASW++: Optimizing Smith-Waterman Sequence Database Searches for CUDA-enabled Graphics Processing Units," *BMC Research Notes,* vol. 2, no. 73.
33. Wirawan, A., Kwoh, C. K., Hieu, N. T., et al. (2008). CBESW: sequence alignment on the playstation 3. *BMC Bioinformatics, 9*, 377.
34. Szalkowski, A., Ledergerber, C., Krahenbuhl, P., et al. (2008). SWPS3 - Fast multi-threaded vectorized Smith-Waterman for IBM cell/B.E. And x86/SSE2. *BMC Res Notes, 1*, 107.
35. Li, J., Ranka, S., & Sahni, S., (2012)."Pairwise sequence alignment for very long sequences on GPUs," in Proc. of ICCABS, pp. 1–6.
36. Li, K.-B. (2003). ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics, 19*(12), 2.
37. Biegert, A., Mayer, C., Remmert, M., et al. (2006). The MPI bioinformatics toolkit for protein sequence analysis. *Nucleic Acids Research, 34*, 5.
38. Henikoff, S., & Henikoff, J. G. (1992). Amino-acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences, U.S.A, 22*, 10915–10919.
39. Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Communications of the ACM, 18*(6), 341–343.
40. Myers, E. W., & Miller, W. (1988). Optimal alignments in linear space. *Computer applications in the biosciences: CABIOS, 4*(1), 11–17.
41. Sanders, J., & Jabdrot, E., (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*: Addison-Wesley Professional.



Huan Truong is currently a Ph.D. student in the Informatics Institute of University of Missouri-Columbia. He earned his B.Sc. degree in Computer Science from Truman State University, MO in 2011. He joined Gavin Conant's lab in 2012 and worked in collaboration with Michela Becchi's lab in 2013. His research interests include high performance computing and novel architectures with their application in biological networks and analysis.



Da Li is currently pursuing the Ph.D. degree in the Department of Electrical and Computer Engineering at University of Missouri-Columbia. His research interests include parallel computing, networking, algorithm acceleration and irregular applications. He received the B.E. degree from Beijing Institute of Technology (BIT), Beijing China, in 2011. He worked as research assistant in the Xilinx-BIT joint lab of high performance networking from July 2009 to May 2011.

Kittisak Sajjapongse is currently a Ph.D. candidate at University of Missouri-Columbia. He received a B.Eng. degree in electrical engineering from Mahidol University, Thailand in 2005 and M.S. degree in computer engineering from University of Missouri in 2010. His research interests include system software for parallel architecture and distributed processing in heterogeneous environments.

Gavin Conant received his Ph.D. from the University of New Mexico. He worked as a post-doctoral fellow at the University of Leipzig and Trinity College, Dublin. His is now an Assistant Professor in the Division of Animal Sciences and the Informatics Institute of the University of Missouri-Columbia. His research interests include metagenomics and the evolution of biological networks.

Michela Becchi received her M.S. and Ph.D. in Computer Engineering from Washington University in St. Louis. She is currently an Assistant Professor in the Electrical and Computer Engineering Department and the Informatics Institute of the University of Missouri-Columbia. Her research interests include high-performance processor architectures, networking systems, and algorithm acceleration on parallel computer architectures.