A Distributed CPU-GPU Framework for Pairwise Alignments on Large-Scale Sequence Datasets

Da Li[1], Kittisak Sajjapongse[1], Huan Truong[2], Gavin Conant[2,3], Michela Becchi[1,2]
[1]Dept. of Electrical and Computer Engineering, [2]MU Informatics Institute, [3]Division of Animal Sciences
University of Missouri
{dlx7f, ks5z9, hntfkb}@mail.missouri.edu, {conantg, becchim}@missouri.edu

***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 paper, we present a hybrid MPI-CUDA framework for computing multiple pairwise sequence alignments on CPU-GPU clusters. Our design targets both homogeneous and heterogeneous clusters with nodes characterized by different hardware and computing capabilities. Our framework consists of the following components: a *cluster-level dispatcher*, a set of *node-level dispatchers*, and a set of *CPU-* and *GPU-workers*. The cluster-level dispatcher progressively distributes work to the compute nodes and aggregates the results. The node-level dispatchers distribute alignment tasks to available CPUs and GPUs and perform dual-buffering to hide data transfers between CPU and GPU. CPU- and GPU-workers perform pairwise sequence alignments using the Needleman-Wunsch algorithm. We propose and evaluate three designs for these GPU workers, all of them outperforming the existing open-source implementation from the Rodinia Benchmark Suite.

Keywords—heterogeneous system; sequence alignment; GPU; cluster

I. 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]. Such acceleration 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 [11]. 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" [12-14]. 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 [15-19]. 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 [20], 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. As GPGPUs are increasingly becoming part of HPC clusters, we developed a distributed framework that can simultaneously leverage the computing power offered by multiple nodes comprising multi-core CPUs and many-core GPUs in various configurations.

Our contributions can be summarized as follows.

- We propose a MPI-CUDA framework for performing many pairwise alignments of large sequence datasets on heterogeneous CPU/GPU clusters.
- We explore different implementations of the Needleman-Wunsch (NW) algorithm on GPU. 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 evaluate our GPU implementations on 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 the considered devices.
- We evaluate our framework on a dataset of about 25,000 unique 16S rDNA genes from the Ribosomal Database [20]. Our experiments show a throughput in the order of 250 and 330 pairwise alignments/sec on low- and high-end GPUs, respectively. In addition, we achieve a throughput of 1,015 pairwise alignments/sec on a 6-node commodity cluster equipped with a low-end GPU per node.

II. Related Work

In recent years GPUs and other accelerator devices have been widely used to accelerate a variety of scientific applications from many domains [21-23]. In particular, a number of biological applications, including BLAST [24], hidden Markov models [25, 26] and structure comparisons [27], have been ported to GPU or FPGA architectures. Most relevant to our work are several sequence alignment algorithms implemented on GPU [22, 28-30]. In Section III, we will provide more background on one of these: the NW implementation in the Rodinia benchmark suite [22].

Among the alignment implementations, Liu et al., [31] 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 [28, 32, 33], their tool provides better performance guarantees for protein database searches. Li et al., [34] 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 [35] aligns multiple protein, RNA or DNA sequences in parallel using MPI. Biegert et al., [36] 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.

III. Background

A. 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; 37]. 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 $(m+1)*(n+1)$. Note that this matrix can be virtual: there are linear space memory implementations of the algorithm (e.g. Hirschberg's algorithm [38]). Each element in M is then computed according to equation (1).

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

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 [20]. 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 [38] are adopted, the trace-back is a linear-time operation accounting for a small fraction of the overall execution time. In Section IV, we focus on the computation of the alignment matrix.

B. 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.

C. Rodinia-NW: Needleman-Wunsch on GPU

The Rodinia benchmark suite [22] 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 every 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 finally copied back to the CPU. 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.

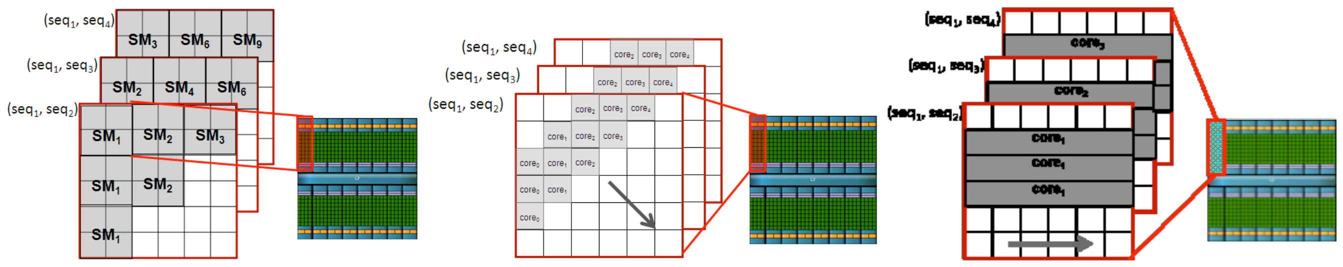
Finally, CUDA does not support global barrier synchronization among thread-blocks within a parallel kernel (an implicit global synchronization takes place at the end of each kernel execution). Since in Rodinia-NW each tile is mapped to a thread-block and tiles must be processed in diagonal strip manner, 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).

IV. Design of GPU-workers

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

A. TiledDScan-mNW: Multiple aligments with tiling

The first method (TiledDScan-mNW) is a directed 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 Figure 1(a), 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 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



(1-a) TiledDScan-mNW (1-b) DScan-mNW (1-c) RScan-mNW

Fig. 1. Exemplification of our 3 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, every alignment matrix is processed by a thread (and mapped onto a core).

and coalesced memory access patterns when storing alignment data from shared memory to global memory.

B. 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 Figure 1(b), 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. [31]. However, [31] 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 [31]) in two ways: first, it avoids the need for complex index translation; second, the more regular data layout leads to better caching properties.

C. 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 Figure 1(c).

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. At every 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 sections so as to fit into shared memory. The size of these sections is configurable. Large sections require more shared memory, which in turn limits the number of active threads on each SM. Small sections (e.g. sections with less than 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:

shmem = 3*sizeof(int)*BLOCK_SIZE*SECTION_SIZE

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 sections of size SECTION_SIZE. By setting the BLOCK_SIZE and the SECTION_SIZE to 32, we use 12KB of shared memory with

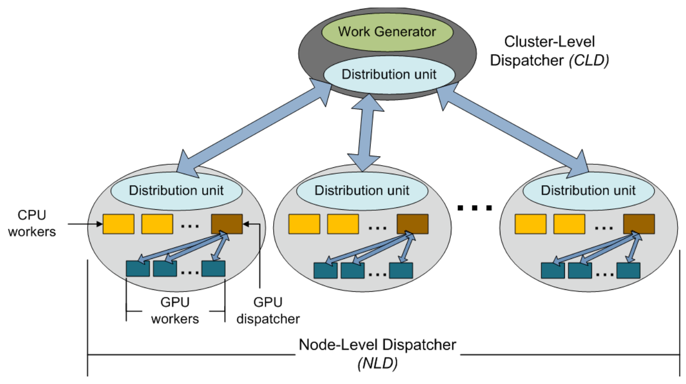


Fig. 2. Design of our distributed framework for CPU-GPU clusters.

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 32MB. 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. [31]. 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 [31], but is a necessary operation in the problem we consider.

V. Design of Our Distributed Framework

A. 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 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.

B. 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 2 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 of CLD and NLD is shown in Figure 3. 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.

C. Node-level architecture

The NLDs consume data from the CLD and dispatch them to CPU and GPU-level workers (Figure 2). 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

```
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
         traceback 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
```

Fig. 3. Pseudo-code of CLD and NLD

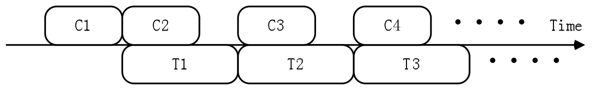


Fig. 4. Exemplification of Dual buffering.

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 IV. 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, which accounts 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 buffering and overlapping kernel execution with GPU-to-CPU data copy. A different choice has been made by Gao et al. [30], 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.

D. GPU-Dispatcher optmizations

Pinned memory - Generally there are three stages on a GPGPU computation. First, the initial data will be copied from the host to the device's global memory. Then, the CPU will launch the kernel, allowing the calculation of the results on the device. After this operation finishes, the data will be copied back to the host memory. With pageable memory, the memory copy operations contribute a significant fraction of the time of the GPGPU computation due to the mandatory involvement of the CPU, as memory allocated on the host may not be physically present and thus may require swapping. Pinned memory guarantees that the memory allocated is always physically present in the host's physical RAM. Pinned memory therefore can provide significant speedups to device-and-host memory transport, as the GPU can handle the transfer using its DMA hardware capabilities without the involvement of the CPU. This approach has two potential advantages. First, the copy operation on pinned memory is usually much faster than for pageable memory [39]. Second, the CPU only needs to pass the list of physical pages to the driver and is then free to do other work.

Table 1: Characteristics of the GPUs used in our evaluation

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

Double buffering - Double-buffering is a common scheme for reducing overhead by overlapping communication and computation. It requires that a non-blocking communication mechanism is 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. Double-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 double-buffering, we can further exploit the concurrency offered by the GPU, by interleaving the two operations (Figure 4). To implement double buffering, we used asynchronous pinned memory operations and CUDA streams.

VI. 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.

A. Experimental setup

Hardware setup – Single GPU experiments have been performed on a variety of low-end and high-end GPUs, listed in Table 1. Cluster level experiments have been performed on two cluster settings (a low-end and a high-end cluster), whose setups are summarized in Table 2.

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

Table 2: Cluster setup

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

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 [20]. The sequences are on average 1,536 bases long.

B. Performance on single GPU

Our first set of experiments is meant to evaluate our GPU implementations and compare them with Rodinia-NW. In Section III.C 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 III.C, 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 Figure 5 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 depending on the underlying GPU device. In Figure 6 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 reports poor performance. This, in general, holds when comparing long sequences on GPUs with 1-5GB device memory. Therefore, we focus on the other schemes.

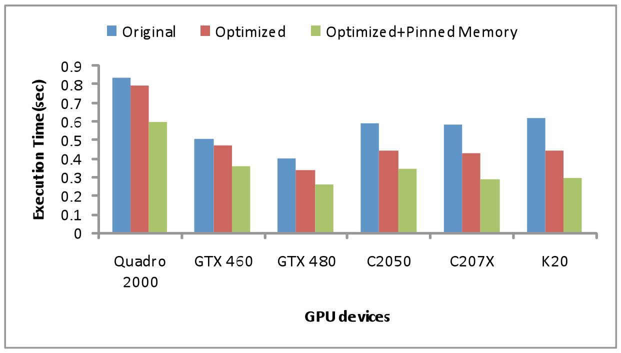
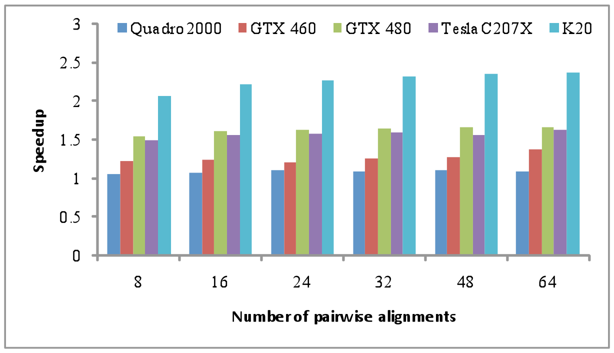
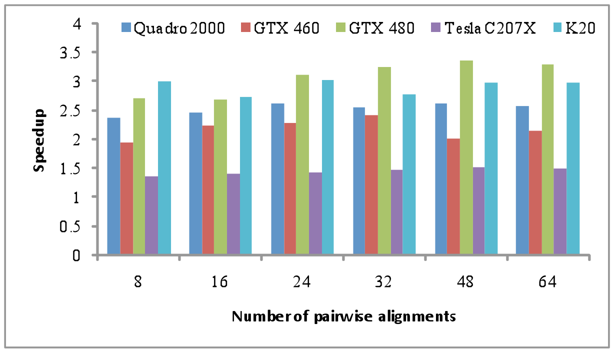


Fig. 5. Evaluation of memory optimizations on Rodinia-NW: original implementation, optimized implementation with initialization of the alignment matrices on GPU, optimized implementation with pinned memory.

Figure 6(a) reports the speedup of TiledDScan-mNW over Rodinia-NW. Note that TiledDScan-mNW performs fewer kernel calls (and therefore, has less kernel overhead) and



(6-a) TiledDScan-mNW



(6-b) DScan-mNW

Fig. 6. Kernel speedup over Rodinia-NW.

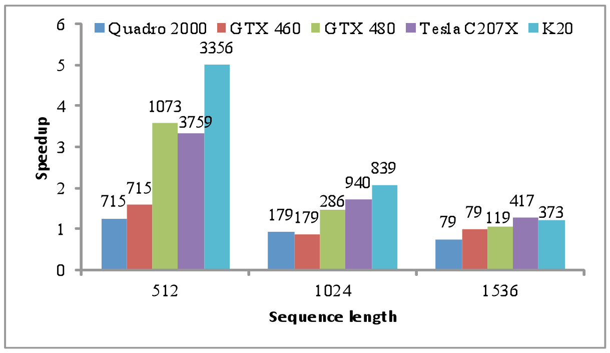


Fig. 7. Kernel speedup of RScan-mNW over Rodinia-NW on sequences of various lengths. The numbers on the bars represent the number of pairwise alignments on the device for each experiment.

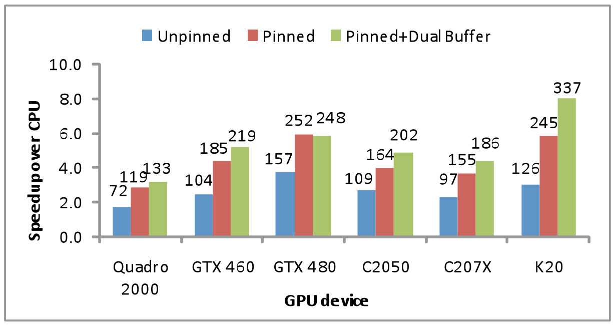


Fig. 8. Speedup reported by DScan-mNW over an 8-threaded CPU implementation running on the 8-core CPU on Node-4. The numbers on the bars represent the processing throughput (number of pairwise alignments/sec).

involves more per-kernel computation (thus leading to increased parallelism). This motivates 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 Figure 6(b), 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 C207x cards. It is somewhat surprising that our approach does not show substantial speedup over Rodinia-NW 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, but 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 motivate the poor performances on Tesla C207x cards, which have a slower memory clock.

Figure 7 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 to use 70% of the global memory capacity). As mentioned in Section IV.C, in RScan-mNW each core computes an alignment matrix: in order to fully utilize the computational resources of the GPU, RScan-mNW needs to perform a large number of parallel sequence alignments. This leads to pressure on the global memory capacity: for long sequences, the GPU global memory becomes the bottleneck, and the performance of RScan-mNW is penalized. RScan-mNW's performance improves when the length of the sequence decreases: in the case of shorter sequences, the global memory can accommodate more alignment matrices, thus leading to higher utilization of the GPU cores. In particular, on 512-base sequences, RScan-mNW gives a speedup over Rodinia-NW up to a factor 5x.

In general, Figure 6 and 7 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 inefficiencies of Rodinia-NW, and suggest that DScan-mNW is preferable on all devices except Tesla C207x. On such cards, TiledDScan-mNW provides better performance. This finding 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 utilize 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 C207x and K20 GPU, respectively.

Figure 8 shows the speedup reported by Dscan-mNW over an 8-threaded OpenMP implementation running on the 8-core CPU on Node-4 (see Table 2). The numbers on 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 double 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 3 b_{SIZE} in order to effectively time the computation. For double buffering, only half of the GPU memory performs alignments at one time, so 6 batches of size $b_{SIZE}/2$ were timed. The performance was measured as the number of sequence pairs compared per second.

As can be observed from Figure 8, switching to pinned memory offers a gain of roughly 1.6x, consistent with previous findings [39]. Not using the OS's virtual memory system could in principle limit the number of sequences that can be processed concurrently. However, our observation is that problem sizes are instead generally limited by the amount of physical memory on the GPU, so we do not consider this CPU-based memory limitation to be a significant disadvantage. The application of double-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, that does not allow the same level

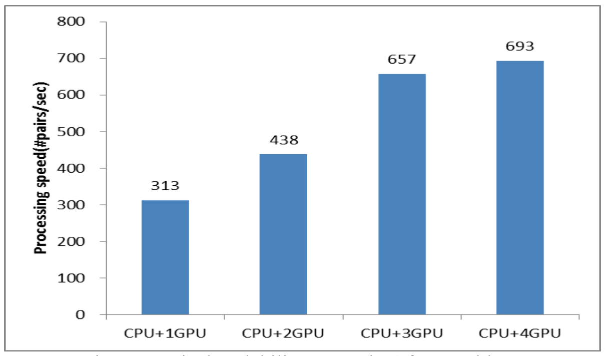


Fig. 9. Vertical scalability on Node-4 from Table 2.

of overlapping of memory transfers and kernel computations possible on other devices. In general, it can be observed that even cheap low-end GPUs (like the GTX460 and GTX480) offer throughput in the order of 200-250 pairwise comparison/sec.

C. Performance on CPU-GPU clusters

In this section, we use the 16S rDNA gene dataset [20] to evaluate our framework's performance and scalability. Our evaluation consists of experiments on single node and on two different clusters (see Table 2). In all experiments, we set the size of the work-lists sent by the CLD to the NLDs to 5,000 pairwise alignments; this allows full utilization 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.

1) Experiments on Single Nodes

a) 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/sec (CPUs only), 359 pairs/sec (GPUs only) and 487 pairs/sec (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.

b) 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 9 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/sec). When we increase the number of GPUs from 1 to 3, the performance scales reasonably well from 313 pairs/sec to 657 pairs/sec (e.g., greater than 2x speedup). However, when the fourth GPU is employed, only a slight improvement is seen (36 extra pairs/sec): 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/sec is achieved. When the third GPU (with an independent controller) is added, performance increases by 219 pairs/sec. 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.

2) 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.

a) High-end cluster

Figure 10 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, respectively. When the framework only uses the available CPUs, the processing speed is between 200 and 400 pairs/sec. When the GPUs are included, the processing speed jumps to 1000~1700 pairs/sec (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/sec added per new node.

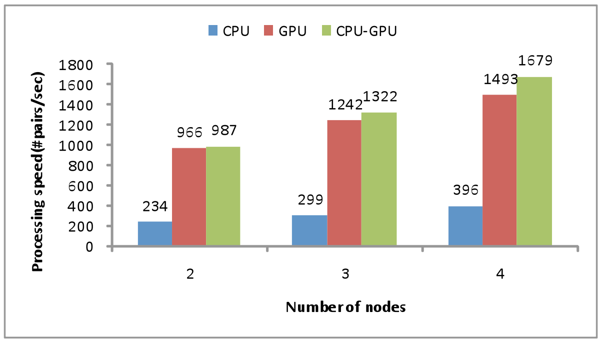


Fig. 10. Performance on our high-end cluster.

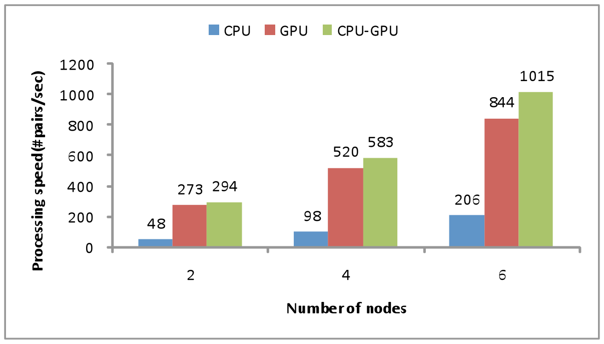


Fig. 11. Performance on our low-end cluster.

b) *Low-end cluster*

Figure 11 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.

VII. Conclusion

We have designed a distributed framework for parallel pairwise sequence alignments on CPU-GPU clusters and shown its efficiency on large bioinformatics datasets. We have explored three GPU-based NW implementations that support concurrent pairwise sequence alignments and have have integrated them into our framework. Our experiments show a throughput in the order of 250 and 330 pairwise alignments/sec on low- and high-end GPUs, respectively. In addition, we achieve a throughput of 1,015 pairwise alignments/sec on a 6-node commodity cluster equipped with a low-end GPU per node.

Acknowledgements

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] S. B. Needleman, and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J Mol Biol,* 48: 443-453, 1970.

[2] T. F. Smith, and M. S. Waterman, "Identification of common molecular subsequences," *J Mol Biol,* 147: 195-197, 1981.

[3] J. D. Thompson, D. G. Higgins, and T. J. Gibson, "CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice.," *Nucleic Acids Res,* 22: 4673-4680, 1994.

[4] D. M. Hillis, C. Moritz, and B. K. Mable, *Molecular Systematics: Second Edition*, Sunderland, MA: Sinauer Associates, 1996.

[5] M. Nei, and T. Gojobori, "Simple methods for estimating the numbers of synonymous and nonsynonymous nucleotide substitutions," *Mol Bio Evol,* 3: 418-426, 1986.

[6] W. R. Pearson, and D. J. Lipman, "Improved tools for biological sequence comparison," *Proc Natl Acad Sci U S A,* 85: 2444-8, 1988.

[7] S. F. Altschul, W. Gish, W. Miller *et al.*, "Basic Local Alignment Search Tool," *J Mol Biol,* 215: 403-410, 1990.

[8] S. F. Altschul, T. L. Madden, A. A. Schaffer *et al.*, "Gapped Blast and Psi-Blast : A new-generation of protein database search programs," *Nucleic Acids Res,* 25: 3389-3402, 1997.

[9] H. Li, J. Ruan, and R. Durbin, "Mapping short DNA sequencing reads and calling variants using mapping quality scores," *Genome Res,* 18: 1851-8, 2008.

[10] B. Langmead, C. Trapnell, M. Pop *et al.*, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biol,* 10: R25, 2009.

[11] D. A. Benson, M. Cavanaugh, K. Clark *et al.*, "GenBank," *Nucleic Acids Res,* 41: D36-42, 2013.

[12] K. Meusemann, B. M. von Reumont, S. Simon *et al.*, "A phylogenomic approach to resolve the arthropod tree of life," *Mol Bio Evol,* 27: 2451-64, 2010.

[13] N. R. Pace, "Mapping the tree of life: progress and prospects," *Microbiol Mol Biol Rev,* 73: 565-76, 2009.

[14] L. W. Parfrey, J. Grant, Y. I. Tekle *et al.*, "Broadly sampled multigene analyses yield a well-resolved eukaryotic tree of life," *Syst Biol,* 59: 518-33, 2010.

[15] O. Beja, M. T. Suzuki, J. F. Heidelberg *et al.*, "Unsuspected diversity among marine aerobic anoxygenic phototrophs," *Nature,* 415: 630-3, 2002.

[16] M. Kim, M. Morrison, and Z. Yu, "Status of the phylogenetic diversity census of ruminal microbiomes," *FEMS Microbiol Ecol,* 76: 49-63, 2011.

[17] S. G. Tringe, and E. M. Rubin, "Metagenomics: DNA sequencing of environmental samples," *Nat Rev Genet,* 6: 805-814, 2005.

[18] J. C. Venter, K. Remington, J. F. Heidelberg *et al.*, "Environmental genome shotgun sequencing of the Sargasso Sea," *Science,* 304: 66-74, 2004.

[19] M. F. Whitford, R. J. Forster, C. E. Beard *et al.*, "Phylogenetic analysis of rumen bacteria by comparative sequence analysis of cloned 16S rRNA genes," *Anaerobe,* 4: 153-63, 1998.

[20] J. R. Cole, Q. Wang, E. Cardenas *et al.*, "The Ribosomal Database Project: improved alignments and new tools for rRNA analysis," *Nucleic Acids Res,* 37: D141-5, 2009.

[21] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program GPUs for general-purpose uses," *SIGARCH Comput. Archit. News,* 34: 325-335, 2006.

[22] S. Che, M. Boyer, J. Meng *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in IEEE International Symposium on Workload Characterization, 2009, 44-54.

[23] "Nvidia Applications Catalog," http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf.

[24] P. D. Vouzis, and N. V. Sahinidis, "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," *Bioinformatics,* 27: 182-8, 2010.

[25] M. C. Schatz, C. Trapnell, A. L. Delcher *et al.*, "High-throughput sequence alignment using Graphics Processing Units," *BMC Bioinformatics*, 2007.

[26] J. P. Walters, X. Meng, V. Chaudhary *et al.*, "MPI-HMMER-Boost: Distributed FPGA Acceleration," *The Journal of VLSI Signla Processing Systems for Signal, Image, and Video Technology,* 48: 6, 2007.

[27] B. Pang, N. Zhao, M. Becchi *et al.*, "Accelerating large-scale protein structure alignments with graphics processing units," *BMC Res Notes,* 5: 116, 2012.

[28] S. A. Manavski, and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC Bioinformatics,* 9 Suppl 2: S10, 2008.

[29] W. Liu, B. Schmidt, G. Voss *et al.*, "Streaming algorithms for biological sequence alignment on GPUs," *TPDS,* 19: 1270-1281, 2007.

[30] Y. Gao, and J. D. Bakos, "GPU Acceleration of Pyrosequencing Noise Removal," in Symposium on Application Accelerators in High Performance Computing (SAAHPC), 2012, 94-101.

[31] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: Optimizing Smith-Waterman Sequence Database Searches for CUDA-enabled Graphics Processing Units," *BMC Research Notes,* 2, 2009.

[32] A. Wirawan, C. K. Kwoh, N. T. Hieu *et al.*, "CBESW: Sequence Alignment on the Playstation 3," *BMC Bioinformatics,* 9, 2008.

[33] A. Szalkowski, C. Ledergerber, P. Krahenbuhl *et al.*, "SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2," *BMC Res Notes,* 1, 2008.

[34] J. Li, S. Ranka, and S. Sahni, "Pairwise sequence alignment for very long sequences on GPUs," in Computational Advances in Bio and Medical Sciences (ICCABS), 2012 IEEE 2nd International Conference on, 2012, 1-6.

[35] K.-B. Li, "ClustalW-MPI: ClustalW analysis using distributed and parallel computing," *Bioinformatics,* 19: 2, 2003.

[36] A. Biegert, C. Mayer, M. Remmert *et al.*, "The MPI Bioinformatics Toolkit for protein sequence analysis," *Nucleic Acids Res,* 34: 5, 2006.

[37] S. Henikoff, and J. G. Henikoff, "Amino-acid substitution matrices from protein blocks," *Proc Natl Acad Sci U S A,* 89: 10915-10919, 1992.

[38] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM,* 18: 341-343, 1975.

[39] J. Sanders, and E. Jabdrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*: Addison-Wesley Professional, 2010.