Large-Scale Pairwise Sequence Alignments on a Large-Scale GPU Cluster

Ibrahim Savran, Yang Gao, and Jason D. Bakos
University of South Carolina

Editor's notes:
This paper presents design of a GPU kernel for performing pairwise sequence alignments for large-scale short sequence datasets generated by next-generation sequencers. This kernel principally performs batch Needleman–Wunsch global alignments. When used with its MPI-based host software, the kernel is scalable and is capable of achieving high throughput alignment when run on a CPU-GPU cluster.
—Partha Pratim Pande, Washington State University

Next-Generation Sequencing (NGS) technologies use parallel sequencing techniques to generate a large set of short DNA fragments. For example, the Roche 454 generates tens of thousands of 400–800 base pair sequences, while the Illumina MiSeq/HiSeq generates millions or billions of 150 base pair sequences. Because of this, NGS pipelines generally must rely on advanced computational tools to extract useful knowledge from the target genome or metagenome. One important class of analysis methods relies on clustering the DNA fragments based on a similarity metric. Although individual alignments are relatively inexpensive, computing pairwise distances requires $O(n^2)$ alignments, becoming prohibitively expensive for large data sets.

In this paper, we describe a graphics processing unit (GPU) kernel that performs batch Needleman–Wunsch (N–W) global alignments. For each alignment, the kernel returns an alignment score divided by the total alignment length. When used with its MPI-based host software, the kernel is scalable and is capable of achieving high-throughput alignment when run on a CPU–GPU cluster. The host software includes a load balancing technique for data sets having sequences of nonuniform lengths.

We evaluate our kernel using the Stampede supercomputer at the Texas Advanced Computing Center (Austin, TX, USA). By executing our kernel on 32 of Stampede's NVIDIA K20 GPUs, we were able to align 256 K (2^{18}) 400 base pair sequences, requiring over 34 billion individual alignments using 79 872 GPU cores.

GPU Computing

Since the introduction of the NVIDIA compute unified device architecture (CUDA) in 2008, GPU computing has become widely adopted by the scientific computing community. There are now several large-scale supercomputer installations equipped with GPU coprocessors, including Titan, currently ranked number one on the Top 500 list containing 18 688 NVIDIA K20× GPUs, Stampede, ranked number seven containing 128 NVIDIA K20 GPUs, and Tianhe-1A, currently ranked number eight containing 7168 NVIDIA M2050 GPUs [1].

Table 1 summarizes the differences in micro-architectural philosophy between NVIDIA GPUs and Intel Xeon CPUs. As compared to CPUs, GPUs devote

Digital Object Identifier 10.1109/MDAT.2013.2290116
Date of publication: 11 November 2013; date of current version: 20 February 2014.

Table 1 Comparison of Intel's largest scale CPU with GPUs used in our tests.

	Intel Xeon E7-8870	NVIDIA K20	NVIDIA GTX 680
Architecture	Westmere-EX	Kepler GK110	Kepler GK104
Processor cores	10	13	8
Threads/core	2 threads/core, superscalar speculative out of order	2048 threads/core, 8 SIMD instructions dispatched per cycle per core in program order	2048 threads/core, 8 SIMD instructions dispatched per cycle per core in program order
Clock rate	2.4 GHz	706 MHz	1.1 GHz
Memory bandwidth	42.6 GB/s	208 GB/s	192 GB/s
Transistors	2.6 billion	7.1 billion	3.5 billion
On chip memory	30 MB L3 cache	1.5 MB L2	512 KB L2
Thermal Dynamic Power	130 Watts	225 Watts	195 Watts

a larger portion of their real estate to functional units but have substantially less onchip cache. GPUs exploit data-level parallelism by interleaving instructions across a large set of active threads, while CPUs exploit instruction-level parallelism by maximizing the number of in-flight instructions from each thread. GPUs use smaller, nonexpandable DRAMs but have substantially higher memory bandwidth than CPUs.

Previous work

Knowledge of DNA sequences has become indispensable for biological research and in numerous applied fields such as diagnostic medicine, biotechnology, and biological systematics. The high demand for low-cost sequencing has driven the development of NGS technologies that parallelize the sequencing process, producing thousands or millions of short sequences at once.

One of the first such high-throughput sequencing technologies was developed by Roche in their 454 pyrosequencer, which provided a means of sequencing tens of thousands of short DNA sequences quickly and efficiently [2]–[8]. However, this technology brought new problems. The 454 data are prone to errors, leading to the realization that computational methods would be necessary to "denoise" the data.

One such tool, AmpliconNoise, is designed for metagenomic data sets and relies on grouping the input sequences into clusters that each represents oversampling of a single taxonomic unit [9]. The clustering is based on the pairwise distances given by the Needleman–Wunsch [10] global sequence alignment. The alignment used in AmpliconNoise differs from traditional sequence alignment in that it requires the use of double-precision floating-point operations when performing alignment score calculation. AmpliconNoise is used to denoise the 454 data in order to reduce the overestimation of the number of unique taxonomic units implied by the data, a common problem in metagenomic sequencing [11]–[19].

Unlike other GPU-based sequence alignment kernels such as CUDASW++ that employ a performance optimization in which the move matrix is discarded, in AmpliconNoise, this movement matrix must be retained in order to calculate the normalized alignment distance required for AmpliconNoise's final distance calculation.

There are several examples in the literature that describe GPU accelerated local and global sequence alignment algorithms such as Needleman–Wunsch, Smith–Waterman [20], and BLAST [21]. However, the emphasis of these efforts is on local sequence alignment for genomic database search, in which a relatively short sequence is aligned against a very long database sequence.

The Needleman–Wunsch alignment method used in AmpliconNoise is inherently more expensive than BLAST. As such, one obvious method for improving the performance of AmpliconNoise is to replace its alignment method with BLAST. This is especially tempting since BLAST has been

parallelized on both GPUs (as mentioned above) but also on traditional distributed memory parallel computers [22]. However, our objective is to achieve GPU acceleration of AmpliconNoise while guaranteeing that the results match those from our software baseline. The authors of AmpliconNoise emphasize accuracy and explicitly chose not to use the less expensive but less sensitive BLAST method because it would compromise the quality of their results. As such, we feel that comparing BLAST to a GPU-accelerated Needleman–Wunsch (N–W) is not a fair comparison.

pGraph is a load balancing method that is specifically designed to address the problem of higher variance in the lengths of protein sequences that is not typical of DNA sequences [23]. pGraph also includes a feature that reduces the number of required sequence comparisons by subjecting each pair to a similarity test before adding that alignment to the work queue. The similarity test is based on checking for long exact matches in the proposed sequence pair, which is implemented using a suffix tree-based filter. While such a filter would benefit our approach, it would violate AmpliconNoise's expectation that a distance be returned for all pairs. More importantly, although using such a filter would allow us to process larger data sets, it would not affect our achieved alignment throughput, which is our target performance metric.

Manavski provided the seminal work in accelerating Smith–Waterman using CUDA [24]. This early work has been improved upon in the development of more recent and more popular libraries such as CUDASW++ [25]. More recently, Razmyslovich has developed an OpenCL implementation of Smith–Waterman [26] that achieves three times the performance of CUDASW++ 2.0 in some situations [27].

To the best of our knowledge, the parallel implementation of Needleman–Wunsch that we used is the only that is sufficiently memory efficient to make it feasible to achieve our target scales, in terms of number of alignments. In other words, all the implementations from the literature, including GPU, FPGA, and MPI-based implementations, either do not provide an alignment length (only the score) or they perform a full traceback which requires large memory which makes them not amenable for massively large alignment workloads.

Needleman–Wunsch global alignment

The Needleman–Wunsch algorithm is a comparison operation between two sequences A and B given an implicit assumption that, when the sequences are not exactly equal, their similarity can be characterized as the number of edit operations that would transform one sequence into the other. Possible edit operations are character substitutions and substring insertions and deletions. The objective of an alignment is to align the matching or substituted characters that are common in both sequences and add blank spaces—or gaps—in one sequence that correspond to characters in the other sequence that are not common to both. The distance "penalty" that is contributed by each edit operation can be specified using a substitution table and a gap penalty.

The algorithm works by constructing two matrices, where each matrix has $k+1$ rows and $l+1$ columns, where k and l are the lengths of the two strings to be compared. The *score matrix* records the alignment score for every possible alignment between the two matrices, while the *movement matrix* provides a path through the matrix, from the bottom-right cell to the upper-left cell, that represents the alignment configuration that yields the minimal alignment score. In this path, a move to the left or up represents a gap that is inserted into the first or second sequence, while a move diagonally represents a matching or substituted character that is common to both sequences.

Each cell of the score and movement matrix is computed as shown in

$$H(i,j) = \min \begin{cases} H(i-1,j-1) + S(A[i], B[j]) \\ H(i-1,j) - d \\ H(i,j-1) - d \end{cases} \quad (1)$$

$$M(i,j) = \begin{cases} \text{diag if } H(i,j) = H(i-1,j-1) + S(A[i], B[j]) \\ \text{left if } H(i,j) = H(i-1,j) - d \\ \text{up if } H(i,j) = H(i,j-1) - d. \end{cases} \quad (2)$$

In these equations, $H(i,j)$ is the score matrix, $S(a,b)$ is the substitution penalty resulting from comparing element $A[i]$ and element $B[j]$, and d is the gap penalty. S and d are specific to the sequencer technology and are represented as

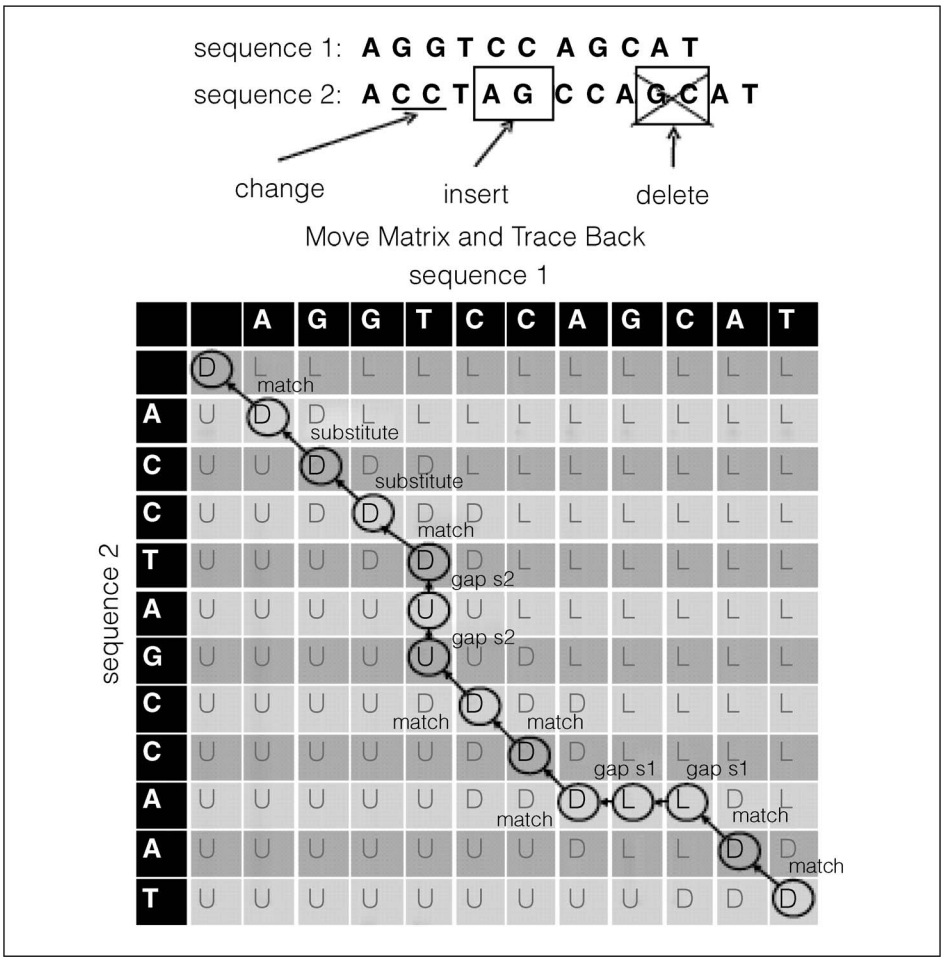

Figure 1. Example N–W alignment between two sequences.

floating-point values. In order to differentiate minor variations between flows, the authors of AmpliconNoise choose to use double-precision floating point to perform the comparisons, score accumulation, and score normalization.

Figure 1 shows an example N–W alignment. In this example, sequence 1 undergoes three edits to produce sequence 2. These sequences are then aligned. The final alignment score is taken from the lower right cell of the resultant score matrix (not shown). The move matrix is depicted in the figure and shows how characters present in sequence 1 but not in sequence 2 produce moves to the left, characters present in sequence 2 but not in sequence 1 produce moves up, and characters that match or are substituted produce a move diagonally. In this example, the final score is divided by the alignment length of 13 to compute the normalized score.

Space optimization

The generation of the two matrices represents a major challenge when performing large-scale batch alignments, as the memory requirement will often become a constraint well before the execution time. In our original kernel implementation, we store only the last row of the score matrix in GPU memory during the alignment procedure [28]. However, since AmpliconNoise must keep track of the total movement distance in order to use it to divide the alignment score, we originally stored the entire movement matrix in memory GPU. This was prohibitively expensive both in terms of memory space and time for performing the "traceback" operation on the movement matrix to determine the alignment length.

In our improved kernel, we store only one row of the movement matrix in GPU memory. In order to avoid storing the entire movement matrix, the kernel maintains only a single vector V, where $V[i]$ represents the accumulated number of minimal alignment moves beginning from the current row and from column i. In addition to this vector, we establish two registers ndist and ldist to hold intermediate values. ndist holds the newly computed number of moves, and ldist holds the previous number of moves from the left cell.

We later discovered that this technique is actually standard technique in dynamic programming despite the fact that it was not used in AmpliconNoise nor was it used in any of the GPU aligners in the literature. We assume this is because it is a relatively obscure technique as it is only applicable in situations where the alignment length is needed but not the recovered alignment itself, which may be a rare requirement (somewhat unique to AmpliconNoise).

Figure 2 depicts this operation. If the current move is determined to be diagonal, then we set $N = V[i-1] + 1$; if the current move is determined

to be left, then we set $N = L + 1$; and if the current move is determined to be up, then we set $N = V[i] + 1$. After this, we set $V[i-1] = L$, $L = N$, and increment i.

Arithmetic intensity

Algorithm 1 shows the version of the Needleman–Wunsch algorithm used in our kernel.

Algorithm 1: Single-Vector Needleman–Wunsch Alignment

```
Input: A, B, S
1 for i = 1 → length(A) do
2   ldist ← 0
3   leftScore ← i × gap_penalty
4   for j = 1 → length(B) do
5     currentScore ← min (
6       Score_{j-1} + S(A_i, B_j),
7       leftScore−d,
8       Score_j − d)
9     if currentScore = Score_{j-1} + S(A_i, B_j) then
10      ndist ← V_{j-1} + 1
11    else if currentScore = leftScore − d then
12      ndist ← ldist + 1
13    else
14      ndist ← V_j + 1
15    end if
16    Score_{j-1} ←leftScore
17    leftScore ← currentScore
18    V_{j-1} ← ldist
19    ldist← ndist
20  end for
21  Score_{j-1} ← leftScore
22  V_{j-1} ← ldist
23 end for
24 return Score_{length(B)}/V_{length(B)}
```

The innermost loop performs all the operations required to calculate a single cell of both the score and the movement matrices. This requires nine double-precision floating-point operations when computing each cell. The algorithm performs the following memory operations when computing each cell.

- Line 6: load 1 B for B_j (recall that A_j is stored in shared memory).

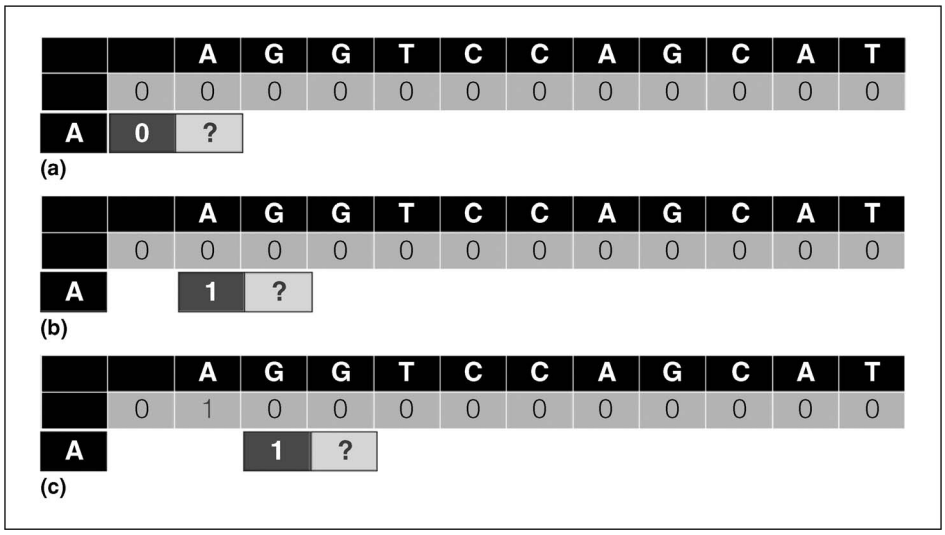

Figure 2. Example showing the movement matrix optimization.

- Lines 6, 8: 16 B loaded from the double-precision score array.
- Lines 10, 12, and 14: only one of these lines is executed, but on two of the lines 4 B are read from the V (distance) array. With our random input data, each line will be executed with equal probability. However, in the most common case, each warp (group of 32 threads) will include at least one thread that executes line 10 and one thread that executes line 14. This will cause warp serialization, subjecting all the threads in the warp to the latency required incurred by all of three branches. Thus, we assume that 8 B will always be read.
- Line 16: 8 B written to the double-precision score array.
- Line 18: 4 B written to the V (distance) array.

In total, each cell update requires accessing 37 B on average. Since our test sequences are of length 400, each alignment requires $400^2 = 160\,000$ cell updates.

Note that when updating the score vector *Score*, writes to this vector must be delayed until after the previous value is no longer needed. *leftScore* is used as a temporary holding place for this updated score before it can be written back to the vector. *currentScore* is a temporary holding place for the current score, which is shifted into *leftScore* for the next iteration. In other words, *leftScore* is the updated score to the left of the current score and *currentScore* is the new value of the current score, but these

values must be maintained in temporary variables since *currentScore* depends on the scores above and to the left from the previous iteration. The alignment distance vector V is treated the same way.

If we begin with the assumption that our kernel is compute bound, and we consider that each of the 13 streaming multiprocessor cores (SMXs) on the NVIDIA K20 can dispatch 128 double-precision operations per cycle, we should be able to achieve a throughput of 1/160 000 alignment/cells × 1/9 cell/ops × (13 × 128) ops/cycle × 706e6 cycles/second = 815 820 alignments/second per GPU.

On the other hand, given our kernel's arithmetic intensity of 37 B per cell, if we assume that the kernel is memory bound, we expect to be able to compute 1/37 cell/B × 1/160 000 alignment/cells × 208 GB/s = 37 726 alignments/second per GPU. Since the second throughput is lower, we conclude that the kernel is actually memory bound, and that this kernel utilizes only 37 726/815 820 = 4.6% of the GPU's computational capability.

In order to compute the pairwise distances among sequences, an input data set with n sequences will perform a Needleman–Wunsch alignment $(n2 - n)/2$ times to construct matrices. For a data set of 2^{18} sequences, there are approximately 34 billion required alignments, which would ideally require 474 min on 32 K20 GPUs.

Kernel implementation

In general, performance tuning GPU kernel code requires that the programmer apply code transformations to maximize the utilization of both GPU resources and memory bandwidth, and a key objective for achieving both these goals is to maximize the number of active threads.

As shown in Figure 3, existing Smith–Waterman and Needleman–Wunsch alignment kernels (including CUDASW++) use a strategy where they employ multiple threads to generate each diagonal of a single score and movement matrix. This is possible since cells along the diagonal can be computed independently. This is an effective strategy for kernels that perform one alignment at a time, since many threads can be utilized during the alignment.

There are two drawbacks to this approach. When assigning one thread per cell in the matrix diagonal, in order to access memory in a coalesced way, the matrices must be organized in a diagonal-major order as opposed to row- or column-major order. Organizing the matrix in this way adds overhead for translating row and column pairs into memory addresses. In addition, there are several "corner cases" that must be considered that require additional conditional execution paths (i.e., if-statements) which cause branch divergences among the threads and degrading performance. Both of these issues also lead to higher register usage per thread that limits the kernel's occupancy, or the number of threads that can execute simultaneously on each of the GPU's processor cores.

As shown in Figure 4, since AmpliconNoise performs a large number of short alignments, we observe that a more effective strategy is to assign one thread to each alignment operation and to perform a large set of alignments in parallel. As compared to the more traditional approach of computing the score matrix diagonals in parallel, this technique leads to fewer divergent branches and allows for lower register usage allowing more threads to be invoked.

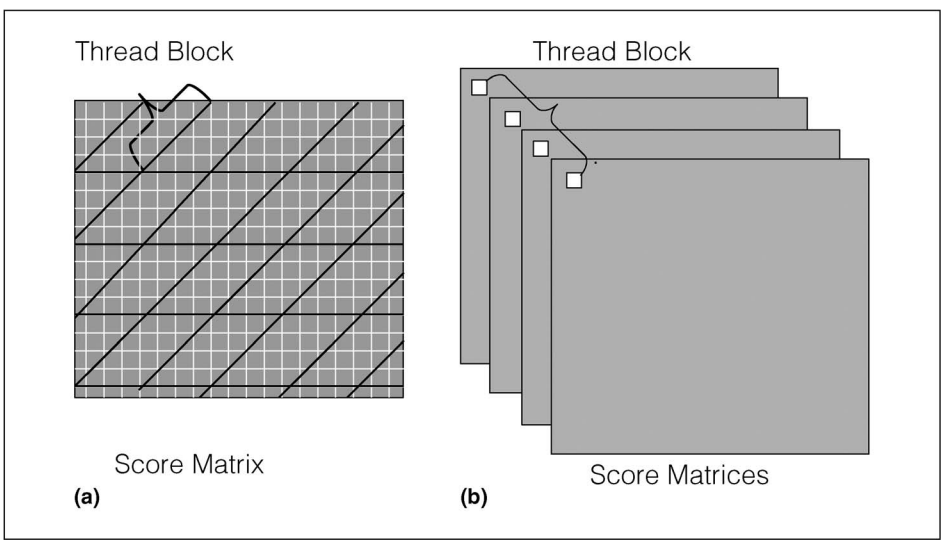

Figure 3. Two data parallel strategies for constructing the N–W matrices: (a) method used in implementations that seek to speed up single alignments, and (b) method used in our approach, which seeks to speed up multiple alignments.

CUDASW++, unlike Razmyslovich's implementation, provides an option for performing multiple alignments in parallel on the GPU but does not provide traceback capability for determining the length of the alignment. Razmyslovich's work, on the other hand, does provide an option for performing traceback, but when enabled, the overall performance is reduced by a factor of 10, as compared to when traceback is not enabled.

Task parallelization

In AmpliconNoise, the Needleman–Wunsch kernel is launched for sequence to sequence alignment. In order to take advantage of the GPU's parallel computing ability, however, we divide the sequences into groups where each group has a predefined number of sequences. Instead of computing the distance one by one, we launch a thread grid to compute all the possible distances between the sequences from each pair of groups.

A single thread performs the construction of one score and move matrix. In order to achieve coalesced memory access of a block of threads, we interleave each group of score and movement matrices, where the group size is 32 to match the warp size. As such, the addresses from 0 to 31 store the first values of 32 matrices; the next block of 32 addresses store the second values of each of 32 matrices; and so forth.

Shared memory optimizations

Using shared memory is a common optimization for increasing the performance of memory bandwidth-bound kernels. In our kernel, we use shared memory to store the substitution matrix in order to lower the number of global loads and stores. In addition, since one thread block is aligning a group of sequences ($B_j - B_{j+\text{Group Size}}$ to sequence A_i), we store A_i in the shared memory to reduce the bandwidth usage by the block.

Multi-GPU implementation and performance overhead

Since each individual alignment is independent, the host can assign each GPU a workload consisting of a subset of the alignments in order to parallelize the pairwise alignments across multiple GPUs. In our multi-GPU implementation, we divide the workload across each GPU using MPI.

When the GPU kernel is invoked, it performs pairwise comparisons between blocks of 32 sequences each. Since each thread performs one alignment, this requires the instantiation of $32^2 = 1024$ threads on the GPU. Each time the kernel in invoked, it can perform up to 128 of these pairwise block comparisons.

Each cluster node receives a set of work units. Each work unit is identified as a starting sequence block C_i and requires the pairwise alignment between blocks C_i and C_{i-1}, C_i and C_{i-2}, C_i and $C_{i-3}, \ldots, C_i$ and C_0. This requires C_i sequence block alignments and, therefore, ceiling (C_i/128) kernel invocations. In other words, each GPU invokes the kernel multiple times but the number of invocations depends on which sequence block is being computed. The number of invocations per GPU can grow quite large with large data sets (thousands of invocations per GPU). This causes the CUDA runtime to add overhead to the execution time.

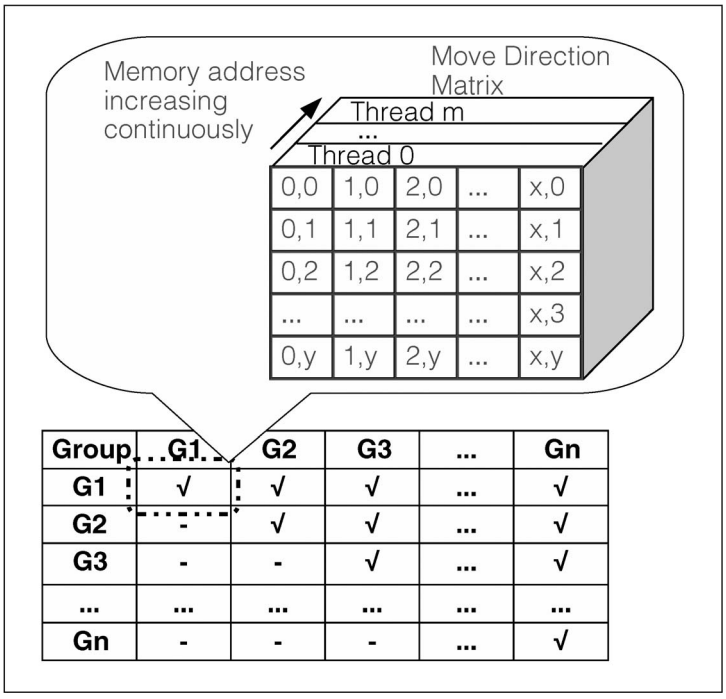

Figure 4. Group assignment and memory arrangement. The ticked cell means we should do the distance computation between the two groups, and for dash we do nothing. The distances between the flows within each group are also computed.

Single GPU performance results

Each node in the TACC Stampede cluster contains dual 2.7-GHz eight-core Intel Xeon E5-2680 CPUs that can all execute 16 MPI processes. Our first set of experimental results seeks to determine

Table 2 CPU versus single GPU execution time in seconds.

Cluster processes or GPU	Execution Time for 8192 Sequences	Execution Time for 6144 Sequences
32	2479	1394
64	1241	698
128	620	698
256	620	349
512	310	175
1024	155	87
GTX 680	1118	634

how many of these cluster nodes are equivalent, in performance, to a single NVIDIA GTX680 GPU for performing a set of pairwise alignments.

Table 2 shows the performance results from aligning, using Stampede's CPUs only, 8K and 6K sequences on 2–64 nodes, and on all 16 processors on each node. Our CPU implementation uses the same optimized algorithm as described in Algorithm 1, which itself is approximately six times faster than the base implementation in AmpliconNoise due to the movement vector optimization. We used the base MPI implementation in AmpliconNoise which distributes the workload uniformly across all nodes. Note that the speedup is nearly ideal as we scale to larger numbers of processors, except for the case when scaling from 128 to 256 processors for the 8K data set and from 64 to 128 processors in the 6K data set. We assume that this is due to communication overheads related to the placement of the MPI processes on the cluster.

We also ran the same data sets using a single NVIDIA GTX 680 GPU (a $500 gaming card). For both data sets, the GPU is approximately equivalent to 64 processors on TACC Stampede.

The thermal design power (TDP) for the NVIDIA GTX 680 is 195 W, while the TDP for one of Stampede's 8-core Intel Xeon E5-4650 CPUs is 130 W. This gives an approximate power consumption of $8 \times 130 = 1040$ W for the CPUs versus 195 W for the GPU, which makes the GPU approximately five times more power efficient.

Multi-GPU performance results

Stampede has 128 nodes that all contain one NVIDIA K20 GPU. For our multi-GPU experiments, we scaled our GPU kernel up to 32 NVIDIA K20s on Stampede using data sets that ranged from 16K to 256K sequences. Our XSEDE allocation for Stampede limits our GPU runs to 12 h, which in turn limits the maximum data set size that we could test. For this test, Stampede's CPUs remained idle while the GPUs executed the alignment kernel.

As shown in Table 3, for each test, we calculated the ideal performance using the memory bandwidth bound derived in the previous section. We calculated the parallelization overhead as the difference between the ideal and actual execution time

Table 3 Multi-GPU runtimes in minutes.

Number of GPUs	Execution time (minutes)	2^{14} (16K) sequences 1.34×10^8 alignments	2^{15} (32K) sequences 5.37×10^8 alignments	2^{16} (64K) sequences 2.15×10^9 alignments	2^{17} (128K) sequences 8.59×10^9 alignments	2^{17} (256K) sequences 3.44×10^{10} alignments
4 x K20	ideal	15	59	237		
	actual	16	66	266		
	overhead	10%	12%	12%		
8 x K20	ideal		30	119	474	
	actual		32	131	531	
	overhead		8%	11%	12%	
16 x K20	ideal		15	59	237	
	actual		16	64	261	
	overhead		11%	7%	10%	
32 x K20	ideal			30	119	474
	actual			30	126	524
	overhead			1%	7%	10%

divided by the ideal execution time. In this case, the parallelization overhead mostly consists of file I/O for the results. The overheads were consistent, from 22% to 25%, and in general were smaller for larger jobs.

Table 4 GPU utilization for nonuniform data sets (16K sequences).

$$\text{Utilization} = \frac{\sum_{i=1}^{p} runtime_i}{p \cdot \max_i runtime_i}$$

	$\sigma^2 = 50$		$\sigma^2 = 75$		$\sigma^2 = 100$	
# nodes/GPUs	naïve	improved	naïve	improved	naïve	improved
4	95.1%	99.5%	95.2%	99.3%	91.8%	99.5%
8	96.4%	99.4%	97.9%	99.4%	96.1%	99.1%
16	91.0%	98.0%	90.2%	97.2%	90.0%	97.1%

GPU load balancing

Our main experimental results assume a fixed sequence size of 400 bp, which is typical of data sets generated in metagenomic shotgun sequencing. Because of the uniformity of the sequence length and thus the alignment workload, there is little work imbalance when using a uniform workload distribution mechanism.

In order to characterize the load imbalance that occurs when the sequences have varying length, we have added a new set of results that measure the GPU utilization for synthetically generated data sets of 16K sequences where the length of each sequence is chosen from a normal distribution having mean 400 while scaling the standard deviation.

Our results show that the utilization does scale with the variance in the sequence length. To address this problem, we developed an improved workload distribution method. In our improved method, we consider each group of 32 consecutive sequences as a group, which corresponds to the number of parallel alignments performed on our GPU kernel, and we assign a block of groups to each GPU based on the estimated cumulative execution time of the groups, as opposed to a naive method where an equal number of alignments is assigned to each GPU.

We estimate the execution time of each group of sequences by observing that the time required for each block is determined by its longest sequence, due to the implicit barrier synchronization after each kernel invocation. Also, each block is aligned against all blocks having an index less than its own index, so this is incorporated into the estimate.

For each group i, we estimate the execution time C_i using

$$C_i = \begin{cases} \frac{m_0}{\mu}, & \text{if } i = 0 \\ \frac{m_i}{\mu} \times \frac{1}{\mu} \times \sum_{0}^{i-1} m_j, & \text{otherwise.} \end{cases} \quad (3)$$

In (3), m_j is the maximum sequence length within group j. We divide the cost of each group by the mean μ of the sequence length (in this case 400) in order to prevent overflow.

In our improved workload distribution method, we first use (3) to estimate the execution time of each group, and then sort the groups according to this value. After this, we assign the sorted groups to the GPUs in a round-robin manner.

As shown in Table 4, our improved workload distribution method consistently achieves 97% or higher utilization.

In this article, we describe our implementation of the normalized floating-point Needleman–Wunsch kernel used in AmpliconNoise, a tool for denoising metagenomic sequences obtained from NGS technology. We compared the performance of a single gaming GPU versus that of a large-scale CPU cluster, and then scaled the kernel to a large set of computing GPUs. We believe that our final result, aligning 256K sequences, is a new record in pairwise alignments, and demonstrates that GPU clusters can be a useful tool for NGS data analysis. ■

Acknowledgment

This work was supported by the National Science Foundation under Grant 0844951. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which was supported by the National Science Foundation under Grant OCI-1053575.

References

[1] *Top 500 List*. [Online]. Available: http://www.top500.org

[2] S. Kumar, T. Carlsen, B. Mevik, P. Enger, R. Blaalid, K. Shalchian-Tabrizi, and H. Kauserud, "CLOTU:

An online pipeline for processing and clustering of 454 amplicon reads into OTUs followed by taxonomic annotation,'' *BMC Bioinf.*, vol. 12, no. 182, 2011, DOI: 10.1186/1471-2105-12-182.

[3] J. R. Cole, Q. Wang, E. Cardenas, J. Fish, B. Chai, R. J. Farris, A. S. Kulam-Syed-Mohideen, D. M. McGarrell, T. Marsh, G. M. Garrity, and J. M. Tiedje, ''The ribosomal database project: Improved alignments and new tools for rRNA analysis,'' *Nucleic Acids Res.*, vol. 37, no. Suppl 1, pp. D141–D145, 2009.

[4] H. Amber, R. Sean, M. Timothy, L. Bertram, and E. Jonathan, ''Introducing WATERS: A workflow for the alignment, taxonomy, ecology of ribosomal sequences,'' *BMC Bioinf.*, vol. 11, no. 317, 2010, DOI: 10.1186/1471-2105-11-317.

[5] P. Ram, N. Viola, and S. Christian, ''CANGS: A user-friendly utility for processing and analyzing 454 GS-FLX data in biodiversity studies,'' *BMC Res. Notes*, vol. 3, no. 3, 2010, DOI: 10.1186/1756-0500-3-3.

[6] F. Juan, L. Antonio, F. No, C. Francisco, P. Guillermo, and C. Gonzalo, ''SeqTrim: A high-throughput pipeline for preprocessing any type of sequence read,'' *BMC Bioinf.*, vol. 11, no. 38, 2010, DOI: 10.1186/1471-2105-11-38.

[7] J. G. Caporaso, J. Kuczynski, J. Stombaugh, K. Bittinger, F. D. Bushman, E. K. Costello, N. Fierer, A. Gonzalez Peña, J. K. Goodrich, J. I. Gordon, G. A. Huttley, S. T. Kelley, D. Knights, J. E. Koenig, R. E. Ley, C. A. Lozupone, D. McDonald, B. D. Muegge, M. Pirrung, J. Reeder, J. R. Sevinsky, P. J. Turnbaugh, W. A. Walters, J. Widmann, T. Yatsunenko, J. Zaneveld, and R. Knight, ''QIIME allows analysis of high-throughput community sequencing data,'' *Nature Methods*, vol. 7, no. 5, pp. 335–336, 2010.

[8] C. Quince, A. Lanzn, T. Curtis, R. Davenport, N. Hall, I. Head, L. Read, and W. Sloan, ''Accurate determination of microbial diversity from 454 pyrosequencing data,'' *Nature Methods* vol. 6, no. 9, pp. 639–641, 2009.

[9] C. Quince, A. Lanzen, R. Davenport, and P. Turnbaugh, ''Removing noise from pyrosequenced amplicons,'' *BMC Bioinf.*, vol. 12, no. 38, 2011, DOI: 10.1186/1471-2105-12-38.

[10] S. Needleman and C. Wunsch, ''A general method applicable to the search for similarities in the amino acid sequence of two proteins,'' *J. Molecular Biol.,* vol. 48, no. 3, pp. 443–453, 1970.

[11] E. Hall, K. Besemer, L. Kohl, C. Preiler, K. Riedel, T. Schneider, W. Wanek, and T. Battin, ''Effects of resource chemistry on the composition and function of stream hyporheic biofilms,'' *Front. Microbiol.*, vol. 3, no. 35, 2012, DOI: 10.3389/fmicb.2012.00035.

[12] L. Tranvik and T. Battin, ''Unraveling assembly of stream biofilm communities,'' *ISME J.*, vol. 6, no. 8, pp. 1459–1468, 2012.

[13] P. Kumar, M. Brooker, S. Dowd, and T. Camerlengo, ''Target region selection is a critical determinant of community fingerprints generated by 16S pyrosequencing,'' *PloS One*, vol. 6, no. 6, 2011, e20956.

[14] A. Lanzen, S. Jørgensen, M. Bengtsson, I. Jonassen, L. Øvreås, and T. Urich, ''Exploring the composition and diversity of microbial communities at the Jan Mayen hydrothermal vent field using RNA and DNA,'' *FEMS Microbiol. Ecol.*, vol. 77, no. 3, pp. 577–589, 2011.

[15] R. H. Nilsson, L. Tedersoo, B. D. Lindahl, R. Kjøller, T. Carlsen, C. Quince, K. Abarenkov, T. Pennanen, J. Stenlid, T. Bruns, K. H. Larsson, U. Kőljalg, and H. Kauserud, ''Towards standardization of the description and publication of next-generation sequencing datasets of fungal communities,'' *New Phytologist*, vol. 191, no. 2, pp. 314–318, 2011.

[16] G. Wang, S. Sherrill-Mix, K. Chang, C. Quince, and F. Bushman, ''Hepatitis C virus transmission bottlenecks analyzed by deep sequencing,'' *J. Virol.*, vol. 84, no. 12, pp. 6218–6228, 2010.

[17] D. Knights, E. Costello, and R. Knight, ''Supervised classification of human microbiota,'' *FEMS Microbiol. Rev.*, vol. 35, no. 2, pp. 343–359, 2011.

[18] J. Bahl, M. C. Y. Lau, G. J. D. Smith, D. Vijaykrishna, S. C. Cary, D. C. Lacap, C. K. Lee, R. Thane Papke, K. A. Warren-Rhodes, F. K. Y. Wong, C. P. McKay, and S. B. Pointing, ''Ancient origins determine global biogeography of hot and cold desert cyanobacteria,'' *Nature Commun.*, vol. 2, 2011, DOI: 10.1038/ncomms1167.

[19] A. Gobet, C. Quince, and A. Ramette, ''Multivariate cutoff level analysis (MultiCoLA) of large community data sets,'' *Nucleic Acids Res.*, vol. 38, no. 15, 2010, DOI: 10.1093/nar/gkq545.

[20] T. F. Smith and M. S. Waterman, ''Identification of common molecular subsequences,'' *J. Mol. Biol* vol. 147, no. 1, pp. 195–197, 1981.

[21] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, ''Basic local alignment search tool,'' *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, 1990.

[22] C. Oehmen and J. Nieplocha, "ScalaBLAST: A scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis," *IEEE Trans. Parallel Distrib. Syst.,* vol. 17, no. 8, pp. 740–749, Aug. 2006.

[23] C. Wu, A. Kalyanaraman, and W. R. Cannon, "pGraph: Efficient parallel construction of large-scale protein sequence homology graphs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 10, pp. 1923–1933, Oct. 2012.

[24] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC Bioinf.*, vol. 9, no. Suppl 2, 2008, DOI: 10.1186/1471-2105-9-S2-S10.

[25] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: Optimizing Smith-Waterman sequence database searches for CUDA enabled graphics processing units," *BMC Res. Notes*, vol. 2, no. 73, 2009, DOI: 10.1186/1756-0500-2-73.

[26] D. Razmyslovich, G. Marcus, M. Gipp, M. Zapatka, and A. Szillus, "Implementation of Smith-Waterman algorithm in OpenCL for GPUs," in *Proc. 9th Int. Workshop Parallel Distrib. Methods Verif./2nd Int. Workshop High Performance Comput. Syst. Biol.*, 2010, pp. 48–56.

[27] Y. Liu, B. Schmidt, and D. Maskell, "CUDASW++ 2.0: Enhanced Smith-Waterman protein database search on CUDAenabled GPUs based on SIMT and virtualized SIMD abstractions," *BMC Res. Notes*, vol. 3, no. 93, 2010, DOI: 10.1186/1756-0500-3-93.

[28] Y. Gao and J. D. Bakos, "GPU acceleration of pyrosequencing noise removal," in *Proc. Symp. Appl. Accelerators High Performance Comput.*, Jul. 10–11, 2012, pp. 94–101.

Ibrahim Savran is currently working toward a PhD in computer science and engineering at the University of South Carolina, Columbia, SC, USA, with focus on high-performance reconfigurable computing, under the guidance of Dr. J. Bakos. His research focuses on large-scale pairwise sequence alignment. He has a BS and an MS in computer engineering from Selçuk University, Konya, Turkey (2003 and 2006, respectively). He is a member of the Association for Computing Machinery (ACM).

Yang Gao is currently working toward a PhD in computer science and engineering at the University of South Carolina, Columbia, SC, USA. He is now working in Dr. Bakos' lab and endeavors to explore computing performance and power efficiency of the nontraditional high-performance computing platforms such as GPU, DSP, as well as mobile processors for implementing scientific application kernels. He is also interested in developing a new technique or programming models to facilitate the algorithm and application development on these platforms. He has a BS in electrical engineering from Fuzhou University, Fuzhou, China (2004) and an MS in electrical engineering from the Shanghai Jiao Tong University, Shanghai, China (2007).

Jason D. Bakos is an Associate Professor of Computer Science and Engineering at the University of South Carolina, Columbia, SC, USA. His research focuses on mapping data- and compute-intensive applications to novel high-performance and embedded platforms. His group is known for their work in mapping applications in computational phylogenetics, large-scale pairwise sequence alignment, frequent itemset mining, and sparse linear algebra, and for their close collaboration with FPGA-based computer manufacturers Convey Computer Corporation, GiDEL, and Annapolis Micro Systems as well as GPU and DSP vendors NVIDIA, Texas Instruments, and Advantech. He has a BS in computer science from Youngstown State University, Youngstown, OH, USA (1999) and a PhD in computer science from the University of Pittsburgh, Pittsburgh, PA, USA (2005). He holds two patents, has published over 30 refereed publications in computer architecture and high-performance computing, was a winner of the ACM/DAC student design contests in 2002 and 2004, and received the U.S. National Science Foundation (NSF) CAREER award in 2009. He is currently serving as an Associate Editor for *ACM Transactions on Reconfigurable Technology and Systems*, as permanent Program Committee Member for the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM) and the IEEE Symposium on Application-specific Systems, Architectures, and Processors (ASAP), and is a member of the IEEE, the IEEE Computer Society, and the Association for Computing Machinery (ACM).

■ Direct questions and comments about this article to Jason D. Bakos, Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208 USA; jbakos@cse.sc.edu.