GPU Acceleration of Pyrosequencing Noise Removal

Yang Gao
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC, USA
Email: gao36@email.sc.edu

Jason D. Bakos
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC, USA
Email: jbakos@cse.sc.edu

***Abstract*—AmpliconNoise [1], an updated version of Pyronoise [2], is a tool for removing noise from metagenomic data recorded by a 454 pyrosequencer. AmpliconNoise has shown to be effective in reducing overestimation of operational taxonomic units (OTUs) and chimera detection. AmpliconNoise's noise removal method relies on clustering a large set of short sequences read by the sequencer. The DNA sequencing algorithm requires the computation of $O(n^2)$ pairwise distances using a global sequence alignment method. Each sequence consists of a few hundred base pairs and a typical dataset contains 10^4 sequences, making the clustering computation extremely expensive. In this paper we describe of GPU kernel implementation of the most computationally expensive module in the AmpliconNoise software package, SeqDist. With our GPU workstation (Intel Core i7 980 @ 3.33GHz + 3 x NVIDIA Tesla C2070) and a typical 454 dataset, our implementation achieves a 8.6X (CUDA-SeqDist) speedup with a single GPU when compared with a 12 MPI ranks of the original tools running on the CPU alone. With three GPUs, we achieve a 2.1X further speedup over the single GPU version, yielding a total speedup of 18.3X. We measure the throughput of our kernel to be 1.4 giga floating-point cell updates per second (GFCUPS) with a single GPU and 2.9 GFCUPS with 3 GPUs, where GFCUPS refers to the unique method by which the score matrix must be updated in the specialized alignment algorithm used in AmpliconNoise.**

***Keywords*-GPU Computing; Metagenomics; Heterogeneous Computing; AmpliconNoise; Pyronoise; Sequence Alignment; Needleman-Wunsch; Smith-Waterman; GPU; CUDA; MPI; Short Reads**

I. Introduction

Since the first Compute Unified Device Architecture (CUDA)-compatible GPU released by NVIDIA in 2008, GPU computing has become widely adopted by the scientific computing community. Modern GPUs are generally organized as a "many-core architecture", where each core consists of a single instruction multiple data (SIMD) architecture having 32 lanes. GPUs also generally have a memory system with higher bandwidth than same-generation CPUs(see Table IV). As a such, GPUs are generally better suited than CPUs for data-parallel kernels.

There has been recent interest in processing and clustering datasets generated by the Roche's 454 sequencing platform [3][4][5][6][7] [8][2][1]. Among these algorithms, AmpliconNoise is of particular interest, as shown in recent metagenomics literature where AmpliconNoise is used to denoise the 454 data in order to reduce the overestimation of the number of unique species implied by the data, a common problem in metagenomic sequencing [9][10] [11][12][13] [14][15][16][17].

The computational cost of 454 de-noising is an obstacle for metagenomics practitioners. To address this problem, the original AmpliconNoise software has been parallelized using the Message Passing Interface (MPI) programming model. Due to the algorithm complexity, however, AmpliconNoise still requires a substantial amount of computing resouce to process a full size dataset generated by a one-time operation of the 454 sequencing equipment, even when run on a substantial parallel computer system.

The computational kernel of AmpliconNoise is the Needleman-Wunsch algorithm [18] to compute the pairwise distances among a large set of short sequences. There are several examples in the literature that describe GPU accelerated local and global sequence alignment algorithms such as Needleman-Wunsch, Smith-Waterman [19], and BLAST [20]. 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 sequence.

Manavski provided the seminal work in accelerating Smith-Waterman using CUDA [21]. This early work has been improved upon in the development of more recent and popular libraries such as CUDASW++ [22]. More recently, Razmyslovich has developed an OpenCL implementation of Smith-Waterman [23] that achieves three times the performance of CUDASW++ 2.0[24] in some situations.

The alignment algorithm 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. As a result, the traditional metric for defining alignment throughput, "Cell Updates Per Second (CUPS)", is not suitable because it is based on integer calculation and thus cannot be directly used to compare the performance of AmpliconNoise alignments. In this paper we propose a new throughput metric that we call "Floating-point Cell Updates Per Second (FCUPS)" for comparing the performance between different implementa-

CFP1225P-ART/12 $26.00 © 2012 IEEE
DOI 10.1109/SAAHPC.2012.15

tions of the alignment method. We do this to discourage direct performances comparisons between our GPU kernel implementation and those that use integer score calculations. In addition, unlike CUDASW++, which employs a performance optimization in which the move matrix is discarded, in the AmpliconNoise alignment this matrix must be retained in order to calculate the "normalized alignment distance", required for AmpliconNoise's final distance calculation.

The remainder of the paper is organized as follows. Section 2 describes the usage of the Needleman-Wunsch algorithm used by AmpliconNoise. Section 3 describes the optimizations we applied to improve the performance of the kernel. Section 4 describes our implementation. Section 5 lists our performance results. Section 6 summarizes this work.

II. Needleman-Wunsch

There are four major components in AmpliconNoise which have been parallelized by MPI, PyroDist, Pyronoise, SeqDist, and SeqNoise. SeqDist is used to compute the distances between each pair of sequence and is the most computationally demanding of the four. Within SeqDist, the pairwise distance computation consumes nearly all of the total execution time (see Table I).

In SeqDist, in order to compute the pairwise distance among the sequences, an input dataset with *n* sequences will perform a Needleman-Wunsch alignment $\frac{n\times(n-1)}{2}$ times to construct matrices. If the length of each sequence is m, the final computation complexity is $O(n^2m^2)$. m is typically several hundred.

The Needleman-Wunsch algorithm is illustrated as below: For two input sequences A and B, we compute

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

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

where $H(i,j)$ is the score matrix. $S()$ is the similarity score resulting from comparing element $A[i]$ and element $B[j]$ obtained from a substitution matrix, etc. d is the penalty for a single gap. $S()$ and d are given by the 454 sequencer dependent data file (Lookup.dat). The penalty values themselves (the content of the table) are represented as floating-point values. In order to differentiate minor variations between flows, the authors of AmpliconNoise choose to use double precision floating point numbers to perform the computation. In (2), $M(i,j)$ is the move matrix. The value in each cell is decided by the selection results from (1).

AmpliconNoise isn't concerned with determining the exact alignment for each pair of sequences, but it does use the move matrix to determine the total alignment length to normalize the alignment score. To do this, the score is divided with length of the alignment, which is calculated by tracing back the move matrix.

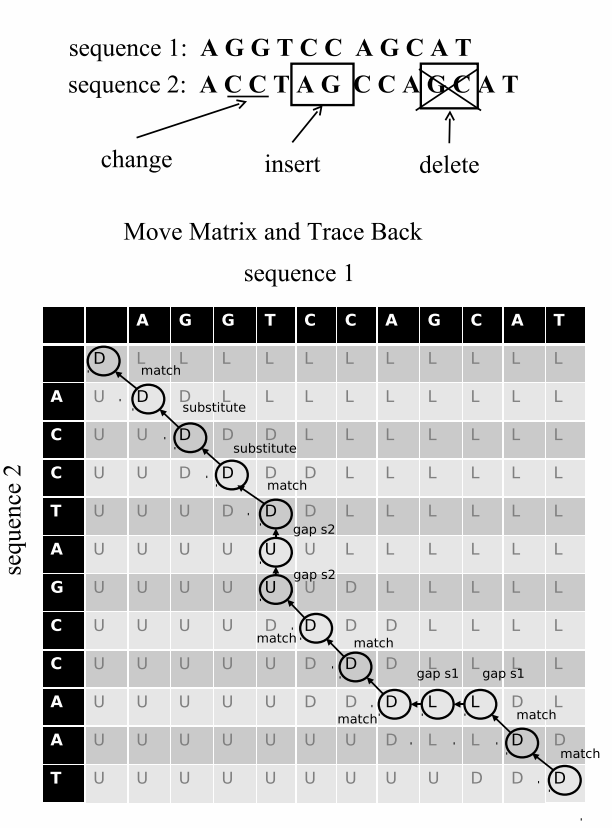

Figure 1. Using N-W method to compute the distance between two sequences.

Figure 1 shows an example N-W computation in AmpliconNoise. In this example, an example sequence, sequence 1, undergoes three edits to produce a second sequence, sequence 2. These sequences are then aligned. The final alignment score is taken from the lower-right cell of the resultant score matrix. The resultant move matrix is depicted in the figure, and shows how characters present in sequence 1 but not in sequence 2 result in moves to the left, while characters present in sequence 2 but not in sequence 1 result in moves up. Move matrices having more moves to the left or up produce longer alignments. In this example, the final score is divided by 13 moves to compute the normalized score.

We summarize the Needleman-Wunsch algorithm in AmpliconNoise as the following:

- It is used to perform pairwise alignment among a large set of short sequences.
- The substitution matrix contains double-precision floating point values and thus requires double-precision

Table I
N-W AND AMPLICONNOISE TIME CONSUMPTION

TOOLS \ DATASET	Random11000	NOTES
PyroDist	1.8%	
Pyronoise	10.4%	
SeqDist	75.8% / 75.9%	The upper value is for N-W methods and the lower value is the total.
SeqNoise	2.7%	
Other	9.1%	This contains all the non-MPI programs

The times are shown as a persentage of the total execution time. Random11000 is a random subset of 11000 flows chosen from a larger 454 dataset.

addition to compute the score, double-precision division to normalize the score, and double precision comparisons when comparing scores.

- A traceback procedure is needed to generate the normalized distance between the two input sequences.

III. METHODS

A. Hot Spot

Table I shows profiling results for the 12-thread CPU execution of the original AmpliconNoise software. We profiled the three primary modules of AmpliconNoise: PyroDist, Pyronoise, SeqDist, and SeqNoise. The time for non-MPI code are summed up in the line "others". For this test we used a 11000-flow dataset that contains a subset of flows from an actual Roche Genome Sequencer FLX titanium-based run. For SeqDist, the execution time required for the alignments is shown above the total execution time. These results show that the alignment is the computational kernel. Note that the MPI version of Pyronoise, which performs an initial clustering of the sequences, consumes 10.4% which will be the focus of future work.

B. Matrix Construction

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

Existing Smith-Waterman and Needleman-Wunsch alignment kernels (including CUDASW++) use a strategy where they use a large set of threads to generate each diagonal in a single score and movement matrix. This is depicted in Figure 2(a). This is possible since cells along the diagonal can be computed independently from one another. 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 considered that require additional conditional execution paths (i.e. if-statements) that cause branch divergences amongst the threads and degrading performance.

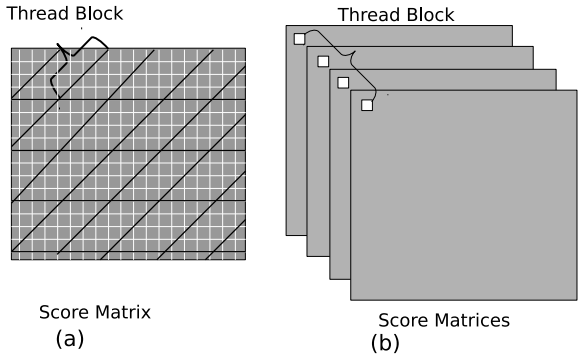

Figure 2. Two Parallel Strategy and Improvement. In (a), a thread block of threads are used to construct a single matrix while in (b) only one thread is assigned to the whole matrix.

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. 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.

CUDASW++, unlike Razmyslovich's implementation, provides an option for performing multiple alignments in parallel on the GPU (see Figure 2(b)) 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.

C. Trace Back

In AmpliconNoise alignments, the traceback procedure is needed to compute the normalized distance. Unlike matrix construction, the control flow of the trace back procedure is unpredictable and data dependent. As such, when tracing back, the threads within one thread block may have different control paths relative to each other. This leads to uncoalesced memory access and divergent branches. Both of these behaviors reduce GPU performance.

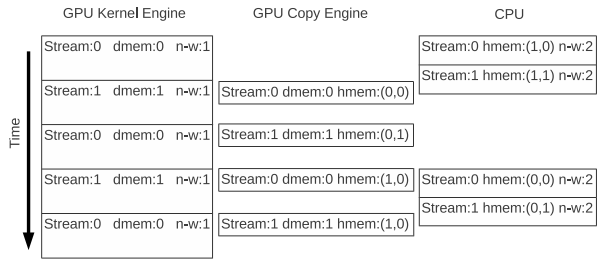

Figure 3. 3-stage GPU stream and CPU co-working pipeline. Under this configuration, two blocks of device memory(dmem) are used for GPU computing and GPU copy engine. In the host side, in order to support the GPU copy and CPU computing at the same, 4 blocks of host memory(hmem) is necessary.

One approach for solving this problem is to use a pipeline, in which the GPU performs the current round of matrix construction while the CPU performs the traceback of the move matrices generated in previous round. This approach is shown in Figure 3.

In this case, the CUDA "stream" execution model can support GPU calculation and memory copy back between GPU and host simultaneously. Despite the fact that the memory bandwidth over PCIe (around 1.5 GB/second in our system) is significantly slower than the GPU's onboard memory, when considering the complexity of the matrix construction ($O(n^2)$ versus that of the trace back $O(n)$), we have found that the execution time required by each pipeline stage is roughly balanced.

However, this method carries with it substantial memory overhead. In this technique, GPU memory must be instanced for two streams (one stream that the GPU is processing and another being transferred to the CPU). In other words, since the CPU is receiving the GPU results while the GPU is processing the next batch of data, the GPU must always contain two complete blocks of data. A further complication is that the CPU must contain both a host and a device version of both data blocks which are for trace back and for data transfer. As such, the CPU will require four instances of the movement matrix.

As shown in Table IV, one of our workstation has 4 GPUs with 4 GB memory each and 16 GB of host memory. For a multiple GPU computation, the memory requirement is even greater and a potential limiter for GPU occupancy.

In order to overcome this problem, we perform the traceback on the GPU. Our results show that the traceback procedure has little effect on execution time, but allows more thread blocks to be invoked.

IV. Implementation

In this section we discuss five specific optimizations in our kernel. Specifically, these include the following:

- task parallelization,
- grid size optimization,

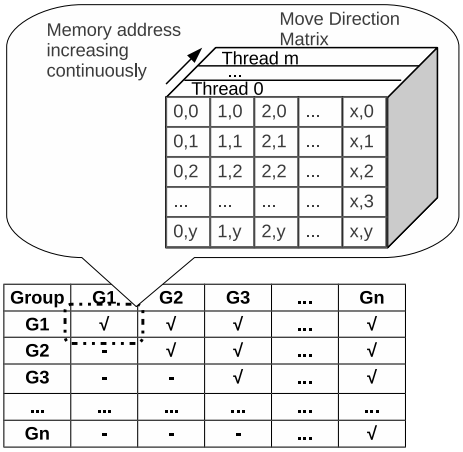

Figure 4. Group assignment and memory arrangement. The ticked cell means we should do the distance computation between the two groups, and for dash we do nothing. The distances between the flows within each group are also computed.

- use of shared memory,
- register usage optimization, and
- scaling to multiple GPUs

A. 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.

As illustrated in the Methods Section, a single thread performs the construction of one score and move matrix as shown in Figure 4. 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.

B. Memory Requirement

454 flows translate into sequences that are typically less than 800 characters. A single thread will thus require at most 800×800 bytes to construct a move matrix, using a single byte for each direction. Here we don't consider the memory cost from the score matrix, because the kernel only maintains two rows of this matrix during operation. Thus, given n 32-thread blocks, the memory usage for the move matrices has an upper bound of $800 \times 800 \times 32 \times n \approx 19.5n$ MB. Since each of our test GPUs has 4 GB we are limited to 192 32-thread blocks, requiring about 3.8 GB.

C. Grid Size

Higher register usage limits thread occupancy, so in order to maximize the number of threads it is necessary to minimize the number of registers required by each thread. Our kernel requires 40 registers/thread. A block containing less than two warps is assigned registers as if it had two warps (two warps per block is minimum). This causes a 32-thread block to require $2 \times 32 \times 40 = 2560$ registers. Since there are 16384 total registers, we can have at most $16384/2560 = 6$ blocks, or 6 warps per GPU processor, which gives $32 \times 6 = 192$ threads per GPU processor. Since the maximum number of threads per GPU processor is 1024, this results in an occupancy of 18.8%. Assigning two warps per block uses the same number of registers and doubles the occupancy to 12 warps per processor, or $32 \times 12/1024 = 37.5\%$, nominally resulting in a speedup of 2. If we continue to increase the warps per block to 3 or 4, the occupancy stays at 12 warps per block. At 5 warps per block, only 2 blocks can be mapped, reducing the occupancy to 10 warps.

Our Tesla T10 GPU has 30 multiprocessors, giving the capacity to schedule 360 warps, or 180 blocks at 2 warps per block. As shown in Table 2, we tested various combinations of one, two, and three warps per block with different numbers of blocks (5, 6 and 7 groups of 32). Interestingly, the 160 block configuration achieved the best execution time, memory throughput, and instruction issue rate. The 192 block version achieves better memory throughput but higher execution time.

D. Shared Memory

Using shared memory is a common optimization for increasing the performance of memory bandwidth-bound kernels. Each SM contains 16 KB of shared memory. The number of concurrent threads in our implementation is $Blocks_{SM} \times Threads_{BLOCK} = 5 \times 64 = 320$ (5 blocks 2 warp configuration), providing 51 bytes of available shared memory per thread. We copied the substitution matrix into shared memory to lower the number of global loads and stores and achieve lower execution time.

E. Register Usage Optimization

According to the CUDA occupancy calculator, in order to achieve higher occupancy, the number of registers would need to be reduced from 40 registers/thread to 32. Register allocation in the CUDA compiler chain is performed by *ptxas* . ptxas supports a stack for spilling registers to local memory through the use of the "-maxregcount" compiler option to limit register usage.

As shown in Table III, when we limit the number of registers per thread to 32, the SM occupancy rises from 0.375 to 0.5. This causes the automatic allocation of 16 bytes local memory per thread. Local memory is a part of the off chip device memory and has a high access time. At 32 registers per thread, we can increase the number warps per SM from 12 to 16, allowing us to launch 480 warps(240 blocks) simultaneously, however this would require more device memory than is available.

Therefore, as shown in III, we actually take 192 blocks the whole card which has shown to provide no advantage over the 160 blocks configuration when processing the 640 sequences dataset.

However, once we scale the question size to 1280 and 2560 sequences, the high occupancy overcomes the penalty from extra memory accessing.

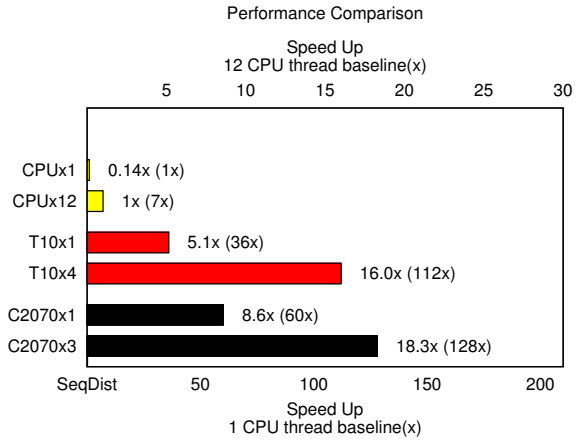

Figure 5. Performance Comparison over C005. Multiple CPU configuration is based on MPI nodes. CPU used in this test is Intel Core i7 980.

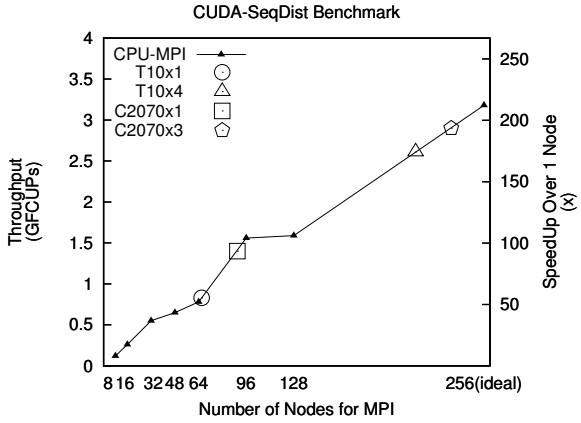

Figure 6. CUDA-FDist Benchmark:MPI vs GPU. CPU used in this test is Intel X5660 .Notice that there is performance difference with Intel Core i7 980 shown in Figure 5.

F. Multi-GPU Implementation

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 SeqDist across multiple GPUs. In our multi-GPU implementation, we divide the workload across each GPU using MPI.

Table II
CUDA-SEQDIST PERFORMANCE

Blocks × Threads	Warps per Block	Streaming Multi-processors (SM)	Ideal Warps per SM[4]	Actual Warps per SM[5]	Processor Occupancy	Run Time (sec)	Memory Throughput (GB/s)	Instr-uction Issue Rate	Notes
32 × 32	1	30	6	1.1	0.188	32.9	17.8	0.18	
192 × 32	1	30	6	6.4	0.188	15.5	35.0	0.24	
192 × 64	2	30	12	12.8	**0.375**	12.7	44.8	**0.34**	6 BG[1]
96 × 96	3	30	9	9.6	0.281	14.5	28.8	0.31	
160 × 64	2	30	12	10.7	**0.375**	**11.3**	**51.7**	**0.34**	5 BG
224 × 64	2	30	12	14.5	**0.375**	12.2	40.0	0.31	7 BG
160 × 64	2	30	12	10.7	0.375	13.6	44.5	0.28	NS[2]
160 × 64	2	30	12	10.7	0.375	11.3	51.7	0.34	NT[3]

Here we choose a dataset which contains 640 sequences to make a trade off to meet both the computation complexity and time efficient needs. All the results are base on CUDA ComputeProfiler ver. 4.0
[1] BG stands for block group which is times of 32
[2] NS stands for no-share memory optimization
[3] NT stands for no-traceback test
[4] This number is reflected in the occupancy calculator which has direct relationship with the SM occupancy.
[5] This number is calculated from the total warps divided by number of SMs. Though technically it's not accurate due to the fraction part, we take this index to reflect the real occupancy.

Table III
REGISTER USAGE OPTIMIZATION

Blocks × Threads	Warps per Block	Streaming Multi-processors (SM)	Ideal Warps per SM	Actual Warps per SM	Processor Occupancy	Run Time (sec)	Kernel Local Memory (Bytes)	Notes
160 × 64	2	30	12	10.7	0.375	43.0	0	SEQ1280[1]
192 × 64	2	30	16	12.8	0.5	42.6	16	SEQ1280[1]
160 × 64	2	30	12	10.7	0.375	164.7	0	SEQ2560[2]
192 × 64	2	30	16	12.8	0.5	162.6	16	SEQ2560[2]

[1] The dataset used in these tests is of length 1280 sequences
[2] The dataset used in these tests is of length 2560 sequences

Table IV
TECHNICAL SPECIFICATIONS OF HARDWARE PLATFORM

CPU	Core i7 980	Xeon L5520	Xeon X5660	Tesla T10	Tesla C2070	GPU
Architecture	Sandy Bridge	Nehalem	Nehalem	GT200	Fermi	Architecture
CPUs	1	2	1-19[1]	4	3	GPUs
Cores/CPU	6	4	6	30	14	MP[2]/GPU
Threads/core	2	2	2	8	32	Cores/MP
Clock rate (GHz)	3.33	2.26	2.8	1.44	1.15	Clock rate (GHz)
Memory Size(GB)	12	16	24	4	6	Memory Size(GB)
Memory Bandwidth(GB/s)	25.6	25.6	32	102	144	Memory Bandwidth(GB/s)

[1] MPI connected cluster.
[2] Multiprocessor.

V. RESULTS AND DISCUSSION

A. Test Platform

The working platform is shown in Table IV. Our platform for development and initial design exploration is a Dell R710 server with a Tesla S1070 GPU 1U system. The S1070 contains 4 Tesla T10 GPUs having the *GT200* architecture. In order to gather baseline results from a more recent CPU and GPU architectures we also ran tests on another workstation with Core i7 980 CPU and 3 C2070 (Fermi architecture) GPU. For our baseline cluster results, we ran tests on a 19 node Nehalem cluster that supports up to 128 concurrent MPI ranks. All of our test datasets are extracted from a microbial community sample of lake environment.

B. Results: SandyBridge CPU vs. GPU

Figure 5 shows the performance when AmpliconNoise processes a subset of a real 454 dataset on multiple platforms. We normalized all results to the performance of a single SeqDist rank running on an Intel SandyBridge CPU. A single T10 GPU achieves 36X speedup and one C2070 GPU achieves 60X speedup. Four T10 GPUs and three

C2070 GPUs achieve 112X and 128X respectively.

C. Benchmark Results:MPI vs. GPU

We also ran tests on a small cluster consisting of 19 nodes of Nehalem Xeon Extreme Edition CPUs with 6 cores each (see Table IV). This cluster bears a 36 port Voltaire 4036 QDR 40Gb/s infiniband switch and each computer has an onboard Mellanox ConnectX VPI (MT26438) QDR 40Gb/s inifiniband adapter with OFED middleware and driver stack installed. We ran performance tests of n nodes x m ranks per node using the following configurations: 2×4, 4×4, 8×4, 12×4, 16×4, 16×6, and 16×8 on that platform. Since we share the cluster with other users, we had to take these configurations in order to get our task scheduled as soon as possible.

We use the throughput index FCUPS as the Y-axis. We choose to down-sample the original dataset for test convenience. We also tested the CUDA-SeqDist with different GPU configurations and put the results into the CPU-MPI line based on its GFCUPs.

Our test cluster doesn't support a full allocation of 256 ranks, so we extrapolate the MPI performance linearly beyond the maximum test scale of 128, although this linear extrapolation for 256 nodes should be taken as an upper bound. Also, since the cluster is shared with other users, our results are averaged over ten runs.

The test results is shown in Figure 6 in which we could find that one T10 GPU is comparable to 64 ranks of Xeon Extreme Edition processors, while four T10 GPU cards outperforms the full capacity of the cluster. A single C2070 GPU achieves nearly 1.5 GFCUPS, which is equivalent to about a 90 node MPI cluster. With three C2070 GPUs we achieve a further speedup of 1.9.

VI. Conclusion

In this paper, we presented an efficient GPU implementation of AmpliconNoise. Combined with AmpliconNoise's native support for MPI, our implementation can also be executed on GPU equipped clusters to achieve extremely high performance.

In this paper we described each optimization and its resultant performance benefit or penalty. These techniques and experiences may be helpful as a general methodology for others who are working to optimize other codes for GPU execution.

Acknowledgment

We would like to acknowledge the ACM-Chem cluster at Interdisciplinary Mathematics Institute (IMI) of University of South Carolina for the usage of their cluster computing utility. And with support from Dr. Robert C. Sharpley(IMI) and Dr. Colin Bennett(Math Department of University of South Carolina), this project became part of NSF EPSCoR. We also want to thank the staffs of IMI for their the generous hardware and software support.

References

[1] C. Quince, A. Lanzen, R. Davenport, and P. Turnbaugh, "Removing noise from pyrosequenced amplicons," *Bmc Bioinformatics*, vol. 12, no. 1, p. 38, 2011.

[2] 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.

[3] 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 bioinformatics*, vol. 12, no. 1, p. 182, 2011.

[4] J. Cole, Q. Wang, E. Cardenas, J. Fish, B. Chai, R. Farris, A. Kulam-Syed-Mohideen, D. McGarrell, T. Marsh, G. Garrity *et al.*, "The ribosomal database project: improved alignments and new tools for rRNA analysis," *Nucleic acids research*, vol. 37, no. suppl 1, pp. D141–D145, 2009.

[5] H. Amber, R. Sean, M. Timothy, L. Bertram, and E. Jonathan, "Introducing WATERS: a workflow for the alignment, taxonomy, and ecology of ribosomal sequences," *BMC Bioinformatics*, vol. 11, 2010.

[6] P. Ram, N. Viola, and S. Christian, "CANGS: a user-friendly utility for processing and analyzing 454 GS-FLX data in biodiversity studies," *BMC Research Notes*, vol. 3, 2010.

[7] 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 Bioinformatics*, vol. 11, 2010.

[8] J. Caporaso, J. Kuczynski, J. Stombaugh, K. Bittinger, F. Bushman, E. Costello, N. Fierer, A. Pea, J. Goodrich, J. Gordon *et al.*, "QIIME allows analysis of high-throughput community sequencing data," *Nature methods*, vol. 7, no. 5, pp. 335–336, 2010.

[9] 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," *Frontiers in Microbiology*, vol. 3, 2012.

[10] L. Tranvik and T. Battin, "Unraveling assembly of stream biofilm communities," *ISME Journal*, vol. 1, p. 10, 2012.

[11] 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, p. e20956, 2011.

[12] A. Lanzén, 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 microbiology ecology*, 2011.

[13] R. Henrik Nilsson, L. Tedersoo, B. Lindahl, R. Kjller, T. Carlsen, C. Quince, K. Abarenkov, T. Pennanen, J. Stenlid, T. Bruns *et al.*, "Towards standardization of the description and publication of next-generation sequencing datasets of fungal communities," *New Phytologist*, 2011.

[14] G. Wang, S. Sherrill-Mix, K. Chang, C. Quince, and F. Bushman, "Hepatitis c virus transmission bottlenecks analyzed by deep sequencing," *Journal of virology*, vol. 84, no. 12, p. 6218, 2010.

[15] D. Knights, E. Costello, and R. Knight, "Supervised classification of human microbiota," *FEMS microbiology reviews*, 2011.

[16] J. Bahl, M. Lau, G. Smith, D. Vijaykrishna, S. Cary, D. Lacap, C. Lee, R. Papke, K. Warren-Rhodes, F. Wong *et al.*, "Ancient origins determine global biogeography of hot and cold desert cyanobacteria," *Nature communications*, vol. 2, p. 163, 2011.

[17] A. Gobet, C. Quince, and A. Ramette, "Multivariate cutoff level analysis (MultiCoLA) of large community data sets," *Nucleic acids research*, vol. 38, no. 15, pp. e155–e155, 2010.

[18] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[19] T. Smith, M. Waterman *et al.*, "Identification of common molecular subsequences," *J. mol. Biol*, vol. 147, no. 1, pp. 195–197, 1981.

[20] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[21] S. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC bioinformatics*, vol. 9, no. Suppl 2, p. S10, 2008.

[22] Y. Liu, D. Maskell, and B. Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," *BMC Research Notes*, vol. 2, no. 1, p. 73, 2009.

[23] D. Razmyslovich, G. Marcus, M. Gipp, M. Zapatka, and A. Szillus, "Implementation of Smith-Waterman algorithm in OpenCL for GPUs," in *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, 2010, pp. 48–56.

[24] Y. Liu, B. Schmidt, and D. Maskell, "CUDASW++ 2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions," *BMC Research Notes*, vol. 3, no. 1, p. 93, 2010.