Achieving TeraCUPS on Longest Common Subsequence Problem using GPGPUs

Adnan Ozsoy
School of Informatics and Computing,
Indiana University, Bloomington
aozsoy@indiana.edu

Arun Chauhan*
School of Informatics and Computing,
Indiana University, Bloomington
achauhan@indiana.edu
*Currently at Google Inc.

Martin Swany
School of Informatics and Computing,
Indiana University, Bloomington
swany@indiana.edu

***Abstract*—In this paper, we describe a novel technique to optimize longest common subsequence (LCS) algorithm for one-to-many matching problem on GPUs by transforming the computation into bit-wise operations and a post-processing step. The former can be highly optimized and achieves more than a trillion operations (cell updates) per second (CUPS)—a first for LCS algorithms. The latter is more efficiently done on CPUs, in a fraction of the bit-wise computation time. The bit-wise step promises to be a foundational step and a fundamentally new approach to developing algorithms for increasingly popular heterogeneous environments that could dramatically increase the applicability of hybrid CPU-GPU environments.**

***Keywords*—Longest Common Subsequence, semi-regular algorithms, GPU, CUDA**

I. Introduction

GPUs tradeoff complex hardware-based support for instruction level parallelism for a large number of simpler processing cores. This has a far reaching impact on application programs. Data-parallel programs with regular control flow and memory-access patterns are able to utilize the GPU hardware effectively, while programs that have thread-dependent control flow or irregular memory access patterns are unable to exploit the performance potential of GPUs. This latter category is often referred to as *irregular* applications, as against the former category of *regular* applications.

In this paper we show that it is possible to redesign algorithms that traditionally result in *irregular* applications[1] to fully exploit the GPU architecture. We do this by using the novel insight that certain *irregular* algorithms have *regular* cores that are well-suited for GPUs. Thus, by appropriately expressing the algorithms in terms of the regular cores that run on GPU and the remaining code that runs on the CPU, an application is able to leverage the best of both architectures. This approach maps well to the increasingly popular heterogeneous (hybrid) design of high performance computers [1], [2]. Moreover, evidence suggests that related algorithms share common regular cores, which justifies investment of time and energy in optimizing these cores. These algorithms fall between the traditional regular algorithms and highly irregular algorithms that exhibit the so-called *amorphous* data parallelism [3]. We call these algorithms *semi-regular* algorithms.

A classic problem that turns out to be semi-regular is that of finding the longest common subsequence (LCS) between two given strings. A *subsequence* of a string of symbols is derived from the original string by deleting some elements without changing their order [4]. For example, the sequence {b,c,e} is a subsequence of {a,b,c,d,e}. Unlike a substring, a subsequence need not be contiguous in the original string.

[1]We call these *irregular algorithms*.

Variants of the LCS problem are widely encountered in several fields. The sequence alignment problem in bioinformatics is the problem of finding similarities between nucleic acid sequences over the alphabet $\{A, T, C, G\}$, where each letter is the initial letter of one of the four types of nucleic acids [4]. In voice and image analysis LCS is used for a variety of tasks such as improving speech recognition [5], evaluating machine translation [6], and image retrieval through structural content similarity [7]. In social networks LCS is used for matching event and friend suggestions [8]. In computer security virus signature matching uses LCS [9]. In data mining, LCS is used for identifying patterns of interest in long sequences of input [10] and database query optimization [11].

In this paper, we present a detailed study of a specific variant of this important problem of LCS matching, called one-to-many LCS matching, or MLCS. We present a novel technique for optimizing MLCS on GPUs by leveraging its semi-regular structure. We identify a regular core of MLCS that consists of highly regular data-parallel bit-vector operations, which is combined with a relatively irregular post-processing step more efficiently performed on the CPUs. These operations combine techniques from Allison and Dix [12] and Crochemore et al. [13]. We make several improvements to achieve more than a trillion cell updates per second (Tera CUPS) on three NVIDIA M2090 Fermi GPUs, which is the first time this level of performance has been achieved in LCS algorithms to the best of our knowledge. We demonstrate that this performance is sustainable for MLCS on a continuous stream of sequences since the post-processing step on the CPU takes only a fraction of the core step on the GPU. We also evaluate our technique on four different GPU devices with varying NVIDIA CUDA compute capabilities.

II. Longest Common Subsequence

The LCS problem for an arbitrary number of input sequences is NP-hard [14]. However, for a constant number of sequences, the problem can be solved in polynomial time. There are two main scenarios in which LCS can be applied, one-to-one matching and one-to-many matching. In the former, only two input sequences are compared and there is one LCS result. In the latter, often referred to as MLCS, there is one *query* sequence and a set of sequences, called *subject* sequences, to which the query sequence is compared. A straightforward way to solve MLCS is to perform one-to-one LCS for each subject sequence. One of the most popular polynomial time solutions to one-to-one LCS problem is using dynamic programming.

A. Dynamic Programming Approach

The dynamic programming approach for LCS is based on the observation that the LCS of two strings X_n of Y_m, of lengths n and m, respectively, can be expressed as:

$$LCS(X_n, Y_m) = \begin{cases} LCS(X_{n-1}, Y_{m-1}) + X(n) \\ \qquad\qquad\qquad \text{if } X(n) = Y(m), \\ \max(LCS(X_n, Y_{m-1}), LCS(X_{n-1}, Y_m)) \\ \qquad\qquad\qquad \text{if } X(n) \neq Y(m) \end{cases} \tag{1}$$

where X_{n-1} represents the substring consisting of the first $n-1$ symbols of X and $X(n)$ is the n$^{\text{th}}$ symbol of X (and similarly for Y).

The solution involves filling a score matrix, H, through a scoring mechanism given in Equation 2, based on Equation 1. The best score is the length of the LCS and the actual subsequence can be found by tracking it back through the matrix. Let m and n be the lengths of two strings to be compared. In order to determine the length of the longest common subsequence in $A = (a_1, a_2, .., a_n)$ and $B = (b_1, b_2, .., b_m)$ we define $H(i, j)$, the length of the longest common subsequence in $(a_1, a_2, ..., a_i)$ and $(b_1, b_2, .., b_j)$, where $0 \leq j \leq m$ and $0 \leq i \leq n$, as follows.

$$H(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ H(i-1, j-1) + 1 & \text{if } a_i = b_j, \\ max(H(i, j-1), H(i-1, j)) & \text{otherwise} \end{cases} \tag{2}$$

The construction of the LCS dynamic programming solution has its foundations in the similarity distance measurements of two strings. One of the early works on this problem, by Levenshtein [15], gives the formula to find the minimum number of single-character edits (insertions and/or deletions) required to transform one sequence into the other. This is often called Levenshtein distance. An example for building a score matrix from the dynamic programming solution is given in Fig. 1. Following Levenshtein's work, a number of

	A	T	C	G	A	G	T
T	0	1	1	1	1	1	1
A	1	1	1	1	2	2	2
T	1	2	2	2	2	2	3
G	1	2	2	3	3	3	3
C	1	2	3	3	3	3	3
A	1	2	3	3	4	4	4
T	1	2	3	3	4	4	5

Fig. 1. Dynamic programming solution example. Matrix is built for sequences {ATCGAGT} and {TATGCAT} and the length of the longest common subsequence can be obtained from the right bottom corner.

efficient algorithms have been designed. The term *edit distance* is often used to refer to an extended form of Levenshtein distance that also includes costs for insertion, deletion, and substitution operations [16]. Needleman and Wunsch (NW) proposed an algorithm that performs a global alignment on two sequences [17]. Smith and Waterman (SW) gave the algorithm for local sequence alignment [18].

B. GPU Architectural Considerations

GPUs have several appealing features that make them good candidates for LCS computation. In this study we focus on NVIDIA GPGPUs. NVIDIA GPUs are usually described in the context of a parallel programming model from NVIDIA based on C and C++, called the Compute Unified Device Architecture, or CUDA. The CUDA programming model consists of a *host* program running on the CPU that initiates the computation and one or more *kernels* that run on the GPU. Each kernel executes over a set of parallel threads that are divided by the CUDA compiler into thread blocks and grids of thread blocks. CUDA threads within one thread block may synchronize using barriers and shared memory. CUDA programming model also has a notion of *compute capability* to summarize the features supported by specific hardware.

NVIDIA GPU hardware is organized as a collection of Streaming Multiprocessors, or SMs, each of which consists of multiple cores. The number of cores depends on the architecture generation and the number of SMs depends on the specific hardware device. For example, the *Fermi* architecture cards have 32 cores per SM for compute capability 2.0 devices. In *Kepler* cards, this number is 192. Devices with these architectures have up to 16 SMs [19]. This results in a potentially large number of concurrent threads. One or more CUDA blocks may be mapped onto each SM.

Scheduling this massively threaded architecture is one of the most critical components of CUDA architecture. The threads are created, managed, scheduled, and executed in groups of 32, called warps. Each thread block is partitioned into warps by a warp scheduler and then scheduled for execution. This style of parallelism is often called SIMT (Single Instruction Multiple Thread), where each thread in a warp executes the same instruction at a given time, but different warps might diverge. Control flow divergence within a single warp is handled by selectively disabling certain threads in the warp, causing performance degradation. Avoiding control flow divergence within warps turns out to be critical for performance.

Another key component in CUDA programming model that requires special attention is memory. GPUs provide very high memory bandwidth through several memory types: global memory, constant memory, texture memory, shared memory, and registers. Global memory resides off-chip and provides a large space of memory. Constant and texture memories are accessed through special caches. Shared memory is specialized hardware on chip that provides very fast access time and can satisfy multiple requests at the same time if the requests do not cause memory bank conflicts. Global memory has a special hardware optimization called *coalescing* that triggers only with regular memory access patterns.

Finally, data transfer latencies can be hidden by leveraging asynchronous kernel execution provided by CUDA API. Depending on the asynchronous engine count of the device, overlapping memory copies of different streams is allowed. Additionally, multiple kernels can be executed at the same time. Asynchronous kernel invocations also allow CPUs to continue working without waiting for kernels to finish, unless an explicit data transfer of the results is requested.

C. Dynamic Programming on GPUs

The construction of the scoring matrix creates three-way dependencies as shown in Fig. 2, where each cell depends on its left, upper, and upper-left neighbors. This dependency prevents parallelization along the rows or columns. A possible solution is to compute all the cells on an anti-diagonal in parallel as illustrated in Fig. 3. However, this suffers from two problems: (a) the parallelism is limited in the beginning and the end of computing the matrix; and (b) memory access patterns are not amenable to hardware coalescing.

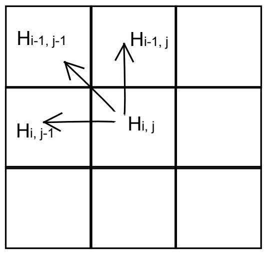
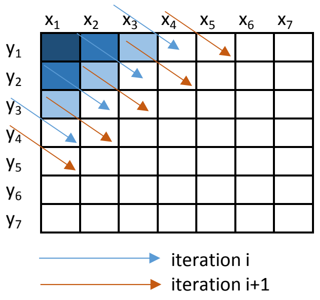


Fig. 2. The dependency in LCS Dynamic Programming solution



x_1 x_2 x_3 x_4 x_5 x_6 x_7
y_1 y_2 y_3 y_4 y_5 y_6 y_7
iteration i
iteration i+1

Fig. 3. The anti-diagonal(wave-front) parallelization of LCS Dynamic Programming solution

Another issue in using dynamic programming approach for LCS on GPUs is memory allocation. Equation 2 requires space proportional to the product of the sequence lengths. One way to reduce the memory requirement is by keeping only the outer boundaries of each submatrix being processed in memory. However, the boundaries need to be recomputed when doing the back-tracking for building the LCS.

As discussed in Section V, prior attempts to parallelize the traditional dynamic programming LCS algorithm on GPUs suffer from both poor distribution of workload and sub-optimal utilization of GPU resources. Currently proposed parallelization approaches to construct the score matrix are limited by the three-way dependencies. These approaches invariably serialize the construction of the actual subsequence and perform all the computation on the GPU, thus failing to leverage the CPU for tasks more efficiently done on CPUs.

III. Toward Tera-CUPS on GPUs

We have developed an approach that eschews dynamic programming in favor of highly data-parallel operations. The asymptotic complexity of our approach remains unchanged compared to the dynamic programming algorithm, but the changed formulation of the algorithm exposes the regular core within the algorithm that can be effectively parallelized on GPUs.

Several observations lead us to our approach, which is optimized for MLCS. We observe that we require matching information of every single element in the sequences. This motivates the computation of a binary matrix that summarizes the matching result of each symbol in the subject sequence with each symbol in the query sequence. Fig. 4 shows such a binary matrix for two example sequences. The computation of such a matrix is highly data parallel and potentially mapped efficiently to GPU threads with homogeneous workload distribution and no control-flow divergence.

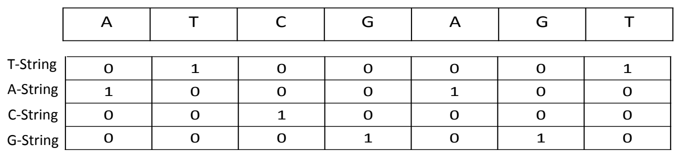
A second observation is that each entry in the binary matrix can be stored as a single bit, leading to a natural packing of the matrix entries. This clearly reduces the space requirement. But, more importantly, packing the matrix elements in this way

	A	T	C	G	A	G	T
T	0	1	0	0	0	0	1
A	1	0	0	0	1	0	0
T	0	1	0	0	0	0	1
G	0	0	0	1	0	1	0
C	0	0	1	0	0	0	0
A	1	0	0	0	1	0	0
T	0	1	0	0	0	0	1

Fig. 4. Binary match matrix.

enables us to use bit operations on words to perform vectorized matching operations on word-length segments of the matrix. This leads to another level of parallelism within the program.

Another observation is that the matches can be precomputed for each symbol in the alphabet for a given query sequence. Then, the matrix can be constructed by simply looking up the pre-computed match for each symbol in the subject sequence. For small alphabets, such as the four-symbol alphabet commonly used for sequence alignment in bioinformatics, this results in good temporal locality of the precomputed data that can be cached. An example in Fig. 5 shows this precomputed data for the sample query string "ATCGAGT."



	A	T	C	G	A	G	T
T-String	0	1	0	0	0	0	1
A-String	1	0	0	0	1	0	0
C-String	0	0	1	0	0	0	0
G-String	0	0	0	1	0	1	0

Fig. 5. Precomputed matching data for the query string "ATCGAGT."

A bit-matrix obtained in this manner does not lead itself directly to LCS. However, certain row operations on it can be used to compute a derived matrix that provides a quick readout of the length of the LCS. In many problems, such as those for detecting virus signatures or patterns in social networks, the actual subsequence is often of less interest than the boolean outcome of whether or not a match of certain length was found, or simply the length of the longest match. In cases where the actual LCS is needed, as in bioinformatics applications, a traditional LCS algorithm implemented on CPUs is adequate to compute the actual subsequence in the small fraction of cases where the matches are sufficiently long to be of interest. Section III-C discusses this in greater detail.

It turns out that the binary match matrix need not be constructed explicitly in order to build the derived matrix. Moreover, we need only the final row of the derived matrix to compute the lengths of the LCS. Usually, in MLCS, we are interested in a certain number of top matches. As a result, this serves as a filter to eliminate the subject sequences that fail to produce sufficiently long matching subsequences.

A. *Bit-vector approach*

In 1986, Allison and Dix proposed solving the LCS problem using bit representation [12]. They also identified the optimization of using precomputed strings, calling them *alphabet-strings*. The computation produces only the final row of the derive matrix and proceeds by computing one row at a time, starting from the top. The number of 1's in the final row is the length of the LCS.

In order to arrive at an efficient way to compute the rows, they first observe that the length of the matched sequence changes by at most one symbol from one row to the next in the original H matrix defined in Equation 2. Thus:

$$H(_{i-1,j-1}) \leq H(_{i,j-1}) \tag{3}$$
$$H(_{i-1,j}) \leq H(_{i,j}) \tag{4}$$
$$|H(_{i,j}) - H(_{i-1,j-1})| \leq 1 \tag{5}$$

A key aspect of Allison and Dix's work is the identification of k-*dominant* matches. In order to define k-dominant, we need to first define the *rank* of a match. We can define a partial order on the matches. A match that occurs in position (i, j) *precedes* a match at (i', j') if $i < i'$ and $j < j'$. A set of matches where each pair consists of matches that can be ordered is called a *chain*. The rank of a match (i, j) is the length of the longest chain of matches terminating at (i, j), i.e., with (i, j) being the greatest match according to the partial order defined above. The match at i,j is k-dominant if it has rank k and for any other pair $[i', j']$ of rank k, either $i' > i$ and $j' \leq j$ or $i' \leq i$ and $j' > j$ [13], [20].

The LCS of two strings can be no longer than the maximum rank of the dominant match, therefore once the k-dominant matches are computed, the LCS problem is solved. Fig. 6 shows an example of the relation between the k-dominant matches and the solution to LCS problem.

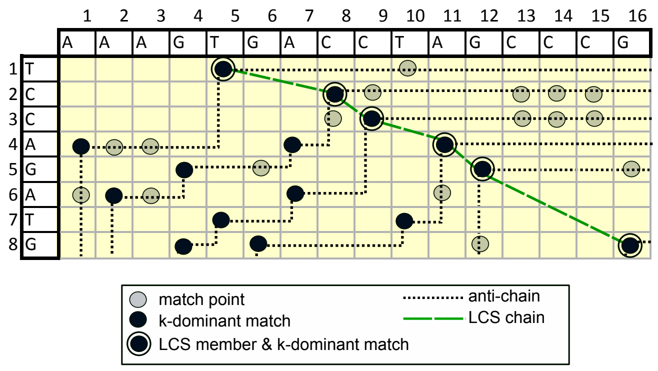


Fig. 6. LCS for x = "TCCAGATG" and y="AAAGTGACCTAGCCCG" and k-dominant matches and the matrix generated by Crochemore's algorithm (Courtesy of Crochemore et al.).

Allison and Dix gave the following equation to calculate the length of the LCS where Row_0 starts with all zeros and M is the precomputed alphabet-string.

$$\begin{aligned} X &= Row_{i-1} \quad OR \quad M_i \\ Row_i &= X \ AND \ ((X - (Row_{i-1} << 1)) \quad XOR \quad X) \end{aligned} \tag{6}$$

In each iteration, the equation follows the projection of the anti-chains and marks all the anti-chain corners in a row. The final number of bit-changes compared to the initial row (number of 1's in the last row) gives the length of the LCS. Crochemore et al. [13] improved the above expression to reduce the number of bit operations from 6 to 4 by starting with Row_0 set to all ones and using precomputed alphabet-strings (M_i) and their inverses (M_i'), as follows.

$$\begin{aligned} Row_i = (Row_{i-1} \ + (Row_{i-1} \ AND \ M_i)) \\ OR \ (Row_{i-1} \ AND \ M_i') \end{aligned} \tag{7}$$

B. Parallelizing and Optimizing on GPUs

After identifying binary matrix match computation as the core of LCS that is most amenable to optimizing for GPUs, we next describe our design decisions and various optimizations that we performed on this core. Recall that we do not need to store the entire matrix, only the last row that gives us the length of the LCS. The last row of the matrix is arrived at by iteratively computing each row of the matrix using Equation 7, starting with the first, and requires keeping only two rows' worth of data in memory. We have implemented both Allison and Dix's and Crochemore et al.'s solutions. Crochemore at al.'s algorithm uses fewer bit operations, which could be expected to lead to a slightly better performance than Allison and Dix's.

a) Intra- vs Inter-task Parallelism: There are two main opportunities for parallelizing the bit-vector computation, running multiple subject sequence matches concurrently (called *inter-task* parallelism) and using multiple threads to perform bit-vector computations as outlined above (called *intra-task* parallelism). Prior work has reported obtaining better performance with the latter [21], [22], [23] for sequence lengths greater than 4K, which is what we use for this study. Section IV justifies the choice of 4K sequence size.

One subject sequence is assigned to each CUDA thread within a CUDA block. In addition, multiple CUDA blocks are concurrently launched on different streaming multiprocessors. Since each subject sequence is processed independently, there is no communication between these tasks.

b) Memory Spaces: One observation from Equations 6 and 7 is that precomputed alphabet-strings are read-only. This makes it possible for alphabet-strings to be placed in cache-able read-only memory. Constant memory is an obvious choice, since the alphabet-strings are small enough to fit within it. Another observation is that alphabet-strings values are used by every thread in the GPU device and loading these values into fast memory, such as shared memory, can also provide benefits. To understand the trade-offs between constant and shared memories we have implemented both versions for each algorithm.

c) Hiding Data-copying Latencies: CUDA enables overlapping data copy and kernel execution to hide data-copying latencies. In order to hide data-copying latencies with asynchronous execution we need to use multiple streams of data copying and computational kernels. We divide the computation into multiple smaller pieces and push them to GPU asynchronously using CUDA streams. Additionally, by splitting data copying and computational kernels into smaller chunks we allow the GPU to execute multiple kernels concurrently. We allocate host memory with page-locked option using the CUDA `cudaMallocHost` API call. This design choice is necessary for asynchronous concurrent execution. Additionally, the range of memory allocated with this function call is tracked by GPU driver, enabling higher bandwidth for reads and writes and automatically accelerating data movement [19].

d) Leveraging Multiple GPUs: Multi-GPU nodes are increasingly common on clusters. In this work, we also investigate the effects on performance when multiple GPUs are

TABLE I. BENCHMARK TESTBEDS

System	GPU	Number of Cores	Device Memory	CUDA Capability	CUDA Version	CPU	Host Memory	OS
FutureGrid	C2075 x 2	448	5375 MB	2.0	5.0	X5660	192 GB	Red Hat 6.3
Keeneland	M2090 x 3	512	5375 MB	2.0	4.2	X5660	24 GB	CentOS 6.3
	GTX 680 x 2	1536	2048 MB	3.0	5.0	AMD PII X6	16 GB	Debian 4.4
	GTX 480	480	1535 MB	2.0	4.2	AMD PII X6	16 GB	Debian 4.4

used. To maximize the benefit from multiple GPUs, we create separate CPU threads to handle the different GPUs. Each thread is assigned $1/n$ of the sequences, where n is the number of GPUs.

e) Streaming Sequences: For very large problems, GPU memory could become a limiting factor. The current maximum memory on a single NVIDIA GPU is about 6 GB. We implemented MLCS in a way that allows sequences to be streamed continuously with repeated kernel launches, which run concurrently with CPU post-processing steps. This establishes a pipeline that can effectively and completely hide data-copying as well as CPU post-processing latencies.

C. Post Processing

The GPU step returns a list of all the lengths for each subject sequence. Post processing is needed to find the top N LCS lengths, where N is a user-specified parameter, and optionally calculating the actual LCS in those cases. Identifying the top N LCS lengths is a relatively short amount of work that is easily done on the CPU.

If the actual subsequences are required then those can also be found using the traditional dynamic programming algorithm, with the score matrix of Equation 2. Note that N is usually much smaller than the number of subject sequences, so this needs to be done for a small fraction of all the sequences that are matched. Moreover, it can be parallelized by using multiple CPU threads, each computing a sequence, as long as $N > 1$. Alternatively, the bit-vector method could also be adapted for CPUs. We have implemented both versions to evaluate their relative performance.

IV. PERFORMANCE EVALUATION

A. Testbed configurations

To evaluate our implementations, we utilized four different resources. Delta GPU cluster is part of FutureGrid—a multi-institutional project supported by the National Science Foundation, involving Indiana University [24]. A second testbed that we used was the Keeneland cluster at Georgia Institute of Technology [25]. Additional workstations in our lab, consisting of GTX 690 (dual 680) and GeForce GTX 480 cards provided the final two test platforms (Table I).

We used synthetic data for the query and subject sequences. In our experiments each sequence is 4K symbols long, motivated by the commonly used sequence length in different applications. For example, many benchmark sequence databases for bioinformatics (Ensembl Dog Proteins, NCBI RefSeqHuman Proteins, TAIR Arabidopsis Proteins, UniProtDB/Swiss-Prot, etc.) 4K size threshold covers more than 99% of the sequences [21], [23]. In GPU-based data compression, past work shows highest performance on 4K chunk size [26]. Collections of string sequences in many applications can reach very large numbers. UniProtDB/Swiss-Prot database includes more than 500,000 subject sequences [21]. Virus signatures number in hundreds of thousands and are growing constantly [27]. Therefore, we tested our implementation with three different subject sequence sizes: 50,000, 188,000 and 720,000.

B. Results

The unit we use to report our results is *cell updates per second (CUPS)*, which is widely used to evaluate performance of LCS algorithms. A cell represents one grid item in the score matrix built in the dynamic-programming approach. In general, total number of cell updates is obtained from the lengths of the sequences being compared. In a one-to-many LCS case, it is the product of the length of the reference sequence and the total lengths of all other sequences. On contemporary machines millions (mega) or billions (giga) of updates per second are common and are abbreviated as MCUPS and GCUPS, respectively. Equation 8 formulates the relation, where R denotes the length of the reference sequence, T denotes the total length of all sequences, and S denotes the time it takes to perform the computation, in seconds.

$$CUPS = \frac{R \times T}{S} \qquad (8)$$

We benchmarked the performance of the bit-vector approach on Crochemore et al.'s algorithm using constant memory for the precomputed alphabet-strings. The tests involve sequences of 4K symbols with the alphabet size four. We use varying number of block sizes and thread counts such that their product is alway equal to a fixed number, which is the number of sequences being matched. The number of threads ranges from 128 to 1024 and the number of blocks ranges from 1470 to 184, respectively, for 188K sequences. Different block / thread number configurations lead to varying device occupancy.

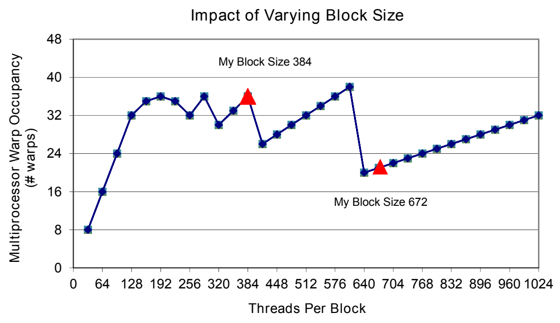


Fig. 7. CUDA Occupancy Calculator (Courtesy of NVIDIA).

CUDA Occupancy Calculator [28] can be used to estimate occupancy of a kernel using the compute capability of the device, number of threads per block, register usage per thread in a block, and the amount of the shared memory used per block. Fig. 7 shows estimated warp occupancy on GTX 480 for

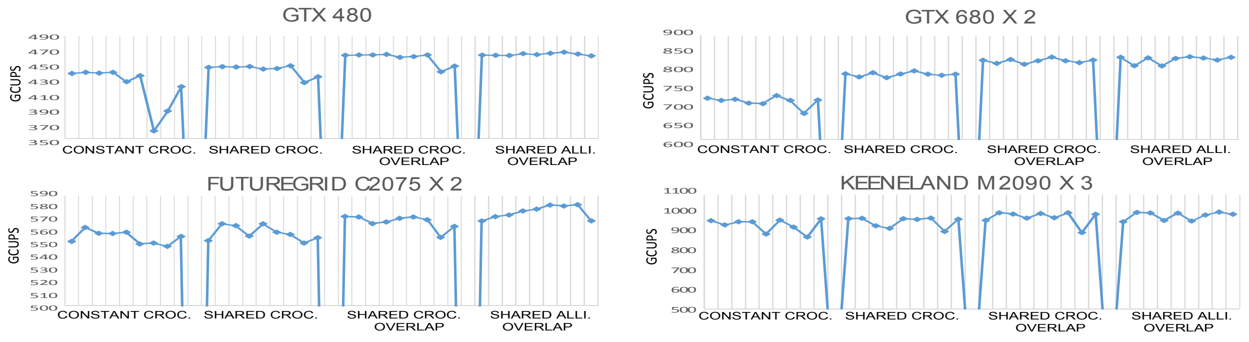


Fig. 8. Combined results for each device and four different implementations.

varying number of threads per block. After increasing initially the occupancy drops to the lowest point at 672 threads and reaches the highest at 384 threads per block. The register usage for this implementation of Crochemore et al.'s algorithm using constant memory is 26 per thread on GTX 480. The register usage information can be obtained by compiling all CUDA input files (`.cu` files) to device-only (`.cubin` files) and passing verbose option, or using the visual profiler. This results in an estimated 44% occupancy for 672 threads and 75% for the 384 threads. In practice, there is excellent correlation between estimated occupancy and observed performance for a given configuration.

The best occupancy for a given kernel varies across different GPU cards, based on the card's compute capability and register usage. Fig. 8 shows occupancy calculations for the four devices we studied for three different implementations of Crochemore et al.'s algorithm and one implementation of Allison and Dix's algorithm.

As indicated before, we implemented versions that kept the alphabet strings in constant memory (”constant croc.”) and shared memory (”shared croc.”). Across all configurations and devices keeping the alphabet strings in shared memory resulted in a gain of 24 CUPS on an average, compared to constant memory.

In order to hide data copying latencies we overlap kernel exceution with memory copies. To achieve that we divide the computation into smaller pieces and use CUDA streams to spawn GPU kernels and initiate data transfers concurrently. This lets us establish pipelines that overlap kernel executions with data transfers for the subsequent kernels. Additionally, concurrent kernel execution is also possible through multiple asynchronous streams. This approach acheives best performance with 16 concurrent CUDA streams, resulting in an average improvement of 22 CUPS over the shared-memory version without such an overlap.

NVIDIA's visual profiler[2] provides a visual feedback of several performance metrics. In our tests, the profiler reported 94.5% of kernel/memory copy hiding (all memory copies except the first stream CPU-to-GPU and last stream GPU-to-CPU memory copies) and fully concurrent execution of kernels.

One concern in using shared memory for alphabet-strings is the potential for memory bank conflicts, leading to reduced performance. Since we exploit inter-task parallelism, each thread operates on a different sequence and follows a different data access pattern. This nondeterministic behavior of access patterns makes it practically impossible to eliminate bank conflicts. Crochemore et al.'s algorithm uses both alphabet-strings and their inverses (Equation 7). On the other hand, Allison and Dix's algorithm only uses alphabet-strings (Equation 6), leading to fewer shared-memory accesses and, potentially, fewer conflicts. The trade-off is a higher number of bit operations in Allison-Dix algorithm. Using NVIDIA's profiler tool we observed around 70M (70,601,250) shared-memory bank conflicts for Crochemore et al.'s algorithm compared to about 52M (52,327,407) for Allison and Dix's. Even with increased number of bit operations this large difference in bank conflicts results in Allison-Dix achieving a higher performance, with an average difference of 4.4 CUPS.

In Fig. 8 the two algorithms with overlapped computations and data transfers are represented by labels “shared croc. overlap” and “shared alli. overlap.” Each set of plots in the figure shows four different implementations. In each implementation, different points show different threads/blocks configurations. The sawtooth shaped graphs in each case is a consequence of varying occupancy and closely follows the occupancy pattern in Fig. 7.

Fig. 9 shows GCUPS achieved by the shared-memory version of Allison-Dix algorithm for different numbers of sequences. Three different sets are used; a small set of 50K sequences, a set of 188K sequences for a full-capacity single kernel call, and a large set of 720K sequences requiring multiple kernel calls. For each data set, different configurations are applied. The two larger data sets achieve similar performance,

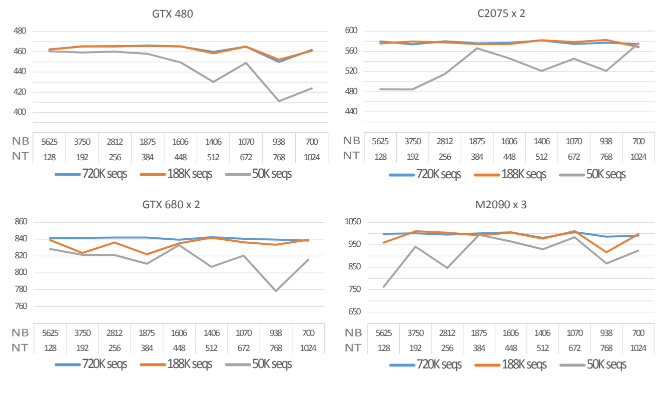


Fig. 9. Comparison of different sequence set sizes. Blue, orange and grey lines indicates 720K, 188K and 50K number of sequences respectively.

[2]For more information https://developer.nvidia.com/nvidia-visual-profiler

indicating that the kernel is able to sustain high performance across multiple invocations.

We also tested the algorithm with different alphabet sizes. Fig. 10 shows the performance results for alphabet sizes 4, 8, 12, 16, 20, and 26. The results show that for the three GPU devices (GTX 480, Tesla C2075, and M2090), an increase in the alphabet size from 4 to 26 characters results in at most 1.9% performance loss. On the GTX 690(680x2), the performs suffers by up to 22%. A possible cause could be an increase in bank conflicts due to bigger shared memory footprint for larger alphabets. GTX 690 stands out by its compute capability (3.0 against 2.0 of all others). The profiler indicates 50% more bank conflicts per GPU on GTX 690 for 26-character alphabet, compared to the other cards. The much larger number of bank conflicts explains the higher performance penalty on GTX 690 for bigger alphabets.

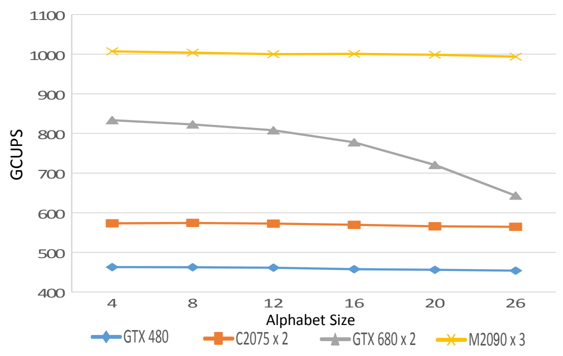


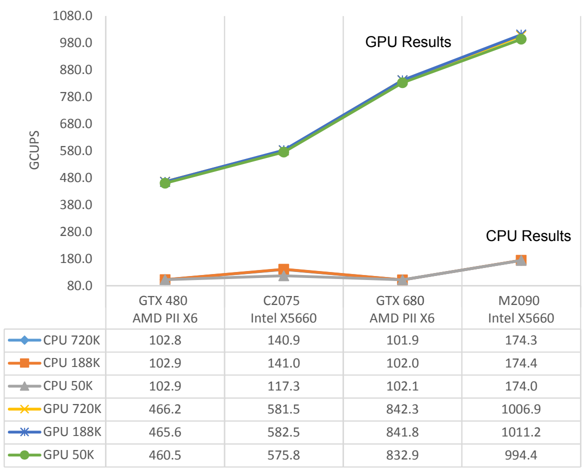
Fig. 10. Comparison of different alphabet sizes.

C. Post-Processing steps on CPU

After the length of LCS is calculated for every sequence, the actual LCS is obtained for the top N of them, N is adjustable by user. This stage is left to CPU for post-processing. POSIX threads are used to parallelize this stage. In our tests, we see 0.01 to 0.015 second latency per actual sequence calculated, not depending on the GPU version. This latency per actual LCS calculated is only 0.016% of GPU computation time for the largest sequence set and 0.23% of the GPU computation time for the smallest sequence set. Additionally, in the multiple runs scenario each run's post-processing step can be overlapped with, and be completely hidden by, the next run's GPU work, except the last one.

D. CPU implementation comparison

We also compare our results with CPU implementation. For that purpose, we parallelize the Allison and Dix's implementation using POSIX threads, and use the `gcc` optimization flag `-O3`. Fig. 11 shows the GCUPS results for three data sets on four different setups. We achieve 8.3X speedup in the best case and 5.7X speedup on average, compared to the parallel CPU implementation. Further, we compared the dynamic programming solution with the bit-vector approach on CPU. We compared serial versions of both these algorithms to identify the benefits of vectorized bit operations on CPUs. For the same configurations the bit-vector version consistently performs about 48X better than the dynamic programming version. However, we emphasize that the bit-vector version does not compute the actual LCS.



	GTX 480 AMD PII X6	C2075 Intel X5660	GTX 680 AMD PII X6	M2090 Intel X5660
◆ CPU 720K	102.8	140.9	101.9	174.3
■ CPU 188K	102.9	141.0	102.0	174.4
▲ CPU 50K	102.9	117.3	102.1	174.0
✕ GPU 720K	466.2	581.5	842.3	1006.9
✳ GPU 188K	465.6	582.5	841.8	1011.2
● GPU 50K	460.5	575.8	832.9	994.4

Fig. 11. Parallel bit-vector LCS (only length), CPU vs GPU.

E. Related work comparison

We listed several related papers in Section V. The reported performance results are either in GCUPS or seconds. For instance, one of the early works by Manavski and Valle reports 3.5 GCUPS performance using Smith-Waterman MLCS. A highly referenced paper, CUDASW++ 2.0 by Liu et al., reports 17 to 30 GCUPS performance on single and double GPUs, respectively, using Smith-Waterman algorithm with both anti-diagonal parallelization (less than 1% of the time) and inter-task approach. Hains et al. improve CUDASW++ and reach approximately 25-30 GCUPS on a single GPU. Khajeh-Saeed et al. investigate the Smith-Waterman algorithm for two large sequences rather than one-to-many alignment, and scaling through hundreds of GPUs. They achieve 1.04 GCUPS per GPU. Kawanami et al. use the Allison bit-vector approach on one-to-one LCS calculation achieving approximately 3 GCUPS. For those studies that did not report GCUPS we derived approximate GCUPS from the given time and speedup numbers. Compared to these efforts, we achieve one to two orders of magnitude better. However, we hesitate to give direct performance comparisons due to the differences in algorithms and LCS approaches.

F. Analysis

There are several key insights resulting from our experiments beyond TeraCUPS performance. One of these is the effect of occupancy on performance. The occupancy of GPU depends mainly on the number of threads used per block and the register usage per thread in a block. In our implementations we used two algorithms, two memory choices, and four different devices. Depending on the four parameters (compute capability of the device, number of threads used per block, register usage per thread in a block, and shared memory in bytes per block), each version has a different configuration that runs better. CUDA Occupancy Calculator gives occupancy of GPU with given configurations. Register signatures for all versions and devices used in this paper are given in Table II. The occupancy information which can be statically computed, gives users the ability to predict performance and tune it manually or automatically for a specific configuration.

Another key observation is the identification of highly regular data-parallel approach for the semi-regular MLCS structure. Using bit-vector operations in GPUs provides a bet-

TABLE II. REGISTER SIGNATURES FOR DIFFERENT VERSIONS

	Constant Croch.	Shared Croch.	Shared Croch. Overlap	Shared Allison Overlap
GTX 480	26	23	23	19
C2075	24	24	24	18
GTX 680	24	25	25	19
M2090	26	23	23	19

ter utilization of the device and leads to a better performance as the results indicate above. The core of the MLCS algorithm has commonalities with other widely used applications such as substring matching and compression. To the best of our knowledge, bit-vector approach on GPUs has not been investigated for those problems.

Finally, scalability of the bit-vector approach is proven through large data set tests. Although, there is extra overhead in handling the multiple runs, the achieved performance is sustained (Fig. 9). This shows that our work is easily scaled to very large data sets.

V. Related Work

The longest common subsequence (LCS) problem has grabbed the attention of many researchers because of its wide applicability in a variety of applications. In this section we summarize related work only for bit-vector approach to LCS and GPU-based parallelization efforts.

A. *Bit-wise LCS*

Solving the LCS problem using bit representation was initially proposed by Allison and Dix [12]. Their approach uses bit operations on bit-vector strings to find the length of the LCS, improving the time complexity by the length of the word. Packing bits into words and using bit operations gave the algorithm word-size parallelism.

Myers [29] gives a practical method for edit distances in the same running time as [12] with the addition of extracting the actual longest common subsequence string, rather than just the length. Crochemore et al. also propose a similar approach but using fewer bit operations than [12] and [29], thus achieving an improved running time [13]. Hyyro applies bit-vector approach to Levenshtein distance with an improvement to Myers algorithm [30].

B. *GPU related work LCS*

One of the early GPU adaptations of LCS for subsequence matching is by Manavski and Valle [31]. They give a solution for one to many sequence alignment problem on GPUs by using dynamic programming solution in which one thread is allocated for the alignment of the two sequences.

Kloetzli et al. give a cache-coherent algorithm in linear-space using a two-level algorithm on the GPUs [32]. The work build on the algorithm by Chowdhury [33] for linear space LCS algorithm. The memoization step saves only the outer boundaries of the matrix to satisfy the linear space requirement, which can be used later in the reconstruction step. Anti-diagonals are parallelized for computation on GPU, an approach introduced in LCS calculation by Edmiston et al. [34].

Liu et al. give a one-to-many sequence alignment approach over a database of query sequences in CUDASW++ implementation [21], [22]. The given implementation uses parallelism either within or across tasks on GPUs, depending on the sequence size. For sequence sizes smaller than the threshold (which is 3072 according to the experimental results), they employ inter-task parallelism in which one thread is responsible for finding the matching information of two sequences. For bigger sizes, intra-task parallelism is used in which a block of threads calculates the longest matching length by using the anti-diagonal approach.

Hains et al. improve the intra-task kernel of CUDASW++ by global memory access improvements, tiling in the anti-diagonal process, extensive register usage and with more flexible threshold selection [23].

Yang et al. investigate the data dependency reduction for the Smith-Waterman algorithm [35]. A higher degree of parallelism is proposed to break the row / column dependency by visiting the previous row / column data for each time. This approach results in better thread utilization than the anti-diagonal approach.

Khajeh-Saeed et al. investigate the single large Smith-Waterman problem rather than one-to-many alignment problem [36]. The work proposes to memoize additional values to reduce dependency and gives a row / column parallel approach similar to Yang's work [35]. It also gives a multi-GPU solution assuming prior knowledge of the upper bound for the length of the alignment.

One recent work that uses both bit-wise operations and GPUs is by Kawanami et al. who give a GPU implementation of the Crochemore's solution to improve Hirschberg's CPU LCS algorithm [37]. They compare their results with Kloetzli but only investigate one-to-one LCS [38].

VI. Conclusion and Future Work

We have presented a novel approach to augment the one-to-many LCS (MLCS) problem on GPUs. The technique elaborates bit-vector operations that exploit the word-size parallelism. Inter-task parallelism is adapted, relying on the independent computation of each one-to-one LCS comparison. We studied two algorithms, by Allison-Dix and Crochemore et al., on GPUs. We then provided the analysis and design steps of GPU-specific optimizations. We used CPU for post-processing to take advantage of the heterogeneous environment. Using the multi-GPU approach, we are able to achieve TeraCUPS performance for length calculation of MLCS problem—a first for LCS algorithms.

Compared to the best known parallel CPU implementations, we achieved up to 8.3x better performance. Our algorithm also shows a sustainable performance with very large data sets. By utilizing both CPU and GPU, we were able to hide data transfer latencies in a heterogeneous environment. By providing design insights and tools specific to NVIDIA GPUs, our work provides a reference for future developments.

One possible future direction to investigate is using bit-vector approach to solve other probleme related to LCS on GPUs, such as Needleman-Wunsch [17] and Smith-Waterman. Hirschberg algorithm [37] is a possible candidate to compute the actual LCS. Another possible direction is to explore problems common to certain string-matching algorithms. For example, substring matching is at the core of the dictionary-based lossless compression and the LCS problem shares certain features with it. A common solution to substring matching based on bit-vector operations could potentially also benefit these related problems.

VII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0910812. This research also used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. The authors would like to thank NVIDIA for their donation of the graphics hardware (GTX 690).

References

[1] *TOP500 List of the world's top supercomputers*, Retrieved February 20, 2013 from http://www.top500.org/lists/2012/11/.

[2] A. L. Shimpi, "Inside the Titan Supercomputer: 299K AMD x86 Cores and 18.6K NVIDIA GPUs," *AnandTech online computer hardware magazine*, October 2012. [Online]. Available: http://www.anandtech.com/show/6421/inside-the-titan-supercomputer-299k-amd-x86-cores-and-186k-nvidia-gpu-cores

[3] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?" *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009.

[4] M. Crochemore and W. Rytter, *Text Algorithms*. Oxford University Press, 1994.

[5] A. Guo and H. Siegelmann, "Time-Warped Longest Common Subsequence Algorithm for Music Retrieval," C. L. Buyoli and R. Loureiro, Eds. Barcelona, Spain: Universitat Pompeu Fabra, Oct. 2004, pp. 258–261.

[6] C.-Y. Lin and F. J. Och, "Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics," in *Proceedings of ACL '04*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2004.

[7] E. G. M. Petrakis, "Image representation, indexing and retrieval based on spatial relationships and properties of objects," 1993.

[8] X. Tian, Y. Song, X. Wang, and X. Gong, "Shortest path based potential common friend recommendation in social networks," in *Cloud and Green Computing, 2nd International Conf. on*, 2012, pp. 541–548.

[9] S.-y. Lu and K.-S. Fu, "A sentence-to-sentence clustering procedure for pattern analysis," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 8, no. 5, pp. 381–389, 1978.

[10] T. S. Han, S.-K. Ko, and J. Kang, "Efficient subsequence matching using the longest common subsequence with a dual match index," in *Proceedings of MLDM '07*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 585–600.

[11] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, Mar. 1988.

[12] L. Allison and T. I. Dix, "A bit-string longest-common-subsequence algorithm," *Inf. Process. Lett.*, vol. 23, no. 6, pp. 305–310, Dec. 1986. [Online]. Available: http://dl.acm.org/citation.cfm?id=8871.8877

[13] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid, "A fast and practical bit-vector algorithm for the longest common subsequence problem," *Information Processing Letters*, vol. 80, pp. 279–285, 2000.

[14] D. Maier, "The complexity of some problems on subsequences and supersequences," *Journal of the ACM*, vol. 25, pp. 322–336, Apr. 1978.

[15] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[16] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. ACM*, vol. 21, no. 1, pp. 168–173, Jan. 1974.

[17] S. B. Needleman and C. D. 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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022283670900574

[18] T. Smith and M. Waterman, "Identification of common molecular subsequences," *J. of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022283681900875

[19] NVIDIA, *CUDA C Programming Guide*, 2012.

[20] M. J. Atallah and S. Fox, Eds., *Algorithms and Theory of Computation Handbook*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1998.

[21] 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, 2009.

[22] Y. Liu, B. Schmidt, and D. L. 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, 2010.

[23] D. Hains, Z. Cashero, M. Ottenberg, W. Bohm, and S. Rajopadhye, "Improving cudasw++, a parallelization of smith-waterman for cuda enabled devices," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '11, 2011.

[24] *FutureGrid, Delta User Manual*, Retrieved June 14, 2012 from https://portal.futuregrid.org/manual/delta.

[25] J. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili, "Keeneland: Bringing heterogeneous gpu computing to the computational science community," *Computing in Science Engineering*, vol. 13, no. 5, pp. 90–95, 2011.

[26] A. Ozsoy, M. Swany, and A. Chauhan, "Pipelined Parallel LZSS for Streaming Data Compression on GPGPUs," *International Conference on Parallel and Distributed Systems*, vol. 0, pp. 37–44, 2012.

[27] O. Erdogan and P. Cao, "Hash-AV; fast virus signature scanning by cache-resident filters," *Int. J. Secur. Netw.*, vol. 2, pp. 50–59, Mar. 2007. [Online]. Available: http://dx.doi.org/10.1504/IJSN.2007.012824

[28] *NVIDIA CUDA Occupancy Calculator*, Retrieved April 20, 2013 from https://developer.nvidia.com/cuda-downloads.

[29] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM*, vol. 46, pp. 1–13, 1999.

[30] H. Hyyro, "A bit-vector algorithm for computing levenshtein and damerau edit distances," *Nordic Journal of Computing*, p. 2003, 2003.

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

[32] J. Kloetzli, B. Strege, J. Decker, and M. Olano, "Parallel longest common subsequence using graphics hardware," in *Proceedings of the 8th Eurographics conference on Parallel Graphics and Visualization*, ser. EG PGV'08, 2008.

[33] R. A. Chowdhury and V. Ramachandran, "Cache-oblivious dynamic programming," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, ser. SODA '06. New York, NY, USA: ACM, 2006, pp. 591–600. [Online]. Available: http://doi.acm.org/10.1145/1109557.1109622

[34] E. W. Edmiston, N. G. Core, J. H. Saltz, and R. M. Smith, "Parallel processing of biological sequence comparison algorithms." *International Journal of Parallel Programming*, vol. 17, no. 3, pp. 259–275, 1988. [Online]. Available: http://dblp.uni-trier.de/db/journals/ijpp/ijpp17.html

[35] Y. S. Jiaoyun Yang, Yun Xu, "An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs," *Processings of the World Congress on Engineering*, vol. 1, 2010.

[36] A. Khajeh-Saeed, S. Poole, and J. B. Perot, "Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors," *Journal of Computational Physics*, vol. 229, no. 11, pp. 4247–58, 2010.

[37] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, no. 6, pp. 341–343, Jun. 1975. [Online]. Available: http://doi.acm.org/10.1145/360825.360861

[38] K. Kawanami and N. Fujimoto, "Facing the multicore-challenge ii," R. Keller, D. Kramer, and J.-P. Weiss, Eds. Berlin, Heidelberg: Springer-Verlag, 2012, ch. GPU accelerated computation of the longest common subsequence, pp. 84–95. [Online]. Available: http://dl.acm.org/citation.cfm?id=2340646.2340658