Towards Tera-scale Performance for Longest Common Subsequence using Graphics Processors

Adnan Ozsoy, Arun Chauhan, Martin Swany
School of Informatics and Computing, Indiana University, Bloomington

Longest Common Subsequence

A **subsequence** of a string of symbols is derived from the original string by deleting some elements without changing their order. ***{b,c,e}*** is a subsequence of ***{a,b,c,d,e}***.

A classic computer science problem is finding the longest common subsequence (LCS) between two or more strings. Widely encountered in several fields:

- The sequence alignment problem in bioinformatics
- Voice and image analysis
- Social networks analysis - matching event and friend suggestions
- Computer security - virus signature matching
- In data mining, pattern identification

Traditional Solutions

Arbitrary number of input sequences is NP-hard. Constant number of sequences, polynomial time.
Two main scenarios :

- One-to-one matching: two input sequences are compared
- One-to-many matching (MLCS) one query sequence is compared to a set of sequences, called subject sequences

A straightforward way to solve MLCS is to perform one-to-one LCS for each subject sequence. Most popular solution to one-to-one LCS problem is using dynamic programming.

Dynamic Programming Approach

Fill 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.

	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

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

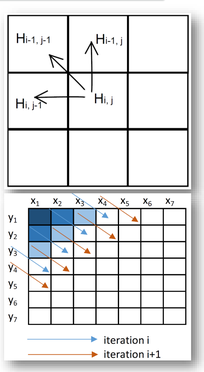
$$H(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ H(i-1, j-1) + 1 & \text{if } a_i = b_j, \\ max(H(i, j-1), H(i-1, j)) & \text{otherwise} \end{cases} \quad 2$$

Challenges

Scoring matrix creates three-way dependencies. This prevents parallelization along the rows/columns. A possible solution is to compute all the cells on an anti-diagonal in parallel as illustrated.

Problems:

1. Parallelism is limited - the beginning and the end of the matrix
2. Memory access patterns are not amenable to coalescing
3. $O(N^2)$ memory requirement
4. Poor distribution of workload
5. Sub-optimal utilization of GPU resources
6. Lack of heterogeneous (CPU/GPU) resource awareness



Objective

Optimizing MLCS on GPUs by leveraging its semi-regular structure and identifying 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.

Proposed Solution

Observations:

- Need matching information for every single element in the sequences
- Binary matrix representation summarizes the matching results
- Such a matrix can be computed efficiently on GPUs
  - Highly data parallel
  - Uniform workload distribution across GPU threads
  - Non control-flow divergence
- Each entry in the binary matrix is stored as a bit
  - Reduces space requirement
  - Adds word-size parallelism through bit operations
- Matches can be pre-computed for each symbol in the alphabet

	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

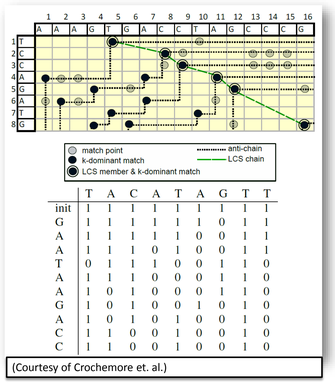
	A	T	C	G	A	G	T
T-String	0	1	0	0	0	0	1
A-String	1	0	0	0	1	0	0
C-String	0	0	1	0	0	0	0
G-String	0	0	0	1	0	1	0

Bit-vector Approach

Row-wise bit operations on the binary matrix can be used to compute a derived matrix that provides a quick readout of the length of the LCS. Allison and Dix compute the length of the LCS using bit representation, in 1986 (Eq 3). The algorithm follows and marks each anti-chain to identify k-dominant matches in each row. Crochemore at al.'s algorithm uses fewer bit operations that lead to better performance (Eq 4).

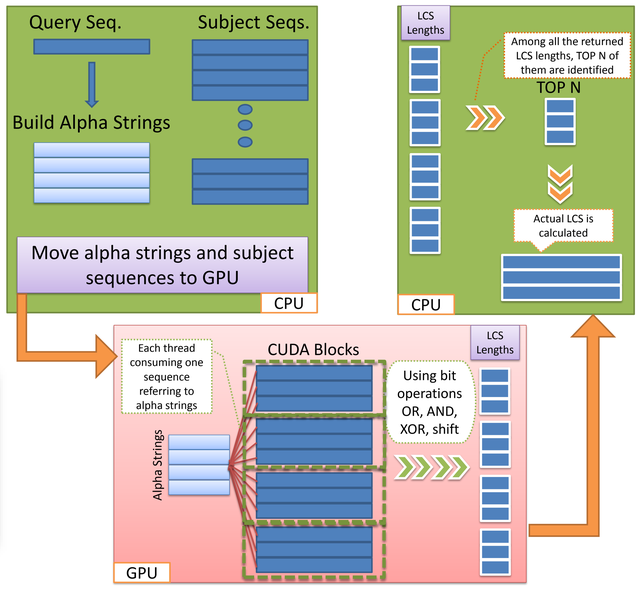
$$\begin{aligned} X &= Row_{i-1} \quad OR \quad M_i \\ Row_i &= X \; AND \; ((X - (Row_{i-1} << 1)) \quad XOR \quad X) \end{aligned} \quad 3$$

$$Row_i = (Row_{i-1} + (Row_{i-1} \; AND \; M_i)) \; OR \; (Row_{i-1} \; AND \; M_i') \quad 4$$



(Courtesy of Crochemore et. al.)

Design Principles



Parallelizing and Optimizing on GPUs

- Intra- vs. Inter-task Parallelism
  - One subject sequence is assigned to each CUDA thread
- Memory Spaces
  - Constant and shared memory usage
- Hiding Data-copying Latencies
  - Hide data-copying latencies with asynchronous execution
- Leveraging Multiple GPUs
- Streaming Sequences
  - Multiple runs of execution to consume large sets

Results

Metrics

The unit that is used to represent our results is cell updates per second (CUPS).
R -length of the query reference
T -total length of subject sequences
S -time it takes to compute

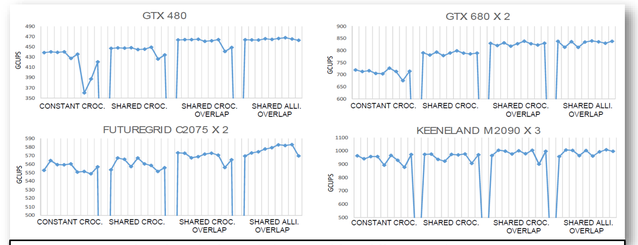
$$CUPS = \frac{RxT}{S} \quad 5$$

Testbed configurations

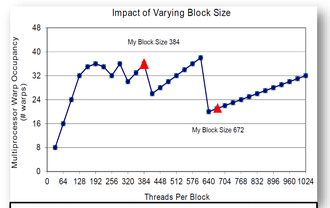
System	GPU	Number of Cores	Device Memory	CUDA Capability	CUDA Version	CPU	Host Memory	OS
FutureGrid	C2075 x 2	448	5375 MB	2.0	5.0	X5660	192 GB	Red Hat 6.3
Keeneland	M2090 x 3	512	5375 MB	2.0	4.2	X5660	24 GB	CentOS 6.3
	GTX 680 x 2	1536	2048 MB	3.0	5.0	AMD PII X6	16 GB	Debian 4.4
	GTX 480	480	1535 MB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

	Constant Croch.	Shared Croch.	Shared Croch. Overlap	Shared Allison Overlap
GTX 480	26	23	23	19
C2075	24	24	24	18
GTX 690	24	25	25	19
M2090	26	23	23	19

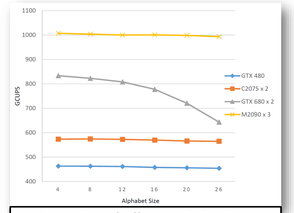
Register signatures for different versions



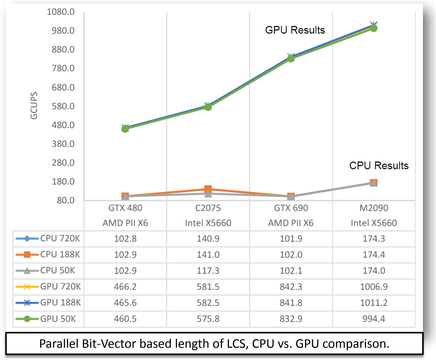
Combined results for each device and four different implementations.



CUDA Occupancy Calculator (Courtesy of NVIDIA).



Comparison of different alphabet sizes.



Parallel Bit-Vector based length of LCS, CPU vs. GPU comparison.

Conclusion & Future Work

- Bit-vector operations that exploit word-level parallelism
- Inter-task parallelism
  - Independent computation: each one-to-one LCS comparison
- Allison-Dix and Crochemore et. al. on GPUs
- Analysis and design steps of GPU specific memory optimizations
- Post-processing
  - Benefit from the heterogeneous environment
- Achieve Tera CUPs for MLCS with three GPUs, a first for LCS algorithms
- 8.3x better performance than multi-threaded CPU implementation on 12 cores
- Sustainable performance with very large data sets
- Two orders of magnitude better performance compared to previous related work

- ❖ Investigate gap-penalty - Needleman-Wunsch, Smith-Waterman
- ❖ Distributing over multiple nodes
- ❖ Common problem space in string matching

Acknowledgements

FutureGrid, Keeneland, Indiana University, NVIDIA Hardware Request Program

Contact Information

aozsoy@indiana.edu, achauhan@indiana.edu, swany@indiana.edu