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

Adnan Ozsoy
Indiana University,
Bloomington, IN 47405
aozsoy@indiana.edu

Arun Chauhan
Indiana University,
Bloomington, IN 47405
achauhan@indiana.edu

Martin Swany
Indiana University,
Bloomington, IN 47405
swany@indiana.edu

1. EXTENDED ABSTRACT

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 poster 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, 12]. 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 [7]. 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.

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]. Usually, a newly discovered sequence is compared against a database of known sequences. The problem can be phrased as a variant of LCS. In voice and image analysis LCS is used for a variety of tasks such as improving speech recognition [5], evaluating machine translation [8], and image retrieval through structural content similarity [10]. In social networks LCS is used for matching event and friend suggestions [13]. 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 [6] and database query optimization [11].

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. 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. The LCS of two strings X_n of Y_m, of lengths n and m, respectively, can be expressed as:

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

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).

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

The construction of the scoring matrix creates three-way dependencies, 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. 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.

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

[1] We call these *irregular algorithms*.

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. 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 the 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 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 pre-compute data that can be cached.

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.

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.

Allison and Dix proposed solving the LCS problem using bit representation, in 1986 [2]. 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. Allison and Dix gave the following equation to calculate the length of the LCS where Row_0 starts with all zeros and M are the precomputed alphabet-strings.

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

In this poster, 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 [2] and Crochemore et al. [3]. 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. Compared to CPU implementation with bit-vector approach, we receive 8.3X times speed up in the best case. Compared to the previous work that adopted dynamic programming solution, our bit-vector approach on GPUs performs one to two orders of magnitude better.

2. REFERENCES

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

[2] L. Allison and T. I. Dix. A bit-string longest-common-subsequence algorithm. *Inf. Process. Lett.*, 23(6):305–310, Dec. 1986.

[3] 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*, 80:279–285, 2000.

[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. pages 258–261, Barcelona, Spain, Oct. 2004. Universitat Pompeu Fabra.

[6] T. S. Han, S.-K. Ko, and J. Kang. Efficient subsequence matching using the longest common subsequence with a dual match index. MLDM '07, pages 585–600, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] 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.

[8] C.-Y. Lin and F. J. Och. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, ACL '04, Stroudsburg, PA, USA, 2004. Association for Computational Linguistics.

[9] S.-y. Lu and K.-S. Fu. A sentence-to-sentence clustering procedure for pattern analysis. *Systems, Man and Cybernetics, IEEE Transactions on*, 8(5):381–389, 1978.

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

[11] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[12] A. L. Shimpi. Inside the Titan Supercomputer: 299K AMD x86 Cores and 18.6K NVIDIA GPUs. *AnandTech online computer hardware magazine*, October 2012.

[13] X. Tian, Y. Song, X. Wang, and X. Gong. Shortest path based potential common friend recommendation in social networks. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 541–548, 2012.