Parallel Approaches to Edit Distance and Approximate String Matching

Cary Yang, Kevin Zhang

Carnegie Mellon University
caryy@andrew.cmu.edu, kkz@andrew.cmu.edu

Abstract

In this paper, we explore approaches to parallelizing the edit distance problem and the related approximate string matching problem. The edit distance is a measure of the number of individual character insertions, deletions, and substitutions requried to transform one string into another string. In the canonical dynamic programming solution to the edit distance, a chain of dependencies renders parallelization extremely difficult; thus, we investigate several different approaches to resolve this issue.

1 Summary

We implemented two different algorithms to solve the edit distance and related approximate string matching problem with CUDA on the GPU and `pthreads` on the CPU and compared the performance of these different algorithms and implementations.

2 Background: Edit Distance

The Edit Distance is a measure of dissimilarity between two strings. Formally, the edit distance between strings S and T is the minimum number of operations required to transform S to T. The valid operations are:

- Insertion: ex: "ab" $\Rightarrow$ "axb"
- Deletion: ex: "abc" $\Rightarrow$ "ac"
- Substitution: ex: "abc" $\Rightarrow$ "xbc"

A recursive definition of edit distance can now be constructed. Let $S = s_1 \dots s_m$ and $T = t_1 \dots t_n$ be two strings of length m and n, respectively. Let $D_{i,j}$ denote the edit distance between the first i characters of S and the first j characters of T.

- In the base case, we have $D_{i,0} = D_{0,i} = i$ for all applicable i.
- For non-zero m, n, $D_{m,n} = D_{m-1,n-1}$ if $s_m = t_n$. Otherwise, $D_{m,n} = 1 + max(D_{m-1,n}, D_{n-1,m}, D_{m-1,n-1})$.

This recursive definition lends itself nicely to a dynamic programming (DP) solution. Namely, we can evaluate in row-major order a $M \times N$ matrix whose (i, j) entry is exactly $D_{i,j}$. In the table below, we see the entries evaluated for the input strings "SPARTAN" and "PART".

String "T"

String "S"	""	P	A	R	T
""	0	1	2	3	4
S	1	1	2	3	4
P	2	1	2	3	4
A	3	2	1	2	3
R	4	3	2	1	2
T	5	4	3	2	1
A	6	5	4	3	2
N	7	6	5	4	3

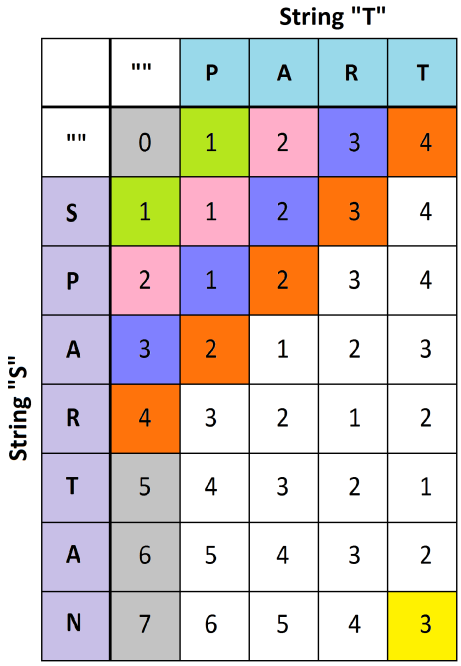
In a straightforward implementation, we may allocate the entire table and evaluate the entries in row order. Clearly, this algorithm runs in $\Theta(M * N)$ time and space. Being a little more keen on space requirements, we may note that at any given moment, only two rows of space is required to obtain the edit distance (lower-right entry indicated in yellow). To accomplish this, we simply "leap-frog" the rows as we evaluate the table. This has the effect of throwing away previous rows once they are not required for the current row. If we force the second input to be the smaller string, this reduces space requirements to $\Theta(min\{M, N\})$.

To analyze possible parallel approaches, we must first determine critical dependencies between the entries of the table. It is not hard to see that in the DP table, each entry depends only on the upper neighbor, the upper-left neighbor, and the left neighbor. In traditional literature, the table is often evaluated in row-major order. This poses a problem for parallelization, as within a row, each entry depends on the previous entry. However, if we process the table in "diagonal-major" order, then new doors are opened for parallel approaches.

3 Approach 1: Parallel Dynamic Programming

3.1 Details

The key observation to our approach is that the entries of a single diagonal row of the DP table can be evaluated independently, provided that previous diagonals have been computed. To see this more clearly, refer to the next image. As can be seen, the entries of the blue diagonal (size 4) can be compuated independently once the values of the green and pink diagonal (sizes 2 and 3) have been evaluated. Similarly, the entries of the orange diagonal can be computed in parallel once the blue diagonal is complete. Our goal was the exploit this avenue of parallelism and implement a multi-threaded algorithm to compute the edit distance between very large strings.



Similar to before, only three diagonals are required at any point in order to obtain the edit distance. Since the length of the longest diagonal is equal to the length of the smaller input, we again have reduced our space requirements to $\Theta(min\{M, N\})$.

Our first task was to implement a serial diagonal-order implementation in c++. Because the diagonals of the table exhibit non-uniform lengths, this proved to be an interesting challenge. Once complete, we immediately observed the following results for two 50k-sized inputs:

- Row-order: [16.849 seconds]
- Diagonal-order: [10.324 seconds]

The runtimes may initially seem odd. After all, the diagonal-order evaluation does no less work than the row-order evaluation; if anything, it requires more work due to the overhead of managing complicated indices. After some consideration, we attribute the apparent speedup to instruction-level parallelism (ILP), an optimization performed by specialized CPU hardware. ILP is not present in the row-order evaluation due to the lack of independence

between the entries of individual rows.

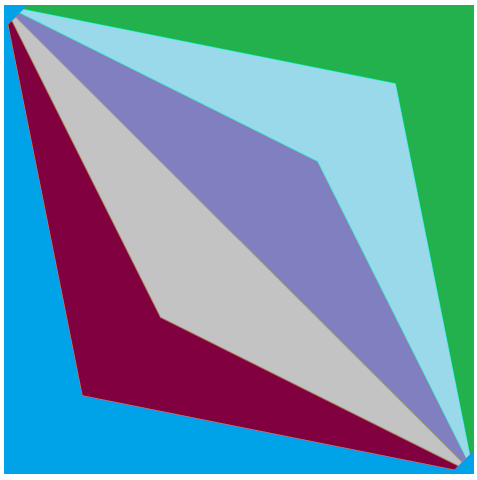
We were now ready to parallelize the diagonal-order approach. The high level design was simple: for each diagonal, split the work as evenly as possible among multiple threads.

3.2 Implementation

Our parallel implementation was designed around the HT-enabled 6-core Intel Xeon processor; however, the concept is easily extensible to other systems.

Our approach can be summarized as follows. As we traverse the table, we allow a single thread to perform all the computations when the size of the diagonal is below a serial threshold. This is to avoid the overhead of running multiple threads on small pieces of work. Once the diagonal is of sufficient length, work is split evenly among all threads. At the end of each diagonal, a synchronization barrier is called for all threads; this is required due to inter-diagonal dependence.

A visual mapping of six threads (one per color) to their respective portion of the DP table is provided below.



As can be seen, a single (blue) thread is responsible for small “caps” in the upper left and lower right corners of the table.

4 Background: Approximate String Matching

Approximate string matching is a very similar problem to edit distance, and as we'll see soon, involves essentially the same computation. The main difference is that where edit distance asks the question "what is the edit distance between these two strings", approximate string matching asks "given a pattern P, a text T, and an integer k, what are all locations j where for some $i < j$, the edit distance between P and $T[i...j]$ is less than k".

An example of this computation follows.

Let the edit distance function be given as $ed(s,t)$. Given P = "match", T = "remachine", and $k = 3$, the algorithm for approximate string matching will output 5, 6, and 7 since:

- $ed(”match”,”mac”) = 2$
- $ed(”match”,”mach”) = 1$
- $ed(”match”,”machi”) = 2$

It may seem like this involves more computation than the original edit distance problem, but it can be solved using exactly the same recurrence, just a different base case and a little post-processing.

Note that the edit distance dynamic programming matrix for the above strings "match" and "remachine" is as follows:

	""	r	e	m	a	c	h	i	n	e
""	0	1	2	3	4	5	6	7	8	9
m	1	1	2	2	3	4	5	6	7	8
a	2	2	2	3	2	3	4	5	6	7
t	3	3	3	3	3	3	4	5	6	7
c	4	4	4	4	4	3	4	5	6	7
h	5	5	5	5	5	4	3	4	5	6

Figure 1: *Dynamic programming matrix for edit distance*

To solve the approximate string matching problem using the same procedure, we first initialize the first row to all 0's, and apply the same recurrence.

	""	r	e	m	a	c	h	i	n	e
""	0	0	0	0	0	0	0	0	0	0
m	1	1	1	0	1	1	1	1	1	1
a	2	2	2	1	0	1	2	2	2	2
t	3	3	3	2	1	1	2	3	3	3
c	4	4	4	3	2	1	2	3	4	4
h	5	5	5	4	3	2	1	2	3	4

Figure 2: *Dynamic programming matrix for approximate string matching*

Then, we simply scan along the bottom row and output all positions where the value is less than k. Note the values 2, 1, and 2 at positions 5, 6, and 7 in the last row of the matrix.

What this transformations does is essentially make insertions until the beginning of the pattern free, allowing us to start the match anywhere in the text. Scanning along the bottom row and taking any values less than a certain threshold instead of just taking the last value makes deletions at the end of the pattern free also, allowing us to end the match against the pattern anywhere in the text. Thus, this modified process finds the edit distance between the pattern and any substring of the text, which is exactly what we want.

Therefore, as we can see, the procedure is analogous to the edit distance procedure, and therefore, contains the same dependencies. To solve with a naive, serial method would result in an $O(mn)$ algorithm, where m is the length of one string and n is the length of the other, exactly like edit distance.

5 Approach 2: Bit-parallel Algorithm

5.1 Details

One observation that can be made about the dynamic programming matrix for both the edit distance computations and the approximate string matching is that the difference in edit distance value between adjacent cells is either -1, 0, or 1.

Furthermore, in approximate string matching, the matrix is fully determined by specifying the vertical deltas between the values in the matrix since the top row is entirely 0.

Given these two facts, we can store column j of the matrix as 2 bit-vectors, Pv_j and Mv_j where the ith bit of Pv_j, or $Pv_j(i)$, is 1 iff the cell (i, j) in the dynamic programming matrix for approximate string matching has a value 1 greater than its neighbor directly above it and $Mv_j(i) = 1$ iff it's the value is 1 less than its neighbor directly above it.

Note that Pv_j and Mv_j are mutually exclusive, that is, $Pv_j \,\&\, Mv_j = 0$, and at all positions where $Pv_j(i) = 0 \wedge Mv_j(i) = 0$, the vertical difference between cell (i, j) and its neighbor above is 0. Therefore, these two bit-vectors fully specify the jth column.

Using these two bit-vectors, we can proceed from one column to the next solely with the use of bit-level operations!

If we let m be the length of the pattern and n be the length of the text, then starting with $Pv_0 = 1^m$ and $Mv_0 = 0$, we can perform the following operations to get Pv_1 and Mv_1 and eventually, the rest of the matrix [3].

$$
\begin{aligned}
Xv_j(i) &= Peq[t_j](i) \text{ or } Mv_{j-1}(i) \\
Xh_j(i) &= \\
&\quad (((Peq[t_j] \,\&\, Pv_{j-1}) + Pv_{j-1}) \oplus Pv_{j-1}) | Peq[t_j] \\
Ph_j(i) &= Mv_{j-1}(i) \text{ or not } (Xh_j(i) \text{ or } Pv_{j-1}(i)) \\
Mh_j(i) &= Pv_{j-1}(i) \text{ and } Xh_j(i) \\
Score_j &= \\
&\quad Score_{j-1} + (1 \text{ if } Ph_j(m)) - (1 \text{ if } Mh_j(m)) \\
Ph_j(0) &= Mh_j(0) = 0^2 \\
Pv_j(i) &= \\
&\quad Mh_j(i-1) \text{ or not } (Xv_j(i) \text{ or } Ph_j(i-1)) \\
Mv_j(i) &= Ph_j(i-1) \text{ and } Xv_j(i)
\end{aligned}
$$

The *Score* variable in the equations accumulates the actual values along the bottom row of the dynamic programming matrix and allows us to determine the locations of minimal distance.

This procedure allows us to process w cells in the dynamic programming matrix at once, where w is the largest word size on the given machine, and is generally 64 for most machines, giving a potential 64x speedup over the serial row implementation described in 3!

5.2 Implementation

We implemented this algorithm both serially on the CPU and in parallel with CUDA on the GPU.

The CPU implementation was reasonably simple to implement: just translate the above algorithm into C++ code. This algorithm was essentially entirely serial, relying only on the bit-level parallelism inherent in the algorithm.

To attempt to extract even more parallelism from this algorithm and take advantage of the numerous multiprocessing units available in modern machines, we went beyond the algorithm's paper and wrote a CUDA implementation that uses arbitrary length bit-vectors represented as chunks of 64-bit segments, one on each multiprocessor. Therefore, theoretically, we could process each column of the matrix simultaneously!

Unfortunately, CUDA has several limitations that reduce the practicality of this approach.

First off, we can go back and see that the final updating step of $Pv_j(i)$ and $Mv_j(i)$ rely on $Ph_j(i-1)$ and $Mh_j(i-1)$. Because the bits are spread across processing units, there would be necessary communication between processing units at this step. However, CUDA's block abstraction does not permit synchronization between blocks, and therefore, we are limited to only 1 block.

Additionally, our code was run on NVIDIA GTX 480 GPUs, which support up to CUDA Compute Capability 2.0, limiting the theoretical maximum number of threads to 1536 and the registers per thread to 63. However, at higher register counts, the number of maximum threads per block is limited even further, and because our kernel uses 41 registers, our actual maximum thread count is 768. Therefore, our CUDA kernel only supports pattern lengths up to 1 block $*$ 768 threads / block $*$ 64 characters $=$ 49,152 characters, which is not large enough to see the benefits of highly parallel computation.

Furthermore, the second step of the computation adds together bit-vectors and relies on the carrying property of addition. Because we have segments that are spread out across multiprocessors that represent parts of the same bit-vector, we need arbitrary precision addition.

First, we tried computing the entire addition serially on one thread, but this was unacceptably slow due to the inherent serial nature of the computation.

We eventually settled on an implementation that uses `scan` as a primitive to perform the carrying propagation.

Here's how that works. First, we allocate a shared array of length equal to the number of processing units. Then, each processing unit stores into the array at its thread index one of the following values:

- `GEN` if the addition of the segment of the bit-vector that this processing unit owns overflowed.
- `PROP` if the addition did not overflow and the result is the maximum allowable value for the bit-vector datatype (`0xFFFFFFFFFFFFFFFF` for 64-bit representations).
- `STOP` if neither of the above are true.

Then, we perform the exclusive scan, with the operator being (in SML):

```
fun propFlags (c, PROP) = c
  | propFlags (_, c) = c
```

This essentially propagates the `GEN` and `STOP` through `PROP`s. Afterwards, we check the scan result at our index and add 1 to our addition result if the array has a `GEN`, or do nothing otherwise.

Results of these two implementations can be seen in the next section.

6 Results

We offer graphs of the runtime of our algorithms at various input sizes and thread counts to show its effectiveness.

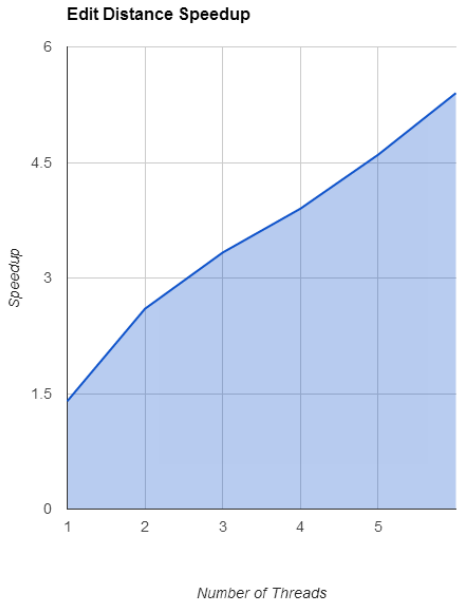
All CPU tests were run on an Intel Xeon W3670 (6 cores, 3.2GHz) processor, and the GPU tests was run on an NVIDIA GTX 480 GPU.

6.1 Edit Distance

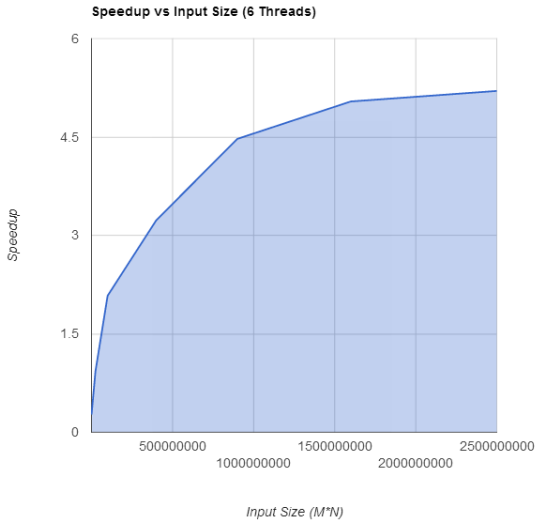
The following is the speedup graph for our parallel diagonal implementation of edit distance (described in section 3.2.). The input sizes of both strings were fixed at 50k, and measurements were taken with respect to a serial row-order implementation.

Recall that ILP plays a crucial row in the performance gain; this is the reason for a 1.4x speedup even on a single thread.

As seen, appreciable speedup is observed up to 6 threads (running on a 6-core Intel Xeon). The speedup takes a significant drop at 7 or more threads, suggesting that the current implementation is compute-bound, as using more threads exchanges computational throughput for latency hiding.



If we instead fix the usage of six threads and change the input size, we obtain the following speedup chart. The x-axis represents the problem size, which is defined as the product of the lengths of two same-sized input strings (recall $M * N$). As shown, our implementation does not achieve maximum speedup until the problem size is around 2.5 billion.



There are many factors which may contribute to a loss of speedup in smaller problem instances. In addition to the typical intuitive understanding of multi-threading overhead, another key observation can be made.

Recall that the number of diagonals for a square $N \times N$ table is on the order of N. Thus, the number of barriers required is also on the order of N. Consequently, the frequency of barrier calls is on the order of $N/N^2 = 1/N$. Thus, when N becomes large, a smaller and smaller fraction of time is spent on synchronization. This reduces overall overhead of using multiple threads, and results in a greater speedup.

6.2 Edit Distance + String Matching

In the below graphs, `ROW`, `DIAG`, and `DIAG (6 CPU)` reference the 3 implementations of the traditional dynamic programming parallelization discussed in Section 3. `BPM-CPU` and `BPM-CUDA` reference the bit-parallel approximate string

matching implementations from Section 5.2.

First, we have a graph of runtimes where the pattern length is kept constant. Recall that the BPM algorithms parallelized along the pattern side of the matrix, so we see some interesting results here.

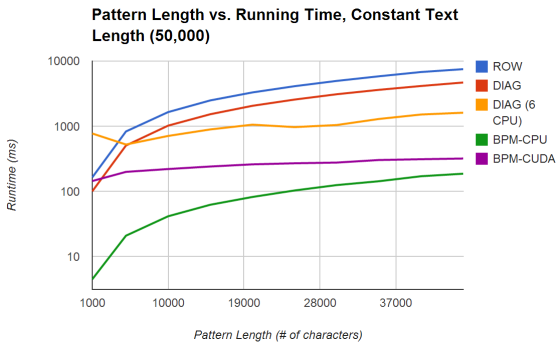


Figure 3: *Graph of running time of our implementations, keeping text length constant.*

The first thing to note is that the first 3 algorithms performed as expected. However, one little difference is that the 6 CPU implementation was actually slower on small enough patterns. We attribute this to the overhead in launching threads for such small pattern lengths.

The BPM algorithms were slightly more interesting. First, we can see that the BPM-CPU implementation is significantly faster than the parallel dynamic programming solutions, however, it did not quite reach its promised 64x speedup, stopping short at 40x speedup. This is most likely due to the fact that the bit-parallel algorithm performs significantly more computation than the serial dynamic programming implementation, which just performs a `min` at each step, and therefore cannot reach the speedup advertised.

Even more interesting is the fact that the CUDA implementation for the bit-parallel algorithm is slower than the CPU implementation! This can be attributed to the fact that, as we learned from earlier in this course, parallel scan is slower than sequential scan until a large number of elements due to the overhead of synchronization and the slower individual GPU multiprocessors. As can be seen from the slopes of the lines in the graph, it is likely that the CUDA implementation would eventually surpass the CPU implementation. However, this could not be tested since, as discussed in Section 5.2, the maximum pattern length we could test was 49,152.

Next, we have a graph of the effects, on running time, of varying text length with a constant (and large) pattern length. Note that all algorithms discussed in this paper are linear with respect to text length.

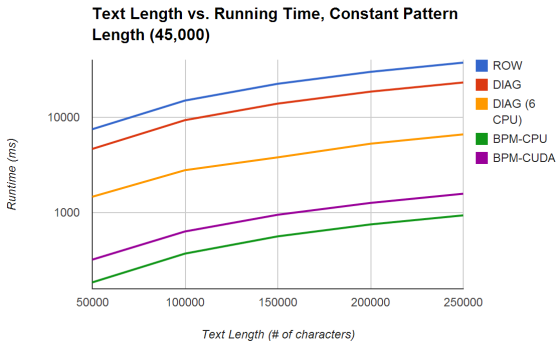


Figure 4: *Graph of running time of our implementations, keeping pattern length constant.*

This data is not particularly interesting, but is included for completeness. As we can see, all implementations are indeed linear with respect to text length, with little to no deviation from that pattern since any overhead caused by the difference in implementation is masked by the large input sizes and runtimes.

7 Future Work

Future work on this topic would involve first attempting to reduce the register count of our CUDA implementation of the bit-parallel algorithm to allow for more threads per block, and therefore, higher input sizes. Additionally, we would also try to acquire and run our CUDA implementation on a graphics card that supports higher CUDA compute capability to allow for more registers and more threads per block, to again allow for higher input sizes.

If these two goals were to be completed, we would probably be able to finally see the CUDA implementation beat the CPU implementation in terms of runtime, since the benefits of parallel scan should become more apparent once higher input sizes are allowed.

After this goal is complete, the next step would be to parallelize row-wise in the matrix instead of column-wise, since the text we're searching in is along the row of the matrix, and is, in practical cases, orders of magnitude larger than the length of the pattern text (in computational biology applications, generally the length of the pattern is around 30 and the length of the text to search in is several billion base pairs [1]). A procedure to do this transposition can be found in [2].

References

[1] Eugene Brown, Mark Chee, Thomas Gineras, David Lockhart, and Gordon Wong. Expression monitoring by hybridization to high density oligonucleotide arrays, 2001.

[2] Kimmo Fredriksson. Row-wise tiling for the myers' bit-parallel approximate string matching algorithm. In *String Processing and Information Retrieval*, pages 66–79. Springer, 2003.

[3] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.

Work by Each Student

Equal work was performed by both project members.