

# Online Approximate String Matching with CUDA

Mikael Onsjö (mikael@is.titech.ac.jp), Yoshinori Aono  
group: “patternmatchers”, supervisor: Osamu Watanabe  
Tokyo Institute of Technology

2009-03-23

## Abstract

Approximate string matching is an important problem in various fields such as natural text searching or when working with large sets of DNA data. We study the bit-parallel approximate string matching algorithms of Baeza-Yates, Navarro [1] and of Hyrrö [2]. We show how to implement these in an efficient and natural way for certain parallel architectures. Specifically we compare the sequential and parallel implementations on an AMD Opteron 2.4 GHz and on an Nvidia T1 processor (GPU) respectively. The speedup in this case is about 50 times (the AMD takes 50 times longer than the T1) when compared for searches of patterns of length 1024 characters in the DNA of the fruit fly, *Drosophila Melanogaster* (165 million base pairs [characters]). E.g., for edit distance 15, the T1 was able to find all matches in 8.5 seconds whereas it took 406 seconds for the AMD.

## 1 Introduction

Approximate string matching is the task of finding all substrings in a body of *text* (of length  $n$ ), that are within a given *edit distance* ( $k$ ) of a given pattern (of length  $m$ ). By “online” we mean that we cannot preprocess the text, this could for instance be because the text is not known in advance or because the text (set of texts) is so large as to make any relevant preprocessing infeasible. This problem is of high importance in many areas, e.g. for search engines when looking for pages relevant to a query and in bioinformatics when analyzing huge sets of DNA sequencing data.

Though an  $O(mn)$  dynamic programming approach [3] was known already in the 1970s, the importance of the problem has inspired significant research and many algorithms have been proposed. Arguably the most “practical” approach has been a nondeterministic finite automaton (NFA) with binary states encoded efficiently into a small number of computer words. This has been studied and gradually improved on in a series of papers with important additions by Baeza-Yates and Navarro [1] (1999) and Hyrrö [2] (2008). These two publications form the basis for the one we present here.

In [2] the size of the NFA is  $(m - k)(k + 1)$  states. If this is less than the word size ( $W$ ) of the machine, the entire NFA can be encoded in a single word and the running time per letter of text is just  $O(1)$ . Particularly in many bioinformatics applications this is, however, not nearly enough as we may well desire to search for patterns of around 1000 characters. For such cases, [1] describes how to encode the NFA efficiently into some number  $w$  of computer words that are then updated sequentially by a normal computer. Even so, this strategy may be the fastest one yet known for the case where  $k \ll m$ , which is the situation in most relevant applications.

We show how to implement the same algorithm in CUDA for a T1 processor from Nvidia. In this case the  $w$  words can be updated virtually in parallel (though technically in warps of 32 or half-warps of 16 at a time) in a natural way, as long as  $w \leq 512$ . This allows, e.g., the parameter combination  $k = 15$  and  $m = 1039$  or longer patterns still if  $k$  is reduced below 10.

For experimentation we use the DNA of the fruit fly as obtained from “www.fruitfly.org” (160MB in fasta [ascii] format, 165 million base pairs) and a pattern of 1024 characters arbitrarily taken from the DNA. We find that the speedup between a sequential 32-bit AMD Opteron, 2.4 GHz (presently on the TSUBAME super computer) and the T1 GPU is around 50 times in the favor of the GPU.

We will proceed by giving a brief description of the general NFA algorithm. The reader who is more interested in the implementational aspects may wish to skip the next section.

## 2 The NFA approach in general

For an extensive explanation of the NFA approach we refer to the papers [1] and [2]. To make this paper more self-contained, however, we give an outline here:

The original NFA is defined by a  $(k+1) \times (m+1)$  matrix of binary states (active=match or inactive=no-match), let the states be called  $s_{ij}$ . States of the first column,  $s_{i0}$ , are always active, signifying that the empty string is always matched. Subsequent columns are labeled in order by the characters of the pattern. The text is scanned once, character by character from start to end. For every new character of text, the entire NFA is updated in parallel according to the following rules, for each state  $s_{ij}$ ,  $i, j > 0$ :

- match:** If  $s_{(i-1)j}$  is active and the current text character matches the pattern character of column  $i$ , then  $s_{ij}$  becomes active.
- replace:** If  $s_{(i-1)(j-1)}$  is active then  $s_{ij}$  becomes active.
- insert:** If  $s_{i(j-1)}$  is active then  $s_{ij}$  becomes active.
- delete:** If  $s_{(i-1)(j-1)}$  becomes active by any rule, then so does  $s_{ij}$ .

and similarly for  $i = 0, j > 0$  but with only the first rule. If a state in the last column becomes active, then there is a corresponding match between the text and the pattern.

It turns out it is not necessary to encode and update the entire original NFA; it suffices to consider the  $(m - k)$  diagonals that start at some  $s_{0i}$ ,  $0 < i \leq m - k$  and extend downwards to the right. Figure 1 illustrates the encoding.

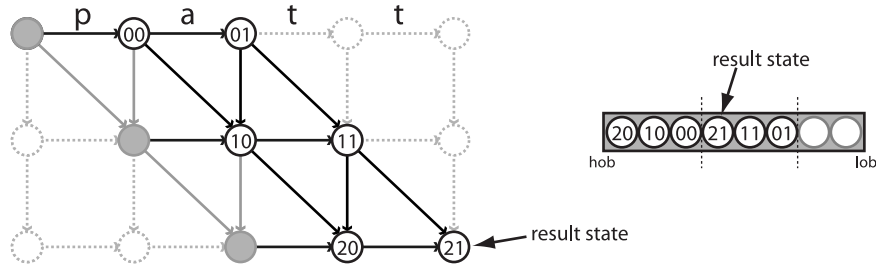


Figure 1: Efficient encoding of small NFA into 8-bit computer word.

The reason for this is that the lower left triangle of the NFA is always active and the upper right triangle such that if any state there becomes active then there will be at least one match. Ignoring the upper right triangle loses some information about how short a match can be made but this is generally considered to be of no concern at this point (formally we are only addressing the question of whether there is *some match within edit-distance k*).

As an example, figure 2 shows what happens as the pattern “tit” is matched against the text “tat” while allowing for a maximum edit distance of one (meaning the number of rows in the NFA is two).

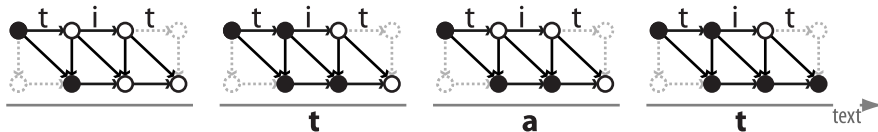


Figure 2: Example of matching.

The NFA encoded in this way can be updated with a relatively small number of basic integer operations, such as are almost always well implemented in the instruction set of relevant processors. For the particulars about the updating process, however, we refer again to the papers [1] and [2].

## 3 CUDA-implementation

An obvious and trivial way to parallelize the string matching task is to simply divide the text into many separate pieces. There are two immediate drawbacks to doing this on a single machine: On one hand the searches have to overlap by the allowed edit distance and on the other hand it seems difficult to handle

memory access in an efficient manner. We note that the architecture of the T1 processor from Nvidia and the language CUDA rather suggest a natural way to extend the bit-parallelism of the NFA algorithm. To understand why, though, we first need to know something about the architecture:

Code on the T1 is executed in blocks of up to 512 threads, this is done virtually in parallel though formally speaking only 16 or 32 operations in a block at a time are ever done strictly in parallel (8 blocks may be processed concurrently on different cores). 16 KB of *shared* memory is attached to each core in such a way that accessing it (with some care) is essentially no more expensive than an operation on a register. This memory is shared between threads in a block but not between blocks. Further it is a readily supported matter to synchronize between threads in a block but much more complicated to do the same between blocks. In addition to the shared memory there is 4GB of *global* memory that is relatively slow to access.

Though each thread (processor core) operates with individual 32-bit words, on a higher level it is possible to think about the operations of a block as being on single huge words, e.g. of size  $512 \times 32 = 16384$ . This is also how we choose to implement the NFA algorithm; each block in CUDA represents an NFA and is responsible for a separate portion of the text. Since the portions have to overlap, it is on one hand desirable to have as few as possible. On the other hand we need at the least 8 blocks to take advantage of all cores in the T1 processor and more blocks still to allow the scheduler its full potential. Experimentally we find that with little dependence on other parameters, a grid size of about 300 is optimal.

Each block reserves 512 32-bit words (uint) shared memory for the NFA (actually padded by an additional uint on each side), 512 words as a buffer for text and  $4 \times 512$  words as a buffer for the compiled pattern (512 words each for the possible characters A, C, G and T). This amounts to about 12 KB of the available 16 KB. Program execution proceeds roughly as follows:

- 1: (CPU) Get data, parameters and pattern to RAM and GPU global memory.
- 2: (CPU) Compile the pattern and put it in global memory.
- 3: (GPU) Execute Kernel per block:
- 4:     Initialize masks and constants.
- 5:     Load compiled pattern *global*  $\rightarrow$  *shared*.
- 6:     **while** more text **do**
- 7:         Load text to buffer *global*  $\rightarrow$  *shared*.
- 8:         **for** each character in text buffer **do**
- 9:             Subroutine: NFA Update
- 10:             Check for match...
- 11:         **end for**
- 12:     **end while**
- 13: (CPU) Get result from GPU.

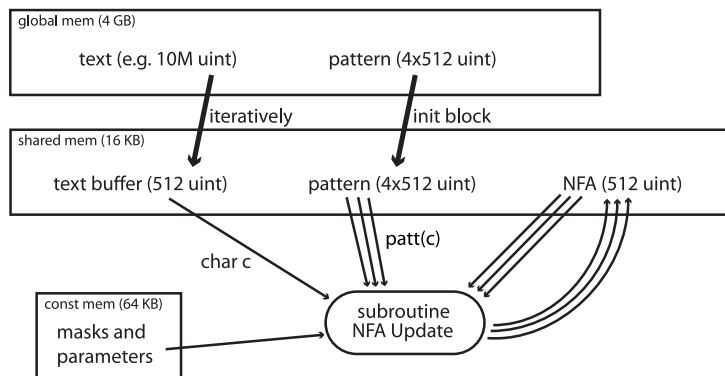


Figure 3: Usage of device memory.

We tried various variation such as to use interchanging buffers for both text and NFA, however it turns out the compiler is much better at scheduling and masking memory latency without such interference. The program was compiled with Nvidia’s *nvcc 2.0* using optimization flag *-O3*.

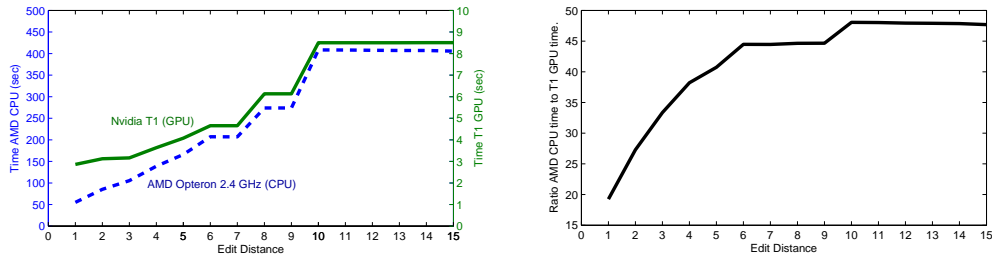
Though the loops unfortunately add some overhead in themselves, obviously the most critical part is (if not the memory operations) the subroutine “NFA Update”. An inspection of the *.ptx* file reveals that

this part is implemented with approximately 30 4-cycle instructions and it is difficult to imagine that this could be much reduced unless a completely different approach was considered.

Several considerations were taken wrt. memory latency and it now appears that all read latency is efficiently masked by arithmetic operations. The text is compressed tightly into integers, each of the 4 DNA characters represented by a two bit combination so as to minimize the amount of data that needs to be moved from global to shared memory. Access to the compiled pattern and the NFA is always done in sequences so that all the 16 memory banks are used and no conflict occurs. Access to the text buffer is always to the same position by all threads so as to enable broadcasting.

## 4 Results and Notes

Sequential and parallel implementations of the NFA algorithm were compared using a sample set of DNA from the fruit fly as obtained from [www.fruitfly.org](http://www.fruitfly.org). This set is about 160MB in uncompressed fasta format and contains slightly more than 162 million DNA characters (A, C, G or T). An arbitrary substring of 1024 characters was selected from the text and the algorithms run for edit distances in the interval  $[1, 15]$ . The result is presented as a graph in figure 4 (note the difference between the axes for the curves!). For  $k = 15$  the GPU was able to find all matches in 8.5 seconds whereas it took the CPU 406 seconds, i.e. 48 times longer.



**Figure 4:** Running times of NFA algorithm with sequential and parallel implementations. The times are plotted (with different scales) in the left figure and the ratio between the two time sequences in the right.

The notable irregularities in both curves in figure 4 correspond to places where the number of NFA diagonals packed into each 32-bit computer word, changes. Since  $m$  is fixed, only  $k$  affects the number of words used and for certain intervals of  $k$ , this remains constant. E.g.  $k \in [8, 9]$  implies that we pack three diagonals into each word whereas  $k \in [10, 15]$  implies two.

Our present implementation assumes that each word contains two or more full diagonals of the NFA, meaning  $k < 16$ . The reason for this is that we wish to make the sub routine “Update NFA” as simple as possible. For larger  $k$  (as for certain cases of  $k$  in the interval  $[1, 15]$ ) it is possible to pack the diagonals more tightly (use almost every bit of the words) by using a more complicated scheme that is also described in [1]. Then, however, the sub routine has to be modified slightly; the best way is probably to have different subroutines for several different cases (e.g. for the cases of one diagonal per word, several diagonal per word or several words per diagonal). For very large  $k$  this is unlikely to be of much interest as there appear to be faster algorithms known, see e.g. [2].

## References

- [1] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23:127–158, 1999.
- [2] H. Hyvrö. Improving the bit-parallel NFA of Baeza-Yates and Navarro for approximate string matching. *Inf. Process. Lett.*, 108(5):313–319, 2008.
- [3] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.