Vol.2009-AL-124 No.3
2009/5/11

Implementation of a Bit-parallel Approximate String Matching Algorithm

Mikael Onsjö [†1] and Osamu Watanabe[†1]

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 Hyyrö[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 Tesla processor (GPU) respectively. The speedup in this case is about 50 times, meaning 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 Tesla 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 of consecutive characters in a body of *text* (of length n), that are within a given *edit distance* (k) of a given pattern (of length m). The edit distance between two strings is defined as the minimum number of edits (character insertion, deletion or replacement) needed in order to transform one string into the other.

This problem is of high importance in many areas, e.g. for search engines when looking for pages relevant to a query or in bioinformatics when analyzing huge sets of DNA sequencing data. Although our work also applies to the former situation (e.g. searching in bodies of English text) we focus on the latter, that is, on searching for long patterns in sequences of the characters A,C,G and T. As will become apparent, the hardware is well suited for this case.

The importance of the problem has inspired significant research and many algorithms have been proposed. Arguably the most "practical" approach has been a non-deterministic 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, in terms of pattern length, not nearly enough as one 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 common computer. Even so, it seems that in many practically relevant situations (e.g. $k << m$) this approach may arguably be the fastest one known.

We show how to implement the same algorithm in CUDA for a Tesla 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 states at a time) in an elegant 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

The original NFA[1),2)] 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

†1 Dep. of Mathematical and Computing Science, Tokyo Institute of Technology

© 2009 Information Processing Society of Japan

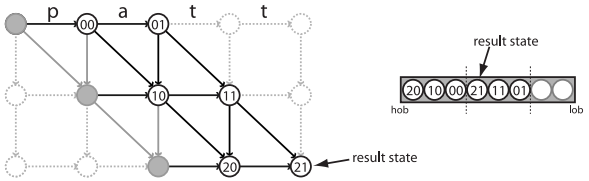

Fig. 1 Efficient encoding of small NFA into 8-bit computer word.

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 that 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 as it is laid out in the following figure, 1, that illustrates the encoding.

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 is at least one match (due to the **delete** rule). 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*).

In practice it is, somewhat counter-intuitively, common to encode active states as zero bits and inactive states as ones. This is because generally machines and programming languages have better support for *shift operations* that introduce zeros at the abandoned side of the word. As an example of how the operations for updating the NFA are implemented, consider the **replace** rule as stated above, and the encoding defined by figure 1. States below the top row obtain their correct assignment by a simple left shift (note that, also perhaps a bit counter-intuitively, each column is encoded "backwards"). Since states on the top row cannot be activated by the **replace** rule, it is prudent to mask the result appropriately (putting top row bits to one explicitly), though this part can be somewhat simplified when all rules are considered togehter. Masking in this case (and this example) entitles taking a bitwise OR with the constant (00100100). In C-style pseud-code, the rule might be written as:

$$\mathrm{NFA}_{\mathbf{rep}} \leftarrow (\mathrm{NFA} << 1) \mid (0^k 1)^{m-k}$$

where the exponential denotes repetition of a bit pattern. The results from different rules can be combined by using a bitwise AND:

$$\mathrm{NFA} \leftarrow \mathrm{NFA}_{\mathbf{mat}} \,\&\, \mathrm{NFA}_{\mathbf{rep}} \,\&\, \mathrm{NFA}_{\mathbf{ins}} \,\&\, \mathrm{NFA}_{\mathbf{del}}$$

though we stress again that this particular organization would introduce a few more operations than what is strictly necessary.

For the **match** rule (and the match rule only), it is obviously necessary to consider the characters of the pattern. It is common to initially (before scanning the text) calculate a constant mask of the same size as the NFA, for each unique character that appears in the text. This mask, call it M[x] for character x, should have a zero in each place for which the corresponding state of the NFA has a *matching transition from the left*. That is, if x happens to appear at position j (counting from 0) of the pattern then for each i should $\mathrm{M[x]}_{ij}$ (if indexed as s above) be zero. Hence if the character x is encountered in the text, the **match** part in the NFA update can be calculated by a right shift of $k+1$ and a bitwise OR with M[x]:

$$\mathrm{NFA}_{\mathbf{mat}} \leftarrow (\mathrm{NFA} >> (k+1)) \mid \mathrm{M[x]}$$

The **insert** and **delete** rules are somewhat more complicated and we refer again to paper[2] for what appears to be the smallest updating implementation in terms of both space and the number of operations required. Paper[1] contains extensive suggestions for how to encode and update NFA's that don't fit into a single machine word.

Finally, as an example figure 2 shows what happens as the pattern "tit" is

© 2009 Information Processing Society of Japan

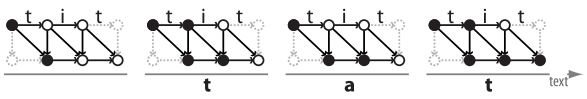

Fig. 2 Example of matching.

matched against the text "tat" while allowing for a maximum edit distance of one (meaning the number of rows in the NFA is two).

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 Tesla 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, see Nvidia's programming manual[3] for details. What follows is a summary of some relevant key points:

Code on the Tesla is executed in several blocks of up to 512 threads each. 16 KB of *shared* memory is attached to each multiprocessor 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** so between blocks. The access for each warp should be either to the same memory position (in which case the read result is broadcasted) or to positions served by different memory banks. 16 banks are organized in an interlaced fashion so that e.g. accessing sequences of memory positions is an efficient approach.

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 4 GB of *global* memory that is relatively slow to access and 64 KB of *constant* memory with a 16 KB cache, that is suitable for such constant as are accessed simultaneously by all threads in a warp.

Though each thread (multiprocessor) operates with individual 32-bit words, on a higher level it is possible to think about the operations of a block as being on

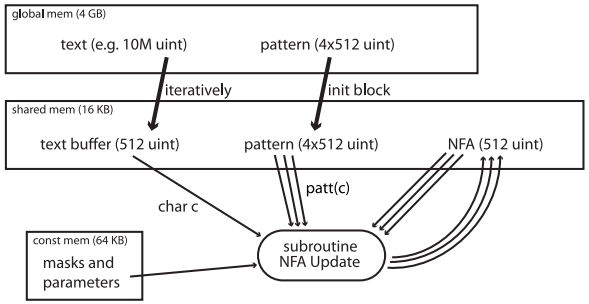

Fig. 3 Usage of device memory.

single huge words, e.g. of size $512 \cdot 32 = 16384$. 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 many in order to take advantage of all multiprocessors and to allow the scheduling its full potential. Experimentally we find that with little dependence on other parameters, about 300 blocks 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 $4 \cdot 512$ 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 in figure 4.

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 assembler-like .ptx file that can be generated with the compiler, reveals that this part is implemented with approximately 30 4-cycle instructions. It is difficult to imagine that this could be much reduced unless a completely different approach was considered. The routine looks roughly as in figure 5 where C_1, C_2 and C_3 are constants that depend on how the NFA is encoded.

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

© 2009 Information Processing Society of Japan

(CPU) Get data, parameters and pattern to RAM and GPU global memory.
(CPU) Compile the pattern and put it in global memory.
(GPU) Execute Kernel per block:
  Initialize masks and constants.
  Load compiled pattern *global* → *shared*.
  while more text **do**
    Load text to buffer *global* → *shared*.
    for each character in text buffer **do**
      Subroutine: NFA Update
      Check for match...
    end for
  end while
(CPU) Get result from GPU.

Fig. 4 Outline of implementation

$cw \leftarrow \mathrm{NFA}[i]$
$cwleft \leftarrow \mathrm{NFA}[i-1]$
$cwright \leftarrow \mathrm{NFA}[i+1]$
$cm \leftarrow M[nexttextchar][i]$
$x_1 \leftarrow (cw >> (k+1)) \mid (cwleft << C_1) \mid cm$
$x_2 \leftarrow ((0^k1)^{m-k} \mid (cw << 1 \,\&\, ((x + (0^k1)^{m-k})^x))$
$x_3 \leftarrow (cw << (k+2)) \mid (cwright >> C_2)$
$\mathrm{NFA}[i] \leftarrow C_3 \,\&\, x_1 \,\&\, x_2 \,\&\, x_3$

Fig. 5 Outline of subroutine NFA-Update for thread i.

needs to be moved from global to shared memory. That is to say, 16 characters of the text are stored in one 32 bit word. 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. 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 6. 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.

The notable irregularities in both curves in figure 6 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).

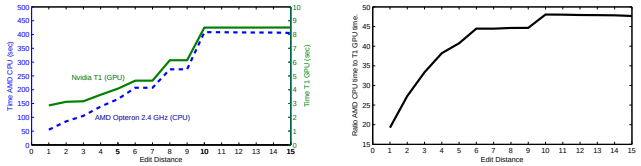

Fig. 6 Running times of NFA algorithm with sequential and parallel implementations. The times are plotted (with different scales) in the left figure and the ratio between the two time sequences in the right.

References

1) Baeza-Yates, R. and Navarro, G.: Faster approximate string matching, *Algorithmica*, Vol.23, pp.127–158 (1999).
2) Hyyrö, H.: Improving the bit-parallel NFA of Baeza-Yates and Navarro for approximate string matching, *Inf. Process. Lett.*, Vol.108, No.5, pp.313–319 (2008).
3) Nvidia: *NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 2.0*, http://developer.download.nvidia.com/.

© 2009 Information Processing Society of Japan