Asian Journal of Science and Applied Technology (AJSAT)
Vol.2.No.1 2014pp 43-48
available at: www.goniv.com
Paper Received :05-03-2014
Paper Published:28-03-2014
Paper Reviewed by: 1. John Arhter 2. Hendry Goyal
Editor : Prof. P.Muthukumar

INCREASING EFFICIENCY FOR HIT DETECTION IN BLASTN

[1]Gomathi.V , [2]A.D. Khamala khannen
Electronics and Communication Engineering,
Smk Fomra Institute Of Technology,
Chennai.
gomathi.rajen@gmail.com

ABSTRACT

For biologists very hard time is with analyzing the uniqueness between two sample sequences such as DNA, RNA and protein sequences. A Bio-sequence represents a single, continuous molecule of nucleic acid or protein. It can be anything from a band on a gel to a complete chromosome. That's to design for a huge database system which finds similarities between two sequences that have biological significance. In such condition we have to compromise in computation time, this can be overcome through implementation BLASTN process. In this paper the BLAST process will be working more efficient by a new approach for biological sequence database scanning. The scanning is performed with reconfigurable FPGA base hardware by comparing sequence one to many sequences from the database. The experimental sequence matching reduces the computation time of BLAST. [1] [2]

Keywords: biological sequence, significance comparing, word-matching, computation time.

1. INTRODUCTION

Database searching is quite big process in many of the application. Searching in a database for a real time system is real tough work to be performed. That to scanning genomic sequence database is a common and often repeated task in molecular biology. The searching should be performed faster because new sequence will be updated very quickly, so the data will be increasing in the database in exponential manner. At this case we are in need of a algorithm which performs very efficiently with the database for accessing the data i.e. biological sequence. One of the most widely and efficient search algorithm tool is BLASTN (Basic Local Alignment Search Tool-Nucleotide). Process of BLAST: Using a heuristic method, BLAST finds similar sequences, not by comparing either sequence in its entirety, but rather by locating short matches between the two sequences as shown in fig. 1.

This process of finding initial words is called seeding. It is after this first match that BLAST begins to make local alignments. While attempting to find similarity in sequences, sets of common letters, known as words, are very important. For example, suppose that the sequence contains the following stretch of letters, AGCTGC. If a BLASTN was being conducted under default conditions, the word size would be 3 letters. In this case, using the given stretch of letters, the searched words would be AGC, GCT, CTG, and TGC. The heuristic algorithm of BLAST locates all common three-letter words between the sequence of interest and the hit sequence, or sequences, from the database. These results will then be used to build an alignment. After making words for the sequence of interest, neighborhood words are also assembled. These words must satisfy a requirement of having a score of at least the threshold *T*, when compared by using a scoring matrix. One commonly-used scoring matrix for BLASTN searches is BLOSUM62, although the optimal scoring matrix depends on sequence similarity. Once both words and neighborhood words are assembled and compiled, they are compared to the sequences in the database in order to find matches. The threshold score *T* determines, whether a particular word will be included in the alignment or not. Once seeding has been conducted, the alignment, which is only 3 residues long, is extended in both directions by the algorithm used by BLAST. Each extension impacts the score of the alignment by either increasing or decreasing it. Should this score be higher than a pre-determined *T*, the alignment will be included in the results given by BLAST. However, should this score be lower than this pre-determined *T*, the alignment will cease to extend, preventing areas of poor alignment to be included in the BLAST results. Note, that increasing the *T* score limits the amount of space available to search, decreasing the number of neighborhood words, while at the same time speeding up the process of BLAST.

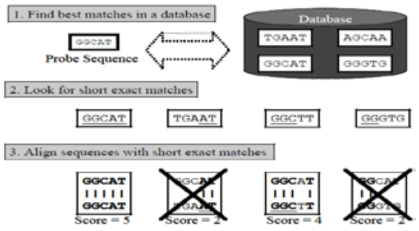


Fig.1.Diagramatic representation of BLASTN

2. ALGORITHM

The paper deals with nucleotide molecular mostly, so more specific BLAST algorithm is required. BLASTN is an algorithm which deals only with nucleotide. This work can also be implemented in the DRC coprocessor system [1] [2]

A. Word matching accelerator architecture

Mainly BLASTN is used to find word matches. A word match is performed by fixing or assuming a string which has fixed length (w) w-referred to as "w-mer, that occurs both in query sequence and database sequence. Word matching flow diagram is as follows:

Stage1: Implementation of parallel bloom filter.

Stage 2: False positive eliminator.

Stage3: Redundancy eliminator.

1) Stage 1: Implementation of parallel bloom filter

The word-matching stage aims to find good alignments containing short exact matches between a query sequence. Such matches such as hash tables or suffix trees. An alternative solution to this filtration problem is to use a Bloom filter. A Bloom filter is defined by a bit-vector of length m, denoted as BF[1,…,m]. A family of k hash functions $h_{i:}$ S→A, 1< i < k, is associated to the Bloom filter, where S is the key space and A={1,…,m} is the address space. A Bloom filter is a simple space-efficient randomized hashing data structure suitable for quick membership tests on FPGA implementation. A Bloom filter works in two steps. **[1] [2]**

1).Programming: For a given set I of keys, I={$x_{1,\ldots,}$$_{Xn}$}, I c S, the Bloom filter's programming process is described as follows. First of all, initialize the bit vector m with zeros, then, for each key x_j E I, compute its k hash values $h_i(x_j)$, I< i < k, subsequently, set the bit vector to one according to the k hash values.

2).Querying: the querying process of the Bloom filter works the same as its programming process. For given key x, compute k hash values $h_i(x)$, I< I < k, is zero, then x E I, otherwise, x is said to be a member of set I with a certain probability. The conventional design for the identification of w-mers using a bloom filter is shown in Fig. 2.

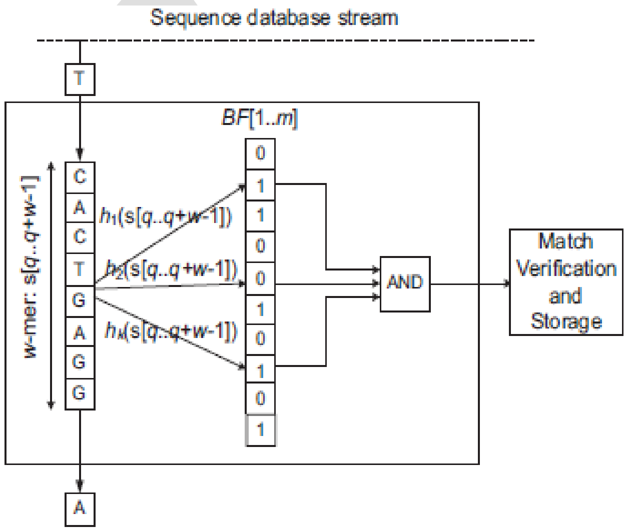


Fig.2.Conventional design for identifying w-mers in a sequence database stream using bloom filter

The bloom filter has been programmed by parsing the query sequence into overlapping substrings of length w in the preprocessing step.Here is an example for the query sub strings. Assume w=3 and the query sequence is "cttgtata" then ,the parsed sub strings are {" ctt","ttg","tgt","gta", "tat","ata"}.Although the conventional bloom filter architecture is efficient for membership test, its direct implementation is not suitable for high performance design on an FPGA.

The computation efficiency will be compromised, if a single key was sent to all hash functions for membership testing, especially under low match rate conditions. Thus, our idea is to divide the *k* hash functions into different groups, with each group used for a different hash query. We apply three techniques to improve the throughput compared to the conventional Bloom filter architecture as shown in Fig 3.1 ,3.2.

1) *Partitioning:* We first partition the Bloom filter vector into a number of smaller vectors, which are then queried by independent hash functions.

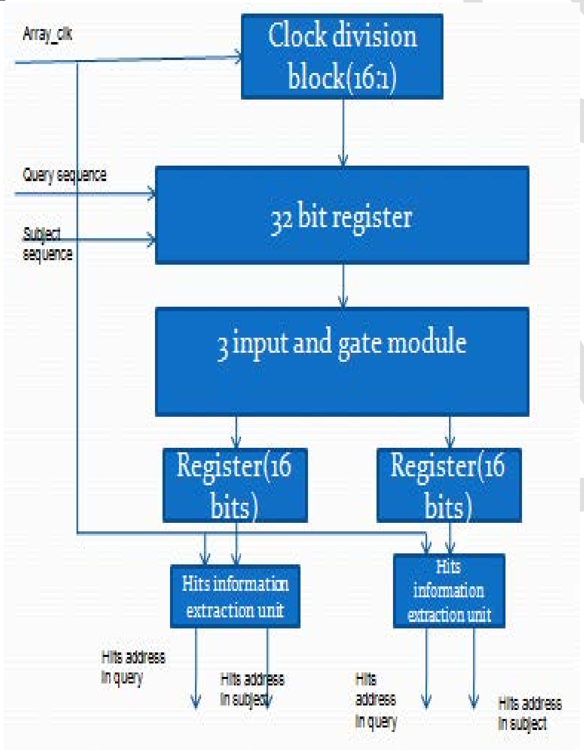
2) *Pipelining:* We further increase the throughput of our design using a new pipelining technique.

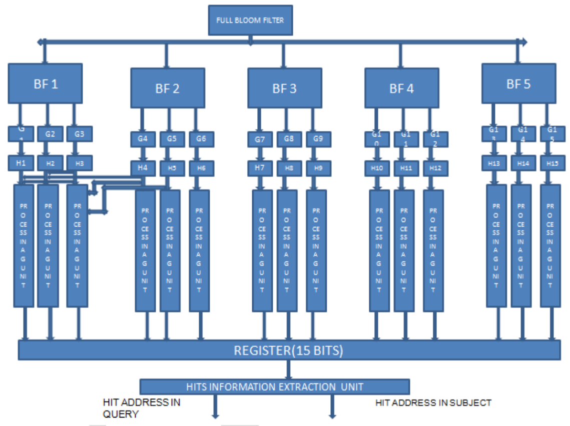
3) *Local stalling:* We use a local stalling mechanism to guarantee all w-mers are tested by the Bloom filter.

In each clock cycle, it can support k/P different hash queries. The hash functions used in the PPBF block are chosen from which it can be efficiently implemented in hardware. Suppose the input bit string X with b bits is represented as $X = < x_1, x_2, \ldots, x_b >$. We calculate the
i-th hash function over X, $h_i(X)$ as

$$h_i(X) = (d_{i1} \cdot x_1) \oplus (d_{i2} \cdot x_2) \oplus \cdots \oplus (d_{ib} \cdot x_b)$$

where "·" is a bitwise AND operator and $\oplus$ is a bitwise XOR operator, d_i are predetermined random numbers in the range $[0, \ldots, m-1]$. Both the AND and XOR operations can be implemented in parallel to shorte





n computation.

Fig.3.1, 3.2 Architecture of Multiple Hits Detection Module.

Parallel Architecture Design of BLAST Algorithm with Multiple Hits Detection.

2) Multiple Hits Detection: Module-Multiple Hits Detection Module is used to detect 3-word hits and record the hits address in the query and the subject sequence. Compared to WPRBS method which could detect at most one hit in only one clock cycle, this design can detect multiple hits in only one clock cycle. The architecture of Multiple Hits Detection Module is shown in figure 3.1,3.2 As the figure illustrated, there is a systolic array with 32 processing units, every 3 units are connected to one 3-input AND gates. Every 16 gates outputs are connected to a 16 bit register. The value of each register is sent to the corresponding hits information extraction units for recording the hits address in the query and the subject sequence. The systolic array and the hits information extraction unit are driven by two different clocks. At each clock out rising edge, query or subject sequence moves forward for one processing unit.

The whole architecture works as follows: first, a query sequence with 32 characters is forwarded into the systolic array so that each processing unit holds a character from the query sequence. Then the subject is driven into the systolic array by each internal clock rising edge. Mean while ,the incoming subject character and the query character which are held by the unit are compared, if they are identity, the logic "1" would be generated; otherwise, the logic "0" would be generated. The comparison result is an input of a 3-input AND gate. A hit is detected when logic "1" is generated from its output. So, the systolic array with 3-input AND gates can detect multiple hits at one internal clock rising edge. Architecture of the processing unit in the systolic array is illustrated in Figure.5. As shown in figure 3.1, 3.2, outputs of 32 3-input AND gate goes into 2 16-bit registers. The Hits information extraction unit detects hits location and records them. The multiple hits detection module is a parallel, pipelined architecture. The systolic array with 32 processing units cooperates with 32 3-input AND gates to detect hits in both sequences. Hits information extraction block records those hits location.

3) Hits Combination Block: If there is a high similarity between query and subject sequence, the multiple hits detection module may output a large amount of hits per clock cycle. For instance, two adjacent hits "ATK" and "TKP" are found but they are actually one hit" ATKP". If they are not combined into one hit, they would have been recorded twice .Hence this block can detect overlapping hits and merge them to reduce verbose hits and maintain the sensitivity of BLAST.

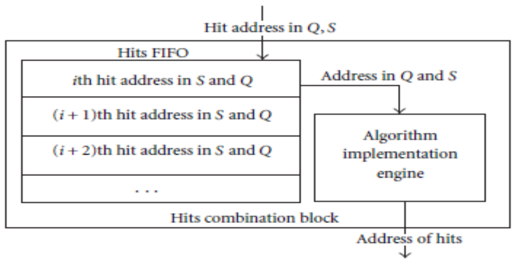

Hit address in Q, S
Hits FIFO
ith hit address in S and Q
(i + 1)th hit address in S and Q
(i + 2)th hit address in S and Q
. . .
Address in Q and S
Algorithm implementation engine
Hits combination block
Address of hits

Fig.4.Hits Combination Block

This block contains a Hits (First In First Out)FIFO buffer which is used to store hits location address from both query and subject sequence. The data flow in this block is shown in Fig. 4.

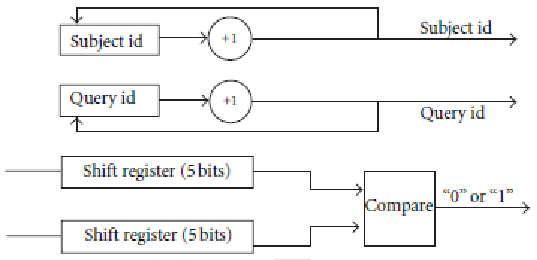


Fig.5.Architecture of processing unit of multiple hits detection module

4) ***Stage 2***: *False positive elimination.*

The objective of this stage is to find all false positive matches generated by bloom filter and get the corresponding position information in the query sequence for the true positive words. The second sub stage of our word-matching accelerator design is false-positive elimination, which includes two objectives:

1) Find all false-positive matches generated by the Bloom filter;
2) Get the corresponding position information in the query sequence for true-positive *w*-mers.

One solution for this sub stage is to use a hash lookup table. The position information of each *w*-mer from the query sequence is stored in the hash table. A hash table with
1 million entries storing position information for a 100-kbase query sequence requires at least 17 Mbits of memory space (17 bits are needed to represent 100 k positions). It is clear that the memory required is significantly greater than that provided by the on-chip BRAMs. Thus, we store the hash table in an external SDRAM attached to the FPGA.
Hash collisions and duplicate keys are two common prob- lems for simple hashing strategies. The former will hash two different queries to the same location, while the latter may miss additional position information. Both of them require extra access to the off-chip DRAM to get the correct data, which could introduce potential performance bottlenecks. In previously reported designs, **[1] [2]**

a perfect hash function has been applied to construct the hash table. A perfect hash function for a set of *n* keys maps each key to a distinct table

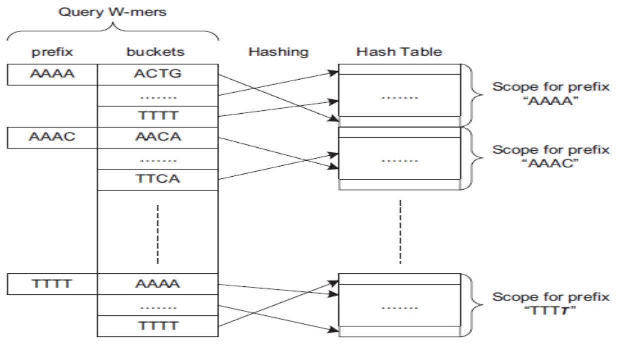


Fig. 6.Hash Table

entry with no collisions among the keys in the set is shown in figure 6. However, a perfect function is not easy to generate, especially when *n* is large. In addition, the representation of the perfect hash function usually needs a significant amount of FPGA resource and may compete with the Bloom filter design. The Mercury BLASTN design implements the hash table using a near- perfect hashing strategy, which bypasses the constraint for a perfect hash function. However, considerable effort is still required to get the “near-perfect” hash functions. Cuckoo hashing is another effective hash strategy used to avoid hash collision, where two independent hash functions are used for a single hash query. However, the additional hash table access may reduce the overall performance and, in rare cases, hash collision can still appear. In our design, we try a less complicated approach with few hash collisions, called a bucket hash. Our idea works as follows. Although it is difficult to find a perfect hash for all *n* keys, it might be easier to find a perfect hash function for a subset of keys, if the size of the subset is small enough. Bucket hashing works as follows. **[1] [2]**

1) Sort the query *w*-mers into different buckets according to their prefix (if the prefix length is properly chosen, the number of *w*-mers in a given bucket is relatively small).

2) Find a simple hash function that is collision-free for all *w*-mers in the same bucket. If it is not possible to find such a perfect hash function, uses the hash function with the minimum hash collisions.

3) Construct a quick lookup table (QLT) which stores the “collision-free” hash functions for each bucket.

*.5) **Stage 3**: Redundancy eliminator*: To avoid repeated generation of the same sequence alignment, we go for this stage. By doing such procedure we can able to reduce the words. We only eliminate “true overlapping” words to the registers unit. If two sequences suggest same alignment matches then one of those matches can be taken the concept of overlapping.

3. PERFORMANCE

Once the word matching stage gets completed then by using verilog language and DRC co processor **[1] [2]** (i.e. a process supports to the main processor of the system) the implementation has been performed. The main processor will be Xilinx Virtex-5 FPGA chip. The DRC has been considered has a co-processor because it stores large volume of off-chip data using the DRC system’s memory which consists of up to 8 GB of DDR2 SDRAM with a maximum bandwidth 3.2 GB/s and 512 MB of low latency RAM with a maximum bandwidth of 1.4 GB/s. In each clock cycle, the parallel Bloom filter can receive up to 16 new *w*- mers to do the membership examination from local buffers.

4. ANALYSIS

Our design reports more 15-mer hits compared to the NCBI BLASTN. In our word-matching accelerator, the Bloom filter only introduces false –positive results with no false negatives, which guarantees no sensitivity loss for the true matches. The off-chip hash table structure, which covers all situations (duplicate hits and collision hits can be examined by looking up the secondary table and the duplicate table, correspondingly), eliminates all the false positives from the Bloom filter stage. Thus, our FPGA design can report all 15-mer hits between the query sequence and the database sequence. In fact, the NCBI software is designed to report fewer hits. It applies several optimizations to accelerate the word-matching search process. For example, for a 100 Kbase query, the software first scans the database sequence for 10-mer matches with step length two, then ,extends one extra base in both sides to get a 15-mer match with the search terminating once a match is found, if a long k-mer (k>15) match exists,15-mer hit losses will appear. The lost hits are expected to be re-examined by the ungapped extension stage which extends hits base by base in both directions. However, sensitivity loss might appear due to optimizations applied in the word-matching stage of NCBI BLASTN software program. For example, for the 100 kbase query sequence, the hits from our FPGA design can generate 1270 HSPS, while the hits from NCBI stage can generate 1212 HSPS.

5. RESULT

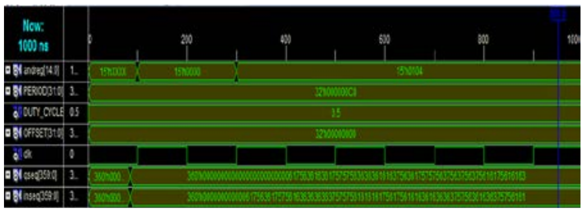


Fig.8.simulation result

The input sequence is a 15 genetic code represented in hexadecimal format. 1 genetic code is represented by 24 bits .The query sequence is yet another 15 genetic code represented in hexadecimal format from the database .Both the sequences are compared, if hit is found between any two sequences output will be 1 else in a case of mismatch output will be 0.Hence output is a 15 bit code which represents the hit between the sequences. The time consumption for comparing 15 genetic code is 300 ns. Refer Fig.8.

6. CONCLUSION

In this paper, I have dealt about increasing the computation time for finding hit detection in BLASTN using three sub stages, a parallel bloom filter, an off-chip hash table, and a match redundancy eliminator. Finding hits is the most computationally time-consuming step in BLASTN. The performance is faster than other approach used in other architectures. The comparison of my technique with that of NCBI BLASTN shows a better performance with limited resource utilization

REFERENCES

[1] .Reconfigurable accelerator for the word-matching stage of blastn-yupeng chen,bertil schmidt,senior member,ieee, and douglas l.maskell,senior member,ieee-1063-8210/31.00-2012 ieee.

[2]. A systolic array-based fpga parallel architecture for the blast algorithm-xinyu guo,hong wang,and vijay devabhaktuni-international scholarly research network-isrn bioinformatics-volume 2012,article id 195658,11 pages doi:10.5402/2012/195658.

[3]genbank statistics at ncbi [online]. Available: http://www.ncbi.
Nlm.nih.gov/genbank/genbankstats.html

[4] s. F. Altschul, w. Gish, w. Miller, e. W. Myers, and d. J. Lipman, “basic local alignment search tool,” j. Molecular biol., vol. 215, pp. 403–410, feb. 1990.

[5] blast algorithm [online]. Available: http://en.wikipedia.org/wiki/ blast

[6] p. Karishnamurthy, j. Buhler, r. Chamberlain, m. Franklin, k. Gyang, a. Jacob, and j. Lancaster, “biosequence similarity search on the mercury system,” j. Vlsi signal process. Syst., vol. 49, no. 1, pp. 101– 121, 2007.

[7] z. Zhang, s. Schwartz, l. Wanger, and w. Miller, “a greedy algorithm for aligning dna sequences,” j. Comput. Biol., vol. 7, nos. 1–2, pp. 203–214, 2000.

[8] w. J. Kent, “blat–the blast-like alignment tool,” genome res., vol. 12, pp. 656–664, mar. 2002.
.