CAAD BLASTn: Accelerated NCBI BLASTn with FPGA Prefiltering

Jin H. Park Yunfei Qiu Martin C. Herbordt
Department of Electrical and Computer Engineering
Boston University,
Boston, MA 02215, U. S. A.

ABSTRACT
The canonical bioinformatics application is determining the biological similarity of a new sequence (protein or DNA) with respect to databases of known sequences. The BLAST algorithm is used for the vast majority of these searches. Of the various BLAST implementations, the one published by NCBI is a recognized standard. In previous work we described FPGA acceleration of the protein version of NCBI BLAST (BLASTp) using our TreeBLAST-based filter. Here we apply this filter to NCBI BLASTn, the DNA version. We show the modifications to the structures of the filtering components needed to handle DNA, as opposed to protein, sequences. The design has been implemented on an Altera Stratix III family chip. Our experimental results show that the speed-up is greater than 12x and the accuracy is 100%.

1. Introduction

BLAST [1] is the most popular heuristic biological sequence alignment searching tool. Two commonly used versions are BLASTp and BLASTn; these manipulate protein-protein and nucleotide-nucleotide sequence alignments, respectively. In recent few years, there have appeared a number of articles on the FPGA acceleration of these two versions [2-6,8-10,12,13]. Due to the different searching criteria, however, handling both versions with a single FPGA structure is problematic. Previously we have shown how our TreeBLAST filter can be applied to BLASTp [4,10]; here we focus on BLASTn.

Operations in BLAST are commonly grouped into three consecutive steps: seeding (word matching), ungapped extension, and gapped extension. FPGA accelerations of BLAST have often accelerated the most computation-intensive parts, particularly the seeding step, with or without restructuring the original algorithm [2,3,8,9,12,13]. More detailed surveys can be found in many articles including [3,12].

In this paper, we describe the design and implementation of CAAD BLASTn and its FPGA components. CAAD BLASTn is a high performance version of NCBI BLASTn with FPGA-based prefiltering. We use Tree-BLASTn and Smith-WatermanN in the filter to reduce the database size. The advantages of prefiltering are described in-depth elsewhere [10]; briefly, this approach enables two conflicting goals to be achieved simultaneously: (i) high performance and (ii) exact agreement of results with those produced by the original NCBI BLAST code.

2. CAAD BLASTn

The global structure of CAAD BLASTn is shown in Figure 1. In fact, the structure is generic for both BLASTn and BLASTp versions, but the detailed implementation of software and hardware components are not identical. The underlying idea of the prefiltering scheme is reducing the database size without altering the sensitivity of the target BLAST system. We now briefly describe the role of each component in CAAD BLASTn.

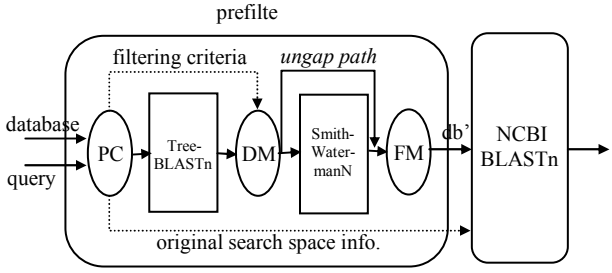


Figure 1. Structure of CAAD BLASTn

The prefilter shown in Figure 1 is transparent to users and consists of five components including two FPGA modules and 3 software modules. When the end-user initiates NCBI BLASTn, the prefilter is dynamically and transparently invoked. It reduces the given database. The reduced database is passed to the original NCBI BLASTn with the original search space information so as to enable the correct calculation of the Evalues for the final report. As a result, the original NCBI BLASTn works with the

reduced database but has the illusion of the original search space information instead of using the reduced space information. The precomputation module (PC), which is implemented in software, is responsible for extracting all needed criteria (heuristics) used in the filter to prevent loss of sensitivity. The Tree-BLASTn module, which is implemented in the FPGA, is the main ungapped filter: it performs operations analogous to those done in the seeding and ungapped-extension phases of NCBI BLASTn. The TreeBLAST logic and its proof are given in [4]. The decision making module (DM), which is implemented in software, is attached to the ungapped filter and provides criteria of reducing the database for either the ungapped or gapped alignment option. In the case gapped alignment, the SmithWatermanN module, which is also implemented in the FPGA, is activated and further reduces the database. In either ungapped or gapped alignment, the reduced database is formatted (in the formatting module FM) since the reduced database is dependent on the query used. The FM module (software) takes a negligible amount of time.

3. Implementation

The acceleration is done through the FPGA components of the filter. In this section we describe the design and implementation details of the two major components, which are TreeBLASTn and SmithWatermanN.

Double-stranded alignment

Unlike BLASTp, BLASTn performs double-stranded search. This is because DNA is double-stranded and genes may occur on either strand. This makes BLASTn alignments more difficult to interpret than BLASTp [2]. NCBI BLASTn reports alignments labeled as either plus/plus or plus/minus. The former represents that the alignment is found from examining the query in the FASTA format against database, while the later uses the reverse-complement of the query. Figure 2 shows sample plus/plus and plus/minus labeled alignments reported by NCBI BLASTn. In Figure 2(b), the subject sequence is flipped for the purpose of display, but in fact, the query is reversed and complemented. The complement rules on the alphabet

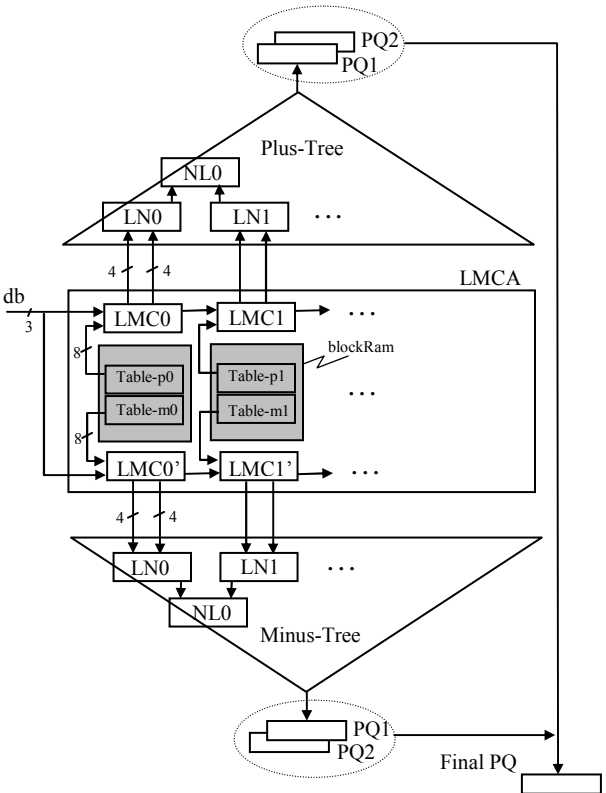
```
(a) Score = 36.2 bits (18), Expect = 0.28
    Identities = 18/18 (100%)
    Strand = Plus / Plus

    Query: 175 acagcacgccgttgatga 192
               ||||||||||||||||||
    Sbjct: 594 acagcacgccgttgatga 611

(b) Score = 34.2 bits (17), Expect = 1.1
    Identities = 17/17 (100%)
    Strand = Plus / Minus

    Query: 154 cggccggcgcggacggg 170
               |||||||||||||||||
    Sbjct: 330 cggccggcgcggacggg 314
```

Figure 2. Double-stranded alignments in NCBI BLASTn

{A,C,G,T} are A↔T and C↔G. For example with the query segment shown in Figure 2(b), its reverse complement form, which is compared against database, is "CCCGTCCGCGCCGGCCG". The double-stranded alignment mechanism is implemented in both TreeBLASTn and SmithWatermanN modules.

Tree-BLASTn

To manipulate the double-stranded search operation in the ungapped filter, we replicate the TreeBLAST logic. We use one copy (Plus-Tree) for the plus-strand search and the other (Minus-Tree) for the minus-strand search. This method is also used in the implementation of SmithWatermanN. The Plus-Tree works with the original FASTA format query and the Minus-Tree works with reverse complement of the query. The global view of our design is shown in Figure 3. The plus and minus trees share one LMCA (letter match cell array) module in which block RAM (BRAM) space is used to store score lookup tables. The LMC (letter match cell) module is used for extracting matching scores for a pair of incoming database residues based on a hash function. When a query is processed on the host side, the lookup table is loaded; it is accessed when database sequences are processed. The definitions of LN (leaf node) and NL (non-leaf node) modules in each tree can be found in [4].



LMCA: letter match cell array, LMC: letter match cell, LN: leaf node, NL: non-leaf node, PQ: priority queue

Figure 3. Global view of TreeBLASTn

Loading and accessing the score lookup table

The full alphabet used in NCBI BLASTn is {A,C,G,T,B,D, H,K,M,N,R,S,V,W,Y} (only a part, {A,C,G,T}, is considered in many studies). We add '$' to serve as a delimiter. Figure 4 shows the 1/-3 scoring scheme used in NCBI BLASTn, including letter '$'.

	A	C	G	T	B	D	H	..	N	..	$
A	1	-3	-3	-3	-3	-2	-2	..	-2	..	0
C	-3	1	-3	-3	-2	-3	-2		-2		0
G	-3	-3	1	-3	-2	-2	-3		-2		0
T	-3	-3	-3	1	-2	-2	-2		-2		0
B	-3	-2	-2	-2	-2	-2	-2		-2		0
...											
$	0	0	0	0	0	0	0	..	0	..	0

Figure 4. Scoring scheme for alphabet {A,C,G,T, .. ,$}

Instead of building one generic table per each query character, we build a table for a pair of query characters. This saves the number of BRAMs required. As shown in Figure 3, we can access two scores simultaneously by accessing a single address. Since we can use two read ports, we store two counterpart tables, i.e., the plus tree and minus tree tables, together in a BRAM. Contents of the counter-part tables are not identical since the minus tree uses the reverse complement of the query. This scheme requires preparing all possible combinations of scores for each pair of query characters. We use the following hashing mechanism to load and access the tables.

Assume that a pair of query characters and a pair of database characters are stored in q_pair(0,1) and db_pair(0,1), respectively, and represented by order number in the alphabet.

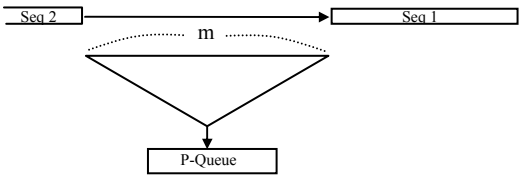
```
when query is processed:
   for i in 0 to alphabet_size -1 loop
      for j 0 to alphabet_size-1 loop
         entry <= scoring_matrix(q_pair(1))(j) &
                  scoring_matrix(q_pair(0))(i);
         if Plus_Tree
            addr_write <= i + j*alphabet_size;
         else (Minus_Tree)
            addr_write <= i + j*alphabet_size + table_size;
      end loop;
   end loop;

when database is processed:
   if Plus_Tree
      addr_read <= db_pair(0) + db_pair(1)*alphabet_size;
   else (Minus-Tree)
      addr_read <= db_pair(0) + db_pair(1)*alphabet_size +
                   table_size;
```

Each lookup table is 16*16 (since our alphabet size is 16). Also, each entry contains two scores (4 bits each) to manipulate the matching of a pair of query characters and a pair of database characters. For example, for a query pair (G,A) and a database pair (C,G), the lookup table contents starts with the matching scores of A-A and G-A at address 0, and ends with the matching scores of A-$ and G-$ at address 255. When the database pair is processed, the LMC module computes the read-address as 33 – the location in which the matching scores of A-G and G-C are stored. In the case of manipulating the minus strand, the address computation for writing and reading requires adding the table_size, i.e., 16*16=256. In the case of manipulating an odd-size query, we simply add a dummy character ($) at the tail. The above hashing mechanism is applicable to an arbitrary size alphabet.

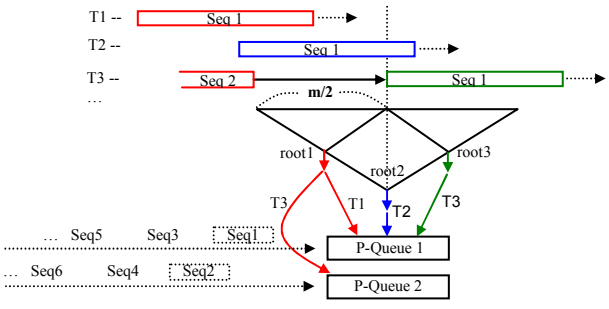
In Figure 3, the LMC0 module accesses two scores (4 bits each) from Table-p0 for leaf node LN0 in the plus tree, while LMC0' accesses another two scores from Table-m0 for leaf node LN0' in the minus tree. Analogous operations are done in all other LMC modules. After the trees are fully utilized, all LMC modules access scores from associated tables in parallel. The final output is collected from outputs of both trees using a priority queue. In fact, each tree yields outputs, which are certain numbers of best scores, using double priority queues. The reason of using the double priority queues is to reduce the processing delay between database sequences. We describe this in more detail below.

Reducing streaming delay between database sequences

Due to the hierarchical structure of the TreeBLAST logic, the processing delay between two database sequences is m (size of query), as shown in Figure 5(a), unless we ignore the tail processing of each sequence by losing the sensitivity of output. To reduce the delay we use a technique that employs three roots and two priority queues. This scheme reduces the delay by half, i.e., m/2 as shown in Figure 5(b), without losing the sensitivity of output. At time T1, when Seq1 enters the tree, only the left sub-tree is utilized and the output is available at root1 and stored in P-Queue1. At time T2, when Seq1 passes



(a) delay = m



(b) delay = m/2

Figure 5. Reducing delay between database sequences

the center line of the tree, the full tree is utilized and the output is available at root2 and stored in P-Queue1. However, the root transition occurs after log(m/2) clocks. At time T3, when Seq1's tail passes the center of the tree, only the right sub-tree is utilized for Seq1 and the output is available at root3 and stored in P-Queue1. This root transition occurs after log(m) clocks. At the same time of T3, the next sequence (Seq2) enters the tree and the left sub-tree is utilized for Seq2 and the output is available at root1 and stored in P-Queue2 while root3 yields the output of Seq1. This makes the processing delay m/2. As shown in Figure 5(b), P-Queue1 serves for Seq1, Seq3, Seq5, .., while P-Queue2 serves for Se2, Seq4, Seq6, …. The priority queue reading operations are overlapped with the store operations without adding any delay.

SmithWatermanN

For the implementation of the gapped filter, we use the analogous method used in the ungapped filter. We replicate the Smith-Waterman systolic array for the plus and minus stranded search operations. PEs in each array access the BRAMs for the score lookup tables in a similar manner as shown in Figure 3.

4. Performance

TreeBLASTn and SmithWatermanN logics are simulated and synthesized using ModelSim and QuartusII, respectively, on a Stratix III EP3SL340 chip. Table 1 shows the statistics. The maximum tree size and array size can be extended by using the folding mechanism we already developed and implemented in our previous works [4,10]. However, the folding mechanism brings about the extended processing time of database residues.

Table 1. Statistics on EP3L340

TreeBLASTn			
Max tree size	Logic util	BlockRam	Max frequency
1,024	70%	9K: 98.5%	89 MHz
SmithWatermanN			
Max array size	Logic util	BlockRam	Max frequency
256	64%	9K: 24.6%	70 MHz

To measure the performance of CAAD BLASTn, we tested a number of nucleotide queries whose lengths are in the range of 100~1000, against the env_nt database (17,669,314 sequences; 7,169,203,781 residues). Without losing generality, the average ratios of the reduced database from the prefilter are 1.20% and 0.10% for ungapped and gapped alignments, respectively. However, the sizes of the reduced database from the TreeBLASTn module for ungapped and gapped alignments are not identical since the criteria used in the decision making module (DM in Figure 1) are not identical. Overall outputs are identical between CAAD BLASTn and NCBI BLASTn. The average time per query of the original NCBI BLASTn (downloaded from the NCBI web site) is about 120 seconds on a 2009-era high-end workstation. The FPGA accelerated time per query is generally much less than 10 seconds for at least a 12x speed-up.

5. Conclusion

We have described the design and implementation of CAAD BLASTn in which an FPGA-based prefiltering method is used. The prefilter has two major components: DNA-sequence versions of TreeBLAST and Smith-Waterman. A factor of 12x speed-up is obtained without losing sensitivity.

Acknowledgment

This work was supported in part by the NIH through award #R01-RR023168-01.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic Local Alignment Search Tool," Journal of Molecular Biology, Vol. 215, No. 3, pp. 403-410, Oct. 1990.

[2] J. Buhler, J. Lancaster, A. Jacob, and R. Chamberlain, "Mercury BLASTN: Faster DNA Sequence Comparison Using a Streaming Hardware Architecture," Proceedings of the Reconfigurable Systems Summer Institute, July 2007.

[3] S. Datta, P. Beeraka, and R. Sass, "RC-BLASTn: Implementation and Evaluation of the BLASTn Scan Function," Proc. 17th IEEE Symposium Field-Programmable Custom Computing Machines, pp. 88-95, 2009.

[4] M. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt, "Single Pass Streaming BLAST on FPGAs," *Parallel Computing*, V33, pp. 741-756, 2007.

[5] B. Harris, A. Jacob, J. Lancaster, et al., "A Banded Smith-Waterman FPGA Acceleration for Mercury BLASTP," Proceedings of the Intl' Conference on Field Programmable Logic and Applications, pp. 765-769, 2007.

[6] A. Jacob, J. Lancaster, and J. Buhler, "Mercury BLASTP: Accelerating Protein Sequence Alignment," ACM Trans. Reconfigurable Technology and Systems, Vol 1, 2.9, 2008.

[7] I. Korf, M. Yandell, and J. Bedell, *BLAST*, O'Reilly, 2003.

[8] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster, "Biosequence Similarity Search on the Mercury System," Proc. 15th IEEE Conf. Application-Specific Systems, Architectures and Processors, pp. 365-375, 2004.

[9] J. Lancaster, J. Buhler, and R. Chamberlain, "Acceleration of Ungapped Extension in Mercury BLAST," Microprocessors and Microsystems, Vol. 33, pp. 281-289, 2009.

[10] J. Park, Y. Qiu, and M. Herbordt, "CAAD BLASTP: NCBI BLASTP Accelerated with FPGA-Based Pre-Filtering," Proceedings of the 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 81-87, 2009.

[11] T. Smith and M. Waterman, "Identification of Common Molecular Subsequences," J. Molecular Biology, V147, pp. 195-197, 1981.

[12] E. Sotiriades and A. Dollas, "A General Reconfigurable Architecture Architecture for the BLAST algorithm," The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, Vol. 48, Issue 3, pp. 189-208, Sept. 2007.

[13] F. Xia, Y. Dou, and J. Xu, "Hardware BLAST Algorithms with Multi-seeds Detection and Parallel Extension," Proceedings of the 4th Workshop on Reconfigurable Computing: Architecture, Tools and Application, Landon, UK, pp. 39-50, 2008.