Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 3, Issue. 3, March 2014, pg.307 – 315

RESEARCH ARTICLE

FPGA Implementation of Wu-Manber Algorithm for BLASTN DNA Sequence Matching

Anitha Ranganathan
M. E. Scholar
Sri Shakthi Institute of Engineering and Technology

Abstract

BLAST is one of the most popular sequence analysis tools used by molecular biologists. Blast is fast and it is ubiquitous within the genomic community. However, because the size of genomic databases is growing rapidly, the computation time of BLAST, when performing a complete genomic database search, is continuously increasing. In order to overcome this time consuming process, we propose a PPBF architecture, which is used to speed up the BLASTN process more than the NCBI BLASTN software running on a general purpose computer.

Index Terms- Bloom filter, Genomic sequence analysis, hash table.

I. Introduction

Scanning genome sequence databases is a common and often repeated task in molecular biology. The need for speeding up these searches comes from the rapid growth of these gene banks: every year their size is scaled by a factor of 1.5 to 2 [1]. One of the most widely used sequence analysis tools to use heuristics is the basic local alignment search tool (BLAST) [2]. BLASTN, a version of BLAST specifically designed for DNA sequence searches, consists of a three-stage pipeline.

Stage 1: Word-Matching detect seeds (short exact matches of a certain length between the query sequence and the subject sequence), the inputs to this stage are strings of DNA bases, which typically uses the alphabet {*A*, *C*, *G*, *T*}.
Stage 2: Ungapped Extension extends each seed in both directions allowing substitutions only and outputs the resulting high-scoring segment pairs (HSPs). An HSP [3] indicates two sequence fragments with equal length whose alignment score meets or exceeds an empirically set threshold (or cutoff score).
Stage 3: Gapped Extension uses the Smith-Waterman dynamic programming algorithm to extend the HSPs allowing insertions and deletions.
The basic idea underlying a BLASTN search is *filtration*. Although each stage in the BLASTN pipeline is becoming more sophisticated, the exponential increase in the volume of data makes it important that measures are taken to reduce the amount of data that needs to be processed. Filtration discards irrelevant fractions as early as possible, thus reducing the overall computation time. Analysis of the various stages of the BLASTN pipeline (see Table I) reveals that the word-matching stage is the most time-consuming part. Therefore, accelerating the computation of this stage will have the greatest effect on the overall performance.
In this paper, we propose a computationally efficient architecture to accelerate the data processing of the word-matching stage based on field programmable gate arrays (FPGA). FPGAs are suitable candidate platforms for high-performance computation due to their fine-grained parallelism and pipelining capabilities.

II. Word Matching Stage

The first stage of BLASTN is used to find “seeds” or word matches. A word match is a string of fixed length *w* (referred to as “*w*-mer”) that occurs in both the query sequence and the database sequence.

TABLE I
PERCENTAGE OF TIME SPENT IN EACH STAGE OF NCBI BLASTN SCANNING THE MOUSE GENOME WITH A PARTIAL HUMAN DNA SEQUENCE (DATA TAKEN FROM [4])

Query size	10 kbase	100 kbase	1 Mbase
Stage 1	86.5%	83.3%	85.3%
Stage 2	13.3%	16.6%	14.65%
Stage 3	0.2%	0.1%	0.05%

Using the alphabet {*A*, *C*, *G*, *T*}, NCBI BLASTN reduces storage and I/O bandwidth by storing the database using only 2 bits per letter (or base). The default *w*-mer length for a nucleotide search is set to 11.
The word-matching stage implementation of NCBI BLASTN
first examines *w*-mers on a byte boundary (i.e., 8-mers).
Subsequently, exact 8-mer matches are extended in both directions to find possible 11-mer matches. If two matching 11-mers occur in close proximity, they are likely to generate the same HSPs. NCBI BLASTN therefore implements a redundancy eliminator to avoid repetitive inspections on the same segment in later stages.
Our FPGA-based accelerator design for BLASTN does not follow exactly the same working mechanism presented in the NCBI BLASTN software. Instead, we have chosen FPGA favorable algorithms to achieve the same functionality. Our word-matching stage design can be decomposed into three substages, as shown in Figure 1. The first substage is a parallel Bloom filter; the second substage is a false-positive eliminator to examine the data passing the parallel Bloom filters, and the last substage eliminates redundant matches. The substage composition is similar to that of Mercury [4], but the detailed architecture is different.

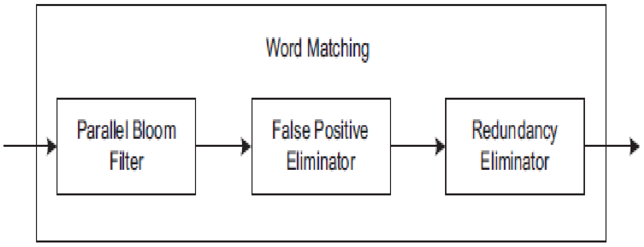

Figure 1. Three substages of word matching stage of BLASTN

A. Parallel Bloom Filter Architecture

The Bloom filter only produces false positive results but never false-negative answers to the query [13]. The conventional Bloom filter architecture is efficient for membership test; its direct implementation is not suitable for the high performance design on an FPGA. Current on-chip memory blocks only provide limited access at the same clock cycle. Pipelining or memory duplication is required for multiple memory access requests. For example, Mercury implements the Bloom filters using the conventional structure. It doubles the clock frequency of the block RAMs to reduce memory consumption, but still requires four copies of the *m*-bit vector to process 16 hash queries at the same clock cycle. The limited on-chip memory becomes the bottleneck for the use of Bloom filter. In contrast, we introduced a novel architecture for the Bloom filter design to provide a better computational efficiency, the parallel partitioned Bloom filter (see Figure2).

We apply three techniques to improve the throughput compared to the conventional Bloom filter architecture.

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.

Although hash collision might appear in both the programming and querying stage, the hash table lookup stage can eliminate all negative influences.

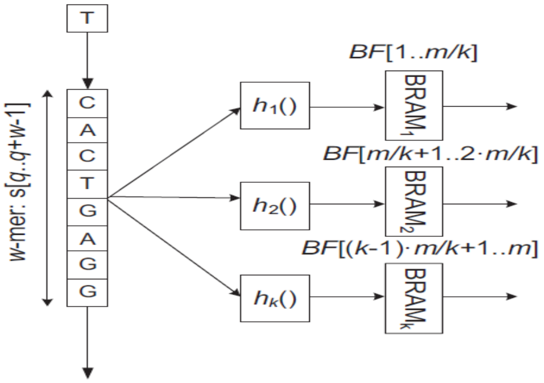

Figure 2. Partitioned Bloom Filter Architecture

Although a local stalling mechanism is applied to guarantee that there is no *w*-mer loss, a processing speed difference would appear among different *w*-

mer buffers. Therefore, we also integrate a *w*-mer scheduler to control the *w*-mer flow. The rule for the w-mer scheduler is:
1) if w-mer bufferi is empty, PPBFi can process data from its nearest w-mer buffer;
2) if only one w-mer is non-empty, all PPBFs can process data from this buffer;
3) if all w-mer buffers are empty, update all w-mer buffers simultaneously.
Buffer cond is a control signal that informs the w-mer scheduler about the empty buffers. Figure 3. is an example containing eight PPBFs, where each PPBF contains two different hash functions.

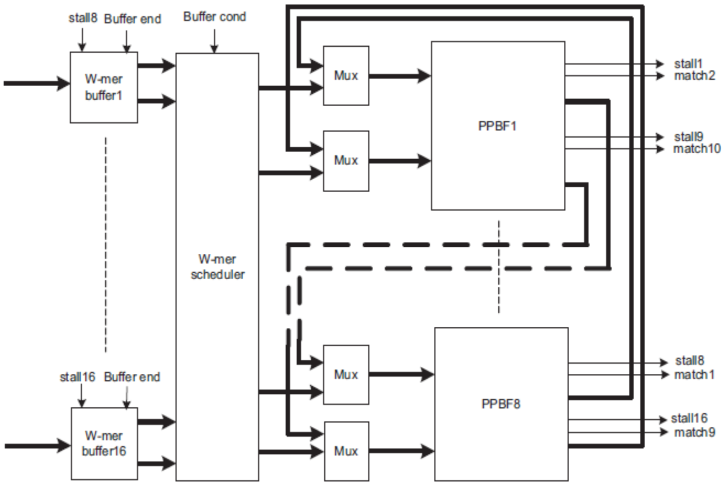

Figure 3. Parallel Bloom Filter Architecture with Eight PPBFs with two hash functions each

B. False-Positive Eliminator Design

The second substage 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 substage 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 problems 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, a perfect hash function [16] 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 entry with no collisions among the keys in the set. 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 [4] implements the hash table using a near perfect hashing strategy [19], which bypasses the constraint for a perfect hash function. However, considerable effort is still required to get the "near-perfect" hash functions. Cuckoo hashing [17] 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) 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.

We constructed a QLT of size 24 kbit with 4 k entries. As the size of query sequence increases, the number of w-mers in the same buckets increases correspondingly. This puts a greater burden on finding a perfect hash function. In rare cases, it is difficult to get a perfect hash function using a simple representation. Under such circumstances, a secondary table is applied to store w-mers that would introduce a collision using a simple hash function. A collision flag "C" is set to identify the colliding keys. As the total number of such w-mers is small, it only has a minor influence on the overall performance. Although bucket hashing can avoid hash collisions for a majority of keys, an additional operation is still required for duplicate keys.

As duplicate keys cannot be distinguished by hash functions, we constructed a separate table, called a duplicate table, to store all duplicate keys. We also set a duplicate flag "D" field to indicate the appearance of a duplicate key. Each element in the primary table, the secondary table, and the duplicate table takes 64 bits. Figure 4 shows the bucket hash data path. The definitions for the primary, secondary, and duplicate table fields are:

En effective slot in the primary table, i.e., query w-mer was programmed into this slot during the programming stage;

D duplicate flag;

C collision flag;

wmer query w-mer;

PTR if the collision flag is "1" then, it indicates a position in the secondary table, if the duplicate flag is "1," then it indicates the address in the duplicate table, otherwise, it indicates the corresponding address in the query sequence;

QPTR indicates the position of the match w-mer in the query sequence;

End A flag that indicates no more duplicates or collisions for the current w-mer;

XX unused bits (set to 0).

There are several different cases for the hash table access. If a w-mer x maps to the primary table entry h(x) without collision and duplication, it only requires a single off-chip memory access. In the duplication-only case, the address we get from the primary table indicates a start location in the duplicate table; we should read out all the position information from this location until an "End" flag is met. If the collision flag is set in the primary table, we should access the secondary table based on the address information; if duplication appears at the same time, the address stored in the secondary table indicates a location in the duplicate table

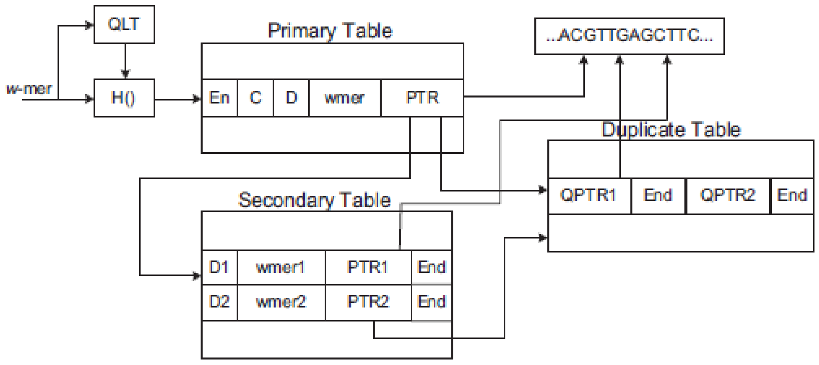

Figure 4. Bucket hash data path

C. Redundancy Eliminator Design

In order to avoid repeated generation of the same sequence alignment during the ungapped extension stage of BLASTN, it uses a redundancy filter to eliminate w-mers that lead to the same ungapped extension range. Each w-mer is represented by an ordered pair (q_j, d_k), where q_j and d_k are indices of the query and database sequence, respectively. The diagonal of this w-mer is defined as $D = q_j - d_k$. Redundancy matches are eliminated by examining their diagonals. In NCBI BLASTN, it also uses the feedback from the ungapped extension stage to eliminate redundancy matches. In contrast, our design is less stringent. We only eliminate “true overlapping” match w-mers, i.e., if two consecutive matches share the same diagonal and they have an overlapping part, we discard the latter as a redundant match. The non-overlapping diagonal will be updated, once a non-overlapping match is found. Although our heuristic is less stringent than NCBI BLASTN’s, there is no significant influence on the overall performance.

III. Conclusion and Future Scope

In this paper, we have presented FPGA-based PPBF architecture to accelerate the word-matching stage of BLASTN, which is a bio-sequence search tool of high importance to Bioinformatics research. Our design consists of three substages, a parallel Bloom filter, an off-chip hash table, and a match redundancy eliminator.

As FPGA-based designs exhibit high performance for parallel computing and fine-grained pipelining, we can expect obvious performance improvements of other applications in Bioinformatics. Therefore, we are also planning to design PPBF architecture with Wu-Manber algorithm to improve its speed.

REFERENCES

[1] *GenBank Statistics at NCBI* [Online]. Available: http://www.ncbi. nlm.nih.gov/genbank/genbankstats.html

[2] 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.

[3] *BLAST Algorithm* [Online]. Available: http://en.wikipedia.org/wiki/BLAST

[4] 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.

[5] 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.

[6] W. J. Kent, "BLAT–the BLAST-like alignment tool," *Genome Res.*, vol. 12, pp. 656–664, Mar. 2002.

[7] B. Ma, J. Tromp, and M. Li, "Patternhunter: Faster and more sensitive homology search," *Bioinformatics*, vol. 18, no. 3, pp. 440–445, 2002.

[8] M. Li, B. Ma, D. Kisman, and J. Tromp, "Patternhunter II: Highly sensitive and fast homology search," *J. Bioinf. Comput. Biol.*, vol. 2, no. 3, pp. 417–439, 2004.

[9] K. Muriki, K. D. Underwood, and R. Sass, "RC-BLAST: Toward a portable, cost-effective open source hardware implementation," in *Proc. 19th Int. Parallel Distrib. Process. Symp.*, vol. 8. 2005, pp. 1–8.

[10] E. Sotiriades, C. Kozanitis, and A. Dollas, "FPGA based architecture for DNA sequence comparison and database search," in *Proc. 20th Int. Parallel Distrib. Process. Symp.*, 2006, p. 8.

[11] D. Lavenier, G. Georges, and X. Liu, "A reconfigurable index FLASH memory tailored to seed-based genomic sequence comparison algorithms," *J. VLSI Signal Process. Syst., Special Issue Comput. Archit. Accelerat. Bioinf. Algorithms*, vol. 48, no. 3, pp. 255–269, 2007.

[12] J. Buhler, J. Lancaster, A. Jacob, and R. Chamberlain, "Mercury BLASTN: Faster DNA sequence comparison using a streaming architecture," in *Proc. Reconfig. Syst. Summer Inst.*, Jul. 2007, pp. 1–7.

[13] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[14] S. Dharmapurikar and J. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *IEEE J. Sel. Areas Commun.*,vol. 24, no. 10, pp. 1781–1792, Oct. 2006.

[15] M. Nourani and P. Katta, "Bloom filter accelerator for string matching," in *Proc. 16th Int. Conf. Comput. Commun. Netw.*, 2007, pp. 185–190.

[16] F. C. Botelho, Y. Kohayakawa, and N. Ziviani, "An approach for minimal perfect hash functions for very large databases," Dept. Comput. Sci., Univ. Federal de Minas Gerais, Belo Horizonte, Brazil, Tech. Rep., 2006.

[17] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[18] *BLAST Programs at NCBI* [Online].Available: http://www.ncbi.nlm.nih. gov/BLAST/

[19] J. Buhler, "Mercury BLAST dictionaries: Analysis and performance measurement," Dept. Comput. Sci. Eng., Washington Univ., St. Louis, MO, Tech. Rep., 2007.