Reconfigurable Accelerator for the Word-Matching Stage of BLASTN

Yupeng Chen, Bertil Schmidt, *Senior Member, IEEE*, and Douglas L. Maskell, *Senior Member, IEEE*

***Abstract*— BLAST is one of the most popular sequence analysis tools used by molecular biologists. It is designed to efficiently find similar regions between two sequences that have biological significance. 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. Thus, there is a clear need to accelerate this process. In this paper, we present a new approach for genomic sequence database scanning utilizing reconfigurable field programmable gate array (FPGA)-based hardware. In order to derive an efficient structure for BLASTN, we propose a reconfigurable architecture to accelerate the computation of the word-matching stage. The experimental results show that the FPGA implementation achieves a speedup around one order of magnitude compared to the NCBI BLASTN software running on a general purpose computer.**

***Index Terms*— Bloom filter, Genomic sequence analysis, hash table, reconfigurable computing.**

I. Introduction

SCANNING genomic 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]. The aim of a scan operation is to find similarities between the query sequence and a particular genome sequence, which might indicate similar functionality from a biological point of view. Dynamic programming-based alignment algorithms can guarantee to find all important similarities. However, as the search space is the product of the two sequences, which could be several billion bases in size, it is generally not feasible to use a direct implementation. One frequently used approach to speed up this time-consuming operation is to use heuristics in the search algorithm. One of the most widely used sequence analysis tools to use heuristics is the basic local alignment search tool (BLAST) [2]. Although BLAST's algorithms are highly optimized for similarity search, the ever growing databases outpace the speed improvements that BLAST can provide on a general purpose PC.

Manuscript received May 19, 2011; revised March 21, 2012; accepted April 12, 2012. Date of publication May 18, 2012; date of current version March 18, 2013.

Y. Chen and D. L. Maskell are with the School of Computer Engineering, Nanyang Technological University, 639798 Singapore (e-mail: chen0511@e.ntu.edu.sg; ASDouglas@ntu.edu.sg).

B. Schmidt is with the Institut fuer Informatik, Johannes Gutenberg University, Mainz 55001, Germany (e-mail: bertil.schmidt@unimainz.de).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2012.2196060

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%

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 a 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.

The rest of this paper is organized as follows. In Section II, we review past research work in this field. Section III gives a brief introduction to our experimental platform. We present a computationally efficient architecture for the word-matching stage in Section IV. The performance of our new architecture is evaluated in Section V. Section VI concludes this paper.

II. Related Work

There have been several approaches to accelerate bio-sequence similarity searches. Some of these approaches use special hardware, while others attempt to solve this problem in software using better algorithms or heuristics. Hybrid approaches employ both the general purpose computer and specialized hardware.

MegaBLAST [5] is used by the National Center for Biotechnology Information (NCBI) as a faster alternative to BLASTN. It achieves a faster processing speed by sacrificing substantial sensitivity. BLAT [6] can quickly find alignments in sequences of high similarity. It indexes the entire database offline before being used in a search. By eliminating the need to scan the database, it achieves more than an order of magnitude speedup comparing to BLASTN. However, a tradeoff is made between the processing speed and the sensitivity. PatternHunter [7] uses a spaced seed model to achieve faster processing speed and higher sensitivity; PatternHunter II [8] implements the optimized multiple seeds scheme to further increase the sensitivity. The spaced seed model is designed to gain profits based on the following constraints: an appropriately chosen model has a significantly higher probability of having at least one hit in a homologous region, compared to BLAST's consecutive seed model, while having a lower expected number of hits. If the spaced seed is not properly designed, there will be too many random hits to slow the subsequent computation. However, to find an optimal seed of given weight and length is NP-hard [8].

FPGAs can provide outstanding performance on parallel data processing, which make them a good option for algorithm acceleration. A small number of designs to speed up BLAST's performance on FPGA devices have been presented. RC-BLAST [9] is an early implementation of BLAST. It first profiles the application to identify the compute-intensive segments. TUC-BLAST [10] accelerates DNA searches for small query sequences (1000 characters) regardless of the database size. It achieves a significant performance improvement compared to the BLAST software, but its performance for large query sequences is not clear. Lavenier [11] indexes both the query sequence and the database sequence based on seed of length w. Afterwards, the neighborhood information to conduct the extension from off-chip Flash memory is extracted. Although Flash memory can provide a large space to store the indexed information, its bandwidth can become the performance bottleneck when throughput rate is the first priority. Mercury [12], [4] accelerates BLASTN's word-matching stage computation by applying parallel Bloom filters [13]. In addition, the Mercury system also provides an efficient near-perfect hashing strategy to eliminate false-positive answers. As BLASTN's word-matching stage is to detect exact matches between two sequences, the Bloom filter architecture is a proper choice to achieve this goal. In this paper, we also accelerate the word-matching stage computation using the Bloom filter architecture. Our approach differs from the Mercury approach in that we apply a partitioned Bloom filter to provide better computational efficiency, and a bucket hashing data structure to ease off-chip hash table accesses. A detailed comparison to Mercury is given in Section V.

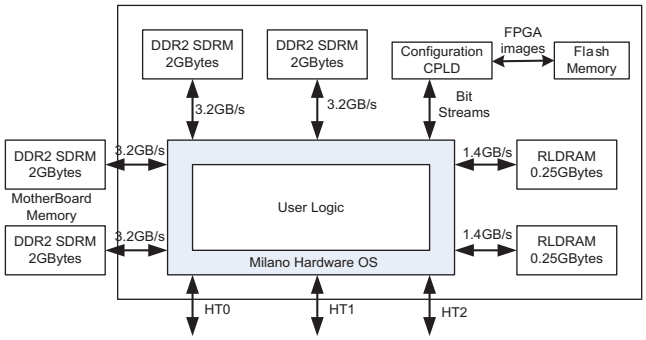


Fig. 1. DRC coprocessor diagram.

The Bloom filter architecture has been used in a number of application fields (e.g., pattern matching). Various Bloom filter architectures have been proposed to achieve different functionalities. Dharmapurikar and Lockwood [14] present a multipattern matching algorithm using multiple parallel Bloom filters on an FPGA. Each Bloom filter detects strings of a unique length. Nourani and Katta [15] combine the Bloom filter and the parallel hash engines to achieve higher throughput. The feed-forward Bloom filter [16] applies a second bit array to reduce scan time and memory requirement for large number of patterns with relatively few matches.

III. Implementation Platform

We have chosen the DRC coprocessor system [17] as our target experiment platform. Accelium is the third generation of DRC coprocessors. It is a high-performance computing system for processing-intensive applications, consisting of three HyperTransport bus and six memory interfaces to the user's logic design. Application images are stored in Flash memory, and are used to configure the FPGA at power-on. Fig. 1 gives the diagram of the coprocessor.

IV. Word-Matching Accelerator Architecture

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. 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

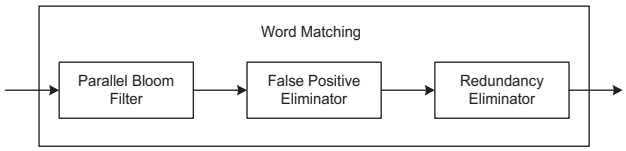


Fig. 2. Three substages of our FPGA-based accelerator for the word-matching stage of BLASTN.

substages, as shown in Fig. 2. 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.

A. Parallel Bloom Filter Architecture

The word-matching stage aims to find good alignments containing short exact matches between a query sequence and a database sequence. Such matches could be computed using data structures 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 $BF[1, \ldots, m]$. A family of k hash functions h_i: $S \rightarrow A$, $1 \leq i \leq k$, is associated to the Bloom filter, where S is the key space and $A = \{1, \ldots, m\}$ is the address space. A Bloom filter is a simple space-efficient randomized hashing data structure suitable for quick membership tests on FPGA implementations.

A Bloom filter works in two steps.

1) *Programming:* For a given set I of n keys, $I = \{x_1,\ldots,x_n\}$, $I \subseteq S$, the Bloom filter's programming process is described as follows. First of all, initialize the bit vector m with zeros, then, for each key $x_j \in I$, compute its k hash values $h_i(x_j)$, $1 \leq i \leq k$, subsequently, set the bit vector to one according to the k hash values (i.e., $BF[h_i(x_j)] := 1$ for all $1 \leq i \leq k$).
2) *Querying:* the querying process of the Bloom filter works the same as its programming process. For a given key x, compute k hash values $h_i(x)$, $1 \leq i \leq k$. If any of the k bits $BF[h_i(x)]$, $1 \leq i \leq k$, is zero, then $x \notin I$, otherwise, x is said to be a member of set I with a certain probability.

One key feature for the Bloom filter is that false-positive answers are possible. This is due to the fact that the hash function could hash two different keys into the same address with low probability. The Bloom filter only produces false-positive results but never false-negative answers to the query [13]. The false-positive probability (FPP) of a Bloom filter is shown in (1)

$$\text{FPP} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k. \quad (1)$$

It is determined by three parameters, the bit-vector length m, the number of keys n, and the number of hash functions k. Obviously, FPP decreases as the bit-vector size m increases, and increases as the number of inserted keys n increases. It can be shown that for a given m and n, the optimal number of hash functions k_{opt} is given by

$$k_{\text{opt}} = \frac{m}{n}\ln 2. \quad (2)$$

The corresponding FPP is then

$$\left(\frac{1}{2}\right)^{k_{\text{opt}}} \approx 0.6185^{\frac{m}{n}}. \quad (3)$$

Hence, in the optimally configured Bloom filter, the false-positive probability decreases exponentially with the size of the bit-vector. If we maintain a fixed FPP, the bit-vector size needs to scale linearly with the inserted key set. Table II shows the required Bloom filter size for FPP values of around 10^{-5} and around 10^{-3}, and varying query lengths. About 2 Mbits of memory is enough for scanning a database with a query sequence of length 100 kbase and 16 hash functions. The Bloom filters have a relatively simple architecture, which makes them suitable for hardware implementation, thus outweighing the drawbacks associated with false positives. Furthermore, all false-positive matches can be eliminated in the next stage and, thus, there is no negative influence on the sensitivity.

TABLE II
BLOOM FILTER MEMORY SIZE (IN BITS) FOR DIFFERENT PARAMETER COMBINATIONS

Query size (n)	No. of hash functions (k)	Bit length of Bloom filter (m)	FPP
1 k	16	23 084	1.53e-5
10 k		230832	
100 k		2308313	
1 k	8	11542	2.91e-3
10 k		115416	
100 k		1154157	

Unlike the NCBI software implementation, which first tests the membership on the byte boundary between the query sequence and the database, our FPGA design conducts the membership test on the w-mer boundary directly. We use the on-chip memory resource (a set of block RAMs) to implement the m-bit vector. Thus, it can provide faster access speed and parallel access in the same clock cycle.

The conventional design for the identification of w-mers using a Bloom filter is shown in Fig. 3. 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 substrings. Assume $w = 3$ and the query is "cttgtata," then, the parsed substrings 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 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

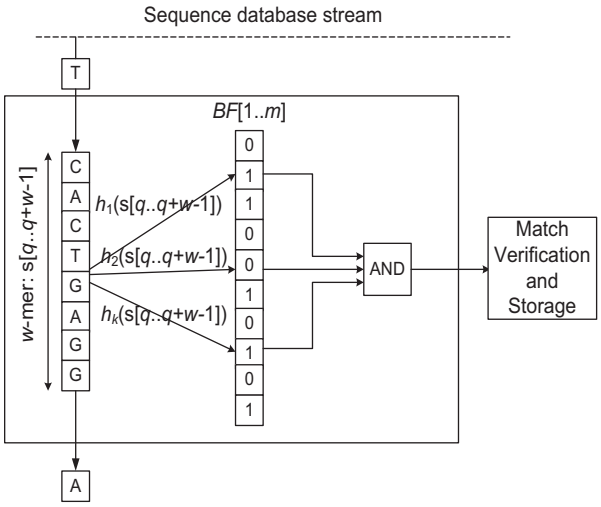


Fig. 3. Conventional design for identifying w-mers in a sequence database stream using a Bloom filter.

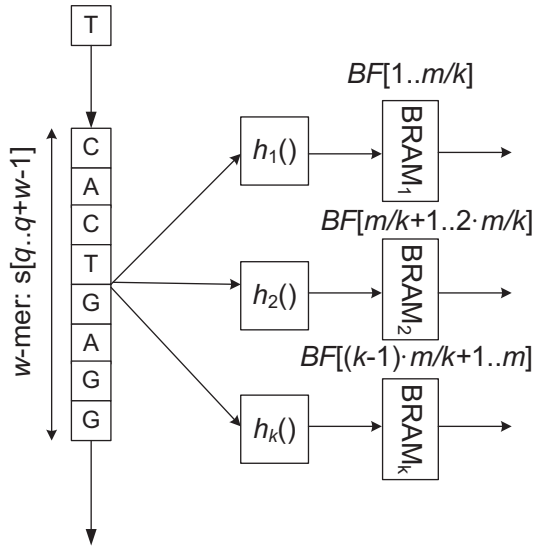


Fig. 4. Partitioned Bloom filter architecture using on-chip BRAM modules.

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 Fig. 4).

In our previous work, we have implemented a 4×4 parallel partitioned Bloom filter [18] to test the computation efficiency introduced by the partitioned architecture. In this paper, we further analyze the influences of different architecture configurations on the partitioned Bloom filter. The detailed analysis is presented in Section V. Based on the query sequence and database sequence, we implement an 8×2 parallel Bloom filter architecture, which theoretically doubles the throughput comparing to the 4×4 architecture under zero-match condition, to gain better performance. Our design takes advantage of the fact that mismatches appear far more frequently than

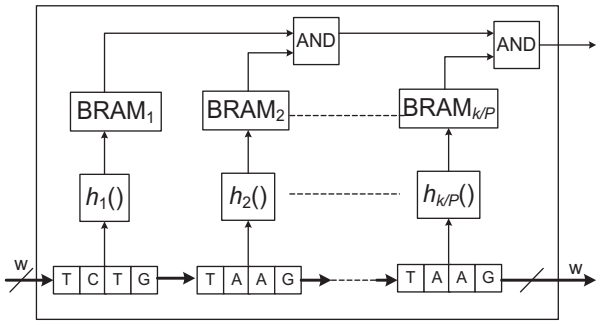


Fig. 5. PPBF architecture with k/P hash functions.

matches in the BLASTN word-matching stage and a match key has much tighter requirements than mismatches (once a key fails in any of the k hash queries, it will be defined as a mismatch, in contrast, a match key should pass all hash queries). 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.

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 our design, we partition a Bloom filter of bit-length m into m/k embedded memory blocks (BRAMs) as shown in Fig. 4, where k is the number of hash functions. This results in a new FPP as given in (4)

$$\text{FPP}_p = \left(1 - \left(1 - \frac{1}{\frac{m}{k}}\right)^n\right)^k. \tag{4}$$

For instance, using the parameters mentioned in Table II, we would get $|\text{FPP}_p - \text{FPP}| \leq 5.5e\text{-}8$ for $k = 16$ and $|\text{FPP}_p - \text{FPP}| \leq 5.5e\text{-}20$ for $k = 8$. Thus, the negative influence on the FPP introduced by the Bloom filter partitioning is negligible.

Our basic building block is a pipelined partitioned Bloom filter with k/P hash functions, denoted as PPBF(k/P), where P is the degree of parallelism. Its architecture is shown in Fig. 5. In each clock cycle, it can support k/P different hash queries. The hash functions used in the PPBF block are chosen from the H_3 family [19], which 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

$$h_i(X) = (d_{i1} \cdot x_1) \oplus (d_{i2} \cdot x_2) \oplus \cdots \oplus (d_{ib} \cdot x_b) \tag{5}$$

where "$\cdot$" 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 shorten the computation

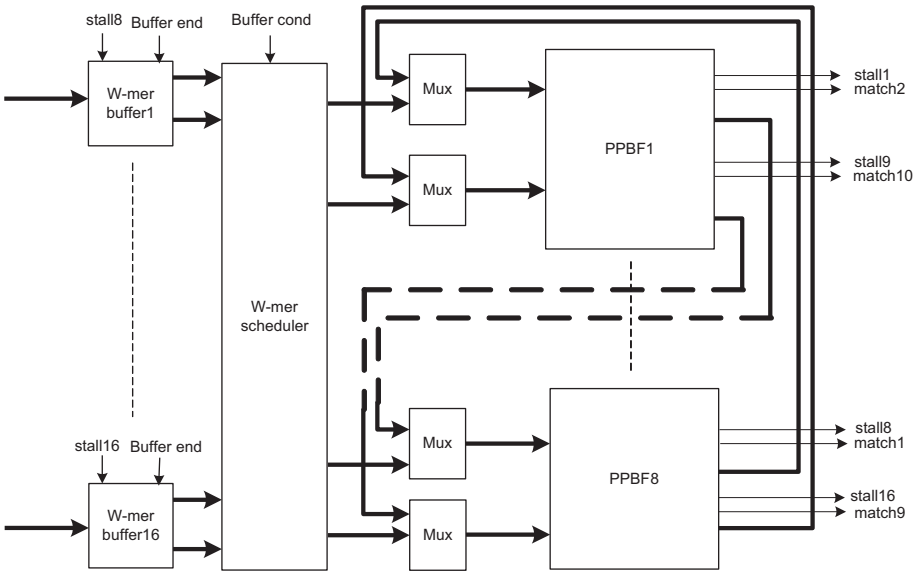


Fig. 6. Parallel Bloom filter architecture consisting of eight PPBFs with two hash functions each.

time on a FPGA. In addition, the total number of hash functions required is small. Thus, the resource needed for the hash function representation is negligible. Although hash collision might appear in both the programming and querying stage, the hash table lookup stage can eliminate all negative influences.

The second step is to implement P PPBFs in parallel. Fig. 6 is an example containing eight PPBFs, where each PPBF contains two different hash functions. This implementation reduces the FPP, making it more computationally efficient. We refer to the design in Fig. 6 as an 8×2 parallel Bloom filter (denoted as PBF8 $\times$ 2). When we use dual port RAM for the "m-bit" vector module, each partitioned memory block can support an additional hash query in the same clock cycle. Thus, the system's throughput can be doubled. Assuming our design works with a 100-MHz clock, it can achieve a theoretical maximum throughput 1.6 Gbases/s under the zero-matching condition (i.e., all querying w-mers fail at the first PPBF they met).

As mentioned before, a w-mer passing one PPBF should be sent to the next PPBF for further examination to reduce FPP, which would introduce a collision between the match w-mer and the new incoming w-mer. To avoid such collision, we applied a local stall mechanism as: if the output match-bit of PPBF_i equals zero for any $1 \leq i \leq P$, indicating that a mismatch is found, then a new w-mer is input to the PPBF, if an output match bit of any PPBF_i equals one, the corresponding w-mer is passed to $\text{PPBF}_{(i+1)}$ for all $1 \leq i \leq P-1$; at the same time, a corresponding input w-mer from the w-mer buffer should be stalled, based on a local stall signal, to prevent a w-mer loss. The match output from PPBF_p is sent to PPBF_1 for further testing. A true match is generated if and only if a w-mer passes all PPBFs.

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 buffer_i is empty, PPBF_i 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.

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 [20] 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

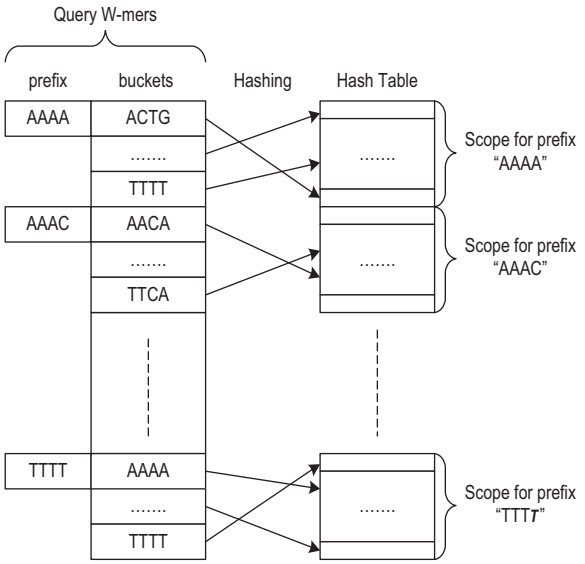


Fig. 7. Hash table construction using bucket hashing.

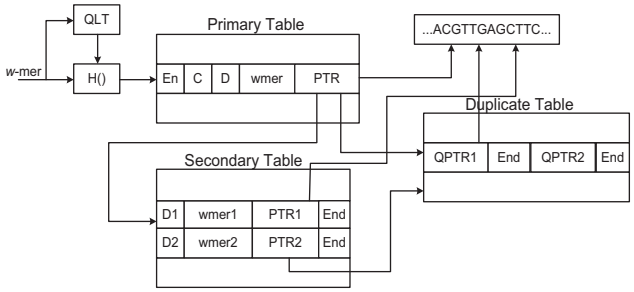


Fig. 8. Bucket hash data path.

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 [24], which bypasses the constraint for a perfect hash function. However, considerable effort is still required to get the "near-perfect" hash functions. Cuckoo hashing [21] 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.

An example of the off-chip hash table construction using the bucket hash strategy is shown in Fig. 7. Query w-mers sharing the same prefix are programmed to a given area in the hash table without hash collision. For example, to support a 10 kbase query sequence, 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. Fig. 8 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

```
address ← h(x)                          // address for primary table
if ((C = 0) and (D = 1))                // only duplication appear
    address ← PTR + DupOffset           // address for duplicate table
    return QPTRs                        // return QPTRs until meet an END
else if (C = 0)                         // collision apepar
    if (D = 0)                          // no duplication jump to secondary table
        address ← PTR + SecOffset
        return PTRs                     // return PTRs until meet an END
    else                                // duplication appear jump to duplicate table
        address ← PTR + DupOffset
        return QPTRs
```

Fig. 9. Hash table control logic.

indicates a location in the duplicate table. The control logic of the hash table lookup is illustrated in Fig. 9. The query w-mers are also stored in the primary table and the secondary table. Each time we conduct a hash lookup, we will also compare the w-mer from the database against the one from the off-chip table. Thus, we can guarantee that there are no false positives after the hash table lookup stage.

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.

V. Performance Analysis

The word-matching stage accelerator has been implemented using Verilog HDL and integrated into the DRC coprocessor system. In the DRC system, a Xilinx Virtex-5 LX330 FPGA chip is available for the user application. A large volume of off-chip data can be stored 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. The final design consumes about 47% of the slice registers, 50% of the LUTs, and 85% of the on-chip memory resource (about 2 Mbits for the m-bit vector in the Bloom filter design). In order to quantify the performance improvements of our word-matching accelerator, we have designed several tests to simulate possible large-scale DNA sequence comparisons.

A. Performance Comparison Against the NCBI Stage 1

We have compared our FPGA implementation to the NCBI BLASTN version 2.2.9 [22], running on a general-purpose workstation containing an Intel Core2 Duo CPU (3.2 GHz) with 8 GB RAM. Both the Virtex-5 LX330 and the Core2 PCU were released in 2006 using 65 nm technology. As both platforms belong to the same technology generation, we think the performance comparison between them is a proper choice. The parameters for both the FPGA and the software program are set to the same default values. We have chosen the mouse genome (mm8, consisting of about 1.4 Gbases after removing unknown characters and Ns, including 19 sequences) as the database sequence. Query sequences of different sizes have been randomly chosen from the human genome (hg18, 10 kbase, 50 kbase, and 100 kbase). The estimated highest clock frequency that our FPGA design can support is 170 MHz. In our experiment, we have tested our design with 133 MHz clock frequency due to the limited choices of clock frequency on the experimental board. Table III shows the performance comparison between NCBI BLASTN Stage 1 and our word-matching accelerator. The database sequences are processed sequence by sequence.

Our design reports more 11-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 11-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 11-mer match with the search terminating once a match is found, if a long k-mer ($k > 11$) match exists, 11-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 the 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 1 can only generate 1212 HSPs. In addition, we have further evaluated the performance on the gapped extension stage. Almost all HSPs generated by the FPGA pass the gapped extension stage, while, for NCBI, a small portion of the HSPs are failed at this stage.

Table III shows a significant speedup compared to the single threaded NCBI software. We have also tested the throughput of the NCBI BLASTN Stage 1 using different numbers of threads. As the computer used in this paper does not support hyper-threading, the two cores can process at most two threads at the same time. This explains why there is no

TABLE III
PERFORMANCE COMPARISON BETWEEN NCBI BLASTN STAGE 1 AND DRC ACCELERATOR

Query	10 kbase	50 kbase	100 kbase
No. of hits (FPGA)	9027656	39686463	82305804
No. of hits (NCBI Stage 1)	5223591	29364313	61974681
No. of HSPs (FPGA[1])	609	850	1270
No. of HSPs (NCBI Stage 2)	609	827	1212
No. of matches (FPGA[1])	609	850	1270
No. of matches (NCBI Stage 3)	609	800	1183
DRC word-matching[2] (Gbase/s)	1.94	1.58	0.756
NCBI Stage 1 1thread[2] (Mbase/s)	233	107	70
Speedup	8.3	14.7	10.8
NCBI Stage 1 2threads (Mbase/s)	466	200	140
Speedup	4.16	7.9	5.4
NCBI Stage 1 4threads (Mbase/s)	466	200	140
Speedup	4.16	7.9	5.4

[1]The number of HSPs and matches are computed using software with the same parameter settings as the NCBI BLASTN.
[2]The throughput reported above does not include loading the database from the hard disk to the local RAM and other preprocessing stages.

TABLE IV
PERFORMANCE COMPARISON USING *E. Coli* QUERY SEQUENCES

Query	10 kbase	50 kbase	100 kbase
No. of hits (FPGA)	3374936	17633252	33991990
No. of hits (NCBI Stage 1)	2736960	15390412	29675898
DRC word-matching (Gbase/s)	2	1.78	1.52
NCBI Stage 1 1thread (Mbase/s)	350	140	107
Speedup	5.7	12.7	14.1
NCBI Stage 1 2threads (Mbase/s)	700	280	200
Speedup	2.8	6.3	7.6

TABLE V
THROUGHPUT OF NCBI STAGE 1 USING CORE i7 CPU WITH FOUR CORES

Query	10 kbase	50 kbase	100 kbase
1 thread (Mbase/s)	233	127	82
2 threads (Mbase/s)	466	233	155
4 threads (Mbase/s)	700	466	280
8 threads (Mbase/s)	1400	700	466

performance difference between 2 threads and 4 threads for NCBI BLASTN. It should be noted that the maximum query sequence size that our design supports in a single database scan is 100 kbase. For query sequence larger than 100 kbase, we have to split the query into multiple segments (e.g., t segments, where each segment is $\leq$ 100 kbase). Then, the database needs to be scanned t times to complete the search process (Mercury [12], [4] has the same problem and also splits large query sequences due to on-chip BRAM limitations). This leads to a t times slowdown in the effective throughput. If more on-chip memory space was available, the Bloom filter could support longer query sequences in a single database scan.

Table III also shows a significant drop in speedup for the 100 kbase query, which has a high number of hits. This is because the Bloom filter needs to stall the pipeline when the match hit FIFO is full if the Bloom filter generates too many hits (i.e. over one hit per clock cycle in average). This is further complicated by the limited bandwidth (maximum 100 MHz $\times$ 64 bit) to the off-chip memory in our Virtex-5 based experimental platform. Theoretically, the off-chip memory can only support one hash lookup table access each cycle at 100 MHz. Thus, when there is a high hit rate, the off-chip memory access becomes a performance bottleneck. To show that our architecture is scalable in terms of the query sequence length, we have conducted another test (shown in Table IV) using a set of query sequences which generate fewer hits. The query sequences are randomly chosen from the *E. coli* genome (NC_008253, 10 kbase, 50 kbase, and 100 kbase, respectively). The performance drop between the 50 kbase query and the 100 kbase query for our FPGA design (Table IV) is only 14.6%, compared to a corresponding performance drop of 52% in Table III.

To examine the effect of match rate (or hit rate: defined as the number of hits divided by the database size), we define the expected number of match w-mers each cycle as the product of the match rate and the degree of parallelism. Then, for the hg18 100 kbase query, the match rate for a w-mer is about 5.86%. Sixteen w-mers are examined in parallel each cycle. Thus, 0.937 w-mers are expected to pass the Bloom filter stage each cycle (omitting the delays introduced by PPBFs to simplify the analysis), which approaches the theoretical upper bound of the off-chip memory access capability. When the off-chip table has to stop the pipeline to prevent hit loss, it leads to a drastic performance degradation. In contrast, for the *E. coli* 100 kbase query, the match rate is only about 2.4% and 0.388 w-mers are expected to pass the Bloom filter each cycle (well within the off-chip hash table's processing capability). When the supported query size each iteration is small, the negative influence of match rate is negligible. However, longer query segments introduce a new constraint on future Bloom filter design, which is not addressed in previous designs from the literature. In fact, the balance between the degree of parallelism, the maximum query size each iteration, and the match rate should be taken into account. One possible solution to moderate the off-chip memory access bottleneck is to use memory with a higher bandwidth, such as DDR3, or provide multiple off-chip memory interfaces.

Furthermore, we have recorded the NCBI BLASTN Stage 1 performance using the same query sequences from human genome on a state-of-the-art Intel Core i7 CPU containing four processor cores (2.93 GHz) and 8 GB RAM. The results are shown in Table V. It can be seen that a better performance is

TABLE VI
THROUGHPUT COMPARISON BETWEEN THE MERCURY AND THE DRC ACCELERATOR

Query	10 kbase	50 kbase	100 kbase	1 Mbase
Mercury (Mbase/s)	1400	700	350	35
DRC accelerator (Mbase/s)	1940	1580	756	80.6

TABLE VII
PERFORMANCE COMPARISON BETWEEN THE MERCURY DESIGN AND OUR 8×2 PBF WITH A 10 kbase QUERY SEQUENCE

	Mercury	DRC 8×2
Total size of block RAM consumed	128 kbit	128 kbit
Total number of hash functions	24	32
w-mers processing in parallel	16	16
FPP	0.35	0.045

achieved as more CPU cores are available. However, our DRC accelerator still reports a slightly better performance compared to the software version running with eight threads. We believe that with the help of newer FPGA technologies, better performances can be expected with the same architecture. As far as we know, the latest Virtex-7 FPGA chip can provide up to 85 MBit of block RAMs [23] (in contrast, our Virtex-5 chip only provides 10 Mbit of block RAMs). This indicates that we could support much longer query sequences or construct the hash lookup tables using the on-chip memories to give a faster access speed. Meanwhile, the Virtex-7 chips use a faster data transfer interface (e.g., DDR3), which also provides the potential to further increase our design's performance.

B. Performance Comparison Against the Mercury

Table VI shows the Stage 1 throughput comparison between the Mercury and our DRC accelerator under different query sizes (the Mercury throughput is reported in [4] also using the mm8 1.4 Gbase mouse genome and the hg18 query sequences). For short query size (e.g., 10 kbase query), both designs achieve a similar throughput. As the query size increases, our design achieves a higher throughput than the Mercury design. For large query sequences ($>$100 kbase), the slowdown due to the need to split the query sequence into multiple segments can be clearly seen. To further explore the influence of larger query sequences, we also conduct an extra experiment with a query sequence of 9 Mbase from the human genome against the mouse genome. The effective throughput is 8.2 Mbase/s (roughly 90 times slower than the 100 kbase query throughput). This additional experiment also proves the correctness of the t-times slowdown prediction (as described in Section V-A).

In Mercury, there are four Bloom filters working in parallel. Each Bloom filter has 6 hash functions and a 32 kbit memory for the bit-array. In total, it consumes 24 hash functions and 128 kbit block RAMs. In order to make a more qualitative comparison between the two designs, we have estimated the performance of our structure using the same amount of resources mentioned in the Mercury implementation. Table VII shows a performance comparison with a 10 kbase query sequence in terms of FPP.

TABLE VIII
EXPECTED MATCHES FOR DIFFERENT ARCHITECTURE CONFIGURATIONS

	2×8 arch	4×4 arch	8×2 arch	16×1 arch
Expected matches per cycle	$4r$	$8r$	$16r$	$32r$

Table VI shows that our 8×2 partitioned Bloom filter achieves a better FPP. We can deduce that, with the same FPP requirement, our 8×2 partitioned Bloom filter can provide a higher degree of parallelism. Furthermore, our Bloom filter design works well under a low match rate condition. For query/database pairs with a high match rate, we can apply a 4×4 Bloom filter structure to reduce the ingest throughput while having no influence on the FPP. A minor negative effect of the partitioned Bloom filter design is that, for a matching w-mer, it takes more cycles to complete all hash queries. However, different configurations of the Mercury's Bloom filter structure have much wider influence on the whole design. As its Bloom filter uses a conventional structure, modifications on its structure will change the total number of hash functions, the FPP, the degree of parallelism, and the number of matches per cycle. If the structure does not fit the constraint on the hash table lookup stage, hits loss will also appear.

The number of PPBFs applied in our PBF design determines the degree of parallelism. As each PPBF can support two hash queries at the same clock cycle, the system's degree of parallelism will be $2 \times m$ (m is the number of PPBFs applied in the design), theoretically. However, it is hard to proclaim an 8-PPBF design is two times faster than a 4-PPBF design. Eight PPBFs working in parallel also indicates more frequent local stalls (introduced by the false positives and the true matches), which can make the 8-PPBF design's performance drop faster. In addition, if the 8-PPBF design generates more than one match each cycle, the performance will drop further to guarantee no sensitivity loss.

The influence of the match rate r also varies depending on different Bloom filter architecture configurations. Assume that we do not count the stalls generated by each parallel partitioned Bloom filter. Table VIII shows the expected number of matches that can pass the partitioned Bloom filters for different configurations. For example, for a match rate of $r = 0.05$, the 16×1 architecture generates 1.6 matches each cycle, which exceeds the processing capability of the hash table access stage. However, the 8×2 architecture has only 0.8 matches. Thus, the 8×2 architecture is a better choice under such a match rate.

If we take stalls into account, the analysis becomes more complicated. To simplify our analysis, we make following assumptions: the match rate is r for each PPBF, the degree of parallelism is m and the number of hash functions in each PPBF is n (e.g., in the 8×2 architecture, $m = 8$ and $n = 2$); the FPP of each hash query is p, only one w-mer

is examined for each PPBF. Then, we can establish an upper bound on the speedup (compared to the conventional Bloom filter architecture that processes 1 w-mer per cycle) as

$$S = \frac{m}{1 + r(m-1) + (1-r)\left(\sum_{k=1}^{m-1} p^{nk}\right)} \tag{6}$$

where $r(m-1)$ represents the true matches from other PPBFs, $(1-r)(\sum_{k=1}^{m-1} p^{nk})$ represents false-positive matches from other PPBFs ahead. From (6), we get a rough performance evaluation of different architectural configurations. However, many other parameters influence the final speedup, such as the distribution of the matches and the processing capability in subsequent stages. For example, for $r = 0.05$ and $p = 0.5$. S for the 8×2 architecture is 4.8 and the S for the 16×1 architecture is 5.97. Although $S_{16\times1}$ is slightly larger than $S_{8\times2}$, in practice, the 8×2 architecture provides a better performance. If two configurations provide a similar S, it is safer, based on our experience, to choose the one with a smaller degree of parallelism. Generally, the memory access is the word-matching stage's performance bottleneck: the on-chip memory resources limit the number of hash functions working in parallel and the query sequence size supported, the off-chip memory limits the number of matches allowed each cycle.

The Mercury design uses a near-perfect hashing strategy to reduce the number of hash collisions (for 25 kbase query, it consumes 16 kbits on-chip memory and generates several hash collisions). However, based on our observations, when hash collision is reduced, duplicate w-mers become the dominating factor that hinders the performance of off-chip memory access. As the query size increases, the number of duplicate w-mers will be much larger than that of hash collision. As duplicate matches cannot be eliminated by any hash function, we apply a simpler method, bucket hashing, to reduce the number of hash collision and mitigate the on-chip memory space consumption. Our test with a 25 kbase query sequence randomly chosen from the human genome results in 222 hash collisions and 1975 duplicate w-mers using bucket hashing. In contrast, if we use a simple hash function with the same query dataset, it would produce 3461 collisions. Meanwhile, the on-chip memory consumption of our bucket hashing strategy is slightly bigger than 5 kbit. Although bucket hashing produces more collisions than the near-perfect hashing, its influence on off-chip table accesses is quite limited, when compared to that of duplicate w-mers. In addition, our on-chip quick lookup table consumes much smaller space than that of the Mercury (5 kbit for our QLT, while 16 kbit for Mercury's displacement matrix), which could avoid competing memory resources with the Bloom filter stage.

In summary, our design can provide better computational efficiency under low match rate conditions than the conventional Bloom filter architecture. The performance bottleneck lies in the memory access, which limits the throughput and parallelism for both the Mercury and our word-matching accelerator.

VI. Conclusion

In this paper, we have presented an FPGA-based reconfigurable 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. Different techniques are applied to optimize the performance of each substage. The comparison of the performance of our word-matching accelerator to that of NCBI BLASTN shows a speedup around one order of magnitude with only modest resource utilization.

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 architecture for Stage 2 of the BLASTN pipeline (ungapped extension) in order to further improve the overall application performance.

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.

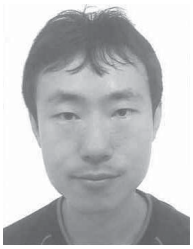
[15] M. Nourani and P. Katta, "Bloom filter accelerator for string matching," in *Proc. 16th Int. Conf. Comput. Commun. Netw.*, 2007, pp. 185–190.

[16] I. Moraru and D. G. Andersen, "Exact pattern matching with feed-forward Bloom filter," in *Proc. Workshop Algorithm Eng. Experim.*, 2011, pp. 1–12.

[17] *DRC Coprocessor Information* [Online]. Available: http://www.drccomputer.com

[18] Y. Chen, B. Schmidt, and D. L. Maskell, "A reconfigurable bloom filter architecture for BLASTN," in *Proc. ARCS 22nd Int. Conf. Archit. Comput. Syst.*, 2009, pp. 40–49.

[19] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Trans. Comput.*, vol. 46, no. 12, pp. 1378–1381, Dec. 1997.
[20] 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.
[21] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
[22] *BLAST Programs at NCBI* [Online]. Available: http://www.ncbi.nlm.nih.gov/BLAST/
[23] *Xilinx Virtex-7 Family* [Online]. Available: http://www.xilinx.com/product/silicon-devices/fpga/virtex-7/index.htm
[24] J. Buhler, "Mercury BLAST dictionaries: Analysis and performance measurement," Dept. Comput. Sci. Eng., Washington Univ., St. Louis, MO, Tech. Rep., 2007.



Yupeng Chen received the Bachelor's and Master's degrees in electrical engineering from the University of Electronic Science and Technology of China, Chengdu, China, in 2000 and 2004, respectively. He is currently pursuing the Ph.D. degree in computer engineering from Nanyang Technological University, Singapore.

His current research interests include high performance computing on field-programmable gate arrays, parallel algorithms, and architectures.

Bertil Schmidt (M'04–SM'07) is a Full Professor and the Chair for Parallel and Distributed Architectures with the University of Mainz, Mainz, Germany. Prior to that, he was a Faculty Member with Nanyang Technological University, Singapore, and the University of New South Wales, Sydney, Australia. His research group has designed a variety of algorithms and tools for bioinformatics mainly focusing on the analysis of large-scale sequence and short read datasets.

Prof. Schmidt was a recipient of the CUDA Research Center Award, CUDA Academic Partnership Award, CUDA Professor Partnership Award, and the Best Paper Award from the IEEE Application-Specific Systems, Architectures, and Processors in 2009. He serves as the champion for Bioinformatics and Computational Biology on gpucomputing.net.

Douglas L. Maskell (SM'03) received the B.Eng. and M.Eng.Sc. degrees in electrical and electronic engineering and the Ph.D. degree in computer systems engineering from James Cook University, Townsville, Australia, in 1980, 1985, and 1996, respectively.

He is currently an Associate Professor with the School of Computer Engineering, Nanyang Technological University, Singapore. His current research interests include the areas of embedded systems, reconfigurable computing, and algorithm acceleration for hybrid high-performance computing systems.