# PREPRINT – NCBI BLASTP on High Performance Reconfigurable Computing Systems

Atabak Mahram, Boston University
Martin C. Herbordt, Boston University

The BLAST sequence alignment program is a central application in Bioinformatics. The *de facto* standard version, NCBI BLAST, uses complex heuristics which make it challenging to simultaneously achieve both high performance and exact agreement. We propose a system that uses novel FPGA-based filters that reduce the input database by over 99.97% without loss of sensitivity. There are several contributions. First is design of the filters themselves, which perform two-hit seeding, exhaustive ungapped alignment, and exhaustive gapped alignments, respectively. Second is the coupling of the filters, especially the two-hit seeding and the ungapped alignment. Third is pipelining the filters in a single design, including maintaining load balancing as data are reduced by orders of magnitude at each stage. Fourth is the optimization required to maintain operating frequency for the resulting complex design. And finally there is system integration both in hardware (the Convey HC1-EX) and software (NCBI BLASTP). We present results for various usage scenarios and find complete agreement and a factor of nearly 5x speed-up over a fully parallel implementation of the reference code on a contemporaneous CPU. We believe that the resulting system is the leading per socket accelerated NCBI BLAST.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Styles—Heterogeneous (hybrid) systems; Pipeline processors*; J.3 [**Life and Medical Sciences**]: Biology and Genetics

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: FPGA-Based Coprocessors, High Performance Reconfigurable Computing, Bioinformatics, Biological Sequence Alignment, Application Acceleration

**ACM Reference Format:**
Atabak Mahram and Martin C. Herbordt, 2014. NCBI BLASTP on High Performance Reconfigurable Computing Systems. *ACM Trans. Reconfig. Technol. Syst.* TBD, TBD, Article 1 ( 2014), 21 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

## 1. INTRODUCTION

A fundamental insight underlying Bioinformatics is that biologically significant polymers such as proteins and DNA can be abstracted into sequences thereby allowing the use of approximate string matching (AM) to determine, e.g., how a newly identified protein is related to those previously analyzed, and how it has diverged through mutation. The motivation for achieving the highest possible performance for biological AM is motivated by at least two factors. First, biological databases are increasing in size at exponential rates far outstripping Moore's Law [Cochrane et al. 2011; Kahn 2011]. And second, AM is being used as a "subroutine" in ever more complex applications

This work was supported in part by the NIH through award #RR01-RR023168-01, by the NSF through award #CNS-1205593, by IBM through a Faculty Award, and facilitated by donations from Gidel and Altera Corporation through the Novo-G Forum program.
Author's addresses: A. Mahram, (Current address) Nvidia; M.C. Herbordt, Department of Electrical and Computer Engineering, Boston University.

such as multiple sequence alignment, genomics, and phylogenetics; in a single run of any one of these, BLAST may be “called” thousands to billions of times.

The most common approach to biological AM is based on Karlin-Altschul statistics [Karlin and Altschul 1990] which allows for insertions and deletions and scoring based on independent character replacement with a scoring matrix. Optimal methods are based on dynamic programming (DP), including the Smith-Waterman algorithm (SW), and have complexity $O(MN)$ where $M$ is the size of the query and $N$ the size of the database. Heuristics such as those used by the BLAST algorithm [Altschul et al. 1990] generally speed AM; these heuristics only rarely cause signficant matches to be missed (see, e.g., [Lam et al. 2008]), although for some applications it is exactly these distant matches that are critical. BLAST acceleration has therefore received much attention with FPGAs being successfully applied. Commercial solutions are currently available from TimeLogic (now part of Active Motif) [Time Logic Corp. 2013] and from Convey [Convey Computer Corporation 2013b]. There have also been several academic efforts [Herbordt et al. 2007; Jacob et al. 2008; Lavenier et al. 2006; Muriki et al. 2005; Park et al. 2009; Sotiriades and Dollas 2007; Xia et al. 2008]. GPU implementations described in published studies [Ling and Benkrid 2010; Liu et al. 2011; Vouzis and Sahinidis 2011] have so far achieved performance up to about 3x times that of a multicore CPU.

Of the many versions of BLAST, NCBI BLAST [NCBI 2013] has become a *de facto* standard. All BLASTs runs through several phases (overview below) and return some number of matches with respect to a statistical measure of likely significance. NCBI BLAST itself is a complex highly-optimized system, consisting of tens of thousands of lines of code and a large number of heuristics beyond those of the original algorithm. Creating an accelerated version that both matches the NCBI BLAST output and delivers significant acceleration is therefore challenging. A now commonly used method is prefiltering [Afratis et al. 2008; Herbordt et al. 2006]. The idea is to quickly reduce the size of the database to a small fraction, and then use the original NCBI BLAST code to process the query. Agreement is achieved by guaranteeing that the filter output is strictly more sensitive than the original code.

In CAAD BLAST (as we call our system) we combine three filter stages, corresponding to the three major phases of BLAST, with the original NCBI BLAST code. As these filters are pipelined there is a significant load balancing problem since the work per filter drops by $5\times$ to $10\times$ at both interfaces. Other challenges addressed are the design and implementation of the filters themselves and the logic necessary to couple them. The result is a highly complex system that requires coordination of dozens of I/O ports, a similar number of internal pipes, and synchronization and orchestration of all these streams. Adding to the difficulty are the heterogeneity of the data streams, the resulting long control paths, and the need for high chip utilization and operating frequency.

The primary result is a transparent FPGA-accelerated NCBI BLASTP, fully tested and validated on the Convey HC-1EX, that achieves both output identical to the original and a factor of $5\times$ to $11\times$ improvement in performance over a multithread optimized reference code running on a similar generation CPU. The FPGA implementation now takes just a few seconds and is limited by the remaining serial code and the time it takes to stream the database through the device.

The rest of this paper is organized as follows. In the next Section we give background in Biological AM, NCBI BLAST, and prospective target architectures. There follows an overview of CAAD BLAST and details of the Two-Hit Filter implementation. After that we describe how the filters are coupled and give details of the FPGA implementation and optimization. We conclude with system integration issues, results, and a discussion.

## 2. BACKGROUND

### 2.1. Basics of AM for Biological Sequences

An alignment of two sequences is a one-to-one correspondence between their characters, without reordering, but with the possibility of some number of insertions or deletions. In biological AM, an alignment score between two (sub)sequences is computed by combining the independently scored character matches, which themselves are determined *a priori* by biological significance. For large databases heuristic methods such as BLAST are popular. BLAST is based on an observation about the typical distribution of high-scoring character matches in the DP alignment tableau: there are relatively few overall, and only a small fraction are promising. This promising fraction is often recognizable as proximate line segments along the main diagonal.

The original BLAST algorithm [Altschul et al. 1990] has three phases: identifying short sequences (words) with high match scores, extending those matches, and merging proximate extensions. In the first phase (seeding), the word size $W$ is typically 3 for BLASTP and significance is determined using a scoring matrix and threshold score $T$ (default 11). Nowadays [Korf et al. 2003; NCBI 2013], the preferred method of seeding looks for diagonals (ungapped alignments) with two hits within a certain distance $A$ (default 40). In the second phase (extension), seeds are extended in both directions to form high-scoring segment pairs (HSPs). Extension stops when it ceases to be promising, i.e., when the drop off from the last maximum score exceeds a threshold $X$. An $Evalue$ (expected value) is computed from the raw alignment score and other parameters. Database sequences with a sufficiently good $Evalue$, as selected by default or by user, are reported. The third phase is nowadays often replaced by a *gapped* extension based on DP – the $O(NM)$ complexity is not onerous when the database size $N$ is a small fraction of the original.

### 2.2. NCBI BLAST Overview

NCBI BLAST adds a number of phases and options, which we sketch here. There are two options, ungapped and gapped. Ungapped alignment proceeds initially as just presented. In gapped alignment, extension and evaluation are triggered only when ungapped alignment satisfies the ungapped threshold. In gapped extension, the extension drop-off threshold $X$ also depends on gap-opening and gap-extension costs.

NCBI BLAST begins the evaluation phase by using an empirically determined cutoff score to keep only statistically significant HSPs. To improve sensitivity, a lower score is tolerated if there are multiple HSPs in a particular database sequence; the more HSPs, the lower the threshold. These multiple HSP scores are combined using Poisson and sum-of-scores methods for ungapped and gapped alignments, respectively. Finally, HSPs are organized into consistent groups and evaluated with the final threshold $Evalue$.

### 2.3. Target Systems

Our target system is the Convey HC-1ex (see Figure 1 and [Bakos 2010; Convey Computer Corporation 2013a]), a hybrid processor with a single four-core Intel CPU (Xeon L5408 2.13GHz) and four Xilinx FPGAs (Virtex-6 XC6VLX760 with user logic clocked at 125MHz). We also support an Altera-based system from Gidel. There is a total of 24GB host and coprocessor memory and the standard Intel I/O chipset; the system runs 64-bit Linux. Host and coprocessors share a virtual address space.

Convey refers to the FPGAs as Application Engines (AEs). The coprocessor also consists of interface logic, called the Application Engine Hub (AEH), which connects the coprocessors to host CPU. It is responsible for fetching and decoding instructions, executing scalar instructions, and routing host memory requests to coprocessor memories.

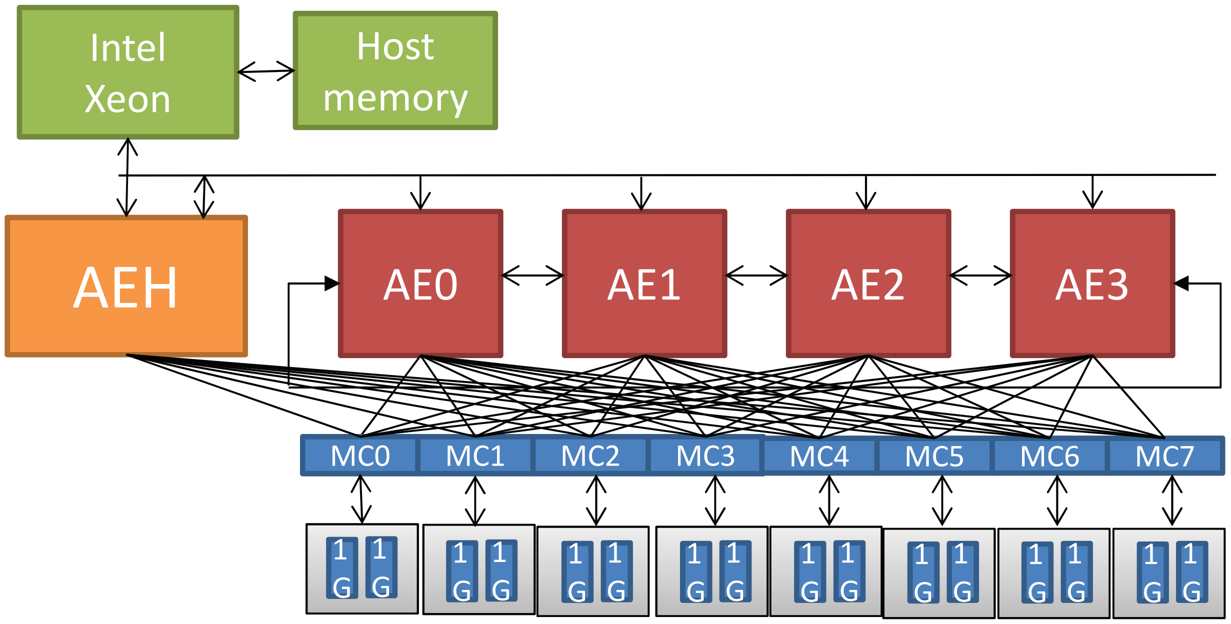

Fig. 1. Block diagram of the Convey HC-1ex.

The coprocessor system has 8 memory controllers that connect the AEH and the AEs to coprocessor memory modules through a full crossbar network. The memory controller subsystem can support up to 16 DDR2 memory channels. The memory subsystem can collectively support up to 8K parallel (independent) requests and 80GB/s total bandwidth.

## 3. CAAD BLASTP OVERVIEW

The design goal of CAAD BLAST is to use the FPGA accelerators to successively reduce the database (DB) without removing any matches that would be returned by NCBI BLAST. Figure 2 shows the overall structure. In the precompute module, the host uses logic from the NCBI code to compute the various parameters needed to determine *cutoffs* and $Evalue$s whose values are necessary to guarantee sensitivity and correct scoring. Three FPGA-accelerated filters replace analogous BLAST functions: The *Two-Hit Filter* (2HF) replaces a similar BLAST seeding routine; the *Exhaustive Ungapped Alignment Filter* (EUAF) replaces ungapped extension; and the *Exhaustive Gapped Alignment Filter* (SW for Smith-Waterman) replaces gapped extension.

Operation is as follows. After the parameters are determined, the next step is to filter the DB with the 2HF and generate a set of hints in the form of a bit vector. These, together with the original DB are then sent to the EUAF and a reduced database DB' is generated. This time only pointers to the relevant sequences are retained. SW is run on DB' to generate a further reduced database DB". Finally, DB" is formatted and sent to NCBI BLAST, together with the *original* parameters and query. The final run through the original NCBI BLAST code ensures that there are no false positives; this includes removing statistically significant sequences generated by CAAD BLAST but not by NCBI BLAST. So that the $Evalue$s match those that would be computed by the original code, we pass the original search space information.

Integration and correctness are described in detail elsewhere [Park et al. 2009]. Briefly, the overall logic is that each filter is either identical to (2HF) or more sensitive

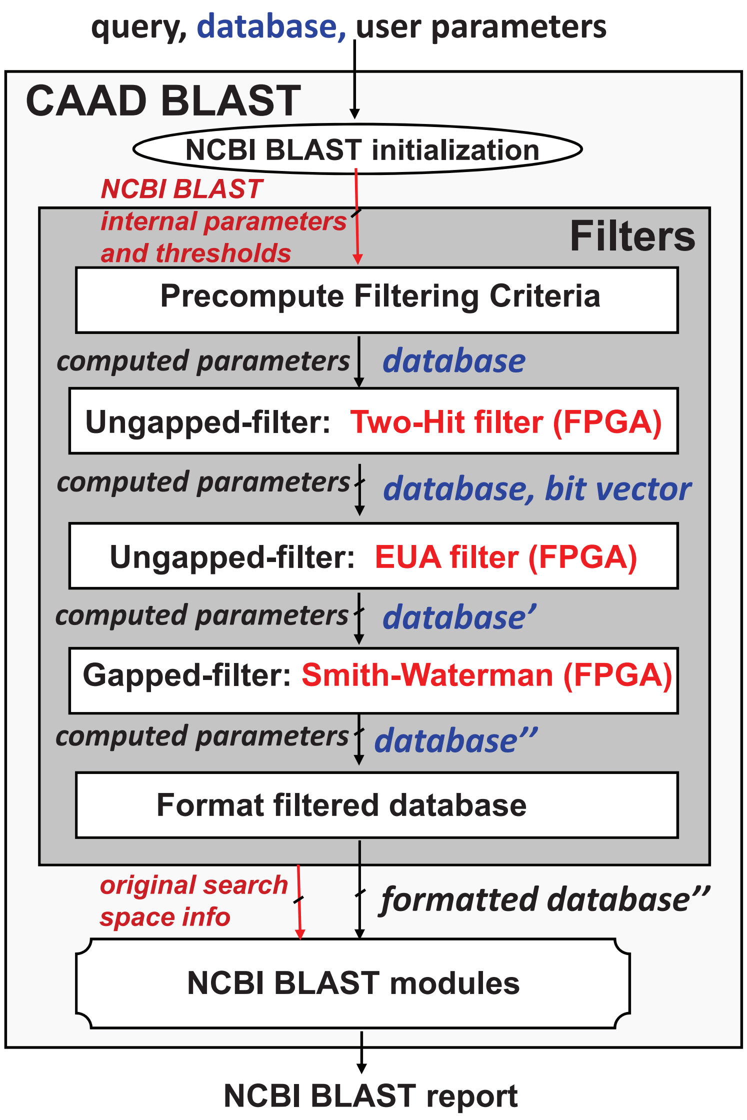
query, database, user parameters
CAAD BLAST
NCBI BLAST initialization
NCBI BLAST internal parameters and thresholds
Filters
Precompute Filtering Criteria
computed parameters
database
Ungapped-filter: Two-Hit filter (FPGA)
computed parameters
database, bit vector
Ungapped-filter: EUA filter (FPGA)
computed parameters
database'
Gapped-filter: Smith-Waterman (FPGA)
computed parameters
database''
Format filtered database
original search space info
formatted database''
NCBI BLAST modules
NCBI BLAST report

Fig. 2. High-level design of CAAD BLAST.

than its NCBI BLAST counterpart (EUAF and SW). In the latter case, the internal parameters in CAAD BLAST are computed dynamically from the NCBI BLAST analogs with an empirically derived function; please see Section 7 for details.

The filters themselves all work on the same principle. Each holds a copy of the query as the database streams through it. The filter size is related to the query size. Generally the filter uses only a fraction of the chip area and so can be replicated some number of times. If the query is very large, then the filter is *folded*: it still operates correctly, but with a slowdown proportional to the number of folds. Each filter thus runs in $O(N)$, assuming that the query sequence is a small multiple of what can fit on a current FPGA, a characteristic of almost all proteins. Large protein databases such

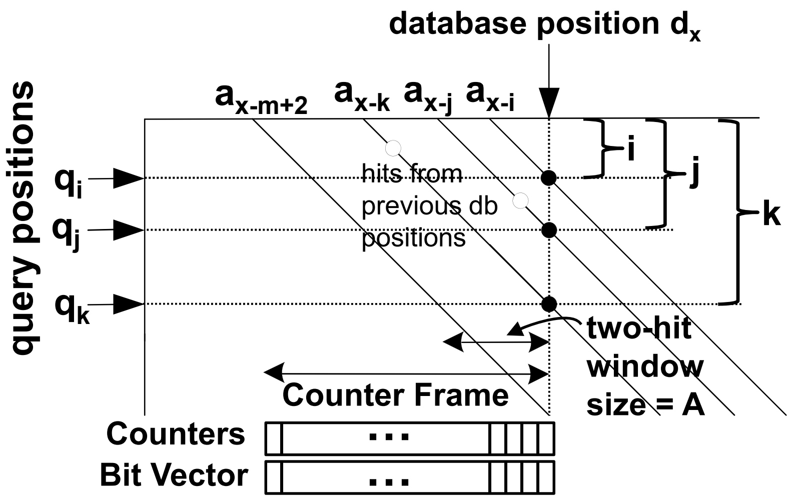

Fig. 3. The Two-Hit filter is processing the $x$th 3-mer in the database sequence. There are three hits. The hit on alignment $a_{x-j}$ is within $A$ of a previous hit, and so is part of a two-hit event. This is determined by comparison with the corresponding Counter value; its bit in the Bit Vector will be set.

as NR currently have over 5GB of data: off-the-shelf FPGA plug-in boards of the type used for computation can hold this in local memory and stream it through the FPGA in a few seconds.

Performance benefit is derived from two sources. The first is that during filtering the DB is processed in a stream with one data element completed per cycle. The second is that most FPGAs can support the processing of multiple streams in parallel. In CAAD BLAST, as is common with accelerated BLASTs, the DB is partitioned to match the number of streams. Since the proteins in the DB are independent, the method of the partition itself is arbitrary: the only constraints are similarity of size and not splitting proteins across partitions.

## 4. TWO-HIT FILTER

The design of the 2HF is generally similar to that used by Mercury BLAST in the seeding pass [Jacob et al. 2008; Krishnamurthy et al. 2007]. There are several algorithmic and implementation differences, however, which have two significant consequences:

— The 2HF does not use heuristics and so has exact agreement with the two-hit seeding algorithm used in NCBI BLAST.
— The 2HF is compact. For our reference design, each filter uses no more than 1% of any chip resource for average sized queries.

In the rest of this section we present the algorithm, some implementation details, and experimental results.

### 4.1. Algorithm Overview

Figure 3 shows the database on the horizontal axis and the query on the vertical axis. Positions of each 3-mer are referred to as $d_x$ for the database and $q_y$ for the query. Each of the $N - M - 2$ possible ungapped alignments between the database and the query is represented by a diagonal; we refer to each diagonal (alignment) as $a_i$. The output of the 2HF is a bit vector where each bit $b_i$ corresponds to an alignment $a_i$ and tells whether or not $a_i$ has passed the filter. That is, an alignment $a_i$ passes the

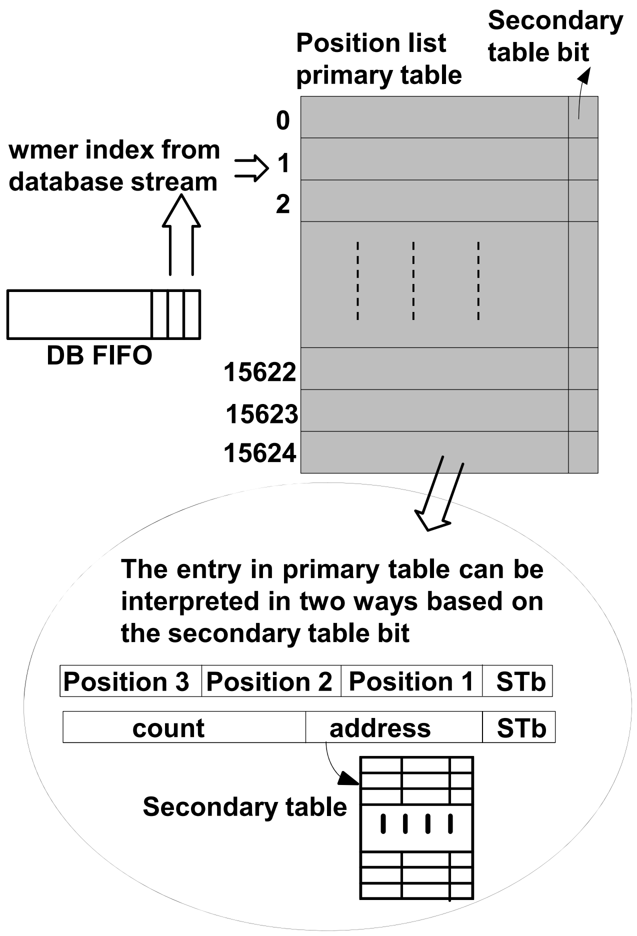
Position list primary table
Secondary table bit
wmer index from database stream
0
1
2
15622
15623
15624
DB FIFO
The entry in primary table can be interpreted in two ways based on the secondary table bit
Position 3
Position 2
Position 1
STb
count
address
STb
Secondary table

Fig. 4. Query 3-mer Position Table.

filter if anywhere on the diagonal there are two hits within the distance threshold $A$ (typically 40). If yes, then $b_i$ is set, otherwise it remains cleared. For each alignment, the corresponding counter in the Counter Frame holds the position of its most recent hit, if any.

The primary data structure is the Query 3-mer Position Table shown in Figure 4. The Position Table stores, for each possible 3-mer, say `WWW`, the positions of all of the 3-mers in the query that exceed the match threshold (typically 11) for that 3-mer. The Position Table has two parts, the primary and the secondary tables. The primary table has an entry for each of the 15625 (25x25x25) possible 3-mers for a typical 25 character alphabet. For any 3-mer, if there are 3 or fewer occurrences in the query, then its primary table entry holds all of those positions. If there are more than 3 occurrences, then the primary table entry contains the number of occurrences and the address in the secondary table where entries for those positions begin. A status bit indicates the record type.

We now give an overview of the operation of the 2HF. On iteration $x$, database 3-mer $d_x$ indexes the Position Table. The query positions where matches occur, if any, are retrieved. Figure 3 shows three hits, at query positions $q_i$, $q_j$, and $q_k$. These correspond,

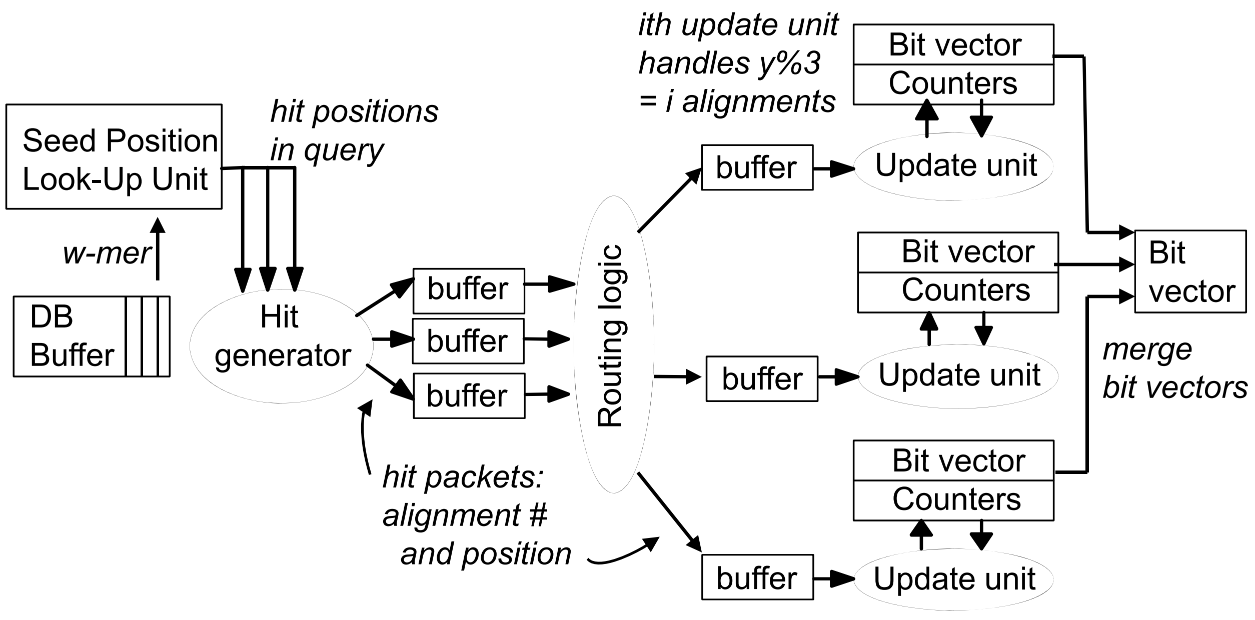

Fig. 5. Block diagram of the Two-Hit Filter.

respectively to the $i$th position on alignment $a_{x-i}$, the $j$th position on alignment $a_{x-j}$, and the $k$th position on alignment $a_{x-k}$.

This hit information is then used to determine whether another hit has occurred on any of these diagonals within the previous $A$ positions (as shown in Figure 3 for alignment $a_{x-j}$). The method is to use a circular list (or *frame*) of $M-2$ counters, one for each alignment where there could be match on the current iteration. For example, for a hit in alignment $a_{x-j}$, the counter $(M-3-j)$ for that alignment is read, compared with $j$, and updated. If the difference between $j$ and the previous value of the counter is less than $A$, then this indicates a two-hit hit event for alignment $a_{x-j}$ and bit $b_{x-j}$ in the bit vector is set. The counters for $a_{x-k}$ and $a_{x-i}$ are also updated.

Critical for keeping the filter logic compact is for the logic required to track each diagonal to be small. In particular, for the 2HF only a single counter plus comparison and update logic is required. That is, only the position of the last hit needs to be saved. This may be non-obvious since on any iteration, any of the last $M-2$ alignments can be affected. For each alignment, however, advancement is monotonic: a hit on a later iteration will never be further back on the diagonal than the previous one.

### 4.2. Implementation Details

*Overall.* The overall structure of the 2HF is given in Figure 5. The goal is to process the database at streaming rate. To support this, the 2HF processes three hits at a time. Both primary and secondary tables are structured to enable the fetch of three query positions per cycle. If the secondary table must be accessed, then streaming rate cannot be maintained and the database stream may need to be throttled. Much of this is prevented, however, by using a judiciously selected FIFO size for the input stream and adjusting the stream rate accordingly. Most performance loss occurs when there are a large number of occurrences $O$ of a 3-mer in the query. The number of cycles per database position, given that there are $O$ occurrences of the 3-mer at that position, is roughly equal to max(1,$\lceil O/3 \rceil$).

*Query 3-mer Position Table.* The primary table has 15625 entries. Each entry has 3 query positions plus 1 status bit. For queries of size less than 1024, this information fits in a single 32-bit word. The secondary table is negligible for small queries, but grows to have size similar to the primary table as query size approaches 2K. The secondary table is again organized to have 3 positions per entry. The FPGA in the reference design has 864 M9K (256x36b) BRAMs and 48 M144K (4Kx32b) BRAMs each of which is dual ported. These support 30-40 Position Tables for small queries and 16 for queries of size 2K.

*Query Position Look-Up Unit.* Input: 3-mer from the current head of the database stream.
Outputs: Positions $q_m$ of that 3-mer in the query (three at a time).
Operation: Translates the 3-mer into the entry position. Fetches corresponding positions, if any. If secondary table look-up is required, fetches 3-mers from there until done.

*Hit-Generator.* Inputs: The current position $d_i$ of the database stream and the query positions $q_m$ for up to three hits at a time from the Query Position Fetch Unit.
Outputs: For each hit, a "hit packet" containing the alignment $a_i$ and the position $p_i$ on that alignment.
Operation: The Hit Generator performs the necessary translation.

*Update Units.* Inputs: Hit information ($a_i$,$p_i$) for up to three hits at a time.
Outputs: Update of the frame counters corresponding to the $a_i$ with the positions of the new hits $p_i$. Update the bit vector.
Operation: Determine for each $a_i$ whether two hits within $A$ positions have occurred. This requires reading, comparing, and updating the frame counters.

*Routing Logic.* To support three parallel updates, the bit vector is interleaved (mod 3). Since multiple hits can be routed to any hit generator on any cycle, hit packets may need to be buffered.

Note that while we have used fixed numbers in the presentation, e.g., 3-mer rather than w-mer and an alphabet of 25 characters, the design supports all standard parameter choices.

### 4.3. Design Decisions and Experimental Results

For the 2HF, performance depends on the number of filters which, in turn, depends on the FPGA resources needed for each filter instance. The logic required is trivial, consisting of less than 1% of that available on the reference FPGA. The on-chip memory required, on the other hand, is the critical resource. Table I shows the number of 2HFs that can be instantiated on an Altera Stratix-III 260E for a selection of sequences from the NR database. Results scale to other FPGAs.

Table I. Various 2HF measures for selected sequences from the NR database.

| Query Size | # of Two-Hit Filters | # of Hits per DB char. | # of excess cycles per DB char. | percent removed (0s) |
|---|---|---|---|---|
| 81 | 38 | 0.064 | 0.0002 | 99.7% |
| 217 | 35 | 0.206 | 0.0100 | 99.2% |
| 490 | 28 | 0.567 | 0.0524 | 98.4% |
| 838 | 25 | 0.891 | 0.2203 | 98.3% |
| 1204 | 21 | 1.244 | 0.3062 | 98.0% |
| 2205 | 14 | 2.570 | 0.8790 | 97.3% |

The primary design decision therefore has to do with the structure of the Position Table, in particular the number of positions per entry in the primary table. For most queries (size $< 2K$) this number (3) falls out immediately from the convenience of packing that number of 11-bit addresses into a single 36-bit word. For larger queries, different configurations are possible, e.g., packing two addresses into one 36-bit word or five into a 72-bit word. The optimization is to trade off table size for number of filters. That is, by having more entries in the primary table, some accesses to the secondary table can be avoided. But the larger table size allows fewer filters to be fit on the FPGA and so fewer database streams to be processed in parallel.

The third and fourth columns in Table I give an indication of this trade off. The number of hits per database character (3-mer) is independent of the structure of the Position Table. For queries up to size 2K, the expected number of hits per position is only slightly more than 1; having three position per entry allows the primary table to account for most 3-mers. For larger query sizes, we have only two addresses and the secondary table must be accessed frequently. The fourth column illustrates this: it shows the number of excess cycles per database character; i.e., the number of extra cycles needed due to accessing the secondary table. For small queries, there are virtually no excess cycles, but for the 2205 query, nearly half the cycles are due to secondary table accesses.

The effectiveness of the 2HF is shown in the rightmost column of Table I. For all but the largest queries, more than 97% of ungapped alignments are tagged for skipping.

## 5. COUPLING THE FILTERS

The EUAF is based on the TreeBLAST scheme described in [Herbordt et al. 2007]. The issue described in this Section is the coupling of the 2HF and the EUAF. The goal is for the EUAF to be able to skip alignments tagged with a 0 by the 2HF. The problem is that the EUAF is already operating at streaming rate. The solution uses two essential properties of the EUAF: it can be folded to trade off performance for area and it can be replicated to process DB sequences in parallel. In an EUAF-only design one or the other might be used for large and small sequences, respectively [Park et al. 2009]. Here it turns out that it is advantageous to use both simultaneously.

*Ideal Skipping.*. The idea behind ideal skipping is, on every cycle, to look ahead in the bit vector to find the next “one” (corresponding to the next alignment to be examined) and then slide the database the correct number of positions. With ideal skipping, the EUAF takes only the number of cycles equal to the number of ones in the bit vector. The additional hardware required, however, is complex. For both the bit vector and the database stream, they must be able, on each cycle, to slide any number of positions up to the maximum number supported. This, in turn, requires that each register in the stream buffer have a multiplexor (MUX) that is large enough for every possible number of positions that could be skipped. It also requires complex routing logic. As a result, support for even a small range of choices makes the logic for general skipping more expensive than the original EUA filter.

*Constant Skipping.*. A better solution is to limit the number of positions that can be skipped to a single number $S$ that is determined experimentally. That is, the database stream skips either $S$ positions or none (and advances either $S$ positions or 1). If there is a sequence of $S$ or more 0s, then $S$ skipping is used, otherwise it is not. This scheme greatly simplifies the MUX logic, but has two drawbacks. The first is that only sequences of 0s of length $S$ or greater can be taken advantage of. All shorter sequences of 0s are useless. This indicates a small $S$ so that most sequences of 0s can be used. The second is that the maximum skip is also limited indicating a large $S$. The optimal $S$ is query dependent but generally follows the query length. A larger $S$ works better with smaller queries as more of their alignments are filtered.

*Folded Skipping.*. Folded skipping is a substantial improvement. Recall that trees can be folded with the addition of a trivial amount of logic. Also that a tree that is folded to $1/F$ its original size requires only $1/F$ the logic of the original, but requires $F$ cycles per alignment rather than 1.

The idea behind folded skipping is to process unfiltered alignments in $F$ cycles (as before), but to process the others in only 1 cycle. The control for this scheme is thus extremely simple: there is no need for complex look-ahead or routing logic. Rather, if the bit-vector value of an alignment is a 0, simply shift the database stream; if the value is a 1, then continue processing the alignment for a delay of another $F-1$ cycles. The hardware cost is a slight increase in control complexity; no other additional logic is needed.

The performance benefit of folded skipping can be demonstrated as follows. Assume that the bit vector for a size $N$ database has $O$ ones. Without skipping, an $F$-folded tree requires roughly $F \times N$ cycles to process the database. With skipping, the number of cycles is $N + O \times (F-1)$. If $F$ is 16 and $N/O$ is 20, then the speed-up is greater than $9\times$. This speed-up is independent of the distribution of 1s.

*Variable Folded Skipping.*. The drawback of folded skipping is that while 0s are processed F$\times$ as fast as 1s, they still take a cycle per character. Since the fraction of 0s $(Z)$ is generally 98%-99% of the stream, processing these null alignments still takes $\frac{Z}{Z+(1-Z)\times F}$ of the cycles, or 75% to 85% for almost all query sequences.

Variable folded skipping works as follows. During the $F$ clock cycles required by the folded skipping mechanism when TreeBLAST is processing an alignment, a seed lookup module continues streaming the database until it finds the next unfiltered diagonal. The seed lookup module finds the next unfiltered alignment by implementing the constant skip mechanism with S=16. That is, each clock cycle it either skips one character or 16 consecutive characters until it finds the next unfiltered alignment. With $F = 16$, $16 \times 16 = 256$ filtered diagonals (0s) can be skipped. The performance gain is dramatic: only a small number of cycles are spent processing 0s, improving performance of the EUAF by more than $4\times$. Note that variable folded skipping addresses another significant issue with the EUAF filter: the need to process artifactual null alignments that are inserted as padding during start-up and tear-down of each database sequence.

## 6. FPGA IMPLEMENTATION

### 6.1. Pipelining the Filters

In early versions of this work we coupled the filters by running them in separate phases and configuring the FPGA to a single filter during any phase. Intermediate results such as the bit vector generated by the 2HF were stored in off-chip memory. This approach achieves high performance by enabling a large number of database streams to be processed in parallel (30 or more for a Stratix-III), but also requires reconfiguration between phases. Both of these characteristics, high number of streams and reconfiguration, have become problematic with new generation devices and system configurations. First, in CAAD BLASTP the number of streams supported increases proportionally with FPGA feature count; the number of streams necessary to fully utilize the resources of Virtex-6/Stratix-IV generation FPGAs may now exceed the bandwidth typically provided to off-chip memory. And second, reconfiguration times continue to grow longer in parallel with FPGA size. When coupled with improved filter performance, we find that reconfiguration time now dominates the execution time. Our approach now is to pipeline the three filters. This is a significant load balancing problem since the work per filter drops by $5\times -10\times$ at both of the interfaces, but must be achieved with little resource overhead.

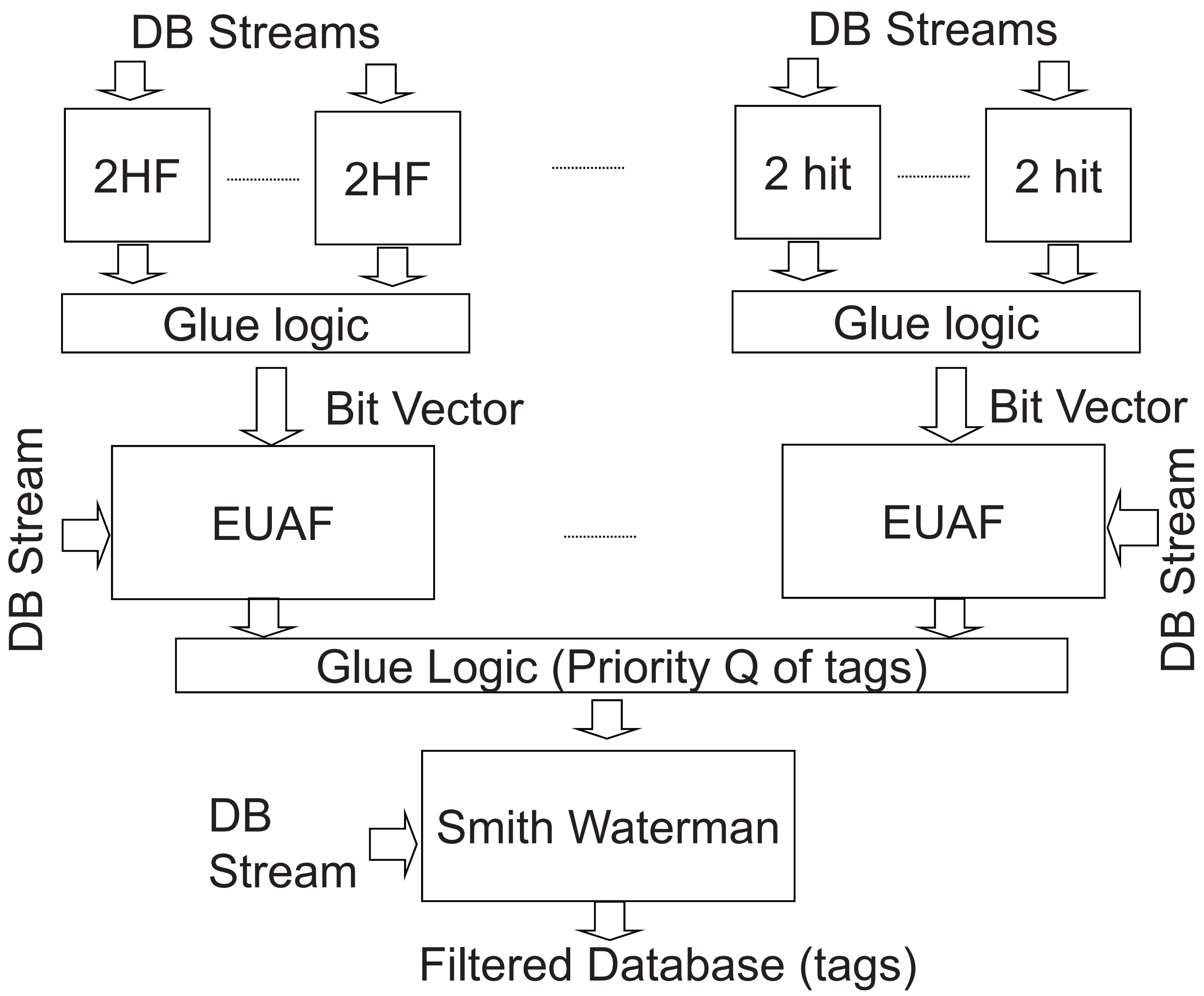

Fig. 6. Block diagram of CAAD BLAST pipelined filters.

Figure 6 shows the overall scheme: there are number of a single filter *banks*. In each bank, parallel database streams feed the 2HFs, which in turn send the 0/1 stream to an EUAF. A copy of the database is streamed to the EUAF where it is coupled with the 0/1 stream. This structure is then replicated some number of times depending on the size of the query and of the FPGA. In the final stage, the highest scoring database sequences from all of the banks are processed with a single SW module.

Speed matching between 2HF and EUAF stages is accomplished as follows. The EUAF processes data (a single sequence) from a single 2HF at a time. Processed sequences from the other 2HFs in the bank are buffered. Through the mechanism described in the previous Subsection, the EUAF is capable of consuming 3 to 5 characters per cycle. That is, data from buffered filtered sequences are transferred to the EUAF $F$ characters at a time (currently $F = 16$). After processing the data of one 2HF, the EUAF starts working on the next sequence from the next 2HF. In order to load balance,

| Query size Fraction 1s | Ratio of two-hit to EUA filters | EUA Filter: Percent idle cycles |
|---|---|---|
| 256 | 4 | 13 |
| 0.008 | ***5*** | ***1.5*** |
| | 6 | 0 |
| 512 | 2 | 14 |
| 0.016 | ***3*** | ***0.14*** |
| 1024 | 2 | 13 |
| 0.020 | ***3*** | ***0.06*** |
| 2048 | 2 | 12 |
| 0.027 | ***3*** | ***0.03*** |

Fig. 7. Balance between two-hit and EUA filters. The number of 1s generated by the two-hit filter is shown together with the query size. Optimal ratio for each query size range is shown in bold.

the database sequences are sorted based by length and multiplexed among multiple 2HFs. As a result the time required to process successive sequences is nearly equal.

Coupling with the SW filter is accomplished as follows. For each database sequence, the EUAF compares the maximum score generated with a constant threshold. If this score is larger than the threshold, the EUAF writes the address of the sequence to a FIFO. The SW unit reads these addresses, streams the subject sequences, and calculates the maximum scores.

## 6.2. Replicating and Balancing the Components

We find the optimal number 2HFs per EUAF by measuring the fraction idle cycles in the EUAF as a function of number of 2HFs and query size. We add 2HFs to each filter bank until almost saturated. The results are shown in Figure 7: 3 to 5 2HFs per EUAF is optimal.

The second column of Figure 8 shows the reduction in database size following the EUAF: This is at least 97% for most query sequences. The SW module can therefore be compacted substantially through folding and still obtain adequate performance. The optimally folded SW consumes characters of the reduced database db at the same rate that characters of the original database db are consumed by the two-hit filters. The raw results are shown in Figure 8. When integrated into the overall system, the number of folds is either 8 or 16 (see Figure 9).

From the preceding discussion we see that a speed matched bank of filters contains from 3 to 5 two-hit filters and 1 EUAF folded to effect $16\times$ replication. A single SW module is shared by all of the filter banks and folded as just described. The number of filter banks themselves that can fit on an FPGA is a function of query size and FPGA resources. Figure 9 shows the results for the Xilinx Virtex-6 XC6VLX7601. Depending on query size, 5, 4, 3, or 1 filter banks can fit for a total of 25, 16, 9, or 3 input streams. Queries of size greater than 2K are run on the host.

## 6.3. Optimization

The initial synthesis returned an unacceptably poor operating frequency, which was not ameliorated by reducing the size of the design until only a small fraction of the potential chip capacity was in use. We have solved this problem through two mecha-

| Query size | Reduction db to db' | Number of Folds for SW Filter (Virtex 6) | Number of Folds for SW Filter (Stratix IV) |
|---|---|---|---|
| 256 | 0.01 | 7 | 4 |
| 512 | 0.03 | 4 | 2 |
| 1024 | 0.05 | 4 | 2 |
| 2048 | 0.07 | 5 | 2 |

Fig. 8. Shown is the average fraction reduction of the database after the EUA filter (and before the SW filter) and the resulting optimal number of folds in the SW filter.

| **Query Size Range # of Filter Banks Total 2-Hit Streams** | **Logic Utilization** | | |
|---|---|---|---|
| | **Component** | **LUTs** | **BRAMs/FIFOs** |
| **Qs<256 Reps=5 2hSt=25** | 5 2HFs | 9050 | 124 |
| | 1 EUAF (16 Folded) | 5365 | 8 |
| | 1 SW (8 Folded) | 7179 | 27 |
| | Total (5 Reps) | 79254 | 687 |
| **Qs<512 Reps=4 2hSt=16** | 4 2HFs | 7355 | 124 |
| | 1 EUAF (16 Folded) | 8747 | 16 |
| | 1 SW (8 Folded) | 14223 | 43 |
| | Total (4 Reps) | 78631 | 603 |
| **Qs<1024 Reps=3 2hSt=9** | 3 2HFs | 6115 | 113 |
| | 1 EUA (16 Folded) | 15384 | 32 |
| | 1 SW (8 Folded) | 28215 | 75 |
| | Total (3 Reps) | 92712 | 510 |
| **Qs<2048 Reps=1 2hSt=3** | 3 2HFs | 6180 | 119 |
| | 1 EUA (16 Folded) | 30075 | 64 |
| | 1 SW (8 Folded) | 56246 | 139 |
| | Total (1 Rep) | 92501 | 322 |
| **Total Available (Virtex VI)** | | 474240 | 720 |

Fig. 9. Per component resource utilization for the Xilinx Virtex-6 and the number of replications for the bank configurations shown for each query size.

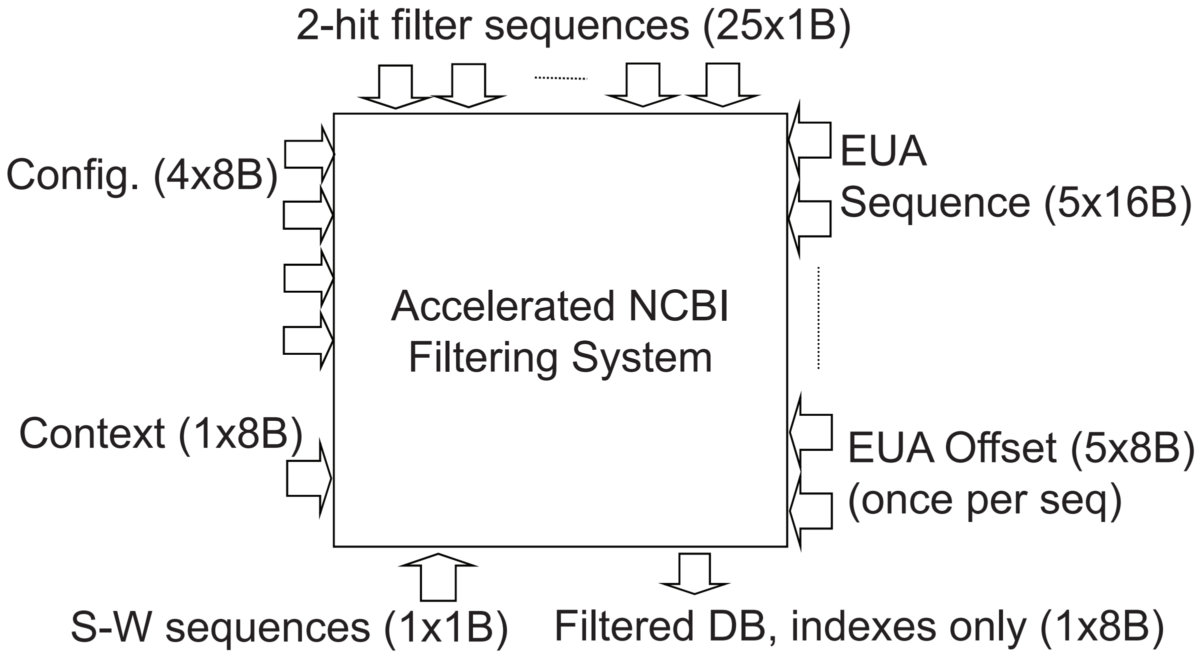
2-hit filter sequences (25x1B)
Config. (4x8B)
EUA Sequence (5x16B)
Accelerated NCBI Filtering System
Context (1x8B)
EUA Offset (5x8B) (once per seq)
S-W sequences (1x1B)
Filtered DB, indexes only (1x8B)

Fig. 10. Accelerated BLAST external interfaces for query size $< 256$. Number of streams and width of each type are shown.

nisms: by floor planning modules with respect to BRAMs and by handling fanout and length of the communication channels.

*6.3.1. RTL-Level Logic Optimizations.* There are two problems that need to be dealt with through RTL-level logic: mapping function I/O to physical I/O and reducing path delay. These are both handled primarily through the creation of three modular communication interfaces: Simple FIFO, Jump FIFO, and a direct register-based interface. Using these we can place each core anywhere on the FPGA and keep its communication off of the critical path by simply specifying an appropriate number of pipeline stages. Other optimizations include replicating registers to reduce fanout and eliminating the reset circuit as much as possible.

The Simple FIFO interface serves as our flexible general purpose inter-module communication mechanism and is used especially to foster module independence and avoid the creation of long paths. Prefetching enables correctness in the presence of back-pressure signals. Figure 10 shows the external I/O interfaces for the Accelerated BLAST configuration that supports sequences $< 256$ characters. Note that there are 26 x 1B streams and 5 x 16B streams operating continuously and a number of others that are used for initialization, data offload, and synchronization. These must be mapped to the physical I/O provided by the Convey HC-1ex: the 16 x 4B memory channels that can operate independently at over 300MHz. The mechanism we use is a *Jump FIFO* interface, a generalization of the Simple FIFO in that it communicates with external memory at a specified address. The Jump FIFOs are mapped to the Convey physical memory interface through the Convey memory crossbar module which routes memory transactions to the correct memory interface.

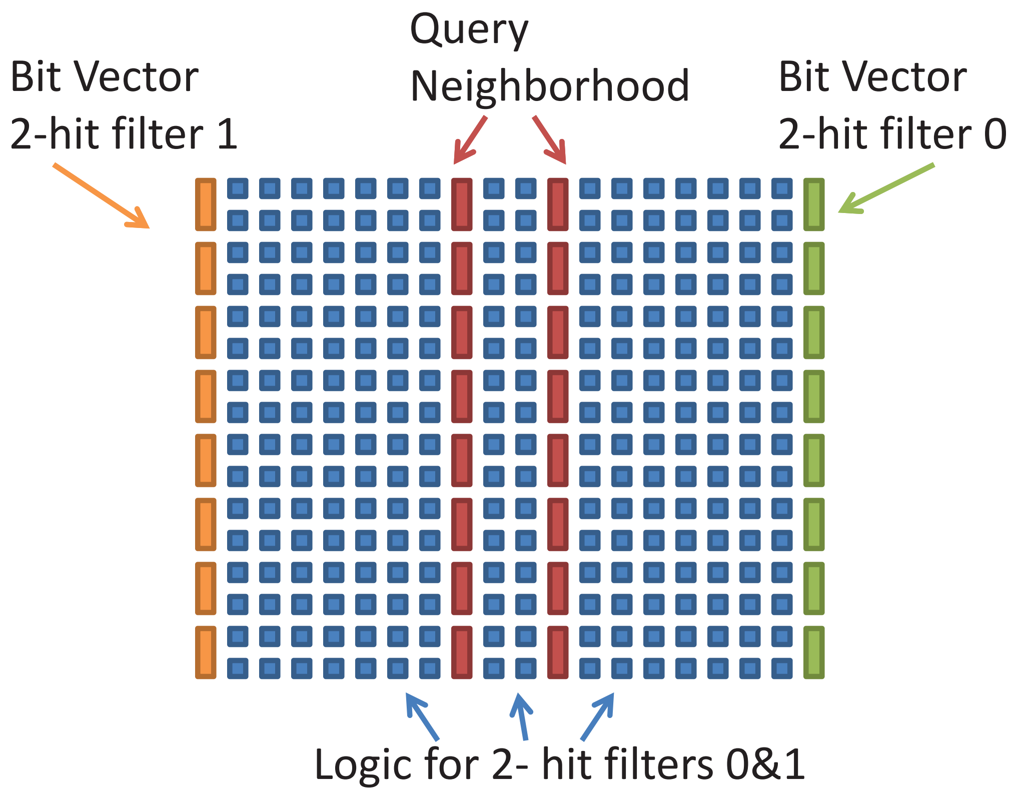
Query Neighborhood
Bit Vector 2-hit filter 1
Bit Vector 2-hit filter 0
Logic for 2- hit filters 0&1

Fig. 11. 2HF after floor planning. Blue cells indicate logic and other colors represent BRAMs.

*6.3.2. Floor Planning.* We apply floor planning in two layers. The first is internally to the two-hit filters, the second is for the higher level modules consisting of the two-hit filters that feed individual EUA filters. We found it sufficient to map BRAMs to particular modules and let the synthesis tools continue handling the logic placement.

While the two-hit filters each require only a small amount of area, their logic is complex and, more significantly, does not lend itself to pipelining. That is, pipelining stages would increase the time required to process each character, violating the basic design contraint: flowing the database through the FPGA at streaming rate of one character per cycle. The most critical path is the lookup of database WMERs in the query (see Figure 4). In the “fast” case there are three or fewer matches in the query. In the “slow” case, there are more and a secondary table must be accessed [Mahram and Herbordt 2010]. Each fetched entry must be processed in one clock cycle meaning that a newly computed address needs to be issued to the position list. As a result, the addressing circuit contains a combinational path that starts with the output of the position list and continues to the address input of the same position list.

The FPGA consists of a pool of CLBs and Block RAMS as shown in Figure 11. We number the BRAM columns from the left from 0 to 11. Of these, 4-7 are used by the interface logic and the API leaving 0-3 and 8-11. To floor plan the two-hit filters, we place the BRAMs for the position lists in a square, minimizing the path length as shown in Figure 11. At the next level, the EUA filter BRAMs are placed as close as possible to those of the two-hit filter (see Figure 12).

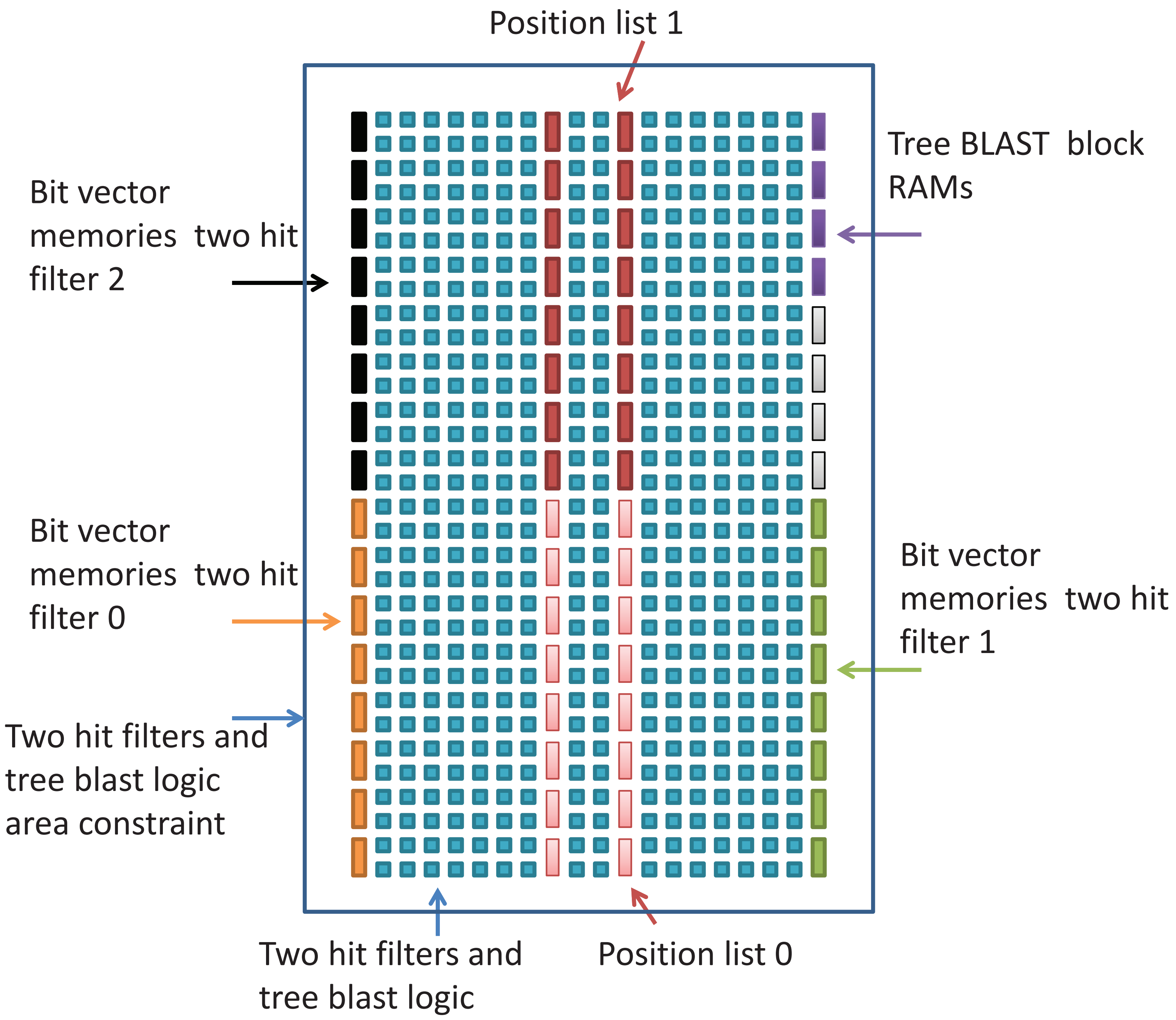
Position list 1
Tree BLAST block RAMs
Bit vector memories two hit filter 2
Bit vector memories two hit filter 0
Bit vector memories two hit filter 1
Two hit filters and tree blast logic area constraint
Two hit filters and tree blast logic
Position list 0

Fig. 12. Module after floor planning.

Table II. Percentage of the orginal 15.6M sequences of the NR database remaining after the EUA and SW filters, respectively.

| Test | DB' reduction % | DB" reduction % |
|---|---|---|
| 1 | 0.13 | 0.035 |
| 2 | 2.15 | 0.034 |
| 3 | 0.02 | 0.014 |
| 4 | 1.28 | 0.034 |

Table III. Various tests of reference and accelerated BLAST for queries up to 256 characters. "FPGA Only" refers to the accelerated part of the computation, i.e., the time to execute the filters. All times are in seconds.

| Test | Ref. Time | 1 FPGA Filter Only | 4 FPGAs Filter Only | Post-filter Search | Post-filter Traceback | 1 FPGA Total | 4 FPGAs Total | 1 FPGA Speedup | 4 FPGAs Speedup | acc % |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 46.5 | 7.1 | 1.9 | 1.5 | 1.9 | 10.5 | 5.3 | 4.4x | 8.8x | 98.4% |
| 2 | 45.6 | 10.5 | 2.9 | 1.5 | 1.7 | 13.7 | 6.1 | 3.2x | 7.5x | 100% |
| 3 | 48.9 | 7.3 | 2.0 | 1.2 | 1.3 | 9.8 | 4.5 | 5.0x | 10.9x | 96.4% |
| 4 | 47.0 | 8.1 | 2.2 | 1.4 | 0.4 | 9.9 | 4.0 | 4.7x | 11.7x | 100% |

## 7. INTEGRATION AND RESULTS

For these tests we use the NR protein database with 15.4M sequences and 5.4G characters. The hardware configuration is as described in Section 2.3. The reference configuration is the Convey processor without the AEs. We chose this reference processor because it is of the same technical generation as the FPGAs in the system. All the reference and accelerated tests were done with the latest NCBI BLAST (BLAST+ 2.2.28) with the -num_threads 4 option; this forces maximum useful parallelism for both the reference code and the CPU part of the accelerated code. In NCBI BLAST, the traceback code which generates the actual alignments is currently not threaded and so is serial in both reference and accelerated tests.

NCBI BLAST provides a wide range of user options that vary such quantities as internal thresholds and the quantity of results provided. The internal thresholds control sensitivity and thus the amount of work to be done. Varying them has comparable effect on both reference and accelerated execution. CAAD BLAST and NCBI BLAST are not identical, however: CAAD BLAST executes exhaustive ungapped and gapped alignments while NCBI BLAST executes gapped and ungapped extensions with heuristics. In order to guarantee no false negatives it may therefore be necessary to increase the sensitivity (lower the threshold) in CAAD BLAST. Note that as long as all false negatives are eliminated this does not change the overall output: the final run of NCBI BLAST still uses the user-specified thresholds and eliminates false positives.[1] In contrast to the sensitiviy parameters, those for output affect primarily the CPU-only part of the accelerated code. The default is to return the top 500 sequences of any possible statistical significance. Given that the traceback code is serial (and Amdahl's Law), permissive output has a disproportionate detrimental effect on the performance of CAAD BLAST.

We run four tests varying the following NCBI BLAST parameters: ungapped extension threshold (for the CAAD BLAST EUAF threshold), Evalue, and number of match sequences returned. We have also experimented with the gapped extension threshold (for CAAD BLAST SW threshold): as expected, no increase in sensitivity is required here so CAAD BLAST always uses this directly. Results are summarized in Tables II and III. We first note the general effectiveness of the filtering mechanisms: depending on thresholds, the EUAF reduces the original database from 97% to 99.98% while SW reduces it by from 99.97% to 99.99%.

[1]These are actually true positives, just not found by NCBI BLAST.

Table IV. Tests 2 and 4 (see text) of reference and accelerated BLAST for *all* queries. "FPGA Only" refers to the accelerated part of the computation, i.e., the time to execute the filters. All times are in seconds.

| Test | Ref. Time | 1 FPGA Filter Only | 4 FPGAs Filter Only | Post-filter Search | Post-filter Traceback | 1 FPGA Total | 4 FPGAs Total | 1 FPGA Speedup | 4 FPGAs Speedup | acc % |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 78.5 | 12.6 | 3.4 | 2.4 | 0.99 | 16.0 | 6.8 | 4.9x | 11.5x | 99.99 |
| 4 | 68.2 | 11.2 | 3.0 | 2.2 | 0.80 | 14.2 | 6.0 | 4.8x | 11.4x | 100 |

*Test 1.*. Reference and CAAD BLAST use default parameters Evalue = 10, max_target_seqs = 500). The EUAF threshold is set to the ungapped extension threshold of NCBI BLAST. In this baseline test we note that there are some false negatives, although none ever appear in the top 100 of returned sequences.

*Test 2.*. For CAAD BLAST the EUAF threshold is reduced by 12. This selection is based on the analysis of the EUAF and SW scores of the missing sequences compared to their corresponding threshold. Since the SW threshold is not changed the reduced databases sizes (DB") are not significantly changed. As a result, the post-filter timing remains the same. Reducing the EUAF threshold increases the FPGA streaming time slightly. The accuracy, however, is improved to 100% (no misses) but with a reduction in performance.

*Test 3.*. The Evalue is reduced from the default value of 10 to 1.0E-5 such that the returned sequences are more statistically meaningful. An Evalue of 10 is generally considered too permissive for this size database. This test assesses the effect of the Evalue on the performance and accuracy. As in Test 1, the thresholds used by CAAD BLAST are those calculated by NCBI BLAST during ungapped and gapped extension. The reduction in Evalue has little effect on the FPGA streaming time. The post-FPGA processing time is reduced, however, producing slightly better speedup. The number of false negatives, however, increases to higher than the original.

*Test 4.*. The EUA threshold is reduced by 20%. Also, both reference and CAAD BLAST are tested with -max_target_seqs 50 which forces the tool to report the top 50 sequences only. The selection of 20% reduction as the EUAF threshold, as opposed to a constant reduction of 12 (in Test 2), is based on the analysis of the scores of the missing sequences in Test 3. Since in Test 3 we used a more restrictive Evalue, the default thresholds are increased. The comparison of the scores of the missing sequences and the default thresholds in the other tests shows that with a 20% reduction in EUA threshold we can achieve 100% agreement. Overall this use case shows optimal performance and accuracy results.

Table IV shows results from Tests 2 and 4 for a set of 600 queries selected randomly from NR. We note that the end-to-end speed-up of CAAD BLAST is around 5x when using a single FPGA and over 11x when using 4 FPGAs. Included in the time is overhead in loading the sequence and in computing and loading all parameters and tables. As is usual when reporting BLASTP results we assume that the database has already been formatted and loaded. Also, as is usual in FPGA studies, we do not include configuration time. In the case of CAAD BLAST this may be a drawback since configurations are optimized for ranges of query sizes.

## 8. DISCUSSION

This work builds on substantial previous efforts, in the overall design, the filters themselves, and in integration with both hardware (the FPGA-based accelerators) and software (NCBI BLAST). As this project has evolved and matured, so has the complexity of the design, especially in coupling the stages logically and for load balancing, which in turn required applying more sophisticated methods of design, optimization, and integration. The result is what we believe to be the fastest BLASTP in terms of per socket acceleration.

The results themselves are necessarily a snapshot of a particular level of technology which has already been surpassed for both CPU and FPGA. We have found that CPU-only NCBI BLASTP continues to scale well as cores are added (8 cores and 8 threads tested so far). Since BLAST in general is trivially parallelizable over the number of streams, we also expect CAAD BLAST to fully scale with emerging FPGA technologies, assuming that no drastic changes are made in resource balance.

## ACKNOWLEDGMENTS

The authors would like to thank Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Josh Model, Jin Park, and Yunfei Qiu who contributed to preliminary versions of this work.

## REFERENCES

P. Afratis, E. Sotiriades, G. Chrysos, S. Fytraki, and D. Pnevmatikatos. 2008. A Rate-Based Prefiltering Approach to BLAST Acceleration. In *Proc. IEEE Conf. on Field Programmable Logic and Applications*.

S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.Journal Lipman. 1990. Basic Local Alignment Search Tool. *J. of Molecular Biology* 215 (1990), 403–410.

J.D. Bakos. 2010. High-Performance Heterogeneous Computing with the Convey HC-1. *Computing in Science and Engineering* 12, 6 (2010), 80–87.

G. Cochrane, I. Karsch-Mizrachi, and Y. Nakamura. 2011. The International Nucleotide Sequence Database Collaboration. *Nucleic Acids Research* 39 (2011), D15–D18.

Convey Computer Corporation 2013a. *Convey HC-2 Architectural Overview*. Convey Computer Corporation, www.conveycomputer.com/ files/4113/ 5394/7097/ Convey_HC-2_Architectural_Overview.pdf.

Convey Computer Corporation 2013b. *Hybrid-Core Computing for High Throughput Bioinformatics*. Convey Computer Corporation, www.conveycomputer.com/ files/2613/ 5085/5888/ ConveyBioinformatics_web.pdf.

M.C. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt. 2006. Single Pass, BLAST-Like, Approximate String Matching on FPGAs. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*.

M.C. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt. 2007. Single Pass Streaming BLAST on FPGAs. *Parallel Comput.* 33, 10-11 (2007), 741–756.

A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. Chamberlain. 2008. Mercury BLASTP: Accelerating Protein Sequence Alignment. *ACM Trans. on Reconfigurable Technology and Systems* 1, 2 (2008).

S.D. Kahn. 2011. On the Future of Genomic Data. *Science* 331 (2011), 728–729.

S. Karlin and S.F. Altschul. 1990. Methods for Assessing the Statistical Significance of Molecular Sequence Features by Using General Scoring Schemes. *Proc. Nat. Acad. Sci.* 87 (1990), 2264–2268.

I. Korf, M. Yandell, and Journal Bedell. 2003. *BLAST: An Essential Guide to the Basic Local Alignment Search Tool*. O'Reilly and Associates.

P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster. 2007. Biosequence Similarity Search on the Mercury System. *Journal of VLSI Signal Processing* 49, 1 (2007), 101–121.

T.W. Lam, W.K. Sung, S.L. Tam, C.K. Wong, and S.M. Yiu. 2008. Compressed Indexing and local alignment of DNA. *Bioinformatics* 24, 6 (2008), 791–797.

D. Lavenier, L. Xinchun, and G. Georges. 2006. Seed-based Genomic Sequence Comparison Using a FGPA/FLASH Accelerator. In *Proc. IEEE Conf. on Field Programmable Technology*. 41–48.

C. Ling and K. Benkrid. 2010. Design and implementation of a CUDA-compatible GPU-basedcore for gapped BLAST algorithm. *Procedia Computer Science* 1, 1 (2010).

W. Liu, B. Schmidt, and W. Mueller-Wittig. 2011. CUDA-BLASTP: Accelerated BLASTP on CUDA-Enabled Graphics Hardware. *IEEE Trans. on Computational Biology and Bioinformatics* 8, 6 (2011), 1678–1684.

A. Mahram and M.C. Herbordt. 2010. Fast and Accurate NCBI BLASTP: Acceleration with Multiphase FPGA-Based Prefiltering. In *Proceedings of the 24th ACM Int. Conf. on Supercomputing*. 73–82.

K. Muriki, K. Underwood, and R. Sass. 2005. RC-BLAST: Towards an Open Source Hardware Implementation. In *Proc. Int. Work. High Perf. Comp. Biology*.

NCBI. Accessed 12/2013. NCBI BLAST home. http://blast.ncbi.nlm.nih.gov/Blast.cgi. (Accessed 12/2013).

J. Park, Y. Qui, and M.C. Herbordt. 2009. CAAD BLASTP: NCBI BLASTP Accelerated with FPGA-Based Pre-Filtering. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*. 81–87.

E. Sotiriades and A. Dollas. 2007. A General Reconfigurable Architecture for the BLAST Algorithm. *Journal of VLSI Signal Processing* 48 (2007), 189–208.

Time Logic Corp. Accessed 7/2013. *DeCypher Biocomputing Platforms*. Time Logic Corp., www.timelogic.com/ catalog/752/biocomputing-platforms.

P.D. Vouzis and N.V. Sahinidis. 2011. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics* 27, 2 (2011), 182–188.

F. Xia, Y. Dou, and J. Xu. 2008. Families of FPGA-based accelerators for BLAST algorithm with multi-seeds detection and parallel extension. In *2nd Int. Conf. Bioinformatics Research and Development*. 43–57.

Received July 2013; revised December 2013; accepted March 2014