

Fast and Accurate NCBI BLASTP: Acceleration with Multiphase FPGA-Based Prefiltering *

Atabak Mahram

Martin C. Herbordt

Computer Architecture and Automated Design Laboratory
Department of Electrical and Computer Engineering; Boston University
Boston, MA 02215; USA; {herbordt|mahram}@bu.edu

ABSTRACT

NCBI BLAST has become the *de facto* standard in bioinformatic approximate string matching and so its acceleration is of fundamental importance. The problem is that it uses complex heuristics which make it difficult to simultaneously achieve both substantial speed-up and exact agreement with the original output. We have previously described how a novel FPGA-based prefilter that performs exhaustive ungapped alignment (EUA) could be used to reduce the computation by over 99.9% without loss of sensitivity. The primary contribution here is to show how the EUA filter can be combined with another filter, this one based on standard 2-hit seeding. The result is a doubling of performance over the previous best implementation, which itself is an order of magnitude faster than the unaccelerated original. Other contributions include new algorithms for both the original EUA and the 2-hit filters and experimental results demonstrating their utility. This new multiphase FPGA-accelerated NCBI BLASTP scales easily and is appropriate for use in large FPGA-based servers such as the Novo-G.

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

Keywords

FPGA-Based Coprocessors, High Performance Reconfigurable Computing, Bioinformatics, Biological Sequence Alignment

1. INTRODUCTION

A fundamental insight of bioinformatics is that biologically significant polymers such as proteins and DNA can be abstracted into character strings (sequences). This allows biologists to use approximate string matching (AM) to

*This work was supported in part by the NIH through award #R01-RR023168-01A1. Web: www.bu.edu/caadlab.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'10, June 2–4, 2010, Tsukuba, Ibaraki, Japan.

Copyright 2010 ACM 978-1-4503-0018-6/10/06 ...\$10.00.

determine, for example, how a newly identified protein is related to those previously analyzed, and how it has diverged through mutation. Fast methods for AM, such as BLAST [2], are based on heuristics, and can match a typical sequence (a query) against a set of known sequences (e.g., the millions in the NR database) in just a few minutes. Moreover, these heuristics only rarely cause significant matches to be missed (< .1% in one study [15]). These remarkable results have only increased the importance of BLAST: it is now often used as the “inner loop” in more complex bioinformatics applications such as multiple alignment, genomics, and phylogenetics.

Acceleration of BLAST is therefore of fundamental importance. For example, massively parallel servers for BLAST have been constructed using the Blue Gene/L [22]. Also, NCBI maintains a large server that processes hundreds of thousands of searches per day [5, 17]. And for generic clusters, mpiBLAST is one of the most popular of several parallel BLAST algorithms [6]. For acceleration, FPGAs have probably been the most popular, with commercial products from TimeLogic [24] and Mitronics [18] and several academic efforts [10, 12, 16, 19, 20, 23].

Of the many versions of BLAST, NCBI BLAST [11] has become a *de facto* standard. Public access is possible either through download of code or directly through the large web-accessible server at NCBI. This standardization motivates the design criteria for accelerated BLAST codes: users expect not only that performance be significantly upgraded, but also that outputs match exactly those given by the original system.

BLAST implementations run through several phases, details of which are given below, and return some number of matches with respect to a statistical measure of likely significance. Besides the variations in the heuristics used, BLAST implementations also vary by target: within the NCBI family, we are primarily concerned with BLASTN (for nucleotide-nucleotide comparisons) and BLASTP (for protein-protein comparisons). Although their logic is similar, we focus here on BLASTP. The changes in logic necessary to support BLASTN are described elsewhere [21].

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. There are also multiple interleaved execution paths. Creating an accelerated version that both matches the NCBI BLAST output and delivers significant acceleration is therefore challenging. One approach is to profile the code and accelerate the most heavily used modules. This can give

agreement of outputs, but may not achieve cost-effective performance: there are many paths that add up to much execution time. Accelerating enough of them may not be viable, especially on an FPGA where code size translates into chip area. A second approach is to restructure the code, modifying or bypassing some heuristics. This can lead to excellent performance, but is unlikely to yield agreement. Academic FPGA-accelerated BLASTs [10, 12, 16, 19, 23, 26] have mostly followed one approach or the other. The methods used by the commercial versions mostly are either not publicly available or follow an academic version [18, 24].

In this work we use a third approach – prefiltering (also suggested previously by Herbordt, et al. [9] and by Afratis, et al. [1]). 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 as follows. The prefiltering is constructed to guarantee that its output is strictly more sensitive than the original code: that is, no matches are missed, but extra matches may be found. The latter can then be (optionally) removed by running NCBI BLAST on the reduced database.

The primary result is a transparent FPGA-accelerated NCBI BLASTP that achieves both output identical to the original and a factor of 25x improvement in performance. The mechanism is the primary intellectual contribution of this work: a pair of highly efficient filters. The first implements two-hit seeding, the second performs exhaustive ungapped alignment. The overall significance of this work is as follows:

- We believe CAAD BLASTP (after the name of our lab) to be at least twice as fast as any previous accelerated BLASTP that achieves agreement with NCBI BLAST.
- Since CAAD BLASTP is transparent NCBI BLASTP, and requires only off-the-shelf components, it is likely to be cost-effective and could achieve widespread use.
- CAAD BLASTP decomposes directly and is easy to parallelize.
- Since FPGAs draw only a small fraction of the power of high-end microprocessors, often less than 10W, FPGA-based BLAST servers are likely to be cost-effective.
- Since CAAD BLASTP is based on prefiltering, integration into other versions of BLAST (e.g., parallel) is straightforward.

The rest of this manuscript is organized as follows. We begin with a review of BLAST, followed by an overview of NCBI BLAST, especially in how it differs from the original algorithm. Then comes the overall design, including the mechanisms we use to guarantee agreement. After that is a description of the filters themselves, followed by some practical concerns and results. We conclude with a discussion and future work.

2. BACKGROUND

2.1 Basics of AM for Biological Sequences

We briefly describe biological sequence matching and the classic BLAST algorithm. For details, please see one of the surveys (e.g., [13]). 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 (i.e., gaps or *indels*). 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. The highest scoring alignment between a query sequence of length m and a database of length n can be found in time $O(mn)$ using dynamic programming (DP) techniques (e.g., Needleman-Wunsch and Smith-Waterman).

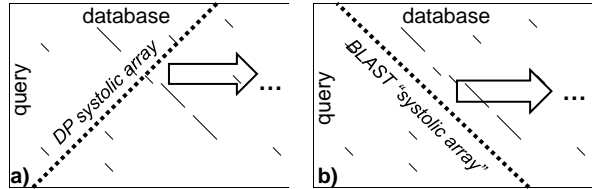


Figure 1: Conceptual alignment tableaux showing distinction between DP and BLAST AM algorithms (after Figure 5.5 in [13]).

Execution of Smith-Waterman is shown graphically in Figure 1a. Depicted therein is a tableau with the query on the vertical and database on the horizontal axes, respectively. Contents are the mn character-character match scores. DP-based methods use a recurrence that is solved along the anti-diagonal which progresses along the database as the computation progresses. The $O(mn)$ complexity results from each match score being involved in a small constant number of operations.

For large databases, however, DP methods are impractical, motivating heuristic methods such as BLAST. 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 has three phases: identifying short sequences (words) with high match scores, extending those matches, and merging proximate extensions. Figure 1b shows conceptually how BLAST reverses the direction of the AM computation: unlike in DP, extensions and mergers progress along the main diagonal. 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, the preferred method of seeding depends on their being two hits on a diagonal (ungapped alignment) 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 Smith-Waterman – the $O(nm)$ is not onerous when 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 (see Figure 2). 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 uses Smith-Waterman to complete gapped extension.

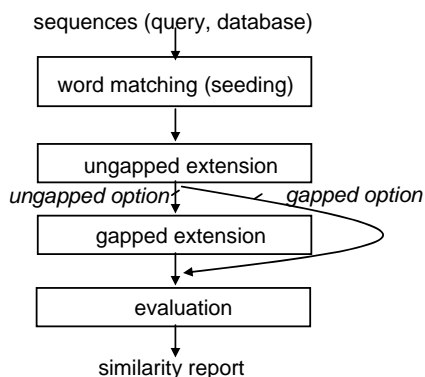


Figure 2: Overview of NCBI BLAST phases.

NCBI BLAST begins the evaluation phase by using an empirically determined cutoff score (*cutoff*) 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*.

The execution flow of the NCBI BLASTP core is shown below with some details added. Note that there is no clear boundary between ungapped and gapped operation. This makes it difficult to substitute directly the modules for ungapped and gapped extension with corresponding FPGA versions (as implied by Figure 2).

```

Seeding;
FOR each database sequence
  Do ungapped extension;
  Calculate cut-off score for HSP \
  linked-list;
  Build HSP list;
  IF gapped alignment
    DO gapped extension for each HSP \
    in list;
  Compute Evalue and reap HSP list \
  (either ungapped or gapped);
END FOR
IF needed
  Compute traceback;
IF gapped alignment,
  Recompute alignments using S-W and reap;
  Execution flow of NCBI BLASTP core
  
```

2.3 Target Systems

We briefly state our assumptions about the target systems with FPGA-based accelerators. They are typical for current products; details of appropriate FPGA-based systems can be found, e.g., in [8].

- The overall system consists of some number of standard nodes. Typical node configurations have 1-4 accelerator boards plugged into a high-speed connection (e.g., the Front Side Bus or PCI Express). The host node runs the main application program. Nodes communicate with the accelerators through function calls.
- Each accelerator board consist of 1-4 FPGAs, memory, and a bus interface. On-board memory is tightly coupled to each FPGA either through several interfaces (e.g., 6 x 32-bit) or a wide bus (128-bit). 4GB-8GB of memory per FPGA is currently standard.
- Besides configurable logic, the FPGA has dedicated components such as independently accessible multiport memories (e.g., 1000 x 1KB) called Block RAMs (or BRAMs) and a similar number of multipliers. FPGAs used in High Performance Reconfigurable Computing typically run at 200 MHz, although with optimization substantially higher operating frequencies can be achieved.

We have been developing CAAD BLAST using a particular *reference system* consisting of a standard high-end PC with a Gidel PROCe III accelerator board. The FPGA is an Altera Stratix III 260E, a 2007-era 65nm device. Its key characteristic for this application is its BRAM count: there are 864 1KB BRAMs and 48 18KB BRAMs.

All algorithms and implementations map immediately onto similar systems (i.e., from other vendors for both boards and FPGAs). CAAD BLAST also maps to multi-FPGA accelerators such as the Gidel PROCStar III, which has four FPGAs, and to large FPGA-based systems. For the latter we are targeting the Novo-G, which has 192 FPGAs [4].

3. CAAD BLAST DESIGN

3.1 Filter Basics

CAAD BLAST uses three FPGA-based filters:

- The Two-Hit Filter is based on the two-hit seeding algorithm. All alignments (all diagonals in Figure 1b) are evaluated as to whether or not they contain a two-hit seed. The output is a bit vector containing a 1/0 for each diagonal depending on whether or not the diagonal contains a seed. We base our Two-Hit Filter on the two-hit seeding algorithm used by Mercury BLAST and described in [12].
- The Exhaustive Ungapped Alignment (EUA) Filter scores every possible alignment between the query and the database. For each sequence in the database it returns the scores of the highest scoring alignments. We base our EUA Filter on the TreeBLAST algorithm described in [10].
- The Exhaustive Gapped Alignment (S-W) Filter is based on the Smith-Waterman algorithm and returns the highest scoring gapped local alignments for each sequence in the database. We base our SW filter on the version of Smith-Waterman described in [3].

All three filters work on the same principal. Each occupies some amount of chip area (in the FPGA) and holds a copy of the query. It then executes as the database streams through it from off-chip memory. The filter size (in chip area) 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 filters still operate correctly, but have reduced performance with a slowdown generally proportional to the query size.

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 as NR currently have nearly 4GB of data: off-the-shelf FPGA plug-in boards (e.g., the XD1000 from XtremeData [27] and the PROCe III from Gidel [7]) can hold this in local memory and stream it through the FPGA in about a second. This assumes the specifications of 333MHz and 16 bytes per cycle with one character per byte.

Why multiple passes? Because for each phase the number of filters per chip and, thus the performance, varies. For example, for typical queries on our reference system: the Two-Hit filter can process the entire memory bandwidth, the EUA filter about a fifth that much, and the SW filter about a quarter of that. Each filter reduces the amount of work that needs to be processed by the next filter. The following scenario is for gapped NCBI BLAST.

- The Two-Hit pass provides “hints” to the EUA filter as to which diagonals can be skipped. As described below, actually skipping diagonals is not cost-effective, but making the EUA filters drastically more compact is. After compaction, the EUA pass is almost as fast as the two-hit pass.
- The EUA filter prunes at least 95% of the database so that it need not be processed by the SW filter. Again, the time is reduced to that of the Two-Hit pass.
- The S-W filter prunes the database to 0.1% of the original. The reduced database is then processed by NCBI BLASTP.

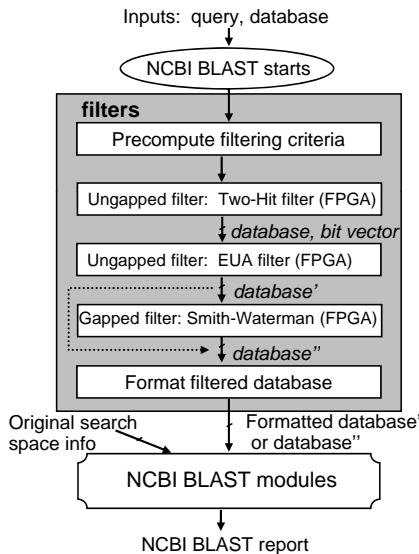


Figure 3: High-level Design of CAAD BLAST.

3.2 CAAD BLAST System Design

As just described, the basic design of CAAD BLAST is to successively reduce the database (say, DB) without removing any potential matches. First, DB is filtered by running the Two-Hit filter and a set of hints generated. These, together with DB are sent to the EUA filter and a reduced database

DB’ is generated. Then, for the gapped option, S-W is run to generate a further reduced database DB”. Finally, DB” (DB’ for the ungapped option) is formatted and sent to NCBI BLAST, together with the *original* parameters and query.

To accomplish this, two problems need to be solved. The first is to get the numbers right. There are two parts: determining the internal thresholds that NCBI BLAST would use, especially *cutoff*, and correctly computing the *Evalues* in the final report. The second and more serious problem is that we need to ensure that DB” both (i) contains all the sequences that NCBI BLAST would return, and (ii) is sufficiently reduced so that the overhead of formatting DB” does not overwhelm any potential performance gain. The methods we use follow those described in [20].

Figure 3 shows the global structure of CAAD BLAST. In the precompute module, the host uses logic from the NCBI code to compute the various parameters needed to determine *cutoffs* and *Evalues* for both ungapped and gapped options. The EUA filter begins with the FPGA using these parameters, together with the query and database, to compute the ungapped alignment scores. For the most promising sequences, scores are returned to the host, which uses them to compute the *Evalues* and specify DB’. For the gapped option, a new threshold is computed and passed to the FPGA where the contents of DB” are determined. Finally, the reduced database (either DB’ or DB”) is formatted to be processed by NCBI BLASTP. To ensure that the *Evalues* match those that would be computed by the original code, we also pass the original search space information.

4. TWO-HIT FILTER

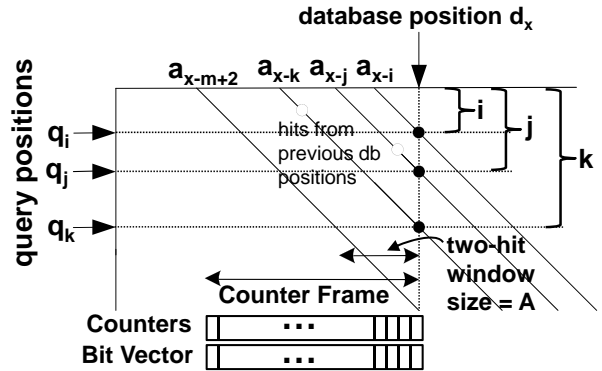


Figure 4: The Two-Hit filter is processing the x th 3-mer in the database sequence. There are three hits. The hit on alignment a_{x-j} is within A of a previous hit, and so is part of a two-hit event. This is determined by comparison with the corresponding Counter value; its bit in the Bit Vector will be set.

The design of the Two-Hit filter is generally similar to that used by Mercury BLAST in the seeding pass [12, 14]. There are several algorithmic and implementation differences, however, which have two significant consequences:

- The Two-Hit filter does not use heuristics and so has exact agreement with the two-hit seeding algorithm used in NCBI BLAST.
- The Two-Hit filter is compact. For our reference design, 38 streams can be processed in parallel for small

proteins (size 100), 28 for average-sized proteins (size 500), and 14 for large proteins (size 2200).

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

4.1 Algorithm Overview

We describe a single filter; the extension to multiple filters operating in parallel follows immediately. We begin with some notation, an overview of the algorithm, and a critical observation.

Figure 4 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 Two-Hit filter 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 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.

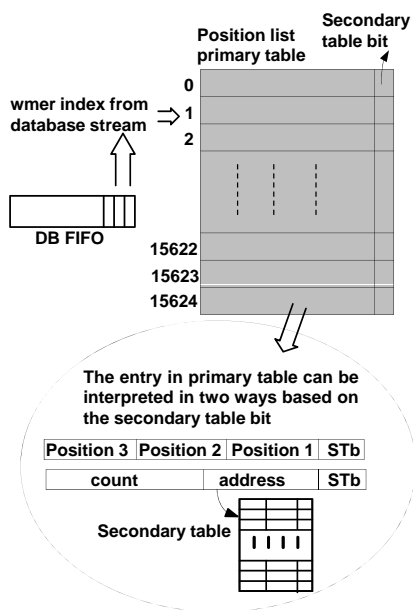


Figure 5: Shown is the Query 3-mer Position Table.

The primary data structure is the Query 3-mer Position Table shown in Figure 5. 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 ($25 \times 25 \times 25$) 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 Two-Hit filter. On iteration x , database 3-mer d_x indexes the Position Table. The query positions where matches occur, if any, are retrieved. Figure 4 shows three hits, at query positions q_i , q_j , and q_k . These correspond, 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 4 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 Two-Hit filter 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

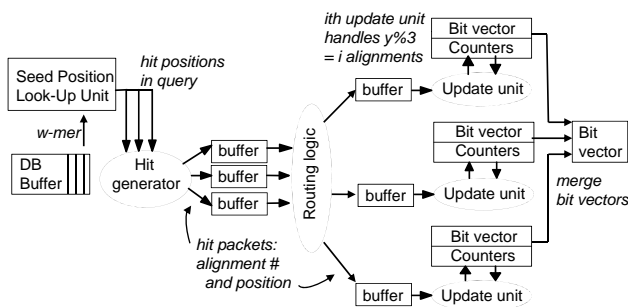


Figure 6: Block diagram of the Two-Hit Filter.

Overall

The overall structure of the Two-Hit filter is given in Figure 6. The goal is to process the database at streaming rate. To support this, the Two-Hit filter 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.

Table 1: Number of Two-Hit filters that fit on the reference design for selected sequences from the NR database.

Query Size	# of Two-Hit Filters	# of Hits per DB char.	# of excess cycles per DB char.
81	38	0.064	0.0002
217	35	0.206	0.0100
490	28	0.567	0.0524
838	25	0.891	0.2203
1204	21	1.244	0.3062
2205	14	2.570	0.8790

4.3 Design Decisions and Experimental Results

For the two-hit filter, 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 1 shows the number of two hit filters that can be instantiated, using the design just described, for a selection of sequences from the NR database.

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 query sizes (less than 1K) this number (3) falls out immediately from the convenience of packing that number of 10-bit addresses into a single 32-bit word. Also for small queries, having, say, 100 filters does little good: that is far more than the number of streams that can be supported by the memory interface in the reference design.

For larger queries, there is the possibility of optimization by trading 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 right two columns in Table 1 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 of size 1K, 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 the query of size 2205, however, the secondary table must be accessed frequently. The rightmost 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.



Figure 7: Graph shows performance as a function of query size for the Two-Hit filter in the reference design. Both the number of filters and the throughput per filter vary with query size.

The performance of the Two-Hit filter phase depends substantially on the query size. There are two effects: the number of filters per chip and the amount of throttling that needs to be done because of references to the secondary table. Experimental results are shown in Figure 7 in terms of cycles per character as a function of query size. For typical protein sequences, size 100 to 500, the throughput is from 25-30 characters per cycle. For large proteins, size 2000 to 2500, the throughput is 6-7 characters per cycle. This enables

processing of the NR database, which has 3.53G residues, in less than 1s for most queries and about 5s for large queries.

5. EXHAUSTIVE UNGAPPED ALIGNMENT FILTER

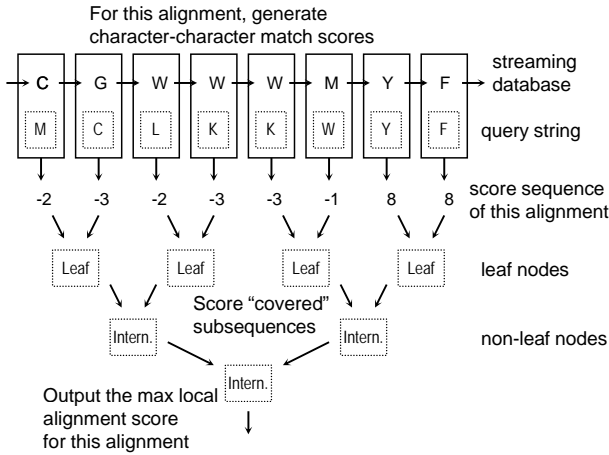


Figure 8: TreeBLAST structure for $m = 8$.

The Exhaustive Ungapped Alignment filter (EUA) is based on the TreeBLAST scheme described in [10]. The TreeBLAST filter has two essential properties. First, it exhaustively evaluates every possible ungapped alignment between query and database. And second, it is fully pipelined: it evaluates the database at streaming rate. The exhaustive nature of TreeBLAST is its strength, but also its weakness: it examines each alignment without regard to any possible information about the likelihood that alignment being significant. In this section we describe the EUA filter: how the TreeBLAST structure can be modified to take advantage of just such likelihood information as is generated from the Two-Hit filter.

5.1 Original TreeBLAST

The key idea behind TreeBLAST is that ungapped alignment can be performed with iterative merging using a tree structure (see Figure 8) that forms a two dimensional systolic array. The database sequence is streamed across the leaves of the tree (top) and one complete score sequence (the set character-character match scores for that alignment) is generated every cycle. The score sequences are processed by the tree, which is also pipelined. For each alignment, the score of the best local alignment emerges after a few cycle delay. The nodes of the tree consist of a some basic comparison logic; the tree size is generally limited by the number of BRAMs on the FPGA. In our reference design, the maximum tree size (unfolded) is about 1800. The structure can be modified in several ways to run more efficiently and to handle various cases.

Folding. To handle queries larger than can fit on chip, the tree is “folded” (see Figure 9). Rather than generating a scoring sequence every cycle, 2^i cycles are required, where i is the number of folds. On each cycle, $1/2^i$ of the score sequence is generated. That is, the tree is used on multiple iterations to handle the sequence. In the left side of Figure 9, the tree in Figure 8 is folded once. During cycle 1 the first

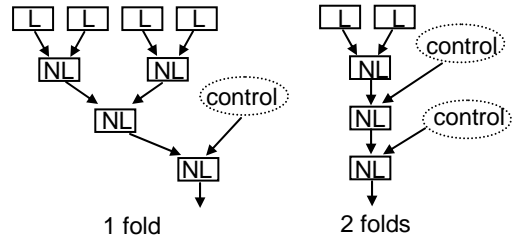


Figure 9: An 8 leaf tree folded once and twice. L = leaf node, NL = non-leaf node.

half of the sequence is scored; during cycle 2, the second half. The alignment score for the first half reaches the root during cycle 4 and is combined with the alignment score of the second half during cycle 5. The right side of Figure 8 shows two folds; processing is analogous.

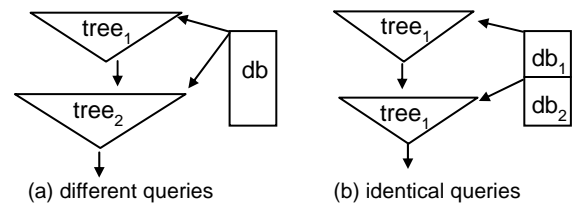


Figure 10: Replicated trees in two configurations: double throughput for one query, or process two queries.

Replication. When queries are small enough to fit multiple trees on the chip, they are replicated to take advantage of available resources. Alternatively, if the query consists of a number of sequences, several can be evaluated simultaneously (Figure 10). Which is used depends on whether the bottleneck lies in the memory interface or on the chip.

5.2 Use of Prior Knowledge

The idea is to couple the database stream with a bit vector indicating which alignments can safely be ignored (as generated by the Two-Hit filter). For example, a one in the bit vector corresponds to a position where the alignment must be processed, a 0 where it can be skipped.

We now look at the skipping mechanism. This necessarily involves a tradeoff between the amount of processing that can be avoided and the hardware support required. As always, the more additional hardware that is required, the greater the reduction in parallelism and thus performance.

General Skipping.

The idea behind general 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. Ideally, general skipping takes only the number of cycles equal to the number of ones in the bit vector. The additional hardware required, however, is complex. For the bit vector, the “look-ahead” logic is similar to a leading one detector used, e.g., in a floating point adder. 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 tree.

Constant Skipping.

The idea behind constant skipping 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. 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.

1) 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.

2) The maximum skip is also limited. No matter how long the 0 sequence, only S will be skipped. This indicates a large S .

The optimal S is query dependent but generally follows the query length. A larger S works best with smaller queries: Their alignments are more much likely to have been filtered.

Folded Skipping.

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 in the bit vector.

The question is why bother folding at all. The answer is that folding gives a way to make the EUA structures (trees) substantially more compact than previously and thus allows them to be replicated. For example, given a database of size N and a query of size M that is handled by a single tree (with a single database stream). This takes N cycles. Now replace the tree with F trees folded to $1/F$ their original size. These now collectively support F database streams, each of which has a throughput that is a substantial fraction of the original.

5.3 Implementation and Performance

The limit on the number of trees is generally given by the query size M , the number of folds F , and the number and size of the BRAMs. For the reference design, the number of columns of the scoring matrix that can fit in an M9K BRAM is 32. Since BRAMs are dual ported, it is most efficient to use them to look up two characters at a time. This places

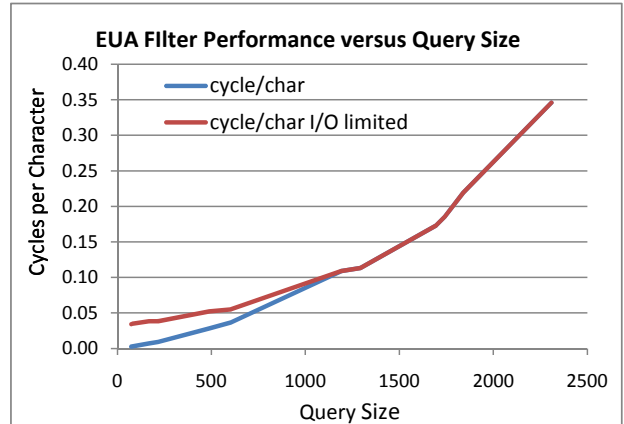


Figure 11: Graph shows performance as a function of query size for the implementation of the EUA filter in the reference design. Both the number of filters and the throughput per filter vary with query size.

a practical limit of 16 on F . Given the 912 BRAMs in the reference design FPGA, the maximum number of EUA filters is $1824 \times F/M$, or 96 for $M = 300$.

It is possible, however, that the memory bandwidth will be limited, perhaps to 32 characters per cycle, rather than 96. The graph in Figure 11 shows performance in cycles per character as a function of query size. The upper graph assumes that the computation is memory bandwidth limited, the lower does not. For the “limited” graph, the range is from 3 to 30 characters per cycle. This enables processing of the NR database in less than 2s for most queries and about 10s for large queries.

6. IMPLEMENTATION AND RESULTS

6.1 System Configuration and Operation

CAAD BLAST operation proceeds as follows. As is usual, we assume that the database is preloaded into staging memory. Unlike NCBI BLAST the database is *unformatted*. The user specifies the query and parameters using the NCBI BLAST interface. For each filter, the FPGA is configured, the sequence is loaded and the filter executed. The Two-Hit filter generates a bit vector which is stored in on-board memory. The EUA filter returns high scores to the host (or to the corresponding node processor). The host uses the scores to determine the sequences in DB’. If gapped alignment, the processing continues similarly with the Smith-Waterman filter to generate DB’’. We use our own implementation of Smith-Waterman that closely follows the version by Wozniak [25]. Folding enables processing of large queries. In the final step, DB’ (or DB’ for ungapped alignment) is formatted and executed with the original NCBI BLAST.

CAAD BLAST runs on standard off-the-shelf FPGA accelerated systems as described in Section 2.3. While we currently assume a high-end FPGA, CAAD BLAST is easily decomposable and also runs well on low-end devices. Assuming sufficient memory bandwidth, the performance is roughly proportional to the number of BRAMs. The size of on-board memory should be sufficient to store the database.

Table 2: Various results for CAAD BLAST. Averages from running sequences of NR versus NR. Original NR database has about 10.3M sequences and 3.53G residues.

NCBI BLASTP	Ungapped	Gapped
exec. time on lab PC	170s	178s
exec. time on NCBI web server	—	12-20s
CAAD BLASTP		
NR' (Sch 2) reduction from NR		
% of sequences remaining	0.116%	—
% of residues remaining	0.338%	—
NR' (Sch 1) reduction from NR		
% of sequences remaining	—	3.24%
% of residues remaining	—	6.04%
NR'' (S-W) reduction from NR		
% of sequences remaining	—	0.054%
% of residues remaining	—	0.088%
Format overhead NR' NR''	1.50s	0.53s
NCBI exec. overhead NR' NR''	1.49s	2.62s

For NR, which is a large protein database, this is less than 4GB when packed. Database sizes are still increasing faster than memory density, but for the next few years we should be able to count on holding databases in the local memories of individual nodes. For multi-FPGA systems the database is distributed across multiple nodes, so memory should not be an issue for the foreseeable future.

6.2 Validation and Performance

Agreement of results from NCBI BLAST and CAAD BLAST has been validated in two ways: through code analysis and from execution. For runs of sequences of NR versus the entire database, all queries have returned identical high scoring sequences and scores for those sequences.

Table 2 contains various results from the filter and reference runs. Our primary reference system is a 2008 64-bit 3GHz Xeon quad processor (Harpertown X5412) with 8GB of memory. We are currently running NCBI BLAST 2.2.20 for reference and for the base code of CAAD BLAST. We compiled each with standard optimization settings and run with default settings. For additional reference we use the web server at NCBI.

We now discuss some of these results. We note that they are averages; there is variation as expected from sequences of widely varying sizes. Scheme1 and scheme2 refer to different methods of using the EUA output to filter the original database (NR) to create a reduced database (NR'). They are appropriate for the ungapped and gapped options of NCBI BLAST, respectively, and are described in detail in [20].

- For gapped BLAST (Scheme 1) a database sequence is retained if it contains at least one HSP that scores above *cutoff*. This is a very low bar and is guaranteed to retain all sequences that NCBI BLAST would hold after extension. NR is reduced by a factor of 17.
- Scheme 2 refers to a more complex method of thresholding, although it still has only a fraction of the complexity of the NCBI algorithm. Up to five HSPs per sequence are processed. While the bar is much higher than in Scheme 1, agreement with NCBI BLAST is still guaranteed. Here NR is reduced by a factor of 296.

- For gapped processing with Smith-Waterman, NR is reduced by a factor of 1136 and generally only a few thousand sequences remain.
- The formatting overhead includes host processing for the filters.

We have implemented all three filters on the reference system which contains a Gidel PROCe III FPGA board. The FPGA is an Altera Stratix-III 260E. This is a high-end device using the 65nm process, but now nearly two generations old. For memory there is 4.5GB of DRAM partitioned into three banks of 2GB, 2GB, and 512MB, respectively. Each bank has a 64-bit interface and can be accessed independently. One of the 2GB and the 512MB banks run at 333MHz; the other 2GB bank runs at 166MHz.

Table 3 contains performance results of the reference design with respect to the NR protein database. Also shown are results for the unaccelerated host PC and the NCBI Server. For CAAD BLAST the S-W time is less than the time for the other filters. Most of the time is in executing the final run of NCBI BLASTP. By percentile we indicate the rough proportion of queries that are smaller than the size shown [5]. Speed-ups over the unaccelerated PC range from 25 \times to 30 \times . The NCBI Server is a large cluster that processes queries in parallel according to load.

7. DISCUSSION AND FUTURE WORK

We have described the design and implementation of a high performance BLAST application accelerated with multiphase FPGA-based prefiltering. We are able to achieve both exact match with NCBI BLAST output and substantial performance improvement. Running with little optimization on three year old hardware it achieves a factor of 25x speed-up. On a new system and with some care we anticipate substantially higher performance, especially for larger queries. New FPGAs have substantially more resources, higher clock frequencies, and higher memory (I/O) bandwidth. The designs described here scale immediately with change of parameters and recompilation (synthesis and place-and-route).

But even on the current device there remains substantial "slack." This lies especially in two places. The first is the extremely conservative heuristics used in creating the reduced databases db' and db'' for gapped alignment. The second is in the overhead involved in the final run. These are coupled. Note that while db'' is typically much less than 0.1% of the original database, processing it with NCBI BLAST takes more than ten times longer than expected. This is because the remaining sequences all have alignments of interest and must undergo substantial processing. If most of this last pass can be eliminated, performance should improve by at least 50%. Since CAAD BLAST already runs Smith-Waterman on these sequences, this seems reasonable.

The filters are usable with any back end. Also, the two-hit filter should be usable by any FPGA-accelerated BLAST, whether it involves prefiltering or not. We have several goals for the next few months. One is to finish porting CAAD BLAST to a 4-FPGA board, the Gidel PROCStar III. Another is to map CAAD BLAST onto the 192-FPGA Novo-G.

Table 3: Performance of the reference design with respect to the 3.53G residue NR database. The gapped option of NCBI BLASTP is used.

query size percentile	2-hit chars/cycle time	EUA chars/cycle time	S-W and Overhead	Total time Accelerated	Total time CPU Only	Total time NCBI Server
up to 500 78th	25/cycle 1.3s	20/cycle 1.3s	3.8s	6.4s	187s	14s
up to 1000 97th	18/cycle 1.9s	11/cycle 2.4s	5.3s	9.4s	292s	20s
up to 2000 99.5th	7/cycle 4.8s	4/cycle 6.6s	7.8s	19.2s	485s	40s

8. REFERENCES

- [1] Afratis, P., Sotiriades, E., Chrysos, G., Fytraki, S., and Pnevmatikatos, D. A rate-based prefiltering approach to BLAST acceleration. In *Proc. IEEE Conference on Field Programmable Logic and Applications* (2008).
- [2] Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. Basic local alignment search tool. *Journal of Molecular Biology* 215 (1990), 403–410.
- [3] Chow, E., Hunkapiller, T., and Peterson, J. Biological information signal processor. In *Proc. International Conference on Application Specific Systems, Architectures, and Processors* (1991), pp. 144–160.
- [4] CHREC: NSF Center for High-Performance Reconfigurable Computing. *Facilities*. www.chrec.org/facilities.html, Web page accessed 1/2010.
- [5] Coulouris, G. BLAST benchmarks. NCBI/NLM/NIH Presentation, June 2005.
- [6] Gardner, N., Feng, W., Archuleta, J., Lin, H., and Ma, X. Parallel genomic sequence-searching on an ad hoc grid: Experience, lessons learned, and implications. In *Supercomputing* (2006).
- [7] GiDEL, Inc. *PROC Boards*. www.gidel.com, Accessed 1/2010.
- [8] Hauck, S., and DeHon, A. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing*. Morgan Kaufmann, 2008.
- [9] Herbordt, M., Model, J., Sukhwani, B., Gu, Y., and VanCourt, T. Single pass, BLAST-like, approximate string matching on FPGAs. In *Proc. FCCM* (2006).
- [10] Herbordt, M., Model, J., Sukhwani, B., Gu, Y., and VanCourt, T. Single pass streaming BLAST on FPGAs. *Parallel Computing* 33, 10-11 (2007), 741–756.
- [11] <http://blast.ncbi.nlm.nih.gov/Blast.cgi>, Accessed 1/2009.
- [12] Jacob, A., Lancaster, J., Buhler, J., Harris, B., and Chamberlain, R. Mercury BLASTP: Accelerating protein sequence alignment. *ACM Transactions on Reconfigurable Technology and Systems* 1, 2 (2008).
- [13] Korf, I., Yandell, M., and Bedell, J. *BLAST: An Essential Guide to the Basic Local Alignment Search Tool*. O’Reilly and Associates, 2003.
- [14] Krishnamurthy, P., Buhler, J., Chamberlain, R., Franklin, M., Gyang, K., and Lancaster, J. Biosequence similarity search on the Mercury system. In *Proc. International Conference on Application Specific Systems, Architectures, and Processors* (2004), pp. 365–375.
- [15] Lam, T., Sung, W., Tam, S., Wong, C., and Yiu, S. Compressed indexing and local alignment of dna. *Bioinformatics* 44, 6 (2008), 791–797.
- [16] Lavenier, D., Xinchun, L., and Georges, G. Seed-based genomic sequence comparison using a FGPA/FLASH accelerator. In *Proc. IEEE Conference on Field Programmable Technology* (2006), pp. 41–48.
- [17] McGinnis, S., and Madder, T. BLAST: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Research* 32 (2004), Web Server Issue.
- [18] Mitrionics. *Mitrion-Accelerated NCBI BLAST for SGI BLAST*. Available at www.mitrionics.se/press, Accessed 1/2010.
- [19] Muriki, K., Underwood, K., and Sass, R. RC-BLAST: Towards an open source hardware implementation. In *Proc. International Work. High Performance Computational Biology* (2005).
- [20] Park, J., Qiu, Y., and Herbordt, M. CAAD BLASTP: NCBI BLASTP accelerated with FPGA-based pre-filtering. In *Proc. FCCM* (2009), pp. 81–87.
- [21] Park, J., Qiu, Y., and Herbordt, M. CAAD BLASTn: Accelerated NCBI BLASTn with FPGA prefiltering. In *Proceedings of the IEEE International Symposium on Circuits and Systems* (2010), p. TBD.
- [22] Rangwala, H., Lantz, E., Musselman, R., Pinnow, K., Smith, B., and Wallenfelt, B. Massively parallel BLAST for the Blue Gene/L. In *Proc. High Availability and Performance Computing Workshop* (2005).
- [23] Sotiriades, E., and Dollas, A. A general reconfigurable architecture for the BLAST algorithm. *Journal of VLSI Signal Processing* 48 (2007), 189–208.
- [24] Time Logic Corp. *Web Site*. www.timelogic.com, Accessed 1/2010.
- [25] Wozniak, A. Using video-oriented instructions to speed up sequence comparison. *Bioinformatics* 13, 2 (1997), 145–150.
- [26] Xia, F., Dou, Y., and Xu, J. Families of FPGA-based accelerators for BLAST algorithm with multi-seeds detection and parallel extension. In *2nd Int. Conf. Bioinformatics Research and Development* (2008), pp. 43–57.
- [27] XtremeData, Inc. *XD1000 Development System*. www.xtremedata.com, Accessed 2/2010.