CAAD BLASTP: NCBI BLASTP Accelerated with FPGA-Based Pre-Filtering

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

***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. Our approach is to prefilter the database. To make this work we have developed a novel heuristic which we append to a previously described structure for ungapped alignment. This enables us to quickly reduce the database by factors of 300 and 1100, for the ungapped and gapped options, respectively, while rejecting no significant sequences. On current hardware we anticipate a speed-up of at least a factor of 10 for NCBI BLASTP, independent of sensitivity settings. This filter is portable to other BLAST codes, and other filters can be similarly integrated into NCBI BLAST.**

I. 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 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 are based on heuristics and allow a typical sequence (a query) to be matched against known sequences (e.g., the millions in the NR database) in just a few minutes. This remarkable result has only increased the importance of AM: it is now often used as the "inner loop" in more complex bioinformatics applications such as multiple alignment, genomics, and phylogenetics. Acceleration of AM is therefore of fundamental importance.

BLAST is the dominant heuristic AM application, or more precisely, family of AM applications. Since the original [2], many versions have been developed that may give slightly different outputs. Of these, NCBI BLAST [6] 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 AM codes: not only must performance be significantly upgraded, users expect the outputs to 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.

Since BLAST uses heuristics, it will typically miss some significant matches; the benefit, however, is a tremendous increase in performance. In typical operation, the user gives parameters that trade off sensitivity (the likelihood that some high-scoring sequences will be missed) with performance.

Note that we distinguish the terms *sensitivity* and *agreement*. We use sensitivity in its usual sense, which is the deviation from the theoretical ideal due to the use of heuristics. We use agreement to refer to the equality of outputs between an accelerated version of a code and the original. For example, an accelerated code can have higher sensitivity than the original, but then will not agree. If the accelerated code is *strictly* more sensitive, then it misses none of the original, and running the result through the original code removes the extra results.

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 while delivering significant acceleration is therefore challenging. One approach is to profile the code and accelerate the most heavily used modules. This can give agreement, but is not likely to achieve cost-effective performance: there are too many paths that add up to too 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 versions of BLAST [7], [9], [11], [13] have mostly followed one approach or the other. The methods used by the commercial versions mostly are either not publically available or follow an academic version [10], [14].

In this work we use a third approach – prefiltering (also suggested previously by us [3] 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. Our prefiltering is guaranteed to be strictly more sensitive than the

This work was supported in part by the NIH through award #R01-RR023168-01, by IBM through a Faculty Award, and facilitated by donations from XtremeData Inc. and Altera Corporation. Email: {parkj|yunfei|herbordt}@bu.edu. Web: http://www.bu.edu/caadlab.

original code: that is, no matches are missed, but extra matches may be found. The latter can then be (optionally) removed by NCBI BLAST.

The primary result is a transparent FPGA-accelerated NCBI BLASTP that achieves both output identical to the original and a factor of 10x improvement in performance. Since the prefiltering mechanism is more sensitive than the original, the user may keep the extra outputs at no cost in performance. A further result is that CAAD BLASTP (as we refer to our system) fully supports both modes of NCBI BLASTP, gapped and ungapped. The overall significance of this work is as follows:

- Since CAAD-BLASTP is transparent NCBI BLASTP, and requires only an off-the-shelf FPGA development board, it is likely to be cost-effective and could achieve widespread use.
- Since CAAD-BLASTP is based on prefiltering, integration into other versions of BLAST (e.g., parallel) is straightforward.
- The prefiltering method and its software support allows other filters (e.g., [1]) to be similarly integrated.
- The structures developed for the filter are applicable to more general problems such as parallel prefix.

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. This includes a summary of the base code and how it has been abbreviated to make the filter cost-effective. There follows a discussion of several FPGA structures necessary to enable use of Tree-BLAST [4] in a production system. These include folding for large queries, replication for small queries, overlapped processing of database sequences, and dealing with arbitrary query sizes. A description of practical concerns and results comes next. We conclude with a discussion and future work.

II. NCBI BLAST

We briefly describe biological sequence matching and the classic BLAST algorithm. For details, please see one of the surveys (e.g., [8]) or previous work (e.g., [4], [7]).

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 $O(nm)$ using dynamic programming (DP) techniques (e.g., Needelman-Wunsch and Smith-Waterman). For large databases, DP methods are impractical, motivating heuristic methods such as BLAST.

The original BLAST algorithm has three phases: identifying short sequences (words) with high match scores, extending those matches, and merging proximate extensions. The third phase is nowadays often replaced by Smith-Waterman – the $O(nm)$ is not onerous when n is a fraction of the original. In the first phase, the word size W is typically 2 or 3 for BLASTP and significance is determined using a scoring matrix and threshold score. In the extension phase, 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.

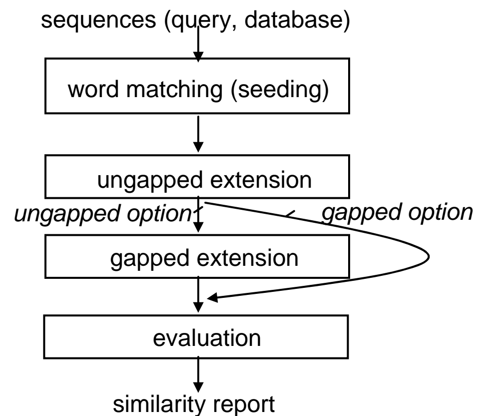

Fig. 1. Conceptual view of NCBI BLAST phases.

NCBI BLAST adds a number of phases and options, which we begin to describe here; further details are given below. There are two options, ungapped and gapped (see Figure 1). 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.

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 1).

```
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

III. CAAD BLAST DESIGN

A. Overview

From previous work by our group and others, we have two essential FPGA components: designs for theoretically ideal ungapped and gapped alignment. These are TreeBLAST [3] and Smith-Waterman [5], [12], [15], respectively. Both hold the query in the FPGA and then stream the database through it. Both run in $O(N)$, assuming that the query sequence is a small multiple of what can fit on a chip, a characteristic of almost all proteins. For larger queries, the slowdown is proportional to the query size. Large protein databases have 2GB of data: off-the-shelf FPGA-based systems (e.g., the XD1000 from XtremeData [17]) can stream this through an FPGA in less than a second. FPGA resources currently constrain this throughput slightly for TreeBLAST, and by about a factor of 10 for Smith-Waterman. This additional overhead for Smith-Waterman is what makes FPGA-based BLAST cost-effective in the first place.

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 TreeBLAST and a reduced database DB' is generated. Then, for the gapped option, Smith-Waterman 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. Details are given below and in Section 3.2. 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. Our methods here are described in Section 3.3.

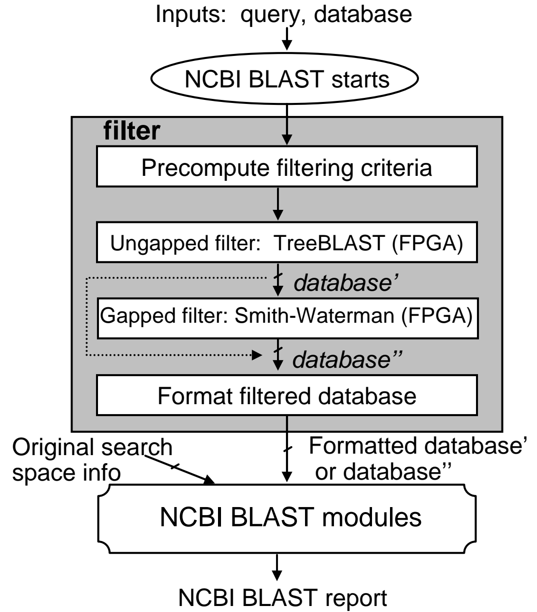

Fig. 2. Overall design of CAAD BLAST.

Figure 2 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 $cutoff$s and $Evalue$s for both ungapped and gapped options. The ungapped 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 $Evalue$s 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 $Evalue$s match those that would be computed by the original code, we also pass the original search space information.

B. Precomputation of Filtering Criteria

We begin with a brief review of the scoring scheme and terminologies used in BLAST. The raw scores in a scoring matrix are determined as $raw_score = original_lod_score * scaling_factor$. From this we get the $normalized_score = raw_score * \lambda$ where λ (appr. 1/scaling_factor) is a matrix-specific constant. The relative entropy H of a scoring matrix is determined as

$$H = -\sum_{i=1}^{r}\sum_{j=1}^{i} q_{ij}\lambda S_{ij}$$

where r, q_{ij}, and S_{ij} are the alphabet size of the matrix, the frequency of pair i, j, and the raw score of pair i, j, respectively. The number of alignments expected is the $Evalue$, with a smaller $Evalue$ implying greater significance. $Evalues$ are computed as follows:

$$Evalue = K * m' * n' * exp^{-\lambda S} \quad (1)$$

where K is a constant and m', n', and λS are the effective query length, effective database length, and normalized score, respectively.

We now describe the effective length computations in more detail. These were derived from analysis of the NCBI BLAST code.

Step1: Access query and database and get their sizes m and n, respectively; the database size is likely to be known.

Step2: Compute λ, K, H for ungapped alignment and gapped alignment separately. In ungapped alignment, these values depend on the query. We skip the details.

Step3: Compute other variables α, β, and $adjust$ (length adjustment) for either ungapped or gapped alignment:

$$adjust = \log(K * m * n)/H.$$

Step4: Compute the effective search space:

$$effective_space = (n - nseqs * adjust) * (m - adjust)$$

where $nseq$ is the number of database sequences.

Step5: Compute the raw score corresponding to the threshold $Evalue$ for both ungapped and gapped alignments. Use the constant values for ungapped and gapped alignments, respectively. The computation is based on Equation 1 in which $m' * n'$ represents the effective search space computed in Step4. The "reverse engineered" raw score threshold is

$$S = ((\log(Evalue) - \log(effective_space) - \log(K))/ - \lambda$$

where $Evalue$ is either from the user or the default (10.0).

Step6: Compute the $gap_trigger$ value using ungapped values of λ and K:

$$gap_trigger = int((c1 * c2 + log(K))/\lambda)$$

where $c1$ and $c2$ are contants used in NCBI BLASTP program, i.e., BLAST_GAP_TRIGGER_PROT (22.0) and NCBI-MATH_LN2 (0.693147), respectively.

Step7: Compute the cutoff score using the gap trigger value and raw score threshold:

$$cutoff = Min(gap_trigger, ceil(S))$$

This cutoff score is used to determine the alignments in the HSP list, while the threshold determines significance of the database sequence.

The length adjustment and effective search space computed in Step3 and Step4 are the "original search space info" shown in Figure 2. They are saved for use by NCBI BLAST in the final phase. The data used by the ungapped-filter module include the cutoff score, effective search space, and constants λ and K.

C. Decisions Made in the Filter

We now describe the decision process used to maximally reduce the database while not losing any sequences.

Recall that NCBI BLAST uses numerous heuristics: the ones selected for a particular database sequence depend on the number of HSPs. In general, the fewer the HSPs, the higher the threshold; and the more the HSPs, the more complex the computation. The first (ideal) filtering method would implement all of these heuristics – in the ungapped case, the final report could then proceed directly from this step. Unfortunately, the overhead for many of them is high: a disproportionate amount of FPGA area would be used to service a small fraction of the candidate sequences. A second method is simply to pass all of the sequences with at least one HSP. We refer to this method as Scheme 1. The problem here is that the HSP threshold has been designed to deal with sequences with 13-14 or more HSPs, and so is artificially low for most significant sequences. When this approach is used, up to 10% of the database can remain.

We use different strategies for the gapped and ungapped cases. For the gapped case, we use the second method (Scheme 1): we keep all sequences with at least one HSP. The reason for allowing this inefficiency is that we make up for it with an FPGA-accelerated Smith-Waterman. Although it is slower than Tree-BLAST, it is more than sufficient to provide efficient gapped filtering on the remaining database.

The ungapped case has no such backup mechanism: the entire remaining database is sent to NCBI BLAST for completion. We therefore use a third method which is based on using only low overhead heuristics.

The major component of the ungapped filter is TreeBLAST. Unlike BLAST, TreeBLAST scans the entire search space without using heuristics and yields scores for all possible alignments in all sequence pairs. Here we add to TreeBLAST a priority queue which, for each database sequence, records the scores of the top n alignments. In our current scheme $n = 5$, but it would go up to the maximum if we implemented all of the NCBI heuristics in the filter.

```
Scheme-2 (optimized)
// Assume: a priority queue of size 5
1)  for each database sequence
2)    if (best_score ≥ cutoff_score) // consider only sig. seqs
3)      if (number of scores over cutoff_score = 1) // single
4)        compute Evalue for single HSP;
5)        if (Evalue ≤ threshold)
6)          pass the sequence;
7)      else // multiple HSPs
8)        if (number of HSPs ≥ 5)
9)          pass the sequence;
10)       else // number of HSPs is 2 – 4
11)         if (best_score's Evalue ≤ threshold)
12)           pass the sequence;
13)         else // compute Evalue for multiple HSPs
14)           compute hypothesis_Evalue using all HSPs;
15)           if (hypothesis_Evalue ≤ threshold)
```

16) pass the sequence;
17) end for;

The optimized scheme addresses the multiple-HSP sequences. There are three cases. First, for the small number of high-HSP-count sequences (≥ 5), we simply pass them to DB'. Second, if there is exactly one HSP, then we compute the $Evalue$ precisely and pass the sequence depending on its value. And third, if the number is 2 - 4, then the $Evalue$ computation is still complex. Rather than computing it precisely, we generate a $hypothesis_Evalue$ (see Appendix), which is guaranteed to be more sensitive than the original, but much simpler to compute. The logic of the third case is shown in the following Lemma.

Lemma: Scheme-2 with the $hypothesis_Evalue$ retains necessary sensitivity (lines 14-16).

Proof: In NCBI BLAST, either a single or multiple $Evalue$s are generated for a sequence with multiple HSPs.

Case 1: Single $Evalue$. NCBI BLAST uses all the HSPs to compute a single $Evalue$.

The $hypothesis_Evalue$ is also computed using all the HSPs (line 14).

Case 2: Multiple $Evalue$s. NCBI BLAST uses various combinations of HSPs to compute a number of $Evalues$. This number, however, does not exceed the total number of HSPs. The optimized scheme always uses all of the HSPs to compute the $hypothesis_Evalue$ (line 14, see Appendix). This is better than any of the multiple $Evalue$s computed in NCBI BLAST. Referring to the code [6]: the term "new_xsum" in function s_BlastEvenGap-LinkHSPs() increases monotonically with additional HSPs, yielding better $Evalue$s.

Scheme-2 is efficient enough so that we are currently running it in software on the host (using scores from the priority queue) with negligible slowdown.

IV. FPGA Algorithm Issues

We describe the necessary steps to get from previously described structures, for TreeBLAST and Smith-Waterman, to a production worthy system. This includes handling a range of cases and certain potential inefficiencies.

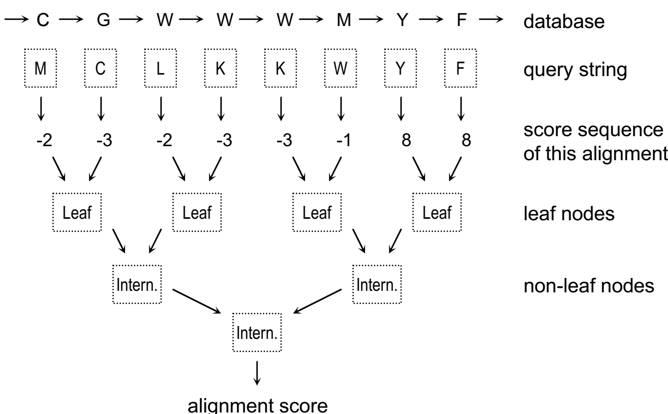

Fig. 3. TreeBLAST structure for $m = 8$.

The basic TreeBLAST structure is the two dimensional systolic array shown in Figure 3. The database sequences are streamed across the leaves of the tree (top) and one complete score sequence (the character-character match scores) is generated every cycle. The score sequences are processed by the tree, which is also pipelined. The score of the best local alignment emerges a few cycles later. This simple structure may need to be varied in four ways.

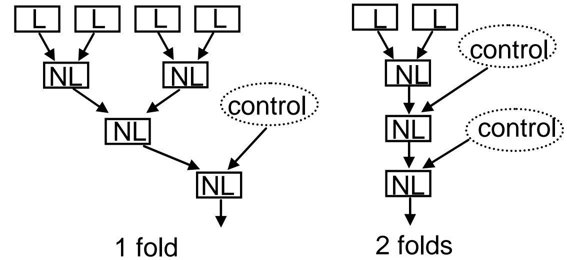

Fig. 4. An 8 leaf tree folded once and twice. L = leaf node, NL = non-leaf node.

Folding. To handle queries larger than can fit on chip, the tree is "folded" (see Figure 4). 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 4, the tree in Figure 3 is folded once. During cycle 1 the first 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 3 shows two folds; processing is analogous.

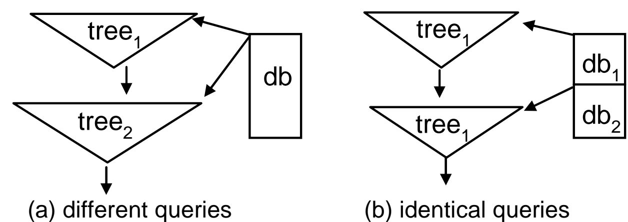

Fig. 5. Replicated trees in two configurations: double throughput for one query, or process two queries.

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 5). Which is used depends on whether the bottleneck lies in the I/O or on the chip.

Tree size $\neq$ power of 2. When queries are not powers of 2, the tree structure is filled out with delay elements as shown in Figure 6. Here a tree of size 1664 is constructed from three trees of size 1024, 512, and 128, respectively. This (and

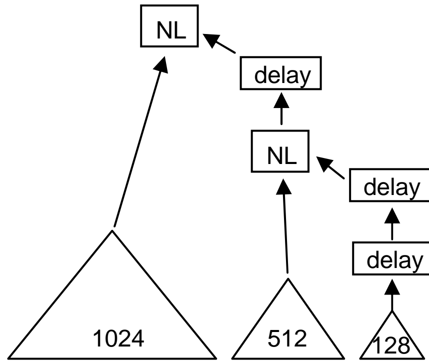

Fig. 6. Tree constructed from variously sized subtrees and delays.

TABLE I
VARIOUS RESULTS FOR CAAD BLAST. AVERAGES FROM RUNNING SEQUENCES OF NR VERSUS NR. ORIGINAL NR DATABASE HAS ABOUT 6M SEQUENCES AND 2.1GB.

NCBI BLASTP	Ungapped	Gapped
exec. time on lab PC	95.9s	99.2s
exec. time on NCBI web server	—	22.5s
# sequences returned	2650	2697
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

folding) works because of the associativity of the TreeBLAST algorithm.

Overlapped sequence processing. A drawback of the tree structure is that only one database sequence can be processed at once. Only one result emerges per cycle; when there are multiple sequences in the tree there is no way to know which the maximal alignment comes from. Correctness can be ensured by inserting m null characters after every sequence, but this degrades performance. The following structure substantially reduces this penalty at a small cost in added logic.

The idea is to sometimes let the subtrees act independently. In this example, let the tree be divided into two subtrees whose roots are the children of the root of the original tree. Then $m/2$ null characters are inserted between two sequences S_1 and S_2 of length n_1 and n_2, respectively. In the first phase, the entire tree processes S_1. When the tail of S_1 enters the tree, null characters begin to be inserted (as before). After $m/2$ null characters (rather than m), we begin inserting S_2 and we enter phase 2. Now the right subtree continues processing S_1, while the left subtree processes S_2. After $m/2$ cycles, S_1 finishes. We return to phase 1 with the entire tree now processing S_2.

Smith-Waterman. We use our own Smith-Waterman that closely follows the version by Wozniak [16]. Folding enables processing of large sequences.

V. RESULTS

A. 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, and the FPGA is configured to the appropriate tree. The sequence is loaded and TreeBLAST executes, returning scores from the FIFO to the host. The host uses the scores to determine the sequences in DB'. If gapped alignment, the FPGA is configured with the appropriate Smith-Waterman array, which then executes using DB' to generate DB". In the final step, DB" (or DB' for ungapped alignment) is formatted and sent to NCBI BLAST.

CAAD-BLAST runs on standard off-the-shelf FPGA-accelerated systems. The FPGA should be sufficiently large such that a tree size of at least a few hundred leaves fits on the chip. The larger Stratix-III and Virtex-5 chips hold trees of size 1000-2000. Memory bandwidth should be sufficient to prevent memory access from being a bottleneck for most queries. For example, the XtremeData XD1000 has a 128-bit interface that streams at 333MHz and thus can drive 24-48 trees, depending on the operating frequency achieved by the FPGA. Given such a memory interface and a large recent FPGA, this would max out the chip (yielding a balanced system) for query sizes of 40-80. In this example, queries larger than that would make the FPGA the bottleneck, while for smaller queries, it would be the memory interface. Memory size should be sufficient to store the database. For NR, which is a large protein database, this is less than 4GB. 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 local memory of typical systems.

B. Validation and Performance

Agreement of results from NCBI BLAST and CAAD BLAST has been validated in two ways. The first is through code analysis, part of which was presented in Section 3.3. The second is from execution. For runs of sequences of NR versus the entire database, all queries have returned the same high scoring sequences; the scores themselves have also been identical.

Table I contains various results from the filter and reference runs. Our primary reference machine is a 2008 64-bit 3GHz Xeon quad processor (Harpertown X5412) with 8GB of memory. We are currently running NCBI BLAST 2.2.17 for reference and for the base code of CAAD BLAST. We compiled 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 these are averages; there is variance as expected from sequences of widely varying sizes.

- Scheme 1 refers to the use of the HSP $cutoff$ as the threshold. NR is reduced by a factor of 17.
- Scheme 2 refers to the $hypothesis_E value$ method in Section 3.3. 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 TreeBLAST and Smith-Waterman on the XtremeData XD1000. The FPGA is an Altera Stratix-II EP2S180. For memory there is 4GB of DRAM with a 128-bit interface that runs at 333MHz. We can fit a tree of size 900 running at 100MHz. For Smith-Waterman, the array size is 200 at 67MHz. Neither design has been optimized: as they are regular they should be able to run substantially faster. For query sequences smaller or larger the tree is either replicated or folded, with proportional effect on performance. For average queries (around 300 amino acids), the 2.1GB NR database is filtered in just over 9 seconds. Smith-Waterman on NR' runs in about 1/3 that time. Overhead adds 3.15 seconds for a total run time of between 15 and 16 seconds and a speed up of about 6x. As the XD1000 does not support easy reconfiguration of the FPGA, we cannot currently run CAAD BLASTP end-to-end.

We have also implemented the filters through post-place-and-route on the Stratix-III EP3SL340. With no optimization, the sizes of the configurations increase to over 2000 and 350, respectively. This nearly doubles performance over the Stratix-II. Currently the operating frequency of TreeBLAST is 160MHz, but Smith-Waterman is still about the same as the Stratix-II; there is likely to be substantial improvement there as well. With an anticipated new system we should be able to process queries end-to-end in well under 9 seconds, including overhead, for a projected speed-up of 11x.

The ungapped option is of particular interest here: it avoids the Smith-Waterman pass, saving some time and complexity. Also, in the NCBI code, ungapped extension is far simpler than the gapped extension. This means that further improvement in the $hypothesis_Evalue$ algorithm would reduce that overhead, perhaps substantially. Together, these could reduce the CAAD BLAST latency on the ungapped option to not much more than that of TreeBLAST alone: that is, 2.8 seconds on a Stratix-III EP3SL340 at 160MHz.

The results returned by gapped and ungapped options are similar, but not identical. For NR sequences queried versus NR, and returning at least 100 sequences, we found the following. Of the top 100, 85.4% agreed; of those missing, 12.8% were in the next 100; the balance (1.8%) were reported further down. The ungapped option may be preferable in applications where BLAST is used as a function, such as in sequencing, and speed is more important than agreement.

VI. Conclusion

We have described the design and implementation of a high performance BLAST application accelerated with FPGA-based prefiltering. We are able to achieve both exact match with NCBI BLAST output and substantial performance improvement. Running unoptimized on two year old hardware it achieves a factor of 6x speed-up. On a new system and with some care we anticipate at least two times that performance.

The filters are usable with any back end. TreeBLAST alone reduces the NR database by factors of 17 and 296 for the HSP-cutoff-only and $hypothesis_Evalue$ heuristics, respectively. For the gapped option, an additional pass with Smith-Waterman reduces the original database by a factor of 1136. Similarly, this method of integrating a front-end filter can be used by other filters as well.

We have several goals for the next few months. One is to achieve at least 200MHz for TreeBLAST and 130MHz for Smith-Waterman on the Stratix-III. Another is to have CAAD BLASTP fully running on a new system we are currently acquiring. Also, we will continue trying to improve the $hypothesis_Evalue$ heuristic.

Longer term we look to apply the TreeBLAST structure to other problems. The most obvious is BLASTN, which has the same logic. Also, the successive merging is certainly useful for other associative kernels. Another possibility is to integrate the filtering structure more directly into the BLAST code, or into a parallel BLAST.

References

[1] P. Afratis, E. Sotiriades, G. Chrysos, S. Fytraki, and D. Pnevmatikatos, "A rate-based prefiltering approach to BLAST acceleration," in *Proc. IEEE Conference on Field Programmable Logic and Applications*, 2008.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, 1990.

[3] M. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt, "Single pass, BLAST-like, approximate string matching on FPGAs," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2006.

[4] ——, "Single pass streaming BLAST on FPGAs," *Parallel Computing*, vol. 33, no. 10-11, pp. 741–756, 2007.

[5] D. Hoang, "Searching genetic databases on SPLASH 2," in *Proc. FCCM*, 1993, pp. 185–191.

[6] http://blast.ncbi.nlm.nih.gov/Blast.cgi, Accessed 1/2009.

[7] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. Chamberlain, "Mercury BLASTP: Accelerating protein sequence alignment," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 1, no. 2, 2008.

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

[9] D. Lavenier, L. Xinchun, and G. Georges, "Seed-based genomic sequence comparison using a FGPA/FLASH accelerator," in *Proc. IEEE Conference on Field Programmable Technology*, 2006, pp. 41–48.

[10] *Mitrion-Accelerated NCBI BLAST for SGI BLAST*, Mitrionics, Available at www.mitrionics.se/press, Accessed 1/2009.

[11] K. Muriki, K. Underwood, and R. Sass, "RC-BLAST: Towards an open source hardware implementation," in *Proc. International Work. High Performance Computational Biology*, 2005.

[12] L. Roberts, "New chip may speed genome analysis," *Science*, vol. 244, no. 4905, pp. 655–656, 1989.

[13] E. Sotiriades and A. Dollas, "A general reconfigurable architecture for the BLAST algorithm," *Journal of VLSI Signal Processing*, vol. 48, pp. 189–208, 2007.

[14] *Web Site*, Time Logic Corp., www.timelogic.com, Accessed 1/2009.

[15] T. VanCourt and M. Herbordt, "Families of FPGA-based algorithms for approximate string matching," in *Proc. International Conference on Application Specific Systems, Architectures, and Processors*, 2004, pp. 354–364.

[16] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison," *Bioinformatics*, vol. 13, no. 2, pp. 145–150, 1997.

[17] *XD1000 Development System*, XtremeData, Inc., www.xtremedata.com, Accessed 2/2009.