Single Pass, BLAST-Like, Approximate String Matching on FPGAs*

Martin C. Herbordt Josh Model Yongfeng Gu Bharat Sukhwani Tom VanCourt

Department of Electrical and Computer Engineering
Boston University; Boston, MA 02215
EMail: {herbordt|jtmodel|maplegu|bharats|tvancour}@bu.edu

Abstract: Approximate string matching is fundamental to bioinformatics, and has been the subject of numerous FPGA acceleration studies. We address issues with respect to FPGA implementations of both BLAST- and dynamic-programming- (DP) based methods. Our primary contributions are two new algorithms for emulating the seeding and extension phases of BLAST. These operate in a single pass through a database at streaming rate (110 Maa/sec on a VP70 for query sizes up to 600 and 170 Maa/sec on a Virtex4 for query sizes up to 1024), and with no preprocessing other than loading the query string. Further, they use very high sensitivity with no slowdown. While current DP-based methods also operate at streaming rate, generating results can be cumbersome. We address this with a new structure for data extraction. We present results from several implementations.

1 Introduction

Approximate string matching (AM) is essential to many important applications. For example, bioinformatics applications use AM to find similarities between DNA (nucleotide) or protein (amino acid) sequences that have diverged through mutation or in the course of evolution. Hamming distance, the number of differing characters, is one way to measure differences between two strings, but does not tolerate insertions or deletions (*indels*). More generalized scoring is based on the probability of particular character mutations and includes indels; it can be handled using dynamic programming (DP) techniques. These have complexity $O(mn)$ for two strings of size m and n, respectively.

With the exploding size of biological databases, DP algorithms have often proven to be impractical. This has spawned heuristic $O(n)$ algorithms, the most famous being BLAST, as well as a host of hardware implementations, particularly of DP methods [2, 3, 6, 9, 10, 12, 15, 16, 20, 23]. Somewhat surprisingly, perhaps, little of this hardware is in general use.

We now summarize the state-of-the-art in FPGA-based AM. DP-based methods are optimal in the sense that with m processing cells, their complexity is proportional the data transfer rate $O(n)$. Their drawbacks, which have prevented their adoption, are their brittleness and the lack of platforms available to the primary users. The first of these issues has been addressed in another recent study [22], while the latter is rapidly being addressed with the proliferation of FPGA-based computational platforms.

BLAST implementations have so far been based closely on the original algorithm [5, 14, 18]. These are substantially faster than the serial version and allow for easy integration into well established systems. They have two drawbacks, however. The first is that they require multiple passes, versus the single pass of the DP-based methods. The second is that in order to process indels, another pass (e.g., using DP) is required, albeit on only a fraction of the database.

There is another significant difference between the FPGA versions of DP and BLAST. Whereas FPGA BLAST easily returns any number of the highest scoring alignments, FPGA DP only returns one, or at most, a small number.

Solutions to these issues are the subject of this paper. We present two FPGA BLAST algorithms that both operate in a single pass at streaming rate. Both algorithms also have very high sensitivity with no performance penalty. Indels are still handled independently. We also present a structure that can be appended to FPGA DP that efficiently extracts high-scoring local alignments. All of these have been implemented on an FPGA development board with a Xilinx Virtex-II Pro XC2VP70 -5 FPGA.

2 Review and motivation

The discussion in the next two subsections covers well-known material; for more detail, please see, e.g., Durbin et al. [8] or Gusfield [11].

*This work was supported in part by the NIH through award #RR020209-01 and facilitated by donations from Xilinx Corporation. Web: http://www.bu.edu/caadlab.

14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)
0-7695-2661-6/06 $20.00 © 2006 IEEE

2.1 Alignment scoring theory

Sequences, or (more commonly) parts of sequences, are considered to have a possible biological relationship if the scoring procedure outlined here yields a score having statistical significance. Typically, one of the sequences has unknown function (e.g. a hypothesized gene) while the other is the database being searched for matches. We refer to the former as the query sequence of length m and the latter as the database of length n.

Since the query is matched with only part of the database at a time, it is convenient to talk about scoring a possible *alignment* of the two sequences. Frequently, we are interested in the best possible matches of any subsequence of the query with the database, a process called *local alignment*. More precisely, an alignment of two sequences is a one-one correspondence between their characters, without reordering, but with the possibility of a number of insertions or deletions (i.e., gaps or *indels*).

The basis of alignment scoring is that character matches can be scored independently, and then combined into an alignment score. Each possible match has an independently generated score, with positive scores for exact or close matches and negative scores for mismatches. These scores are available *a priori*.

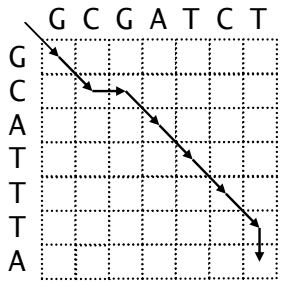
We refer to the sequence of initial character-character scores as the *ScoreSequence* for the alignment. If no indels are considered, then the alignment is said to be *ungapped*, and the alignment score is generated by summing the score sequence. Gaps are handled by adding a penalty per gap based on the length of the gap. Usually the first indel in a gap is assigned a larger penalty than its successors; various, generally simple, functions are used to generate gap penalties.

2.2 Scoring algorithms

A simple procedure for scoring ungapped alignments "slides" the database over the query, and then, for each alignment, computes the score. This results in an $O(mn)$ algorithm. Finding the maximum local alignment can be done with the same complexity using the following procedure:

SimpleScoring
Traverse ScoreSequence
  Add next_score to current_score
  If current_total > max_score, update max_score
  If current total score < 0, set current_total to 0

The naive extension of the above algorithm to deal with gaps has potentially unbounded complexity, but a clever technique based on dynamic programming (DP) reduces the complexity back to $O(mn)$. Variations yield the well-known Needleman-Wunsch and Smith-Waterman algorithms, for global and local alignment, respectively. The basic idea is now described.




```
Alignment: GCGATcT-
           GC-ATtTA
```

Figure 1: Alignment example: Indels are indicated by hyphens; mismatches by lower case.

The Needleman-Wunsch algorithm for aligning two strings is normally presented as a 2D array, such as that shown in Figure 1. Each axis represents one of the strings to be aligned, and steps along each axis represent character positions within the string. Throughout this paper we use the convention that the query string is along the vertical dimension and the database along the horizontal. The algorithm proceeds as if there were a cursor in each string. When both cursors step concurrently, that represents a match in one character position, whether or not the characters in that position are the same. If one string's cursor steps but the other cursor holds its position, that represents a character in the first string being skipped, i.e. a gap being opened in the comparison.

The alignment shown is drawn as one path through the 2D array of possibilities. Finding the highest scoring alignment is an iterative process that scores all cells of the array and determines the highest-scoring path through the array. Comparison starts as if the cursors in the two strings were set to position 0, the position just before the first character in each string. The score $S_{i,j}$ for grid cell (i, j) is computed using the following recurrence relation [8]:

$$S_{i,j} = \begin{cases} if\ (i,j) = 0,0 & 0 & (1) \\ else\ if\ i = 0 & S_{0,j-1} - S_{gap} & (2) \\ else\ if\ j = 0 & S_{i-1,0} - S_{gap} & (3) \\ else & max \begin{cases} S_{i-1,j-1} + s(q_i, r_i) & (4) \\ S_{i-1,j} - S_{gap} & (5) \\ S_{i,j-1} - S_{gap} & (6) \end{cases} \end{cases}$$

Line (1) is the base step of the recurrence, Lines (2) and (3) represent the left end-gap, and Lines (4-6) represent the interior of the array. There, the decision is made to extend the alignment by one position along both strings (4), or to assume a gap in one string or

the other (5 or 6). The comparison function $s(q_i, r_j)$ determines goodness of match between two characters, q_i and r_j. The S_{gap} value represents the penalty for skipping a character in performing the alignment; the more common affine function, $S_{gap} = S_{open} + S_{cont} * len$, only increases the complexity of the recurrence slightly.

The score at the lower right corner, $S_{m,n}$, represents the end-to-end goodness of match between the two strings. When asking the question, "Is string A more similar to B or to C?", the result depends only on the scores for the A/B alignment and the A/C alignment. Other times, however, the experimenter is interested in seeing which parts of the two strings are similar. In that case, a second (traceback) pass is made over the computation array, starting with that final score $S_{m,n}$, and following the highest preceding score back to the origin. Local alignment requires only slight modification to the recurrence relation.

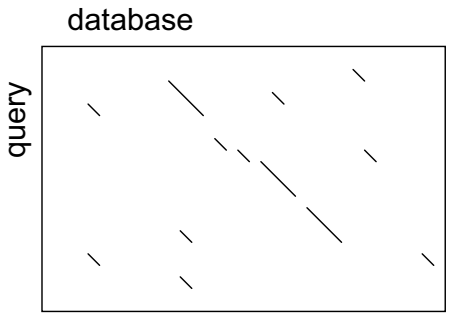


Figure 2: Shown is a tableau formed by all-to-all character matching, and used in DP-based methods. Matches tend to cluster along alignments of biological interest (after Figure 5.5 in Korf, et al. [13]).

Although $O(mn)$ is a remarkable improvement over the naive algorithms, it is still far too great for large databases. A heuristic algorithm, BLAST, generally runs in $O(n)$ time, and is often sufficiently sensitive. BLAST is based on an observation about the typical distribution of high-scoring character matches in the DP alignment tableau (see Figure 2). There are relatively few overall, and only a small fraction are promising. This promising fraction is often recognizable as proximate line segments parallel to the main diagonal.

We now sketch the classic BLAST algorithm [1]. There are three phases: identifying contiguous high scores (parallel to the main diagonal), extending them along the diagonal, and attempting to merge nearby extensions which may or may not be on the same diagonal. The third phase, which accounts for gaps, is nowadays often replaced by a pass of Smith-Waterman on the regions of the database identified as of possible interest. The $O(mn)$ complexity of Smith-Waterman is not as significant when only run on small parts of the database; this (effectively) makes this final pass $O(m^2)$ where $m << n$.

Detail of the first two phases follows. The first is called seeding and identifies positions in the database where a group of contiguous characters (a *word*) have a high match score against a word in the query string. Although the word size W is a variable, the most common sizes are 3 for amino acids and 11 for nucleitides. The seed threshold T is also a variable. The second phase extends the seeds using the SimpleScoring procedure outlined above. There is one difference, however: rather than extending until the current score reaches zero, extension is curtailed when the current score is X (another variable) less than the maximum.

In order to reduce the number of seeds that are extended, many implementations add another filtering step: seeds are not extended unless there is another *collinear* seed within some number of characters (ungapped), usually 40.

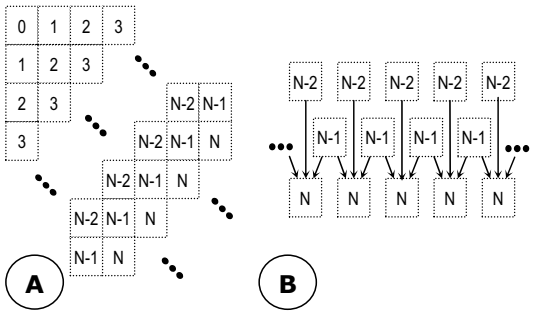


Figure 3: Shown is a dynamic programming based computation array. (A) 2D structure of the computation, showing the order in which grid cells can be evaluated. (B) Linear computation structure corresponding to the evaluatable cells at one time step.

2.3 FPGA algorithms

The $O(mn)$ complexity of the DP algorithms spawned not only heuristic alternatives, but also a raft of special purpose hardware to accelerate the original algorithm [2, 3, 6, 9, 10, 12, 15, 16, 20, 23]. Most implementations follow the construction shown in Figure 3. Because of the dependencies in the DP recurrence, computation can proceed in a wave-front along the diagonal as shown in Figure 3a. Only the computation cells on that diagonal require hardware: Figure 3b shows those computation cells, along with the storage for the previous results on which those computations depend. If the number of cells is greater than m, the size of the query string (see e.g. [22]), the FPGA algorithm runs in $O(n)$. The constant is the time-per-character required to pump the database through the array.

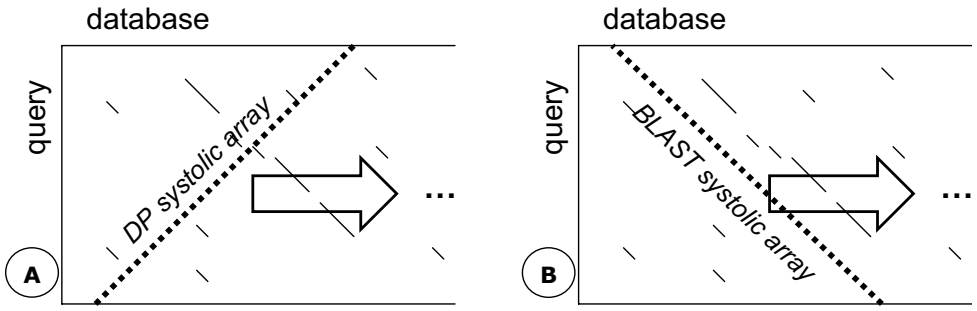


Figure 4: Alignment tableau showing distinction between DP and BLAST systolic arrays. While both hold the query string, and traverse the tableau one character at a time, the flow of the database through the array is reversed.

There have been fewer BLAST implementations, perhaps because the software version is already fast. Still, the importance of the application and the potential for additional performance make its acceleration an important topic. Current published FPGA implementations concentrate on the first two passes and closely follow the serial algorithm [5, 14, 18]. The algorithm used by TimeLogic [21] is not publically available.

3 Single Pass BLAST

3.1 Motivation

From the previous section, it appears that direct FPGA implementations of BLAST (FPGA/BLAST) have a hard time competing with FPGA-based DP (FPGA/DP). FPGA/DP requires only a single pass (and no preprocessing), and handles gaps. In this section we address the former issue with two FPGA/BLAST algorithms that operate at streaming or near streaming rate.

Still, why FPGA/BLAST when FPGA/DP is already so fast? Although it is not possible to be asymptotically faster (as in the serial case), the basic cell turns out to be simpler. This has two consequences: cycle time, and the number of processing cells—and so the size of the query string—that can fit on the chip. Another issue is cultural: BLAST is widely used and well understood.

"BLAST" has been used to describe a variety of algorithms based on the description in the previous section. There are two issues here. The first is that, as the third pass is already the highly efficient FPGA/DP, we do not include that (in this section). As a consequence, gaps need not be considered. The second issue is that the algorithms in this section make some of the sensitivity parameters irrelevent. In particular, in the algorithms below, many aspects of very high sensitivity are achieved with no impact on performance. This has multiple benefits. It yields an even more drastic improvement in performance over serial performance with comparable settings. Also, alignments are likely to be returned that have been missed with sensitivity parameters set at their nominal levels.

3.2 Algorithm basics

Before describing the algorithms themselves, we make an observation about a fundamental distinction between DP and BLAST-like methods. The DP wavefront keeps track of the highest scoring m paths, independent of their twists and turns; therefore, the DP systolic array is perpendicular to the main alignment diagonal (as shown in Figure 4a).

In contrast, BLAST-like methods look for matches (seeds), and then extend these seeds *along* the alignment diagonals. Reducing this to first principles, we could do the equivalent work (with much more processing, but with maximal sensitivity) by successively processing each alignment diagonal in its entirety, e.g., by using **SimpleScoring**. A sketch of a systolic implementation is shown in Figure 4b. The result, however, would be $O(m)$ processing time for each of n alignments; impossibly slower, even than the serial BLAST algorithm. It is possible, however, to create a streaming FPGA/BLAST based on this structure, as we now show. We now present two basic ideas, followed by a "strawman" algorithm.

1. The structure in Figure 4b is used to perform m character matches in parallel and so generate, in a single cycle, the *ScoreSequence* for a particular ungapped alignment. The hardware to implement this for typical queries is a fraction of a typical high-end FPGA. The only non-obvious detail is that, since the query string is held in place, only a single column of the matching array needs to be associated with each element, not the entire table (i.e. 20 entries rather than 400).

2. The hardware to implement **SimpleScoring** consists primarily of the m cells needed to store the *ScoreSequence* as it awaits processing. The processing then

takes m cycles.

SimpleScoring2 – TwoDSystolicBLAST
Construction: One m-length one-dimensional match-scoring array and m copies of a SimpleScoring processor, each with associated m-length FIFOs.
On each cycle i:

1. Generate the ScoreSequence for alignment i
2. Transfer the ScoreSequence to the $i\%m$th FIFO
3. Foreach of m SimpleScoring processors, process the next score.

This algorithm clearly performs ungapped alignments with maximal sensitivity and at the streaming rate of one entire alignment per cycle. Just as clearly, the requirement of m^2 register elements makes it impractical for FPGAs in the forseeable future. We now show two ways of addressing this problem and reducing the logic requirement, including computational storage (but not including the database!) to $O(m)$.

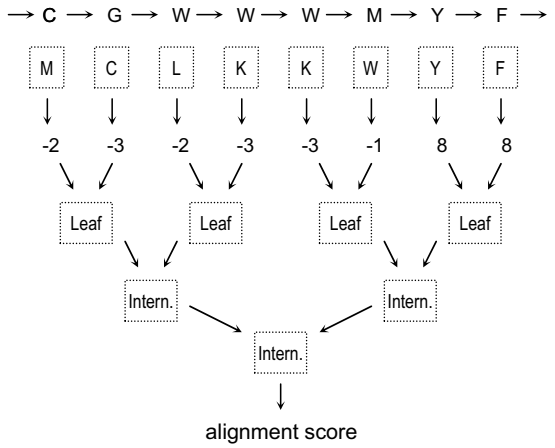


Figure 5: Shown is the TreeBLAST algorithm. The systolic array holds the query string while the database flows through systolically. Scores are evaluated by pipelined, level-by-level, tree traversal.

3.3 TreeBLAST

3.3.1 Description

The key idea behind TreeBLAST is that SimpleScoring can be performed with iterative merging using a tree structure (as shown in the lower part of Figure 5), and that the tree nodes require only a small amount of logic. Further, the tree structure can be pipelined level-by-level. As a result, ungapped alignment scores of maximum sensitivity are generated every cycle. Most significantly, only $m - 1$ nodes are required; these fit on current FPGAs for all but the largest queries.

As with TwoDSystolicBLAST, TreeBlast begins with a one dimensional systolic array that outputs broadside the m character-character ScoreSequence. These match scores are then iteratively combined into subsequence scores using the following logic. Note that only four words of storage are required, but that there are two different node types. The latter reflects the nature of the algorithm, with basis and induction steps.

Procedure TreeBLAST

```
Structure Node
 LeftRunScore  // score of run being extended
               // to the left.  if 0, then no
               // active run
 RightRunScore // same for right extension.
 MaxScore  // score of maximal local alignment
           // within sequence covered by this
           // node.
 Sum  // Sum of all character scores within
      // sequence covered by this node

LeafNode // Generate the first level of tree
         // from input scores (left & right)
 Sum = left + right
 IF (left >= 0 && right >= 0)
  RightRunScore = LeftRunScore = MaxScore = Sum
 IF (left < 0 && right < 0)
  RightRunScore = LeftRunScore = MaxScore = 0
 IF (left >= 0 && right < 0)
  LeftRunScore = MaxScore = left
  IF (SUM > 0) RightRunScore = Sum
  ELSE RightRunScore = 0
 IF (left < 0 && right >= 0)
  RightRunScore = MaxScore = right
  IF (Sum > 0) LeftRunScore = Sum
  ELSE LeftRunScore = 0

NonLeafNode // Generate Left and Right
 LeftRunScore = Max(Left.LeftRunScore,
                 Right.LeftRunScore+Left.Sum)
 RightRunScore = Max(Right.RightRunScore,
                  Left.RightRunScore+Right.Sum)
 Max = Max(Left.Max,Right.Max
         Left.RightRunScore+Right.LeftRunScore)
 Sum = Left.Sum + Right.Sum
```

3.3.2 Proof of correctness

The idea is that a small amount of information about any ScoreSequence is sufficient to characterize, with that same information, a concatenation between two such sequences. As this information includes the score of the maximum local alignment, this procedure is sufficient to find the maximum local alignment within any sequence constructed by pair-wise concatenation of any number of subsequences. We begin with some definitions. The score of sequence x is denoted with "$|x|$." Basic is the idea of a run. A run is a sequence being evaluated with, say SimpleScoring, that currently has a positive score, and so can be extended by a further merge. Somewhat tricky is that runs can possibly be

extended in either direction.
cover ≡ The subsequence of the original ScoreSequence that is "covered" by a node.
maximum ≡ A subsequence that comprises the maximally scoring local alignment within a sequence. The maximum for a cover can be null.
lrun, rrun ≡ Sequences of characters that, if extended, could result in a new maximum. Runs have direction: an lrun extends to the left from a maximum, an rrun extends to the right. The right end character of an rrun is the right end character of the sequence. The opposite is true of lruns.
remainder, runmax ≡ A run consists of two parts: the sequence runmax comprising a local maximum (perhaps, but not necessarily, *the* maximum); and a remainder sequence that causes its score to be $<$ | runmax |.
Also, $0 \leq |\text{run}| \leq |\text{maximum}|$ and any $|\text{remainder}| \leq 0$.
When we concatenate two sequences to form a new sequence, we refer to attributes xxx of the the inputs as L.xxx and R.xxx, and of the output as N.xxx.

Theorem: TreeBLAST performs the ungapped alignment shown in SimpleScoring in a single pass and O(m) space. The score of the maximum local ungapped alignment for the alignment of the query sequence with the $m + i$th m-length subsequence of the database appears in variable *Max* of the root node in cycle $m + i + \log m + 1$.
Proof: Following the algorithm, we use an induction with a basis and induction steps. The basis step is executed by the leaf nodes, the induction steps by the internal nodes. The proof builds up through four Lemmas: the first three refer to the inductive (merging) steps, the fourth to the basis step. We begin by examining what happens in a simplified case: merging an arbitrary lrun and rrun into lrun‖rrun.

Lemma 1: For two runs L.rrun and R.lrun that are merged to form the sequence L.rrun‖R.lrun:
|N.maximum| =
Max(|L.rrun‖R.lrun|,|R.lrunmax|,|L.rrunmax|).
ProofL1: Clearly |R.lrunmax| and |L.rrunmax| are possible values for |N.maximum|. No other subsequences of L.rrun or R.lrun are candidates: if they were, then they would be the L.rrunmax and R.lrunmax respectively. All other candidates are extensions of either L.rrun or R.lrun into the other sequence. Lemma 1 claims that of all the extensions of, say, L.rrun, only the entire concatenation with R.lrun can be N.maximum. The proof requires showing that no extension of L.rrun, other than all of R.lrun, can have a higher score than the terms shown. The other direction is analogous.

Using the above notation, we partition R.lrun and rewrite the concatenation as follows: L.rrun‖R.lremainder‖R.lrunmax. There are now three cases to be considered. It must be shown that N.maximum cannot be any of the following:
1. L.rrun concatenated exactly with R.lremainder.
2. L.rrun extended a character within R.lremainder.
3. L.rrun extended through R.lremainder to a character within R.lrunmax, but short of the end.

Case 1: R.lremainder < 0, so appending all of it to L.rrun does give N.maximum.
Case 2: We rewrite R.lremainder as an arbitrary concatenation R.l.lremainder‖R.r.lremainder. Now for Case 2 to be true,
$|\text{L.rrun}| + |\text{R.l.lremainder}| >$
$|\text{L.rrun}| + |\text{R.l.lremainder}| + |\text{R.r.lremainder}| + |\text{R.lrunmax}|$
But then $0 > |\text{R.r.lremainder}| + |\text{R.lrunmax}|$ and $|\text{R.r.lremainder}|$ would be more negative than $|\text{R.lrunmax}|$ is positive. This is impossible, however, since it implies that R.lrun would have been—at some point during its construction—extended through a negative score, which violates the definition of local alignment.
Case 3: We rewrite R.lrunmax as the concatenation R.l.lrunmax‖R.r.lrunmax. For Case 3 to be true,
$|\text{R.l.lrunmax}| > |\text{R.l.lrunmax}| + |\text{R.r.lrunmax}|$.
But this is impossible because then $|\text{R.r.lrunmax}| < 0$ and |R.lrunmax| would not be maximal.

Lemma 2: For two runs L.rrun and R.lrun that are being merged to form the sequence L.rrun ‖ R.lrun, the score of the new rrun is given by
|N.rrun| =Max(|R.lrunmax|,|L.rrun‖R.sum|)
and R.lrunmax = R.maximum = R.rrun.
ProofL2: Either N.rrun is L.rrun extended all the way through R.lrun, or it is R.rrun. The latter is the same as R.lrunmax and R.maximum. With this lemma, the following two starting points of N.rrun are not allowed: (i) somewhere in L.rrun other than at the beginning, and (ii) somewhere in R.lrun other than the beginning of R.lrunmax. For (i), this would require discarding part of L.rrunmax; for (ii), this would mean adding part of R.lremainder. The rest of the argument follows in the style of **ProofL1**.

Lemma 3: Given two arbitrary sequences L. and R., both with known scores for maximum, lrun, rrun, and sum. Then their concatenation yields N. with the same scores computed as indicated in the procedure **TreeBLAST**.
ProofL3: $|\text{N.}| = |\text{L.}| + |\text{R.}|$ is immediate. The other two parts follow from Lemmas 1 and 2 and the induction. Observe that any sequence is the concatenation of the following parts, any of which can be null:
lremainder lrunmax seq1 maximum seq2 negseq2 rrunmax rremainder.
If maximum is not the same as rrunmax or lrunmax, then scores of seq1 and seq2 are both more negative than the score of maximum is positive. More importantly, the complexity of the sequence does not change the

induction, i.e., that we know the scores of lrun, rrun, maximum and sum. Then
|N.maximum| =
Max(|L.rrun||R.lrun|,|R.lrunmax|,|L.rrunmax|).
follows by the same method as Lemma 1, and
|N.rrun| =Max(|R.rrun|,|L.rrun||R.sum|)
follows by the same method as Lemma 2.

Lemma 4: The leaf nodes correctly compute, for sequences of size 2, the scores for lrun, rrun, maximum, and sum. The leaf nodes therefore provide the basis for the induction.

ProofL4: Follows by inspection.

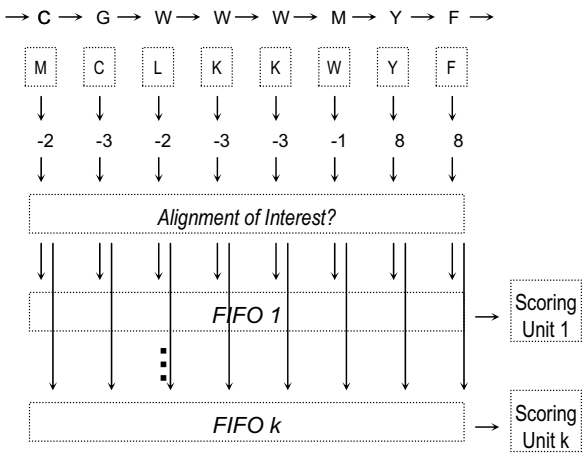


Figure 6: Shown is the structure of the ServerBLAST algorithm. The systolic array holds the query string while the database flows through systolically. Alignments of interest are detected and passed on to the array of scoring servers.

3.4 ServerBLAST

3.4.1 Description

The ServerBLAST structure is shown in Figure 6; it consists of three parts:

1. The same 1D systolic array as in TreeBLAST. The array holds the query string while the database passes through systolically at a rate of one character per cycle. The ScoreSequence for each alignment is determined in a single cycle and fed broadside to the next stage.

2. Alignment of interest (AOI) checker. The scoring sequence is evaluated in a single cycle to determine whether it is of sufficient interest for extension. Currently, the AOI checker looks for a single seed. The word size W and threshold T are parameters. The AOI checker, together with stage 1, comprise a dynamic instantiation of the BLAST seeding phase. Instead of simply tallying hits in a data structure, however, we forward the entire ScoreSequence directly to stage 3 for immediate processing.

3. k "servers" as in **SimpleScoring2**. If the ScoreSequence represents an AOI, then the entire ScoreSequence is fed broadside into the next available server. This stage emulates the BLAST extension phase.

The idea behind ServerBLAST is to stream the database through the systolic array, while the k servers process only the relatively infrequent AOIs. As long as the servers are not all busy, the database streams without pausing. If all the servers are busy and an AOI is encountered, the database stream must wait until a server becomes free.

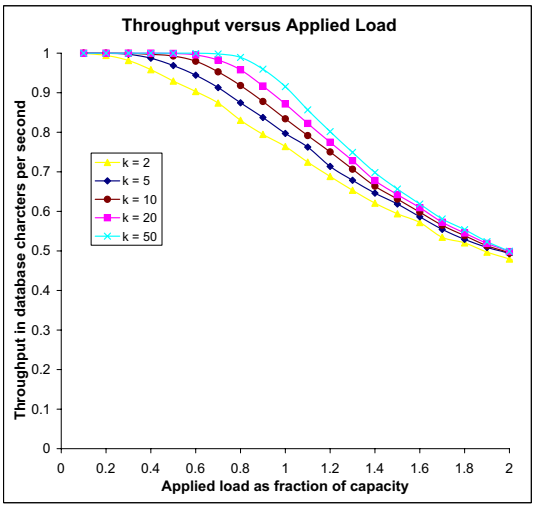


Figure 7: Shown is a graph of ServerBLAST performance: throughput versus applied load for various numbers of servers k.

3.4.2 Performance

ServerBLAST depends on the classic BLAST assumption of low seed rates. We now examine how this assumption applies here. We first examine what the performance is with certain stochastic assumptions in a queuing simulation. Then we see how these are met.

The service time T_s is deterministic and equal to the length of the query string. The arrival rate λ is the frequency of AOIs per alignment and depends on the database and query. For the queuing simulation, we assume exponentially distributed arrivals.

In the first approximation, if $\lambda < k/T_s$, then ServerBLAST may operate at nearly full streaming rate. Whether it actually does or not depends on the distribution of AOIs in the database. If $\lambda > k/T_s$, then ServerBLAST reduces to a k-way parallel processor: the total processing time of the database is equal to the total number AOIs $\times T_s/k$.

We observe that ServerBLAST resembles an MG/k queuing system with finite queues (size = 1). Queuing systems with multiple servers generally flatten

0-7695-2661-6/06 $20.00

the applied load versus average latency graph, and ServerBLAST is no exception. What we are interested in, however, is somewhat different: the average latency per alignment.

We now describe Figure 7, a graph of throughput versus applied load for various k from 2 to 50. The query size (T_s) = 512, but the graphs do not change appreciably over plausible sizes. Capacity is the maximum throughput of AOIs; the applied load is normalized to fraction-of-capacity and is = $\lambda \times T_s/k$. Note that unlike systems with infinite queues, it makes sense for the arrival rate to be higher than the service rate. For all k, we observe the "first approximation" behavior: what varies is where the transition is made from fully streaming (throughput = 1 alignment/cycle), to $k - way$ server (throughput = 1 / (applied-load-as-a-fraction-of-capacity)). For all k, the transition has been made completely by the time applied load approaches 2 times capacity. Systems with $k = 2$ begin the transition around .4; for higher k, the transition is not begun until the applied load approaches 1.

For a real simulation, we aligned the LIM2 protein (about 200 AAs) against the *e coli* database (1.7M AAs). We used the BLOSUM62 scoring matrix, and set the seed threshold at 14. We found that approximately one out of eleven alignments was an AOI. The distribution was somewhat more regular than exponential with the effect of moving the transition towards a higher applied load. Note that for the applied load to be less than the capacity, k should be at least 20. This is plausible for high-end current generation FPGAs. A k of 10, however, is still sufficient to achieve a throughput of .5 alignments per cycle.

For higher λ, or substantial larger amino acid query strings (nucleitide units are much smaller), a different AlignmentofInterest? design is appropriate. The most obvious alternative is to require two seeds per alignment, rather than only one. A simple structure is possible that (effectively) sets the BLAST seed separation parameter to 0. If the new AlignmentofInterest? unit is pipelined through p stages, then the additional chip area is equal to p of the k servers. We currently estimate fewer than 5 cycles this unit. Given that it reduces λ by a large factor (more than 10× in our *e coli* simulation), this is the likely way to go.

4 Filtering

The FPGA/BLAST algorithms of the previous section have the additional advantage that high scoring results are easily extracted. For TreeBLAST, a priority queue is simply appended to the root of the tree; for ServerBLAST, the structure is only slightly more complicated. DP-based methods, however, are not amenable to such simple structures. The reason (illustrated in Figure 2) is that the FPGA/DP array processes m alignments simultaneously; in contrast, FPGA/BLAST only processes one. As a result, the FPGA/DP array is sufficient to retrieve only a single maximum, not the other highest scoring alignments.

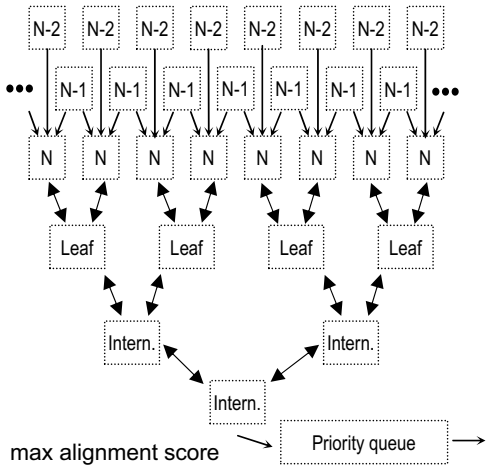


Figure 8: Shown is the DP filtering structure appended to the DP systolic array. The highest local alignments percolate towards the root where they are entered into a priority queue.

This situation is rectified by appending a *priority tree* to the DP systolic array from Figure 3 (as shown in Figure 8). The inputs to the leaf nodes are the current local maxima in each node of the systolic array. On each cycle, the scores in each child node are compared with that in the parent node, and exchanged if necessary so that the maximum of the three ends up in the parent node. This way, the highest alignment scores percolate to the root of the tree. The root score is then entered into a priority queue as is done with the FPGA/BLAST algorithms.

The number of local alignments retrieved, as with the FPGA/BLAST algorithms, is limited only by the size of the priority queue and the frequency with which it is drained. Note that it is possible for the priority tree to lose high scoring local alignments, but only with very low probability. For this to happen, several independent high scoring local alignments would need to be generated in adjoining leaf nodes on the same cycle.

The area cost of the *priority tree* is significantly less than the base FPGA/DP array.

5 Implementation and Results

Our primary target system is an Annapolis Microsystems WildstarII-Pro board with two Xilinx Virtex-II Pro XC2VP70 -5 FPGAs. Only one of the FPGAs is currently used. The board has a 133MHz 32-bit

PCI interface and, for these experiments, is housed in a Dell workstation-class PC with a 2.8GHz Xeon processor, 2GB of memory, and running Windows XP. The databases are stored on a 150 GB IDE-connected NTFS drive. Database sequences are streamed using DMA code from the Annapolis Micro Systems software library. The routines used for disk I/O and performance measurement are the C++ fstream libraries and "Dskspd" utility from Microsoft. We have also configured TreeBLAST onto a Xilinx XC4VLX160 through post place-and-route. None of the designs has been optimized beyond using good digital design practices: e.g., no floor planning has been done.

The primary metrics for the FPGA component of performance are size of the query that can fit on chip, the operating frequency, and, in the case of ServerBLAST, the number of servers that fits on chip. We have so far considered only protein alignment; nucleic acid alignment is far simpler, but in some cases may require more resources per query. For protein queries, we have a target size of 1024: this appears to support the vast majority of such queries [7]. In any case, the query size is based on basic biology and so unlikely to change much, whereas the next generation of FPGAs will allow us effortlessly to increase the hardware resources applied. Also, it is often possible to partition large queries (e.g., for database versus database queries) into constituent sequences.

TreeBLAST. For TreeBLAST, we index the scoring arrays using the block RAMs (BRAMs). By using dual porting and overlapping the indexing, we score four amino acid pairs per block RAM per cycle; therefore the BRAM count limits the query size to slightly over 1200. The limiting factor, however, is the number of slices. For the VP70, a tree can be built that supports queries up to size 600 with cycle time of 9ns, yielding performance of 110M database amino acids per second (Maa/sec). We have also implemented TreeBLAST with a "folded tree." This allows us to use each systolic array cell for multiple character evaluations, with a proportional reduction in throughput. In the singly-folded version, queries of up to size 1200 can be processed at 55Maa/sec.

We have also implemented TreeBLAST on a Xilinx XC4VLX160 through post place-and-route. Here, we achieve our target sequence size of 1024 without folding. The clock delay is 5.6ns for a throughput of 178 Maa/sec. This last design uses 90% of the slices, 88% of the block RAMs, and 78% of the lookup tables.

ServerBLAST. Recall that ServerBLAST uses the same front end as TreeBLAST. The other critical resource comes from trying to maximize k, the number of servers. For query size of 600, we easily achieve $k = 5$ and 100Mhz operating frequency. For smaller queries, such as LIM2/*e coli* (previously described), k can be much larger.

DP. In previous work we implemented a large number of variations of FPGA/DP [22]. Perhaps the most "vanilla" of these holds a query of size 150 and has an operating frequency of 40MHz. The database is processed at one character per cycle.

DP-plus-filtering. Adding the priority tree filtering network yields an operating frequency of 33MHz and a query size of 120.

ServerBLAST, DP, and DP-plus-filtering should all benefit analogously to TreeBLAST when implemented on the Xilinx XC4VLX160.

Operation of TreeBLAST-based alignment. Query sequence, database, and scoring matrix are specified by the user. The FPGA is initialized with the query sequence and scoring matrix. The database is streamed from disk or memory through the FPGA; high scores and their corresponding database position are returned. Conversion from raw scores to final output is done using code derived from the NCBI source. Planned, but not yet completed, the highest-scoring sequences will be sent on to the second FPGA on the board for gapped alignment using a DP algorithm.

Validation. We chose to simulate blastp. Ecoli was the source of our query strings, and drosophilia the subject (as per http://www.bioperf.org/). Ten sequences from ecoli were chosen at random and concatenated into a query sequence. The drosophila database was left in FASTA format. Parameters were as follows: neighborhood size = 3, threshold for seed extension = 11, dropoff for ungapped extension = 7 bits, reporting cutoff = 66 (Raw Score). Our validation target was NCBI BLAST for Win32 version 2.2.13. Ungapped BLAST was run, and an e-value threshold was chosen to yield the same reporting cutoff (S3 value in the NCBI BLAST report) as our simulator. Since we were not calculating sum statistics for collinear alignments in our simulator, the significant alignments from the NCBI report which passed under the e-value threshold due to a lower sum statistic, rather than a lower raw score, were manually discarded. Under these conditions, identical pairwise alignments were obtained for two sets of 10 queries.

Performance. The total execution time consists primarily of the time to stream the database from either memory or disk through the FPGA. Memory versus disk depends on the relative sizes of memory and database. In our configuration, we achieved a transfer rate from disk to FPGA of 55MB/sec, and memory to FPGA of 320 MB/sec. As long as the query fits on chip, latency = database size/"bottleneck rate." In our configuration (VP70 -5), queries with respect to the FASTA NR amino acid database (1.8GB) were processed in the following times: 16.4 seconds (direct) and 32.8 seconds (folded). The estimated time for the same configuration using an XC4VLX160 is 12.4 seconds. Smaller databases are proportionally faster. Pa-

rameter settings are equivalent to most sensitive.

Comparison with alternatives. NCBI BLAST was installed on the same PC. Queries of size 1K required 7 minutes, 37 seconds format time and 9 minutes, 23 seconds run time. Only a small fraction of this time was used for disk access. The same queries to the NCBI server using the standard web interface required between 20 and 30 seconds. The NCBI server consists of several hundred CPUs [17]. Individual queries are processed by multiple (but not all) resources; databases are often cached in memory [19].

6 Discussion and Future Work

We have presented two algorithms for accelerated FPGA/BLAST, as well as an extension to FPGA/DP that eases data extraction. Our hardware implementations show that most common queries, as observed in the workload at the NCBI servers [7], fit on current generation FPGAs. The BLAST cell is somewhat smaller and simpler than the DP cell: FPGA/BLAST queries can be four times larger, and FPGA/BLAST operation is three times faster, than FPGA/DP. On the other hand, FPGA/DP handles gaps. Combining TreeBlast with FPGA/DP, where the latter handles only the highest scoring alignments, provides high performance gapped alignment.

In previous work we determined FPGA/DP to be 150× to 400× faster than PC implementations. Such a determination is harder with BLAST: performance is highly workload dependent, both in query size and selection; e.g., the latter allows databases to be cached in memory. Even so, for cases in which the FPGA implementation is at a disadvantage, we have achieved performance comparable to that of the dedicated server farm at NCBI. And, the performance of the FPGA-based system is independent of sensitivity.

Work remains in analyzing the biological implications of increased sensitivity.

The extension to larger systems (i.e., for higher throughput and lower response time) is immediate: queries and the databases are partitioned across multiple PC/FPGA components. The methods described here are also compatible with integrated approaches, as is being carried out Muriki et al. [18], and with embedding directly into an I/O device [4].

Performance limits of this approach are determined by disk I/O, FPGA interface, and FPGA operating frequency. Standard system techniques should enable disk I/O rates of 1GB using conventional desktop systems. The FPGA interface in a dedicated system (e.g. NUMALink in the SGI/RASC) is advertised to support this rate. Current generation FPGAs are likely to be limited to 500MHz, however, setting the overall limit per FPGA at 500Maa/sec. A realistic per-FPGA goal, given current technology, is perhaps half that.

References

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

[2] Bluethgen, H.-M., and Noll, T. A programmable processor for approximate string matching with high throughput rate. In *Proc. ASAP* (2000), pp. 309–316.

[3] Borah, M., Bajwa, R., Hannenhalli, S., and Irwin, M. A SIMD solution to the sequence comparison problem on the MGAP. In *Proc. ASAP* (1994), pp. 336–345.

[4] Chamberlain, R. Embedding applications within a storage appliance. In *Proc. HPEC* (2005).

[5] Chang, C. BLAST implementation on BEE2.

[6] Chow, E., Hunkapiller, T., and Peterson, J. Biological information signal processor. In *Proc. ASAP* (1991).

[7] Coulouris, G. BLAST benchmarks. NCBI/NLM/NIH Presentation, June 2005.

[8] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. *Biological sequence analysis*. Cambridge University Press, Cambridge, U.K., 1998.

[9] Dydel, S., and Bala, P. Large scale protein sequence alignment using FPGA reprogrammable logic devices. In *Field Programmable Logic and Applications* (2004).

[10] Guccione, S., and Keller, E. Gene matching using JBits. In *Proc. FPL* (2002), pp. 1168–1171.

[11] Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge U. Press, Cambridge, U.K., 1997.

[12] Hoang, D. Searching genetic databases on SPLASH 2. In *Proc. FCCM* (1993), pp. 185–191.

[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. ASAP* (2004), pp. 365–375.

[15] Liptov, R., and Lopresti, D. Comparing long strings on a short systolic array. In *Systolic Arrays*, W. Moore, A. McCabe, and R. Uquhart, Eds. Adam Hilger, 1986.

[16] Lopresti, D. P-NAC: A systolic array for comparing nucleic acid sequences. *IEEE Computer 20*, 7 (1987).

[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] Muriki, K., Underwood, K., and Sass, R. RC-BLAST: Towards an open source hardware implementation. In *Proceedings of the International Workshop on High Performance Computational Biology* (2005).

[19] NCBI. Programming with BLAST: BLAST scripting. NCBI/NLM/NIH Presentation, February 2005.

[20] Roberts, L. New chip may speed genome analysis. *Science 244*, 4905 (1989), 655–656.

[21] Time Logic Corp. *Web Site*. www.timelogic.com, 2003.

[22] VanCourt, T., and Herbordt, M. Families of FPGA-based accelerators for approximate string matching. *Microprocessors and Microsystems in press* (2006).

[23] Yu, C., Kwong, K., Lee, K., and Leong, P. A Smith-Waterman systolic cell. In *Field Programmable Logic and Applications* (2003), pp. 375–384.

0-7695-2661-6/06 $20.00