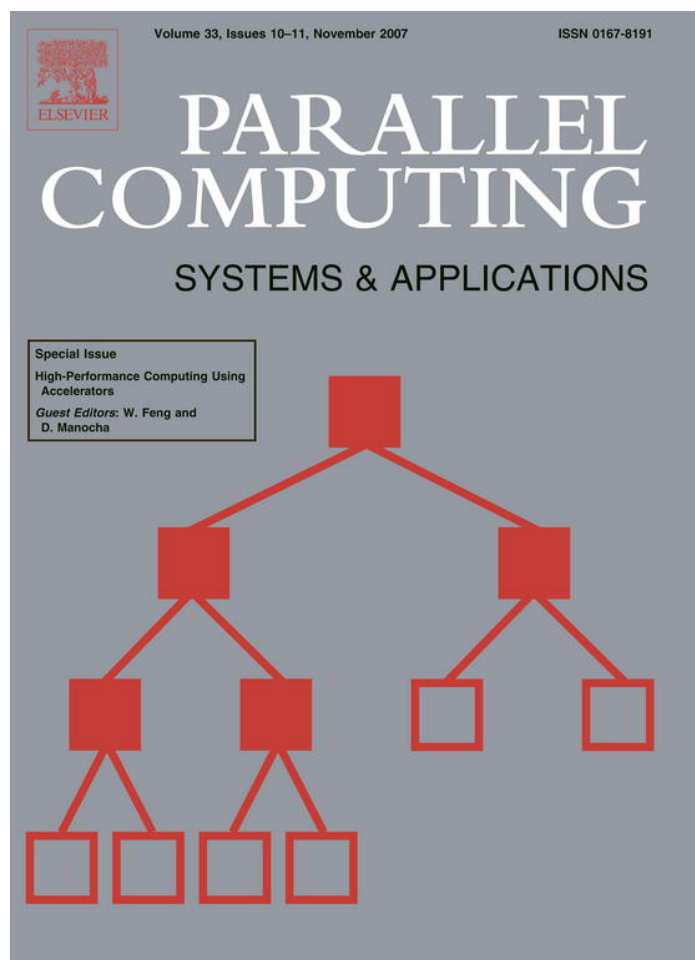


Provided for non-commercial research and education use.  
Not for reproduction, distribution or commercial use.



This article was published in an Elsevier journal. The attached copy is furnished to the author for non-commercial research and education use, including for instruction at the author's institution, sharing with colleagues and providing to institution administration.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

<http://www.elsevier.com/copyright>



ELSEVIER

Available online at [www.sciencedirect.com](http://www.sciencedirect.com)

Parallel Computing 33 (2007) 741–756

---



---

**PARALLEL  
COMPUTING**


---



---

[www.elsevier.com/locate/parco](http://www.elsevier.com/locate/parco)

# Single pass streaming BLAST on FPGAs <sup>☆,☆☆</sup>

Martin C. Herbordt <sup>\*</sup>, Josh Model <sup>1</sup>, Bharat Sukhwani, Yongfeng Gu,  
Tom VanCourt <sup>2</sup>

*Department of Electrical and Computer Engineering, Boston University, Boston, MA 02215, United States*

Received 4 May 2007; received in revised form 14 September 2007; accepted 24 September 2007

Available online 1 October 2007

---

## 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 contribution is a new algorithm for emulating the seeding and extension phases of BLAST. This operates in a single pass through a database at streaming rate, and with no preprocessing other than loading the query string. Moreover, it emulates parameters turned to maximum possible 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 showing order of magnitude acceleration over serial reference code. A simple extension assures compatibility with NCBI BLAST.

© 2007 Elsevier B.V. All rights reserved.

*Keywords:* Configurable computing; High performance computing; Bioinformatics; Computational accelerators

---

## 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

---

<sup>☆</sup> This work was supported in part by the NIH through awards #RR023168-01 and #RR020209-01, and facilitated by donations from Xilinx Corporation, SGI, and XTremeData, Inc.

<sup>☆☆</sup> A preliminary version of this work was presented at the 2006 IEEE Symposium on Field-Programmable Custom Computing Machines.

<sup>\*</sup> Corresponding author.

*E-mail addresses:* [herbordt@bu.edu](mailto:herbordt@bu.edu) (M.C. Herbordt), [jtmodel@bu.edu](mailto:jtmodel@bu.edu) (J. Model), [bharats@bu.edu](mailto:bharats@bu.edu) (B. Sukhwani), [maplegu@bu.edu](mailto:maplegu@bu.edu) (Y. Gu), [tvancour@bu.edu](mailto:tvancour@bu.edu) (T. VanCourt).

*URL:* <http://www.bu.edu/caadlab> (M.C. Herbordt).

<sup>1</sup> Now at the MIT Lincoln Laboratory.

<sup>2</sup> Now at Altera Corporation.

(*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 [1], as well as a host of hardware implementations, particularly of DP methods [3,4,7,10,11,14,18,19,22,27]. 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 to 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 [25], 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 [6,15,17,20]. 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. Further discussion of related work appears in Sections 2 and 6.

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 a new FPGA BLAST algorithm, TreeBLAST, that operates in a single pass at streaming rate. Significantly, TreeBLAST emulates BLAST with that program's parameters turned to maximum possible 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 and evaluated for performance and validity.

A preliminary version of this work was presented at the 2006 IEEE Symposium on Field-Programmable Custom Computing Machines [13]. There are many additions: the integration of TreeBLAST into a complete FPGA-based BLAST, an extension of TreeBLAST to handle large queries, performance improvements, additional validation experiments, a new (simplified) proof of TreeBLAST correctness, and an appreciation of the importance of our DP results extraction structure.

## 2. Review and motivation

The discussion in the next two subsections covers well-known material; for more detail, please see, e.g., [9] or [12].

### 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-to-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 character 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 – for evaluating one alignment

Traverse ScoreSequence

    Get next character match score into next\_score

    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.

The Needleman–Wunsch algorithm for aligning two strings is normally presented as a 2D array, such as that shown in Fig. 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 (see, e.g., [9]):

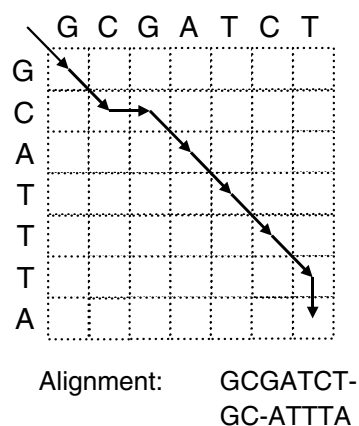


Fig. 1. Alignment example: indels are indicated by hyphens; mismatches by lower case.

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

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  $S_{\text{gap}}$  value represents the penalty for skipping a character in performing the alignment. The more common affine function for computing multiple successive skips,  $S_{\text{gap}} = S_{\text{open}} + S_{\text{cont}} * \text{len}$  – where  $S_{\text{open}}$  is the penalty for opening a gap and  $S_{\text{cont}}$  the penalty for continuing a gap – 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.

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 Fig. 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; e.g., Krishnamurthy et al. [17], find that, for a set of BLASTn experiments, the final pass accounts for well under 1% of the run time. This (effectively) makes the final pass  $O(m^2)$  where  $m \ll n$ .

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 parameter, the most common sizes are 3 for amino acids and 11 for nucleitides. The seed threshold  $T$  is also a parameter. The second phase extends the seeds using the SimpleScoring proce-

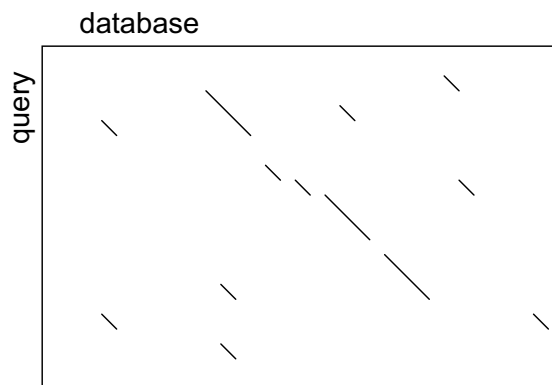


Fig. 2. Shown is a tableau formed by all-to-all character matching, and used in DP-based methods. Matches tend to cluster along alignments of biological interest (after Fig. 5.5 in [16]).

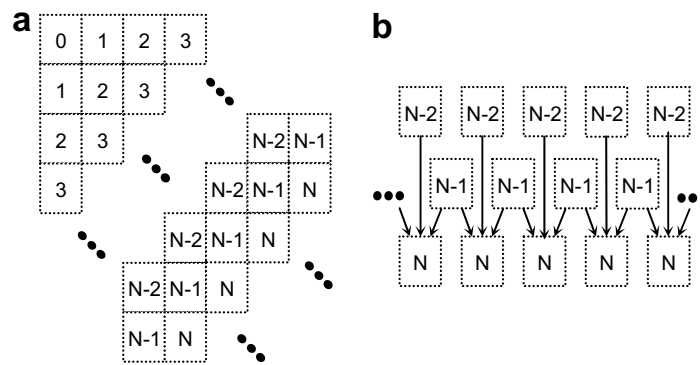


Fig. 3. Shown is a dynamic programming based computation array. (a) 2D structure of the computation, showing the order in which grid cells can be evaluated. (b) Linear computation structure corresponding to the evaluable cells at one time step.

dure 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 parameter) 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.

### 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 [3,4,7,10,11,14,18,19,22,27]. Most implementations follow the construction shown in Fig. 3. Because of the dependencies in the DP recurrence, computation can proceed in a wavefront along the diagonal as shown in Fig. 3a. Only the computation cells on that diagonal require hardware: Fig. 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. [25]), the FPGA algorithm runs in  $O(n)$ . The constant is the time-per-character required to pump the database through the array.

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 [6,15,17,20]. The algorithm used by TimeLogic [24] 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 pre-processing), and handles gaps. In this section we address the former issue with an FPGA/BLAST algorithm that operates at streaming rate.

Still, why FPGA/BLAST when FPGA/DP is already so fast? Although it is not possible to be asymptotically faster (e.g., a reduction from  $O(mn)$  to  $O(n)$  in the serial case), the basic cell turns out to be simpler. This has two consequences: a reduction in cycle time, and an increase in 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 algorithm in this section makes some of the sensitivity parameters irrelevant. In particular, 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 when sensitivity parameters have been set at their nominal levels.

### 3.2. Algorithm basics

Before describing the algorithm itself, we make an observation about a fundamental distinction between DP- and BLAST-based 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 Fig. 4a).

In contrast, BLAST-based 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 Fig. 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 present two basic ideas, followed by a “strawman” algorithm. The result is that processing is performed with a throughput of one alignment per cycle (per database stream).

1. A structure along the diagonal shown in Fig. 4b is used to perform  $m$  character matches in parallel and so generate, in a single cycle, the *ScoreSequence* for a particular ungapped alignment (see Fig. 5). The hardware to implement this for typical queries is a fraction of a 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., for proteins, 20 entries rather than 400).
2. The hardware to implement each of  $m$  copies of **SimpleScoring** consists of an  $m$ -length queue needed to store the *ScoreSequence* as it is processed, plus some arithmetic logic to perform the sums and compares. The processing then takes  $m$  cycles. See Fig. 5.

#### SimpleScoring2 – $m$ -waySystolicBLAST

Construction: One  $m$ -length one-dimensional match-scoring array and  $m$  copies of a SimpleScoring processor, each with an  $m$ -length queue.

On each cycle  $i$ :

1. Generate the *ScoreSequence* for alignment  $i$
2. Transfer the *ScoreSequence* to the  $i \% m$ th FIFO
3. For each of  $m$  SimpleScoring processors, advance the queue to process the next character match score with the associated Scoring Unit.

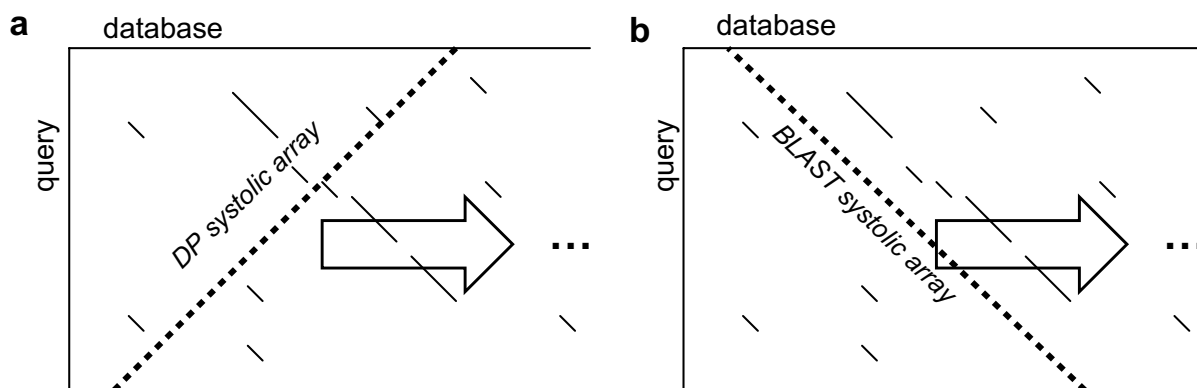


Fig. 4. Alignment tableau showing distinction between DP and BLAST systolic arrays. While both hold the query string, and traverse the tableau one character at a time, the flow of the database through the array is reversed.

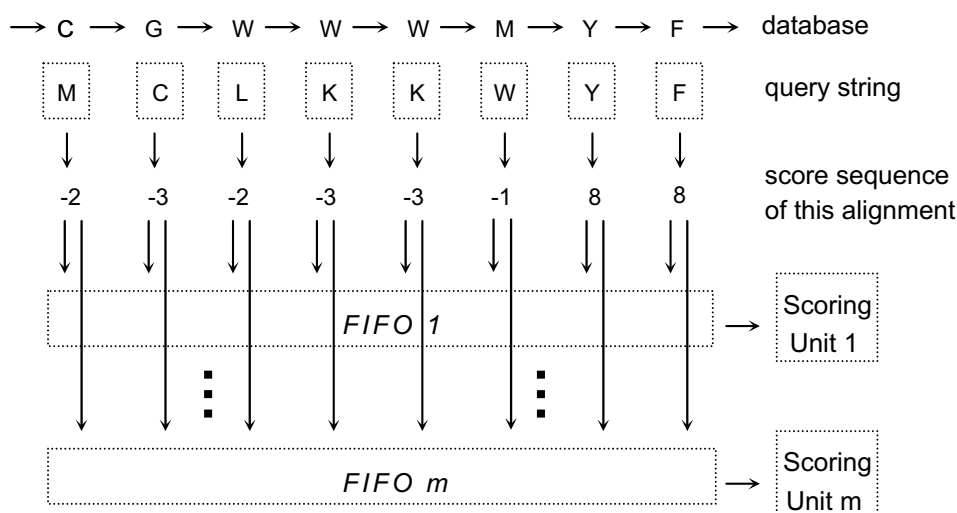


Fig. 5. Shown is the SimpleScoring2 algorithm. The systolic array holds the query string while the database flows through systolically. Each alignment is scored by one of the queue/processor units.

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 foreseeable future for sequence lengths of biological interest. We now show how to address this problem and reduce the logic requirement, including computational storage, to  $O(m)$ .

### 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 Fig. 6), 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 most queries. For large queries, the tree can be folded; for small queries, the tree can be replicated.

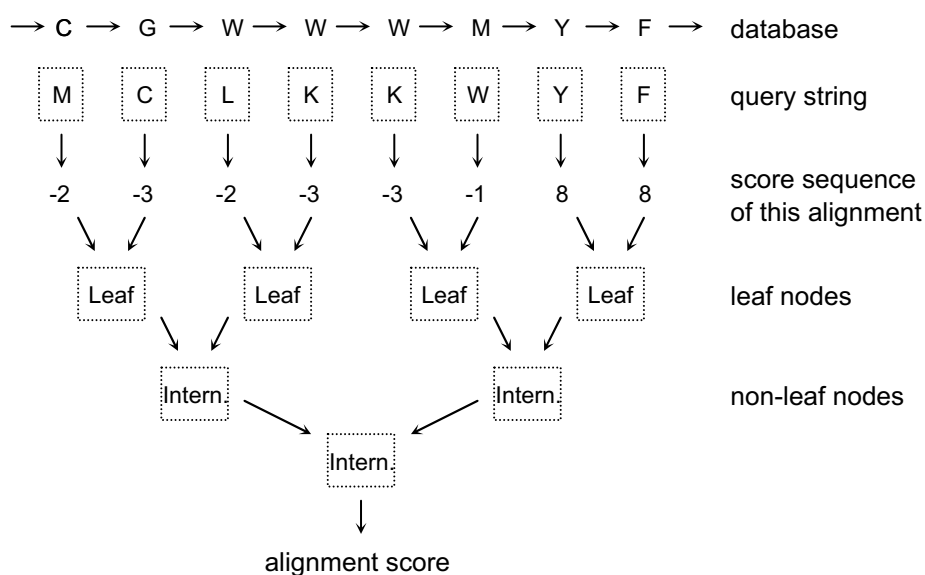


Fig. 6. Shown is the TreeBLAST algorithm. The systolic array holds the query string while the database flows through systolically. Scores are evaluated by pipelined, level-by-level, tree traversal.

As with TwoDSystolicBLAST, TreeBlast begins with a one-dimensional systolic array that outputs broad-side 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 step. We begin with some definitions.

Run  $\equiv$  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 be extended in either direction.

Cover  $\equiv$  The subsequence of the original ScoreSequence that is “Covered” by a node.

Maximum  $\equiv$  A subsequence that comprises the maximally scoring local alignment within a sequence. The Maximum for a Cover can be null.

LeftRun, RightRun  $\equiv$  Sequences of characters that, if extended, could result in a new Maximum. Runs have direction: a LeftRun extends to the left, a RightRun extends to the right. The right-most (left-most) character of a RightRun (LeftRun) is the right-most (left-most) character of the sequence.

Remainder  $\equiv$  In sequences with a LeftRun (RightRun), the part of the sequence that is not in the LeftRun (RightRun).

When we concatenate two sequences to form a new sequence, we refer to attributes xxx of the inputs as Left.xxx and Right.xxx, and of the output as New.xxx.

### Procedure TreeBLAST

#### Structure Node

Sum // Sum of all character scores within sequence covered by this node

MaxScore // Score of maximal local alignment within sequence covered by this node.

LeftRunScore // Score of run being extended to the left. If 0, then no active run.

RightRunScore // Same for right extension.

LeafNode // Generate the first level of tree from a  
// pair of input scores (Left & Right)

Sum = Left + Right

LeftRunScore = MAX(Left, Sum, 0)

RightRunScore = MAX(Right, Sum, 0)

MaxScore = MAX(Sum, Left, Right, 0)

NonLeafNode // Merge Left and Right nodes

LeftRunScore = MAX(Left.LeftRunScore, Right.LeftRunScore + Left.Sum)

RightRunScore = MAX(Right.RightRunScore, Left.RightRunScore + Right.Sum)

MaxScore = 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 constant amount of information about a ScoreSequence of any length 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.

**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 MaxScore of the root node in cycle  $m + i + \log m + 1$ .*

**Proof.** Following the algorithm, we use an induction with basis and induction steps. The basis step is executed by the leaf nodes, the induction steps by the internal nodes. In both parts we show that the computation of Sum, LeftRunScore, RightRunScore, and MaxScore is correct:

**Basis Step:**

**Sum** – By definition,  $\text{Sum} = \text{Left} + \text{Right}$ .

**LeftRunScore (RightRunScore is analogous)** – There are five possible cases which generate three possible values:

1. Left and Right are both positive.  
Then  $\text{LeftRunScore} = \text{Right} + \text{Left}$
2. Right is positive and Left is negative with  $\text{ABS}(\text{Right}) > \text{ABS}(\text{Left})$ .  
Then  $\text{LeftRunScore} = \text{Right} + \text{Left}$
3. Left is positive and Right is negative.  
Then  $\text{LeftRunScore} = \text{Left}$
4. Left and Right are both negative.  
Then  $\text{LeftRunScore} = 0$
5. Right is positive and Left is negative with  $\text{ABS}(\text{Right}) < \text{ABS}(\text{Left})$ .  
Then  $\text{LeftRunScore} = 0$

**Maximum** – There are four cases depending on whether none, one (Left or Right), or both of Left and Right contribute.

**Induction Step:**

**Sum** – By definition,  $\text{Sum} = \text{Left.Sum} + \text{Right.Sum}$ .

**LeftRunScore** – There are four cases:

1.  $\text{Left.LeftRunScore} > 0$  and  $\text{Right.LeftRunScore} > \text{Left.Remainder}$ .  
Then  $\text{New.LeftRunScore} = \text{LeftRunScore} + \text{Left.Remainder} + \text{Right.LeftRunScore} = \text{Left.Sum} + \text{Right.LeftRunScore}$
2.  $\text{Left.LeftRunScore} > 0$  and  $\text{Right.LeftRunScore} \leq \text{Left.Remainder}$ .  
Then  $\text{New.LeftRunScore} = \text{Left.LeftRunScore}$
3.  $\text{Left.LeftRunScore} = 0$  and  $|\text{Right.LeftRunScore}| > |\text{Left.Sum}|$ .  
Then  $\text{New.LeftRunScore} = \text{Left.Sum} + \text{Right.LeftRunScore}$
4.  $\text{Left.LeftRunScore} = 0$  and  $|\text{RightRunScore}| \leq |\text{Left.Sum}|$ .  
Then  $\text{New.LeftRunScore} = 0$

Referring to Fig. 7, in (a) we see case 2 while in (b) we see cases 1 and 3.

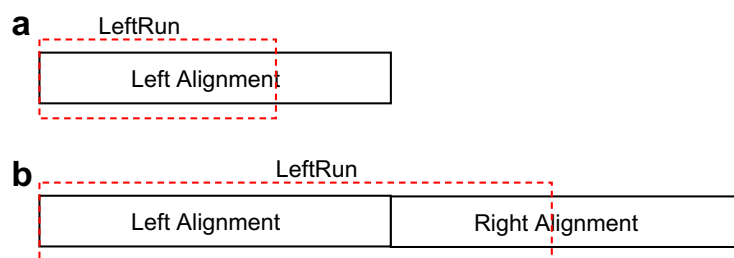


Fig. 7. Shown are the two possible non-trivial terms for computing the LeftRunScore.

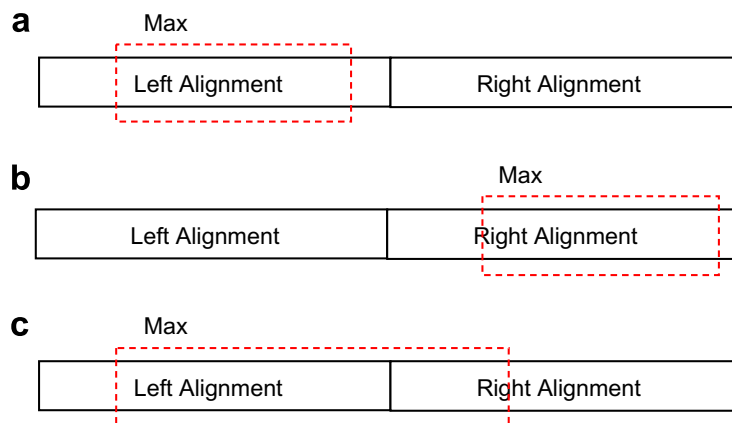


Fig. 8. Shown are the three possible non-trivial terms for computing the MaxScore.

**MaxScore** – There are three cases:

1. New.Maximum is entirely within Left.  
Then  $\text{New.MaxScore} = \text{Left.MaxScore}$
2. New.Maximum alignment is entirely within Right.  
Then  $\text{New.MaxScore} = \text{Right.MaxScore}$
3. New.Maximum overlaps Right and Left.  
Then New.Maximum must necessarily consist of the concatenation of following two alignments: (i) the maximum alignment in Right that can be obtained by starting at the right-most value and moving left, and the maximum alignment in Left that can be obtained by starting at the left-most value and moving right. Since these two alignments are exactly  $\text{Left.RightRun}$  and  $\text{Right.LeftRun}$ , respectively, the result of this case is  $\text{New.MaxScore} = \text{Left.RightRunScore} + \text{Right.LeftRunScore}$ .

Referring to Fig. 8(a), (b), and (c) illustrate cases 1, 2, and 3, respectively.  $\square$

#### 4. Extracting results from the Smith–Waterman structure

The FPGA/BLAST algorithm of the previous section has the additional advantage that the highest-scoring result for each ungapped alignment is easily extracted: a priority queue is simply appended to the root of the tree. DP-based methods, however, are not amenable to such simple structures. The reason (illustrated in Fig. 2) is that the FPGA/DP array processes  $m$  alignments simultaneously; in contrast, FPGA/BLAST only processes one.<sup>3</sup> This is a necessary consequence of being able to exhaustively score all *ungapped* alignments. As a result, the FPGA/DP array is sufficient to retrieve only a single maximum, not the other highest-scoring alignments.

This issue becomes critical when DP is used to process TreeBLAST alignments. Recall that the first two passes of BLAST, and the TreeBLAST emulation, are used to indicate the foci of interest to be examined further by a DP pass. The problem is that, unlike BLAST, TreeBLAST does not distinguish among (possibly) *multiple* “good” scoring subsequences (local alignments) that could occur within a single alignment, returning only the score of the highest. If DP also only returns a single score, then information critical to the BLAST user may have been lost: in typical BLAST usage, not just the maximum alignment is considered important, but rather some number of those scoring highest.

This situation is rectified by appending a *priority tree* to the DP systolic array from Fig. 3 (as shown in Fig. 9). The inputs to the leaf nodes are the current local maxima in each node of the systolic array. On each

<sup>3</sup> In TreeBLAST, the processing of multiple alignments can overlap, as is done in our pipeline implementation, but in a way that is easily separable.

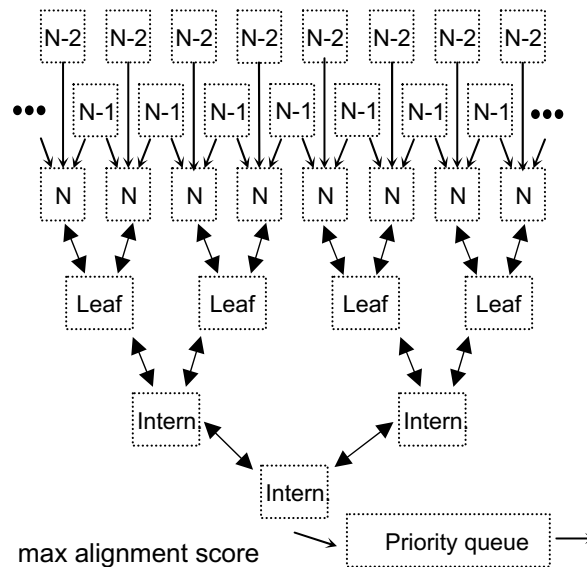


Fig. 9. Shown is the mechanism for extracting results from the DP appended to the DP systolic array. The highest local alignment scores percolate towards the root where they are entered into a priority queue.

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

### 5.1. Basic operation and generic FPGA implementation issues

#### 5.1.1. Basic operation

The 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, their corresponding database sequences, and positions in those sequences are returned. The highest-scoring sequences (HSSs) are sent on to the (reconfigured) FPGA for gapped alignment using Smith–Waterman. The HSSs passed from TreeBLAST to Smith–Waterman are filtered using the same scoring threshold that is used by NCBI BLAST. Conversion from raw scores to final output is done using code derived from the NCBI source.

As previously described, TreeBLAST uses no heuristics and so returns a superset of results returned by standard BLAST codes. There is some concern that only those alignments be returned that would be from, say, NCBI BLAST. This is accomplished by adding a pass on the host that uses NCBI BLAST itself. The original query is reprocessed with respect to just the highest-scoring sequences, and unwanted sequences are necessarily removed. For reasonable thresholds and non-trivial databases, the set of returned alignments is <.01% of the database size; the increased processing is therefore negligible.

#### 5.1.2. Generic FPGA implementation issues

In previous sections, the algorithms are presented independent of both target hardware and character–character scoring function. We now consider some issues in implementing TreeBLAST on a generic FPGA and for

sequences of biological interest. Here we describe protein alignment; nucleic acid alignment is simpler and uses the same basic mechanisms. There are three issues: character–character scoring to derive the score sequence, processing the score sequence to obtain the alignment score, and streaming the data base through the FPGA.

1. For proteins, character–character scoring is a table look-up into a preselected 2D scoring matrix. On an FPGA, for each character in the query sequence, the corresponding column of the scoring matrix is held in one of the hundreds of independently addressable block RAMs (BRAMs) available on typical high-end FPGAs. There are two optimizations. One is scoring multiple characters with each look-up, rather than only one. The size of the look-up table increases exponentially with the number of characters; with current FPGA BRAMs, two-character tables fit easily, but those for three characters do not. The second optimization uses the multiple ports of the BRAMs to enable two look-ups per BRAM per cycle. Combining these optimizations allows for the scoring of four pairs of amino acids per BRAM per cycle.
2. The logic for processing the score sequence follows directly from Procedure TreeBLAST in Section 3.3.1, with the following addition and optimization. The size of the processing tree depends on the amount of logic available on the FPGA, a quantity typically only loosely related to the number of BRAMs. Therefore, tree size is only coincidentally related to the size of the query and to the size of the score sequence that can be processed per cycle. There are two cases, depending on whether the query is larger or smaller than the available processing logic. Queries that are larger than the tree size are handled by folding the tree. This allows the use of each systolic array cell in the tree for multiple character evaluations, with a proportional reduction in throughput. The number of folds is limited only by the availability of buffer space; for current BRAMs, at least 32-1 folding is reasonable. For queries smaller than half the tree size, a simple optimization is possible: In this case the tree can be replicated and multiple streams processed in parallel.
3. The rate the database is streamed through the FPGA depends on the number of streams and the throughput per stream. Since the I/O capacity of modern FPGAs is high, with several hundred signal pins and numerous multi-Gb communication connections, the streaming rate is likely to be limited by the capability of the system to get the database to the FPGA.

## 5.2. Sample implementation

The target system for our experiments is an Annapolis Microsystems WildstarII-Pro board with two Xilinx Virtex-II Pro XC2VP70 -5 FPGAs, although only one is used. This board is housed in a Dell workstation-class PC with a 2.8 GHz Xeon processor, 2GB of memory, and running Windows XP. One of the FPGAs is used for TreeBLAST, and then reconfigured for the DP pass. The board has a 133 MHz 32-bit PCI interface. The databases are stored on a 150GB 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.

On the VP70, the BRAM count limits the query size that can be handled without folding to slightly over 1200. The limiting factor for this chip, however, is the number of slices, which reduces that number to 600. The cycle time is 9 ns. We have also implemented TreeBLAST on a Xilinx XC4VLX160 through post place-and-route. Here, we obtain a query size of 1024 without folding, and a clock of 5.6 ns. This last design uses 90% of the slices, 88% of the block RAMs, and 78% of the lookup tables.

For DP, in previous work we implemented a large number of variations [25]. Perhaps the most “vanilla” of these holds a query of size 150 and has an operating frequency of 40 MHz. The database is processed at one character per cycle. Adding the priority tree filtering network yields an operating frequency of 33 MHz and a query size of 120. DP and DP-plus-filtering should both benefit analogously to TreeBLAST when implemented on the Xilinx XC4VLX160.

### 5.3. Verification and validation

#### 5.3.1. Verification

The raw scores generated by TreeBLAST were verified through various reference programs, including NCBI BLASTp.

#### 5.3.2. Validation

For validation we investigate TreeBLAST output with respect to NCBI BLAST. This is an issue because TreeBLAST performs only ungapped alignment, and because it is not possible for TreeBLAST to emulate NCBI input parameters such as word size and threshold, TreeBLAST sensitivity being maximal by default. For operation we assume the procedure outlined in Section 5.1: TreeBLAST, filtering of HSSs, then Smith–Waterman.

1. For any given alignment, are the *E*-values returned by the TreeBLAST-based system the same as those returned by NCBI BLAST?

The *E*-value, or expectation value, describes how often an alignment with a given score is expected to occur at random. Matching *E*-values for a given alignment is critical because this measure provides a common statistical basis for comparison among all possible alignments. *E*-values depend on query and database compositions and the raw score, with query and database parameters typically being computed off-line. It therefore suffices to return identical alignments (and therefore raw scores), and to use the same *E*-value computation to guarantee correct *E*-values.

2. In comparison with NCBI BLAST, does TreeBLAST miss any alignments? Because of its inherently maximal sensitivity, we expect TreeBLAST to return more alignments than one-hit ungapped NCBI BLAST; the opposite would indicate a problem.

We begin by describing the process algorithmically, then describe specific experiments. TreeBLAST scores all possible segment pairs; HSSs are selected using the same filtering used by NCBI BLAST. NCBI BLAST on the other hand, uses heuristics to select a subset of segment pairs for scoring. Therefore, if TreeBLAST has been implemented correctly, it should not miss any alignments returned by NCBI BLAST.

For validation, we emulate BLASTp. We compare with both the standard two-hit algorithm [2] as well as the older one-hit version. In both cases the query source was *E. coli* and the subject the non-redundant protein database nr. Fifteen source sequences were chosen at random. Reporting cutoffs were chosen to return the top 100 matches. Also for both, the parameters were as follows: neighborhood size = 3, threshold for seed extension = 11, dropoff for ungapped extension = 7 bits. For all experiments, TreeBLAST returned all alignments returned by NCBI BLAST, and with identical score and location. In comparison to the one-hit algorithm, TreeBLAST typically returned two additional alignments not returned by NCBI BLAST; for the two-hit algorithm, this number is typically 5–10.

3. In comparison to the first two passes of NCBI BLAST, does TreeBLAST send too many alignments to the DP pass? By eliminating the filtering, will the DP phase be overwhelmed with work?

We find that this is not the case. Algorithmically, the reason is that the filtering of HSSs is unrelated to how candidate HSSs are generated. In the same experiments as just described, we find that the one-hit algorithm only sends less than 20% more HSSs to the DP pass than the two-hit algorithm. Streaming FPGA/DP easily handles this increased workload.

### 5.4. Performance

We begin by discussing performance of implementations of the TreeBLAST algorithm. Performance of the entire TreeBLAST-based system is described below. As TreeBLAST is a streaming algorithm, its throughput depends simply on the bandwidth of the tightest bottleneck. We divide the considerations into two categories, getting streams to the FPGA, and streaming data through the FPGA; we begin with the latter.

Bandwidth through the FPGA depends on the operating frequency, the number of streams, and for queries larger than the tree size, the number of folds. As stated above, for the Xilinx Virtex-2 VP70 queries up to size 600 can be processed without folding at 110 MHz, for a throughput of 110 M database amino acids per second

(Maa/s). In folded versions, queries of up to size 1200 can be processed at 55 Maa/s, up to size 2400 at 22.5 Maa/s, and so on. For the V4 LX160 (in simulation only), queries up to 1024 can be processed at 178 Maa/s, with similar reductions in performance for larger queries. For smaller queries, however, there is a proportional increase in performance. For example, on the V4 for typically sized queries of 300 residues, the throughput is over 500 Maa/s.

Getting streams to the FPGA depends on the overall system, with many possible alternatives. In our configuration, we achieved a transfer rate from disk to FPGA of 55MB/s, and from host memory to FPGA of 320MB/s. Queries with respect to the FASTA nr amino acid database (1.8GB), which fits in host memory, were processed in the following times: 8.2 s (query size 200–300), 16.4 s (query size 300–600), and 32.8 s (query size 600–1200). If the VP70 were replaced with a Xilinx Virtex-4 LX160, we estimate the processing time of, e.g., queries of size 300–500 to be less than 6 s; the limit here is the bandwidth of the I/O bus. In both cases, queries with respect to smaller databases are proportionally faster.

Given the simplicity of the algorithm and the accurate characterization of basic parameters in FPGA systems, it is possible to estimate the performance of TreeBLAST in a tightly coupled system such as, e.g., the XtremeData XD1000 [26], the SGI RASC RC100 [23], or the SRC SR-7 [21]. Each of these has sufficient memory proximate the FPGA to hold a protein database such as nr, and sufficient memory/FPGA bandwidth to support 16 byte-wide streams. For expected workload, we examine statistics gathered by NCBI [8] for BLASTp. For database selection, 80% of queries are with respect to nr; the distribution of query sizes also follows the distribution of sequence sizes in nr with 300 being the average and over 90% of queries having fewer than 1000 residues. We weight these potential queries by size; assuming a Xilinx Virtex-2 VP100, queries of 513–1024 have weight 1, 257–512 weight .5 (two trees), 1025–2048 weight 2 (a single fold), etc. and obtain a weighted average of .61. This yields an average response time of 10 s. For large numbers of sequences tighter packing is possible (e.g., large queries with small). Then the average number of streams increases from 1.6 to 2.3 and the average response time reduces to 7 s. Table 1 shows results of running an FPGA-bound system such as those just described, for various FPGAs, over a range of query sizes with respect to the nr database. For reference, some sample results from running NCBI BLASTp on a PC (described in Section 5.2) are given. Standard settings were used.

We now extend the performance discussion to complete “end-to-end” BLAST implementations based on TreeBLAST (FPGA/BLAST). Proper performance comparison of FPGA/BLAST with NCBI BLAST run on a generic PC (PC/BLAST) requires accounting for the gapped extension (DP) phase as well as other differences in pre- and post-processing.

The FPGA acceleration of the DP phase has been researched extensively with speed-ups in the hundreds common (see, e.g., [25]). The obvious method of implementing FPGA/BLAST is therefore to have the FPGA process both ungapped and gapped phases. The performance implications are as follows. Depending on the query and the database, and on BLASTp versus BLASTn, PC/BLAST spends from .03% to 34% of the time performing the DP phase [15,17]. One potential concern is that with FPGA accelerators, the critical resource is available logic: both FPGA/BLAST and FPGA/DP benefit from using the whole chip. The time to reconfigure the FPGA, however, is small in comparison to processing of all but the simplest queries. To summarize: For FPGA/BLAST, the total additional processing time for adding the gapped extension is almost always less than 10% of that required for the rest of the computation, and usually less than 1%.

Table 1

Results in seconds of TreeBLAST assuming an FPGA-bound configuration for various query sizes and FPGAs with respect to the nr database

Platform	Query length			
	200	500	1000	2000
Xilinx Virtex-II Pro VP70	8	16	32	64
Xilinx Virtex-II Pro VP100	4	8	16	32
Xilinx Virtex-4 LX160	2.5	5	10	20
NCBI BLAST on a PC	49 ± 3	110 ± 6	163 ± 7	324 ± 9

NCBI BLAST is end-to-end and does not include time required for preprocessing the database; to compare, less than 10% needs to be added to the FPGA numbers (see text).

Two other differences relate to preprocessing. The first is that PC/BLAST requires that the database be preprocessed to create the seed index, a step not required by FPGA/BLAST. Another is the relative time to load the database from disk into memory. For a given disk I/O system, the time is similar for PC and FPGA versions. But because FPGA/BLAST is so much faster, a latency that is negligible for PC/BLAST can dominate in FPGA/BLAST. At least for proteins, however, even large databases (e.g., nr, which also accounts for most queries) fit in memory for common PC configurations. Once loaded, the database can be used for any number of queries.

## 6. Discussion and future work

We have presented an algorithm for accelerated FPGA/BLAST. There are two parts: TreeBLAST, which emulates the first two passes of the common BLAST algorithm, and an extension to the well-known FPGA/DP algorithm that eases data extraction. We are able to support all but the largest queries without splitting, although for queries greater than twice the average (on the Xilinx Virtex-2 VP70, three times the average on the V-2 VP100 and V4 LX160) some slowdown results.

The BLAST cell is somewhat smaller and simpler than the DP cell: FPGA/DP queries must be folded four times as much as FPGA/BLAST queries; also, 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 150–400× faster than PC implementations. Such a determination is harder with BLAST: performance is highly workload dependent, both in query size and selection. Even so, FPGA/BLAST achieves substantial speed-up over the serial version. And, the performance of the FPGA-based system is independent of sensitivity parameter settings.

Since the preliminary version of this work was published [13], Jacob et al. have presented a different version of FPGA/BLAST, Mercury BLASTp [15], based more closely on the original BLAST algorithm. This version obtains similar performance to that described here (both versions have better performance than described in [13]), and allows some manipulation of parameters. Mercury BLASTp makes compromises, some of which are as follows: for performance, word size of 4 is used to emulate a word size of 3; the two-hit algorithm is modeled by a heuristic; and, by certain use of fixed size structures, it may miss some seeds. Still, the correlation – at the level of finding high-scoring database sequences – between NCBI BLAST and Mercury BLASTp is very high.

To compare the approaches: the advantage of Mercury BLAST is its reduction of work in the extension phase through the use of Bloom filters. It is also fully integrated into a production system and well characterized. The advantage of TreeBLAST is its fidelity to the original concept of sequence alignment: because of this it emulates BLAST as if that program were running with highest possible sensitivity; and it does this without loss of performance. Another advantage of TreeBLAST is its simplicity. It remains to be determined whether one or the other will have significantly better performance once both algorithms have been tuned and ported to future generation FPGA technology. For example, relative performance is likely to depend on characteristics of the associated memory system, such as the number of memory banks. That is, Mercury BLAST reduces the amount of data that is examined per query (after preprocessing), but requires random access; TreeBLAST looks at the entire database, but does this entirely with data streams.

We now describe some future work. Important is analyzing the biological implications of increased sensitivity. The extension to parallel systems (i.e., for higher throughput and lower response time) is immediate: queries and databases are partitioned across multiple PC/FPGA components. The methods described here are also compatible with integrated approaches, as is being carried out by Muriki et al. [20], and with embedding directly into an I/O device [5].

## Acknowledgements

We would like to thank A. Jacob and J. Lancaster for pointing out the significance of DP filtering, and for a very useful discussion on various issues in accelerating BLAST with FPGAs. We also thank the anonymous referees for their many helpful comments.

**References**

- [1] S. Altschul, W. Gish, W. Miller, E. Myers, D. Lipman, Basic local alignment search tool, *Journal of Molecular Biology* 215 (1990) 403–410.
- [2] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, D. Lippman, Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, *Nucleic Acids Research* 25 (17) (1997) 3389–3402.
- [3] H.-M. Bluthgen, T. Noll, A programmable processor for approximate string matching with high throughput rate, in: *Proc. ASAP*, 2000, pp. 309–316.
- [4] M. Borah, R. Bajwa, S. Hannenhalli, M.A Irwin, SIMD solution to the sequence comparison problem on the MGAP, in: *Proc. ASAP*, 1994, pp. 336–345.
- [5] R. Chamberlain, Embedding applications within a storage appliance, in: *Proc. HPEC*, 2005.
- [6] C. Chang, C. BLAST Implementation on BEE2, 2004.
- [7] E. Chow, T. Hunkapiller, J. Peterson, Biological information signal processor, in: *Proc. Int. Conf. Application Specific Systems, Architectures, and Processors*, 1991, pp. 144–160.
- [8] G. Coulouris, BLAST benchmarks, NCBI/NLM/NIH Presentation, June 2005.
- [9] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, *Biological Sequence Analysis*, Cambridge University Press, Cambridge, UK, 1998.
- [10] S. Dydel, P. Bala, Large scale protein sequence alignment using FPGA reprogrammable logic devices, in: *Proc. Field Prog. Logic and Applications*, 2004.
- [11] S. Guccione, E. Keller, Gene matching using JBits, in: *Proc. Field Prog. Logic and Applications*, 2002, pp. 1168–1171.
- [12] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Cambridge, UK, 1997.
- [13] M. Herbordt, J. Model, B. Sukhwani, Y. Gu, T. VanCourt, Single pass, BLAST-like, approximate string matching on FPGAs, in: *Proc. Field Prog. Custom Computing Machines*, 2006.
- [14] D. Hoang, Searching genetic databases on SPLASH 2, in: *Proc. FCCM*, 1993, pp. 185–191.
- [15] A. Jacob, J. Lancaster, J. Buhler, R. Chamberlain, FPGA-Accelerated seed generation in Mercury BLASTP, in: *Proc. Field Prog. Custom Computing Machines*, 2007.
- [16] I. Korf, M. Yandell, J. Bedell, *BLAST: an Essential Guide to the Basic Local Alignment Search Tool*, O'Reilly and Associates, 2003.
- [17] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, J. Lancaster, Biosequence similarity search on the Mercury system, in: *Proc. Int. Conf. Application Specific Systems, Architectures, and Processors*, 2004, pp. 365–375.
- [18] R. Liptov, D. Lopresti, Comparing long strings on a short systolic array, in: W. Moore, A. McCabe, R. Uquhart (Eds.), *Systolic Arrays*, Adam Hilger, 1986.
- [19] D. Lopresti, P-NAC: a systolic array for comparing nucleic acid sequences, *IEEE Computer* 20 (7) (1987) 98–99.
- [20] K. Muriki, K. Underwood, R. Sass, RC-BLAST: towards an open source hardware implementation, in: *Proc. Int. Workshop on High Performance Computational Biology*, 2005.
- [21] D. Poznanovic, SRC-7 system characteristics and design considerations, Presentation, Reconfigurable Systems Summer Institute, July 2006.
- [22] L. Roberts, New chip may speed genome analysis, *Science* 244 (4905) (1989) 655–656.
- [23] Silicon Graphics, Inc., SGI RASC RC100 Blade, <[www.sgi.com/pdfs/3920.pdf](http://www.sgi.com/pdfs/3920.pdf)>, 2007.
- [24] Time Logic Corp., Web site <[www.timelogic.com](http://www.timelogic.com)>, 2007.
- [25] T. VanCourt, M. Herbordt, Families of FPGA-based accelerators for approximate string matching, *Microprocessors and Microsystems* 31 (2) (2007) 135–145.
- [26] XtremeData, Inc., XD1000 Development System, <[www.xtremedata.com](http://www.xtremedata.com)>, 2007.
- [27] C. Yu, K. Kwong, K. Lee, P.A. Leong, Smith–Waterman systolic cell, in: *Proc. Field Prog. Logic and Applications*, 2003, pp. 375–384.