FPGA-accelerated seed generation in Mercury BLASTP

**Arpith Jacob, Joseph Lancaster,
Jeremy Buhler, and Roger D. Chamberlain**

Arpith Jacob, Joseph Lancaster, Jeremy Buhler, and Roger D. Chamberlain, “FPGA-accelerated seed generation in Mercury BLASTP,” in *Proc. of 15th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2007.

Dept. of Computer Science and Engineering
Washington University
Campus Box 1045
One Brookings Dr.
St. Louis, MO 63130

BECS Technology, Inc.
9487 Dielman Rock Island Industrial Dr.
St. Louis, MO 63132

FPGA-accelerated seed generation in Mercury BLASTP *

Arpith Jacob[1], Joseph Lancaster[1], Jeremy Buhler[1], and Roger D. Chamberlain[1,2]
{*jarpith,jmlancas,jbuhler,roger*}*@cse.wustl.edu*
[1]Department of Computer Science and Engineering
Washington University in St. Louis, St. Louis, MO 63130
[2]BECS Technology, Inc., St. Louis, MO 63132

Abstract

BLASTP is the most popular tool for comparative analysis of protein sequences. In recent years, an exponential increase in the size of protein sequence databases has required either exponentially more runtime or a cluster of machines to keep pace. To address this problem, we have designed and built a high-performance FPGA-accelerated version of BLASTP, Mercury BLASTP. *In this paper, we focus on seed generation, the first stage of the BLASTP algorithm. Our seed generator is capable of processing database residues at up to 219 Mresidues/second for 2048-residue queries. The full Mercury BLASTP pipeline, including our seed generator, achieves a speedup of* $37\times$ *over the popular NCBI BLASTP software on a 2.8 GHz Intel P4 CPU, with sensitivity more than 99% that of the software. Our architecture can be generalized to accelerate the seed generation stage in other important biocomputing applications.*

1. Introduction

Comparative sequence analysis is widely used in computational biology to study the evolutionary relationship between two sequences. Biologists compare a sequence of unknown function, termed the *query*, against a database of sequences with known function to detect known sequences with high similarity. Similarity between query and database sequences is represented by an *alignment*, such as the example in Figure 1. A good alignment of the two sequences matches up many pairs of identical or biologically similar residues (characters) while keeping dissimilar pairs and unaligned residues to a minimum. Good alignments provide evidence of common ancestry between proteins, which can imply shared structure and function.

```
Query:  QAPGTLIGASRD--EDELPVKGISNLNNMAMFSVS
         |||| |   ||  +| ||| |+|++| + + +++
   Db:  DAPGTRI--ERDVQKDRLPVTGLSSINKVVLLNLA
```

Figure 1. Alignment of two protein sequences. Identical and biologically similar residue pairs are marked by vertical lines and pluses, respectively.

Over the last two decades, databases of known sequences have grown at an exponential rate, driven in part by whole-genome sequencing of many organisms. For example, Figure 2 shows the rapid growth of the TrEMBL protein database [19] caused in part by new proteins inferred from whole-genome DNA sequences. This growth has raised the computational cost of sequence comparison with existing software, to the point that analyzing the proteins from a newly sequenced genome has become a computational bottleneck. For example, consider the status of BLASTP [1], the single most popular protein comparison software tool. The U.S. National Center for Biological Information (NCBI) makes their version of BLASTP available through a public web server. In 2004, this server was backed by a Linux cluster of around 200 CPUs that processed 1.4×10^5 queries on a typical weekday. Furthermore, NCBI planned to double their computing capabilities to keep up with demand [13].

In this work, we address the BLASTP bottleneck by accelerating protein database searches using a combination of FPGAs and general-purpose CPUs. Our implementation, *Mercury BLASTP*, departs substantially from prior accelerators targeting Smith-Waterman, a slower but more accurate algorithm for aligning two sequences [17]. The biological community has accepted BLASTP's results as a *de facto* standard of accuracy, so Smith-Waterman's higher accuracy is of limited interest, given that even hardware-accelerated versions of the latter are no faster than software BLASTP. In contrast, we accelerate the BLASTP algorithm

*This research was supported by NIH/NGHRI grant 1 R42 HG003225-01 and NSF grants CCF-0427794 and DBI-0237902. R.D. Chamberlain is a principal in BECS Technology, Inc.

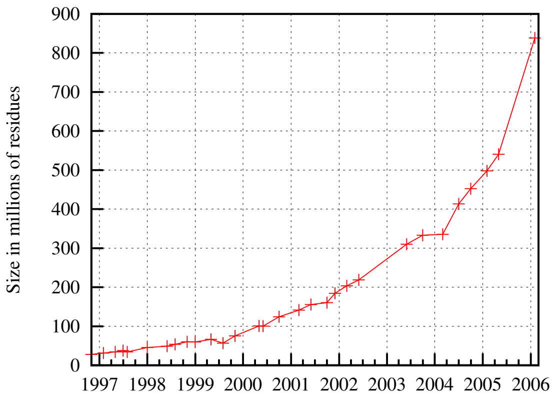


Figure 2. Growth of UniProtKB/TrEMBL protein database.

itself, which is actually a pipeline of several different computations. To maximize acceptance of our accelerator in the biological community, we strive to produce results at least as good as NCBI BLASTP's and to match that implementation's command-line interface and output format.

This paper focuses on accelerating *seed generation*, the first stage of BLASTP. We describe the architecture of our new FPGA-based seed generation stage and analyze its performance as part of the fully implemented *Mercury BLASTP* pipeline. Using our stage design, Mercury BLASTP can scan a protein database at 219 Mresidues/second vs. a 2048-residue query, producing results better than 99% identical to those of NCBI BLASTP. Our design can also be used to accelerate similar computations in other bioinformatics applications, such as HMMERhead [15] and PhyloNet [21].

2. Related work

The Smith-Waterman dynamic programming algorithm for sequence alignment [17] has been extensively targeted for parallelization in hardware. FPGA accelerators for this computation [6, 7, 22] report 1-2 orders of magnitude speedup. Unfortunately, hardware Smith-Waterman, even running $100\times$ faster than a software implementation on current CPUs, run at about the same speed as the NCBI BLASTP software on the same CPUs.

Thanks to commercial availability of large FPGAs, several other recent works have accelerated all or part of the BLAST algorithm family. Not all of these implementations address BLASTP; for example, [18] and [11], and [14] target BLASTN, the version of BLAST for DNA sequences, which requires a somewhat different approach to acceleration. We describe our own BLASTN accelerator, Mercury BLASTN, in [9].

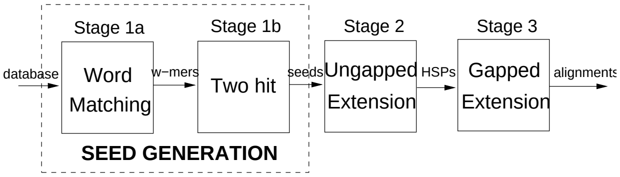


Figure 3. Pipelined stages of BLASTP.

TreeBLASTP [5] is a recent FPGA-based accelerator for BLASTP-like computations. Mercury BLASTP offers higher throughput and larger query support than TreeBLASTP (219 Mresidues/second for 2048-residue queries vs. 110 Mresidues/second for 600-residue queries) on similar FPGA technology.

We note that all the above accelerators may produce significantly different results than the BLAST software. Indeed, none of them report their sensitivity compared to software BLAST. In our experience with the biological community, an accelerated BLAST implementation must not only yield a significant speedup but also produce results similar to or better than the standard software implementations. Biological end users of BLAST have in the past been reluctant to adopt accelerated solutions for fear of missing alignments that might otherwise have been found with NCBI BLAST [3]. Mercury BLASTP therefore aims to closely mimic the results of the standard software. We show below that Mercury BLASTP yields results almost identical to those of NCBI BLASTP on our test computation.

DeCypherBLAST [20] is a commercial product to accelerate BLASTP, utilizing FPGA-based processing engines attached to high-end CPUs. Given the closed nature of DeCypherBLAST, we lack sufficient information to fairly compare its performance to that of Mercury BLASTP.

The majority of high-performance BLAST solutions today are based on multiprocessor clusters [12, 16]. Although BLAST can be made to scale well with cluster size, clusters typically have high acquisition, maintenance, and energy costs when compared to single-node solutions. Our BLASTP implementation could replace a small computing cluster; alternatively, it could be used as a building block for a larger system that delivers performance equivalent to today's thousand-node clusters with many fewer nodes.

3. The BLAST algorithm

BLAST, the ***B****asic* ***L****ocal* ***A****lignment* ***S****earch* **T**ool [1], is the most popular package for biological sequence analysis. BLAST uses an efficient heuristic approach to identify strong alignments between a query sequence and a database without the full computational cost of Smith-Waterman. BLAST's computation is divided into three stages (Figure 3): seed generation, ungapped extension, and gapped extension. The seed generation stage identifies *seed matches*, or short patterns of highly similar residues, between the query and database sequences. Seed matches are passed to ungapped extension, where the region around each match is inspected. This stage separates those matches that occur by chance from those that are parts of longer pairs of similar substrings that align without gaps, called *high-scoring segment pairs (HSPs)*. HSPs whose total alignment score exceeds a user-determined threshold are passed to *gapped extension*, which further extends them using a Smith-Waterman-like algorithm allowing for gaps in either sequence. A pair of sequences that successfully passes all three stages is finally reported to the user. BLAST's search strategy rapidly discards most pairs of proteins that contain no meaningful alignment, resulting in a large speedup compared to the traditional approach of running the full Smith-Waterman algorithm on each pair.

This work focuses on accelerating BLASTP's seed generation stage. Our accelerators for the other two stages of BLASTP are described in [4, 10].

3.1. Seed generation in detail

Seed generation accepts a query sequence Q and a database D and emits a list of paired positions (q, d) in the two sequences at which seed matches occur. Seed generation consists of two substages: *word matching* (1a) and *two-hit* (1b).

The word matching substage finds pairs of highly similar w*-mers*, or substrings of length exactly w, between Q and D. More precisely, word matching generates all pairs (q, d) such that $\sum_{i=0}^{w-1} \delta(Q[q+i], D[d+i]) \geq T$, where δ is a biologically motivated table of scores for all residue pairs, and T is a user-defined score threshold. These pairs are called *word matches*.

To find word matches efficiently, stage 1a uses a precomputed *neighborhood* $N(w, T)$ of the query sequence, which is a list of all w-mers that score at least T when paired with some w-mer of the query. Part of a neighborhood for a w-mer with $w = 3$ is illustrated in Figure 4. If a database w-mer occurs in the query's neighborhood, then it is part of a word match with the query. The neighborhood of a query may be stored in a direct lookup table for efficient occurrence testing.

The *two-hit* substage filters the stream of word matches generated by the word matching substage to produce *seed matches*. Filtering is necessary because, for small w, w-mer matches between two proteins occur at a high rate by chance alone. The two-hit heuristic exploits the observation that HSPs of biological interest are typically much longer than a single word and hence are likely to generate several nearby word matches. Formally, a seed match is a pair of word matches (q, d) and (q', d') such that $d - q = d' - q'$ and $w \leq d - d' < A$. The constant A, again specified by the user, is termed the *window length*. Larger A's detect more real HSPs but also increase the rate at which two chance word matches happen to form a seed match.

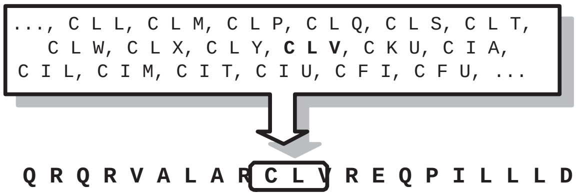

..., C L L, C L M, C L P, C L Q, C L S, C L T,
C L W, C L X, C L Y, C L V, C K U, C I A,
C I L, C I M, C I T, C I U, C F I, C F U, ...
Q R Q R V A L A R C L V R E Q P I L L L D

Figure 4. Neighborhood of query w-mer CLV in a protein sequence.

Table 1. Execution profile and match rates of the BLASTP pipeline.

	Word Match.	**Two-hit**	**Ungap. Ext.**	**Gap. Ext.**
% time	30.96%	19.29%	15.85%	33.60%
Match Rate	3.873×	0.043	0.003	0.031

3.2. Execution Profile of BLASTP

Table 1 shows the performance characteristics of NCBI BLASTP for a typical workload. Seed generation dominates, accounting for up to half of execution time. However, to achieve more than a 2× speedup, the other stages must be accelerated as well. All stages except word matching are extremely good filters, discarding over 95% of their input. In contrast, word matching is a net expander of data, generating 3.9 w-mer matches on average per input residue from the database. The two-hit stage discards most of these w-mers. Hence, although it is the least expensive stage of the pipeline, stage 1b is crucial to reducing the workload of the computationally more expensive downstream stages.

4. Hardware architecture

We now present our hardware architecture for the seed generation stage. We first describe high-throughput word matching and two-hit units, then detail techniques to handle the high data generation rate of the word matching substage. We introduce a novel work division scheme to distribute word matches to replicated copies of the two-hit unit and design a switching network to efficiently route matches from the word matching modules to the two-hit units.

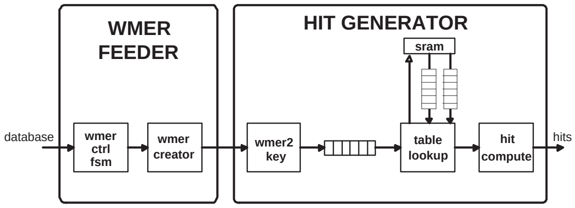


Figure 5. Word matching hardware design.

4.1. Word matching substage

The word matching module (Figure 5) is divided into two logical components: the w-mer feeder and the hit generator. The w-mer feeder converts a database residue stream into a stream of words to be scanned against the query neighborhood. Up to 12 database residues are accepted in each clock cycle by the w-mer control FSM. The w-mer creator block extracts the w-mer starting at each database position, producing up to 12 w-mers per clock cycle.

The hit generator produces word matches from an input w-mer by probing a direct memory lookup table to which every possible w-mer maps. For our word size $w = 4$, this table is too large to be stored in on-chip block RAM and so is kept in off-chip SRAM. Because five bits are needed to represent each residue, using a w-mer directly to address a table would require a prohibitively large 32^w entries, of which only 20^w would be valid. We instead compute the address of a w-mer r as a polynomial expression with exactly the required range: $H(r) = 20^{w-1}r[w-1] + 20^{w-2}r[w-2] + ... + r[0]$. For fixed w, this computation, which is carried out by the *wmer2key* module, is easily realized using small lookup tables and an adder tree.

The table lookup module finds all word matches between each database w-mer and the query. Our current realization of this module uses a 32-bit addressable SRAM storing positions in a 2048-residue query sequence, though our design generalizes to other query lengths and memory word sizes. We divide SRAM into a *primary table*, with 20^w 32-bit entries, and a *duplicate table*. Each entry in the primary table stores at most three query positions to which a w-mer maps. w-mers mapping to more than three positions are instead stored in the duplicate table. To indicate whether a w-mer maps to more than three positions, the primary table entry includes a *duplicate bit*. If this bit is set, the remaining bits of the entry hold a pointer into the duplicate table and a count of matching words, which are stored consecutively. Figure 6 illustrates the data path for a w-mer lookup, including primary table entries with and without duplicate table pointers.

Lookups into the duplicate table increase the w-mer service time and so reduce the throughput of the word matching substage. It is therefore desirable that w-mer lookups be satisfied by a single probe into the primary table whenever possible. We implement a *delta encoding* scheme to efficiently pack n query positions qp plus a duplicate bit into D bits, where D is the width of a primary table entry.

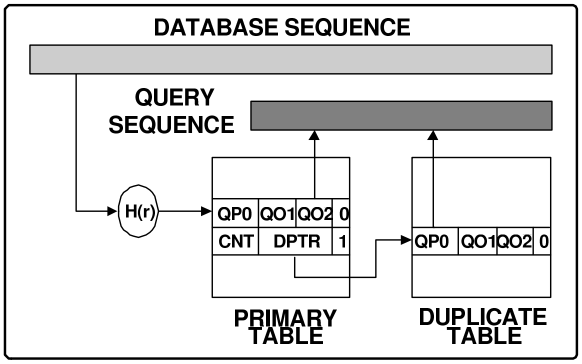


Figure 6. Lookup table data path.

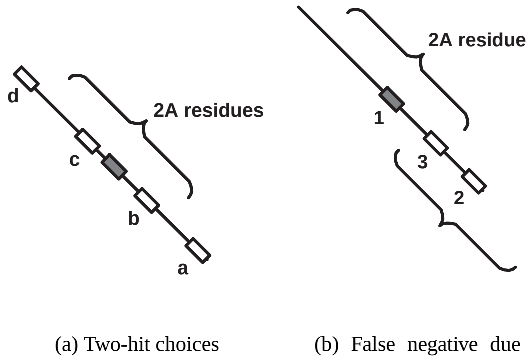
We describe the delta encoding scheme for $D = 32$ and $w = 11$, though it works whenever $n|qp| > D - 1$ but $|qp| + (|qp| - 1)(n - 1) \leq D - 1$. To pack three 11-bit query positions into 31 bits, we store one query position using a full 11 bits, then store the differences between the first and second and the second and third positions, modulo $2^{11} = 2048$, as 10-bit offset values. If an entry contains (qp_0, qo_1, qo_2), the three query positions are decoded in hardware as follows: qp_0, $(qp_0 + qo_1) \bmod 2048$, and $(qp_0 + qo_1 + qo_2) \bmod 2048$. In contrast, a naive approach using 11 bits per position would be able to store just two positions plus a duplicate bit in a 32-bit entry.

To ensure correctness of delta encoding, we must handle two special cases. Firstly, for three or more sorted query positions, 10 bits are sufficient to represent the differences between all but (possibly) one pair (qp_i, qp_j). The solution is to start the encoding by storing qp_j in the first 11 bits. For example, query positions 10, 90, and 2000 are encoded as $(2000, 58, 80)$. Secondly, if there are only two query positions with a difference of exactly 1024, we introduce a dummy position 2047, then proceed as usual. For example, query positions 70 and 1094 (and the dummy 2047) are encoded as $(1094, 953, 71)$. The dummy position is easily ignored, since valid word starts in the query range from 0 to $2047 - w$.

With the use of delta encoding, 82% of w-mer lookups can be satisfied in a single probe, which would otherwise satisfy only 67% of lookups. All query positions in our experiments can be retrieved in six SRAM accesses, versus nine with the naive implementation.

4.2. Two-hit substage

Recall that the two-hit substage is responsible for recognizing pairs of word matches that fall within A database positions of each other. The two matches (q, d) and (q', d')



(a) Two-hit choices

(b) False negative due to out-of-order word match

Figure 7. Examples of the two-hit computation.

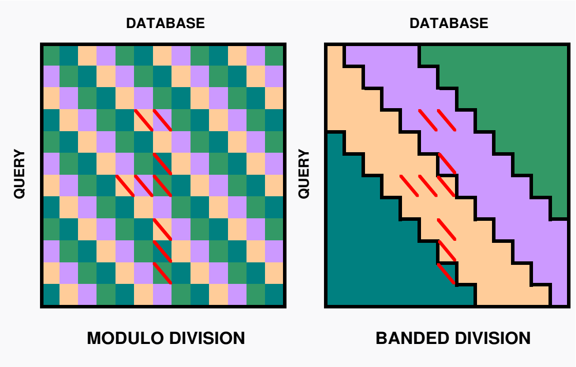


Figure 8. Modulo division of diagonals results in more equal distribution of hits.

must also have $d - q = d' - q'$; this shared difference is called the *diagonal* of the matches.

Two-hit recognition is implemented using an array that stores the database position of the most recently encountered word match on each diagonal. The diagonal array is of total size $2M$, where M is the query length. At any position in the database, word matches to that position can occur only in a window of M diagonals. Hence, as the database is scanned left to right, the array stores only this active window, reusing array cells that correspond to no-longer-possible diagonals. For a query size $M = 2048$ and 32-bit database positions, the diagonal array can be implemented in eight block RAMs.

A straightforward two-hit implementation works only if matches on a given diagonal arrive in the order of their database positions. However, as detailed in Section 4.4, word matches may not always arrive at the two-hit substage in this order. We implement the following heuristic to deal with out-of-order word matches: if a match falls at most A positions prior to the most recently seen match, discard it; else, declare it a seed match by itself and forward it to ungapped extension. This heuristic discards out-of-order word matches that are likely part of the same HSP as a word match in the array, while passing on those matches that are likely part of a distinct HSP.

Figure 7(a) illustrates the choices that the two-hit computation must make. The most recently recorded word match on the diagonal is shown in gray. Matches c and b are within A positions before and after this match, respectively. According to our heuristic, match c would be discarded, while matches b and d would cause a seed match to be generated. Match a would not by itself cause a seed match but would overwrite the position recorded in the array. As shown in Figure 7(b), our treatment of out-of-order word matches is not perfect but may miss cases that should generate seed matches. Suppose word matches arrive as shown in the order 1, 2, 3. Match 2 overwrites match 1 but does not cause a seed match, while match 3 is discarded. Hence, no seed match is generated. However, as shown in Section 6, seed matches lost by our heuristic have a negligible effect on sensitivity.

4.3. Two-hit replication to handle high word match rate

With our implementation's word size $w = 4$, the word matching stage generates roughly two word matches per database residue. Although our implementation uses dual-ported block RAMs, the two-hit logic, which is pipelined for speed, uses one read and one write port in each clock cycle. Hence, a two-hit unit can consume only a single word match per clock. Processing more word matches at once requires replication of the two-hit logic.

Replication of the entire diagonal array in each copy of the two-hit unit would require that all copies be kept coherent, leading to a multi-cycle update phase and a corresponding loss in throughput. Instead, we partition the diagonals, which can be processed independently of each other, across b two-hit units as follows. A word match (q, d) is processed by the j^{th} two-hit unit if $d - q \equiv j - 1 \pmod{b}$. Note that we use the low-order rather than the high-order bits of the diagonal to select the two-hit unit. In practice, word matches tend to occur in clusters within a band of nearby diagonals; hence, as Figure 8 illustrates, our modulo scheme for distributing matches tends to partition work among the two-hit units more evenly than allocating a contiguous band of diagonals to each unit.

4.4. Increasing throughput with lookup table replication

With the downstream stages capable of handling word matches at greater than one match per clock, throughput

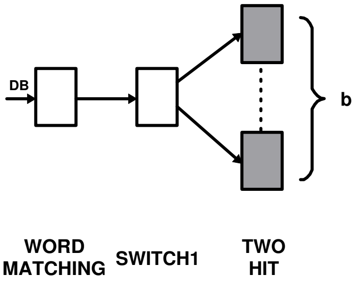


Figure 9. Switch 1: routing 3 hits from a single word matcher to b two-hit units.

of seed generation becomes limited by the word matching substage. The rate at which this substage generates word matches is constrained by the bandwidth of the off-chip SRAM. To speed up the pipeline, we use multiple word matching modules in parallel, each accessing an independent off-chip SRAM resource. Adjacent database w-mers are distributed by the feeder stage to each of h lookup tables as they are requested.

The use of h independent lookups has an unintended consequence on the generated stream of word matches. Since the time to look up a w-mer varies with the number of matches it generates, the word matchers lose synchronization and generate word matches that are out of order with respect to their database positions. While a limited degree of “out-of-orderness” can be handled by our two-hit logic, it is desirable to upper-bound how unordered word matches can be to guarantee that sensitivity will not suffer arbitrarily. In our design, two successively generated word matches may be out of order by at most $(\ell + \lceil L/3 \rceil) \times (h - 1)$ residues, where $L = 15$ is the maximum number of query positions per w-mer stored in our lookup table, and $\ell = 4$ cycles is the latency of access to the duplicate table.

4.5. Routing between multiple parallel substages

An important component of our architecture is the routing of word matches from h hit generators to b two-hit modules. We use a two-phase switching network for this purpose. Switch 1 routes word matches from a single lookup table to one of b queues, while switch 2 routes matches from h queues to a single two-hit unit.

Switch 1 (Figure 9) independently routes up to three word matches from a lookup module in a single clock cycle. Routing is simplified due to the modulo division of diagonals in the two-hit substage: if b is a power of two, i.e. 2^t, the lower t bits of a word match’s diagonal identify its target two-hit unit. In case of a collision, priority is given to the word match with the lowest database position.

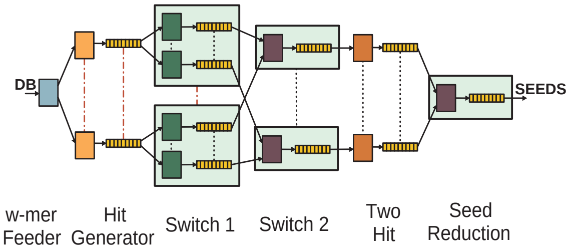


Figure 10. Seed generation logic, showing routing.

With the addition of multiple lookup tables, additional switching circuitry is required to route all word matches to their corresponding two-hit modules. First, we replicate switch 1 for each word matcher, routing its word matches to one of b queues. Second, switch 2, which is replicated for each two-hit unit, selects one word match per cycle from among the appropriate queues for all modules. This design can route any of the $3h$ word matches generated by the lookup tables to any of the b two-hit modules.

Figure 10 illustrates the complete architecture of the seed generation hardware. In addition to switches 1 and 2, the design includes a seed reduction tree, which collects seed matches from all b two-hit units into a single stream to be forwarded to the ungapped extension stage.

5. Hardware parameter space exploration

We now investigate the parameters of the seed generation stage and characterize their effect on the BLASTP hardware pipeline. The parameter values used by software may not yield an efficient hardware implementation; hence, it is crucial to study the performance impact of different parameter choices. Furthermore, an accelerated implementation must produce results similar to software, so the chosen parameters must not compromise sensitivity.

5.1. Maximizing sensitivity

We ran NCBI BLASTP (default word length $w = 3$ and threshold $T = 11$) to compare the *E. coli* K12 proteome (4,242 sequences) to a 2004 release of the GenBank Non-Redundant database (2.3 million sequences). The resulting alignments became our *gold standard*, against which we measured the quality of results produced in our experiments with other parameter values. An alignment in the standard was considered “found” in an experiment (i.e. a true positive TP) if some experimentally produced alignment overlapped at least half its residues in both query and database; otherwise, it was counted as a false negative (FN). We calculated sensitivity as $TP/(TP + FN)$. Alignments

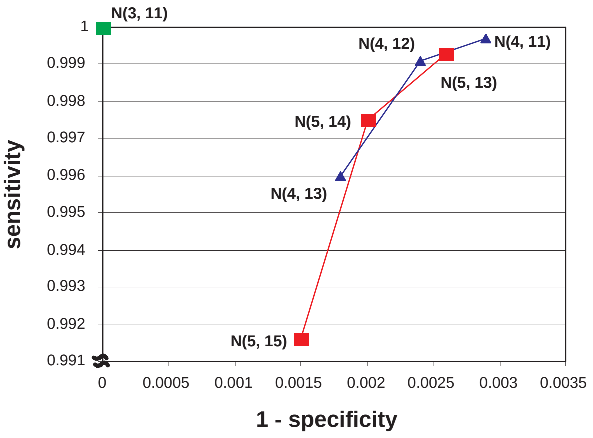


Figure 11. Result quality of BLASTP algorithm for various neighborhoods.

found in an experiment but not in the standard were for our purposes considered false positives (FP). We measured specificity as $TP/(TP + FP)$. We note that sensitivity is the more important statistic, as high-scoring alignments not found by NCBI BLASTP might still be biologically meaningful.

Figure 11 shows receiver operating characteristic (ROC) curves for our experiments. The X-axis represents (1 - specificity); the Y-axis, sensitivity. Longer word lengths w increase the probability of missing a word match in a biologically meaningful alignment. For example, searching with a neighborhood $N(5, 13)$ is less sensitive than searching with $N(4, 13)$. The sensitivity of a search at a fixed w is higher for a lower threshold value T. We consider parameters with sensitivity greater than 99.5% as candidates for use in Mercury BLASTP.

5.2. Resource constraints

The most stringent hardware resource constraint in stage 1 is the capacity of off-chip SRAM required to store the lookup table accessed by the word matching stage. The capacity of the SRAM limits the maximum table size in stage 1a and so impacts the word length, threshold, and maximum query sequence length that the implementation can support.

The effect of these three parameters on the table size (primary plus duplicate) is illustrated in Table 2. For a fixed query sequence length, table size increases exponentially with word size. For example, neighborhoods with $w = 4$ require under 2 MB, while those with $w = 5$ require at least 16 MB. The *occupancy rate* measures the fraction of all w-mers (out of 20^w) present in the neighborhood of a typical protein query of given size. The high expected occupancy rates justify our use of a direct lookup table rather than a sparse hashing scheme.

Table 2. Effects of parameters on lookup table size.

	2048		4096	
N(w, T)	**Occ. rate**	**Table size**	**Occ. rate**	**Table size**
$N(3, 11)$	95%	77KB	99%	134KB
$N(4, 13)$	85%	928KB	96%	1.6MB
$N(4, 14)$	70%	743KB	88%	1.1MB
$N(5, 14)$	80%	16MB	95%	25.7MB

Biological protein sequences are typically about 300 residues long. However, our hardware can support significantly longer queries. It is therefore advantageous to concatenate smaller queries into a composite composite sequence that can be compared to the database in one pass. Reducing the number of passes over the database reduces the overall search time. On the other hand, a longer query increases the number of w-mers in the query's neighborhood, which can increase the size of the duplicate table. For example, the neighborhood $N(4, 13)$ of a 4096-residue query requires 1.7× more table space than a 2048-residue query with the same neighborhood.

For economic reasons, our implementation uses 1 MB of SRAM per lookup table. This constraint precludes use of $w > 4$ if sensitivity is to be maintained. Throughput considerations lead us to maximize word length, which minimizes the rate of word matches; hence, we select $w = 4$ and $T = 13$, the lowest threshold that fits our SRAM. With these parameters, query length is restricted to 2048 residues.

5.3. Maximizing throughput

We now develop a mean-value performance model to estimate the throughput of the BLASTP hardware pipeline at various points in the reduced parameter space. When executing in a pipelined fashion, overall throughput is limited to the minimum throughput achieved by any one resource:

$$Tput_{pipe} = \min(Tput_{1a}, Tput_{1b}, Tput_2, Tput_3),$$

where $Tput_{1a}, Tput_{1b}, Tput_2$, and $Tput_3$ are the throughputs (in Mresidues/second) of the word matching, two-hit, ungapped extension, and gapped extension stages. Throughput of each stage is expressed as an equivalent number of database residues processed per second. Throughput expressions for ungapped and gapped extension are available in [4, 10]; we omit them here for brevity.

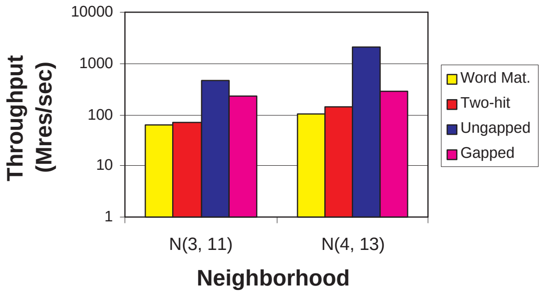


Figure 12. Effect of word length on pipeline throughput.

The throughput of seed generation is computed as

$$Tput_{1a} = \frac{h \times f}{c_{1a}} \qquad Tput_{1b} = \frac{b \times f}{c_{1b} \times r_{1a}}$$

In the following discussion, we set $h = 1$ and vary b so that the two-hit module is not the bottleneck in the pipeline. Based on our synthesis results, we use a clock frequency $f = 130$ MHz for the seed generation stage. The average input processing times c in stages $1a$ and $1b$ are 1.3684 clocks/database residue and 1 clock/w-mer, respectively. The number of w-mers into the two-hit unit is determined by the word match generation rate r_{1a} (expressed as w-mers/database residue) of the word matching stage. All these parameter values were determined empirically using simulations on large biological datasets.

The throughput of seed generation is inversely proportional to c_{1a} and r_{1a}, which in turn depend on the neighborhood parameters. Using a neighborhood $N(4, 13)$ instead of $N(3, 11)$ reduces c_{1a} from 2.18 to 1.37 clocks/w-mer, increasing throughput from 60 to 95 Mresidues/second. The word match generation rate also drops by approximately half, reducing the load on the two-hit stage. The drop in load increases the throughput of all downstream stages, as shown in Figure 12.

To summarize, our final hardware implementation uses a neighborhood of $N(4, 13)$, which increases throughput versus the NCBI BLASTP default parameters while maintaining high sensitivity. Concatenated queries of up to 2048 residues are supported in each pass over the database.

6. Results

We have implemented Mercury BLASTP on the Mercury system [2], which provides disk-based, high-throughput computation on reconfigurable logic. The system's host machine contains two 2.0 GHz AMD Opteron CPUs with 8 GB of memory, running Linux, and two prototyping co-processor boards connected via the PCI-X bus to the host. Interfacing drivers to the boards are provided by Exegy, Inc[1]. The first board contains a Xilinx Virtex-II 6000 FPGA (used for BLASTP stages 1 & 2); the second, a Xilinx Virtex-II 4000 FPGA (used for stage 3). Up to three synchronous 1 MB SRAM modules are available on the first board. In this configuration, we have demonstrated sustained data throughput from disk to FPGA well over the requirement for Mercury BLASTP.

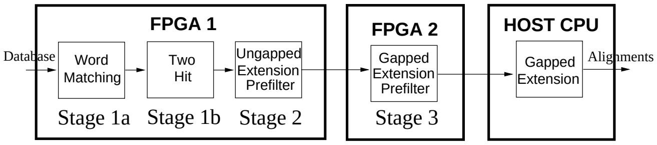


Figure 13. Mercury BLASTP hardware/software partition.

The seed generation hardware has been built post-place-and-route with one, two, and three word match generator modules. The three configurations use one, four, and eight two-hit units, respectively, with the hardware synthesizing to a clock frequency of 130 MHz, 110 MHz, and 100 MHz. The expected throughput (using the mean value model in Section 5.3) of the seed generation stage is 95, 160 and 219 Mresidues/second, respectively, for 2048-residue queries. In each case, word matching is the limiting stage of the Mercury BLASTP pipeline.

We now examine stage 1 as part of the entire Mercury BLASTP deployment shown in Figure 13. Seed generation (with two word match generators and four two-hit modules) and ungapped extension run at 110 MHz and 85 MHz, respectively, on the XC2V6000, consuming 63% of the slices and 77% of the block RAMs. Gapped extension runs at 60 MHz on the XC2V4000, consuming 48% of the slices and 58% of the block RAMs. Our implementation uses two FPGAs due to the large number of block RAMs required by all stages combined; however, the latest Virtex FPGAs from Xilinx can fit the entire design on a single chip.

We have integrated Mercury BLASTP with the original NCBI BLAST code to preserve the BLASTP user interface, including command-line options and input/output format. Query sequences are packed into 2048-residue bins using the first-fit-decreasing bin packing algorithm; this packing is done transparently to NCBI BLAST. Neighborhood table generation is done online as part of query setup for the hardware [8]. The database is streamed in a single pass per query through the three hardware filtering stages. Seed matches that extend into high-scoring gapped alignments are passed to the BLASTP software, which does final gapped extension

[1] http://www.exegy.com/

and prepares the alignments for reporting to the user.

Performance: We compared Mercury BLASTP to NCBI BLASTP (release 2.2.10), the BLAST software used by the majority of the bioinformatics community. All BLASTP parameters were set to their default values, except for the alignment cut-off score (E-value), which we set to 10^{-5} (a reasonable value for a large database search). NCBI BLASTP was run on a single 2.8 GHz Pentium 4 workstation with 1 GB of memory. We measured the runtime of a search of the entire *E. coli* K12 proteome (4,242 sequences; 1.35 Mresidues) against a 2006 release of the GenBank Non-Redundant database (4 million sequences; 1.39 Gresidues). We included setup and search time but excluded time to produce the final alignments for printing. The total execution time of this search on the baseline workstation was 101 hours and 51 minutes. In comparison, Mercury BLASTP completed the same search in 2 hours and 44 minutes, giving a speedup of $37\times$.

Sensitivity: Output alignments returned by the software were compared against those returned by Mercury BLASTP, using the same criterion described above for determining identity of alignments in the two outputs. Mercury BLASTP achieved sensitivity of 99.4% versus NCBI BLASTP. The software returned a total of 5.92×10^6 alignments of which Mercury BLASTP detected 5.88×10^6. Furthermore, Mercury BLASTP found an additional 11,218 significant gapped alignments not found by NCBI BLASTP. We conclude that on this benchmark, Mercury BLASTP returns output of comparable quality to NCBI BLASTP.

7. Conclusion

In this work, we have presented an FPGA hardware design to accelerate the seed generation stage of BLASTP. Our design closely matches the behavior of the standard BLASTP software while overcoming issues caused by high data generation and filtering rates. This stage has been implemented as part of the Mercury BLASTP system, which achieves software-like sensitivity at more than an order of magnitude speedup over a NCBI BLASTP on a modern workstation. To the best of our knowledge, Mercury BLASTP is the first FPGA accelerator for the entire BLASTP pipeline.

The FPGA devices used in our work is from an older generation. The latest PCIx-based accelerator cards in use by our partners at Exegy, Inc. contains two FPGAs from the Xilinx Virtex-4 family. Each of the two FPGAs has two larger attached SRAMs than in our current design. With the larger logic and block RAM resources, we will be able to fit the entire Mercury BLASTP pipeline on a single FPGA clocked at higher frequencies. The larger capacity SRAMs will be able to accomodate query neighborhoods of 4096 residues, hence halving the number of passes of the database. We will also be able to run two parallel copies of the Mercury BLASTP engine on the two FPGAs. Overall, we expect a speedup of $4\times$ over our current implementation simply by moving to the latest generation FPGA technology.

The seed generation hardware architecture described here can be used to accelerate the first stage of a number of other biosequence analysis tools that use neighborhood-based seeded alignment. Examples include other BLAST variants, such as BLASTX, and tools that use seeds to accelerate searches using multiple sequence alignments, such as PSI-BLAST [1], PhyloNet [21], and the HMMERhead [15] software for protein motif finding. Applying our design to these systems will further demonstrate the utility of FPGAs for accelerating even complex heuristic algorithms for biosequence analysis.

References

[1] S. F. Altschul et al. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucl. Acids Res.*, 25(17):3389–3402, Sep 1997.

[2] R. D. Chamberlain et al. The Mercury System: Exploiting truly fast hardware for data search. In *Proc. 3rd Int'l Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, pages 65–72, September 2003.

[3] S. Eddy. Private communication, 2006.

[4] B. B. Harris. Acceleration of gapped alignment in BLASTP using the Mercury system. Technical Report WUCSE-2006-47, Dept. of Computer Science and Engineering, Washington University in St. Louis, August 2006.

[5] M. Herbordt et al. Single pass, BLAST-like approximate string matching on FPGAs. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 217–26, 2006.

[6] J. D. Hirschberg et al. Kestrel: a programmable array for sequence analysis. In *IEEE Int'l Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 25–34, 1996.

[7] D. T. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, 1993.

[8] A. C. Jacob. Design and analysis of an accelerated seed generation stage for BLASTP on the Mercury System. Master's thesis, Washington University in St. Louis, May 2006.

[9] P. Krishnamurthy et al. Biosequence similarity search on the *Mercury* system. *Journal of VLSI Signal Processing*, 2007. In press.

[10] J. Lancaster et al. Acceleration of ungapped extension in Mercury BLAST. *Int'l J. of Embedded Sys.*, 2007. In press.

[11] D. Lavenier, S. Guyetant, S. Derrien, and S. Rubini. A reconfigurable parallel disk system for filtering genomic banks. In *Engineering of Reconfigurable Systems and Algorithms*, pages 154–166, 2003.

[12] H. Lin et al. Efficient data access for parallel BLAST. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, page 72.2, 2005.

[13] S. McGinnis and T. L. Madden. BLAST: at the core of a powerful and diverse set of sequence analysis tools. *Nuc. Acids Res.*, 32:20–5, 2004.

[14] K. Muriki et al. RC-BLAST: Towards a portable, cost-effective open source hardware implementation. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, page 196.2, 2005.

[15] E. Portugaly and M. Ninio. HMMERHEAD - accelerating HMM searches on large databases. In *Proc. Int'l Conf. on Research in Molecular Biology (RECOMB)*, pages 250–251, 2004.

[16] H. Rangwala et al. Massively parallel BLAST for the Blue Gene/L. In *High Availability and Performance Computing Workshop*, 2005.

[17] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[18] E. Sotiriades, A. Dollas, and C. Kozanitis. Some initial results on hardware BLAST acceleration with a reconfigurable architecture. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2006.

[19] Swiss Institute of Bioinformatics. Growth of Swiss-Prot. `http://www.expasy.org/sprot/relnotes/#SPstat`, 2006.

[20] Timelogic, Inc. Timelogic DeCypher BLAST. `http://www.timelogic.com/`.

[21] T. Wang and G. D. Stormo. Identifying the conserved network of cis-regulatory sites of a eukaryotic genome. *Proc. Natl. Acad. Sci. USA*, 102:17400–5, 2005.

[22] Y. Yamaguchi et al. High speed homology search with FPGAs. In *Pacific Symp. on Biocomputing*, pages 271–282, 2002.