Published in final edited form as:
ACM Trans Reconfigurable Technol Syst. 2008 June ; 1(2): 9. doi:10.1145/1371579.1371581.

Mercury BLASTP: Accelerating Protein Sequence Alignment

Arpith Jacob,
Dept. of Computer Science and Engineering, Washington University in St. Louis

Joseph Lancaster,
Dept. of Computer Science and Engineering, Washington University in St. Louis

Jeremy Buhler,
Dept. of Computer Science and Engineering, Washington University in St. Louis

Brandon Harris, and
Dept. of Computer Science and Engineering, Washington University in St. Louis, and BECS Technology, Inc., St. Louis, Missouri

Roger D. Chamberlain
Dept. of Computer Science and Engineering, Washington University in St. Louis, and BECS Technology, Inc., St. Louis, Missouri

Abstract

Large-scale protein sequence comparison is an important but compute-intensive task in molecular biology. 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 running time 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 describe the architecture of the portions of the application that are accelerated in the FPGA, and we also describe the integration of these FPGA-accelerated portions with the existing BLASTP software. We have implemented Mercury BLASTP on a commodity workstation with two Xilinx Virtex-II 6000 FPGAs. We show that the new design runs 11-15 times faster than software BLASTP on a modern CPU while delivering close to 99% identical results.

General Terms

Algorithms; Design; Performance

Additional Key Words and Phrases

bioinformatics; biological sequence alignment

Categories and Subject Descriptors: C.1.3 **[Processor Architectures]:** Other Architecture Styles—*Heterogeneous (hybrid) systems; Pipeline processors*; J.3 **[Life and Medical Sciences]:** Biology and genetics

1. Introduction

Comparative sequence analysis is widely used in computational biology to study the evolutionary relationship between protein sequences. Biologists compare a protein of unknown function, termed the *query*, against a database of sequences with known function to detect sequences with high similarity. Similarity between query and database sequences is described 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.

The classical method for computing the best alignment of two proteins is the Smith-Waterman algorithm [Smith and Waterman 1981], which uses dynamic programming. However, the cost of this algorithm, which scales as the product of the sequence lengths, rapidly becomes prohibitive for comparing a query protein to an entire database. To make large-scale proteomic comparison feasible, the biological community has resorted to heuristic methods. In particular, one heuristic-based comparison tool, BLASTP [Altschul et al. 1997], dominates this community. BLASTP uses a *seeded alignment* heuristic to limit Smith-Waterman comparison to pairs of proteins that are *a priori* likely to be highly similar. This heuristic avoids almost all the work that Smith-Waterman would normally perform while still delivering results of sufficient quality to satisfy biologists. Because of its wide adoption, BLASTP's output has become a *de facto* standard against which alignments found by other proteomic comparison tools are compared.

While BLASTP delivers an immense improvement in efficiency – roughly two orders of magnitude – over Smith-Waterman, it too is subject to ever larger computational demands, both from larger biosequence databases and from increasing genomic sequencing capacity. Over the last two decades, public biosequence databases have grown at an exponential rate, driven in part by widespread whole-genome sequencing and gene prediction in many organisms. For example, Figure 2 shows the rapid growth of the TrEMBL protein database [Swiss Institute of Bioinformatics 2006] over the last decade.

At the same time, more and faster genome sequencing has resulted in a high rate of new protein queries for BLASTP. A single modern genome sequencing machine [Margulies et al. 2005] can sequence a complete bacterial genome in a day, producing 4000-6000 new putative protein sequences. Simply comparing all these sequences to the existing public databases with BLASTP would take more than one day of CPU time today. More ambitious sequencing projects in the near future, focused on eukaryotic genomes, could potentially produce tens of thousands of candidate proteins in a day. A second source of high computational load for the BLASTP application is the aggregation of queries from many different sources to a central BLAST server. Consider, for example, the BLAST web server maintained by the U.S. National Center for Biological Information (NCBI). In 2004, this server was backed by a Linux cluster of ~200 CPUs that processed 1.4×10^5 queries on a typical weekday – more than one query per second. Furthermore, NCBI planned to double their computing capabilities to keep up with demand [McGinnis and Madden 2004].

In short, increases in database size and rate of query generation, as well as the popularity of the BLASTP application itself, have raised the computational cost of sequence comparison to the point that running BLASTP can be a bottleneck to proteomic analysis. Architectural solutions to this bottleneck must exploit parallelism in the BLASTP computation to raise the application's throughput.

In this work, we address the BLASTP bottleneck by implementing an engine for accelerated protein database search, using a combination of FPGAs and general-purpose CPUs. Our implementation, *Mercury BLASTP*, departs substantially from prior accelerators targeting the slower but more accurate Smith-Waterman computation. The full BLASTP algorithm is actually a pipeline of several different computational stages: discovery of short, highly similar *seed matches* between query and database, followed by two successive phases of dynamic programming extension to determine which matches represent meaningful biological similarity between sequences. The three stages contribute roughly equally to BLASTP's total running time in software, so we target all of them for speedup via implementation on our FPGA platform. Key design choices of our implementation include a streaming architecture, which allows all stages to run in parallel; fast generation of seed matches using multiple parallel SRAM lookups; and small, fixed-size dynamic programming extension steps that effectively filter out most uninteresting seed matches in hardware before they reach the software. These choices lead to a final design that achieves significant speedup relative to the BLASTP software, not just relative to the much slower Smith-Waterman algorithm.

To maximize acceptance of our accelerator in the biological community, we strive to produce an implementation at least as sensitive to biologically meaningful alignments of its inputs as NCBI BLASTP and to match the latter's command-line interface and output format. An important limitation of previously published accelerators for proteomic comparison is that few have included measurements of their *sensitivity* relative to the BLASTP software. Without such measurements, there is little reason for BLASTP users to trust the results produced by these systems. We therefore demonstrate that our BLASTP implementation can compete on *both* speed and sensitivity.

We have implemented Mercury BLASTP on a commodity workstation with a pair of Xilinx Virtex-II 6000 FPGAs. Mercury BLASTP can scan a protein database with throughput 11-15 times faster than the BLASTP software on a modern server while delivering results approximately 99% identical to those returned by the software. Our design can also be used to accelerate similar computations in other bioinformatics applications, such as HMMERhead [Portugaly and Ninio 2004] and PhyloNet [Wang and Stormo 2005].

The remainder of this work is organized as follows. After reviewing related work, we briefly describe the core computation of the BLASTP software to motivate our design. We then describe in detail the hardware architectures of the various stages of Mercury BLASTP, as well as the rationale for choosing numerical parameters used by the implementation. The following section describes the NCBI BLASTP-based software that communicates with the Mercury BLASTP FPGA designs. Finally, we report performance of Mercury BLASTP on real-world proteomic sequence comparisons.

2. Related Work

The Smith-Waterman dynamic programming algorithm for sequence alignment [Smith and Waterman 1981] has been extensively targeted for parallelization in hardware. FPGA accelerators for this computation [Hirschberg et al. 1996; Hoang 1993; Yamaguchi et al. 2002] report 1-2 orders of magnitude speedup over software. Unfortunately, hardware Smith-Waterman, even running 100× faster than a software implementation on current CPUs, runs 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 family of biosequence comparison algorithms. BLASTN, the algorithm for comparing DNA sequences, has been the target of several implementations. While BLASTN and BLASTP have significant differences, they share a similar seeded alignment hardware architecture.

Our prior work includes an end-to-end BLASTN accelerator, Mercury BLASTN [Buhler et al. 2007; Krishnamurthy et al. 2007; Krishnamurthy et al. 2004; Lancaster et al. 2005; 2008], on which our BLASTP accelerator is based. Here, we accelerate all three stages on the FPGA unlike in BLASTN, where the third stage requires an insignificant fraction of total execution time. Neighborhood matching in BLASTP generates a large number of matches, translating into far more work in the first stage. Mercury BLASTP therefore uses direct memory lookup tables and parallel two-hit units to cope with this increased workload. In contrast, BLASTN produces far fewer matches in the first stage, and so we use compact on-chip Bloom Filters to build a high-throughput design. Mercury BLASTN is I/O limited unlike Mercury BLASTP.

SGI along with Mitrionics have recently released an FPGA accelerated version of BLASTN for the SGI RASC appliance. The accelerator is implemented on the Mitrion Virtual Processor, a soft-core parallel processor implemented on an FPGA. The Mitrion-C implementation is available as an open-source download[1]. They report that a dual-blade SGI RASC implementation is 16× faster than a single Itanium 2 processor core and 10× faster than a quad-core AMD Opteron processor. The SGI/Mitrionics work is a direct implementation (in a high level language) of our design for Mercury BLASTN, first published in 2004 [Krishnamurthy et al. 2004], on a newer-generation FPGA. Their implementation further validates our architecture and design choices. SGI/Mitrionics have not released a BLASTP accelerator to date.

RDisk [Lavenier et al. 2003] is another FPGA-based approach to accelerating stage 1 of BLASTN, claiming 60 Mbases/sec throughput using a single disk. However, their work does not accelerate the entire BLAST application. Other BLASTN accelerators [Muriki et al. 2005; Sotiriades et al. 2006] display varying degrees of success. However, none of them show end-to-end numbers.

We next compare Mercury BLASTP with TreeBLASTP [Herbordt et al. 2006; Herbordt et al. 2007], a recent FPGA-based accelerator for BLASTP-like computations. Rather than following the original NCBI BLAST heuristic, TreeBLASTP uses a one-dimensional systolic array to directly perform ungapped extension thereby eliminating seed generation. The array requires as many processing elements as the size of the query sequence. While the asymptotic time complexity of the two designs remains the same (linear in the size of the database), the systolic array requires more logic and block RAM resources on the FPGA. The authors claim to achieve higher sensitivity than NCBI BLASTP but have validated their hardware only on small datasets (fifteen query sequences).

In contrast, our design implements the BLASTP heuristic using off-chip SRAM as the primary resource thus freeing up on-chip logic and memories to implement all stages on a single newer-generation FPGA. In terms of scalability, supporting larger query sequences requires larger capacity SRAMs in our design, while TreeBLASTP requires larger FPGAs. While the average protein sequence is small (about 300 residues), supporting larger queries enables us to perform a query packing optimization. In our design, a 1 MB SRAM supports query sequences of size 2048 on an older-generation Xilinx Virtex-II 6000 FPGA. TreeBLASTP supports a query size of 1024 on the newer-generation Xilinx Virtex-4 LX 160 FPGA and is on-chip resource limited.

The three stages of Mercury BLASTP (seed generation, ungapped extension and gapped extension) are designed to operate in a *single* pass where each stage is connected in a pipeline, with the database being streamed through. While we currently use two older-generation FPGAs for this purpose, all three stages can be easily placed on a single newer-generation FPGA. TreeBLASTP requires another FPGA to implement the Smith-Waterman gapped extension

[1] http://mitc-openbio.sourceforge.net/

stage in parallel. The authors of TreeBLASTP do not report end-to-end numbers but estimate a performance of 178 Mresidues/second for 1024-residue queries in the ungapped extension stage.

We note that none of the above accelerators other than Mercury BLASTP report their sensitivity compared to software BLAST. In our experience with the biological community, an accelerated implementation must not only yield a significant speedup but more importantly produce results similar to or better than the standard software implementations. Biologists using the BLAST software have in the past been reluctant to adopt accelerated solutions for fear of missing alignments that might otherwise have been found by the software[2]. Mercury BLASTP therefore aims to closely replicate the software's results. We have integrated our FPGA accelerator with the NCBI BLAST software and show that Mercury BLASTP yields results almost identical to those of NCBI's BLAST software on our test computations.

DeCypherBLAST [Timelogic, Inc.] is a commercial product to accelerate BLASTP, utilizing FPGA-based processing engines attached to high-end CPUs. Given the closed nature of DeCypherBLAST and our lack of access to a DeCypher machine, we have insufficient information to compare its performance to that of Mercury BLASTP.

The majority of high-performance BLAST solutions today are based on multiprocessor clusters [Lin et al. 2005; Rangwala et al. 2005]. 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* [Altschul et al. 1997], is the most popular software 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. This section describes the architecture of the existing BLASTP software, which served as the starting point for our accelerated implementation. While multiple software implementations of BLASTP exist, our discussion reflects NCBI's implementation, which is the most popular, as of version 2.2.10.

As described above, BLASTP's goal is to produce alignments between a query and a sequence database, both of which consist of strings over a protein alphabet Σ of twenty possible residues. Alignments reported between the query and a database sequence may align only a substring of each; in bioinformatics parlance, they are *local* rather than global. The quality of an alignment is judged by its score, which is computed as follows. Each pair of aligned residues (x, y) is assigned a score $\delta(x, y)$, where δ is a $|\Sigma| \times |\Sigma|$ matrix of (mostly negative) small integer scores that assigns higher scores to pairs of identical residues and to pairs of residues that are biologically similar. δ is defined by the biological end users of BLASTP, either from empirical observation or from an evolutionary model of mutation [Dayhoff et al. 1978; S. and Henikoff 1992]. Each run of k consecutive unaligned residues in an alignment is assigned a *gap penalty* $-g_o - k \cdot g_e$, where g_o and g_e are constants. The total score of an alignment is then the sum of scores for all its aligned residue pairs, plus the sum of penalties for all its gaps. This score is compared to a threshold to determine whether the alignment is worth reporting to the user.

[2]S. Eddy, personal communication

BLAST's search for high-scoring alignments is divided into three stages, as shown in 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 their alignment using a Smith-Waterman-like algorithm allowing for gaps in either sequence. A pair of aligned proteins that successfully passes all three stages is finally reported to the user along with its alignment. 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.

3.1 Seed generation

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* (stage 1a) and *two-hit* (stage 1b), as illustrated in Figure 4.

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 $\sum_{i=0}^{w-1} \delta(Q[q+i], D[d+i]) \geq T$, for some numerical threshold T. These pairs are called *word matches*.

To find word matches efficiently for any fixed δ, stage 1a uses a precomputed *neighborhood* $N(w, T)$ of the query sequence, which is a list of all w-mers which, when paired with some w-mer of the query, exhibit a total score of at least T under the residue-pair scoring matrix δ. Part of a neighborhood for a w-mer with $w = 3$ and typical residue pair scoring is illustrated in Figure 5. If a database w-mer occurs in the query's neighborhood, then it is part of a word match with the query. All w-mers in the neighborhood of a query are stored in a lookup table, against which each successive w-mer from the database is checked to discover matches.

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 groups of several nearby word matches. Formally, a seed match is a pair of word matches starting at positions (q, d) and (q', d'), respectively, such that $d - q = d' - q'$ and $w \leq d - d' < A$, for a specified constant A. Larger A's detect more real HSPs but also increase the rate at which two word matches arising independently by chance happen to form a seed match. Seed matches are detected efficiently by tracking, for each possible difference $d - q$, the most recent word match (by database position) seen with that difference.

3.2 Ungapped extension

Recall that an HSP is a pair of intervals of common length in two sequences that are aligned without residue insertions or deletions. BLASTP's ungapped extension computes, for each seed match, the highest-scoring HSP (under the matrix δ) that contains that seed match. Extension proceeds in two steps. The pair of aligned w-mers constituting the match are first extended backwards toward the beginnings of the two sequences, then extended forward toward their ends. An HSP's score is the sum of the scores of its constituent residue pairs, so extending an HSP by one residue pair adds that pair's score under δ to the overall score of the HSP. The end of the HSP in each direction is chosen so as to maximize the total score of that direction's

extension. If the total score of an HSP exceeds a user-defined threshold, it is passed on to gapped extension.

For long protein sequences, considering possible extensions an HSP all the way to the ends of the two sequences is computationally expensive. In the vast majority of cases, seed matches appear by chance without an underlying biologically meaningful similarity, so this work would typically be wasted without finding any sufficiently high-scoring HSP. BLASTP therefore implements early termination of extension, using an *X-drop* mechanism. The algorithm tracks the highest score achieved by any extension of the seed match considered thus far. If the current direction's extension scores at least X below this maximum, further extension in that direction is terminated. A more detailed explanation of this heuristic is given in [Lancaster et al. 2005; 2008].

3.3 Gapped extension

The gapped extension stage of BLASTP uses a modified version of the Smith-Waterman algorithm to compute the highest-scoring local alignment between two sequences. We briefly review this algorithm here to inform the detailed discussion of our implementation later.

Let x and y be sequences of lengths m and n, and let $M_{i,j}$ be the score of an optimal (highest-scoring) alignment between any suffixes of strings $x[1..i]$ and $y[1..j]$ ($0 < i \leq m$, $0 < j \leq n$). Smith-Waterman computes $M_{i,j}$ by the following dynamic programming recurrence:

$$\begin{aligned} M_{i,j} &= \max\{M_{i-1,j-1}+\delta(x[i],y[j]),I_{i,j},D_{i,j},0\} \\ I_{i,j} &= \max\{M_{i,j-1} - g_o,I_{i,j-1}\} - g_e \\ D_{i,j} &= \max\{M_{i-1,j} - g_o,D_{i-1,j}\} - g_e. \end{aligned}$$

$M_{i,j}$ is the best of four possibilities: the best alignment may align residues $x[i]$ and $y[j]$, or it may leave either $x[i]$ or $y[j]$ unaligned, or (if all these possibilities yield alignments with negative scores) it may leave $x[1..i]$ and $y[1..j]$ entirely unaligned, with score 0. The I and D portions of the recurrence track the scores of optimal alignments ending with a gap in x or y, respectively. The recurrence is initialized with $M_{0,j} = M_{i,0} = 0$ and $I_{0,j} = D_{i,0} = -\infty$. An optimal local alignment then has score $\max_{i,j} M_{i,j}$, which is computed in time $\Theta(mn)$. While the above algorithm only finds an optimal alignment's score, it is possible to recover the underlying alignment from the intermediate values computed by the recurrence.

The classical Smith-Waterman algorithm described above differs from BLASTP's implementation in one key respect: while Smith-Waterman explores all possible local alignments between two proteins to find the one with highest score, BLASTP's gapped extension receives a “hint,” in the form of an HSP, indicating where in these proteins a good alignment may be found. To exploit this hint, BLASTP chooses a fixed pair of coordinates (q_0, d_0) within the HSP, then extends the alignment forward and backward from this point in each sequence, this time allowing for gaps in either. The result is the highest-scoring local alignment that passes through the point (q_0, d_0).

The dynamic programming recurrence used by extension is the same as that for Smith-Waterman, except for its choice of initialization conditions. BLASTP's gapped extension stage again uses an X-drop heuristic, similar to that used for ungapped extension, to decide when to terminate extension in each direction. More details are given in [Altschul et al. 1997].

3.4 Execution profile of BLASTP

Table I 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

Mercury BLASTP accelerates the BLASTP pipeline by exploiting opportunities for fine-grained parallelism in its various stages. We strive to maintain the same basic stage algorithms as software BLASTP, so as to maximize our results' agreement with those of the standard software implementation. However, the limitations of the target FPGA platform, in particular its limited on-chip and near-chip storage, sometimes demand changes to achieve high performance in practice.

This section describes in detail the algorithmic approaches and architectures used to accelerate each of the three stages of BLASTP.

4.1 Seed generation

Accelerating seed generation requires managing a computation with low input but high output bandwidth. A *w*-mer read from the database typically occurs in the neighborhood of multiple query locations; hence, as shown in Table I, the first sub-stage of seed generation typically *expands* each input *w*-mer into a stream of several word hits. Efficiently retrieving these hits and filtering them to produce two-hit seed matches is the key challenge faced by this stage. Although both word matching and two-hit generation allow for substantial parallelism, the need to parallelize *both* computations at once requires careful bookkeeping to produce output similar to that of a purely serial implementation.

4.1.1 Word matching substage—The word matching module (Figure 6) 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. As described in Section 5, implementation constraints dictate a word size $w = 4$; for this value of w, the lookup 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: $H(r) = 20^{w-1}r[w-1] + 20^{w-2}r[w-2] + \ldots + r[0]$. For fixed w, this computation, which is carried out by the *w-mer2key* 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 7 illustrates the data path for a *w*-mer lookup, including primary table entries with and without duplicate table pointers.

Every lookup into the duplicate table increases a *w*-mer's service time and so reduces the throughput of word matching. It is therefore desirable that *w*-mer lookups be satisfied with as few duplicate table accesses as possible, ideally none. To reduce the average number of duplicate table lookups per *w*-mer, 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.

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 $2047 - w$.

In empirical BLASTP experiments comparing bacterial proteomes, we found that by using delta encoding, we could satisfy 82% of *w*-mer lookups with a single SRAM probe; with a naive encoding using 11 bits per query position, only 67% of lookups were so satisfied. No *w*-mer resulted in more than six SRAM accesses with the delta-encoded representation; with a naive implementation, up to nine accesses were sometimes needed.

4.1.2 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') 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 increasing order of their database positions. However, as detailed in Section 4.1.4, 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 8(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 8(b), our treatment of out-of-order word matches is not perfect and 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 below, our implementation exhibits high sensitivity even though it sometimes loses seed matches.

4.1.3 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 9 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.

4.1.4 Increasing throughput with lookup table replication—With the downstream stages capable of handling word matches at greater than one match per clock, throughput 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 $(l + \lceil L/3 \rceil) \times (h-1)$ residues, where $L = 15$ is the

maximum number of query positions per w-mer stored in our lookup table, and $l = 4$ cycles is the latency of access to the duplicate table.

4.1.5 Routing between multiple parallel substages—Because our architecture permits both multiple hit generators and multiple two-hit modules, it must support routing the output of each hit generator to any of several two-hit modules. To address this problem with h hit generators and b two-hit modules, Mercury BLASTP implements a two-phase switching network between these substages. 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 10) 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.

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 11 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.

4.2 Ungapped extension

The BLASTP software's ungapped extension algorithm, with its X-drop heuristic, allows it to recover HSPs of arbitrary length while limiting the average search space for a given seed match. However, the algorithm's flexibility makes its implementation in an FPGA difficult because the regions of extension around a seed match can, in theory, be arbitrarily long. Hence, the software's algorithm is not suitable for fast implementation in an FPGA. Mercury BLASTP therefore takes the more FPGA-friendly approach by finding the best extension within a small, fixed-size window around the seed match.

A closely related ungapped extension design for BLAST on DNA sequences was reported in our earlier work [Lancaster et al. 2005; 2008]. This section combines a brief review of that design with more detailed discussion of BLASTP-specific adaptations to accommodate the score matrix, δ, which is not used in DNA comparisons.

4.2.1 Algorithm—Extension for a given seed match is performed in a single forward pass over a fixed-size window. These features of our approach simplify hardware implementation and expose opportunities to exploit fine-grain parallelism and pipelining that are not easily accessed in NCBI BLASTP's algorithm. Our extension algorithm is given as pseudocode in Algorithm 1.

Algorithm 1 Mercury BLASTP extension algorithm pseudocode.

```
1:  procedure EXTENSION(seed match)
2:      Calculate window boundaries
3:      Γ ← γ ← 0                                  • Initialize max and running score to 0
```

```
4:    B ← B_max ← E_max ← 0                                  • Initialize beginning and end values to 0
5:    for i ← 1, L_w do                                      • Iterate across window
6:      γ ← γ + δ(q_i, s_i)
7:      if γ > 0 then
8:        if γ > Γ and i > SeedMatchEnd then                  • Update best HSP
9:          Γ ← γ
10:         B_max ← B
11:         E_max ← i
12:       end if
13:     else if i < SeedMatchStart then
14:       B ← i + 1
15:       γ ← 0
16:     end if
17:   end for
18:   if Γ > T or B_max = 0 or E_max = L_w then
19:     return True
20:   else
21:     return False
22:   end if
23: end procedure
```

Extension begins by calculating the limits of a fixed window of length L_w, centered on the first word match of a seed match, in both query and database stream. The appropriate substrings of the query and the stream are fetched into buffers. Once these substrings are buffered, the extension algorithm begins.

Extension implements a dynamic programming recurrence that simultaneously computes the start and end of the best HSP in the window. First, the score contribution of each residue pair in the window is computed, using the same score matrix, δ, as the software implementation. These contributions can be calculated independently in parallel for each pair. Then, for each position i of the window, the recurrence computes the score γ_i of the best (highest-scoring) HSP that terminates at i, along with the position B_i at which this HSP begins. These values can be updated for each i in constant time. The algorithm also tracks Γ_i, the score of the best HSP ending at *or before* i, along with its endpoints B_{max} and E_{max}. Note that Γ_{Lw} is the score of the best HSP in the entire window. If Γ_{Lw} is greater than a user-defined score threshold, the seed passes the filter and is forwarded to software ungapped extension.

Two subtleties of Mercury BLASTP's algorithm should be explained. First, our recurrence requires that the HSP found by the algorithm pass through its original seed match; a higher-scoring HSP in the window that does not contain this seed match is ignored. This constraint ensures that if two distinct biological features appear in a single window, the seed matches generated from each have a chance to generate two independent HSPs. Otherwise, both seed matches might identify only the feature with the higher-scoring HSP, causing the other feature to be ignored. Second, if the best HSP intersects the bounds of the window, it is passed on to software regardless of its score. This heuristic ensures that HSPs that might extend well beyond the window boundaries are properly found by software, which has no fixed-size window limits, rather than being prematurely eliminated.

4.2.2 Architecture—Figure 12 shows an overview of the architecture of the ungapped extension stage. The controller is responsible for managing the stage. The query is stored in on-chip block RAM, while the database flows through an on-chip circular buffer. As each seed match arrives at the stage's input, a window of residues centered on the seed is extracted from this buffer by the *window lookup module* and passed on to the dynamic programming hardware.

Figure 13 shows a detailed view of the window lookup module. The query is stored on-chip in block RAMs and is loaded at the beginning of a database search. The database is too large to buffer in its entirety, so a specialized circular buffer, also in block RAMs, was built to hold the needed portion of the database to form the windows which are also stored in block RAMs.

The compute window bounds module takes in a seed match and is responsible for calculating the boundaries of the window, requesting the window from the buffer modules, and finally constructing the final window from the raw output of the buffer. The darkened boxes are pipeline registers to keep the input data in lock-step with the lookup process.

After the window is retrieved from the buffers, it needs further processing to appropriately align the two sequences since it is possible that a seed does not fall on a word boundary. A four stage shifting tree was built to efficiently implement this functionality, since the synthesizer did an extremely poor job implementing it from a behavioral specification in VHDL. Finally, an aligned window flows to the scoring module.

A potential limit on the implementation of this stage is the number of simultaneous accesses required to the buffered query and database sequences. Because on-chip block RAMs support a limited number of accesses per cycle (at most two in our hardware), multiple copies of the data must be kept to satisfy all consumers in the design. To reduce the amount of on-chip RAM needed for buffering, the block RAMs in this stage are time-multiplexed to allow four accesses in a single clock cycle. Figure 14 shows a block diagram illustrating the design of a quad-ported block RAM. Quad-porting the block RAMs halves the number of physically dual-ported memories required for this application. Note that quad-porting the block RAMs does not increase the size of the RAM; it simply increases the ability to access this memory. A notable limitation of this technique is the requirement of a frequency-doubled clock, which can lower the maximum frequency at which a design can operate.

The ungapped extension scoring function is implemented as a pipelined systolic array. Saturating arithmetic is used both to shorten the critical paths of the compute logic and to reduce area usage. This improvement is possible since the reduction in the number of bits used to represent the recurrence variables outweighs the increased complexity of using saturating arithmetic.

The first stage of the array, the score table, determines the score, δ, for each residue pair. Since these scores are all independent, they are computed in parallel. This is illustrated in Figure 15.

Next, the scores for the window are passed to the scoring stages shown in Figure 12. Each scoring stage implements two steps of the recurrence, for a total of 32 scoring stages for the selected window size. The final step compares the score to a threshold and makes a decision whether to keep or discard the scores. Note that the number of arrows between scoring stages decreases along the pipeline in Figure 12. This is due to the fact that each residue pair is only inspected once and the new algorithm only moves in one direction, so that the unneeded data can be discarded after it has been used. If an HSP scores above the threshold (or meets the other two conditions mentioned in the previous discussion) it is sent to the next stage for gapped extension.

4.3 Gapped extension

Mercury BLASTP's gapped extension stage, like the foregoing ungapped stage, restricts extension to a fixed-size window. Within this window, the stage implements a slimmed-down Smith-Waterman recurrence centered on an HSP to decide whether the HSP should be subjected to full-length gapped extension by the BLASTP software.

4.3.1 Algorithm—The gapped extension procedure of software BLASTP, while substantially less expensive than full Smith-Waterman between protein, computes an irregular pattern of cells, depending on the dynamic values of their computed scores. Figure 16(a) illustrates one such pattern. While variability in the set of cells computed is easily supported in software, it is more challenging for a hardware implementation. Mercury BLASTP therefore implements *banded Smith-Waterman* [Altschul and Gish 1996], an extension procedure that always computes a regular, fixed-size band of cells while still performing much less computation than full Smith-Waterman.

Banded Smith-Waterman scores only those alignments that fall entirely within a fixed-size band of diagonals centered on the input HSP, as shown in Figure 16(b). In what follows, we will view this band as being composed of *antidiagonal strips* (or simply "antidiagonals") running perpendicular to the diagonals, i.e. sets of cells (i, j) with the same value of $i + j$. The geometry of a band is defined by two parameters: the *band length* λ, which is the number of antidiagonals in the band, and the *band width* ω, which is the number of cells on each antidiagonal. It can be shown that this band covers exactly $\omega+\frac{\lambda}{2}$ residues in each of the two sequences Given a band, the banded Smith-Waterman algorithm seeks the highest-scoring alignment that is completely confined to the band. The basic recurrence is identical to full Smith-Waterman except that an infinite penalty is assessed on alignment paths that leave the band.

An efficient hardware implementation of banded Smith-Waterman exploits the following parallelism inherent in the algorithm Call the set of three values $M_{i,j}$, $I_{i,j}$, and $D_{i,j}$ the i, jth *cell* of the computation Cell i, j is dependent on only three other cells for its values, namely cells $i - 1, j$; $i, j - 1$; and $i - 1, j - 1$. Hence, all ω cells in a band with the same sum of indices $i + j$ (i.e. those on the same antidiagonal) may be computed simultaneously, once the cells on lower-numbered antidiagonals have been computed. The banded computation therefore proceeds along successive antidiagonals in stair-step fashion, computing all cells of each antidiagonal in parallel.

To ensure that the optimal alignment found in the band is associated with the HSP at its center, Mercury BLASTP imposes the constraint that the alignment must cross the antidiagonal at the center of the HSP. This constraint is implemented by not clamping the alignment score to zero (as in the standard Smith-Waterman recurrence) once the center is crossed, thus forbidding the alignment to start anew. Moreover, the algorithm returns only the best alignment score observed on any antidiagonal *after* that of the HSP's center, rather than the best score overall.

4.3.2 Architecture—Banded Smith-Waterman, like the classical algorithm, can be accelerated in hardware using a systolic array design that pipelines the computation of successive antidiagonals. This section describes such a design, with emphasis on the work required to make it compatible with the streaming computational model of BLASTP and the strictures of the banded algorithm.

An overview of our design is shown in Figure 17. It is divided into three pieces: control, buffering and storage, and the actual banded Smith-Waterman computation.

4.3.3 Control—Control is implemented using three finite-state machines and several FIFOs. Each state machine is responsible for a single task. The receive FSM accepts incoming commands and data, processes the commands, and directs the data to the appropriate buffer. All commands and data to leave the stage are queued, and the send FIFO pulls them out and sends them downstream (to software). The compute FSM is responsible for controlling the banded Smith-Waterman computation (see Figure 18). It serves the important functions of calculating the band geometry, initializing the computational core, stopping an alignment when it passes or fails, passing successful alignments to the send FSM, and resetting the computation core to perform the next alignment.

4.3.4 Storage and buffering—Several parameters and tables need to be stored in addition to the query and the database. The requisite parameters are the length of the band, λ, the extend penalty, *e*, and the open-extend penalty, *d*.

The query is, at most, 2048 residues in length, or 1.25 KB, so it easily fits in a single block RAM. Since residues are consumed sequentially starting from an initial offset, the query buffer provides a FIFO-like interface. The initial address is loaded, after which each residue request by the compute FSM increments a counter in the query buffer.

The database is too large to be stored on chip, so only the active portion is locally buffered. The buffer size is determined mainly by the need to support HSPs arriving out of order with respect to their database positions – a problem caused by out-of-order seed match generation in stage 1. To accommodate out-of-order arrivals, the buffer keeps a window of 2048 residues behind the database start of the current HSP. The typical distance between out-of-order HSPs is only 40 residues, so almost all such cases of out-of-order arrival are resolved by the buffer. In the exceptional case that an HSP falls too far behind its predecessor in the stream, the buffer flags an error, causing the failing HSP to be passed on unconditionally for processing in software.

To avoid the need to stall the systolic array once a Smith-Waterman computation begins, the database buffer blocks a pending computation on an HSP until it has received the full window of $\omega+\frac{\lambda}{2}$ database residues surrounding the HSP. The compute FSM is responsible for waiting for the buffer to signal its readiness to proceed with the computation.

4.3.5 Banded Smith-Waterman core—Computation of the cells on each antidiagonal is handled by the banded Smith-Waterman core. This core is implemented as a standard systolic array that computes the ω cells of each antidiagonal in parallel. Figure 18 shows its main components: the Smith-Waterman cell array, the MID register block (retaining the recurrence values *M*, *I*, and *D*), the score block, the database-query shift register, and the pass-fail block.

Each cell in the systolic array implements logic to compute the recurrence for a single Smith-Waterman cell. It consists of four adders, five maximizers, and a two-input mux to clamp scores either to zero, for cells before the HSP's center, or to negative infinity, for cells after the center. Clamping to zero is part of the basic Smith-Waterman recurrence, while clamping to negative infinity helps to prevent the algorithm from returning alignments unrelated to the HSP.

The cell values computed by the array are stored in the *MID register block*. Because only the score of the optimal alignment is computed and not the alignment itself, only the two most recent antidiagonals need be stored. As shown in Figure 19, four registers in each cell store the *M*, *I*, and *D* values computed in the previous clock cycle and the *M* value computed two cycles prior. Some of the local dependencies of a cell vary according to whether it is on an odd or even antidiagonal (the leftmost antidiagonal is odd). For odd cells, $M_{i-1,j}$ and $I_{i,j-1}$ are retrieved

from its left neighbor, while for even cells $M_{i,j-1}$ and $D_{i-1,j}$ are retrieved from its right neighbor. Multiplexers are used in the MID register block to fetch the appropriate data dependencies for each antidiagonal.

The *score block* generates a signed 8-bit δ score for each residue pair considered by a cell. As in the ungapped design, δ is stored as a table in on-chip block RAM and addressed by a concatenated residue pair. On our FPGA, each block RAM provides two independent read ports; we therefore use $\omega/2$ block RAMs to service the systolic cell array.

To perform gapped extension on a pair of sequences, the entire query is first loaded into on-chip block RAMs, after which the database sequence is streamed in through the database buffer. The active query and database residues are stored in a pair of parallel-tap *shift registers*, whose values are read by the score block. Residues are shifted in, one per clock cycle, during the computation of odd antidiagonals for the subject and even antidiagonals for the query.

The *pass-fail block* simultaneously compares the ω cell scores in an antidiagonal against a threshold. If any cell value exceeds the threshold, the HSP is deemed significant and is immediately passed through to software for further processing. We implement the following optimization to terminate extension early in some cases. Once an alignment crosses the HSP, its score is never clamped to zero but may become negative. If we observe only negative scores in all cells on two consecutive antidiagonals, extension is terminated. Most HSPs that yield no high-scoring gapped alignment are rapidly discarded by this optimization.

The worst-case running time of gapped extension per HSP is exactly $5 + \omega + \lambda$ clock cycles (5 to compute band geometry and initialize the database buffer, ω to load the shift registers with initial residues, and λ for the score computation). Using the start table described below as well as optimizations to support early alignment termination, gapped extension saves an average of 56% of the naive algorithm's running time for $\omega = 65$ and $\lambda = 1601$ on typical protein datasets.

4.4 Supporting packed query sequences

Protein sequences are typically only about 300 residues long. However, the hardware on which Mercury BLASTP is implemented possesses sufficient SRAM and block RAM to support query sequences of over 2000 residues. For every query loaded into the hardware, the entire database must be fed through the accelerator; hence, it is advantageous to concatenate several small query sequences into one larger *composite query* that is compared to the database in a single pass. Such *query packing* reduces the number of passes over the database and hence the overall search time.

Query packing is supported by all three stages of Mercury BLASTP. Seed match generation trivially supports hashing of concatenated queries in a single table. The two extension stages, however, must treat the components of each packed query differently, because each query protein comes with its own associated thresholds for retaining HSPs and gapped alignments. Ungapped extension simply uses the lowest threshold of any query protein in each concatenated block, relying on later stages to discard HSPs that should otherwise have been suppressed. Gapped extension takes a more sophisticated approach, using *threshold* and *start* tables to support packed query sequences. The threshold table maps each position in a packed query to the threshold score corresponding to the component query sequence at that location. The start table is used to identify the start of the current component at any point in the packed query. Knowing this start permits extension to begin at the start of an HSP's component, rather than considering the full composite query. Table II shows an example of these two tables.

5. Hardware Parameter Selection

We now investigate the parameters of the algorithm 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 Performance model

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

$$\mathrm{Tput}_{\mathrm{pipe}}=\min\left(\mathrm{Tput}_{1a},\mathrm{Tput}_{1b},\mathrm{Tput}_{2},\mathrm{Tput}_{3}\right),$$

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 input database residues processed per second.

The throughput of seed generation is computed as

$$\mathrm{Tput}_{1a}=\frac{h\times f_1}{c_{1a}}\qquad \mathrm{Tput}_{1b}=\frac{b\times f_1}{c_{1b}\times r_{1a}}$$

Based on our synthesis results, we use a clock frequency f_1 = 110 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[3]. 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.

We model the normalized throughput of stage 2 as

$$\mathrm{Tput}_{2}=\frac{f_2}{c_2\times(r_{1a}r_{1b})}$$

where the ungapped extension stage runs at f_2 = 85 MHz and is capable of accepting one seed per clock cycle (i.e., c_2 = 1). The product $r_{1a}r_{1b}$ represents the number of seeds passed into stage 2 per input residue.

Finally, the normalized throughput of stage 3 is modeled as

[3]Technically, c is the inter-arrival time, in clocks, for each stage, since the stages are internally pipelined.

$$\text{Tput}_3 = \frac{f_3}{c_3 \times (r_{1a} r_{1b} r_2)}$$

where the gapped extension stage runs at f_3 = 90 MHz. The value c_3 represents the average number of clock cycles required to process an input HSP in the gapped extension stage, and $r_{1a}r_{1b}r_2$ represents the number of HSPs passed into stage 3 per input residue.

5.2 Parameters for Seed Generation

The key parameters for the seed generation are word length w and neighborhood score threshold T, which control both the sensitivity of the computation and the time spent in its bottleneck table lookup operation. We first describe a range of these parameters that empirically achieve high sensitivity, then indicate how our hardware resource constraints shaped the final parameter selections. We also discuss the effect of query length on the performance of this stage.

5.2.1 Ensuring high sensitivity—To generate a baseline for measuring 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 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 20 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.

5.2.2 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 IV. 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.

As described above, Mercury BLASTP can concatenate multiple short protein queries to form one long query, thereby reducing the number of database passes and hence the running time needed to process a large query set. However, 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. Limits on the memory available for the lookup table therefore imply limits on feasible query length, even if the query sequence itself is short enough to fit on chip.

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. In addition, 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 (for a single copy of the word matching module). The word match generation rate r_{1a} also drops by approximately half, from 3.873 to 2.007, reducing the load on the downstream stages.

5.3 Parameters for ungapped extension

The parameters of importance in the ungapped extension stage are the window length L_w and score threshold T. The window length must be long enough to accurately distinguish between interesting HSPs and seed matches which occur by chance alone. Increasing the window length allows ungapped extension to more accurately make this distinction by using a higher value for T. However, the window must be made as small as is practical since the amount of FPGA resources used scales linearly with L_w. Additionally, decreasing T by too much will cause a large load increase to the downstream stages. To summarize, the goal is to use the smallest L_w with the highest T while still making good decisions on whether to discard a seed match.

In order to quantitatively evaluate the minimum necessary window length, NCBI BLAST was instrumented to count the number of comparisons performed per seed match in its ungapped extension stage. Random samples from the E. coli proteome were searched against the GenBank Non-Redundant database. For each query, histograms were collected representing the number of comparisons performed for each seed match (and hence, the size of the search space). Figure 21(a) shows the measurement for all seed matches, regardless of whether the seed passed the threshold or not. The graph shows us that, after inspecting approximately 60 residues, more than 95% of the seeds can be decided. However, this does not tell us how much of the work being performed by software extension is useful. To gage this, statistics were gathered on only the seed matches which scored lower than T (and were then rejected by ungapped extension). Figure 21(b) shows the number of bases inspected before a low-scoring HSP was rejected. In this case, slightly fewer bases were inspected before rejecting a seed match. Note that these distributions are very sensitive to T.

To strike a balance between resource utilization, sensitivity and performance, Mercury BLASTP uses $L_w = 64$ and a slightly lowered (as compared to NCBI BLASTP default) score threshold of 16. Empirical evaluation of sensitivity was carried out in [Lancaster et al. 2008], which indicates these parameters are sufficient in terms of both sensitivity and throughput.

Note that neither of the above parameters has an impact on the throughput of the ungapped extension stage, which can accept a single seed each clock (i.e., $c_2 = 1$), independent of the parameter values. The impact of the parameters is limited to FPGA area consumed (for L_w) and sensitivity (for both L_w and T).

5.4 Parameters for gapped extension

Stage 3 throughput is dependent on the band geometry, specifically the band length λ and band width ω. The goal of high sensitivity dictates that these parameters be made as large as possible,

to most closely approximate full Smith-Waterman on the input HSP. However, band width impacts the width of the stage's systolic array, and hence overall area, while band length impacts time spent in the stage per HSP processed.

Figure 22 plots the average HSP processing time, c_3, in clocks, as a function of band length for several band widths, while Figure 23 plots the estimated stage 3 throughput as a function of band length for the same band widths. Variation in band width within the limits dictated by available logic area empirically had little effect on throughput. However, throughput is inversely proportional to band length, which determines the number of update steps to the systolic array per HSP.

Our final design uses parameters $\omega = 65$ and $\lambda = 1601$ with a maximum estimated throughput of 423 Mres/sec. The band length was chosen empirically to minimize the number of significant gapped alignments lost relative to the NCBI BLASTP software. Larger bandwidths were feasible to implement but were found not to significantly improve sensitivity.

6. Mercury Blastp Software Architecture

Mercury BLASTP requires tight co-ordination between the FPGA resources and software running on the host CPU. In the current implementation, stages 1 and 2 of the pipeline are deployed on one FPGA and stage 3 is deployed on a second FPGA. In the following section we describe the software infrastructure supporting the Mercury BLASTP system. In particular, we discuss the preprocessing stages of neighborhood generation and query packing and their effect on the BLASTP pipeline.

6.1 Architectural overview

The software is organized as a multi-threaded application consisting of independently executing components communicating via queues. A major goal in the design of the software system was to integrate the Mercury code into the existing NCBI BLAST package. The NCBI BLASTP pipeline was modified and FPGA resources substituted for the software equivalents.

There are two main advantages to using the NCBI codebase. Fundamental support routines such as I/O processing, query filtering, and the generation of sequence statistics can be reused. Further, support for additional BLAST programs such as BLASTX and TBLASTN can be added with minimal work at a later time. Secondly, the user interface, including command-line options, input sequence format, and output alignment format is preserved. This facilitates transparent migration for end users and seamless integration with the large set of applications designed to work with NCBI BLAST.

The BLASTP initialization code executes part of the traditional NCBI pipeline that creates the state for the search process. The Mercury query data structures are then loaded and search parameters initialized in hardware. The database is then streamed through the first FPGA executing the first two stages of the BLASTP pipeline. HSPs generated by ungapped extension are multiplexed with the database stream and sent back to the host CPU. Banded gapped extension on the second FPGA consumes the stream and tags HSPs that will likely lead to gapped alignments. Finally, the NCBI BLASTP software pipeline is resumed on the host to filter the output of the second FPGA and produce the final gapped alignments. For ease of integration, results from the second FPGA are inserted into the software BLASTP pipeline at its ungapped extension stage and are validated using both the software's ungapped and gapped filters.

6.2 Neighborhood generation

Much of the query pre-processing time in the BLASTP pipeline is spent generating the neighborhood of a query, which is then encoded into the tables used by the word matching module. A naive algorithm for neighborhood generation would score all possible 20^w w-mers against every w-mer in the query sequence, adding those that score greater than or equal to T into the neighborhood. Indeed, this implementation is essentially what is found in NCBI BLASTP.

NCBI BLASTP's neighborhood generation is both memory-intensive and computationally expensive, degrading exponentially at longer word lengths. We now describe a prune-and-search algorithm that has the same worst-case bound but shows practical improvements in speed. The algorithm divides the search space into a number of independent partitions, each of which is inspected recursively. At each step, it is possible to determine if there exists at least one w-mer in the partition that must be added to the neighborhood. This decision can be made without the costly inspection of all w-mers in the partition. Such w-mer partitions are pruned from the search process. Another advantage of this class of algorithms is that they can be easily parallelized. We describe a vector implementation using SIMD technology available on the host CPU that further speeds up neighborhood generation.

6.2.1 Prune-and-search neighborhood—Given a query w-mer r, an alphabet Σ, and a scoring matrix δ, the neighborhood of the w-mer is computed using the following recurrence. The neighborhood $N(w, T)$ of the query Q is the union of the individual neighborhood of every query w-mer $r \varepsilon Q$.

$$N(w,T) = \bigcup_{r \varepsilon Q} G^r(\varepsilon, w, T)$$

$$G^r(x,w,T) = \bigcup_{a \varepsilon \Sigma} \begin{cases} \{xa\} & \text{if } |x| = w-1 \text{ and } S^r(x)+\delta(r_w,a) \geq T, \\ G^r(xa,w,T) & \text{if } |x| < w-1 \text{ and } S^r(x)+\delta(r_{|x|+1},a)+C^r(|x|+1) \geq T, \\ \varphi & \text{otherwise.} \end{cases}$$

$$S^r(x) = \begin{cases} 0 & \text{if } x=\varepsilon, \\ S^r(y)+\delta(r_{|x|},a) & \text{otherwise, where } x=ya. \end{cases}$$

$$C^r(i) = \begin{cases} \max_{a \varepsilon \Sigma} \delta(r_w,a) & \text{if } i=w-1, \\ \max_{a \varepsilon \Sigma} \delta(r_i,a)+C^r(i+1) & \text{otherwise.} \end{cases}$$

$G^r(x, w, T)$ is the set of all w-mers in $N^r(w, T)$ having the prefix x. We term x a *partial* w-mer. The base is $G^r(x, w, T)$ where $|x| = w - 1$ and the target is to compute $G^r(\varepsilon, w, T)$. At each step of the recurrence, the prefix x is extended by one character $a \varepsilon \Sigma$. The pruning process is invoked at this stage. If it can be determined that no w-mers with a prefix xa exist in the neighborhood, all such w-mers are pruned; otherwise the partition is recursively inspected (lines 2 and 3 of the recurrence). The score of xa is also computed and stored in $S^r(xa)$. The base case of the recurrence occurs when $|xa| = w - 1$. At this point it is possible to determine conclusively if the w-mer scores above the neighborhood threshold.

We now describe the pruning step in more detail. During the extension of x by a, the highest score of any w-mer in $N^r(w, T)$ with the prefix xa is determined. This score is computed as the sum of three parts: the score of x against $r_{1..|x|}$, the pairwise score of a against the character $r_{|x|+1}$ and the highest score of some suffix string y and $r_{|x|+2..w}$ with $|xay| = w$. The three score values are computed by constant-time table lookups into S^r, δ, and C^r respectively. $C^r(i)$ holds the score of the highest scoring suffix y of some w-mer in $N^r(w, T)$, where $|y| = w - i$. This is easily computed in linear time using the scoring matrix.

A stack implementation for the computation of $G^r(\varepsilon, w, T)$ is shown in Algorithm 2. We define $\sum_b'$ to be the alphabet sorted in descending order of the pairwise score against character b in δ. The w-mer extension is done in this order, causing the contribution of the δ lookup in the left-hand side of the expression on line 11 to progressively diminish with every iteration.

Algorithm 2 Stack implementation of the prune-and-search algorithm

```
1:  procedure PS-NEIGH(w, T, r) w-mer r                    • Generate neighborhood N(w, T) for query
2:      G ← φ                                              • Initialize neighborhood set
3:      STACK.PUSH(ε)                                      • Initialize target of recurrence
4:      repeat
5:          x ← STACK.POP( )                               • Pop next partial w-mer
6:          for all a ε Σ'_{r|x|+1} do                     • Cycle through alphabet, sorted by pairwise score
7:              if |x| = w − 1 then                        • Base case
8:                  if S^r(x) + δ(r_w, a) ≥ T then
9:                      G ← G ∪ {x · a}
10:                 end if
11:             else if S^r(x) + δ(r_{|x|+1}, a) + C^r_{|x|+2} ≥ T then
12:                                                        • Partition contains at least one w-mer in neighborhood: store for later search
13:                 STACK.PUSH(x · a)
14:             else                                       • All remaining partitions guaranteed to score poorer: prune
15:                 break for
16:             end if
17:         end for
18:     until STACK.EMPTY( )
19:     return G
20: end procedure
```

6.2.2 Vector implementation—In the Prune-and-search algorithm described, extension of a partial w-mer by every character in the alphabet can be done independently. We exploit the resultant data parallelism by vectorizing the computation in the *for* loop of Algorithm 2.

Algorithm 3 shows a vector implementation. In each iteration of the loop, the algorithm extends a partial w-mer by *VECTOR_SIZE* characters. Lines 21-27 perform the comparison operation with the returned mask value being inspected to determine the result.

We use SSE2 extensions available on the ×86-family host CPU for our implementation. A vector size of 16 and signed 8-bit integer data values are used. Saturated signed arithmetic is used to detect overflow/underflow. The alphabet size is increased to the nearest multiple of 16 by introducing dummy characters, and the scoring matrix is extended accordingly.

Algorithm 3 Vector implementation of the prune-and-search algorithm

```
1:  procedure PS-NEIGH-VECTOR(w, T, r)
2:                                                  • Generate neighborhood N(w, T) for query w-mer r using vector instructions
3:      T⃗ ← {T − 1, …, T − 1}                       • Initialize threshold vector
4:      G ← φ
5:      STACK.PUSH(ε)
6:
7:      repeat
8:          x ← STACK.POP( )
9:          Σ″ ← Σ′_{r|x|+1}                        • Retrieve sorted alphabet list for this w-mer residue
10:         δ″ ← δ′ (r_{|x|+1})                     • Retrieve corresponding pairwise scores
11:         S⃗ ← {S^r(x), …, S^r(x)}
12:         C⃗ ← {C^r_{|x|+2}, ⋯ , C^r_{|x|+2}}
13:         P⃗ ← VECTOR-ADD(S⃗, C⃗)                   • Precompute loop invariant vector
14:
15:                                                 • Cycle through alphabet, VECTOR_SIZE characters per iteration
16:         for i ← 1, |Σ|, VECTOR_SIZE do
17:             δ⃗ ← {δ″(Σ″_i), …, δ″ (Σ″_{i+VECTOR_SIZE})}   • Initialize score vector
18:
19:         if |x| = w − 1 then                     • Base case
20:             A⃗ ← VECTOR-ADD(S⃗, δ⃗)
21:             mask ← VECTOR-CMPGT-GET-MASK(A⃗, T⃗)   • vector set bit, if op1 > op2
22:             pos ← 0
23:             while mask do                       • Locate neighborhood w-mers in vector
24:                 G ← G ∪ {x · Σ″_{i+pos}}
25:                 mask ← mask ≫ 1
26:                 pos ← pos + 1
27:             end while
28:         else
29:             A⃗ ← VECTOR-ADD(P⃗, δ⃗)
30:             mask ← VECTOR-CMPGT-GET-MASK(A⃗, T⃗)
31:             pos ← 0
32:             while mask do                       • Locate partitions in vector not pruned
33:                 STACK.PUSH (x · Σ″_{i+pos})
```

```
34:                    mask ← mask ≫ 1
35:                    pos ← pos + 1
36:                  end while
37:                end if
38:              end for
39:            until STACK.EMPTY( )
40:            return G
41:          end procedure
```

6.2.3 Results—Table V compares the neighborhood generation times of the three algorithms for 2048-residue query sequence. The benchmark machine was a 2.0 GHz AMD Opteron workstation with 6GB of memory.

The prune-and-search algorithm is 5× faster than the NCBI BLAST enumeration method for $w = 4$. and over 60× faster at $w = 6$. The vector implementation shows a speedup of 3× over the sequential version.

At the default Mercury BLASTP neighborhood of $N(4, 13)$, the naive algorithm consumes approximately 10% of the program execution time. This is especially critical because the rest of the pipeline remains idle until the neighborhood is generated. In contrast, the vectorized prune-and-search implementation is 19× faster, consuming just 0.5% of the execution pipeline.

6.3 Query bin packing

Query bin packing is an optimization intended to speed up the BLAST search process. The Mercury BLASTP pipeline stages are designed to operate at full utilization on query sequences of 2048 residues. However, the average protein sequence is only 300 residues in length, causing the downstream stages to remain idle a large portion of the time. By concatenating multiple short query sequences and processing them in a single pass over the database, the total execution time can be reduced.

However, a number of caveats must first be addressed. To ensure alignments generated do not cross sequence boundaries, an invalid sequence control character is used to separate them. The word matching stage detects and rejects w-mers crossing these boundaries. Similar safeguards are present in the downstream extension stages as described in Section 4.4. In software, the HSP co-ordinates returned by the hardware stages must also be translated to the reference system of the individual components.

The process of packing a set of sequences in an online configuration must be optimized to reduce the overhead to a minimum. The query packing problem is identical to the one-dimensional bin packing problem and is known to be NP-hard. We compare the performance of the *Next Fit* (*NF*) and the *First Fit* (*FF*) algorithms. These algorithms can be improved by first sorting the query list by decreasing sequence lengths before applying the packing rules.

6.3.1 Performance of bin packing—The approximation algorithms were run on 4,241 sequences (1,348,939 residues) of the Escherichia coli k12 proteome with the bin size set to 2048 residues. An optimal solution for this input set uses 661 bins.

As shown in Table VI First Fit performs best, with the sorted list using just one more bin than the optimal solution. This good performance can be attributed to the large number of relatively small query sequences in the data set. Figure 24 shows the histogram of input query sequence lengths. The distribution is heavily biased toward smaller sequences, with 60% of the input set

being less than 300 residues. We believe this data to be representative of protein sequences in general and expect to achieve near-optimal bin packing in practice.

The Mercury BLASTP pipeline is stalled during the query bin packing preprocessing computation. First Fit keeps every bin open until the entire query set is processed. In the case of certain configurations, such as when Mercury BLASTP is used to service requests from a web server this is not feasible. The Next Fit algorithm may be used instead. Since only the most recent bin is inspected in this case, all previously closed query bins may be dispatched for processing in the pipeline.

7. Results

We have implemented Mercury BLASTP on the Mercury prototyping system [Chamberlain et al. 2003], which provides high-throughput data movement between direct-attached disk storage and reconfigurable logic. The system's host processors include two dual-core 2.4 GHz AMD Opteron CPUs with 16 GB of memory, running 64-bit Linux, and two prototyping FPGA co-processor boards connected via the PCI-X bus to the host. Interfacing drivers to the boards are provided by Exegy, Inc[4]. Both boards contain a Xilinx Virtex-II 6000-6 FPGA. The first board runs BLASTP stages 1 & 2; the second runs stage 3. Three synchronous 1 MB SRAM modules are available on the first board. Using this configuration, we have demonstrated sustained data throughput from disk to FPGA well over the requirement for Mercury BLASTP [Chamberlain and Shands 2005].

We now examine the entire Mercury BLASTP deployment shown in Figure 25. Seed generation (with two word match generators, using 2 of the 3 SRAMs, and four two-hit modules) and ungapped extension run at 110 MHz and 85 MHz, respectively, consuming 63% of the slices and 77% of the block RAMs on the first FPGA. Gapped extension runs at 90 MHz, consuming 33% of the slices and 48% of the block RAMs. Our implementation uses two FPGAs due to the large number of block RAMs required by all stages combined; however, the current generation Virtex FPGAs from Xilinx can easily 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. Individual protein 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. 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 and prepares the alignments for reporting to the user.

To quantify the performance of the entire Mercury BLASTP accelerator, we compared it to NCBI BLASTP running on a modern, general-purpose computing cluster with two 2.4 GHz Opteron CPUs and 4 GB of RAM per machine running 64-bit Linux. The individual times of the 8 machines were summed to get the total time to execute NCBI BLASTP for each experiment. Rather than compare Mercury BLASTP's running time to that of the NCBI BLASTP version (2.2.9) from which it was originally derived, we instead used the most recent version of NCBI BLASTP (2.2.17) available at the time of writing. The newer BLASTP software, which represents the end result of a several-year rewrite of BLAST's core functionality, is more than twice as fast as 2.2.9 in our tests and so is a faster competitor to our accelerated implementation.

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

We built NCBI BLASTP with all available compiler optimizations of gcc 3.4. BLASTP runs were performed single-threaded, one on each core of the machine. BLASTP was run with an E-value threshold of 10^{-5} and default parameters otherwise. Reported runtimes include query setup and the time spent in the three stages of the pipeline but do *not* include time spent formatting the final alignments for printing.

To measure the output quality of our implementation, we computed its sensitivity to the alignments returned by software BLASTP. Sensitivity was measured as the fraction of alignments from the software baseline's output that were also found by Mercury BLASTP. Alignments in the two outputs that overlap by more than 50% were considered to be the same. We also report the number of alignments found by the baseline but not by Mercury BLASTP ("Alignments Lost") and vice versa ("New Alignments").

For our sensitivity measurements, we compared our implementation to NCBI BLASTP 2.2.10, which is substantially identical to 2.2.9. We did not use 2.2.17 for sensitivity testing because it dynamically modifies the default scoring matrix δ used to score alignments, which yields significantly different output. For example, we found that 2.2.17's output in the third of our experiments below included only 89.2% of the alignments produced for the same experiment by 2.2.10. Our implementation presently hard-codes the default scoring matrix used by 2.2.10, so we preferred a baseline that isolates the sensitivity effects of our design choices from those caused by unrelated changes in scoring.

We compared Mercury BLASTP and NCBI BLASTP on the following four large proteomic comparisons:

1. *E. coli* K12 proteome (1.35 Mres) vs. GenBank Non-Redundant (NR) database (1.39 Gres);
2. *B. thetaiotaomicron* proteome (1.85 Mres) vs. GenBank NR;
3. *Y. pestis* KIM proteome (1.27 Mres) vs. all other bacterial proteomes in Gen-Bank (282 Mres).
4. *H. sapiens* putative proteins (16.45 Mres) vs. GenBank NR. Due to the enormity of the data generated from this experiment, sensitivity analysis was performed on a random sample of 10% of the query. However, the timing numbers reflect the execution of the entire query in both software and Mercury BLASTP.

These comparisons represent typical annotation tasks that would be performed on proteins predicted from a newly sequenced genome. The new proteins are compared to databases of known proteins, and any statistically significant similarities are recorded as evidence that a known protein and new protein are evolutionarily related. GenBank NR is a database of all known or predicted proteins in the NCBI GenBank archive. In all our tests, query sequences were filtered to remove low-complexity regions.

Tables VII and VIII respectively show the speedup and the sensitivity of Mercury BLASTP relative to the software baseline for our experiments. Mercury BLASTP averages more than an order of magnitude faster than the software baseline for all four experiments, with larger databases giving greater speedups, despite dated FPGA technology. Furthermore, close to 99% of all alignments found by NCBI BLASTP were also detected by our FPGA solution – an insignificant loss in quality compared to the differences between releases of NCBI BLASTP itself. Mercury BLASTP also finds many new, statistically significant alignments not reported by the NCBI software.

8. Conclusion

In this work, we have presented our design of Mercury BLASTP, an FPGA-accelerated implementation of the standard protein sequence comparison tool. Our design closely matches the behavior of the standard BLASTP software while overcoming issues caused by high data generation and filtering rates. The Mercury BLASTP system achieves software-like sensitivity at more than an order of magnitude speedup over NCBI BLASTP on a modern workstation. To the best of our knowledge, Mercury BLASTP is the first FPGA accelerator for the entire BLASTP pipeline.

Overall, Mercury BLASTP is shown to execute at least 10 times faster than NCBI BLASTP running on a modern processor, maintaining close to 99% sensitivity. These results are measured using experimental runs that are indicative of BLASTP usage common in the biological research community.

We expect further performance gains in our implementation from updating it to more recent hardware platforms. In particular, the FPGA devices used in this work are from an older generation. The latest PCI-X based accelerator cards in use by our partners at Exegy, Inc. contain 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 accommodate 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× over our current implementation simply by moving to the latest generation FPGA technology.

Our successful FPGA design for BLASTP extends to bioinformatics applications beyond pure pairwise sequence comparison. Modern biosequence databases are increasingly organized into families of related sequences, such as the same gene in multiple organisms, or groups of proteins from the same functional family. These families are represented as *profiles* – ordered lists of positions, each of which describes the distribution of residues at one point in the sequence. For example, a profile for a family of proteins may show that 70% have residue L in their first position, 20% have residue I, and so forth. Just as BLASTP compares a query sequence to a database of profiles, more recent comparison tools such as PhyloNet [Wang and Stormo 2005] can compare profiles to each other to discover similarity between sequence families. Other tools, such as PSI-BLAST [Altschul et al. 1997] and IMPALA [Schaffer et al. 1999], compare profiles to sequences to discover instances where a sequence is likely a member of a known family.

The design of Mercury BLASTP naturally extends to implement comparisons involving profiles. At a high level, the structure of efficient profile-based comparison tools is similar to BLASTP, including seed matching and extension stages. Extension uses the same dynamic programming approaches described for BLASTP, with the scoring matrix δ on residue pairs replaced by a more complex function δ' on pairs of profile positions. Hence, implementing profile extension in Mercury BLASTP requires only that we replace the scoring blocks of our designs with logic implementing the new scoring functions.

Seed matching between a query profile and a database of sequences is well-defined if δ' is defined for a residue and a profile position; this is the case for PSI-BLAST and related applications. For such applications, one may define the (w, T)-neighborhood of a profile as the set of all w-mers that score at least T when compared to some w contiguous positions in the profile. With this definition, our neighborhood hash-based design can be used unchanged, except for minor modifications to the software that generates neighborhoods. When the

database consists of profiles, as in PhyloNet, the common approach is to apply vector quantization to map all profile positions to one of a small number of representative residue distributions. The resulting quantized sequence of positions can be scored against a query analogously to a sequence of residues, again using a neighborhood hashing strategy. Quantization of profile columns may be performed either offline or as part of the input processing in the Stage 1 FPGA design.

In the end, our experience designing and building Mercury BLASTP yields two general lessons for the designers of all types of accelerators. One important lesson is that it is possible to achieve significant performance gains for real-world, interesting applications using FPGAs, even when the computational workload in the original application is not concentrated in a single hot spot. In particular, a streaming approach accelerates our application not only by exploiting the parallelism of individual application stages but also by pipelining them. The second lesson is the importance of setting, and then testing, goals for output quality as part of the design process. For Mercury BLASTP, the need to closely match NCBI BLASTP's output steered us both to a design that closely follows the structure of the BLASTP software and to parameter choices that yield acceptable sensitivity versus the software baseline. Recognizing such constraints, and then testing them rigorously as part of evaluating the implementation, is key for any design that, like ours, must compromise between acceleration and fidelity to an ideal computation.

Acknowledgments

The two lead authors, A. Jacob and J. Lancaster, contributed equally to the content of this paper. This work was supported by NSF Career grant DBI–0237902, NSF grant CCF–0427794, and NIH/NGHRI grant 1 R42 HG003225–01 (the latter through BECS Technology, Inc.). R.D. Chamberlain is a principal in BECS Technology, Inc.

References

Altschul SF, et al. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. Nucl Acids Res 1997 Sep 17;25:3389–3402. [PubMed: 9254694]

Altschul SF, Gish W. Local alignment statistics. Methods: a Companion to Methods in Enzymology 1996;266:460–80.

Buhler, JD.; Lancaster, JM.; Jacob, AC.; Chamberlain, RD. Mercury BLASTN: Faster DNA sequence comparison using a streaming hardware architecture. Proc. of Reconfigurable Systems Summer Institute; 2007.

Chamberlain, RD., et al. The Mercury System: Exploiting truly fast hardware for data search. Proc. Int'l Workshop on Storage Network Architecture and Parallel I/Os (SNAPI); 2003. p. 65-72.

Chamberlain, RD.; Shands, B. Streaming data from disk store to application. Proc. Int'l Workshop on Storage Network Architecture and Parallel I/Os (SNAPI); 2005. p. 17-23.

Dayhoff MO, Schwartz R, Orcutt BC. A model of evolutionary change in proteins. Atlas of Protein Sequence and Structure 1978;5:345–52.

Herbordt, M., et al. Single pass, BLAST-like approximate string matching on FPGAs. Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM); 2006. p. 217-26.

Herbordt MC, Model J, Sukhwani B, Gu Y, VanCourt T. Single pass streaming BLAST on FPGAs. Parallel Comput 2007;33:10–11. 741–756.

Hirschberg, JD., et al. Kestrel: A programmable array for sequence analysis. IEEE Int'l Conf. on Application-Specific Systems, Architectures, and Processors (ASAP); 1996. p. 25-34.

Hoang, DT. Searching genetic databases on Splash 2. IEEE Workshop on FPGAs for Custom Computing Machines (FCCM); 1993. p. 185-191.

Krishnamurthy P, Buhler J, Chamberlain R, Franklin M, Gyang K, Jacob A, Lancaster J. Biosequence similarity search on the Mercury system. Journal of VLSI Signal Processing 2007;49:101–121.

Krishnamurthy, P.; Buhler, J.; Chamberlain, R.; Franklin, M.; Gyang, K.; Lancaster, J. Biosequence similarity search on the Mercury system. Proc. of the Application-Specific Systems, Architectures and Processors (ASAP); 2004. p. 365-375.

Lancaster, J.; Buhler, J.; Chamberlain, RD. Acceleration of ungapped extension in Mercury BLAST. Proc. of 7th Workshop on Media and Streaming Processors; 2005.

Lancaster J, Buhler J, Chamberlain RD. Acceleration of ungapped extension in Mercury BLAST. Int'l J of Embedded Sys. 2008In press

Lavenier, D.; Guyetant, S.; Derrien, S.; Rubini, S. A reconfigurable parallel disk system for filtering genomic banks. Engineering of Reconfigurable Systems and Algorithms (ERSA); 2003. p. 154-166.

Lin, H., et al. Efficient data access for parallel BLAST. Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS); 2005. p. 72.2

Margulies M, Egholm M, Altman WE, Attiya S, Bader JS, et al. Genome sequencing in microfabricated high-density picoliter reactors. Nature 2005;437:326–7. [PubMed: 16163333]

McGinnis S, Madden TL. BLAST: At the core of a powerful and diverse set of sequence analysis tools. Nuc Acids Res 2004;32:20–5.

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

Portugaly, E.; Ninio, M. HMMERHEAD - accelerating HMM searches on large databases. Proc. Int'l Conf. on Research in Molecular Biology (RECOMB); 2004. p. 250-251.

Rangwala, H., et al. Massively parallel BLAST for the Blue Gene/L. High Availability and Performance Computing Workshop; 2005.

H S, Henikoff JG. Amino acid substitution matrices from protein blocks. Proc Natl Acad Sci U S A 1992 Nov 22;89:10915–10919. [PubMed: 1438297]

Schaffer AA, Wolf YI, Ponging CP, Koonin EV, Aravind L, Altschul SF. IMPALA: Matching a protein sequence against a collection of PSI-BLAST-constructed position-specific score matrices. Bioinformatics 1999;15:1000–11. [PubMed: 10745990]

Smith TF, Waterman MS. Identification of common molecular subsequences. Journal of Molecular Biology 1981;147:195–197. [PubMed: 7265238]

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

Swiss Institute of Bioinformatics. Growth of Swiss-Prot. 2006. http://www.expasy.org/sprot/relnotes/#SPstat

Timelogic, Inc. Timelogic DeCypher BLAST. http://www.timelogic.com/

Wang T, Stormo GD. Identifying the conserved network of cis-regulatory sites of a eukaryotic genome. Proc Natl Acad Sci USA 2005;102:17400–5. [PubMed: 16301543]

Yamaguchi Y, et al. High speed homology search with FPGAs. Pacific Symp on Biocomputing 2002:271–282.

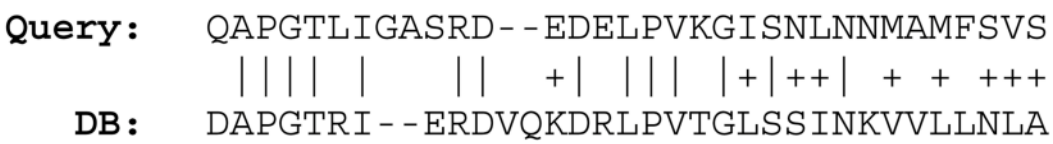


Fig. 1.
Alignment of two protein sequences. Identical and biologically similar residue pairs are marked by vertical lines and pluses, respectively.

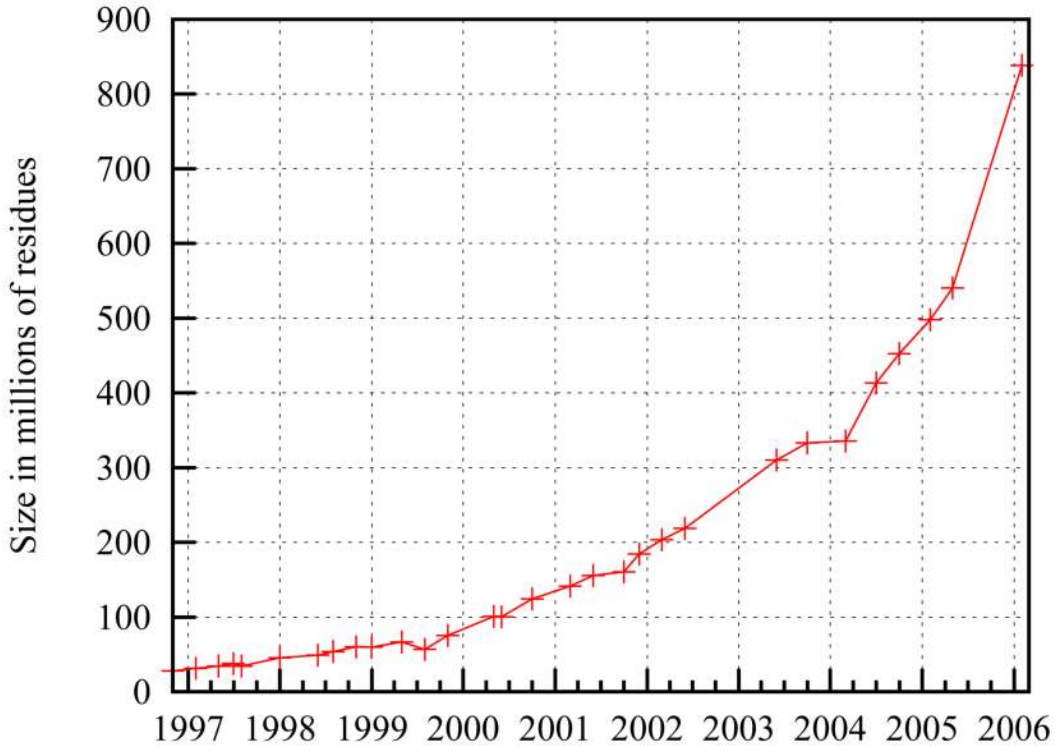


Fig. 2.
Growth of TrEMBL protein database.

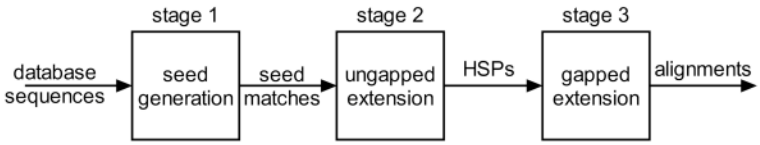


Fig. 3.
The BLASTP computational pipeline.

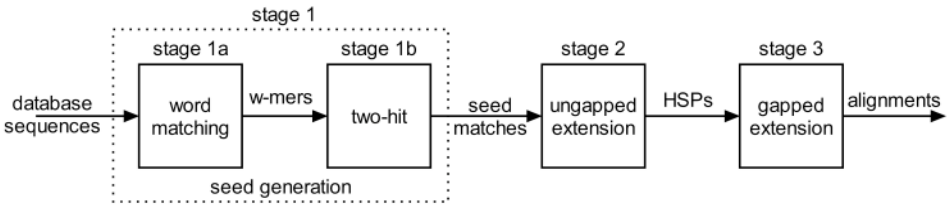


Fig. 4.
Decomposition of BLASTP stage 1.

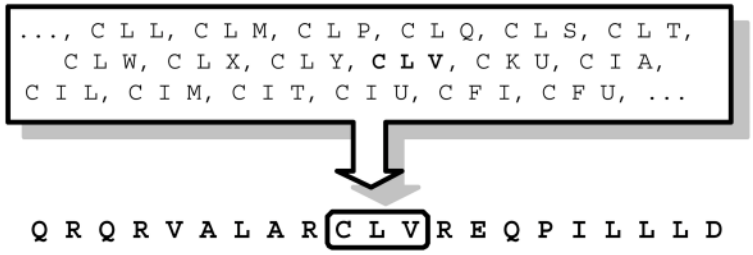


Fig. 5.
Neighborhood of query *w*-mer CLV in a protein sequence.

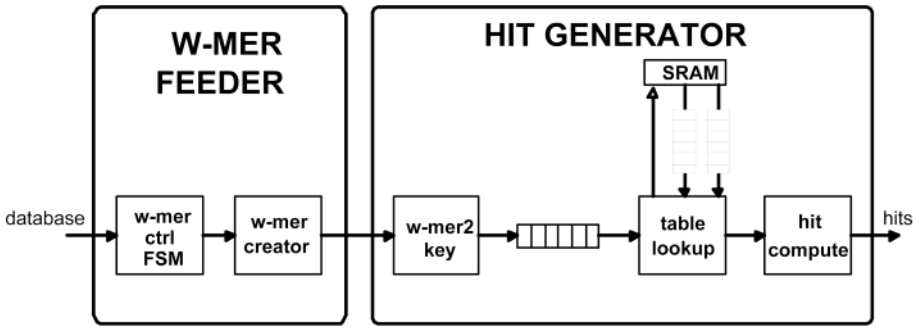


Fig. 6.
Word matching hardware design.

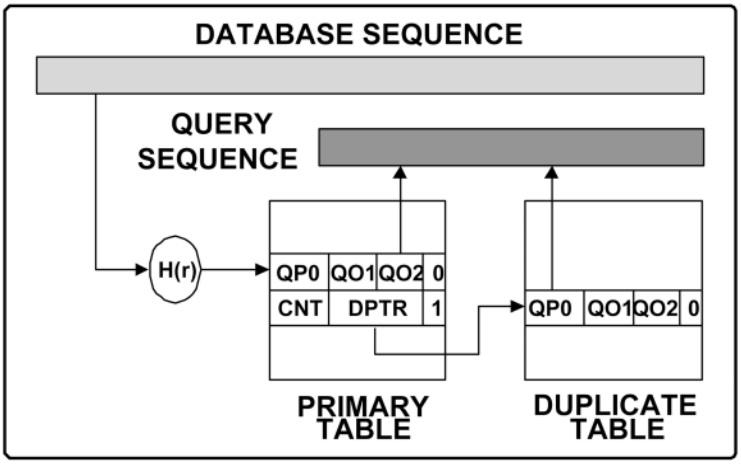
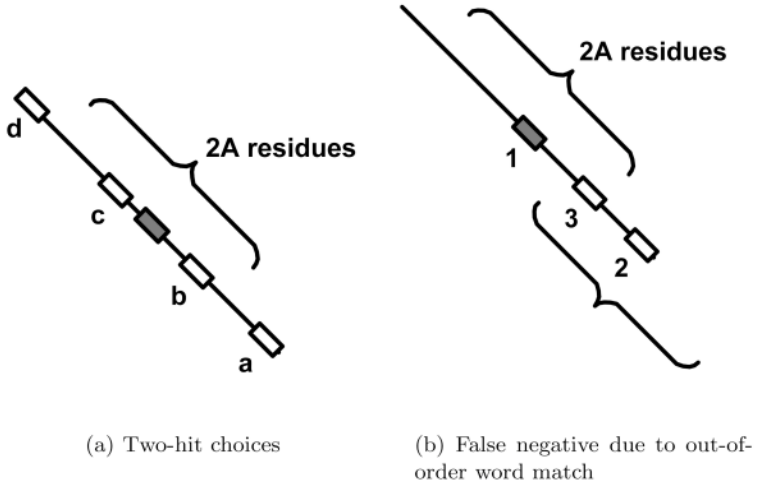


Fig. 7.
Lookup table data path.



(a) Two-hit choices

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

Fig. 8.
Examples of the two-hit computation.

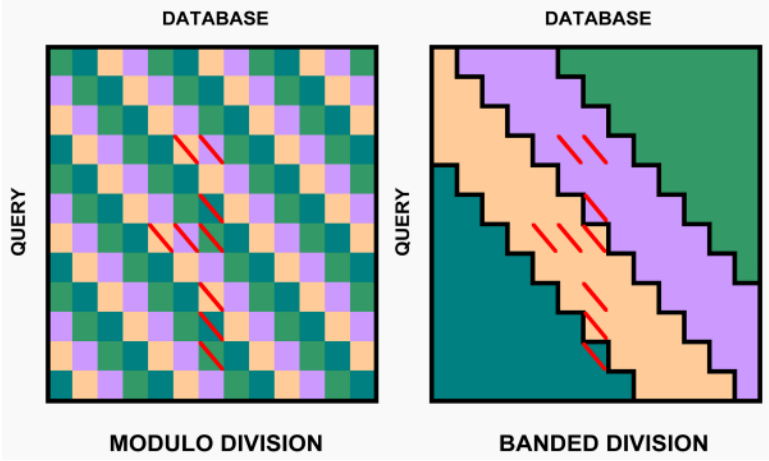


Fig. 9.
Modulo division of diagonals results in more equal distribution of hits.

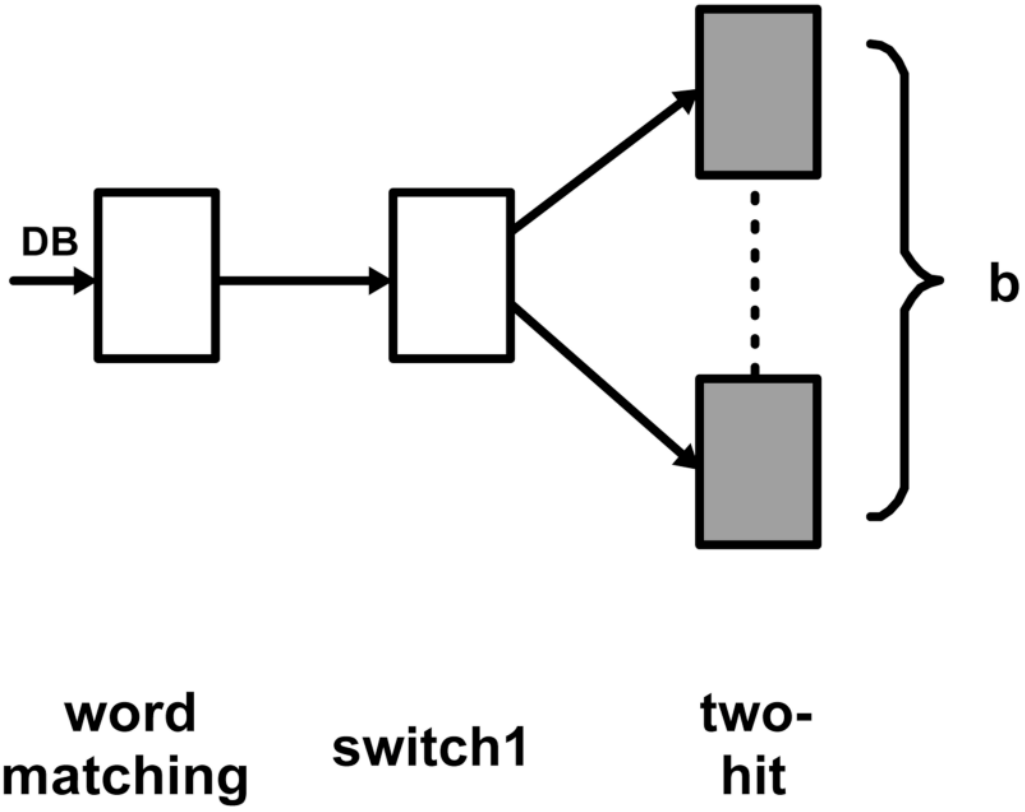


Fig. 10.
Switch 1 – routing 3 hits from a single word matcher to *b* two-hit units.

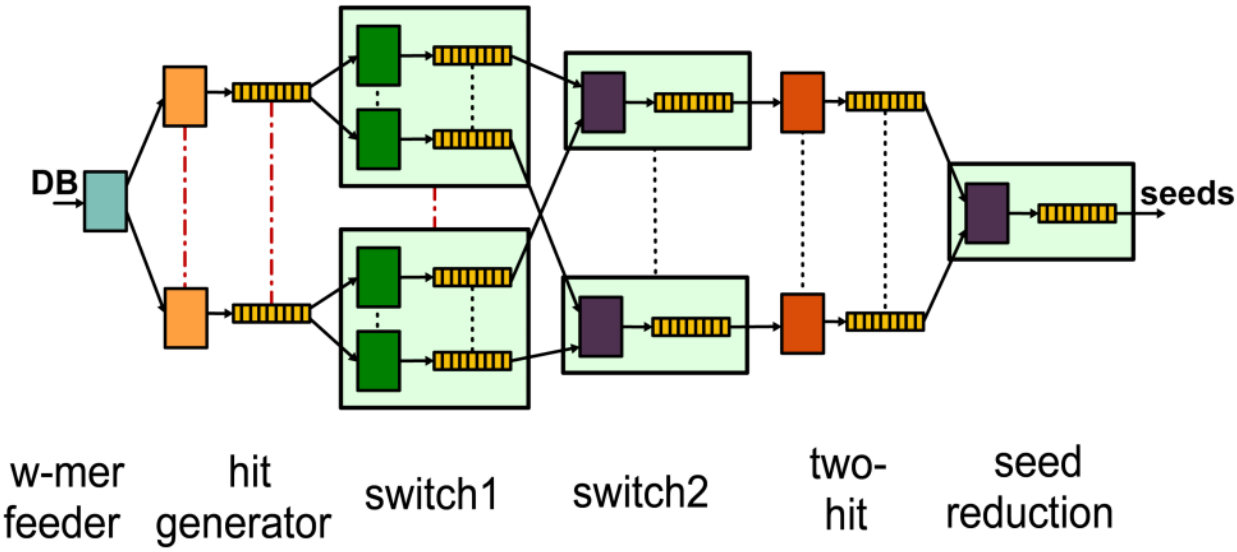


Fig. 11.
Seed generation logic, showing routing.

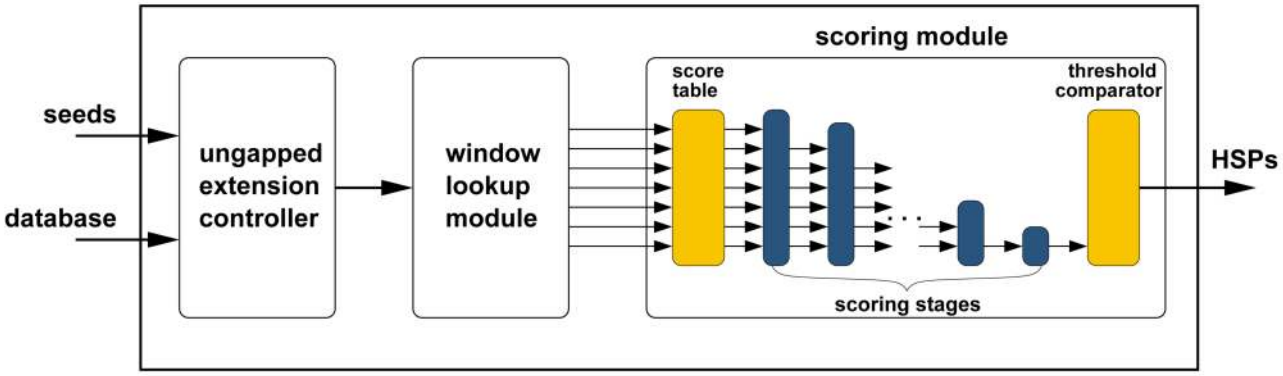


Fig. 12.
Overview of stage 2 architecture.

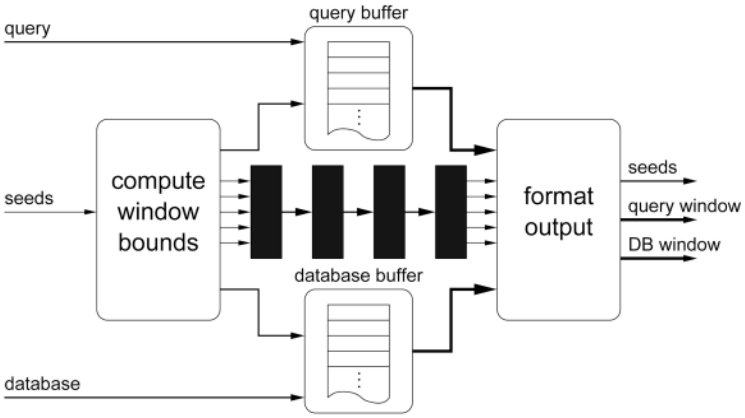


Fig. 13.
Top-level diagram of the window lookup module.

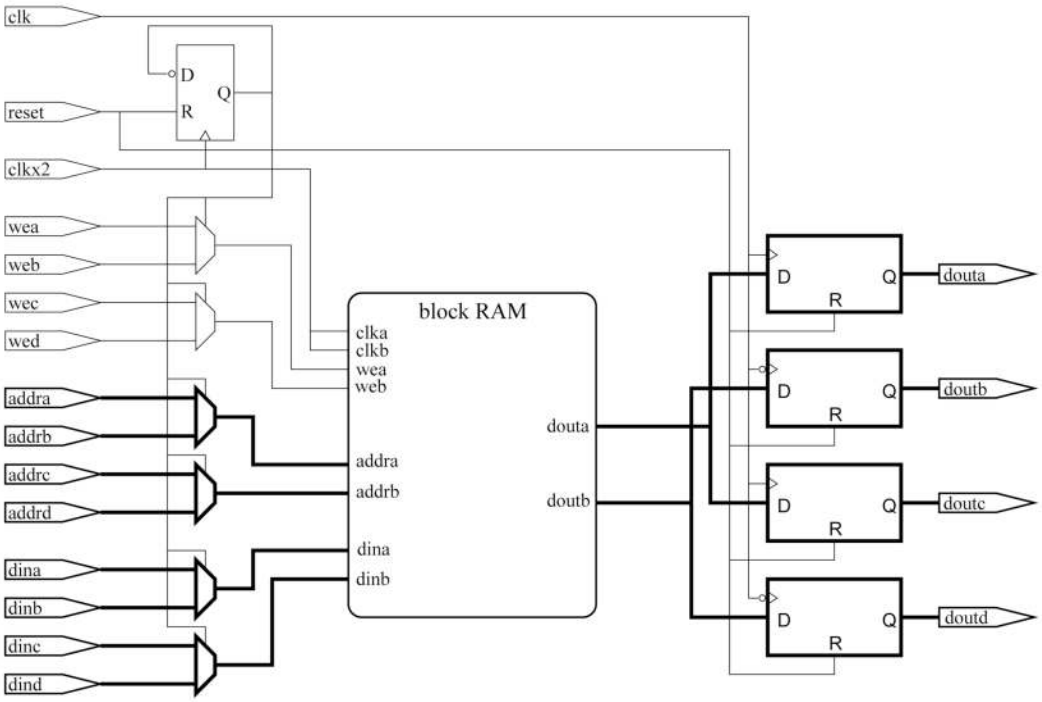


Fig. 14.

Diagram of a time-multiplexed block RAM to provide four independent ports. The wires shown in bold represent multi-wire paths. *clkx2* is a frequency-doubled clock which is phase-aligned to *clk*.

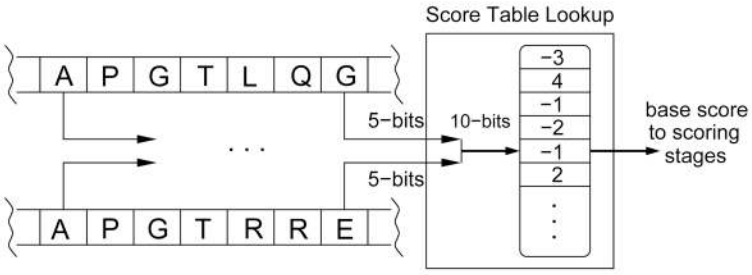


Fig. 15.
Illustration of the first stage of the scoring. The residue pairs are used to index a set of parallel lookup tables that retain the pair score.

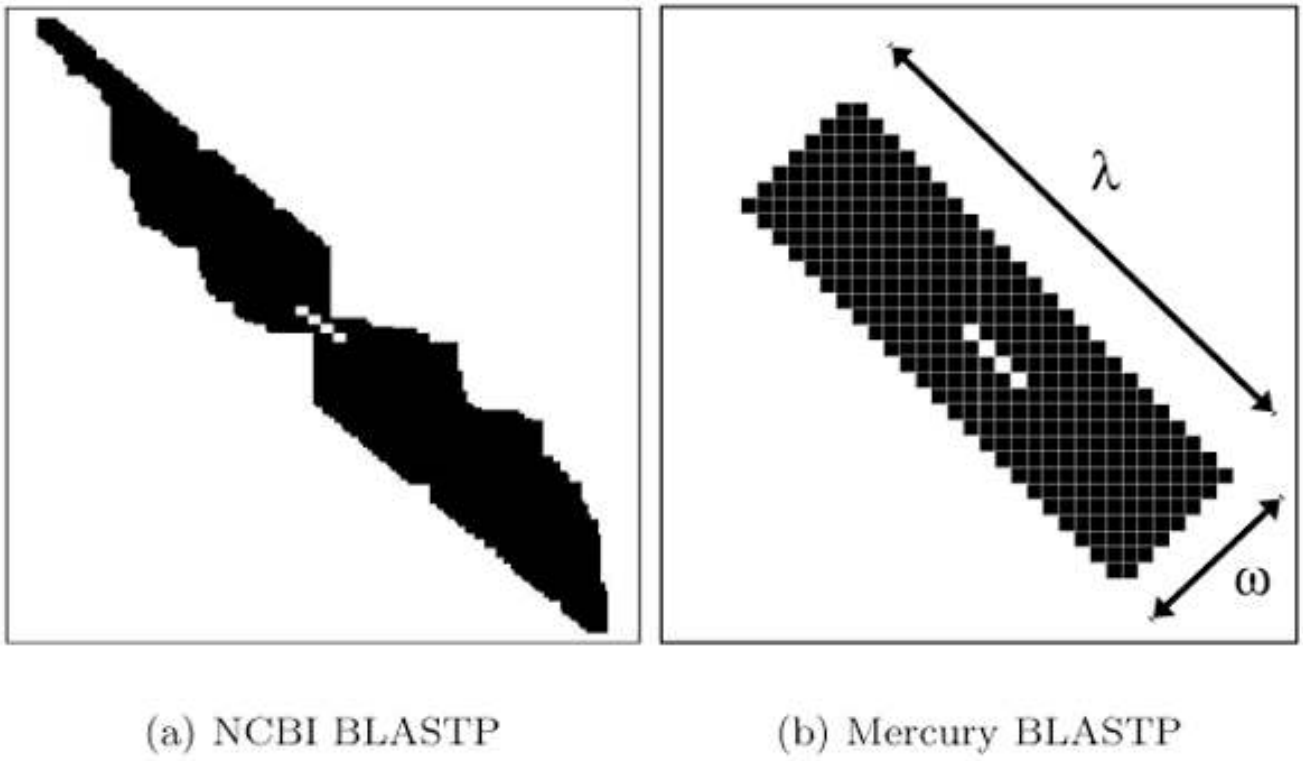


Fig. 16.
Typical structure of gapped extension in (a) NCBI and (b) Mercury BLASTP. X- and Y-axes indicate position within query and subject proteins. Cells computed by each method are shaded, with seed match in white. NCBI BLAST figure is from [Altschul et al. 1997].

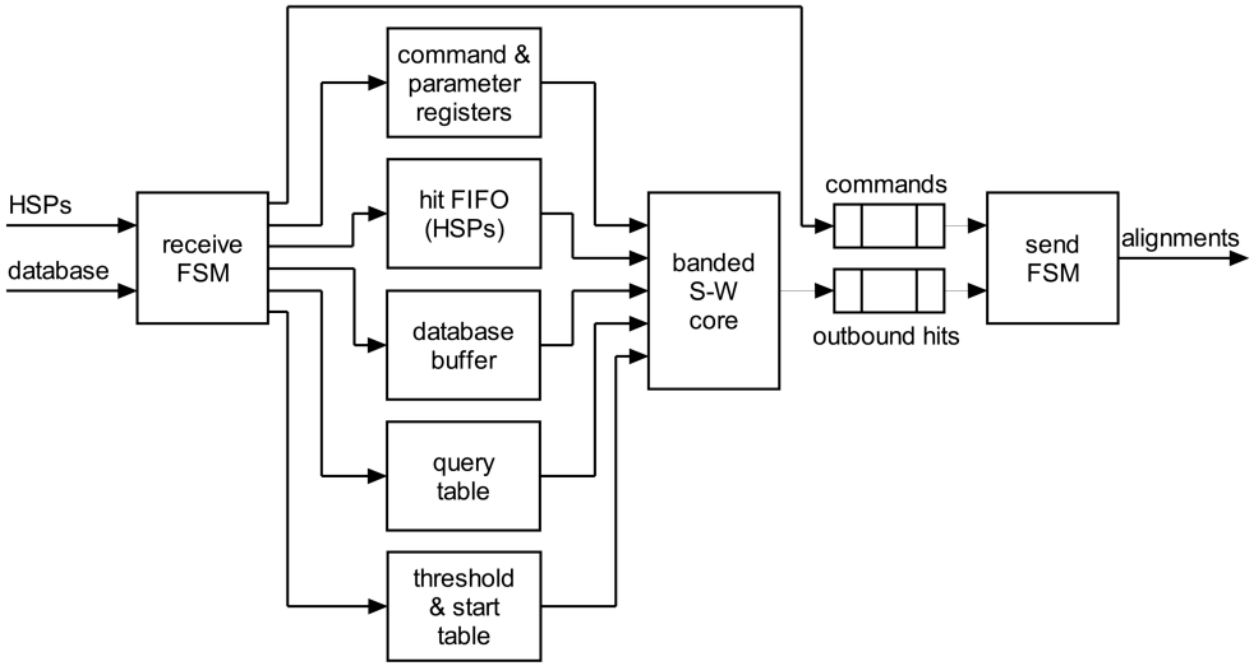


Fig. 17.
Overview of stage 3 design.

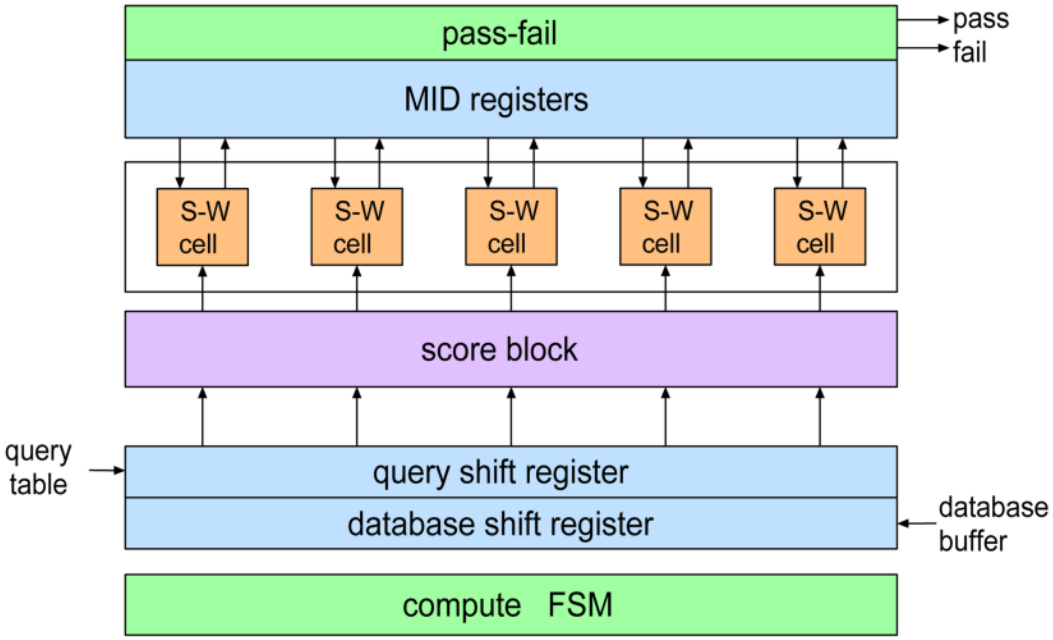


Fig. 18.
Design of banded Smith-Waterman core with $\omega = 5$.

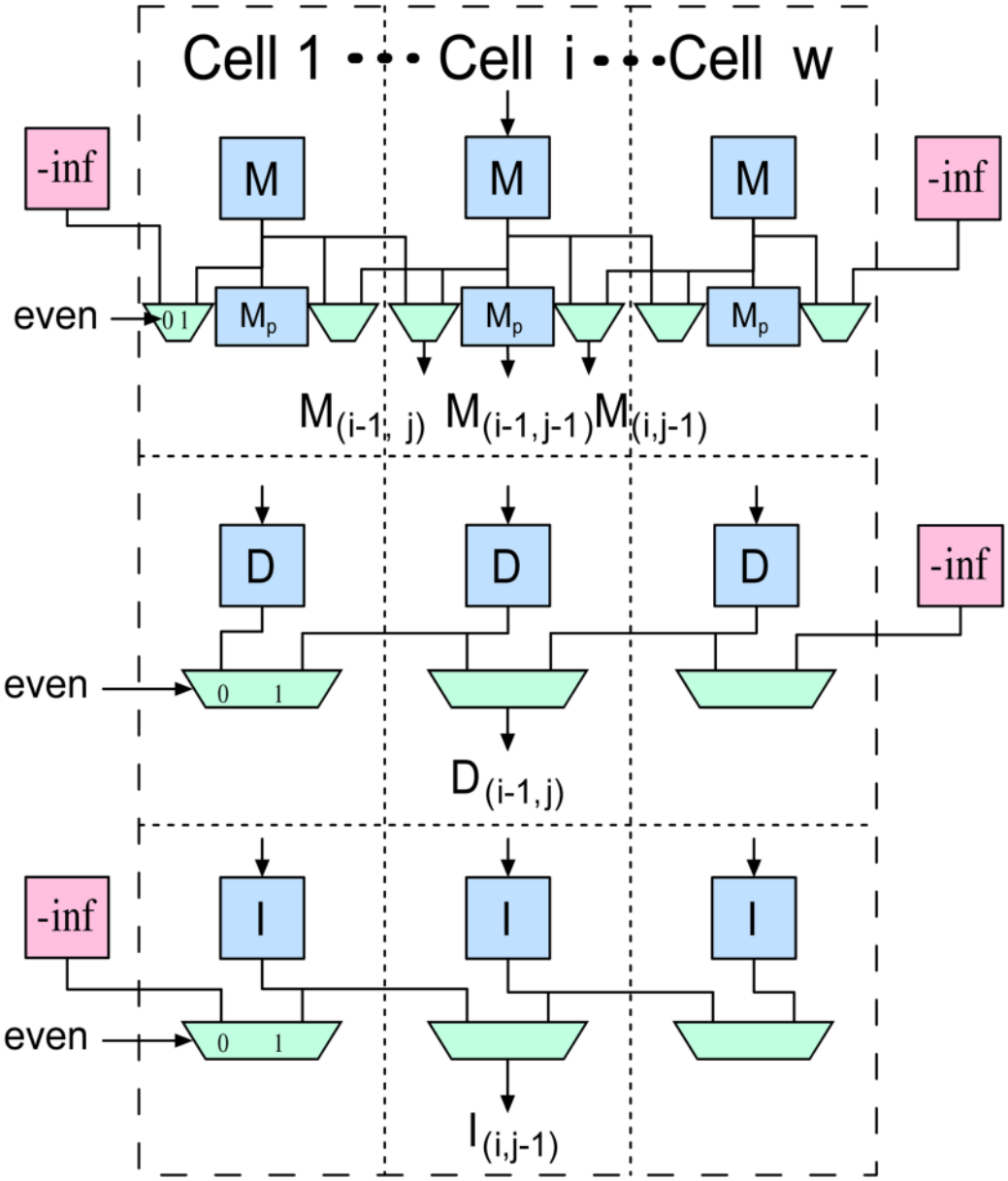


Fig. 19.
Design of MID register block.

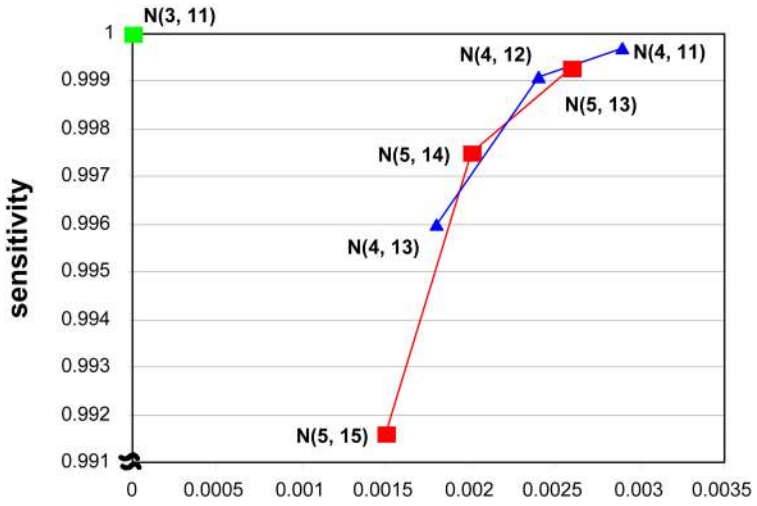
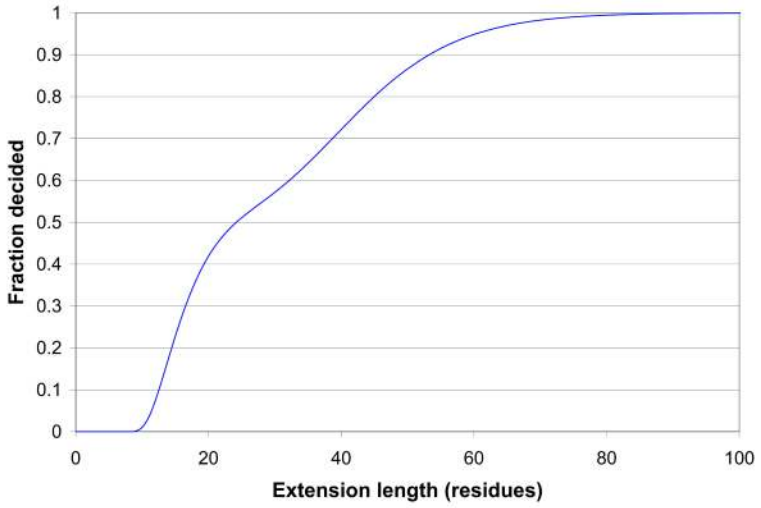
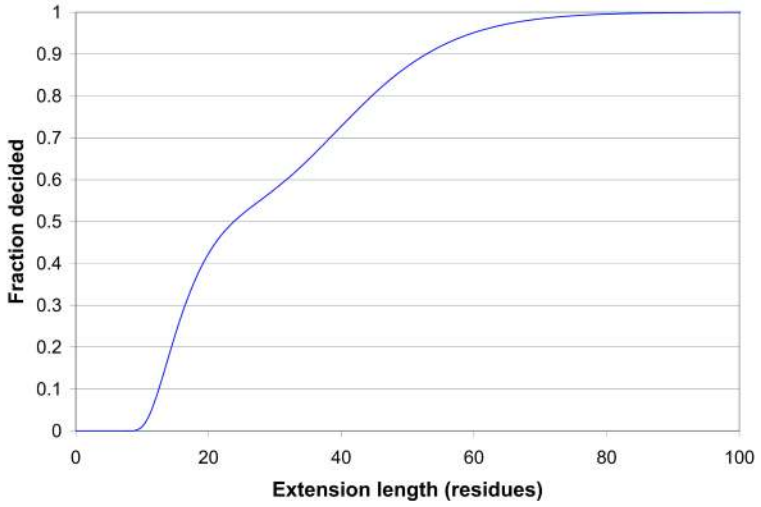


Fig. 20.
Result quality of BLASTP algorithm for various neighborhoods.



(a) CDF of all extensions



(b) CDF of failed extensions

Fig. 21.
CDFs of the length of ungapped extensions measured in NCBI BLASTP. All measurements use the default scoring threshold.

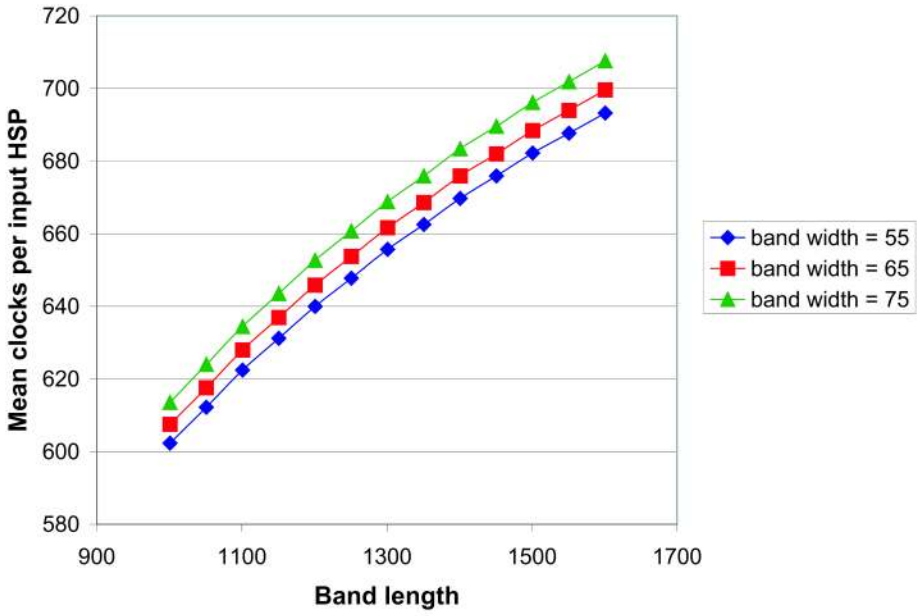


Fig. 22.

Compute time (in clocks) for stage 3 (c_3) as a function of band length λ for several band widths ω.

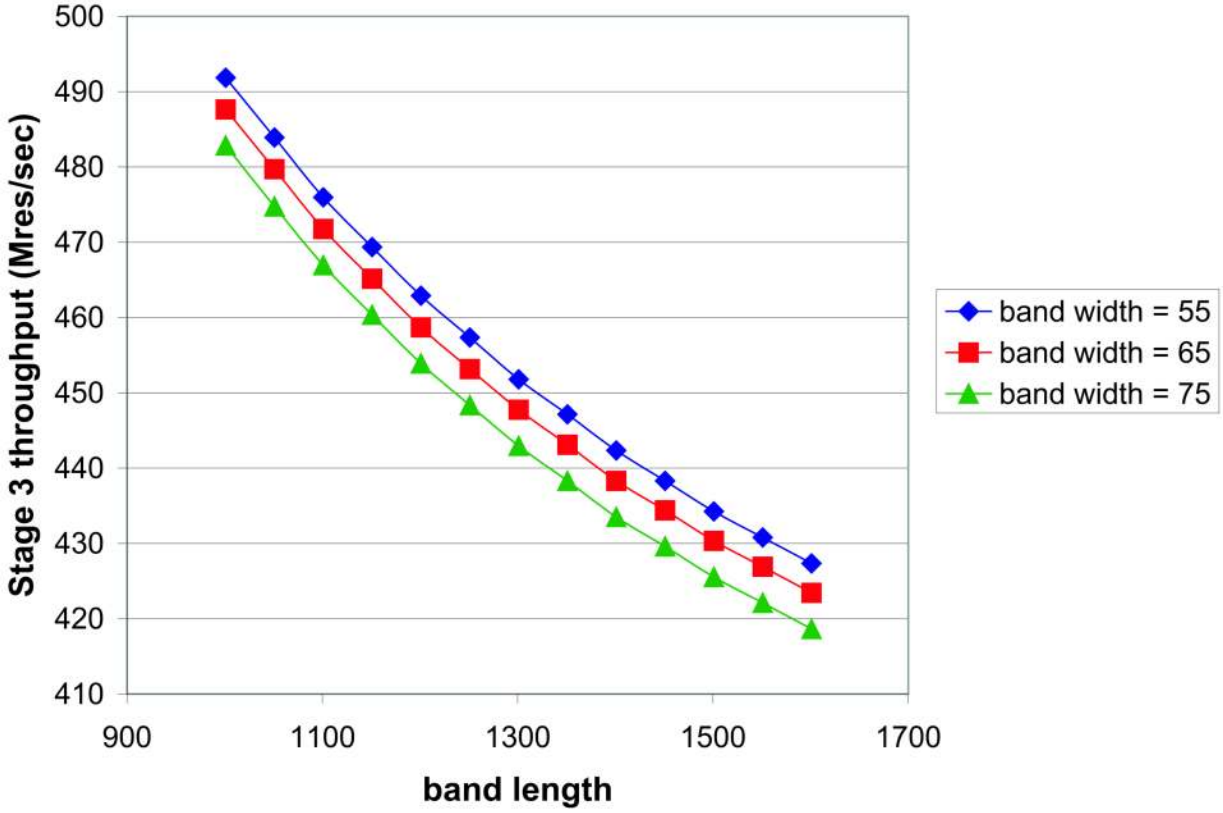


Fig. 23.
Stage 3 throughput as a function of band length λ for several band widths ω.

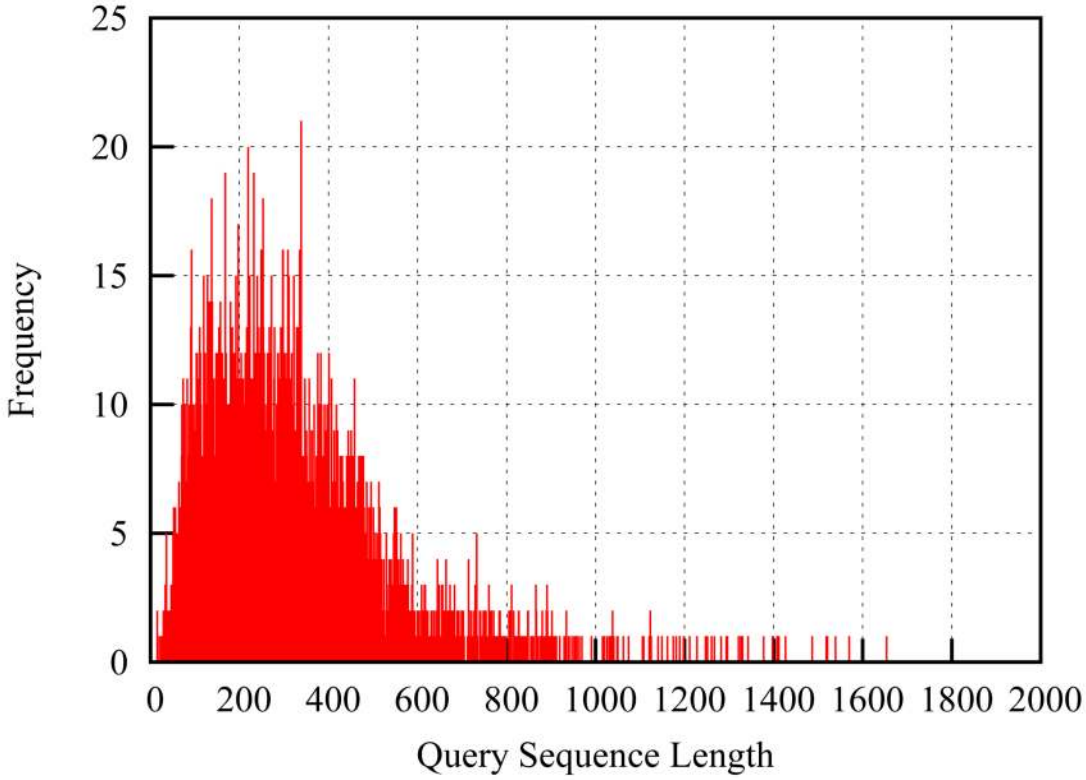


Fig. 24.
Histogram of query sequence lengths in the E.coli proteome

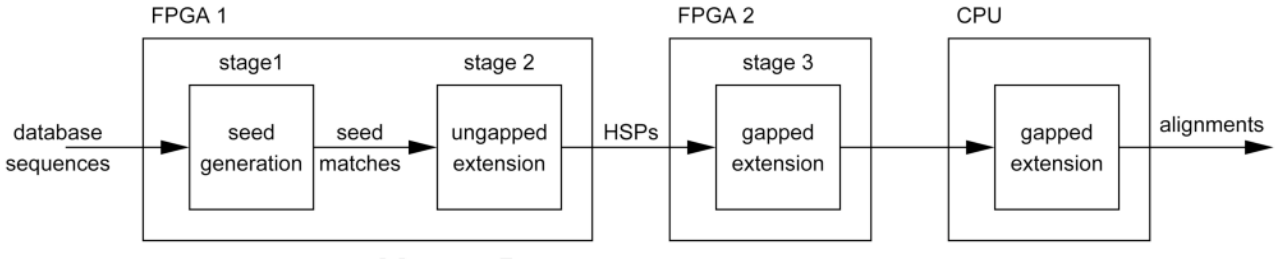


Fig. 25.
Mercury BLASTP hardware/software partition.

Table I

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

Table II

An illustration of a packed query and its threshold and start tables.

Position	Query	Threshold	Start
0	*S*	10	0
1	*W*	10	0
2	*M*	10	0
3	*	-	-
4	*G*	8	4
5	*H*	8	4
6	*M*	8	4
7	*	-	-

Table III

Performance model parameters.

Parameter	Units	Meaning
h		Number of parallel word matching units
b		Number of parallel two-hit units
f_1	MHz	clock frequency for stage 1
f_2	MHz	clock frequency for stage 2
f_3	MHz	clock frequency for stage 3
c_{1a}	clocks	mean time in stage 1a for each input residue
c_{1b}	clocks	mean time in stage 1b for each input *w*-mer
c_2	clocks	mean time in stage 2 for each input seed
c_3	clocks	mean time in stage 3 for each input HSP
r_{1a}	matching *w*-mers/input residue	stage 1a input match rate
r_{1b}	seeds/input *w*-mer	stage 1b filter rate
r_2	HSPs/input seed	stage 2 filter rate
r_3	alignments/input HSP	stage 3 filter rate
$Tput_{1a}$	Mres/sec	stage 1a (word matching) throughput
$Tput_{1b}$	Mres/sec	stage 1b (two-hit) throughput
$Tput_2$	Mres/sec	stage 2 (ungapped alignment) throughput
$Tput_3$	Mres/sec	stage 3 (gapped alignment) throughput
$Tput_{pipe}$	Mres/sec	overall pipeline throughput

Table IV

Effects of parameters on lookup table size.

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

Table V

Comparison of runtimes (in seconds) of various neighborhood generation algorithms

N(w, T)	NCBI-BLAST	Prune-Search	Vector-Prune-Search
N(4, 13)	0.4470	0.0780	0.0235
N(4, 11)	0.9420	0.1700	0.0515
N(5, 13)	25.4815	1.3755	0.4430
N(5, 11)	36.2765	2.6390	0.7835
N(6, 13)	1, 097.2388	16.0855	5.2475

Table VI

Performance of query bin packing approximation algorithms

	Bins	
Algorithm	**Unsorted**	**Sorted**
NF	740	755
FF	667	662

Table VII

Execution time of Mercury BLASTP compared to the baseline system.

Experiment	Baseline Time	Mercury Time	Speedup
1	28.7 h	1.9 h	15.11×
2	40.5 h	2.7 h	15.29×
3	5.7 h	.41 h	13.82×
4	346.4 h	31.15 h	11.12×

Table VIII

Sensitivity of Mercury BLASTP compared to the baseline system.

Experiment	Sensitivity	Alignments Lost	New Alignments
1	99.40%	36,686	10,747
2	99.46%	20,945	10,952
3	99.18%	21,855	5,053
4	98.83%	118,166	14,813