Report Number: WUCSE-2006-48

2006-01-01

# Design and analysis of an accelerated seed generation stage for BLASTP on the Mercury system - Master's Thesis, August 2006

Arpith Jacob

NCBI BLASTP is a popular sequence analysis tool used to study the evolutionary relationship between two protein sequences. Protein databases continue to grow exponentially as entire genomes of organisms are sequenced, making sequence analysis a computationally demanding task. For example, a search of the E. coli. k12 proteome against the GenBank Non-Redundant database takes 36 hours on a standard workstation. In this thesis, we look to address the problem by accelerating protein searching using Field Programmable Gate Arrays. We focus our attention on the BLASTP heuristic, building on work done earlier to accelerate DNA searching on the Mercury platform.... **Read complete abstract on page 2.**

# Design and analysis of an accelerated seed generation stage for BLASTP on the Mercury system - Master's Thesis, August 2006

Arpith Jacob

Complete Abstract:

NCBI BLASTP is a popular sequence analysis tool used to study the evolutionary relationship between two protein sequences. Protein databases continue to grow exponentially as entire genomes of organisms are sequenced, making sequence analysis a computationally demanding task. For example, a search of the E. coli. k12 proteome against the GenBank Non-Redundant database takes 36 hours on a standard workstation. In this thesis, we look to address the problem by accelerating protein searching using Field Programmable Gate Arrays. We focus our attention on the BLASTP heuristic, building on work done earlier to accelerate DNA searching on the Mercury platform. We analyze the performance characteristics of the BLASTP algorithm and explore the design space of the seed generation stage in detail. We propose a hardware/software architecture and evaluate the performance of the individual stage, and its effect on the overall BLASTP pipeline running on the Mercury system. The seed generation stage is 13x faster than the software equivalent, and the integrated BLASTP pipeline is predicted to yield a speedup of 50x over NCBI BLASTP. Mercury BLASTP also shows a 2.5x speed improvement over the only other BLASTP-like accelerator for FPGAs while consuming far fewer logic resources.

2006-48

# Design and analysis of an accelerated seed generation stage for BLASTP on the Mercury system - Master's Thesis, August 2006

Authors: Arpith Jacob

Corresponding Author: arpith@arpith.com

Web Page: http://www.arpith.com/

Abstract: NCBI BLASTP is a popular sequence analysis tool used to study the evolutionary relationship between two protein sequences. Protein databases continue to grow exponentially as entire genomes of organisms are sequenced, making sequence analysis a computationally demanding task. For example, a search of the E. coli. k12 proteome against the GenBank Non-Redundant database takes 36 hours on a standard workstation.

In this thesis, we look to address the problem by accelerating protein searching using Field Programmable Gate Arrays. We focus our attention on the BLASTP heuristic, building on work done earlier to accelerate DNA searching on the Mercury platform. We analyze the performance characteristics of the BLASTP algorithm and explore the design space of the seed generation stage in detail. We propose a hardware/software architecture and evaluate the performance of the individual stage, and its effect on the overall BLASTP pipeline running on the Mercury system.

The seed generation stage is 13x faster than the software equivalent, and the integrated BLASTP pipeline is predicted to yield a speedup of 50x over NCBI BLASTP. Mercury BLASTP also shows a 2.5x speed

Type of Report: Other

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE

---

# DESIGN AND ANALYSIS OF AN ACCELERATED SEED GENERATION STAGE FOR BLASTP ON THE MERCURY SYSTEM

by

Arpith Chacko Jacob

Prepared under the direction of Jeremy Buhler and Roger Chamberlain

---

A thesis presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE

August 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

# DESIGN AND ANALYSIS OF AN ACCELERATED SEED GENERATION STAGE FOR BLASTP ON THE MERCURY SYSTEM

by

Arpith Chacko Jacob

ADVISOR: Jeremy Buhler and Roger Chamberlain

August 2006

Saint Louis, Missouri

NCBI BLASTP is a popular sequence analysis tool used to study the evolutionary relationship between two protein sequences. Protein databases continue to grow exponentially as entire genomes of organisms are sequenced, making sequence analysis a computationally demanding task. For example, a search of the E. coli. k12 proteome against the GenBank Non-Redundant database takes 36 hours on a standard workstation.

In this thesis, we look to address the problem by accelerating protein searching using Field Programmable Gate Arrays. We focus our attention on the BLASTP heuristic, building on work done earlier to accelerate DNA searching on the Mercury platform. We analyze the performance characteristics of the BLASTP algorithm and explore the design space of the seed generation stage in detail. We propose a hardware/software architecture and evaluate the performance of the individual stage, and its effect on the overall BLASTP pipeline running on the Mercury system.

The seed generation stage is 13x faster than the software equivalent, and the integrated BLASTP pipeline is predicted to yield a speedup of 50x over NCBI BLASTP. Mercury BLASTP also shows a 2.5x speed improvement over the only other BLASTP-like accelerator for FPGAs while consuming far fewer logic resources.

To my beloved parents

# Contents

# List of Tables

# List of Figures

# Acknowledgments

I am deeply indebted to my advisors, Dr. Jeremy Buhler and Dr. Roger Chamberlain, for their tremendous patience, support, guidance and insights over the past two years.

I would like to thank the members of my research group, Joseph Lancaster, Brandon Harris, Praveen Krishnamurthy and Richard Crowley for the incisive discussions and their constructive criticisms. We spent many hours in the lab beating the hardware into submission. It was truly a rewarding experience!

Exegy Inc. provided invaluable assistance during the development of the hardware on the Mercury platform. I especially acknowledge the support of Mr. Berkley Shands for help with the FPGA communication wrappers.

I am grateful for the support from our sponsors, the National Institutes of Health and the National Science Foundation. This research is supported by the NIH/NGHRI grant 1 R42 HG003225-01 and the NSF career grant DBI-0237902.

Finally, I would like to thank my parents and my brother for their support and love throughout these years.

Arpith Chacko Jacob

*Washington University in Saint Louis*
*August 2006*

# Chapter 1

# Introduction

Sequence analysis is a commonly used tool in computational biology to help study the evolutionary relationship between two sequences, by attempting to detect patterns of conservation and divergence. It measures the similarity of two sequences by doing inexact matching, using biologically meaningful mutation probabilities. A high-scoring alignment of the two sequences matches as many identical residues as possible while keeping differences to a minimum, thus recreating a hypothesized chain of mutational events that separates them.

Biologists use high-scoring alignments as evidence in deducing homology, i.e., that the two sequences share a common ancestor. Homology between sequences implies a possible similarity in function or structure, and information known for one sequence can be applied to the other. Sequence analysis helps to understand an unidentified sequence using existing information. Considerable effort has been spent in collecting and organizing information on existing sequences. An unknown DNA or protein sequence, termed the query, can be compared to a database of annotated sequences such as GenBank [4] or Swiss-Prot [10] to detect homologs.

Sequence databases continue to grow exponentially as entire genomes of organisms are sequenced, making sequence analysis a computationally demanding task. For example, since its release in 1982, the GenBank DNA database has doubled in size approximately every 18 months [5]. The International Nucleotide Sequence Databases comprising DNA and RNA sequences from GenBank, European Molecular Biology Laboratory's European Bioinformatics Institute (EMBL-Bank), and the DNA Data Bank of Japan recently announced a significant milestone in archiving 100 gigabases of sequence data [7]. The Swiss-Prot protein database has experienced a corresponding growth (Figure 1.1 [6]) as newly sequenced genomic DNA are translated into proteins. Existing sequence analysis tools are fast becoming outdated in the post-genomic era.

In this thesis, we look to address this problem by accelerating protein database searches using programmable logic on Field Programmable Gate Arrays (FPGAs). We focus our attention on the popular BLASTP heuristic, building on work done earlier to accelerate DNA searching [38] on the Mercury platform. We analyze the performance characteristics of the major stages of the BLASTP algorithm and propose a hardware design for the initial seed generation stage. We evaluate the

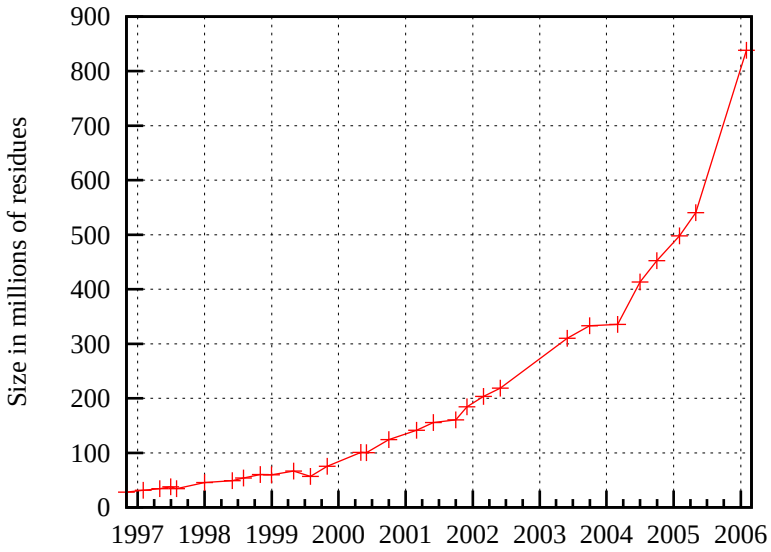



Figure 1.1: Growth of UniProtKB/TrEMBL protein databases

performance of the individual stage and its effect on the overall pipeline running on the Mercury system. We achieved a speedup of 13x for the seed generation stage over the software equivalent. The Mercury BLASTP pipeline is expected to run over 50x faster than a typical protein search on a standard workstation.

In the following sections we introduce sequence analysis, describe the BLASTP algorithm in detail and analyze its performance characteristics. Finally, we survey the state of the art in high-throughput protein sequence analysis.

## 1.1 Sequence analysis

Pairwise comparison aims to find biologically similar regions between two sequences. It performs a side-by-side comparison of residues to generate an alignment reflecting their similarity (Figure 1.2). Comparison assigns a similarity score from a scoring matrix to each residue pair. Nucleotide comparison assigns a positive score for a perfect residue match and a negative score for a mismatch. Protein comparison uses a table of biologically meaningful log-odds scores to score a pair of residues. In addition, a gap in an alignment corresponding to the absence of a residue is scored by an affine penalty $a + nb$, where $a$ is the gap initiation and $b$ the gap extension cost for each of the $n$ gap positions.

An optimal alignment is a highest scoring alignment of two sequences. Dynamic programming is an efficient method to find such an alignment avoiding combinatorial explosion. The Needleman-Wunsch [47] global alignment algorithm matches two sequences in their entirety. Smith-Waterman [52, 30] locally aligns two sequences by looking for conserved subsequences. Since the entire search space is inspected by comparing every residue of the two sequences against each other, dynamic programming is guaranteed to find the optimal alignment.

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

Figure 1.2: Alignment of two protein sequences

Dynamic programming algorithms run at a cost dependent on the product of the lengths of the two sequences. Even for small sequences, this becomes prohibitively expensive to run on general-purpose computers. In addition to the large time complexity, Smith-Waterman suffers from a number of other disadvantages. The simple traceback procedure used to retrieve an optimal alignment requires space proportional to the product of the sequence lengths. The space complexity is of greater concern than the time complexity on memory-limited workstations. Linear space traceback requires a more sophisticated divide-and-conquer algorithm [46].

As a result of the high cost of Smith-Waterman, a number of fast heuristic algorithms like BLAST [15] and FASTA [48] have been developed. Heuristics are employed to rapidly identify "hotspots," i.e. locations where a good alignment is likely to be present. The more expensive dynamic programming technique is only performed in this greatly reduced search space. In addition to having lower space and time complexity, BLAST is able to detect sub-optimal alignments. However, sensitivity is reduced, and there is no guarantee that the optimal alignment will be found.

### 1.1.1 NCBI BLAST

The ***B**asic **L**ocal **A**lignment **S**earch* **T**ool, or BLAST [13, 15], is the most popular sequence analysis package. The BLAST heuristic relies on identifying high-scoring segments between a pair of sequences using efficient data structures and algorithms. These segments are then used as seeds for a full dynamic programming alignment procedure. This heuristic results in a huge speed advantage over Smith-Waterman, at the cost of reduced sensitivity. The original BLAST suite of programs [14] also introduced the concept of statistical "significance" of an alignment, i.e., how likely it is that two unrelated sequences are similar simply by chance [35]. The BLAST package provides a variety of programs with specific functions that are summarized in Table 1.1. In addition, enhancements to identify distant homologs (PSI-BLAST) and to rapidly compare highly similar nucleotide sequences (MegaBLAST) are included in the standard distribution.

Table 1.1: BLAST programs

| Program | Function |
|---|---|
| *BLASTN* | Nucleotide query vs. nucleotide database |
| *BLASTP* | Protein query vs. protein database |
| *BLASTX* | Nucleotide query translated into its six reading frames vs. protein database |
| *TBLASTN* | Protein query vs. nucleotide database translated into its six reading frames |
| *TBLASTX* | Nucleotide query vs. nucleotide database, both translated into their six reading frames |

Two BLAST implementations, NCBI BLAST [15, 13] maintained by the National Center for Biotechnology Information and WU-BLAST [12] by Warren Gish at Washington University in St. Louis, are available. We concentrate our efforts on NCBI BLAST (Oct 20, 2004 release 2.2.10) as its source code is available in the public domain.

## 1.2 The BLAST algorithm

The BLAST algorithm is based on two heuristics published in 1990 [14] (BLAST1) and 1997 [15] (BLAST2). We focus on the BLAST2 algorithm, noting important distinctions from the original when relevant. Though we describe the algorithm for protein-to-protein comparisons, it can be easily extended to DNA searches with minor modifications.

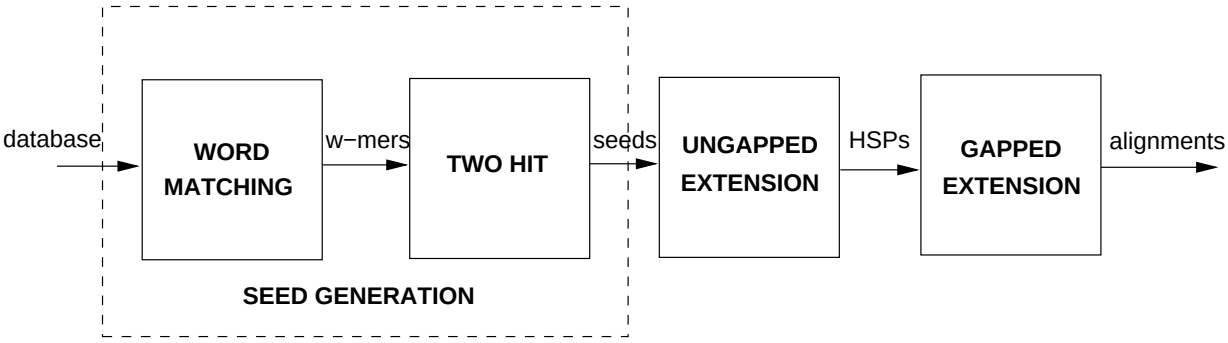



Figure 1.3: Pipelined stages in BLASTP

The BLAST algorithm is divided into three stages (Figure 1.3): Seed Generation, Ungapped Extension, and Gapped Extension. The key observation in the BLAST technique is the high likelihood of the presence of short aligned words in an alignment. In the Seed Generation stage word matches, or hits, are identified between the query and the database sequence. These matches are passed as seeds into the ungapped extension phase to identify High-scoring Segment Pairs (HSPs). An HSP is a pair of continuous subsequences of residues (identical or not, but without gaps at this stage) of equal length, at some location in the query and the database sequence. Statistically significant HSPs are then passed into the gapped extension phase, where a Smith-Waterman-like dynamic programming

algorithm is performed. An HSP that successfully passes through all three stages is reported to the user.

The seed generation stage of the pipeline is the focus of this thesis and is described in detail in the next section. The remaining stages involve work done by other members of our research group and are only briefly described. A more detailed exposition is available in [15, 13].

### 1.2.1 Stage 1: Seed Generation

Let $S$ be a string or a sequence defined over an alphabet $\Sigma$. A *w-mer* or a word $r$ is defined as a string of exactly $w$ characters. A string $S$ can be split into exactly $|S|-w+1$ overlapping w-mers. If $r \in S$, $pos(r)$ refers to the index location of the first character of $r$ in $S$. Seed generation consists of the word matching and two-hit stages placed back-to-back. The input to seed generation is a query sequence $Q$ and a database $D$; the output, a list of w-mer pairs $(q, d) \in Q \times D$, termed *seeds* or *hits*.

The first stage aims to find w-mer matches between the query and the database. W-mer matches indicate regions of interest between the two sequences. Reading in a query $Q$ and a database $D$, the output of this stage is a list of all word pairs $(q, d) \in Q \times D$ such that $\Sigma_{i=1}^{w} \delta_{q_i,d_i} >= T$, i.e. a pairwise comparison of characters of the two words yields a score greater than or equal to the threshold. $\delta$ is a scoring table/function defined for all pairs of characters in $X$. In order to compute the list of word pairs efficiently, a neighbourhood $N(w, T)$ of the query sequence is defined for a fixed word length $w$ and a threshold value $T$. The neighbourhood is defined for every query w-mer as the list of all possible database w-mers whose pairwise comparison score is greater than or equal to $T$. Linear scanning of overlapping words in the database sequence, using a lookup table constructed from the neighbourhood of the query helps in quick identification of hits.

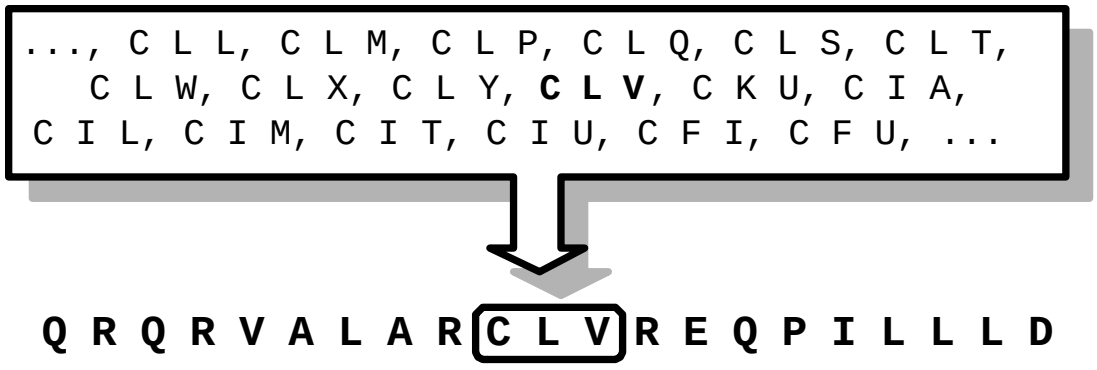


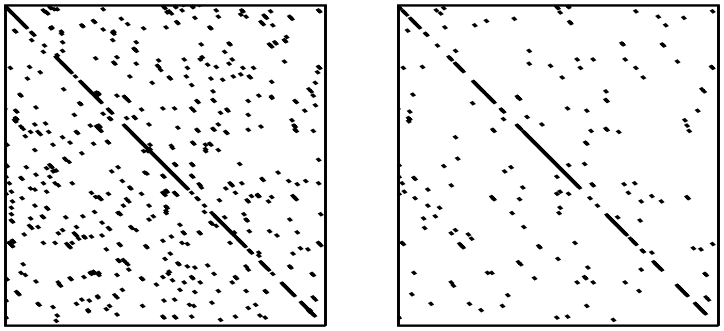

Figure 1.4: Neighbourhood of a single query w-mer in a protein sequence

A two-hit stage is frequently employed to filter an input hit stream generated by the word matching stage. A two-hit is defined as a pair of hits $(q, d)$ and $(q', d')$ such that $pos(d) - pos(q) = pos(d') - pos(q')$, $pos(d) - pos(d') < A$ and $pos(d) - pos(d') \geq w$. $A$ is a scalar, termed the window length.

The output of this stage termed seeds, is a list of all hits $(q, d)$ such that there exists a two-hit whose second hit is $(q', d')$.

### Neighbourhood Generation

Due to the high degree of conservation in DNA sequences, BLASTN word matches are simply pairs of exact matches in both sequences (with the default word length being 11). Building the neighbourhood involves identifying all $|S| - w + 1$ overlapping w-mers in the query sequence $S$. Amino acids in protein sequences readily mutate into other, functionally similar amino acids. Hence, BLASTP looks for shorter (typically of length 3) non-identical pairs of substrings that have a high similarity score. The neighbourhood $N(w, T)$ is generated by identifying all possible amino acid subsequences of size $w$ that match each overlapping w-mer in the query sequence. All such subsequences that score at least $T$ (called the neighbourhood threshold) when compared to the query w-mer are added to the neighbourhood. BLAST compares each query w-mer against an enumerated list of all $|\Sigma|^w$ possible words to determine the neighbourhood.



(a) Word length = 3, Threshold = 9 (b) Word length = 3, Threshold = 11

Figure 1.5: Speed vs. sensitivity tradeoff: increasing the neighbourhood threshold decreases the number of hits in the seed generation stage. The database sequence is plotted on the X-axis and the query sequence on the Y-axis.

There is a tradeoff between speed and sensitivity when selecting the neighbourhood parameters. Increasing the word length or the neighbourhood threshold decreases the neighbourhood size and therefore reduces the computational costs of seed generation, since fewer hits are generated. However, this comes at the cost of decreased sensitivity. Fewer word matches are generated from the smaller neighbourhood, reducing the probability of a hit in a biologically relevant alignment.

**Word Matching**

The neighbourhood of a query is stored in a direct lookup table indexed by w-mers. A linear scan of the database performs a lookup on each overlapping w-mer $d$ at database offset $pos(d)$. The table lookup yields a linked list of query offsets $pos(q_1), pos(q_2), \ldots, pos(q_n)$ which correspond to hits $(q_1, d), (q_2, d), \ldots, (q_n, d)$. Hits generated from a table lookup may be further processed to generate seeds for the ungapped extension stage.

**One-hit:** BLASTN and the initial version of BLASTP consider each hit in isolation. In the one-hit approach, a single hit is considered sufficient evidence of the presence an HSP and is used to trigger a seed. A neighbourhood $N(4, 17)$ yields sufficient hits to detect similarity between typical protein sequences. A large number of these seeds, however, are spurious and must be filtered by expensive seed extension.

**Two-hit:** The two-hit refinement is based on the observation that HSPs of biological interest are typically much longer than a word. Hence, there is a high likelihood of generating multiple hits in a single HSP. In the two-hit method, hits generated by the word matching stage are not passed directly to ungapped extension, instead being recorded in a diagonal array. The presence of two hits in close proximity on the same diagonal (noting that there is a unique diagonal associated with any HSP that does not include gaps) is the necessary condition to trigger a seed. The decreased seed generation rate drastically reduces time spent in later stages.

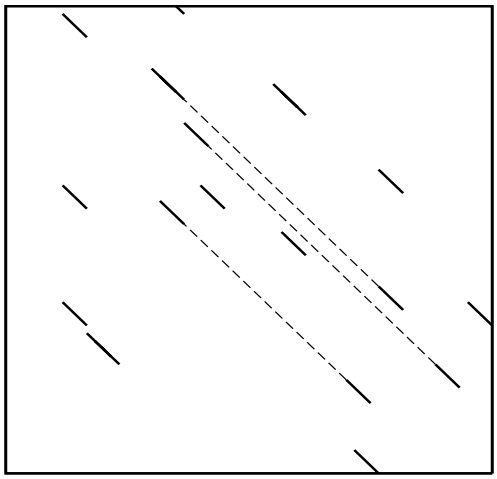


Figure 1.6: The two-hit method requires two adjacent word matches on the same diagonal – indicated by the dashed line – to generate a seed. The database sequence is plotted on the X-axis and the query sequence on the Y-axis.

In order to attain comparable sensitivity to the one-hit algorithm, a more permissive neighbourhood of $N(3, 11)$ is used. Although this increases the number of hits generated by the word matching stage, only a fraction pass as seeds for ungapped extension. The BLAST2 literature [15] reports an approximately 3.2x increase of hits in the word matching stage, but an 86% reduction in the number of seeds generated by the two-hit configuration. Since far less time is spent filtering hits than extending them, there is a significant savings in the computational cost.

### 1.2.2 Stage 2: Ungapped Extension

The input to this stage is a list of seeds; the output, all HSPs that score above a threshold after extension on a single diagonal, using the X-drop algorithm. Extension along a diagonal centered on a seed is done using an X-drop procedure [39]. Match/mismatch scores are computed using a scoring matrix. Seeds are extended in either direction so long as the score continues to increase. If an extension causes the score of an HSP to fall more than a threshold $X_u$ below the best score seen so far, the process is terminated. Statistically significant HSPs are passed along the pipeline for further inspection.

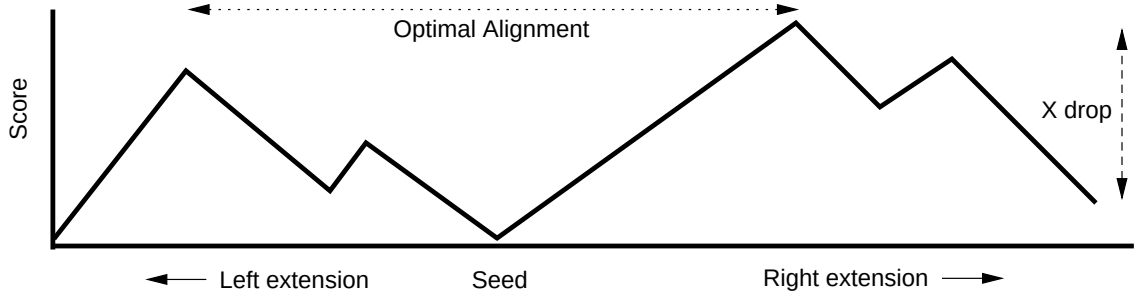



Figure 1.7: Ungapped Extension in BLAST

In order to avoid the generation of redundant seeds, the length of an HSP extension is recorded in the diagonal array used by the two-hit algorithm. Feedback from this stage permits recognition of hits that are part of the same HSP, which can then be safely ignored.

### 1.2.3 Stage 3: Gapped Extension

The input to this stage is a list of HSPs; the output, a list of high scoring alignments. HSPs passed into this phase are extended using rigorous dynamic programming permitting gaps. While full Smith-Waterman performs a fixed amount of computation proportional to the product of the sequence lengths, the BLAST gapped extension stage is adaptive. Centered on the seed, gapped extension proceeds along either direction exploring fewer cells, which are selected so as to enclose

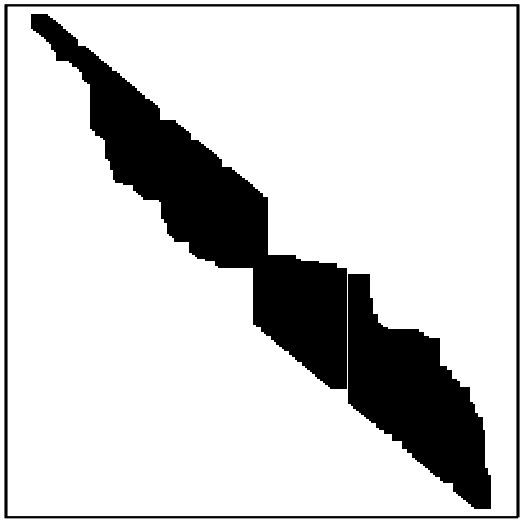


Figure 1.8: Gapped Extension in BLAST. The database sequence is plotted on the X-axis and the query sequence on the Y-axis.

an optimal alignment. By reducing the number of cells computed, a fraction of the work of full Smith-Waterman is done. An X-drop procedure similar to the one described in the previous stage is employed.

Since BLAST looks at multiple hits between the query and the database, more than one high scoring alignment can be retrieved. An E-value, or an Expectation Value, measures the statistical significance of an alignment. It denotes the number of chance occurrences in the database when using a specific scoring system. The lower the E-value of an alignment, the more likely it is to be biologically significant.

## 1.3 Scale of the problem

While the fundamental BLAST algorithm has undergone little change since the late 1990's, advancements in general-purpose microprocessor technology have provided necessary speed enhancements. However, the exponential growth of sequence data has exposed serious limitations to this strategy. For example, the BLAST server on the NCBI website makes use of a Linux cluster consisting of around 200 CPUs [44]. NCBI reported processing 140,000 queries on a typical weekday in 2004 and planned to double their computing capabilities to keep up with demand [44].

To analyze the scale of the problem, we measured the performance of BLASTP, running it with default parameters on a benchmark machine. We compared the GenBank Non-Redundant (NR)

database against query sequences of various lengths, selected randomly from the Escherichia coli k12 proteome. Details of the benchmark machine and the dataset are presented in chapter 2. Timings are averaged over twenty queries for each length and reported at 95% confidence intervals.

Table 1.2: NCBI BLASTP runtimes for various query sizes

| **Query Length** (residues) | **BLASTP Runtime** |
|---|---|
| 500 | $1m30s \pm 3.42\%$ |
| 1000 | $2m17s \pm 2.44\%$ |
| 2000 | $3m48s \pm 2.46\%$ |
| 5000 | $8m43s \pm 4.52\%$ |
| $Ecoli - k12$ | $36h14m20s$ |

Table 1.2 lists the execution time of NCBI BLASTP for query sequences of various sizes. An approximately linear increase in running time is observed with an increase in the query length. NCBI BLASTP requires over 36 hours to compare the entire E. coli. K12 proteome against NR.

Recognizing the importance of an accelerated BLASTP implementation, we look to use programmable logic on Field Programmable Gate Arrays (FPGAs) for a solution. The Mercury system [23] provides the infrastructure to support high-throughput disk-based computation on reconfigurable hardware attached to general-purpose workstations. Data from disks is streamed directly through pipelined logic blocks, typically being filtered by progressively more complex computations before being sent for post-processing on the attached workstation. Hardware/software codesign is necessary to ensure efficient implementation of an application. Previous work [38] on the Mercury system targets a two-order-of-magnitude acceleration of the NCBI BLASTN algorithm.

In this work, we describe the redesign of the Seed Generation stage as part of Mercury BLASTP. The acceleration of the ungapped and gapped extension stages are detailed in [39, 31].

## 1.4 Related work

In this section, we survey research targeting acceleration of homology search. We first list algorithmic advances in the field that improve the BLAST heuristic to increase speed and sensitivity. We then look at parallel implementations on special-purpose hardware of the Smith-Waterman and BLAST algorithms.

### 1.4.1 Heuristic improvements

**FASTA:** [48] Introduced prior to BLAST, FASTA uses a similar seeding strategy to identify locations of high similarity. Identical pairs of subsequences, called $k$-tuples (k = 1 or 2 for protein

searches), are identified between the query and database sequence. Diagonals with the highest density of matches are identified by scoring its k-tuples and penalizing intermediate mismatches. This step corresponds to the neighbourhood strategy in BLAST. The highest-scoring diagonals, which are likely to form part of a significant alignment, are combined using Smith-Waterman dynamic programming. This is similar to gapped extension in BLAST, except that dynamic programming is restricted to a narrow band centered on the highest scoring diagonal. BLAST builds on techniques introduced in FASTA and is of comparable sensitivity, with fewer false hits and an order of magnitude speedup [14].

**BLAT:** [36] The **B**LAST-like **A**lignment **T**ool is a DNA and protein sequence analysis package written by Jim Kent at UCSC. It was designed to quickly find alignments in sequences of high similarity. A table of all non-overlapping w-mers (w = 4 or 5 for proteins) in the *database* is pre-computed and loaded into memory. Scanning involves a lookup of every overlapping w-mer in the query. Since the database rather than the query is indexed, the scanning phase is extremely quick. To reduce the number of hits generated, a longer word length is used, and only non-overlapping exact matches in the database are identified.

BLAT "clumps" hits into buckets based on their database locations and then sorts them by their diagonals. Hits within a short window on a diagonal and in close proximity in the database sequence are located. Homologous regions in the database can then be identified from hits in high scoring clumps. This stage is similar to the two-hit stage in BLAST; the more elaborate technique is necessary to handle the large number of hits generated on a table lookup.

BLAT is about 50x faster than existing protein alignment tools for sequences that are at least 80% identical. In addition to perfectly matching w-mers, BLAT allows near-perfect matches, where up to one pair of residues may mismatch. The increase in sensitivity comes at a cost of decreased speed. BLAST, with its sophisticated neighbourhood seeding and gapped extension, is more sensitive and therefore suitable for more divergent sequences. This is important since homologous sequences have a percent identity as little as 20%.

The memory requirements of BLAT are considerably increased since both the database index and sequences must be loaded into main memory. The lookup table of the translated human genome alone is 2.5 gigabytes in size.

**PatternHunter:** [42, 41] The major contribution of PatternHunter is the novel seeding strategy used to improve sensitivity and selectivity of the seed generation stage. While BLAST forms a word from $k$ contiguous residues, PatternHunter inspects only $w < k$ residues from a window of size $k$. The positions of the $w$ residues in *spaced* seeds is specified by a model. For example, 1110111 inspects $k = 7$ residues but creates words of size $w = 6$ (termed the weight of the model); here the fourth residue is ignored.

The spaced seeding strategy significantly improves sensitivity because the independence of match events of neighbouring w-mers in a sequence is increased. When a word and its shifted copy in a sequence share many residues, a mismatch in the first word directly affects the number of mismatches in the second. With reduced independence of consecutive words, their match probabilities decrease, diminishing the sensitivity. This effect is reduced by spaced seeds since fewer residue positions are shared between consecutive w-mers. By increasing the weight of a word and using a spaced seed model, fewer false positives are generated in the seed generation stage, without a loss in sensitivity. A further sensitivity improvement in PatternHunter considered multiple spaced seeds, where a hit may be generated by any one of multiple seed models.

*Vector* seeds generalize this concept for protein searches. A vector seed is specified by a pair $(v, T)$, where $v$ is the model and $T$, the threshold. The sum of the score of aligned pairs of residues at positions specified by a 1 in the model must exceed $T$ to be considered a match. tPatternHunter [37] uses multiple vector seeds for protein sequences [55].

There are two main disadvantages to these approaches. Firstly, increasing the weight of a seed model increases the size of the hash table. This is especially true for protein sequences where a neighbourhood is generated. Secondly, finding optimal seed models is NP-hard. Further, optimal seed models are specific to a particular dataset. Use of multiple seeds require multiple hash tables, exacerbating the problem and making it impractical for memory-constrained machines.

**MUMmer:** [26] MUMmer replaces the hashing strategy of the word matching phase by a more efficient data structure, the suffix tree. The suffix tree of a sequence is a compact representation of all suffixes in an input sequence. It is possible to locate every suffix by traversing a unique path from the root to a leaf node. MUMmer creates a suffix tree for the first genome, and then adds suffixes from the second. Maximal unique matches (MUMs) are quickly identified using this data structure and inspected further. A clever implementation of the suffix construction algorithm can be done in linear time.

The time requirements of MUMmer are better than the BLAST hashing scheme. The suffix tree approach finds application in searching large genomes that are biologically close to each other. However, it fails to find alignments without exact matches, unlike BLAST which uses the neighbourhood strategy.

### 1.4.2 Smith-Waterman accelerators

The Smith-Waterman algorithm has been studied extensively for parallelization in hardware. Fine-grained parallelism exploits the data dependency structure in the Smith-Waterman recurrence to speed up pairwise comparison on tightly-coupled processing elements. To compute the value of a cell in the dynamic programming matrix, cell values immediately to the left, above, and upper-left must be known. This allows the simultaneous computation of cells on an anti-diagonal. Coarse-grained

implementations perform many sequence alignments in parallel on loosely-coupled processors. Each computational unit aligns the query against an independent subset of the database.

Smith-Waterman accelerators can be classified into four categories: Workstation clusters, Supercomputers, VLSI devices and Reconfigurable Hardware.

*General-purpose workstations* are an inexpensive and widely available resource. **S**ingle **I**nstruction stream, **M**ultiple **D**ata stream (SIMD) architectures are available on most modern workstations to handle the demands of multimedia processing. SIMD instructions operate on multiple data words (between 2 and 16) packed in well-defined fields in special registers. An operation on these registers individually calculates the result for each data word in a single clock cycle. Various implementations include VIS on Sun UltraSparc processors, MMX/SSE/3DNow! on x86 processors, and AltiVec on the PowerPC. Wozniak [54] used the Visual Instruction Set on the UltraSPARC to achieve a two-fold speed increase. Rognes et al. [50] report a six-fold speedup on a single Pentium III workstation using MMX instructions. Clusters of workstations have been used to exploit coarse-grained parallelism. Martins et al. [43] report a speedup of 90 on a 120 node PC cluster using their EARTH architecture.

*Supercomputers* provide high-throughput sequence analysis, though they are expensive and not widely available. BLAZE, an implementation for the massively parallel 4096 processor MasPar MP1104 [20], achieved a speedup of more than four orders of magnitude when compared to workstations.

*VLSI devices* are usually explicitly designed for a single application. They achieve high performance but are more expensive than general-purpose workstations. Programmable VLSI based architectures execute a class of similar applications. The UCSC Kestrel Parallel Processor [18] is a SIMD processor with a 512 array of 8-bit processing elements capable of running a wide range of scientific applications. They report a speedup of two orders of magnitude when compared with an UltraSPARC-II machine. VLSI devices, however, cannot be updated to exploit advancements in technology. They are also constrained by their initial design. For example, 8-bit processing elements on the Kestrel make it unsuitable for aligning large sequences.

*Reconfigurable Hardware* systems are flexible devices that can execute a wide array of applications. They achieve performance approaching that of VLSI devices at a fraction of the cost. A number of FPGA systems such as DeCypher [3], SPLASH 2 [33], Cray XD1 [2], and others [56] accelerate Smith-Waterman, reporting several orders of magnitude speedup. These systems can easily be ported to newer generations of FPGAs with minimum re-design.

It is important to note that Smith-Waterman is essentially the gapped extension stage of the BLAST algorithm. Acceleration of Smith-Waterman alone in hardware is insufficient to perform high-throughput sequence analysis. To illustrate this point, we measured the performance of a software implementation of Smith-Waterman, ssearch [48]. On our benchmark machine, ssearch computes the dynamic programming matrix at approximately 89 million cell updates per second. A hardware

implementation running 100x faster is estimated to take approximately 33 hours to compare the entire E. coli. K12 proteome against NR (this does not include post-processing, i.e., time spent retrieving significant alignments). In contrast, NCBI BLASTP running on a single general-purpose workstation performs the same comparison in approximately 36 hours.

The above result shows that a highly accelerated implementation of Smith-Waterman on an FPGA runs in about the same time as NCBI BLASTP running on a Pentium 4 workstation. We conclude that the BLASTP heuristic must be accelerated in order to realize a significant speed improvement for protein sequence analysis.

### 1.4.3 BLAST accelerators

The majority of BLAST accelerators run on a cluster of workstations. A few have been designed to run on FPGA devices.

**Faster Search Algorithm BLAST:** [21, 22] (FSA-BLAST[1]) employs software optimization and modifications to the BLAST algorithm. The lookup table in stage 1 is replaced by a deterministic finite automaton that is engineered for fast, cache-conscious operation. A *semi-gapped* extension stage is added between stages 2 and 3 to further filter data. Here, a dynamic programming recurrence similar to the ungapped extension stage is used, but with gaps allowed only at every $n^{th}$ residue in the two sequences. Finally, the recurrence of the gapped extension phase is modified to disallow adjacent gaps in the two sequences, leading to reduced computation per cell. An overall speedup of 20-30% over NCBI BLASTP is reported.

**Apple/Genentech BLAST:** [1] (AG-BLAST) Apple Computer and Genentech provide an open-source version of NCBI BLAST customized to use Altivec instructions on PowerMac G4 and G5 processors. The modifications are in the seed generation stage. AG-BLAST used with word lengths 20 - 40 provide a two-fold speed increase over MegaBLAST. However, the use of large word lengths makes it unsuitable for searching divergent sequences.

**BLAST clusters:** The embarassingly parallel nature of BLAST can be exploited to run on a cluster of nodes. Query segmentation splits the set of query sequences and runs each on individual nodes of a cluster. BLAST searches a subset of the queries against the entire database on each node. This approach provides a near linear scalability if the database can fit in main memory. Alternatively, the database can be segmented, with the same query being processed against different subsets of the database. NCBI-BLAST implements a native multi-threaded search that can take advantage of SMP systems. **M**essage **P**assing **I**nterface BLAST (mpiBLAST) [25] is capable of running on a diverse set of architectures including Beowulf clusters, exhibiting near linear scalability on small numbers of nodes. SGI **H**igh **T**hroughput **C**omputational BLAST (HTC-BLAST) [9] is a distributed

[1] http://www.fsa-blast.org/

cluster implementation of BLAST on SGI Origin 300 servers that enables high-throughput homology searching. A BLASTX comparison of a large number of query sequences against the NR protein database on a 32-processor cluster yields a 30x speedup over a single machine. A commercial offering, TurboBLAST [17], runs on many parallel computing environments including heterogeneous workstations, parallel supercomputers, and grids. Paracel BLAST [8] is designed to run on high-end Sun clusters.

Cluster implementations can significantly decrease turn-around time on high-throughput BLAST searches. Distributed resources can be harnessed to search large queries or databases which would be infeasible on a single node. However, they scale poorly as more nodes are added to the cluster, since more time is spent formatting databases and collating results. Equal-size database segments on each node need not mean equal workload on the nodes. A large number of homologous sequences in a database segment can cause an imbalance in the load. Clusters typically also have high operational costs when compared to single-node solutions.

**FPGA Accelerators:** Rdisk[2] [40] is an FPGA based system to accelerate stage 1 of BLASTN. Reconfigurable logic is attached close to a hard disk, providing on-the-fly filtering capabilities. Rather than using lookup tables, the pattern matching computation is performed between a database word and all query words. This computation can proceed in parallel for all query words and requires processing elements proportional to the size of the query. Rdisk reports a throughput of 60 Mbases/sec for nucleotide searching. Such an approach, however, is not easily extensible to protein searches due to the more complicated neighbourhood strategy and the two-hit technique.

DeCypherBLAST [11] is a commercial product running on FPGA based engines attached to high-end servers. Scarcity of information on this offering makes a side-by-side comparison impossible.

RC-BLAST [45] is a recent implementation of the BLAST word matching phase on FPGAs. The work illustrates the difficulties faced in accelerating heuristic algorithms on FPGAs. The final FPGA implementation was slower than software version, although this was attributed to the limitations of the technology used by the authors.

TUC-BLAST [53] is an FPGA solution to accelerate DNA searches of small query sequences (1000 bases). The basic computation unit is a hit finder and an extension unit. The former stores a hash table of the query in on-chip block RAMs to detect hits. The extension unit performs ungapped extension to detect significant alignments. High-throughput searching is achieved by replication of the basic computation units. The basic idea is similar to our approach but is not easily extended to protein searches without a significant redesign of the architecture. Storing the hash table of the neighbourhood of a protein query in block RAMs on currently available FPGAs is infeasible due to its large size. Additionally, without the two-hit algorithm, the hit finder module quickly overwhelms the extension unit when performing protein searches.

[2] http://www.irisa.fr/symbiose/lavenier/rdisk.html

TreeBLASTP [32] is the only FPGA-based accelerator for BLASTP that we are aware of. Published in early 2006, this work accelerates seed generation and ungapped extension. The seed generation phase is similar to the one-hit approach. High-scoring word matches are detected using dynamic programming (thus eliminating lookup tables), and then passed to ungapped extension servers. Since two-hit filtering is not performed, larger word lengths and threshold values must be used so as to not overwhelm ungapped extension. The authors claim a database processing rate of 170 million amino acids per second on query sizes of 1024 residues on the latest FPGA. The effect of decreased sensitivity due to the higher neighbourhood threshold must be factored into these results.

An extensive literature survey of accelerated sequence analysis applications has established the need for faster solutions, specifically for BLAST. The limited number of BLAST accelerators, in particular for protein matching, highlights the difficulties faced in designing a hardware amenable architecture for seed generation. The focus of this thesis is a hardware/software design for an accelerated seed generation architecture.

## 1.5 Contributions

The subject of this thesis is part of a larger body of work done in the STTR group to accelerate sequence matching. Former and current members include Joseph Lancaster, Brandon Harris, Praveen Krishnamurthy and Richard D. Crowley. This research was conducted under the supervision of Dr. Jeremy Buhler and Dr. Roger Chamberlain.

The focus of this thesis is the seed generation stage. We designed and implemented this algorithm in hardware after a careful analysis to preserve the quality of the results. We achieved a speedup of 13x over the software equivalent. The performance of the seed generation stage was considered in the context of the BLASTP pipeline and is expected to run over 50x faster than a typical protein search on a standard workstation.

The author's specific contributions to the NCBI BLASTP acceleration effort are summarized below. Significant contributions by other members are highlighted at the relevant sections.

1. Contributions to the hardware design

   (a) The application profile of NCBI BLASTP was studied and the bottleneck stages identified. The performance characteristics of the pipeline were investigated, and the necessary acceleration required in each stage was determined.

   (b) The source code of the seed generation stage was examined in detail and the computationally expensive logic blocks identified. To enable high-throughput sequence analysis, the effect of various parameters and two algorithms on the performance was analysed.

The case for preserving NCBI BLAST-like sensitivity was established, and the effect of the selected set of parameters on the quality of results was studied.

(c) An initial design with a highly optimized implementation of the word matching stage running on the host CPU and the latter stages on an FPGA was considered. This was determined to be insufficient to meet the performance goal, and the requirement of placing all stages in hardware was established.

(d) The word matching hardware architecture was designed for the seed generation stage. An encoding scheme to pack query positions in SRAM was used to increase the throughput of this stage. The design was implemented in VHDL and optimized for high performance.

(e) A pipelined, high performance design for the two-hit module was implemented in hardware. The effect of the lack of feedback from the ungapped extension stage on the workload of the pipeline was characterized.

(f) Software simultations of the word matching and two-hit modules were written for use as a benchmark in the comparison of results generated by the hardware.

(g) Significant contributions were made in parallelizing the word matching module and in the design of switch1. Other contributors include Brandon Harris, Dr. Roger Chamberlain and Dr. Jeremy Buhler. The initial implementation was done by Richard D. Crowley. An optimized high performance implementation was later written by the author.

(h) The architecture for parallel two-hit modules and modulo division of work was designed. Praveen Krishnamurthy contributed to this design.

(i) Significant contributions were made to the design of the switch2 module along with Brandon Harris and Richard D. Crowley. An initial implementation of the switch was written by Richard D. Crowley. A high speed version of switch2 and the seed reduction tree was written by the author.

(j) The effect of out-of-order hits due to parallel word matching modules was closely examined. The stages in the seed generation hardware were designed to minimize this effect.

(k) The parallel seed generation hardware, including multiple copies of the word matching and two hit modules, was implemented and verified for functional correctness.

2. Contributions to the software design

(a) The necessity of an efficient algorithm to generate the neighbourhood of a query was established. An optimized prune-and-search neighbourhood generation implementation was written for this purpose. Dr. Jeremy Buhler was a major contributor to the design of this algorithm.

(b) A vector implementation of the prune-and-search algorithm was designed and implemented for high-performance neighbourhood generation.

(c) An encoding scheme for lookup table generation for use in the word matching stage was designed and implemented.

(d) A survey of bin packing approximation algorithms was performed, and the algorithms applied to query packing. The quality of results of the approximation algorithms was studied and the appropriate ones selected based on the deployment of the Mercury system.

(e) A multi-threaded, high performance software architecture for the Mercury BLASTP system was designed.

(f) The software design was implemented and interfaced to the Exegy FPGA communication wrapper. The Mercury software system was incorporated with the NCBI BLASTP source code to realize an integrated application.

(g) A reconfigurable logic test tool (rltesttool) to perform stability testing of FPGA logic blocks based on a scripted set of commands, was written to aid the hardware development process.

The rest of the thesis is organized as follows. Chapter 2 studies the various seed generation parameters affecting overall system performance and settles on an optimal set. Chapter 3 describes the parallel hardware architecture in detail. Chapter 4 details the necessary algorithmic modifications required in the software supporting the BLASTP pipeline. Chapter 5 reports performance numbers, and Chapter 6 concludes.

# Chapter 2

# Design of an accelerated seed generation stage

In this chapter we explore the various parameters of the NCBI BLASTP algorithm and study their effect on system performance. We first identify the bottleneck stages of BLASTP and characterize the workload of each stage. We then proceed to study the design space of the seed generation stage, to help build a hardware-amenable implementation.

## 2.1 Performance characteristics of NCBI BLASTP

To analyze the performance of the various stages of BLASTP, we determined its execution profile by running it with its default parameters (word length 3, threshold 11, E-value 10, reporting 500 best matches with alignments displayed for the first 250). Our benchmark machine was a single 2.8 Ghz Pentium 4 workstation with an L2 cache of 512 KB and 1 GB of RAM. We compared the GenBank Non-Redundant (NR) database[1] (3,292,813 sequences; 1,128,164,434 residues) against query sequences of various lengths chosen randomly from the Escherichia coli k12 proteome[2] (4,242 sequences; 1,351,322 residues). Timings were averaged over twenty queries for each length and reported at 95% confidence intervals.

Table 2.1: Percentage of execution time spent in the various stages of NCBI BLASTP

| **Query Length** (*residues*) | **Word Matching** | **Two-hit** | **Ungapped Extension** | **Gapped Extension** | **I/O & Pre/Post processing** |
|---|---|---|---|---|---|
| 512 | $45.99 \pm 3.23\%$ | $15.74 \pm 2.91\%$ | $12.74 \pm 3.87\%$ | $24.74 \pm 8.16\%$ | $0.73 \pm 9.22\%$ |
| 1024 | $36.61 \pm 2.87\%$ | $17.50 \pm 3.42\%$ | $14.15 \pm 5.03\%$ | $31.27 \pm 6.82\%$ | $0.42 \pm 7.95\%$ |
| 2048 | $30.96 \pm 2.22\%$ | $19.29 \pm 2.28\%$ | $15.85 \pm 2.07\%$ | $33.60 \pm 3.68\%$ | $0.24 \pm 4.84\%$ |
| 4096 | $28.72 \pm 2.66\%$ | $20.27 \pm 1.96\%$ | $16.64 \pm 1.86\%$ | $34.14 \pm 3.35\%$ | $0.16 \pm 7.22\%$ |

[1] ftp://ftp.ncbi.nih.gov/blast/db/{nr.00.tar.gz, nr.01.tar.gz}
[2] ftp://ftp.ncbi.nih.gov/genbank/genomes/Bacteria/Escherichia_coli_K12/U00096.faa

Table 2.1 shows the execution profile of NCBI BLASTP. Seed generation dominates, accounting for up to half the execution time. While time spent in the gapped extension stage in BLASTN is virtually negligible [38], protein matching spends close to a third of its execution time in this stage. A BLASTP accelerator must necessarily speed up all three stages of the pipeline to achieve an overall performance boost. Note that as the query size increases, the percentage of time spent in the word matching stage progressively decreases, while the rest of the pipeline shows a gradual increase. Longer query sequences increase the probability of a hit in the word matching stage, leading to more work for the latter stages. We ignore disk I/O and pre/post processing times (including query filtering, alignment traceback, and output formatting) in further analysis since it takes less than one percent of the total execution time.

Table 2.2: Percentage of execution time spent in the various stages of NCBI BLASTP for various E-values at a query size of 2048

| **E-Value** | **Word Matching** | **Two-hit** | **Ungapped Extension** | **Gapped Extension** | **I/O & Pre/Post processing** |
|---|---|---|---|---|---|
| $10^{-5}$ | $30.99 \pm 2.28\%$ | $18.92 \pm 2.19\%$ | $16.21 \pm 7.28\%$ | $33.59 \pm 4.12\%$ | $0.23 \pm 6.70\%$ |
| $10^{-2}$ | $31.03 \pm 2.39\%$ | $19.09 \pm 2.46\%$ | $15.95 \pm 1.46\%$ | $33.65 \pm 3.67\%$ | $0.23 \pm 7.85\%$ |
| 10 | $30.96 \pm 2.22\%$ | $19.29 \pm 2.28\%$ | $15.85 \pm 2.07\%$ | $33.60 \pm 3.68\%$ | $0.24 \pm 4.84\%$ |

Table 2.3: Percentage of execution time spent in the various stages of NCBI BLASTP for different organisms at a query size of 2048

| **Query** | **Word Matching** | **Two-hit** | **Ungapped Extension** | **Gapped Extension** | **I/O & Pre/Post processing** |
|---|---|---|---|---|---|
| E.coli K12 | $30.96 \pm 2.22\%$ | $19.29 \pm 2.28\%$ | $15.85 \pm 2.07\%$ | $33.60 \pm 3.68\%$ | $0.24 \pm 4.84\%$ |
| D.melanogaster (Chr3) | $32.57 \pm 3.96\%$ | $19.07 \pm 3.70\%$ | $15.45 \pm 3.69\%$ | $32.58 \pm 7.40\%$ | $0.27 \pm 6.92\%$ |
| P.falciparum (Chr14) | $27.04 \pm 5.61\%$ | $15.51 \pm 5.59\%$ | $13.60 \pm 8.25\%$ | $43.47 \pm 5.91\%$ | $0.25 \pm 9.89\%$ |
| S.enterica chol. | $31.98 \pm 2.19\%$ | $19.53 \pm 2.04\%$ | $16.02 \pm 2.48\%$ | $32.16 \pm 3.96\%$ | $0.24 \pm 6.91\%$ |
| S.pneumoniae R6 | $30.22 \pm 2.49\%$ | $18.20 \pm 2.70\%$ | $15.29 \pm 3.00\%$ | $36.00 \pm 4.24\%$ | $0.23 \pm 7.97\%$ |

The execution profile remains the same for various E-values (Table 2.2). A lower E-value leads to a more restrictive threshold in the gapped extension phase and thus only affects the I/O and post processing time. We observe similar results (Table 2.3) when other organisms are used as the query sequence dataset. Though we restrict ourselves to the Escherichia coli k12 proteome in further analysis and our performance measurements, we expect to achieve similar results with other query organisms.

Table 2.4 shows the time $t_i$ spent in stage $i$ of the pipeline per input database residue, w-mer match, seed, or HSP. Later stages of the pipeline are computationally more expensive. Ungapped extension spends 2-5x more time per input item as compared to seed generation, while full Smith-Waterman consumes several orders of magnitude more time than ungapped extension.

Table 2.5 shows the data match rates across the pipeline. The match rate, $p_i$, represents the probability that an output item from stage $i$ is produced by a single input into that stage. Here, $1a$

Table 2.4: Time $t_i$ spent in stage $i$ per input item

| **Query Length** ($residues$) | **Word Matching** ($\mu sec/residue$) | **Two-hit** ($\mu sec/w-mer$) | **Ungapped Extension** ($\mu sec/seed$) | **Gapped Extension** ($\mu sec/hsp$) |
|---|---|---|---|---|
| 512 | 0.0312 | 0.0112 | 0.2193 | 163.6821 |
| 1024 | 0.0401 | 0.0103 | 0.2043 | 155.5074 |
| 2048 | 0.0619 | 0.0100 | 0.1926 | 156.6640 |
| 4096 | 0.1043 | 0.0096 | 0.1873 | 147.2897 |

Table 2.5: Data match rates $p_i$ of the various stages of NCBI BLASTP

| **Query Length** ($residues$) | **Word Matching** ($r_{1a}$) | **Two-hit** ($p_{1b}$) | **Ungapped Extension** ($p_2$) | **Gapped Extension** ($p_3$) |
|---|---|---|---|---|
| 512 | 0.9566x | 0.0412 | 0.0026 | 0.0507 |
| 1024 | 1.8561x | 0.0409 | 0.0029 | 0.0486 |
| 2048 | 3.8733x | 0.0425 | 0.0026 | 0.0314 |
| 4096 | 7.6555x | 0.0422 | 0.0026 | 0.0419 |

and $1b$ refer to the word matching and two-hit stages respectively. All stages except word matching are extremely good filters, discarding more than 95% of their input and leaving progressively less work for later stages. Unlike BLASTN, the word matching stage of BLASTP is a net expander of data. The hit generation rate, $r_{1a}$, of this stage represents the number of w-mer matches generated per input residue of the database. Inexact matching combined with a short word length results in the data expansion characteristic of this stage. The two-hit stage – computationally, the least expensive in the pipeline – discards close to 95% of these w-mers. This two-hit design for the seed generation stage results in high sensitivity at low computational cost.

Protein sequences tend to be relatively small in size (the average sequence length in the NR database is 319 residues). Hence, short sequences may be concatenated and processed in a single pass to decrease overall run time. Conversely, longer sequences may be split into smaller, overlapping chunks and processed over multiple runs. However, as was noted previously, the length of a query sequence has an effect on the execution profile of BLASTP. Linear growth in the number of word matches with increasing query length has the potential to overwhelm later stages. This places a limit to the performance benefit observed by increasing the size of the query sequence processed in a single pass.

To characterize the performance of BLASTP, we consider the throughput of individual stages, as well as the entire pipeline. Throughput, $Tput_i$, for stage $i$ is the number of input items that can be processed per second and is given by $1/t_i$. The throughput of the entire pipeline is the number of input database residues that can be processed per second. On a sequential computational resource, the average compute time per input residue of the database is given by $t = t_{1a} + r_{1a}t_{1b} + r_{1a}p_{1b}t_2 + r_{1a}p_{1b}p_2t_3$. The throughput of the run is then $1/t$.

Table 2.6: Throughput of NCBI BLASTP

| **Query Length** (*residues*) | **Word Matching** ($10^6 res./s$) | **Two-hit** ($10^6 w-mers/s$) | **Ungapped Extension** ($10^6 seeds/s$) | **Gapped Extension** ($10^6 hsps/s$) | **NCBI BLASTP** ($10^6 res./s$) | **I/O data rate** ($MB/s$) |
|---|---|---|---|---|---|---|
| 512 | 32.05 | 89.29 | 4.56 | 0.0061 | **14.85** | 8.85 |
| 1024 | 24.94 | 97.09 | 4.89 | 0.0064 | **9.18** | 5.47 |
| 2048 | 16.16 | 100.00 | 5.19 | 0.0064 | **5.02** | 2.99 |
| 4096 | 9.59 | 104.17 | 5.34 | 0.0068 | **2.76** | 1.65 |

Table 2.6 gives the throughput of individual stages in millions of input items per second. The seed generation phase processes input items at a rate several times faster than the downstream stages. Throughput is strongly dependent on the query size. While throughput increases slightly in the two-hit and extension stages for longer queries, word matching performance progressively degrades. This is explained by the linear growth with query size of the w-mer generation rate in the word matching stage. A longer query sequence increases the number of matches expected purely by chance and not part of a meaningful alignment. This leads to an increase in the time spent per input residue in the word matching stage.

An important observation is the low throughput of BLASTP when compared to BLASTN. While the former runs at only 15 million residues per second for a 512-residue query, BLASTN is capable of executing an order of magnitude faster for even 10,000-base query sequences. Assuming 5 bits to encode a residue (20 amino acids), we compute in Table 2.6 the disk I/O rate required to support the different throughputs. A 2048-residue query sequence requires a sustained database transfer rate of only 3 MB/sec from disk to processor. The Mercury platform is capable of data transfer to its computational unit at over 700 MB/sec. Hence, an accelerated implementation of BLASTP on the Mercury system running even 100x faster than the software will not be I/O limited.

We investigate the effect of improved individual stages on the performance of the NCBI BLASTP pipeline. Figure 2.1 graphs the overall system performance as a function of the throughput of individual stages, for 512-residue queries. In each case, the throughput of a single stage is increased, while the remaining stages run at the speeds indicated in Table 2.6. When accelerating the seed generation stage, we assume a two-hit module capable of ingesting hits at the rate produced by the word matching module.

While improved ungapped and gapped extension stages provide a small increase in overall performance, an accelerated seed generation stage has the most impact. The seed generation stage is the bottleneck stage of the pipeline and must be significantly accelerated for any overall performance improvement. The graph also shows that acceleration of just a single stage produces only a modest increase in the throughput of the pipeline. To achieve a significant order of magnitude speedup, it is necessary that all three stages are accelerated. In this work we characterize the design of an accelerated seed generation stage. Ungapped and gapped extension stages are dealt with in [39, 31].

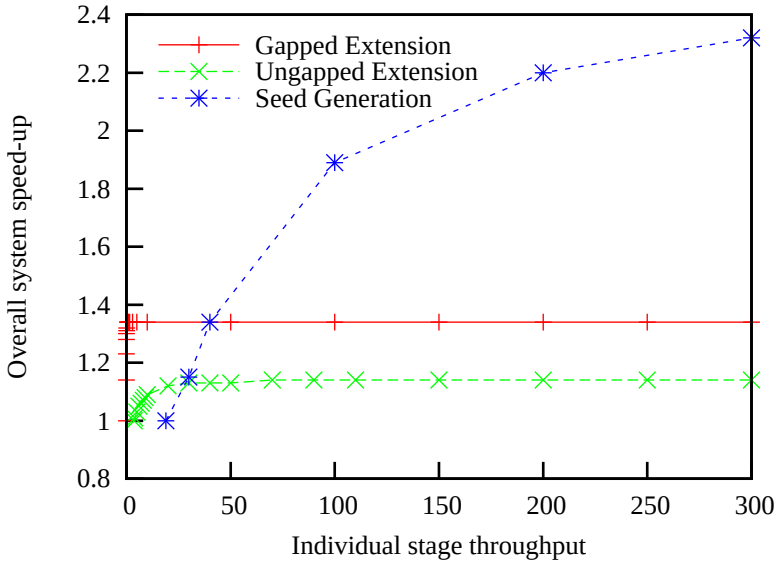



Figure 2.1: Speedup of NCBI BLASTP with improved performance of an individual stage

## 2.2 Software acceleration attempts

The word matching module of the seed generation stage uses a direct lookup table to generate hits. This table stores the neighbourhood of the query sequence and is used for efficient scanning against the database. The size of this lookup table increases exponentially with the length of the w-mer. For example, a 3-mer table requires just a few KB, while a 4-mer table requires close to 1MB of storage space. A hardware implementation of the word matching stage must store this table in memory. While currently available FPGAs provide on-chip block RAMs, they are typically limited to small capacities. Off-chip SRAM/DRAM modules must be used for applications with larger memory requirements. However, modern workstations have ample storage space in the form of caches and memory modules. To take advantage of this resource, we investigated a design that uses an accelerated software implementation of the word matching stage, with the remaining stages running on an FPGA.

A number of optimizations were employed to decrease the size of the lookup table and improve the performance of the word matching module. The alphabet size was reduced to 20, discarding w-mers containing ambiguity characters. Using base-20 in place of the existing base-32 arithmetic to generate w-mer indices into the table further reduced its size. Lookup table locations were aligned at word boundaries to reduce misaligned memory read penalties. The word matching loop was unrolled several iterations, and memory was explicitly prefetched to reduce the number of cache misses.

This software-hardware approach yielded limited success in accelerating the word matching stage. The major limiting factor is the memory bottleneck: the database and the lookup table are too large

to fit in cache memory. Database w-mer lookups follow a random access pattern, further increasing the number of cache misses. We conclude, therefore, that a hardware implementation that solves the memory resource constraint is necessary to achieve a significant speedup of the seed generation stage.

## 2.3 Design space exploration

In this section, we investigate the parameters that control the seed generation stage and characterize their effect on the BLASTP pipeline. The various parameters are studied to converge on a set of values that maximizes the throughput of a hardware implementation. However, the limited resources available on an FPGA are a constraint that must be carefully considered. The design space exploration is treated as a throughput optimization problem given certain resource constraints.

An accelerated implementation of the BLASTP algorithm must not only yield a significant speedup but also produce results similar to the standard software implementation. The parameters used in the seed generation algorithm have a considerable impact on sensitivity. As the popularity of BLAST has soared, it has become an important component of commercial research. However, institutions have been reluctant to adopt faster, alternative heuristic solutions for fear of "missing" hits that might otherwise have been found with NCBI BLAST. This has resulted in the limited adoption of a number of commercial products, including other BLAST-like accelerators [29]. In our design of Mercury BLASTP, we place great importance in preserving its output characteristics. A set of parameter values that maximizes the pipeline throughput must not come at the expense of sensitivity of the search result.

An FPGA has limited logic and memory resources on board. Memory is of particular concern in the word matching stage. The lookup module probes a table containing the neighbourhood of a query. In general, this table is too large to be stored in on-chip block RAMs and requires an off-chip SRAM resource. The speed of the SRAM and its capacity are the resource constraints considered in this section. The former decides the maximum speed of the lookup module; the latter limits the size of the query neighbourhood.

The word length and threshold parameters used to generate the neighbourhood of a query sequence directly influence the throughput of the pipeline. Longer word lengths or higher threshold values increase throughput. However, this comes at the cost of decreased sensitivity, and so a careful tradeoff must be made. The throughput of a BLASTP search can be increased by concatenating multiple query sequences, and processing them in a single pass of the database. However, this increases the neighbourhood size, which may exceed the capacity of the SRAM. Finally, we consider the original one-hit algorithm for seed generation and examine it as an alternative to two-hit.

In the following sections, we first study the effect of the neighbourhood parameters on the sensitivity of a BLASTP search. We then consider the resource constraints to narrow down the parameter space. In order to measure the throughput of a hardware design, we first develop a mean-value performance model. Using this, we select a set of parameters that maximizes the throughput of the seed generation stage.

### 2.3.1 Sensitivity

Alignments generated by BLASTP searches at different neighbourhood parameters were compared for sensitivity. NCBI BLASTP was run with a neighbourhood of $N(3, 11)$ at a cut-off E-value of $10^{-5}$. The resultant HSPs reported were designated as the "gold standard." The restrictive E-value is justified, since the default of 10 generates a large number of biologically spurious hits purely by chance. The neighbourhood parameters were then varied and the search results compared against the gold standard.

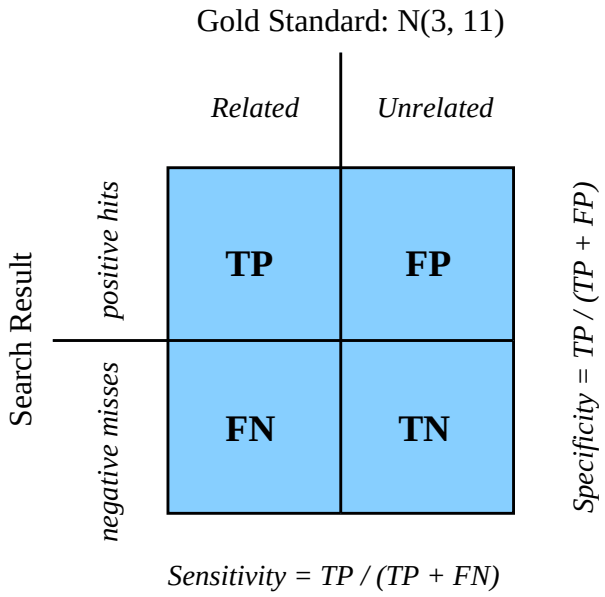



Figure 2.2: Classification of search results

An HSP in the gold standard is designated as detected in the search result (true positive TP) if the latter contains an HSP between the same query/database pair whose alignment overlaps at least half the original query and database residues. In practice, greater than 99.5% of true positives exhibited total overlap in our experiments. An HSP in the gold standard that fails to be identified by a search result is termed a false negative (FN). Sensitivity is measured as $TP/(TP + FN)$, where a number closer to 1 signifies a larger coverage of the gold standard. False positives (FP) are those HSPs

detected by the search result, but not present in the gold standard. The specificity is TP/(TP + FP), where a number closer to 1 signifies increased accuracy of the search. These definitions are summarized in Figure 2.2.

### Neighbourhood parameters

To study the effect of the neighbourhood parameters on the sensitivity of NCBI BLASTP, we compared an earlier release of the GenBank Non-Redundant (NR) database (2,321,957 sequences; 787,608,532 residues) against the entire Escherichia coli k12 proteome (4,242 sequences; 1,351,322 residues). Table 2.7 shows the sensitivity and specificity of BLASTP searches for various neighbourhoods. The gold standard of $N(3, 11)$ recorded 2,623,223 HSPs between the NR database and the Escherichia coli k12 proteome.

Longer word lengths or higher threshold values increase the probability of missing a seed in a biologically meaningful alignment. The sensitivity of a search at a particular word length is higher for a lower threshold value. For example, $N(4, 11)$ and $N(5, 13)$ perform extremely well. Increasing the threshold shows a slow decline in sensitivity followed by a rapid fall off as the threshold value increases beyond a critical point.

Table 2.7: Sensitivity of the two-hit BLASTP algorithm for various neighbourhoods

| **N(w, T)** | **# HSPs detected** ($TP$) | **# HSPs missing** ($FN$) | **# New HSPs detected** ($FP$) | **Sensitivity** ($TP/(TP+FN)$) | **Specificity** ($TP/(TP+FP)$) |
|---|---|---|---|---|---|
| $N(4, 11)$ | $2,622,471$ | 752 | $7,658$ | 0.9997 | 0.9971 |
| $N(4, 12)$ | $2,620,928$ | $2,295$ | $6,332$ | 0.9991 | 0.9976 |
| $N(4, 13)$ | $2,612,600$ | $10,623$ | $4,588$ | 0.9960 | 0.9982 |
| $N(5, 13)$ | $2,621,263$ | $1,960$ | $6,841$ | 0.9993 | 0.9974 |
| $N(5, 14)$ | $2,616,699$ | $6,524$ | $5,262$ | 0.9975 | 0.9980 |
| $N(5, 15)$ | $2,601,312$ | $21,911$ | $3,958$ | 0.9916 | 0.9985 |
| $N(5, 17)$ | $2,475,696$ | $147,527$ | $14,542$ | 0.9438 | 0.9942 |
| $N(5, 18)$ | $2,334,988$ | $288,235$ | $32,385$ | 0.8901 | 0.9863 |

Specificity remains high for the various neighbourhood parameters and is much less of a concern. It is important to note that the gold standard itself is a result of statistical analysis using computational methods and does not necessarily imply biological similarity. Hence, though we aim to reduce false positives to produce a result identical to default NCBI BLASTP, the newly discovered HSPs are in fact statistically significant. A neighbourhood of $N(4, 12)$, for example, detects more than twice as many new statistically significant HSPs as it loses, while still running faster than default BLASTP. While it would have been preferable to use a gold standard based on a known list of biologically relevant HSPs, as far as we are aware, no such list exists for large databases.

We consider the neighbourhood parameters having a sensitivity greater than 99.5% as prime candidates for the target implementation.

**Seeding strategy**

Recent research [37] suggests that the seeding strategy used in the word matching stage affects the search sensitivity. While BLASTP uses a consecutive subsequence of $k$ residues as its seed, a vector seed uses a discontiguous subsequence of residues. As described in section 1.4.1, a vector seed $(v, T)$ is specified by a model $v \in \{0,1\}^*$ and a threshold value $T$. Positions indicated by a 1 in the seed model are checked for a positive match, while all others are ignored. A seed match is generated when the sum of the scores of the residue pairs is greater than or equal to $T$.

Vector seeds increase the independence of match events at adjacent database residues, and have been shown to improve sensitivity and specificity for DNA searches [42, 41]. The construction of vector seeds during the database scanning phase is expensive in software, however, a hardware implementation can be easily adapted for this purpose. We briefly investigated the use of vector seeds to improve the sensitivity of protein matching.

Table 2.8: Sensitivity of vector seeds

| **v, T** | **# HSPs detected** $(TP)$ | **# HSPs missing** $(FN)$ | **# New HSPs detected** $(FP)$ | **Sensitivity** $(TP/(TP+FN))$ | **Specificity** $(TP/(TP+FP))$ |
|---|---|---|---|---|---|
| $(1,1,1,1),13$ | $2,324,875$ | $8,405$ | $3,973$ | 0.9964 | 0.9983 |
| $(1,0,1,1,1),13$ | $2,330,785$ | $2,495$ | $5,494$ | 0.9989 | 0.9976 |
| $(1,1,1,0,0,0,1),13$ | $2,331,113$ | $2,167$ | $5,564$ | 0.9991 | 0.9976 |
| $(1,0,1,1,1),14$ | $2,320,649$ | $12,631$ | $4,042$ | 0.9946 | 0.9983 |

The word matching stage of the NCBI BLASTP software was modified to use a vector seeding strategy. 3000 sequences randomly selected from the Escherichia coli k12 proteome were compared against the GenBank Non-Redundant (NR) database. The gold standard was the default seed $(1,1,1),11$, which detected 2,333,280 HSPs. Sensitivity results are summarized in Table 2.8. Vector seeds show a slight increase in sensitivity when compared to their consecutive counterparts at the same threshold. For example, $(1,1,1,0,0,0,1),13$ loses 6238 fewer and finds 1591 more HSPs than $(1,1,1,1),13$. The resultant sensitivity is greater than 99.9% of the gold standard.

However, unlike for DNA searches, the advantage of using vector seeds for proteins is minimal. Vector seeds do not show a significant increase in sensitivity; we do not consider this further. The default seed $(1,1,1,1),13$ is used in our implementation.

### 2.3.2 Constraints

The off-chip SRAM used to store the neighbourhood of a query sequence is a critical resource. The query neighbourhood is packed into a direct lookup table stored in the SRAM. The packing scheme is described in detail in the next chapter. Increasing the word length increases the number of possible

unique w-mers in the neighbourhood and so the size of the table. Similarly, a lower threshold value increases the number of w-mers that map to the query sequence. The table size is also directly proportional to the length of the query sequence.

### Word and query sequence lengths

Table 2.9 shows the variation of the lookup table size with the neighbourhood parameters and the query sequence length. The data was averaged over twenty sequences of various lengths generated from the Escherichia coli k12 proteome. The key space is the number of unique w-mers for a particular word length and is computed as $|\Sigma|^w$, where the size of the alphabet is 20. The occupancy rate measures the fraction of the key space populated by w-mers in a neighbourhood. The large occupancy rates observed, especially for longer query sequences, justifies the use of a direct lookup table instead of a hashing mechanism.

For a fixed query sequence length, the size of the neighbourhood, and hence the lookup table, increases exponentially with word length. For example, neighbourhoods with a word length of 4 require under 2MB to store the lookup table, while those with a word length of 5 require a minimum of 10MB.

Table 2.9: Word and query lengths versus neighbourhood size

| | | **512** | | | **2048** | | | **4096** | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **N(w, T)** | **Key space** | **Occ. rate** | **Neigh. w-mers** | **Table size** | **Occ. rate** | **Neigh. w-mers** | **Table size** | **Occ. rate** | **Neigh. w-mers** | **Table size** |
| $N(3, 11)$ | $8 \times 10^3$ | 61% | $8,960$ | 35K | 95% | $36,399$ | 77K | 99% | $73,801$ | 134K |
| $N(4, 13)$ | $1.6 \times 10^5$ | 41% | $91,147$ | 634K | 85% | $371,257$ | 928K | 96% | $755,569$ | 1.6M |
| $N(4, 14)$ | $1.6 \times 10^5$ | 28% | $56,349$ | 628K | 70% | $230,267$ | 743K | 88% | $469,918$ | 1.1M |
| $N(5, 14)$ | $3.2 \times 10^6$ | 35% | $1,442,167$ | 12.3M | 80% | $5,873,872$ | 16M | 95% | $11,943,444$ | 25.7M |
| $N(5, 15)$ | $3.2 \times 10^6$ | 24% | $913,301$ | 12.3M | 65% | $3,734,387$ | 14M | 85% | $7,609,670$ | 18.5M |

The growth of the table size is more controlled with an increase in the query sequence length. A longer query sequence increases the number of w-mers in the neighbourhood, but the key space remains the same. For a neighbourhood of $N(4, 13)$, the data shows a modest 2.5x increase in the table size from a 512-residue to a 4096-residue query.

The SRAM available in our implementation is limited to 1MB of storage space. This excludes the use of neighbourhoods with a word length greater than 4 for 2048-residue query sequences.

### 2.3.3 Throughput

Now that the sensitivity of the search and the memory constraints have been considered, we estimate the hardware throughput for various points in the reduced parameter space. We develop a mean-value performance model to estimate the throughput of the BLASTP hardware pipeline. The pipeline executes word matching ($1a$), two-hit ($1b$), ungapped extension ($2a$), and gapped extension ($3a$) on the FPGA. The throughputs for the ungapped and gapped extension prefilters are computed from the implementations described in [39, 31].

#### Performance model

When executing in a pipelined fashion, the overall throughput is limited to the minimum throughput achieved by any one resource:

$$Tput_{pipe} = min(Tput_{1a}, Tput_{1b}, Tput_{2a}, Tput_{3a}) \text{ Mresidues/second}$$

where $Tput_{1a}, Tput_{1b}$, $Tput_{2a}$, and $Tput_{3a}$ are the throughputs of the word matching, two-hit, ungapped extension, and gapped extension stages respectively.

The throughput of the word matching stage is a function of its data input rate and its w-mer processing time. The target frequency of the FPGA implementation of the word matching stage is 140 MHz. This stage is capable of accepting up to 12 database residues per clock cycle. The w-mer processing time (expressed in clock cycles), $\mu$, is dependent on the neighbourhood parameters, the size of the query sequence and the design of the w-mer lookup module. Throughput of stage $1a$ is computed as:

$$Tput_{1a} = \frac{h \times f_1}{\mu} \text{ Mresidues/second}$$

where $h$ is the number of parallel copies of the w-mer lookup module and $f_1 = 140$ MHz is the clock frequency of this hardware stage. Note that each input w-mer in stage $1a$ corresponds to a single residue of the database.

An off-chip memory lookup table is accessed to detect query hit positions corresponding to a database w-mer. The neighbourhood parameters and the query size determine the number of query hit positions stored in this table. Additionally, the organization of the table decides the number of memory probes required to retrieve all hits corresponding to a w-mer. We compute $\mu$ empirically for

every set of neighbourhood parameters and query sequence lengths. The organization of the lookup table is described in the next chapter.

Table 2.10: Calculation of $\mu$ for a neighbourhood of $N(4, 13)$ and a 2048-residue query. Note: a w-mer is never satisfied in two probes (Refer Chapter 3)

| **Table probes** | **Probability** | **Value** |
|---|---|---|
| 1 | 0.8262 | 0.8262 |
| 3 | 0.1548 | 0.4643 |
| 4 | 0.0175 | 0.0699 |
| 5 | 0.0015 | 0.0074 |
| 6 | 0.0001 | 0.0007 |
| **Weighted average:** | | $\mu = 1.3684$ |

Table 2.10 shows the calculation of the empirical value for $\mu$. The data was averaged over NCBI BLAST searches of twenty 2048-residue query sequences against the NR database, using a neighbourhood of $N(4, 13)$. The weighted average $\Sigma g_i a_i$, is computed from the probability $g_i$ that a database w-mer is satisfied in exactly $a_i$ probes.

The normalized throughput of the two-hit stage is a function of its data input rate and the hit generation rate of the word matching module. A clock frequency of $f_1 = 140$ MHz is targeted for the hardware two-hit stage, and it is capable of accepting one hit per clock cycle. Throughput of stage $1b$ is computed as

$$Tput_{1b} = \frac{b \times f_1}{r_{1a}} \text{ Mresidues/second}$$

where $b$ represents the number of parallel copies of the two-hit module running in hardware.

We measure the normalized throughput of stage $2a$ as

$$Tput_{2a} = \frac{f_2}{r_{1a} p_{1b}} \text{ Mresidues/second}$$

where the ungapped extension prefilter runs at $f_2 = 75$ MHz on the FPGA, and is capable of accepting one seed per clock cycle. $r_{1a} p_{1b}$ represents the number of seeds passed into stage $2a$ per input residue.

Finally, the normalized throughput of stage $3a$ is computed as

$$Tput_{3a} = \frac{f_3}{\mu_{3a} \times (r_{1a} p_{1b} p_{2a})} \text{ Mresidues/second}$$

where the gapped extension prefilter runs at $f_3 = 66$ MHz on the FPGA. $\mu_{3a}$ represents the average number of clock cycles required to process an input HSP in the gapped extension stage. We use $\mu_{3a} = 700$ in our calculations. $r_{1a}p_{1b}p_{2a}$ represents the number of HSPs passed into stage $3a$ per input residue.

### One-hit versus two-hit

One of two candidate algorithms may be implemented in the seed generation hardware stage. We establish the superiority of the two-hit over the one-hit algorithm by making an important observation that affects the performance of the hardware implementation. To compare the two algorithms, we measured their match rates and throughputs by performing BLASTP searches of twenty 2048-residue sequences against the NR database. Tables 2.11 and 2.12 display the results.

Table 2.11: Match rates of the one-hit and two-hit algorithms

| **Algorithm** | **N(w, T)** | **Word Matching** ($r_{1a}$) | **Two-hit** ($p_{1b}$) | **Ungapped Extension** ($p_{2a}$) |
|---|---|---|---|---|
| two-hit | $N(3,11)$ | 3.8727 | 0.0424 | 0.0025 |
| | $N(4,13)$ | 2.0067 | 0.0183 | 0.0091 |
| | $N(4,17)$ | 0.2097 | 0.0026 | 0.0843 |
| one-hit | $N(3,11)$ | 3.8727 | – | 0.0001 |
| | $N(4,13)$ | 2.0067 | – | 0.0003 |
| | $N(4,17)$ | 0.2097 | – | 0.0011 |

The two-hit algorithm uses a shorter word length combined with a smaller threshold value to increase sensitivity. As a consequence, the word matching rate $r_{1a}$ increases drastically. However, the two-hit modules act as strong filters discarding a large fraction of input hits. One-hit seed generation, on the other hand, passes a much larger fraction of seeds to ungapped extension. For example, one-hit seed generation at $N(3,11)$ produces 24x more work for ungapped extension than its two-hit counterpart. Most of these seeds are discarded, as evidenced by the very low match rate in stage $2a$ of the one-hit pipeline. Relaxation of the one-hit neighbourhood parameters reduces this rate but comes at the cost of decreased sensitivity.

Table 2.12: Throughput of the one-hit and two-hit pipelines

| **Algo.** | **N(w, T)** | $\boldsymbol{\mu}$ (*clks/w-mer*) | $\boldsymbol{Tput_{1a}}$ ($10^6 res./s$) | $\boldsymbol{b}$ | $\boldsymbol{Tput_{1b}}$ ($10^6 res./s$) | $\boldsymbol{Tput_{2a}}$ ($10^6 res./s$) | $\boldsymbol{Tput_{3a}}$ ($10^6 res./s$) | $\boldsymbol{Tput_{pipe.}}$ ($10^6 res./s$) |
|---|---|---|---|---|---|---|---|---|
| two-hit | $N(3,11)$ | 2.1843 | 192 | 6 | 217 | 457 | 225 | 192 |
| | $N(4,13)$ | 1.3684 | 307 | 5 | 349 | 2,048 | 282 | 282 |
| | $N(4,17)$ | 1.0019 | 419 | 1 | 668 | 136,324 | 2,032 | 419 |
| one-hit | $N(3,11)$ | 2.1843 | 192 | – | – | 19 | 171 | 19 |
| | $N(4,13)$ | 1.3684 | 307 | – | – | 37 | 171 | 37 |
| | $N(4,17)$ | 1.0019 | 419 | – | – | 358 | 427 | 358 |

Throughput comparison for the hardware pipeline is done with $h = 3$ copies of the lookup module. The high hit generation rate when using the two-hit algorithm is dealt with by replicating the two-hit modules, using up to six copies for $N(3, 11)$. In this case, the throughput of the ungapped extension prefilter is extremely high, owing to the excellent filtering capabilities of stage 1. The performance of this configuration (except for $N(4, 13)$) is limited by the word matching module.

In the one-hit pipeline, the ungapped extension prefilter is overwhelmed by the high stage $2a$ input rate, especially at shorter word lengths. The throughput of stage 2a in the one-hit pipeline is up to several orders of magnitude lower than its two-hit counterpart. A similar effect is seen on the gapped extension stage. Using a more permissive neighbourhood of $N(4, 17)$ decreases $r_{1a}$, and so alleviates this problem to a certain extent. However, stage 2a remains the bottleneck of the pipeline.

This highlights an important advantage of the two-hit algorithm. The two-hit module may be replicated relatively cheaply to handle the high hit production rate. The ungapped extension stage by comparison consumes far more resources and is both costly and more complex to replicate. We conclude, therefore, that the two-hit pipeline configuration scales better with the number of lookup modules and is more suited for a hardware implementation.

### Neighbourhood

The performance of the seed generation stage is affected by the w-mer processing time, and the hit generation rate of the word matching stage. The throughput of the word matching module is inversely proportional to $\mu$. An increase in $r_{1a}$ requires two-hit module replication to keep up with the increased hit production rate. The values of both these parameters can be reduced by using more permissive neighbourhoods. Increasing the word length or the threshold value decreases $\mu$ and $r_{1a}$. We study the effect of various neighbourhood parameters on the throughput of the BLASTP hardware pipeline.

Table 2.13: Match rates of the two-hit algorithm for various neighbourhood parameters

| **N(w, T)** | **Word Matching** ($r_{1a}$) | **Two-hit** ($p_{1b}$) | **Ungapped Extension** ($p_{2a}$) |
|---|---|---|---|
| $N(3, 11)$ | 3.8727 | 0.0424 | 0.0025 |
| $N(4, 11)$ | 5.5245 | 0.0418 | 0.0023 |
| $N(4, 12)$ | 3.3531 | 0.0280 | 0.0047 |
| $N(4, 13)$ | 2.0067 | 0.0183 | 0.0091 |
| $N(5, 13)$ | 2.7100 | 0.0183 | 0.0089 |
| $N(5, 14)$ | 1.6741 | 0.0123 | 0.0166 |
| $N(5, 15)$ | 1.0176 | 0.0081 | 0.0295 |
| $N(5, 17)$ | 0.3595 | 0.0033 | 0.0838 |
| $N(5, 18)$ | 0.2095 | 0.0021 | 0.1337 |

Table 2.13 shows the match rates for various w-mer lengths, neighbourhood thresholds, and two-hit window lengths. The data was averaged over BLASTP searches of twenty 2048-residue query

sequences selected from the Escherichia coli k12 proteome against the NR database. In each configuration of the hardware pipeline, a single copy of the lookup module ($h = 1$) and a variable number of two-hit modules are used. The hit production rate for the default neighbourhood parameter is 3.8727 hits/residue. This drops by half for a neighbourhood of $N(4, 13)$, and by three-fourths for $N(5, 15)$. There is also a corresponding decrease in the workload of downstream stages. Smaller values of $r_{1a}$ translate to reduced workload in the two-hit stage and increased individual throughputs of downstream stages.

Table 2.14: Throughput of the two-hit algorithm for various neighbourhood parameters

| **N(w, T)** | $\boldsymbol{\mu}$ $(clks/w\text{-}mer)$ | $\boldsymbol{Tput_{1a}}$ $(10^6 res./s)$ | $\boldsymbol{b}$ | $\boldsymbol{Tput_{1b}}$ $(10^6 res./s)$ | $\boldsymbol{Tput_{2a}}$ $(10^6 res./s)$ | $\boldsymbol{Tput_{3a}}$ $(10^6 res./s)$ | $\boldsymbol{Tput_{pipe.}}$ $(10^6 res./s)$ |
|---|---|---|---|---|---|---|---|
| $N(3, 11)$ | 2.1843 | 64 | 2 | 72 | 457 | 225 | 64 |
| $N(4, 11)$ | 2.9383 | 48 | 2 | 51 | 325 | 176 | 48 |
| $N(4, 12)$ | 1.9495 | 72 | 2 | 84 | 799 | 212 | 72 |
| $N(4, 13)$ | 1.3684 | 102 | 2 | 140 | 2,048 | 282 | 102 |
| $N(5, 13)$ | 1.6473 | 85 | 2 | 103 | 1,509 | 212 | 85 |
| $N(5, 14)$ | 1.2332 | 114 | 2 | 167 | 3,632 | 275 | 114 |
| $N(5, 15)$ | 1.0689 | 131 | 2 | 275 | 9,084 | 388 | 131 |
| $N(5, 17)$ | 1.0042 | 139 | 1 | 389 | 62,736 | 941 | 139 |
| $N(5, 18)$ | 1.0010 | 140 | 1 | 668 | 170,808 | 1,606 | 140 |

Table 2.14 summarizes the pipeline throughput for various neighbourhood parameters. The w-mer processing time in the lookup module falls by over a clock cycle for $N(5, 15)$, from 2.1843 for the default parameters. This has a direct effect on the throughput of the word matching stage, with the former configuration running twice as fast as the latter. The effect on the throughput of the two-hit and ungapped stages is magnified, running almost 4x and 20x faster respectively. We conclude that increasing the word length has a favourable impact on the throughput of the hardware pipeline.

As discussed in the previous sections, we only consider word lengths of 3 and 4 for our hardware implementation because of the memory constraint. A configuration with a neighbourhood of $N(4, 13)$ produces the highest throughput, running 1.5x faster that the default parameters.

### Query sequence length

Having established the use of a neighbourhood of $N(4, 13)$, we now consider the optimum query sequence length. The length of a typical protein sequence is around 300 residues. Performing a search against a proteome requires a database scan against every one of its queries. If instead, they are packed to form larger composite sequences, the number of database scans can be reduced. Since the throughput remains almost unchanged for the larger query sequence, the runtime is reduced. We have already considered the effect of the query sequence length on the lookup table size. An increase in query size also leads to a larger match rate in the word matching module, and may overwhelm downstream stages.

Table 2.15: Variation of throughput with query length

| **Query length** (residues) | $\boldsymbol{\mu}$ ($clks/w\text{-}mer$) | $\boldsymbol{Tput_{1a}}$ ($10^6 res./s$) | $\boldsymbol{b}$ | $\boldsymbol{Tput_{1b}}$ ($10^6 res./s$) | $\boldsymbol{Tput_{2a}}$ ($10^6 res./s$) | $\boldsymbol{Tput_{3a}}$ ($10^6 res./s$) | $\boldsymbol{Tput_{pipe.}}$ ($10^6 res./s$) |
|---|---|---|---|---|---|---|---|
| 512 | 1.0085 | 139 | 1 | 284 | 8,609 | 1,174 | 139 |
| 2048 | 1.3684 | 102 | 2 | 140 | 2,048 | 282 | 102 |

Table 2.15 shows the variation of throughput with query length for the neighbourhood of $N(4, 13)$. As expected, the 2048-residue query has a 4x higher match rate in the word matching stage (data not shown). However, this higher match rate is easily handled by two-hit replication. While the throughput of the downstream stages decreases for the longer query sequence, word matching remains the bottleneck. In the word matching stage, the average number of clock cycles required to satisfy an input w-mer increases only slightly for the 2048-residue sequence. Consequently, throughput remains fairly constant resulting in decreased runtime for packed searches.

We use a query length of 2048 residues in our hardware implementation.

### 2.3.4 Summary

To summarize, we considered the design space of NCBI BLASTP as a throughput maximization problem, constrained by the memory resources available to the hardware implementation. A number of neighbourhood parameters with high sensitivity were considered. The 1MB capacity of the off-chip SRAM limited the word length to 4 and the query sequence length to under 4096 residues. Finally, the two-hit algorithm using a neighbourhood of $N(4, 13)$ and a query sequence of 2048 residues were selected to maximize throughput.

# Chapter 3

# Mercury BLASTP seed generation architecture

In this chapter we discuss the hardware design of Mercury BLASTP. We first describe the salient features of the Mercury architecture and briefly summarize past work done on an accelerated implementation of BLASTN. We then give a general description of the Mercury BLASTP design and describe in detail the seed generation stage.

## 3.1 Mercury architecture

The Mercury system [23] supports disk-based, high-throughput computation on reconfigurable logic. The goal of the design is to place low-complexity, high-throughput, application-specific logic blocks near the data source. Streams of data are continuously pumped at disk access speeds through this resource, performing one or more physically pipelined functions. High-complexity tasks are performed by more resource-intensive, general-purpose units present downstream, such as a microprocessor. The Mercury system is ideally suited for a computational pipeline that performs a large part of its computation in easily parallelizable, low-cost logic blocks placed upstream. In a typical design, FPGA-based co-processors receive data from disks over a high-bandwidth I/O bus. One or more microprocessors on the host, running a conventional operating system, manage the functioning of the FPGA as well as performing high-complexity tasks. Hence, the Mercury system is ideal for hardware-software codesign problems. Applications previously implemented on the Mercury system include text search and DNA sequence matching.

The Mercury system used for implementing BLASTP consists of two 2.0 GHz AMD Opteron CPUs with 6 GB of memory. The host CPU runs a customized version of Linux 2.6.16-rc5 with Mercury kernel drivers and an API provided by Exegy Inc[1]. Two prototyping co-processor boards are connected to the processors and the disk subsystem (SCSI Ultra320 10,000 RPM disk drive) via

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

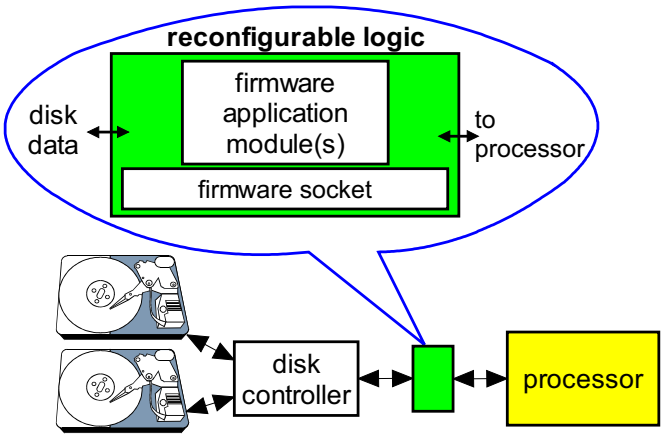



Figure 3.1: Mercury system architecture

the PCI-X bus. The first board contains a Xilinx Virtex-II 6000 FPGA with 33,792 slices and 144 18-Kbit on-chip block RAM memory, while the second board contains a Xilinx Virtex-II 4000 FPGA with 23,040 slices and 120 18-Kbit on-chip block RAM memory. In addition, up to four synchronous SRAM modules (GS88032BT-150), each providing 1MB of off-chip memory are available. On this configuration, members in our group [24] have demonstrated sustained data throughput of over 700 MB/sec from the disk to the FPGA.

## 3.2 Mercury BLASTN

In this section we compare and contrast Mercury BLASTN with BLASTP. A detailed description may be found in [38, 39]. The BLASTN heuristic consists of a similar pipeline; however the distribution of execution time is different, being spent primarily in the seed generation stage. The most significant observation from the execution profile in Table 3.1 is the negligible time spent performing gapped extension. As a result of these characteristics, only the first two stages were implemented in hardware. The Mercury BLASTN deployment is illustrated in Figure 3.2.

Table 3.1: Percentage of pipeline time spent in each stage of NCBI BLASTN [39]

| **Query Size** (bases) | **Stage 1** | **Stage 2** | **Stage 3** |
|---|---|---|---|
| 10 K | 86.53±1.33% | 13.24±1.99% | 0.23±0.02% |
| 25 K | 83.89±2.56% | 15.88±4.40% | 0.22±0.04% |
| 50 K | 82.63±2.94% | 17.28±4.96% | 0.09±0.01% |
| 100 K | 83.35±1.28% | 16.58±2.17% | 0.08±0.01% |
| 1 M | 85.39±3.34% | 14.68±5.24% | 0.03±0.01% |

The seed generation stage of BLASTN is substantially different in its design. Since DNA uses an alphabet of size 4, exact matches of a given length are statistically more probable than in proteins.

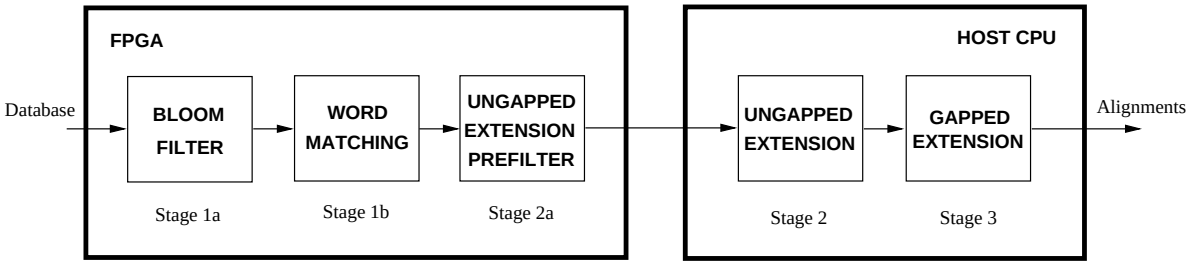



Figure 3.2: Mercury BLASTN: hardware/software deployment [38]

BLASTN uses w-mers of size 11 rather than a neighbourhood seeding strategy, along with the one-hit algorithm, where a single word match triggers a seed. Further, unlike in BLASTP, the word matching stage is a strong filter. A parallel Bloom filter [19] stage is employed to rapidly inspect database residues, discarding more than 90% of input w-mers. A Bloom filter is a probabilistic algorithm that is used to test the presence of input w-mers in a large member set. This can be done using far fewer resources than a direct lookup table and is easily implemented in hardware using on-chip block RAMs. A number of such parallel blocks can support high-throughput scanning of the database (16 residues/clk). A near-perfect hashing scheme [38] is used to query off-chip SRAM, to retrieve word matches of flagged w-mers at the rate of one per clock. Query positions corresponding to an exact match are passed directly from the SRAM as seeds for ungapped extension. The need to look up the SRAM is a potential bottleneck in the pipeline; however, the Bloom filter greatly reduces the load in this phase.

An ungapped extension prefilter is employed on the hardware to further filter 99% of seeds emitted by the first stage. HSPs emitted by the ungapped extension prefilter are then passed through NCBI BLASTN ungapped and gapped extension stages running on the host CPU. The probability of a database residue generating a seed that passes through the entire hardware pipeline is very small ($< 4.72 \times 10^{-7}$). Hence the downstream stages are able to keep up with the ingest rate, whilst running on the microprocessor.

The NCBI BLAST software code was modified to call the FPGA routines to perform the initial stages, while running latter stages in parallel with the hardware on the host CPU. This preserves the familiar BLAST interface for the end user, with the Mercury system running transparently. A speedup of 30x over the software-only deployment was targeted, with the seed generation stage running at 1.4Gbases/sec for 10k queries (or a 700MB/sec database ingest rate).

## 3.3 Mercury BLASTP: Overview

Mercury BLASTP is a hardware/software architecture consisting of substantial hardware and software components. As described earlier, all three stages of NCBI BLASTP must be accelerated in

order to achieve a substantial speedup. An overview of the Mercury BLASTP deployment is depicted in Figure 3.3.

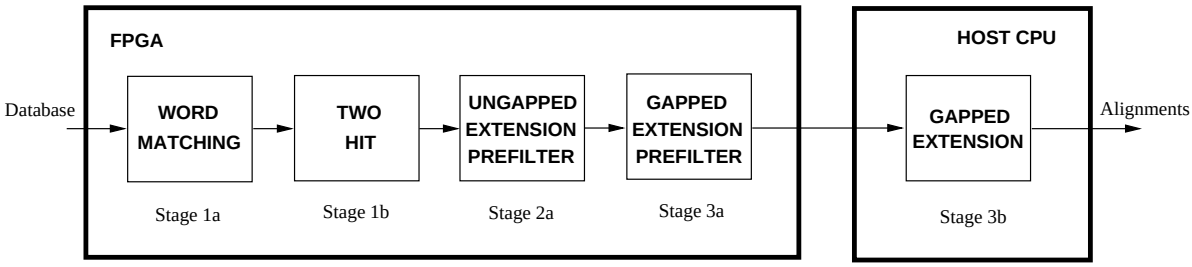



Figure 3.3: Mercury BLASTP: hardware/software deployment

The seed generation stage includes word matching and the two hit modules. A prefilter stage as in Mercury BLASTN is of little performance benefit, as the probability of a database w-mer producing a word match in the query's neighbourhood is high ($\approx 0.85$ for 2048 residue queries). Stage 1a accesses the query neighbourhood stored in off-chip SRAM(s) to generate w-mer matches. Stage 1b seeks pairs of word matches in close proximity, which indicates an alignment of interest. The second word match is passed as a seed into the ungapped extension phase.

Stage 2a of the pipeline is a dynamic programming filter that extends a seed within a fixed window of residues. Gaps are disallowed, so the computation can be implemented as a high throughput filter. Ungapped HSPs scoring above a threshold are passed to stage 3a, where the costly banded Smith-Waterman gapped extension computation is performed. This stage is a substitute for the X-drop gapped extension algorithm described in Chapter 1. While the software equivalent places no bounds on gapped extension, the hardware stage is limited to a fixed rectangular search space. Gapped HSPs scoring above a threshold and those crossing the search boundary are sent to the host CPU. This approach minimizes false negatives, but introduces a few false positives. The NCBI BLASTP software pipeline is resumed at the gapped extension phase (stage 3b), filtering out false positives and generating alignments of significant matches.

The seed generation hardware is described in detail in the next section. The next chapter deals with the software architecture and algorithms needed to support the hardware. In this section, we briefly summarize the design of the remaining hardware stages.

### 3.3.1 Ungapped Extension

The design and implementation of the Mercury BLASTP ungapped extension stage was done by another member of our research group, Joseph Lancaster, and is described in detail in [39]. The ungapped extension algorithm is a heuristic used to decide if a seed is likely to lead to a significant alignment. As the name implies, residue pairs are scored in the region surrounding the seed; gaps,

or residue insertion and deletions, are disallowed. This simplifies the general Smith-Waterman recurrence and can be computed efficiently.

The NCBI BLASTP ungapped extension X-drop algorithm described in chapter 1 is ill-suited for a hardware implementation. However, a decision on a significant fraction of input seeds can be made by an extension of a fixed window of residues on either side of the seed. This leads to an FPGA-friendly hardware implementation. The prefilter always scores left to right in a fixed window centered on the seed. The recurrence can modified to ensure that the HSP always passes through the seed. HSPs that score above the threshold are passed directly to the gapped extension stage.

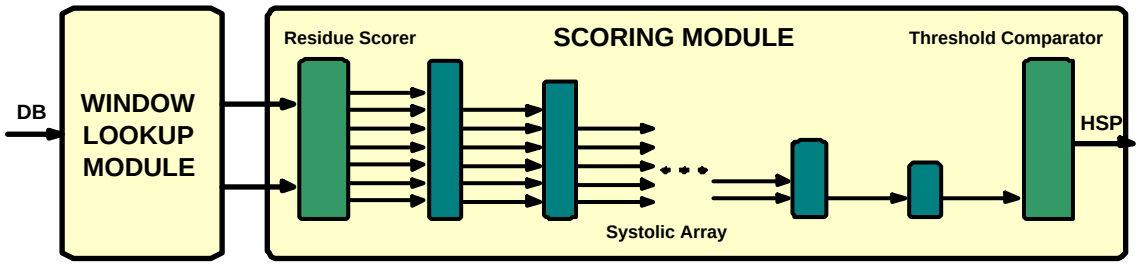



Figure 3.4: Ungapped extension prefilter hardware design

Figure 3.4 shows the design of the prefilter module. A window lookup module retrieves the query and database sequences in the prefilter window. The entire query is stored in block RAMs, while a circular buffer holds the required portion of the database. A scorer module compares corresponding residues of the query and database sequences, assigning a score based on a scoring table (BLOSUM62) stored in on-chip lookup tables. A systolic array of scoring stages implements the dynamic programming recurrence and emits the score of the highest-scoring HSP. Finally, a threshold comparator flags both HSPs that score above a threshold value and those that intersect the window bounds for passage to the next stage. The hardware module is fully pipelined and accepts one seed per clock.

The design of the ungapped extension prefilter has two major consequences for the seed generation hardware. Firstly, seeds passed into the prefilter must be approximately sorted by database position. As the database is streamed through the window lookup module, the circular buffer is advanced to accommodate incoming data, based on the database offset of the most recent seed. Hence there is no guarantee that a window of database residues will be available for seeds that appear out of order (or out of order by more than a bounded number of residues). Secondly, the seed generation hardware and the ungapped extension prefilter run in parallel, with little synchronization between them. In NCBI BLASTP, feedback from stage 2 to the two-hit module helps determine seeds that are part of a previously extended HSP, discarding them in the process. The effect on the workload of the prefilter in the absence of this information in hardware is studied in section 3.4.2.

### 3.3.2 Gapped Extension

We briefly describe work done by a member of our group, Brandon Harris, in accelerating the gapped extension phase of the BLASTP pipeline. Further detail is available in [31]. Gapped extension is the most computationally expensive stage of the BLASTP pipeline. The upstream prefilters are designed with the sole purpose of reducing the input rate into this phase. The NCBI X-drop gapped extension algorithm is unsuitable for a hardware implementation due to the lack of a fixed search boundary. We advocate the use of banded Smith-Waterman, a fixed-window alternative that is also used in WU-BLAST. This hardware stage acts as a filter on the input HSP stream. Gapped HSPs scoring above a threshold are passed to the host CPU, where the unbounded gapped extension is performed.

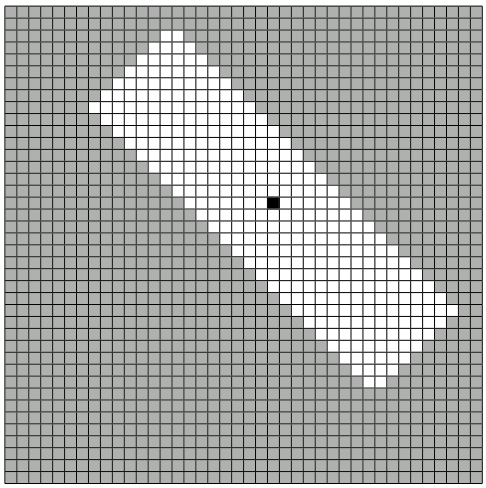


Figure 3.5: Banded Smith-Waterman: fixed-window gapped extension centered on a seed

As illustrated in figure 3.5, the search is limited to a fixed rectangular band centered around the seed. The dimensions of this band are selected according to the required sensitivity and specificity of the prefilter. A systolic array of cells simultaneously computes the score of an entire anti-diagonal in the rectangular window. The computation proceeds left to right until all anti-diagonals are scored, as the database is streamed through the array. The systolic array maintains the score of the best HSP passing through the seed, which is compared to the threshold at the end of the computation. Unlike ungapped extension, the Banded Smith-Waterman prefilter is a multi-cycle ($\approx 700$) stage.

Figure 3.6 shows the block diagram of the gapped extension prefilter. The entire query sequence is pre-loaded into on-chip block RAMs, while the database is streamed through a FIFO. During the fixed-width extension, residues from both sequences are shifted through a register set. Parallel taps from the query and database shift registers feed the systolic cell array. The score of each database and query residue pair for each cell is computed by a lookup into the scoring matrix stored in block RAMs. The Smith-Waterman computation is performed in each cell element, and the results are stored in the VEF register block. The stage 3 controller initiates the extension process when an

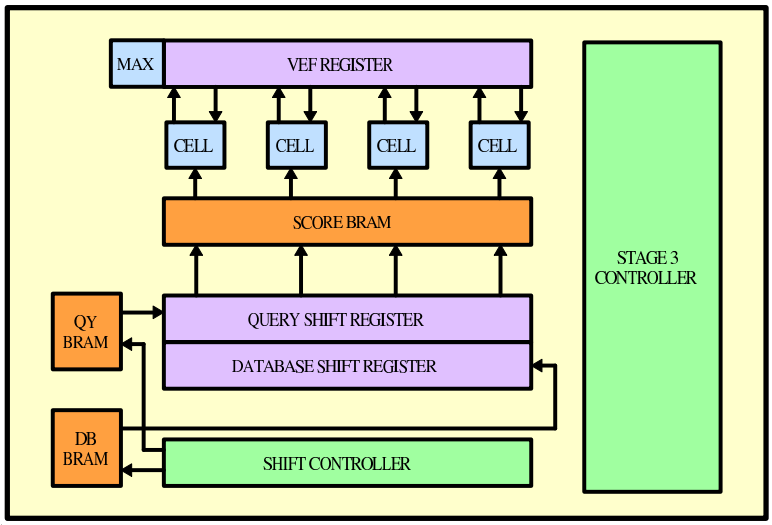



Figure 3.6: Gapped extension hardware

HSP and its corresponding database window is received from the ungapped extension prefilter. The maximum score in the systolic cell array is extracted and compared to the threshold upon completion of the extension process.

## 3.4 Seed Generation

The seed generation hardware closely reflects the functionality of NCBI BLASTP. In keeping with the goal of producing comparable results, the same heuristics are reproduced in hardware. It is important that the seed generation hardware process the database stream at a high throughput, since it is the bottleneck stage of the pipeline. The design decisions and the highly pipelined logic of the hardware are motivated by a high-speed implementation.

The designs of the word matching and the two-hit unit are described in detail in this section. We investigate approaches to replicating these modules for increased throughput and design an interconnecting switch. Finally, we discuss the current instantiation of Mercury BLASTP and the implementation parameters used.

### 3.4.1 Word Matching

Figure 3.7 shows the block diagram of the word matching module in hardware. It is divided into two logical components: the w-mer feeder and the hit generator. The w-mer feeder receives a database stream from the DMA engine and constructs fixed length words to be scanned against the query

neighbourhood. Twelve 5-bit database residues are accepted in each clock cycle by the w-mer control finite state machine. The output of this stage is a w-mer and its position in the database. The word length is defined by the user at compile time.

The w-mer creator block is a structural module that generates the w-mer at each database position. Simple modifications enable various word lengths, masks (discontiguous residue position taps), or even multiple w-mers based on different masks. Another function of the module is to flag invalid w-mers. While NCBI BLASTP supports an alphabet size of 24 (20 amino acids, 2 ambiguity characters and 2 control characters), Mercury BLASTP is restricted to 20. W-mers that contain residues not representing the twenty amino acids are flagged as invalid and discarded by the seed generation hardware. This stage is also capable of servicing multiple consumers in a single clock cycle. Up to $M$ consecutive w-mers can be routed to downstream sinks based on independent read signals. This functionality is important to support multiple parallel hit generator modules. Care is also taken to eliminate dead cycles; the w-mer feeder is capable of satisfying up to $M$ requests in every clock cycle.

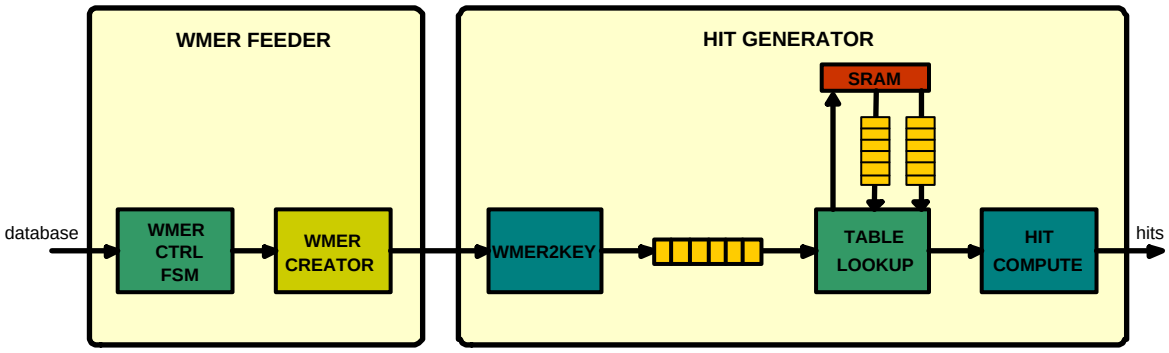
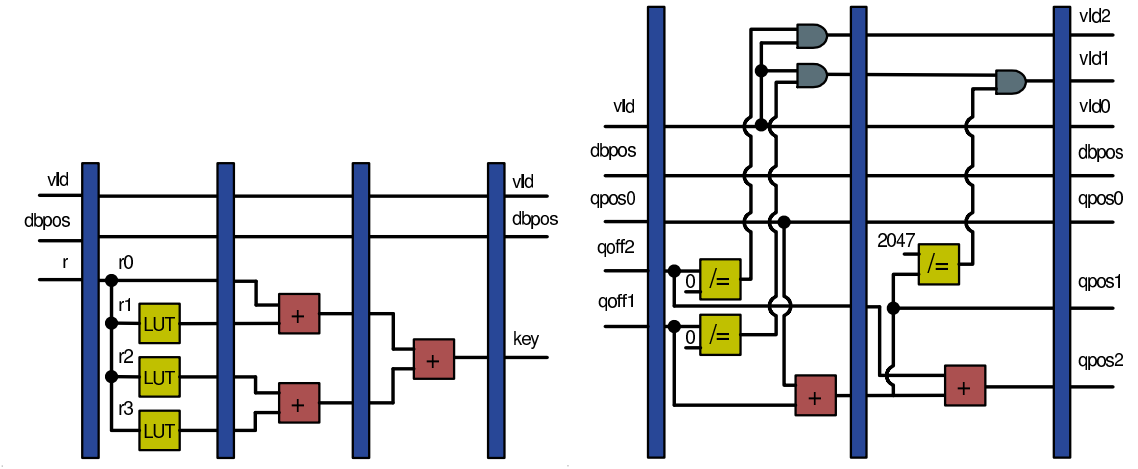



Figure 3.7: Word matching hardware design

The hit generator produces hits from an input w-mer by querying a lookup table stored in off-chip SRAM. The pipeline consists of the wmer2key, table lookup, and hit compute modules. A direct memory lookup table stores the position(s) in the query sequence to which every possible w-mer maps. As mentioned earlier, the twenty amino acids are represented using 5 bits. A direct mapping of a w-mer requires a prohibitively large lookup table with $32^w$ entries, out of which only $20^w$ specify valid w-mers. Therefore, a change of base is done in the wmer2key module as follows: $20^{w-1}r_{w-1} + 20^{w-2}r_{w-2} + ... + r_0$, where $r_i$ is the $i^{th}$ residue of the w-mer. For a fixed word length (which is set during compile time), this computation is easily realized in hardware.

Figure 3.8(a) shows the three-stage w-mer-to-key conversion for $w = 4$. A database w-mer $r$, at position *dbpos* is converted to the key in stages. Simple lookup tables are used in place of hardware multipliers (since the alphabet size is fixed) to multiply each residue in the w-mer. The result is aggregated using an adder tree.




(a) W-mer to key conversion module (b) Hit compute module

Figure 3.8: Word matching stages

The table lookup module generates hits for each database w-mer. The query neighbourhood is stored in off-chip SRAM, in a primary and duplicate table. We describe the implementation for a 32-bit addressable SRAM, storing query positions for a 2048-residue query sequence.

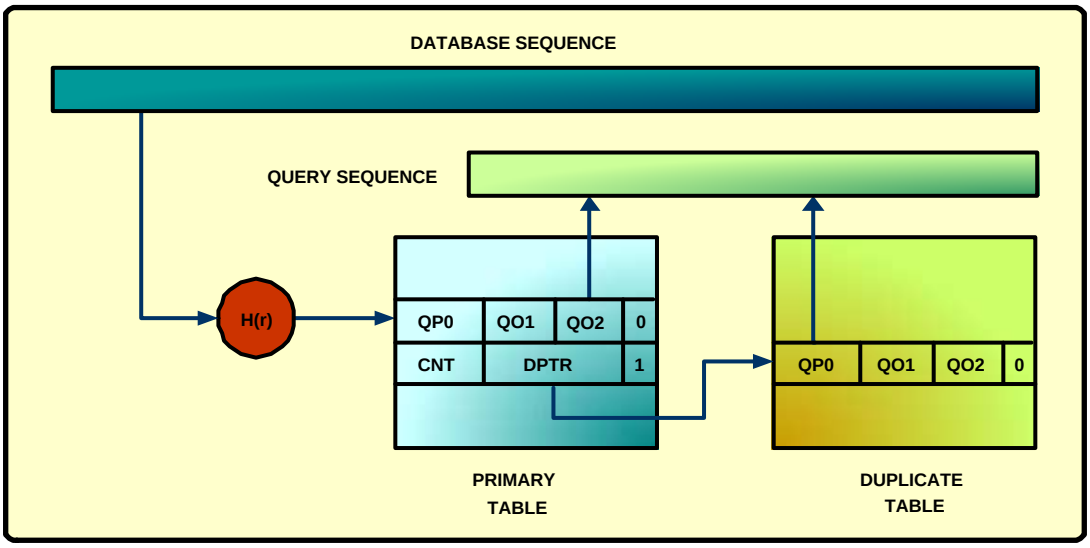



Figure 3.9: Lookup table datapath

The primary table is a direct memory lookup table containing $20^w$ 32-bit entries, one for every possible w-mer. Unlike for DNA matching, hashing is unsuitable for proteins since a large fraction of the key space is non-empty (Refer Table 2.9). Each primary table element stores up to three query positions that a w-mer maps to. Since a w-mer may map to more than three positions in the query, the primary table entry is extended to hold a duplicate bit. If set, the primary table element

does not store any query positions. Instead, the remaining bits hold the duplicate table pointer and an entry count value. Duplicates are stored in consecutive memory locations in the duplicate table, starting at the address indicated. The number of duplicates for each w-mer is limited by the size of the count field and the amount of off-chip memory available. Figure 3.9 illustrates the data path for a w-mer lookup.

**Algorithm 1** Encode query positions

```
1:  procedure ENCODE(n, qp_0, ..., qp_{n-1})
2:      if n = 0 then
3:          return ADD(2047, 0, 0)                          ▷ Non-matching w-mer
4:      else if n = 2 and (qp_1 − qp_0) = 1024 then
5:          qp_2 ← 2047; n ← 3                              ▷ Special case #2
6:      end if
7:
8:      s ← 0
9:      for i ← 1, n − 1 do
10:         if qp_i − qp_{i−1} ≥ 1024 then
11:             s ← i                                       ▷ Special case #1
12:         end if
13:     end for
14:     return ADD(qp_s, ..., qp_{n−1}, qp_0, ..., qp_{s−1})
15: end procedure
```

Lookups into the duplicate table reduce the throughput of the word matching stage. It is essential that such lookups are kept to a minimum, and that most w-mer lookups are satisfied by a single probe into the primary table. As noted in section 2.3.3, the word matching stage generates approximately two query positions per w-mer lookup, when used with the default parameters. To decrease the number of SRAM probes for each w-mer, the 11-bit query positions are packed three in each primary table entry. To achieve this packing in 31 bits, we use the following encoding scheme: the first query position is stored in the first 11 bits, followed by two unsigned 10-bit offset values, i.e. $(qp_0, qo_1, qo_2)$. The three query positions are decoded as follows: $qp_0$, $(qp_0 + qo_1)$ *mod* 2048, and $(qp_0 + qo_1 + qo_2)$ *mod* 2048. The result of each addition is an 11-bit query position.

The encoding of the query positions in the lookup table is done during the pre-processing step on the host CPU. There are two special cases that need to be handled. Firstly, for three or more sorted query positions, 10 bits are sufficient to represent the difference 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 of the table entry. 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, a dummy value of 2047 is introduced, after which the solution to the first case applies. For example, query positions 70 and 1094 are encoded as $(1094, 953, 71)$. Query position 2047 is recognized as a special case and ignored in the hit generation module. This can be done without loss of information since query w-mer positions only range from $[0 \ldots 2047 - w]$. The encoding process is summarized in Algorithm 1.

As a result of the encoding scheme used, query positions may be retrieved out of order by the word matching module. This, however, is of no consequence to the downstream stages, since the hits remain sorted by database position.

Table 3.2: SRAM access statistics in the word matching module, for a neighbourhood of $N(4, 13)$

| | **% of DB w-mers satisfied** | |
|---|---|---|
| **SRAM probes** | **Offset-encoded** | **Naive** |
| 1 | 82.6158 | 67.5121 |
| 2 | 82.6158 | 67.5121 |
| 3 | 98.0941 | 91.3216 |
| 4 | 99.8407 | 98.0941 |
| 5 | 99.9889 | 99.6233 |
| 6 | 100.0000 | 99.9347 |
| 7 | 100.0000 | 99.9889 |
| 8 | 100.0000 | 99.9985 |
| 9 | 100.0000 | 100.0000 |

Table 3.2 reveals the effect of the query encoding scheme on the SRAM access pattern in the word matching stage. The table displays the percentage $f_i$ of database w-mer lookups that are satisfied in $a_i$ or fewer probes into the SRAM. The data is averaged for a neighbourhood of $N(4, 13)$, over BLASTP searches of twenty 2048-residue query sequences compiled from the Escherichia coli k12 proteome, against the NR database. The most important observation is that 82% of the w-mer lookups can be satisfied in a single probe (returning up to three query positions) when using the encoded representation. The naive scheme would satisfy only 67% of lookups (returning up to two query positions), thus reducing the overall throughput. Note, in case that the duplicate bit is set, the first probe returns the duplicate table address (and zero query positions). All fifteen query positions are retrieved in 6 SRAM accesses when the encoding scheme is used; this increases to 9 otherwise.

Decoding the query positions in hardware is done in the hit compute module. The two stage pipeline is depicted in Figure 3.8(b) and the control logic in Algorithm 2. The circuit accepts a database position *dbpos*, a query position $qpos0$, and up to two query offsets $qoff1$ and $qoff2$. Two back-to-back adders generate $qp_2$ and $qp_3$. Each query offset represents a valid position if it is non-zero. Additionally, the dummy query position $qp_2 = 2047$ is discarded. The circuit outputs up to three hits at the same database position.

**Algorithm 2** Lookup table control logic

```
1: procedure LOOKUP(r, dp)                                   ▷ Lookup database w-mer r
2:     key ← 20^{w-1} r_{w-1} + 20^{w-2} r_{w-2} + ... + r_0   ▷ Base conversion
3:     entry ← sram[key]                                     ▷ Read primary table entry
4:     if d = 1 then                                         ▷ Duplicate bit set
5:         entry ← sram[dup_ptr]
6:         return dp, HITCOMPUTE(qp_0^0, qo_1^0, qo_2^0), ...  ▷ Return query positions from duplicate table
7:     else if qp_0 = 2047 then
8:         return NULL                                       ▷ Non-matching w-mer
9:     else
10:        return dp, HITCOMPUTE(qp_0, qo_1, qo_2)           ▷ Return query positions from primary table
11:    end if
12: end procedure
13:
14: procedure HITCOMPUTE(qp_0, qo_1, qo_2)                   ▷ Compute query positions from offsets
15:    qp_1 ← qp_0 + qo_1
16:    qp_2 ← qp_1 + qo_2
17:    if qo_1 = 0 or qp_1 = 2047 then
18:        qp_1 ← NULL                                       ▷ Second query position is invalid or dummy value
19:    end if
20:    if qo_2 = 0 then
21:        qp_2 ← NULL                                       ▷ Third query position is invalid
22:    end if
23:    return qp_0, qp_1, qp_2
24: end procedure
```

### 3.4.2 Two-hit

The diagonal of a hit $(q_i, d_i)$ is defined as $D_i = d_i - q_i$. Given two hits $(q_i, d_i)$ and $(q_j, d_j)$, the two hit algorithm generates a seed $(q_j, d_j)$ when $D_i = D_j$ and $d_j - d_i \geq w$, $d_j - d_i < A$, where $A$ is the window length. The algorithm can be efficiently implemented using a data structure to store the database positions of seeds encountered on each diagonal [49]. The diagonal array is implemented using on-chip block RAMs of size equal to $2M$, where $M$ is the size of the query sequence. As the database is scanned left to right, all diagonals $D_k < d_k - M$ are no longer used and may be discarded. $D_i$ indexes the array and wraps around to reuse memory locations corresponding to discarded diagonals. For a query size of 2048 and 32-bit database positions, the diagonal array can be implemented in eight block RAMs.

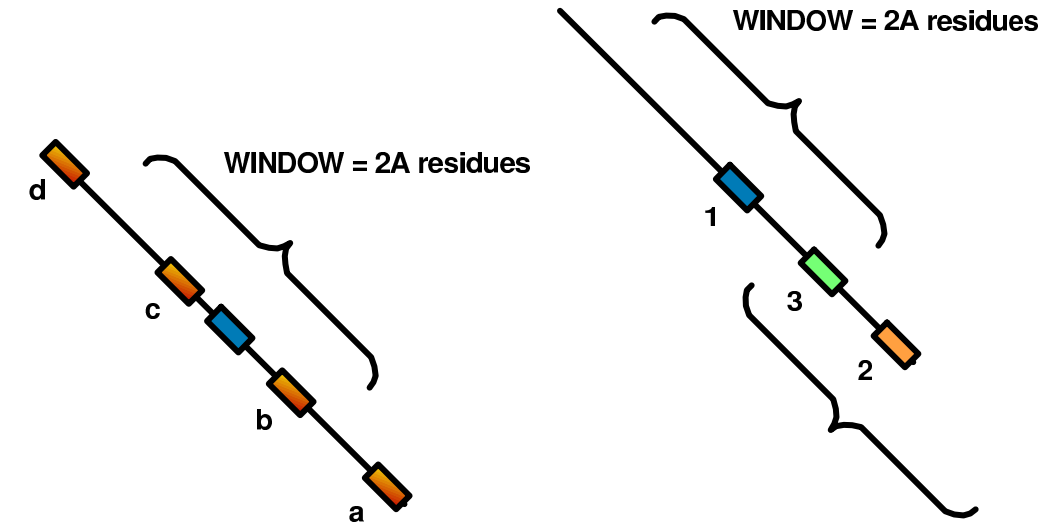
Algorithm 3 sketches the two-hit algorithm. Line 9 ensures that at least one word match has been encountered on the diagonal, before generating a seed. This is accomplished by checking for the initial zero value (database positions range from $1..N$). A valid seed is generated if the word match does not overlap and is within $A$ residues to the right of the last encountered w-mer (Line 10). Finally, the latest hit encountered is recorded in the diagonal array, on Line 5.

**Algorithm 3** Two-hit

```
1:  procedure TWO-HIT(q_i, d_i)
2:      D_i ← d_i − q_i                          ▷ Compute diagonal index: result is a signed value
3:      d_p ← DIAG_ARRAY[D_i]      ▷ Look up last encountered database position on the diagonal
4:
5:      if d_i − d_p >= w then                   ▷ Update diagonal if non-overlapping word match
6:          DIAG_ARRAY[D_i] ← d_i
7:      end if
8:                                               ▷ Condition to check for a valid two-hit seed
9:      if d_p ≠ 0 then                  ▷ At least one word match encountered on this diagonal
10:         if d_i − d_p >= w and d_i − d_p < A then   ▷ Non-overlapping w-mer within right window
11:             return q_i, d_i
12:         else if d_i − d_p < −A then          ▷ Out-of-order w-mer by more than A residues
13:             return q_i, d_i
14:         end if
15:     end if
16: end procedure
```

As will be detailed in the next section, the two-hit module must be capable of handling hits received out of order, without an appreciable loss in sensitivity or increase in workload of downstream stages. Our heuristic (Line 12) is to do one of the following: if the hit is within $A$ residues to the left of the last recorded hit, discard it; else, forward it to the next stage. In the former case, the out-of-order hit is likely part of an HSP that was already inspected – assuming the last recorded hit was passed for ungapped extension – and can be safely ignored. In practice, for $A = 40$, most out-of-order hits fall into this category (due to the design and implementation parameters).

Figure 3.10(a) shows the choices for two-hit computation on a single diagonal, upon the arrival of a second hit. If it is within the right window (hit $b$), it is forwarded to the next stage; if instead it is beyond $A$ residues (hit $a$), it is discarded. An out-of-order hit (hit $c$) within the left window is discarded, while hit $d$, which is beyond $A$ residues, is passed on for ungapped extension. The heuristic to handle out-of-order hits may lead to false negatives. Figure 3.10(b) illustrates this point, showing three hits numbered in their order of arrival. When hit 2 arrives, it is beyond the right window of hit 1 and is discarded. Similarly, hit 3 is found to be in the left window of hit 2 and discarded. A correct implementation would forward both hits 2 and 3 for extension. The heuristic,




(a) Two-hit algorithm choices (b) Loss in sensitivity due to an out-of-order hit

Figure 3.10: Two-hit computation performed in a diagonal

though not perfect, handles out-of-order hits without increasing the workload of downstream stages. The effect on sensitivity was empirically determined to be negligible.

Figure 3.11 illustrates the two-hit module. A input hit $(dbpos, qpos)$ is passed in along with its corresponding diagonal index, $diag\_idx$. The hit is checked in the two-hit logic, and sent downstream (i.e. $vld$ is high) if it passes. The two-hit logic is pipelined into three stages to enable a high-speed design. This increases the complexity of the design since data has to be forwarded from the later stages. The Diagonal Read stage performs a lookup into the block RAM using the computed diagonal index. The read operation uses the second port of the block RAM and has a latency of one clock cycle. The first port is used to update a diagonal with the last encountered hit in the Diagonal Update stage. A write collision condition is detected upon a simultaneous read/write to the same diagonal, and the most recent hit is forwarded to the next stage. The second stage performs the two-hit check and implements the three conditions discussed. The most recent hit in a diagonal is selected from one of three cases: a hit from the previous clock cycle (forwarded from the Diagonal Update stage), a hit from the last but one clock cycle (detected by the write collision check), or the value read from the block RAM. The two-hit condition checks are decomposed into two stages to decrease the length of the critical path, e.g: $d_i - d_p < A$ becomes $tmp = d_i - A$ and $tmp < d_p$. A seed is generated when the requisite conditions are satisfied.

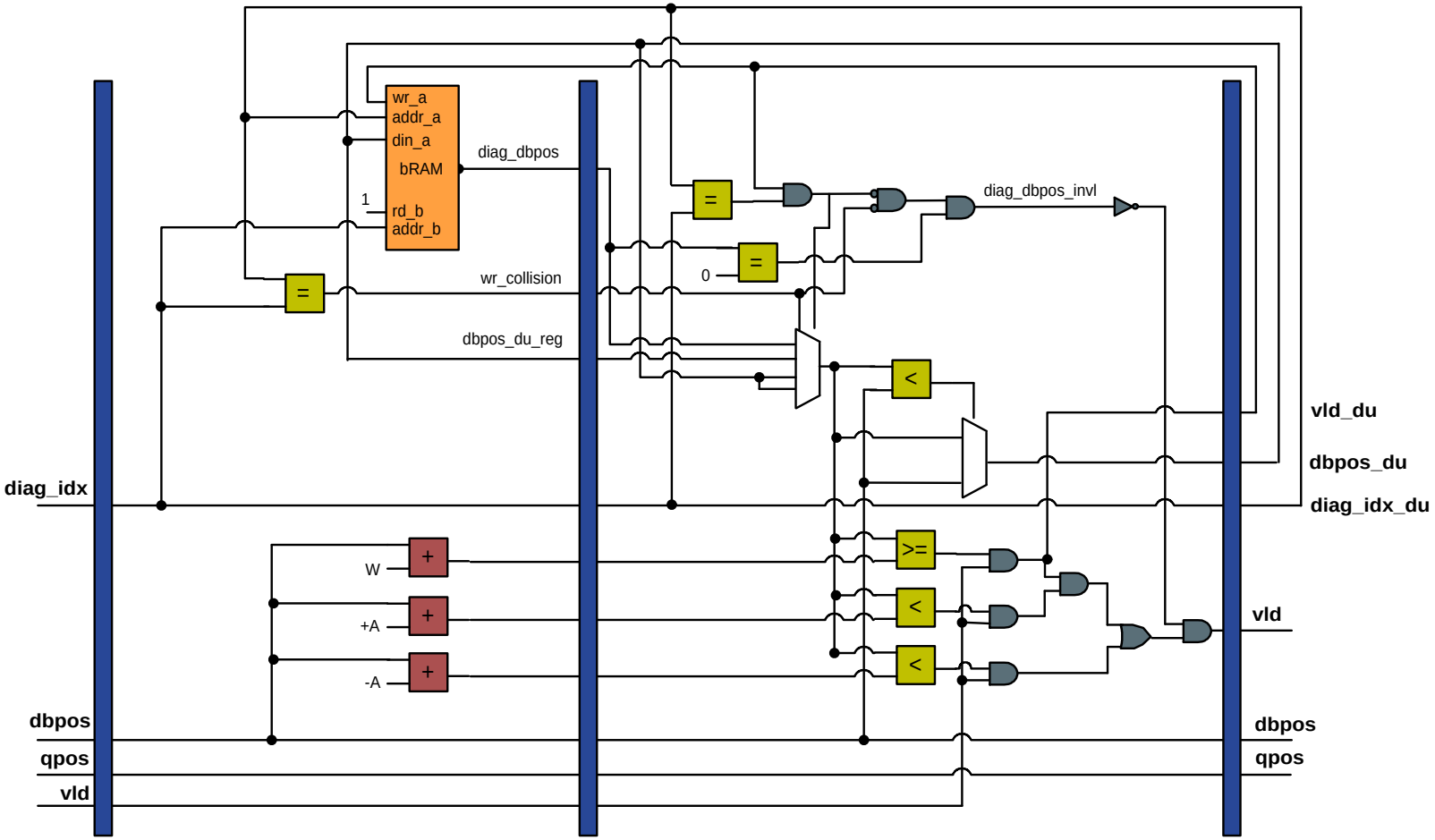



Figure 3.11: Two-hit module

NCBI BLASTP employs a redundancy filter to discard seeds present in the vicinity of HSPs inspected in the ungapped extension stage. The furthest database position examined after extension is recorded in a structure similar to the diagonal array. In addition to passing the two-hit check, a hit must be non-overlapping with this region to be forwarded to the next stage. This feedback characteristics of the filter make it a challenging candidate for hardware implementation.

Table 3.3: Increase in seed generation rate without feedback from NCBI BLASTP stage 2

| **Query Length** (*residues*) | **N(w, T)** | **Rate increase** (%) |
|---|---|---|
| 2000 | $N(3, 11)$ | 0.2191 |
| 2000 | $N(4, 13)$ | 0.2246 |
| 2000 | $N(5, 14)$ | 0.2784 |
| 3000 | $N(3, 11)$ | 0.2222 |
| 3000 | $N(4, 13)$ | 0.2205 |
| 3000 | $N(5, 14)$ | 0.2743 |
| 4000 | $N(3, 11)$ | 0.2359 |
| 4000 | $N(4, 13)$ | 0.2838 |
| 4000 | $N(5, 14)$ | 0.3956 |

We measured the effect of the lack of the NCBI BLASTP extension feedback on the seed generation rate of the first stage. Table 3.3 shows the increased seed generation rate for various query sizes and neighbourhoods. The experimental setup is similar to that described in the previous chapter. The data suggests a modest increase in workload for ungapped extension, of less than a quarter of one percent. The reason for this is that the two-hit algorithm is already an excellent filter, approximately performing the role of the redundancy filter. We conclude that feedback from stage 2 has little effect on system throughput and choose to ignore it in our design.

### 3.4.3 Two-hit replication

As noted in section 2.3.3, the word matching stage generates hits at the rate of approximately two per database residue for a neighbourhood of $N(4, 13)$. The two-hit module, with the capacity to process only a single hit per clock cycle, becomes the bottleneck in the pipeline. Processing multiple hits, however, poses a substantial challenge due to the physical constraints of the implementation. Concurrent access to the diagonal array is limited by the dual-ported block RAMs on the FPGA. Since one port is used to read a diagonal and the other to update it, no more than one hit can be processed in the two-hit module at a time. In order to address this issue, the two-hit logic must be replicated, with hits being evenly distributed among the copies. A straightforward replication of the entire diagonal array requires that all copies be kept coherent, leading to a multi-cycle update phase and a corresponding loss in throughput. Efforts to time-multiplex access to block RAMs (for example, quad-porting by running them at twice the clock speed of the two-hit logic) proved impractical because the two-hit logic already runs at a high clock speed.

An important observation is that the two-hit computation for a w-mer is performed on a single diagonal and is independent of the values of all other diagonals. Rather than replicating the entire diagonal array, the diagonals can instead be evenly divided among $b$ two-hit modules. A hit $(q_i, d_i)$ is processed by the $j^{th}$ two-hit copy if $D_i \bmod b = j - 1$. This modulo division scheme also increases the probability of equal work distribution between the $b$ copies. Hits generated by the word matching phase tend to be clustered around a few high scoring residues; hence, an alternate division of the diagonal array into $b$ bands leads to an uneven partitioning of hits (Figure 3.12). The routing of a hit to its two-hit unit is also simplified. If $b$ is a power of two, i.e. $2^t$, the lower $t$ bits of $D_i$ act as the two-hit identifier. If not, the modulo operation is precomputed for all possible $D_i$ values and stored in on-chip lookup tables.

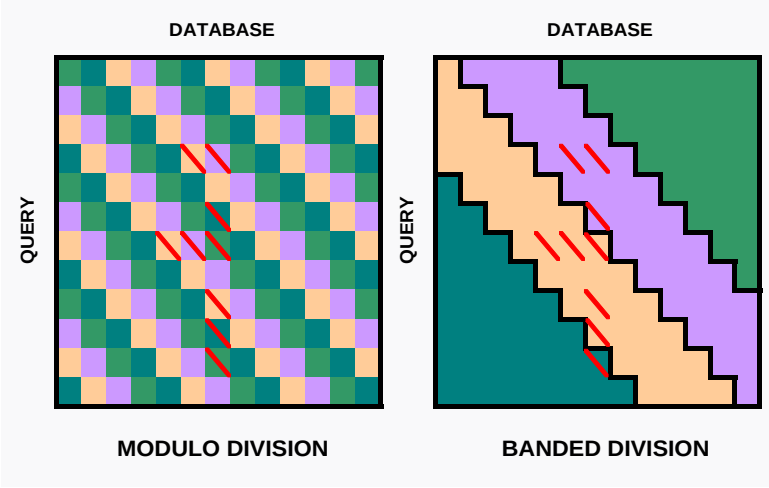



Figure 3.12: Two-hit work distribution: modulo division of diagonals achieves a more equal distribution of clustered hits than banded division

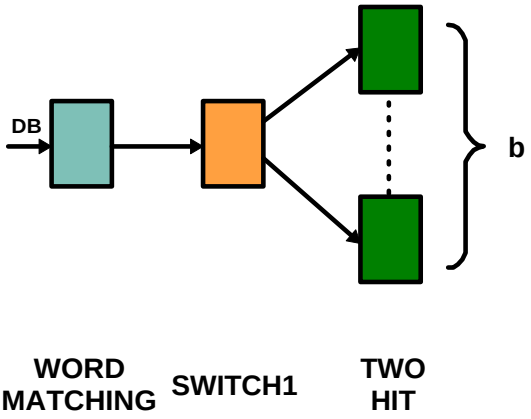



Figure 3.13: Two-hit replication: hits from a single word matching module are routed to $b$ two-hit modules

**Interconnecting Switch 1**

An important part of the design is the switch required to route hits from the word matching stage to the two-hit units. Figure 3.14 displays the hardware design of the $3 \times b$ interconnecting switch (Switch 1) between a single word matching stage and $b = 2$ two-hit modules. The word matching module generates up to three hits per clock cycle ($dbpos, qpos0, diag\_idx0, vld0$, ...), which are stored in a single entry of an interconnecting FIFO. All hits in a FIFO entry share the same database position and must be routed to their appropriate two-hit units before the next triple can be processed. The routing decision is made independently, in parallel, and locally at each two-hit unit. Hits sent to the two-hit modules are ($dbpos0, qpos0$) and ($dbpos1, qpos1$).

A decoder for each hit examines $t = 1$ low-order bits of the diagonal index. The decoded signal is passed to a priority encoder at each two-hit unit to select one of the three hits. In case of a collision, priority is given to the higher-ordered hit. Information on whether a hit has been routed is stored in a register ($routed0/1/2$ in the figure) and is used to deselect a hit that has already been sent to its two-hit unit. This is decided by examining if the hit is valid, is being routed to a two-hit unit that is not busy, or has already been routed previously. The *read* signal is asserted once the entire triple has been routed. Each two-hit unit always accepts at least one available hit every clock cycle. With the word matching module generating two hits on average per clock cycle, $b = 2$ two-hit modules are sufficient to eliminate the bottleneck from this phase.

### 3.4.4 Hit generator replication

With downstream stages capable of handling the seed generation rate of the first stage, the throughput of Mercury BLASTP is limited by the word matching phase. The design is constrained by the lookup into off-chip SRAM. A logical solution to speed up the pipeline is to run multiple hit generation modules in parallel, each accessing an independent off-chip SRAM resource with its own copy of the tables. Adjacent database w-mers are distributed by the feeder stage to each of $h$ hit generation modules. Hits generated by each copy are sent to the two-hit modules. The number of two-hit modules is increased to keep up with the larger hit generation rate.

The use of $h$ independent modules has an unintended consequence on the generated hit stream. The w-mer processing time is variable due to the possibility of duplicate query positions. This characteristic causes the lookup stages to lose synchronization and generate hits that are out of order with respect to the database positions. Out-of-order hits may be discarded in the hardware stages. This however, leads to decreased search sensitivity. Alternatively, hits that are out of order by more than a fixed window of database residues in the extension stages may be forwarded to the host CPU without inspection. This increases the false positive rate and has an adverse effect on the throughput of the pipeline.

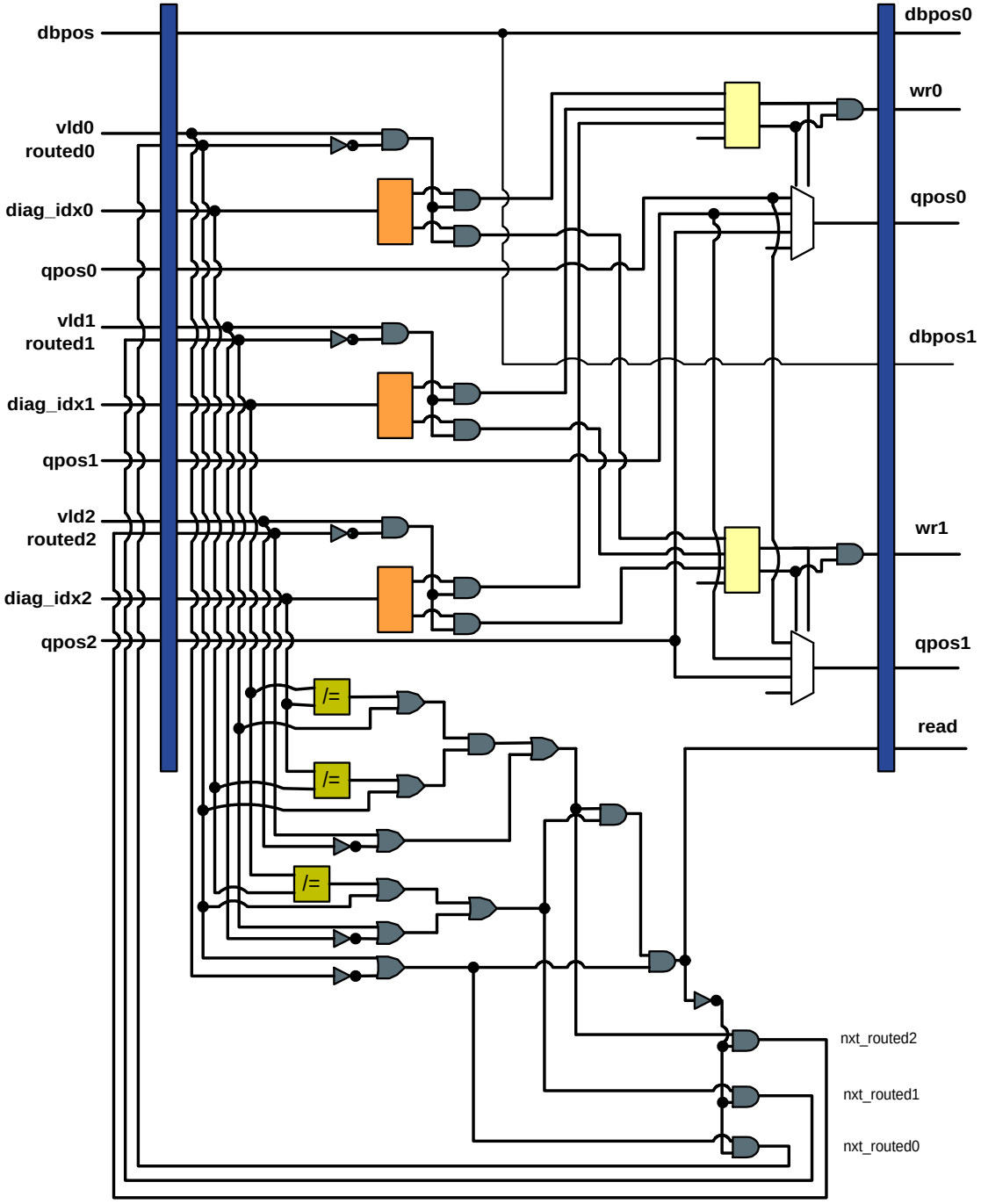



Figure 3.14: Switch1

This problem may be tackled in one of three ways. First, the $h$ modules may be deliberately kept synchronized. On encountering a duplicate every module stalls until all duplicates are retrieved, before the next set of w-mers is accepted. This approach quickly degrades in performance: as $h$ grows, the probability of the modules stalling increases, and the throughput decreases drastically. Another approach is to stall the modules only if they get out of order by more than the downstream tolerance. We advocate a slightly different solution. The number of duplicates for each w-mer in the lookup table is limited to $L$, requiring a maximum processing time of $l = \lceil L/3 \rceil$ clock cycles in our implementation. This automatically limits the distance the hits can get out of order in the worst case to $(d_t + l) \times (h - 1)$ database residues, without the use of additional hardware circuitry. Here, $d_t$ is the latency of access into the duplicate table. The downstream stages can then be designed for this out-of-order tolerance level. In our implementation $d_t = 4$ and $L = 15$. The loss in sensitivity due to the pruning of hits outside this window was experimentally determined to be negligible.

**Interconnecting Switch 2**

With the addition of multiple hit generation modules, additional switching circuitry is required to route all $h$ hit triples to their corresponding two-hit modules. This is done in two phases. Firstly, a triple from each hit generation module is routed to $b$ queues (one for each copy of the two-hit unit), using the interconnecting switch 1. A total of $h \times b$ queues, each containing a single hit per entry, are generated. Finally, a new interconnecting switch (2) is required at each two-hit unit to select hits from one of $h$ queues. This two-phase switching mechanism successfully routes any one of $3 \times h$ hits generated by the word matching stage to any one of the $b$ two-hit units.

Figure 3.15 shows the single stage hardware design of switch 2 with $h = 4$. Hits ($dbpos0, qpos0$, ...) each with a valid signal, must be routed to a single output port ($dbpos\_out, qpos\_out$). The data reduction circuit is designed to not introduce out-of-order hits. Parallel comparators ($\frac{h \times (h-1)}{2}$ in number) inspect the first element of all $h$ queues to detect the hit at the lowest database position. This hit is then passed directly to the two-hit module and cleared from the input queue.

Figure 3.16 illustrates the final architecture of the Mercury BLASTP seed generation hardware. The w-mer feeder block accepts the database stream from the host CPU, generating up to $h$ w-mers per clock. Hit triples from the hit generator modules are routed to one of $b$ queues in each of the $h$ switch 1 circuits. Switch 2 then reduces data from $h$ input streams and feeds the two-hit units.

The final piece of the design is the seed reduction module. Seeds generated from $b$ copies of the two-hit units are reduced to a single stream and forwarded to the ungapped extension phase. An attempt is again made to maintain order. The hardware circuit is identical to switch 2, except that a reduction tree is used. For a large number of input queues ($> 4$), the single-stage design described earlier has difficulty routing at high clock speeds. For $b = 8$, the reduction is performed in two stages: two 4-to-1 followed by a single 2-to-1.

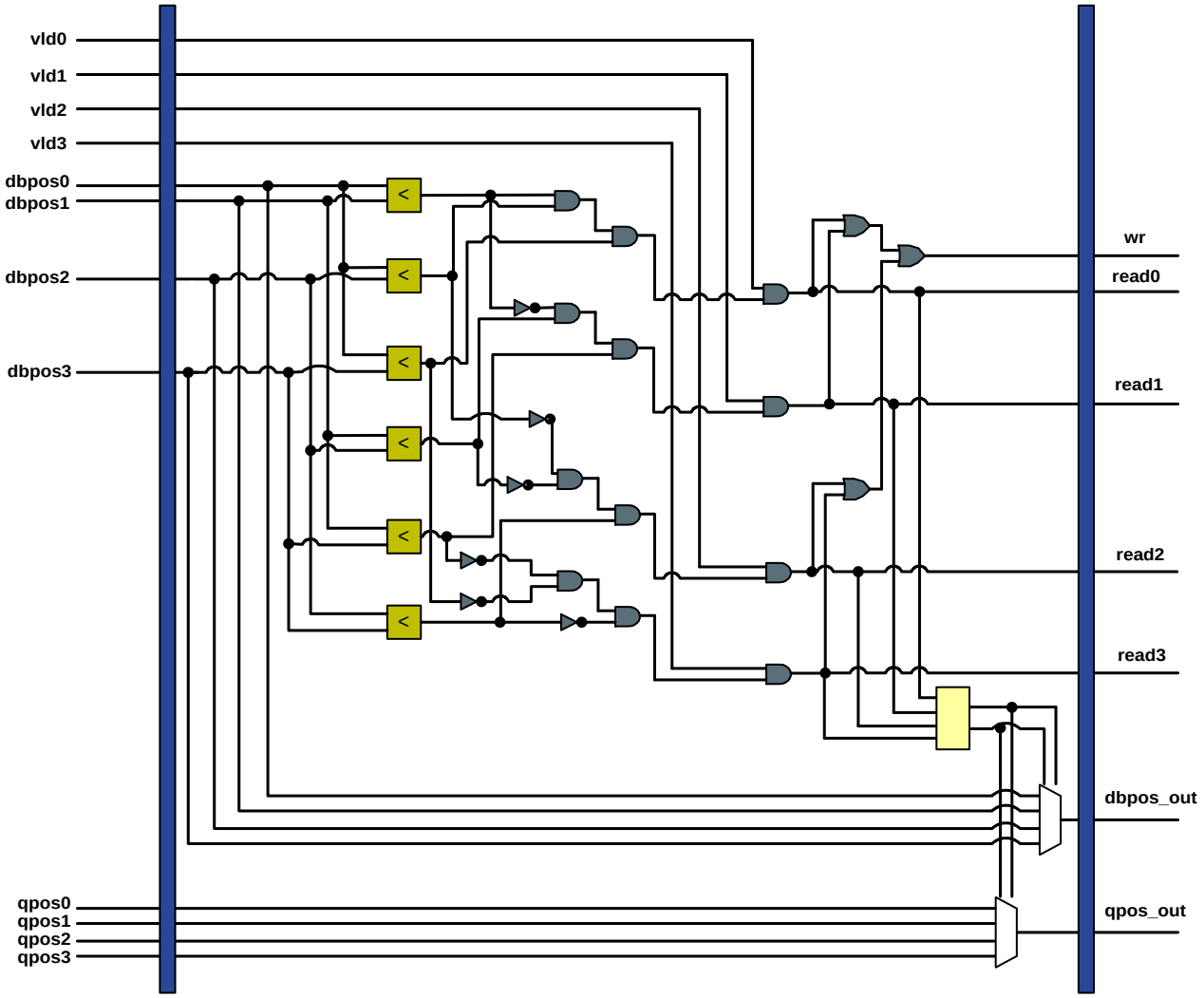



Figure 3.15: Switch2

We also note that seed reduction is not required to operate as fast as the rest of the design, since the two-hit stage generates seeds at less than one per clock cycle.

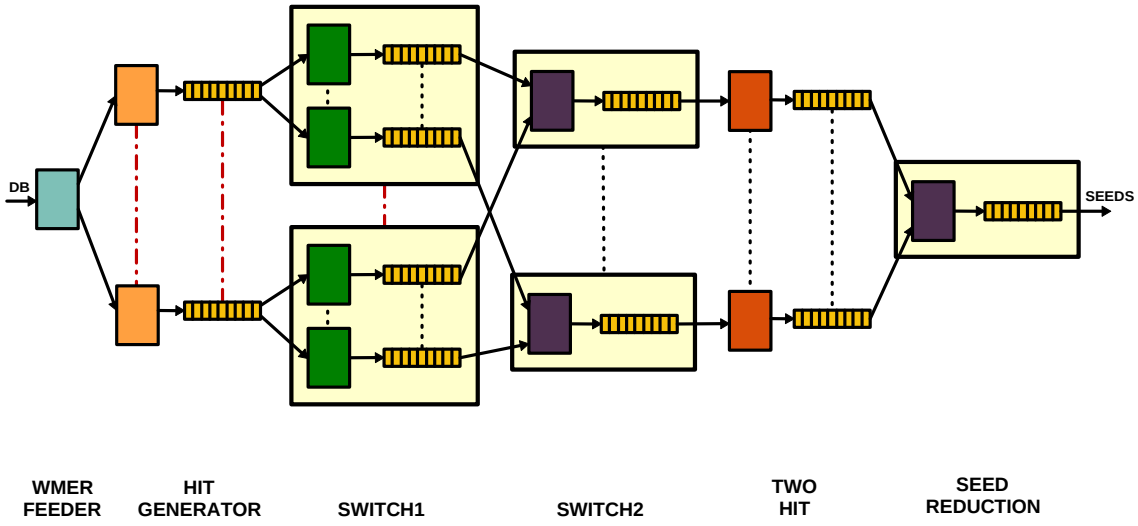



Figure 3.16: Seed Generator Hardware Design

## 3.5 Mercury BLASTP deployment

We conclude this chapter by detailing the instantiation parameters of Mercury BLASTP used in our implementation. The seed generation stage supports a query sequence of up to 2048 residues, and uses a neighbourhood of $N(4, 13)$. A database sequence of up to $2^{32}$ residues is supported. We use $h = 3$ copies of the hit generation module, and $b = 8$ parallel copies of the two-hit unit.

A dual-FPGA solution is used in our implementation of Mercury BLASTP, with seed generation and ungapped extension on the first FPGA and gapped extension running on the second. The database sequence is streamed from the host CPU to the first card. HSPs generated after ungapped extension are sent back to the host CPU, where they are interleaved with the database sequence and resent to the gapped extension stage. Significant hits are finally sent back to the host CPU to resume the software pipeline.

Modules in the Mercury system communicate via separate 64-bit data and control buses. Module-specific commands program the lookup table and clear the diagonal array in the two-hit modules. The seed generation and ungapped extension modules communicate via two independent data paths. The standard data communication channel is used to send seed hits, while a new bus is used to stream the database sequence. All modules respect backpressure signals asserted to halt an upstream stage when busy.

# Chapter 4

# Mercury BLASTP software architecture

Mercury BLASTP requires tight co-ordination between the FPGA resource and software running on the host CPU. In the following chapter, we describe the software infrastructure supporting the Mercury BLASTP system. We give an overview of the architecture and discuss the salient algorithms in detail. Finally, we briefly summarize the FPGA communication interface and describe the modifications undertaken to the NCBI BLASTP software pipeline.

## 4.1 Architectural overview

The software is organized as a multi-threaded application consisting of independently executing components communicating via queues. Figure 4.1 illustrates the hardware/software architecture of Mercury BLASTP. The software code falls into three categories: Mercury BLASTP support routines, FPGA interface code, and the NCBI BLAST software. The first category comprises code written specifically to populate data structures, such as the word matching lookup table, used in the hardware. The FPGA interface code uses the Exegy API to perform low-level communication tasks with the Mercury modules. 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 stage. 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.

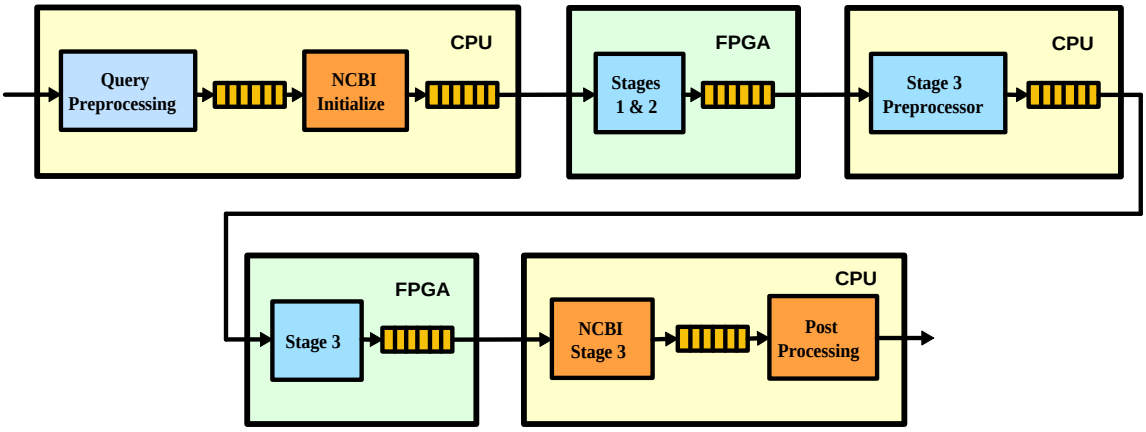



Figure 4.1: Software architecture

Query pre-processing involves preparing the necessary data structures required by the Mercury BLASTP hardware and the NCBI BLAST software pipeline. The input query sequences are first examined to mask low complexity regions (short repeats, or residues that are over-represented), which would otherwise generate statistically significant but biologically spurious alignments. SEG filtering replaces residues contained in low complexity regions with the "invalid" control character. The query sequence is packed, 5 bits per base in 60-bit words, and encoded in big-endian format for use by the hardware. Three main operations are then performed on the input query sequence set. Query bin packing concatenates smaller sequences to generate composites of the optimal size for the hardware. The neighbourhood of all w-mers in the packed sequence is generated and lookup tables created for use in the word matching stage. It is essential that query sequences are pre-processed at a high enough rate to prevent starvation of the hardware stages. We study these algorithms in detail in the next section.

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. Finally, the database is streamed through the FPGA device. The BLAST sequence database files are augmented to include a version encoded specifically for Mercury BLASTP. The ingest rate into the card is modulated by a backpressure signal propagated backwards from the hardware modules.

The FPGA device on the first card executes the first two stages of the BLASTP pipeline. HSPs generated by ungapped extension are sent back to the host CPU, where they are multiplexed with the database stream. Banded gapped extension on the second card consumes the stream to generate significant HSPs. The NCBI BLASTP pipeline is resumed on the host CPU at the X-drop gapped extension stage, and alignments are generated after post-processing.

The FPGA communication wrappers, the Mercury device driver, and the hardware DMA engine were provided by Exegy Inc.

## 4.2 Neighbourhood generation

A substantial portion of the pre-processing time in the BLASTP pipeline is spent generating the neighbourhood of a query. A naive algorithm, similar to the one used in NCBI BLASTP, scores 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 neighbourhood. The lookup table for the word matching module is generated from the neighbourhood using the encoding scheme specified in the previous chapter.

The NCBI BLASTP implementation is both memory-intensive and computationally expensive, degrading exponentially at longer word lengths. We 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 neighbourhood. 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 neighbourhood generation.

### 4.2.1 Prune-and-search neighbourhood

Given a query w-mer $r$, an alphabet $\Sigma$, and a scoring matrix $\delta$, the neighbourhood of the w-mer is computed using the following recurrence. The neighbourhood $N(w,T)$ of the query $Q$ is the union of the individual neighbourhoods of every query w-mer $r \,\epsilon\, Q$.

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

$$G^r(x,w,T) = \bigcup_{a\,\epsilon\,\Sigma} \left\{ \begin{array}{ll} \{xa\} & \text{if } |x| = w-1 \text{ and } S^r(x) + \delta_{r_w,a} \geq T, \\ G^r(xa,w,T) & \text{if } |x| < w-1 \text{ and } S^r(x) + \delta_{r_{|x|+1},a} + C^r(|x|+1) \geq T, \\ \phi & \text{otherwise.} \end{array} \right\}$$

$$S^r(x) = \left\{ \begin{array}{ll} 0 & \text{if } x = \epsilon, \\ S^r(y) + \delta_{r_{|x|},a} & \text{otherwise, where } x = ya. \end{array} \right\}$$

$$C^r(i) = \left\{ \begin{array}{ll} \max_{a \,\epsilon\, \Sigma} \delta_{r_w,a} & \text{if } i = w-1, \\ \max_{a \,\epsilon\, \Sigma} \delta_{r_i,a} + C^r(i+1) & \text{otherwise.} \end{array} \right\}$$

$G^r(x, w, T)$ 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 $G^r(\epsilon, w, T)$. At each step of the recurrence, the prefix $x$ is extended by one character $a \,\epsilon\, \Sigma$. The pruning process is invoked at this stage. If it can be determined that no w-mers with a prefix $xa$ exist in the neighbourhood, 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 neighbourhood 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 $S^r$, $\delta$, 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 $G^r(\epsilon, w, T)$ is shown in Algorithm 4. The algorithm does a depth-first search of the neighbourhood, extending a partial w-mer by every character in the alphabet. We define $\Sigma_b^{'}$ to be the alphabet sorted in descending order of the pairwise score against character $b$ in $\delta$. The w-mer extension is done in this order, causing the contribution of the $\delta$ lookup in the left-hand side of the expression on line 12 to progressively diminish with every iteration. Hence, as soon as a partition is pruned, further extension by the remaining characters in the list can be halted.

As partial w-mers are extended, a larger number of partitions are discarded. The fraction of the neighbourhood discarded at each step depends on the scoring matrix $\delta$ and the threshold $T$. While in the worst case the algorithm takes exponential time in $w$, in practice the choice of the parameters leads to a significant improvement in speed over naive enumeration.

**Algorithm 4** Stack implementation of the prune-and-search algorithm

```
1: procedure PS-NEIGH(w, T, r)          ▷ Generate neighbourhood N(w, T) for query w-mer r
2:     G ← φ                                                  ▷ Initialize neighbourhood 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 neighbourhood: 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
```

### 4.2.2 Vector implementation

To further improve performance of the implementation, we take advantage of vector instructions available on modern microprocessors. Single Instruction, Multiple Data (SIMD) instructions exploit data parallelism in algorithms by performing the same operation on multiple data values. The instruction set architectures of modern general purpose processors are augmented with SIMD instructions that offer increasingly complex functionality. Existing extensions include SSE2 [27] on x86 architectures and AltiVec [28] on PowerPC cores.

Sample SIMD instructions are illustrated in Figure 4.2. The vector addition of four signed 8-bit operand pairs is done in a single clock cycle, decreasing the execution time to one-fourth. The number of data values in the SIMD register (*Vector Size*) and their precision are implementation-dependent. The *Cmpgt-Get-Mask* instruction checks to see if signed data values in the first vector are greater than those in the second. This operation is performed in two steps. First, a result value of all ones if the condition is satisfied or zero otherwise is created. Second, a sign extended mask is formed from the most significant bits of the individual data values. The mask is returned in an integer register that must be inspected sequentially to determine the result of the compare operation.

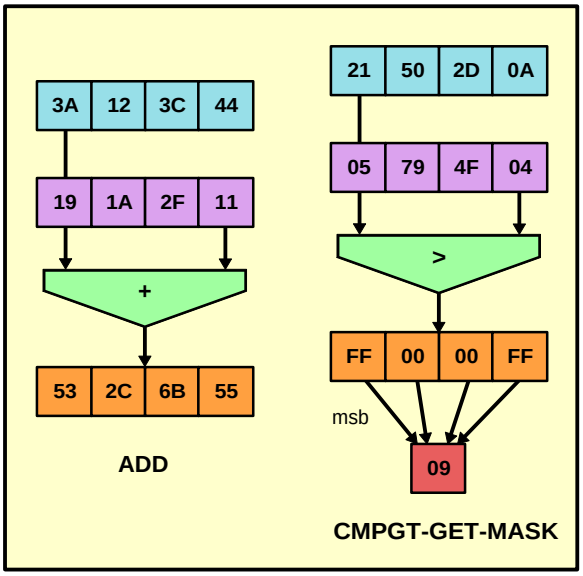



Figure 4.2: Single Instruction Multiple Data (SIMD) operations

Prune-and-search algorithms partition a search problem into a number of subinstances that are independent of each other. In our case, the extension of a partial w-mer by every character in the alphabet can be done independently of each other. We exploit the resultant data parallelism by vectorizing the computation in the *for* loop of Algorithm 4.

Algorithm 5 shows the vector implementation of the prune-and-search algorithm. As in the sequential version, each partial w-mer is extended by every character in the alphabet. However, each iteration of the loop performs $VECTOR\_SIZE$ such simultaneous extensions. As noted previously, a sorted alphabet list is used for extension. The sequential add operation is replaced by the vector equivalent, *Vector-Add*. Lines 21-27 perform the comparison operation and inspect the result. The returned mask value is shifted right, and the least significant bit is inspected to determine the result of the comparison operation for each operand pair. Appropriate actions are executed according to this result. The lack of parallelism in statements 22-27 results in sequential code.

We use SSE2 extensions available on the host CPU for our implementation. A vector size of 16 and signed 8-bit integer data values were used. The precision afforded by this implementation is sufficient for use with typical parameters without overflow or underflow exceptions. Saturated signed arithmetic is used to detect overflow/underflow and clamp the result to the largest/smallest value. The alphabet size is increased to the nearest multiple of 16 by introducing dummy characters, and the scoring matrix is extended accordingly.

**Algorithm 5** Vector implementation of the prune-and-search algorithm

1: **procedure** PS-NEIGH-VECTOR($w, T, r$)
2: ▷ Generate neighbourhood $N(w,T)$ for query w-mer $r$ using vector instructions
3: $\vec{T} \leftarrow \{T-1, \ldots, T-1\}$ ▷ Initialize threshold vector
4: $G \leftarrow \phi$
5: STACK.PUSH($\epsilon$)
6:
7: **repeat**
8: $x \leftarrow$ STACK.POP( )
9: $\Sigma^{''} \leftarrow \Sigma^{'}_{r_{|x|+1}}$ ▷ Retrieve sorted alphabet list for this w-mer residue
10: $\delta^{''} \leftarrow \delta^{'}_{r_{|x|+1}}$ ▷ Retrieve corresponding pairwise scores
11: $\vec{S} \leftarrow \{S^r(x), \ldots, S^r(x)\}$
12: $\vec{C} \leftarrow \{C^r_{|x|+2}, \ldots, C^r_{|x|+2}\}$
13: $\vec{P} \leftarrow$ VECTOR-ADD($\vec{S}, \vec{C}$) ▷ Precompute loop invariant vector
14:
15: ▷ Cycle through alphabet, VECTOR_SIZE characters per iteration
16: **for** $i \leftarrow 1, |\Sigma|, VECTOR\_SIZE$ **do**
17: $\vec{\delta} \leftarrow \{\delta^{''}_{\Sigma^{''}_i}, \ldots, \delta^{''}_{\Sigma^{''}_{i+VECTOR\_SIZE}}\}$ ▷ Initialize score vector
18:
19: **if** $|x| = w-1$ **then** ▷ Base case
20: $\vec{A} \leftarrow$ VECTOR-ADD($\vec{S}, \vec{\delta}$)
21: $mask \leftarrow$ VECTOR-CMPGT-GET-MASK($\vec{A}, \vec{T}$) ▷ vector set bit, if $op1 > op2$
22: $pos \leftarrow 0$
23: **while** mask **do** ▷ Locate neighbourhood w-mers in vector
24: $G \leftarrow G \cup \{x \cdot \Sigma^{''}_{i+pos}\}$
25: $mask \leftarrow mask \gg 1$
26: $pos \leftarrow pos + 1$
27: **end while**
28: **else**
29: $\vec{A} \leftarrow$ VECTOR-ADD($\vec{P}, \vec{\delta}$)
30: $mask \leftarrow$ VECTOR-CMPGT-GET-MASK($\vec{A}, \vec{T}$)
31: $pos \leftarrow 0$
32: **while** mask **do** ▷ Locate partitions in vector not pruned
33: STACK.PUSH($x \cdot \Sigma^{''}_{i+pos}$)
34: $mask \leftarrow mask \gg 1$
35: $pos \leftarrow pos + 1$
36: **end while**
37: **end if**
38: **end for**
39: **until** STACK.EMPTY( )
40: **return** $G$
41: **end procedure**

### 4.2.3 Results

Table 4.1 compares the neighbourhood generation times of the three algorithms discussed. The run times are averaged over twenty runs on a 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 5x faster than the NCBI BLAST enumeration method for $w = 4$. The performance of the naive implementation degrades drastically with increasing word lengths. For example, at $w = 6$, the prune-and-search algorithm is over 60x faster.

Table 4.1: Comparison of runtimes (in seconds) of various neighbourhood 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 |

The vector implementation shows a speedup of 3x over the sequential version. At first, this seems unusually low given simultaneous inspection of 16 characters of the alphabet during extension. We explain the discrepancy as follows. Since the alphabet size of amino acids is 20, two iterations of the *for* loop are required, resulting in an efficiency of only 62%. Additionally, the large sequential block of code in the *for* loop acutely affects the performance. Nevertheless, a consistent order of magnitude speedup is observed over the naive algorithm.

At the default Mercury BLASTP neighbourhood of $N(4, 13)$, the naive neighbourhood generation algorithm consumes approximately 10% of the program execution time. This is especially critical because the rest of the pipeline remains idle until the neighbourhood is generated. In contrast, the vectorized prune-and-search implementation is 19x faster, consuming just 0.5% of the execution time. These savings are even more pronounced for sensitive neighbourhoods at lower thresholds.

## 4.3 Query bin packing

Query bin packing is an optimization intended to speed up the BLAST search process. Multiple short query sequences are concatenated and processed in a single pass over the database. Sequences larger than the maximum supported size are broken into smaller, overlapping chunks and processed over several passes of the database. Query bin packing is especially relevant for Mercury BLASTP: the maximum query size supported is 2048 residues, while the average protein sequence in typical sequence databases is only 300 residues.

Sequence packing reduces the overhead of each pass, and so ensures that the resources available are fully utilized. 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. The HSP co-ordinates returned by the hardware stages must be translated to the reference system of the individual components. Finally, the process of packing a set of sequences in an online configuration must be optimized to reduce the overhead to a minimum. We address this latter issue in this section.

### 4.3.1 Approximate bin packing algorithms

In the query bin packing problem, we are given a list of $L = (q_1, q_2, \ldots, q_n)$ query sequences, each of length $l_i \; \epsilon \; (0, 2048]$ that must be packed into a minimum number of bins, each of capacity 2048. This is the classical one-dimensional bin packing problem and is known to be NP-hard. A number of approximation algorithms have been suggested that can be guaranteed to use no more than a constant factor of bins used by the optimal solution. We briefly describe two of the most popular algorithms [34] and study their application to query bin packing.

Let $B_1, B_2, \ldots$ be a list of bins indexed by the order of their creation. Let $B_k^l$ be the sum of the lengths of the query sequences packed in bin $B_k$. In the *Next Fit* ($NF$) algorithm, the query $q_i$ is added to the most recently created bin $B_k$ if $l_i \leq 2048 - B_k^l$. Otherwise, $B_k$ is closed and $q_i$ is placed in a new bin $B_{k+1}$, which now becomes the active bin. This algorithm is guaranteed to use not more than twice the number of bins used by the optimal solution.

The *First Fit* ($FF$) algorithm attempts to place the query $q_i$ in the first bin in which it can fit, i.e. the lowest indexed bin $B_k$, such that the condition $l_i \leq 2048 - B_k^l$ is satisfied. If no bin with sufficient room exists, a new one is created with $q_i$ as its first sequence. The $FF$ algorithm uses no more than 17/10 the number of bins used by the optimal solution.

These algorithms can be improved by first sorting the query list by decreasing sequence lengths before applying the packing rules. The corresponding algorithms are *Next Fit Decreasing* ($NFD$) and *First Fit Decreasing* ($FFD$). It can be shown that $FFD$ uses no more than 11/9 the number of bins used by the optimal solution.

### 4.3.2 Results

The approximation algorithms discussed were implemented and their performance assessed over 4,241 sequences (1,348,939 residues) of the Escherichia coli k12 proteome. The length of each query sequence was increased by one to accommodate the sequence control character. The capacity of

each bin was set to 2048 residues. Bin packing was performed either in the original order of the sequences in the input file, or after sorting by decreasing sequence length.

Table 4.2: Performance of query bin packing approximation algorithms

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

An optimal solution for this input set uses $1,353,180/2048 = 661$ bins. Table 4.2 shows the number of bins required for each of the packing rules. In general, both algorithms perform considerably better than the worst case. $FF$ performs best on the sorted list of query sequences, 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 4.3 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.

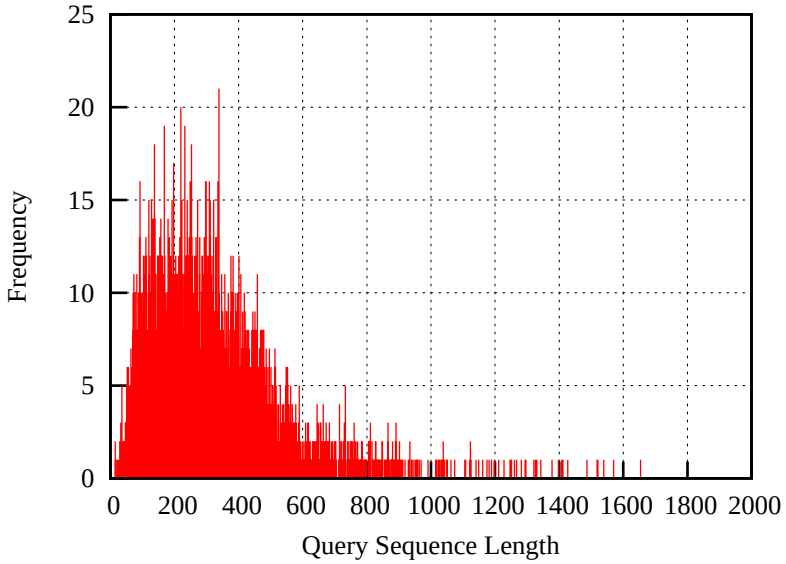



Figure 4.3: Histogram of query sequence lengths in the E.coli proteome

Sorting the list of input query sequences is possible when they are known in advance. In the case of certain configurations, such as when Mercury BLASTP is used to service requests from a web server, this is not feasible. In the on-line approach where sequences cannot be sorted, the best rule for packing is $FF$ which uses just six more bins than the optimum.

The Mercury BLASTP pipeline is stalled during the query bin packing pre-processing computation. $FF$ keeps every bin open until the entire query set is processed. The $NF$ algorithm may be used if

this pre-processing time becomes a major concern. Since only the most recent bin is inspected in this case, all previously closed query bins may be dispatched for processing in the pipeline. However, $NF$ significantly increases the number of database passes required and causes a corresponding increase in execution time.

In our implementation, the input query sequence set is assumed to be known in advance, and $FFD$ is the preferred packing rule. The Escherichia coli k12 proteome dataset is packed into query bins in under 0.17 seconds on the host CPU.

# Chapter 5

# Results

We evaluate the performance of the seed generation stage as part of the Mercury BLASTP pipeline. The throughput of the implementation is the main metric.

Sensitivity of searches conducted on Mercury BLASTP is also an important factor. We considered the sensitivity of the hardware implementation during the design phase. The seed generation algorithm implemented in hardware closely matches that of NCBI BLASTP. We expect to achieve near identical results and thus maintain the sensitivity of the search. The downstream hardware stages were also designed with this consideration; their sensitivity analysis is treated in [39, 31].

## 5.1 Implementation status

Mercury BLASTP is work in progress. The seed generation hardware has been implemented in VHDL and tested for functional correctness in simulation. A configuration of the seed generation hardware with a single copy of the word matching stage and eight copies of the two-hit module was implemented on a Xilinx Virtex-II 6000 through post place-and-route. This configuration has been verified in hardware at only 60 MHz due to stability issues experienced with the SRAM at higher clock speeds. This is a fault with the SRAM controller, and is being resolved by other members of our research group.

A configuration with three word matching stages and eight two-hit modules is still to be tested on the FPGA. We believe a clock speed of 100 MHz is a reasonable estimate for this configuration. The ungapped extension hardware prefilter is estimated to run at 75 MHz on the same FPGA [39]. The version for DNA sequences has been placed-and-routed at 100 MHz. The gapped extension module runs at 66 MHz on a Xilinx Virtex-II 4000 [31].

The software algorithms described in the previous chapter have been implemented, and a standalone program was written to test the seed generation hardware module. We are in the process of integrating the three hardware stages of the Mercury pipeline. Software to support the two-card FPGA solution and an integrated NCBI BLAST application are also in preparation.

## 5.2 Performance

We consider two configurations of the seed generation stage. MBP/1/2 implements one copy of the word matching stage and two parallel copies of the two-hit module. This is a configuration that has been placed and routed at 130 MHz on the FPGA. The second configuration, MBP/3/8, is the target configuration for Mercury BLASTP.

We estimate the performance of Mercury BLASTP based on the mean-value model described in Section 2.3.3. The throughput equation is extended to accomodate the final configuration of the Mercury BLASTP pipeline shown in Figure 3.3. The throughput equations are summarized below.

$$
\begin{aligned}
Tput_{pipe} &= min(Tput_{1a}, Tput_{1b}, Tput_{2a}, Tput_{3a}, Tput_{3b}) \text{ Mresidues/second} \\
Tput_{1a} &= \frac{h \times f_1}{\mu} \text{ Mresidues/second} \\
Tput_{1b} &= \frac{b \times f_1}{r_{1a}} \text{ Mresidues/second} \\
Tput_{2a} &= \frac{f_2}{r_{1a} p_{1b}} \text{ Mresidues/second} \\
Tput_{3a} &= \frac{f_3}{\mu_{3a} \times (r_{1a} p_{1b} p_{2a})} \text{ Mresidues/second} \\
Tput_{3b} &= \frac{1}{t_{3b} \times (r_{1a} p_{1b} p_{2a} p_{3a})} \text{ Mresidues/second}
\end{aligned}
$$

A newly considered pipeline resource in this model is the NCBI BLASTP gapped extension stage ($3b$) running on the host CPU at the end of the pipeline. The average time spent processing an input HSP in this stage is given by $t_{3b}$. The HSP input rate per database residue into this stage is $r_{1a}p_{1b}p_{2a}p_{3a}$.

The values for these parameters are summarized in Table 5.1, and the throughput in Table 5.2. The seed generation stage has a maximum throughput of 95 Mresidues/second for MBP/1/2 and 219 Mresidues/second for MBP/3/8. This translates to a 13x speedup over its software equivalent running on the benchmark machine. In both configurations, the hardware pipeline is limited by the word matching module.

However, a further increase in the number of word matching modules ($h > 3$) is limited by the performance of the gapped extension hardware prefilter. A tradeoff can be made at this point to decrease the filtering efficiency of $3a$ for better performance of this stage. This increases the number of HSPs forwarded into the software gapped extension stage, which is able to handle the higher input rate.

Table 5.1: Parameter values for the performance model

| Parameter | Value | Unit | Comment |
|---|---|---|---|
| $r_{1a}$ | 2.0067 | hits/residue | $1a$ hit production rate |
| $p_{1b}$ | 0.0183 | seeds/hit | $1b$ match rate |
| $p_{2a}$ | 0.0091 | HSPs/seed | $2a$ match rate [39] |
| $p_{3a}$ | 0.0292 | HSPs out/HSP in | $3a$ match rate [31] |
| $\mu$ | 1.3684 | 1a clks/w-mer | $1a$ service time |
| $\mu_{3a}$ | 700 | 3a clks/HSP | $3a$ service time [31] |
| $f_2$ | 75 | MHz | $2a$ clock frequency |
| $f_3$ | 66 | MHz | $3a$ clock frequency |
| $t_{3b}$ | 156 | $\mu$sec/HSP | $3b$ service time |

Table 5.2: Throughput of the two configurations of Mercury BLASTP

| Parameter | MBP/1/2 | MBP/3/8 | Unit | Comment |
|---|---|---|---|---|
| $f_1$ | 130 | 100 | MHz | clock frequency of stage 1 |
| $h$ | 1 | 3 | | number of copies of $1a$ |
| $b$ | 2 | 8 | | number of copies of $1b$ |
| $Tput_{1a}$ | 95 | 219 | Mresidues/sec | $1a$ throughput |
| $Tput_{1b}$ | 130 | 399 | Mresidues/sec | $1b$ throughput |
| $Tput_{2a}$ | 2,048 | 2,048 | Mresidues/sec | $2a$ throughput |
| $Tput_{3a}$ | 282 | 282 | Mresidues/sec | $3a$ throughput |
| $Tput_{3b}$ | 654 | 654 | Mresidues/sec | $3b$ throughput |
| $Tput_{pipe}$ | 95 | 219 | Mresidues/sec | overall pipeline throughput |
| Input rate | 61 | 140 | MB/sec | data input rate |

We also measured the input rate required from the I/O subsystem to support the throughput of the hardware pipeline. Database residues are encoded in five bits, and are packed twelve in a 64-bit word. The highest input rate required is 140 MB/sec, which is easily supported on the Mercury platform.

### 5.2.1 Benchmark comparison

In order to quantify the speedup of Mercury BLASTP, we compared its performance to the software version of NCBI BLASTP. The benchmark machine was a single 2.8 GHz Pentium 4 workstation with an L2 cache of 512 KB and 1 GB of RAM. We measured the runtime of a search of the entire Escherichia coli k12 proteome (4,242 sequences; 1,351,322 residues) against the GenBank Non-Redundant (NR) database (2,321,957 sequences; 787,608,532 residues). Post-processing time (generation and display of alignments) is excluded from our analysis. NCBI BLASTP performs a single pass of the database against each query sequence, resulting in an execution time exceeding 36 hours.

To measure the search time of Mercury BLASTP, the input query sequences were packed into 2048-residue bins. Query sequences larger than this length were split across multiple bins, with a ten percent overlap. Mercury BLASTP requires 664 passes of the database to process the packed query sequences. The average runtime of a single pass is calculated as $787,608,532/Tput_{pipe}$ seconds. The total search time evaluates to 92 and 40 minutes for the two configurations.

We also estimated the performance of TreeBLAST [32], a BLASTP-like FPGA accelerator, on this dataset. This implementation processes 1024-residue query sequences in the seed generation stage at the rate of 170 million database residues per second. The query sequences were packed into 1,368 1024-residue bins, requiring a search time of 106 minutes.

Table 5.3: Comparison of Mercury BLASTP against the benchmark

| **System** | **Passes** | **Runtime/pass** (sec) | **Total Runtime** (sec) | **Speedup** |
|---|---|---|---|---|
| NCBI BLASTP | 4,242 | variable | 130,460 | 1 |
| TreeBLASTP | 1,368 | 4.63 | 6,338 | 20 |
| MBP/1/2 | 664 | 8.29 | 5,505 | 23 |
| MBP/3/8 | 664 | 3.60 | 2,388 | 54 |

Table 5.3 summarizes these results. MBP/3/8 runs 54x faster than the benchmark machine. The speedup achieved is a result of both the hardware accelerator and the fewer passes required due to query bin packing performed on the host CPU. In contrast, TreeBLAST achieves an acceleration of only 20x, below that of the MBP/1/2 configuration.

Since we have considered the entire BLASTP pipeline in our performance model, we believe it to be reasonably accurate. However, there is still pre-processing (query bin packing, neighbourhood and table generation) and post-processing (generation of alignments) for each run that must be accounted for. We do not aim to accelerate the post-processing stage. Generation and formatting of output alignments for user display is done offline.

The pre-processing steps however count toward the cost of the hardware implementation. The acceleration efforts described in Chapter 4 were an attempt to decrease this execution time. Query bin packing reads in a set of input sequences and packs them into multiple bins. This entire computation is done on the host CPU in under 0.17 seconds. The neighbourhood and the lookup table are generated in 0.08 seconds for each bin, for a total execution time of 53.12 seconds. Before each pass of the database, the lookup table must be written into the off-chip SRAM. This operation consumes 262,144 clock cycles, and executes in 0.0027 seconds per pass on a 100 MHz system. The total time spent in the pre-processing steps is approximately 55 seconds, which is only 2.3% of the total search time.

The pre-processing time is dominated by the neighbourhood and table generation. We note however, that the multi-threaded software architecture runs the pre-processing stages in parallel to the hardware pipeline and so mitigates its effect on the overall performance.

## 5.3 Area report

We report the FPGA resource requirements of stage 1 with a single copy of word matching, and two copies of the two-hit module. The design uses 4,393 (13%) slices and 24 (17%) block RAMs on a Xilinx Virtex-II 6000. We estimate that the MBP/3/8 configuration with the ungapped extension stage and the DMA engine will occupy 85% of the slices of the FPGA.

TreeBLASTP requires the larger Xilinx Virtex-4 LX160 FPGA. It uses 60,825 (90%) slices and 254 (88%) block RAMS. The large area requirement is due to a systolic array used for word matching. Mercury BLASTP, in contrast, uses a lookup table stored in off-chip SRAM to perform a similar function.

# Chapter 6

# Conclusion and future work

## 6.1 Conclusion

The rapid growth of genomic data has made high-throughput protein searching an increasingly demanding task. Workstation clusters running NCBI BLASTP have been relied upon thus far for the necessary speed improvement. We addressed this issue by designing a hardware/software architecture to accelerate BLASTP using Field Programmable Gate Arrays.

The focus of this thesis was the seed generation stage. We designed and implemented this algorithm in hardware after a careful analysis to preserve the quality of the results. The important problem of excessive hits generated in the lookup module was handled by a novel work division scheme in the two-hit stage. We achieved a speedup of 13x over the software equivalent. The performance of the seed generation stage was considered in the context of the BLASTP pipeline including the software subsystem, to realize the entire BLASTP application. Mercury BLASTP is expected to run over 50x faster than a typical protein search on a standard workstation.

We believe Mercury BLASTP is the first FPGA accelerator for BLASTP that considers the entire sequence analysis pipeline. The design shows a 2.5x performance boost over the best known BLAST-like FPGA implementation while still consuming fewer resources on the FPGA.

## 6.2 Future work

Immediate future work includes building and testing of a configuration of the seed generation stage with multiple copies of the word matching module. The two-card FPGA solution with the integrated hardware stages is to be completed. Finally, the software interface, as well as the necessary modification to the NCBI BLAST codebase to support the Mercury system is underway. Support for future enhancements may include vector seeding in the word matching module. We believe there is a demand for commercial BLASTP accelerators and intend to explore this option.

The basic ideas of the two-hit algorithm and the work division scheme can be applied to accelerate other problems in computational biology. Electronic PCR (e-PCR) [51] is a tool used to detect landmarks (fixed subsequences) in a DNA sequence that are used to construct genomic maps. The software aims to detect primer (subsequence) pairs, i.e. a forward and reverse primer in a query string, separated by no more than $M$ bases. The current implementation first scans the sequence for the forward primer. Upon its detection, a scan of a window of $M$ bases is performed to locate the reverse primer, and thus declare a match.

We propose speeding up multiple simultaneous detection of forward/reverse primer pairs in an input sequence, by using a two-hit like data structure. The primer-pair array, which corresponds to the diagonal array, holds a distinct entry for each primer pair. In such an algorithm, the sequence is scanned simultaneously for both forward and reverse primers. Upon the detection of a forward primer, its position is recorded in a unique location of the primer-pair array. When a reverse primer is detected, a lookup is done in the array to retrieve the most recent position of the corresponding forward primer. A match is reported if it is within the window specified. This algorithm could conceivably be accelerated in hardware using a Bloom Filter based scanning phase, followed by a two-hit like module.

The basic hardware pipeline using the seeding heuristic can also be adapted for profile-based sequence searches. HMMER is an application that searches for protein families (termed a profile), by comparing a probabilistic representation of multiple proteins against a database. A dynamic programming algorithm is used to optimally compare a profile against an input sequence. Heuristics to speed up this process include the use of patterns constructed from the profile, to enable a high-speed prefilter. Such a pipeline is a prime candidate for acceleration using the Mercury BLASTP architecture.

# Appendix A

# Glossary

**Alignment:** A side-by-side comparison of two sequences reflecting their similarity. Residues may be paired across both, reflecting conservation; different, reflecting divergence, or absent in either sequence, represented by gaps. Global alignment compares the sequences in their entirety, while local alignment looks for subsequences of interest.

**Alignment Score:** The score of an alignment is computed by summing the individual scores of residue pairs in an alignment. This involves the match/mismatch score as defined by the scoring matrix and penalties for gaps introduced in either sequence.

**Database:** Refers to an organized list of sequences that have been archived by the molecular biology community. An example is the Swiss-Prot protein database.

**E-value:** Expectation Value; measures the statistical significance of a match. Denotes the number of chance occurrences in the database when using a specific scoring system. The lower the E-value of a match, the more likely it is to be biologically significant.

**Gap Penalty:** An affine gap penalty of $a + kb$ refers to a gap initiation cost of $a$ and an extension cost of $b$ for each of the $k$ gaps. This is used for the gapped alignment of two sequences.

**Neighbourhood:** The neighbourhood of a query w-mer is the list of all possible database w-mers of length $w$, whose pairwise comparison score is greater than or equal to the threshold $T$. A neighbourhood $N(w, T)$ of a query sequence is the union of the individual neighbourhoods of all query w-mers.

**Optimal Alignment:** An alignment that scores the highest based on a set of parameters. Usually, when referring to "an alignment" of two sequences, we imply any optimal alignment.

**Query:** A sequence that is compared against entries in a database to find biologically related matches.

**Residue:** Refers to a single nucleotide or amino acid in a DNA or protein sequence respectively.

**Scoring Matrix:** Defines the parameters used to evaluate the similarity of two sequences. DNA sequence comparison simply assigns a positive score for a match between two residues, and a negative score for a mismatch. Protein comparison uses a table of biologically meaningful log-odds scores for its evaluation.

**Sequence:** A chain of nucleotides or amino acids that is represented as a string of characters and forms a DNA or protein sequence respectively.

**W-mer:** A sequence of exactly $w$ residues. Also known as a word.

# References


[1] Apple/Genentech BLAST. http://www.apple.com/acg/.

[2] Cray XD1 Smith-Waterman solution. http://www.cray.com/products/xd1/smithwaterman.html.

[3] DeCypherSW - Smith-Waterman solution. http://www.timelogic.com/decypher_sw.html.

[4] GenBank. http://www.ncbi.nlm.nih.gov/Genbank/.

[5] Growth of GenBank. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[6] Growth of Swiss-Prot. http://www.expasy.org/sprot/relnotes/#SPstat.

[7] Growth of the DNA sequence database maintained by the International Nucleotide Sequence Database Collaboration. http://www.nlm.nih.gov/news/press_releases/dna_rna_100_gig.html.

[8] Paracel BLAST. http://www.paracel.com/.

[9] SGI high throughput computational BLAST. http://www.sgi.com/industries/sciences/chembio/papers.html#bio.

[10] Swiss-Prot. http://www.ebi.ac.uk/swissprot/.

[11] Timelogic DeCypher BLAST. http://www.timelogic.com/.

[12] WU-BLAST. http://blast.wustl.edu/.

[13] S F Altschul and W Gish. Local alignment statistics. *Methods Enzymology*, 266:460–80, 1996.

[14] S F Altschul, W Gish, W Miller, E W Myers, and D J Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–10, 1990.

[15] S F Altschul, T L Madden, A A Schaffer, J Zhang, Z Zhang, W Miller, and D J Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, Sep 1997.

[16] Abdullah N. Arslan, Omer Egecioglu, and Pavel A. Pevzner. A new approach to sequence comparison: normalized sequence alignment. In *RECOMB '01: Proceedings of the fifth annual international conference on Computational biology*, pages 2–11, New York, NY, USA, 2001. ACM Press.

[17] R.D. Bjornson, A.H. Sherman, S.B. Weston, N. Willard, and J. Wing. TurboBLAST: A parallel implementation of BLAST built on the turbohub. 2002.

[18] Andrea Di Blas, David M. Dahle, Mark Diekhans, Leslie Grate, Jeffrey D. Hirschberg, Kevin Karplus, Hansjörg Keller, Mark Kendrick, Francisco J. Mesa-Martinez, David Pease, Eric Rice, Angela Schultz, Don Speck, and Richard Hughey. The UCSC Kestrel parallel processor. *IEEE Trans. Parallel Distrib. Syst.*, 16(1):80–92, 2005.

[19] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[20] Douglas L. Brutlag, Jean-Pierre Dautricourt, Ron Diaz, Jeff Fier, Bruce Moxon, and Richard Stamm. BLAZE$^{\text{tm}}$: An implementation of the Smith-Waterman sequence comparison algorithm on a massively parallel computer. *Computers & Chemistry*, 17(2):203–207, 1993.

[21] Michael Cameron, Hugh E. Williams, and Adam Cannane. Improved gapped alignment in BLAST. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 1(3):116–129, 2004.

[22] Michael Cameron, Hugh E. Williams, and Adam Cannane. A deterministic finite automaton for faster protein hit detection in BLAST. *Journal of Computation Biology.*, 2005.

[23] Roger D. Chamberlain, Mark A. Franklin Ron K. Cytron, and Ronald S. Indeck. The Mercury System: Exploiting truly fast hardware for data search. *Proc. of Int'l Workshop on Storage Network Architecture and Parallel I/Os*, pages 65–72, September 2003.

[24] Roger D. Chamberlain and Berkley Shands. Streaming data from disk store to application. *Proc. of 3rd Int'l Workshop on Storage Network Architecture and Parallel I/Os*, pages 17–23, September 2005.

[25] Aaron E. Darling, Lucas Carey, and Wu chun Feng. The design, implementation, and evaluation of mpiBLAST. *ClusterWorld Conference & Expo and the 4th International Conference on Linux Clusters.*, 2003.

[26] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucl. Acids. Res.*, 27(11):2369–2376, 1999.

[27] Keith Diefendorff. Katmai enhances MMX. *Microprocessor Report*, 10/5/98.

[28] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[29] Sean Eddy. Private communication, 2006.

[30] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[31] Brandon B. Harris. Acceleration of gapped alignment in BLASTP using the Mercury system. *Master's project, Washington University in St. Louis*, August 2006.

[32] Martin Herbordt, Tom VanCourt, Yongfeng Gu, Josh Model, and Bharat Sukhwani. Single pass approximate string matching on FPGAs. In *FCCM*, 2006.

[33] Dzung T. Hoang. Searching genetic databases on Splash 2. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, Los Alamitos, CA, April 1993. IEEE Computer Society Press.

[34] Dorit S. Hochbaum, editor. *Approximation algorithms for NP-hard problems.* PWS Publishing Co., Boston, MA, USA, 1997.

[35] S Karlin and S F Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc Natl Acad Sci U S A*, 87(6):2264–8, 1990.

[36] W. J. Kent. BLAT–the BLAST-like alignment tool. *Genome Research*, 12(4):656–664, April 2002.

[37] Derek Kisman, Ming Li, Bin Ma, and Li Wang. tPatternHunter: gapped, fast and sensitive translated homology search. *Bioinformatics*, 21(4):542–544, February 2005.

[38] Praveen Krishnamurthy, Jeremy Buhler, Roger Chamberlain, Mark Franklin, Kwame Gyang, Arpith Jacob, and Joseph Lancaster. Biosequence similarity search on the mercury system. *To appear in Journal on VLSI Signal Processing.*

[39] Joseph M. Lancaster. Design and Evaluation of a BLAST Ungapped Extension Accelerator. Master's thesis, Washington University in St. Louis, St. Louis, MO. USA. 63130, May 2006.

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

[41] Ming Li, Bin Ma, Derek Kisman, and John Tromp. PatternHunter II: Highly Sensitive and Fast Homology Search. *Journal of Bioinformatics and Computational Biology*, 2(3):417–439, 2004. Early version in GIW 2003.

[42] Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, March 2002.

[43] W. S. Martins, Juan del Cuvillo, F. J. Useche, Kevin B. Theobald, and Guang R. Gao. A Multithreaded Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison. In *Pacific Symposium on Biocomputing*, pages 311–322, 2001.

[44] Scott McGinnis and Thomas L Madden. BLAST: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Research*, 32(Web Server issue):W20–5, 2004.

[45] Krishna Muriki, Keith D. Underwood, and Ron Sass. RC-BLAST: Towards a portable, cost-effective open source hardware implementation. In *IPDPS*, 2005.

[46] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[47] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two sequences. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[48] William R. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in Enzymology*, 183:63–98, 1990.

[49] Pavel A. Pevzner and Michael S. Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, 13(1/2):135–154, 1995.

[50] Torbjørn Rognes and Erling Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000.

[51] Gregory D. Schuler. Sequence mapping by electronic PCR. *Genome Research*, 5(7):541–550, May 1997.

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

[53] Euripides Sotiriades, Apostolos Dollas, and Christos Kozanitis. Some initial results on hardware BLAST acceleration with a reconfigurable architecture. In *IPDPS*, 2006.

[54] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences*, 13(2):145–150, 1997.

[55] Jinbo Xu, Daniel G. Brown, Ming Li, and Bin Ma. Optimizing multiple spaced seeds for homology search. *Journal of Computational Biology*, 2005.

[56] Yoshiki Yamaguchi, Tsutomu Maruyama, and Akihiko Konagaya. High speed homology search with FPGAs. In *Pacific Symposium on Biocomputing*, pages 271–282, 2002.

# Vita

Arpith Chacko Jacob

| | |
|---|---|
| **Date of Birth** | April 13, 1982 |
| **Place of Birth** | Manipal, India |
| **Degrees** | B.E. Computer Science and Engineering, May 2003 |
| **Publications** | P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, A. Jacob and J. Lancaster. "Biosequence Similarity Search on the Mercury System." *To appear in Journal on VLSI Signal Processing.* |

August 2006