Report Number: WUCSE-2006-21

2006-01-01

Design and Evaluation of a BLAST Ungapped Extension Accelerator, Master's Thesis

Joseph M. Lancaster

The amount of biosequence data being produced each year is growing exponentially. Extracting useful information from this massive amount of data is becoming an increasingly difficult task. This thesis focuses on accelerating the most widely-used software tool for analyzing genomic data, BLAST. This thesis presents Mercury BLAST, a novel method for accelerating searches through massive DNA databases. Mercury BLAST takes a streaming approach to the BLAST computation by offloading the performance-critical sections onto reconfigurable hardware. This hardware is then used in combination with the processor of the host system to deliver BLAST results in a fraction of the time... **Read complete abstract on page 2.**

Design and Evaluation of a BLAST Ungapped Extension Accelerator, Master's Thesis

Joseph M. Lancaster

Complete Abstract:

The amount of biosequence data being produced each year is growing exponentially. Extracting useful information from this massive amount of data is becoming an increasingly difficult task. This thesis focuses on accelerating the most widely-used software tool for analyzing genomic data, BLAST. This thesis presents Mercury BLAST, a novel method for accelerating searches through massive DNA databases. Mercury BLAST takes a streaming approach to the BLAST computation by offloading the performance-critical sections onto reconfigurable hardware. This hardware is then used in combination with the processor of the host system to deliver BLAST results in a fraction of the time of the general-purpose processor alone. Mercury BLAST makes use of new algorithms combined with reconfigurable hardware to accelerate BLAST-like similarity search. An evaluation of this method for use in real BLAST-like searches is presented along with a characterization of the quality of results associated with using these new algorithms in specialized hardware. The primary focus of this thesis is the design of the ungapped extension stage of Mercury BLAST. The architecture of the ungapped extension stage is described along with the context of this stage within the Mercury BLAST system. The design is compact and performs over 20× faster than that of the standard software ungapped extension, yielding close to 50× speedup over the complete software BLAST application. The quality of Mercury BLAST results is essentially equivalent to the standard BLAST results.

2006-21

Design and Evaluation of a BLAST Ungapped Extension Accelerator, Master's Thesis, May 2006

Authors: Joseph M. Lancaster

Corresponding Author: lancaster@wustl.edu

Abstract: The amount of biosequence data being produced each year is growing exponentially. Extracting useful information from this massive amount of data is becoming an increasingly difficult task. This thesis focuses on accelerating the most widely-used software tool for analyzing genomic data, BLAST. This thesis presents Mercury BLAST, a novel method for accelerating searches through massive DNA databases. Mercury BLAST takes a streaming approach to the BLAST computation by offloading the performance-critical sections onto reconfigurable hardware. This hardware is then used in combination with the processor of the host system to deliver BLAST results in a fraction of the time of the general-purpose processor alone.

Mercury BLAST makes use of new algorithms combined with reconfigurable hardware to accelerate BLAST-like similarity search. An evaluation of this method for use in real BLAST-like searches is presented along with a characterization of the quality of results associated with using these new algorithms in specialized hardware. The primary focus of this thesis is the design of the ungapped extension stage of Mercury BLAST. The architecture of the ungapped extension stage is described along with the context of this stage within the Mercury BLAST system. The design is compact and performs over 20× faster than that of the standard software

Type of Report: Other

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DESIGN AND EVALUATION OF A BLAST UNGAPPED EXTENSION ACCELERATOR

by

Joseph M. Lancaster

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

May 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

DESIGN AND EVALUATION OF A BLAST

UNGAPPED EXTENSION ACCELERATOR

by

Joseph M. Lancaster

ADVISOR: Jeremy Buhler and Roger Chamberlain

May 2006

Saint Louis, Missouri

The amount of biosequence data being produced each year is growing exponentially. Extracting useful information from this massive amount of data is becoming an increasingly difficult task. This thesis focuses on accelerating the most widely-used software tool for analyzing genomic data, BLAST. This thesis presents *Mercury BLAST*, a novel method for accelerating searches through massive DNA databases. Mercury BLAST takes a streaming approach to the BLAST computation by offloading the performance-critical sections onto reconfigurable hardware. This hardware is then used in combination with the processor of the host system to deliver BLAST results in a fraction of the time of the general-purpose processor alone.

Mercury BLAST makes use of new algorithms combined with reconfigurable hardware to accelerate BLAST-like similarity search. An evaluation of this method for use in real BLAST-like searches is presented along with a characterization of the quality of results associated with using these new algorithms in specialized hardware. The primary focus of this thesis is the design of the ungapped extension stage of Mercury BLAST. The architecture of the ungapped extension stage is described along with the context of this stage within the Mercury BLAST system. The design is compact and performs over 20× faster than that of the standard software ungapped extension, yielding close to 50× speedup over the complete software BLAST application. The quality of Mercury BLAST results is essentially equivalent to the standard BLAST results.

To my lovely wife, Katie

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank both of my advisers for their guidance and support throughout this work.

In addition, I would like to thank all the members of our research group for fruitful discussions and ideas.

This research would not have been possible without our sponsors, NIH and NSF. We are truly grateful for your support. This research is supported by NIH/NGHRI grant 1 R42 HG003225-01 and NSF Career grant DBI-0237902.

Finally, I would like to thank my wife and family for their encouragement, and especially their patience. They have always supported me in all walks of life.

Joseph M. Lancaster

Washington University in Saint Louis
May 2006

Chapter 1

Introduction

Databases of genomic DNA and protein sequences are an essential resource for modern molecular biology. Computational search of these databases can show that a DNA sequence acquired in the lab is similar to other sequences of known biological function, revealing both its role in the cell and its history over evolutionary time. A decade of improvement in DNA sequencing technology has driven exponential growth of biosequence databases such as NCBI GenBank [17], which has doubled in size every 12–16 months for the last decade and now stands at over 60 billion characters. Figure 1.1 shows the growth of GenBank over time. Technological gains have also generated more novel sequences, including entire mammalian genomes [14, 25], which will further increase the load on search engines.

1.1 Similarity Search

As new sequences arise from different genomic sources, there is an increasing need to efficiently extract useful information from them. Changes occur in genomes over time due to mutations. A mutation can cause a base to be replaced with a different base, a base to be dropped from the genome, or a new base to be inserted into the sequence. These mutations correspond to single-character substitutions, deletions, and insertions, respectively. An *alignment* is a comparison of two or more biological sequences with differences in the sequences annotated. Biologists want to compare biological sequences in this way because alignments allow the biologist to form hypotheses regarding their evolutionary relationship. Matching bases in the alignment

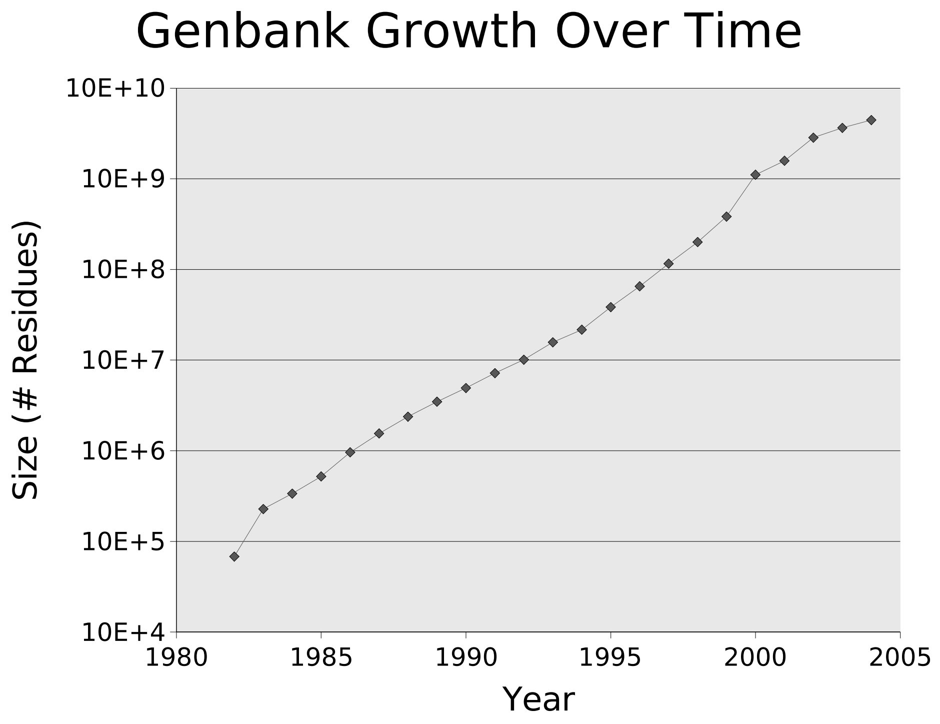

Figure 1.1: Growth of Genbank size over time. The number of bases is doubling every 12–16 months.

are assumed to have descended from the same ancestral base, while mismatching or missing bases are considered to have diverged from the ancestor. A common method of generating an alignment is by calculating their statistical similarity. Statistical similarity is a tool used to objectively measure how close sequences are evolutionarily related and is commonly calculated by measuring their *string edit distance*. String edit distance gives a direct indication of how "good" an alignment is.

Figure 1.2 shows an example of an alignment between two DNA sequences. The DNA alphabet is made of up 4 characters: A, C, G, and T. A hyphen in one sequence represents a deletion from that sequence (or an insertion into the other sequence). The first DNA sequence is shown across the top horizontal line, the middle line annotates the type of edit, and the third line is the second DNA sequence. A vertical line between two bases means that the bases are an exact match, while the absence of a line indicates either an insertion or deletion or a mismatch. The bases bounded by the dashed box denote an exact word match, a w-mer, of length 4 (e.g., $w = 4$ here). This is discussed in more detail in Chapter 2.

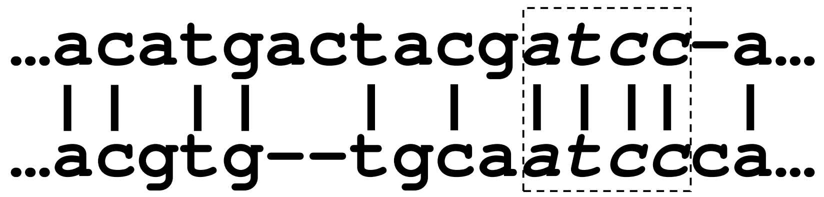

Figure 1.2: An example of a short DNA-to-DNA alignment. The bases inside the dashed box represent an exact word match of length 4.

The fastest known algorithm for calculating the edit distance between strings is the Smith-Waterman dynamic programming algorithm [23]. Smith-Waterman finds the optimal alignment of two strings with the fewest number of edits (i.e. single-character substitutions, insertions, or deletions). Many variants of this algorithm are implemented as the core of genetic comparison software. Let the input to Smith-Waterman be two strings of sizes m and n, and let C be a constant factor. Then Smith-Waterman runs in time Cmn. Table 1.1 gives approximate Smith-Waterman comparison run times for various sized genome-to-genome comparisons assuming C, the number of cell updates per second, is 120 MCUPS (Cell Updates Per Second). This constant factor was extrapolated from a cell fill rate of 30 MCUPS on a single 933 MHz Pentium III [26] to the fill rate of a newer generation processor. It is evident that, even with the smaller genome comparisons, directly calculating the edit distance for two strings is infeasible on today's computers. The smallest genome to genome comparison in Table 1.1 would take 7.2 years to complete. While using a hardware-accelerated implementation of Smith-Waterman is clearly faster, only the first row in Table 1.1 is even remotely feasible. Since executing Smith-Waterman directly (using either a software or hardware approach) leads to excessive run times, heuristics are used in practice to find localized pairs of small regions with small edit distances in a fraction of the time.

1.2 BLAST

The most widely used software for efficiently comparing biosequences to a database is BLAST, the **B**asic **L**ocal **A**lignment **S**earch **T**ool [1, 2, 3]. BLAST compares a *query sequence* to a biosequence database to find sequences that differ from the query by the fewest edits. Because direct measurement of edit distance between sequences is

Table 1.1: Estimated run times of Smith-Waterman for a general purpose CPU and specialized hardware. The software estimates use a cell fill rate of 120 MCUPS (Cell Updates Per Second). This number corresponds roughly to a single 3 GHz Pentium 4. The custom hardware uses a cell fill rate of 13.9 GCUPS which is given in [19].

Sequence 1 (# Bases)	**Sequence 2 (# Bases)**	**Software Runtime**	**Hardware Runtime**
D. melanogaster (180 Mbases)	D. pseudoobscura (150 Mbases)	7.2 Years	23 Days
H. sapiens (2.9 Gbases)	M. musculus (2.5 Gbases)	1,916 Years	16.5 Years
Z. mays (2.5 Gbases)	T. aestivum (16 Gbases)	11 Millennia	91 Years

generally infeasible (as shown above), BLAST uses a variety of heuristics to identify small portions of a large database that are worth comparing carefully to the query.

BLAST is a pipeline of computations that filter a stream of characters (the database) to identify meaningful matches to a query. To keep pace with growing databases and queries, this stream must be filtered at increasingly higher rates. Even with these heuristics, BLAST searches still take a substantial amount of time, as is discussed in detail in Chapter 2. Running BLAST on the smallest genome comparison in Table 1.1 still takes 12 CPU-hours to run, assuming the system described in Chapter 2 and extrapolating up from the throughput given in Table 2.3. Even with the BLAST heuristics, the largest genome-to-genome comparison in Table 1.1 still takes 2 CPU-years to complete.

The use of NCBI BLAST is growing every day. NCBI receives over 100,000 BLAST search requests each day. With a rapidly-growing demand for BLAST results and increasingly large databases to be searched, there is a clear need for a faster BLAST.

A natural approach to accelerating BLAST is through software techniques. There have been many attempts to do this. Some examples are [28], [18], [10], [11], [16], and [5]. A more detailed discussion of these approaches is given in Chapter 2. While there are many improvements gained from these approaches, none of them report even an order of magnitude speedup over software BLAST for large queries. Hence, the problem of slow BLAST seems to be an open one.

One path to higher performance is to develop a specialized processor that offloads part of BLAST's computation from a general-purpose CPU. A method of accelerating BLAST that has been deployed in the past is to offload only the gapped alignment onto a custom processor. In Chapter 2 we show why it is necessary to accelerate other portions of the program in order to achieve significant speedups. Past examples of processors that accelerate or replace BLAST include the ASIC-based Paracel GeneMatcher^{TM} [21] and the FPGA-based TimeLogic DecypherBLAST^{TM} engine [24]. While these accelerators do give clear benefits over software BLAST, they tend to be costly and suffer from swift obsolescence cycles.

We have developed a new accelerator design, the FPGA-based Mercury BLAST engine [13]. Mercury BLAST exploits fine-grained parallelism in BLAST's algorithms and the high I/O bandwidth of current commodity computing systems to deliver 1–2 orders of magnitude speedup over software BLAST on a card suitable for deployment in a laboratory desktop. Mercury BLAST is specifically designed to handle large genome-to-genome comparisons which are becoming more common as new, entire genomes are sequenced. Some possible motivations for performing such a search are to establish a mapping between orthologous parts of the sequences, to compare genomes to expressed mRNA sequences, and to compare databases to each other.

Mercury BLAST is a multistage pipeline, parts of which are implemented in FPGA hardware, others in software. This work describes a key part of the pipeline, *ungapped extension*, that sifts through exact word matches between query and database and decides whether to perform a more accurate but computationally expensive comparison between them. Ungapped extension attempts to grow a larger but possibly inexact word by allowing mismatches to appear in the extended word. If the extended word, called a *high-scoring segment pair* (HSP), has few enough mismatches, it is considered worthwhile to inspect in gapped extension. If not, the HSP is discarded from the pipeline.

Our design illustrates a fruitful approach to accelerating variable-length string matching that is robust to character substitutions. The implementation is compact, runs at high clock rates, and can process one pattern match every clock cycle.

1.3 Economic Analysis

Developing an FPGA-based solution requires significant design and implementation efforts not present in a software solution in addition to increased costs due to using non-commodity hardware. To speed up large BLAST searches, the search has traditionally been partitioned and performed on a cluster of general purpose processors. In order to justify a custom FPGA-based solution such as Mercury BLAST, one must examine the economic benefits of such a solution over the common method. To compare the cost of the traditional method to Mercury BLAST one can compare the cost and speed per unit. Let X be the cost per 1U rack space of Mercury BLAST (which should be enough space to hold the entire Mercury BLAST system). Let Y be the cost per 1U rack space of a cluster computer. Let S be the speedup of Mercury BLAST over a 1U cluster computer. Then Mercury BLAST is a better value when $X < Y \times S$ holds true.

To illustrate this analysis, let $X = \$50,000$ and let $Y = \$2,200$. Figure 1.3 shows the relative benefit of using an FPGA-based approach assuming the FPGA-based approach can achieve 40× speedup over a 1U single-processor cluster computer (equivalently, 80× speedup over a 1U dual-processor cluster computer). Figure 1.3 shows a clear benefit to using Mercury BLAST in this situation resulting in a cost savings of almost $45,000 compared to the traditional cluster system using 32-U of rack space.

1.4 Contributions

The following list outlines the author's specific contributions:

- New Knowledge
    1. Identified the need for a different ungapped extension algorithm for use in reconfigurable hardware. To accomplish this, the source code from NCBI BLAST was studied and a preliminary hardware system was designed based on the original NCBI BLAST ungapped extension algorithm.

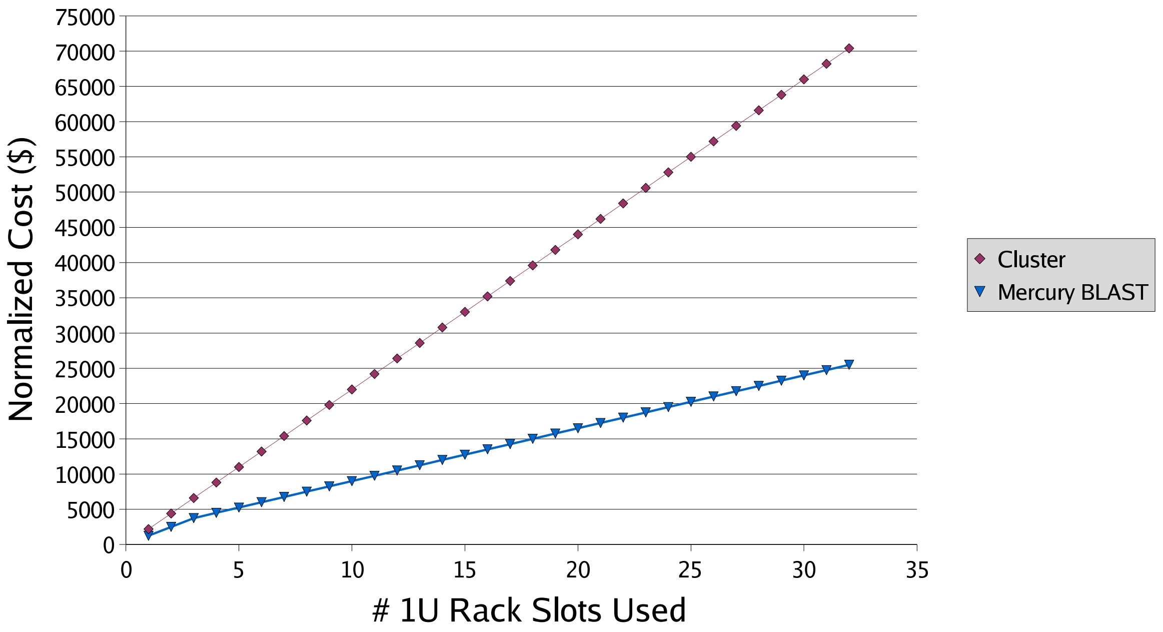

Figure 1.3: Normalized cost of two different BLAST systems. The graph assumes a 1U Mercury BLASTN system costs \$50,000 and is 40× faster than a 1U single-processor cluster computer costing \$2,200.

2. Developed a new algorithm for ungapped extension. The motivation for doing so was based on the design drawbacks from the hardware implementation of NCBI BLAST ungapped extension. The new algorithm was designed to increase throughput and reduce resource utilization in hardware while maintaining the quality of results from the algorithm.

3. The behavior of NCBI BLAST with the new ungapped extension algorithm was characterized. To accomplish this, NCBI BLAST was instrumented with new monitoring functions to gather detailed performance statistics about the execution across varied inputs. A software emulator of the new ungapped extension algorithm was inserted in the pipeline to evaluate the new algorithm and the results were compared to the original BLAST software. Three difference configurations of the pipeline were evaluated: original BLAST pipeline, BLAST pipeline with ungapped extension replaced with new algorithm, and BLAST pipeline with new ungapped extension as an added pipeline stage in front of the standard BLAST ungapped

extension stage. Each configuration was evaluated using a statistically significant sampling of the human genome of various sizes as the query versus the mouse genome as the database.

4. A program was created which compares the output of BLAST (i.e. gapped alignments) from two different configuration. Since no good quantitative metric for comparing gapped alignments was known, a new overlap metric was created for comparing gapped alignments. This overlap metric allows objective measurement of gapped alignments which gave us a basis for comparing the quality of results.
5. A high-performance ungapped extension filter was designed, implemented, and tested in reconfigurable hardware.
6. A time-multiplexed design to allow a physically dual-ported Block RAM to masquerade as a quad-ported Block RAM was designed, implemented and tested.
7. Stage 1 was designed with significant input from the author. Among other contributions, a minimal, perfect hashing strategy was developed to reduce the complexity of the hash lookup logic as well as decrease the load load and size of the external memory needed to store the hash tables.
8. A redundant hit filter, similar in functionality to the one used in NCBI BLASTN, yet more suitable for hardware was designed. The redundancy filter was characterized using a software emulator scheme similar to the methods used for ungapped extension.
9. The redundancy filter was implemented and tested in hardware.

Items 8 and 9 are only briefly discussed in this thesis. The reader is referred to [13] for a detailed exposition.

- New Infrastructure and Implementations

1. A new framework for measuring the behavior of NCBI BLAST across multiple executions with different input sequences was developed and has been expanded for use with NCBI BLASTP.
2. An infrastructure for gathering statistics from multiple BLAST runs to gather aggregate performance statistics was created.

3. The hardware prototyping system infrastructure was deployed and tested.
4. Stage 1 and stage 2 of Mercury BLASTN were integrated, tested, and debugged. This system is running many independent clock domains.

1.5 Organization of Thesis

The rest of the thesis is organized as follows. Chapter 2 gives a fuller account of the BLAST computation, illustrates the need to accelerate ungapped extension, and discusses related research. Chapter 3 describes our accelerator design and details its hardware architecture. Chapter 4 evaluates the quality of results and throughput of our implementation, and Chapter 5 concludes and discusses future work.

Chapter 2

Background and Related Work

2.1 The BLAST Computation

BLAST's search computation is organized as a three-stage pipeline, illustrated in Figure 2.1. The pipeline is initialized with a query sequence, after which a database is streamed through it to identify matches to that query. We focus on BLASTN, the form of BLAST used to compare DNA sequences; however, many of the details described here also apply to BLASTP, the form used for protein sequences. The following discussion of BLAST closely follows the description in [13].

The first pipeline stage, *word matching*, detects substrings of fixed length w in the stream that perfectly match a substring of the query; typically, $w = 11$ for DNA. We refer to these short matches as *w-mers*. Each matching w-mer is forwarded to the second stage, *ungapped extension*, which extends the w-mer to either side to identify a longer pair of sequences around it that match with at most a small number of mismatched characters. These longer matches are *high-scoring segment pairs (HSPs)*, or *ungapped alignments*. Finally, every HSP that has both enough matches and sufficiently few mismatches is passed to the third stage, *gapped extension*, which uses

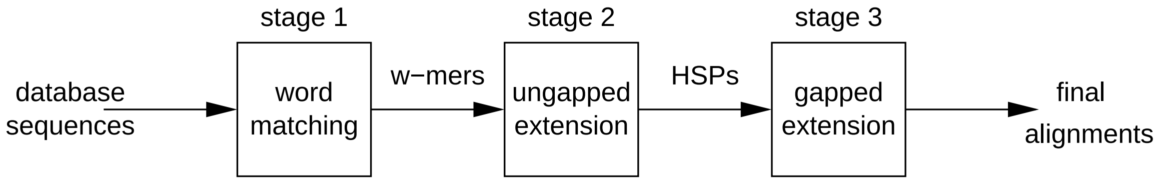

Figure 2.1: NCBI BLAST pipeline.

the Smith-Waterman dynamic programming algorithm [23] to extend it into a *gapped alignment*, a pair of similar regions that may differ by arbitrary edits. BLAST reports only gapped alignments with many matches and few edits.

Although each stage of BLASTN is more compute-intensive than the previous, each successive stage also discards a large fraction of its input. Table 2.1 quantifies the data reduction at each stage of the BLASTN pipeline for various query lengths. The pass rate, p_i, represents the probability that an output from stage i is generated from an individual input to that stage. For stage 1, p_1 measures the number of matches generated per DNA base read from the database. For stage 2, p_2 measures the number of HSPs generated for each w-mer received from stage 1. For stage 3, p_3 measures the number of gapped alignments generated for each HSP received from stage 2. The results clearly show the data reduction trend at each stage, with a very small percentage of HSPs arriving for processing in the expensive gapped alignment stage. As the query increases in size, the number of w-mers generated by random chance in stage 1 also increases. This places the burden on stage 2 to filter out these spurious hits before gapped alignment.

In the performance predictions that follow, we will consider the throughput of individual stages of the pipeline as well as the throughput of the entire pipeline. To make throughputs comparable, they are normalized to be in units of input bases per second from the database. When executing on a single computational resource (i.e., software running on a single processor), the average compute time per input base can be expressed as $t_1 + p_1t_2 + p_1p_2t_3$, where t_i is the compute time for stage i for each input item (base, match, or alignment) to stage i. The normalized throughput is then $Tput = 1/(t_1 + p_1t_2 + p_1p_2t_3)$.

Table 2.1: Pass rates p across pipeline stages [13]

Query Size (bases)	**Stage 1** (p_1)	**Stage 2** (p_2)	**Stage 3** (p_3)
10 k	0.00858	0.0000550	0.320
25 k	0.0205	0.0000619	0.141
50 k	0.0411	0.0000189	0.194
100 k	0.0841	0.0000174	0.175
1 M	0.851	0.0000172	0.096

To quantify the computational cost of each stage of BLASTN on a general-purpose CPU, we measured the standard BLASTN software published by the National Center for Biological Information (NCBI), v2.3.2, with default parameters on a 2.8 GHz Intel P4 workstation with 512 KB of L2 cache and 1 GB of RAM, running Linux. We compared a database containing the non-repetitive fraction of the mouse genome (1.16×10^9 characters) to queries of various lengths selected at random from the human genome. CPU time was measured separately for each of the three pipeline stages.

The length of a typical query sequence in BLASTN is application-dependent. For example, a short DNA sequence obtained in a single lab experiment may be only a few hundred bases, while in genome-to-genome comparison, a query (one of the genomes) may be billions of bases long. A BLAST implementation should support the largest computationally feasible query length, both to accommodate long individual queries and to support the optimization of "query packing," in which multiple short queries are concatenated and processed in a single pass over the database with enough invalid sequence between them to ensure the boundary is never crossed. Conversely, queries longer than the maximum feasible length may be broken into pieces with some overlap, each of which is processed in a separate pass.

In our experiments, we tested queries of 10 kbases, 25 kbases, 50 kbases, 100 kbases, and 1 Mbase, both to simulate different applications of BLASTN and to assess the impact of query length on the performance of our firmware implementation. One megabase is a reasonable upper bound on query size for NCBI BLASTN with standard parameters, since it generates 11-mer word matches by chance alone at a rate approaching one match for every base read from the database. Timings were averaged over at least 20 queries randomly sampled form the human genome for each length, and each query's running time was averaged over three identical runs of BLASTN. It should be noted that, given a query sequence of length n, BLASTN compares the database to both the sequence and its DNA reverse-complement, effectively doubling the query length. The reverse-complement of a DNA strand is formed by first reversing the order of the bases and then transforming each base into its complementary base. The performance numbers reported in this section and throughout the rest of the thesis reflect such "double-stranded" queries.

Table 2.2 gives the distribution of times spent in each stage of NCBI BLASTN for various query sizes. Averaged times are given with 95% confidence intervals. Time spent in stage 1 dominated that spent in later pipeline stages, stage 2 takes a significant fraction of the pipeline time, while time spent in stage 3 was almost negligible. Although later stages are computationally more intensive, each stage is such an efficient filter that it discards most of its input, leaving later stages with comparatively little work.

Table 2.2: Percentage of pipeline time spent in each stage of NCBI BLASTN [13]

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%

From the measured running times of our experiments and the size of the mouse genome database, we computed the throughput (in Mbases from the database per second) achieved by NCBI BLASTN's pipeline for varying query sizes. The results are shown in the first row of Table 2.3. Throughput depends strongly on query length. To explain this observation, we used the predicted filtering efficiencies p_i for each pipeline stage and the distribution of running times by stage to estimate the average time spent to process each base in stage 1, each word match in stage 2, and each ungapped alignment in stage 3. These results are shown in the remaining rows of the table. While the overhead per input remains constant for stage 2 and actually decreases for stage 3, the cost per base in stage 1 grows linearly with query length. This cost growth derives from the linear increase in the expected number of matches per base that occur purely by chance, in the absence of any meaningful similarity. These chance matches can propagate to stage 3 but are quickly discarded which explains why the t_3 small queries can spend more time per alignment than larger queries.

Table 2.3: Summary of performance results for software runs of NCBI BLASTN [13]

Query Size (bases)	**10 kbases**	**25 kbases**	**50 kbases**	**100 kbases**	**1 Mbase**	**Units**
Throughput	67.0	29.2	14.9	8.76	0.648	Mbases/sec
Stage 1 (time per base, t_1)	0.0129	0.0287	0.0553	0.0951	1.32	μsec/base
Stage 2 (time per match, t_2)	0.231	0.265	0.281	0.225	0.264	μsec/match
Stage 3 (t_3)	71.3	60.4	81.8	58.9	34.4	μsec/alignment

Table 2.4: Performance model parameters. Query size is 25 kbases (double stranded), and the pass fractions for stage 2 are with the most permissive cutoff score of 16.

Parameter	Units	Meaning
p_1	matches/base	stage 1 pass fraction [13]
p_2	HSPs/match	stage 2 pass fraction
t_3	μsec/HSP input	stage 3 execution time [13]
Tp_2	Gmatches/sec	stage 2 throughput (not normalized)
$Tput_1$	Gbases/sec	stage 1 throughput [13]
$Tput_2$	Gbases/sec	stage 2 throughput
$Tput_3$	Gbases/sec	stage 3 throughput
$Tput_{overall}$	Gbases/sec	overall pipeline throughput

Our profile illustrates that, to achieve more than about a 6x speedup of NCBI BLASTN on large genome comparisons, one must accelerate *both* word matching (stage 1) *and* ungapped extension (stage 2). Mercury BLASTN therefore accelerates both these stages, leaving stage 3 to NCBI's software. Our previous work [13] described how we accelerate word matching.

To explore the benefit to be gained from accelerating stage 2, we develop the following performance model of the system. This model assumes that the hardware ungapped extension constitutes the entire ungapped extension stage . There is no software ungapped extension executed in the pipeline. When executing in a heterogeneous pipeline, the overall throughput is determined by the minimum throughput achieved on any one resource. The performance model presented here assumes that stage 1 and stage 2 are accelerated in hardware. Stage 3 is executed in software. The throughput is given by

$$Tput_{overall} = \min\left(Tput_1, Tput_2, Tput_3\right),$$

where $Tput_1$, accelerated stage 1 throughput, is 1.4 Gbases/sec from [13]; $Tput_2$, stage 2 throughput normalized to input bases per second, is expressed as $Tput_2 = Tp_2/p_1$; and $Tput_3$, throughput of software stage 3, is expressed as $Tput_3 = 1/p_1p_2t_3$. Tp_2 is the input to the model, representing the throughput of the hardware ungapped extension stage. Table 2.4 summarizes the model parameters.

With stage 1 (word matching) accelerated as described in [13], the performance of stage 2 (ungapped extension) directly determines the performance of the overall pipelined application. Figure 2.2 graphs $Tput_{overall}$ as a function of the performance attainable in stage 2 (i.e., Tp_2) (quantified by the ingest rate of stage 2 alone, in million matches per second).

The software profiling shows an average execution time for stage 2 alone of 0.265 μs/match, which corresponds to a throughput of 3.8 Mmatches/s, plotted towards the left of Figure 2.2. As we increase the performance of stage 2, the overall pipeline performance increases proportionately until stage 2 is no longer the bottleneck stage. Figure 2.3 shows the speedup of the overall system as a function of Tp_2. The lower speedup for the 20 kbase query is because the software runs faster for smaller query lengths. In order to reach peak performance, stage 2 must have a throughput higher than approximately 35 Mmatches/sec. Precisely how we accomplish this is the subject of this thesis.

2.2 System Infrastructure

As the name implies, Mercury BLAST has been targeted to the *Mercury* system [6]. The *Mercury* system is a prototyping infrastructure designed to accelerate disk-based computations using FPGA co-processors. Figure 2.4 shows the architecture of the *Mercury* system. This system exploits the high I/O bandwidth available from modern disks by streaming data directly from the disk medium to the FPGA. Software running on the host processor initiates a data stream off the disk subsystem over the PCI-X bus, through the FPGA co-processor, and finally back over the PCI-X bus to the host processor. The configuration of the *Mercury* system is ideal for hardware-software codesign. The application can easily be divided up into portions that execute on the reconfigurable hardware and those that execute exclusively in software. The *Mercury* system is especially well suited to applications that process large volumes of data on their input and can filter out much of the data in the initial stages of the computation, leaving the higher-complexity computations to be done on a much smaller set. Fortunately, BLAST is one such application. A detailed explanation of the *Mercury* System is found in [6].

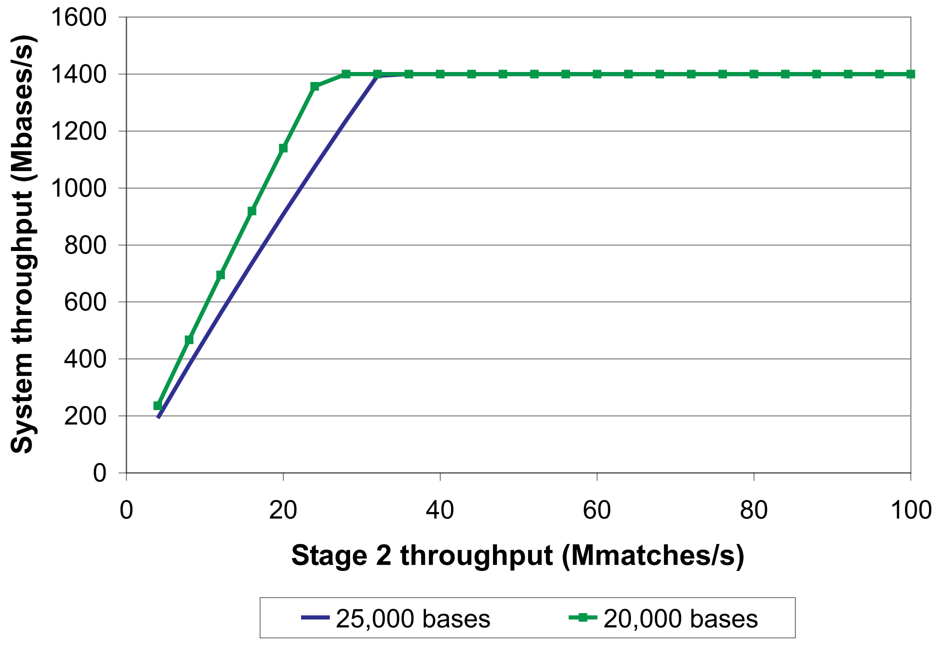

Figure 2.2: Throughput of overall pipeline as a function of ungapped extension throughput for queries of sizes 20 kbases and 25 kbases.

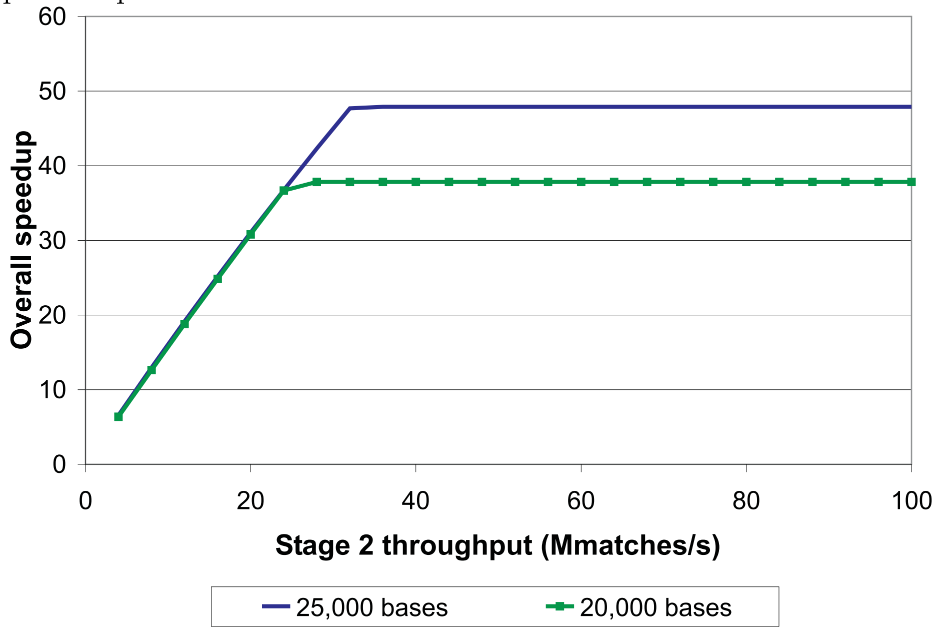

Figure 2.3: Speedup of overall Mercury BLASTN pipeline as a function of ungapped extension throughput for queries of sizes 20 kbases and 25 kbases over software BLAST.

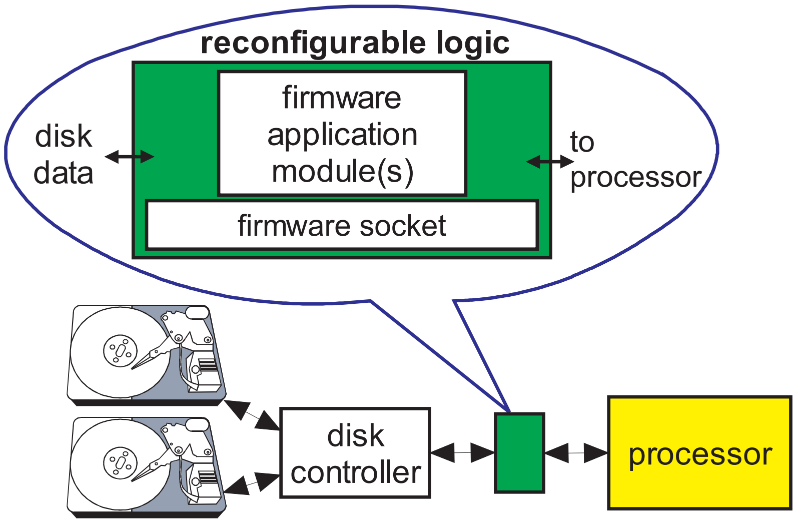

Figure 2.4: Mercury System Architecture [6]

The version of the *Mercury* system used for implementing Mercury BLAST consists of a dual-processor host system with a prototyping co-processor board and a SCSI disk subsystem. The two host CPUs are 2.0 GHz AMD Opteron processors with a total of 6 GB of memory (4 GB installed on one processor and 2GB on the other). The two processors are connected to each other, and to the PCI-X, through Hypertransport links. The disk subsystem is a SCSI Ultra320 10,000 RPM disk drive. The hardware prototyping board used is connected to the host CPU through the PCI-X bus. The board contains a single Xilinx Virtex-II 6000 FPGA with 8,448 Configurable Logic Blocks (CLBs). Each CLB is divided into 4 slices with each slice providing two 4 input Look Up Tables (LUTs) and two registers. The FPGA also contains 144 18 kbit Block RAMs which provide dedicated on-chip memory and 144 18x18 dedicated multipliers. External SRAM modules are attached through expansion connectors on the board to provide larger memories for the hardware applications.

Definition of Terms

The following terms are found in this thesis and defined here for convenience of the reader.

- *Base*: The basic elements of DNA. A single base is one of {A—C—T—G}.

- *Biosequence Database*: A typically large sequence of symbols (e.g. bases) which forms the one half of the BLAST input.
- *CUPS*: Cell Updates Per Second. Similar to cell fill rate, except the units are explicitly stated in the name.
- *Cell Fill Rate*: The rate at which dynamic programming cells are computed. Usually stated in units "cells / second."
- *DNA*: Deoxyribonucleic acid. One of the two forms of nucleic acid in living cells. DNA is the genetic material for all life forms and many viruses
- *DNA Sequencing*: The technique for determining the order of nucleotides in a DNA molecule.
- *E-value*: The expected number of HSPs with score at least S that occur in a database search by chance alone. The lower the E-value the more significant the result.
- *Gapped Alignment*: The result of a gapped extension. This is differentiated from an HSP only by the inclusion of gaps as a possible edit.
- *Gapped Extension*: The process of locally aligning two sequences allowing all edit operations. This typically performed using a variant of the Smith-Waterman algorithm.
- *GenBank*: An annotated collection of all publicly available DNA sequences. One of the largest sequence repositories in the world.
- *Genome*: A genome describes the entire genetic makeup of a living organism.
- *HSP*: High-scoring Segment Pair. The result of an ungapped extension alignment.
- *Mutations*: An alteration of the nucleotide sequence in a DNA molecule.
- *Ports (RAM)*: The number of ports on a physical RAM device refers to the number of independent, simultaneous read or write operations that can be performed in a single clock cycle.

- *Prefilter*: In this context, a prefilter refers to an additional filter inserted in front of an existing filter to further improve performance.
- *Protein*: A polymer compound consisting of amino acids.
- *Query Sequence*: A biological sequence to be compared to a biosequence database. A query sequence can be a new genetic sequence to be compared to existing sequences. Forms one half of a BLAST input.
- *RNA*: Ribonucleic acid. The other form of nucleic acid in living cells.
- *Redundant Hit*: A redundant hit is a w-mer that overlaps, or is very close to, a previously inspected w-mer.
- *Seeded Alignment*: An alignment method by which the gapped alignments are required to have arisen from a word match (i.e., seed).
- *Sensitivity*: Sensitivity is a measure of how closely a set of heuristics match an optimal (or baseline) output.
- *Sequence Alignment*: An arrangement of two or more sequences denoting their similarity.
- *Specificity*: Specificity is a measure of the ability of an algorithm to identify the important areas of the sequences while ignoring the uninteresting segments.
- *String Edit Distance*: String edit distance refers to the number of single-character edit operations (substitutions, insertions, or deletions) to change one string into another.
- *Ungapped Extension*: A substitution-robust method of locally aligning sequences. This method does not allow insertion or deletion of characters.
- *w-mer*: A contiguous word match of length w.
- *Word Match*: A common occurrence of a pattern of bases in two biological sequences, usually of fixed length.

2.2.1 Literature Review

There have been several approaches to improving biosequence similarity search methods. Some of these approaches use specialized hardware while others attempt to improve biosequence search methods using purely software techniques. Many approaches, including the one described in this paper, use hybrid techniques employing both a general purpose computer as well as specialized hardware. The following gives an overview of other attempts at improving the state of the art of biosequence similarity search.

Software tools exist that seek to accelerate BLASTN-like computations through algorithmic improvements. MegaBLAST [28] is used by NCBI as a faster alternative to BLASTN; it explicitly sacrifices substantial sensitivity relative to BLASTN in exchange for improved running time. The SSAHA [18] and BLAT [10] packages achieve higher throughput than BLASTN by requiring that the entire database be indexed offline before being used for searches. By eliminating the need to scan the database, these tools can achieve more than an order of magnitude speedup versus BLASTN; however, they must trade off between sensitivity and space for their indices and so in practice are less sensitive. In contrast, Mercury BLASTN aims for at least BLASTN-equivalent sensitivity.

Another approach to improving the performance of BLAST can be found in [5]. This software achieves speedup over standard NCBI BLASTP by adding an extra dynamic programming stage after ungapped extension, dubbed *semi-gapped alignment*, to filter out even more unfruitful hits before gapped alignment. The same paper offers another, orthogonal technique called *restricted insertion alignment* which can be applied to either semi-gapped or fully-gapped alignment to decrease the runtime. While it is unclear how fruitful accelerating Smith-Waterman is for Mercury BLASTN, semi-gapped alignment may offer an opportunity to further improve the performance of Mercury BLASTP by adding yet another pipeline stage to the hardware.

Other software, such as DASH [11] and PatternHunter II [16], achieves both faster search and higher sensitivity compared to BLASTN using alternative forms of pattern matching and dynamic programming extension. DASH's reported speedup over BLASTN is less than 10-fold for queries of 1500 bases, and it is not clear how it performs at our query sizes, which are an order of magnitude larger. DASH's authors have also reported on a preliminary FPGA design for their algorithm [12].]Pattern-Hunter II achieves only a two-fold reported speedup relative to BLASTN, albeit with substantially greater sensitivity.

In hardware, numerous implementations of the Smith-Waterman dynamic programming algorithm have been reported in the literature, using both non-reconfigurable ASIC logic [7, 8] and reconfigurable logic [9, 20, 27]. These implementations focus on accelerating gapped alignment, which is heavily loaded in proteomic BLAST comparisons but takes only a small fraction of running time in genomic BLASTN computations. Our work instead focuses on accelerating the bottleneck stages of the BLASTN pipeline, which reduces the data sent to later stages to the point that Smith-Waterman acceleration is not necessary.

While one could in principle dispense with the pattern matching and ungapped extension stages of BLASTN given a sufficiently fast Smith-Waterman implementation, no such implementation is likely to be feasible with current hardware. The projected data rate of 1.4 Gbases/s for a 25 kbase query, if achieved by a Smith-Waterman implementation, would imply computation of around 10^{14} dynamic programming matrix cells per second. In contrast, existing FPGA implementations report rates of less than 10^{10} cells per second.

High-end commercial systems have been developed to accelerate or replace BLAST [21, 24]. The Paracel GeneMatcher$^{\text{TM}}$ [21] relies on non-reconfigurable ASIC logic, which is inflexible in its application and cannot easily be updated to exploit technology improvements. In contrast, FPGA-based systems can be reprogrammed to tackle diverse applications and can be redeployed on newer, faster FPGAs with minimal additional design work. RDisk [15] is one such FPGA-based approach, which claims a 60 Mbase/sec throughput for stage 1 of BLAST using a single disk.

Two commercial products that do not rely on ASIC technology are BLASTMachine2$^{\text{TM}}$ from Paracel [21] and DeCypherBLAST$^{\text{TM}}$ from TimeLogic [24]. The highest-end 32-CPU Linux cluster BLASTMachine2$^{\text{TM}}$ performs BLASTN with a throughput of 2.93 Mbases/sec for a 2.8 Mbase query. The DeCypherBLAST$^{\text{TM}}$ solution uses an

FPGA-based approach to improve the performance of BLASTN. This solution has throughput rate of 213 kbases/sec for a 16 Mbase query.

Chapter 3

Design Description

As was mentioned in the previous chapter, the nature of the BLASTN computation made it necessary to accelerate both the word matching and ungapped extension stages in reconfigurable hardware to attain reasonable speedups for BLASTN. A profile of BLASTP software suggests that it is also necessary to accelerate the corresponding two stages in hardware as well as the final stage of the BLAST pipeline, gapped extension. This thesis focuses on the acceleration of the ungapped extension stage.

We first briefly describe the hardware-based accelerator for word matching in Mercury BLAST, which was implemented by other members of our research group. Next, a new algorithm for ungapped extension in BLAST is presented. We end with a description of the accelerator for ungapped extension for Mercury BLAST.

3.1 Word Matching Accelerator

This section summarizes the *Mercury* BLASTN word matching accelerator which has previously been described in [13] and was developed by other members in our research group. We begin with a general introduction to word matching and how it is performed in BLASTN. Then, we describe the design of the hardware accelerated word matching module for *Mercury* BLASTN.

3.1.1 Word Matching in NCBI BLASTN

As described earlier, BLAST is organized as a sequence of algorithms, with each algorithm becoming more sophisticated and computationally expensive. The strategy is to spend as little time as possible executing the more expensive algorithms on irrelevant data. One popular approach to effectively achieving this goal is *seeded alignment*. BLAST implements seeded alignment by finding *word matches* of a fixed length between the query and the database. The idea here is that if there is a high concentration of matching bases in some part of the query and database, then there is a higher likelihood that there are biologically significant matches near there than chance alone, hence, it should be inspected with more scrutiny.

Formally, a word match is a string of some fixed length w (referred to as a "w-mer") that occurs in both the query and the database. In NCBI BLASTN, w-mers are deemed worthy of further inspection if they are at least ≈ 11 bases in length. However, the vast majority of these w-mers are present from chance alone, which illustrates the need for closer inspection. To speed up the word matching stage, the NCBI BLASTN implementation does word matching by first searching for two-byte words (i.e. an 8-mer) that lie on byte boundaries. Figure 3.1 illustrates the possible positions of an 11-mer relative to byte boundaries. It is clear from Figure 3.1 that every 11-mer will be found by first finding 8-mers and then looking at 3 bases on both sides of the 8-mer. Once a byte-aligned 8-mer is found, bases on each side of the match are inspected to attempt to construct an 11-mer.

If two 11-mers occur close to each other in both the query and database, they are likely to have arisen from the same biological feature. Hence, to avoid duplicate inspection in the later stages, NCBI BLASTN implements a *redundancy elimination* filter at the output of ungapped extension. The redundancy elimination filter checks whether each 11-mer match overlaps or is sufficiently close to a previously discovered match. If this is the case, the current 11-mer is suppressed, since the biological feature that the 11-mer arose from must have already been inspected using the previous 11-mer that was close by. Word matches that pass the redundancy filter are then further inspected by the ungapped extension filter.

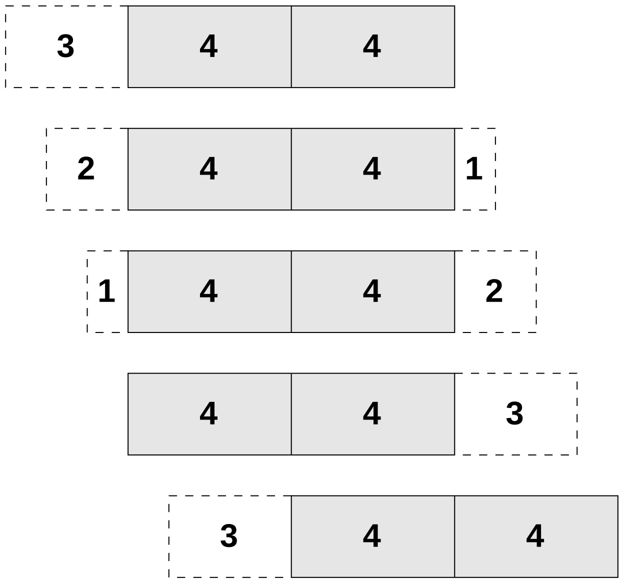

Figure 3.1: Illustration of possible 11-mer positioning relative to a byte-aligned 8-mer. The solid boxes represent aligned bytes in memory. The dashed boxes represent the possible locations of the 11-mer relative to the 2 aligned bytes. The numbers inside the boxes represent the number of bases in the box.

3.1.2 Word Matching in an FPGA

The word matching accelerator for *Mercury* BLASTN is divided into 3 sub-stages, illustrated in Figure 3.2. Even though the goal is functionality similar to NCBI BLASTN, the mechanisms by which this are achieved are very different. The word matching accelerator implements a *prefilter* using Bloom filters for the first substage, a lookup stage for the second, and finally performs redundancy elimination. All w-mers that pass through all three stages are forwarded to stage 2.

A Bloom filter [4] is a probabilistic method to quickly test membership in a large set. Bloom filters produce no false negatives but produce some false positives with a rate that varies with the configuration of the filter. Bloom filters can be efficiently implemented in hardware and allow many queries to occur in a single clock cycle. A Bloom filter is used to reduce the load to the external hash table used in stage 1b.

The second substage of the word matching accelerator is the hash lookup stage. This stage is responsible for taking a word out of the Bloom filter stage, hashing it, and

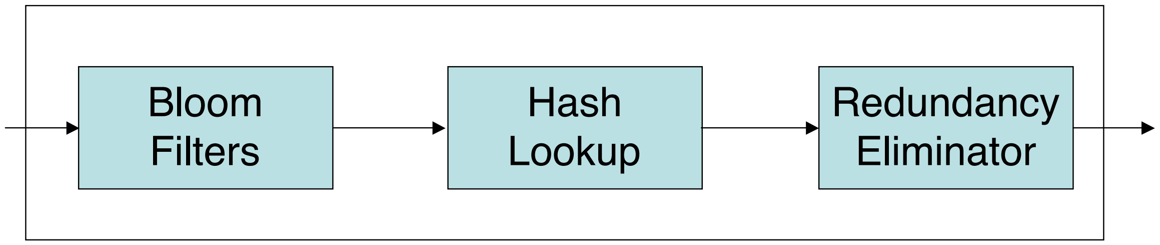

Figure 3.2: Division of word matching for *Mercury* BLASTN. Stage 1a implements a Bloom filter, stage 1b hashes the word and does a lookup into the hash table, and stage 1c removes redundant word matches.

querying the hash table to test membership. The hash table is stored in external SRAM. In addition to testing for membership, the table also stores the offset, of the matches in the query so that they can be inspected more closely in later stages. The hashing scheme used here is the new near-perfect hashing scheme described in [13]. Hashing in this way allows for dramatic reductions in hash table size while maintaining very good performance.

The redundancy filter takes the seed offsets from the hash table and checks to see if they overlap, or are very close to previous 11-mers. If so, the seed is dropped; otherwise, it is passed on to stage two. The redundancy filter is implemented in the FPGA using Block RAMs to store the diagonal positions. The diagonal table is proportional to the size of the query. This redundancy filter technique, however, does not use feedback from stage 2 to make decisions on whether or not to drop a w-mer. Instead of a strict overlap parameter, a *trailing gap* parameter is defined which determines the minimum number of bases away a w-mer must lie to be considered distinct. More detail on the redundancy filter method used can be found in [22].

3.2 Ungapped Extension Accelerator

This section describes the design of the ungapped extension FPGA accelerator for BLAST. Ungapped extension is used in both the BLASTN and the BLASTP implementations, with minor differences. We explain the motivation for performing ungapped extension and show the method that is implemented in NCBI BLAST.

Next, we describe the design of a new ungapped extension algorithm which can be used as a prefilter in reconfigurable logic or as a standalone ungapped extension filter.

3.2.1 Motivation for Ungapped Extension in BLAST

The purpose of extending a w-mer is to determine, as quickly and accurately as possible, if the w-mer is from chance alone or if it may have greater significance. Ungapped extension must decide whether each w-mer found during the word matching stage is worth inspecting by the more computationally intensive gapped extension. It is important to distinguish between spurious w-mers as early as possible in the BLAST pipeline because the stages are increasingly more complex the farther down the pipeline you go. There exists a delicate balance between the stringency of the filter and its sensitivity, i.e. the number of truly biologically significant alignments that are found. A highly stringent filter is needed to minimize time spent in fruitless gapped extension, but the filter must not throw out w-mers that legitimately identify long query-database matches with few differences. For FPGA implementation, this filtering computation must also be parallelizable and simple enough to fit in a limited area.

Mercury BLAST implements stage 2 guided in part by lessons learned from the implementation of word-matching described in [13]. It deploys an FPGA-based ungapped extension stage which is well suited for deployment as a *prefilter* in front of NCBI BLAST's software ungapped extension. This design exploits the speed of FPGA implementation to greatly reduce the number of w-mers passed to software while retaining the flexibility of the software implementation on those w-mers that pass. A w-mer must pass both hardware and software ungapped extension before being released to gapped extension. Since the performance focus is on overall throughput, adding an additional processing stage which is deployed on dedicated hardware is often a useful technique. Alternatively, the ungapped extension firmware can be used as the only ungapped extension filter in the pipeline. If correctly parametrized, this method has the potential advantage of lowering the burden on the CPU.

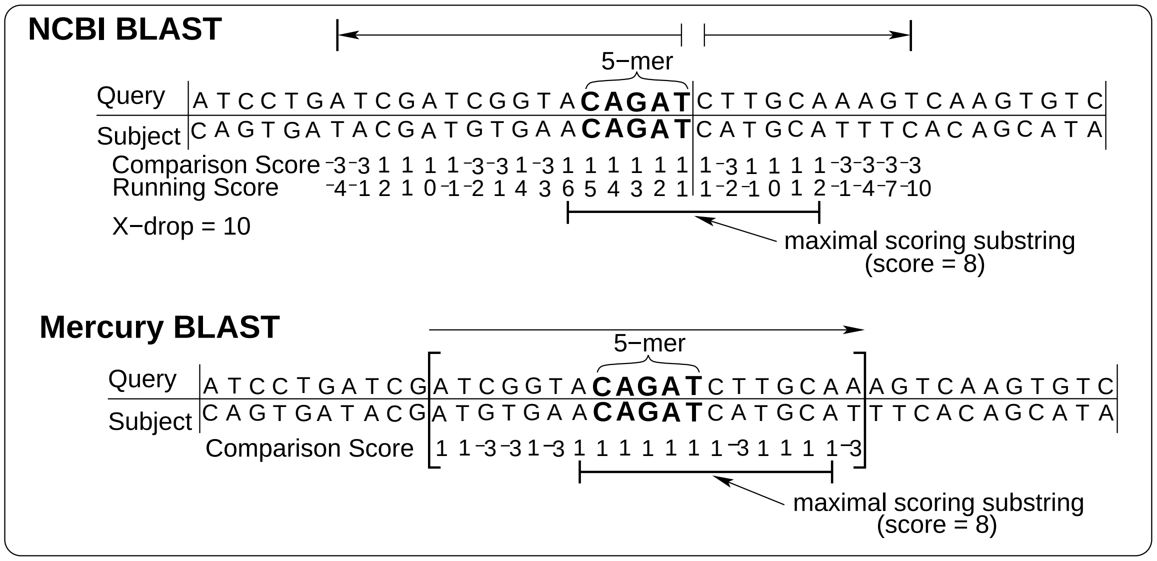

Figure 3.3: Example of NCBI and Mercury ungapped extension. The parameters used here are $L_w = 19$, $w = 5$, $\alpha = 1$, $\beta = -3$, and X-drop= 10. NCBI BLAST ungapped extension begins at the end of the w-mer and extends left. The extension stops when the running score drops 10 below the maximum score (as indicated by the arrows). The same computation is done in the other direction. The final substring is the concatenation of the best substrings from the left and right extensions. Mercury BLAST ungapped extension begins at the leftmost base of the window (indicated by brackets) and moves right, calculating the best scoring substring in the window. Note that even though in this example the algorithms gave the same result, this is not necessarily the case in general.

In the next sections, we briefly describe NCBI BLAST's software ungapped extension stage, then describe Mercury BLAST's hardware stage. Figure 3.3 shows an example illustrating the two different approaches to ungapped extension.

3.2.2 NCBI BLAST Ungapped Extension Algorithm

NCBI BLAST's ungapped extension of a w-mer into an HSP runs in two steps. The w-mer is extended back toward the beginnings of the two sequences, then forward towards their ends. As the HSP extends over each character pair, that pair receives a reward $+\alpha$ if the characters match or a penalty $-\beta$ if they mismatch. An HSP's score is the sum of these rewards and penalties over all its pairs. The end of the HSP in each direction is chosen to maximize the total score of that direction's extension. If the final HSP scores above a user-defined threshold, it is passed on to gapped extension.

For long sequences, it is useful to terminate extension before reaching the ends of the sequences, especially if no high-scoring HSP is likely to be found. BLAST implements early termination by an X*-drop* mechanism. The algorithm tracks the highest score achieved by any extension of the w-mer thus far; if the current extension scores at least X below this maximum, further extension in that direction is terminated.

Ungapped extension with X-dropping allows BLAST to recover HSPs of arbitrary length while limiting the average search space for a given w-mer. However, because the regions of extension can in principle be quite long, this heuristic is not very suitable for fast implementation in an FPGA. Note that even though extension in both directions can be done in parallel, this was not sufficient to achieve the speedups we desired.

3.2.3 Ungapped Extension Accelerator Design

Mercury BLAST takes a different, more FPGA-friendly approach to ungapped extension. Since the NCBI BLAST ungapped extension algorithm was not ideal for implementation in hardware, we first reexamined the methods used for ungapped extension. Instead of performing the extension in two steps, ungapped extension for a given w-mer is performed in a single forward pass over a fixed-size window. These features of our approach simplify hardware implementation and expose opportunities to exploit fine-grain parallelism and pipelining that is not easily accessed in NCBI BLAST's algorithm. Our extension algorithm, *Ungapped_Extend*, is given as pseudocode in Figure 3.4.

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

```
Ungapped_Extend($w$–mer)
  Calculate window boundaries
  $\Gamma = \gamma = 0$
  $B = B_{max} = E_{max} = 0$

  for $i = 1...L_w$
    if $q_i = s_i$
      $\gamma = \gamma + \alpha$
    else
      $\gamma = \gamma - \beta$

    if $\gamma > 0$
      if $\gamma > \Gamma$ and $i > WmerEnd$
        $\Gamma = \gamma$
        $B_{max} = B$
        $E_{max} = i$
    else if $i < WmerStart$
      $B = i + 1$
      $\gamma = 0$

  if $\Gamma > T$ or $B_{max} = 0$ or $E_{max} = L_w$
    return True
   else
    return False
```

Figure 3.4: *Mercury* BLASTN ungapped extension algorithm pseudocode. BLASTP ungapped extension is performed the same way with a base-dependent scoring mechanism.

Ungapped_Extend implements a dynamic programming recurrence that simultaneously computes the start and end of the best HSP in the window. First, the score contribution of each character pair in the window is computed. For BLASTN, the same bonus $+\alpha$ and penalty $-\beta$ as the software implementation are used. Similarly for BLASTP, the same base-pair dependent score matrices are used in the hardware accelerator. These contributions can be calculated independently in parallel for each base-pair. Then, for each position i of the window, the recurrence computes the score γ_i of the best (highest-scoring) HSP that terminates at i, along with the position B_i at which this HSP begins. These values can be updated for each i in constant time. The algorithm also tracks Γ_i, the score of the best HSP ending at *or before* i, along with its endpoints B_{max} and E_{max}. Note that Γ_{L_w} is the score of the best HSP in the entire window. If this value is greater than a user-defined score threshold, the w-mer passes the prefilter and is forwarded to software ungapped extension.

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

3.2.4 Architecture

Figure 3.5 shows the organization of the application pipeline for BLASTN. The input stream flows from the disk, is delivered to the hardware word matching module (which also employs a prefilter), and then passes into the ungapped extension prefilter. The output of the prefilter goes to the processor for the remainder of stage 2 (NCBI ungapped extension) and stage 3 (gapped extension). The prefilter algorithm lends itself

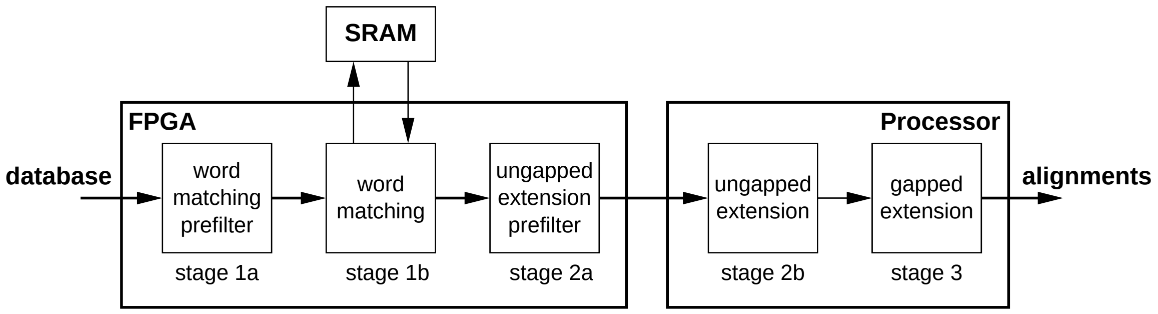

Figure 3.5: Overview of Mercury BLASTN hardware/software deployment when using both hardware and software ungapped extension.

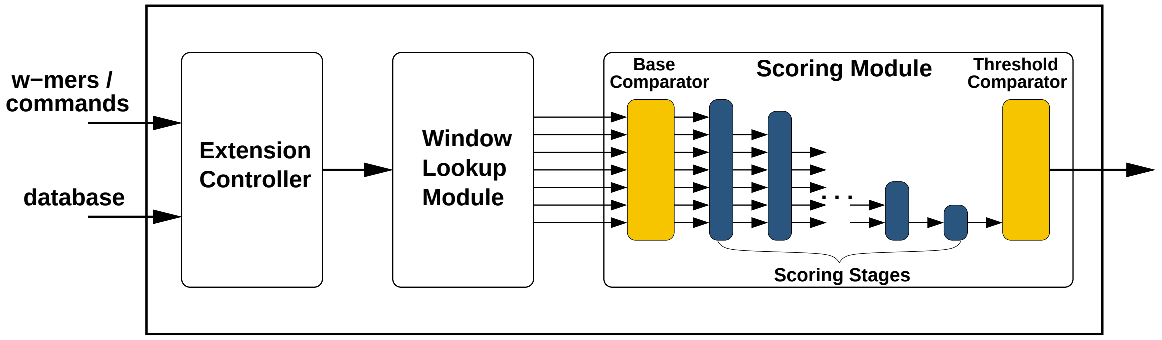

Figure 3.6: Ungapped extension prefilter design.

to hardware implementation despite the sequential expression of the computation in Figure 3.4.

The ungapped extension prefilter design is fully pipelined internally and accepts one match per clock. Typically in the Mercury System, each module is connected to one upstream module and one downstream module by a 64-bit data bus and a collection of various control signals. However, the w-mer matching stage logically generates more output than input so two independent, 64-bit data paths are utilized between the word matching stage and the ungapped extension stage. The w-mers and commands are sent on one path, and the database is sent on the other. The module is organized as 3 pipelined macro-stages as illustrated in Figure 3.6.

The controller parses the input from the host system to distinguish between commands and data on the input stream. Commands are supported by an active control valid signal, and the commands are sent in a standard format developed for the Mercury system. Commands allow for online assignment of various parameters used in

Mercury BLAST without the need for reconfiguring the FPGA. These commands can also reset individual stages without having to reset the entire FPGA. Commands are supported to configure parameters such as match score, mismatch score, and cutoff thresholds. Allowing the user to set these parameters at runtime leaves the trade-off between sensitivity and throughput to his or her discretion. The complete set of supported commands for the ungapped extension stage can be found in Appendix A.

All w-mer matches and the database flow through the controller into the window lookup module. This module is responsible for fetching the appropriate substrings of the stream and the query. Figure 3.7 shows the overall structure of the module. The query is stored directly on-chip using the dual-ported Block RAMs on the FPGA. The query is loaded once at the beginning of the BLAST computation and is fixed until the end of the database is reached. The size of the query is limited to the amount of Block RAMs that are allocated for buffering it. In the current implementation, a maximum query size of 65,536 is supported using 8 Block RAMs. The database stream is buffered in a similar fashion as the query, except that we use a circular buffer to retain the necessary section of the stream that windows formed from arriving w-mers might need. Since the w-mer generation is done in the first hardware stage, only a relatively small amount of the stream needs to be buffered to accommodate all extension requests. The database buffer was built to allow a fixed-distance of out-of-order w-mers to be processed correctly. This is important in the BLASTP implementation since the w-mers from the word matching stage will possibly be out of order.

The window lookup module is organized as a 6-stage pipeline. Extensive pipelining is necessary to keep up with requests from the previous stage, which may come once per clock. The first stage of the pipeline calculates the beginning of the query and database windows based off the incoming w-mer and a configurable window size. The offset of the beginning of the window is then passed to the buffer modules which begin the task of reading a superset of the window from the Block RAMs. One extra word of data must be retrieved from the Block RAMs because there is no guarantee that the window boundaries will fall on a word boundary. Hence, one extra word is fetched on each lookup so that the exact window can be constructed from a temporary buffer holding a window size worth of data plus the extra word. The next four stages of the pipeline move the input data lock-step with the buffer lookup process and ensure

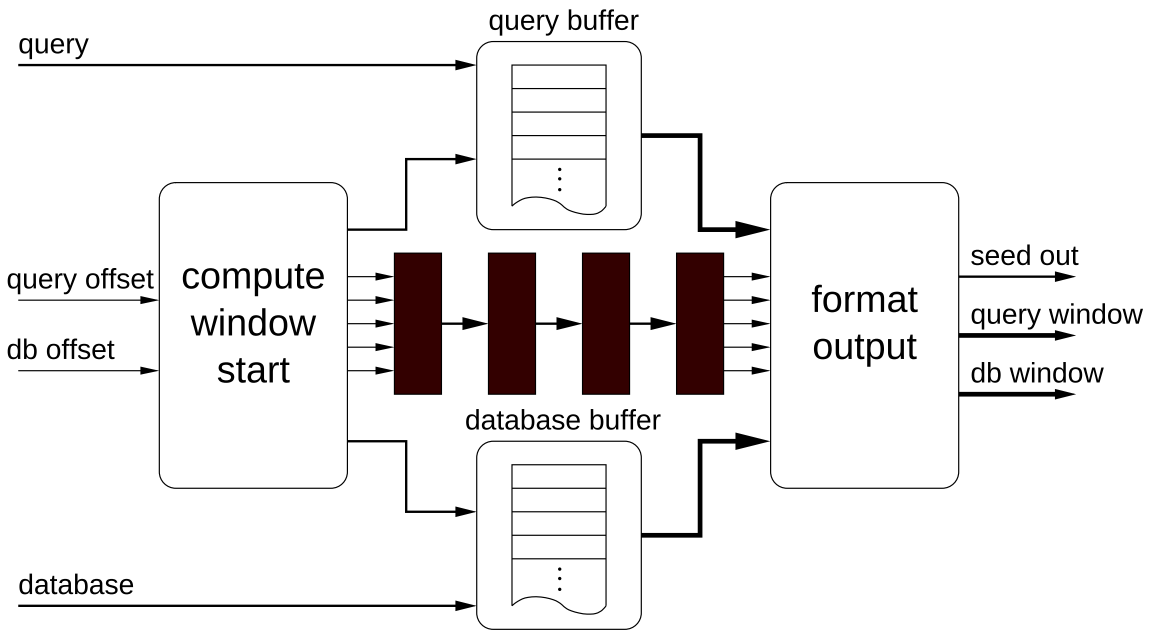

Figure 3.7: Top-level diagram of the window lookup module. The query is streamed in at the beginning of each BLAST search. A portion of the database stream flows into a circular buffer which holds the necessary portion of the stream needed for extension. The controller takes in a w-mer and is responsible for calculating the bounds of the window, requesting the window from the buffer modules, and finally constructing the final window from the raw output of the buffer.

that pipeline stalls are handled in a correct fashion. In the final stage, the superset of the query and database windows arrive to the top level module. The correct window of the buffers is extracted and registered as the output.

The window lookup module as well as the Bloom filters for stage 1 use *quad-ported* Block RAMs. The quad-ported Block RAMs are built on top of the physically dual-ported Block RAM structures. Figure 3.8 shows a block diagram illustrating the design of a quad-ported Block RAM. The existing dual-ported Block RAMs are time-multiplexed to allow two accesses per cycle. This presents four ports to the user, which may be used to read and write the Block RAMs with the same restrictions as the standard dual-ported Block RAMs. Having a quad-ported structure is advantageous in many situations, as it allows double the number of reads or writes to be performed in a single clock cycle, minimizing the number of Block RAMs needed for some designs. Note that quad-porting the Block RAMs does not increase the size of the RAM; it simply increases the ability to access this memory. Time-multiplexing a resource such as a Block RAM allows Mercury BLASTN to support a query that is twice as large as would be otherwise possible. A notable limitation of this technique is the requirement of a frequency-doubled clock, which can lower the maximum frequency at which a design can operate.

After the window is fetched, it is passed into the scoring module and registered. The scoring module implements the recurrence of the extension algorithm. Since the computation is too complex to be done in a single cycle, the scorer is extensively pipelined.

Figure 3.9 illustrates the first stage of the scoring pipeline. This stage, the base comparator, compares every base pair and assigns a score to each base pair in the window. For BLASTN, the base comparator assigns a reward α to each matching base pair and a penalty $-\beta$ to each mismatching pair. The calculation of the comparison scores is done in a single cycle, using L_w comparators. The score computation is the same for BLASTP, except there are many choices for the score. In BLASTP, the α and $-\beta$ are replaced with a value retrieved from a lookup table which is indexed by the concatenation of the two bases. After the scores are calculated, they are stored for use in later stages of the pipeline.

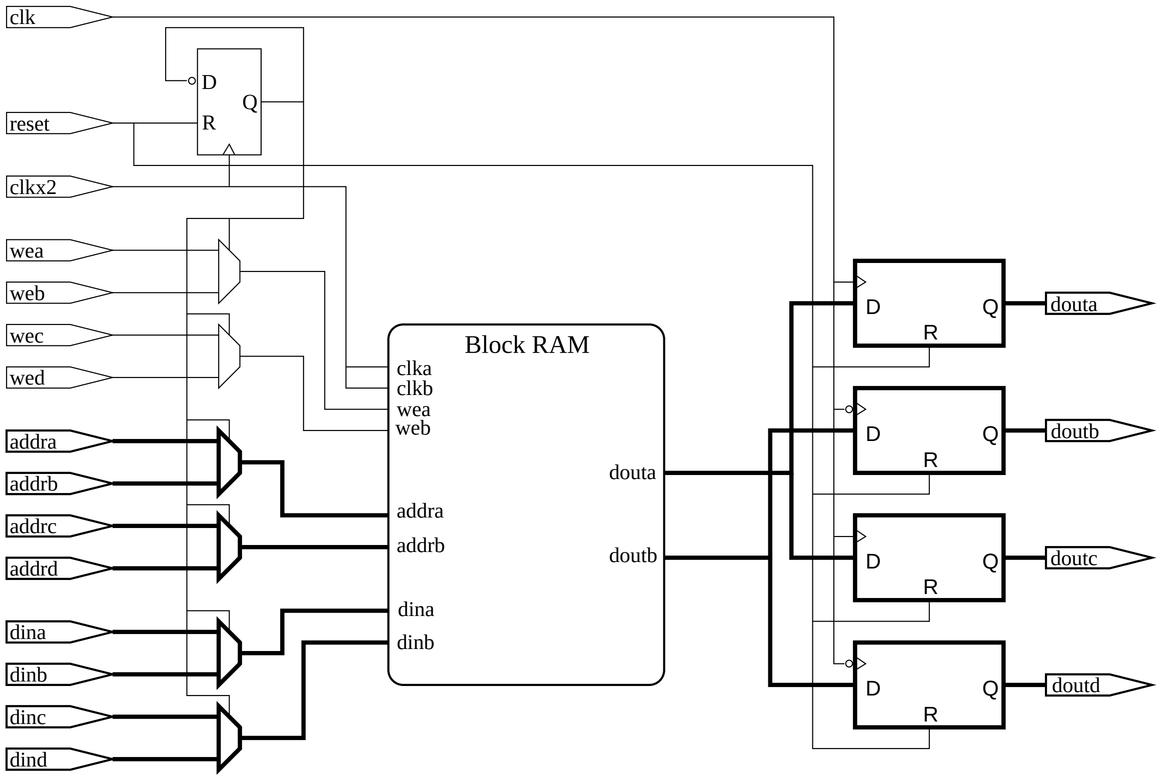

Figure 3.8: Diagram of a time-multiplexed Block RAM to provide four independent ports. The wires shown in bold represent multi-wire paths. *clkx2* is a frequency-doubled clock which is phase-aligned to *clk*.

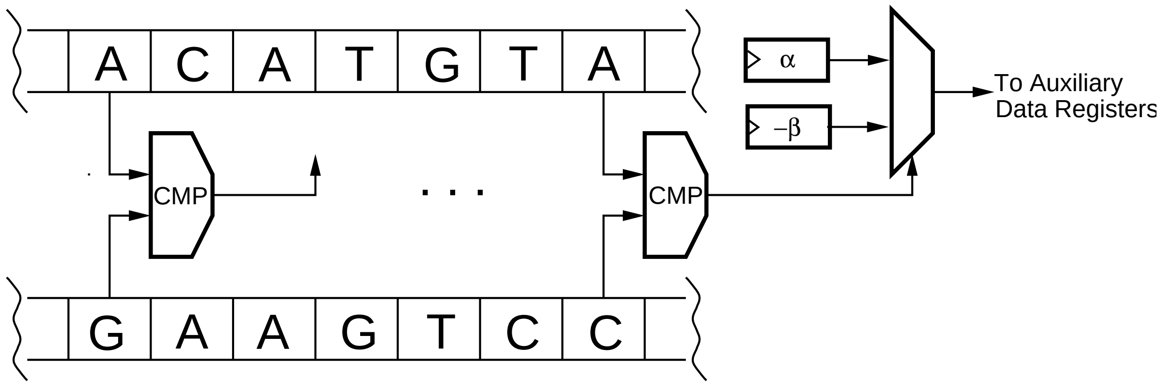

Figure 3.9: The base comparator stage computes the scores of every base pair in the window in parallel. These scores are stored in registers which are fed as input to the systolic array of scorer stages.

The scoring module is arranged as a classic systolic array. The data from the previous stage is read on each clock, and output to the following stage on the next clock. Figure 3.6 shows successive scoring stages decreasing in size. While the amount of information about the state of the computation is the same for each scorer stage, the auxiliary information (i.e., comparison scores) for the previous stages is discarded as a w-mer moves down the pipe. For instance, after the comparison score for base i has been used, that comparison score is not needed again since the the later stages will only be looking at successive base pairs. Figure 3.10 shows the nature of the data movement down the pipeline. The darkened registers hold the auxiliary data needed for the pipeline. Since there are L_w w-mers in the pipeline, a copy of the necessary auxiliary information for each w-mer is passed in-step with the calculation on that w-mer. As data moves from left to right, the total amount of auxiliary data decreases linearly until the last scoring stage. At the last stage, there are only two comparison scores stored. While Figure 3.10 depicts a maximally-pipelined scoring module in which each stage computes only one step of the extension recurrence, the current implementation is able to sustain high clock rates while computing *two* steps of the recurrence in each stage. This conserves hardware resources which can be allocated to other parts of Mercury BLASTN.

The next section of the pipeline is the scoring stages. Each of these stages contains the logic to implement essentially lines 12-19 of the algorithm described in Figure 3.4. Figure 3.11 shows the interface of an individual scoring stage. The values shown entering from the left are the state of computation from the previous stage, which are updated by the combinational logic and stored in the registers shown on the right. The data entering from the top of the module is the supporting information for a given w-mer, which is independent of the state of the computation. In order to sustain a high clock frequency design, each scoring stage computes two iterations of the loop per clock cycle, resulting in $L_w/2$ scoring stages for a complete calculation. Hence, there are $L_w/2$ independent w-mers being processed simultaneously in the scoring stages of the processor when the pipe is full.

The final pipeline stage of the scoring module is the threshold comparator. The comparator takes the fully-scored segment and makes a decision to discard or keep the w-mer. This decision is based on the score of the alignment as compared to the user-defined threshold, T, and the position of the maximal scoring substring. If the

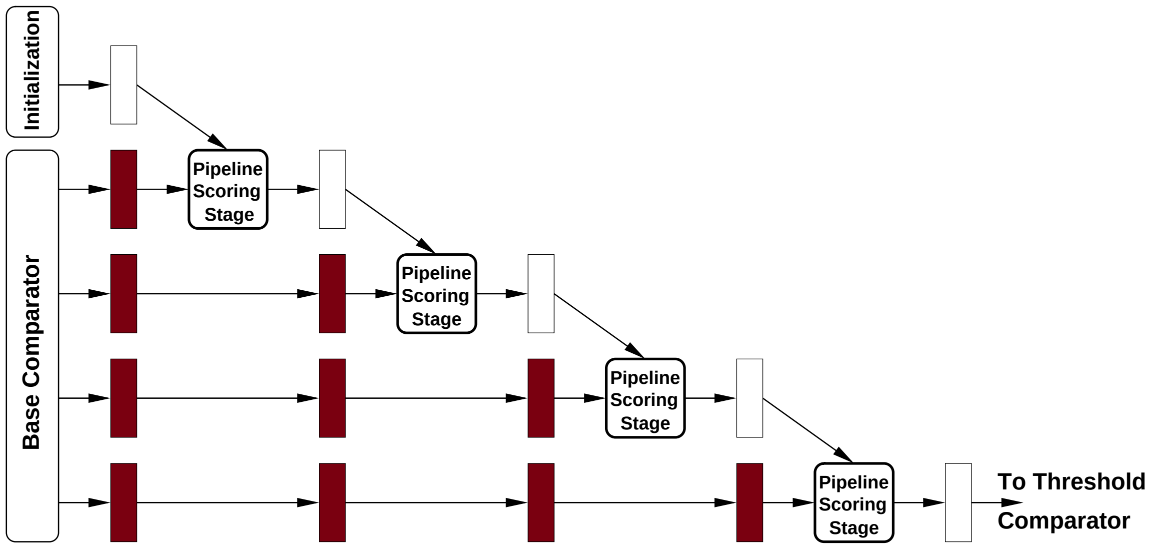

Figure 3.10: Depiction of the systolic array of scoring stages. The dark registers hold auxiliary data which is independent of the state of the computation. The data flows left to right on each clock cycle. The light registers are the pipeline calculation registers used to transfer the state of the computation from a previous scoring stage to the next. Each column of registers contains an independent w-mer in the pipeline.

maximum score is above the threshold, the w-mer is passed on. Additionally, if the maximal scoring substring intersects either boundary of the window, the w-mer is also queued for further inspection regardless of the score. Otherwise, the w-mer is discarded.

Figures 3.12 through 3.16 show an example of a single 4-base window being processed in the systolic array. The light objects contain valid data while the darkened objects are idle. Note that even though this pipeline is capable of processing 5 independent windows simultaneously, this example only shows the processing of one window for clarity. The two recurrence variables shown, Γ and Bmax, represent the global maximum score and the beginning position of the maximal scoring substring, respectively. A positive reward value of 1 is used for matching bases and a negative penalty of -3 is used for mismatching bases in the window.

First, the recurrence variables are initialized and stored in registers while the comparison scores are calculated and stored in registers concurrently. Then, one step of the recurrence (i.e. one base comparison score is consumed) is performed, and the

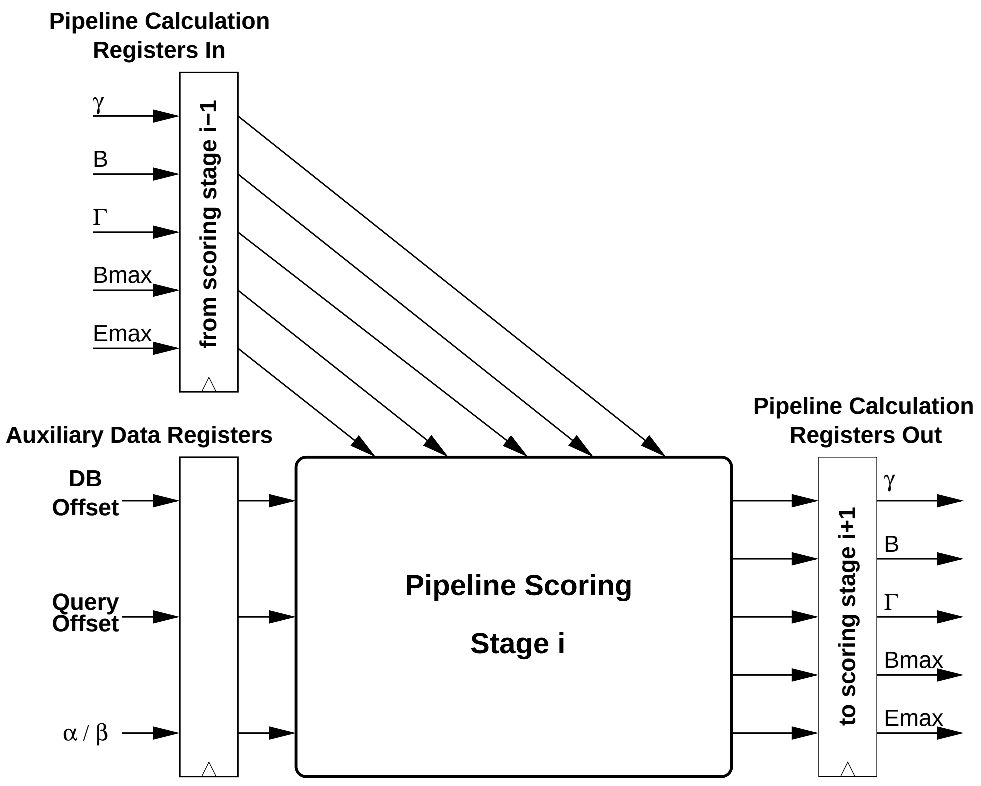

Figure 3.11: Detailed view of an individual scoring stage.

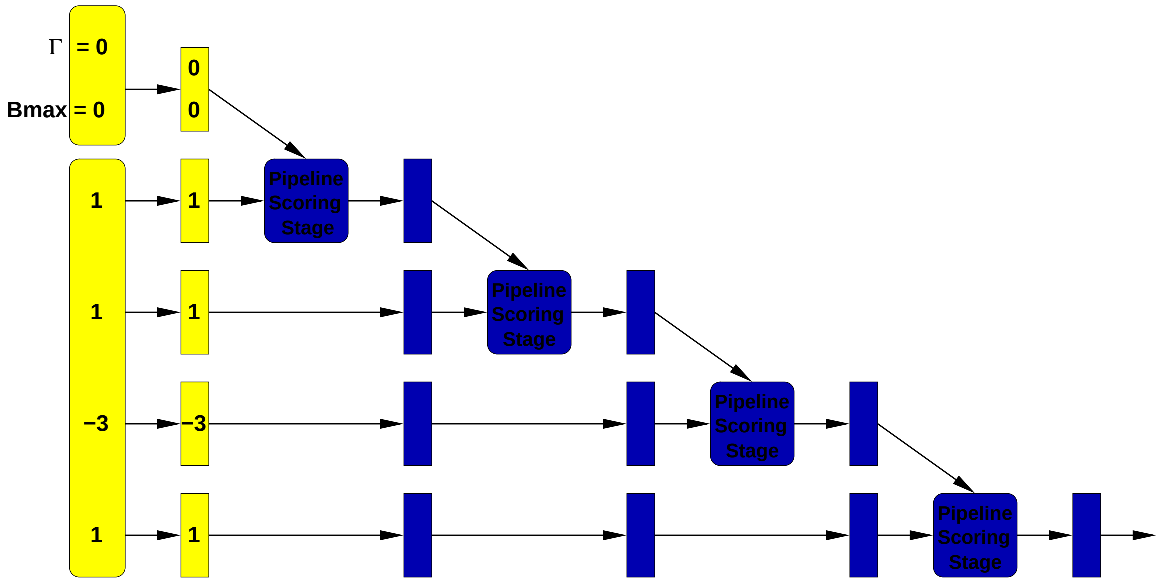

Figure 3.12: Example of a single w-mer propagating through the systolic array. First, the recurrence variables (only 2 of which are shown) are initialized, and the comparison score for each base is calculated. These values are then stored in registers.

result is stored in the output registers. After a base comparison score has been inspected it is never used again because of the unidirectional nature of the recurrence, so it is discarded. The recurrence is repeated until the final base comparison score is consumed, and the result is stored in registers. If Γ is greater than a user-defined threshold, it is passed on for further inspection. Otherwise, it is discarded.

Figure 3.17 illustrates another possible configuration of Mercury BLASTN. Here, the hardware ungapped extension design described in this chapter is used as the only ungapped extension processing in the pipeline. Instead of being additionally filtered by the software, HSPs output from the hardware filter are processed directly in the software gapped extension stage. Eliminating the software ungapped extension stage can reduce the load on the CPU. However, at lower score thresholds, stage 2a can be a less stringent filter, leading to an increased number of attempted gapped alignments. In this case, the advantage of eliminating stage 2b is lessened. An advantage of not using software ungapped extension, however, is that a significant number of new alignments can be discovered that are dropped from the pipeline configuration in Figure 3.5. The performance effects of the two configurations are discussed in more detail in Chapter 4.

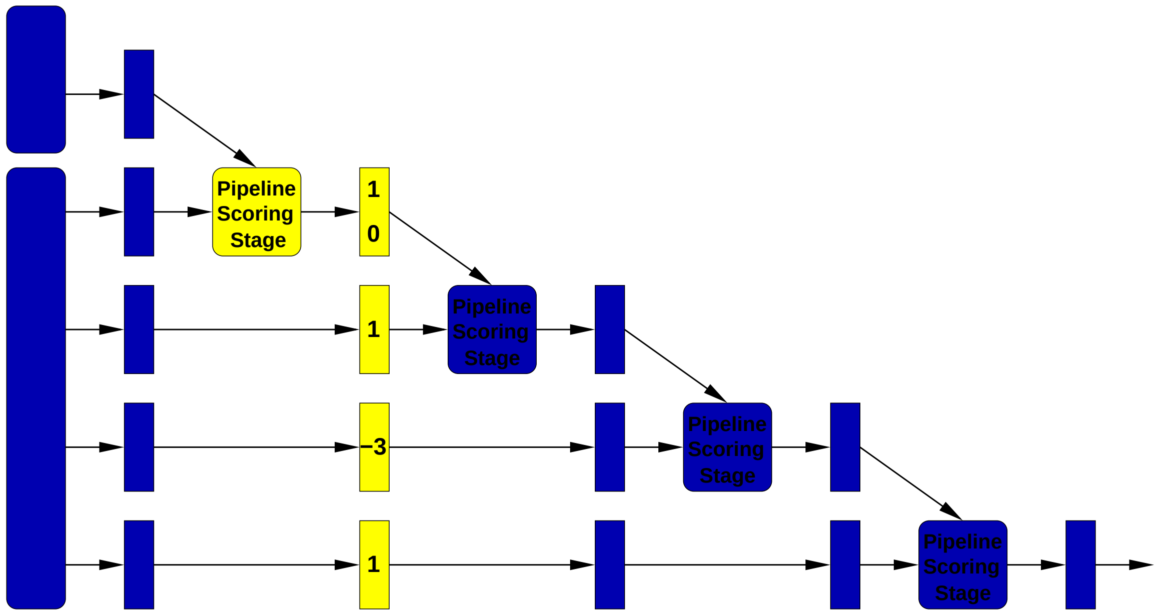

Figure 3.13: Example of a single w-mer propagating through the systolic array. The first step of the recurrence is performed, and the output is stored in registers.

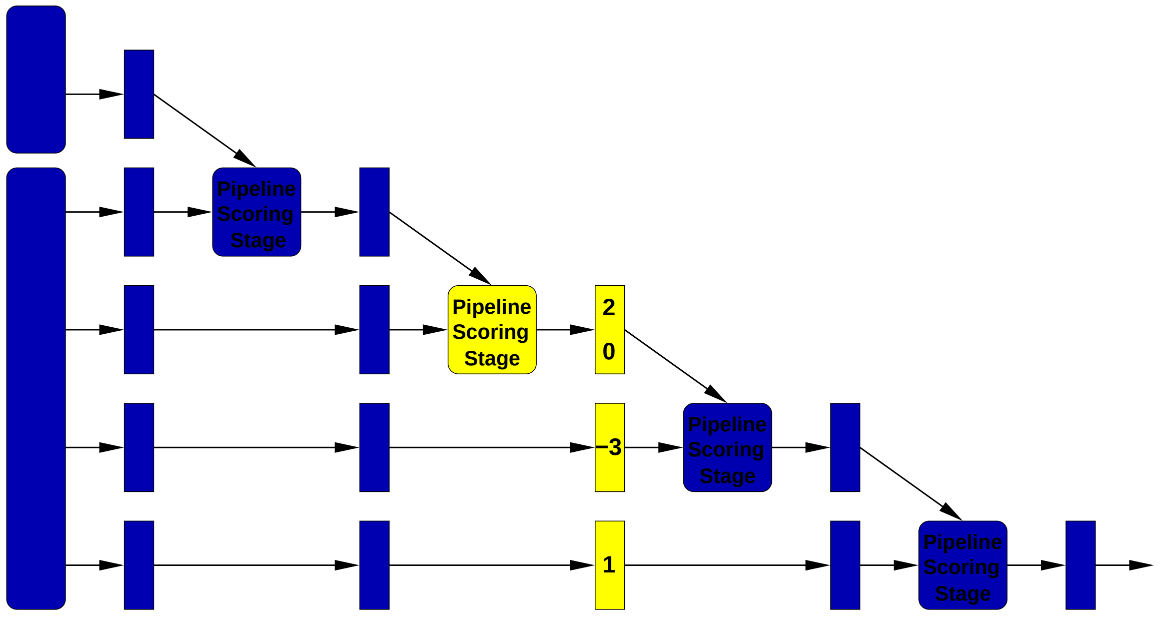

Figure 3.14: Example of a single w-mer propagating through the systolic array. The second step of the recurrence is performed, and the output is stored in registers.

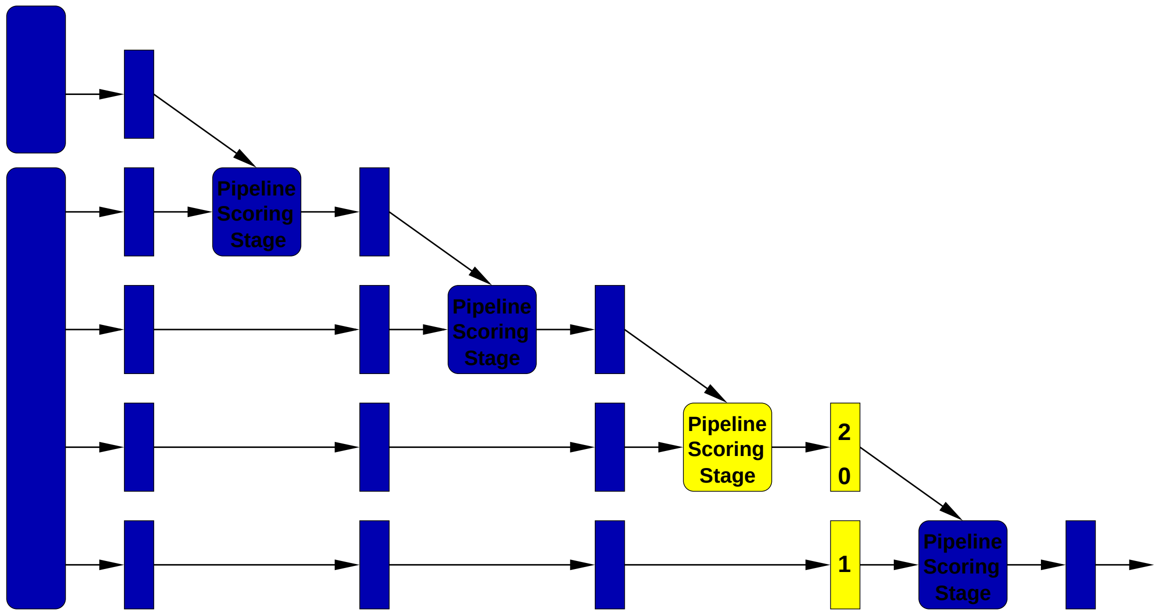

Figure 3.15: Example of a single w-mer propagating through the systolic array. The third step of the recurrence is performed, and the output is stored in registers.

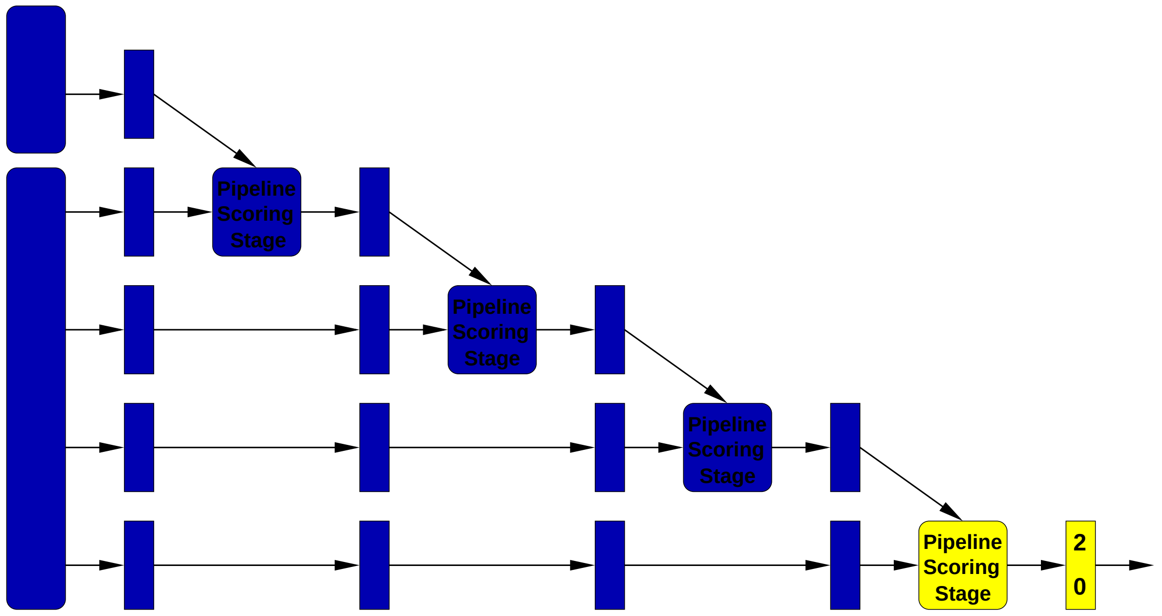

Figure 3.16: Example of a single w-mer propagating through the systolic array. The final step of the recurrence is completed, and the result is stored in registers.

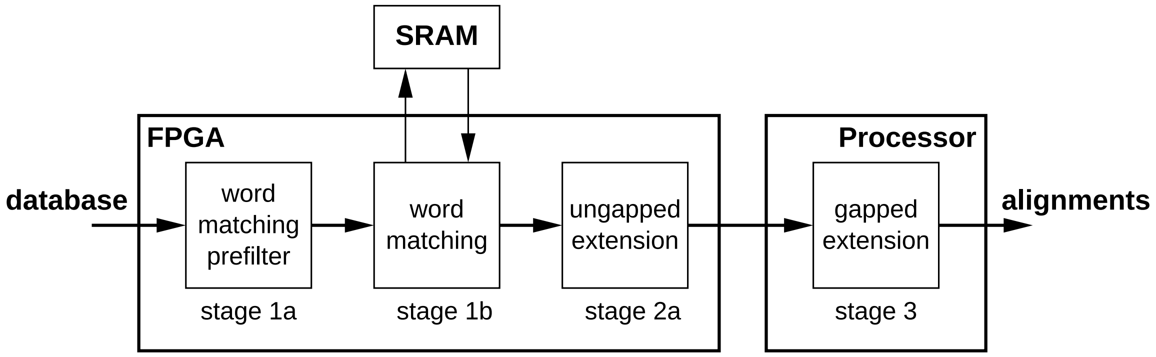

Figure 3.17: Alternative Mercury BLASTN hardware/software deployment, with no software ungapped extension is performed.

Chapter 4

Design Performance

This chapter evaluates the performance of the ungapped extension hardware filter and its effects on the entire system. There are many aspects to the performance of an ungapped extension filter. First, the accelerator must be able to do a similar enough computation that the quality of results is essentially equivalent to NCBI BLAST. Second, the ungapped extension filter must filter out as many w-mers as possible. Finally, the throughput of the ungapped extension stage must be high enough that it is not a bottleneck in the system. This chapter describes the performance of the hardware ungapped extension filter in terms of the above qualities. We give a characterization of the sensitivity of the filter, show the stringency of the ungapped extension stage, characterize the throughput in two different pipeline configurations, and conclude with some examples of typical FPGA resource utilization.

4.1 Quality of Results: Sensitivity

The quality of BLAST's results is measured primarily by its *sensitivity*, the number of statistically significant gapped alignments that it discovers. Because we are trying to improve the performance of BLAST without sacrificing sensitivity, we compare Mercury BLAST's sensitivity to that of the NCBI BLAST software, taking the latter as our standard of correctness.

Formally, sensitivity is defined as follows:

$$\text{Sensitivity} = \#\text{ New Alignments} \;/\; \#\text{ Original Alignments}\ ,$$

where "# New Alignments" is the number of statistically significant gapped alignments discovered by Mercury BLAST, and "# Original Alignments" is the number of similarities returned from NCBI BLAST given the same measure of significance. Measurements of sensitivity vary depending on how stringently the user chooses to filter NCBI BLAST's output. The numbers reported here correspond to a BLAST E-value of 10^{-5}, which is reasonably permissive for DNA similarity search.

Two parameters of the ungapped extension stage affect the quality of its output. First, the score cutoff threshold used affects the number of alignments that are produced. If the cutoff threshold is set too high, the filter will incorrectly reject a large number of statistically significant alignments. Conversely, if the threshold is set too low, the filter will generate many false positives and will negatively affect system throughput. Second, the length of the window can affect the number of false negatives that are produced. In particular, alignments that have enough mismatches before the window boundary to be below the score threshold but have many matches immediately outside the boundary will be incorrectly rejected. The larger the window size, the higher the score threshold that can be used without diminishing the quality of results.

We measured the sensitivity of BLASTN with our ungapped extension design using a modified version of the NCBI BLASTN code base. A software emulator of the new ungapped extension algorithm (stage 2a) was placed in front of the standard NCBI ungapped extension stage (stage 2b). In addition, another configuration was evaluated where the hardware ungapped extension stage was the only ungapped extension performed in the pipeline. We used BLAST with this emulator to compare sequences extracted from the human and mouse genomes. The queries were samples of the human genome of the following sizes: 100 10 kbase samples, 50 100 kbase samples, and 20 1 Mbase samples. The database stream was the mouse genome with low-complexity and repetitive sequences removed. Statistics were gathered for both which show how many w-mers arrived at each ungapped extension stage, and how many passed. These statistics were collected for three different configurations: NCBI BLASTN ungapped extension alone (the baseline configuration), the hardware emulator and NCBI ungapped extension combined to form the entire stage 2, and the hardware emulator as the only ungapped extension in the pipeline.

To compare the results of NCBI BLAST to those of Mercury BLAST, a Perl script was written to quantify how many gapped alignments produced by each implementation were also produced by the other. The test for whether an alignments exists in both sets is more complicated than a simple equality check, since the two BLASTs can produce highly similar (and equally useful) but non-identical alignments. Gapped alignments were therefore compared using the following *overlap* metric. For each alignment A output from one BLAST run, the overlap metric determines if any alignment A' from the other run overlaps A in at least a fraction f of its bases in each sequence. In our experiments, $f = 1/2$. If one gapped alignment overlaps another gapped alignment by more than f, it is considered to be the same alignment for purposes of the sensitivity measurement.

Figure 4.1 illustrates the sensitivity of the Mercury BLAST system when the hardware ungapped filter is combined with the NCBI ungapped extension filter to make up the full stage 2 (i.e., the configuration of Figure 3.5). Using a window size of 64 to 256 bases, the combined filters yielded sensitivity in excess of 99.6%; in absolute terms, the worst observed false negative rate was only 36 false negatives out of 11616 significant alignments. Larger window sizes resulted in higher sensitivity, up to 100%; however, increasing the window size above 96 yielded diminishing returns for the additional logic consumed. A window size of 64 reduced sensitivity below 99.9%, but this loss can be compensated by lowering the score threshold. The confidence intervals for window sizes greater than 64 are very small overall ($< 0.024\%$). For a window length of 64, the confidence interval increases significantly with score threshold. This is caused by a small number of samples differing significantly from the mean. For instance, one sample contains only 3 HSPs found by the original algorithm for a score threshold of 20 and the new algorithm is only finding 2 of them with a window length of 64 and score threshold of 20. The sensitivity for that sample is 66.67%, significantly farther from the mean than the rest of the data. We conclude that using the hardware prefilter in this configuration does not noticeably degrade the quality of results.

Figure 4.2 shows the sensitivity of a similar set of measurements, except that NCBI ungapped extension is not executed (i.e., the configuration of Figure 3.17). All hits that come out of the hardware filter are passed for processing in software gapped extension. The results show a similar trend to the case where both filters are used,

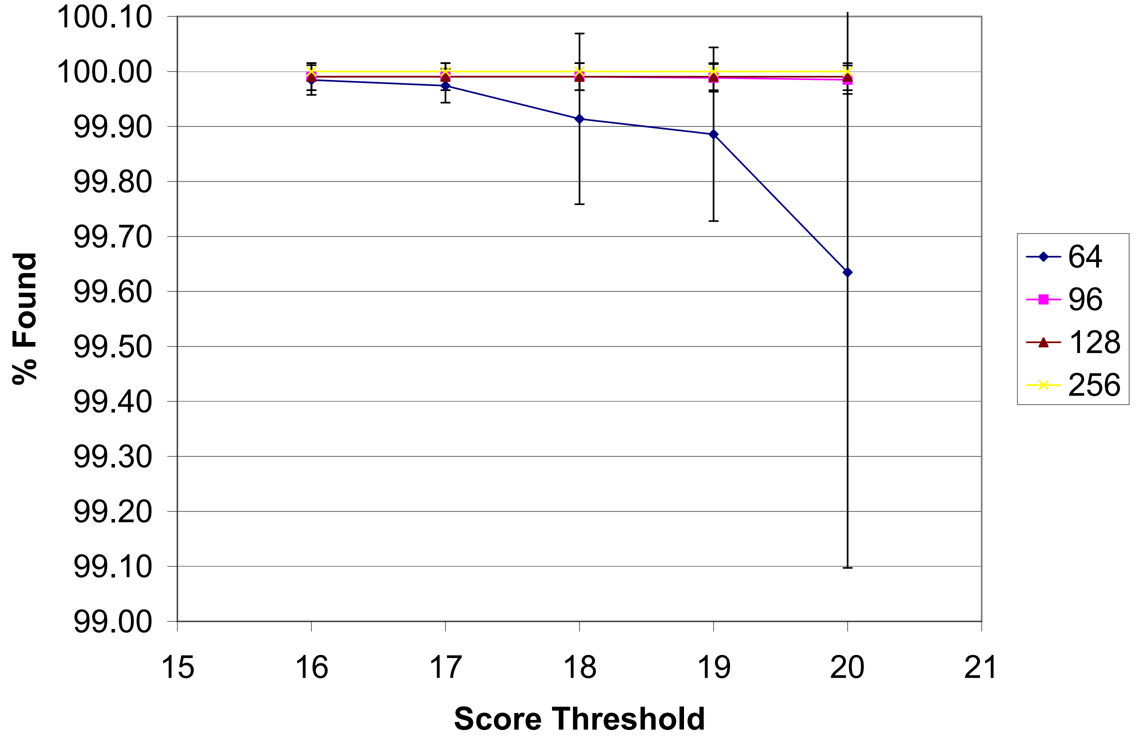

Figure 4.1: Mercury BLASTN sensitivity using the deployment shown in Figure 3.5. The four curves represent window lengths of 64, 96, 128, and 256 bases. Error bars represent 99% confidence intervals.

except that the sensitivity overall is slightly higher. The confidence intervals follow the same trends as in Figure 4.1.

Figure 4.3 shows sensitivity results from the same configuration, but with the newly discovered alignments included into the count. This configuration does have the advantage that new alignments can be discovered that were not included in the stock NCBI blast setup, raising the sensitivity above 100%. The confidence intervals follow the same trends as the ones in Figure 4.1 as well.

4.2 Efficiency of Filtration: Specificity

Specificity measures how effectively the ungapped extension stage discards insignificant or chance matches from its input. High specificity is desirable for computational efficiency, since fewer matches out of ungapped extension lowers the computational burden on software ungapped and gapped extension. An effective filter exhibits both high sensitivity and high specificity.

For BLASTN ungapped extension, specificity is measured as follows:

$$\text{Specificity} = 1 - (\#\ \text{HSPs out} \ / \ \#\ \text{matches in})\ ,$$

To quantify the specificity of our implementation, we gathered statistics during the aforementioned experiments on how many w-mers (matches) arrived at the ungapped extension stage, and how many of these produced ungapped alignments that passed the stage's score threshold.

Specificity can have a direct impact on system throughput, since the larger the volume of output from ungapped extension yields a longer runtime in software gapped extension. The next section will discuss the impact of specificity on system throughput.

Figure 4.4 shows the specificity of Mercury BLASTN ungapped extension for various score thresholds and window lengths. In this graph the ungapped extension stage consists of the hardware filter alone (i.e., stage 2a), *without* NCBI BLAST's software

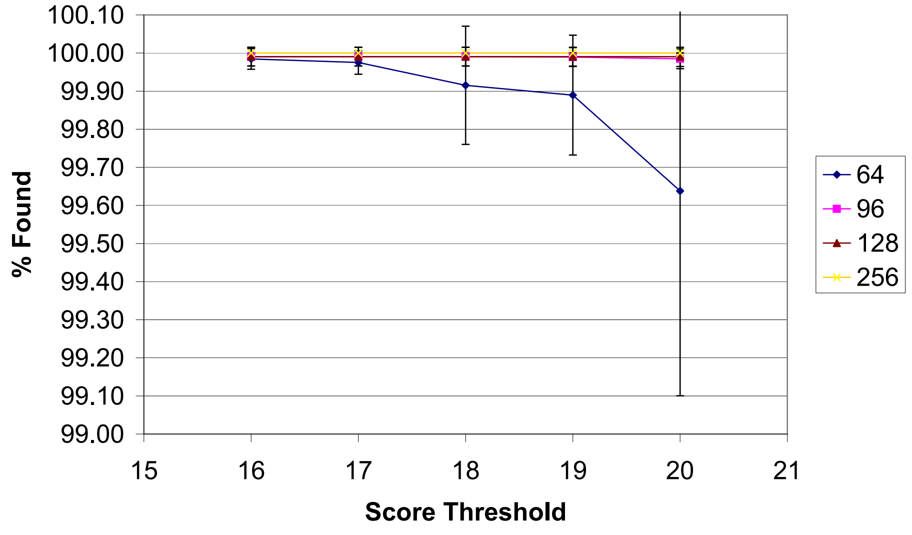

Figure 4.2: Mercury BLASTN sensitivity for the configuration shown in Figure 3.17 (i.e., *only* the hardware prefilter) not including newly discovered alignments. The four curves represent window lengths of 64, 96, 128, and 256 bases. Error bars represent 99% confidence intervals.

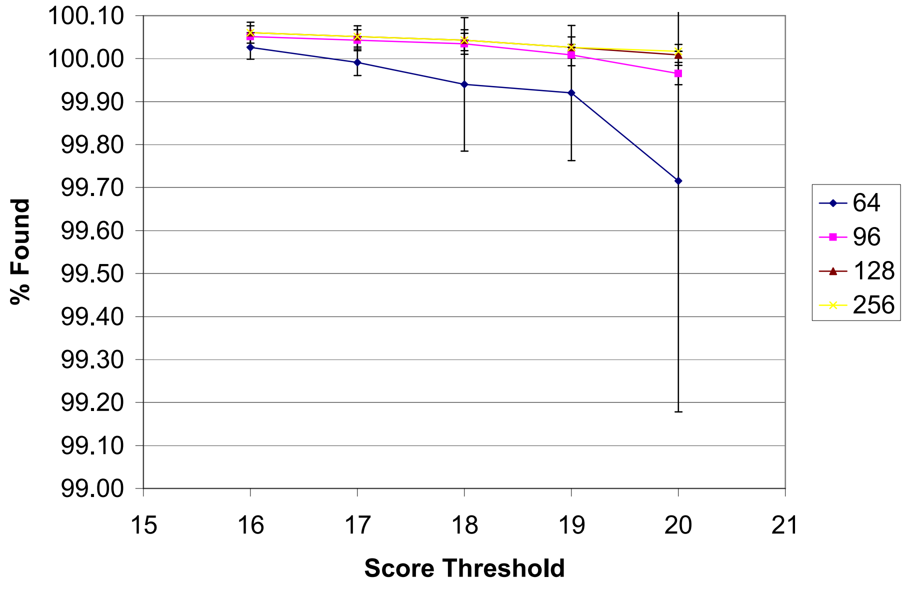

Figure 4.3: Mercury BLASTN sensitivity for the same configuration shown in Figure 3.17 *and* including newly discovered alignments in the count. The four curves represent window lengths of 64, 96, 128, and 256 bases. Error bars represent 99% confidence intervals.

ungapped filter. As the score threshold increases, the hardware passes fewer w-mers, and so the specificity of the filter increases. Specificity is not strongly influenced by window size. At a score threshold of 18, the hardware prefilter is approximately as specific as the original NCBI ungapped extension stage.

Figure 4.5 shows the specificity of the combined hardware filter and software ungapped extension filter. As expected the specificity is essentially constant, with a minuscule increase at the highly stringent score threshold of 20.

4.3 Performance

Since the goal of this thesis is to develop a BLAST accelerator, overall throughput is important. Because there are other, more computationally expensive stages downstream, the filter's stringency needs to be as high as possible. Second, high throughput must be achieved without inadvertently dropping a large percentage of the significant alignments (i.e., the false negative rate must be minimal), as described in earlier sections. This section describes the performance with respect to throughput and speedup for the ungapped extension stage consisting of the hardware filter only, with both hardware and software ungapped extension, and finally for the entire Mercury BLASTN application.

The throughput of the Mercury BLASTN ungapped extension prefilter is a function of the data input rate. The ungapped extension hardware stage accepts one w-mer per clock and runs at 100 MHz on a current FPGA. Hence the maximum throughput of the prefilter is $Tp_2 = 1$ input match/cycle $\times$ 100 MHz = 100 Mmatches/second. This gives a speedup of 25$\times$ over the software ungapped extension executed on the baseline system described earlier.

We now return to the performance graphs given in Chapter 2 in more detail. Figure 4.6 illustrates the throughput of the system with various speedups in stage 2a. These throughput numbers are given for the pipeline configuration of Figure 3.17. As stated in Chapter 2, the throughput of the entire system is directly dependent on the speedup that is achieved in stage 2. The throughput peaks at 1,400 Mmatches/second where the system is I/O limited by the PCI-X bus. In terms of stage 2a performance,

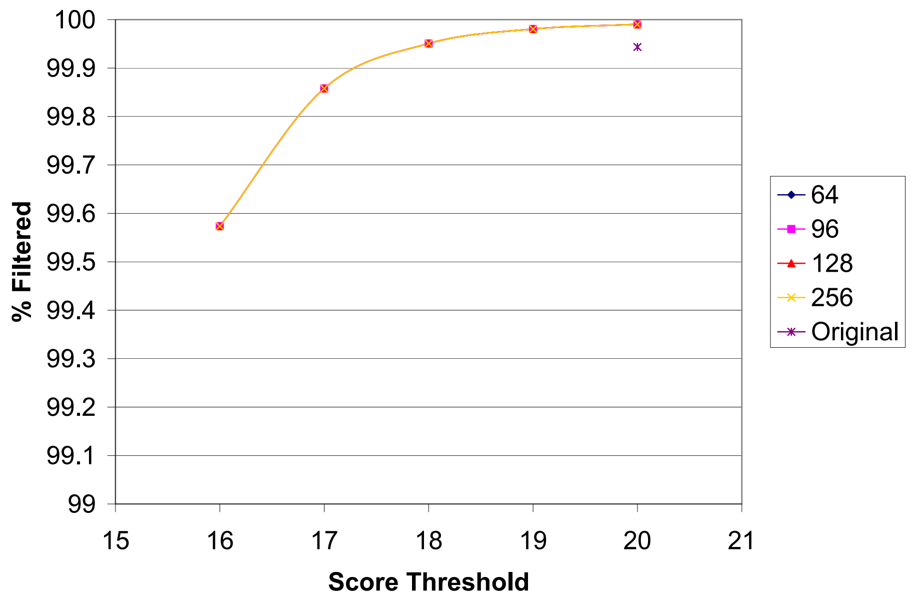

Figure 4.4: Mercury BLAST specificity for stage 2a alone. The four curves represent window lengths of 64, 96, 128, and 256 bases. The individual point represents the value for NCBI BLAST stage 2. 99% confidence intervals are less than 0.0001%.

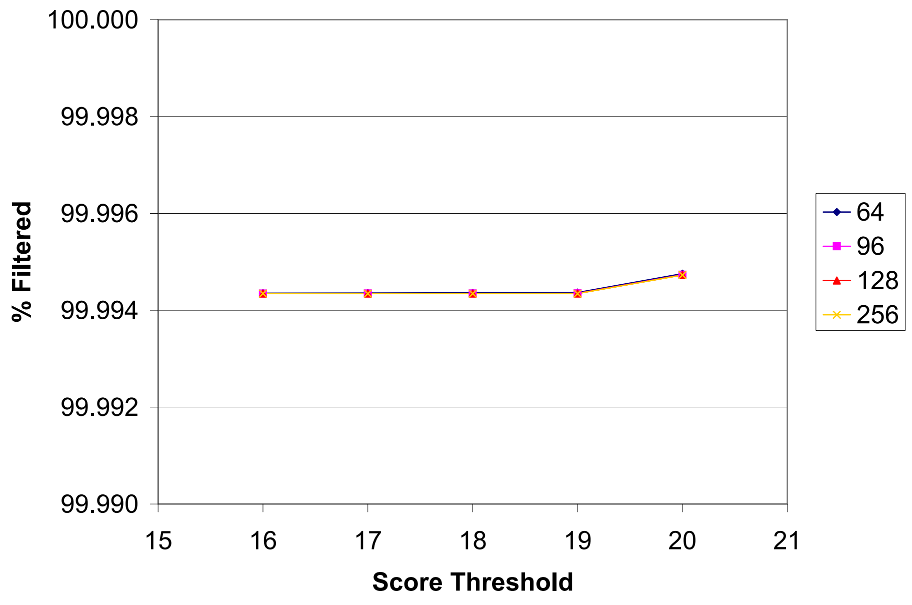

Figure 4.5: Mercury BLAST sensitivity for complete stage 2. The four curves represent window lengths of 64, 96, 128, and 256 bases. 99% confidence intervals are less than 0.0001%.

this corresponds to a stage 2a throughput of 25 - 33 Mmatches/second depending on the supported query size. The design capability is currently to the far right of the graph, supporting an ingest rate of 100 Mmatches/second.

Figure 4.7 plots the speedup of the Mercury BLASTN accelerator as a function of stage 2a throughput. The results here show a very similar trend to that Figure 4.6, except that the maximum speedups achieved are substantially different for the two query sizes. This is because the throughput of the baseline system is significantly different for the two different sizes; however, the throughput of our hardware system is essentially constant up to our maximum supported query size. If the query is larger than the maximum supported size, the query is divided into multiple parts and processed in more than one run. Stage 2a throughputs above around 33 Mbases/second result in maximum system speedup of approximately 48× the baseline software system. As mentioned above, the stage 2a design easily supports this maximum speedup.

The previous performance model assumes that the only other resource executing after hardware ungapped extension is stage 3. Since there are other possible deployments of Mercury BLASTN, we now develop a new performance model which assumes that NCBI ungapped extension is executed in software after stage 2a. To explore the impact that stage 2a performance has on the overall streaming application when used as a prefilter, we use the following mean-value performance model. Overall pipeline throughput for the deployment of Figure 3.5 is

$$Tput_{overall} = \min(Tput_1, Tput_{2a}, Tput_{2b3}),$$

where $Tput_1$ is the maximum throughput of stage 1 (both 1a and 1b) executing on the FPGA, $Tput_{2a}$ is the maximum throughput of stage 2a executing on the FPGA (concurrently with stage 1), and $Tput_{2b3}$ is the maximum throughput of stages 2b and 3 executing on the processor.

For the above expression to be correct, all of the throughputs must be normalized to the same units; we will normalize to input DNA bases per unit time. This normalization can be accomplished with knowledge of the fractions of input bases that survive each of the stages of filtering. Call the w-mers from stage 1 "matches" and

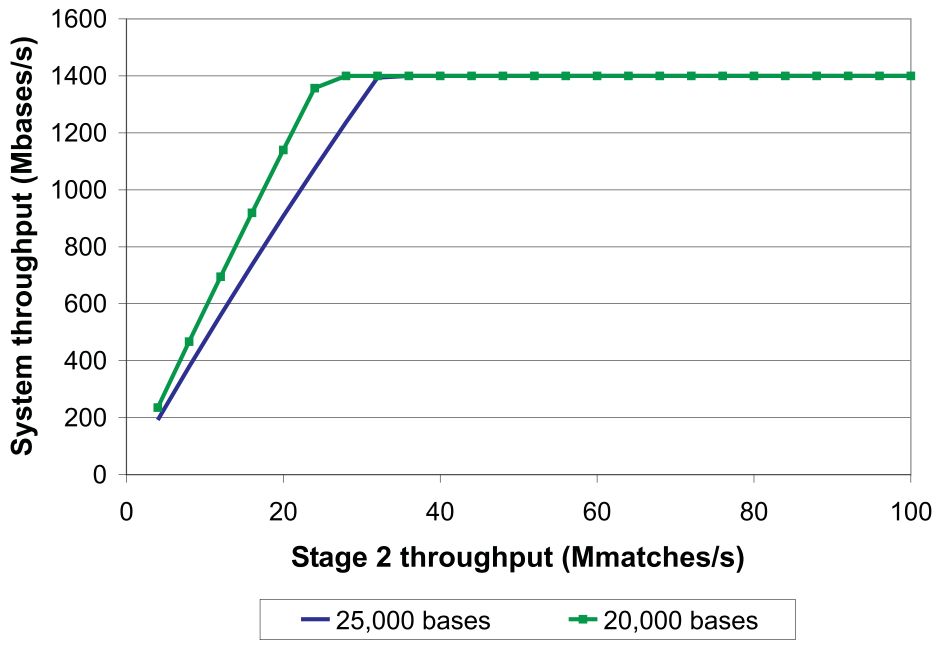

Figure 4.6: Throughput of overall pipeline as a function of ungapped extension throughput for queries of sizes 20 kbases and 25 kbases.

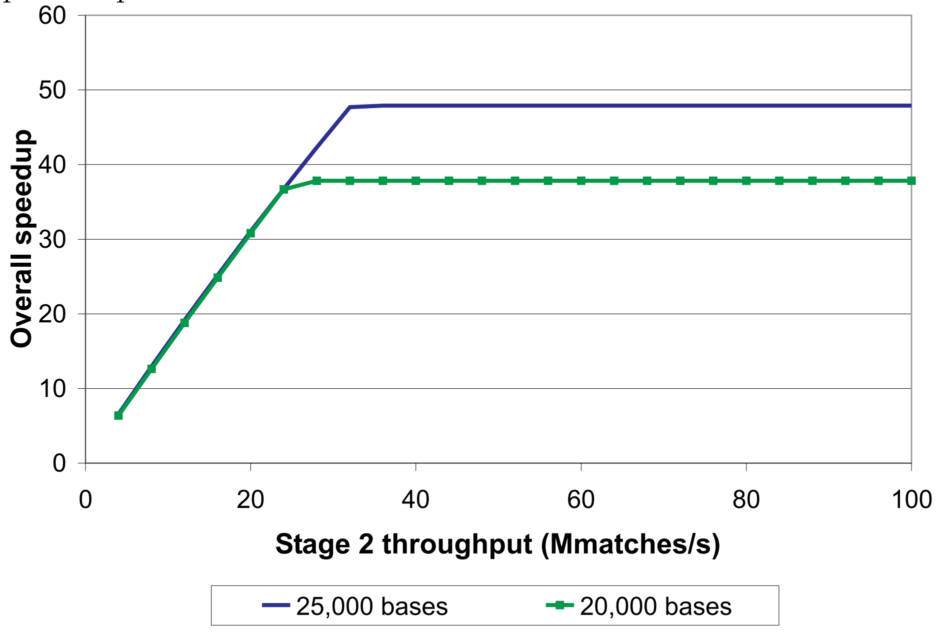

Figure 4.7: Speedup of overall pipeline as a function of ungapped extension throughput for queries of sizes 20 kbases and 25 kbases.

Table 4.1: Performance model parameters. Query size is 25 kbases (double stranded), and the pass fractions for stages 2a and 2b are with the most permissive cutoff score of 16.

Parameter	Value	Units	Meaning
p_1	0.0205	matches/base	stage 1 pass fraction [13]
p_{2a}	0.0043	HSPs/match	stage 2a pass fraction
p_{2b}	0.0133	HSPs out/HSPs in	stage 2b pass fraction
t_{2b}	0.265	μsec/HSP input	stage 2b execution time [13]
t_3	60.4	μsec/HSP input	stage 3 execution time [13]
$Tput_1$	1.4	Gbases/sec	stage 1 throughput [13]
$Tput_{2a}$	4.9	Gbases/sec	stage 2a throughput
$Tput_{2b3}$	10.6	Gbases/sec	processor throughput
$Tput_{overall}$	1.4	Gbases/sec	overall pipeline throughput

the HSPs from stages 2a and 2b "alignments." We define p_1 as the pass fraction from stage 1 (matches out per base in), p_{2a} as the pass fraction from stage 2a (alignments out per match in), and p_{2b} as the pass fraction from stage 2b (alignments out per alignment in). With this information, we compute the normalized throughput of stage 2a as $Tput_{2a} = (100 \text{ Mmatches/sec})/p_1$. Finally, we need the time required to process alignments in software (both in stage 2b and in stage 3). We define t_{2b} as the execution time per input alignment for the stage 2b software and t_3 as the execution time per input alignment for the stage 3 software. The normalized throughput of the stages executing on the software can then be expressed as

$$Tput_{2b3} = \frac{1}{p_1 p_{2a}(t_{2b} + p_{2b} t_3)}.$$

Table 4.1 provides the above parameters and their values for a 25 kbase, double-stranded query. The values of p_1, t_{2b}, t_3, and $Tput_1$ come from [13]. Clearly, the overall throughput is limited by the capacity of stage 1.

It it important to note that performance of stage 1 in the current implementation of Mercury BLASTN is limited by the input rate of the I/O subsystem. Hence, as newer interconnect technologies, such as PCI-Express, become more readily available, the throughput of our system will increase significantly. It is likely that the next generation of the Mercury system will use PCI-Express to deliver even higher throughput to the hardware accelerator. Since the downstream pipeline stages are clearly capable of sustaining higher throughputs, any improvement in stage 1 throughput will translate directly into greater overall throughput.

Given the clear ability of stage 2a to surpass the minimum performance needs of the pipeline, it is useful to revisit the advantages of the different pipeline configurations shown in Figure 3.5 and 3.17. Having stage 2a substantially faster than necessary allows flexibility in the deployment. If there are tight hardware area constraints, a smaller window length with a lower threshold can be used in conjunction with stage 2b to offload some of the ungapped extension processing to software without decreasing sensitivity or slowing down the pipeline. If there are sufficient hardware resources available, it can be advantageous to offload as much of the software computation as possible. This will free up more of the CPU for post-processing.

We must be cautious when interpreting the above performance model. There is significant software pre-processing and post-processing that must be performed to setup a Mercury BLASTN run. For instance, if the query is too large, it must be split into smaller queries that will fit in the system with a small amount of padding on each to make sure that alignments on the query boundaries are not lost. In other cases, query packing may need to be done if there are lots of small queries to be processed. Also, a hash table must be generated for each query that is processed, among other peripheral tasks. Finally, the processing and formatting of the output for viewing is not included in this performance model.

4.4 Resource Utilization

As mentioned in the previous chapter, the Mercury ungapped extension stage is parameterizable and can be configured for different window lengths. Currently, the filter exists with window lengths of 64, 96, and 128 bases. Table 4.2 gives the resource usage for each of these design points. For comparison, the full Mercury BLASTN

Table 4.2: FPGA resource usage and utilization of the hardware ungapped extension stage in isolation. The three rows show the resource usage for window sizes of 64, 96, and 128 bases on a Xilinx Virtex-II 6000 FPGA.

Window Size	Slices Used (% Utilized)	Block RAMs Used (% Utilized)
64	9174 (27%)	13 (9%)
96	11700 (35%)	18 (12%)
128	15226 (45%)	18 (12%)

design, including both stages 1 and 2a, utilizes approximately 54% of the logic cells and 134 Block RAMs of our FPGA platform with a stage 2a window size of 64 bases.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

With an exponential increase in genetic information available, the need for faster biosequence search methods are evident. Entire mammalian genomes are being sequenced leading to increasingly longer search queries. Biosequence similarity search can be accelerated practically by processors designed to filter high-speed streams of character data. This thesis describes a portion of our Mercury BLASTN search accelerator, focusing on the performance-critical ungapped extension stage.

To address this problem, we created and evaluated a new algorithm for ungapped extension in BLAST. The algorithm was designed to be a prime candidate for acceleration in hardware. Our highly parallel and pipelined implementation of this algorithm yields quality of results comparable to those obtained from software BLASTN while running over 20× faster than software ungapped extension alone. The design is fast and compact, clocking at over 100 MHz on a current FPGA and consuming less than 30% of the logic gates available. Accelerating ungapped extension to this degree enables the entire Mercury BLASTN accelerator to run approximately 50× faster than a standard PC. The design exists in working silicon and has been tested with other stages of Mercury BLASTN on a single Xilinx Virtex II FPGA.

5.2 Future Direction

5.2.1 Improved BLASTN

To further improve Mercury BLASTN, many enhancements are planned. First, the software processing time must be optimized. Currently, the post-processing that BLAST performs to collect and format the output has not been accelerated. To get good overall performance this needs to be addressed, since post-processing can take more than 15% of the total execution time. Also, the ungapped extension stage may be configured to perform more iterations of the dynamic-programming recurrence in a single clock cycle. Since Mercury ungapped extension stage is clearly not a bottleneck, a slower clock speed and a long window length can be used to improve the sensitivity to higher than that of NCBI BLASTN.

5.2.2 BLASTP

The BLASTP pipeline spends even more time executing the ungapped extension stage. Our ungapped extension design is suitable not only for BLASTN but also for other forms of BLAST, particularly the BLASTP algorithm used on proteins, for other applications of ungapped sequence alignment. Porting the current implementation to BLASTP requires support for more bits per character (5, vs. 2 for DNA) and a richer scoring function for individual character pairs; however, it requires essentially no further changes. This stage in used our within our in-progress design for Mercury BLASTP and expect that it will prove similarly successful in that application.

Appendix A

Module Command Reference

As mentioned in Chapter 3, the hardware ungapped extension stage accepts a number of commands. Some of the commands, global commands, are general commands which are used for all stages, including the infrastructure for moving data in and out of the hardware. Other commands, module-specific commands, are custom commands intended to be interpreted correctly by a particular module. All commands are encoded as two ASCII characters, shown below in parenthesis after the names of the commands. Every module supports global commands, and most modules have individualized commands to configure various aspects of a design at runtime. The listing below shows all the global commands, as well as the custom commands for ungapped extension, along with a brief description of their meaning.

Global Commands:

- **Reset (RS)**

  Resets either the entire module chain, or an individual module, depending on the ID field of the command.

- **Query (QY)**

  Requests status information from all modules, or an individual module, depending of the ID field of the command. Each module queried will respond with on or more Query Response command.

- **Query Response (QR)**

  A command that is generated in response to a Query command. This command is what is received to the end used to indicate the status of one or more modules.

- **Passthrough (PS)**

  Forces one or more modules to enter debug mode, where all input is passed through the module(s) unchanged. This command is useful for sanity-checking the software infrastructure.

- **Start of Data (SD)**

  This command informs the modules that a new data stream is incoming.

- **End of Data (ED)**

  This command informs the modules that the end of a data stream has been reached.

Module-specific Commands:

- **Start of Query (SQ)**

  Start of Query marks the beginning of an incoming query stream.

- **End of Query (EQ)**

  Marks the end of the query stream.

- **Set Parameters (SP)**

  This command is used to set the word length, match score, mismatch score, and score threshold for stage 2a.

- **Query Length (QL)**

  Sets the length of the query in the module.

- **Database Length (DL)**

  Sets the length of the database in the module.

- **Start of Database (SB)**

  Indicates that the database stream is incoming.

References

[1] S. F. Altschul and W. Gish. Local alignment statistics. *Methods: a Companion to Methods in Enzymology*, 266:460–80, 1996.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, et al. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–10, 1990.

[3] S. F. Altschul, T. L. Madden, A. A. Schäffer, 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*, 25:3389–402, 1997.

[4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, May 1970.

[5] M. Cameron, H. Williams, and A. Cannane. Improved gapped alignment in blast. *IEEE Transactions on Computational Biology and Bioinformatics*, 1(3):116–14, 2004.

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

[7] R. K. Singh et al. BioSCAN: a dynamically reconfigurable systolicarray for biosequence analysis. In *Proceedings of CERCS 96*, 1996.

[8] J. D. Hirschberg, R. Hughley, and K. Karplus. Kestrel: a programmable array for sequence analysis. In *Proceedings of IEEE International Conference on Application-specific Systems, Architecture, and Processors*, pages 23–34, 1996.

[9] D. T. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–91, 1995.

[10] W. J. Kent. BLAT: the BLAST-like alignment tool. *Genome Research*, 12:656–64, 2002.

[11] G. Knowles and P. Gardner-Stephen. DASH: localizing dynamic programming for order of magnitude faster, accurate sequence alignment. In *Proceedings of the 3rd International IEEE Computer Society Computational Systems Bioinformatics Conference*, pages 732–35, 2004.

[12] G. Knowles and P. Gardner-Stephen. A new hardware architecture for genomic and proteomic sequence alignment. In *Proc. of IEEE Computational Systems Bioinformatics Conf.*, 2004.

[13] P. Krishnamurthy, J. Buhler, R. D. Chamberlain, M. A. Franklin, K. Gyang, and J. Lancaster. Biosequence similarity search on the Mercury system. In *Proc. of the 15th IEEE International Conf. on Application-Specific Systems, Architectures, and Processors*, pages 365–375, 2004.

[14] E. S. Lander et al. Initial sequencing and analysis of the human genome. *Nature*, 409:860–921, 2001.

[15] D. Lavenier, S. Guytant, S. Derrien, and S. Rubin. A reconfigurable parallel disk system for filtering genomic banks. In *ERSA'03, Engineering of Reconfigurable Systems and Algorithms*, 2003.

[16] M. Li, B. Ma, D. Kisman, and J. Tromp. Patternhunter II: highly sensitive and fast homology search. *Journal of Bioinformatics and Compuational Biology*, 2:417–39, 2004.

[17] National Center for Biological Information. Growth of GenBank, 2002. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[18] Z. Ning, A. J. Cox, and J. C. Mullikin. SSAHA: a fast search method for large DNA databases. *Genome Research*, 11:1725–9, 2001.

[19] T. Oliver, B. Schmidt, and D. Maskell. Hyper customized processors for biosequence database scanning on FPGAs. In *Proc. of ACM/SIGDA 13th Int'l Symp. on Field-Programmable Gate Arrays*, pages 229–237, February 2005.

[20] N. Pappas. Searching biological sequence databases using distributed adaptive computing. Master's thesis, Virginia Polytechnic Institute and State University, 2003.

[21] Paracel, Inc. http://www.paracel.com.

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

[23] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–97, March 1981.

[24] TimeLogic Corporation. http://www.timelogic.com.

[25] R. H. Waterston et al. Initial sequencing and comparative analysis of the mouse genome. *Nature*, 420:520–562, 2002.

[26] B. West, R. D. Chamberlain, R. S. Indeck, and Q. Zhang. An FPGA-based search engine for unstructured database. In *Proc. of 2nd Workshop on Application Specific Processors*, pages 25–32, December 2003.

[27] Y. Yamaguchi, T. Maruyama, and A. Konagaya. High speed homology search with FPGAs. In *Pacific Symposium on Biocomputing*, pages 271–282, 2002.

[28] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7:203–14, 2000.

Vita

Joseph M. Lancaster

Date of Birth	April 26, 1980
Place of Birth	Athens, Tennessee
Degrees	B.S. Electrical Engineering, May 2003 B.S. Computer Engineering, May 2003
Professional Societies	Institute of Electrical and Electronic Engineers
Publications	P. Krishnamurthy, J. Buhler, R. D. Chamberlain, M. A. Franklin, K. Gyang, and J. Lancaster. Biosequence similarity search on the Mercury system. In *Proc. of the 15th IEEE International Conf. on Application-Specific Systems, Architectures, and Processors*, pages 365–375, 2004. J. Lancaster, J. Buhler, R. D. Chamberlain. Acceleration of Ungapped Extension in Mercury BLAST. In *Proc. of the 7th Workshop on Media and Streaming Processors*, November, 2005.

May 2006

Short Title: Design of an Ungapped Extension Accelerator   Lancaster, M.S. 2006