Report Number: WUCSE-2006-47

2006-01-01

Acceleration of Gapped Alignment in BLASTP Using the Mercury System

Brandon B. Harris

Protein databases have grown exponentially over the last decade. This exponential growth has made extracting valuable information from these databases increasingly time consuming. This project presents a new method of accelerating a commonly used program for performing similarity searching on protein databases, BLASTP. This project describes the design and implementation of Mercury BLASTP, a customized hardware accelerated variant of BLASTP. This project focuses on the gapped alignment stage of Mercury BLASTP and provides design details and implementation results.

2006-47

Acceleration of Gapped Alignment in BLASTP Using the Mercury System

Authors: Brandon B. Harris

Corresponding Author: bbh2@wustl.edu

Abstract: Protein databases have grown exponentially over the last decade. This exponential growth has made extracting valuable information from these databases increasingly time consuming. This project presents a new method of accelerating a commonly used program for performing similarity searching on protein databases, BLASTP. This project describes the design and implementation of Mercury BLASTP, a customized hardware accelerated variant of BLASTP. This project focuses on the gapped alignment stage of Mercury BLASTP and provides design details and implementation results.

Type of Report: Other

Acceleration of Gapped Alignment in BLASTP Using the Mercury System

Brandon Harris
bbh2@wustl.edu

Abstract

Protein databases have grown exponentially over the last decade. This exponential growth has made extracting valuable information from these databases increasingly time consuming. This project presents a new method of accelerating a commonly used program for performing similarity searching on protein databases, BLASTP. This project describes the design and implementation of Mercury BLASTP, a customized hardware accelerated variant of BLASTP. This project focuses on the gapped alignment stage of Mercury BLASTP and provides design details and implementation results.

1 Introduction

Proteomics, the study of protein structure and function, is an important part of biology. Advances in technology have allowed biologists to sequence proteins at an ever increasing rate. By comparing a newly sequenced protein to large protein databases, biologists are able to find other, similar proteins, which may help them understand the function and evolution of the newly sequenced protein. Proteins with similar subsequences are likely to have similar structure and function. Protein databases like NCBI GenBank have grown at an exponential rate over the last decade, doubling in size every 12-16 months[2] . This dramatic growth in size has made searching for similar proteins increasingly difficult.

The similarity of two proteins is based on the edit distance between them. Each protein is represented as a sequence of amino acids, or *residues*. There are 20 different residues as well as several degenerate symbols which can represent more than one residue. Each residue is represented by a letter in the alphabet (eg, MDLLYRVKT). The edit distance measures the number of substitutions, insertions, and deletions required to transform one sequence to another. Calculating the edit distance for all pairings of sequences is computationally expensive and impractical, so biologists often use a tool called BLAST, the **B**asic **L**ocal **A**lignment **S**earch **T**ool [6][7]. BLAST uses heuristics to accelerate the search process by identifying regions in proteins which are likely to be similar. BLAST is capable of many different types of genomic searches, but we will concentrate on BLASTP, the version for comparing protein sequences to protein databases.

Even with the speed-up BLASTP offers, protein searches can take days or weeks. One approach to alleviate this problem is to use dedicated hardware to accelerate or replace parts

of the BLAST computation. Examples of this approach include the Paracel GeneMatcher and the TimeLogic DecypherBLAST engine [3][4]. The GeneMatcher system was an ASIC-based system which quickly became obsolescent, and the DecypherBLAST engine based on FPGAs is still commercially available.

Here we will present part of a new design based on the *Mercury* system to accelerate BLASTP. Mercury BLAST is designed as a high speed streaming pipeline utilizing the high bandwidth I/O of modern computer systems [10]. Using an FPGA to exploit both the sequentiality and fine grain parallelism of the BLAST computation, Mercury BLAST is able to achieve 1-2 orders of magnitude of speed up over software BLAST. The Mercury system is suitable for deployment on a server or on a laboratory workstation. Mercury BLASTP is a multi-stage stage pipeline which performs successively more stringent and computational complex filters to find similarities between protein. Here we present a successful design for the third stage of the pipeline which computes the score of a *gapped alignment*. I conceived and implemented a design which is highly parallel, has a single clock cycle recurrence and offers high sensitivity. I built a software simulator to verify the results of the hardware and measured the performance and sensitivity of the hardware.

The remainder of the paper is organized as follows. Section 2 gives a background on sequence alignment, the Smith-Waterman algorithm, the BLAST computation, and a banded variant of the Smith-Waterman algorithm. Section 3 provides an overview of the Mercury BLASTP system and gives design details for the gapped alignment stage of Mercury BLASTP. Section 4 presents the performance and sensitivity of our implementation, and Section 5 concludes.

2 Background

2.1 Sequence Alignment

An alignment is a relationship between two sequences or subsequences where in each column, a character in one sequence is paired to a corresponding character in the other or to nothing at all. A pairing is considered a *match* if the two characters are the same and a *substitution* if they differ. The goal of sequence alignment is to find the alignment with the most matches and fewest substitutions and unmatched characters.

If we define a scoring function Ψ which defines the value of each possible match or substitution, then we can assign a score to an alignment. For example, suppose we define Ψ such that matching vowels score +2, matching consonants or spaces score +1 and all substitutions score -1. Consider an example with the following two sequences:

```
Jack went up the hill
Jack went to the hill
```

The highest scoring, or *optimal*, alignment pairs the first letter of the first sequence to the first letter of the second sequence, and each correspond letter thereafter. The score of this alignment is +8 for matching vowels, +11 for matching consonants, +4 for matching spaces and -2 for substituting "to" for "up", for a total of +21. We can write this alignment with the substitutions marked in gray.

An issue we must consider is that these sequences may be part of larger sequences. Consider the example:

```
Jack went up the hill to fetch a pail
On Sunday Jack went to the hill
```

These sentences are related, but only locally. The first example produced a global alignment, where all the characters were paired to a corresponding character. We can define a local alignment, which is the alignment of two subsequences. Local alignments are more suitable than global alignments for comparing sequences of different length, and they can be used to show regions of similarity between two sequences. The highest scoring local alignment of the above two sentences is:

```
          Jack went up the hill to fetch a pail
On Sunday Jack went to the hill
```

This alignment scores the same as the first example, or +21. The substituted characters are written in dark gray and the characters not used by the alignment are written in light gray for the sake of example but are usually not show in the alignment at all.

The preceding two examples are examples of *ungapped* alignment, that is each character was paired to a corresponding character. This approach fails to identify similar regions if any characters are inserted or deleted. Consider the following two sentences:

```
Jack went up the hill
Jack went down the hill
```

Using ungapped alignment, there is no way to make both "Jack went" and "the hill" align. The optimal ungapped local alignment of these two sequences is +12 for matching "Jack went ".

To accommodate the insertion or deletion of characters, we introduce *gaps* into one of the sequences. A character can be paired with a gap rather than pairing with a character from the other sequence. We define a function Γ which yields the cost of a gap. For this example, we define *Gamma* as -2 for each gap introduced. Using Γ and Ψ, we find that the optimal *gapped alignment* is:

```
Jack went up-- the hill
Jack went down the hill
```

Where each "-" represents an insertion in first sequence. Note that an insertion in the first sequence is the same as a deletion is the second; here "d" and "n" are effectively deleted because they are not paired with any character. This alignment scores +23 for the matching characters, -2 for the substitutions, and -4 for the two gaps, for a total of +17.

2.1.1 Protein Sequence Alignments

The purpose of protein alignment is to find proteins which share regions of similarity. Regions of similarity are likely to have similar function and may indicate that the proteins have a common ancestor. Because biologists are interested in the regions, local alignments are performed. For proteins alignments, Ψ can represent the relative probability that two

residues will match or mismatch if the proteins shared a common ancestor. Thus by finding the highest scoring alignments between two protein sequences we can determine the likelihood that they shared an ancestor and how they have diverged. Ungapped alignments are sufficient for finding closely-related proteins but are less likely to detect distantly related proteins. Gapped alignments are more sensitive but also harder to compute. As an example consider the two protein sequences:

```
MDLLYRVKTLWAATPASWPG
MDELVKTLRAPTPVHTSWRR
```

Subject to the definition of Ψ the ungapped alignment of these two sequences could be:

```
AMDLLYRVKTLWAATPASWPG
  AMDELVKTLRAPTPVHTSWRR
```

Subject to the definition of Ψ and Γ the gapped alignment of these two sequences could be:

```
AMDLLYRVKTLWAATPA--SWPG
AMDEL--VKTLRAPTPVHTSWRR
```

2.2 Smith-Waterman

We will examine the Smith-Waterman (**SW**) algorithm used by BLASTP and Mercury BLASTP to calculate gapped sequence alignments[16]. SW is dynamic programing algorithm which is guaranteed to find a local alignment between two sequences which requires the fewest possible edits. SW is a very general algorithm, but we will only consider the most common variant, affine Smith Waterman[11]. Affine SW requires that the cost of a gap be expressed in the form of $o + k * e$ where o is the cost of a gap existing, k is the length of the gap and e is the cost of extending the gap length by 1. In practice o is usually costly, around -12, while e is less costly, around -3. Because we never have gaps of length 0, we define d as $o + e$, the cost of gap of length 1.

Consider a database sequence x and a query sequence y; let m be the length x and n be the length of y. We define two variables i and j and let x_i and y_j denote the i^{th} and j^{th} residues of the x and y sequences respectively. Any alignment which ends at position (i, j) must end with a pairing of x_i and y_j, a gap in x representing an insertion in x, or a gap in y representing a deletion in x. The alignment which ends in a gap in x must either extend an alignment which ended with a gap in x at x_{i-1} or add a gap to an alignment which ended at x_{i-1}. Similarly, the alignment which ends in a gap in y must either extend an alignment which ended with a gap in y at y_{j-1} or add a gap to an alignment which ended at y_{j-1}. The highest scoring alignment ending at (i, j) must select the highest scoring of each of its options or start a new local alignment with zero edits. The optimal alignment must end at some position (i, j), so by computing all possible (i, j) we must find the optimal alignment.

We define the functions $M(i, j)$, which is the score of the highest scoring alignment which ends at (i, j); $I(i, j)$ which is the score of the highest scoring alignment which ends in an insertion at (i, j); and $D(i, j)$, which is the score of the highest scoring alignment which ends in a deletion at (i, j). We define a substitution matrix s such that $s(x_i, y_j)$ gives the score of matching x_i and y_j. Local alignments may start at any of the positions $(i, 0)$ for $i = 0 \dots m$

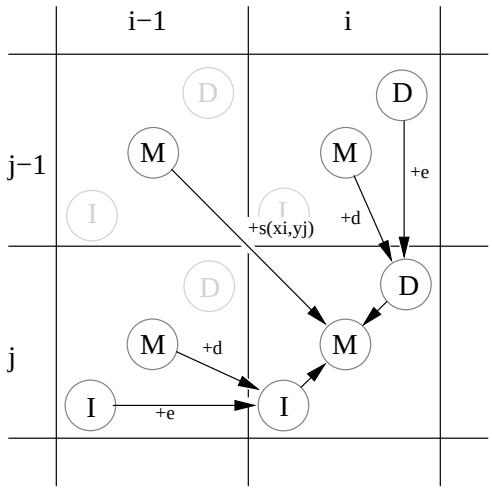

Figure 1: Smith Waterman recurrence depicted graphically.

or $(0, j)$ for $j = 0 \ldots n$, so we set the initial condition that $M(0, j) = 0$ for $j = 0 \ldots m$ and $M(i, 0) = 0$ for $i = 0 \ldots n$. No local alignment may begin with a gap, so we set the initial conditions that $I(0, j) = -\infty$ for $j = 0 \ldots m$ and $D(i, 0) = -\infty$ for $i = 0 \ldots n$. The SW recurrence is expressed as:

$$I(i,j) = max \begin{cases} M(i-1,j) + d \\ I(i-1,j) + e \end{cases}$$
$$D(i,j) = max \begin{cases} M(i,j-1) + d \\ D(i,j-1) + e \end{cases}$$
$$M(i,j) = max \begin{cases} M(i-1,j-1) + s(x_i, y_j) \\ I(i,j) \\ D(i,j) \\ 0 \end{cases}$$

Graphically as in Figure (1).

The combination of the three values, $M(i, j)$, $I(i, j)$ and $D(i, j)$, is known as cell (i, j). There are several important things to recognize about this recurrence. First, each cell is dependent solely on the cell to its left, above, and upper-left, a fact we will exploit later. Second, $M(i, j)$ is never negative. This fact allows us to find strong local alignments regardless of the strength of the global alignment because a local alignment is never penalized by a negative scoring section before it. Lastly, this algorithm runs in $O(mn)$ time and space.

In most biology applications, the majority of alignments are not statistically significant and are discarded. Since allocating and initializing mn space takes significant time, it is common to run linear-space Smith Waterman[15] as a prefilter to full SW. Linear SW is an adaptation of SW which allows the computation to be done in linear space but gives only the score and not the actual alignment. Alignments with high enough scores are then recomputed with SW to get the path of the alignment.

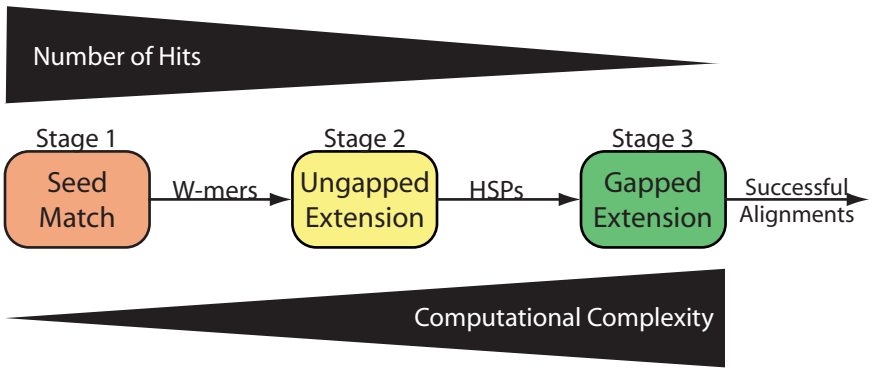

Figure 2: Depiction of BLAST software pipeline.

2.3 BLAST

BLAST is designed around the idea that even distantly similar proteins are likely to have very similar regions. To take advantage of this idea, BLAST breaks the search process down into 3 stages as in Figure (2). Stage 1 searches for exactly matching words of length W, called *W-mers*. W-mers are then passed to stage 2, which performs an ungapped alignment around each W-mer. If the ungapped alignment scores high enough, that is there are many matches and few mismatches, then that is called a High Scoring Pair (*HSP*). HSPs are passed to Stage 3 which performs a gapped alignment around the HSP. Successful gapped alignments are passed to post-processing and output to the user. An important aspect of this pipeline is that each stage filters the number of results that are passed onto the next stage and that each stage is more computationally expensive.

The process of performing an alignment in a targeted area is called seeded alignment. BLAST uses the XDrop algorithm which is based on Smith-Waterman to perform the seeded gapped alignment. XDrop actually performs two alignments, an alignment to the left of the seed and an alignment to the right of the seed, so the combined alignment must go through the seed. The XDrop algorithm runs in $O(mn)$ time and requires $O(min(m, n))$ space.

2.4 Banded Smith Waterman

Banded Smith Waterman (**BSW**) is a special variant of SW used in Mercury BLASTP. As the name suggests, BSW fills a band surrounding an HSP rather than doing a complete fill of the search space. Unlike XDrop, the band has a fixed width and a maximum length. A *diagonal* is a set of positions (i, j) for which $i - j$ is a constant and conversely, an *anti-diagonal* is a set of positions for which $i + j$ is constant. An ungapped alignment is confined to a single diagonal. No alignment, gapped or ungapped, can include an anti-diagonal more than once. We define the band width, ω, as the number of cells in each anti-diagonal and the band length, λ, as the number of anti-diagonals in the band. See Figure (3). We define the term *band geometry* as the specification of a band width and band length and express it in the form λ-ω. The total number of cell fills required is ω*λ. By computing just a band centered around an HSP, we can reduce the search space significantly over using SW. It is

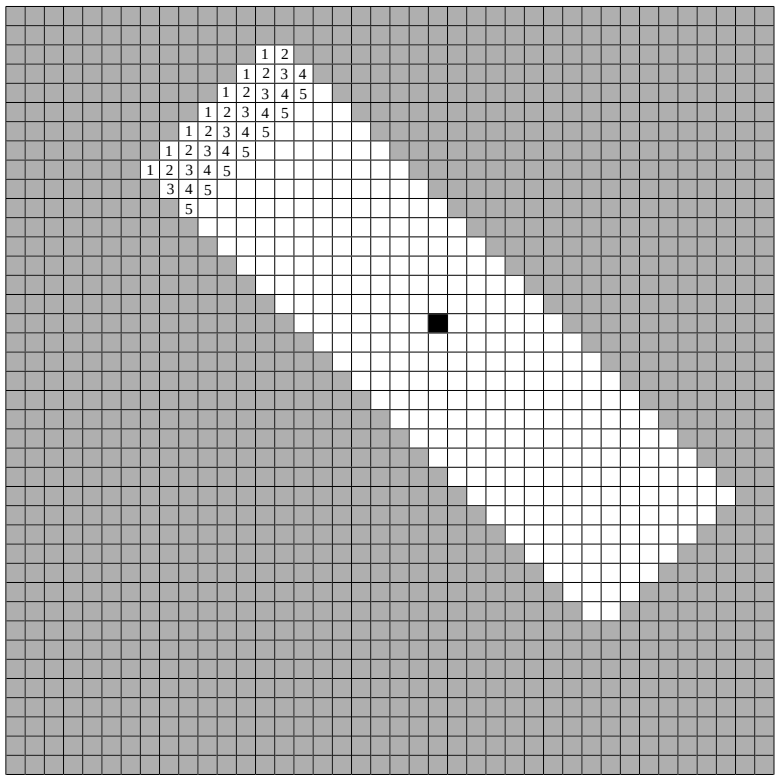

Figure 3: Banded Smith Waterman showing the first 5 anti-diagonals to be computed. $\omega = 7$, $\lambda = 48$

important to note that the maximum number of residues examined in both the database and query is $\omega + \frac{\lambda}{2}$.

The goal of each alignment is to determine if that HSP is part of a statistically significant alignment. Unlike XDrop, BSW does not force the alignment to go through the HSP, but we do impose the constraint that it cross the anti-diagonal the HSP is on. This prevents alignments from ending before the HSP or starting after the HSP. To enforce this constraint we specify that only scores which come after the HSP can indicate a successful alignment. After the HSP we allow scores to become negative, which means that every alignment after the HSP must include the HSP.

As with SW, each cell is dependent only on its left, upper and upper-left neighbors. Thus it makes sense to compute along the anti-diagonal. The order of this computation can be a bit deceiving because the order of anti-diagonal computation does not proceed in a diagonal fashion but rather a stair stepping fashion. That is, after the first anti-diagonal is computed, the second anti-diagonal is immediately to the right of the first and the third is immediately below the second. This can be seen in Figure (3). We make a distinction between odd anti-diagonals and even anti-diagonals where the $1^{st}, 3^{rd}, 5^{th} \ldots etc$ anti-diagonals computed are odd and the $2^{nd}, 4^{th}, 6^{th} \ldots etc$ are even. We will always start numbering the anti-diagonals at one and thus always start with an odd anti-diagonal.

Mercury BLASTP uses a linear-space version of BSW which stores only the previous two anti-diagonal computed. Linear-space BSW computes the score of the optimal alignment

but it does not yield the path of the optimal alignment.

3 Design Description

Mercury BLASTP is a three stage hardware pipeline, similar to NCBI BLASTP, and software to eliminate false positives and find the actual alignment. Each hardware stage of Mercury BLASTP performs the same function as the three stages of NCBI BLASTP: seed matching, ungapped extension and gapped extension, but each uses a slightly different algorithm. The seed matching stage and ungapped extension stage have been implemented and are described in detail in [12] [13].

There have been several other approaches used to accelerate the gapped alignment computation. NCBI BLAST uses prefiltering heuristics to limit the number of gapped alignments which must be performed. Several projects have used parallel processors to compute multiple cells concurrently[14][9][8]. Decypher produces a dedicated Smith-Waterman engine on an FPGA[1]. However, the Mercury BLASTP gapped alignment is unique in that it combines the heuristic seeding approach used by NCBI BLASTP with dedicated Smith-Waterman hardware, which unlike other Smith-Waterman hardware operates only in a band around the seed.

3.1 Mercury BLASTP

Because of the limited memory resources on the board and the availability of very high speed I/O, Mercury BLASTP is designed using a streaming paradigm. A query is loaded into the on-chip memory, and the database is streamed through the card. Each run produces all the significant alignments for that query.

Rather than stream a list of discrete database sequences, the sequences are concatenated along with a sequence separator character. In the hardware, a separator character automatically scores negative infinity, and so no alignments will cross sequence boundaries.

We have taken a similar approach with the query sequence. Rather than stream the database for every query sequence, we pack several query sequences into each run, again separating them with a separator character. Stage 1 uses a hash lookup table which must be stored in off-chip SRAM. The size of the table is positively correlated to the size of the query but must all fit into the available SRAM. Because of the size of the available SRAM the query is limited to a maximum length of 2048 residues. Most proteins are 300-400 residues in length, so each concatenated query typically contains 6-7 distinct query sequences.

To represent the protein alphabet we number the 20 amino acids, listed in alphabetical order, 0-19. In most scoring systems, degenerate symbols all score the same, so we represent all degenerate symbols with the value of 20. We also need the special sequence separator character, which we gave the value of 21. Thus we can represent our entire alphabet with 5 bits.

The Mercury system uses a very simple interface between each component: a 64-bit data bus, a data valid signal, a command valid signal, and a wait signal. Both data and commands are sent over the 64-bit data bus. Commands have a 16-bit control field, 16-bit command type, and a 32-bit parameter. HSPs are an offset position into the query and database. Since

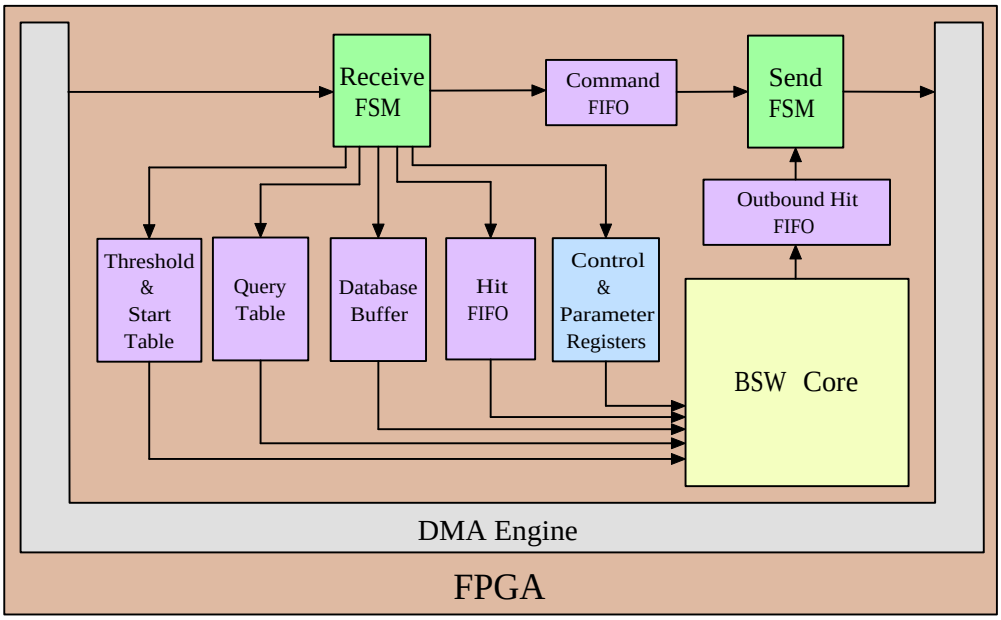

Figure 4: Overview of the Mercury BLASTP Stage 3 design. Arrows represent the flow of data and each box represents a logical component.

the query requires an 11-bit offset and the database requires a 32-bit offset the total size of an HSP is 43 bits. This makes HSPs unsuitable as commands and so they are multiplexed into the normal data-stream along with the database. The database stream is packed 12 residues per 64 bits. We use the highest-order bit, which is otherwise unused, to differentiate between the database stream and HSPs.

3.2 Implementation

The design of Stage 3 can be thought of in three categories: control, buffering and storage, and BSW computation. An overview of the design is shown in Figure (4). Control handles all commands to and from software, directing data to the appropriate buffer and sending successful alignments to software. Both the database and the HSPs must be buffered and the query and its supported data structures must be stored. The BSW computation is performed by an array of elements which execute the recurrence. The design has been implemented targeting a Xilinx Virtex-II part; all references to specific hardware refer to the Virtex-II implementation. The design has been deployed and tested on both the Xilinx XC2V6000 and XC2V4000 parts; for specification for these parts see [5].

3.2.1 Control

Control is implemented using three state-machines and several FIFOs. Each state-machine is responsible for one task. The **receive** state-machine accepts incoming commands and data, processes the commands, and directs the data to the appropriate buffer. For a list of supported commands see Appendix A. All commands and data to leave the system are queued into a FIFO, and the **send** state-machine pulls them out and sends them to the

software. The **compute** state-machine is responsible for controlling the BSW computation. It serves the important functions of calculating the band geometry, initializing the computational core, stopping an alignment when it passes or fails, passing successful alignments to the send state-machine, and resetting the computation core to perform the next alignment. Alignments may be stopped at any time by deasserting a valid signal which runs to all computational modules.

3.2.2 Storage and Buffering

There are several parameters and tables which need to be stored in addition to the query and database. The requisite parameters are the length of the band, λ, the extend penalty, e, and the open-extend penalty, d. Each parameter is set using a separate command from software and are stored in registers in the hardware.

The hardware relies on two tables, the **threshold** table and the **start** table, examples of which can be seen in Table (1). The thresholds to determine what is a significant alignment are based on the length of the query sequence, but because we concatenate multiple queries into a single run, an HSP can be from any one of the query sequences. In order to determine the correct threshold for an HSP, we provide a look up table of the threshold at any position in the concatenated query sequence. This means the hardware does not have to know which query sequence an HSP comes from to determine the threshold for the alignment. This uses only 1 block RAM and no logic while providing a simple and fast decoding mechanism which places no restrictions on the number of queries which can be concatenated. We use a similar technique to improve running time by decreasing the average band length. The start of the band frequently falls before the start of the query sequence. Rather than calculate values that will be reset once we reach the sequence separator, the hardware performs a look up into the start table to determine the minimum starting position for a band. Again the hardware is unaware of which query sequence an HSP comes from but rather performs the lookup based on the HSP's query position. A table of the clock cycles saved by using the start table is provided in Section (4.1).

The query is, at most, 2048 residues in length, or 1.25 Kbytes, so it easily fits in a single Virtex-II block RAM. Since residues are consumed sequentially starting from an initial offset the query buffer provides a FIFO-like interface. The initial address is loaded and then the compute state machine can request the next residue which increments a counter in the query buffer.

The database is too large to be stored on-chip, so the only active portion is stored in a circular buffer. The circular buffer has several unique properties. Because of the design of Mercury BLASTP Stage 1, HSPs do not always arrive in ascending database position. To accommodate these out of order arrivals the buffer keeps a window of 2048 residues behind the current database offset of the current HSP. The typical out of orderness is around 40 residues, so almost all out of orderness is supported. In the exceptional case that an HSP is too far behind, the buffer flags an error, and it is sent to software for further processing. Another special feature of the buffer is that, because once the computation is started it is difficult to stall, the buffer does not service a request until it has buffered the next $\frac{\lambda}{2} + \omega$ residues. This is implemented using a FIFO-like interface similar to the query buffer. The primary difference is that after loading the initial address, the compute state machine must

Position	Query	Threshold	Start
0	S	10	0
1	W	10	0
2	M	10	0
3	A	10	0
4	T	10	0
5	*	-	-
6	G	8	6
7	H	8	6
8	L	8	6
9	D	8	6
10	*	-	-
11	M	8	11
12	S	8	11
13	H	8	11
14	L	8	11
15	*	-	-

Table 1: Sample query, threshold and start tables. In this example, the threshold is set to twice the length of the sequence. In practice, proteins are much longer and the threshold function is more complex.

wait for the database buffer to signal it is ready. This only happens once the buffer has buffered the next $\frac{\lambda}{2} + \omega$ residues.

3.2.3 Banded Smith-Waterman Computation

The BSW computation is carried out by the Stage 3 Smith-Waterman core. We exploit the parallelism of the BSW algorithm that each value in an anti-diagonal can be computed concurrently. To compute ω values simultaneously, the hardware has ω Smith-Waterman computational units, or *cells*. Because there are ω cells and the computation requires one clock cycle, each anti-diagonal can be computed in a single clock cycle. A cell computing $M(i, j)$, is dependent on $M(i-1, j)$, $M(i, j-1)$, $M(i-1, j-1)$, $I(i-1, j)$, $D(i, j-1)$, x_i, y_j, $s(x_i, y_j)$, e, and d. Many of these resources can be shared between cells, for example $M(i+1, j-1)$ which is computed concurrently with $M(i, j)$ is also dependent on $M(i+1-1, j-1) = M(i, j-1)$. Rather than arrange the design in a per-cell fashion, we arranged the BSW in blocks which provide all the dependencies of one type for all cells. The primary advantage of this is that the internal implementation of each block can change as long as it provides the same interface. The design is broken into the MID register block, the pass-fail block, the subject-query shift register, the score block, and the cell block. The relationship between each of these parts can be seen in Figure (5).

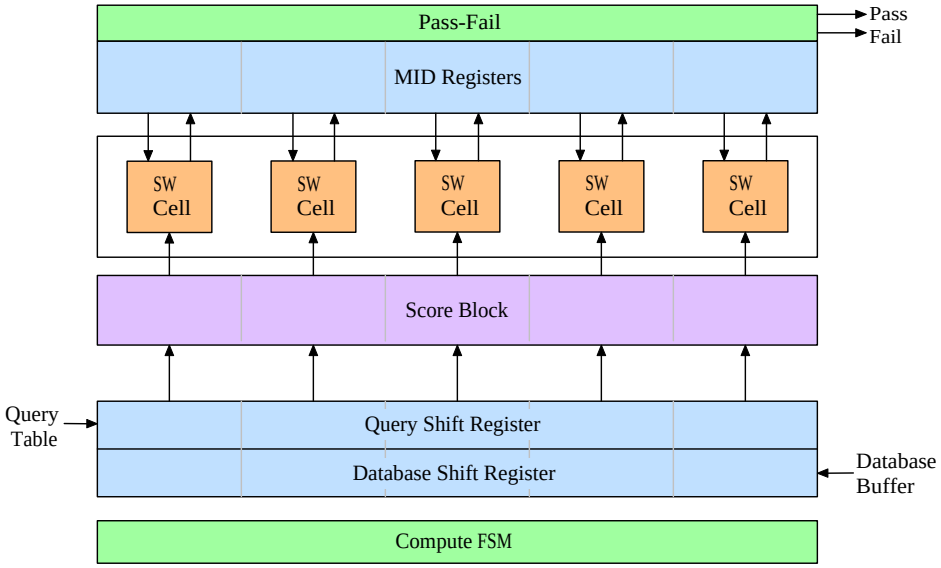

Figure 5: Design of the Banded Smith-Waterman core with $\omega = 5$.

3.2.4 MID Register Block

All of the values computed by each cell are stored in the MID block. Each cell is dependent on three M values, one I value, and one D value, but the three M values are not unique to each cell. Cells adjacent on the anti-diagonal use both the M value above the lower cell and to the left of the upper cell. Thus while there are five inputs, there are only four values per cell. This means that we allocated $4 * \omega$ value registers to store M, I, and D values computed in the previous clock cycle and the M value computed two clock cycles prior. We introduce the terminology M_I which is the M value which is used to compute that cell's I value, that is $M(i-1, j)$ for a cell $M(i, j)$, and similarly, M_D which is used to compute the D value. We define M_2, the M value computed two clock cycles before, that is $M(i-1, j-1)$ for a cell $M(i, j)$. On an odd anti-diagonal, each cell is dependent on the M_D and D it computed in the previous clock cycle, the M_I and I that its left neighbor computed in the previous clock cycle, and M_2. On an even diagonal, each cell is dependent on the M_I and I it computed in the previous clock cycle, the M_D and D that its right neighbor computed in the previous clock cycle, and M_2. To implement this, the MID block uses registers and two input multiplexers. See Figure (6). The control hardware generates a signal to indicate whether the current anti-diagonal is even, which is used as the select on the muxes. To prevent alignments from starting on the edge of the band, scores from outside the band are set to negative infinity.

3.2.5 Pass-Fail Block

The pass-fail block is responsible for determining if an alignment passes or fails. It has ω comparators which compare the threshold to the current scores stored in the M registers of MID block. If any of the M values meets or exceeds the threshold, then the pass-fail module raises a flag, the alignment is stopped and the controller will pass the HSP to software as

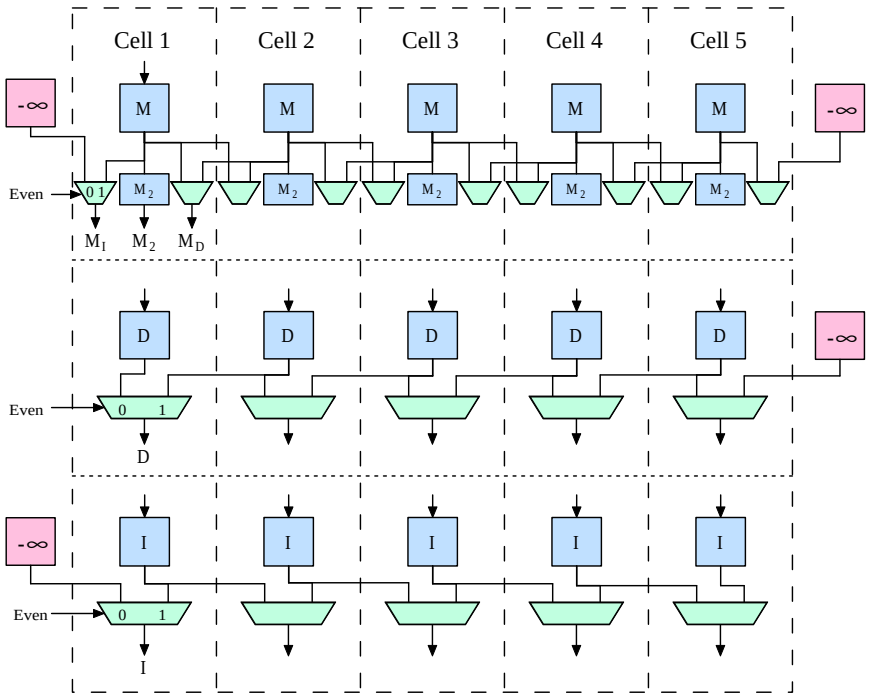

Figure 6: Design of the MID Register Block. Registers in blue, muxes in green, and constants in red.

a valid hit. Most alignments never exceed the threshold, so there is very little increase in performance from stopping passing-alignments early. Once an alignment passes the HSP, the scores can start to become negative. Any alignment which includes a negative scoring portion is the optimal alignment which passes through the HSP but is not an optimal local alignment. Any alignment which has all negative scores for two consecutive anti-diagonals will have no more optimal local alignments which include the HSP. We consider such an HSP to be insignificant because it is not contained in any optimal alignment which exceeds the threshold. The pass-fail block detects if two consecutive anti-diagonals have all negative scores and raises a flag, signaling the controller to stop the alignment.

3.2.6 Database-Query Shift Register

The complete query is stored in block RAM and the database is stored in its own buffer also implemented with block RAMs. The key design challenge was that each ω cells needed access to a different residue of both the query and the database. The solution to the problem is recognizing the locality of the dependencies. Assume that cell 1, the bottom-left most cell, is computing $M(i, j)$ and is therefore dependent on $s(x_i, y_j)$. By the geometry of the cells we know that cell ω is computing $M(i + (\omega - 1), j - (\omega - 1))$ and is therefore dependent on $s(x_{i+(\omega-1)}, y_{j-(\omega-1)})$. So at any point in time the cells must have access to ω consecutive values of both the database and query. The clock cycle following the example above, the system will be dependent on $x_{i+1} \ldots x_{i+1+(\omega-1)}$ and $y_j \ldots y_{j-(\omega-1)}$. So the system needs one new residue and can discard one old residue per clock cycle but must keep the other $\omega - 1$ residues.

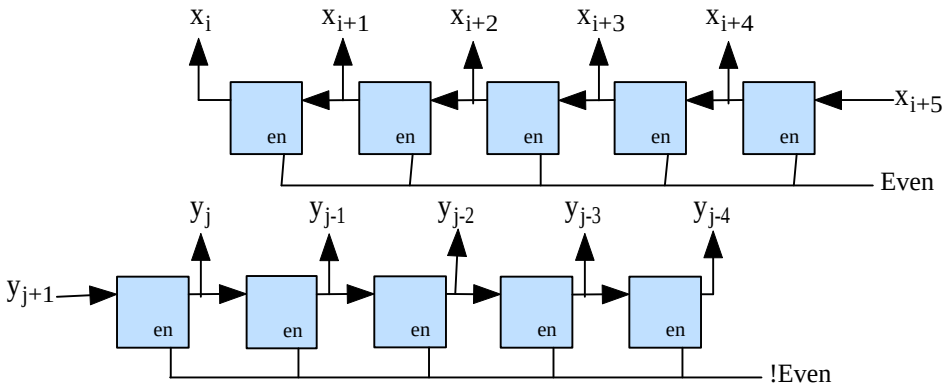

Figure 7: Design of the Database-Query Shift Register.

We implemented this with a pair of parallel tap shift registers, one for the query and one for the database. Each of these is a series of registers who's outputs are connected to the input of an adjacent register and output to the scoring block. See Figure (7). The registers have an enable signal which allows shifting only when desired. During normal running, the database is shifted on even anti-diagonals, and the query is shifted on odd anti-diagonals. The shift actually occurs at the end of the cycle, but the score block introduces a one clock cycle delay so this has the effect of shifting the scores at the end of a odd anti-diagonal for the database and an even anti-diagonal for the query which is the desired result.

3.2.7 Score Block

The score block takes in $x_i \ldots x_{i+(\omega-1)}$ and $y_j \ldots y_{j-(\omega-1)}$ from the database-query shift register and produces the required $s(x_i, y_j) \ldots s(x_{i+(\omega-1)}, y_{j-(\omega-1)})$. This is implemented using a look up table in block RAM. We concatenate the x and y value to generate the address for the look up table. Since each residue is represented with 5 bits, the total address space is 10 bits. Each score value can be represented as a signed 8 bit value so the total size of the table is 1 Kbyte, small enough to fit in one block RAM. Because each residue pair may be different, the design must be able to service all requests simultaneously and independently. Since each block RAM provides two independent ports, we used $\frac{\omega}{2}$ block RAMs to replicate the lookup table. The block RAMs take one cycle to produce data, so the inputs must be sent one clock cycle ahead.

3.2.8 Cell Block

The cell block is the logic which actually computes the SW recurrence. It is comprised solely of combinational logic because all the values are stored in the MID register block. See Figure (8). Each cell consists of four adders, five maximizers, and a two-input mux. Internally, each maximizer is implemented as a comparator and a mux. The two-input mux is used to select the minimum value, either zero for all the anti-diagonals before the HSP or negative infinity after the HSP.

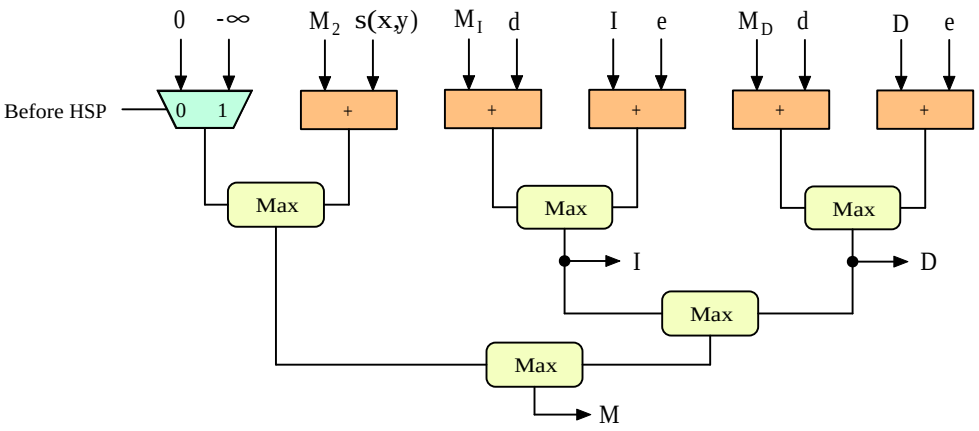

Figure 8: Design of a single Smith-Waterman Cell.

4 Results

The design has been implemented in VHDL and Verilog, and deployed on a Xilinx Virtex-II 4000 and 6000. The targeted XC2V4000 is "-4" speed grade, and the design has been placed, routed, and tested at 66 MHz. The targeted XC2V6000 is a "-6" speed grade, and the design has been placed, routed, and tested at 75 MHz. For testing and collection of data a XC2V4000 was used with the design running at 60 MHz. Performance data is quoted for the 66 MHz implementation. Hardware utilization data for varying ω can be found in Appendix B.

Our target platform is a dual Opteron workstation with 6 Gbytes of memory and two channels of 100 MHz PCI-X. We use two Virtex-II development boards each connected to its own PCI-X channel. Stages 1 and 2 are deployed on the first card, and stage 3 on the second. The database is sent to the first card and is processed by stage 1 and 2 and the resulting stream with the database and HSPs interleaved is sent to stage 3 through main memory. Currently each stage is being tested and debugged independently. The software infrastructure to convert an NCBI BLASTP search to a Mercury BLASTP search is still under development. We used software to synthesize the input to stage 3 because Stages 1 and 2 are still undergoing testing and debugging. The data was collected by running NCBI BLAST and writing the HSPs generated to file. Based on the design of stages 1 and 2 we expect the results to be very similar to those produced by BLAST. The hits were then interleaved with the database and sent to stage 3 for processing. Because the process of collecting data from software, reformatting it for the hardware and running it through the card is very time consuming, the results presented here were derived using software simulation, but samples have been verified against the hardware results.

We have not developed a model to describe the out of orderness introduced by stage 1, so when interleaving the hits and database, hits are placed immediately following the corresponding database position, and no out of orderness is introduced. Similarly, no model has been developed to classify how the inter-arrival time of HSPs affects performance, so we perform a mean value analysis. We first evaluate the performance of stage 3 as a stand alone design and then we evaluate the system performance. We then discuss the sensitivity and

		λ				
		35	45	55	65	75
ω	1001	147.0	141.0	135.0	129.2	123.5
	1051	163.0	156.6	150.5	144.4	138.4
	1101	178.8	172.2	165.8	159.4	153.2
	1151	195.8	189.1	182.4	175.9	169.4
	1201	212.7	205.8	198.9	192.1	185.4
	1251	230.6	223.5	216.5	209.5	202.7
	1301	248.3	241.1	233.9	226.8	219.7
	1351	266.9	259.5	252.2	245.0	237.7
	1401	285.3	277.8	270.4	263.0	255.6
	1451	304.6	297.0	289.4	281.8	274.3
	1501	323.8	316.0	308.2	300.6	292.9
	1551	343.7	335.8	327.9	320.1	312.3
	1601	363.6	355.5	347.5	339.5	331.6

Table 2: Mean number of clock cycles saved by using **start** table for varying band geometries.

specificity trade-offs of different band geometries.

4.1 Performance

The performance of stage 3 is dependent on the band geometry. Assuming the relevant portion of the database is already in the database buffer, the worst case running time per HSP is $5 + \omega + \lambda$. The hardware requires 5 clock cycles to compute the band geometry and initialize the database buffer; it then takes ω clock cycles to load the shift register with the initial values and finally λ clock cycles to compute the score of the optimal alignment. The mean clock cycles per HSP is much lower because for two reasons. The first is that using the **start** table prevents the band from extending beyond the start of an individual query sequence. We define κ as the position of the HSP relative to the individual query sequence. The number of anti-diagonals computed before the HSP is then $min(2\kappa + \omega, \frac{\lambda}{2})$. It is 2κ because there are twice as many anti-diagonals as residues. The center of the band intersects the start of the query at 2κ, but the triangular tail adds an additional ω anti-diagonals. Table (2) shows the mean number of clock cycles saved per HSP for several different band geometries. The second reason the mean cycles per HSP is less than the worst case is that alignments tend to pass or fail before reaching the end of the band. A failure can be the result of having no positive scoring alignments or reaching the end of either the query sequence or the subject sequence. Table (3) lists the mean number of clock cycles saved by early termination of the alignment.

Mercury BLASTP is a streaming pipeline, so latency is insignificant in the overall system performance. The primary metric for determining system performance in a pipeline is throughput. We calculate the throughput of stage 3 by dividing the clock frequency, 66 MHz, by the number of clock cycles required to process each HSP. This yields a throughput measured in thousands of HSPs per second (**kHSP/s**) and these values can be found in Table (4).

		λ				
		35	45	55	65	75
ω	1001	272.6	293.1	297.6	298.2	297.8
	1051	295.8	317.3	322.1	322.7	322.4
	1101	319.0	341.4	346.5	347.3	346.9
	1151	342.4	365.7	371.1	371.9	371.6
	1201	365.8	390.0	395.6	396.5	396.2
	1251	389.4	414.4	420.2	421.2	420.9
	1301	413.0	438.8	444.9	445.9	445.7
	1351	436.8	463.3	469.5	470.7	470.4
	1401	460.6	487.7	494.2	495.4	495.2
	1451	484.5	512.3	519.0	520.2	520.1
	1501	508.4	536.8	543.7	545.0	544.9
	1551	532.4	561.4	568.5	569.9	569.7
	1601	556.4	586.0	593.3	594.7	594.6

Table 3: Mean number of clock cycles saved by terminating the alignment early because it has passed, failed or reached the end of one of the sequences.

		λ				
		35	45	55	65	75
ω	1001	107.3	109.8	109.6	108.7	107.6
	1051	105.4	108.0	107.8	106.9	105.8
	1101	103.6	106.3	106.0	105.1	104.0
	1151	102.1	104.8	104.6	103.6	102.6
	1201	100.6	103.3	103.1	102.2	101.1
	1251	99.4	102.1	101.9	101.0	99.9
	1301	98.1	100.8	100.7	99.8	98.7
	1351	97.0	99.8	99.6	98.7	97.6
	1401	95.9	98.7	98.6	97.7	96.6
	1451	95.0	97.8	97.7	96.8	95.7
	1501	94.0	96.9	96.8	95.9	94.8
	1551	93.3	96.1	96.0	95.1	94.0
	1601	92.5	95.3	95.2	94.3	93.3

Table 4: Maximum throughput of stage 3 measured in thousands of HSPs per second.

		λ				
		35	45	55	65	75
ω	1001	321.1	328.7	327.9	325.1	321.9
	1051	315.5	323.3	322.6	319.8	316.5
	1101	310.2	318.0	317.3	314.5	311.3
	1151	305.6	313.6	312.9	310.1	306.9
	1201	301.1	309.2	308.6	305.8	302.6
	1251	297.3	305.5	304.9	302.1	298.9
	1301	293.5	301.7	301.2	298.5	295.3
	1351	290.3	298.5	298.1	295.4	292.2
	1401	287.0	295.4	294.9	292.2	289.0
	1451	284.2	292.6	292.2	289.6	286.4
	1501	281.4	289.9	289.5	286.9	283.7
	1551	279.1	287.5	287.2	284.6	281.4
	1601	276.7	285.2	284.9	282.3	279.1

Table 5: Maximum throughput of stage 3 measured in millions of residues per second.

We measure the throughput of Mercury BLASTP in millions of residues per second (**mRes/s**). Based on the hit generation rate and filtering rate of stages 1 and 2, the mean inter-arrival time of HSPs is 2992.43 residues[12]. We use this value to convert the performance data in measured in kHSP/s into mRes/s so that the performance of stage 3 can be compared to other stages. We calculate mRes/s by multiplying 2992.43 times the measured kHSP/s. Values for the throughput in mRes/s for varied band geometries are found in Table (5). This simplified analysis assumes that stage 3 never has to wait for database or software down stream, and while this is unrealistic, it useful as as an upper bound.

The target performance of Mercury BLASTP is a 30x speedup over a baseline commodity PC. The baseline system is a 2.8 GHz Pentium 4 with 1 GB of RAM. The time measured to run the Escherichia coli k12 proteome (4,242 sequences; 1,351,322 residues) against the GenBank Non-Redundant (NR) database (2,321,957 sequences; 787,608,532 residues) is 130,460 seconds. In order to meet the 30x target, Mercury BLASTP must run this search in 4348.7 seconds. Mercury BLASTP must perform 664 passes through the database for this search, so the total number of residues to pass through the pipeline is $664 * 787,606,532 = 522,970,737,248$. To perform this search in the alloted time of 4,348.7 seconds the database must stream at an average rate of 120.26 MRes/s. The throughput of the system is limited by stage 1 which has a maximum throughput of 219 MRes/s, equivalent to 54x.

4.2 Quality of Results

There are two important metrics for classifying the quality of the results produced by stage 3, sensitivity and selectivity. These terms as defined as:

$$\text{Sensitivity} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

$$\text{Selectivity} = \frac{\text{true negatives}}{\text{true negatives} + \text{false positives}}$$

We measure both sensitivity and selectivity using the XDrop implementation in NCBI BLASTP as the standard for measuring true positives and true negatives. Table (6) shows the sensitivity and selectivity of varying band geometries.

To increase sensitivity, we allow the threshold to be artificially lowered. We define a scalar ϕ which will be multiplied by all thresholds during the generation of the threshold table. This increases sensitivity but decreases selectivity, and more HSPs are passed to software. We define the match rate p as the probability that an HSP will be passed to software. The software is capable of processing 6,410 HSPs per second, and HSPs are processed by the hardware pipeline at a mean rate of 73,184 HSPs per second. Thus in order for software to keep up with the hardware, $p < \frac{6{,}410}{73{,}184} = .0875$. In other words, stage 3 must filter out at least 91.25% of HSPs. Table (7) shows the impact ϕ has on p and sensitivity and selectivity.

The system we are deploying this design on places stage 3 on its own XC2V4000 so there is sufficient hardware resources to set $\omega = 75$. We choose $\lambda = 1601$ because the performance of the system is limited by stage 1 and we set $\phi = .6$ which offers a sensitivity nearly indistinguishable from 1 and f$p = .995$. These selections are logical for the current deployment of the system but my change in the future. Future deployments may place stage 3 on the same FPGA as the other stages and fewer hardware resources may be available or performance enhancements may be made to other stages of the pipeline requiring more performance.

5 Conclusion

Protein databases are growing at an exponential rate which has made the task of searching them increasingly time consuming. The increasing importance of proteomics is driving a need for accelerated protein similarity searches. These searches can be accelerated using customized hardware such as the FPGA-based Mercury BLASTP. This paper describes the design of the gapped extension stage, one of the three pipeline stages in Mercury BLASTP.

We first verified the quality of results and the exploitable parallelism of the known banded Smith Waterman algorithm. We conceived an original design to execute the banded Smith-Waterman algorithm and developed a simulator to verify its performance. We implemented the design in reconfigurable logic using VHDL and Verilog and deployed it on a Xilinx FPGA. We showed that the design is capable of running searches over 54x faster than a standard PC.

5.1 Future Work

The next step is to integrate the three stages of Mercury BLASTP and build the supporting software infrastructure. Once combined the end-to-end system performance and quality of results can be measured. Additional features to be added to stage 3 include reducing the block RAM using by double clocking them to provide 4 accesses a clock. This will reduce the overall size of the design and increase the likelihood of a single-chip solution for all three

Sensitivity

		λ				
		35	45	55	65	75
ω	1001	0.99474	0.99605	0.99631	0.99635	0.99636
	1051	0.99570	0.99705	0.99734	0.99737	0.99739
	1101	0.99635	0.99785	0.99814	0.99818	0.99819
	1151	0.99691	0.99843	0.99873	0.99879	0.99880
	1201	0.99728	0.99881	0.99911	0.99917	0.99918
	1251	0.99754	0.99907	0.99937	0.99943	0.99944
	1301	0.99770	0.99926	0.99955	0.99960	0.99962
	1351	0.99778	0.99935	0.99964	0.99970	0.99971
	1401	0.99787	0.99943	0.99973	0.99978	0.99981
	1451	0.99793	0.99947	0.99979	0.99984	0.99988
	1501	0.99802	0.99949	0.99982	0.99988	0.99990
	1551	0.99804	0.99951	0.99983	0.99989	0.99992
	1601	0.99804	0.99951	0.99984	0.99990	0.99992

Selectivity

		λ				
		35	45	55	65	75
ω	1001	0.99924	0.99913	0.99904	0.99897	0.99892
	1051	0.99923	0.99911	0.99902	0.99896	0.99891
	1101	0.99921	0.99910	0.999	0.99894	0.99889
	1151	0.99919	0.99907	0.99897	0.99891	0.99886
	1201	0.99917	0.99905	0.99896	0.99889	0.99884
	1251	0.99917	0.99904	0.99895	0.99888	0.99883
	1301	0.99916	0.99904	0.99894	0.99887	0.99882
	1351	0.99915	0.99903	0.99893	0.99886	0.99881
	1401	0.99914	0.99902	0.99892	0.99886	0.99881
	1451	0.99914	0.99902	0.99892	0.99885	0.99880
	1501	0.99914	0.99901	0.99891	0.99885	0.99879
	1551	0.99913	0.99900	0.99891	0.99884	0.99879
	1601	0.99913	0.99900	0.99890	0.99883	0.99878

Table 6: Sensitivity and selectivity measured against the XDrop algorithm of different band geometries. These values were collected from the software simulation of stage 3. (Note: These values are for a small sample size and are pending update subject to further analysis.)

		ϕ					
		.50	.60	.70	.80	.90	1.0
ω	35	0.99982	0.99966	0.99939	0.99889	0.99842	0.99770
	45	0.99999	0.99995	0.99985	0.99971	0.99954	0.99926
	55	0.99999	0.99999	0.99997	0.99989	0.99979	0.99955
	65	1.00000	0.99999	0.99999	0.99992	0.99983	0.99960
	75	1.00000	0.99999	0.99999	0.99993	0.99984	0.99962

Table 7: Effect on sensitivity of varying ϕ given $\lambda = 1301$.

stages. An adaptation to the banded Smith-Waterman algorithm which forces the alignment to go through the seed can be investigated to study its affects on quality of results.

Appendix A Module Command Reference

Stage 3 hardware accepts a number of commands. Commands are divided into two categories, global and module specific. Global commands are used by all stages, including the infrastructure for moving and in and out of the hardware. 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 stage 3, 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.

- **Start of Start Table (ss)**

  Marks the begining of the start table stream.

- **End of Start Table (es)**

  Marks the end of the start table stream.

- **Start of Threshold Table (st)**

  Marks the begining of the threshold table stream.

- **End of Start Table (et)**

  Marks the end of the threshold table stream.

- **Set Band Length (ln)**

  Sets the length of the band λ.

- **Set Gap Extend (ex)**

  Sets the gap entend penalty, e.

- **Set Gap Extend + Extend (oe)**

  Sets the gap open + extend penalty, d.

- **All Hits (ah)**

  Similar to Passthrough mode, but each HSO is processed and sent to software with a single bit set to indicate a failed HSP.

Appendix B Hardware Resource Usage for Stage 3

			XC2V4000		XC2V6000	
ω	SLICES	BRAM	% SLICES	% BRAM	% SLICES	% BRAM
35	7,622	54	33.1%	45.0%	22.6%	37.5%
45	8,749	59	38.0%	49.2%	25.9%	41.0%
55	9,612	64	41.7%	53.3%	28.4%	44.4%
65	10,332	69	44.8%	57.5%	30.6%	47.9%
75	11,408	74	49.5%	61.7%	33.8%	51.4%

References

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

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

[3] Paracel Genematcher. `http://www.timelogic.com/`.

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

[5] Xilinx Virtex-II Platform FPGAs: Complete Data Sheet. `http://direct.xilinx.com/bvdocs/publications/ds031.pdf`.

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

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

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

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

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

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

[12] Arpith C. Jacob. Design and Analysis of An Accelerated Seed Generation Stage for BLASTP on the Mercury System. Master's thesis, Washington University in St. Louis, St. Louis, MO. USA. 63130, August 2006.

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

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

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

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