2016

Regular Expression Synthesis for BLAST Two-Hit Filtering

Jordan Bradshaw
University of South Carolina

Regular Expression Synthesis for BLAST Two-Hit Filtering

by

Jordan Bradshaw

Bachelor of Science
University of South Carolina, 2009

Master of Engineering
University of South Carolina, 2011

Submitted in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2016

Accepted by:

Jason D. Bakos, Major Professor

Duncan Buell, Chair, Examining Committee

John R. Rose, Committee Member

Yan Tong, Committee Member

Phil Moore, Committee Member

Lacy Ford, Senior Vice Provost and Dean of Graduate Studies

ACKNOWLEDGEMENTS

First and foremost, I must thank my advisor, Dr. Jason Bakos for working with me to complete this research and my graduate studies. The difficulties surrounding the time that I had to devote to this work made it challenging for both of us, and I appreciate being given the opportunity to work with him despite them. He has always been a source of useful advice and direction throughout the entire process, and there is no doubt I would not have completed this work without him. Likewise, I thank Rasha Karakchi for assistance in implementing some of the design work covered in this dissertation.

I would also like to thank my other committee members, Dr. Duncan Buell, Dr. John Rose, Dr. Yan Tong and Dr. Phil Moore for taking their time to serve on my committee.

My family has also always been supportive, and I will always appreciate them for everything they have and will do for me.

Lastly, I must thank my friends. They have always been an endless source of interesting distractions, without which I may have finished years sooner but would be a much less happy person for it.

ABSTRACT

Genomic databases are exhibiting a growth rate that is outpacing Moore's Law, which has made database search algorithms a popular application for use on emerging processor technologies. NCBI BLAST is the standard tool for performing searches against these databases, which operates by transforming each database query into a filter that is subsequently applied to the database. This requires a database scan for every query, fundamentally limiting its performance by I/O bandwidth. In this dissertation we present a functionally-equivalent variation on the NCBI BLAST algorithm that maps more suitably to an FPGA implementation. This variation of the algorithm attempts to reduce the I/O requirement by leveraging FPGA-specific capabilities, such as high pattern matching throughput and explicit on-chip memory structure and allocation. Our algorithm transforms the database—not the query—into a filter that is stored as a hierarchical arrangement of three tables, the first two of which are stored on-chip and the third off-chip. Our results show that it is possible to achieve speedups of up to 8x based on the relative reduction in I/O of our approach versus that of NCBI BLAST, with a minimal impact on sensitivity. More importantly, the performance relative to NCBI BLAST improves with larger databases and query workload sizes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In the field of computational biology, there is often a need to identify a genetic or protein sequence, or to recognize related sequences. Identifying sequences is performed by comparing samples against databases of known sequences in vast search operations, sometimes against billions of possible matches. Performing this search efficiently is the subject of a large body of prior work, and contributions to this area have the potential for significant impact for computational biology. As of 2015, BLAST, a popular database search algorithm, has been cited in over 50,000 other publications.

Genomic database search relies on approximate string matching, and the process of comparing two non-equal sequences in an attempt to discover how closely they line up is a problem known as sequence alignment. Smith-Waterman is a dynamic programming algorithm that serves as the basis for several sequence alignment algorithms, and is optimal in the sense that it always provides the best alignment of sequences for a given scoring criteria. Dynamic programming techniques have the drawback that they require quadratic time and memory, and for problems of an interesting size they are often infeasible to use in practice.

As a result, algorithms have been derived from it that seek to improve its running time without a significant sacrifice in sensitivity in the alignment. BLAST is an example of a heuristic algorithm that is run before traditional dynamic programming algorithms in an effort to rapidly discover potential matches between sequences. BLAST is much

faster, but has the drawback that its heuristic approach may miss some sequence alignments because it is not exhaustive.

Due to the importance of sequence alignment and its common usage, BLAST has been the subject of substantial research, with many proposed enhancements and implementations on specialized hardware such as FPGAs. While great speedups over CPU implementations have been achieved using such hardware, most state of the art implementations focus on directly translating the hash table-based design used in the reference implementation into FPGA hardware.

BLAST is fundamentally memory bound, and translating the traditional hash table-based algorithm to FPGAs does not address this limitation. BLAST's runtime is bounded by how long it takes to transfer the database being searched from memory or disk, and traditional designs must pay this transfer penalty for every query. Thus, improving BLAST's performance requires some means of reducing the transfer size, either by compressing or indexing the database. FPGAs are well suited to this approach since they are able to efficiently perform decompression or rapid table lookups as an early stage of a computational pipeline.

In this work we develop an alternate implementation of BLAST's two-hit filtering stage using sets of regular expressions built from protein databases that allow us to eliminate the need to scan the entire database for each query sequence, as well as the need for a large DRAM-backed hash table. By leveraging these regular expressions and eliminating the hash table, we can devote the FPGA's resources to pattern matching.

Pattern matching on FPGAs has a long history in the literature, such as network intrusion detection and real time packet inspection using databases of regular expressions

that match malicious traffic. However, to the best of our knowledge, there are no existing designs that utilize this approach for genomic database search.

In this research, we address the gap in the literature regarding the use of finite automata on FPGAs for genomic database search. Specifically, we present a novel implementation of BLAST's two-hit filtering stage utilizing finite automata in the form of synthesized regular expressions. We show that this implementation of BLAST results in a substantial reduction in I/O, which is the bottleneck for state of the art FPGA designs for BLAST. By reducing the I/O bottleneck, we achieve projected speedups of nearly 8x compared to these designs, while maintaining high sensitivity and selectivity in the search operation.

CHAPTER 2

BACKGROUND

2.1 SEQUENCE ALIGNMENT

Sequence alignment is a way of arranging sequences of data to find regions of similarity. It is commonly used in bioinformatics to find functional or ancestral similarities between DNA, RNA or protein sequences, but it also has applications in other fields, such as natural language processing and the social sciences [1].

These problems are usually solved by arranging the sequences in question into a matrix and trying to find the best corresponding regions between them using dynamic programming techniques.

2.2 SMITH-WATERMAN AND DYNAMIC PROGRAMMING

Dynamic programming is a technique used in optimization problems where a problem has optimal substructure and where it repeatedly solves the same subproblems. In this case, having optimal substructure means that an optimal solution to a problem can be found by finding an optimal solution to its subproblems. By reusing the solutions to subproblems, such a problem can be solved much faster than by naively resolving its constituent components.

The first described sequence alignment algorithm to use dynamic programming in this fashion was presented by Needleman et al [2], and is known as the Needleman-Wunsch algorithm. A very similar algorithm, later used as the basis for the much faster

FASTA and BLAST algorithms, was described by Smith et al [3], and is known as the Smith-Waterman algorithm.

Both algorithms construct a table using two strings to compare on its axes and a scoring system to fill in the table cells, which is then used to find the best alignments of the two strings. A brief example of the Needleman-Wunsch algorithm follows.

Suppose we have the following sequences: A C C T G and A C G T C and want to find the best alignment between the two. We begin by arranging the two sequences on the axes of a table, as shown in Figure 2.1.

		A	C	C	T	G
	0					
A						
C						
G						
T						
C						

Figure 2.1: Empty Needleman-Wunsch Table

The table is populated by going from cell-to-cell and comparing the characters in the corresponding row and column. For matches, the score increases by 1, for mismatches it decreases by 1, and for insertions or deletions (called "indels"), it also decreases by 1. This change is added to the highest previous value, which comes from the left, top or upper-left neighboring cells, and represents a previous alignment step. When populating the cell, it is important to keep track of which cell was used as the prior value, as this will be used to trace back the alignment. In the following figures, this will be represented as arrows pointing to the prior values. Note that it's possible to have

multiple equal choices, represented by multiple outgoing arrows. These represent multiple equally valid alignments.

The first row and column are simple: they can only choose from the left or top cell's value, and since they have no character to compare to, are always a mismatch. After populating these cells, the table should appear like in Figure 2.2.

		A	C	C	T	G
	0	-1	-2	-3	-4	-5
A	-1					
C	-2					
G	-3					
T	-4					
C	-5					

Figure 2.2: Partial Needleman-Wunsch Table

Proceeding from here, the next step would be to compare A to A. Since this a match, the score goes up by 1, and the highest prior value is 0. The table would now appear like in Figure 2.3. By continuing the procedure for the remaining cells, the table can be completed, as shown in Figure 2.4.

		A	C	C	T	G
	0	-1	-2	-3	-4	-5
A	-1	1				
C	-2					
G	-3					
T	-4					
C	-5					

Figure 2.3: Partial Needleman-Wunsch Table

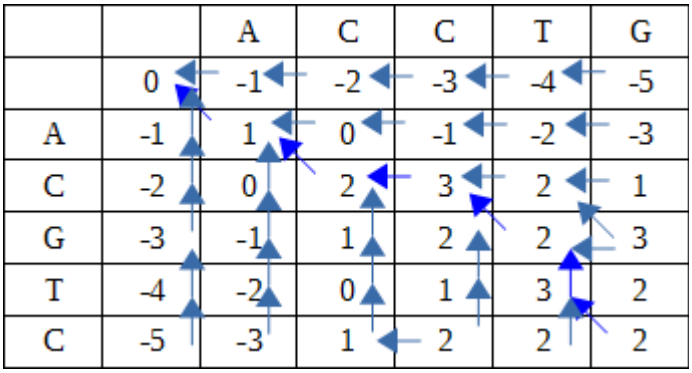


Figure 2.4: Completed Needleman-Wunsch Table

Once the table has been populated, alignments can be found by backtracking through the table, starting with the bottom right cell and working backward to a score of 0. Diagonal arrows represent a match or mismatch, while vertical arrows represent an insertion from the query (on the left) and horizontal arrows represent a deletion (denoted by a dash in the alignment). A possible alignment is shown by the highlighted arrows, and given by: A C - G T C

Different scoring systems can be used for different problems. For example, if insertions or deletions are strongly discouraged, then a severe penalty can be imposed for them. For genetic sequence alignment, it is typical to use a value of 1 for a match and either a value of 0 or -1 for a mismatch.

The Smith-Waterman algorithm, by comparison, is better suited to finding local alignments, which are subregions of sequences with high similarity. This is done by setting the minimum value of any table cell to 0, which allows any number of mismatches to occur between matching elements without drastically reducing the score.

2.3 BLAST AND HEURISTIC APPROACHES

Although Smith-Waterman is optimal in the matches found for a given substitution matrix, its greatest drawback is that, as a dynamic programming algorithm, it

uses quadratic time and space and is infeasible to use directly on large problems. To address this shortcoming, several approaches using heuristic algorithms were devised to improve the alignment speed with a minimal sacrifice in sensitivity.

FASTA was one of the first heuristic algorithms for sequence alignment, proposed by Pearson et al [4][5]. Its speed benefit comes from identifying promising small local alignments first, rather than building an entire scoring table. FASTA works by defining a *ktup* parameter, which is the length of matching elements that must be found for a local alignment to be considered. A *ktup* value of 4, for example, means that a sequence of 4 characters must match exactly between the two sequences before it will be considered further. These local alignments, known as regions of identity, are then scored by the Wilbur-Lipman algorithm [6]. The Wilbur-Lipman algorithm finds local alignments by using the Dot Matrix projection proposed by Maizel et al. [7], and finding diagonal runs of at least *ktup* length.

The Dot Matrix projection takes two sequences and places them at the top and left sides of a matrix, and marks every cell in the matrix where the value of the column and row are identical. Diagonal lines running from the upper left to lower right corner represent possible alignments of the two sequences, and for consecutively marked cells on a diagonal, the sequences are identical. An example of this is shown in Figure 2.5.

FASTA proceeds by rescoring the best 10 diagonals (termed regions in FASTA) by using a scoring matrix, which allows amino acid substitutions that are common in nature to maintain a positive score. This improves the sensitivity of the algorithm, and by not considering gaps or deletions at this stage FASTA retains high speed.

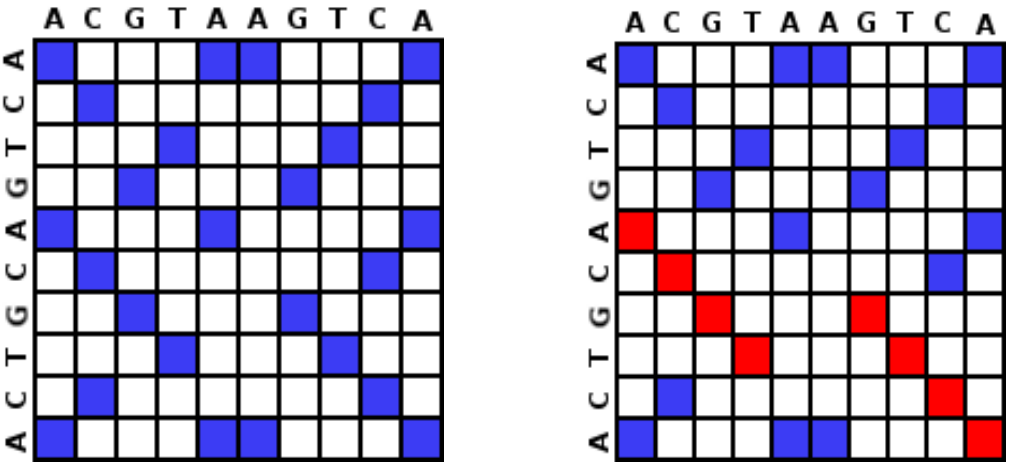


Figure 2.5: Dot Matrix comparison of two sequences. The left shows the raw comparison, and the right shows FASTA regions of *ktup* = 4 highlighted

The next step of FASTA is to try to merge local alignments that are on the same diagonal, using a penalty metric similar to the insertion and deletion metrics used by Smith-Waterman. Any merged alignments are then rescored, and the best scoring results are returned.

Altschul et al. later proposed an algorithm inspired by FASTA, known as BLAST [8]. BLAST also employs heuristics to perform local alignments, but differs from FASTA in that it instead relies on the concept of *seeding* and uses a user defined cutoff *threshold* for the seeding operation.

The scores computed in BLAST are based on statistical analysis performed by Karlin et al. [9], who provided models that give the probabilities of combinations of protein sequences occurring. Commonly occurring amino acids are given comparatively low scores, while rarely occurring amino acids are given higher scores. The score of a sequence of amino acids can be summed up, with a high score effectively indicating a sequence that is less likely to occur by chance and thus more significant. These models

can also be used to describe the probability of mismatches between sequences, which is used to allow significant but different subsequences to be related to one another.

The threshold used in the BLAST search is directly related to the score and is used as a filter so that only subsequences of comparatively high value are considered. A subsequence with a low score is unlikely to be useful in helping to distinguish between two sequences, so a minimum threshold is enforced. Seeds whose sequence score is below the threshold are not considered.

Seeding itself is the primary difference between BLAST and FASTA, and proceeds by decomposing the search query into a set of *W-mers*, which are overlapping sequences of a fixed length. For example, for protein searches in BLAST, *W-mers* of size 3 are typically used. If the following query was being considered,

A B C C D

then the following W-mers would be produced:

A B C B C C C C D

Each W-mer's *self-score* is then computed using a matrix of scores, which is in turn produced using probabilities as described above. As an example, consider the scoring matrix in Figure 2.6.

	A	B	C	D
A	5	-4	-1	-6
B	-4	3	-1	-6
C	-1	-1	1	-6
D	-6	-6	-6	8

Figure 2.6: Sample BLAST Scoring Matrix

Using this scoring matrix, these seeds have self scores of:

A B C	Score: 9 (A to A = +5, B to B = +3, C to C = +1)
B C C	Score: 5 (B to B = +3, C to C = +1, C to C = +1)
C C D	Score: 10 (C to C = +1, C to C = +1, D to D = +8)

If we were using a threshold value of 9, then only the seeds A B C and C C D would be considered. B C C would be too likely to occur at random to be worth considering.

The process of finding hits in a database sequence uses the same scoring matrix and proceeds by "sliding" the seed over the database query, checking the possibly mismatched characters between the seed and database query for their scores, and summing them up. If the summed score is above the threshold, then a hit is reported.

For example, if we consider the second seed we kept from above, C C D, and the database entry A D C C D, then the seeding operation would proceed as follows:

A D C C D	
C C D	Score: -13 (A to C = -1, D to C = -6, C to D = -6)
C C D	Score: -11 (D to C = -6, C to C = +1, C to D = -6)
C C D	Score: 10 (C to C = +1, C to B = +1, D to D = +8)

If a threshold value of 9 were being employed, then this would produce a single hit for the seed C C D at position 2.

After all of the hits for the query sequence above a threshold value have been found and recorded, BLAST proceeds by performing gap extension in a similar fashion to FASTA. Seed hits are arranged along a diagonal determined by the offsets into the query and database string, and seed hits along the same diagonal are subject to merging.

Extensions are performed by adding a single character at a time from the query and database and using the scoring matrix to evaluate the new score. If the score drops by a threshold, then extension terminates, but if it continues then several seeds may be consolidated into a single hit. Extensions are performed such that seeds are merged first, if possible, before extending outward. A visualization of this, from Gapped BLAST [10] is shown in Figure 2.7.

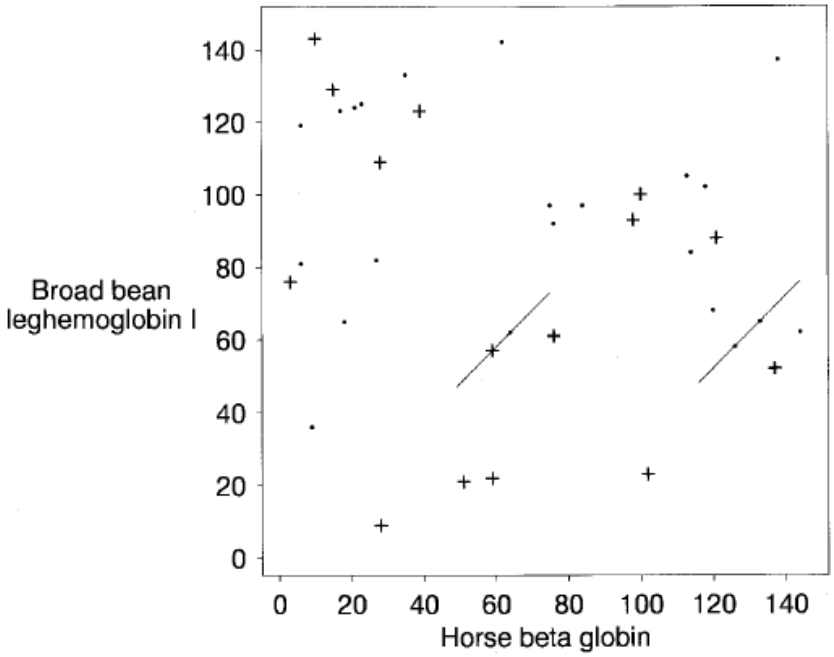


Figure 2.7: Example of seed hits from Gapped BLAST (Becchi). Hits indicated by a plus sign have a score of at least 15. Dots are a score of 11-14. The lines indicate hits along the same diagonal that were close enough to trigger potential extensions (a distance of 40 in this case).

After all of the seeds have been evaluated for extension, the highest scoring results are returned.

BLAST's importance and ubiquity in computational biology has led to much effort in improving its performance. A common way of improving the speed that BLAST operates at is to use specialized hardware platforms, such as FPGAs and GPUs.

2.4 FPGAS

FPGAs (Field Programmable Gate Arrays) are a type of reprogrammable hardware that are often used to create high speed application-specific hardware designs that may offer substantially higher performance than conventional software implementations on CPUs for some problems [11].

While FPGAs can be used as a prototyping tool for embedded system design, they are often used as hardware accelerators to speed up CPUs. They have drawbacks however, having comparatively high cost and requiring their designs to be specified in specialized hardware description languages. They are also not suited to all applications, and there are situations where a CPU can outperform them [12].

FPGAs can be well suited to computation using finite automata however [13][14], and it is this aspect that we are interested in. See section 2.6 for background information on finite automata.

2.5 GPUS (GRAPHICAL PROCESSING UNITS)

GPUs are specialized processors that were created to accelerate the computationally expensive process of rendering 3D graphics for computer design tools and computer games. Rendering 3D graphics requires repetitive processing of very large numbers of pixels and vertices, which trended toward the development of highly parallel processing architectures with extremely high memory bandwidths to accommodate these requirements.

GPUs operate using a SIMD architecture – single instruction, multiple data. In effect, this means that multiple processors are executing the same instruction in a shared program, but operating on different streams of data. GPU programming languages, such as Nvidia's CUDA [15] and Khronos Group's OpenCL [16] organize work into a *grid*, which consists of a 3D array of threads and data.

One consequence of this design model is that since each GPU processor runs its threads in lockstep in a SIMD manner, each thread must be executing the same instruction. If they diverge, the processor replays the program and masks off the diverging threads, reducing performance. In the worst case, only a single thread is active as all individual threads take different paths.

Memory bandwidth is also highly sensitive to locality, and especially in older CUDA versions, memory bandwidth suffers extremely if processing elements don't access *coalesced* memory addresses that are adjacent to one another.

GPUs are able to perform a wide range of high speed calculations when the operations are independent, highly parallel and free of complex control flow, but the limitations explained above prevent GPUs from achieving high performance on all problems.

2.6 FINITE AUTOMATA

As an abstract model, finite automata, or finite state machines, represent one of the simplest forms of computation. Informally, they are a series of states connected by transitions, with one state considered *active*, a *starting* state, and a set of *accepting* states. Each transition is labeled by a character from an input alphabet, such that the currently active state transitions to the connected state if that character is provided as input to the

machine. The goal is to determine whether an input string from that alphabet is *recognized* by the machine, which is indicated by one of the accepting states being the currently active state when the entire input string has been processed. The set of strings recognized by a finite automaton is said to be the language it recognizes. An example of such a machine is shown in Figure 2.8.

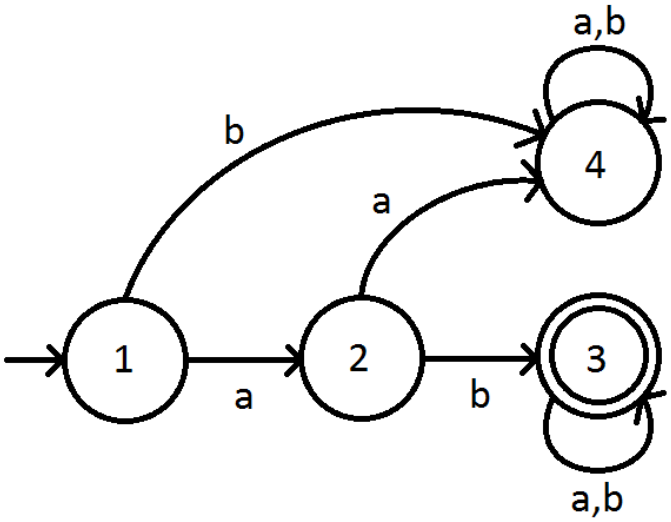


Figure 2.8: Example of DFA. This DFA recognizes any string of length at least 2 which begins with "ab." In this example, $\Sigma = \{a,b\}$, $S = \{1,2,3,4\}$, $s_0 = \{1\}$, δ is represented by the image above, and $F = \{3\}$.

Formally, finite automata are expressed as the 5-tuple $(\Sigma, S, s_0, \delta, F)$ where these are defined as:

- Σ – The input alphabet, which is a finite and non-empty set of symbols
- S – The set of states, which is finite and non-empty
- s_0 – The starting state

- δ – The transition function, which maps each state to another state for each element of the input alphabet
- F – The final (accepting) states, which is finite but may be empty

Finite automata can be further classified into one of many varieties. Two of the most commonly described versions are the DFA (Deterministic Finite Automaton) and NFA (Nondeterministic Finite Automaton). DFAs are characterized by allowing only a single state to be active at once and by requiring that each state have exactly one transition for each element of the alphabet. NFAs instead allow each state to transition to multiple, or even no, other states for each element of the alphabet. While this has implications for how the two machines are defined, in theory they have the same expressive power and algorithms exist to transform NFAs into equivalent DFAs [17], although the resulting DFA may be exponentially larger in the number of states.

In this sense, DFAs and NFAs represent opposite ends of computation and storage complexity spectra [18]. NFAs may have multiple active states, and in the worst case every state may be active. This means that processing an input character may require every state to be evaluated, resulting in a time complexity of O(n), where n is the number of states. NFAs are efficient to store, and require O(n) storage for n states. An equivalent DFA may take up to $O(2^n)$ states in the worst case, but since only a single state is ever active at once processing each input character takes constant time O(1).

By their nature of attempting to recognize input strings as belonging to some language, finite automata are well suited to the problem of string matching, which is an important subproblem of sequence alignment. Specifically, finite automata can be

constructed to recognize regular expressions, which are a compact way of specifying languages of strings.

2.7 REGULAR EXPRESSIONS

Regular expressions are a means of specifying *regular languages*, which are in turn defined as languages that can be recognized by some finite state machine [19]. As a result, regular expressions are effectively a means of expressing finite automata in a compact form.

There are multiple common syntaxes for describing regular expressions, but as an example we will consider a subset of the POSIX Extended Regular Expression (ERE) syntax [20]. ERE regular expressions are given as a sequence of characters consisting of anything from the language alphabet plus a number of metacharacters which give special meaning to the regular characters. These metacharacters are shown in Table 2.1.

Table 2.1: POSIX ERE Metacharacters

Character(s)	Meaning
.	Matches any single character from the alphabet.
[ ]	A bracket expression. This matches any single character contained within the brackets. For example, [abc] would match any of a, b or c.
( )	A grouped subexpression, for use with the alternation operator '|'.
*	Matches the preceding element any number of times, including not at all.
+	Matches the preceding character at least once, but any number of times.
?	Matches the preceding character zero or one times.
|	Alternation, matching either the expression before or after the operator. For example, abc|def matches "abc" **or** "def".

Using these metacharacters, it is possible to build compact expressions that can recognize large classes of strings. Some examples are shown in Table 2.2.

Table 2.2: Example Posix ERE regular expressions.

Expression	Recognized Language
ab.*	Equivalent to the DFA in the previous section: it recognizes any string of length at least 2 that begins with "ab."
(a|b).*	Matches any string beginning with 'a' or 'b'.
[abc]a+b	Matches strings beginning with 'a', 'b' or 'c', followed by at least one 'a' and ending with 'b'.
a?[bcd]	Matches one of 'b', 'c' or 'd', optionally prefixed by 'a'.

As mentioned previously, regular expressions are defined as being equivalent to finite automata, so it is possible to transform any regular expression into its equivalent NFA or DFA using Thompson's Construction Algorithm [21]. Thompson's Construction Algorithm defines a set of rules that are applied recursively to a regular expression to generate its equivalent NFA by transforming metacharacters into sets of states that are connected together. This NFA can in turn be converted to a DFA separately [22].

CHAPTER 3

RELATED WORK

In this chapter, we summarize various implementations of the BLAST algorithm. We begin by presenting software implementations and move on to specialized hardware implementations. We then present examples of using regular expressions to perform pattern matching on FPGAs.

3.1 SEQUENCE ALIGNMENT

In this section we describe various software implementations of BLAST. First we describe the well-known baseline implementation from NCBI, then move on to describe alternative implementations and their advantages and disadvantages.

3.1.1 NCBI BLAST

The original implementation was created and is still maintained by NCBI (the National Center for Biotechnology Information), which also maintains large online catalogs of protein and genetic data for performing sequence alignments [23]. NCBI's implementation sees extensive use, with their web interface seeing over 100,000 queries per day as of 2004 [24], and is commonly used as a baseline in the literature. NCBI BLAST follows the general algorithm for BLAST outlined in the previous section, but specific details on its implementation follow.

On initialization, NCBI BLAST constructs a hash table to store the offsets of any seeds from the query sequence being considered. The hash function uses a mapping from the alphabet being considered to integers, using the fewest possible bits per character.

For example, BLASTN (BLAST for nucleotides) has a four letter alphabet: A C G T. By mapping these characters to the integers 1-4, only 2 bits ($\log_2 4 = 2$) are needed per character. This bit representation of each character is then packed together into a single memory word and used as the hash value for a possible seed. Packing the values allows the seed to be stored compactly, and subsequent seed values can be calculated by a single left shift and binary OR operation to append the next character.

The advantage of this approach is that it allows repeated seed values to be collapsed into a single entry in the hash table and save memory. If a seed value appears multiple times in the query sequence, up to three offsets can be stored directly in the hash table, and if there are more a dynamically sized array is allocated and pointed to in the table. This allows efficient use of the memory hierarchy in the case of relatively few repeated seed values, since the hash table entries should fit into the CPU's cache and only relatively rarely would a more expensive lookup into main memory be needed.

Substitutions are handled by computing every possible permutation of each seed value over the alphabet and computing a "self score" against the original seed using a scoring matrix that maps each character in the alphabet to all others and gives a positive score for likely substitutions or negative scores for unlikely substitutions. If the new seed value has a high enough score then it is retained and added to the list of possible seeds that can generate hits, else it is discarded.

Searching the database sequences is performed by dividing the sequence up into seed values in the same way as for the query string. Each sequence is hashed in the same way, and if a hit is found in the hash table, the offset into the database query is stored.

After searching is completed, all of the seeds are arranged into diagonals based on the query and database hit offsets, and extensions are performed along the diagonals using a substitution matrix and scoring threshold. The extender attempts to merge multiple hits on a single diagonal by extending toward nearby hits first, and if successful, extends as far as possible past the hits until the score drops below the given threshold.

3.1.2 DFA SEARCHING

One alternate way to perform the seeding and searching stage of BLAST is to implement it using finite automata instead of a hash table. One such implementation is given by Cameron et al. [25], where they achieve a 15-30% increase in performance over the NCBI implementation using DFAs and careful management of data so as to efficiently use the CPU cache. They go on to describe that NCBI's oldest implementations used a complex DFA structure before moving on to the hash table currently employed, and the DFA structure was reasonable for the available workstations of the time that rarely had onboard caches.

Their approach is to carefully plan the order of the DFA's state table so as to improve cache performance. They do this by clustering frequently used states in the center of the table, predicted by using the expected frequency of the state's amino acid sequence. The outgoing edges from each state are also ordered by frequency. Lastly, the authors store the query positions outside of the state table, reducing the probabilities of cache misses when the positions are referenced. Together, these changes improve memory locality and thus cache performance.

This structure also uses considerably less memory than the old NCBI BLAST DFA and hash table architectures. By using state minimization and smaller position variables (16-bits where possible), their approach uses as little as 2% of the memory.

The authors note that their approach requires more computation since there are more pointers to dereference, but the results they present imply that it is more than compensated for by the improved memory performance.

3.1.3 MASKING

One method for reducing the number of seeds found, which has a dramatic impact on performance due to the slowness of the extension operations, is to allow nonconsecutive matches to be considered for alignments.

PatternHunter [26] is an example of such an implementation, which uses bit masks to denote positions in the W-mer that must match between the source and query strings. For example, the bit mask 1 0 1 would require that the first and third characters in a 3-mer to match, and not care about the second. PatternHunter calls these patterns models, and using a specific model that is tuned against BLAST, they were able to achieve a 4-5 fold decrease in runtime and memory consumption compared to BLAST. They also demonstrate that it produces significantly better results than MegaBLAST [27], which it also outperformed in runtime and memory consumption. However, PatternHunter is slower than BLAST for small data sets (hundreds of kilobytes).

Masking in this fashion, also called *spaced seeds*, is also used by other tools, such as BLAT [28] and BLASTZ [29] and even recent versions of MegaBLAST.

3.1.4 DISTANCE SEARCHING

Rather than trying to compute the similarity between two alignments, it's possible to use a distance metric instead. MegaBLAST is an example of an implementation that computes distances between two sequences and uses that to rapidly cull entire diagonals from being extended. MegaBLAST uses a simplistic scoring scheme that assigns small positive and negative scores for matches and mismatches respectively, with the assumption that it can greedily discard diagonals after a small drop in score. This has the advantage of being potentially very fast, but loses some sensitivity, and the authors don't provide any performance results.

3.1.5 COMPRESSED ALPHABETS

The size of the hash table generated by NCBI BLAST limits the ability to use larger seed sizes in an attempt to improve performance. This is because the hash table grows very rapidly, at a rate of n^{32} table slots where n is the length of the seed. NCBI BLAST puts a hard limit on the seed size at 5 for this reason.

Sergey et al. [30] proposed a new means of encoding seed values in the hash table in order to allow longer seeds to be used. Their algorithm compresses the protein alphabet by grouping together amino acids that are "similar" enough that not distinguishing between them does not severely affect the accuracy of BLAST. By compressing the alphabet in this manner, they are able to extend BLAST so that it can accept seed sizes of up to 7, although the authors say that results for seeds of size 8 or larger were "not promising."

They use these longer seeds to improve BLAST's performance by also implementing fractional threshold values. The increased length of the seeds allows these

more precise threshold values to be specified, which can maintain the same level of sensitivity while generating substantially fewer seeds. This is implemented by converting the fractional threshold into integer thresholds for use in other phases of the BLAST search, using amino acid background probabilities and rounding.

The authors are able to improve BLAST's runtime by up to 30% using these alterations.

3.2 HARDWARE IMPLEMENTATIONS OF BLAST

In this section we describe various implementations of BLAST using specialized hardware, such as PLAs, FPGAs and GPUs.

3.2.1 SEEDING ON FPGAS

Some FPGA implementations of BLAST focus primarily on the seeding operation. This is likely due to the fact that this takes up 70-90% of the time spent in BLAST [31]. As a result, the greatest benefits are expected to be seen by accelerating this portion of the algorithm, although by Amdahl's Law the theoretical maximum global speedup would be a factor of about 10x.

RC-BLAST was an early attempt to implement the seeding operation on an FPGA, using a Xilinx XC4085XLA FPGA. RC-BLAST implements a lookup table similar to NCBI BLAST, and as the authors discovered, this was poorly suited to the architecture of the FPGA they chose. RC-BLAST was about 5x slower than the software version using a conventional processor of the time, which the authors attribute to poor RAM speeds, PCI bus speeds and poor utilization of the FPGA's resources.

A more successful implementation of BLAST was realized with TreeBLAST [32], which was implemented on a Xilinx Virtex-II Pro XC2VP70-5 FPGA. TreeBLAST is an

example of an implementation using systolic arrays, which are a form of pipelining computations by pushing data elements through an array of specialized processors one cycle at a time [33]. TreeBLAST uses a tree merging structure to calculate alignment scores, which allows it to compute a score every clock cycle while reducing the number of registers needed. The query is stored in an array of registers, with each register storing the binary representation of an element of the query. The database entry is systolically "slid" over the registers, and each register compares to its corresponding entry in the database entry. The scores are fed forward, two at a time, into a leaf, which continues until a single score is left, representing the alignment of the query against the database at that position. A high level view of TreeBLAST can be seen in Figure 3.1.

The performance of TreeBLAST varies with the size of the query, but results of up to about 35x speedup per query were reported compared to NCBI BLAST on a 2.8Ghz Xeon processor, at a rate of $1.7x10^8$ characters per second peak throughput achieved with nucleotide searches.

3.2.2 FULL BLAST ON FPGAS

There are also full implementations of BLAST on FPGAs, which also perform extension of hits in hardware.

One early implementation of BLAST on an FPGA that included the full algorithm was performed by Chen Chang on a BEE2 system [34]. The BEE2 is a specialized hardware system that uses conventional hardware where possible, such as DRAM modules, but includes a fully programmable FPGA as its processing elements (in this case, a Xilinx Virtex 2 Pro). BLAST is implemented in a similar way to NCBI BLAST, using a hash table to index seed values. However, one of the design goals is to maximize

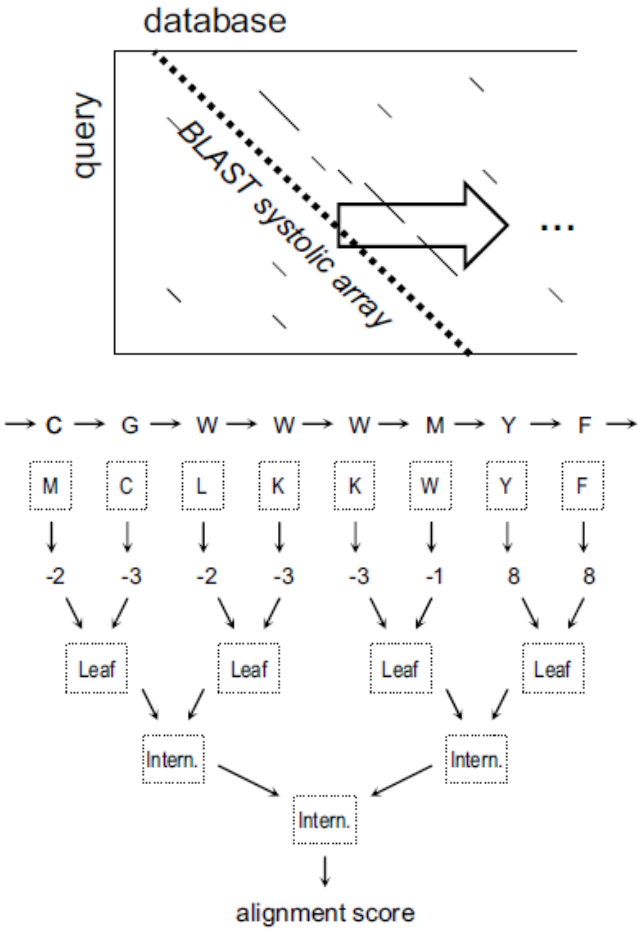


Figure 3.1: TreeBLAST. Top: How TreeBLAST proceeds along the diagonals in a "wavefront." Bottom: The tree merging structure used by TreeBLAST. Each level represents a clock cycle.

memory efficiency since it is the primary bottleneck, and as a result the hash table values are bitpacked to eliminate storage waste. Hit indexing and expansion are done using a second table that is indexed by the alignment's diagonal, which is expanded in parallel to maximize performance.

The author goes on to show that performance is highly dependent on memory access patterns, and a lot of effort was put into optimizing them. The overall

performance was a substantial improvement over CPU implementations, with 4 BEE2 systems operating at about 10x the speed of a 128-node cluster of 667 Mhz CPUs.

The TUC (Technical University of Crete) architecture is a different take on BLAST that implements a highly parallel system to increase throughput [35]. TUC is fundamentally a systolic system which pushes 2-bit encodings of nucleotide characters into a shift array, which stores 11 characters in total as is the standard for BLASTn. The 11 characters are combined into a single register value, which is then sliced off into a 10-bit 'tag' which is used to index a 1000 word RAM of w-mer positions, which takes up the remaining 12 bits. This RAM is prepopulated before the system begins by finding all seeds in the query and noting down their indices. If hits are found the position and tag are forwarded to an extension unit which performs extensions in both directions as per the normal BLAST extension algorithm.

TUC is able to achieve extremely high throughput by arranging multiple copies of this computational hardware into independent nodes, each of which can operate on a database query. The authors were able to fit 69 parallel computational units onto their Xilinx Virtex 4 FPGA, which were able to achieve a speedup of 215.12 against a single core of a 2Ghz Xeon processor running NCBI BLAST. The peak throughput for a single computational unit is 1.21×10^8 characters per second. Curiously, they report that 69 computational units only perform 5.76x as well as 16 cores of the same Xeon processor, implying better than linear speedup for NCBI BLAST on a CPU.

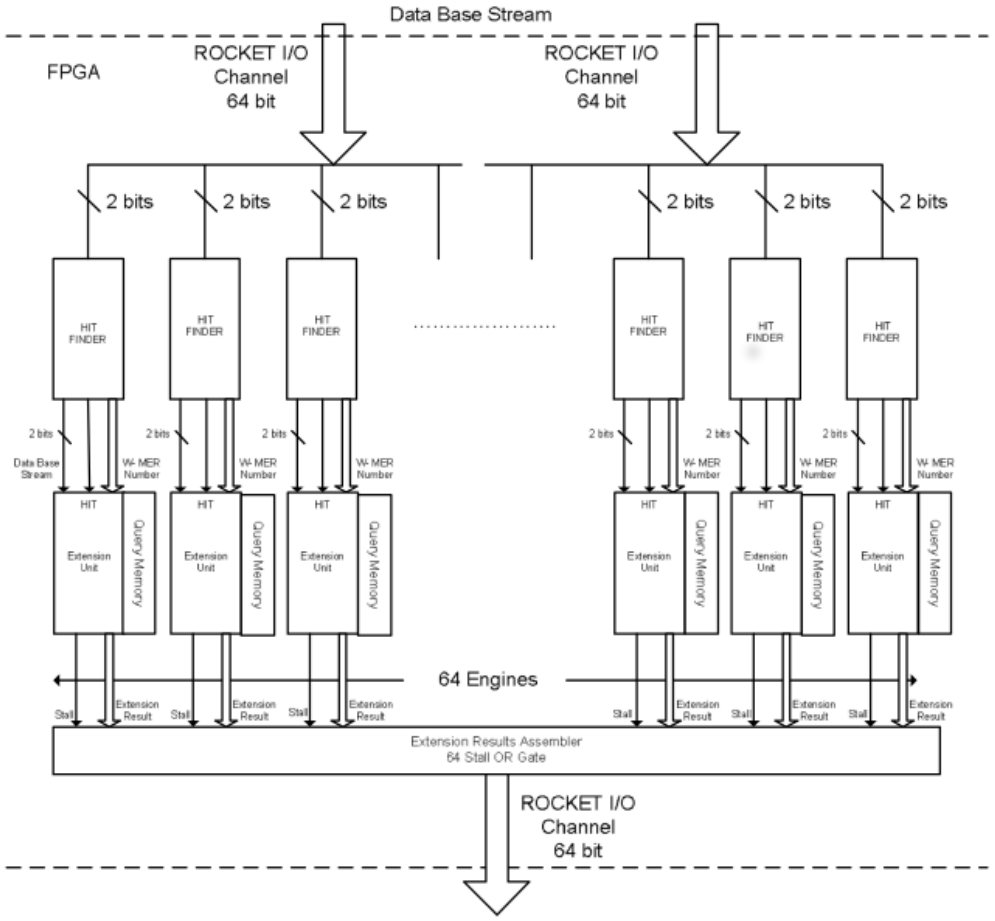


Figure 3.2: TUC functional unit overview. Rocket I/O channels feed into finder units, which feed into extender units, which are aggregated before being returned to the host.

Another architecture was designed by Guo et al. [36], which uses systolic arrays in the seeding phase to find multiple hits per clock cycle. This is accomplished by feeding 32 characters from the database sequence in at a time and forwarding groups of three comparator units into binary AND gates. Any AND gates that output a high signal are in effect indicating a hit, so up to 10 hits can be detected per clock cycle.

Although the authors don't go into details on how their gapped or ungapped extensions work, they show that it achieves considerable speedup over NCBI BLAST in all cases. BLASTp is about 17x faster for large queries, but for small queries there is

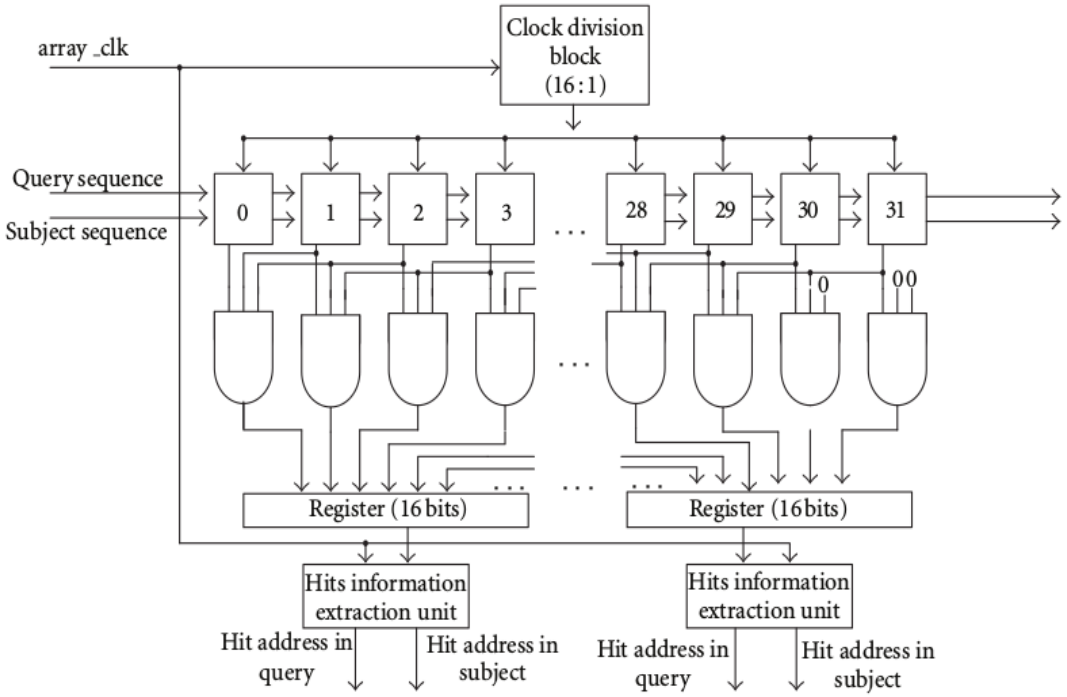


Figure 3.3: Guo et al's architecture overview. Characters are shifted through the systolic array, where groups of AND gates aggregate multiple hits per clock cycle.

relatively little improvement in speed. Peak performance was cited as 1.445 x 10^{10} characters per second.

Mahram and Herbordt designed a full FPGA implementation of NCBI BLASTp that makes use of successive layers of filters, similar to the Bloom filters used by MercuryBLAST (described later), to dramatically reduce the workload and thus achieve high performance. The first step is to apply a filter that discards most database queries that are not promising matches to the search query, reducing the database size by 97-99.7%. Afterward, depending on which type of extension is to be performed, additional filters are applied to further reduce the work load, where each filter is tuned to give results at least as good as NCBI BLASTp's results. Total peak speedup is approximately

5x over the CPU of the Convey system used in the tests, compared to a single FPGA card.

3.2.3 SEQUENCE ALIGNMENT ON GPUS

Relatively few implementations of BLAST exist for GPUs, but several implementations of Smith-Waterman are described in the literature. These implementations are able to take advantage of the very high computational throughput and memory bandwidth of GPUs to perform the calculations needed for the dynamic programming algorithm. Performance for Smith-Waterman implementations is usually given in CUPS – cell updates per second, and is a number that reflects how many cells in the dynamic programming matrix are populated per second.

Liu et al. [37] produced the earliest implementation, predating modern GPGPU languages and implementing Smith-Waterman in programmable graphics pipelines. They were able to achieve about a 20x speedup over CPU implementations of the time, peaking at around 0.65 GCUPS.

Manavski et al. [38] designed an implementation using CUDA that was able to achieve about 3x speedup over Liu's implementation, peaking at about 1.89 GCUPS. The authors also give performance comparisons to NCBI BLAST on CPUs, and by using two GeForce 8800 GTX cards they were able to outperform BLAST by a factor of 2.4. Ligowski et al. [39] revised this design, which uses horizontal strides over the matrix to achieve a 4x improvement over Manavski's implementation, peaking at 7.5 GCUPS per GPU.

In 2010, Ling et al. [40] produced a BLAST implementation using CUDA that was able to achieve a peak of 2.7x better performance than a Pentium 4 at 3.4 Ghz. Their

design followed the NCBI algorithm closely and used the GPU to search for many seeds at once in parallel while leveraging its very high memory bandwidth to maximize performance. Extensions are performed in a separate kernel and require some synchronization between threads, so performance hinges on the number of hits found by the first kernel and their distribution.

Vouzis et al. [41] later produced an implementation that modified NCBI BLAST directly to replace the ungapped word finder and extender with kernels that ran on both the GPU and CPU at the same time to maximize performance. To minimize performance degradation from the SIMD nature of GPU kernels, they sorted database samples so that each thread operated on approximately the same length strings. Some operations are performed exclusively on the CPU, such as gapped extensions, which are not well suited to the massively parallel SIMD architecture. Using a Fermi C2050 GPU and a six core Xeon at 2.67 Ghz, the authors were able to get a peak of slightly over 3x speedup when using only a single CPU core, but with all six cores they only managed about 1.5x speedup.

3.2.4 OTHER HARDWARE IMPLEMENTATIONS

MercuryBLASTN [42] is an example of BLASTn implemented on the Mercury system, which is a specialized hardware platform designed to use reconfigurable logic in the form of an FPGA to process streams of data from a disk backed storage system [43].

MercuryBLASTN implements only the hit finding component of BLAST, but uses a multi-stage design to do so that achieves significant speedup. Hit finding is performed by sending in 11-character nucleotide seeds from the query sequence, which are then compared against streamed database entries by using a Bloom filter. Bloom

filters are a means of quickly testing membership in a large set through the application of hashing functions on the set [44], but have the drawback that they may report false positives when queried due to loss of information in the hashing step.

In MercuryBLASTN, Bloom filters are used to reduce the amount of data that must be checked by later stages of the hit finding algorithm. Database entries are run through the bloom filter using the query seeds and hits are forwarded on to the second stage. The second stage is a hash table implementation similar to NCBI BLAST's hash table, and is used to store indexes for hits from the first stage. The third and final stage is a redundancy filter that is used to eliminate false positives generated from the first stage and cull hits that aren't at the maximum extent along a diagonal as they would be unnecessary in the extension phase.

Performance for the hit finder reaches a peak speedup of about 45x over NCBI BLAST's hit finder running on a CPU, but overall performance is hindered by the fact that the remaining parts of the BLAST algorithm are not implemented in hardware. Overall peak performance was only about 7x better than NCBI BLAST, at about 2.128 $x10^9$ characters per second.

An implementation of BLASTp on the Mercury system was described by Jacob et al. [45], called MercuryBLASTP. Unlike Mercury BLASTN, MercuryBLASTP implements the entire BLAST algorithm in hardware and does not use Bloom filters to process potential hits. Instead, a lookup table, similar to the hash table used by NCBI BLAST, is used to perform hit finding. By using an encoding scheme that only stores the difference in position between hits, the authors are able to pack more hits directly into the table without having to resort to secondary lookups, which improves performance. To

improve performance and get around hardware limitations on the number of ports available to on chip memories, the extension phase divides the results into diagonals using a modulo operation that efficiently maps the hits to multiple extender units without requiring them all to share data. With this architecture, the authors achieved a speedup of about 37x over a 2 Ghz AMD Opteron processor.

3.3 PATTERN MATCHING WITH REGULAR EXPRESSIONS ON FPGAS

In this section, we present some implementations of pattern matching on FPGAs, with a particular focus on doing so using regular expressions and finite automata.

Regular expression matching has been implemented in hardware as far back as the use of PLAs, predating FPGAs, such as the implementation by Floyd et al [46]. Their implementation is a means of encoding regular expressions directly into hardware circuitry through the use of lookup tables that store next state transitions for specific input characters. Their design implements regular expression matching in NFAs directly, allowing multiple states to be active at once.

Sidhu et al [47] describe one of the first regular expression matching designs on FPGAs that use NFAs. Their design uses an extension of the One-Hot Encoding scheme which allows multiple states to be active at once. One-Hot Encoding is a means of implementing finite automata wherein the set of states are represented as a bit string with only one bit active (hot) at a time. The active bit is used to index into a table indicating which state should be active next for a given input. In their implementation, each state is represented as a flip-flop, having on or off status. By connecting the output of a flip-flop to multiple flip-flop inputs, a single state can trigger more than one other state and trivially implement a state machine where multiple states are active. Furthermore, their

design is able to implement a regular expression as an NFA at run time, using only O(n) time and $O(n^2)$ space, where *n* is the length of the regular expression of interest.

Using DFAs with One-Hot Encoding has the advantages of being simple and usually allowing the FPGA to run at a higher clock rate, which can otherwise be constrained by complex logic introducing delays in the circuit. DFAs created from NFAs (which are in turn generated from regular expressions) can potentially have exponentially large numbers of states, so doing this in a naive fashion is rarely possible for interesting problem sizes.

Various authors have proposed means of either reducing the complexity of the regular expression to combat this, such as removing ambiguities [48], or by proposing extensions to DFAs, such as counters to recognize previously seen parts of the expression [49], that enable the state tree to be collapsed in some fashion.

Becchi et al. [50] provide several techniques for reducing the complexity of NFAs generated from regular expressions, focusing on reducing the number of states or transitions. The authors propose a reduction operation to eliminate epsilon transitions (those that occur on no input character), as they tend to increase the total number of states as well as the average number of active states. They also propose an algorithm that is able to reduce the number of transitions in the NFA by reducing it to matching on a reduced alphabet. If multiple elements of the alphabet always follow the same transitions, they can be merged into a single character in a reduced alphabet. Lastly, the authors propose a multi-stride NFA that processes multiple input characters at a time. They present an algorithm for generating a 2-NFA from a normal NFA, which has

approximately the same number of states but may have more transitions. Alphabet reduction can help reduce the number of transitions afterward.

An observation by Korenek et al. [51] is that oftentimes only a small number of the total NFA states might be active at a time, and go on to show that for network intrusion detection using regular expressions, as little as 3% of the states might be active when other states are active. They use a split NFA / DFA architecture that keeps states that are always active alone in memory and use a DFA to process them, while using an NFA for the other states. It is unclear how well this would perform for regular expressions used for biosequencing however.

CHAPTER 4

APPROACH

In this chapter we provide an overview of our proposed replacement of BLAST's two-hit filtering stage. Unlike NCBI BLAST, which performs this as separate hit finding and HSP generation steps, our approach combines both into a single operation that can be offloaded onto accelerator hardware.

We have developed a framework for decomposing FASTA protein databases into regular expressions that directly map to HSPs, which are then used to build an index for the database. When queries are fed into the regular expressions, matches can be cross referenced into the index to generate hits that can be evaluated like BLAST's normal seed hits. An overview of our framework is shown in Figure 4.1.

4.1 DATABASE PREPROCESSING

Before we can search a database, we must preprocess it to generate a set of tables that are used at runtime to rapidly filter the database to a subset that is passed on to be aligned. Preprocessing the database consists of building up a set of regular expressions that represent potential HSPS, then indexing the database by these HSPs.

4.1.1 GENERATING REGULAR EXPRESSIONS

Database preprocessing begins by generating a set of regular expressions that are able to map queries to database records. To do this, we have extended and modified the normal hit finding phase of BLAST, which as described in Chapter 2, is the stage where BLAST attempts to find significant segments (called seeds) of a sequence using its

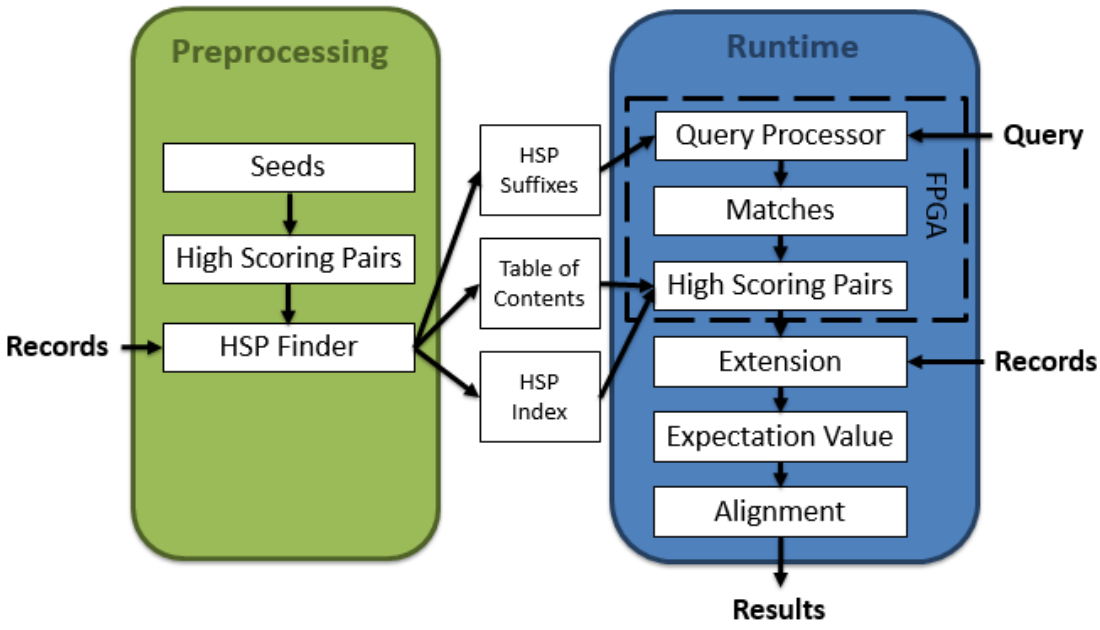


Figure 4.1: Overall Approach. The database is preprocessed to produce the HSP suffix table and HSP index table. Queries are filtered through both tables to produce a set of potential database matches.

scoring criteria. Instead of just locating seeds in a database sequence, we seek to fully form sets of High Scoring Pairs (HSPs) at this stage to generate regular expressions.

First, we seek to establish the practicality of our approach by estimating the number of HSP regular expressions that will be needed to describe a database. There are a finite number of possible seeds for a given seed length and alphabet size, which is further constrained by the threshold parameter. Protein BLAST is typically performed with a seed length of 3 with a standard protein alphabet consisting of 23 amino acids represented by the uppercase characters from the English alphabet. Thus, before elimination of seeds by the threshold parameter, there are 23^3, or 12,167 possible combinations of characters used to generate seeds.

Recall that after finding seed hits in a query or database record, BLAST proceeds to combine nearby pairs of hits into HSPs. NCBI BLAST applies two criteria to possible

HSP candidates before accepting them: pairs of hits must not overlap and they must appear within a certain distance of one another in the original sequence. For NCBI BLAST, this HSP Window Size is typically specified as a distance of 40 characters.

Based on this, we can design a regular expression template that all HSPs would match, as shown in Figure 4.2.

ABC.{0,40}DEF

Figure 4.2: HSP Regular Expression Template

The first three characters represent the first seed hit, or **prefix**, that wold produce the HSP, while the second three characters represent the second hit, or **suffix** of the HSP. The wildcard, ranging from 0 to 40 characters, represents the HSP Window Size, forcing matches to be equivalent to two seed hits that would be close enough to generate an HSP in NCBI BLAST's two-hit filtering stage.

Because there are a finite number of seeds, there are a finite number of possible unique HSPs needed to describe all possible hits in a database of arbitrary size. In the worst case, every possible seed is paired up with every possible seed, including itself, in some sequence, giving us a maximum of $12{,}167^2$ or about 148 million possible HSPs. Applying the seed score threshold reduces this number by reducing the number of seeds meeting the minimum self-score, as shown in Figure 4.3.

As Figure 4.3 shows, a typical threshold value of 13 gives around 130 million HSPs. 130 million HSPs is too large to fit in an on-chip filter, so we are forced to find some way to further reduce the number of HSP regular expressions.

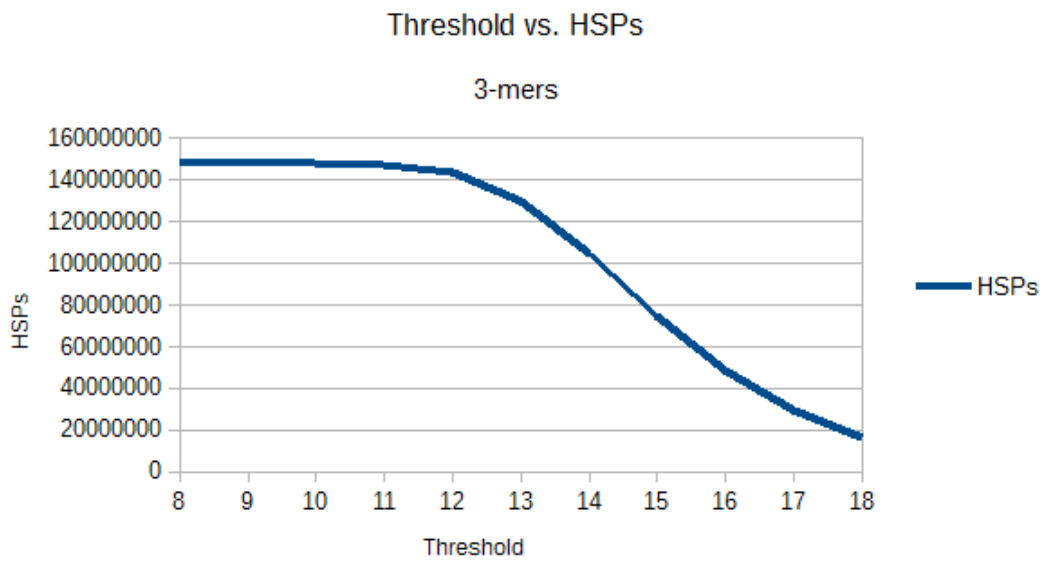


Figure 4.3: Total HSPs By Threshold. Increasing the threshold results in fewer candidate seeds, and as a result, possible HSPs.

As Figure 4.3 shows, even very high threshold values (which reduce sensitivity) are not enough to reduce the number of HSPs to an acceptable level. To accomplish a substantial reduction in the number of seeds and HSPs, we use 2-character seeds, or 2-mers, instead of 3-mers. Using 2-mers instead of 3-mers reduces the number of possible seeds down to 23^2, or 529. In turn, the maximum number of possible HSPs drops to 529^2, or 279,841. This represents a savings of 99.8% in the number of regular expressions needed. To compensate for reducing the size of the seeds, we have to use a lower threshold value, and in Chapter 5 we present data to show that this does not substantially alter the results of the search.

In addition to these base regular expressions, we must also allow inexact matches like BLAST. While the wildcard HSP window allows insertions and deletions, we must modify our regular expressions to allow substitutions in the prefixes and suffixes. This

does not alter the total number of HSPs, but does influence the indexing operation by allowing mismatches when scanning the database records.

In NCBI BLAST this inexact matching is handled by enumerating all of the possible substitutions for each seed and computing the substitution scores. Any seed whose score remains equal to or above the threshold is kept and used in the seed finding phase.

We produce similar behavior by also enumerating all possible substitutions for each seed and retaining those seeds whose scores remain equal to or above the threshold. In order to express the allowed substitutions in the regexes, all prefixes and suffixes that are generated from a seed are grouped together in alternation clauses. An example of a regular expression allowing substitutions is shown in Figure 4.4.

Continuing the example in Figure 4.4, a database sequence containing the subsequence H I A B C C Q could now be matched up against a query that contained the subsequence H J A B C C E.

4.1.2 INDEXING FASTA DATABASES

After the set of possible HSPs has been generated, the FASTA protein database is scanned and a set of index files are generated. The first, which we term the **HSP Index File**, contains a comprehensive index that maps all of the HSP regular expressions to HSP hits in the input database. Each hit stores the index of the FASTA database record that produced the hit, the offset into the record where the hit begins and how long the match was in the database record. This file is larger than the input database and its size grows linearly with input size, which we cover in further detail in Chapter 5.

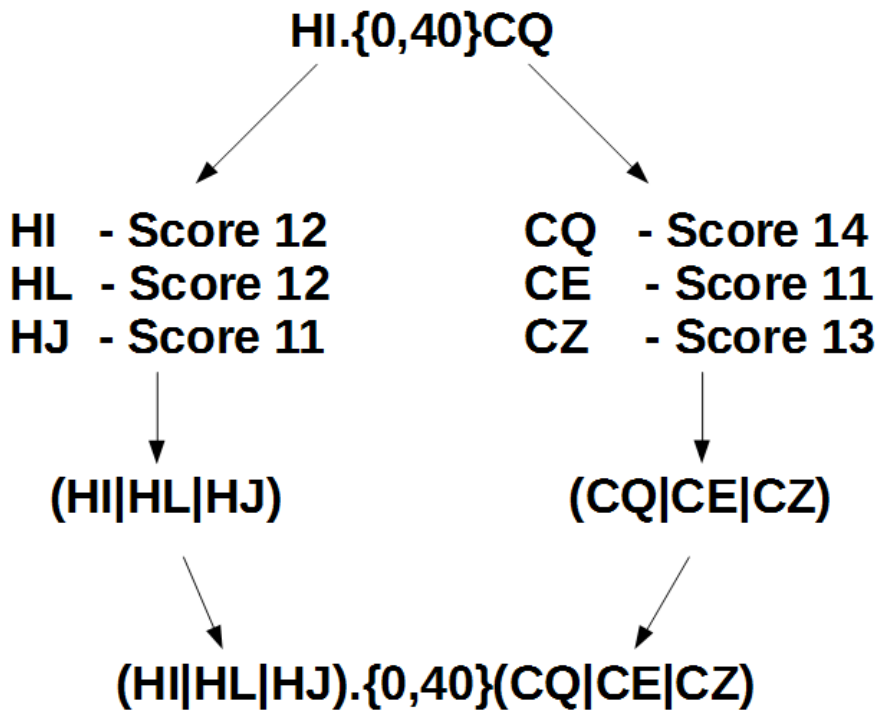


Figure 4.4: HSP Regular Expression With Substitutions. This assumes a threshold value of 11.

The hits in the HSP Index Table are sorted so that all of the hits for a given regular expression can be read in a single batch operation. To permit this, a secondary, small and fixed size table of contents is generated that maps each combination regular expression and match length (0 through 40) to its corresponding hits in the HSP Index Table.

Lastly, the HSP regular expressions are stored in a specialized encoded format, called the **HSP Suffix Table**, that allows the FPGA hardware to identify hits in a query sequence as it is streamed in. Details for the FPGA implementation and the format of this table are presented later in this chapter.

The indexing operation proceeds by tokenizing each database record into overlapping 2-character chunks, or 2-mers. These 2-mers then have their self-scores

evaluated using a scoring matrix, and 2-mers with a sufficiently high self score are kept, and others are discarded. An example of this is shown in Figure 4.5.

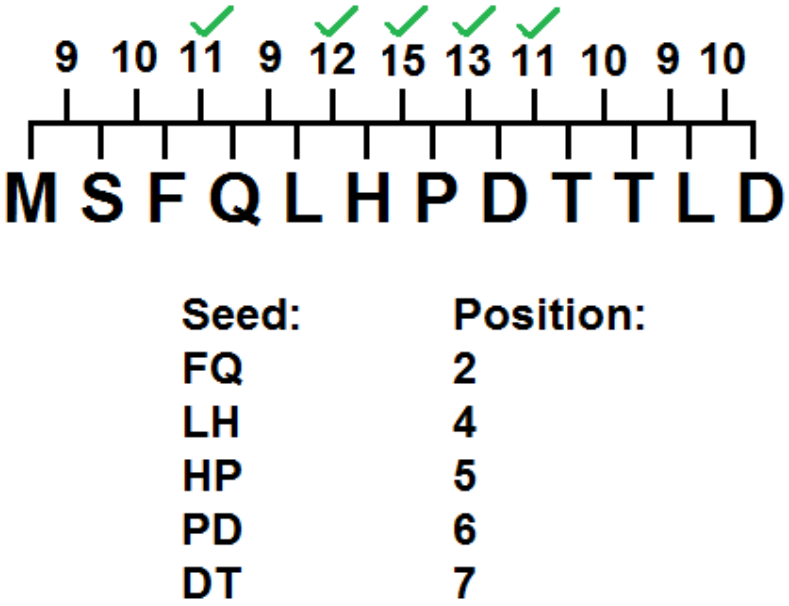
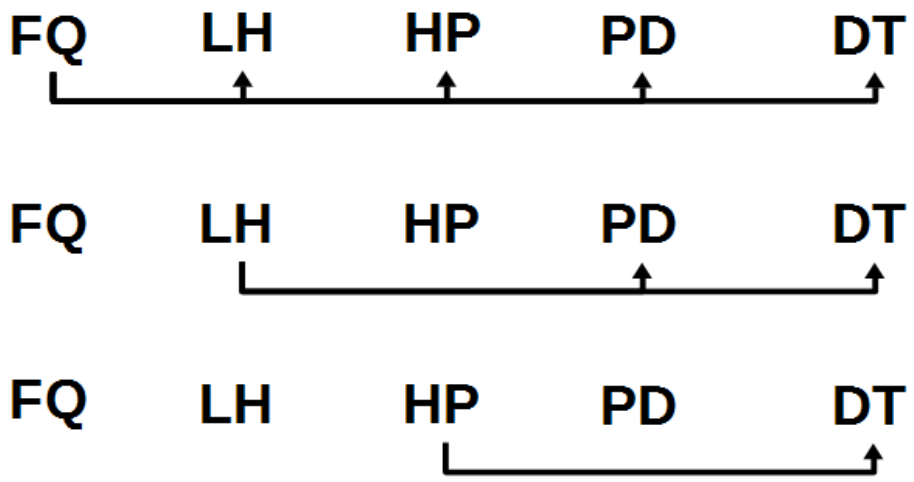


Figure 4.5: 2-mer Tokenization Example. Scores for each 2-mer are shown above the component characters, and are computed using self scores from the BLOSUM62 scoring matrix. All seeds with a score equal to or above the threshold (11 in this example) are listed below the sequence.

After tokenizing the record, the 2-mers are paired up into HSP hit candidates. All of the 2-mers are paired up with all non overlapping 2-mers following them, within the HSP Window Size. By producing HSPs from all possible combinations of hits, we maintain high sensitivity and mirror the behavior of NCBI BLAST.

An example of the pairing process is shown in Figure 4.6, using the seeds generated in the example from Figure 4.5.



Resulting HSP Regular Expressions:

FQ.{0,40}LH	LH.{0,40}PD	HP.{0,40}DT
FQ.{0,40}LH	LH.{0,40}DT	
FQ.{0,40}LH		
FQ.{0,40}LH		

Figure 4.6: Hit Pairing Example. Note that pairs that would overlap in the original sequence are not used to generate HSPs (see Figure 4.4 for the original sequence).

As Figure 4.6 illustrates, the starting and ending 2-mers for each hit correspond to a resulting regular expression HSP. For regular expressions that allow mismatches through alternations, the hit is associated with whichever regular expression contains the prefix and suffix of the hit. Once the corresponding regular expression has been identified for the hit, the hit's starting offset into the database record, length and the record number of the corresponding database record are stored in the HSP Index File for use during the two-hit filter stage while evaluating queries.

4.2 RUNTIME BEHAVIOR

At runtime, the filter tables produced during the preprocessing step are used by the FPGA to transform a batch of queries into a set of database records that must be aligned. In this section, we first describe the overall query matching behavior and then provide a more thorough description of the FPGA architecture.

4.2.1 QUERY HIT FINDING

In NCBI BLAST, the tokenization and HSP generation phases described in Chapter 3 are done for both the query and all database records at runtime. This is necessary because NCBI BLAST does not have a pregenerated database index.

Our approach performs an offline indexing operation for database records, but the query that is being searched for is not available until runtime. To generate hits for the query, we now leverage the HSP regular expressions that were generated during the database indexing phase.

For every query, every HSP regular expression is evaluated and all of the matches and their lengths are recorded. The regular expressions are evaluated in an ungreedy fashion, ensuring that no hits are missed. Details on specific implementations of the regular expression evaluation stage are presented in the following sections.

After all of the matches from the query have been located, they must be cross referenced to the HSP hits generated during the offline indexing phase. Each regular expression that generated at least one match in the query has its hits loaded from the HSP Index File, as well as all of the database records associated with its hits. The final list of HSP candidates is generated by producing diagonals using the regular expression matches from the query against database hits.

In NCBI BLAST the concept of *diagonals* is used to represent alignments that occur at different offsets between two sequences. In NCBI BLAST, this is computed by finding the distance between hits that occur in the query sequence and candidate database sequence. NCBI BLAST only extends hits that occur on the same diagonal.

In our approach, we must also take the length of matches into account, since a change in the length of the match between the query and sequence would indicate that the prefix and suffix seeds would have different diagonals in NCBI BLAST. Since we use the same extension algorithm as NCBI BLAST, which requires seeds forming HSPs to be on the same diagonal, we must enforce the same constraint that query matches and database HSP hits be of the same length. For each query match and database HSP hit of the same length, a candidate HSP is generated and placed on the diagonal calculated as the difference between the query match starting position and the database HSP hit starting position.

An example of an insertion causing a possible HSP to be discarded is shown in Figure 4.7. Note that while this HSP is discarded, other HSP matches could still be found at later positions in the two sequences, but would now be on a different diagonal.

After all of the hits have been extended and their expectation values calculated, alignment proceeds exactly as in NCBI BLAST.

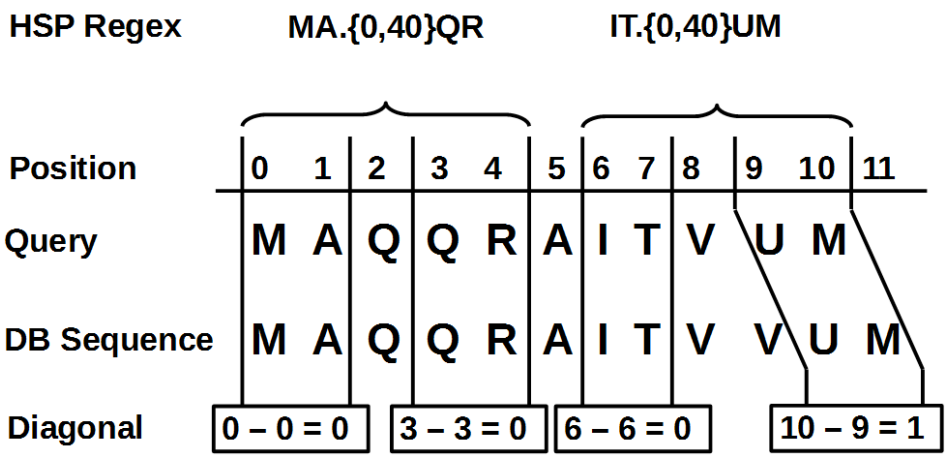


Figure 4.7: Diagonal Calculation Example. The insertion in the database sequence causes the second HSP match length to be longer (6 characters) than on the query (5 characters). Since this would cause the corresponding seeds to have different diagonals in NCBI BLAST, it is discarded.

4.2.2 FPGA PATTERN MATCHER DESIGN

An overview of our FPGA design is shown in Figure 4.8.

The design consists of a query processor, which is given a query stream to process and outputs a stream of HSP matches on the query indicating which HSP generated the match, where it begins and how long the match is. The query processor is composed of a set of processing elements (PEs), an HSP suffix table and an arbitrator for aggregating output hits from the PEs.

Each PE contains the logic needed to recognize an HSP hit from the incoming character stream. It takes as input the current characters from the stream that are being evaluated, the encoded corresponding suffixes that generate a hit from the suffix table and busy signals from the other PEs. Using counters, it is able to record the starting cycle and ending cycle of a match (needed for the start and length), and to abort matching once the HSP Window Size has been reached.

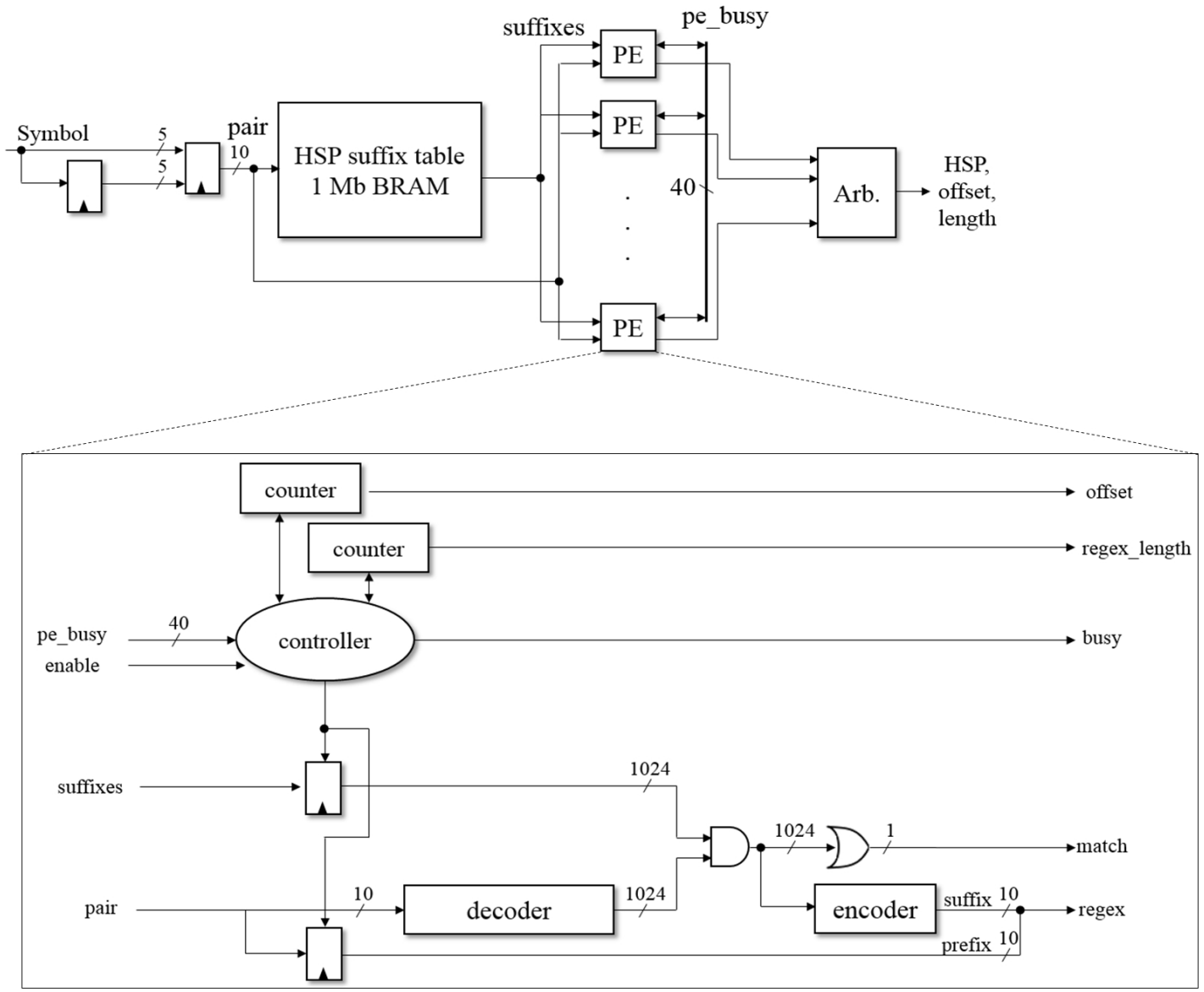


Figure 4.8: FPGA Two-Hit Filter Architecture

The HSP suffix table is a RAM that contains an encoding of HSPs in a compact and rapidly accessible form. Each address line of the RAM corresponds to the beginning of an HSP that uses a modified ASCII encoding. Since the protein alphabet consists of 23 characters, each character can be represented uniquely by 5 bits, meaning that the beginning of an HSP requires 10 bits of address space, or 1024 lines. Each line is a bit vector where each column represents an ending suffix. For every regular expression HSP that begins with a given HSP prefix, all of its suffix values are encoded in the RAM by converting the suffix to a 10-bit value and toggling the corresponding column in the prefix's row. Each line must be 1024 bits wide to accommodate this, leading to a total usage of 1 Mb per query processor. An example of encoding a single HSP regular expression in the HSP Suffix Table is shown in Figure 4.9.

During evaluation, the query is streamed into the query processor, which buffers incoming characters to create 10-bit character hashes that can be used to directly address the prefix table. This hash, as well as its corresponding prefix table value, are passed to all of the query processor's PEs, which begin to compare against it.

Each PE will begin to match on the incoming character hash only if all of the PEs before it are currently busy. On its first active cycle, it stores the incoming hash value as its prefix. On subsequent cycles, it begins counting the length of a possible match and begins comparing the incoming hash to the prefix table's output using a 1024-bit wide decoder. If the decoder detects that the prefix table has a column matching the incoming character hash, then a match is reported. If the counter exceeds the HSP Window Size, then the PE resets and goes idle.

HSP Regex: (AA|AR).{0,40}(AA|AC|CD|PT)

Seed Encodings

AA = 00000 00000$_b$ = 0
AB = 00000 00001$_b$ = 1
AC = 00000 00002$_b$ = 2
AR = 00000 10001$_b$ = 17
CD = 00010 00011$_b$ = 67
PT = 01111 10011$_b$ = 499

Address	Value (each column is 1 bit)												
	0	1	2	...	66	67	68	...	498	499	500	...	1023
0	1	0	1	...	0	1	0	...	0	1	0	...	0
1	1	0	1	...	0	1	0	...	0	1	0	...	0
..	...	...	...	...	...	...	...	...	...	...	...	...	...
17	1	0	1	...	0	1	0	...	0	1	0	...	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
1023	1	0	1	...	0	1	0	...	0	1	0	...	0

Figure 4.9: HSP Suffix Table Generation. Rows are addressed by the encoded version of each prefix, and columns are toggled for each suffix of each prefix.

Each query processor contains 42 PEs, which are enough to ensure that with an HSP Window Size of 40, there are enough to process a character every cycle without needing to stall. We verified that 42 PEs are enough through the use of the software model described below in Section 4.5.

Hits generated by the query processors are stored in DRAM, which are then processed by the HSP generator to produce a final list of HSP hits to be passed back to the host. The HSP generator is responsible for sorting the hits by HSP ID and length so that database hits can be referenced from the HSP Index File with the minimum number of transactions possible. It then calculates the diagonal for each hit by subtracting the

database record hit start position from the query hit start position, and adds it to the final hit to be sent back to the host for extension and alignment.

4.3 SOFTWARE MODEL

To evaluate the feasibility of this approach, a software model was implemented using C++ that emulated the functional behavior of the FPGA model. We used this model to prove the correctness of the algorithm in the absence of a complete physical implementation, and to verify the minimum number of PEs needed per query processor to ensure no stalls would be needed to process a character every cycle.

This model also allowed us to make estimations of performance on a real system. We were able estimate the I/O overhead of loading HSP hits from disk during the seed search phase, and gather timing information on how long the host system would spend performing an equivalent search operation in software.

4.4 SUMMARY

In this chapter we provided an overview of the offline database preprocessing algorithm needed to build the HSP Hit Index table, as well as an overview of the FPGA two-hit filter architecture that would leverage the regular expressions generated by the preprocessing stage to perform streaming rate pattern matching on query sequences.

In Chapter 5, we present results gathered from our software emulation model of the FPGA architecture that show projected performance on a real world system, accounting for the measured overhead of reading the HSP Hit Index table from disk. We also show results comparing the relative time spent in the two-hit filter stage by NCBI BLAST and provide estimated maximum speedups.

CHAPTER 5

EXPERIMENTAL EVALUATION

In this chapter, we provide experimental results showing that our framework has comparable sensitivity to NCBI BLAST as well as performance estimations that show that we expect significant speedups on real hardware. We also present projections on disk utilization for our index files as well as hardware utilization for both the FPGA architecture.

5.1 RESULT VERIFICATION

To verify that our framework produces equivalent results to NCBI BLAST, we used the four protein databases NR [52], Uniref50, Uniref90 and Uniref100 [53]. NR is a commonly used protein database aggregating records from several other sources in a non-redundant fashion. The Uniref databases are generated from the UniProt Knowledgebase, which serves as another hub of aggregating protein sequences from multiple sources. The Uniref databases attempt to merge multiple sequences from different sources which have high overlap into a single sequence to reduce the size of the database and search times at the loss of resolution. Uniref50 combines sequences of 50% identity (that is, sequences which are identical in 50% or more of their characters), Uniref90 combines sequences of 90% identity, and Uniref100 combines sequences of exact identity. We chose these databases with the expectation that we would see different numbers of extensions and alignments for a given query due to the varying levels of redundancy.

We performed tests by selecting random sequences from these databases and feeding them back in as queries using NCBI BLAST and our framework. All of our tests were performed using the default parameters for NCBI BLAST: threshold of 13 and seed size of 3, and using threshold 11 and seed size 2 for our framework. We chose this threshold based on backward extrapolation from the original BLAST paper, which suggested a threshold step of 2 for each step of seed length gives similar sensitivities [8].

Our experiments show that highly significant alignments, that is, those with very low expectation values, are equivalent between our framework and NCBI BLAST. However, both NCBI BLAST and our framework return alignments with much higher expectation values, and these results were less consistent. In Tables 5.1 and 5.2 we present an example of the output obtained by running the database sequence gi|49187252 from the NR database as a query against the first 10,000 records of the NR database in both NCBI BLAST and our framework.

In the following tables, the SW Rank column indicates the relative ranking of the alignment in a list sorted by expectation value when Smith-Waterman is run directly between the query and database records. Running Smith-Waterman on the query record and its database counterpart gives the highest possible score and lowest possible expectation value, for example, and thus receives a SW Rank of 1. This gives optimal alignments for every database record and allows us to directly compare the results from NCBI BLAST and our framework by showing gaps (indicating missed alignments) in the results.

For brevity's sake we present only this example, but its results are typical of all queries tested from the four databases mentioned.

Table 5.1: Results of Using gi|49187252 as a Query in NCBI BLAST against NR.

Sequence Name	Score (Bits)	E-Value	SW Rank
gi|49187252	299	6.53e-86	1
gi|15925512	75.1	1.17e-18	2
gi|15924184	59.7	5.1e-14	3
gi|22537123	30.8	2.54e-05	4
gi|19746167	28.9	9.65e-05	5
gi|66823573	28.5	1.26e-04	6
gi|15924499	26.9	3.67e-04	9
gi|22537145	26.6	4.79e-04	10
gi|30265243	25.8	8.17e-04	13

Table 5.2: Results of Using gi|49187252 as a Query in Our Framework against NR.

Sequence Name	Score (Bits)	E-Value	SW Rank
gi|49187252	299	6.53e-86	1
gi|15925512	75.1	1.17e-18	2
gi|15924184	59.7	5.1e-14	3
gi|19746167	28.9	9.65e-05	5
gi|66823573	28.5	1.26e-04	6
gi|15924499	26.9	3.67e-04	9
gi|22537145	26.6	4.79e-04	10
gi|66819553	25.8	8.17e-04	11
gi|85083718	25.8	8.17e-04	14

In addition to the SW Rank, we are also interested in the E-value of each alignment. The E-Value column indicates the number of times that one would expect to

see a sequence with such a score by chance, so smaller values are better. We can use E-values to gauge whether an alignment is significant, irrespective of its relative SW Rank.

As expected, since the query was taken straight from the database without modification, it matches itself with a very high score, a very low E-Value and is the highest ranked match when performing Smith-Waterman directly on the results.

There are several observations we can take away by comparing the results of NCBI BLAST and our framework. First, both agree in the first three alignments, indicating that for this query, there is no difference between BLAST implementations for high scoring alignments.

Past this, it is clear that there are discrepancies between the two. NCBI BLAST manages to align against the 4th highest ranked alignment, where our framework does not. Similarly, our framework finds an alignment the 11th and 14th highest ranking alignments where NCBI BLAST does not. It should be noted that while missing the 4th highest ranking alignment sounds significant, by this point the E-Values are ten orders of magnitude higher, and therefore, the alignments are correspondingly less significant. Both NCBI BLAST and our framework begin to experience large gaps in SW Rank between alignments at this point.

The major cause of differences in the results is that NCBI BLAST has an advantage over our framework when using higher thresholds for seed generation. This is because when processing a query, NCBI BLAST will use a 3-mer from the query even if it is below the threshold score. Our framework is unable to do this because it must index the database without any queries available and so, to achieve completely equivalent results it would be forced to use a much lower threshold, with a corresponding increase in

the size of the HSP Index File. This is the reason that our framework missed the alignment for gi|22537123.

5.2 DATA TRANSFER RATIO

As described in Chapter 4, we assume that, independent of the storage medium, performance is I/O bound. Calculations are performed by the FPGA quickly enough that the majority of time spent will be waiting on data from the database or HSP Index Table, whether they are stored in RAM or on disk. We then are interested in the amount of data transferred by by our framework and how it compares to state of the art FPGA implementations, but not in specifics on the hardware model such as random access latency or throughput.

To gather these numbers, we modified our framework to count the number of database HSP hits read from the HSP Index File, each of which is 8 bytes, as well as the total size of all the database records for which at least one HSP hit was matched to a query. If we assume the records are of average size for the database then we can calculate the number of bytes of the database that we must read.

In Figure 5.1 below, we show the bytes read from the HSP Index Table when searching for batches of 20, 40, 60, 80 and 100 queries randomly selected from the NR database using the first 200K records of the database.

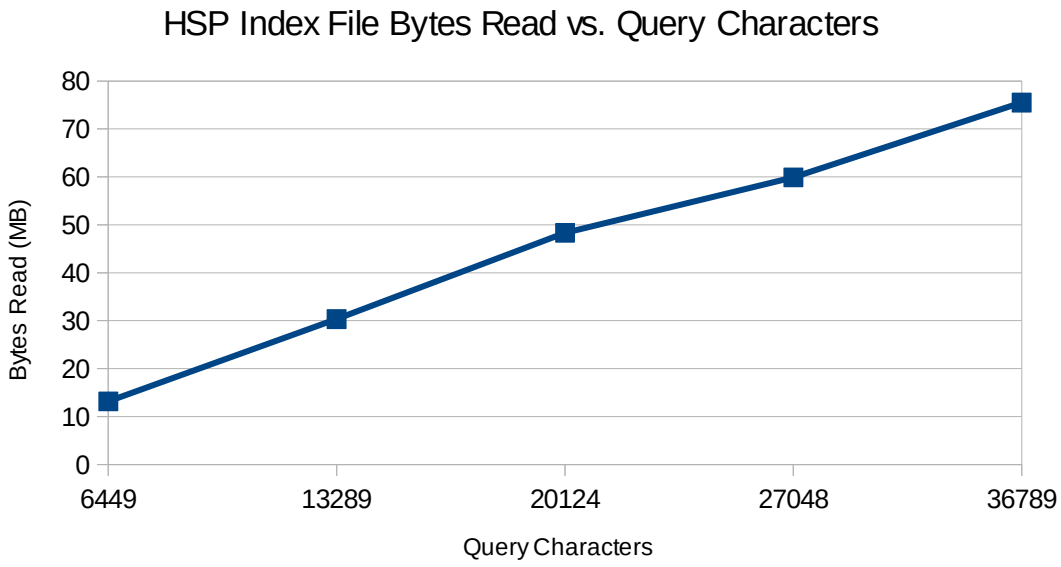


Figure 5.1: HSP Index Table bytes read vs. Query Characters for query batches against the first 200K records of NR.xz

From the figure we can see that the number of bytes read is linear on the number of query characters being processed. To place these numbers into perspective, the database used contained the first 200K records of NR and was approximately 800MB in size. By comparison, 100 queries totaling 36,789 characters required less than 80MB of data to be transferred, indicating that the overhead for the HSP Index Table is small compared to rereading the database.

To also account for the database transfer required, we present the number of records for which there was at least one match below in Figure 5.2:

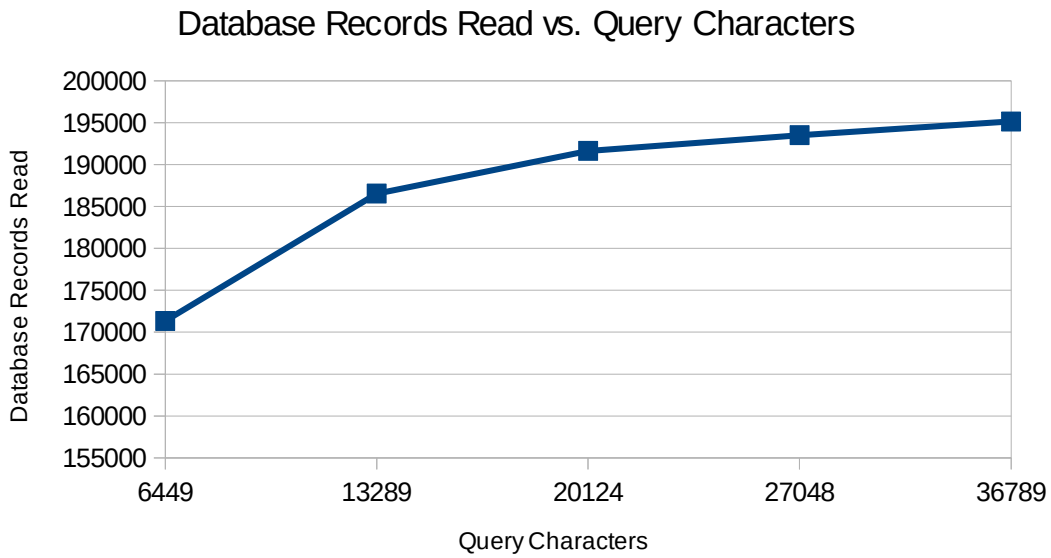


Figure 5.2: Database records read vs. Query Characters for query batches against the first 200K records of NR.

Figure 5.2 shows that the number of database records read rapidly approaches the limit of the full size of the database. For the sample database, searching for 20 queries used approximately 170K out of 200K records, or 85%, but this reaches 195K out of 200K, or 97.5% by the time we search for 100 queries. In practice, this would mean that for large numbers of queries we expect to have to read most of the database.

5.3 TWO-HIT FILTER SPEEDUP

State of the art implementations of BLAST on FPGAs, such as CAAD BLAST [31], must reread the entire database for each query so that the two-hit filtering stage can be reconfigured for the query. Our approach eliminates this requirement and only needs to read the database in its entirety at most once for a batch of queries, but still has the overhead of reading the HSP Index Table for any matches found in the queries.

We expect speedup in the two-hit filter if the overhead for reading the HSP Index Table is less than the need to reread the database per query. We define speedup as:

$$Speedup = \frac{BytesRead_{Database} + BytesReads_{HSP\ Index\ File}}{BytesTotal_{Database} \times Queries}$$

Our projections of speedup for the two-hit filter using the same selection of queries and database are shown below in Figure 5.3:

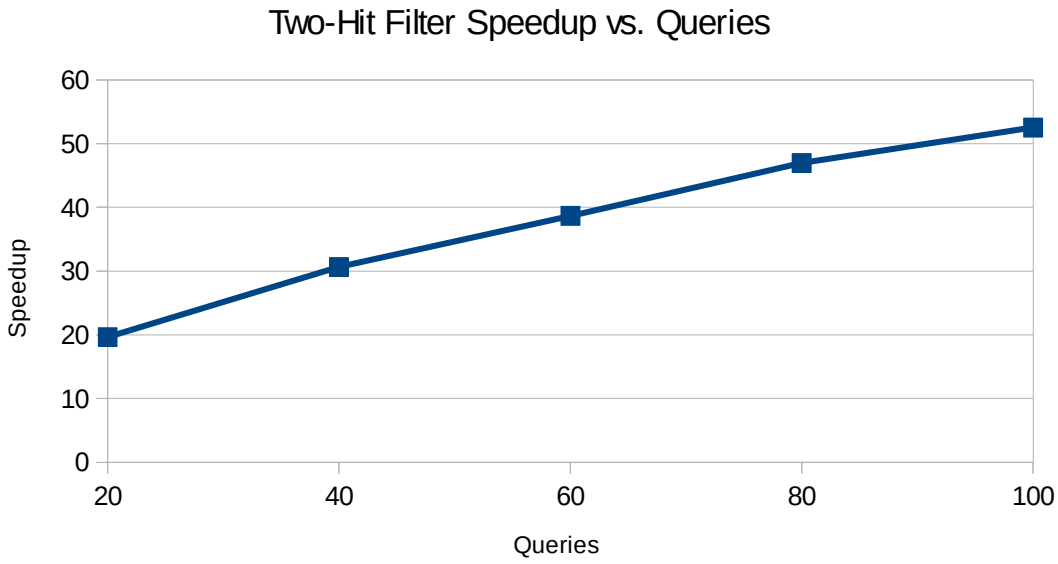


Figure 5.3: Two-Hit Filter Speedup vs. Queries for query batches against the first 200K records of NR.

Given the total bytes read from the HSP Index Table shown in Figure 5.1, it is unsurprising that we are able to realize substantial speedup with increasing numbers of queries. For these queries, each query, on average, requires only approximately 1% of the database's size in bytes to be read from the HSP Index Table, as opposed to 100% for CAAD BLAST.

The two-hit filter speedup does not vary significantly with the size of the database. The total bytes read from the HSP Index Table is directly proportional to the

size of the database used to generate it, causing the ratio of total bytes read vs. rereading the database per query to remain nearly constant.

5.4 TWO-HIT FILTER TIME RATIO

While we are able to achieve substantial speedups in the two-hit filter stage of BLAST, the two-hit filter is only part of the total runtime. Extension and alignment come afterward and are not accelerated by our framework. As such, our overall speedup is limited by the time spent in the two-hit filter.

To estimate the time spent in the two-hit filter, we modified NCBI BLAST and inserted timing code around its word finding and HSP generation routines. We then ran it on a Dell PowerEdge R710 server using a single thread of a Xeon E5520 CPU and the same queries we examined in our framework. We gathered the time spent in the two-hit filter stage and compared it to the overall time spent executing NCBI BLAST to determine the ratio of execution time that we could accelerate.

The relative time spent in the two-hit filter is shown in Figures 5.4 and 5.5.

From Figure 5.4 we can see that there is little change from simply using more or longer queries. When using the first 200K records of the NR database, NCBI BLAST ranges from spending 78% of its total runtime in the two-hit filter for a batch of 20 queries totaling 6,449 characters up to only 81% for using nearly 6 times the query characters. We can also see that from these batches of queries, the total relative time spent does not strictly increase with more queries.

Figure 5.5 however shows a clear increase in the relative time spent in the two-hit filter with increasingly larger databases. Using the same 20 queries totaling 6,449 characters, NCBI BLAST ranges from using 82% of its runtime at 200K records up to

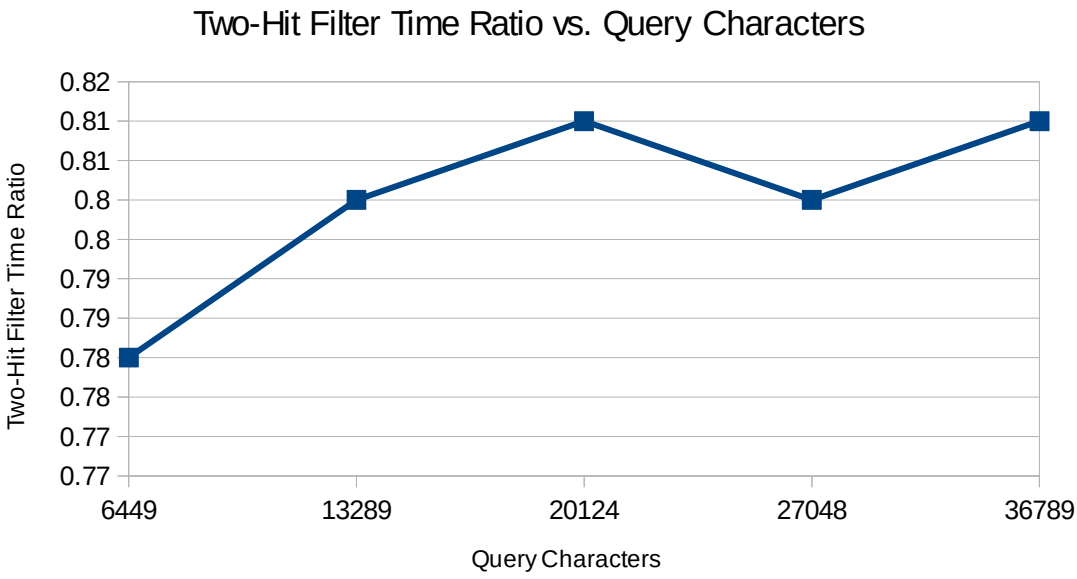


Figure 5.4: Relative Time Spent in Two-Hit Filter

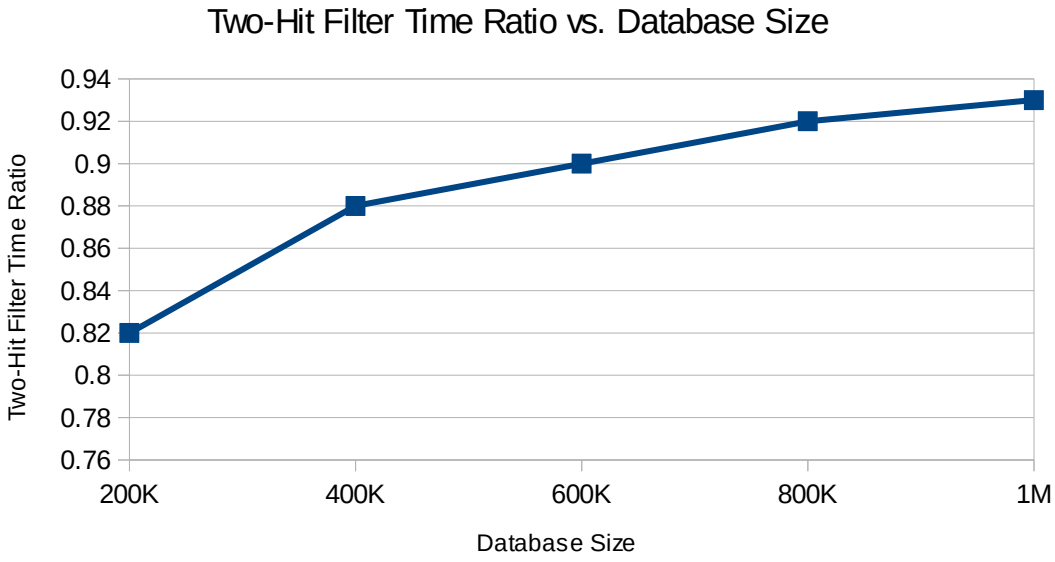


Figure 5.5: Two-Hit Filter Speedup vs. Database Size using 20 queries totaling 6,449 characters.

93% for 1 million records. Since our speedup is dependent on accelerating the two-hit filter, we thus conclude that using searching larger databases will see more benefit than smaller databases.

5.5 OVERALL SPEEDUP

In order to calculate the overall expected speedup of our approach, we must consider both the speedup of the two-hit filter and limit it by the relative time spent by BLAST in the two-hit filter. We define speedup as follows:

$$Speedup = \frac{T_{TOTAL}}{(T_{TOTAL} - T_{TWO-HIT}) + T_{TWO-HIT} / Speedup_{TWO-HIT}}$$

$Speedup_{TWO-HIT}$ was defined previously as the ratio of data transferred between our framework and CAAD BLAST. As a result, we expect overall speedup to increase with more queries, which would improve the two-hit filter speedup, and with larger databases, which would increase the relative time spent in the two-hit filter. We show these effects below in Figures 5.6 and 5.7.

Using increasing numbers of queries shows a small improvement in overall speedup for the same size database, ranging from about 4x to 5x. In this case, overall speedup is limited by the relative time spent in the two-hit filter, which we can improve by increasing the size of the database being searched.

A clear improvement in overall speedup is shown by ranging from 200K to 1 million records using the same 20 queries. In this case we get from about 4x to nearly 8x overall projected speedup.

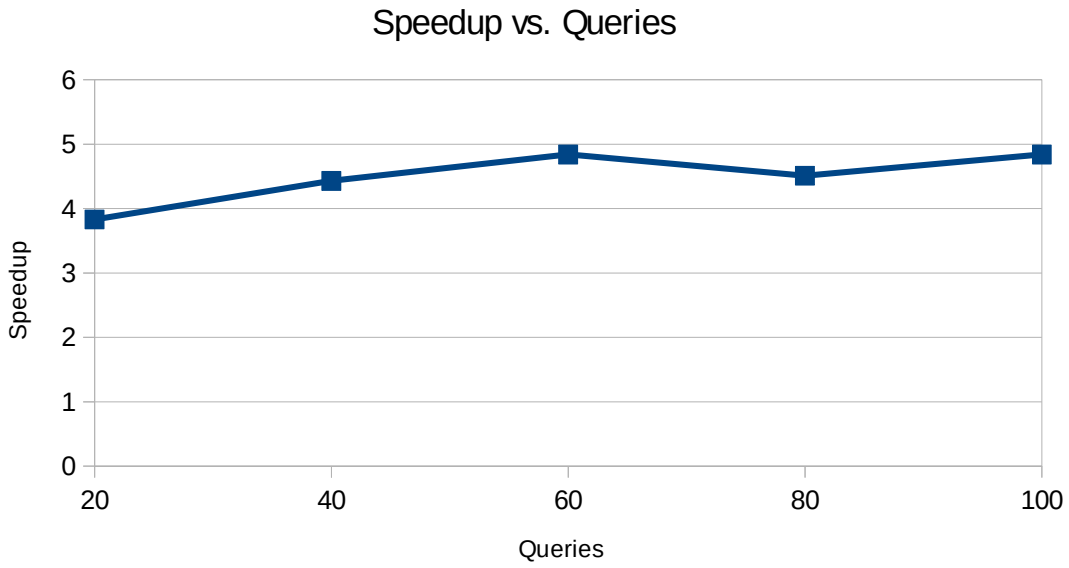


Figure 5.6: Overall Speedup vs. Queries using the first 200K records of the NR database.

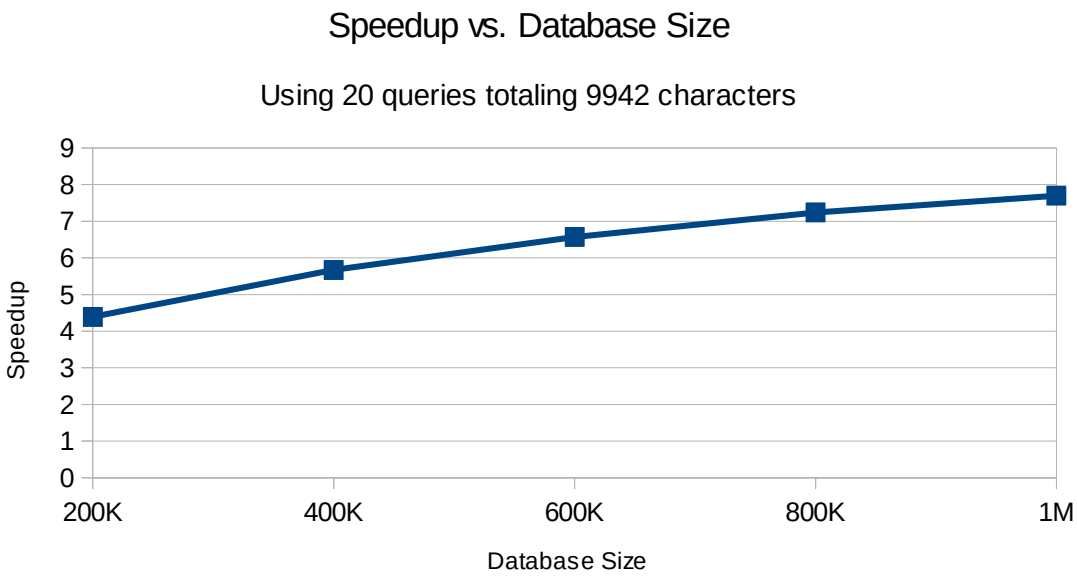


Figure 5.7: Overall Speedup vs. Database Size using 20 random queries totaling 9,942 characters.

5.6 HSP INDEX TABLE SIZE

In this section we provide results showing the size of the HSP Index File as a function of the input database size. As mentioned in Chapter 4, this file is larger than the input database because it is an exhaustive index of all HSPs contained in the database, for which there is considerable overlap. The results in Table 5.3 were generated using a threshold value of 11 for 2-character seeds.

Table 5.3: HSP Index File Size as a Function of Database Size

Entries Processed	NR		Uniref50	
	Size of Raw Database	Size of HSP Index File	Size of Raw Database	Size of HSP Index File
200K	81.3 MB	0.8 GB	93.3 MB	0.92 GB
400K	153 MB	1.4 GB	166 MB	1.5 GB
600K	221 MB	2.0 GB	265 MB	2.4 GB
800K	291 MB	2.8 GB	365 MB	3.5 GB
1M	361 MB	3.5 GB	467 MB	4.5 GB
Full (est.)	3.7 GB	38.5 GB	6.7 GB	72 GB

As the results show, the HSP Index File is substantially larger than the input database, from 10x-11x the size. Despite its large size, the file is sorted and organized such that only a single disk seek is required before all hits to a relevant HSP regular expression can be read, emphasizing disk bandwidth over seek times.

The size is also a function of the seed score threshold, and using higher thresholds would decrease the size of the HSP Index File while sacrificing sensitivity. While we have not performed rigorous analysis on the effects of using thresholds other than 11 on search results, below we present data showing the effects on the HSP Index File's size when using different threshold values.

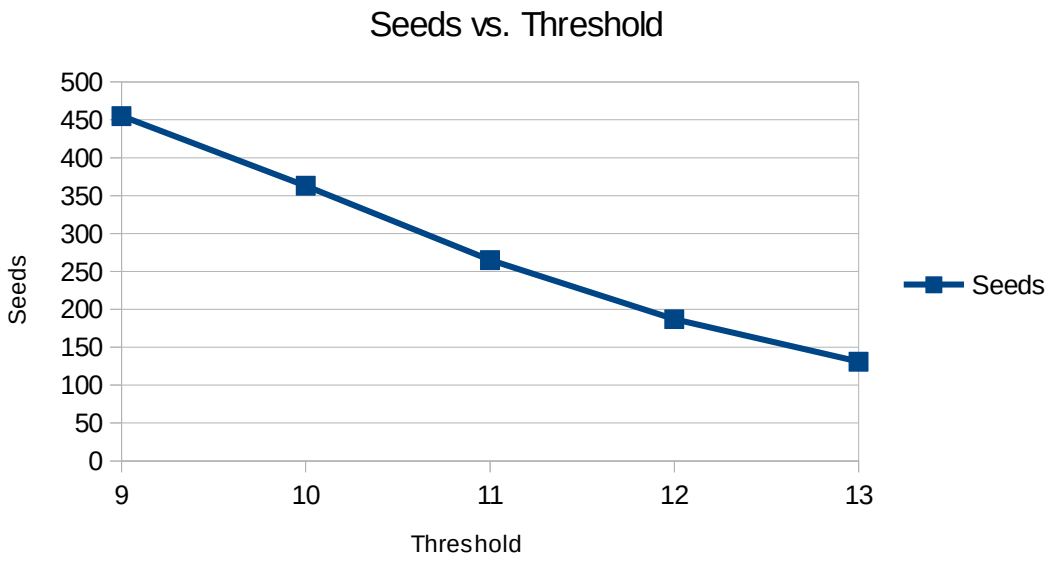


Figure 5.8: Seed Growth as a Function of Threshold

As Figure 5.8 shows, the number of seeds decreases with increasing threshold, indicating fewer seeds whose self scores meet the minimum threshold value. For these thresholds the relationship is roughly linear.

In Figure 5.9, the number of candidate HSPs that these seeds translate to is shown. Since the number of candidate HSPs is the square of the number of seeds, we see a quadratic falloff with increasing threshold and fewer seeds.

While we chose a threshold value of 11 based on preliminary results from Altschul et. al [8], Figure 5.9 shows that a threshold value of 11 also represents a point where increasing the threshold produces diminishing returns on reducing the number of HSPs.

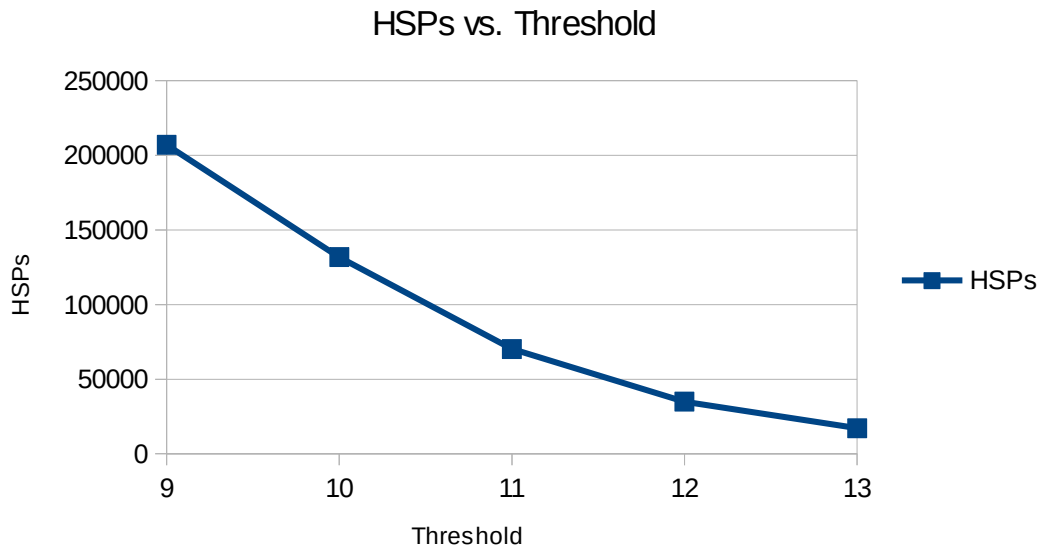


Figure 5.9: Total HSP Growth as a Function of Threshold

We would expect that the size of the HSP Index File would correlate strongly to the number of HSPs, which is borne out and shown below in Figure 5.10.

Just like the number of HSPs, the size of the HSP Hit Index decreases quadratically with increasing threshold. Again we can see that a threshold of 11 gives a natural point at which increasing the threshold further produces diminishing returns on reducing the file size.

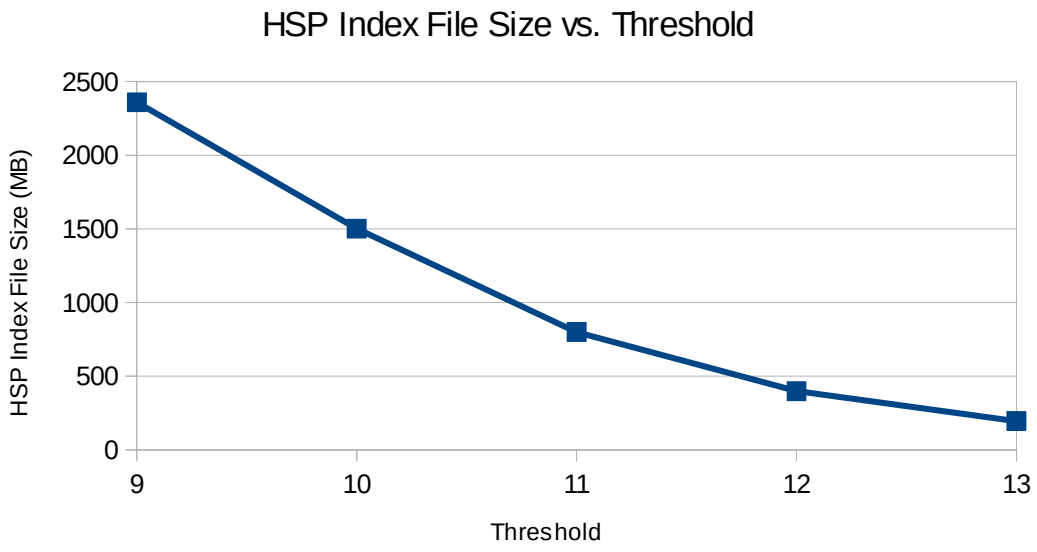


Figure 5.10: HSP Index File Growth as a Function of Threshold, using the first 200K records of the NR database.

CHAPTER 6

CONCLUSION

In this dissertation we have presented a novel alternative architecture for the two-hit filter stage of BLAST that utilizes regular expressions implemented as dedicated processing elements in an FPGA to efficiently search a genomic database. Our design implements the seed finding and HSP creation stages in a single query HSP detection step and uses a preprocessed table of HSPs from a subject database to produce a set of final HSPs that can be extended and aligned on a host processor.

We have shown that our framework presents the possibility for significant speedups over state of the art BLAST implementations on FPGAs, such as CAAD BLAST, through a substantial reduction in data I/O allowed by reading only a small fraction of the database per query processed. Our results indicate that we can attain increasing speedups for the two-hit filter by using increasing numbers of queries, while searching increasingly large databases leads to greater overall speedups.

We have also shown that while the preprocessing has limitations that prevent it from producing identical results to NCBI BLAST at comparable seed thresholds, the results are accurate for high scoring matches and only very poor matches differ between our framework and NCBI BLAST.

6.1 FUTURE RESEARCH DIRECTIONS

Future work could involve improving the FPGA implementation or improving the offline database indexing operation.

6.1.1 FPGA ARCHITECTURE

The FPGA implementation presented in this dissertation implements only the two-hit filter components of BLAST in hardware, which is a limitation on our maximum speedups. By moving the extension and alignment stages onto the FPGA, such as in CAAD BLAST, we could accelerate more of the algorithm and reduce some communication between the host and coprocessor.

6.1.2 DATABASE INDEXING

The need to generate the HSP Index File reduces the flexibility of our framework, since it fixes the threshold parameter at the time of generating the index. Furthermore, generating the index is time intensive and uses a considerable amount of disk space.

Future research could focus on ways to improve the speed of generating this index as well as its final size. There is substantial redundancy in the index caused by many HSPs being generated for the same database sequence characters, which mirrors the behavior of NCBI BLAST but is very inefficient in space. It is possible that some of these HSP candidates may be unnecessary, or there may be better ways to store them that require less space.

BIBLIOGRAPHY

1. Abbott A., Tsay A. (2000). "Sequence Analysis and Optimal Matching Methods in Sociology, Review and Prospect". *Sociological Methods and Research* 29 (1): 3–33.

2. Needleman, Saul B., and Christian D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins." *Journal of molecular biology* 48.3 (1970): 443-453.

3. Smith, Temple F., and Michael S. Waterman. "Identification of common molecular subsequences." *Journal of molecular biology* 147.1 (1981): 195-197.

4. Pearson, William R., and David J. Lipman. "Improved tools for biological sequence comparison." Proceedings of the National Academy of Sciences 85.8 (1988): 2444-2448

5. Lipman, David J., and William R. Pearson. "Rapid and sensitive protein similarity searches." *Science* 227.4693 (1985): 1435-1441.

6. Wilbur, W. John, and David J. Lipman. "Rapid similarity searches of nucleic acid and protein data banks." *Proceedings of the National Academy of Sciences*80.3 (1983): 726-730.

7. Maizel Jr, Jacob V., and Robert P. Lenk. "Enhanced graphic matrix analysis of nucleic acid and protein sequences."*Proceedings of the National Academy of Sciences of the United States of America* 78.12 (1981): 7665.

8. Altschul, Stephen F., et al. "Basic local alignment search tool." Journal of molecular biology 215.3 (1990): 403-410

9. Karlin, Samuel, and Stephen F. Altschul. "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes." *Proceedings of the National Academy of Sciences* 87.6 (1990): 2264-2268.

10. Becchi, Michela. "Data Structures, Algorithms and Architectures for Efficient Regular Expression Evaluation." Diss. Washington U in St. Louis, 2009. 01 May 2009. Web. 1 Apr. 2015.

11. Fidanci, Osman Devrim, et al. "Performance and overhead in a hybrid reconfigurable computer." *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003

12. Harkins, John, et al. "Performance of sorting algorithms on the SRC 6 reconfigurable computer." *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 2005

13. Floyd, Robert W., and Jeffrey D. Ullman. "The compilation of regular expressions into integrated circuits." Journal of the ACM (JACM) 29.3 (1982): 603-622.

14. Sidhu, Reetinder, and Viktor K. Prasanna. "Fast regular expression matching using FPGAs." Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on. IEEE, 2001.

15. "CUDA Toolkit Documentation, v 6.5." http://docs.nvidia.com/cuda/cuda-c-programming-guide/. 2015.

16. "OpenCL – The open standard for parallel programming of heterogeneous systems." https://www.khronos.org/opencl/. 2015.

17. Rabin, M. O.; Scott, D. (1959). "Finite automata and their decision problems". *IBM Journal of Research and Development* 3 (2): 114–125

18. Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. "Introduction to automata theory, languages, and computation." *ACM SIGACT News* 32.1 (2001): 60-65.

19. Sheng Yu (1997). "Regular languages". In Grzegorz Rozenberg and Arto Salomaa. *Handbook of Formal Languages: Volume 1. Word, Language, Grammar*

20. *ISO/IEC 9945-2:1993 Information technology – Portable Operating System Interface (POSIX) – Part 2: Shell and Utilities, successively revised as ISO/IEC 9945-2:2002 Information technology – Portable Operating System Interface (POSIX) – Part 2: System Interfaces, ISO/IEC 9945-2:2003, and currently ISO/IEC/IEEE 9945:2009 Information technology – Portable Operating System Interface (POSIX®) Base Specifications, Issue 7*

21. *Ken Thompson (Jun 1968). "Programming Techniques: Regular expression search algorithm". Communications of the ACM 11 (6): 419–422.*

22. Rabin, M. O.; Scott, D. (1959). "Finite automata and their decision problems". *IBM Journal of Research and Development* 3 (2): 114–125

23. "BLAST: Local Alignment Search Tool" http://blast.ncbi.nlm.nih.gov/Blast.cgi. 2015.

24. McGinnis, S. and Madden, T. (2004). Blast: at the core of a powerful and diverse set of sequence analysis tools. Nucleic Acids Research, 32:W20–W25.

25. Cameron, Michael, Hugh E. Williams, and Adam Cannane. "A deterministic finite automaton for faster protein hit detection in BLAST." Journal of Computational Biology 13.4 (2006): 965-978.

26. Ma, Bin, John Tromp, and Ming Li. "PatternHunter: faster and more sensitive homology search." Bioinformatics 18.3 (2002): 440-445.

27. Zhang, Zheng, et al. "A greedy algorithm for aligning DNA sequences." Journal of Computational biology 7.1-2 (2000): 203-214.

28. Kent, W. James. "BLAT—the BLAST-like alignment tool." Genome research 12.4 (2002): 656-664.

29. Mahram, Atabak, and Martin C. Herbordt. "NCBI BLASTP on High-Performance Reconfigurable Computing Systems." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.4 (2015): 6.

30. Shiryev, Sergey A., et al. "Improved BLAST searches using longer words for protein seeding." *Bioinformatics* 23.21 (2007): 2949-2951.

31. Muriki, Krishna, Keith D. Underwood, and Ron Sass. "RC-BLAST: Towards a portable, cost-effective open source hardware implementation." Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International. IEEE, 2005.

32. Herbordt, Martin C., et al. "Single pass, BLAST-like, approximate string matching on FPGAs." Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on. IEEE, 2006.

33. Kung, Hsiang Tsung, and Charles E. Leiserson. "Algorithms for VLSI processor arrays." Introduction to VLSI systems (1980): 271-292.

34. Chang, Chen. "BLAST implementation on BEE2." *Electrical Engineering and Computer Science, Univ. of Cal Berkeley* (2004).

35. Sotiriades, Euripides, Christos Kozanitis, and Apostolos Dollas. "FPGA based architecture for DNA sequence comparison and database search." Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International. IEEE, 2006.

36. Guo, Xinyu, Hong Wang, and Vijay Devabhaktuni. "A Systolic Array-Based FPGA Parallel Architecture for the BLAST Algorithm." International Scholarly Research Notices 2012 (2012).

37. Liu, Weiguo, et al. "Bio-sequence database scanning on a GPU." *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006

38. Manavski, Svetlin A., and Giorgio Valle. "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment." *BMC bioinformatics* 9.Suppl 2 (2008): S10.

39. Ligowski, Lukasz, and Witold Rudnicki. "An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases." *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009.

40. Ling, Cheng, and Khaled Benkrid. "Design and implementation of a CUDA-compatible GPU-based core for gapped BLAST algorithm." *Procedia Computer Science* 1.1 (2010): 495-504.

41. Vouzis, Panagiotis D., and Nikolaos V. Sahinidis. "GPU-BLAST: using graphics processors to accelerate protein sequence alignment." *Bioinformatics*27.2 (2011): 182-188.

42. Krishnamurthy, Praveen, et al. "Biosequence similarity search on the Mercury system."The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology 49.1 (2007): 101-121.

43. Chamberlain, Roger D., et al. "The Mercury system: Exploiting truly fast hardware for data search." *Proc. of Int'l Workshop on Storage Network Architecture and Parallel I/Os*. 2003.

44. Bloom, Burton H. "Space/time trade-offs in hash coding with allowable errors."*Communications of the ACM* 13.7 (1970): 422-426.

45. Chamberlain, Roger D., et al. "The Mercury system: Exploiting truly fast hardware for data search." *Proc. of Int'l Workshop on Storage Network Architecture and Parallel I/Os*. 2003.

46. Floyd, Robert W., and Jeffrey D. Ullman. "The compilation of regular expressions into integrated circuits." Journal of the ACM (JACM) 29.3 (1982): 603-622.

47. Sidhu, Reetinder, and Viktor K. Prasanna. "Fast regular expression matching using FPGAs." Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on. IEEE, 2001.

48. Sidhu, Reetinder, and Viktor K. Prasanna. "Fast regular expression matching using FPGAs." Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on. IEEE, 2001.

49. Smith, Randy, et al. "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata." ACM SIGCOMM Computer Communication Review. Vol. 38. No. 4. ACM, 2008.

50. Becchi, Michela, and Patrick Crowley. "Efficient regular expression evaluation: theory to practice." *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2008.

51. Korenek, Jan, and Vlastimil Kosar. "Efficient mapping of nondeterministic automata to FPGA for fast regular expression matching." *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*. IEEE, 2010.

52. NR Database, available from http://nih.gov.

53. Uniprot Protein Databases, available from http://www.uniprot.org/downloads.