Building BLAST for Coprocessor Accelerators Using Macah

by

Ben Weintraub

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering
University of Washington

June 2008

Presentation of work given on:

Thesis and presentation approved by:

Date:

Abstract

The problem of detecting similarities between different genetic sequences is fundamental to many research pursuits in biology and genetics. BLAST (Basic Local Alignment and Search Tool) is the most commonly used tool for identifying and assessing the significance of such similarities. With the quantity of available genetic sequence data rapidly increasing, improving the performance of the BLAST algorithm is a problem of great interest. BLAST compares a single query sequence against a database of known sequences, employing a heuristic algorithm that consists of three stages arranged in a pipeline, such that the output of one stage feeds into the input of the next stage. Several recent studies have successfully investigated the use of Field-Programmable Gate Arrays (FPGAs) to accelerate the execution of the BLAST algorithm, focusing on the first and second stages, which account for the vast majority of the algorithms execution time. While these results are encouraging, translating algorithms like BLAST that contain somewhat complex and unpredictable control flow and data access patterns into versions suitable for implementation on coprocessor accelerators like FPGAs turns out to be quite difficult using currently available tools. Such architectures are usually programmed using Hardware Description Languages (HDLs), which are significantly more difficult to learn and use than standard programming languages. In this paper, an accelerated version of the BLAST algorithm is presented, written in a new language called Macah, which is designed to make the task of programming coprocessor accelerators easier for programmers familiar with the widely-known C language.

1 Introduction

BLAST (Basic Local Alignment Search Tool) is a fast heuristic algorithm used for approximate string matching against large genetic databases. BLAST takes as inputs a query string and a database string (usually much longer than the query), and returns a list of approximate matches between the two, each with an associated alignment and numeric score. The algorithm was first described in [1], and since then has become one of the most widely used software tools by researchers in genetics and bioinformatics – the original BLAST paper was the most widely cited paper of the 1990's, with over 10,000 citations. BLAST is used for a variety of purposes, including matching newly-isolated genes against databases of known genes in order to make predictions about gene function, and tracking evolutionary changes in genetic material over time. Figure 1 shows the format of an example result from a sample BLAST search.

```
Mus musculus chromosome 2 genomic contig, strain C57BL/6J
Length=36511446

 Features in this part of subject sequence:
   myosin IIIA


 Score =  333 bits (368),  Expect = 2e-88
 Identities = 454/636 (71%), Gaps = 12/636 (1%)
 Strand=Plus/Minus

Query  16     CAAGTAGGTCTACAAGACGCTACTTCCCCTATCATAGAAGAGCTTATCACCTTTCATGAT  75
              ||| | ||| ||||||||||| || |||||||| |||||||| || || |  || ||||||
Sbjct  46215  CAACTTGGTTTACAAGACGCCACATCCCCTAtTTATAGAAGAACTAATAAATTTCCATGAT  46156

Query  76     CACGCCCTCATAATCATTTTCCTTATCTGCTTCCTAGTCCTGTATGCCCTTTTCCTAACA  135
              ||| | || |||||  ||||||| ||  ||| | |||||||| |||  | | |  |||| |
Sbjct  46155  CACACACTAATAATTGTTTTCCTAATTAGCTCCTTAGTCCTCTATATCATCTCGCTAATA  46096
```

Figure 1: A (truncated) sample BLAST result, generated by searching for the human gene COX2 in the mouse genome (*Mus musculus*). The name of the matching sequence from the database is shown (top), along with notable biological features within the matched portion of the database string, some scoring information, and the alignment itself between the matched portions of the query and database strings.

Prior to the development of BLAST, the Smith-Waterman string alignment algorithm was the primary tool used for doing approximate string matching against genetic databases. Smith-Waterman is a traditional dynamic-programming algorithm, and is guaranteed to find the best local alignment of the two input strings, because it calculates a score for each such possible alignment and choosing the best one. The running time of the Smith-Waterman algorithm is $O(mn)$, where m is the length of the query string and n is the length of the database string, as it is with BLAST, but BLAST works by quickly and probabilistically eliminating the vast majority of potential alignments which do not appear promising. While the size of query strings has remained relatively constant and

small (on the order of thousands or tens of thousands of characters), the size of genetic databases is much larger (typically several gigabytes), and has grown exponentially, doubling approximately every 18 months and spurring the development of heuristics such as BLAST that sacrifice some search sensitivity and accuracy for greatly reduced running time.

Since the initial development of BLAST in 1990, the size of genetic databases has continued to grow at an exponential rate, meaning that improving BLAST performance has remained an important goal. Algorithmic modifications such as MegaBLAST [15] that make further speed/sensitivity tradefoffs have been proposed and are used in certain situations where performance is critical and sensitivity of the search is secondary, however accelerated versions of BLAST that do not sacrifice sensitivity compared to the original BLAST algorithm are more desirable.

1.1 Previous Hardware Acceleration Work

Much approximate string matching acceleration work has focused on the simpler Smith-Waterman algorithm, which is more easily accelerated using specialized hardware. Numerous FPGA and GPU-based accelerated Smith-Waterman implementations have been created, many yielding impressive speedups over the standard software implementation. However, even with good hardware-acceleration, Smith-Waterman still has trouble competing performace-wise with unaccelerated software BLAST.

Several multi-threaded BLAST implementations leveraging multiprocessor systems and scientific computing clusters are available, the most prominent being mpiBLAST [5] and mpiBLAST-PIO [13] (targeted specifically at the IBM's Blue Gene/L). These accelerated BLAST implementations have shown impressive speedups, but such approaches will not be the focus of this paper.

Coprocessor accelerator architectures offer greatly reduced power consumption and higher spatial density as compared to traditional multi-processor or cluster systems, making them attractive for accelerated BLAST implementations . In recent years, several FPGA-accelerated versions of BLAST have been produced, both commercially and in the research realm. Commercial implementations include TimeLogic's Tera-BLAST and the Mitrionics' Open Bio Project [9] (the source code of which has been released under an open-source license). FPGA BLAST implementations in the research realm include, RC-BLAST [10], Tree-BLAST [7], TUC BLAST [12], and Mercury BLAST [8] [3].

Of these various implementations, Mercury BLAST is perhaps most interesting for several reasons. In addition to the fact that it the authors report a significant speedup over NCBI BLAST, the *functional* deviations of Mercury BLAST from the original BLAST algorithm are minimal, meaning it produces results that are very similar to if not exactly the same as those of NCBI BLAST. Also, it is the basis for the Mitrion-C Open Bio Project's BLAST implementation, which appears to have gained at least some real-world adoption. For this reason, the acceleration approach presented in this paper will follow the general structure of the Mercury BLAST system, albeit with some modifications which

will be discussed in more detail in later sections.

2 BLAST Overview

The National Center for Biotechnology Information (NCBI) maintains the most widely-used BLAST software implementation (hereafter referred to as 'NCBI BLAST'). NCBI BLAST includes several variants for dealing with searches over nucleotide sequences, protein sequences, and translated versions of each of these. All utilize the same basic algorithm, but contain minor variations. The nucleotide variant of BLAST, called `blastn`, will be the focus of this paper, though many of the ideas presented should be applicable to the other variants as well. The BLAST algorithm utilizes a three-stage filtering pipeline to search for approximate matches between the query and database strings, with each stage acting as a filter, eliminating a significant proportion of its input and passing on promising candidate matches to the next stage.

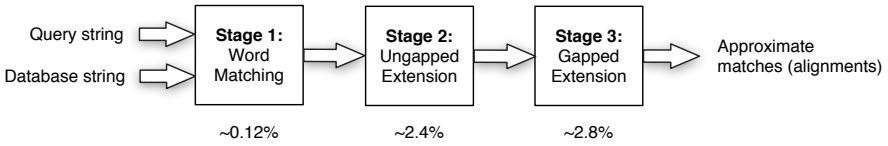

Figure 2: The 3-stage pipeline structure of the BLAST algorithm. Each stage acts as a filter, selecting a small proportion of its input to be passed on to the next stage for further investigation. The approximate percentages of the inputs that are 'passed' by each stage are shown below each stage.

Measurements for the pass rates of each stage shown in figure 2 were taken on the same system and over the same set of query and database strings used for performance profiling in section 2.2. These pass rates will be discussed in more detail after an explanation of each of the stages of the BLAST algorithm.

It is also important to note that the database string is pre-processed by the NCBI `formatdb` tool in order to compress it so that each nucleotide is represented using only 2 bits (for the possible values of A, C, T, and G) rather than a full byte.

Stage 1 consists of *word matching*, that is, finding short, fixed-length, exact matches (words) that are common to both the query sequence and the database sequence. The length of these word matches varies between the different BLAST variants, and is also configurable, but in nucleotide BLAST, the default length of such words is 11 nucleotides. Using a longer words will result in fewer matches, and thus will decrease the running time of the search, but will also decrease the search's sensitivity, while using shorter words will increase the number of word matches, and thus the running time while also increasing the sensitivity of the search.

Figure 3: A diagram showing the decomposition of input strings into words used in stage 1 of the BLAST algorithm

For each word in the database string, two questions are relevant: does the word *exist* in the query string, and if so, at which *location(s)* in the query string can it be found? Each nucleotide is represented using 2 bits, meaning that each word can be represented using 22 bits when the default word length of 11 nucleotides is used. Because most modern CPUs can operate only on bytes at a time rather than individual bits, NCBI BLAST splits stage 1 of the BLAST algorithm into two sub-steps. The database is first scanned along byte-boundaries, using a word size of 8 nucleotides (2 bytes), then each of these hits is extended by a small number of nucleotides on either side to bring it up to the size of a full word.

In order to facilitate this process, NCBI BLAST builds two data structures: a presence vector and a lookup table. The presence vector is simply a bit vector, consisting of $2^{16} (= 65536)$ bits, and can quickly answer the question of whether a given word is present in the query string. The bit positions corresponding to words that are present within the query string are turned on, and the rest are left as zeros. If a given word from the database is present within the query string, the lookup table is then used to extract its location(s) from the query string. Finally, the short exact matches are extended by several nucleotides on either side to bring them up to the full word length required.

Each word match consists of a pair of indices, one pointing to the beginning of the word in the query string, and the other pointing to the start index of the word within the database string. In stage 2 of the algorithm, each word match is extended in either direction, aligning corresponding character pairs from the query and database strings before and after the initial word match, and assigning each character pairing a score – positive if the paired characters are identical and negative otherwise. The score of the extension is calculated as the sum of all individual character pair scores. This *ungapped extension* is continued until the current score drops a certain threshold below the highest score seen so far, and only extensions that achieve scores above a certain threshold are passed on to stage 3 of the computation.

In stage 3, *gapped extension*, each successfully extended word match is extended in either direction using the standard Smith-Waterman alignment algorithm, which allows for gaps in the alignment between the query and database strings. Matches that yield a final score in this stage above a certain threshold

are then finally reported to the user.

Recall now the pass rates for each stage given in figure 2. It should be noted that these pass rates (especially for stages 1 and 2) can vary significantly based on the size of the query string. In particular, as the length of the query string increases, the probability of finding within it a particular word increases until it approaches one (when the number of words in the query string approaches the number of possible words of the specified length). This has a ripple effect into stage 2, forcing it to discard a higher proportion of initial word matches because they do not correspond to biologically significant.

2.1 Common Usage Characteristics

NCBI also maintains a publicly-accessible web interface to BLAST running on its own servers, and tracks statistics about the number and type of BLAST searches performed by site visitors. Though they do not give a complete picture of BLAST usage, these usage statistics are helpful in determining the most common parameters used in BLAST searches. According to [4], as of 2005, NCBI processes over 150,000 BLAST searches per day. More than 80% of these searches are against the 'nr' or 'nt' databases (for proteins and nucleotides, respectively). These databases, which contain a compilation of non-redundant genetic data pulled from a variety sources, are updated frequently and are quite large – currently abut 1.8 and 6.3 gigabytes, respectively. Additionally, more than 90% of the nucleotide searches are for query strings of less than 2000 base-pairs.

In summary, the most common usage of BLAST seems to be searching large databases with relatively short query strings. NCBI has released a set of databases and queries [4] specifically designed to mimic the characteristics of the most common BLAST searches, to be used for benchmarking purposes without the practical issues involved in downloading and using the multi-gigabyte 'nt' and 'nr' databases. This information further emphasizes the importance of the size of the database string to the running time of BLAST.

2.2 BLAST Profiling

In [8], the authors profile BLAST running with three different query string lengths (10K, 100K, and 1M bases), finding that stage 1 of the BLAST computation takes up an average of about 85% of the total pipeline processing time, while stage 2 takes up an average of about 15% of the running time, and the amount of time spent in stage 3 is negligible. The authors of [10], doing profiling on a single query on a small database with an older version of NCBI BLAST, also concluded that stage 1 of the BLAST algorithm accounts for about 80% of the program running time.

While the profiling results in [8] are useful for understanding the performance of BLAST with larger queries, no benchmarks for query sizes under 10000 base-pairs are reported (recall that according to NCBI statistics, over 90% of nucleotide BLAST searches performed through their website are with

queries of 2000 base-pairs or less). The results reported in [10] seem to be in agreement with those in [8], however, they are by no means exhaustive, covering only a single query on a single database and using an old version of NCBI BLAST.

In order to confirm these profiling results, and also to get a better picture of BLAST performance characteristics on smaller queries, I used the `oprofile` tool to profile `blastn` running over the 102 sample queries that comprise the BLAST benchmarks for `blastn` released by NCBI [4]. All queries were run against the benchmark nucleotide database `benchmark.nt`, also included with the BLAST benchmarks download. Results for this profiling are given in table 1 below.

Stage 1	Stage 2	Stage 3	I/O and other
67.73%	9.77%	12.48%	10.02%

Table 1: Profiling data collected using `oprofile` on NCBI `blastall` while running 102 benchmark queries against the `benchmark.nt` database. Profiling was performed using the `blastall` executable built from the 3/17/08 release of the NCBI C Toolkit on a 2.2 GHz AMD Athlon64 processor with 1 gigabyte of RAM running Ubuntu Linux 8.04.

While these results show broad agreement with the results in [8] and [10], they do differ somewhat. It should be noted that the profiling results given in [8] include only 'pipeline time', which presumably means that time not spent in one of the three core stages (the 'I/O and other' category in table 1) is not included in the reported percentages. The most striking difference beyond this is the much greater significance of stage 3 to the total running time in these results. The reason for this is unknown, but two factors may have contributed to the increased proportion of time spent in stage 3. First, the queries tested in these results are shorter than those used in [8], with lengths ranging from about 100-10000 bp, and an average length of 2068 bp, and these shorter queries may be computationally less intensive during stages 1 and 2. Second, the proportion of matches making it to stage 3 may be significantly higher when using the benchmark data set because of a higher degree of similarity between the queries and the database string.

It is worth noting that any unusual performance characteristics specific to short queries may be avoided using a slight algorithmic modification known as 'query packing', wherein multiple short queries are combined into a single longer query and run through the standard BLAST algorithm, with results being sorted based on their query of origin after the fact. In any case, all sets of profiling results suggest that stage 1 (word matching) should be the first target for hardware acceleration, because of its dominance of the overall running time.

3 Coprocessor Accelerators

3.1 Abstract Architecture

The abstract architecture targeted by the BLAST implementation presented in this report comes from the work of Ylvisaker, Van Essen, and Ebeling in [14]. An understanding of this abstract architecture and the constraints it imposes is crucial to allowing the programmer to evaluate potential acceleration schemes and optimizations, and determine their suitability for coprocessor acceleration. The abstract coprocessor accelerator architecture consists of a standard *sequential processor* connected to an *accelerator engine* (see Figure 2). The accelerator engine may be either an FPGA or another type of hybrid reconfigurable processor containing many parallel processing elements. The accelerator engine has direct access to a small amount ot *workspace memory* (implemented as Block-RAM on an FPGA) and a number of simple processing elements coordinated by a small amount of control logic.

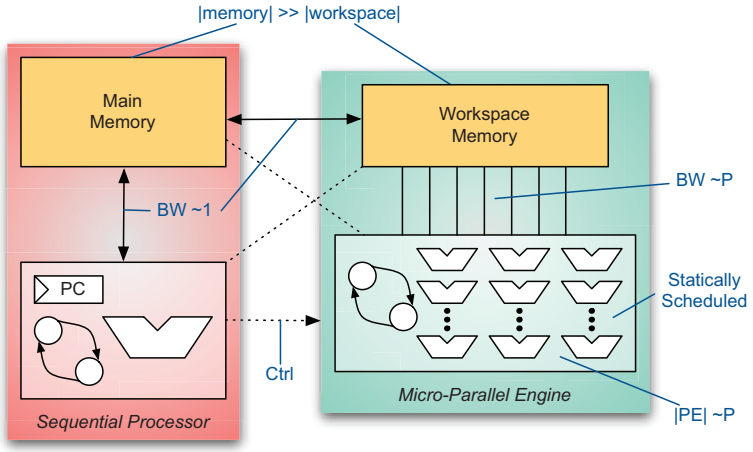

Figure 4: A block diagram of abstract coprocessor accelerator architecture, consisting of a sequential processor (left) connected to an accelerator engine consisting of a small amount of workspace memory and a set of processing elements. Originally presented in [14]

.

3.2 Constraints

There are several important constraints to consider when designing applications for coprocessor accelerator architectures. Firstly, it is important to recognize that the accelerator engine does not have direct access to the main memory of the sequential processor, meaning that before it can operate on a given piece of data, it must be sent across the I/O channel connecting the two processors.

The bandwidth of this channel is finite, and it can easily be the limiting factor in I/O-intensive computations. Because of this fact, it is desirable to make as much use of a given piece of data as possible once it has been sent across this channel, maximizing the ratio of computation to communication. Secondly, the amount of memory directly accessible by the accelerator engine (the workspace memory) is quite limited in size. While the exact amount varies across target hardware platforms, the size of the workspace memory will be on the order of that of an L2 cache on a modern microprocessor.

3.3 Macah

Macah is a new programming language, being developed at the University of Washington by the authors of [14], with the goal of providing a C-level language that can be used to write efficient programs taking advantage of the parallelism offered by coprocessor accelerator architectures. It is implemented as a set of extensions to the standard C language, and is designed to be easier to use than hardware description languages (HDLs) such as Verilog, which are the primary existing solutions for programming coprocessor accelerator architectures. Macah has also been designed with the goal of being able to target multiple specific hardware platforms using the same code base, provided that such systems conform to the abstract system architecture discussed at the beginning of this section. The Macah compiler provides the programmer with facilities that abstract many (but not all) of the details involved with scheduling concurrent operations on the accelerator engine and communicating between the sequential and parallel processors. Several of these are discussed in more detail below.

Kernel blocks allow the partitioning of application code into portions that will execute on the standard sequential processor and portions that will execute on the parallel processor. Each kernel block represents a portion of code that will execute on the parallel processor, while the rest of the code executes on the sequential processor. *Streams* provide an abstraction layer for the I/O channel connecting the sequential and parallel processors, allowing primitive types as well as more complex data structures to be serialized and sent back and forth between the two sides. *Reader and writer functions* handle incoming and outgoing data to and from the sequential processor across streams.

The Macah compiler can be configured to output either C code to be fed into a standard C compiler (for simulation and testing purposes), or a combination of C and Verilog (with C being used for the sequential parts of the computation, and Verilog being used to implement the kernel blocks). When compiling code inside a kernel block into Verilog, Macah will build a data-flow graph for the variables in the kernel block, and use this graph locate loops within the code that can be pipelined, or unrolled and executed entirely in parallel. An important optimization that is performed by the compiler in this process is *array scalarization*, in which arrays that are accessed inside of loops are transformed into multiple scalar variables so that they may be accessed in a parallel fashion. To assist the compiler, Macah provides the programmer with an uppercase *FOR* loop construct, which informs the compiler that a given loop should be unrolled

and executed in parallel.

4 Accelerated BLAST Structure

The accelerated BLAST implementation presented in this paper consists of a set of Macah code that interfaces with NCBI BLAST to replace NBCI BLAST's implementation of the word matching stage of the BLAST algorithm (see figure 4). The overall structure of the new stage 1 is similar to that of NCBI BLAST in that two different data structures constructed from the query string are used: a *Bloom filter* replaces the presence vector in NCBI BLAST and determines whether a given word is present within the query string, and a *cuckoo hash table* is used to extract the position of a given word from the query string, as well as to verify the results from the Bloom filter. Stages 2 and 3 (ungapped and gapped extension, respectively) remain unchanged in this implementation.

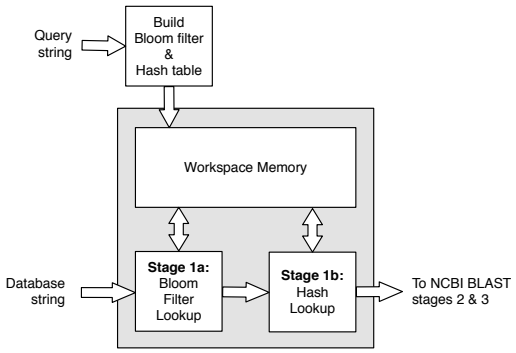

Figure 5: An overview of the accelerated BLAST structure. The Bloom filter and hash table are built from the query string on the sequential processor, and stored in the workspace memory of the accelerator engine, while the database string is streamed through, and word matches are streamed back to the sequential processor and sent on to stages 2 and 3 of the algorithm, which run unmodified.

The Bloom filter and hash table are built during program initialization, and are kept resident in the workspace memory of the accelerator engine throughout the execution of the program, while the database string is sent through using a Macah stream. Each successful word match is sent across another stream from the accelerator engine back to the sequential processor for further investigation by the later stages of the algorithm. This approach means that the space efficiency of the Bloom filter and hash table is of crucial importance, as both of these structures must be able to fit within the limited workspace memory available.

Currently, a single kernel block is used to implement the entire word matching stage, however a version that separates out the two sub-stages (using the

Bloom filter and the hash table) into separate kernel blocks connected by streams so that they may be pipelined and execute on different data concurrently is planned. The details of the Bloom filter and hash table are presented in the following sections.

5 Bloom Filter Implementation

The Bloom filter, first described by Burton Bloom in [2], is a space-efficient, probabilistic data structure used for quickly checking the membership of a given element within a set of other elements. Krishnamurthy et al. first proposed applying the Bloom filter as a pre-filter in front of stage 1 of BLAST in [8] in order to quickly eliminating the vast majority of words from the database string that do not have exact matches within the query string. This report will explain the usage and characteristics of a standard Bloom filter, and propose a small modification to the original structure that will make the structure better suited to implementation on coprocessor accelerator architectures.

A Bloom filter consists of a bit vector of m bits, along with k unique hash functions. Initially, all of the bits in the bit vector are set to 0. The Bloom filter is then built by encoding a set of of elements into the bit vector in the following manner: for each element, k different indices into the bit vector are generated using the hash functions $h_1 - h_k$ and the bit values at each of the corresponding positions within the bit vector are set to 1. Any collisions are simply ignored. To check for the presence of an element in the Bloom filter, k indices for the element are generated using the k hash functions, and each corresponding position within the bit vector is checked. If all k of the values are set to one, then the Bloom filter will return true for the given input, meaning there is a high probability that it exists within the set of encoded elements, otherwise it will return false, meaning the input element definitely is not a member of the set of encoded elements.

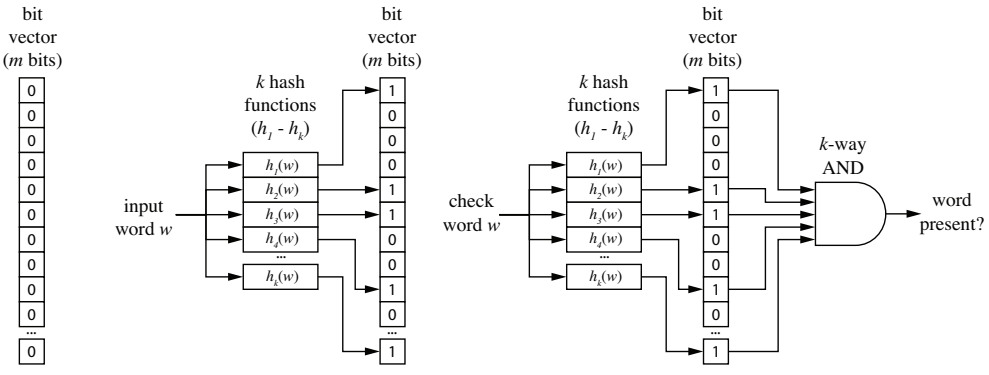

Figure 6: An illustration of Bloom filter operation: initialization (left), encoding a new word (center), and checking for the presence of a word (right)

Note that while false positives are possible with the Bloom filter, false negatives are not. False positives result from collisions, and so the probability of false positives depends on the probability of collisions within the bit vector, which in turn depends on the size of the bit vector, the number of items encoded, and the number of hash functions used.

In [2], Bloom calculates the false-positive rate P of a Bloom filter using k hash functions as:

$$P = (1 - \phi)^k$$

Where ϕ represents the expected proportion of bits in the bit vector that are still set to zero. He further defines ϕ in terms of m, the size of the bit vector, and n, the number of items that have been inserted into the filter so far:

$$\phi = (1 - k/m)^n$$

Thus giving a final error rate for the Bloom filter as:

$$P = (1 - (1 - k/m)^n)^k$$

A Bloom filter consisting of $2^18 = 262144$ bits (32 kilobytes), utilizing a $k = 6$ hash functions therefore can support queries of up to $2^15 = 32768$ words with a false positive rate of less than 2.2%:

$$P = (1 - (1 - 6/2^{18})^{2^{15}})^6 \approx 0.022$$

5.1 Bloom Filter Partitioning

One issue with implementing a Bloom filter in Macah is serialization of the k memory accesses to the bit vector for each lookup – though the k different hash functions can be computed in parallel, queries on the bit values at each corresponding position within the bit vector will be serialized if a single table is used.

To avoid this serialization, it is desirable to partition the bit vectorinto a set of k smaller bit vectors, each of size m/k. Each of the k hash functions is associated with a single bit vector, and will only generate indices into that array. The set of tables may be stored as a two-dimensional array, and because each hash function will be generating an index into its own table, the Macah compiler will be able to scalarize the first dimension of the array, meaning all k hashes and lookups may be executed in parallel. The design of the partitioned Bloom filter as compared to the standard Bloom filter can be seen in figure 7.

Recall from the previous section that the false-positive rate P of a Bloom filter using k hash functions can be calculated as:

$$P = (1 - \phi)^k$$

Where ϕ represents the expected proportion of bits in the bit vector that are still set to zero, and is defined in terms of m (the number of bits in the bit vector) and n, the number of elements encoded in the bit vector as:

$$\phi = (1 - k/m)^n$$

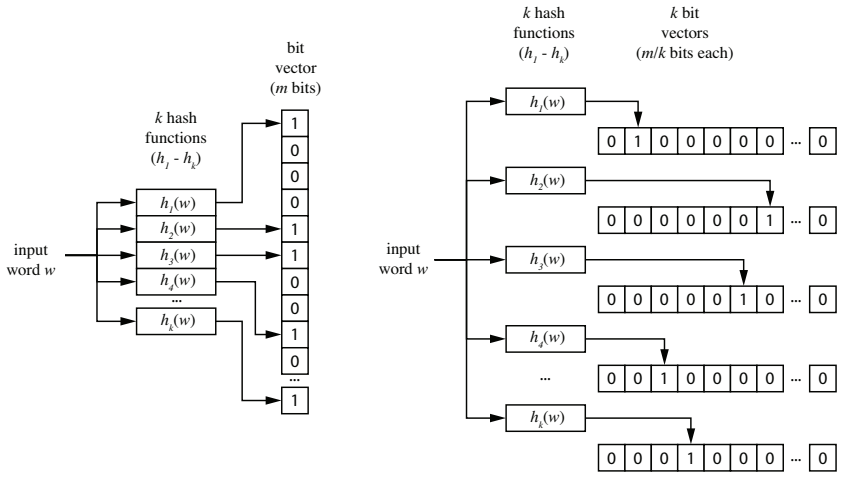

Figure 7: A standard Bloom filter (left) vs. a partitioned Bloom filter (right)

If m' is defined to be the size of the smaller tables in a partitioned Bloom filter, then the proportion of bits in the bit vector that are still set to 0 in the partitioned Bloom filter, ϕ_p, can be similarly defined as:

$$\phi_p = (1 - 1/m')^n$$

Recall from above, however, that $m' = m/k$, giving:

$$\phi_p = \left(1 - \frac{1}{m/k}\right)^n = (1 - k/m)^n$$

Meaning that $\phi_p = \phi$, and thus the false positive rate for the partitioned Bloom filter, P_p, is equal to the false-positive rate for the standard Bloom filter.

5.2 Hash Function Reuse

In [8], the authors chose to use the H_3 hash function for both their Bloom filter and hash table because of its amenability to efficient hardware implementation. Because it is used so frequently in both parts of stage 1, choosing a fast hash function is critical to the overall performance of stage 1. The H_3 hash function is described in [**?**], and consists of a number of terms XORed together (one term for each bit of the input key). The H_3 hash function h_q, as defined in [**?**], takes an input key x of width i bits, and produces an output index $h_q(x)$ that is j bits wide, making use of q, a randomly-generated $i \times j$ matrix in the following manner:

$$h_q(x) = x(1) \cdot q(1) \oplus x(2) \cdot q(2) \oplus \ldots \oplus x(i) \cdot q(i)$$

Where $x(n)$ represents bit n from the input word repeated j times, and $q(n)$ represents row n (a string of j bits) from the matrix q. During stage 1 of the

BLAST algorithm, the Bloom filter and hash table are held in the workspace memory while the database is streamed through and examined word by word. As can be seen in figure 3, adjacent words from the database differ by only two characters (the first and last). Because the database words are always examined in the order in which they appear in the database during stage 1, and each word must be examined, this overlap between adjacent words can be exploited to reduce the amount of work needed to generate hash functions for each word, as shown in figure 8.

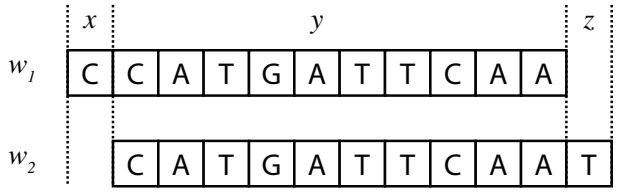

Figure 8: Two adjacent database words, showing the overlapping region y. Given a hash function h, $h(w_2)$ can be calculated from $h(w_1)$ by 'removing' the old portion (x) and 'adding' the new portion, z using the formula $h(w_2) = h(w_1) \oplus h(x) \oplus h(z)$, thus reducing the number of bits that must be hashed per word from 22 to 4 when using the default word size of 11 nucleotides.

6 Hash Table Implementation

Each word from the database string that passes through the Bloom filter must next be verified against the query string, both to ensure that it is not a false positive generated by collisions in the Bloom filter, and to discover the position(s) of the word within the query string. The standard approach, used in the NCBI BLAST, Mercury BLAST, and Mitrionics BLAST systems, is to use a hash table mapping each word from the query string to its position(s) within the query string.

The authors of [8] use a complex displacement hashing scheme that makes use of a large primary hash table, a smaller, more sparsely-populated secondary hash table to resolve collisions efficiently, and a duplicate table to store multiple query positions for a single word. This scheme seems to work well, but it requires the use of an significant amount of off-chip SRAM in order to store the hash table and associated data structures (the authors claim they are able to support queries of up to 17.5 kbases using a 1MB SRAM with their scheme).

Cuckoo hashing, a relatively new hashing technique first described in 2001 by Pagh et al. in [11] offers much better space efficiency than displacement hashing without sacrificing any lookup performance, making it more suitable for BLAST implementations targeted at coprocessor accelerator architectures, with their limited workspace memory. As will be shown, cuckoo hashing is also more naturally suited to coprocessor parallelization. Cuckoo hashing as

originally described makes use of two tables, T_1 and T_2, along with two hash functions, h_1 and h_2. Each hash function is associated with a single table, and maps input words to positions within its associated table, thus each input word x can be alternately stored in either $T_1[h_1(x)]$ or $T_2[h_2(x)]$.

When attempting to insert a word x into the hash table, $T_1[h_1(x)]$ and $T_2[h_2(x)]$ are first examined to see if they are occupied. If not, x is placed into one of the unoccupied slots. If, however, $T_1[h_1(x)]$ is occupied by the key x', and $T_2[h_2(x)]$ is occupied by the key x'', either x' or x'' will be chosen for eviction, and pushed to its alternate spot in the opposite table, making room for x. If the evicted key finds its alternate location occupied, it evicts the current occupant, propagating the chain of evictions down until all keys have been assigned to one of their two alternate locations, or until a cycle is entered (a very rare case in which re-hashing with two new hash functions is performed). As in [3], a separate duplicate table is used to store multiple query positions for a single word, supporting up 4 distinct query positions for a given word.

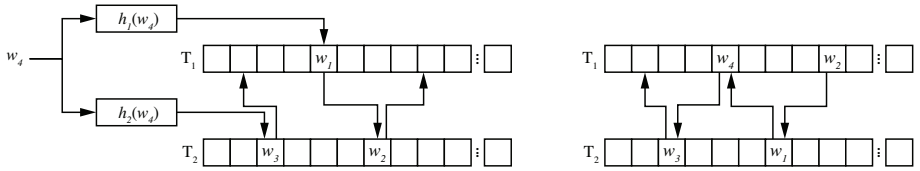

Figure 9: A diagram showing the an example cuckoo hash table before the insertion of a new word, w_4 (left) and afterwards (right). The new word is first run through the two hash functions (h_1 and h_2) in order to generate indices into the tables T_1 and T_2. Since both of the possible positions for are already occupied (by w_1 and w_3), w_1 is evicted to its alternate position in T_2, in turn forcing w_2 to its alternate position in T_1, and leaving a space for w_4.

This results in a hashing scheme with worst-case constant lookup time, in which *at most* 2 hash lookups must be performed in order to locate a given key, because each key must reside in one of its two alternate locations if it is in the table at all. Additionally, because the two hash lookups needed are independent and generate indices into separate tables, they can be performed in parallel. Because the size of the query string is usually quite small compared to the size of the database string, and because the query hash table must only be built once and not subsequently modified, the somewhat worse insertion performance of Cuckoo hashing as compared to more traditional hashing schemes is unimportant in BLAST.

The standard Cuckoo hashing implementation as described above will accommodate space utilization of up to 50%, however, in [6], the authors generalize binary Cuckoo hashing (using two tables and two hash functions) to d-ary Cuckoo hashing, utilizing d tables and hash functions in order to achieve much higher space utilization (91% for $d = 3$, and 97% for $d = 4$). Because each insertion probe or lookup requires d hash table positions to be calculated and checked, this results in a performance hit for both insertion and lookup on stan-

dard serial hardware. However, on parallel hardware, the extra hash function calculation and table lookup can be done in parallel, meaning that *increasing d to a modest value of 4 yields space utilization near 100% with no significant lookup performance hit.*

In summary, the use of 4-way Cuckoo hashing of the words from the query string makes it possible to fit reasonably-sized queries into a very small workspace memory. The actual query size that can be supported varies based on the details of the implementation, but assuming hash table entries of 64 bits each (the value used in Mitrion-C BLAST), and a space utilization of 95%, a query of 17500 kbases can be accommodated using 4 tables totaling less than 150 kilobytes in size, leaving 50 kilobytes for a duplicate table while still yielding a 5x space reduction over Mercury BLAST's hashing scheme.

7 Conclusions

Because the Macah compiler's Verilog code path is a work in progress, it is unfortunately not yet possible to synthesize and run the Macah BLAST version that has been produced (though it is possible to build compile to C for correctness testing purposes), and so no performance comparisons with NCBI BLAST can be made. However, the results of this investigation have been encouraging in that they have shown the relative ease with which a new and complex application can be ported to the Macah language.

Profiling results have supported the conclusion of other researchers that stage 1 of BLAST is the most significant contributor to the overall running time of the BLAST algorithm, even for the shorter query sizes typical of those that are submitted to NCBI's servers. It has been shown that the stage 1 acceleration approach presented by the authors of [8] and [3] of using a Bloom filter and hash table combination is applicable to the Macah language and abstract coprocessor accelerator architecture with only minimal modifications to avoid memory access serialization (Bloom filter partitioning) and reduce workspace memory utilization (cuckoo hashing). A method of re-using partially computed hash values during stage 1 has also been proposed. All of these optimizations are informed by a knowledge of the abstract model for coprocessor accelerator architectures.

8 Future Work

For the further development of accelerated BLAST in Macah, the most important and obvious next step is to use a completed version of the Macah compiler to synthesize a hardware implementation of the algorithm and measure its performance characteristics in comparison to NCBI BLAST. Performance against NCBI BLAST should be evaluated using a range of different query lengths, and query splitting should also be added in order to accommodate long queries that cannot fit within the workspace memory.

The acceleration of stage 2 of the algorithm, which accounts for a smaller but not insignificant amount of the total work, would also be a logical next step. The authors of [3] have proposed a method of accelerating stage 2 that should be fairly easy to implement in Macah. Based on the profiling results in section 2.2, it may also be worthwhile to replace BLAST stage 3 with an accelerated version as well. Such a modification would not be particularly difficult, given that a method for accelerating the standard Smith-Waterman alignment algorithm on FPGAs is well known.

Finally, determining the applicability of the modifications discussed in this paper to other BLAST variants (most notably `blastp`, used for protein searching) is another potentially fruitful path for future study.

9 Acknowledgements

This project would not have been possible without the mentorship and assistance of my advisors, Scott Hauck and Benjamin Ylvisaker. I have learned and continue to learn a great deal from them. I would also like to acknowledge the other undergraduate students working on Macah applications for their support, as well as my friends and family who have patiently listened to more talk about BLAST than should reasonably be expected from anyone.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

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

[3] Jeremy D. Buhler, Joseph M. Lancaster, Arpith C. Jacob, and Roger D. Chamberlain. Mercury blastn: Faster dna sequence comparison using a streaming hardware architecture. In *Reconfigurable Systems Summer Institute*, July 2007.

[4] George Coulouris. Blast benchmarks (powerpoint presentation). *NCBI FTP (ftp://ftp.ncbi.nih.gov/blast/demo/benchmark)*, 2005.

[5] Aaron E. Darling, Lucas Carey, and Wu C. Feng. The design, implementation, and evaluation of mpiblast.

[6] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time, 2003.

[7] Martin C. Herbordt, Josh Model, Yongfeng Gu, Bharat Sukhwani, and Tom Vancourt. Single pass, blast-like, approximate string matching on fpgas. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 217–226, 2006.

[8] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster. Biosequence similarity search on the mercury system, 2004.

[9] Mitrionics. Mitrion-c open bio project. *http://mitc-openbio.sourceforge.net/*.

[10] K. Muriki, K. D. Underwood, and R. Sass. Rc-blast: towards a portable, cost-effective open source hardware implementation. pages 8 pp.+, 2005.

[11] Rasmus Pagh and Flemming F. Rodler. Cuckoo hashing. *Lecture Notes in Computer Science*, 2161:121–??, 2001.

[12] Euripides Sotiriades and Apostolos Dollas. A general reconfigurable architecture for the blast algorithm. *The Journal of VLSI Signal Processing*, 48(3):189–208, 2007.

[13] Oystein Thorsen, Brian Smith, Carlos P. Sosa, Karl Jiang, Heshan Lin, Amanda Peters, and Wu-Chun Feng. Parallel genomic sequence-search on a massively parallel system. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 59–68, New York, NY, USA, 2007. ACM.

[14] Benjamin Ylvisaker, Brian Van Essen, and Carl Ebeling. A type architecture for hybrid micro-parallel computers. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 99–110, Washington, DC, USA, 2006. IEEE Computer Society.

[15] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning dna sequences. *J Comput Biol*, 7(1-2):203–214, 2000.