Acceleration of Ungapped Extension in Mercury BLAST

Joseph Lancaster
Jeremy Buhler
Roger Chamberlain

Joseph Lancaster, Jeremy Buhler, and Roger Chamberlain, "Acceleration of Ungapped Extension in Mercury BLAST," in *Proc. of the 7th Workshop on Media and Streaming Processors*, November 2005.

Washington University
Dept. of Computer Science and Engineering
Campus Box 1045
One Brookings Dr.
St. Louis, MO 63130

Acceleration of Ungapped Extension in Mercury BLAST

Joseph Lancaster
Washington University
in St. Louis
One Brookings Drive
St. Louis, MO 63130
lancaster@wustl.edu

Jeremy Buhler
Washington University
in St. Louis
One Brookings Drive
St. Louis, MO 63130
jbuhler@wustl.edu

Roger D. Chamberlain
Washington University
in St. Louis
One Brookings Drive
St. Louis, MO 63130
roger@wustl.edu

ABSTRACT

The amount of biosequence data being produced each year is growing exponentially. Extracting useful information from this massive amount of data efficiently is becoming an increasingly difficult task. There are many available software tools that molecular biologists use for comparing genomic data. This paper focuses on accelerating the most widely-used software tool, BLAST. Mercury BLAST takes a streaming approach to the BLAST computation by offloading the performance-critical sections to specialized hardware. This hardware is then used in combination with the processor of the host system to deliver BLAST results in a fraction of the time of the general-purpose processor alone.

This paper is the first dissemination of the design of the ungapped extension stage of Mercury BLAST. The architecture of the ungapped extension stage is described along with the context of this stage within the Mercury BLAST system. The design is compact and runs at 96 MHz on available FPGAs making it an effective and powerful component for accelerating biosequence comparisons. The performance of this stage is 25× that of the standard software distribution, yielding close to 50× performance improvement on the complete BLAST application. The sensitivity is essentially equivalent to that of the standard distribution.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems

General Terms

Algorithms, Design, Performance

Keywords

BLAST, Biosequence analysis, Sequence alignment, FPGA acceleration

MSP-7 Workshop at MICRO-38 Barcelona, Spain, 2005

1. INTRODUCTION

Databases of genomic DNA and protein sequences are an essential resource for modern molecular biology. Computational search of these databases can show that a DNA sequence acquired in the lab is similar to other sequences of known biological function, revealing both its role in the cell and its history over evolutionary time. A decade of improvement in DNA sequencing technology has driven exponential growth of biosequence databases such as NCBI GenBank [7], which has doubled in size every 12–16 months for the last decade and now stands at over 45 billion characters. Technological gains have also generated more novel sequences, including entire mammalian genomes [6, 11], to keep search engines busy.

The most widely used software for efficiently comparing biosequences to a database is BLAST, the **B**asic **L**ocal **A**lignment **S**earch **T**ool [1, 2, 3]. BLAST compares a *query sequence* to a biosequence database to find other sequences that differ from it by a small number of *edits* (single-character insertions, deletions, or substitutions). Because direct measurement of edit distance between sequences is computationally expensive, BLAST uses a variety of heuristics to identify small portions of a large database that are worth comparing carefully to the query.

BLAST is a pipeline of computations that filter a stream of characters (the database) to identify meaningful matches to a query. To keep pace with growing databases and queries, this stream must be filtered at increasingly higher rates. One path to higher performance is to develop a specialized processor that offloads part of BLAST's computation from a general-purpose CPU. Past examples of processors that accelerate or replace BLAST include the ASIC-based Paracel GeneMatcher^{TM} [8] and the FPGA-based TimeLogic DecypherBLAST^{TM} engine [10]. Recently, we have developed a new accelerator design, the FPGA-based Mercury BLAST engine [5]. Mercury BLAST exploits fine-grained parallelism in BLAST's algorithms and the high I/O bandwidth of current commodity computing systems to deliver 1–2 orders of magnitude speedup over software BLAST on a card suitable for deployment in a laboratory desktop.

Mercury BLAST is a multistage pipeline, parts of which are implemented in FPGA hardware. This work describes a key part of the pipeline, *ungapped extension*, that sifts

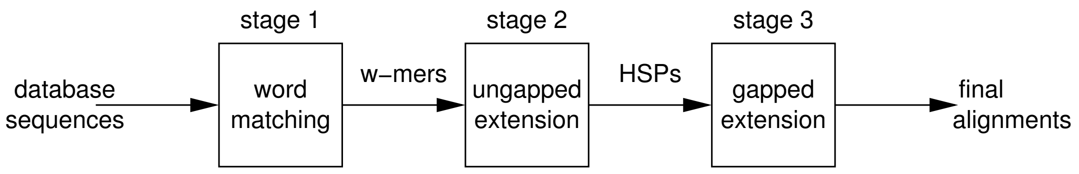

Figure 1: NCBI BLAST pipeline.

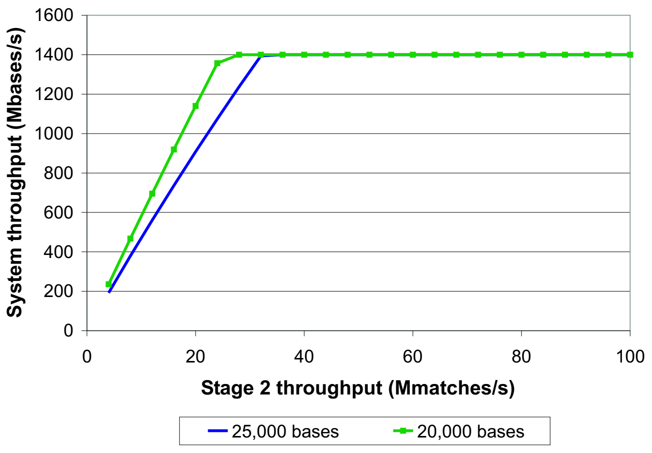

Figure 2: Throughput of overall pipeline as a function of ungapped extension throughput for queries of sizes 20 kbases and 25 kbases.

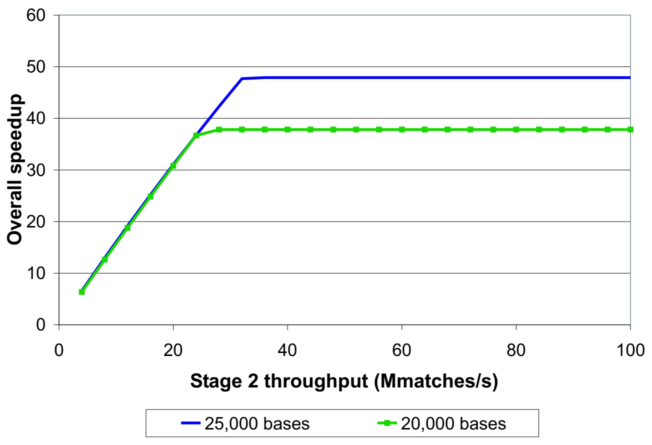

Figure 3: Speedup of overall pipeline as a function of ungapped extension throughput for queries of sizes 20 kbases and 25 kbases.

through pattern matches between query and database and decides whether to perform a more accurate but computationally expensive comparison between them. Our design illustrates a fruitful approach to accelerating variable-length string matching that is robust to character substitutions. The implementation is compact, runs at high clock rates, and can process one pattern match every clock cycle.

The rest of the paper is organized as follows. Section 2 gives a fuller account of the BLAST computation and illustrates the need to accelerate ungapped extension. Section 3 describes our accelerator design and details its hardware architecture. Section 4 evaluates the sensitivity and throughput of our implementation, and Section 5 concludes.

2. BACKGROUND: THE BLAST COMPUTATION

BLAST's search computation is organized as a three-stage pipeline, illustrated in Figure 1. The pipeline is initialized with a query sequence, after which a database is streamed through it to identify matches to that query. We focus on BLASTN, the form of BLAST used to compare DNA sequences; however, many of the details described here also apply to BLASTP, the form used for protein sequences.

The first pipeline stage, *word matching*, detects substrings of fixed length w in the stream that perfectly match a substring of the query; typically, $w = 11$ for DNA. We refer to these short matches as *w-mers*. Each matching w-mer is forwarded to the second stage, *ungapped extension*, which extends the w-mer to either side to identify a longer pair of sequences around it that match with at most a small number of mismatched characters. These longer matches are *high-scoring segment pairs (HSPs)*, or *ungapped alignments*. Finally, every HSP that has both enough matches and sufficiently few mismatches is passed to the third stage, *gapped extension*, which uses the Smith-Waterman dynamic programming algorithm [9] to extend it into a *gapped alignment*, a pair of similar regions that may differ by arbitrary edits. BLAST reports only gapped alignments with many matches and few edits.

To quantify the computational cost of each stage of BLASTN, we profiled the standard BLASTN software published by the National Center for Biological Information (NCBI), v2.3.2, on a comparison of 30 randomly selected 25,000-base queries from the human genome to a database containing the non-repetitive fraction of the mouse genome (1.16×10^9 characters). NCBI BLAST was profiled on a 2.8 GHz Intel P4 workstation running Linux. The search spent 83.9% of time in word matching, 15.9% in ungapped extension, and the remaining time in gapped extension. While execution time varied with query length (roughly linearly), the per stage percent execution times are relatively invariant with query length.

Our profile illustrates that, to achieve more than about a 6× speedup of NCBI BLASTN on large genome comparisons, one must accelerate *both* word matching *and* ungapped extension. Mercury BLASTN therefore accelerates both these stages, leaving stage 3 to NCBI's software. Our previous work [5] described how we accelerate word matching. This work represents the first description of Mercury BLAST's approach to ungapped extension.

With stage 1 (word matching) accelerated as described in [5], the performance of stage 2 (ungapped extension) directly determines the performance of the overall pipelined application. Figure 2 shows the application throughput (quantified by the ingest rate of the complete pipeline, in million bases per second) as a function of the performance attainable in stage 2 (quantified by the ingest rate of stage 2 alone, in million matches per second). The figure includes results for both 25,000-base queries and 20,000-base queries. Figure 3 plots the resulting speedup (for both query sizes) over NCBI BLAST executing on the reference system described above.

The software profiling shows, for 25,000-base queries, an average execution time for stage 2 alone of 0.265 μs/match. This corresponds to a throughput of 3.8 Mmatches/s, plotted towards the left of Figures 2 and 3. As we increase the performance of stage 2, the overall pipeline performance increases proportionately until stage 2 is no longer the bottleneck stage. Precisely how we do this is the subject of this paper.

Profiling of typical BLASTP computations reveals that ungapped extension accounts for an even larger fraction of the overall computational cost. The design described here for BLASTN ports with minimal changes to become an efficient stage 2 for BLASTP as well.

3. DESIGN DESCRIPTION

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

Mercury BLASTN implements stage 2 guided in part by lessons learned from the implementation of stage 1 described in [5]. It deploys an FPGA ungapped extension stage that acts as a *prefilter* in front of NCBI BLASTN's software ungapped extension. This design exploits the speed of FPGA implementation to greatly reduce the number of w-mers passed to software while retaining the flexibility of the software implementation on those w-mers that pass. A w-mer must pass both hardware and software ungapped extension before being released to gapped extension. Here, we are exploiting the streaming nature of the application as a whole. Since the performance focus is on overall throughput, not latency, adding an additional processing stage which is deployed on dedicated hardware is often a useful technique. In the next sections, we briefly describe NCBI BLASTN's software ungapped extension stage, then describe Mercury BLASTN's hardware stage. Figure 4 shows an example illustrating the two different approaches to ungapped extension.

3.1 NCBI BLAST Ungapped Extension

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

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

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

3.2 Mercury BLASTN's Approach

Mercury BLASTN takes a different, more FPGA-friendly approach to ungapped extension. Extension for a given w-mer is performed in a single forward pass over a fixed-size window. These features of our approach simplify hardware implementation and expose opportunities to exploit fine-grain parallelism and pipelining that are not easily accessed in NCBI BLASTN's algorithm. Our extension algorithm is given as pseudocode in Figure 5.

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

Extension implements a dynamic programming recurrence that simultaneously computes the start and end of the best HSP in the window. First, the score contribution of each character pair in the window is computed, using the same bonus $+\alpha$ and penalty $-\beta$ as the software implementation. These contributions can be calculated independently in parallel for each pair. Then, for each position i of the window, the recurrence computes the score γ_i of the best (highest-scoring) HSP that terminates at i, along with the position B_i at which this HSP begins. These values can be updated for each i in constant time. The algorithm also tracks Γ_i, the score of the best HSP ending at *or before* i, along with

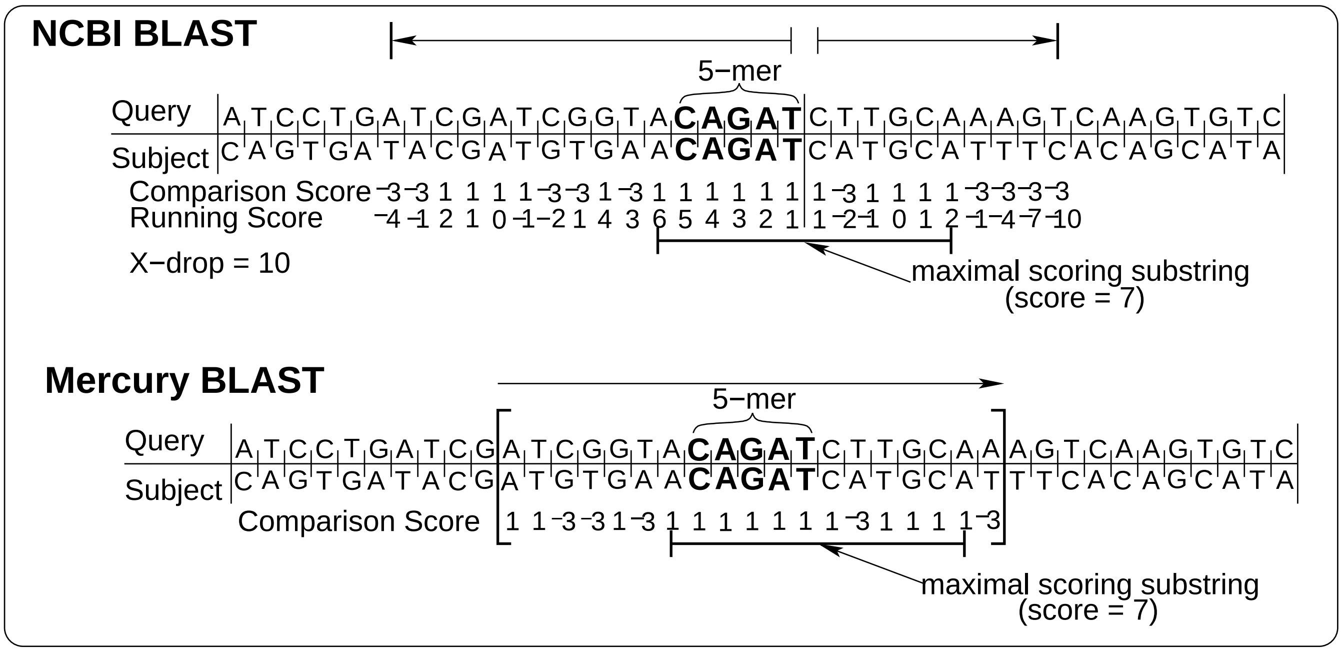
NCBI BLAST
5−mer
Query
Subject
A T C C T G A T C G A T C G G T A C A G A T C T T G C A A A G T C A A G T G T C
C A G T G A T A C G A T G T G A A C A G A T C A T G C A T T T C A C A G C A T A
Comparison Score −3 −3 1 1 1 1 −3 −3 1 −3 1 1 1 1 1 1 1 −3 1 1 1 1 −3 −3 −3 −3
Running Score −4 −1 2 1 0 −1 −2 1 4 3 6 5 4 3 2 1 1 −2 −1 0 1 2 −1 −4 −7 −10
X−drop = 10
maximal scoring substring (score = 7)
Mercury BLAST
5−mer
Query
Subject
Comparison Score 1 1 −3 −3 1 −3 1 1 1 1 1 1 1 −3 1 1 1 1 −3
maximal scoring substring (score = 7)

Figure 4: Examples of NCBI and Mercury ungapped extension. The parameters used here are $L_w = 19$, $w = 5$, $\alpha = 1$, $\beta = -3$, and X-drop$= 10$. NCBI BLAST ungapped extension begins at the end of the w-mer and extends left. The extension stops when the running score drops 10 below the maximum score (as indicated by the arrows). The same computation is then performed in the other direction. The final substring is the concatenation of the best substrings from the left and right extensions. Mercury BLAST ungapped extension begins at the leftmost base of the window (indicated by brackets) and moves right, calculating the best-scoring substring in the window. In this example, the algorithms gave the same result, but this is not necessarily the case in general.

its endpoints B_{max} and E_{max}. Note that Γ_{L_w} is the score of the best HSP in the entire window. If Γ_{L_w} is greater than a user-defined score threshold, the w-mer passes the prefilter and is forwarded to software ungapped extension.

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

3.3 Implementation

As the name implies, Mercury BLAST has been targeted to the Mercury system [4]. The Mercury system is a prototyping infrastructure designed to accelerate disk-based computations. This system exploits the high I/O bandwidth available from modern disks by streaming data directly from the disk medium to the FPGA. Figure 6 shows the organization of the application pipeline for BLASTN. The input comes

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

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

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

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

Figure 5: Mercury BLAST Extension algorithm pseudocode.

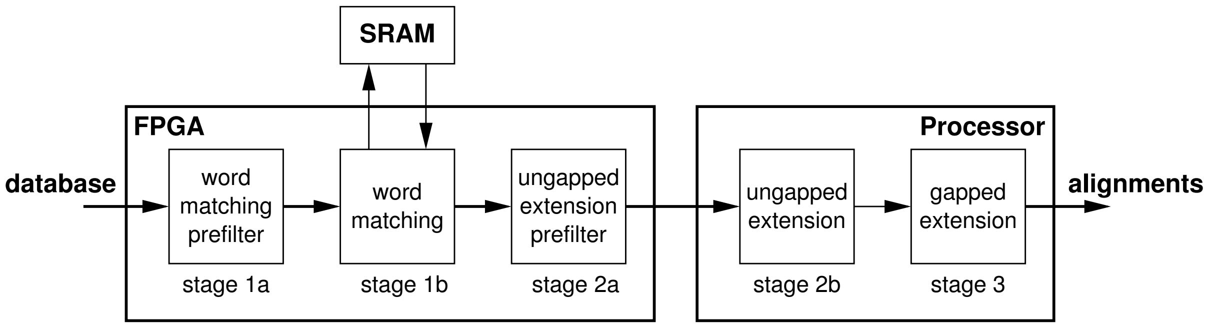

Figure 6: Overview of Mercury BLASTN hardware/software deployment.

from the disk, is delivered to the hardware word matching module (which also employs a prefilter), and then passes into the ungapped extension prefilter. The output of the prefilter goes to the processor for the remainder of stage 2 (ungapped extension) and stage 3 (gapped extension). The prefilter algorithm lends itself to hardware implementation despite the sequential expression of the computation in Figure 5.

The ungapped extension prefilter design is fully pipelined internally and accepts one match per clock. The prefilter is parameterizable with the parameters chosen based on a number of design-specific constraints. Commands are supported to configure parameters such as match score, mismatch score, and cutoff thresholds. Thus, the trade-off between sensitivity and throughput is left to the discretion of the user. Since the w-mer matching stage generates more output than input, two independent data paths are used for input into stage 2. The w-mers and commands are sent on one path, and the database is sent on the other. The module is organized as 3 pipelined stages as illustrated in Figure 7.

The controller parses the input to demultiplex the shared command and data stream. All w-mer matches and the database flow through the controller into the window lookup module. This module is responsible for fetching the appropriate substrings of the database stream and the query to form the alignment window. The query is buffered on-chip using the dual-ported BRAM on the FPGA. The database stream is buffered in a circular buffer which is also created from BRAMs. This buffer retains only the portion of the stream needed to form the windows for the input w-mers. As mentioned earlier, the w-mer generation is done in the first hardware stage. Only a small amount of the database stream needs to be buffered to accommodate all input w-mers because they arrive from stage 1 in-order with respect to the database stream. Since the BRAMs are a highly-utilized resource in stage 1, the BRAMs are time-multiplexed to create a quadported BRAM structure. This allows stage 2 to use half the number of BRAMs for buffering that would otherwise be necessary. After the window is fetched, it is passed into the scoring module and stored in registers. The scoring module implements the recurrence of the extension algorithm. Since the computation is too complex to be done in a single cycle, the scorer is extensively pipelined.

Figure 8 illustrates the first stage of the scoring pipeline. This stage, the base comparator, assigns a *comparison score* to each base pair in the window. For BLASTN, the base comparator assigns a reward α to each matching base pair and a penalty $-\beta$ to each mismatching pair. The score computation is the same for BLASTP, except there are many more choices for the score. In BLASTP, the α and $-\beta$ are replaced with a value retrieved from a lookup table that is indexed by the concatenation of the two bases. The calculation of all comparison scores is done in a single cycle, using L_w comparators. After the scores are calculated, they are stored for use in later stages of the pipeline.

The scoring module is arranged as a classic systolic array. The data from the previous stage are read on each clock, and results are output to the following stage on the next clock. As Figure 7 shows, storage for comparison scores in successive pipeline stages decreases in every stage. This decrease is possible because the comparison score for window position i is consumed in the ith pipeline stage and may then be discarded, since later stages inspect only window positions $> i$. This structure of data movement is shown in more detail in Figure 9. The darkened registers hold the necessary comparison scores for the w-mer being processed in each pipeline stage. Note that for ease of discussion, Figure 9 shows a single comparison score being dropped for each scorer stage; however, the actual implementation consumes two comparison scores per stage.

Figure 10 shows the interface of an individual scoring stage. The values shown entering the top of the scoring stage are the state of dynamic programming recurrence propagated from the previous scoring stage. These values are read as input to combinational logic and the results are stored in the output registers shown on the right. Each scoring stage in the pipeline contains combinational logic to implement the dynamic programming recurrence shown in lines 12-19 of the algorithm described in Figure 5. The data entering from the left of the module are the comparison scores and the database and query positions for a given w-mer, which are independent of the state of the recurrence. In order to sustain a high clock frequency design, each scoring stage computes only two iterations of the loop per clock cycle, resulting in $L_w/2$ scoring stages for a complete calculation. Hence, there are $L_w/2$ independent w-mers being processed simultaneously in the scoring stages of the processor when the pipe is full.

The final pipeline stage of the scoring module is the threshold comparator. The comparator takes the fully-scored seg-

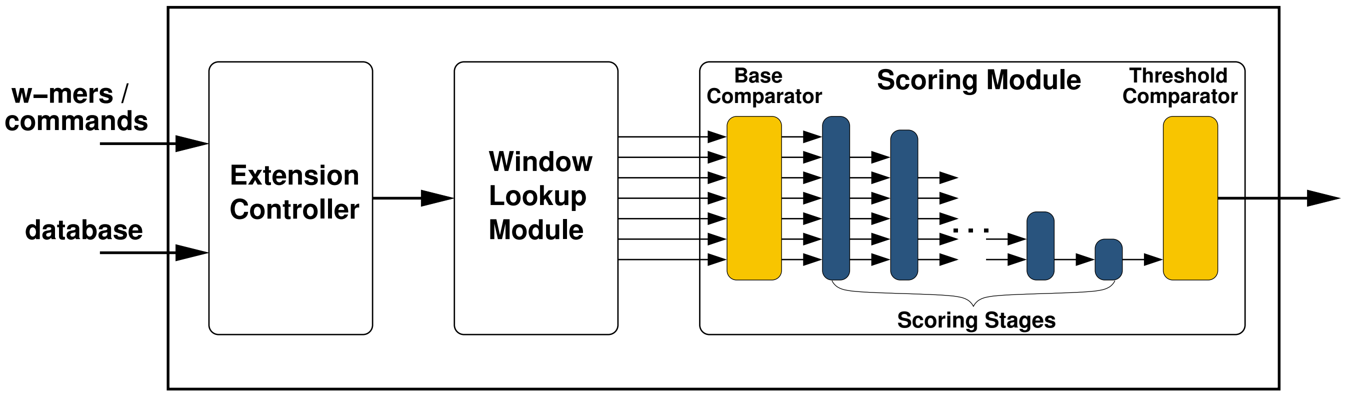

Figure 7: Ungapped extension prefilter design.

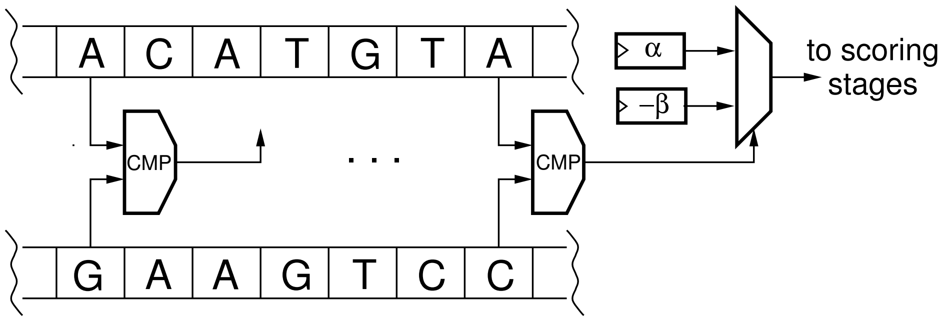

Figure 8: Illustration of the base comparator for BLASTN. The base comparator stage computes the scores of every base pair in the window in parallel. These scores are stored in registers which are fed as input to the systolic array of scorer stages.

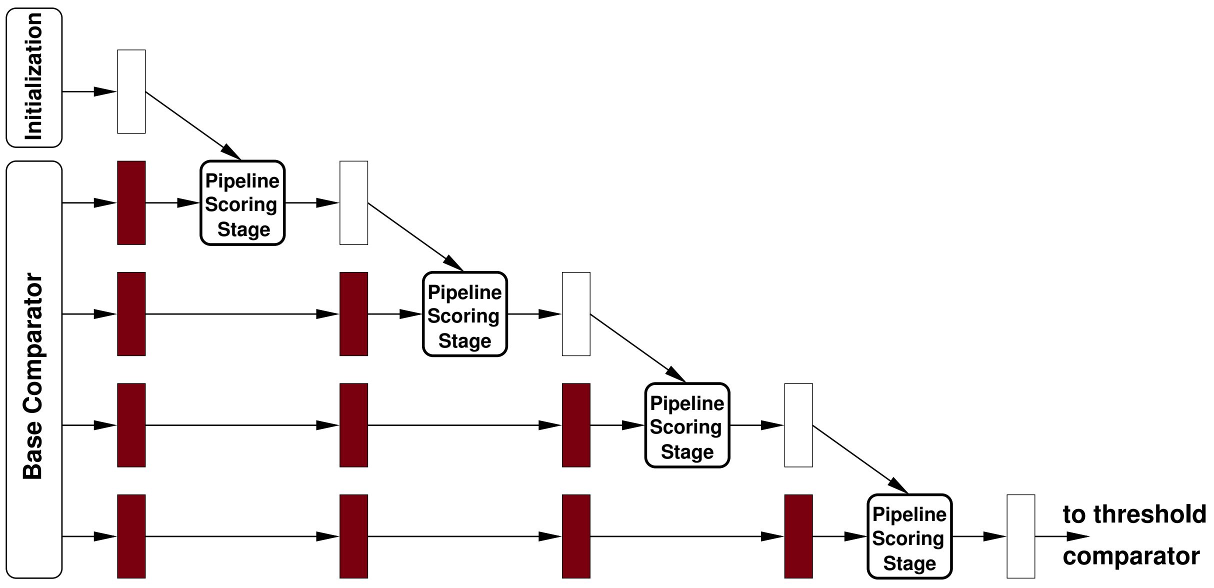

Figure 9: Depiction of the systolic array of scoring stages. The dark registers hold data which is independent of the state of the recurrence. The data flow left to right on each clock cycle. The light registers are the pipeline calculation registers used to transfer the state of the recurrence from a previous scoring stage to the next. Each column of registers contains an different w-mer in the pipeline.

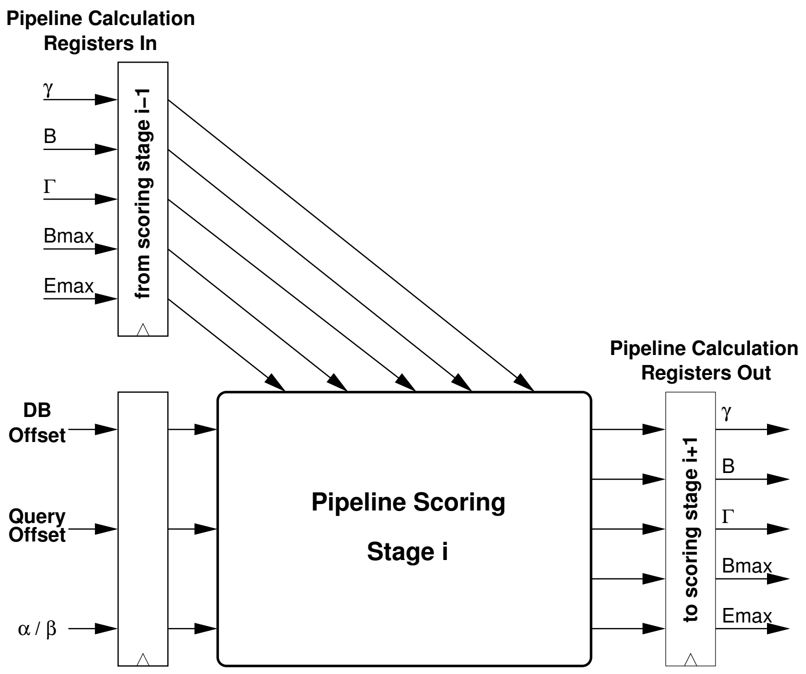

Figure 10: Detailed view of an individual scoring stage.

ment and makes a decision to discard or keep the w-mer. This decision is based on the score of the alignment relative to a user-defined threshold T, as well as the position of the highest-scoring substring. If the maximum score is above the threshold, the w-mer is passed on. Additionally, if the maximal scoring substring intersects either boundary of the window, the w-mer is also passed on, regardless of the score. If neither condition holds, the w-mer is discarded.

In the current implementation, the ungapped extension prefilter stage utilizes approximately 38% of the logic cells and 27 BRAMs on a Xilinx Virtex-II 6000 series FPGA, including the infrastructure for moving data in and out of the FPGA itself. The design runs at 96 MHz, processing one w-mer per clock. The full Mercury BLASTN design utilizes approximately 65% of the logic cells and 134 BRAMs.

4. RESULTS

There are many aspects to the performance of an ungapped extension filter. First is the individual stage throughput. The ungapped extension stage must run fast enough to not be a bottleneck in the overall pipeline. Second, the ungapped extension stage must effectively filter as many w-mers as possible, since downstream stages are even more computationally expensive. Finally, the above must be achieved without inadvertently dropping a large percentage of the significant alignments (i.e., the false negative rate must be limited). We will describe throughput performance first for the ungapped extension stage alone, then for the entire BLASTN application. After describing throughput, we discuss the sensitivity of the pipeline to significant sequence alignments.

The throughput of Mercury BLASTN ungapped extension is a function of the data input rate. The ungapped extension stage accepts one w-mer per clock and runs at 96 MHz. Hence the maximum throughput of the filter is 1 input match/cycle × 96 MHz = 96 Mmatches/second. This gives a speedup of 25× over the software ungapped extension executed on the baseline system described earlier.

Table 1: BLASTN sensitivity results using the hardware prefilter. For all experiments, the window size is 64 characters.

Score Threshold	Reject Fraction	Percent Found
20	0.999902	99.72
18	0.999506	99.92
17	0.997221	99.95
16	0.995735	99.96

To explore the impact this stage 2 performance has on the overall system, we return to the graphs of Figures 2 and 3. Above approximately 35 Mmatches/s, the overall system throughput is at its maximum rate of 1400 Mbases/s, with a speedup of 48× over that of software NCBI BLASTN for a 25,000-base query and a speedup of 38× over software for a 20,000-base query.

Two parameters of ungapped extension affect its sensitivity. First, the score threshold used affects the number of HSPs that are produced. Second, the length of the window can affect the number of false negatives that are produced. HSPs which have enough mismatches before the window boundary to be below the score threshold but have many matches immediately outside the boundary will be incorrectly rejected.

To evaluate the functional sensitivity of hardware ungapped extension, measurements were performed using an instrumented version of NCBI BLASTN. A software emulator of the new ungapped extension algorithm was placed in front of the standard NCBI ungapped extension stage. Statistics were gathered which show how many w-mers arrived at the ungapped extension stage, and how many passed. These statistics were collected both for NCBI BLASTN ungapped extension alone and with the hardware emulator in place. The dataset was generated from the human and mouse genomes. The queries were statistically significant samples of various sizes (e.g., 10 kbase, 100 kbase, and 1 Mbase). The database stream was the mouse genome with low-complexity and repetitive sequences removed.

Table 1 summarizes the results for a window size of 64 bases, which is the window size used in the current hardware implementation. A score threshold of 20 corresponds to the default value in NCBI BLASTN ungapped extension. The reject fraction is the measured ratio of output HSPs over input w-mers. This value quantifies the effectiveness of the overall stage 2 at filtering w-mers so they need not be processed in stage 3. The percent found is the percentage of gapped alignments present in the output of NCBI BLASTN that are also present in the output of Mercury BLASTN. Using a window length of 64 bases, the ungapped extension prefilter is able to filter out between 99.5% and 99.99% of all its input. For instance, a threshold of 17 gives a good tradeoff between a high reject fraction while keeping the vast majority (99.95%) of the significant HSPs. This translates into 5 missed HSPs out of 10334 HSPs found by NCBI BLAST.

5. CONCLUSION

Biosequence similarity search can be accelerated practically by a pipeline designed to filter high-speed streams of character data. We have described a portion of our Mercury BLASTN search accelerator, focusing on our design for its performance-critical ungapped extension stage. Our highly parallel and pipelined implementation yields results comparable to those obtained from software BLASTN while running 25× faster and enabling the entire accelerator to run approximately 40× to 50× faster. Currently a functionally complete implementation has been deployed on the Mercury system and is undergoing performance tuning.

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

Acknowledgement

This research is supported by NIH/NGHRI grant 1 R42 HG003225-01 and NSF Career grant DBI-0237902.

6. REFERENCES

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

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

[3] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–402, 1997.

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

[5] P. Krishnamurthy, J. Buhler, R. D. Chamberlain, M. A. Franklin, K. Gyang, and J. Lancaster. Biosequence similarity search on the Mercury system. In *Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP04)*, pages 365–75, 2004.

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

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

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

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

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

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