Published in final edited form as:
Microprocess Microsyst. 2009 June 1; 33(4): 281–289. doi:10.1016/j.micpro.2009.02.007.

Acceleration of Ungapped Extension in Mercury BLAST

Joseph Lancaster,
Department of Computer Science and Engineering, Washington University in St. Louis, E-mail: lancaster@wustl.edu

Jeremy Buhler, and
Department of Computer Science and Engineering, Washington University in St. Louis, E-mail: jbuhler@wustl.edu

Roger D. Chamberlain
Department of Computer Science and Engineering, Washington University in St. Louis, and BECS Technology, Inc., St. Louis, MO, E-mail: 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 such tool, BLAST. Mercury BLAST takes a streaming approach to the BLAST computation by off loading 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 presents 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 100 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.

Keywords

BLAST; biosequence analysis; sequence alignment; FPGA acceleration

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 Gen-Bank [National Center for Biological Information, 2002], which has doubled in size every 12–16 months for the last decade and now stands at over 59 billion characters. Technological

Correspondence to: Joseph Lancaster.

gains have also generated more novel sequences, including entire mammalian genomes [Lander et al., 2001, Waterston et al., 2002], 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 [Altschul and Gish, 1996, Altschul et al., 1990, Altschul et al., 1997]. 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™ [Paracel, 2006] and the FPGA-based TimeLogic DecypherBLAST™ engine [Timelogic, 2006]. Recently, we have developed a new accelerator design, the FPGA-based Mercury BLAST engine [Krishnamurthy et al., 2004]. 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 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. Section 5 describes related work, and Section 6 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. This paper focuses 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, *un-gapped 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 [Smith and Waterman, 1981] 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 proffiled 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 nonrepetitive 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* un-gapped extension. Mercury BLASTN therefore accelerates both these stages, leaving stage 3 to NCBI's software. Our previous work [Krishnamurthy et al., 2004] described how we accelerate word matching. This work describes Mercury BLAST's approach to ungapped extension.

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 [Krishnamurthy et al., 2004]. 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 that is deployed on dedicated hardware is often a useful technique. In the next sections, we briefly describe NCBI BLASTN's software un-gapped extension stage, then describe Mercury BLASTN's hardware stage. Figure 2 shows an example illustrating the two different approaches to ungapped extension.

3.1 NCBI BLASTN 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 parallelism 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 3.

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 its endpoints B_{max} and E_{max}. Note that Γ_{Lw} is the score of the best HSP in the entire window. If Γ_{Lw} 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 [Chamberlain et al., 2003]. The Mercury system is a prototyping infrastructure designed to accelerate disk-based computations. Its organization is illustrated in Figure 4. Data flows off the disks into an FPGA. The FPGA provides reconfigurable logic that has its function specified via HDL.

Results of the processing performed on the FPGA are delivered to the processor. By delivering the high-volume data directly to the FPGA, the processor can be relieved of the requirement of handling the bulk of the original data set. Associated with the re-configurable logic is a configuration store that maintains a number of FPGA configurations.

Figure 5 shows the organization of the application pipeline for BLASTN. The input comes from the disk, is delivered to the hardware word matching module (which also employs a prefilter), and then passes into the un-gapped 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 3.

The ungapped extension prefilter design is fully pipelined internally and accepts one match per clock. 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 6.

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 stream and the query. Figure 7 shows the overall structure of the module. The query is stored on chip using the dual-ported BlockRAMs on the FPGA. The query is loaded once at the beginning of the BLAST computation and is fixed until the end of the database is reached. The size of the query is limited to the amount of BlockRAM that is allocated for buffering it. In the current implementation, a maximum query size of 65,536 is supported using 8 BlockRAMs. The subset of the database stream needed for scoring is retained in a circular buffer also constructed using BlockRAMs. Since the *w*-mer generation is performed in the first hardware stage, only a relatively small amount of the database stream needs to be buffered to accommodate extension requests. The database buffer is large enough to accommodate *w*-mers whose positions in the stream are out of order by up to a fixed amount. While not critical for BLASTN, this extra capacity is important in the BLASTP implementation, since the *w*-mers from the word matching stage can be out of order.

The window lookup module is organized as a 6-stage pipeline. The first stage of the pipeline calculates the beginning of the query and database windows based on the incoming *w*-mer and a configurable window size. The off-set of the beginning of the window is then passed to the buffer modules, which begin the task of reading a superset of the window from the BlockRAMs. One extra word of data must be retrieved from the BlockRAMs because there is no guarantee that the window boundaries will fall on a word boundary. Hence, one extra word is fetched on each lookup so that the exact window can be constructed from a temporary buffer holding a window size worth of data plus the extra word. The next four stages of the pipeline move the input data in lock step with the buffer lookup process and ensure that pipeline stalls are handled in a correct fashion. In the final stage, the superset of the query and database windows arrive to the format output module. The correct window of the buffers is extracted and registered as the output.

After the window is fetched, it is passed into the scoring module. 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 –β to each mismatching pair. All comparison scores are calculated 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 score computation is the same for BLASTP, except that there are many more choices for the score. In BLASTP, the alphabet consists of 20 amino acids rather than the 4 bases of BLASTN. As a result, amino acids are represented by 5 bits each. The α and –β are replaced with a value retrieved from a lookup table that is indexed by a pair of of amino acids. This is shown in Figure 9. The remainder of the stage 2 design is common to both BLASTN and BLASTP.

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 6 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 *i*th 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 10. 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 10 shows a single comparison score being dropped for each scorer stage; however, the actual implementation consumes two comparison scores per stage.

Figure 11 shows the interface of an individual scoring stage. The values shown entering the top of the scoring stage are the state of the dynamic programming recurrence as 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 13–22 of the algorithm described in Figure 3. 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, 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 segment 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. This second condition ensures that a high-scoring alignment that extends beyond the window boundary is not spuriously discarded. If neither of the above conditions holds, the *w*-mer is discarded.

In the current design, the window size, L_w, is set at the time the design is synthesized, while the scoring parameters, α and –β, and score threshold, *T*, are set at run time.

4 RESULTS

There are many aspects to the performance of the un-gapped extension stage of the BLAST pipeline. First is the individual stage throughput. The ungapped extension prefilter must run fast enough not to 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 signifficant alignments (i.e., the false negative rate must be limited). We will describe throughput performance first for the ungapped extension prefilter alone, then for the entire BLASTN application. After describing throughput, we discuss the sensitivity of the pipeline to significant sequence alignments.

4.1 Performance

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

To explore the impact that stage 2a performance has on the overall streaming application, we use the following mean-value performance model. Overall pipeline throughput for the deployment of Figure 5 is

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

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

For the above expression to be correct, all of the throughputs must be normalized to the same units; we will normalize to input DNA bases per unit time. This normalization can be accomplished with knowledge of the fractions of input bases that survive each of the stages of filtering. Call the *w*-mers from stage 1 "matches" and the HSPs from stages 2a and 2b "alignments." We define p_1 as the pass fraction from stage 1 (matches out per base in), p_{2a} as the pass fraction from stage 2a (alignments out per match in), and p_{2b} as the pass fraction from stage 2b (alignments out per alignment in). With this information, we compute the normalized throughput of stage 2a as $Tput_{2a}$ = (100 Mmatches/sec)/p_1. Finally, we need the time required to process alignments in software (both in stage 2b and in stage 3). We define t_{2b} as the execution time per input alignment for the stage 2b software and t_3 as the execution time per input alignment for the stage 3 software. The normalized throughput of the stages executing on the software can then be expressed as

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

Table 1 provides the above parameters and their values for a 25-kbase, double-stranded query. The values of p_1, t_{2b}, t_3, and $Tput_1$ come from [Krishnamurthy et al., 2004]. Clearly, the overall throughput is limited by the capacity of stage 1.

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

with the I/O limitations of the existing infrastructure, Mercury BLASTN has a speedup over software NCBI BLASTN of approximately 48× (for a 25-kbase query, NCBI BLASTN has a throughput of 29 Mbas-es/sec [Krishnamurthy et al., 2004] vs. Mercury BLASTN throughput of 1.4 Gbases/sec).

4.2 Quality of results: sensitivity

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

Formally, sensitivity is defined as follows:

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

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

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

We measured the sensitivity of BLASTN with our un-gapped extension design using a modified version of the NCBI BLASTN code base. A software emulator of the new ungapped extension algorithm (stage 2a) was placed in front of the standard NCBI ungapped extension stage (stage 2b). We used BLAST with this emulator to compare sequences extracted from the human and mouse genomes. The queries were statistically significant samples of various sizes (10 kbase, 100 kbase, and 1 Mbase). The database stream was the mouse genome with low-complexity and repetitive sequences removed.

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

experiments, $f = 1/2$, an arbitrary but reasonable fraction given that in practice almost all overlaps are complete (100% overlap).

Figure 12 illustrates the sensitivity of the Mercury BLAST system when the hardware ungapped filter is combined with the NCBI ungapped extension filter to make up the full stage 2 (i.e., the configuration of Figure 5). Using window sizes of 64 to 256 bases, the combined filters yielded sensitivity in excess of 99.6%; in absolute terms, the worst observed false negative rate was only 36 false negatives out of 11616 significant alignments. Larger window sizes resulted in higher sensitivity, up to 100%; however, increasing the the window size above 96 yielded little benefit for the additional logic consumed. A window size of 64 reduced sensitivity below 99.9%, but this loss can be compensated by lowering the score threshold. We conclude that using the hardware prefilter in this configuration does not noticeably degrade the quality of results.

4.3 Efficiency of filtration: specificity

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

For BLASTN ungapped extension, specificity is measured as follows:

Specificity = 1 − (# alignments out /# matches in), which for stage 2a is $1 - p_{2a}$ and for the complete stage 2 is $1 - p_{2a}p_{2b}$. To quantify the specificity of our implementation, we gathered statistics during the aforementioned experiments on how many *w*-mers (matches) arrived at the ungapped extension stage, and how many of these produced ungapped alignments that passed the stage's score threshold.

Figure 13 shows the specificity of Mercury BLASTN un-gapped extension for various score thresholds and window lengths. In this graph, the ungapped extension stage consists of the hardware filter alone (i.e., stage 2a), *without* NCBI BLAST's software ungapped filter. As the score threshold increases, the hardware passes fewer *w*-mers, and so the specificity of the filter increases. Specificity is not strongly influenced by window size. At a score threshold of 18, the hardware prefilter is approximately as specific as the original NCBI BLASTN software's ungapped extension stage.

Figure 14 shows the specificity of the combined hardware filter and software ungapped extension filter. As expected, the specificity is essentially constant, with only a miniscule increase at the highly stringent score threshold of 20.

4.4 Resource utilization

As mentioned earlier, the Mercury ungapped extension filter is parameterizable and can be configured for different window lengths. Currently, the filter exists with window lengths of 64, 96, and 128 bases. Table 2 gives resource usage information for each of these design points. For comparison, the full Mercury BLASTN design, including both stages 1 and 2a, utilizes approximately 54% of the logic cells and 134 BlockRAMs of our FPGA platform with a stage 2a window size of 64.

5 RELATED WORK

Several research groups have implemented systems to accelerate similarity search, both in software alone and in hardware.

Software tools exist that seek to accelerate BLASTN-like computations through algorithmic improvements. MegaBLAST [Zhang et al., 2000] is used by NCBI as a faster alternative to BLASTN; it explicitly sacrifices substantial sensitivity relative to BLASTN in exchange for improved running time. The SSAHA [Ning et al., 2001] and BLAT [Kent, 2002] packages achieve higher throughput than BLASTN by requiring that the entire database be indexed offline before being used for searches. By eliminating the need to scan the database, these tools can achieve more than an order of magnitude speedup versus BLASTN; however, they must trade off between sensitivity and space for their indices and so in practice are less sensitive. In contrast, Mercury BLASTN aims for at least BLASTN-equivalent sensitivity.

Other software approaches, such as DASH [Knowles and Gardner-Stephen, 2004a] and PatternHunter II [Li et al., 2004], achieve both faster search and higher sensitivity compared to BLASTN using alternative forms of pattern matching and dynamic programming extension. DASH's reported speedup over BLASTN is less than 10-fold for queries of 1500 bases, and it is not clear how it performs at our query sizes, which are an order of magnitude larger. DASH's authors have also reported on a preliminary FPGA design for their algorithm [Knowles and Gardner-Stephen, 2004b], while PatternHunter II achieves only a two-fold reported speedup relative to BLASTN, albeit with substantially greater sensitivity.

In hardware, numerous implementations of the Smith-Waterman dynamic programming algorithm have been reported in the literature, using both nonreconfigurable ASIC logic [Singh et al., 1996, Hirschberg et al., 1996] and reconfigurable logic [Hoang, 1995, Pappas, 2003, Yamaguchi et al., 2002]. These implementations focus on accelerating gapped alignment, which is heavily loaded in proteomic BLAST comparisons but takes only a small fraction of running time in genomic BLASTN computations. Our work instead focuses on accelerating the bottleneck stages of the BLASTN pipeline, which reduces the data sent to later stages to the point that Smith-Waterman acceleration is not necessary.

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

High-end commercial systems have been developed to accelerate or replace BLAST. The Paracel GeneMatcher™ [Paracel, 2006] relies on non-reconfigurable ASIC logic, which is infiexible in its application and cannot easily be updated to exploit technology improvements. In contrast, FPGA-based systems can be reprogrammed to tackle diverse applications and can be redeployed on newer, faster FPGAs with minimal additional design work. RDisk [Lavenier et al., 2003] is one such FPGA-based approach, which claims 60 Mbases/sec throughput for stage 1 of BLAST using a single disk.

Two commercial products that do not rely on ASIC technology are BLASTMachine2™ from Paracel [Paracel, 2006] and DeCypherBLAST™ from Time-Logic [Timelogic, 2006]. The highest-end 32-CPU Linux cluster BLASTMachine2™ performs BLASTN with a throughput of 2.93 Mbases/sec for a 2.8-Mbase query. The DeCypherBLAST™ solution uses an FPGA-based approach to improve the performance of BLASTN. This solution has throughput rate of 213 kbases/sec for a 16-Mbase query.

6 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 close to 50× faster. Currently, a functionally complete implementation of the accelerator 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. We are using this stage in our in-progress design for Mercury BLASTP and expect that it will prove similarly successful in that application.

Acknowledgments

This research is supported by NIH/NGHRI grant 1 R42 HG003225-01 and NSF grants DBI-0237902 and CCF-0427794. R. Chamberlain is a principal in BECS Technology, Inc. This work is an extension of the previously published paper, J. Lancaster, J. Buhler, and R. Chamberlain, "Acceleration of Ungapped Extension in Mercury BLAST," in *Proc. of 7th Workshop on Media and Streaming Processors*, November 2005.

References

Altschul SF, Gish W. Local alignment statistics. Methods: a Companion to Methods in Enzymology 1996;266:460–80.

Altschul SF, Gish W, Miller W, Myers EW, et al. Basic local alignment search tool. Journal of Molecular Biology 1990;215:403–10. [PubMed: 2231712]

Altschul SF, Madden TL, Sch•affer AA, Zhang J, Zhang Z, Miller W, Lipman DJ. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research 1997;25:3389–402. [PubMed: 9254694]

Chamberlain RD, Cytron RK, Franklin MA, Indeck RS. The *Mercury* system: Exploiting truly fast hardware for data search. Proceedings of International Workshop on Storage Network Architecture and Parallel I/Os 2003:65–72.

Hirschberg JD, Hughley R, Karplus K. Kestrel: a programmable array for sequence analysis. Proceedings of IEEE International Conference on Application-Specific Systems, Architecture, and Processors 1996:23–34.

Hoang DT. Searching genetic databases on Splash 2. IEEE Workshop on FPGAs for Custom Computing Machines 1995:185–91.

Kent WJ. BLAT: the BLAST-like alignment tool. Genome Research 2002;12:656–64. [PubMed: 11932250]

Knowles G, Gardner-Stephen P. DASH: localizing dynamic programming for order of magnitude faster, accurate sequence alignment. Proceedings of the IEEE Computational Systems Bioinformatics Conference 2004a:732–35.

Knowles G, Gardner-Stephen P. A new hardware architecture for genomic and proteomic sequence alignment. Proceedings of the IEEE Computational Systems Bioinformatics Conference 2004b:730–31.

Krishnamurthy P, Buhler J, Chamberlain RD, Franklin MA, Gyang K, Lancaster J. Biosequence similarity search on the Mercury system. Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures, and Processors 2004:365–75.

Lander ES, et al. Initial sequencing and analysis of the human genome. Nature 2001;409:860–921. [PubMed: 11237011]

Lavenier D, Guytant S, Derrien S, Rubin S. A reconfigurable parallel disk system for filtering genomic banks. Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms 2003:154–66.

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

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

Ning Z, Cox AJ, Mullikin JC. SSAHA: a fast search method for large DNA databases. Genome Research 2001;11:1725–9. [PubMed: 11591649]

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

Paracel. 2006. Paracel, Inc. http://www.paracel.com

Singh RK, et al. BioSCAN: a dynamically reconfigurable systolic array for biosequence analysis. Proceedings of CERCS '96. 1996

Smith TF, Waterman MS. Identification of common molecular subsequences. Journal of Molecular Biology 1981;147(1):195–97. [PubMed: 7265238]

Timelogic. 2006. TimeLogic Corporation. http://www.timelogic.com

Waterston RH, et al. Initial sequencing and comparative analysis of the mouse genome. Nature 2002;420:520–62. [PubMed: 12466850]

Yamaguchi Y, Maruyama T, Konagaya A. High speed homology search with FPGAs. Pacific Symposium on Biocomputing 2002:271–82. [PubMed: 11928482]

Zhang Z, Schwartz S, Wagner L, Miller W. A greedy algorithm for aligning DNA sequences. Journal of Computational Biology 2000;7:203–14. [PubMed: 10890397]

Biography

Biographical notes: Joseph Lancaster received his BS degrees in Computer Engineering and Electrical Engineering from the University of Tennessee, Knoxville in 2003. In 2006, he received a MS degree in Computer Engineering from Washington University in St. Louis. He is continuing his education at the same institution and is currently a DSc candidate in Computer Engineering. His current research areas are FPGA design, HW/SW codesign, high-throughput computational biology, hardware design automation, and dynamically reconfigurable hardware techniques.

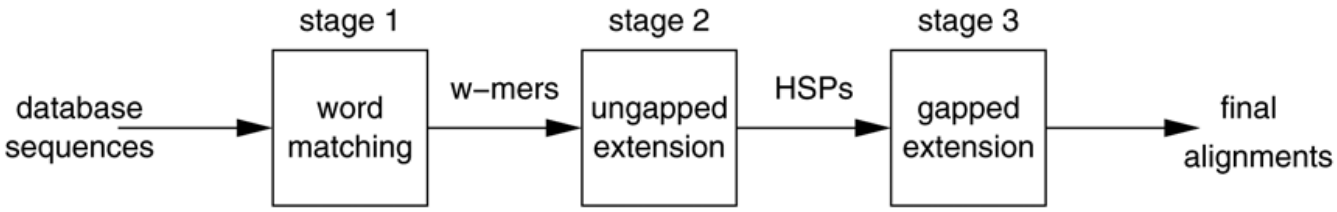

Figure 1.
NCBI BLAST pipeline.

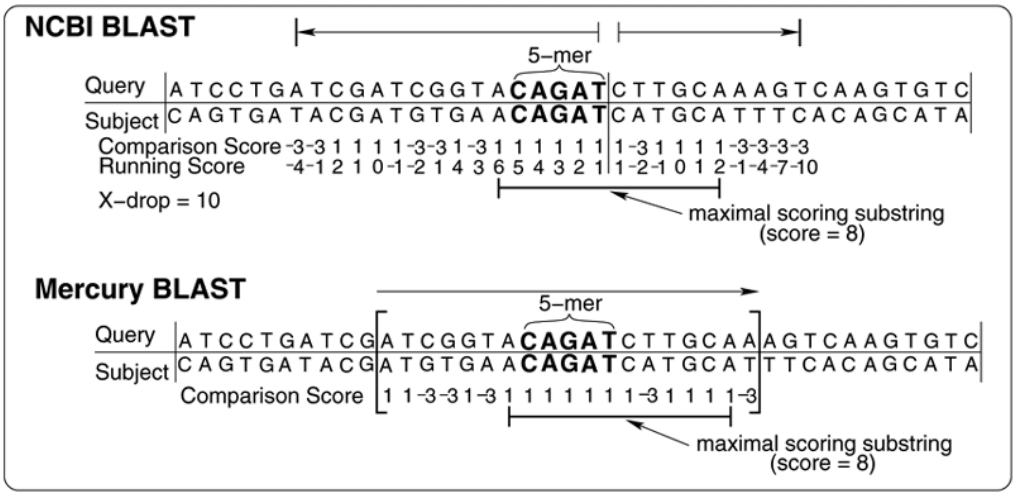

Figure 2.

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.

```
1  Extension(w–mer)
2      Calculate window boundaries
3      Γ = γ = 0
4      B = B_max = E_max = 0
5
6      for i = 1...L_w
7          if q_i = s_i
8              γ = γ + α
9          else
10             γ = γ − β
11         endif
12
13         if γ > 0
14             if γ > Γ and i > WmerEnd
15                 Γ = γ
16                 B_max = B
17                 E_max = i
18             endif
19         else if i < WmerStart
20             B = i + 1
21             γ = 0
22         endif
23     endfor
24
25     if Γ > T or B_max = 0 or E_max = L_w
26         return True
27     else
28         return False
29     endif
```

Figure 3.
Mercury BLASTN Extension algorithm pseudocode.

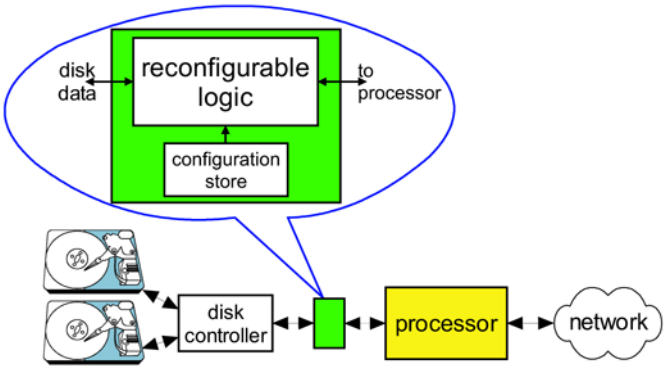

Figure 4.
Mercury system organization.

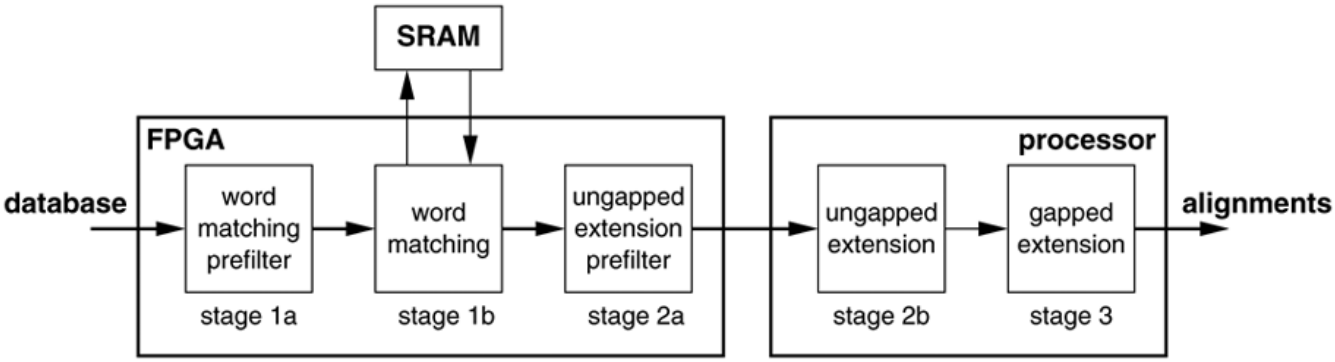

Figure 5.
Overview of Mercury BLASTN hardware/software deployment.

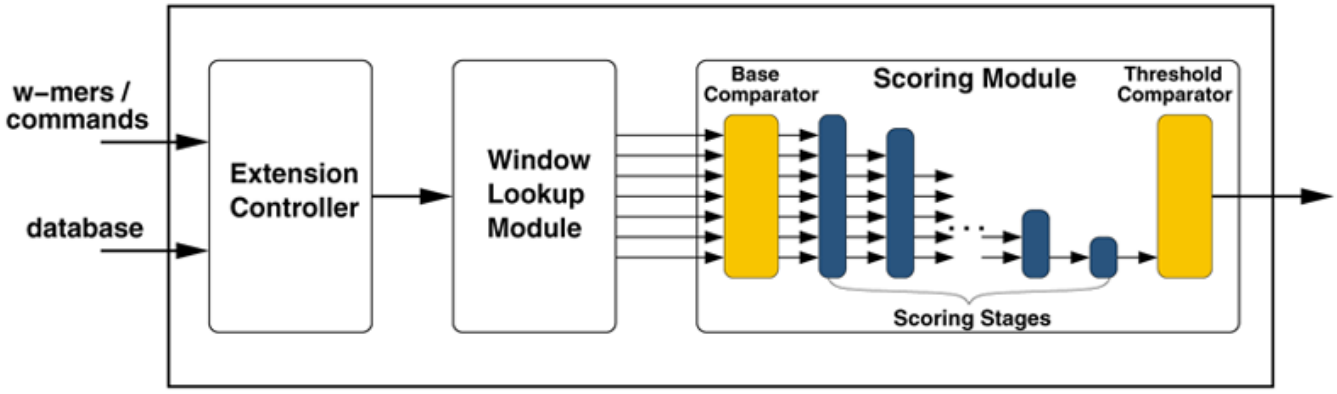

Figure 6.
Ungapped extension prefilter design.

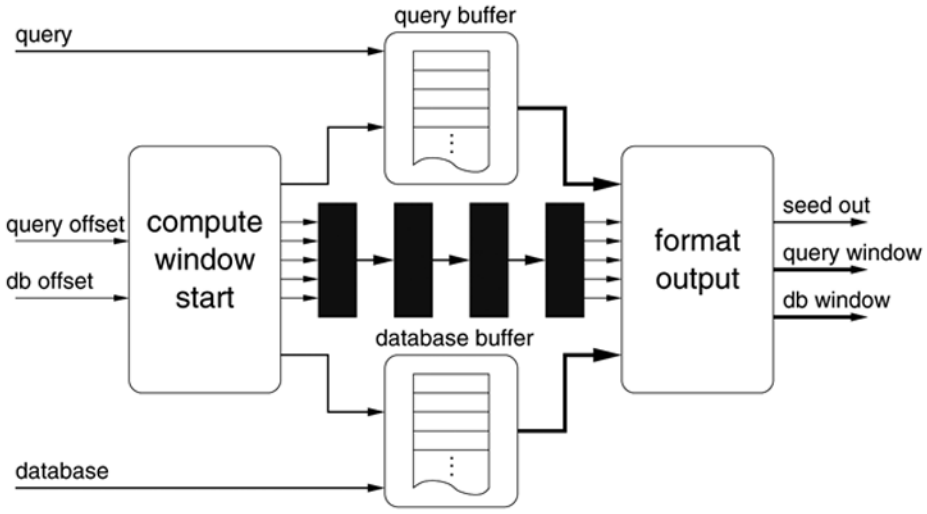

Figure 7.

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

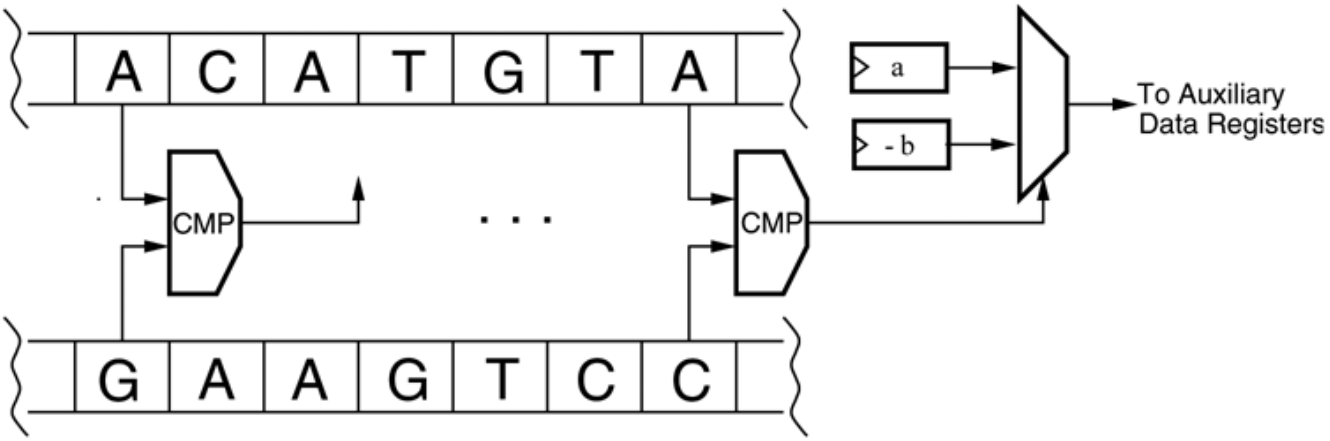

Figure 8.
Illustration of the base comparator for BLASTN. The base comparator stage computes the scores of every base pair in the window in parallel. These scores are stored in registers which are fed as input to the systolic array of scorer stages.

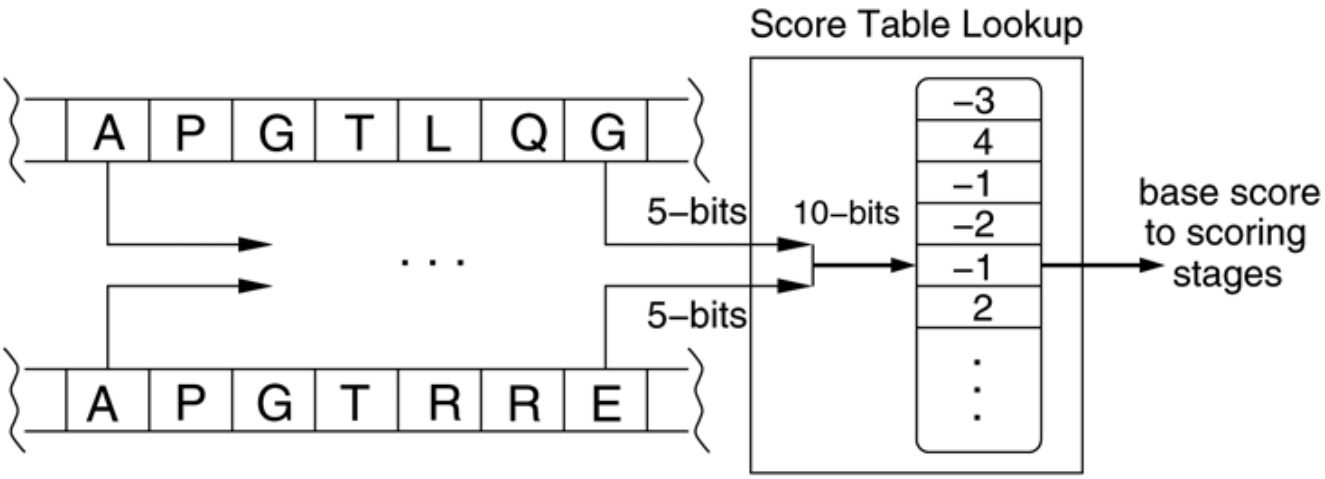

Figure 9.
Illustration of the first stage of the scoring for BLASTP. The amino acid pairs are used to index a set of parallel lookup tables that retain the pair score.

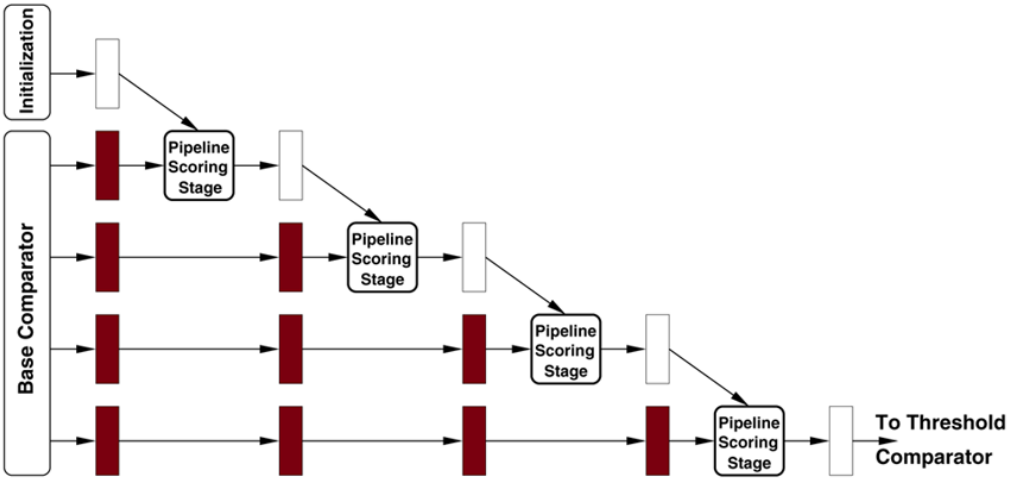

Figure 10.

Depiction of the systolic array of scoring stages. The dark registers hold data that 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 a different *w*-mer in the pipeline.

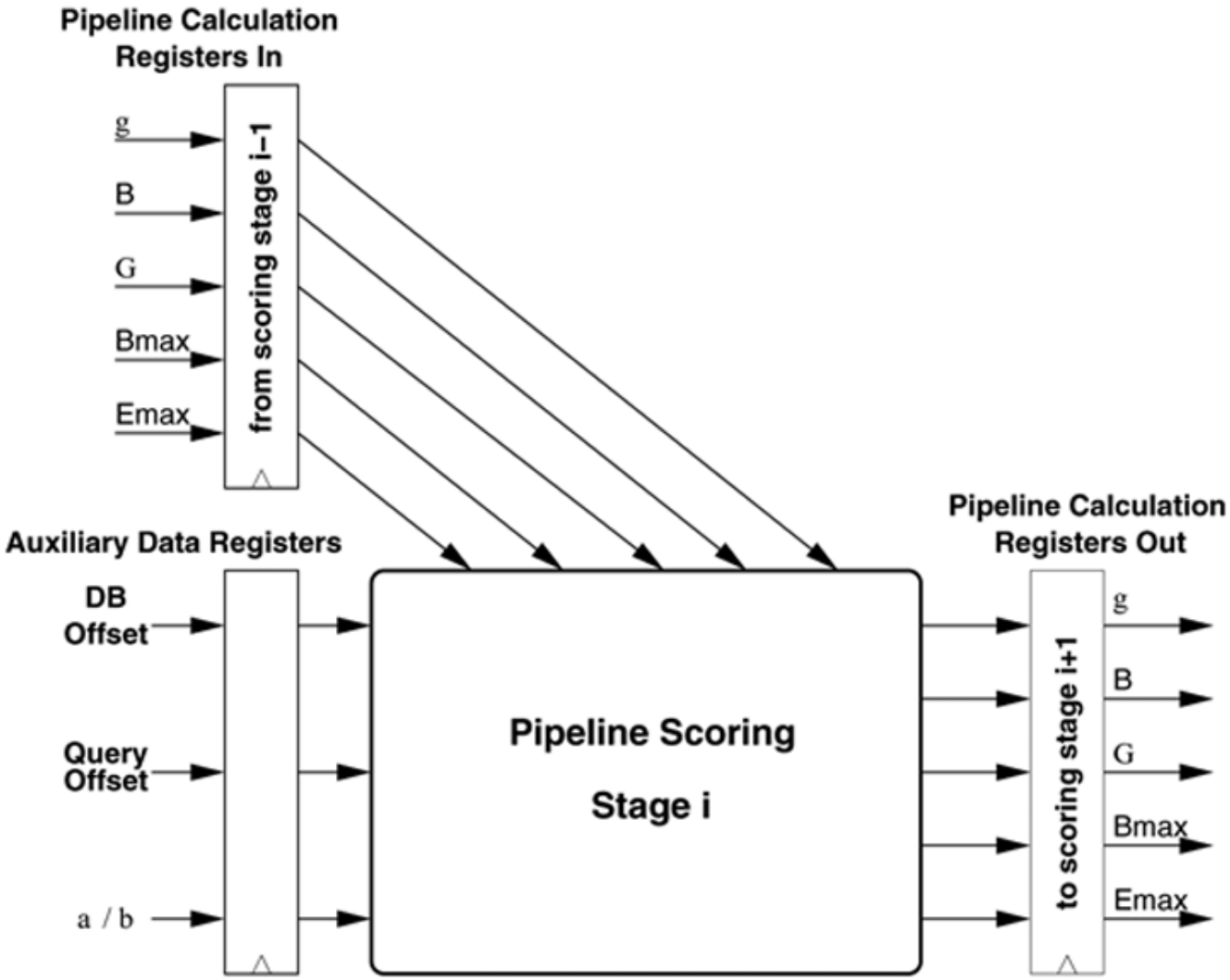

Figure 11.
Detailed view of an individual scoring stage.

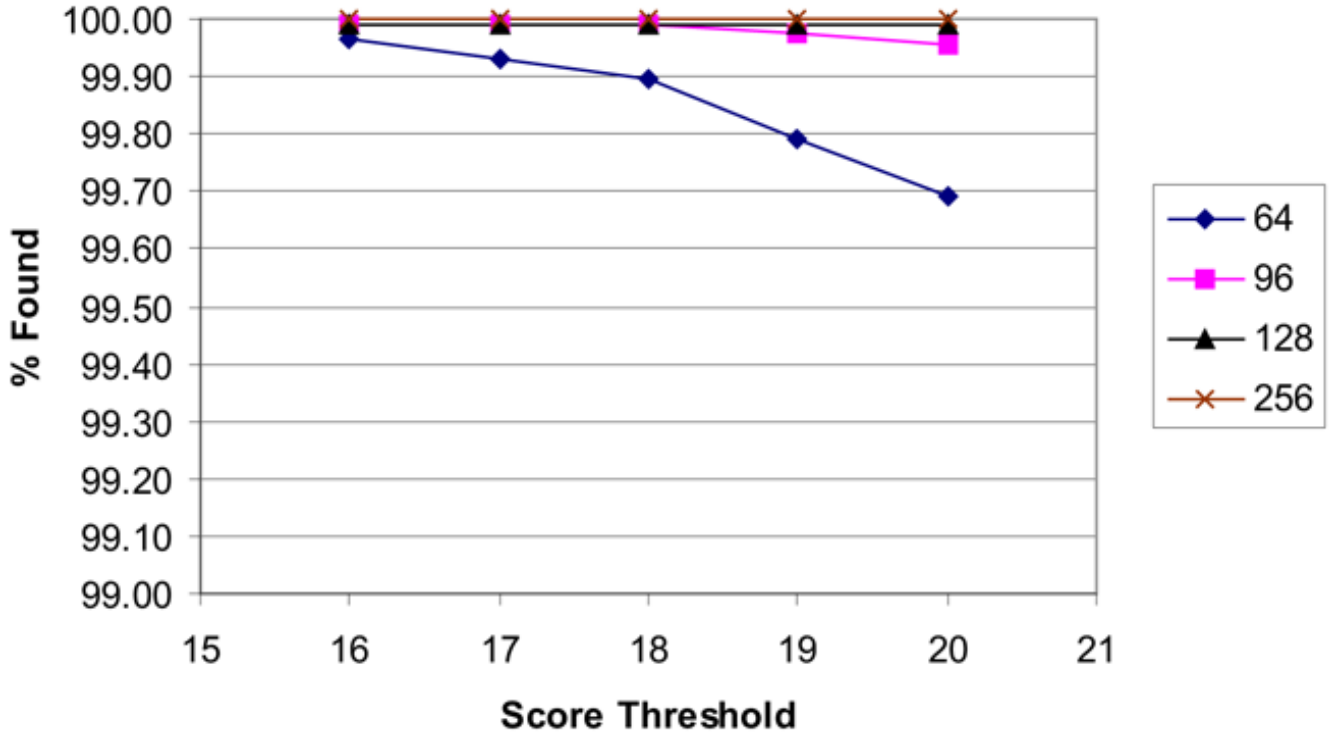

Figure 12.
Mercury BLAST sensitivity. The four curves represent window lengths of 64, 96, 128, and 256 bases.

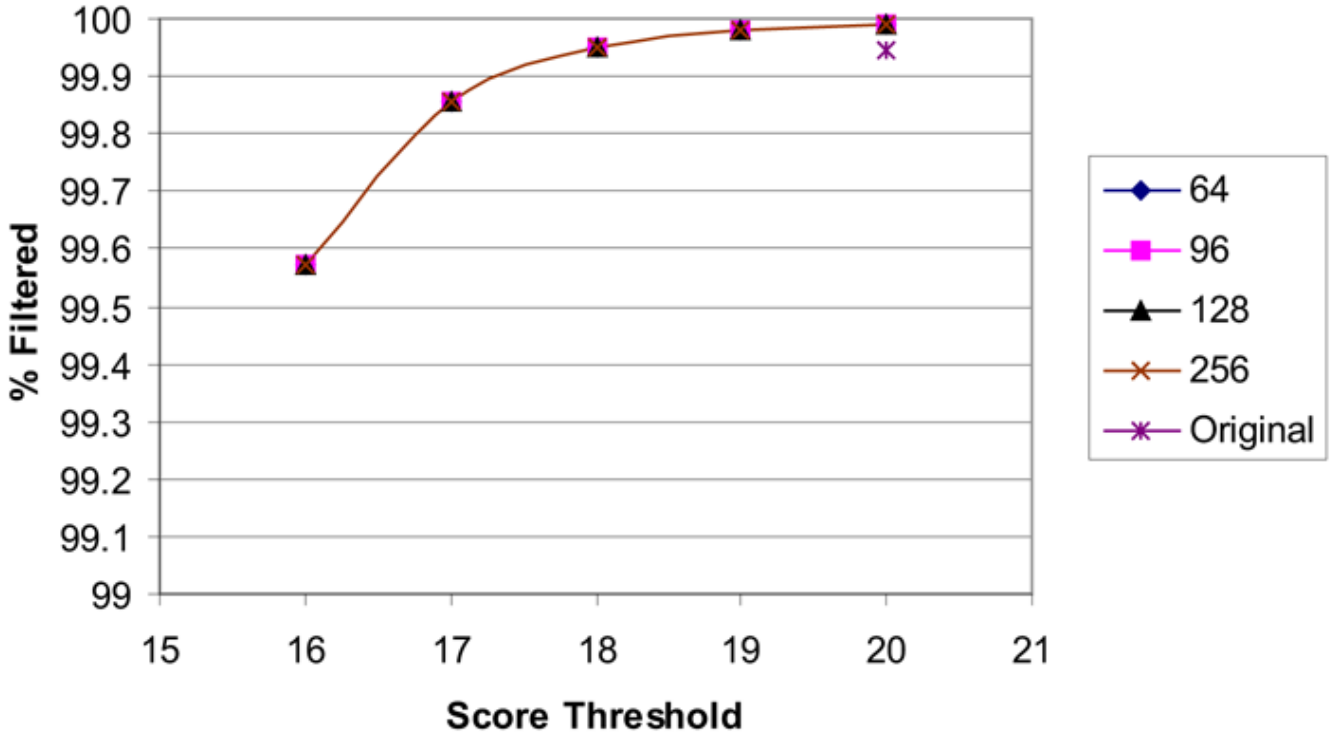

Figure 13.
Mercury BLAST specificity for stage 2a alone. The four curves represent window lengths of 64, 96, 128, and 256 bases. The individual point represents the value for NCBI BLAST stage 2.

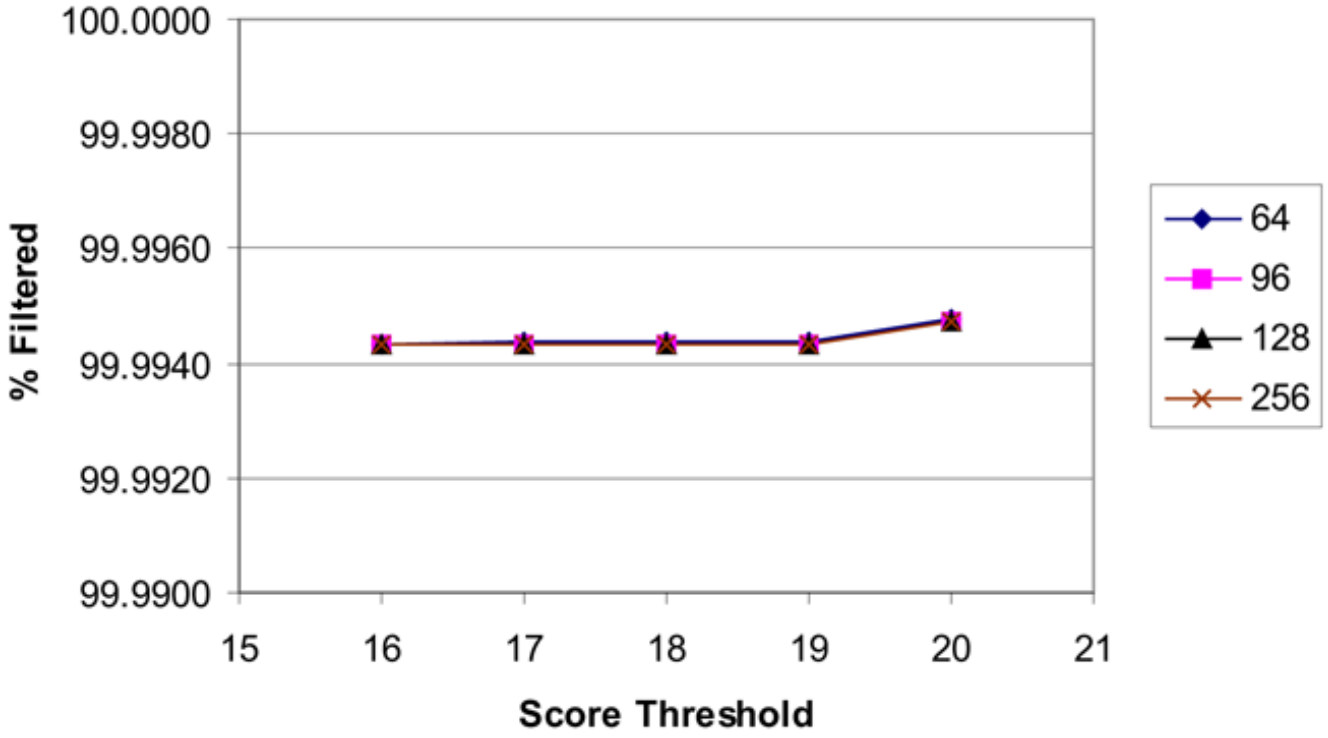

Figure 14.
Mercury BLAST sensitivity for complete stage 2. The four curves represent window lengths of 64, 96, 128, and 256 bases.

Table 1

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

Parameter	Value	Units	Meaning
p_1	0.0205	matches/base	stage 1 pass fraction [Krishnamurthy et al., 2004]
p_{2a}	0.0043	alignments/match	stage 2a pass fraction
p_{2b}	0.0133	alignments out/alignment in	stage 2b pass fraction
t_{2b}	0.265	μsec/alignment input	stage 2b execution time [Krishnamurthy et al., 2004]
t_3	60.4	μsec/alignment input	stage 3 execution time [Krishnamurthy et al., 2004]
$Tput_1$	1.4	Gbases/sec	stage 1 throughput [Krishnamurthy et al., 2004]
$Tput_{2a}$	4.9	Gbases/sec	stage 2a throughput
$Tput_{2b3}$	10.6	Gbases/sec	processor throughput
$Tput_{overall}$	1.4	Gbases/sec	overall pipeline throughput

Table 2

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

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