BIOINFORMATICS

Vol. 00 no. 00 2007
Pages 1–8

Mercury BLASTN: Faster DNA Sequence Comparison using a Streaming Hardware Architecture

Jeremy D. Buhler*, Joseph M. Lancaster, Arpith C. Jacob, and Roger D. Chamberlain

Department of Computer Science and Engineering, Washington University, St. Louis, MO 63130

ABSTRACT

Motivation: Large-scale DNA sequence comparison, as implemented by BLAST and related algorithms, is one of the pillars of modern genomic analysis. One way to accelerate these computations is with a streaming architecture, in which processors are arranged in a pipeline that replicates the multistage structure of the algorithm. To achieve high performance, the processor hardware implementing the critical seed matching and ungapped extension stages of BLAST should be specialized to execute these stages as quickly as possible. However, accelerating these stages requires solving two key problems: first, the seed matching stage is not of a form which has traditionally been amenable to hardware acceleration; and second, the accelerated implementation of BLAST should retain sensitivity at least comparable to that of the original software.

Results: We describe Mercury BLASTN, an FPGA-based implementation of BLAST for DNA. Mercury BLASTN combines a Bloom filtering approach to seed matching with a modified ungapped extension algorithm to overcome barriers to placing the early stages of BLAST onto hardware. On a previous-generation FPGA hardware platform, Mercury BLASTN runs 5 to 11 times faster than NCBI BLASTN current-generation general-purpose CPUs, with the prospect of a further eight-fold speedup on current-generation FPGAs. Moreover, its sensitivity to significant DNA sequence alignments is 99% of that observed with software NCBI BLASTN.

Availability: Academic users should contact the authors for information on acquiring a prototype of the Mercury BLASTN system.

Contact: jbuhler@cse.wustl.edu

1 INTRODUCTION

Fast biological sequence comparison is a crucial technology for modern molecular biology. High-throughput comparison is the preferred way to annotate functional elements in a newly sequenced genome, as well as the basis of tools that discover correspondences between related genomes [3, 5, 6]. Because of the exponential growth of biosequence databases such as NCBI's GenBank [21] and the growing numbers and sizes of genomes used for comparative analysis, the sequence comparison problem, though extensively studied, remains a computational bottleneck for genomics.

Efficient heuristics based on seeded alignment [23], in particular the BLAST family of algorithms [1, 2, 27], today dominate large-scale biosequence comparison. Because of these methods' long track records and acceptance by researchers, extensive effort has been invested to improve their performance. These efforts include multiprocessor implementations [10, 20, 26] that parallelize comparison across a collection of identical processing elements, ideally achieving performance proportional to the number of processors used. However, building a high-performance multiprocessor implementation of seeded alignment demands careful load balancing and communication management and requires each processing element to implement a complex, multistage seeded alignment computation.

In contrast to the multiprocessor approach, a *streaming* implementation of seeded alignment organizes processors into a linear pipeline. As data streams through the pipeline, the processors for each pipeline stage perform one part of the computation, sending their results to the following stage. This simple architecture avoids many of the complexities of multiprocessors. Streaming implementations achieve high computational throughput by running all stage processors in parallel. Moreover, since each processor implements only one pipeline stage, rather than the whole computation, its architecture may be specialized to perform that stage as efficiently as possible. Specialization can practically be achieved by implementing the stage processors in reconfigurable hardware, such as field-programmable gate arrays (FPGAs). With the right architecture, speedups of one to two orders of magnitude can be realized relative to sequence comparison performed in software alone.

Most previous streaming architectures for biosequence comparison [12, 13, 28, 33] have focused on accelerating the classic Smith-Waterman dynamic programming algorithm [29]. Unfortunately, using Smith-Waterman *instead* of seeded alignment, even with today's fastest reported FPGA implementations, is not performance competitive with software BLASTN on a modern CPU. Moreover, Smith-Waterman is not the computational bottleneck for large genomic DNA comparisons; rather, the bottleneck is in *seed matching*, the initial search to detect which regions of a query sequence and a database are similar enough to investigate further. An orthogonal issue with previous architectures is that, regardless of how they are implemented, few have been benchmarked for their sensitivity relative to software BLASTN. Hence, there is little reason for BLAST users to trust the results produced by these systems. Overcoming all these issues is essential to produce an accelerator that is both competitive in performance and acceptable to biologists.

This work reports the development of Mercury BLASTN, a streaming implementation of a BLAST-like algorithm for DNA sequence comparison. Unlike most previous streaming accelerators for biosequence comparison, Mercury BLASTN uses FPGA hardware to accelerate the performance-critical early stages of seeded alignment, while using the standard NCBI BLASTN software for later, non-bottleneck stages. On large DNA sequence comparisons, Mercury BLASTN on previous-generation FPGA hardware runs 5 to 11

*to whom correspondence should be addressed

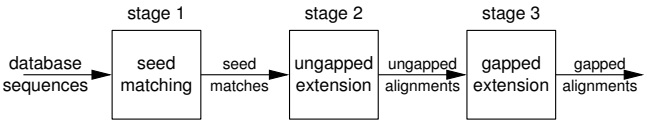

Fig. 1. Software NCBI BLASTN functional pipeline.

times faster than NCBI BLASTN software on a current-generation general-purpose CPU, while delivering results 99% identical to those from the software. On FPGA hardware available today, we anticipate a further eight-fold speedup. Our successful approach may be used to accelerate a large class of biosequence comparison tools based on seeded alignment. More concretely, a single Mercury BLASTN system is competitive with a medium-size multiprocessor for high-performance BLASTN tasks.

The rest of this paper is organized as follows. Section 2 briefly reviews the structure of the BLASTN algorithm, details the rationale for our design, and reviews related work. Section 3 then describes Mercury BLASTN, including the techniques used to specialize processing units to its various pipeline stages. Section 4 compares the speed and sensitivity of Mercury BLASTN to that of the standard NCBI BLASTN software on large genomic BLAST computations. Finally, Section 5 concludes with plans for future improvements and generalization of our tool.

2 BACKGROUND AND DESIGN RATIONALE

The BLASTN algorithm compares one or more *query* DNA sequences to a database of other DNA sequence. Like most seeded alignment tools, it is logically organized as a pipeline of computational stages, as shown in Figure 1. The first stage, *seed matching*, recognizes short patterns, or seed matches, that appear in both query and database. It emits all such co-occurrences (i, j), where i and j respectively indicate a seed match's location in the query and database. BLASTN's second stage, *ungapped extension*, determines whether there exists a strong ungapped sequence alignment, or HSP (high-scoring segment pair), between the query and database in the vicinity of the seed match at diagonal offset $j - i$. Finally, *gapped extension* determines whether a high-scoring gapped alignment exists in the vicinity of each HSP.

A streaming implementation of BLASTN uses a pipeline of processors that reflects the structure of Figure 1. Each stage of the computation is allocated to its own processor. Communication is via a simple, unidirectional stream: each processor takes input from its predecessor and sends results to its successor in the pipeline. If all processors are equally powerful, then the speed of the implementation is determined by the most computationally loaded processor, which may become a bottleneck. We must therefore evaluate which stages of BLASTN are most heavily loaded and ensure that these stages receive the computational resources needed for high performance.

For large DNA comparisons, BLASTN's computational load is heavily biased toward the early stages of its pipeline. For example, in a comparison of the human and mouse genomes, we observed [16] that NCBI BLASTN spends roughly 50% of its time in seed matching and 50% in ungapped extension, with negligible time in gapped extension. We therefore focus on accelerating the parts of our streaming architecture devoted to these first two stages of BLASTN.

Our approach to accelerating the bottleneck stages of BLASTN is to specialize the architectures of these stages to perform their tasks more efficiently. To create specialized processors, we turn to field-programmable gate arrays, which are a fast, inexpensive option for implementing specialized architectures. A computing architecture can be specified using standard hardware description languages, then compiled to run on a particular family of FPGA, without the need for expensive circuit fabrication. Moreover, a single FPGA, like a general-purpose CPU, can be reused for different computations by simply reprogramming it with new architectures. Although FPGAs today are typically clocked an order of magnitude more slowly than conventional CPUs, they can yield much higher performance provided that the tasks to be accelerated can be formulated as a large number of parallel logical or arithmetic operations. For such tasks, a modern FPGA can outperform software on a general-purpose CPU by an order of magnitude or more, while consuming much less power and generating much less heat than a multiprocessor of equivalent performance.

Our main contribution in this work is to show how to modify the bottleneck stages of the BLASTN algorithm to exploit the power of an FPGA, while producing results substantially identical to those of the standard NCBI BLASTN software. In particular, we show how to speed up the critical seed matching stage, which does not use dynamic programming and so is not helped by techniques previously used to accelerate Smith-Waterman. We also show how to reimplement BLASTN's own ungapped extension algorithm as an efficient hardware filter that prevents most fruitless extensions from reaching the software.

2.1 Related Work

Parts of the Mercury BLASTN architecture were previously described in [16, 17]. However, this work is the first description of the completed system and its performance.

Much previous work describes hardware accelerators for the Smith-Waterman algorithm [12, 13, 28, 33]. Commercial implementations of such accelerators include the (now defunct) Paracel GeneMatcher and Compugen Bioaccelerator. Although these accelerators can outperform Smith-Waterman on a general-purpose CPU by up to 100-fold, their rate of comparisons is much less than that of a BLASTN accelerator. Published designs compute fewer than 10^{10} Smith-Waterman cells per second; for the 17.5-kbase query size used by Mercury BLAST, these designs can compare the query to fewer than 10^6 bases per second from the database. In contrast, Mercury BLAST consumes roughly 10^9 database bases per second.

Several other accelerator designs have been described for BLAST-like computations. RDisk [18] is an FPGA-based design that reports a throughput of 60 Mbases/sec for stage 1 of the BLASTN pipeline. However, they do not implement a full pipeline and so do not report either end-to-end performance or sensitivity compared to software. Another FPGA design, TUC BLASTN [30] reports a 23-fold speedup over a Pentium workstation, using an FPGA with twice the logic and four times the on-chip memory of our hardware. However, this performance is predicted, not measured, and no sensitivity estimates are given relative to software. Herbordt *et al.*[11] report an FPGA-based BLASTP accelerator. Finally, the TimeLogic DeCypher engine is a commercial accelerator that offers a BLASTN replacement; unfortunately, we lack sufficient information to compare its performance fairly to Mercury BLASTN.

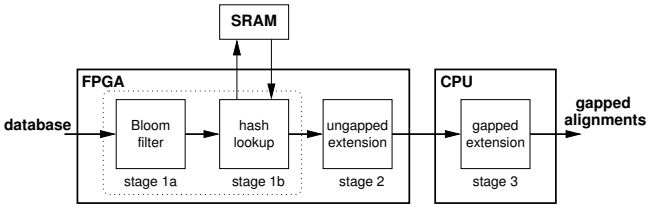

Fig. 2. Deployment of BLASTN onto the Mercury system. The dotted box denotes the sub-stages that make up stage 1.

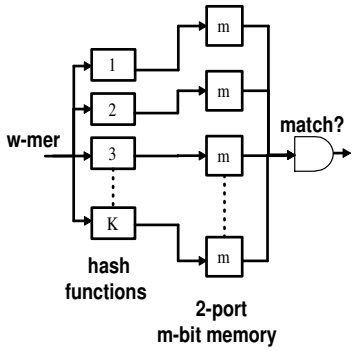

Fig. 3. Bloom filter *test* operation.

Multiprocessor versions of BLASTN distribute the query sequences, the database, or both across a number of identical processors. They require careful tuning to avoid communication bottlenecks, which occur when the overhead of distributing inputs, collecting outputs, and managing the computation prevents processors from carrying out the core BLASTN operation. Failure to overcome this issue leads to much worse than linear speedup with the number of processors; for example, SGI's HT BLAST [8] reports only a 12-fold speedup on a cluster of 64 CPUs. Examples of well-tuned multiprocessor BLASTs include mpiBLAST [10, 20] and BlueGene/L BLAST [26], which scale to over 1000 processors.

A streaming implementation of BLASTN avoids much of the complex communication of a multiprocessor. However, to achieve large speedup, a streaming implementation of BLASTN must execute the rate-limiting stages of the pipeline significantly faster than the general-purpose CPUs in a multiprocessor. Our work demonstrates that such acceleration is feasible and practical. A hybrid between streaming and multiprocessing is also possible, in which an array of streaming engines process different queries against a single, common database stream. Such a system design, which has been used in, e.g., the TimeLogic DeCypher engine, can achieve high scalability with a fraction of the processors needed by a general-purpose cluster.

A number of algorithmic improvements to BLASTN-like tools have been proposed to increase search speed. MegaBLAST [34], SSAHA [22], and BLAT [14] achieve speedups of an order of magnitude or more over BLASTN. However, the longer seed lengths and other heuristics used by these tools diminish their sensitivity relative to BLASTN, causing fewer significant alignments to be returned. In contrast, PatternHunter II [19] and DASH [15] report both higher speed and greater sensitivity than BLASTN. PatternHunter reports only a 2-fold speedup. DASH reports at most a 10-fold speedup for small queries (1500 bases or less), but it is unclear whether the same performance benefit would hold for the longer query sizes, tens of kilobases or more, that are used in large-scale genomic DNA and mRNA comparisons. In summary, speeding up BLASTN in software appears to involve a tradeoff: one can either have order-of-magnitude speedup *or* BLASTN-equivalent sensitivity, but achieving both at once is elusive. Mercury BLASTN's goal is to preserve as closely as possible the sensitivity of software BLASTN, while still delivering a large speedup on queries tens of kilobases in length.

3 METHODS

Mercury BLASTN is built on the *Mercury* system [9], a platform for processing high-speed data streams. The platform is concretely realized as a workstation containing a PCI-X add-in card, containing an FPGA co-processor, and a pair of AMD Opteron CPUs. Inputs from disk or system RAM flow over the PCI-X bus to the FPGA, while outputs flow from the FPGA back to the host CPUs for software post-processing.

Figure 2 shows a block diagram depicting the various stages of Mercury BLASTN. The query sequence is first loaded on-chip, after which the database is streamed through the FPGA in a single pass. The FPGA performs seed matching (stage 1) and a limited form of ungapped extension (stage 2) and returns ungapped alignments to the CPUs, which perform more complete ungapped extension and gapped extension (stage 3). The stages on the FPGA eliminate a large fraction of their input, removing most of the load from the host CPUs and allowing them to keep pace.

The next two sections describe the architecture and algorithms of the first two stages. More detail is available in [16, 17].

3.1 Stage 1 - Seed Matching

The seed matching stage of BLASTN is typically implemented in software as a table lookup: all contiguous words in the query of some length w (hereafter called w-mers) are hashed into a table in memory, and each w-mer in the database is looked up in the table to find seed matches. This scheme is rate-limited by the cost of doing a table lookup for each position (or each few positions) in the database.

Mercury BLASTN's stage 1 implements a table-based seed matching algorithm similar to BLASTN (and quite distinct from traditional dynamic programming-based architectures). Unfortunately, the tables used are too large to fit on an FPGA, so we cannot simply eliminate references to external memory. Instead, Mercury BLASTN employs a filtering scheme on-chip that nearly eliminates unsuccessful table lookups, which constitute 99% or more of the lookups performed in software for DNA queries of 20 kilobases or less.

Our implementation of seed matching is divided into two substages (Figure 2, labeled stages 1a and 1b). A Bloom filter stage (1a) rapidly tests the input database stream on-chip to identify all w-mers in the database that will hit in the table, plus a small number of false positives. The hash lookup stage (1b) then performs the

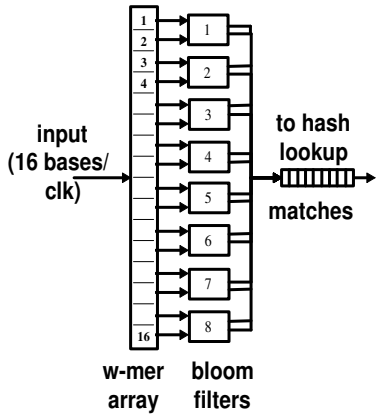

Fig. 4. Bloom filters as used in stage 1a.

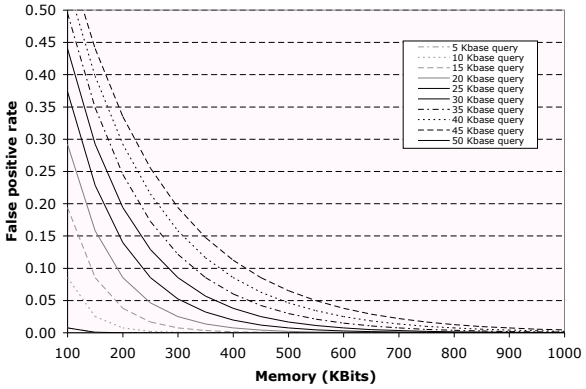

Fig. 5. Theoretical false positive rate of a Bloom filter vs. memory size for different query lengths

lookups and retrieves the matching positions in the query for each database w-mer. This architecture greatly reduces the memory bandwidth bottleneck and so permits meaningful acceleration of stage 1 over software alone.

3.1.1 Bloom filters as a database prefilter. A *Bloom filter* [4] is a probabilistic algorithm used to test membership in a large set, where multiple hash functions each look up an independent array of space-efficient bit-vectors. A set of query w-mers are *programmed* by setting the bit at the memory location indicated by each hash function. A w-mer *test* operation yields a *match* when all memory locations identified by the hash functions are set. This operation is illustrated in Figure 3; here k hash functions are used to test an identical number of m-bit-vectors.

A Bloom filter produces no false negatives but may generate false positives at a rate f determined by the number of w-mers programmed into it and the length of its memory vector. The rate f can be modeled as $f = (1 - e^{-Nk/m})^k$, where N is the number of entries programmed into the filter (query size). Figure 5 shows the false positive rate of a Bloom filter as a function of memory size for different query lengths. We design our filters to keep the false positive rate to at most a few percent for the query sizes of interest, thereby eliminating 98-99% of memory table lookups that would otherwise be necessary.

A Bloom filter stage is ideally suited for a hardware rather than a software implementation due to its inherent parallelism, and it can be implemented efficiently using on-chip memory for the filters. We use $k = 6$ hash functions of the hardware-friendly H_3 family [25], computed in parallel, each of which references a bit-vector implemented using two on-chip block RAMs. Since block RAMs on our FPGA are dual-ported, each bit-vector is shared between two input w-mers. Using a bit-array of size $m = 32768$, query sequences between 10-15 kbases can be supported in each pass with a negligible false positive rate. Finally, we replicate the setup in Figure 3 eight times to support sixteen parallel w-mer test operations as shown in Figure 4. Using this setup, we are able to process the database at roughly 10^9 bases/second, as quickly as it can be delivered to the FPGA over the PCI-X bus.

Up to 16 database w-mers can be processed in a single clock cycle in this stage, matching on average one w-mer to the query sequence. Serialization of matched w-mers is done using a 16-to-1 reduction tree, with buffering to hold bursts of matches from the same group of 16.

3.1.2 Hash Lookup. The second substage of stage 1 maps a database w-mer to its query position(s) using a hash table. The hash table is implemented in an external SRAM attached to the FPGA, supporting a single lookup per clock cycle. Despite the filtering performed by the Bloom filter in stage 1a, the SRAM remains a potential bottleneck for our design if the hashing scheme chosen requires multiple table probes per w-mer searched. Our design, which uses a variant of Tarjan and Yao's displacement hashing [31], balances the need for as few probes (ideally just one) per w-mer as possible with the limited table size possible with our SRAM, and with the need to compute the hash functions efficiently on-chip at a high rate.

We organize our lookup table into *primary* and *secondary* tables. To create a mapping from a set S of query w-mers, each consisting of $2w$ bits, into the primary table of size 2^a for some fixed a, we construct a hash function h_1 of the form

$$h_1(s) = A(s) \oplus \tau[B(s)],$$

where $A : \{0,1\}^{2w} \rightarrow \{0,1\}^a$ and $B : \{0,1\}^{2w} \rightarrow \{0,1\}^b$ are easily-computed H_3 hash functions of the w-mer s, $\oplus$ is the bitwise XOR operation, and τ is a *displacement table* of 2^b small integers. The functions A and B are chosen so that the pairs $(A(s), B(s))$ are distinct for all keys $s \in S$. The displacement table can then be exploited to resolve collisions as follows: if w-mers s_1 and s_2 have $A(s_1) = A(s_2)$ but $B(s_1) \neq B(s_2)$, try to choose distinct values for $\tau[B(s_1)]$ and $\tau[B(s_2)]$ to ensure that $h_1(s_1) \neq h_1(s_2)$. We set $w = 11$, we set $a = 17$, $b = 10$, and permit each entry of τ to be up to 8 bits; these parameters were chosen empirically so as to produce almost no collisions for query sequences of lengths 10-15 kbases. We therefore map w-mers to a table of 2^{17} entries in SRAM, using a modest 8 kbits of space on-chip for τ.

Although our hash function produces few collisions on the inputs of interest, it is not a perfect hash – the known constructions for such hashes require tables larger than our available SRAM size and/or hash functions that are not amenable to computation on an FPGA. To resolve collisions, we map all w-mers in the query that collide in our primary table to the secondary table, using a second hash function h_2 of the form $h_2(s) = C(s)$, where $C : \{0,1\}^{2w} \rightarrow$

$\{0,1\}^c$. The secondary table contains 2^{16} entries in our implementation. This table is sparsely populated, so it resolves essentially all collisions from the primary table in one additional probe.

A database w-mer may occur at more than one position in the query sequence. We record the positions of such w-mers in a *duplicate* table, which occupies part of our SRAM. Accesses to the duplicate table are only needed when a w-mer occurs at least twice in the query (indicated by a bit in its hash table entry), and each such access can return up to three query positions.

Using an SRAM of size 1 MB, our hashing scheme nearly always produces a perfect primary hash for 10-15 kbase query sequences, achieving one probe per w-mer. With the additional support of the secondary table, query sequences up to 17.5 kbases are supported without seriously degrading hash lookup performance. The time taken to generate the hash function and the associated tables on our host CPU is well under a second per query.

3.2 Stage 2 - Ungapped Extension

Ungapped extension of a seed match must decide, as quickly and accurately as possible, whether or not the seed forms part of a high-scoring alignment of the query and database. Because almost all seed matches produced by BLASTN stage 1 do *not* occur in such alignments, ungapped extension is a strong filter that prevents the more expensive gapped extension stage from being overloaded.

As a filter, ungapped extension must carefully control both false positives, so as to not negatively affect the speed of gapped extension, and false negatives, so as not to affect the sensitivity of the pipeline. NCBI BLASTN implements a linear-time dynamic programming heuristic that does a good job of balancing these often opposing goals. Unfortunately, although accelerating dynamic programming is one of the strengths of FPGAs, other aspects of the heuristic are less compatible with an efficient and sensitive hardware implementation. We therefore developed a simplified stage 2 design that serves as a filter in front of the original NCBI BLASTN ungapped extension software stage and removes most of the computational burden from that stage.

In what follows, we first review the algorithm used by NCBI BLASTN, then describe our variant algorithm and how it is implemented in hardware.

3.2.1 NCBI BLASTN Ungapped Extension. NCBI BLAST's extension of a seed match into an ungapped alignment proceeds in two stages. The seed match is extended backwards toward the beginning of the two sequences, then forward towards their ends. As the algorithm extends over each base pair, that pair receives a reward $+\alpha$ if the bases match or a penalty $-\beta$ if they do not match. An ungapped alignment's score is the sum of these rewards and penalties over all its pairs. The beginning and end of the ungapped alignment is chosen to maximize its total score. If the final ungapped alignment scores above a user-defined threshold, it is passed on to gapped extension.

To maintain efficiency, it important to terminate ungapped extensions long before reaching the ends of the query and database sequences, especially if no high-scoring ungapped extension is likely to be found. NCBI BLASTN implements early termination by an *X-drop* mechanism. The algorithm tracks the highest score achieved by extension thus far; if the current extension scores at least X below this maximum, further extension in that direction is terminated. This mechanism allows BLAST to recover ungapped extensions of arbitrary length while simultaneously limiting the cost of extension for most seed matches.

The X-drop strategy used by the NCBI BLASTN software poses a challenge for hardware implementation because it does not *a priori* limit the size of the region explored by extension. Trying to reproduce this algorithm on an FPGA would necessitate maintaining a very large window of the database on-chip to handle the worst-case extensions. This window would go largely unused for most extensions, almost all of which terminate within a few bases of the seed's endpoints.

3.2.2 Mercury BLASTN Ungapped Extension. Mercury BLASTN takes a different, more FPGA-friendly approach to ungapped extension. We reexamined NCBI BLASTN's heuristic and developed a modified algorithm with limited resource usage, and therefore greater hardware efficiency.

Instead of performing extension in two steps, Mercury BLASTN makes a single forward pass over a fixed-size window anchored around the seed match. In other words, we explicitly set the number of pairs inspected for each seed match instead of allowing a flexible termination scheme like that of NCBI BLASTN. In a single pass through the window, the start and end of the best ungapped alignment in the window are found by our algorithm. If this alignment scores above a desired threshold (which in general is lower than that used by the BLASTN software), it is passed on to a software stage that implements the full BLASTN ungapped extension algorithm. This compromise implementation eliminates nearly all false positives, i.e. all seed matches that do not yield strong ungapped alignments, before they reach software, while permitting arbitrarily high-scoring extensions to be found and scored accurately.

We modify the dynamic programming recurrence used for ungapped extension to impose two additional constraints. First, the ungapped alignment must pass through the seed match. This ensures that, if two distinct biological features appear in a single window, a seed match generated from each has the possibility of generating two independent ungapped alignments. Otherwise, the higher-scoring feature in the window would be found twice. Second, if the best ungapped alignment intersects either the beginning or end of the window, it is passed on to later stages regardless of its score. This heuristic ensures that ungapped alignments that might extend well beyond the window boundaries are properly found by downstream stages.

Figure 6 illustrates the differences between the two methods. Both algorithms give the same result in this example, but this is not necessarily the case in general. A large number of experiments were run with an instrumented version of NCBI BLASTN to determine reasonable minimum values for a window length that still met our sensitivity goal of 99%. A window length of 64 bases is the one used in this accelerator; however, it is parameterizable if other lengths are needed.

3.2.3 Implementation Details. Figure 7 shows an overview of the architecture of the ungapped extension accelerator. The query is stored in on-chip block RAM, while the database flows through an on-chip circular buffer. As each seed match arrives at the stage's input, a window of bases centered on the seed is extracted from this buffer by the *window lookup module* and passed on to the dynamic programming hardware.

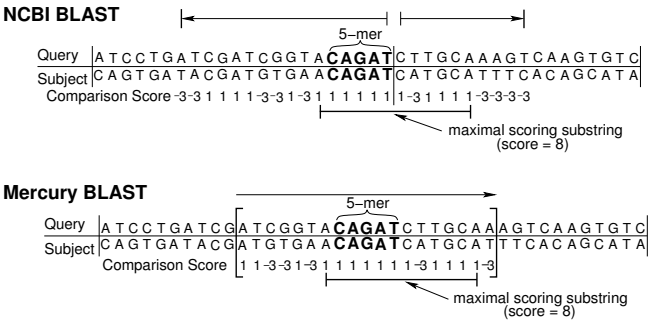

Fig. 6. Example illustrating the different approaches to ungapped alignment. The arrows indicate the direction in which the base pairs are inspected.

A potential limit on the implementation of this stage is the number of simultaneous accesses required to the buffered query and database sequences. Because on-chip block RAMs support a limited number of accesses per cycle (at most two in our hardware), multiple copies of the data must be kept to satisfy all consumers in the design. To reduce the amount of on-chip RAM needed for buffering, the block RAMs in this stage are time-multiplexed to allow access from four users in a single clock cycle. Quad-porting the block RAMs halves the number of physically dual-ported memories required for this application. With this optimization, the stage supports query sizes up to 64 kbases and a database buffer of 32 kbases.

The ungapped extension algorithm in Mercury BLASTN is implemented as a pipelined systolic array. Saturating arithmetic was used both to shorten the critical paths of the compute logic and to reduce area usage. This improvement is possible since the reduction in the number of bits used to represent the recurrence variables outweighs the increased complexity of using saturating arithmetic.

The first stage of the array, the base comparator, computes the reward or penalty score for each base pair. Since these scores are all independent, they are computed in parallel. Next, the scores for the window are passed to the scoring stages shown in Figure 7. Each scoring stage implements two steps of the recurrence, for a total of 32 scoring stages for the current window size. The final step compares the score to a threshold and makes a decision whether to keep or discard the scores. Note that the number of arrows between scoring stages decreases as along the pipeline in Figure 7. This is due to the fact that each base pair is only inspected once and the new algorithm only moves in one direction, so that the unneeded data can be discarded after it has been used. If an ungapped alignment scores above the threshold (or meets the other two conditions mentioned in the previous discussion) it is sent to software for gapped extension.

3.3 Integration with Software

Mercury BLASTN is a combined hardware-software architecture. The software portion of the system uses a heavily augmented version of NCBI BLASTN. The BLASTN software was modified to pass its inputs to a software library that sets up the query and associated tables, downloads it to the hardware, and sends a specially formatted copy of the database through the hardware to execute the search. Results from the hardware are collected and passed on to the unmodified NCBI BLASTN ungapped and gapped extension code, just as if these results had come from BLASTN's own seed matching stage. Query setup, downloading to the hardware, receipt of

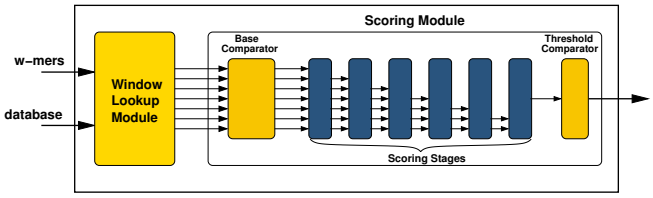

Fig. 7. Overview of stage 2 architecture.

results, and computation in later stages of NCBI BLASTN are all performed concurrently by different threads of execution within the modified BLASTN program. The actual running of NCBI BLASTN stages 2 and 3 requires only about 5% of one Opteron CPU.

To efficiently support comparisons on many small queries, Mercury BLASTN can pack these queries into chunks as large as the hardware will support (currently 17.5 kbases), using a fast approximation algorithm for the bin-packing problem. Very large queries are split into chunks of at most this size, with some overlap to avoid losing alignments near a chunk boundary. In each case, searching each chunk requires one pass over the entire database.

To the user, the Mercury BLASTN system appears nearly identical to regular NCBI BLASTN, with the same user interface. The system includes a modified version of the `formatdb` utility to produce the database format required by the hardware.

4 RESULTS

The goal of Mercury BLASTN is to deliver NCBI BLASTN-quality sensitivity in a fraction of the time needed by a general purpose CPU. In this section, we quantify how well Mercury BLASTN achieves these aims on two large-scale DNA comparisons typical of the kinds of large BLAST runs that might be performed for genome annotation.

Our tests pitted Mercury BLASTN against a modern, general-purpose Linux workstation containing a 3.0 GHz Pentium D CPU and 1.5 GB of RAM, running 64-bit Linux. We built a recent version of NCBI BLAST (v2.2.15) for this platform, using all available compiler optimizations of gcc 3.4. (We note that this version of BLAST uses a new, extensively rewritten implementation of the core algorithms that is roughly 2.5x faster than versions of BLASTN released prior to 2005.) BLASTN runs were performed single-threaded on one core of the CPU on an otherwise idle system.

Our Mercury BLASTN system contains two 2.0 GHz AMD Opteron processors and one FPGA co-processor prototyping board. The FPGA is a Xilinx Virtex-II 6000 part (XC2v6000) and has SRAM connected onboard. The Mercury BLASTN software is built around an older version of NCBI BLASTN (v2.2.10), also running on 64-bit Linux (CentOS 4). The software portion of the BLASTN pipeline ran on one CPU of the system, with support functions such as query preparation running on the other CPU.

For all experiments, we ran both versions of BLASTN with an E-value threshold of 10^{-5} and default parameters otherwise. Sequences were filtered for low-complexity regions before comparison. Reported runtimes include time spent in query setup and the three stages of the BLASTN pipeline but exclude time spent formatting the output for printing, which is extraneous to the main comparison.

The two experiments performed were as follows:

Table 1. Executing time of Mercury BLASTN compared the baseline system.

Experiment	Baseline Time	Mercury Time	Speedup
1	101 min	20 min	5.05x
2	218 min	19 min	11.47x

Table 2. Sensitivity results of Mercury BLASTN compared to the baseline system.

Experiment	Sensitivity	Alignments Lost	New Alignments Found
1	98.64%	8428	6089
2	99.01%	96	185

1. 3,975 randomly sampled human cDNA sequences (9 Mbases after removing known repeats and Ns), from Release 21 of the NCBI RefSeq cDNA library, against all other vertebrate cDNAs (586 Mbases after removing known repeats and Ns). Such a run would be used, e.g., to identify homologs of cDNAs from a library sequencing experiment.
2. Human chromosome 22 (hg18, 21 MBases after removing known repeats and Ns) against the entire mouse genome (mm8, 1.5 Gbases after removing known repeats and Ns). Such a run would be used for large-scale comparative genomics.

Tables 1 and 2 respectively show the speedup and the sensitivity of Mercury BLASTN relative to the software baseline on our comparisons. Sensitivity was measured by counting the fraction of alignments from the baseline output that also appeared in the Mercury BLASTN output. Alignments in the two outputs that overlap by more than 50% in both sequences are considered to be the same. The "Alignments lost" column counts significant alignments from the baseline not found by Mercury BLASTN, while the "New Alignments Found" column counts significant alignments found by Mercury BLASTN but not by the baseline.

Mercury BLASTN exhibits speedups ranging from 5x to more than an order of magnitude over the software baseline on the benchmark computations; put another way, one instance of Mercury BLASTN could replace 5-10 CPU cores of a multiprocessor with a single FPGA. These speedups are achieved while retaining 98.5-99% of all alignments found by NCBI BLASTN. In addition, Mercury BLASTN reports roughly as many significant new alignments as it loses existing ones, suggesting that observed differences in sensitivity are "in the noise" for these search tasks.

The results presented here characterize the performance of Mercury BLASTN on an FPGA co-processor that is one generation behind the currently available state of the art. In the near future, we plan to deploy our architecture on a new co-processor card containing two newer FPGAs from the Xilinx Virtex 4 family. The two FPGAs can process the same datastream for two queries in parallel. Moreover, each FPGA will have larger SRAMs and more on-chip block RAM, allowing us to double our query size. Finally, we plan to time-multiplex the block RAMs in stage 1 analogously to our current optimization in stage 2, which will enable a further doubling of the query size. Overall, the new hardware should allow us to run the Mercury BLASTN pipeline with two simultaneous queries, each four times larger than those used by the current implementation. As a result, we will be able to perform benchmarks like those described above in one-eighth as many passes over the database, for an eight-fold speedup over our current hardware prototype.

5 CONCLUSION

BLAST is one of the most important computational tools of modern genomics, making it worthwhile to invest substantial time and effort in making it run faster. We have explored acceleration of BLAST for DNA sequences using a specially adapted streaming architecture. Our solution achieves sensitivity nearly equal to that of NCBI BLASTN on real-world comparison tasks, with an observed speedup of 5-11x over a current-generation CPU on old FPGA hardware and an anticipated speedup of 40-88x on current-generation hardware. Our performance advantage comes from a combination of careful attention to which stages require acceleration, a stage 1 design that eliminates unnecessary traffic to memory, and a stage 2 design that adapts ungapped dynamic programming to work with limited hardware resources.

Besides porting Mercury BLASTN to a newer FPGA platform, we have plans for other extensions and improvements. One advantage of our FPGA-based hashing scheme is that it costs no more to use a discontiguous seed pattern than to use the standard 11-mer seed of NCBI BLASTN. Such patterns have been shown to yield sensitivity equal to that of BLASTN for longer seed lengths [7, 19], which translates to reduced traffic to memory for our stage 1 implementation. Discontiguous seeds should therefore allow us to eliminate a potential implementation bottleneck as the rest of the FPGA hardware gets faster.

The techniques used to accelerate seed matching in Mercury BLASTN should apply to other seeded alignment tools as well. Examples of such tools include other BLAST-family algorithms, such as BLASTP for protein; PSI-BLAST [2] and HMMERHEAD [24] for protein motif finding; and PhyloNet [32] for regulatory motif finding. Currently, we have a working FPGA prototype for BLASTP. The implementation challenges for these other applications are slightly different, since some, especially BLASTP, produce seed matches at a much higher rate than BLASTN; however, other architectural techniques, such as distributing table lookups across multiple memory modules, can address these challenges.

Ultimately, a limiting factor on the utility of streaming architectures for seeded alignment is the work required to implement them. Even with modern hardware description languages, the time investment needed to build a specialized architecture remains significant. However, work like that described here is identifying architectural features that lead to high-performance streaming implementations. We therefore hope to eventually produce a library of modules that can be tweaked and composed to quickly produce high-performance implementations for a variety of sequence analysis algorithms.

ACKNOWLEDGMENT

This work was supported by NIH award 1 R42 HG003225-01.

REFERENCES

[1]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.

[2]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.

[3]M. Blanchette, W. J. Kent, C. Riemer, L. Elnitski, A. F. A. Smit, K. M. Roskin, R. Baertsch, K. Rosenbloom, H. Clawson, E. D. Green, D. Haussler, and W. Miller. Aligning multiple genomic sequences with the threaded blockset aligner. *Genome Research*, 14:708–15, 2004.

[4]B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, May 1970.

[5]N. Bray and L. Pachter. MAVID: constrained ancestral alignment of multiple sequences. *Genome Research*, 14:574–9, 2004.

[6]M. Brudno, C. Do, G. M. Cooper, M. F. Kim, E. Davydov, NISC Comparative Sequencing Program, E. D. Green, and S. Batzoglou. LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA. *Genome Research*, 13:813–20, 2003.

[7]J. Buhler, U. Keich, and Y. Sun. Designing seeds for similarity search in genomic DNA. *Journal of Computing and Systems Science*, 70:342–363, 2005.

[8]N. Camp, H. Cofer, and R. Gomperts. SGI high throughput computational BLAST. SGI White Paper, September 1998.

[9]R. D. Chamberlain et al. 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.

[10]A. E. Darling et al. The design, implementation, and evaluation of mpiBLAST. *4th Int'l Conf. on Linux Clusters*, 2003.

[11]M. Herbordt et al. Single pass, BLAST-like approximate string matching on FPGAs. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

[12]J. D. Hirschberg, R. Hughley, and K. Karplus. Kestrel: a programmable array for sequence analysis. In *Proceedings of IEEE International Conference on Application-specific Systems, Architecture, and Processors*, pages 23–34, 1996.

[13]D. T. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, 1993.

[14]W. J. Kent. BLAT: the BLAST-like alignment tool. *Genome Research*, 12:656–64, 2002.

[15]G. Knowles and P. Gardner-Stephen. DASH: Localizing dynamic programming for order of magnitude faster, accurate sequence alignment. In *Proceedings of the 3rd International IEEE Computer Society Computational Systems Bioinformatics Conference*, pages 732–35, 2004.

[16]P. Krishnamurthy et al. Biosequence similarity search on the *Mercury* system. *Journal of VLSI Signal Processing*, 2007. In press.

[17]J. Lancaster et al. Acceleration of ungapped extension in Mercury BLAST. In *Proc. of 7th Workshop on Media and Streaming Processors*, Nov. 2005.

[18]D. Lavenier et al. A reconfigurable parallel disk system for filtering genomic banks. In *Engineering of Reconfigurable Systems and Algorithms*, pages 154–166, 2003.

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

[20]H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel BLAST. In *Proc. 19th Int'l Parallel and Distributed Processing Symposium*, page 72b, 2005.

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

[22]Z. Ning, A. J. Cox, and J. C. Mullikin. SSAHA: A fast search method for large DNA databases. *Genome Research*, 11:1725–9, 2001.

[23]W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Nat. Acad. Sci. USA*, 85:2444–8, 1988.

[24]E. Portugaly and M. Ninio. HMMERHEAD – accelerating HMM searches on large databases. Poster at 8th Ann. Int'l Conf. on Research in Computational Molecular Biology, 2004.

[25]M. V. Ramakrishna et al. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers*, 46:1378–1381, 1997.

[26]H. Rangwala, E. Lantz, R. Musselman, K. Pinnow, B. Smith, and B. Wallenfelt. Massively parallel BLAST for the Blue Gene/L. In *High Availability and Performance Computing Workshop*, 2005.

[27]S. Schwartz, W. J. Kent, A. F. A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller. Human-mouse alignments with BLASTZ. *Genome Research*, 13:103–7, 2003.

[28]R. K. Singh et al. BioSCAN: a dynamically reconfigurable systolic array for biosequence analysis. In *Proceedings of CERCS 96*, 1996.

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

[30]E. Sotiriades, C. Kozanitis, and A. Dollas. Some initial results on hardware BLAST acceleration with a reconfigurable architecture. In *HiCOMB Workshop*, April 2006.

[31]R. E. Tarjan and A. C. C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.

[32]T. Wang and G. D. Stormo. Identifying the conserved network of cis-regulatory sites of a eukaryotic genome. *Proc. Natl. Acad. Sci. USA*, 102:17400–5, 2005.

[33]Y. Yamaguchi et al. High speed homology search with FPGAs. In *Pacific Symposium on Biocomputing*, pages 271–282, 2002.

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