2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines

Two-Hit Filter Synthesis for Genomic Database Search

Jordan A. Bradshaw, Rasha Karakchi, Jason D. Bakos
Department of Computer Science
University of South Carolina
Columbia, SC USA
Email: bradshja@email.sc.edu, karakchi@email.sc.edu, jbakos@cse.sc.edu

***Abstract*—Advancements in genomic sequencing technology is causing genomic database growth to outpace Moore's Law. This continues to make genomic database search a difficult problem and a popular target for emerging processing technologies. The de facto software tool for genomic database search is NCBI BLAST, which operates by transforming each database query into a filter that is subsequently applied to the database. This requires a database scan for every query, fundamentally limiting its performance by I/O bandwidth. In this paper we present a functionally-equivalent variation on the NCBI BLAST algorithm that maps more suitably to an FPGA implementation. This variation of the algorithm attempts to reduce the I/O requirement by leveraging FPGA-specific capabilities, such as high pattern matching throughput and explicit on chip memory structure and allocation. Our algorithm transforms the database—not the query—into a filter that is stored as a hierarchical arrangement of three tables, the first two of which are stored on chip and the third off chip. Our results show that—while performance is data dependent—it is possible to achieve speedups of up to 8X based on the relative reduction in I/O of our approach versus that of NCBI BLAST. More importantly, the performance relative to NCBI BLAST improves with larger databases and query workload sizes.**

Keywords—reconfigurable computing, heterogeneous computing, FPGA, BLAST, approximate string matching, regular expression, pattern matching, high-performance computing, sequence alignment, automata processor, database search, computational biology, bioinformatics, comparative genomics, genomic analysis

I. Introduction

Many tasks in genomic sequence comparison rely on approximate string matching based on assigning biologically-significant scores to specific aspects of dissimilarity (or "edits") between two character strings. Dynamic programming methods such as the well-known Smith-Waterman [1] and Needleman-Wunsch [2] algorithms are optimal but their quadratic memory requirement makes them too expensive to match each query against the entire database. NCBI BLAST (Basic Local Alignment Search Tool) [3] addresses this problem by identifying a subset of the database that contains likely matches, to which it later applies Smith-Waterman.

NCBI BLASTP (protein BLAST), the focus of our work, requires, as input, a character substitution matrix $S(a,b)$ and a threshold. Using these, it generates a list of n-character strings called *seeds* (for BLASTP, typically $n = 3$) whose "self-score" (score when aligned against an exact match of itself) meets or exceeds the given threshold. BLAST then identifies exact or approximate matches of these in both the query and database. These are called *hits*. BLAST marks any pair of hits comprised of the same two seeds separated by the same relatively small number of intervening characters in both query and database as a *high scoring pair (HSP)* on the same *diagonal.* For each of these, BLAST extends the HSP by incorporating an increasing number of the HSP's neighboring characters until the slope of the running score becomes negative. At this point, if the *expectation value*, computed as a function of the score, is sufficiently low, BLAST performs a Smith-Waterman alignment between the query and the corresponding database entry.

NCBI BLASTP and each of its recent FPGA implementations need to scan the entire database for every query, making each query I/O bound (or memory bound if the databases can fit entirely in RAM) [4,5,6]. As such, our approach to BLAST acceleration is to reduce the I/O requirement. Specifically, our contribution is a hardware-centric approach for designing the two-hit filter, which operates by combining hit detection and two-hit filtering as a single hardware operation. To do this, we transform the database into a set of high scoring pairs (HSPs) and store it as a hierarchical series of indexed tables. The first on-chip table is the *HSP Suffix Table* and stores the set of valid HSPs relative to the substitution matrix and threshold. The second on-chip table is the *HSP Table of Contents,* which matches each HSP to a location in a larger off-chip database called the *HSP Index Table.* The HSP Index Table maps HSPs and lengths to the original database. Together, these tables allow the FPGA to accept a batch of queries and convert each into a corresponding subset of the original database against which to align using Smith-Waterman. These tables can reasonably fit in larger FPGA devices [7].

II. Related Work

As far as the authors know, the state-of-the-art in FPGA-based BLAST is embodied by CAAD BLAST, the most recent of a lineage of FPGA BLAST implementations from Herbordt's team from Boston University [8]. CAAD BLAST is implemented across four Virtex-6 LX760 FPGAs. The coprocessor comprises three filter stages, including the two-hit filter (the focus of our proposed approach), the extended ungapped alignment, and Smith-Waterman. As in NCBI BLASTP, the filters are constructed using each query—as opposed to the database as in our proposed approach—and subsequently used to filter the database. These filters must be reconstructed and the database scanned for every query. In order to maximize throughput, the filters are replicated multiple times and the database is partitioned such that one element from each database partition is read per cycle. Despite being implemented in custom hardware, CAAD BLAST incorporates many of the same design decisions as NCBI BLAST. For example, its two-hit filter is implemented using a table structure that is identical to NCBI BLAST, comprised of 25^n rows (where n = seed size) and three columns (i.e. each entry holds up to three occurrences

This material is based upon work supported by the National Science Foundation under Grant No. 1421059.

978-1-5090-2356-1/16 $31.00 © 2016 IEEE
DOI 10.1109/FCCM.2016.24

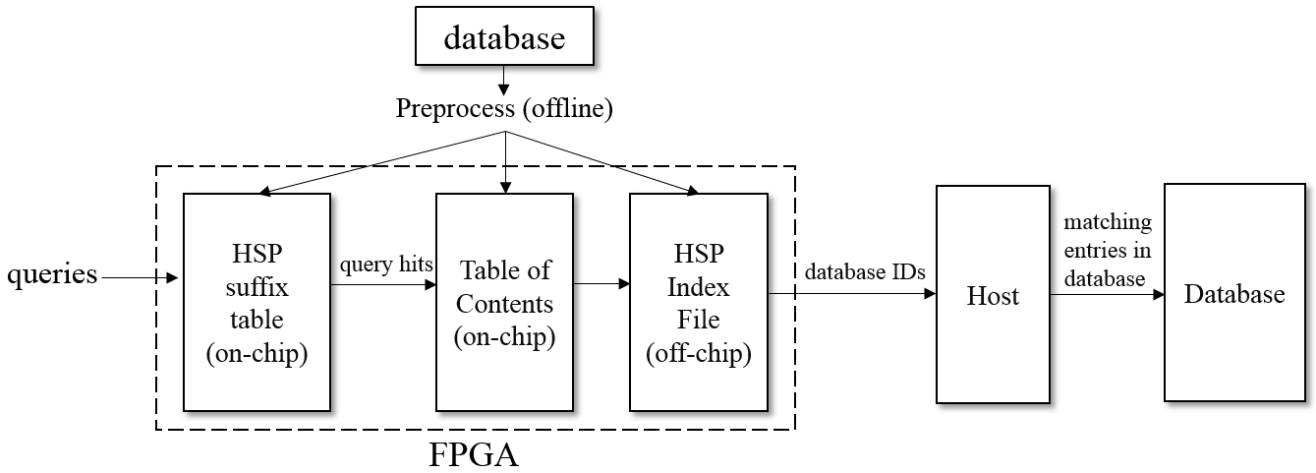

Fig. 1. Overall approach. The database is preprocessed to produce the HSP Suffix Table, Table of Contents, and HSP Index Table. The design filters queries through these three tables to produce a set of potential database matches. Each of these are aligned in the traditional way, but the execution time of BLAST is dominated by the first two filter stages implemented by these tables.

of the corresponding seed in the query and a secondary table holds additional entries). As the database characters are streamed into its pipeline, the design references the table row corresponding to the *n* most recent characters. When a hit is detected, a counter activates to determine the number of cycles that elapse between hits. The entire system achieves a speedup of up to 11X vs. CPU, which is impressive when considering that the FPGAs have only 3X the memory bandwidth of a CPU.

III. Approach

Figure 1 shows the data flow of our overall approach. The design on the FPGA begins by identifying all the valid HSPs in each query. For each of these "query HSP hits", the design uses a secondary table to find the corresponding offset and length of its section in the off-chip HSP Index Table. This provides a list of the corresponding "database HSP hits", a list of pointers to each of the original database entries that contain the HSP with its corresponding length. Each query HSP and database HSP having the same length is returned as output from the FPGA to the host for downstream processing (extension filter and potentially Smith-Waterman).

This approach is more amenable for FPGA implementation because (1) there is no need to re-initialize filter tables for each query to provide higher throughout for batched queries, (2) it reduces the I/O requirement, in which the kernel needs only to read a subset of the database for each query (at the cost of needing to read the HSP Index File), and (3) it exposes fine grain parallelism in the HSP matching stage.

A. Database Preprocessing

The first step is to determine how many of the 25^n *n*-mers have a self-score that meets the given threshold. The self-score is calculated as $\sum_i^n S(s_i, s_i)$ where *s* is the seed and *S()* is the substitution matrix. Each seed has a corresponding set of non-matching *n*-mers that, when scored against the original seed, give a score that still meets the threshold, i.e. $\sum_i^n S(s_i, d_i)$, where d_i is the potential substitution. The resulting set of seeds are matched against the database. Any pair of hits on the same database sequence that occur within 40 characters are combined to form a high scoring pair (HSP). In this case, the first and second seed hits represent the beginning and end—or *prefix* and *suffix*—of the HSP candidate. Using three-character seeds produces too large a table so we reduce the seed size to two-characters, e.g. AB,DE (requires a threshold adjustment of -2).

The preprocessing step then scans the database to identify each occurrence of each of these HSPs having a matching length of no more than 44 and having a minimum self-score value of $\frac{Tn}{2}$, where *T* is the seed threshold and *n* is the length of the HSP. Each of these matches and corresponding length is stored in the HSP Index File with a reference back to the corresponding database entry and offset. Each HSP may have multiple hits, and may hit across multiple records.

B. Runtime Behavior

Figure 1 depicts the runtime behavior. For each query, the design on the FPGA generates a set of *query HSP hits*, comprised of a two-character prefix and two-character suffix. Using these, the design indexes a table of contents, which stores the corresponding offset and length of the HSP's section in the HSP Index File. For each HSP the HSP Index File returns a list of corresponding database hits. Each of these points to database entry and match length. Each of these whose length matches the query hit is returned to the host as output.

Our profiling shows that the subsequent execution time of the extension filter is negligible as compared to the time required to access the sequence from the disk. Consequently, nearly all of NCBI BLAST's execution time is spent in the seeding and two-hit filtering stages, which themselves are dominated by I/O [9].

To characterize the growth of the HSP Index File we generated HSP Index Tables for subsets of popular, widely-cited biological protein databases of various sizes (NR [10] and Uniref [11]). Our results show that the file is approximately 10x larger than the input database and does not vary significantly with the input data.

C. Filter Architecture

As shown in Figure 1, the accelerator hardware converts each query into a set of database matches, where each database match corresponds to a matching HSP of the same total length between the query and database entry.

The most substantial component of the accelerator design is the query processor that generates the set of corresponding HSP hits for each query. Each HSP hit is comprised of a prefix-suffix pair, total HSP length, query ID, and query offset. Each query processor contains one HSP Suffix Table and 42 processing elements (PEs). The HSP Suffix Table provides a 1024-bit value that represents the valid suffixes for each pair of consecutive characters reach. Each PE can track only one HSP at a time, but it contains a sufficient number of PEs to hide the worst case latency of the PE (the maximum length of an HSP). This way it is not necessary to stall the query processor until the output buffer becomes full.

Whenever a PE becomes available, it latches the most recently-received pair of input characters as a prefix, latches its corresponding set of valid suffixes (as a bit vector), and then activates a counter. Each subsequent two-character sequence is decoded into a bit array that bit-wise AND'ed with the suffix bit vector. Any one-bits within the output of the AND operation marks an HSP hit. Any HSP hits that occur before the counter reaches 40 are reported. The matching suffix is generated using an encoder on the output of the AND-operation. Another counter tracks the offset into the query, allowing both counters provide the offset and length for each detected HSP. When synthesized and implemented on a Xilinx Zynq 7020, one query processor requires 31258/5320 (58.8%) slice LUTs.

The design produces the final list of database HSP hits by cross referencing the query hits with the HSP Index File. The query hits are sorted by HSP ID and length, allowing the design to read all of the relevant database hits from the HSP Index File using a minimum number of accesses.

To minimize I/O back to the host, the design returns a set of a query hits that match each database entry returned. Each query hit includes the offset between query hit and database hit. Each diagonal is passed to the extension and alignment stage.

IV. Experimental Results

A. I/O Overhead

The throughput of our design is bounded by I/O, which is comprised of transfers from the database and the HSP Index File. To characterize overall speedup, we first must measure the I/O requirements of accessing the HSP Index File and compare it to the overall database size.

Figure 2 shows the number of bytes read from the HSP Index File as a function of the number of query characters processed when we perform searches of 20-100 queries randomly selected from the NR database using the first 200K records of the NR database as the search space. The bytes read follow a linear relationship with the number of query characters. About 75MB is read for 100 queries (about 37K query characters), compared to the full database size of about 800MB.

Figure 3 shows the total number of database records read as a function of the number of query characters processed, using the first 200K records of the NR database. The application need only read database records if there is an HSP from the HSP Index File corresponding to it from at least one query. Because it quickly converges to the database size, it is advantageous to batch queries into the largest batch allowed by the HSP buffer in order to minimize the frequency of repeated reads from the database.

B. Word Finder Speedup

State-of-the-art FPGA-based BLAST implementations must re-read the entire database to rebuild the filters for each query. Our proposed approach read hits from the HSP Index File, and at most, reads each database entry once per batch of queries.

We calculate the speedup of our approach using a technology-independent approach, where we assume that I/O is the performance bottleneck and that the HSP Index File and database are stored in a memory or disk having the same effective bandwidth. We estimate speedup relative to current NCBI BLAST algorithm as follows:

$$Speedup = \frac{BytesRead_{Database} + BytesRead_{HSPIndex}}{NumQueries \times DatabaseSize}$$

Figure 4 plots the resulting speedup versus query workload,. This follows from our results that show that each query requires reading less than the full database's size from the HSP Index File.

C. Overall Speedup

To estimate whole-application speedup, we must consider the relative time spent in the word finder component of BLAST vs. the full execution time. Our experiments show that the word finder requires a varying amount of overall execution time that increases with the size of the database being searched, ranging from 75% for 20 queries against a 200K record database up to 93% for the same 20 queries against a 1M record database.

Figure 5 shows our projected overall speedups for databases ranging from 200K records to 1M records, assuming that word finder speedup is a function of I/O reduction and that overall speedup is limited by the relative time spent in the word finder.

V. Conclusions

In this paper we describe a new algorithm for BLAST that is more amenable to FPGA acceleration than NCBI BLAST. We show that this algorithm can substantial reduce the I/O requirement when assuming that the HSP finder can operate on the FPGA with zero time overhead and when the HSP Suffix table and Table of Contents tables can fit within FPGA memory. In future work we will analyze the impact of query batching on a deployed system.

References

[1] Temple F. Smith, Michael S. Waterman, "Identification of Common Molecular Subsequences," Journal of Molecular Biology 147: 195–197, 1981, doi:10.1016/0022-2836(81)90087-5.

[2] Saul B. Needleman, Christian D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," Journal of Molecular Biology 48 (3): 443–53. doi:10.1016/0022-2836(70)90057-4, 1970.

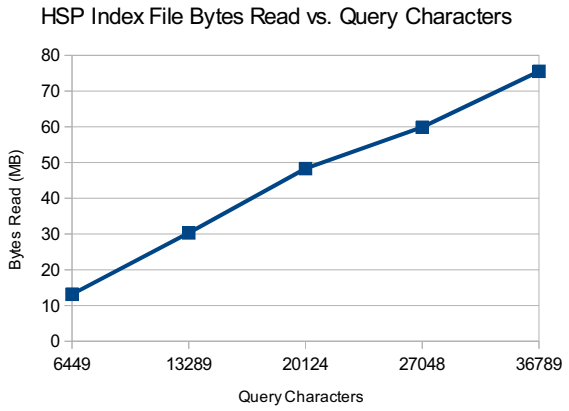

Fig. 2. HSP Index File bytes read as a function of query characters, using the first 200K records from NR as the processed database.

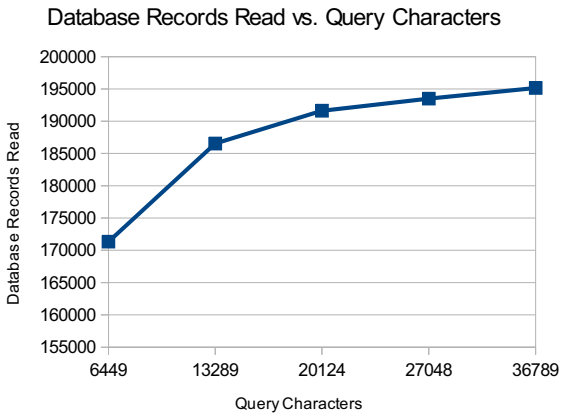

Fig. 3. Database records read as a function of query characters, using a 200K record database.

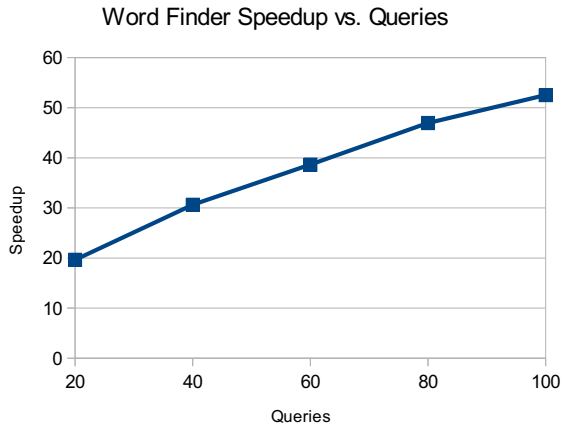

Fig. 4. Word Finder Speedup as a function of queries, using the first 200K records from NR as the processed database.

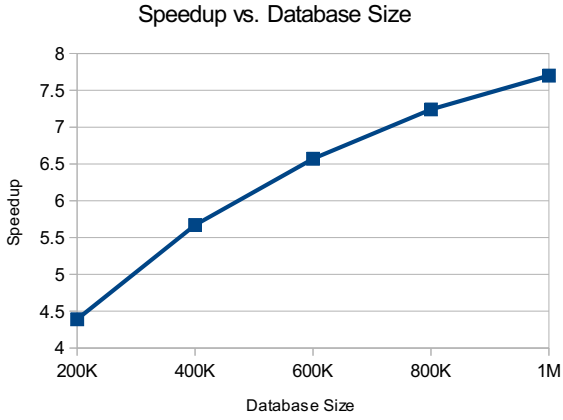

Fig. 5. Overall speedup as a function of database size, using 20 randomly selected queries from the NR database.

[3] S. F. Altschul, W. Gish, W. Miller, E.W. Myers, D. Journal Lipman, “Basic local alignment search tool,” Journal of Molecular Biology, 215 (190), 403-410.

[4] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman, “Gapped BLAST and PSI-BLAST: a new generation of protein database search programs,” Nucleic Acids Research, 1997, Vol. 25, No. 17, 3389–3402.

[5] Xinyu Guo, Hong Wang, Vijay Devabhaktuni, "A Systolic Array-Based FPGA Parallel Architecture for the BLAST Algorithm," ISRN Bioinformatics Volume 2012, Article ID 195658, doi:10.5402/2012/195658.

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

[7] Xilinx UltraScale+ FPGA Product Tables and Product Selection Guide, http://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf, retrieved Sept. 2015.

[8] Atabak Mahram, Martin C. Herbordt, "NCBI BLASTP on High-Performance Reconfigurable Computing Systems," Transactions on Reconfigurable Technology and Systems (TRETS), Volume 7 Issue 4, 2015.

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

[10] NR Database, available from http://nih.gov.

[11] Uniref100 Database, available from http://www.uniprot.org/downloads.