High-Level Synthesis of a Genomic Database Search Engine

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

***Abstract*—Genomic database search is an I/O-bound problem, so avoiding unnecessary I/O transactions is a key consideration for improving search throughput. Many approximate search tools such as NCBI BLAST perform a database scan for each query, lacking a mechanism to avoid access to portions of the database that offer no potential for a match. In this paper we present an approach for using an FPGA-based pattern filter to convert each search query into a set of potential database matches that reduces the average portion of the database accessed per query. The approach is based on a hardware design for a pattern filter that can achieve a sustained recognition rate of one pattern per cycle. We used Vivado HLS to design the filter. Despite the presence of loop-carried dependencies, the final design meets the maximum possible throughout as constrained by the code's arithmetic intensity and available memory bandwidth. In this paper we describe the filter implementation and our code tuning methodology.**

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

I. INTRODUCTION

Genomic database search relies on a type of approximate string matching that is based on assigning biologically-significant contributions and penalties to specific aspects of similarity and dissimilarity between two character strings representing DNA or protein data. The Smith-Waterman [1] and NCBI BLAST (Basic Local Alignment Search Tool) [2,3] algorithms are well-suited for this task but both require a full database scan for each query. This is not a serious limitation in cases where queries are infrequent or when the database can fit entirely in DRAM. However this becomes a major performance bottleneck when large sets of queries are searched against a very large database stored in non-volatile storage.

NCBI BLAST is more commonly-used than Smith-Waterman for database search because it works on the principle of database filtering, in which it identifies a subset of the database that contains likely matches and then applies Smith-Waterman to this subset. This filtering process is based on searching for simple patterns shared between the query and each of the database entries. These patterns are comprised of a pair of short, fixed-length words, called *seeds*, which appear within the same relatively short distance in both query and database entry. Given a substitution matrix and score threshold, the seeds are comprised of the subset of all *n*-character orderings whose characters are among the highest-scoring against themselves and others.

Any pair of seeds that occur within a user-defined distance of each other are called *high scoring pairs (HSPs)*. For each HSP of the same length in both the query and database entry, BLAST computes the score contribution of each of the intervening characters. If each these contributions are positive, BLAST will flag the corresponding database entry as a likely match and send it for further processing. Matching the query HSPs against the database HSPs consumes nearly all of NCBI BLAST execution time.

Our approach to BLAST acceleration is to avoid the need to read all the database entries by pre-computing a table of all database HSP patterns and cross-reference these against each set of query HSP patterns [4]. Although the set of database HSPs is larger than the original database, only a small portion of it and the original database need be accessed per query. This approach, which filters the query using the database, is an inverted form of NCBI BLAST, which filters the database using the query.

This paper describes a new FPGA-based design for identifying HSPs in the query that match HSPs in the database. The design uses an on-chip table to store HSPs as a set of start and end seeds (referred to as prefix and suffix below), and a second, off-chip table that stores the corresponding set of pattern lengths that appear in the database. The length of each HSP candidate identified by the on-chip table is compared against the off-chip table. The design hides the latency of the off-chip table through pipelining. The design involves complex control and data movement, making it an ideal candidate for implementation using high-level synthesis tools. Using Vivado HLS, our design achieves a consistent throughput of 5.1 million query characters per second on a Virtex-7.

II. RELATED WORK

Previous work has explored the potential performance benefits of performing genomic database search on FPGAs. As far as the authors know, CAAD BLAST is the current state-of-the-art in published FPGA-based BLAST implementations [5]. CAAD BLAST is implemented across four Virtex-6 LX760 FPGAs and performs all components of the BLAST algorithm on the FPGAs. CAAD BLAST is designed to generate results that exactly match those of NCBI BLAST. To guarantee this, the CAAD BLAST is a literal translation of the NCBI BLAST algorithm into hardware; it closely follows the implementation of NCBI BLASTP, where each query acts as a filter applied to the database; i.e. for each query, the software must reconstruct the filter and scan the database. The entire system achieves a speedup of up to 11X vs. the software, which is remarkable when considering that the FPGAs have only 3X the memory bandwidth of a CPU.

This material is based upon work supported by the National Science Foundation under Grant No. 1421059.
978-1-5090-3707-0/16/$31.00 ©2016 IEEE

Other FPGA BLAST implementations that do not guarantee exact result equivalency to NCBI BLAST use this freedom to restructure the algorithm to more efficiently map to hardware. Specifically there is previous work in restructuring the BLAST two-hit filter as a systolic array, where the query and database sequences are streamed into the FPGA alongside each other and compared with an array of parallel comparators [6]. This approach is notable in that it avoids the need for hash tables but must repeatedly read the query while scanning the database, requiring the FPGA to transact approximately twice as much data as compared to CAAD BLAST and achieving less performance.

III. Background

As described in Section 1, NCBI BLASTP uses seeds as a method for identifying points of interest in both the query and database. To build the seed set, BLAST builds an initial set of all *n*-character sequences (where *n* is normally 3 for BLASTP) whose score relative to itself (as determined by the given substitution table) exceeds a user-defined threshold value. From each of the members of this initial set, BLAST adds any *n*-character sequence whose score against it also exceeds the threshold.

For protein BLAST (BLASTP), an HSP is comprised of a pair of seeds separated by up to 40-wildcard characters. This type of pattern is represented as a regular expression such as ABC.{0,40}DEF, where ABC is the "prefix" seed and DEF is the "suffix" seed. Note that one prefix instance in the query starting at one offset may initiate multiple HSP occurrences having corresponding suffixes at different offsets. A query may contain several instances of this type of pattern. In the worst case a query could contain up to 44 HSP patterns for every character starting with character offset 44.

Using the default settings in NCBI BLAST of $n = 3$ over the 24-character protein alphabet there are $24^6 = 191$ million HSP permutations, of which 130 million are valid seeds under NCBI BLASTP's default threshold of 13 and commonly-used BLOSSUM62 substitution table. Storing all the valid HSPs on chip is impractical using three-character so we reduce the seed size to two, e.g. AB,{0,40}DE and make a corresponding adjustment to the threshold, making it 11. This gives $24^4 = 331$K HSP permutations, of which 179K are valid under the conditions previously described.

IV. Approach

A. Database Preprocessing

Figure 1 shows a top-level view of our approach. During the preprocessing phase, a software tool generates three tables from a given database and associated substitution matrix and threshold. These tables are the *suffix table*, *table of contents*, and the *HSP Index Table*.

1) Suffix Table

The suffix table is of a fixed size of $24^2 \times 24^2 = 576 \times 576$ bits, which we round to 1024 x 1024 = 1 Mb for mapping to BRAM, and is intended for on chip storage. It is addressed using the 10-bit prefix hash, computed by subtracting 65 and masking off the lower 5 bits of both ASCII characters of the prefix, and then concatenating the resulting values. Each row of the suffix table is a 1024-bit bit vector where bit *n* represents the validity of the corresponding suffix, where *n* is the hash of the suffix (computed using the same method as the prefix). The hardware uses the suffix table to identify valid HSPs in the query.

2) HSP Index Table

The preprocessing step scans the database and identifies each occurrence of each valid HSP having a total length in the range of [4,44] and having a minimum total self-score value of $\frac{Tn}{2}$, where *T* is the seed threshold and *n* is the match length. The preprocessing tool stores the corresponding database index and offset of each of these occurrences into the HSP Index Table. Each HSP may have multiple hits, and may hit across multiple records.

We generated HSP Index Tables for subsets of widely-cited biological protein databases of various sizes (NR [7] and Uniref [8]). Our results show that the resulting HSP Index Table is approximately 10X the size of the input database, irrespective of the database contents.

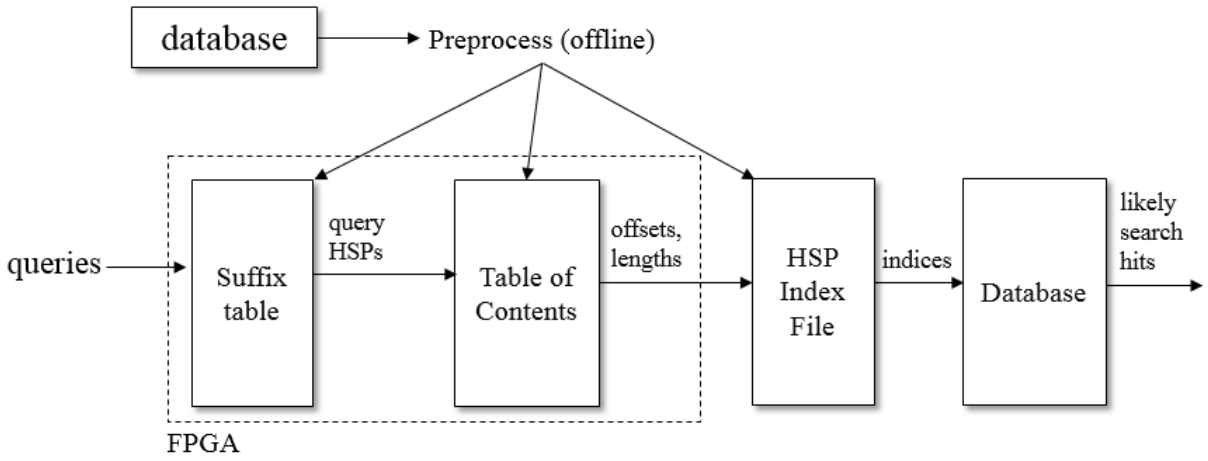

Fig. 1. Overall approach. The database is preprocessed to produce the HSP Suffix Table, Table of Contents, and HSP Index Table. The design filters queries through these three tables to produce a set of potential database matches. Each of these are aligned in the traditional way, but the execution time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

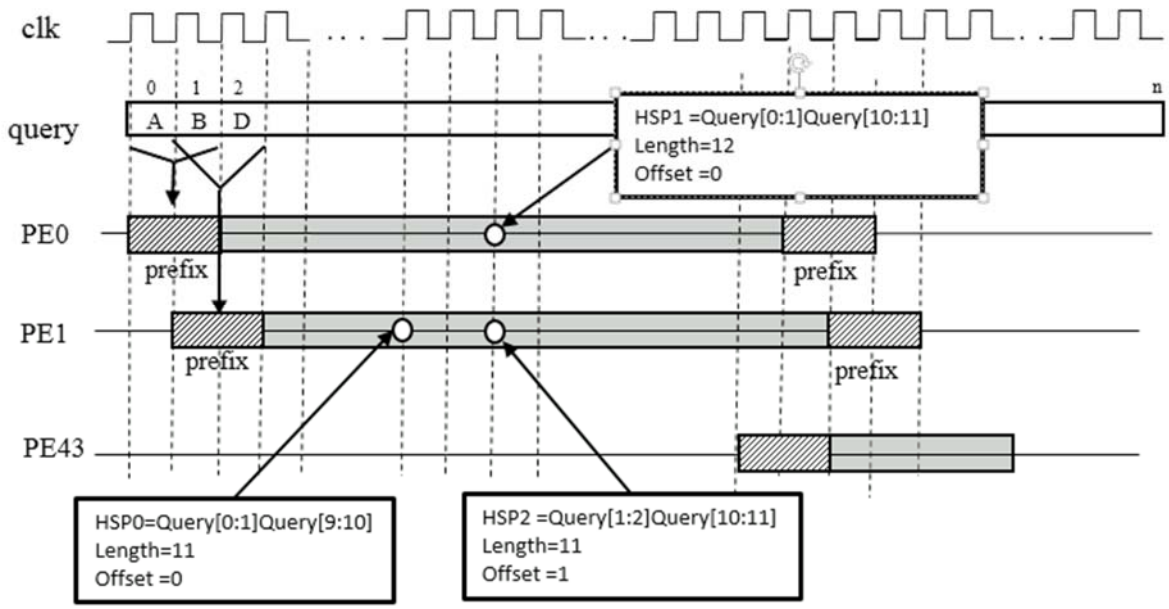

Fig. 2: Operation of query processor.

3) Table of Contents

The table of contents is indexed HSP and length, i.e. 20-bit HSP (10 bits for both the prefix and suffix) and its corresponding length. Each entry in the table of contents is either null (if the HSP and length do not appear in the database) or points to an offset and length in the HSP Index Table. There are 24^4 HSP permutations and 40 possible lengths, giving 24^4 x 40 = 13 million entries in the table. Each entry provides the corresponding 8-byte offset and 4-byte length to the corresponding section in the HSP Index Table, giving a total size of 128 MB.

B. Runtime Behavior

Figure 1 depicts the runtime behavior. For each query, the design generates a set of *query HSP hits*, comprised of a four-character prefix-suffix pair and length. If the corresponding value stored in the table of contents is non-null, the design returns the contents of the table of contents to the host CPU, which will access the HSP Index Table and perform subsequent filtering steps.

C. Initial Query Processor Design

In our previous work we described our initial design for the query processor, a streaming architecture that converts each query into a set of HSP hits. We designed this version in HDL but it only generated hits from the suffix table and was not able to access the table of contents due to the a relatively complex dependency chain of table accesses. Our new design, described in C for Vivado HLS, has pipeline stages for accessing the suffix table, table of contents, and output arrays. The pipeline's data introduction interval (II) is matched to the DRAM bandwidth.

The operation of the query processor relative to the query string is shown in Figure 2. The query processor contains one HSP Suffix Table and 44 pipelined processing elements (PEs). PE p can detect all HSPs whose prefix begins with query character c, where c mod 44 = p. The query processor contains a sufficient number of PEs such that there is one PE to track all HSPs starting with any character in the query.

Whenever a PE begins tracking HSPs at a particular offset, it latches the most recently-received pair of input characters as a prefix candidate, latches that prefix's corresponding set of valid suffixes as a 1024-wide suffix bitvector (stored as sixteen 64-bit words in C), and then activates a counter.

The PE treats each subsequent two-character sequence as a potential 10-bit suffix hash. It decodes each of these into a 1024-wide bitvector and computes its bitwise-AND with the suffix bitvector. Any one-bits in this result indicates a query HSP hit. The PE encodes this result to compute the matching suffix value. Another counter tracks the offset into the query, allowing both counters provide the offset and length for each detected HSP.

D. Theoretical Performance Impact

In previous work we reported the results shown in Figure 3, which extrapolates the potential performance improvement of

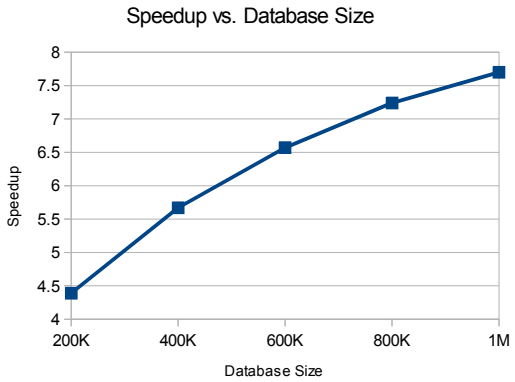

Fig. 3. End-to-end speedup of NCBI BLAST as a function of database size, using 20 randomly selected queries from the

this approach by measuring the reduction in total I/O during the HSP finder stage as compared to NCBI BLAST while also considering the relative amount of execution time spent finding HSPs as compared to the other portions of the code.

V. Query Processor Design

This paper describes our new query processor design, which is implemented as a systolic array using Vivado HLS 2016.1.

A. Query Processing Algorithm

Algorithm 1 shows a sketch of the algorithm for detecting HSPs in the query. It also closely resembles the initial code before optimization.

The outermost loop (line 2) iterates once for each query character. The loop body begins by considering each symbol and its predecessor as a potential suffix (line 3). The inner loop (line 4) iterates once per processing element. For each that is actively tracking an HSP (line 5), check to see if its maximum length is exceeded (line 6) and if so, deactivate it (line 7). Otherwise, if the bitwise-AND of the decoded form of the suffix (conversion from 10-bit binary to 1024-bit unary) is non-zero, the length greater than three, and there is an entry in the ToC corresponding the resultant HSP and length, then output the HSP (lines 8-11). To output an HSP, set the entries in output arrays for its length, query offset, HSP Index Table offset, and HSP Index Table length (lines 12-17). If the PE is not active and the current symbol has not been considered as a potential suffix, then allocate a new PE (lines 19-22).

The runtime operation of the code is illustrated in Figure 2. Each of the 44 PEs identifies all valid HSPs within each consecutive block of 44 query characters, with each PE's block offset from the next PE by one character. Each PE may identify multiple suffixes corresponding to its latched prefix within a single 44-character span.

B. Design Considerations

Seven arrays (the query, two arrays that form the table of contents, and the five output arrays) are allocated in off-chip DRAM with shared access with the host CPU. The host CPU performs the back end of the BLAST algorithm using the query HSP hits and their corresponding addresses in the HSP Index File as inputs. The host is responsible for retrieving the query HSP hits from the HSP Index Table and performing the extension and Smith-Waterman phases.

We associate an AXI master interface with each of the seven input and output arrays. These ports share access to the device's DRAM controller through an AXI crossbar to the FPGA's DRAM controller. The generated pipeline schedules a cycle for each access to each AXI master port, though the pipeline will only perform the read and write accesses when needed. For example, the code reads arrays *ToC.offsets* and *ToC.lengths* on lines 10, 14, and 16, but the latter two are accessed only on an HSP hit. Likewise, the pipeline schedules accesses to the output arrays $hsps_{length}$, $offsets_{query}$, $offsets_{index}$, and $lengths_{index}$ on lines 12, 13, 14, and 16 but it only performs the writes on an HSP hit. This way, the required memory bandwidth depends on the frequency of HSP hits. Lower hit frequency will reduce contention for the DRAM controller and minimize pipeline stalls.

ALGORITHM 1: Query HSP Search

Inputs: *query* (array of 8-bit characters),
$length_{query}$ (scalar),
ToC.offsets (array of 64-bit values),
ToC.lengths (array of 32-bit values)

Outputs: $offsets_{query}$ (array of 8-bit values),
$hsps_{length}$ (array of 8-bit values),
$offsets_{index}$ (array of 64-bit values),
$lengths_{index}$ (array of 32-bit values)

Internal State:
suffix_table (1024x1024 bit table)
PE.active (bit vector; execution state of PEs)
PE.suffixes (array of 44 1024-bit vectors)
PE.offset (array of 44 16-bit values)
PE.prefixes (array of 44 16-bit values)
current_symbol_initiate (flags current prefix)
suffix (current suffix candidate)

```
1   PEactive0..numPEs := {0,0,...,0}
2   for i in 1 to length_query loop
3     suffix := concat(query[i-1],query[i])
4     for j in 0 to numPEs – 1 loop
5       if PE.active[j] then
6         if (i – PE.offset[j]) > 44 then
7           PE.active[j] := false
8         else if (decode(suffix) & PE.suffixes[j]) and
9               ((i – PE.offset[j]) > 3) and
10              ToC.offsets[concat(PE.prefix[j],suffix) x
11              (i – PE.offset[j])] != null then
12              hsps_length [i*44+j] := i – PE,offset[j]
13              offsets_query[i*44+j] := PE.offset[j]
14              offsets_index[i*44+j] := ToC.offsets
15                    [concat(PE.prefix[j],suffix) * length]
16              lengths_index[i*44+j] := ToC.lengths
17                    [concat(PE.prefix[j],suffix) * length]
18      else
19        PE.active[j] := true
20        PE.offset[j] := i - 1
21        PE.prefix[j] := concat(query[i-1],query[i])
22        PE.suffixes[j] := suffix_table[PE.prefix[j]]
```

Each iteration of the inner loop schedules reads of 16 bytes (rounded up to the nearest multiple of 4): one byte from *query*, 8 bytes from *ToC.offsets,* and 4 bytes from *ToC.lengths.*

Each iteration of the outer loop writes 20 bytes (rounded up to the nearest multiple of 4): 2 bytes to $offsets_{query}$, one byte to $hsps_{length}$, 8 bytes to $offsets_{index}$, and 4 bytes to $lengths_{index}$.

Achieving a throughput of one cycle per inner loop iteration requires in the worst case 36 bytes of memory access per cycle, or 10.8 GB/s at 300 MHz.

We allocate all the internal arrays on BRAM. The suffix array is a ROM, initialized from our database formatting tool.

C. Code Optimizations

Our initial implementation was a literal translation of Algorithm 1 to C code. In order to provide additional flexibility for the scheduler we completely unrolled the inner loop and instructed the compiler to pipeline the *i*-loop with minimal iteration interval (II). Our objective is to achieve an iteration interval equal to 44, the number of PEs required to identify the maximum number of HSPs per query character.

Our first optimization was to use a variable to register the value of *query*[*i*-1] in order to avoid the need for an additional access to the *query* array.

Our second optimization was to manually convert all the internal arrays into scalar variables in order to encourage the compiler to map as many arrays as possible to LUTs.

Our third optimization was to transform the code for decoding the suffix value and matching its decoded value against the corresponding bit in PE's suffix table row (line 8 in Algorithm 1) by eliminating all control structures such as if-statements and for-loops. We store the 1024-bit suffix bit vector as sixteen 64-bit variables, each corresponding to a group of 64 consecutive bits in the 1024-bit value. For each decoded suffix value we calculate its corresponding bit position within its surrounding group of 64 consecutive bits by masking off the low-order 6 bits of the suffix value:

$$suffix_{mod64} = suffix \;\&\; 0x3F$$

With this, we calculate an intermediate value *masks* by isolating the bit position calculated above and repositioning it into a new position corresponding to its group number:

$$masks = \bigcup_{i=0}^{15} (PE.suffixes[i] \gg suffix\_mod64) \ll i$$

We calculate the group number by dividing the suffix by 64:

$$suffix_{div64} = suffix \gg 6$$

We then mask off the group number against its corresponding bit position in *masks*:

$$suffix_{match} = masks \gg suffix_{div64} \& 1.$$

D. Scalability

Table I shows the resource requirements as the query processor is scaled from 12 to 44 processing elements (note that the design requires 44 processing elements for correct operation). The LUT and DSP requirements scale linearly, with approximately 5,000 LUTs and one DSP required for each PE. The number of BRAMs is constant irrespective of the number of PEs.

Table II shows the pipeline performance as the number of PEs is scaled from 12 to 44. In each case the compiler achieves the minimum II—equal to the number of PES—as constrained by the number of AXI master interfaces (since each additional PE requires an additional access to the ToC). The pipeline depth increases by one cycle for each PE. The clock rate is consistent at 369 MHz for each scale. We calculate the pipeline throughput as *clock rate* / *II*. At 44 PEs we achieve a throughput of 5.1 million query characters per second.

TABLE I: RESOURCE USAGE OF SYNTHESIZED PIPELINES ON VIRTEX-7 AS REPORTED BY VIVADO HLS 2016.1

# PEs	LUTs	DSPs	BRAMs
12	59349	14	50
16	78990	18	50
20	98655	22	50
24	117989	26	50
28	137654	30	50
32	157276	34	50
36	176948	38	50
40	196258	42	50
44	216002	46	50

TABLE II: STATIC PERFORMANCE RESULTS OF QUERY PROCESSOR PIPELINE ON VIRTEX-7 AS REPORTED BY VIVADO HLS 2016.1

# PEs	II	Pipeline Depth	Clock rate (MHz)	Throughput (Mchars/s)
12	12	40	369	9.2
16	16	44	369	8.4
20	20	48	369	7.7
24	24	52	369	7.1
28	28	56	369	6.6
32	32	60	369	6.2
36	36	64	369	5.8
40	40	68	369	5.4
44	44	72	369	5.1

E. DRAM Memory Latency

Vivado HLS assumes a default read latency of five cycles for all top-level input arrays mapped to an AXI master interface. Of these, four cycles are associated with the AXI master interface and one additional cycle for the ap_bus interface. Vivado HLS makes this assumption even when the port is used to access an off-chip DRAM having potentially higher latency. If the DRAM latency is greater than the latency assumption in the generated design, the pipeline will stall on every access, increasing the effective II value reported by the compiler and reducing expected query throughput.

Since DRAM latency varies with the specific type of DRAM on the target platform, we use Vivado HLS's ap_bus latency directive to explicitly set the read latency of the modeled DRAM and sweep it in order to evaluate its impact on the resource usage and throughput of the generated design.

Table III shows the results of this analysis. The first column shows the latency setting for the ap_bus (does not include the additional four cycles for the AXI master interface). Even when setting the DRAM latency as high as 30 cycles, Vivado HLS is able to maintain an II of 44 and a clock rate of 369 MHz while only increasing the pipeline latency. By deepening the pipeline

in this way we avoid the potential for lost throughput from DRAM latency.

TABLE III: RESOURCE USAGE OF SYNTHESIZED PIPELINES ON VIRTEX-7 AS REPORTED BY VIVADO HLS 2016.1

DRAM Latency (cycles)	II	Pipeline Depth	Clock rate (MHz)
5	44	80	369
10	44	90	369
15	44	100	369
20	44	110	369
25	44	120	369
30	44	130	369

F. Comparison with CPU

Using OpenMP we developed a multithreaded version of query processor for software execution. For this we used the same code as with Vivado HLS including the corresponding code optimizations. The decoder implementation in particular will reduce the number of branch instructions as compared to the original unoptimized version.

To parallelize with OpenMP we made further adjustments to the code. The outer for-loop, which iterates over each query character, is made a parallel region and all its variables are made private. Within the loop, the code block for each processing element is prefaced with the following if condition that effectively distributes the pool of PEs across the threads:

if (PE_number mod thread_count) == thread_number then

When running the software implementation we initialize the *ToC.offsets* array to zero, which prevents the software from ever writing any of the output arrays. This makes the software behavior deterministic and provides an upper bound for the throughput. In a separate run we initialize the ToC such that all query HSPs are reported in the output arrays. This gives us a lower bound on throughput.

TABLE IV: NUM_THREADS VS PERFORMANCE FOR SOFTWARE IMPLEMENTATION

	Upper bound		Lower bound		
Threads	Time/ char (ns)	Thrp't. (Mc/s)	Time/ char (ns)	Thrp't. (Mc/s)	Ave. Thrp't
1	115	8.7	421	2.4	5.6
2	87	11.5	256	3.9	7.7
3	120	8.3	219	4.6	6.5
4	143	7.0	219	4.6	5.8
5	164	6.1	273	3.7	4.9
6	200	5.0	277	3.6	4.3
7	215	4.7	298	3.4	4.1
8	237	4.2	320	3.1	3.7

We ran each test on a 3.50 GHz four-core (eight logical core) Intel i5-4690K. Since the CPU applies constant frequency scaling according to load, our execution times are averaged over 5,000 runs using 100K character queries.

Our results are listed in Table IV. The best performance is with two threads, achieving between 3.9 and 11.5 million characters per second in the best and worst case. This is compared to our FPGA implementation which achieves a consistent performance of 5.1 million characters per second after the pipeline is filled.

The software performs best with two threads, which indicates that the performance is driven more by memory performance than computational resources. The FPGA performance falls between the upper and lower bounds of the two-thread case and always outperforms the lower bound.

The FPGA suffers from three disadvantages as compared to the CPU: (1) the FPGA has less than half the peak DRAM bandwidth, (2) despite having random access, the 128 MB size of the table of contents is sufficiently small to potentially benefit from the CPU's 6 MB L3 cache, and (3) as previously described the FPGA pipeline schedule always allocates time for the worst case memory behavior.

VI. CONCLUSIONS

In earlier work we developed a new NCBI BLASTP-compatible database search algorithm optimized for throughput on an FPGA-based coprocessor. This paper builds on this work by describing the design of the query processor using the Vivado high-level synthesis compiler. By using the HLS compiler we are able to implement pipelined, indirect referencing through multiple off-chip arrays. Specifically the design reads query data, tracks character patterns, uses the resultant patterns to address another off-chip array, and write its contents to off-chip output arrays. Our design achieves a throughput consistent with the memory bandwidth constraints, reaching a final throughput of 5.1 million query characters per second. In future work we will add downstream processing in our FPGA design.

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] 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.

[3] 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.

[4] Jordan Bradshaw, Rasha Karakchi, Jason D. Bakos, "Two-Hit Filter Synthesis for Genomic Database Search", Proc. 24th IEEE International Symposium on Field-Programmable Custom Computing Machines.

[5] 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.

[6] 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.

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

[8] Uniref100 Database, available from http://www.uniprot.org/downlo