FHAST: FPGA-Based Acceleration of Bowtie in Hardware

Edward B. Fernandez, Jason Villarreal, Stefano Lonardi, and Walid A. Najjar

Abstract—While the sequencing capability of modern instruments continues to increase exponentially, the computational problem of mapping short sequenced reads to a reference genome still constitutes a bottleneck in the analysis pipeline. A variety of mapping tools (e.g., Bowtie, BWA) is available for general-purpose computer architectures. These tools can take many hours or even days to deliver mapping results, depending on the number of input reads, the size of the reference genome and the number of allowed mismatches or insertion/deletions, making the mapping problem an ideal candidate for hardware acceleration. In this paper, we present FHAST (FPGA hardware accelerated sequence-matching tool), a drop-in replacement for Bowtie that uses a hardware design based on field programmable gate arrays (FPGA). Our architecture masks memory latency by executing multiple concurrent hardware threads accessing memory simultaneously. FHAST is composed by multiple parallel engines to exploit the parallelism available to us on an FPGA. We have implemented and tested FHAST on the Convey HC-1 and later ported on the Convey HC-2ex, taking advantage of the large memory bandwidth available to these systems and the shared memory image between hardware and software. A preliminary version of FHASTrunning on the Convey HC-1 achieved up to 70x speedup compared to Bowtie (single-threaded). An improved version of FHAST running on the Convey HC-2ex FPGAs achieved up to 12x fold speed gain compared to Bowtie running eight threads on an eight-core conventional architecture, while maintaining almost identical mapping accuracy. FHAST is a drop-in replacement for Bowtie, so it can be incorporated in any analysis pipeline that uses Bowtie (e.g., TopHat).

Index Terms—Short-read mapping, genome re-sequencing, FPGAs, reconfigurable hardware

1 Introduction

The estimated two thousand DNA sequencing instruments in research facilities, universities and hospitals around the world, have the potential to generate 15 petabytes of data in a year [2]. Sequencing capacity has been increasing 3x-5x a year, far exceeding Moore's law. For the vast majority of sequencing projects, the first step after cleaning/trimming the reads involves mapping the reads to the reference genome. The problem of *short read alignment* or *short read mapping* is a computationally intensive operation that involves matching hundreds of million of short strings (called *reads*) against a reference genome, which can reach in the billion of bases pairs.

Besides the sheer volume of reads to be processed, one of the challenge of these kind of data intensive applications is that they are irregular. Traditional techniques for exploiting locality, such as caching, are not effective for these applications, hence long memory latencies have an amplified impact on their performance. One objective of multithreaded architectures, as proposed in the Tera MTA [3], [4] and later the Cray XMT [5], is to mask long memory latencies by context switching between concurrent-ready threads in the processor. Traditional multithreaded architectures have a fixed data-path, configured by an instruction set, that supports a pre-determined number of concurrent threads (i.e., a fixed number of thread register files, etc). FPGAs provide an opportunity to explore the potentials of customized multithreaded architectures where the data-path, control and registers are tailored to the target computation.

In this paper we propose a customized multithreaded architecture that is implemented on an FPGA. The structure of the data-path and the number of thread states are designed for the specific target application. On this hardware architecture we have designed a novel short read alignment tool called FHAST (FPGA hardware accelerated sequencing-matching tool). FHAST implements the approximate string matching algorithm based on the FM-index data structure [6], [7] which in turn is based on the Burrows-Wheeler transform (BWT) [8]. In [9] we have described the basic algorithm, implemented on a single FPGA, with no multithreading, for finding exact matches of reads in the genome. Here we describe a multithreaded implementation for approximate string matching. The current implementation of FHAST was designed on the Convey HC-1 (originally described in [1]) and later ported on the Convey HC-2ex. Its novel features are: (1) it is multithreaded and supports up to 512 concurrently executing threads on a single accelerator FPGA of the Convey HC-1/HC-2ex; (2) it supports exact and approximate string matching (up to two mismatches), single-end and paired-end alignment, and reports any number of matched locations; (3) it is a drop-in replacement for the popular Bowtie short read alignment tool [10] (i.e., FHAST's output format is exactly the same of Bowtie so it can be used any sequence

- *E.B. Fernandez and S. Lonardi are with the Department of Computer Science and Engineering, University of California, Riverside, CA. E-mail: ecfernan@yahoo.com, stelo@cs.ucr.edu.*
- *J. Villarreal and W.A. Najjar are with the Department of Computer Science and Engineering, University of California, Riverside, CA and are also with Jacquard Computing Inc., Riverside, CA. E-mail: villarre@gmail.com, najjar@cs.ucr.edu.*

Manuscript received 11 June 2014; revised 27 Dec. 2104; accepted 6 Feb. 2015. Date of publication 23 Feb. 2015; date of current version 5 Oct. 2015.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCBB.2015.2405333

analysis pipeline where BOWTIE is used, e.g., TOPHAT for RNA-Seq). We have compared the execution times of FHAST against BOWTIE (running eight threads on a eight cores conventional architecture) for zero, one and two mismatches on the Convey HC-2ex. The observed speedup of FHAST compared a eight-threads BOWTIE to is up to about 12x. The preliminary (older) version of FHAST running on the Convey HC-1 achieved a speedup up to 70x compared to single-threaded BOWTIE.

2 METHODS

The FM-index [6], [7] is a data structure composed by the Burrows-Wheeler transform [8] of the text database (the reference genome in this case) and suffix array checkpoints [11]. The FM-index allows one to determine whether a pattern (read) occurs in the text in linear time in the size of the pattern (see [6], [7] for details). During the execution of the search, a partially-matched pattern is described by two pointers to the FM-index (called *top* and *bottom*) that specify the range of locations a suffix of the pattern appears in the text. These two pointers are updated at each processed character of the pattern (see below for the rules). If at any one time the two pointers are equal or if *top* is less than *bottom*, the search is terminated and the algorithm declares that the pattern does not occur in the text. Instead if the last character in the pattern is reached, the range between top and bottom indicates the number of occurrences of that pattern in the text. We remind the reader that in this application, not only each read (pattern) needs to be searched in the genome (text) but also its reverse complement.

We have adapted the FM-index to make it suitable for FPGA implementation. The first incarnation of our design was described in [9] (and later refined in [1]). The scheme [9] only allowed exact matching, and had no multithreading. In that scheme the Burrows-Wheeler transform of the text is represented as two tables, called C-table and I-table. The I-table is an array with a number of entries equal to the number of symbols in the alphabet (four in this case). The I-table stores the position of the first occurrence of each character in the text after the text has been sorted lexicographically. The C-table is a two dimensional matrix, with a number of rows equal to the length of the text, and a number of columns equal to the number of symbols (four in this case). If $BTW(text)$ is the Burrow-Wheeler transform of the input text (reference genome), entry (i, j) of the C-Table represents the number of occurrences of symbol j in the prefix of $BWT(text)$ of length i. Given the I-Table and the C-Table, if s is the current symbol to be processed, the rules to update top and bottom pointers are as follows

$$top_{new} = C[top_{current}, s] + I[s]$$
$$bottom_{new} = C[bottom_{current}, s] + I[s].$$

In [9] and [1], the C-table and I-table are allocated in block RAMs and LUTs of the FPGA, respectively. The main practical limitation of [9] is related to the size of memory available on the FPGA, which limited the size of the genome that could be processed. Large genomes (e.g., human genome) had to be split into chunks and processed in batches. This imposed limits on the achievable speed up of the algorithm because of the overhead involved in processing multiple chunks. The implementation described here overcomes this main limitation by using the external memory to store the C-table and uses multiple threads to hide long memory latencies.

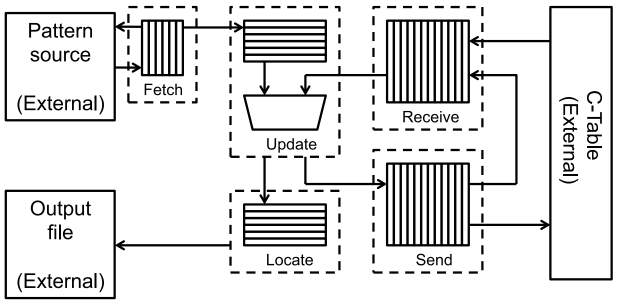

Fig. 1. The block diagram of the exact string matching architecture (patterns are the short reads, the C-table stores the FM-index of the reference genome).

2.1 Exact String Matching Architecture

In this section we briefly describe the customized multithreaded architecture that implements exact string matching in FHAST. As said, each read is processed by an individual thread.

Fig. 1 shows the block diagram implementing the exact matching engine using external memory. The implementation consists of five main blocks, namely *fetch*, *update*, *send*, *receive*, and *locate*. Each block consists of queues that are used to hide memory latency while performing other tasks. The C-table and the list of reads are allocated on external memory. The I-table is allocated on LUTs of the FPGA. The *fetch* block requests read from external memory and generates unique ID for each read, so that the system can track them.

The *update* block inserts the reads from the *fetch* block into the *send* block. The *update* block determines if a read requires further processing or if the read has been determined to be a match or mismatch.

If the read needs more processing, the *update* block forwards the read to the *send* block, which issues addresses to access the C-table for the top and bottom pointers. The I-table is also accessed simultaneously using the last character of the read. As addresses are issued to external memory, the *send* block places state information of the read into the *receive* block. The information in the *receive* block waits for data returned from external memory for further processing. Data is returned from memory in the same order it was requested.

The *send* block continuously issues addresses of different reads to the external memory and read information to the *receive* block until the address queue of the external memory or queue of the *receive* block is full. This achieves the multithreading: multiple reads are waiting in queues for memory while other reads are processed. When the memory returns the data, the *receive* block merges it with the waiting thread and passes it to the *update* block.

The *update* block determines if the processing of a read is complete. Two conditions can lead to termination. In the first, the algorithm determines that the read actually occurs

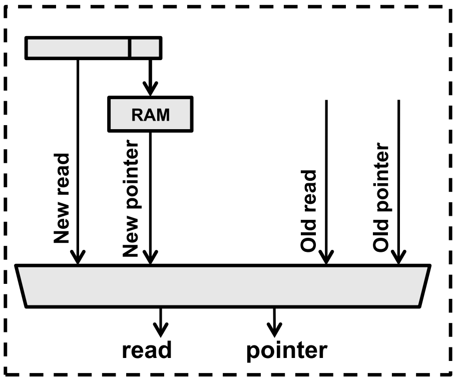

Fig. 2. The *update* block uses a block RAM of precomputed addresses to improve the performance of the string matching engine.

in the text. This happens when the two pointers, after processing the last character of the read, identify a non-empty range. In this case the read is passed into the *locate* block to report the match. In the second, the algorithm determines that the read does not occur in the text: in this case the pointers have an empty range. In both cases, a new read from the *fetch* block is introduced to keep the engine full.

The queues are used to store the request to and the responses from the memory system. As said, the idea behind the multithreading execution implemented in FHAST is to mask the memory latency and hence increase the parallelism. The memory system on the Convey machine is fully buffered and allows several hundred outstanding memory requests per memory channel. The number of outstanding memory requests, in a multithreaded system, is a measure of the effective parallelism. Whenever a queue is close to being full a stall signal is raised that propagates backwards stalling all the circuits upstream. This process continues until the fetching of new reads from memory is stopped. Such events would increase the execution time. Obviously, the design aims at avoiding such a situation by making the queue size large enough. Beyond a certain value, the size of the queue does not have any effects on the overall execution time: this is the value chosen for the queue size.

The performance of the hardware implementation strictly depends on the number of external memory requests. To reduce this number, the memory addresses are precomputed for all character combinations up to a specific length (prefix) such that each combination of characters represent a range for the C-table. Instead of initializing the address to the first and last rows of the C-table as indicated in the modified algorithm, we instead initialize the top and bottom pointers to the precomputed values.

We store the precomputed values in a block RAM and use the last l characters to access the precomputed values. Fig. 2 shows the structure of the *update* block including precomputed addresses. The *update* block decides if a new or old pattern is passed to the *send* block.

2.2 Approximate String Matching Architecture

In order to maintain modularity and future expandability, our approximate string matching architecture uses multiple exact matching engines. If n is the number of allowed mismatches, the architecture needs $n+1$ exact matching engines. Fig. 3 show the architecture that handles up to two

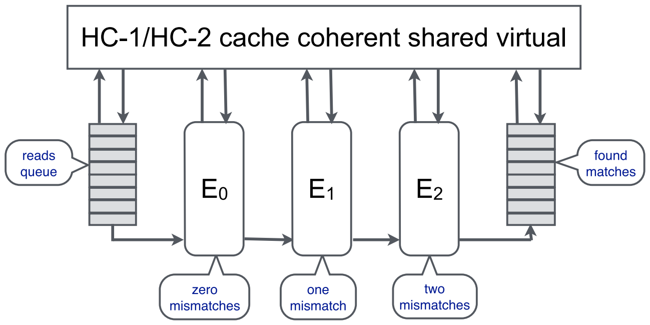

Fig. 3. Block diagram of approximate matching architecture using n exact string matching architectures.

mismatches and illustrates the connections between the three exact matching engines. Engine E_0 handles exact string matching and requests reads from external memory. If a read fails on engine E_0 at some position k, it is passed to engine E_1. Engine E_1 receives the read, the value $k-1$, and the value of *top* and *bottom* pointers for character $k-1$. The data is passed to a *replace* block that searches the pattern with up to one mismatch. The heuristic search implemented in the *replace* block is discussed below. A failing read on engine E_1 is passed to engine E_2 to detect hits with two mismatches. Reads that pass any of the three matching engines are moved to the *locate* block that determines their location in the genome.

As said, the *replace* block executes a heuristic to find approximate matches. If a read failed on engine E_0, the algorithm creates three copies of the read with the same ID. The failing character of each copy is replaced by the other three nucleotide symbols. Each copy becomes a new thread and it is inserted in the queues of searching blocks. Fig. 4 illustrates the replacement of the failing character by other characters.

A "copy" flag is set for each new read copy as it is inserted in the queue of the *update* block for engine E_i. If a read with the copy flag fails on its first iteration on the new engine E_i, the read is terminated immediately and it is not passed to engine E_{i+1}. If a read with the copy flag succeeds on its first iteration on engine E_i, then the copy flag is reset. If the reads fails on any of the following iterations, new copies are created again and passed to engine E_{i+1}.

Fig. 5 shows the block diagram of the *replace* block inserted is the *update* block of engine E_1. Engine E_1 accepts

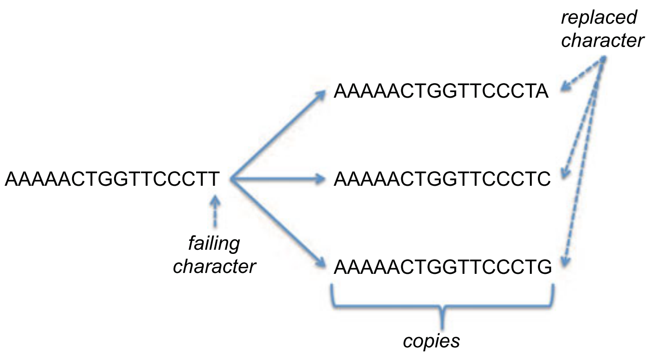

Fig. 4. A failing character in the read induces three new copies of the read, where each read has the failing character replaced by one of the other three nucleotides.

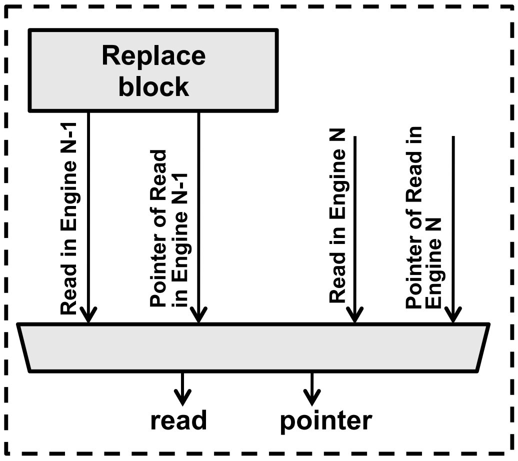

Fig. 5. Update block including the replace block for approximate matching.

reads from the *replace* block instead of the *fetch* block. The *update* block then selects reads from the previous engine when processing of a read on engine E_1.

Reads that exit from the matching engines are passed to a *locate* block that identifies the location of the occurrences of the read in the genome. The data passed to the *locate* block are the read ID and the values of the two pointers. The architecture of the *locate* block is similar to *send* and *receive* blocks. Fig. 6 shows the block diagram of the *locate* block: it consists of queues for sending addresses to external memory and waiting for data returning from memory.

The *locate* block sends the top pointer as an address to the suffix array placed in external memory. The external memory returns the location which is later written to the output file. If a read exists at multiple locations, the algorithm sends multiple address requests (until it reaches the bottom pointer) to the memory for the required locations.

Our approximate string matching heuristics is somewhat different from that of BOWTIE [10]. BOWTIE also uses the exact matching algorithm based on the FM-index for successively longer query suffixes of the pattern. However, if the range between top and bottom pointers becomes empty (i.e., that suffix does not occur in the text), then BOWTIE selects an already-matched query position and substitute a different symbol there, introducing a mismatch into the alignment. BOWTIE only selects the position(s) which yield a modified suffix that occurs at least once in the text. The exact-matching search resumes from just after the substituted position.

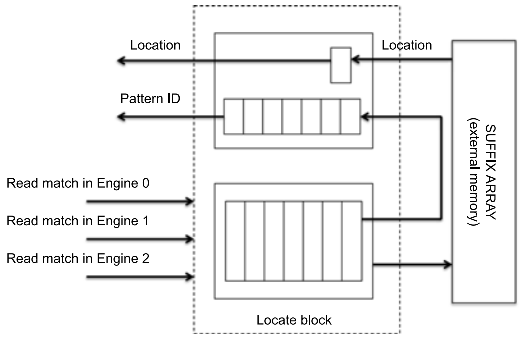

Fig. 6. The *locate* block have send and receive queues similar to *send* and *receive* blocks for finding the location of a read from the FM-index stored in main memory.

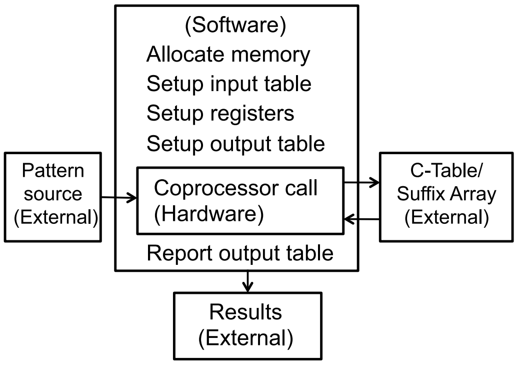

Fig. 7. The software is responsible on memory allocation and reporting, while the hardware carries out the approximate searching algorithm.

In contrast, in order to maximize the efficiency of the backtracking implementation in hardware, FHAST only substitutes the last symbol where a mismatch occurred. As a consequence, some of the alignments reported by BOWTIE can be missed by FHAST, in particular alignment involving two mismatches. In order to reduce the number of missed two-mismatches alignments, we split a read in half, and we use two additional engines to match them against the reference genome, starting from the center and working outwards.

3 Experimental Results

In this section we describe the implementation of FHAST on the Convey Computers HC-1, and later ported on the Convey Computers HC-2ex.

Fig. 7 illustrates the role of the hardware in searching for the reads on the text. The software performs memory allocation for the C-table, the suffix array, and the reads. It also takes care of writing the results to external memory. After allocating the necessary memory and setting up the coprocessor registers, the host CPU calls the coprocessor to perform the approximate search algorithm. Software then writes results to an output file in BOWTIE format.

We initially conducted our experiments on the Convey HC-1 hybrid core computing system. The Convey HC-1 is composed of a dual core Intel Xeon processor running at 2.13 GHz as the host processor and four Xilinx Virtex 5 FPGAs as coprocessor. The Convey HC-2ex has instead 4 Virtex 6 FPGAs (see Fig. 8). Both systems have a memory with peak bandwidth of 76.8 GB/s and a memory clock rate of 150 MHz. All processors, both host processor and FPGA coprocessors, have one shared cache coherent memory. We implemented a design for a read length up to 101 symbols that supports up to two mismatches using only one FPGA in the coprocessor. The design is synthesized with place and route: on the Xilinx Virtex 5 (XC5VLX330) FPGA, the designed occupied about 46 percent of the FPGA. We set the frequency to 150 MHz, that is the maximum operating frequency of the memory controllers of the Convey HC-1/HC-2ex.

3.1 Results on the Convey Computers HC-1

In this first set of experiments, we compared the performance of FHAST to BOWTIE [10], which is the most popular tool to map reads to a reference genome, along with BWA. We executed BOWTIE on two general-purpose architectures,

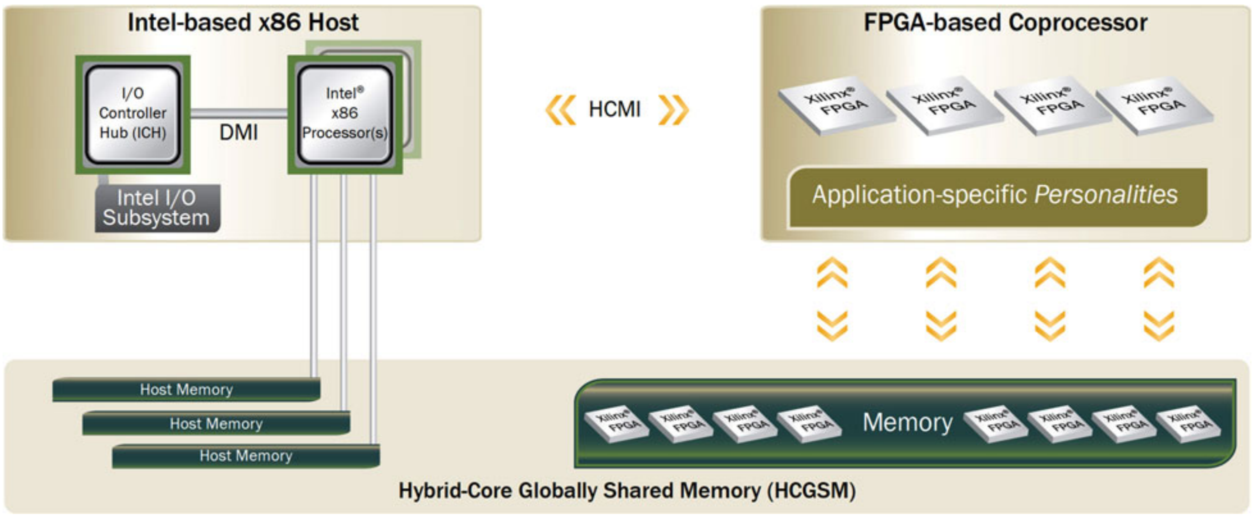

Fig. 8. Architecture of the Convey HC-2ex heterogeneous multiprocessor (figure courtesy of Convey Computers).

that we named below CPU1 and CPU2. CPU1 is the general-purpose CPU on the Convey HC-1 (2.13 GHz Xeon L540B, two dual-cores, 6 MB Cache, with 192 GB of RAM). CPU2 is a 2.27 GHz Xeon E5520, two quad-cores, 8 MB cache, with 24 GB of RAM.

We measured the execution time of FHAST and BOWTIE on a data set composed by 18 million reads where each read was 101 bases long, against human chromosome 14 (about 107 Mb). Table 1 shows the execution time of FHAST and BOWTIE (on CPU1 and CPU2) allowing up to mismatches. The table shows longer execution time for BOWTIE than FHAST. Observe that while there is a significant difference in the execution time of FHAST between searching exact and approximate matches, there is no significant difference between searching for one and two mismatches. The execution time for BOWTIE instead increases significantly as more mismatches are allowed. Fig. 9 shows the speed up of FHAST over BOWTIE for the two CPUs. Observe that the highest speed up is achieved in detecting two mismatches. By masking memory latency, the customized multithreading approach allows us to achieve up to 70x speedup on a 150 MHz FPGA over conventional CPUs.

We have also evaluated FHAST on all four accelerator FPGAs (AEs) of the Convey HC-1. The execution of FHAST relies on pre- and post-processing of the read data in software. The breakdown of the FHAST execution time, in software and hardware, shows the software phases are the limiting factor but the hardware achieves a near linear speedup (data not shown).

TABLE 1
Comparing the Execution Time of FHAST and BOWTIE; FHAST is Running on a Single FPGA, BOWTIE on a Single Core; the Data Set was Composed by 18 Million Reads (101 Bases Each) Matched against Human Chromosome 14

# mismatches	FHAST (*sec*)	BOWTIE (*sec*)	
		CPU1	CPU2
0	55	715	404
1	71	1,924	1,142
2	73	5,410	3,698

The multithreaded execution time for BOWTIE, using up to 16 cores, and the speedup over a single thread, on the same data set is shown in Table 2. The execution time of FHAST on four FPGAs is 138 seconds which is 2.43x lower than that of BOWTIE with 16 cores (336 seconds).

3.2 Results on the Convey Computers HC-2ex

On the second round experiments following the publication of the preliminary version of this manuscript [1], we ported FHAST on the Convey Computers HC-2ex. In this new set of experiments, we used the Convey HC-2ex for both the software (CPU based) as well as the hardware (FPGA based) implementations. As said, the HC-2ex has four Virtex-6 LX760 FPGAs in the AE, two Intel Xeon processors E5-2643 four-core 3.3 GHz (used in the software implementation), and 96 GB shared memory (used by both implementations). FPGAs are running at 150 MHz. BOWTIE was run in multithreaded mode, namely with eight threads on four cores, which we determined to be the optimal number of threads for BOWTIE on this architecture.

The only significant hardware change in porting the software to the HC-2ex was the addition of two additional engines to improve the accuracy of FHAST to handle the

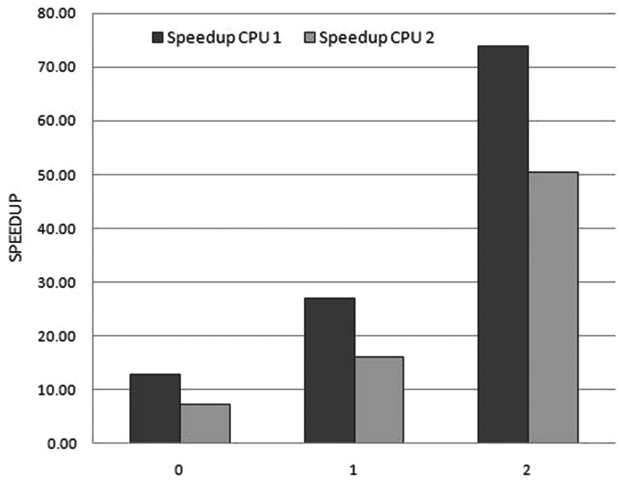

Fig. 9. Speed up of FHAST compared to BOWTIE for exact matches, one and two mismatches.

TABLE 2
Multithreaded Execution Time of Bowtie: Speedup over Single Thread

# Threads	1	2	4	8	16
Time (*sec*)	3,325	1,772	896	501	336
Speedup	1	1.88	3.71	6.64	9.90

two-mismatches case. We noticed in the previous implementation [1] that the number of hits reported by FHAST for two mismatches was quite low compared to Bowtie. As said above, in this new design in order to handle the two-mismatches case we split a read in half, and we use two additional engines to match them against the reference genome, starting from the center and working outwards. With this heuristics FHAST was able to catch more two-mismatch hits that were obscured by the prefix calculations, but incurred additional computational cost. In software we added more options to FHAST, such as supporting the `fastq` format, as well paired-end reads alignment.

End-to-end experimental results are reported in Tables 3-6. As said, the reported speedup is compared to Bowtie running on eight threads. The conversion from `fastq` to a bit-stream is done in FHAST in software and incurs a significant overhead compared to processing raw files. Tables report the performance of FHAST for both file-types. Reads for *Staphylococcus aureus* and *Rhodobacter sphaeroides* were obtained from `gage.cbcb.umd.edu` [12]. Human reads were obtained from NIH Sequence Read Archive (accessions ERR231578, ERR231579 and ERR231582). *Staphylococcus aureus*' genome is about 2.8 Mb, *Rhodobacter sphaeroides*' genome is 4.6 Mb, and *Homo sapiens*' genome about 3,234 Mb. The observed speed-up ranges from 0.53 to 11.99X. Observe that (1) for most of the cases, FHAST's speedup gets higher as the number of mismatches increase; (2) FHAST's advantage over Bowtie increases on larger genomes. In general, FHAST is faster than Bowtie running on eight-threads on eight-cores conventional architecture.

4 Tool Availability

FHAST is available in the public domain and free for academic use. FHAST can be downloaded from http://www.cs.ucr.edu/~stelo/pub/Bio-FHAST.tar.gz The compressed archive contains FHAST, some utilities and a user manual. FHAST runs on Convey Computers HC-1/HC-2ex. As in Bowtie, FHAST requires one to build the `ebwt` (FM-index) for the reference genome using the `bowtie_build` tool.

TABLE 3
Performance Statistics for *Staphylococcus Aureus* Reads (Reporting All Locations)

				0 mismatches		1 mismatch		2 mismatches	
Data set	format	len	# reads	Speedup	mapped	Speedup	mapped	Speedup	mapped
fragments	fastq	101	2,588,208	1.83X	100%	1.68X	99.2%	3.02X	96.9%
shortjump	fastq	37	6,988,140	2.55X	100%	1.87X	95.4%	3.68X	91.5%
fragments	raw	101	2,588,208	0.87X	100%	0.93X	99.6%	1.83X	98.9%
shortjump	raw	37	6,988,140	0.79X	100%	0.64X	98.8%	1.58X	95.9%

TABLE 4
Performance Statistics for *Rhodobacter Sphaeroides* Reads (Reporting All Locations)

				0 mismatches		1 mismatch		2 mismatches	
Data set	format	len	# reads	Speedup	mapped	Speedup	mapped	Speedup	mapped
fragments	fastq	101	4,101,736	1.65X	100%	1.71X	99.7%	3.37X	98.8%
shortjump	fastq	101	4,101,736	1.96X	100%	1.66X	95.3%	3.37X	93.5%
fragments	raw	101	4,101,736	0.73X	100%	0.75X	99.9%	1.54X	99.6%
shortjump	raw	101	4,101,736	0.79X	100%	0.64X	99.9%	1.44X	99.5%

TABLE 5
Performance Statistics for Human Reads Mapped to Chromosomes 14 (Reporting One Location, Reads are 101 bp); for Reference, on ERR231578 (Raw), Bowtie Took 193.8s (0 Mismatches), 186.9s (1 Mismatch), 443.7s (2 Mismatches), while Fhast Took 31.7s (0 Mismatches), 32.7s (1 Mismatch), 37.0s (2 Mismatches)

			0 mismatches		1 mismatch		2 mismatches	
Data set	format	# reads	Speedup	mapped	Speedup	mapped	Speedup	mapped
ERR231578	fastq	57,767,429	2.37X	100%	2.36X	99.92%	4.16X	99.12%
ERR231579	fastq	62,851,893	2.41X	100%	2.33X	99.92%	4.11X	99.02%
ERR231582	fastq	50,781,875	2.28X	100%	2.33X	99.84%	4.21X	99.62%
ERR231578	raw	57,394,559	6.12X	100%	5.71X	100%	11.99X	99.22%
ERR231579	raw	62,445,390	6.62X	100%	4.99X	100%	11.75X	99.13%
ERR231582	raw	50,596,558	6.43X	100%	5.45X	100%	11.81X	99.82%

TABLE 6
Performance Statistics for Human Reads Mapped to the Reference Genome (Reporting One Location, Reads are 101 bp)

			0 mismatches		1 mismatch		2 mismatches	
Data set	format	# reads	Speedup	mapped	Speedup	mapped	Speedup	mapped
ERR231578	fastq	57,767,429	1.16X	100%	1.16X	99.92%	3.50X	99.20%
ERR231579	fastq	62,851,893	1.24X	100%	1.09X	99.92%	3.58X	99.18%
ERR231582	fastq	50,781,875	1.08X	100%	1.05X	99.84%	3.25X	99.63%
ERR231578	raw	57,394,559	1.22X	100%	1.04X	100%	7.36X	99.33%
ERR231579	raw	62,445,390	1.24X	100%	1.19X	100%	4.85X	99.31%
ERR231582	raw	50,596,558	1.23X	100%	1.05X	100%	4.53X	99.87%

5 Related Works

The first use of FPGAs in bioinformatics and computational biology appeared in early nineties, with the objective to accelerate DNA sequence alignment [13]. In fact, a significant body of work focused on accelerators for dynamic programming algorithms such as the Smith-Waterman [14] algorithm (see, e.g., [15], [16], [17], [18], [19]). For instance, in [15], the authors use a systolic array to take advantage of parallelism inherent to the algorithm. In [18], the systolic array structure is automatically generated using a compiler. In [19], the Smith-Waterman algorithm is implemented on a supercomputing platform using FPGAs as coprocessors. The platform included a highly pipelined system that reduces FPGA resource utilization.

Algorithms based on "seeds-and-extend that perform DNA sequence matching (BLAST-like) have also been explored (see, e.g., [20], [21]). For instance, in [20], the objective is to accelerate the seed generation phase of BLAST. In [21], BLAST is implemented on an FPGA with an optimized verification phase.

Besides using seeds and dynamic programming, finite automata have also been used for exact sequence matching. The implementation of the Aho-Corasick algorithm [22] on FPGAs has been explored in [23] where protein sequences are matched on a reference genome. A brute force approach has also been implemented in [9]. In this approach, the patterns to be searched are placed in hardware registers and the genome is streamed through the hardware infrastructure.

More recently, the focus has shifted to the analysis of next-generation sequencing data, and specifically to the problem of mapping short-reads to a reference genome. A significant number of papers have been published that either use FPGAs (see, e.g., [1], [9], [24], [25], [25], [26], [27], [28], [29], [30], [31]) or GPUs (see, e.g., [32], [33]). For instance in [24], the authors propose a hybrid system for short read mapping utilizing both FPGA-based hardware and CPU-based software: the hardware implements parallel block-wise alignment structure to approximate the conventional dynamic programming algorithm. In general, the claimed speedups compared to software solution for all the proposed FPGAs designs in the literature has a wide range and it is not easy to compare between them because they are tested on different data sets. As our results demonstrate, the performance of these systems critically depends on the size of the reference genome, the number of allowed mismatches, the number and the length of the reads. It also depends on whether we are supposed to return all the hits or a fixed number of them. The sensitivity of the mapping tool (i.e., the number of hits found compared to Bowtie or BWA) is also often not reported. In [26], the authors introduce a FPGA-based solution to the short read mapping problem which achieves a 31x speedup versus Bowtie on eight CPU cores. In [27], the speedup of their accelerator over a six-cores CPU ranges from 22.2x to 42.9x. In [28], the author focus on long reads mapping: their FPGA-based platform achieves a 1.8x-3.2x speedup versus the BWA-SW aligner. In [29], the actors claim that their FPGA tool to be up to 293 times faster than BWA (single-threaded) on an Intel X5650 CPU and 134 times faster than SOAP3 on an NVIDIA GTX 580 GPU. In [30], the proposed single FPGA is populated with specialized filters based on a novel bidirectional backtracking version of the FM-index. Their alignment time can be up to 18.1 times faster than BWA running on dual Intel X5650 CPUs.

A good survey on hardware acceleration for computational genomics and bioinformatics appeared recently in [34].

6 Conclusion

In this paper we have described and demonstrated an FPGA-based customized multithreading solution to the problem of short-read mapping. We compared the FHAST's execution time and output results to Bowtie, which is a widely used tool for sequencing reads. Preliminary experimental results on the Convey Computers Hc-1 show that FHAST achieves a speedup of up to 70x over Bowtie. The new version that runs on the Convey Computers HC-2ex has higher sensitivity for higher number of mismatches, and the speed up compared to Bowtie running on eight-cores is up to 12x. In both cases, allowing more mismatches increases the speed up compared to Bowtie. This is because the execution time of Bowtie dramatically increases while only a minimal increase in execution time is observed in FHAST. FHAST could handle even a higher number mismatches by adding more engines without any significant increase in execution time (provided that sufficient area is available on the FPGAs). The overall performance of FHAST is somewhat limited by the computational cost of the software to pre- and post-processing necessary to prepare the data and format the results in the same format as Bowtie. FHAST is a drop-in replacement for Bowtie: it reads input reads in `fastq` format, it uses the FM-index produced by Bowtie, it handles paired-end read mapping, and produces the same output format of Bowtie.

ACKNOWLEDGMENTS

In this work W. Najjar and E. Fernandez were supported in part by US National Science Foundation Awards 0905509, 0811416 and by a Cisco Systems research grant. S. Lonardi was supported by NIH Award AI85077-01A1 and US National Science Foundation Awards DBI-1062301, IIS-1302134. J. Villarreal was supported at Jacquard Computing Inc. by AFRL contract FA9453-09-C-0173. A preliminary version of this work was presented at the IEEE High Performance Extreme Computing Conference 2012 (HPEC'12), Waltham, CA, and appeared in its *Proceedings* [1]. S. Lonardi is the corresponding author.

REFERENCES

[1] E. Fernandez, W. Najjar, S. Lonardi, and J. Villarreal, "Multithreaded FPGA acceleration of DNA sequence mapping," in *Proc. IEEE Conf. High Perform. Extreme Comput.*, Sep. 2012, pp. 1–6.
[2] M. C. Schatz and B. Langmead, "The DNA data deluge," *IEEE Spectrum*, vol. 50, no. 7, pp. 28–33, Jul. 2013.
[3] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith, "Exploiting heterogeneous parallelism on a multithreaded multiprocessor," in *Proc. 6th Int. Conf. Supercomput.*, 1992, pp. 188–197.
[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The tera computer system," in *Proc. 4th Int. Conf. Supercomput.*, 1990, pp. 1–6.
[5] J. Feo, D. Harper, S. Kahan, and P. Konecny, "Eldorado," in *Proc. 2nd Conf. Comput. Frontiers*, 2005, pp. 28–34.
[6] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. 41st Annu. Symp. Found. Comput. Sci.*, 2000, pp. 390–398.
[7] P. Ferragina and G. Manzini, "An experimental study of an opportunistic index," in *Proc. 12th Ann. ACM-SIAM Symp. Discr. Algorithms*, 2001, pp. 269–278.
[8] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, CA, USA, Tech. Rep. 124, 1994.
[9] E. Fernandez, W. Najjar, E. Harris, and S. Lonardi, "Exploration of short reads genome mapping in hardware," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Aug. 31, 2010-Sep. 2, 2010, pp. 360–363.
[10] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biol.*, vol. 10, no. 3, p. R25, 2009.
[11] U. Manber and G. Myers. (1990). Suffix arrays: A new method for on-line string searches," in *Proc. 1st Ann. ACM-SIAM Symp. Discr. Algorithms*, pp. 319–327. [Online]. Available: http://dl.acm.org/citation.cfm?id=320176.320218
[12] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, G. Marais, M. Pop, and J. A. Yorke. (2011). GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research* [Online]. *22(3)*, Available: http://genome.cshlp.org/content/early/2012/01/12/gr.131383.111.abstract
[13] D. Hoang, "Searching genetic databases on Splash 2," in *Proc. IEEE Workshop FPGAs Custom Comput. Mach.*, Apr. 1993, pp. 185–191.
[14] T. Smith and M. Waterman. (1981). Identification of common molecular subsequences. *J. Molecular Biol.* [Online] *147 (1)*, pp. 195–197. Available: http://www.sciencedirect.com/science/article/pii/0022283681900875
[15] G. Caffarena, S. Bojanic, J. A. Lopez, C. Pedreira, and O. Nieto-Taladriz, "High-speed systolic array for gene matching," in *Proc. ACM/SIGDA 12th Int. Symp. Field Programmable Gate Arrays*, 2004, pp. 248–248.
[16] M. Gok and C. Yilmaz, "Efficient cell designs for systolic smith-waterman implementations," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Aug. 2006, pp. 1–4.
[17] K. Benkrid, Y. Liu, and A. Benkrid, "A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 4, pp. 561–570, Apr. 2009.
[18] B. Buyukkurt and W. Najj, "Compiler generated systolic arrays for wavefront algorithm acceleration on FPGAs," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Sep. 2008, pp. 655–658.
[19] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform," in *Proc. 1st Int. Workshop High-Perform. Reconfigurable Comput. Technol. Appl.: Held Conjunction SC07*, 2007, pp. 39–48.
[20] A. Jacob, J. Lancaster, J. Buhler, and R. Chamberlain, "FPGA-accelerated seed generation in mercury blastp," in *Proc. 15th Annu. IEEE Symp. Field-Programmable Custom Comput. Mach.*, Apr. 2007, pp. 95–106.
[21] M. Herbordt, J. Model, Y. Gu, B. Sukhwani, and T. VanCourt, "Single pass, blast-like, approximate string matching on FPGAs," in *Proc. 14th Annu. IEEE Symp. Field-Programmable Custom Comput. Mach.*, Apr. 2006, pp. 217–226.
[22] A. V. Aho and M. J. Corasick. (1975, Jun.). Efficient string matching: An aid to bibliographic search. *Commun. ACM* [Online]. *18(6)*, pp. 333–340. Available: http://doi.acm.org/10.1145/360825.360855
[23] Y. Dandass, S. Burgess, M. Lawrence, and S. Bridges, "Accelerating string set matching in FPGA hardware for bioinformatics research," *BMC Bioinform.*, vol. 9, no. 1, p. 197, 2008.
[24] Y. Chen, B. Schmidt, and D. Maskell. (2013). A hybrid short read mapping accelerator. *BMC Bioinform.* [Online]. *14(1)*, p. 67. Available: http://www.biomedcentral.com/1471-2105/14/67
[25] O. Knodel, T. Preusser, and R. Spallek, "Next-generation massively parallel short-read mapping on FPGAs," in *Proc. IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors*, Sep. 2011, pp. 195–201.
[26] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *Proc. IEEE 20th Int. Symp. Field-Programm. Custom Comput. Mach.*, 2012, pp. 161–168.
[27] W. Tang, W. Wang, B. Duan, C. Zhang, G. Tan, P. Zhang, and N. Sun, "Accelerating millions of short reads mapping on a heterogeneous architecture with FPGA accelerator," in *Proc. 20th Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, Apr. 2012, pp. 184–187.
[28] P. Chen, C. Wang, X. Li, and Z. Xuehai, "Accelerating the next generation long read mapping with the FPGA-based system," *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 11, no. 5, pp. 840–852, Sep./Oct. 2014.
[29] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Reconfigurable acceleration of short read mapping," in *Proc. IEEE 21st Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, Apr. 2013, pp. 210–217.
[30] J. Arram, W. Luk, and P. Jiang, "Reconfigurable filtered acceleration of short read alignment," in *Proc. Int. Conf. Field-Programmable Technol.*, Dec. 2013, pp. 438–441.
[31] Y. Xin, B. Liu, B. Min, W. X. Li, R. C. Cheung, A. S. Fong, and T. F. Chan. (2013). Parallel architecture for DNA sequence inexact matching with burrows-wheeler transform. *Microelectron. J.* [Online]. *44(8)*, pp. 670–682. Available: http://www.sciencedirect.com/science/article/pii/S0026269213001225
[32] J. Salavert Torres, I. Blanquer Espert, A. Tomas Dominguez, V. Hernendez, I. Medina, J. Terraga, and J. Dopazo, "Using GPUs for the exact alignment of short-read genetic sequences by means of the Burrows–Wheeler transform," *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 9, no. 4, pp. 1245–1256, Mar. 2012.
[33] C. M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T. W. Lam, "SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, Mar. 2012.
[34] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *IEEE Design Test*, vol. 31, no. 1, pp. 19–30, Jan./Feb. 2014.

Edward B. Fernandez received the PhD degree from the Department of Computer Science and the Engineering at University of California, Riverside, CA. He is currently with Freescale Semiconductor Inc.

Jason Villarreal recievd the BS, MS, and PhD degrees from the Department of Computer Science and Engineering at the University of California, Riverside, CA. He is currently with Xilinx Inc.

Stefano Lonardi received the *Laurea cum laude* from the University of Pisa, Italy in 1994 and the PhD degree in the summer of 2001 from the Department of Computer Sciences, Purdue University, West Lafayette, IN. He also received the doctorate degree in electrical and information engineering from the University of Padua, Italy, in 1999. He is the professor and the vice chair of the Department of Computer Science and Engineering at University of California, Riverside, CA. During the summer of 1999, he was an intern at Celera Genomics, the Department of Informatics Research, Rockville, MD. His research interest includes design of algorithms, computational molecular biology, data compression and data mining. He has published more than 100 papers in computational biology, data mining, and theoretical computer science journals and refereed international conferences. In the year 2005, he received the CAREER award from National Science Foundation. He has received funding from US National Science Foundation, NIH, DARPA, USAID and USDA. He currently serves on the Steering Committee of this journal. He is a senior member of the IEEE.

Walid A. Najjar received the BE degree in electrical engineering from the American University of Beirut in 1979, and the MS and PhD degrees in computer engineering from the University of Southern California in 1985 and 1988, respectively. He is a professor in the Department of Computer Science and Engineering at the University of California Riverside. His areas of research include computer architectures and compilers for parallel and high-performance computing, embedded systems, FPGA-based code acceleration, and reconfigurable computing. From 1989 to 2000, he was on the faculty of the Department of Computer Science at Colorado State University, before that he was with the USC-Information Sciences Institute. He is a fellow of the IEEE and the AAAS.