Proc. of IEEE 2012 High Performance Extreme Computing Conference,
Waltham, MA, September 11 - 12, 2012.

Multithreaded FPGA Acceleration of DNA Sequence Mapping

Edward B. Fernandez, Walid A. Najjar, Stefano Lonardi
University of California Riverside
Riverside, USA
{efernand,najjar,lonardi}@cs.ucr.edu

Jason Villarreal
Jacquard Computing Inc.
Riverside, USA
Jason@jacquardcomputing.com

***Abstract*—In bioinformatics, short read alignment is a computationally intensive operation that involves matching millions of short strings (called *reads*) against a reference genome. At the time of writing, a representative run requires to match tens of millions of reads of length of about 100 symbols against a genome that can consists of a few billion characters. Existing short read aligners are expected to report all the occurrences of each read as well as allow users to control the number of allowed mismatches between reads and reference genome. Popular software implementations such as Bowtie [8] or BWA [10] can take many hours or days to execute, making the problem an ideal candidate for hardware acceleration. In this paper, we describe FHAST (FPGA Hardware Accelerated Sequencing-matching Tool), a hardware accelerator that acts as a drop-in replacement for short read alignment software. Our architecture masks memory latency by executing many concurrent hardware threads accessing memory simultaneously and consists of multiple parallel engines to exploit the parallelism available to us on an FPGA. We have implemented and tested FHAST on the Convey HC-1 [9], taking advantage of the large amount of memory bandwidth available to the system and the shared memory image between hardware and software. By comparing the performance of FHAST against Bowtie on the Convey HC-1 we observed up to ~70X improvement in total end-to-end execution time, reducing runs that take several hours to a few minutes. We also favorably compare the rate of growth when expanding FHAST to utilize multiple FPGAs against multiple CPUs in Bowtie.**

***Index Terms*—bioinformatics, short read matching, hardware acceleration, FPGA, multithreaded.**

I. Introduction

Besides the sheer volume of data, one major challenge of big data and data intensive applications is that they are irregular. Traditional techniques for exploiting locality, such as caching, are 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 [1, 2] and later the Cray XMT [3], 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 supported are designed for the specific target application. We evaluate this model using a novel hardware accelerated DNA sequence matching tool called FHAST (*FPGA Hardware Accelerated Sequencing-matching Tool*). FHAST implements a heuristic based on the FM-index string-matching algorithm [4,5] operating on the Burrows-Wheeler transform (BWT) of the genome [6]. In [7] we have described the algorithm, implemented on an FPGA, with no multithreading, for finding exact matches of reads in the genome. The current implementation of FHAST is implemented on the Convey Computer HC-1. Its novel features are: (1) it is multithreaded and supports up to 512 concurrently executing threads on a single accelerator FPGA of the HC-1. (2) It supports exact as well as approximate matches with one and two mismatches and reports any number of matched locations. (3) It can be used as a drop-in replacement for the Bowtie sequence-matching tool [8]. We have compared the execution times of FHAST to Bowtie for zero, one and two mismatches. The observed speedup ranges from 7x to 70x.

The rest of this paper is organized as follows: Section II discusses the background of FM-Index as a searching algorithm. Section III and IV discusses of implementation of exact and approximate matching. Section V discusses our experiment setup and evaluation of results. Section VI reports on related work and Section VII states our conclusions.

II. The FM-index String Matching Algorithm

The FM-index string-matching algorithm [3] operates on the Burrows-Wheeler transform (BWT) [1] of the text. The data structure of the FM-index that indicates a matching pattern is composed of two pointers (*top* and *bottom*) that specify the range of locations a pattern of specific length appears in the text. These two pointers are updated with every character of the *read* (the read is the short string being matched against the text of the genome). If at any one time the two pointers are equal or if *top* is less than *bottom*, the search is terminated and that read does not exist in the genome. After processing the last character in the read, the range between top and bottom indicates the number of locations that read exists on the genome.

We have modified this algorithm to make it suitable for FPGA implementation [7]. In that scheme the Burrows-Wheeler transform (BWT) of the text is represented as two tables: C-table and I-table. The C and I tables are placed in block RAMs and LUTs of the FPGA respectively. Hardware accesses these two tables during the search.

The main limitation of our approach in [7] is the size of memory available on the FPGA. Very large texts must be divided into sections and processed in batches. This limits the inherent speed up of the algorithm because the search is effectively performed on sections sequentially instead of performing the entire search on the text. Our implementation will remedy this main limitation by using external memory to store the C-table and use multiple threads to hide long memory latencies, each thread representing one read during the search. This is achieved through queues where data requested in memory is returned in the same order on how it is requested.

III. String Matching Architecture

In this section we describe the customized multithreaded architecture that implements the FHAST system. Each read is a thread.

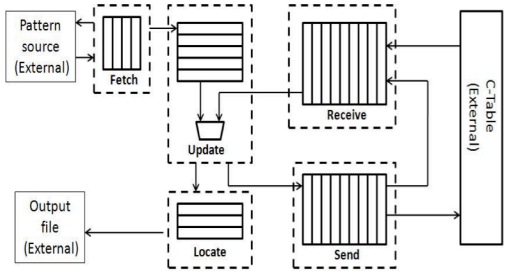

Figure 1: Block diagram of exact string matching algorithm using external memory.

A. Architecture

Figure 1 shows the block diagram implementing exact matching using external memory. The implementation consists of five main blocks: the *fetch*, *update*, *send*, *receive*, and *locate* blocks. Each block consists of queues that are used to hide memory latency while performing other tasks. The C-table and list of reads are placed on external memory. The I-table is placed on LUTs of the FPGA. The *fetch* block requests external memory for readsand generates an ID for each read.

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 intothe *receive* block.This information in the *receive* block waits for data returned from external memory for further processing. Data is returned from memory in issue order.

The *send* block continuously issuesaddresses of different reads to 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 multithreaded search where multiple reads are waiting in queues for memory while other reads are processed. When memory returns data the *receive* block merges it with the waiting thread and passes it to the *update* block.

The *update* block decides if a processing of a read terminates based on one of two conditions: 1) The read is determined to have existed on the text. This happens when the two pointers, after processing the last character of the read, represent a range. In this case the read is passed into the locate block to report the match. 2) A read does not exist on the text when data returned from memory represent an empty range of locations where a read can possibly occur. In this case the read is discarded. In both cases, a new read fro m the *fetch* block is introduced to keep the engine full.

B. Improving Performance

The performance of the hardware previously discussed highly depends on the number of external memory requests. To reduce this number, the memory address are precalculated for all character combinations up to a specificlength such that each combination of characters represent a range for every 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 precalculated values.

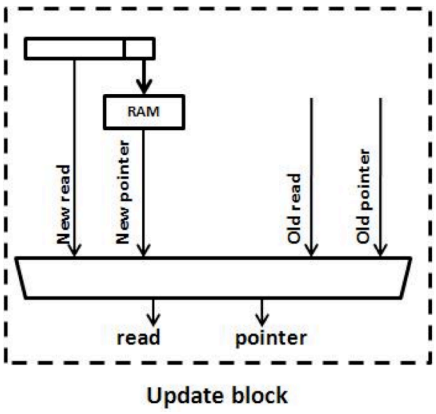

Figure 2: Update block including a block RAM for pre-calculated addresses.

We store the precalculated values in a block RAM and use the last *l* characters to access the precalculated values. Figure 2 shows the a detailed *update* block including precalculation. The *update* block decides if a new or old pattern is passed to the *send* block.

IV. Overall Approximate String Matching Architecture

For minimal change to the structure of the exact matching architecture, we use multiple exact matching engines to expand the capability to approximate matching. For every n allowed mismatch, we use $n + 1$ exact matching engines. Figure 3 is a

block diagram that shows the connection between exact matching engines.

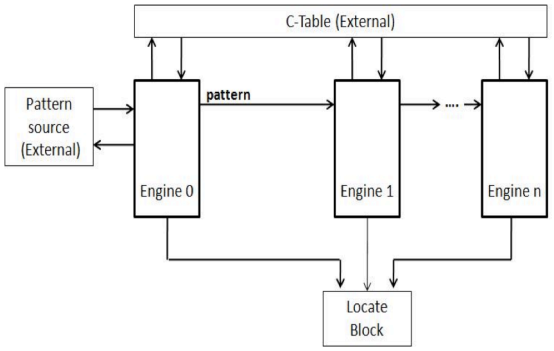

Figure 3: Block diagram of approximate matching architecture using n *exact string matching architectures.*

A. Architecture

Engine 0 handles exact string matching and requests reads from external memory. When a read on iteration *k* fails on Engine 0, it is passed to Engine 1. The important information passed to the next engine is: the read, the iteration count

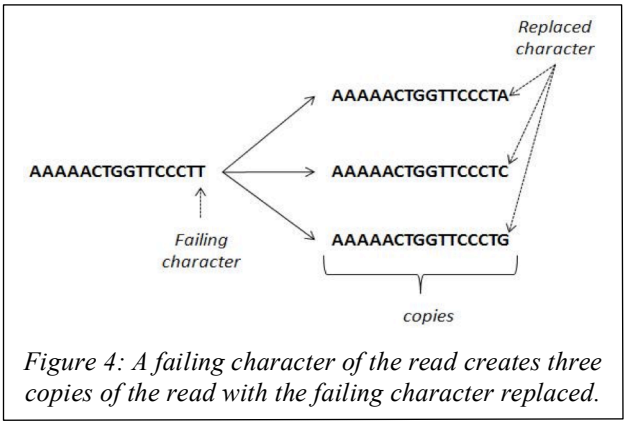

Figure 4: A failing character of the read creates three copies of the read with the failing character replaced.

reduced by one, and address of pointers on iteration *k-1*. The data passed are inputted to a *replace* block that follows a heuristic that enables mismatched search. The heuristic and the *replace* block are discussed in the next section. Reads that successfully pass Engine 1 register as a matching read with one mismatching character. A failing read on Engine 1 is passed to Engine 2 to detect reads with two mismatching characters. Reads that pass the three matching engines are all passed to the locate block that determines the location of the read in the text.

B. Replace Block

The *replace* block executes the heuristic allowing approximate matching. The heuristic creates three copies of the failing read on Engine 0 with each copy having the same accepted ID. The failing character of each copy is replaced by the other three characters of the alphabet. Each copy becomes a thread and inserted to the queues of the searching blocks. Figure 4 shows an example of the heuristic showing the replacement of the failing character by other characters.

A flag *f* is set for each read copy as each is inserted to the queue of the *update* block of Engine 1. This flag indicates that

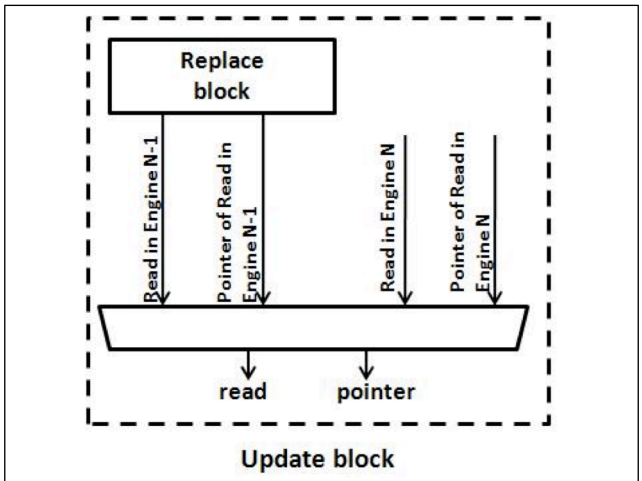

Figure 5: Update *block including the* replace *block for approximate matching.*

a read copy is recently inserted to Engine 1. If the copy fails on its first iteration on the new engine, the copy finishes processing and is no longer passed to a succeeding engine. If the copy proceeds to the next iteration on Engine 1, then the flag *f* of the copy is reset. If the copy fails on any succeeding

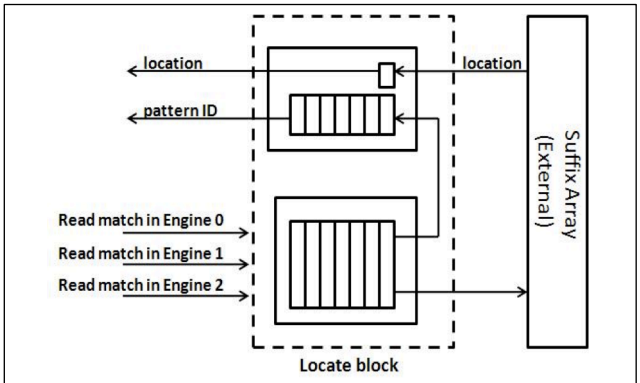

Figure 6: Locate block have send and receive queues similar to send and receive blocks for finding the location of a pattern from the suffix array residing in main memory.

iterations, the copy is passed to Engine 2 where new copies are created again.

Figure 5 shows the block diagram of the *replace* block inserted on the *update* block of Engine 1. Engine 1 accepts 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 ends on Engine 1.

C. Locate Block

Reads that exit from the matching engines are passed to a *locate* block that searches the location of the pattern on the text. The data passed to the *locate* block for each read are the pattern

ID and the last pointers returned from memory. The architecture of the *locate* block is similar to *send* and *receive* blocks. Figure 6 shows the block diagram of the locate block consisting of queues for sending addresses to external memory and waiting data returning from memory.

The *locate* block sends the top pointer as an address to the

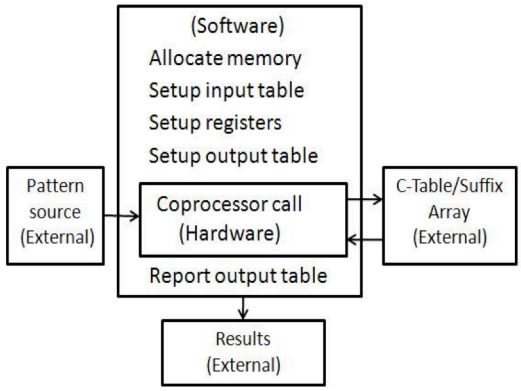

Figure 7: Role of software is mainly on memory allocation and reporting. Hardware performs the search algorithm.

suffix array placed in external memory. External memory returns the location that is written to the output file. If a read exists at multiple locations, we send multiple addresses to memory for the required locations.

V. Implementation and Evaluation

In this section we describe the implementation of FHAST on the Convey Computers HC-1.

A. FHAST Hardware/Software Implementation

Figure 7 illustrates the role of the hardware in searching for the reads on the text. The software performs memory allocation for reading the C tables, the suffix arrays, the reads and writing the results to external memory. After allocating memory and setting up the coprocessor registers, the host CPU calls the coprocessor to perform the search algorithm that writes matching patterns to memory allocated by software. Software then writes results to an output file.

We conducted our experiments on the Convey HC-1 hybrid core computing system 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. All processors, both host processor and FPGA coprocessors, have one shared cache coherent memory. Each FPGA has16 memory channels from eight memory controllers. This memory subsystem supports a peak bandwidth of 80 GB/s. We implemented designs for a pattern length of 101 characters that supports 0, 1, and 2 mismatches using only one FPGA in the coprocessor. The design is synthesized with place and route on a Xilinx Virtex 5 (XC5VLX330) FPGA that occupies 23,923 slices or 46% of the FPGA. We set the frequency to 150 MHz that is the maximum operating frequency of the memory controllers of the Convey HC1.

TABLE I. Specifications of CPU running Bowtie

	CPU 1	CPU 2
CPU type	Xeon L540B	Xeon E5520
# of cores	2 dual cores	2 quad cores
Memory size	192 GB	24 GB
Cache size	6 MB	8 MB
Frequency	2.13 GHz	2.27 GHz

B. Analysis and Comparison to Bowtie

We compare our results to the Bowtie software tool used for mapping DNA sequences. We executed the Bowtie software tool using two systems whose specifications are shown in Table 1. CPU 1 is the host CPU on the HC-1. We measured the execution time of searching 18 million unique reads with 101 characters in length on Chromosome 14 of the human genome having a length of 107 million characters.

Table 2 shows the execution time in seconds of FHAST and Bowtie running in both systems in detecting the reads having zero, one and two mismatches. The table shows longer execution time of Bowtie running in both CPUs compared to FHAST. Notice that there is a significant difference in the execution time of FHAST between searching exact and approximate matches. This is because the reverse of a read is required in the search to detect mismatches. This additional copy of a read represents an additional thread that uses up bandwidth that lengthens the execution time. Also notice that with FHAST there is no significant difference in execution time between searching for one and two mismatches.

This small difference in execution time is due to the simultaneous searching of reads in the three engines concurrently. The execution time for Bowtie increases significantly as more mismatches are allowed. Figure 7 shows the speed up of FHAST over Bowtie for the two CPUs. Observe that the highest speed up is achieved in detecting two mismatches where Bowtie execution time is the longest. *By masking memory latency, the customized multithreading approach allows us to achieve better than 70x speedup on a 150 MHz FPGA over large CPUs.*

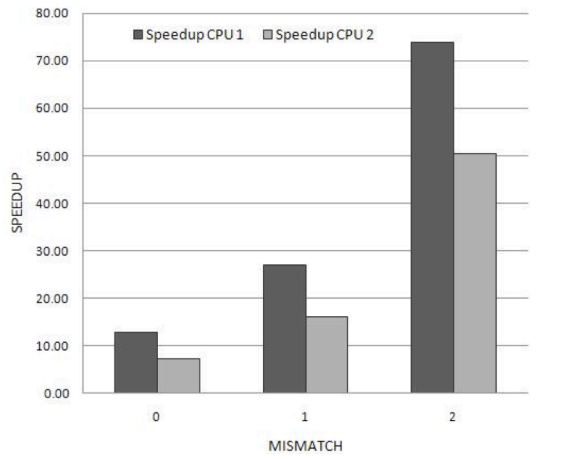

Figure 7: Speed up of FHAST and Bowtie for exact matches, one and two mismatches.

TABLE II. EXECUTION TIME (IN SECONDS) OF FHAST AND BOWTIE. FHAST IS RUNNING ON A SINGLE FPGA, BOWTIE ON A SINGLE CORE, 18 MILLION READS MATCHED IN CHROMOSOME 14.

Mismatch	FHAST	Bowtie	
		CPU1	CPU2
0	55.43	715	404
1	71.17	1924	1142
2	73.25	5410	3698

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.

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 III. Note that the execution time of FHAST, on four FPGAs, (138 seconds) is significantly lower than that of Bowtie with 16 cores (336 seconds) by a factor of 2.43.

TABLE III. MULTITHREADED EXECUTION TIME OF BOWTIE AND SPEEDUP OVER SINGLE THREAD

Threads	1	2	4	8	16
Time (s)	3325	1772	896	501	336
Speedup	1	1.88	3.71	6.64	9.90

VI. RELATED WORKS

The first use of FPGA in accelerating edit distance calculations on DNA strings appeared in [21]. Subsequent work focused, mostly, on accelerators for dynamic programming algorithms such as Smith-Waterman algorithm. In [15], the algorithm was parameterized based on the length of the pattern, allowed mismatches, and number of symbols. In [12, 13, 16, 17], systolic arrays were used to execute the algorithm. In [13], the target is optimizing the cell that is the fundamental block in faster execution. In [12], the focus is using systolic array to take advantage of parallelism inherent to the algorithm. In [16], the systolic array structure is automatically generated using a compiler. In [17], the Smith-Waterman algorithm was implemented on a supercomputing platform using FPGAs as coprocessors. The platform included a highly pipelined system that reduces FPGA resource utilization.

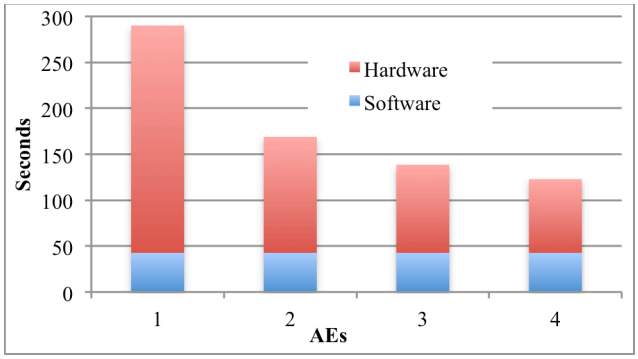

Figure 8: Breakdown of the FHAST execution in hardware and software times using 1 to 4 Accelerator Engines (FPGAs)

Algorithms based on seeds and hash tables that perform DNA sequence matching have also been explored. Seeds are shorter sequences of reads where a hash table is built based on the location of the seed on the genome. The index of the hash tables serves as reference if reads appear on the genome. The entire read is then verified on the location listed on the hash table. One tool that uses seeds is BLAST [18] and it has also been implemented on FPGAs. In [20], the seed generation phase of BLAST was accelerated. In [19], BLAST is implemented on an FPGA with an optimized verification phase. A combination of seeds and the Smith-Waterman algorithm has also been implemented in [11]. This implementation first maps the seeds using the hash table previously discussed and the then uses the Smith-Waterman algorithm for the alignment.

Besides using seeds and dynamic programming, finite automata have also been used for exact sequence matching. The implementation of the Aho-Corasick algorithm in FPGAs has been explored in [14] where protein sequences are matched on a reference genome. The brute force approach has also been implemented in [22]. In this approach, the *reads* are placed in registers and the genome is streamed while the searching is performed by direct comparison of characters.

VII. CONCLUSION

In this paper we have described and demonstrated an FPGA-based customized multithreading approach to hide long memory latencies using the FHAST tool for matching DNA short reads. We compared the FHAST's execution time and output results to Bowtie, which is a widely used tool for sequencing reads. Experimental results show that FHAST achieves a speedup of up to 70x over Bowtie. 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 when allowing for more mismatches. FHAST could handle even a higher number mismatches by adding more engines without any significant increase in execution time. The parallel execution of FHAST is limited by the software pre- and post-processing necessary to prepare the data for hardware processing and display the results in the same format as Bowtie. However, in spite of this overhead, FHAST executing on four FPGAs is 2.43 times than Bowtie on 16 cores.

ACKNOWLEDGMENT

In this work W. Najjar and E. Fernandez are supported in part by NSF Awards 0905509, 0811416 and by a Cisco Systems research grant. S. Lonardi is supported by NIH Award AI85077-01A1 and NSF Award DBI 1062301. J. Villarreal is supported at Jacquard Computing Inc. by AFRL contract FA9453-09-C-0173.

REFERENCES

[1] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. "Exploiting heterogeneous parallelism on a multithreaded multiprocessor." In Proc. of the 6th Int. Conf. on Supercomputing, ICS '92, pages 188–197, New York, NY, USA, 1992. ACM.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. "The Tera Computer System." In Proc. of the 4th Int. Conf. on Supercomputing, ICS '90,pages 1–6, New York, NY, USA, 1990. ACM.

[3] J. Feo, D. Harper, S. Kahan, and P. Konecny. "ELDORADO." In Proc. of the 2nd Conference on Computing Frontiers, CF '05, pages 28-34, New York, NY, USA, 2005. ACM.

[4] P. Ferrragina and G. Manzini, "Opportunistic Data Structures with Application." In Proc. 41st IEEE Symposium on Foundations of Computer Science, pp. 390-398, 2000.

[5] P. Ferrragina and G. Manzini, "An Experimental Study of an Opportunistic Index." In Proc. 12th ACM-SIAM Symposium on Discrete Algorithms, pp. 269-278, 2001.

[6] M. Burrows and D.J. Wheeler, *A Block-sorting Lossless Data Compression Algorithm,* SRC Research Report, May 1994.

[7] E. Fernandez, W. Najjar, and S. Lonardi, "String Matching in Hardware using the FM-Index." In Proc. IEEE Int. Symp. on Field-Programmable Custom Computing Machines, FCCM 2011, pages 218-225, Salt Lake City, UT, USA, 2011.

[8] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and Memory-Efficient Alignment of Short DNA sequences to the Human Genome." Genome Biology, 2009.

[9] www.conveycomputer.com.

[10] H. Li and R. Durbin, "Fast and Accurate Short Read Alignment with Burrows-Wheeler Transforms." Bioinformatics, 2009.

[11] C. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, W. Ruzzo, "Hardware Acceleration of Short Read Mapping." in Proc. IEEE Int. Symp. on Field-Programmable Custom Computing Machines, FCCM 2012.

[12] G. Caffarena, S. Bojanic, J. Lopez, C. Pedreira, and O. Nieto-Taladriz. "High-Speed Systolic Array for Gene Matching." Int. Symp.on Field Programmable Gate Arrays (FPGA), 2004.

[13] M. Gok and C. Yilmaz, *Efficient Cell Designs for Systolic Smith-Waterman Implementations*". Int. Conf. on Field Programmable Logic and Applications (FPL), 2006.

[14] Y. Dandass, S. Burgess, M. Lawrence, and Susan Bridges. *Accelerating String Set Matching in FPGA Hardware for Bioinformatic Research*. BMC Bioinformatics, April, Vol. 9, No. 197, pp1471-2105, 2008.

[15] K. Benkrid, Y. Liu, and A. Benkrid. *A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment*. IEEE Trans. on Very Large Scale Integration Systems, Vol. 17, No. 4, pp. 561-570. April 2009.

[16] B. Buyukkurt and W. Najjar. *Compiler Generated Systolic Arrays for Wavefront Algorithm Acceleration on FPGAs.* In Proc. of 18th Int. Conf. on Field Programmable Logic and Applications (FPL), 2008.

[17] P. Zhang, G. Tan, and G. Gao. *Implementation of the Smith-Waterman algorithm on a Reconfigurable Supercomputing Platform.* Conference on High Performance Networking and Computing, 2007.

[18] BLAST. http://blast.ncbi.nlm.nih.gov/Blast.cgi

[19] M. Herbordt, J. Model, Y. Gu, B. Sukhwani, and T. Van Court. *Single Pass, BLAST-Like, Approximate String Matching on FPGAs.* In Proc. of IEEE Symp. on Field Programmable Custom Computing Machines (FCCM), 2006.

[20] A. Jacob, J. Lancaster, J. Buhler, and R. Chamberlain. *FPGA accelerated seed generation in Mercury BLASTP*. In Proc. of IEEE Symp. on Field Programmable Custom Computing Machines (FCCM), 2007.

[21] D.T. Hoang; *Searching genetic databases on Splash 2*,Proc. IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), 1993.

[22] E. Fernandez, W. Najjar, E. Harris and S. Lonardi, *Exploration of Short Reads Genome Mapping in Hardware,* In Proc. of 20th Int. Conf. on Field Programmable Logic and Application, 2010.