Parallel Accelerators For GlimmerHMM Bioinformatics Algorithm

Nafsika Chrysanthou, Grigorios Chrysos, Euripides Sotiriades, Ioannis Papaefstathiou
Microprocessor and Hardware Laboratory
Department of Electronic and Computer Engineering
Technical University of Crete
Chania, Greece
nasia.c@gmail.com, {chrysos, esot, ygp}@mhl.tuc.gr

Abstract— **In the last decades there is an exponential growth in the amount of genomic data that need to be analyzed. A very important problem in biology is the extraction of the biologically functional genomic DNA from the actual genome of the organisms. There have been proposed many computational biology algorithms that solve the gene finding problem which utilize various approaches; GlimmerHMM is considered one of the most efficient such algorithms. This paper presents two different accelerators for the GlimmerHMM algorithm. One of them is implemented on a modern FPGA platform exploiting the parallelism that reconfigurable logic offers and the other one utilizes a GPU (Graphic Processing Unit) taking advantage of a highly multithreaded operational environment. The performance of the implemented systems is compared against the one achieved when the official distribution of the algorithm is executed on a high-end multi-core server; the speedup initiated, for the most compute intensive part, is up to 200x for the FPGA-based system and up to 34x for the GPU-based system.**

Keywords- Gene finding, FPGA, GPU, bioinformatics

I. INTRODUCTION

Gene identification is one of the most important steps in the process of understanding the genome of the different species. This process refers to identifying biologically functional stretches of sequences (genes) in genomic DNA. Gene identification becomes very important nowadays due to the large number of surveys on the genomes of different organisms performed by biologists.

Many approaches for the gene finding problem have been proposed so far; certain methods do not have high accuracy whereas others need huge processing time. In general, there are three main approaches to the gene identification problem: homology-based approaches [1], Ab Initio approaches [2], like Glimmer [3] and GlimmerHMM [4], and comparative genomics methods [5]. The difference between these algorithms is on the methodology used for solving the gene finding problem. Each one of the above algorithms has several advantages and disadvantages; and they are used according to the accuracy needed and the kind of the organism that the experiments take place on (i.e. prokaryotes or eukaryotes).

Hidden Markov Models (HMM) are widely used for gene finding. The GlimmerHMM [4] algorithm is based on the "Generalized HMM" method for gene identification and since it is considered one of the most efficient algorithms for the gene finding problem, it is widely used in several Biological Research Institutes (over 100 as reported in [4]). Also, it is considered as an official benchmark in computational biology and it is included in the Bioperf Benchmark [6]. GlimmerHMM is used for gene finding in eukaryotes' genomes and it has very high accuracy (i.e. 97% - 98%). The high execution time even on a high-end computer, especially in case of huge genomes, is one of the main drawbacks of this method; this drawback led us to the implementation of two accelerators for speeding up this highly efficient algorithm.

FPGAs offer high flexibility and high performance for processing tasks that use fast on-chip memories, reconfigurable resources [7] as well as built-in DSPs. On the other hand, GPUs support the efficient execution of multiple threads on multiple data simultaneously offering high levels of parallelism.

This paper presents two different accelerators for the GlimmerHMM algorithm. First, we have analyzed the algorithm and identified its most computationally intensive parts. Then we have implemented those highly CPU intensive functions to both a modern FPGA and a high-end GPU; to the best of our knowledge, this is the first FPGA-based as well as the first GPU-based systems proposed for the acceleration of the GlimmerHMM algorithm. Finally, the performance of the implemented systems is compared against that achieved when the official GlimmerHMM software suite is executed on a high-end multi-core server. As the real-world performance results demonstrate, the FPGA-based system is 200 times faster than a high-end server, when executing the most time consuming function of the algorithm, while the GPU achieves a speedup of 34x when compared with this same server; this accelerated tree-traversal function is also utilized in numerous other applications, like data mining, ray tracing and IP lookup.

The rest of this paper is organized as follows: Section 2 presents the related work while Section 3 describes and analyzes the GlimmerHMM algorithm. Sections 4 and 5 present the implementations on an FPGA and on a GPU, respectively. Section 6 presents our real-world performance results while Section 7 concludes our work.

II. RELATED WORK

Gene finding and gene-related HMMs are both very hot topics in bioinformatics today. The HMMER algorithm is

978-3-9810801-7-9/DATE11/©2011 EDAA

considered as the most widely used tool for the genetic sequence search on databases. Walters et al. [8] implemented the HMMER algorithm on a hybrid system consisting of reconfigurable devices and an MPI cluster and they achieved a speedup of up to 30x in a system with a large number of CPUs and 2 FPGA devices. Oliver et al. in [9, 10] mapped the Plan7 Viterbi algorithm, which is the basic stage of the HMMER algorithm, on a Spartan 3 FPGA achieving acceleration of up to 30x against the execution of this same stage on a high-end AMD CPU. Takagi et al. in [11] propose a fully parallel implementation of the Viterbi algorithm which, when it is executed on a high end FPGA, it achieves an up to 360x speedup for certain sequences vs. the execution of the official software on a Core2Duo CPU. Chrysos et al. in [12] describe the first FPGA-based implementation of the Glimmer algorithm, which is a gene finding algorithm tailored to prokaryote organisms and based on the Interpolated Markov model; this latter model is also used in the GlimmerHMM algorithm. This architecture maps the scoring phase of the algorithm on an FPGA achieving a speedup of up to 33x vs. the execution on a high end CPU.

Moreover, many similar bioinformatics algorithms have been developed on GPU platforms. Horn et al. in [13] present the implementation of the Viterbi method for the HMMER algorithm on a cluster of NVIDIA GPUs achieving linear speedup with the number of nodes. Ganesan et al. in [14] re-designed the *hmmsearch* program in order to exploit its parallelism on a modern GPU platform. Their system significantly outperforms other existing methods for unsorted searches on large databases.

The key difference between all those previous systems and the ones we propose in this paper is that our implementations are the first to target the acceleration of the GlimmerHMM algorithm which is considered to probably be the most efficient such algorithm currently available; as a result no direct comparison between our work and these previous implementations is possible.

III. GLIMMERHMM ALGORITHM

A. Description of GlimmerHMM algorithm

The GlimmerHMM (Gene Locator and Interpolated Markov Model ER Hidden Markov Model) algorithm is used mainly for eukaryotic organisms and it is based on the Generalized Hidden Markov Model (GHMM) [4]. Hidden Markov Models (HMMs) provide a precise probabilistic method for modeling gene sequences and therefore they are very frequently used to find new genes in DNA sequences.

The GlimmerHMM algorithm is divided into three phases: **training phase, identification phase** and **resolving overlap phase**. The algorithm uses long sequences, which are similar to already known genes from other organisms, as training data.

The training phase of the algorithm takes as input the genes of one or more organisms and it constructs four interpolated Markov Models (IMM). The IMM is a generalization of the fixed-order Markov chains. The chains based on IMM are represented with a fourth order tree structure. This tree consists of eight node-levels while each node contains four probabilities. Three of the training IMMs describe the three reading frame positions of the coding region of the input genome, whereas the fourth IMM describes the non-coding regions of the input training dataset.

The second phase of the algorithm takes as input a number of DNA sequences and using a sliding window, which is 12-base wide; it records the scores of all the 12-bases contexts. All those scores for each sequence are summed up concluding to a final score for each sequence. The four IMMs, which were constructed in the previous phase, are used for this scoring process. The score for each input DNA sequence represents the probability that this certain sequence describes a specific gene of the organism. Finally in the third phase, the method resolves the candidate gene sequences that overlap.

B. GlimmerHMM algorithm SW analysis

Several versions of the GlimmerHMM algorithm exist, each incorporating certain improvements to the initial version. In this work we used the latest distributed version of the algorithm (GlimmerHMM v3.1 [15]), as distributed officially by the Center for Bioinformatics and Computational Biology (CBCB) of the University of Maryland.

The official training datasets of the algorithm include five different genomes. All of these five official distributed sets of four IMMs each were used in our experiments. The provided software suite implements the three phases that were described above in three separate programs: *build-icm*, which builds the Interpolated Markov Models, *glimmer2* which records the score of the candidate gene and *extract*, which extracts all non-overlapping "open-reading" frames.

The GlimmerHMM software is unix-based, and the most recent version of the algorithm is designed for single core system. Intel's V-Tune Performance Analyzer for Linux was used for the assessment of the algorithm's performance. The *glimmer2* program implements the scoring phase and is the most computationally demanding part of the algorithm: it accounts for more than 90% of the total execution time, including data transfers. The function *get_prob_of_window*, which computes the probability score of a 12-base context that resulted from the sliding of the window of the candidate gene, takes from 21% and up to 55% of the total execution time of the algorithm's second phase (or in other words from 20% to over 50% of the total execution time), as shown in Table 1. This led us to the conclusion that a parallel implementation of the *get_prob_of_window* function will significantly accelerate the execution of the GlimmerHMM scheme.

TABLE I. EXECUTION TIMES FOR GLIMMER2 PROGRAM AND *GET_PROB_OF_WINDOW* FUNCTION FOR ALL THE TRAINING DATASETS AND FOR A SMALL 2209344 BP INPUT (LEISHMANIA BRAZILIENSIS)

Training Dataset	Glimmer2 program time (sec)	Get_prob_of_window function time (sec)	% function time Compared to Glimmer2 execution time
Arabidopsis	7.77	4.25	54.70%
Zebrafish	7.99	4.21	52.69%
Rice	10.78	4.17	38.68%
Human	13.83	3.91	28.27%
Celegans	20.08	4.13	20.57%

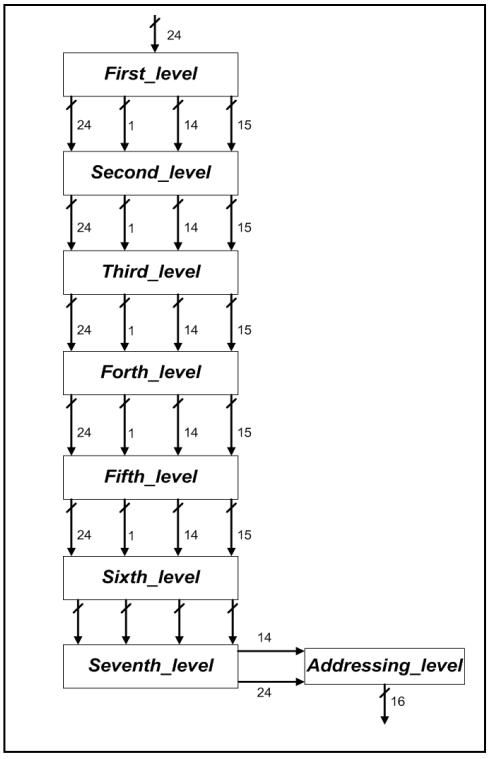

Figure 1. Block diagram of the tree-structure IMM architecture.

C. *Get_prob_of_window function analysis*

As described above, the function *get_prob_of_window* is the most time consuming function of the algorithm. This function is executed for all the 12-base contexts of the forward and the reverse input DNA strands. The *get_prob_of_window* function takes as input the 12-base context, one of the three corresponding IMMs for each context (depending on the reading frame) and it traverses the tree structure, computing at the final step a scoring probability for this context. After that, the process is repeated for the non-coding IMM and another scoring probability is calculated. Finally, the function takes the logarithm of these two values and subtracts them concluding to the final value, which represents the scoring probability of the input context. As it is clear, most of the execution time of the function is consumed on the traversing of the tree-structure IMMs and the calculation of the logarithms.

IV. FPGA-BASED SYSTEM ARCHITECTURE

This section describes the implementation of the GlimmerHMM algorithm on a modern FPGA. The system was implemented using Xilinx ISE Design Suite 10.1 and Xilinx EDK 10.1. Initially, a parallel architecture that implements the *get_prob_of_window* function of the algorithm is presented. Then we present the final system implementation containing both the CPU server and the FPGA sub-system.

A. *FPGA-based function architecture*

This section describes the mapping of the software function *get_prob_of_window* on reconfigurable logic. As mentioned above, the function *get_prob_of_window* takes as input a 12-base context and an IMM, which will be traversed, and its output is a probability score. For the hardware implementation we performed a certain matching of the genetic bases to bit-values (i.e., A = “00”, C = “01”, G = “10”, T = “11”).

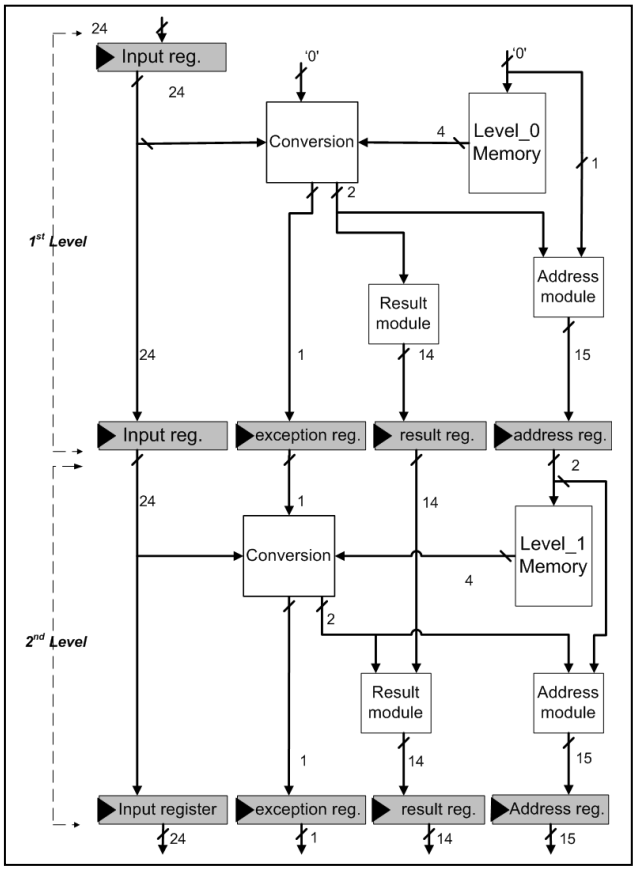

Figure 2. The first two levels of our architecture.

As mentioned above, each IMM is an eight-level fourth-order tree in which each node consists of four single precision floating point probabilities. After the tree traversal completes, the algorithm calculates the logarithm of the probability value based on the traversal path. The implementation of the IMM on a reconfigurable device is presented in Figure 1. The implemented IMM consists of eight pipeline stages, each of which represents the set of all nodes of the tree data structure at a given depth (tree level). This results to a fully pipelined system with a throughput of 1 tree-traversal per cycle.

The first level of the IMM takes as an input a 24-bit value, which actually is the 12-base context of the input sequence, and this value is passed to the next level of the IMM in each cycle. Figure 2 shows the first two levels of the IMM tree. These two levels are repeated so as to create the first seven levels of the IMM implementation. Each level consists of a memory module, named as level_x, where the traversing information is kept. The address module is utilized in the tree traversing as it computes, in each cycle, the address of the next level memory. The result module constructs the address that will be used as an index to the eighth level memory. This memory will produce the final logarithm of probability for this specific 12-base input context.

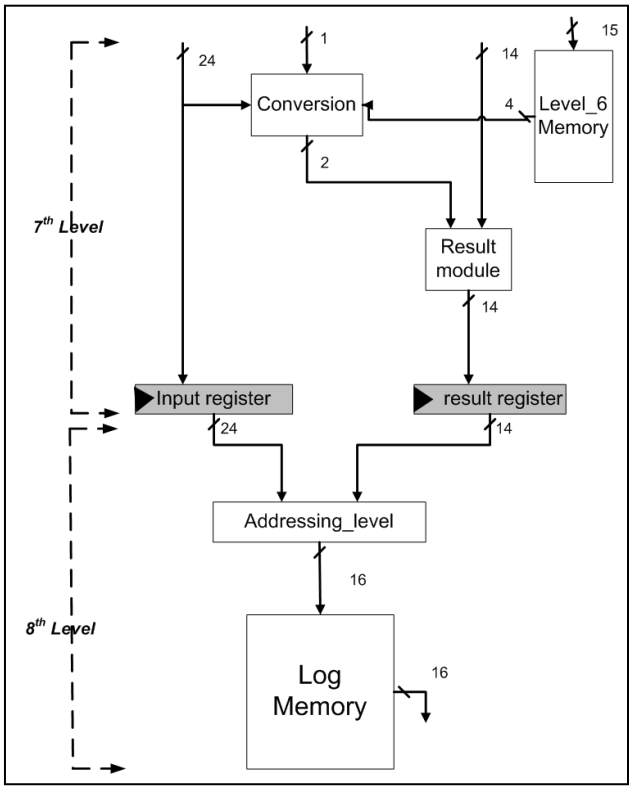

Figure 3. The last two levels of the IMM architecture.

The final value that is returned by the *get_prob_of_window* function is the logarithm of the value that comes from the IMM traversal. Figure 3 presents the last two levels of the implemented IMM. Actually, the last level consists of the "Log memory", a lookup table that stores the logarithm of all the probabilities produced in the training phase. As the design was restricted by the FPGA resources, namely the number of Block Memories, we reduced the width of our arithmetic to 16-bit floating point; as our exhaustive experimental results demonstrate this reduction in the internal accuracy resulted in absolutely no difference in the final results of the scheme.

As mentioned in the previous section, each training dataset of the algorithm consists of four different IMMs, three for the coding regions and one for the non-coding one, which are traversed for each 12-base input context. This led to the implementation of four parallel tree structures, each one of them holding the information of a specific IMM. The final value reported by the function comes from the subtraction of the logarithm of the non-coding IMM from that of the coding IMM; in order to implement this subtraction certain 16-bit floating point subtractors are utilized.

As the implementation is fully-pipelined, a shift register has also been used in order to keep the 12-base input context. Finally, taking advantage of the two ports of the built-in Block memories, the implemented system takes as input two 12-base contexts per cycle and it produces simultaneously 6 values. The architecture of the implemented module is shown in Figure 4.

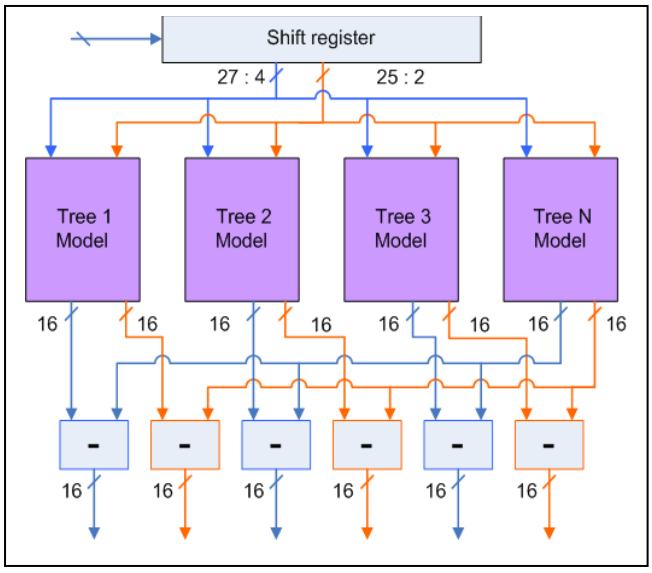

Figure 4. The hardware architecture of get_prob_of_window function

B. *GlimmerHMM FPGA-based system architecture*

This section describes the integration of the developed hardware module with a high-end CPU server. The *get_prob_of_window* core is being implemented on a relatively small Virtex-5 FPGA. The FPGA is connected through PCI-Express with a high-end server, where the rest of the GlimmerHMM algorithm is executed. The server formats the data appropriately, sends them to the FPGA board, waits for the results, and once it has received then continues with execution of the rest of the algorithm.

A communication interface was built for the data exchange between the implementation platform and the server. The PCI-Express interface consists of two FIFOs where the data that go to and from the FPGA are placed; and FSM controls the full process. The block diagram of the complete system is shown in Figure 5. Despite the exploitation of high levels of parallelism in the implemented architecture, the communication interface was the bottleneck of the full system and it reduced significantly the overall performance.

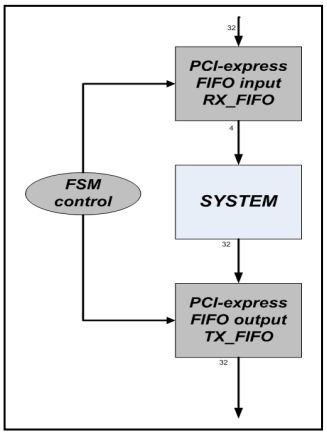

Figure 5. Block diagram of the system

TABLE II. FPGA RESOURCES FOR THE FUNCTION AND THE FULL FPGA-BASED SYSTEM IMPLEMENTATION

FPGA Resources	Function FPGA utilization	System FPGA utilization
Slices	4617 / 69120 (6%)	21,936 / 69,120 (31%)
Block RAMs	132 / 148 (89%)	147 / 148 (99%)

V. GPU SYSTEM ARCHITECTURE

The GlimmerHMM algorithm was mapped on an NVIDIA GPU using the Compute Unified Device Architecture (CUDA), an extension of C/C++ that enables users to write scalable multi-threaded programs for GPUs. CUDA programs can be executed on GPUs with NVIDIA's Tesla unified computing architecture. CUDA programs contain a sequential part, called a kernel, which is the multithreading part of the implementation. The number of threads executing each kernel is algorithm-depended and it is user-defined.

A. CUDA-GlimmerHMM architecture

The CUDA-GlimmerHMM architecture maps the function *get_prob_of_window* as a CUDA kernel and the rest of the program is executed on a high-end server. Considering the calculation of the three probabilities for each 12-base context as a task, many approaches were implemented for parallelizing the probability scoring phase of the input sequence. Actually, the GlimmerHMM algorithm for very long sequences breaks the input DNA sequence in 1,000,000-bases long, overlapped segments and it processes them individually and sequentially.

First, the inter-task parallelization was applied in the kernel implementation. Namely, each probability scoring task for each one of the 1,000,000 genetic bases was assigned to exactly one thread. Second, intra-task parallelization was applied by increasing the parallel threads to 3,000,000 and assigning to each one of them the calculation of one of the three reading frames probabilities associated with each 12-base context. Third, as the algorithm calculates the probability scores for the forward and the reverse strand, we doubled the parallelization factor to up to 6,000,000 threads supporting the coinstantaneous probability calculations of the two strands. Also, we used coalesced global memory accesses so as to reduce the overall memory access overhead. The data of the four IMMs were stored in the texture memory of the GPU offering cached data accesses, an approach that further accelerated the overall execution time. Finally, another GPU tailored optimization was the downscaling of the data accuracy to single precision floating point; this accuracy reduction as described in the next section does not affect the final results of the algorithm at all.

VI. SYSTEMS PERFORMANCE

This section describes the performance of the two hardware based implemented systems.

A. Systems' resources

The FPGA-based GlimmerHMM system was implemented on a Virtex-5 FPGA (XC5VLX110T). The resources used for the implementation of the get_prob_of_window function and the full system are presented in Table 2; this module is clocked at 126 MHz, whereas the clock rate for the total system was 62.5 MHz, which is the operation frequency of the PCI-Express. It is clear that the bottleneck of our innovative system is the intercommunication of the FPGA with the CPU. The CUDA-GlimmerHMM system was implemented on a 1.24 GHz NVIDIA GeForce GTX 280 with 1 GB global memory and 30 complex processors (consisting of 240 ALUs).

B. Systems' performance

We performed several tests on both implemented systems, using all the officially distributed test cases by the National Center of Biotechnology Information (NCBI) [16], for validation. The results from both systems compared identical to those of the official software (i.e. GlimmerHMM v3.1). For a baseline for comparisons we used the GlimmerHMM algorithm which was compiled with the highest optimization flags and executed on a 4-core Intel Xeon E5430 server with 12 GB RAM and Ubuntu 8.1 OS.

Tables 3 and 4 detail the performance of the accelerated function and of the entire implemented system respectively. In Table 3 we can see that the FPGA-based implementation of *get_prob_of_window* function significantly outperforms both the standard software as well as the GPU implementation, achieving a speedup of two orders of magnitude; this speedup is also relatively insensitive to the input size (except for the really small input Gallus Gallus). The GPU results show that the efficient mapping of the function as a kernel can offer a speedup of up to 34x when compared with the execution of the same function in a high-end 4-core system; however, small input sizes drastically reduce the thread-parallelization for the GPU system; as a result data transfers become the performance bottleneck and performance can actually deteriorate.

TABLE III. PERFORMANCE OF THE HARDWARE-BASED FUNCTION IMPLEMENTATION

Training Dataset	Input DNA sequence	SW Execution Time (sec)	FPGA-based Function Implementation		GPU-Based Function Implementation	
			Execution Time (sec)	Speedup vs. SW	Execution Time (sec)	Speedup vs. SW
Arabidopsis	Gallus Gallus (1028 bp)	0.02	8.84 * 10^-6	2262 x	0.80	0.03 x
	Leishmania Braziliens (2209344 bp)	4.25	0.019	223.68 x	0.97	4.38 x
	Homo sapiens (100878317 bp)	183.09	0.87	210.44 x	5.95	30.77 x
	Monodelphis (214340800 bp)	387.42	1.84	210.55 x	11.98	32.34 x
Celegans	Gallus Gallus (1028 bp)	0.01	8.84 * 10^-6	1131 x	0.80	0.01 x
	Leishmania Braziliens (2209344 bp)	4.13	0.02	206.50 x	0.91	4.54 x
	Homo sapiens (100878317 bp)	185.50	0.87	213.21 x	6.02	30.81 x
	Monodelphis (214340800 bp)	397.24	1.84	215.89 x	11.68	34.01 x

TABLE IV. PERFORMANCE OF THE HARDWARE-BASED GLIMMERHMM SYSTEM IMPLEMENTATION

Training Dataset	Input DNA sequence	SW Execution Time (sec)	FPGA-based System Implementation			GPU-Based System Implementation	
			Execution Time (sec)	Speedup vs. SW (measured)	Speedup vs. SW (calculated)	Execution Time (sec)	Speedup vs. SW
Arabidopsis	Gallus Gallus (1028 bp)	0.03	0.03	1.00 x	1.00 x	0.84	0.04 x
	Leishmania Braziliens (2209344 bp)	7.77	5.98	1.30 x	1.86 x	4.22	1.84 x
	Homo sapiens (100878317 bp)	403.50	309.01	1.31 x	1.89 x	219.05	1.84 x
	Monodelphis (214340800 bp)	866.47	732.88	1.18 x	1.72 x	475.20	1.82 x
Celegans	Gallus Gallus (1028 bp)	0.25	0.24	1.04 x	1.07 x	1.09	0.23 x
	Leishmania Braziliens (2209344 bp)	20.08	15.92	1.26 x	1.61 x	15.60	1.29 x
	Homo sapiens (100878317 bp)	675.08	578.66	1.17 x	1.52 x	473.45	1.43 x
	Monodelphis (214340800 bp)	1404.13	1015.97	1.38 x	1.91 x	967.22	1.45 x

As shown in Table 4, the GPU system implementation outperforms the other two system implementations achieving a speedup of up to 1.5x for the worst case and 1.9x for the best case. This acceleration is due to the very fast data movement from and to the GPU. The FPGA implementation achieves a 1.3x speedup over the same high-end server. The main reason for this limited speedup comes from the fact that the driver feeding the PCI-Express interface of the FPGA board does not saturate the PCI-Express channel. We calculate that using an efficient PCI-Express driver that uses interrupts (as the GPU card) the FPGA speedup would be up to 1.9x for certain cases as demonstrated in Table 4.

Another and probably more important, contribution of this work is the extremely fast FPGA-based implementation of the *get_prob_of_window* function itself; similar tree-traversal functions (such as KD-tree traversal or Trie-based IP lookup) are used in numerous applications nowadays and this is the first time that this function has been implemented on an FPGA.

VII. CONCLUSIONS

This paper presents and compares two accelerating approaches for a very important bioinformatics application named GlimmerHMM. This application is utilized in the modern Gene Finding Schemes and no accelerator for it has been presented so far.

The first accelerator is implemented on a modern relatively low-cost FPGA and it achieves a more than two orders of magnitude speedup over a high-end multi-core server, when they both execute the most CPU intensive part of the algorithm. The second accelerator utilizes a high-end GPU and triggers a 34x speedup over the same high-end multi-core server when executing this exact same function. As a result the FPGA system is significantly more efficient when executing this certain part while it is less expensive than both systems. When the IO system is also taken into account, the FPGA-based accelerator is outperformed by the GPU because our FPGA board IO uses a low performance PCI-Express link and driver on the server side reducing the board IO rate.

Finally, it is important to note that despite the communication intensive nature of the algorithm, both the GPU and the FPGA systems achieve a speedup which is near to the maximum theoretical gain for the entire algorithm.

We also note that the most time consuming function is the traversal of an n-order tree. Similar functions are utilized in numerous other applications such as data mining, ray tracing, IP lookup, etc, and all those applications can take advantage of the impressive speedup triggered by our proposed architecture on a relatively low-cost FPGA.

REFERENCES

[1] MA Andrade, CP Ponting, TJ Gibson, P Bork, "Homology – based method for identification of protein repeats using statistical significance estimates", Journal of Molecular Biology, vol. 298, pp.521-537, 2000.

[2] IM Meyer, R Durbin, "Comparative ab initio prediction of gene structures using pair HMMs", Bioinformatics, vol. 18, 2002.

[3] A. L. Delcher, D. Harmon, S. Kasif, O. White, and S. L. Salzberg, "Improved microbial gene identification with GLIMMER", Nucleic Acids Research, vol. 27, pp. 4636- 4641, 1999.

[4] W.H. Majoros, M. Pertea and S.L. Salzberg, "TigrScan and GlimmerHMM: two open-source ab initio eukaryotic gene-finders", Bioinformatics, pp. 2878-2879, 2004.

[5] M. Kellis, N. Patterson, B. Birren, B. Berger, E. S. Lander, "Methods in comparative genomics: genome correspondence, gene identification and motif discovery", Journal of Computational Biology, vol.11, 2004

[6] D. Bader, Y. Li, T. Li, V. Sachdeva, "BioPerf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications", Proceedings of the 2005 IEEE International Symposium on Workload Characterization, 2005.

[7] K. Papadimitriou, A. Anyfantis, A. Dollas, "An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems" , IEEE Transactions on Instrumentation and Measurement, Vol. 59, 2010.

[8] J. P. Walters, X. Meng, V. Chaudhary, T. F. Oliver, L. Yuan Yeow, B. Schmidt, D. Nathan, J. I. Landman, "MPI-HMMER-Boost: Distributed FPGA Acceleration", VLSI Signal Processing, vol. 48, 2007

[9] T. F. Oliver, B. Schmidt, Y. Jakop, D. L. Maskell, "High Speed Biological Sequence Analysis With Hidden Markov Models on Reconfigurable Platforms", IEEE Transactions on Information Technology in Biomedicine, vol. 13, pp 740-746, 2009.

[10] T.F. Oliver, L.Y. Yeow, B. Schmidt, "Integrating FPGA acceleration into HMMer". Parallel Computing, vol 34, pp 681-691, 2008.

[11] T. Takagi, T. Maruyama, "Accelerating HMMER search using FPGA". International Conference on Field Programmable Logic (FPL), 2009.

[12] G. Chrysos, E. Sotiriades, I. Papaefstathiou and A. Dollas, "A FPGA based Coprocessor for Gene finding using Interpolated Markov Model (IMM)". Poster at International Conference on Field Programmable Logic and Application (FPL), 2009.

[13] D.R. Horn, M. Houston, P. Hanrahan, "ClawHMMER: A Streaming HMMer-Search Implementatio". Proceedings of the 2005 ACM/IEEE conference on Supercomputing, pp. 11–20, 2005.

[14] N. Ganesan, R.D. Chamberlain, J. Buhler, M. Taufer, "Accelerating HMMER on GPUs by Implementing Hybrid Data and Task Parallelism". ACM International Conference on Bioinformatics, 2010.

[15] http://www.cbcb.umd.edu/software/GlimmerHMM/

[16] http://www.ncbi.nlm.nih.gov/sites/genome