2013 International Conference on Parallel and Distributed Systems

Evaluating Optimization Strategies for HMMer Acceleration on GPU

Samuel Ferraz Nahri Moreano
School of Computing
Federal University of Mato Grosso do Sul, Brazil
Email: {samuel, nahri}@facom.ufms.br

***Abstract*—Comparing a biological sequence to a family of sequences is an important task in Bioinformatics, commonly performed using tools such as HMMer [1], [2]. The Viterbi algorithm is applied as HMMer main step to compute the similarity between the sequence and the family. Due to the exponential growth of biological sequence databases, implementations of the Viterbi algorithm on several high performance platforms have been proposed. Nevertheless, few implementations of the Viterbi algorithm use GPUs as main platform. In this paper, we present the development and optimization of an accelerator for the Viterbi algorithm applied to biological sequence analysis on GPUs. Some of the optimizations analyzed are applied to the sequence comparison problem for the first time in the literature and others are evaluated in more depth than in related works. Our main contributions are: (a) an accelerator that achieves speedups up to 102.90 and 60.46, with respect to HMMer2 and HMMer3 execution on a general purpose computer, respectively; (b) the use of the multi-platform OpenCL programming model for the accelerator; (c) a detailed evaluation of several optimizations such as memory, control flow, execution space, instruction scheduling, and loop optimizations; and (d) a methodology of optimizations and evaluation that can also be applied to other sequence comparison algorithms, such as the HMMer3 MSV.**

***Keywords*-Sequence-profile alignment; Viterbi algorithm; GPU; Accelerator; Optimization; OpenCL**

I. Introduction

The exponential growth of biological sequence databases has been making the tasks of sequence comparison and classification more and more computationally demanding. As examples, the protein database UniProtKB TrEMBL [3] contains 23,165,610 sequences and its size almost doubles in every two years; the sequence family database Pfam [4] contains 13,672 families and it is currently 6× larger than its first version.

Given a family of sequences represented by a profile HMM (Hidden Markov Model) [5], a newly identified sequence is compared to the profile HMM using a sequence-profile comparison tool and a similarity score is produced. The sequence is classified as part of the family or not, based on a significance threshold applied to the score.

HMMer [1], [2] is a widely used sequence-profile comparison tool and is based on the Viterbi algorithm [6]. However, depending on the sizes of the sequence and family databases, HMMer can take up to 500 days to perform the comparison process using a conventional processor [7]. Therefore, several implementations of the Viterbi algorithm on high performance platforms have been proposed.

GPUs (Graphic Processing Units) combine low cost, massively parallel processing capability and increasing programmability, allowing researchers to use them as a high performance computing platform. Nevertheless, few implementations use this platform for the sequence-profile comparison problem.

This work evaluates the use of GPUs for the sequence-profile comparison problem. We perform a detailed analysis of several memory optimizations for the data structures of the problem. We also apply and evaluate optimizations such as control flow optimization, execution space and occupancy tunning, and loop unrolling combined with instruction scheduling. Some of the optimizations are applied to the sequence comparison problem for the first time in the literature, while others are evaluated in more depth than in related works.

An optimized accelerator for sequence-profile comparison is presented, evaluated and compared to other accelerators. Our accelerator is the first described in literature to use the multi-platform OpenCL programming model [8] and achieves higher performance than other GPU accelerators.

This paper is organized as follows. Section II introduces profile HMMs, the sequence-profile comparison problem, and the Viterbi algorithm. In Section III we describe related works in sequence-profile comparison accelerators. Section IV presents a basic accelerator developed and preliminary results obtained. In Section V we describe and analyze the optimizations applied. Section VI presents an optimized accelerator and its performance evaluation. Finally, in Section VII we summarize the results and suggest future works.

II. Sequence-Profile Comparison

A sequence family is a set of sequences with similar functionalities, similar 2D/3D structure, or common evolutionary history. A multiple alignment of the sequences in a family can be used to produce a profile HMM, a theoretical model that represents statistically the similarities among the sequences [5]. This model employs a set of states to record the residue distribution at each position of the multiple alignment, and to describe the variations in the family.

The classification of a newly identified sequence as part of a family allows inferring the function and/or structure of the sequence. Therefore, the sequence is compared to the profile HMMs representing several known families, in search for similarities. The sequence-profile comparison process aligns the sequence of interest to the profile HMM, and produces a similarity score. If the score is significant, the sequence is

1521-9097/13 $31.00 © 2013 IEEE
DOI 10.1109/ICPADS.2013.21

classified as part of the family and the alignment is used to insert the sequence into the HMM.

Figure 1 shows a profile HMM H following the Plan7 architecture [5] and representing a sequence family. When a sequence S is compared to H, the residues of S are aligned to the states of H. Each Match state M_j emits a residue of S, matching it to a column of the multiple alignment of the family. An Insert state I_j allows residues of S to be emitted between consecutive Match states. Each Delete state D_j allows S to jump Match states without emitting residues. Special states N, C, B, E, and J allow different alignment algorithms to be applied, such as global alignment, local alignment with respect to S, local alignment with respect to H, and multi-hit alignment. There are residue emission probabilities associated to Match and Insert states, as well as transition probabilities associated to state transitions. A group of $\{I_j,\ M_j,\ D_j\}$ states is called the node j of the profile HMM.

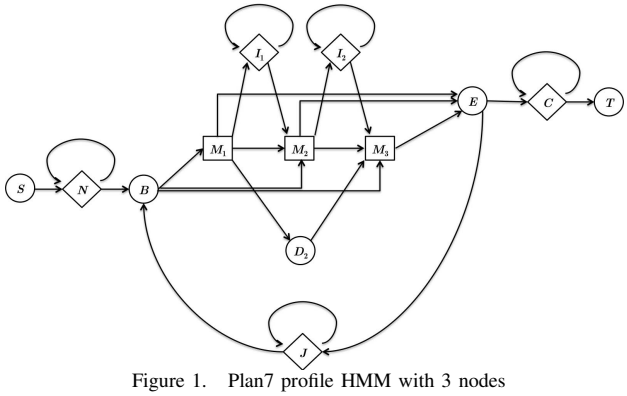

Figure 1. Plan7 profile HMM with 3 nodes

Given a sequence $S = s_1 s_2 \dots s_{|S|}$ to be aligned to a profile HMM H of Q nodes, there may be many different alignments that generate S using H, each one producing a different similarity score. The Viterbi algorithm applies the dynamic programming technique to find the best alignment of S to H, i.e., the one with highest score. It calculates:

- Matrices M, I, and D, for Match, Insert, and Delete states, respectively, where the element $[i, j]$ of each matrix is the score of the best alignment that emits the first i residues of S and reaches the corresponding state in node j;
- Vectors N, B, E, C, and J, where the i-th element of each vector is the score of the best alignment that emits the first i residues of S and reaches the state associated to the vector.

The following recurrence equations describe the Plan7 Viterbi algorithm, where Pt_{t_1,t_2} is the transition probability from state t_1 to t_2 and $Pe_t(s)$ the emission probability of residue s at state t.

$$M[i][j] = Pe_{M_j}(s_i) + max \begin{cases} M[i-1,j-1] + Pt_{M_{j-1},M_j} \\ I[i-1,j-1] \ + Pt_{I_{j-1},M_j} \\ D[i-1,j-1] \ + Pt_{D_{j-1},M_j} \\ B[i-1] \qquad + Pt_{B,M_j} \end{cases}$$

$$I[i][j] = Pe_{I_j}(s_i) + max \begin{cases} M[i-1,j] + Pt_{M_j,I_j} \\ I[i-1,j] \ + Pt_{I_j,I_j} \end{cases}$$

$$D[i][j] = max \begin{cases} M[i,j-1] + Pt_{M_{j-1},D_j} \\ D[i,j-1] \ + Pt_{D_{j-1},D_j} \end{cases}$$

$$N[i] = N[i-1] + Pt_{N,N}$$

$$E[i] = max \begin{cases} M[i,1] \ + Pt_{M_1,E} \\ M[i,2] \ + Pt_{M_2,E} \\ \quad \vdots \\ M[i,Q] + Pt_{M_Q,E} \end{cases}$$

$$J[i] = max \begin{cases} J[i-1] + Pt_{J,J} \\ E[i] \qquad + Pt_{E,J} \end{cases}$$

$$B[i] = max \begin{cases} N[i] + Pt_{N,B} \\ J[i] \ + Pt_{J,B} \end{cases}$$

$$C[i] = max \begin{cases} C[i-1] + Pt_{C,C} \\ E[i] \qquad + Pt_{E,C} \end{cases}$$

The time complexity of the Viterbi algorithm for Plan7 HMMs is $O(Q \times |S|)$. Despite its polynomial execution time, due to the exponential growth of sequence and sequence family databases, the Viterbi algorithm can be very computationally demanding, in both time and memory space. One of the simplest alternatives to improve Viterbi algorithm performance is to calculate matrices elements in parallel. However, the data dependencies between them, shown in Figure 2, prevent the parallel computation of any two elements in a matrix.

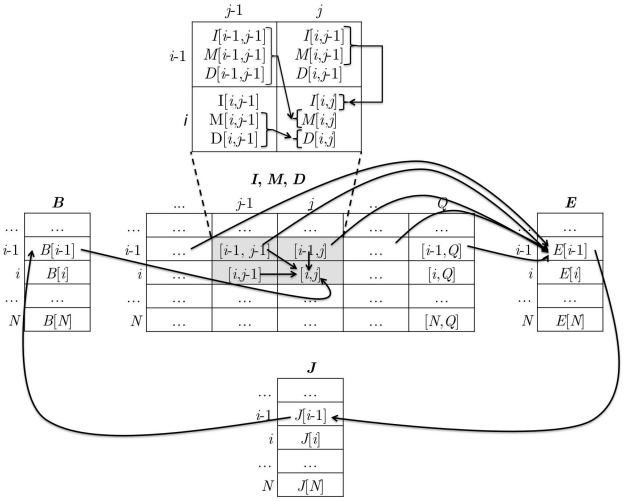

Figure 2. Data dependencies in the Plan7 Viterbi algorithm

When comparing a set of sequences to a profile HMM, there are no dependencies between the score matrices corresponding to two different sequences, which can be computed in parallel, exploiting the coarse-grained sequence parallelism.

The Viterbi algorithm calculates the score of the best alignment of S to H, however, it does not return the alignment itself, which is the sequence of states of H that generates S with highest score. A traceback algorithm, executed only when the similarity score is superior to a threshold, calculates the best alignment using the Viterbi algorithm matrices.

It is possible to execute the Viterbi algorithm storing only two rows of the score matrices and vectors, the current row and the previous one. In this case the Viterbi algorithm works as a filter, with reduced memory requirements. However, if the resulting score is significant, it is necessary to re-execute

the algorithm, this time storing the entire matrices, to be able to perform the traceback.

III. Related Work

HMMer [1], [2] is a set of programs that uses profile HMMs for biological sequence comparison and classification. One of main operations of this tool is hmmsearch, which compares a sequence database to a sequence family profile HMM. HMMer2 hmmsearch operation uses the Viterbi algorithm as its main step, while HMMer3 uses the Viterbi algorithm as the second step of its filter pipeline. The first step is the MSV (Multiple Segment Viterbi) algorithm, which is a simplified version of Viterbi algorithm with less data dependencies. HMMer is widely used and many optimizations have been proposed to improve its performance.

ClawHMMer [9] is an implementation of HMMer2 hmmsearch operation on GPU using the Brook language. It uses profile HMMs with only M, I, and D states and exploits sequence parallelism. The sequence database is sorted by sequence length and divided into batches, in order to fit in GPU memory and provide load balance, with sequences in the same batch having similar lengths. The accelerator works as a filter, storing only two rows of the score matrices, and traceback is performed on host if necessary. ClawHMMer, executing on a ATI R520, reached a speedup of 36 compared to HMMer2 executing on Intel Pentium 4 2.8GHz.

GPU-HMMer [10] implements HMMer2 hmmsearch operation as a filter on GPU, using the CUDA programming model. It uses Plan7 profile HMMs and exploits sequence parallelism. The sequence database is sorted to provide load balance and the inner loop of Viterbi algorithm is unrolled. Score matrices are stored in GPU global memory and accessed with coalescency, while transition and emission probabilities are kept in constant and texture memory, taking advantage of caching. Using a GeForce GTX 8800 Ultra, GPU-HMMer reached speedups between 12 and 38.6 compared to HMMer2 executing on AMD Athlon 2.2GHz.

CuHMMer [11] also implements HMMer2 hmmsearch operation as a filter on GPU with CUDA, using Plan7 profile HMMs and exploiting sequence parallelism. The main difference is that, instead of keeping the host idle while the GPU executes, CuHMMer executes HMMer on the host in parallel to the GPU. The sequences are grouped based on their length to provide load balance, and transition and emission probabilities are stored in GPU shared or texture memory. CuHMMer, executing on a GeForce GTX 8800, reached speedups between 13 and 45 compared to HMMer2 executing on AMD Athlon64 X2 Dual Core processor.

Du et al. [12] implements HMMer2 hmmsearch operation on GPU using CUDA and profile HMMs with only M, I, and D states, allowing only global sequence-profile alignments. Once the J state does not exist, the cyclical data dependency shown in Figure 2 is broken and it is possible to calculate M, I, and D anti-diagonal cells in parallel, one anti-diagonal at a time. They implement three different approaches: storing the whole score matrices in GPU global memory, accessing them with coalescency, and performing traceback on the GPU; storing only two anti-diagonals on GPU, copying old anti-diagonals back to the host using overlapping, and performing traceback on the host, but without re-executing Viterbi algorithm; preprocessing the sequences on the host in order to find homologous segments among them and executing Viterbi algorithm on GPU for the other segments. Using a GeForce GTX 9800, they reached speedups between 1.97 and 72.21 compared to HMMer2 executing on Intel Dual Core 2.83GHz.

Ganesan et al. [13] implements HMMer2 hmmsearch operation on GPU using CUDA and Plan7 profile HMMs. They iterate the Viterbi algorithm recurrences, allowing cells of the same row of M and D score matrices to be calculated in parallel, while successive rows are computed sequentially. This way, they exploit both data and sequence parallelism. Using a cluster of four NVIDIA Tesla C1060 GPUs, they reached a speedup of 100 compared to HMMer executing on AMD Opteron 2.33GHz.

Li et al. [14] implements a speculative version of the HMMer3 MSV filter on GPU using CUDA, applying optimizations such as memory coalescing, asynchronous transfers between CPU and GPU, load balancing by sorting the sequences, and loop unrolling. Auxiliary threads are executed on CPU to control I/O, prepare the data for the GPU and run the other steps of HMMer3 pipeline (Viterbi and Forward algorithms). Using a NVIDIA Tesla C2050 GPU, they reached a speedup of up to 6.5 compared to HMMer3 SSE serial version executing on Intel Xeon E5506.

Most FPGA (Field-Programmable Gate Array) accelerators [7], [15]–[17] for the Viterbi algorithm implement a systolic array and eliminate state J, exploiting anti-diagonal data parallelism, but decreasing similarity score accuracy. Strategies to reduce the accuracy loss are applied, by duplicating the profile HMM [7] or recalculating part of the score matrices [17]. Some accelerators exploit limited sequence parallelism [18] or combine anti-diagonal and sequence parallelism, by interleaving two sequences [16]. Abbas and Derrien [19] implement an accelerator for the HMMer3 MSV and Viterbi algorithms in FPGA, rewriting the recurrence equations to expose more parallelism. Cluster-based solutions are also used to improve HMMer performance, exploiting sequence parallelism [20], [21]. In general, FPGA accelerators achieve good performance results, at the expense of accuracy loss, while cluster solutions produce accurate similarity results, however with smaller performance gains.

IV. Basic Accelerator and Preliminary Results

The execution platform used in this work consists of a GPU GeForce GTX 460 1GB GDDR5 connected to a host computer (through a PCI-Express interface) with 4GB and an AMD Athlon II X3 processor. The accelerator programs were written using OpenCL 1.0 and were executed on the CUDA architecture. OpenCL and CUDA have many similarities, however OpenCL may suffer from a small performance loss on NVIDIA GPUs compared to programs using CUDA [22]. One of the main advantages of OpenCL is the portability,

allowing the execution of the same code in several different architectures.

We used in the experiments the profile HMMs corresponding to the 20 families with highest number of sequences (top twenty) from Pfam [4] sequence family database. These HMMs have length between 23 and 488, and 173 on average. A subset of 70,000 sequences with length between 550 and 1,500 was randomly extracted from the UniProtKB Swiss-Prot database [23] and used in preliminary experiments. The final optimized accelerator was evaluated using the entire database, composed of more than 500,000 sequences.

We developed a basic GPU accelerator for the Viterbi algorithm. Given a set of N sequences $S = \{S_1, S_2, ..., S_N\}$ to be compared to a profile HMM H of length Q, we execute in parallel N instances of the Viterbi algorithm, each one comparing a different sequence S_k to the same HMM H, exploiting sequence parallelism.

Since the Viterbi algorithm instances are independent, each instance must have its own set of score matrices and vectors. Table I shows, in the second column, the memory space required to store the entire score structures when executing the Viterbi algorithm on GPU for the complete Swiss-Prot database and a profile HMM with the average length of the top twenty families. The other structures (sequences, transition and emission probabilities) are much smaller than the scores, fitting into GPU global memory. Therefore, there should be at least 370GB available on GPU to store the score structures, an amount that is not supported by current GPUs.

In order to reduce the amount of memory required, our GPU accelerator works as a filter. Instead of storing the entire score matrices and vectors, only two rows are kept. Table I shows, in the third column, the memory space required to execute the Viterbi filter algorithm in parallel on GPU for the same typical data inputs. There should be at least 2GB available on GPU to store the score structures, an amount easily supported by several current GPUs.

Table I
MEMORY SPACE REQUIREMENTS OF COMPLETE AND FILTER VITERBI ALGORITHM ON GPU

Score structures	Viterbi algorithm with complete score structures	Viterbi filter algorithm with reduced score structures
M, I, D	$3\times Q\times 4\times \sum_{k=1}^{N} \lvert S_k\rvert$ bytes	$3\times Q\times 4\times 2\times N$ bytes
N, B, E, C, J	$5\times 4\times \sum_{k=1}^{N} \lvert S_k\rvert$ bytes	$5\times 4\times 2\times N$ bytes
Typical total size	388,684,295.06KB	2,170,768.25KB

When the similarity score produced is significant (a hit is obtained), it is necessary to re-execute the Viterbi algorithm on the host, keeping the entire score structures in order to perform traceback. Nevertheless, the comparison of the entire Swiss-Prot database to the top twenty profile HMMs resulted in only 0.29% hits, on average. The memory requirement reduction combined with the small hit rate justify the use of Viterbi algorithm as a filter. Figure 3 shows the workflow implemented by our GPU accelerator.

In our basic GPU accelerator, all data structures required to execute the Viterbi algorithm are stored in global memory and no optimizations are applied. We compared the performance of the accelerator to HMMer2 and HMMer3 using the entire SwissProt sequence database, for each profile HMM. Table II shows the average execution time of the basic accelerator, HMMer2, and HMMer3, where the accelerator time includes the Viterbi algorithm execution on GPU and all host-GPU data transfers. The HMMer2 and HMMer3 execution times correspond to the sequential execution on the host of the main filters of the hmmsearch operation of these tools, with SSE2 instructions disabled in the latter.

Table II
PERFORMANCE COMPARISON BETWEEN BASIC GPU ACCELERATOR AND HMMER2 AND HMMER3

Average execution time (s)	Basic accelerator	HMMer2	HMMer3
	494.84	548.44	423.12

The basic accelerator is, on average, only 1.11 times faster than HMMer2 and it performs worse than HMMer3. These results show that implementing algorithms on GPU in a similar way to implementations on general-purpose processors may not produce performance gains over conventional implementations, stressing the importance of applying optimizations in order to achieve performance improvements when using GPUs.

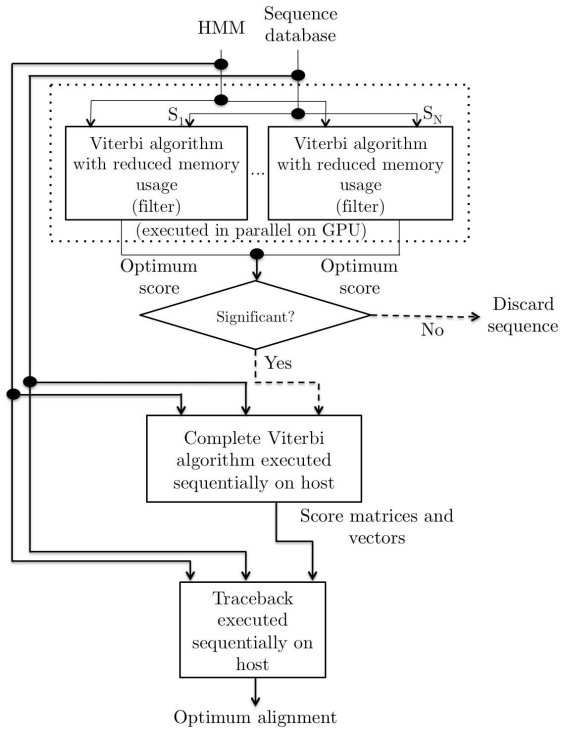

Figure 3. Workflow of the proposed accelerator on GPU

V. OPTIMIZATION STRATEGIES

This section presents the optimizations applied to our basic GPU accelerator and the results achieved.

A. Memory Optimizations

The correct use of each kind of GPU memory is important to achieve good performance. We reorganize the allocation

and access of the Viterbi algorithm data structures in order to find out the best memory to allocate each structure and the best way to access it.

In OpenCL, the instances of the Viterbi algorithm are executed by work-items organized in work-groups, which are assigned to the GPU multiprocessors. Work-items in the same work-group are divided in groups of 32 work-items, called warps, and executed in lockstep using SIMD (Single Instruction, Multiple Data) instructions. In order to optimize global memory accesses, work-items in the same warp must access words belonging to the same 128-byte memory segment, generating a coalesced access, which is served by only one 128-byte memory transaction [24].

The score matrices and vectors are the most frequently accessed structure of the Viterbi algorithm and consume much memory space, as seen in Table I, therefore the global memory is the only one that fits them. If the score matrices M are sequentially allocated in global memory, when each work-item k (associated to sequence S_{k+1}) in a warp access the element $M_{k+1}[1,1]$, uncoalesced accesses are generated to different memory segments, requiring one 128-byte memory transaction for each work-item. Therefore, we interleave the M matrices elements in a way that when each work-item k in a warp access the element $M_{k+1}[1,1]$, a coalesced access is generated to the same memory segment, requiring only one 128-byte memory transaction to serve all work-items. This strategy is applied to all score structures to achieve memory coalescency and improve performance.

Even when accesses are coalesced, if the number of sequences is not multiple of 32, some extra transactions are needed to serve some warps accessing scores. We can avoid these extra transactions by applying memory padding to the score structures in order to make their sizes multiple of 32.

We also evaluated allocating the score structures in private memory, which is an off-chip memory whose scope is local to the work-item. Private memory is mapped into global memory and used by the compiler to hold automatic variables when it determines they do not fit into registers. Figure 4 shows score matrix M allocated in private memory and how it would be automatically mapped into global memory.

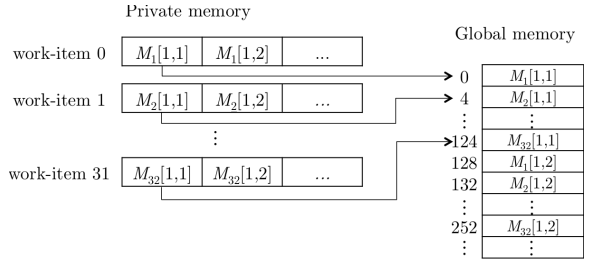

Figure 4. Matrix M allocated in private memory and mapped into global memory

As private memory resides in global memory, it is subject to the same requirements for memory coalescing, except for the fact that accesses to private memory are naturally coalesced as long as work-items in a warp access the same relative address of the structure [25].

Figure 5 shows how the position $[i,j]$ of the score matrix M is accessed in global memory and in private memory, respectively. In Figure 5(a), the expression $i \times Q \times N + j \times N$ represents an offset applied by each work-item to access the portion of M score matrix which stores the elements $M[i,j]$ of all sequences. Figure 5(b) shows that index computations are simpler in private memory, because the matrix is accessed as an ordinary kernel variable and each work-item views its matrix as a separated structure.

$$M[i \times Q \times N \ + \ j \times N \ + \ work\_item\_id]$$

(a) Access in global memory

$$M[i \times Q \ + \ j]$$

(b) Access in private memory

Figure 5. Accesses to the position $[i,j]$ of score matrix M allocated in different memories

Table III shows the average execution time of the GPU accelerator with different strategies for the allocation and access of the score structures. Although coalescency in global memory along with padding achieved an excellent performance gain, the use of private memory brought even more gain, showing that coalescency is more effective in this memory, since it is easier to achieve and it simplifies index computations.

The sequences are allocated in global memory due to the same memory limitations faced with the score structures. The same strategy applied to the score structures is used to achieve coalescency with the sequences, with the difference that, in order to facilitate interleaving, the sequences are sorted by length. The sequence access coalescency achieved an average reduction of 8% in the execution time of the basic accelerator. This reduction is smaller than the one achieved with the score coalescency because the sequences are accessed less frequently than scores.

Table III
PERFORMANCE COMPARISON OF STRATEGIES FOR SCORE STRUCTURES ALLOCATION AND ACCESS

Strategy	Average execution time (s)
Basic accelerator (scores in global memory)	144.13
Scores in global memory with coalescency	20.43
Scores in global memory with coalescency and padding	18.10
Scores in private memory	6.70

The transition probabilities are divided in two categories: regular transitions, involving states M, I, and D of the profile HMM, and special transitions, which connect only the special states B, N, E, C, and J, and are much smaller than regular ones, since they do not depend on the length of the HMM. Figure 6 shows the access pattern of work-items in a warp to the transition probabilities. Since all work-items use the same profile HMM, they always request the same transition probability. The transition probabilities are a small structure and it is possible to allocate it in the constant memory or local memory, besides the global memory.

The constant memory is a small part of the global memory which is constant and whose access is as fast as reading from registers, as long as work-items in a warp read from the

same address, which is the exact pattern shown in Figure 6. Allocating regular transitions in constant memory is quite simple and requires changing only one parameter.

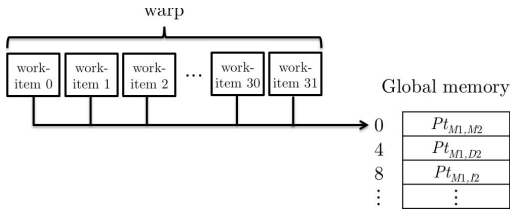

Figure 6. Access pattern of work-items in a warp for the transition probabilities

The local memory is a small on-chip memory inside each GPU multiprocessor. It is extremely fast and, when used properly, has an access latency up to 100× shorter than global memory [24]. Local memory is shared only by work-items in the same work-group, thus, each work-group must have a copy of the regular transitions in its local memory. The work-items copy the regular transitions from global memory to local memory coalescedly, in a way that no extra transactions are generated. In order to avoid serialization of accesses from different work-groups to the same transition probability, the probabilities are replicated in global memory for each work-group. Finally, a synchronization barrier is needed to ensure the work-items execute only after all the transitions are copied to local memory.

We allocated special transitions exclusively in global memory, but replicating transitions for each work-item and interleaving them, in order to produce coalesced accesses. We also evaluated allocating special transitions in local memory, but with some differences with respect to regular transitions. Since there are only a few special transitions, only a few work-items per work-group copy the transitions from global memory to local memory, while the others are kept inactive. We replicate the special transitions in local memory for each warp, so that no barrier synchronization is needed. Finally, we allocated special transitions in constant memory, however, no significant performance gain was achieved, probably because this structure is accessed less frequently than the regular transitions.

Table IV shows the results of the memory optimizations applied to the transitions. The use of local memory and constant memory for regular transitions produced similar small performance gains, although using local memory consumes more global memory (transitions were replicated in global memory) and inhibits the use of local memory for other structures that might achieve better performance gains, since this memory is very small. For special transitions, the use of global memory with coalescency and local memory also produced similar performance gains, but two conclusions can be draw. First, although special transitions are accessed less frequently than regular ones, using local memory for special transitions achieved a better performance than using it for regular transitions, due to the absence of barrier synchronization. Second, the performance gain obtained with replication and coalescency of special transitions in global memory shows that eliminating concurrent accesses of work-items in the same warp to the same position of the global memory can produce performance gains.

Table IV
PERFORMANCE COMPARISON OF STRATEGIES FOR TRANSITION PROBABILITIES STRUCTURES ALLOCATION AND ACCESS

Strategy	Avg. execution time (s)
Basic accelerator (transitions in global memory)	144.13
Regular transitions in constant memory	143.40
Regular transitions in local memory	142.54
Special transitions in global mem. w/ coalescency	140.44
Special transitions in local memory	140.99

The emission probabilities have different characteristics. They are organized as two different matrices, for emission probabilities of M and I states, and are shared among all work-items. We reorganized these matrices in global memory, transposing them, in order to improve the spatial locality. Nevertheless, it did not produce performance gains.

Since the work-items handle different sequences and the emission probabilities they need depend on the residues of the sequences, it is not possible to predict the access pattern for these probabilities. As a consequence, it is not possible to reorganize the probabilities in order to generate coalesced accesses to global memory. Their pattern of access is also not suitable for allocation in constant memory. Therefore, local memory is the one that best fits emission probabilities, because it is faster than global memory and the accesses do not have to be coalesced. We also padded some positions in the emission probabilities structure in order to avoid bank conflicts during the accesses of a warp.

Allocating emission probabilities in local memory improved the execution time in 7%, on average, compared to the basic accelerator, which allocates the probabilities in global memory. The longer the profile HMM, the better the performance improvement, because the emissions probabilities are accessed more often. Nevertheless, the insufficient space in local memory for the emission probabilities of very long profile HMMs prevent its use.

B. General Optimizations

Instruction scheduling is the process of determining the execution order of program instructions to reduce delays caused by conflicts between instructions [26]. Static instruction scheduling is usually performed by compilers during the optimization and code generation phases. Conflicts between instructions do not influence the GPU performance as long as there are available warps to execute while other warps execute long-latency arithmetic operations [24]. This happens when GPU occupancy, the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps [24], is higher than 18.75% (for devices with compute capability 1.2 or above).

Using the NVIDIA occupancy calculator, we determined the occupancy of 50% for our basic accelerator, using 256 work-items per work-group. Therefore, conflicts between instructions should not cause delays during the execution of our accelerator. Nonetheless, we applied instruction scheduling manually in the high-level source program of our accelerator,

increasing the distance between dependent instructions. We also applied the register renaming technique [26], in order to eliminate false dependencies between instructions, but increasing the amount of registers needed to execute the accelerator. To illustrate this approach, Figures 8(a) and 8(b) show the main section of the Viterbi kernel before and after instruction scheduling and register renaming were applied, respectively.

Loop unrolling consists in replacing the body of a loop with several copies of the same body, reducing the loop overhead and facilitating instruction scheduling. We applied loop unrolling to the Viterbi algorithm inner loop in our accelerator, using unroll factors 2, 4, and 8. We also evaluated combining loop unrolling and instruction scheduling.

Figure 7 compares the performance of the basic accelerator to the loop unrolled accelerator, with different unroll factors, each one with and without instruction scheduling. Instruction scheduling alone achieved performance gains over the basic accelerator, showing that, even when the GPU occupancy is higher than the recommended threshold, increasing the distance between dependent instructions can improve the performance. It also shows that the code generated by the GPU compiler is not effectively scheduled. The loop unrolling itself did not produce a significant performance gain. However, when combined with instruction scheduling, the gain is potentialized and, as the unroll factor increases, the performance gain also increases. The best performance is achieved with instruction scheduling combined to loop unrolling with factor 8. We did not applied a bigger unroll factor because the number of registers needed increases as we increase the unroll factor and we had already reached the limit of registers per work-item.

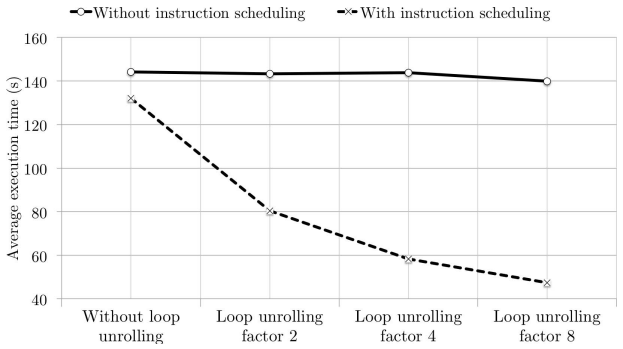

Figure 7. Performance comparison of accelerators with and without instruction scheduling and loop unrolling

It is possible to improve the performance by overlapping data transfers between host and GPU with GPU execution. We applied this technique to our accelerator, by overlapping the GPU execution with the copy of part of the sequences from host to GPU. However, it did not produce performance gains, because the time spent with data transfers is too small compared to the time spent executing the accelerator. In the basic accelerator, more than 99% of the execution time is spent in GPU computation.

In a GPU execution, a divergence happens when work-items in the same warp take different execution paths, and these paths are executed sequentially until the work-items join the same execution path again. Divergences can reduce execution parallelism on GPU and should be minimized [24]. There are three control flow structures in the Viterbi algorithm that can generate divergences: the outer loop, that iterates over the sequence residues; the inner loop, that iterates over the HMM nodes; and the conditional statements used to check the conditions of the dynamic programming recurrences shown in Section II. We created some synthetic sets of sequences to evaluate the impact of the divergences caused by each control flow structure. The inner loop does not cause divergences, because all the Viterbi algorithm instances handle the same profile HMM, then the work-items iterate over the same number of nodes. The outer loop generates divergences for sequences with different lengths. However, these divergences did not cause performance loss. Finally, the conditional statements did not generate divergences, because they guard only a few instructions and, in these cases, the compiler replaces the conditional branches with predicated instructions [24].

VI. Optimized Accelerator and Results

Based on the experiments performed, we ranked the proposed optimizations according to the performance gains they achieved, as follows:

1) Scores in private memory with coalescency;
2) Loop unrolling with unroll factor 8;
3) Instruction scheduling;
4) Sequences in global memory with coalescency and sorted by their lengths;
5) Special transition probabilities in local memory, replicated for each warp;
6) Regular transition probabilities in constant memory.

After determining the optimizations that provide performance gains, we need to evaluate how they behave in combination with the others. Table V shows the results of the execution of the accelerators created by applying the optimizations listed previously, one at a time, in that order. The subset of 70,000 sequences from SwissProt database and the profile HMM Oxidored_q1 from Pfam database, with length 270, were used in this experiment. The different accelerators evaluated are identified by the numbers of the optimizations applied, as defined in the previous optimization list.

Table V
GPU accelerator performance combining optimizations

Accelerator / Optimizations	Execution time (s)
Basic	220.85
1	10.60
1 + 2	11.94
1 + 2 + 3	8.04
1 + 3	10.40
1 + 2 + 3 + 4	5.20
1 + 2 + 3 + 4 + 5	5.17
1 + 2 + 3 + 4 + 5 + 6	4.36

The coalesced access of the scores in private memory (optimization 1) is the strategy that achieves the highest performance gain, because memory coalescency is a important optimization and the score structures are the most frequently accessed by the work-items.

```
for i = 1 to |S| do
    ...
    for j = 1 to Q do
        aux = M[i_previous, j-1] + Pt_{M_{j-1},M_j}
        if I[i_previous, j-1] + Pt_{I_{j-1},M_j} > aux then
            aux = I[i_previous, j-1] + Pt_{I_{j-1},M_j}
        end if
        if D[i_previous, j-1] + Pt_{D_{j-1},M_j} > aux then
            aux = D[i_previous, j-1] + Pt_{D_{j-1},M_j}
        end if
        if B[i_previous] + Pt_{B,M_j} > aux then
            aux = B[i_previous] + Pt_{B,M_j}
        end if
        M[i_current, j] = aux + Pe_{M_j}(s_i)
        aux = M[i_previous, j] + Pt_{M_j,I_j}
        if I[i_previous, j] + Pt_{I_j,I_j} > aux then
            aux = I[i_previous, j] + Pt_{I_j,I_j}
        end if
        I[i_current, j] = aux + Pe_{I_j}(s_i)
        aux = M[i_current, j-1] + Pt_{M_{j-1},D_j}
        if D[i_current, j-1] + Pt_{D_{j-1},D_j} > aux then
            aux = D[i_current, j-1] + Pt_{D_{j-1},D_j}
        end if
        D[i_current, j] = aux
        ...
    end for
    ...
end for
```

(a) Before instruction scheduling and register renaming

```
for i = 1 to |S| do
    ...
    for j = 1 to Q do
        aux_M = M[i_anterior, j-1] + Pt_{M_{j-1},M_j}
        aux_I = M[i_anterior, j] + Pt_{M_j,I_j}
        aux_D = M[i_atual, j-1] + Pt_{M_{j-1},D_j}
        if I[i_anterior, j-1] + Pt_{I_{j-1},M_j} > aux_M then
            aux_M = I[i_anterior, j-1] + Pt_{I_{j-1},M_j}
        end if
        if I[i_anterior, j] + Pt_{I_j,I_j} > aux_I then
            aux_I = I[i_anterior, j] + Pt_{I_j,I_j}
        end if
        if D[i_atual, j-1] + Pt_{D_{j-1},D_j} > aux_D then
            aux_D = D[i_atual, j-1] + Pt_{D_{j-1},D_j}
        end if
        if D[i_anterior, j-1] + Pt_{D_{j-1},M_j} > aux_M then
            aux_M = D[i_anterior, j-1] + Pt_{D_{j-1},M_j}
        end if
        I[i_atual, j] = aux_I + Pe_{I_j}(s_i)
        D[i_atual, j] = aux_D
        if B[i_anterior] + Pt_{B,M_j} > aux_M then
            aux_M = B[i_anterior] + Pt_{B,M_j}
        end if
        M[i_atual, j] = aux_M + Pe_{M_j}(s_i)
        ...
    end for
    ...
end for
```

(b) After instruction scheduling and register renaming

Figure 8. Instruction scheduling and register renaming applied to the Viterbi kernel, in order to reduce execution delays caused by conflicts between instructions

The combination of loop unrolling (optimization 2) to the first optimization caused a small performance loss, that was eliminated by the insertion of instruction scheduling (optimization 3). In order to evaluate if the instruction scheduling would achieve more performance gain combined with optimization 1 without loop unrolling, we implemented the accelerator version 1+3, which produced a worse performance than version 1+2+3, confirming that loop unrolling itself does not produce performance gain, but potentialize the gain of instruction scheduling.

The combination of optimization 4, the coalescency of sequences, to the previous ones, produced a good performance gain, which was enhanced by sorting the sequences, reducing wasted bandwidth during accesses to the score structures.

Allocating special transition probabilities in local memory (optimization 5) combined with previous optimizations achieved a small performance gain, proportional to the gain produced when it was applied to the basic accelerator. Optimization 6, the allocation of regular transition probabilities in constant memory, achieved a much higher performance gain when combined to optimizations 1+2+3+4+5 than when applied to the basic accelerator. Therefore, some memory optimizations may cause more impact when most of memory accesses are already optimized.

The final optimized accelerator includes optimizations 1, 2, 3, 4, 5, and 6. In order to evaluate its performance, we executed this accelerator using the entire SwissProt sequence database, sorted by length and divided in batches of 70,000 sequences due to memory limitations. The batches were executed one at a time, for each top twenty profile HMM. The sorting is performed only once and the time spent on it was negligible. Table VI shows the optimized accelerator execution time and the speedup it achieves with respect to the basic accelerator, to HMMer2, HMMer3, and GPU-HMMer [10]. The performance is also reported using the throughput measure CUPS (Cell Updates per Second), which indicates how many cells of the dynamic programming matrices are computed in one second.

The optimized GPU accelerator is on average 44.45 times faster than the basic one, even though both of them exploit the same amount of sequence parallelism, using exactly the same GPU as execution platform. Therefore, the performance gain is obtained exclusively from the optimizations applied. This shows how important it is to use the GPU resources efficiently in order to achieve performance gains.

The speedup of the optimized accelerator with respect to HMMer2 ranged from 40.35 to 102.90, with average of 49.27, as shown in Table VI. The accelerator CUPS is uniform in all executions, ranging from 8.36 to 10.05 GCUPS, with average of 8.99 GCUPS. The CUPS achieved by HMMer2 for the families Pkinase and Response_reg are much lower than for other families, probably due to a bad performance accessing host memory hierarchy. This explains the anomalous speedups of 101.75 and 102.90 achieved by our accelerator with respect to HMMer2, when processing these two families.

Table VI
PERFORMANCE COMPARISON BETWEEN OPTIMIZED GPU ACCELERATOR AND HMMER2, HMMER3, AND GPU-HMMER

Profile HMM family	Optimized accelerator					GCUPS (10^9 CUPS)			
	Execution time (s)	Speedup wrt. Basic accelerator	HMMer2	HMMer3	GPU-HMMer	Optimized accelerator	HMMer2	HMMer3	GPU-HMMer
ABC_tran	7.52	44.41	43.14	41.30	14.31	9.07	0.21	0.22	0.63
Cytochrom_B_C	6.52	44.53	42.99	54.10	14.24	9.06	0.21	0.17	0.64
Cytochrom_B_N	12.01	44.82	42.71	50.27	15.22	9.00	0.21	0.18	0.59
Pkinase	16.62	45.38	101.75	48.93	15.36	8.97	0.09	0.18	0.58
Response_reg	7.05	44.70	102.90	30.57	14.54	9.18	0.09	0.30	0.63
RVP	6.36	44.67	45.01	24.90	14.28	9.11	0.20	0.37	0.64
RVT_1	13.69	44.10	43.02	25.94	14.89	8.97	0.21	0.35	0.60
zf-C2H2	1.68	35.63	40.35	33.57	12.86	8.36	0.21	0.25	0.65
adh_short	10.81	44.05	42.28	37.19	14.51	8.89	0.21	0.24	0.61
COX1	28.98	44.08	42.87	24.36	16.05	8.82	0.21	0.36	0.55
HTH_1	3.89	43.19	42.46	25.77	14.03	9.03	0.21	0.35	0.64
Helicase_C	5.04	43.47	42.38	41.37	13.99	9.00	0.21	0.22	0.64
Oxidored_q1	17.28	44.10	42.64	39.25	15.88	8.96	0.21	0.23	0.56
WD40	2.68	40.37	40.94	35.54	12.94	8.64	0.21	0.24	0.67
BPD_transp_1	11.90	44.58	42.42	29.27	14.86	8.94	0.21	0.31	0.60
Acetyltransf_1	4.80	48.89	47.44	28.80	15.36	10.05	0.21	0.35	0.65
HATPase_c	7.13	44.01	42.57	31.07	14.20	9.00	0.21	0.29	0.63
RVT_thumb	4.54	43.32	42.27	23.62	13.87	8.99	0.21	0.38	0.65
MFS_1	22.74	44.51	43.42	60.46	15.83	8.91	0.21	0.15	0.56
GP120	31.41	45.16	43.04	38.73	16.37	8.88	0.21	0.23	0.54
Average	11.13	44.45	49.27	38.01	15.28	8.99	0.20	0.27	0.61

The speedup of the optimized accelerator with respect to HMMer3 ranged from 23.62 to 60.46, with average of 38.01. The CUPS achieved by HMMer3 is less uniform than HMMer2 because the number of calculated cells depends on its pipeline of filters, which can discard some sequences without executing all steps. We also compared our accelerator with HMMer3 with SSE2 instructions enabled and we achieved speedups up to 4.15. Our accelerator optimized the Viterbi algorithm, which is not the main step of HMMer3, and nevertheless achieved excellent speedups. If the MSV algorithm were included and optimized in our accelerator, an even better performance would be achieved.

From the main implementations of the Viterbi algorithm on GPUs, described in Section III, GPU-HMMer [10] is the only one with source code available. We executed and evaluated GPU-HMMer on the same platform used for our optimized accelerator and with the same families and sequences. Our accelerator is on average 15.28 faster than GPU-HMMer, even though the latter exploits the same sequence parallelism of ours. This experiment enabled us to factor out differences of platform and GPU generations and to compare the accelerators using the same execution platform and the same input data.

Table VII compares our accelerator to the main implementations of the Viterbi algorithm on GPUs, described in Section III, with respect to the speedups reported. GCUPS results are not reported in these works. The solutions proposed by ClawHMMer [9] and Du *et al.* [12] did not implement the full Plan7 architecture and both achieved maximum speedups lower than the speedups produced by our accelerator. GPU-HMMer [10] and CuHMMer [11] implemented the full Plan7 architecture and achieved a maximum speedup lower than the average speedup produced by our accelerator. Ganesan *et al.* [13] also implemented the full Plan7 architecture and achieved an average speedup higher than the average speedup of our accelerator. However, they used four GPUs Tesla C1060, each one containing 240 cores operating at 1.3GHz and 4GB of global memory. Li *et al.* [14] implemented the MSV algorithm and achieved speedups up to 6.5 with respect to HMMer3 SSE serial execution. They used a Tesla C2050 GPU, with 448 cores and 3GB of global memory. In our experiments we used only one simpler GPU containing 336 cores and 1GB of global memory. A GPU with more memory would allow us to process more sequences in parallel and less batches serially, producing better speedups.

Table VII
PERFORMANCE COMPARISON BETWEEN MAIN GPU ACCELERATORS FOR VITERBI ALGORITHM AND OUR OPTIMIZED ACCELERATOR

Implementation	HMM architecture	GPU	Speedup
[9]	M, I and D	ATI R520	36 (max.)
[10]	Plan7	GeForce GTX 8800 Ultra	12 to 38.6
[11]	Plan7	GeForce GTX 8800	13 to 45
[12]	M, I and D	GeForce 9800 GTX	1.97 to 72.21
[13]	Plan7	4 Tesla C1060	100
[14]	MSV	Tesla C2050	6.5 (max.)
Optimized accelerator	Plan7	GeForce GTX 460	40.35 (min.) 49.27 (avg.) 102.83 (max.)

VII. CONCLUSION AND FUTURE WORK

This paper proposed an accelerator on GPU for the sequence-profile comparison problem. The accelerator was evaluated using an intermediate GPU and real sequence database and sequence families, achieving an excellent performance, with average and maximum speedup of 38.01 and 60.46, respectively, when compared to the widely used HMMer3 tool executing on a conventional processor. Compared to HMMer2, better speedups were achieved, with average and maximum of 49.27 and 102.90, respectively. Besides,

it is the first sequence-profile comparison tool presented in the literature to use the OpenCL programming model. The performance achieved is also higher than the performance reported by other GPU accelerators described in the literature, allowing the comparison of huge sequence databases to long profile HMMs in a few seconds. Using a more powerful GPU, our accelerator would achieve an even better performance.

We performed an extensive analysis of the viability of many optimizations, which is not found in literature for the sequence-profile comparison problem. This analysis can contribute to the development of optimized GPU solutions to other problems, such as HMMer3 MSV algorithm, which is very similar and even simpler than the Viterbi algorithm.

The experiments showed that the main optimizations to be exploited on GPU involve the efficient use of the memory hierarchy through memory coalescency and reduction of memory transactions. Besides, simple optimizations such as loop unrolling and instruction scheduling help reducing conflicts between instructions, producing performance gains.

To the authors' knowledge, this is the first time the GPU global memory is used through private memory mapping for allocating the score structures, allowing more natural coalescency of accesses and simpler index computations.

Our final accelerator achieved an excellent performance, however, the basic accelerator also exploited sequence parallelism and did not produce performance gains. In this basic solution, the Viterbi algorithm was implemented without taking the characteristics of the platform into account, showing that GPU solutions without any optimizations may not produce performance gains.

Throughout the experiments we verified that classic compiler optimizations, such as instruction scheduling, were not effectively applied by the GPU tool chain. A very interesting research subject is to investigate the GPU compilation process and develop new optimizations for GPU compilers.

REFERENCES

[1] S. Eddy, *HMMER User's Guide – Biological Sequence Analysis Using Profile Hidden Markov Models – Version 2.3.2*, Howard Hughes Medical Institute, http://hmmer.janelia.org/, 2003, accessed July, 2012.

[2] ——, *HMMER User's Guide – Biological Sequence Analysis Using Profile Hidden Markov Models – Version 3.0*, Howard Hughes Medical Institute, http://hmmer.janelia.org/, 2010, accessed July, 2012.

[3] EMBL-EBI, "UniProtKB/TrEMBL Protein Database Release 2012-07 Statistics," http://www.ebi.ac.uk/uniprot/TrEMBLstats/, 2012, accessed July, 2012.

[4] M. Punta *et al.*, "The Pfam Protein Families Database," *Nucleic Acids Research*, vol. 40, no. D1, pp. 290–301, 2012.

[5] S. Eddy, "Profile Hidden Markov Models," *Bioinformatics*, vol. 14, no. 9, pp. 755–763, 1998.

[6] G. D. Forney, "The Viterbi Algorithm," *Proc. of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[7] R. P. Maddimsetty *et al.*, "Accelerator Design for Protein Sequence HMM Search," in *ACM ICS*, 2006, pp. 288–296.

[8] Khronos Group, "OpenCL – The Open Standard for Parallel Programming of Heterogeneous Systems," http://www.khronos.org/opencl, 2012, accessed July, 2012.

[9] D. Horn, M. Houston, and P. Hanrahan, "ClawHMMER: A Streaming HMMer-Search Implementation," in *ACM ICS*, 2005, p. 11.

[10] J. P. Walters *et al.*, "Evaluating the Use of GPUs in Liver Image Segmentation and HMMER Database Searches," in *IEEE IPDPS*, 2009, pp. 1–12.

[11] P. Yao *et al.*, "CuHMMer: A Load-balanced CPU-GPU Cooperative Bioinformatics Application," in *HPCS*, 2010, pp. 24–30.

[12] Z. Du, Z. Yin, and D. A. Bader, "A Tile-based Parallel Viterbi Algorithm for Biological Sequence Alignment on GPU with CUDA," in *IEEE IPDPS*, 2010, pp. 1–8.

[13] N. Ganesan *et al.*, "Accelerating HMMER on GPUs by Implementing Hybrid Data and Task Parallelism," in *ACM BCB*, 2010, pp. 418–421.

[14] X. Li, W. Han, G. Liu, H. An, M. Xu, W. Zhou, and Q. Li, "A Speculative HMMER Search Implementation on GPU," in *IEEE IPDPS*, 2012, pp. 735–741.

[15] K. Benkrid, P. Velentzas, and S. Kasap, "A High Performance Reconfigurable Core for Motif Searching Using Profile HMM," in *NASA/ESA AHS*, 2008, pp. 285–292.

[16] A. C. Jacob *et al.*, "Preliminary Results in Accelerating Profile HMM Search on FPGAs," in *IEEE IPDPS*, 2007, pp. 1–8.

[17] Y. Sun *et al.*, "Accelerating HMMer on FPGAs Using Systolic Array Based Architecture," in *IEEE IPDPS*, 2009, pp. 1–8.

[18] S. Derrien and P. Quinton, "Parallelizing HMMER for Hardware Acceleration on FPGAs," in *IEEE ASAP*, 2007, pp. 10–17.

[19] N. Abbas and S. Derrien, "Accelerating HMMER on FPGA using Parallel Prefixes and Reductions," INRIA, France, Tech. Rep. 7370, 2010.

[20] J. P. Walters, R. Darole, and V. Chaudhary, "Improving MPI-HMMER's Scalability With Paralell I/O," in *IEEE IPDPS*, 2009, pp. 1–11.

[21] K. Jiang *et al.*, "An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence-Search on a Massively Parallel System," *IEEE TPDS*, vol. 19, no. 1, pp. 15–23, 2008.

[22] J. Fang, A. L. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *ICPP*, 2011, pp. 216–225.

[23] Swiss Institute of Bioinformatics, "UniProtKB/Swiss-Prot Protein Knowledgebase Release 2012-07 Statistics," http://web.expasy.org/docs/relnotes/relstat.html, 2012, accessed July, 2012.

[24] NVIDIA Corporation, *OpenCL Best Practices Guide*, 2010.

[25] ——, *OpenCL Programming for the CUDA Architecture*, 2012.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.