2015 IEEE International Parallel and Distributed Processing Symposium Workshops

Fine-Grained Acceleration of HMMER 3.0 via Architecture-aware Optimization on Massively Parallel Processors

Hanyu Jiang
Department of Electrical and Computer Engineering
Stevens Institute of Technology
NJ, USA.
hjiang5@stevens.edu

Narayan Ganesan
Department of Electrical and Computer Engineering
Stevens Institute of Technology
NJ, USA.
nganesan@stevens.edu

***Abstract*—HMMERsearch used for protein Motif finding which is a probabilistic method based on profile hidden Markov models, is one of popular tools for protein homology sequence search. The current version of HMMER (version 3.0) is highly optimized for performance on multi-core and SSE-supported systems while maintaining accuracy.**

The computational workhorse of the HMMER 3.0 task-pipeline, the MSV and P7Viterbi stages together consume about 95% of the execution time. These two stages can prove to be a significant bottleneck for the current implementation, and can be accelerated via architecture-aware reformulation of the algorithm, along with hybrid task and data level parallelism. In this work we target the core-segments of HMMER3 hmmsearch tool viz. the MSV and the P7Viterbi and present a fine grained parallelization scheme designed and implemented on Graphics Processing Units (GPUs). This three-tiered approach, parallelizes scoring of a sequence across each warp, multiple sequences within each block and multiple blocks within the device. At the fine-grained level, this technique naturally takes advantage of the concurrency of threads within a warp, and completely eliminates the overhead of synchronization.

The HMM used for the MSV and P7Viterbi segments share several core features, with few differences. Hence the techniques developed for acceleration of the MSV segment can also be readily applied to the P7Viterbi segment. However, the presence of additional D-D transitions in the HMM for P7Viterbi induces sequential dependencies. This is handled by implementing the Lazy-F procedure as in HMMER 3.0 but for SIMT architectures in a warp-synchronous fashion.

Finally, we also study scalability across multiple devices of early Fermi Architecture. Compared to the core-segments, MSV and P7Viterbi of the optimized HMMER3 task pipeline, our implementation achieves up to 5.4-fold speedup for MSV, 2.9-fold speedup for P7viterbi and 3.8-fold speedup for combined pipeline of them on a single Kepler GPU while preserving the sensitivity and accuracy of HMMER 3.0. Multi-GPU implementation on Fermi architecture yields up to 7.8x speedup.

***Keywords*-Protein Motif Finding, HMM, Viterbi**

I. Introduction

Protein motif detection is key to identifying conserved protein domains within family of proteins as well as deducing its structure and function within the genome. The HMMER [1], [2] suite of programs is widely used for protein motif finding, building profile HMMs, scanning an entire database of HMMs for all motifs etc. The current version, HMMER, ver 3.0, is a significant improvement over its predecessor, ver 2.0 due to the scoring system used to compute the statistical significance of alignment scores. Among the suite of tools in HMMER, `HMMersearch` is used to detect a query motif among a target database of sequences. The wide applicability of motif finding, the rapid growth of the set of protein families as well as the set of known sequences has made it target of many acceleration attempts. Although the list of acceleration attempts for HMMER 2.0 [3] is not exhaustive, some representative contributions include [4], [5], [6], [7], [8], [9], [10], [11]. While HMMER 2.0 used Viterbi algorithm (for optimal alignment) to compute the scores, HMMER 3.0 follows a scoring system that computes the total log-likelihood ratios summed over all possible alignments, via the Forward-Backward algorithm [2]. Optimal alignment scores are useful in studying similarity between individual sequences (as in BLAST or Smith-Waterman algorithms for local alignment), the Forward scores are more meaningful in alignment of target protein sequences against a probabilistic model such as the HMM. Although, computing the Forward scores requires higher computational throughput (FLOPS) than Viterbi, it is amenable to parallelization. Viterbi algorithm on the other hand, imposes sequential dependencies within the dynamic programming matrix that is harder to parallelize. It is shown that [3] distribution of high-scores of optimal alignment (via Viterbi algorithm) is Gumbel distributed with parameter $\lambda = \log 2$ and that of Forward scores (total log-likelihood ratio sums) is exponentially distributed with the same $\lambda = \log 2$. Hence, it is expected that the high-scoring tails of Viterbi and Forward scores to agree with each other. This enables designing an efficient task pipeline that can pre-filter out sequences based on Viterbi scores that are not expected to score high via the Forward algorithm. Although, this pipeline removes the load off the Forward score computing stage, the Viterbi based pre-filtering is still as expensive as the scoring system employed in HMMER 2.0. In order to mitigate the computational workload on the Viterbi stage a heuristic Multiple-Segment-Viterbi (MSV) is introduced that is analogous to word hit and ungapped extension stages implemented in BLAST [12]. The MSV stage employs a much simpler Hidden Markov Model for scoring that eliminates sequential dependencies between the dynamic programming cells. Hence, the MSV score calculation can be "Vectorized" on a parallel machine while preserving the statistical distribution of significant alignments with high fidelity. Through choice of sensitivity parameters of MSV scores in HMMER 3.0, an 8-bit saturating scoring system was used whose computation can be easily vectorized by 16, 8-bit SIMD registers thus achieving 16-fold speedup on a commodity processor.

A. *Previous Work*

Due to extensive computational and scoring optimization procedures implemented in HMMER 3.0 [2], it is extremely unlikely to improve the performance further either on CPU or GPU based platforms with generic optimization techniques alone. The previous version, HMMER 2.0, is based on Viterbi algorithm which imposes sequential dependencies and proves to be the computational bottleneck. Several strategies were proposed to accelerate the underlying Viterbi score calculation, resolve dependencies and extract parallelism. `HMMERsearch` was initially parallelized for clusters via MPI in [4]. The state loop was also vectorized to process 24 HMM states in SIMD fashion or 8 state triplets at once. Implementations of `HMMERsearch` for graphics processing units (GPUs) were carried out in Claw-HMMER [5], GPU-HMMER [6]. Optimizations included the use of GPU texture and shared memory to efficiently store and retrieve the calculated partial score values. Partial prefix sums were used [7] to break the chain of dependencies in computation of Viterbi scores. This helped extract a hybrid task and data-level parallelism in order to solve the load imbalance problem that arises due to variations in sequence lengths. In addition to multiprocessor systems, a number of attempts to accelerate implementation of the HMMER recurrence have been carried out for FPGAs [8], [9], [10]. An extensive review of various acceleration attempts was compiled in [11].

However, unlike previous version of HMMER 2.0 which has been target of numerous acceleration attempts, there exist only a handful of acceleration attempts aimed to improve the performance of key segments of HMMER 3.0 pipeline. The main reason is that HMMER 3.0 is already about 100- to 1000- fold faster than HMMER 2.0 [1], implemented on the same commodity processor with SSE support. Hence alternative architectures such as FPGA [13] have been explored as an accelerator hardware for MSV and P7Viterbi segments. The Viterbi algorithm was rewritten for parallelization via prefix sums approach on the FPGA and is able to achieve comparable performance for P7Viterbi implemented on dual-core processors. However the hardware limitations on the FPGA makes this implementation suitable for smaller models (upto 512) and tiling larger models into several dataflow partitions.

GPU based works have also been reported to partially accelerate HMMER 3.0. [14] implemented a speculative method to reduce times of accessing global memory which results in speedup. This approach aims to reduce the execution time of original reduction loop empirically. In addition, the sequences were processed by one thread at a time within the MSV filter. Partial optimization was proposed in [15], which parallelizes the P7Viterbi part without considering the D-D path dependency. Although this approach claims that 14x speedup than original functions, it sacrifices probabilistic inference and sensitivity.

This rest of the paper is organized as follows. Section II provides the overview of HMMERs probabilistic inference method with heuristic algorithms in HMMER 3.0. Section III presents the GPU architecture-aware optimizations designed to accelerate the MSV and P7Viterbi segments respectively. Section IV analyzes the results, performance and scalability across varying model sizes, sequence databases and the number of GPUs. Section V and VI present discussion and conclusion followed by current and future works.

II. HMMER 3.0 Protein Motif Finding

HMMER 3.0 task pipeline is optimized for computational efficiency that employs heuristics to eliminate vast majority of low scoring sequences as well as parallelization techniques to accelerate Viterbi score computation. For a sample model size of 400 positions and the 'Envnr' sequence database consisting of 6.5 million sequences, 2.2% of the sequences cross the MSV threshold to be passed on to the P7Viterbi stage. Only 0.1% of all the sequences are passed on to the Forward-scoring stage. The corresponding execution time is close to 80% for MSV, and 15% for P7Viterbi and 4.9% for Forward-Backward stage.

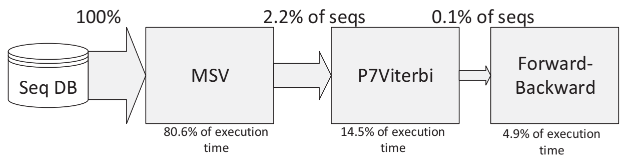

Figure 1. HMMER3 Task Pipeline

The Multiple Segment Viterbi (MSV) stage detects contiguous match alignment that is analogous to the ungapped high scoring pairs implemented in BLAST. Although BLAST uses a two-stage filter to detect and extend the ungapped alignments, the uniform entry/exit probability in the MSV model allows for partial matches upto the size of the query motif. The MSV heuristic HMM is shown in Figure 2. The full Plan 7 Viterbi is shown in Figure 3.

III. GPU Acceleration

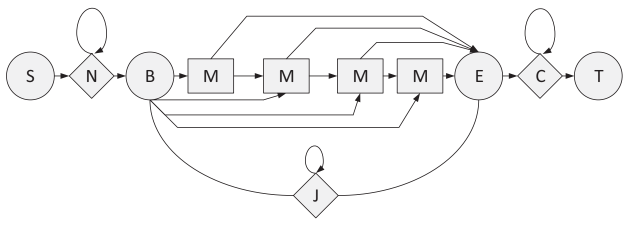

Figure 2. MSV Profile HMM

Since the majority of the execution time is spent in the MSV filtering stage, it is a prime candidate for acceleration. As vast majority of input sequences are also eliminated in this stage, any improvement in the performance will greatly impact the efficiency of the pipeline. The MSV stage exhibits regular and well-behaved dependencies that can be easily parallelized. However, in order to exceed the performance of the highly optimized MSV filter for the CPU platform in HMMER 3.0, it is imperative to go beyond generic parallelization techniques and exploit architecture-specific features.

The model simplifications in the MSV as shown in Figure 2 compared to the full core model used in P7Viterbi stage (Figure 3) eliminates the "$Delete$" states (D) that induce sequential dependencies between the cells of the dynamic programming (DP) matrix within each row. The "$Insert$" states (I) that induce dependencies to the previous rows are also eliminated, leaving only the "$Match$" states (M) that induce a diagonal dependency to cells in the previous row. Although, this simplified model is a great fit for parallelization, it contains a time consuming step that is ignored by previous acceleration attempts, viz. the synchronization overhead. The data dependency between current row and previous row of the dynamic programming matrix is shown in Figure 4. Therefore, an in-place updating of the DP row stored in the shared memory via multiple threads requires 2 synchronization calls in order to avoid potential racing hazards. The first synchronization is issued once the threads read their dependencies from shared memory. The second one is issued after the threads finish writing the new values back to the shared memory. In addition to updating cells in each row, computation of X_E which entails determining maximum of all elements in row (via parallel reduction [16]) will incur further synchronization calls. Secondly, the P7Viterbi stage incurs additional sychronization overhead due to the profile-HMM complexity. These synchronization calls impose significant overhead and degrade the performance of the entire thread-block on any parallel architecture. In the CUDA programming model, the warp schedulers select warps for execution at random within the thread block, which forces active threads to enter the idle state waiting for other threads to complete. This results in longer execution time and consumption of on-chip resources per thread within each thread-block(a single DP row). The problem is further amplified by the fact that the total number of rows is equal to the total number of collective residues contained within all sequences in the database (typically billions of residues), which can severely limit the performance.

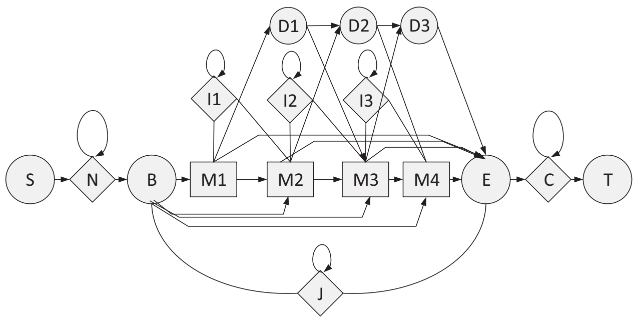

Figure 3. P7Viterbi Profile HMM

In the CPU implementation of HMMER 3.0, the cells within the row are updated in parallel via 16, 8-bit SIMD registers without any synchronizations. Hence in order to match or achieve better performance any acceleration attempt must avoid unnecessary synchronization or totally eliminate them if possible.

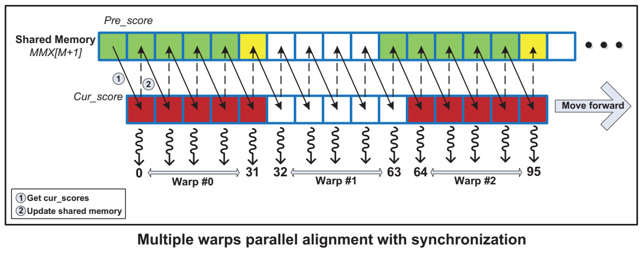

Figure 4. Multiple thread-warps with synchronization. Each thread-warp is executed synchronously by the hardware. In the absence of synchronization calls before read and write to shared memory, the cells in yellow at each warp boundary is subject to racing hazards (i.e., warp #0 and warp #1 are not selected in the same instruction issue time by SIMT unit).

A. MSV Acceleration

• **Warp-Synchronous Execution:** We exploit the fact that every 32 threads within a thread-warp are always executed synchronously by the current CUDA programming model. Hence by having a single warp update all the cells within each row, the need for synchronizations can be eliminated. In order to avoid data dependency between warps, each thread-warp processes a different sequence. The thread-warp loops over the size of the model to cover a single row of the DP matrix moving on to the successive row of the same sequence (next residue) until the entire sequence is scored. Optimal performance of shared memory for frequent reading and writing can be obtained by successive addressing of DP cells within the row. However, care must be exercised so as to avoid overwriting cells (yellow-marked) at the warp boundary due to the diagonal dependence as shown in Figure 5.

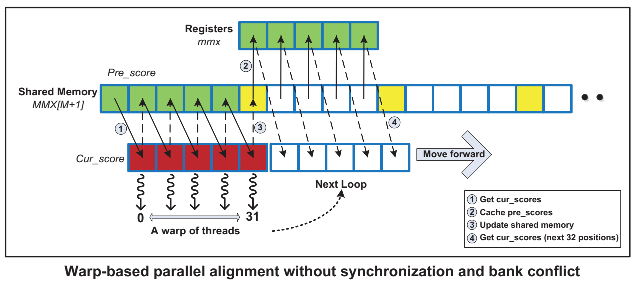

Figure 5. Single thread-warp without synchronization. Each thread-warp processes a single row by iterating till the length of the row (size of the model). Step ① to ④ is the logical sequence of instructions execution.

In order to eliminate overwriting problem with our warp-synchronous execution, double-buffering of on-chip memory is adopted. Prior to writing the values of cells back to the shared memory, the 32 dependence values corresponding to the next iteration are read into local registers. Step ① to ④ shown in Figure 5 illustrate the sequence of double-buffering in our case. Taking advantage of dual instruction dispatch units per warp scheduler of Kepler architecture, the first two independent instructions, step ① and ②, are also able to be executed concurrently, which further hides the latency of sequential execution. This method ensures that values at the warp boundary are not overwritten while updating each row in-place. It is repeated for every iteration using the same registers until the entire row is covered, before moving on to the next row for the same sequence.

In the event that a single warp finished the processing of a sequence, it automatically continues working on the next available sequence in the database asynchronously, independent of other warps within the block. This again eliminates need for any block-level coordination or stalling due to synchronization, and helps keep active threads always busy. This achieves true independence between warps and eliminates the need for synchronization throughout the course of entire execution. Other optimizations that enable achieving maximum performance include:

• **Intrinsic Conflict-Free Access:** The on-chip shared memory is fully utilized for buffering temporary values of each DP row. Due to the byte scoring system of HMMER 3.0 used in MSV, each cell of the DP matrix is represented by a single byte. As the cells are stored consecutively, every group of four threads can access a single word (4 Bytes) from a single bank of the shared memory without any bank conflicts, where each thread accesses a sub-word (8 bits). The next group of four sub-words can be similarly accessed from the successive bank, which ensures no two words fall in the same bank accessed by threads of the same warp. This naturally avoids any contention for the shared memory banks, thus achieving conflict-free access and ensuring maximum bandwidth. This optimization is achieved even when model parameters such as the transition probabilities and emission probabilities are stored and accessed within the shared memory.

• **Warp-Shuffled Reduction:** The computation of X_E, the maximum score of "$Match$" to "End" states, requires max-reduction operations over private data buffered in registers. Traditionally, this requires load and store operations via shared memory for successive binary reductions by halving the size of the array at the end of each iteration marked by a synchronization call [16]. The procedure is repeated until the final value is obtained after $\log(N)$ iterations. The use of shared memory, regular synchronization calls, as well as uneven workload on threads makes this implementation time-consuming and precious on-chip resources.

The NVIDIA Kepler series GPUs with compute capacity 3.x supports warp-shuffle instructions that enable direct exchange of private data between threads within the same warp [17]. Original separate load and store operations are merged into a single step. By using warp-shuffle instructions, like Butterfly Exchange (XOR) [18], the reduction operation can be performed with (a) even workload on all threads, (b) synchronize-free exchange of register values between threads without any shared memory, (c) and automatic broadcast of maximum value over all threads of a warp [18] which is a necessary step for next residue alignment.

• **Residue Packing:** Since the design is aimed at processing large sequence databases, it is necessary to optimize and reduce bandwidth to global memory. Each sequence residue could belong to one of 20 standard amino acids, 6 degenerate symbols and 3 gap types, as shown in Figure 6, requiring 5-bits to encode each residue with digitized values between 0 and 28. Hence, 6 consecutive residues could be packed into a single 32-bit word (int cell) that is intrinsic data type supported by CUDA. As for redundant cells (marked by red color in Figure 6 appended at the end, they are all assigned to 31 as a flag of loop termination.

The pseudo-code for synchronize-free parallel MSV alignment is shown in Algorithm 1. Lines 1-5 set up the sequence id corresponding to unique `threadIdx.y`. The `while` loop in Line 6 is executed across all warps, such that each warp receives a unique sequence id. The `while` loop in Line 10, is over each row of the DP matrix corresponding to each residue within a single sequence. Line 13, loads the row values to the register in sets of 32, corresponding to each warp. The `For` loop in Line 14, covers the entire row. Line 17 describes read-before-write in order to avoid racing hazards followed by writing the cell values to shared

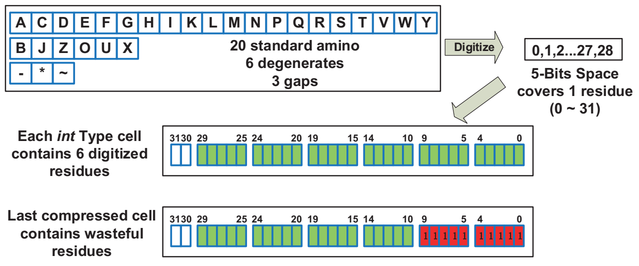
A C D E F G H I K L M N P Q R S T V W Y
B J Z O U X
- * ~
20 standard amino
6 degenerates
3 gaps
Digitize
0,1,2...27,28
5-Bits Space covers 1 residue (0 ~ 31)
Each *int* Type cell contains 6 digitized residues
31 30 29 25 24 20 19 15 14 10 9 5 4 0
Last compressed cell contains wasteful residues
1 1 1 1 1 1 1 1 1 1

Figure 6. Compression of residues. 6 consecutive sequence residues are packed into a single 32-bit word.

memory in Line 18. Line 20 shows the warp shuffle to compute max for each row, and the final scores are written back to the shared memory in Line 23.

Algorithm 1 Parallel MSV Alignment: In current execution configuration each thread-warp is indexed by a unique `threadIdx.y`. Threads with different `threadIdx.y` fall in different warps with `blockDim.x=32`.

```
Require: Sequences from database
Ensure: MSV score for each sequence (in nats)
 1: $row \leftarrow blockIdx.y * blockDim.y + threadIdx.y$
 2: $(Idx, Idy) \leftarrow (threadIdx.x, threadIdx.y)$
 3: $duty\_span \leftarrow rows\_in\_block * gridDim.y$
 4: $count \leftarrow 0$
 5: while $row + duty\_span * count < total\_seq$ do
 6:     $seqid \leftarrow row + duty\_span * count$
 7:     $i \leftarrow 0$
 8:     $len \leftarrow length[seqid]$
 9:     $mmx, share\_mem \leftarrow 0$
10:     while $i < len$ do
11:         $res \leftarrow$ decode $seq[seqid][i]$
12:         $xE \leftarrow 0$
13:         Load($mmx$)   ▷ Load first 32 values of previous row
    from Shared Memory
14:         for $p \leftarrow Idx; p < hmm\_size; p+ = 32$ do
15:             $temp \leftarrow \max(mmx, xB) + bias - \text{em}(res, p)$
16:             $xE \leftarrow \max(xE, temp)$
17:             Load($mmx$)                  ▷ Load next 32 values
18:             $share\_mem[Idy][p + 1] \leftarrow temp$
19:         end for
20:         $xE \leftarrow$ warp_shuffle_max($xE$) Update $xE, xJ, xB$
21:         $i+ = 1$
22:     end while
23:     $Final\_score[seqid] \leftarrow score(xJ)$   ▷ Save Final Score
24:     $count+ = 1$
25: end while
```

B. P7Viterbi Acceleration

The next highest execution time in the pipeline is occupied by the P7Viterbi stage, whose acceleration will only have a modest impact on the pipeline efficiency. The P7Viterbi profile HMM as shown in Figure 3 consists of the core model states along with "$Insert$" and "$Delete$" states. The presence of the "$Delete$" imposes sequential dependence between cells within the same row of the dynamic matrix. Fortunately, since the D-D path is only rarely taken, Lazy-F evaluation as described in [19], implemented in HMMER 3.0, can be used to evaluate the dependent cells. All the architecture-aware optimization techniques implemented for the MSV stage can also be applied to the P7Viterbi as the data dependency pattern for the "$Match$" states is identical to the MSV filter.

The pseudo-code for P7Viterbi stage is shown in Algorithm 2. The inner `For` loop in Line 14 mainly calculates current values of M, I and D states. However, unlike M and I, the D score ($temp\_d$) is a partial value that is only contributed by M-D transition path. As for the D-D path, a delayed checking of any dependencies that can improve this D score is set to detect whether D-D update is necessary. Any D scores with such necessity have to be updated by Lazy-F step in Line 25.

Algorithm 2 Parallel P7Viterbi Alignment: In current execution configuration each thread-warp is indexed by a unique `threadIdx.y`. Threads with different `threadIdx.y` fall in different warps with `blockDim.x=32`.

```
Require: Sequences from database
Ensure: Viterbi score for each sequence (in nats)
 1: $row \leftarrow blockIdx.y * blockDim.y + threadIdx.y$
 2: $(Idx, Idy) \leftarrow (threadIdx.x, threadIdx.y)$
 3: $duty\_span \leftarrow rows\_in\_block * gridDim.y$
 4: $count \leftarrow 0$
 5: while $row + duty\_span * count < total\_seq$ do
 6:     $seqid \leftarrow row + duty\_span * count$
 7:     $i \leftarrow 0$
 8:     $len \leftarrow length[seqid]$
 9:     $mmx, imx, dmx, share\_mem \leftarrow -32768$
10:     while $i < len$ do
11:         $res \leftarrow$ decode $seq[seqid][i]$
12:         $xE \leftarrow -32768$
13:         Load($mmx, imx, dmx$)          ▷ Load first 32 values
14:         for $p \leftarrow Idx; p < hmm\_size; p+ = 32$ do
15:             $temp\_i \leftarrow \max(V\_ii, V\_mi)$
16:             $temp\_m \leftarrow \max(V\_bm, V\_mm, V\_im, V\_dm)$
17:             $temp\_d \leftarrow temp\_m + T_{md}$
18:             $xE \leftarrow \max(xE, temp\_m)$
19:             Load($mmx, imx, dmx$)    ▷ Load next 32 values
20:             $shared\_mem[Idy][p + 1] \leftarrow (temp\_m/i/d)$
21:         end for
22:         $xE \leftarrow$ warp_shuffle_max($xE$)
23:         $Dmax \leftarrow$ warp_shuffle_max($Dmax$)
24:         Update $xC, xJ, xB$
25:         Update $Delete$ state by Lazy_F
26:         $i+ = 1$
27:     end while
28:     $Final\_score[seqid] \leftarrow score(xC)$   ▷ Save Final Score
29:     $count+ = 1$
30: end while
```

• **Parallel Lazy-F:** Considering the significance of on-chip memory resources to our strategy, we implement parallel Lazy-F for SIMT processors as shown in Figure 7, in order to handle sequential dependencies caused by D-

D transitions. Within each warp, the threads concurrently calculate the D-D transition scores and checks the needs of updating their local D scores. If none of the D scores within the current warp need updating, then the D scores are final. This procedure is also warp-based with no demand of synchronization.

Within parallel inner loop, a warp-vote instruction `__all(MD_score > DD_score)` is used as a conditional statement to ensure that all current 32 positions of shared memory store highest D scores. Otherwise, the inner loop is repeated until this statement is true. In the event that scores of specific positions are influenced by the D-D transition, only those affected positions need to be updated in parallel instead of examining and updating every position in sequence. Since a large number of positions do not require the D-D transition, this update can be ignored which greatly reduces the time to evaluate the "*Delete*" score, which is one of the primary bottleneck in other acceleration attempts for HMMER 3.0. Compared to traditional method of prefix sums [13], the Lazy-F strategy requires fewer on-chip memory resources and instructions, which is beneficial to warp-based algorithms that largely consume shared memory and registers.

C. Three-Tiered Parallelization

Both MSV and P7Viterbi algorithm are built under the framework of three-tiered parallelization. This is a fine-grained strategy that (a) parallelly scores each sequence via a single warp, (b) concurrently processes multiple protein sequences in the same thread-block and (c) resides multiple blocks on a single streaming multiprocessor (SM on Fermi / SMX on Kepler) within entire thread-grid as shown in Figure 8. This framework is an architecture-aware optimization that fully takes advantage of available hardware resources and intrinsic features of CUDA programming model, which can be easily applied to other data-independent, large-scaled and intensively computational problems.

IV. Results

The GPU accelerated MSV and Viterbi tasks in the pipeline were tested against various HMMs of sizes 48, 100, 200, 400, 800, 1002, 1528, 2405 representative of motifs of different protein families from small to large in the Pfam HMM database [20]. The Pfam database (pfamA and pfamB) consists of a total of 34,831 protein families with 84.5% of models of size 400 or lesser, 14.4% of the models with size between 400 and 1000, and 1.1% of modes of size 1000 or greater. NVIDIA-Kepler series Tesla K40 was used for the single GPU implementation and HMMER 3.0 utilizing multi-core and SSE capabilities on Intel Core i5 quad core CPU running 64-bit Ubuntu at 3.4GHz was used for baseline speedup calculations. This benchmark environment is similar to [21] which proposed a fine-grained optimization of BLAST on GPU.

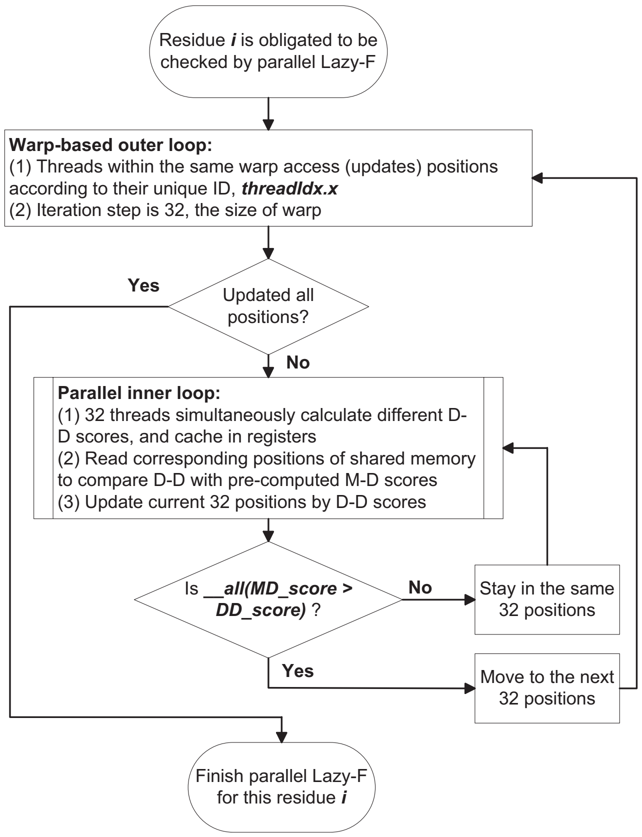
Residue ***i*** is obligated to be checked by parallel Lazy-F
Warp-based outer loop:
(1) Threads within the same warp access (updates) positions according to their unique ID, ***threadIdx.x***
(2) Iteration step is 32, the size of warp
Yes
Updated all positions?
No
Parallel inner loop:
(1) 32 threads simultaneously calculate different D-D scores, and cache in registers
(2) Read corresponding positions of shared memory to compare D-D with pre-computed M-D scores
(3) Update current 32 positions by D-D scores
Is ***__all(MD_score > DD_score)*** ?
No
Stay in the same 32 positions
Yes
Move to the next 32 positions
Finish parallel Lazy-F for this residue ***i***

Figure 7. The Implementation of Parallel Lazy-F. Selected residues need pass through this checking procedure to update D state scores.

Figure 9 shows stage-wise speedup obtained for MSV and Viterbi stages for two different sequence databases, Swissprot database consisting of 459,565 sequences with a total of 171,731,281 residues and Envnr database consisting of 6,549,721 sequences with a total of 1,290,247,663 residues. Two different configurations: 1) storing the entire model parameters (transition, emission probabilities) in the shared memory and 2) storing the model parameters in the

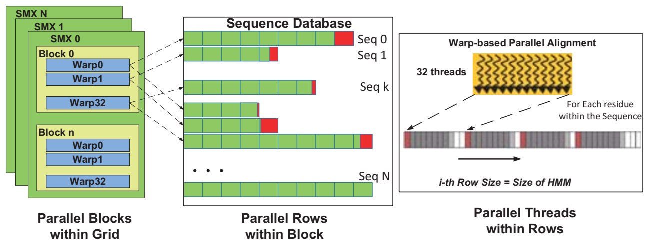
SMX N
SMX 1
SMX 0
Block 0
Warp0
Warp1
Warp32
Block n
Warp0
Warp1
Warp32
Sequence Database
Seq 0
Seq 1
Seq k
Seq N
Warp-based Parallel Alignment
32 threads
For Each residue within the Sequence
i-th Row Size = Size of HMM
Parallel Blocks within Grid
Parallel Rows within Block
Parallel Threads within Rows

Figure 8. Three-Tiered Parallelization. This framework consists of three levels: (a) multiple blocks resident on a single SMX; (b) multiple sequences (row) processed within a single block; (c) each residue alignment processed by a warp of 32 threads. The red-colored parts represent wasteful residues.

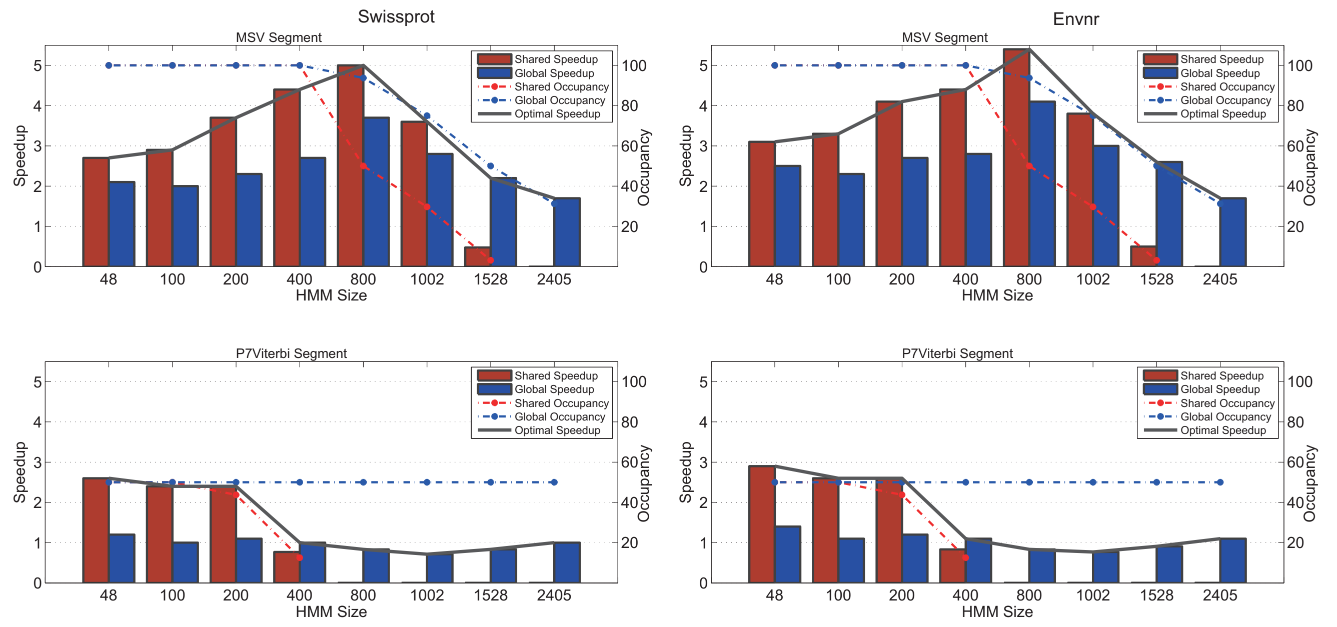

Figure 9. Speedup for Individual Stages of `HMMsearch`, for Swissprot and Envnr databases. The red and blue bars show the speedup obtained via shared memory and global memory configurations. The red curve shows the device occupancy for the shared memory configuration and the blue curve for global memory configuration. The black curve shows the optimal speedup strategy by switching between shared memory configuration for smaller models to global memory configuration for larger models. Occupancy refers to the ratio of the total number of resident threads (warps) and the maximum theoretical number of threads per multiprocessor.

global memory were implemented according to consumption of on-chip memory resources which is directly influenced by size of models. For MSV models that simplify the core HMM models of size 1528 could be accommodated within the shared memory. The device occupancy is 100% for models of size less than 400 and achieves a peak speedup of 5.0x for model size 800. However, due to increased shared memory usage for larger models, the device occupancy drastically decreases. The device occupancy can however be increased for large models by storing the model parameters in the global memory. For models of size less than about 1002, the shared memory configuration gives better performance for both Swissprot and Envnr databases. As for models of size larger than 1002 approximately, only few threads can be resided on multiprocessors with shared memory configuration, and the global memory configuration gives better performance due to increased device occupancy though accessing to global memory leads higher latency. The optimal speedup strategy would switch between shared and global memory configurations based on a threshold of size 1002 for MSV stage.

The speedup obtained bears a strong correlation to the occupancy hence as a thumb-rule increasing the device occupancy increases the performance for both MSV as well as P7Viterbi stages. As the majority of use-case models, about 98.9% of Pfam database, have size less than 1002, the presented technique will offer greater benefits to vast majority of common use cases.

Secondly, due to increased register and shared memory usage for computing Viterbi scores and buffering the core model of P7Viterbi stage, the device peak occupancy is limited to 50% with the speedup up to 2.9x and decreases rapidly for models of size greater than 200. The amount of available registers per SM/SMX becomes main limiting factor of occupancy in this case. The red curve shows the device occupancy for the shared memory configuration and the blue curve for global memory configuration.

Figure 10 shows the overall speedup obtained for the combined MSV, P7Viterbi stages implemented on the GPU. The maximum speedups reach to 3.0-fold and 3.8-fold for Swissprot and Envnr databases respectively.

A. *Multi-GPU systems*

Finally, the above acceleration strategies were implemented on multi-GPU platforms in order to study the scalability of the application. Since the processing of the sequence database can be easily parallelized across multiple devices without any dependencies, the expected speedup gained via multi-GPU implementation is almost linear. Using the same databases, Figure 11 shows the overall speedups of up to 5.6x and 7.8x obtained for the combined MSV, P7Viterbi stage implemented on four GTX 580s based on NVIDIA-Fermi architecture.

The multi-GPU implementation was also demonstrates the portability of the application to earlier NVIDIA-Fermi based architectures that are still widely being used, however, with some key performance differences. First, the Fermi GPU architecture is not equipped with inter-thread exchange and broadcast of private thread data, hence some of the warp shuffle reduction operations are carried out with the help of shared memory. This increases shared memory usage and decreases device occupancy. Second, Fermi architecture is equipped with 32KB of registers per SM as opposed to 64KB of registers on the Kepler, which further decreases its occupancy. Neverthless, the acceleration strategies are robust enough to perform on par with highly-optimized implementation of HMMER 3.0 on quad-core Intel i5 CPUs and demonstrate linear speedup with number of GPUs.

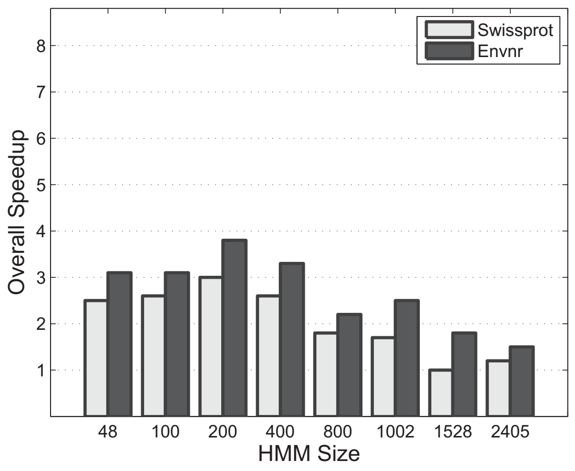

Figure 10. Overall speedup for Swissprot and Envnr databases on a single NVIDIA Tesla K40.

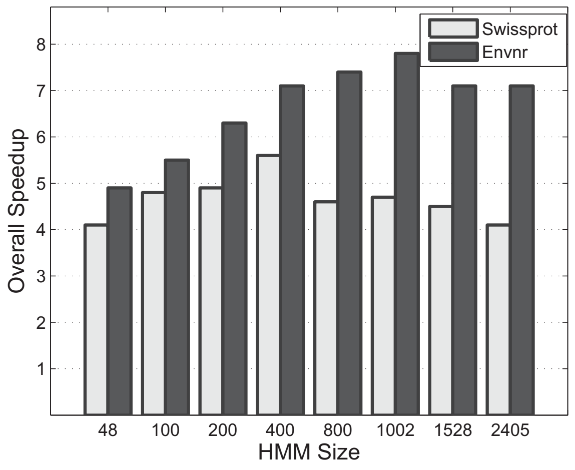

Figure 11. Overall speedup for Swissprot and Envnr databases on 4 GTX 580 GPUs based on NVIDIA-Fermi Architecture.

V. Discussion

It is observed that the Envnr database yields better speedup compared to the Swissprot database. This is due to the fact the MSV stage yields higher speedup compared to Viterbi stage. A greater execution time ratio of MSV to P7Viterbi would yield a higher overall speedup. The execution time ratio is directly dependent on the percentage of sequences that pass each successive stage. Hence protein sequences that have a higher degree of homology to query model will have lower MSV to P7Viterbi execution ratio than protein sequences that have a lower degree of homology. Hence the overall speedup obtained is not only dependent on the acceleration methodologies but also on the similarity between the target database and query model.

These two core algorithms within `HMMERSearch` application are memory-bandwidth bound, as the innermost loop in both the MSV as well as P7Viterbi have low arithmetic intensity due to the amount of data read and the number of arithmetic instructions performed. Hence any further improvements on the performance of the application would directly depend on the performance of shared memory and global memory like available size, bandwidth, access speed, etc.

VI. Conclusion and Future Work

In this work, we demonstrate the payoffs of architecture-aware optimizations implemented on the GPUs for the HMMERsearch application. The three-tiered parallelization scheme designed and implemented on Graphics Processing Units(GPUs) included optimizations at the warp-level and synchronize-free processing of target sequences. Additional improvements such as warp shuffle reduction, conflict-free shared memory access etc, improved the performance compared to generic parallelization techniques. Furthermore, the entire sequence database was parallelized across multiple warps, blocks and devices. This fine-grained strategy is fit for optimizing other analogous applications. We also propose a parallel Lazy-F implemention for resolving strong D-D dependency with less time consumption.

Our method takes advantage of the hardware cache configuration of the GPU architecture. We explore different cache configurations for strong scalability analysis. This cache-aware method switches between the (write-enabled) shared memory for smaller HMM sizes and global memory for larger HMMs in order to increase occupancy and maintain strong scalability despite device limitations. Finally, we also study scalability across multiple devices in order to enhance the applicability of the proposed technique to earlier architectures.

One of the bottlenecks of the Viterbi algorithm is the Lazy-F evaluation which iteratively recalculates the delete states if the D-D path is found to be taken by the optimal alignment. While the number of D-D transitions is very low for smaller models, it can prove to be expensive for larger models with as much as 80% of D-D transitions being taken. Hence, in order to accelerate evaluation of sequential dependencies, parallel prefix sums can be employed to establish a upper bound in the number of iterations. Our previous work demonstrated the applicability of this technique and is currently being investigated for the present application. Heterogeneous computing platform such as FPGA and GPUs are also currently being explored to accelerate the application.

Acknowledgment

The authors would like to thank the Xilinx University Program(XUP) and the NVIDIA-Professor partnership for their generous support and donation helpful in carrying out this research.

References

[1] S. Eddy, "Profile hidden markov models," *Bioinformatics*, vol. 14, pp. 755–763, 1998.

[2] ——, "Accelerated profile HMM searches," *PLoS Comput Biol*, vol. 7, no. 10, 2011, doi:10.1371/journal.pcbi.1002195.

[3] ——, "A probabilistic model of local sequence alignment that simplifies statistical significance estimation," *PLoS Comput Biol*, vol. 4, no. 5, 2008, doi:10.1371/journal.pcbi.1000069.

[4] E. Lindahl, "Altivec HMMer, version 2.3.2," http://powerdev.osuosl.org/project/hmmerAltivecGen2mod/.

[5] D. Horn, M. Houston, and P. Hanrahan, "ClawHMMER: A streaming HMMer-search implementation," in *Proc. of ACM/IEEE Supercomputing Conf.*, 2005.

[6] J. P. Walters, V. Balu, S. Kompalli, and V. Chaudhary, "Evaluating the use of gpus in liver image segmentation and hmmer database searches," in *Proc. of IPDPS*, 2009.

[7] N. Ganesan, R. D. Chamberlain, J. Buhler, and M. Taufer, "Accelerating HMMER on GPUs by implementing hybrid data and task parallelism," in *Proc. of the First ACM Int. Conf. on Bioinformatics and Computational Biology (ACM BCB)*, 2010.

[8] R. Maddimsetty, J. Buhler, R. Chamberlain, M. Franklin, and B. Harris, "Accelerator design for protein sequence HMM search," in *Proc. 20th ACM International Conference on Supercomputing*, 2006.

[9] T. Oliver, L. Y. Yeow, and B. Schmidt, "Integrating FPGA acceleration into HMMer," *Parallel Computing*, vol. 34, no. 11, pp. 681–691, 2008.

[10] T. Takagi and T. Maruyama, "Accelerating HMMER search using FPGA," in *IEICE Tech. Rep., RECONF2009-6*, vol. 109, no. 26, 2009, pp. 31–36.

[11] X. Meng and Y. Ji, "Modern computational techniques for the HMMER sequence analysis," *ISRN Bioinformatics*, no. 252183, 2013, doi:10.1155/2013/252183.

[12] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[13] N. Abbas, S. Rajopadhye, and P. Quinton, "Accelerating HMMER on FPGA using parallel prefixes and reductions," in *International Conference on Field-Programmable Technology (FPT)*, 2010, pp. 37–44.

[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 26th IPDPS Workshop and PhD Forum*, 2012, pp. 73–74.

[15] S. Quirem, F. Ahmed, and B. K. Lee, "CUDA acceleration of P7Viterbi algorithm in HMMER 3.0," in *IEEE 30th IPCCC*, 2011, pp. 1–2.

[16] M. Harris, "Optimizing parallel reduction in CUDA," http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[17] NVIDIA, "NVIDIAs next generation CUDA compute architecture: Kepler gk110," http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[18] NVIDIA-Developer, "Faster parallel reduction," http://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/.

[19] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, pp. 156–161, 2007.

[20] Pfam-Database, "Pfam, version 27.0," http://pfam.xfam.org/.

[21] J. Zhang, H. Wang, H. Lin, and W. chun Feng, "cuBLASTP: Fine-grained parallelization of protein sequence search on a GPU," in *28th International Parallel and Distributed Processing Symposium*, 2014, pp. 251–260.