Acceleration of Single- and Multiple-Segment Viterbi Algorithms for Biological Sequence-Profile Comparison on GPU

Alcides C. de Araújo Neto  Nahri Moreano
Federal University of Mato Grosso do Sul, Brazil
Email: alcides.araudio@hotmail.com, nahri@facom.ufms.br

Abstract—Over the past few decades the amount of biological data in genomic databases grew up in an exponential rate. Tools such as HMMER use the Viterbi algorithm to find biological sequences that are homologue to a family of sequences represented by a statistical model called profile HMM. Due to the quadratic time complexity of the Viterbi algorithm, this search procedure can demand long execution times depending on database size, sequence size, profile HMM length, and platform used. This paper presents the development and optimization of a high performance solution for the problem of sequence-profile comparison on GPU. We performed a detailed evaluation of several optimizations such as memory optimizations and padding, loop unrolling, multiple streams to enable computation and transfers overlapping, vectorized access to data structures, and tiling. The proposed solution achieved speedups up to 8.8 and 471.7, with respect to HMMER 3.1 execution on a quad-core computer, with and without the use of vector instructions respectively.

Keywords: Sequence-profile alignment, Viterbi algorithm, HMMER, GPU.

1. Introduction

In the past years, new DNA sequencing technologies have been causing genomic databases to grow in an almost exponential rate. The protein database UniProtKB/TrEMBL [1], for instance, nearly doubled its size every two years since 2000. As a consequence, a huge amount of new genomic data needs to be analyzed, in order to determine their functional content.

The sequence-profile comparison problem, i.e., determining if a newly identified biological sequence is homologous to a known family of sequences is a task of great importance in Bioinformatics. The classification of the new sequence as part of the family allows inferring the function and/or structure of the sequence. HMMER [2], [3], a software solution for conventional computers, is one of the main tools used for this purpose and is based on an important algorithm called Viterbi algorithm [4].

Given the quadratic time complexity of the Viterbi algorithm, comparisons of large sequence databases, long sequences, and long families may result in lengthy execution times. With the rapid growth of biological databases, these execution times become even more critical. Therefore, there is a need for high-performance solutions capable of comparing large amounts of sequences and families in a short time, by the adoption of heuristics and/or exploitation of parallelism.

This paper presents the development of a solution to the sequence-profile comparison problem, using a GPU as execution platform. We apply and evaluate several optimizations such as memory optimizations, memory padding, loop unrolling, multiple streams to enable computation and transfers overlapping, vectorized access to data structures, and tiling. The goal is to achieve a high-performance solution that allows the analysis of large biological databases in an efficient way.

To the authors’ knowledge, this is the first GPU-based system proposed for the acceleration of the sequence-profile comparison, which implements the new SSV (Single Segment Viterbi) algorithm, introduced in HMMER 3.1 version, besides the MSV (Multiple Segment Viterbi) algorithm.

This paper is organized as follows. Section 2 introduces the basic concepts needed to understand the sequence-profile comparison problem. In Section 3 we describe related works in parallel sequence-profile comparison. Section 4 presents our GPU solution to this problem and the optimizations we applied to the solution. Section 5 presents the GPU solution experimental performance evaluation and discusses the results obtained. Finally, in Section 6 we draw some conclusions.

2. Sequence-Profile Comparison

A profile HMM (Hidden Markov Model) is a statistical model that represents a family of sequences describing the similarity between members in the form of discrete states. It is based on a multiple alignment of the sequences in the family and represents how conserved each column of the multiple alignment is and which symbols are more likely [5].

A profile HMM representing a family can be used to search for new members of that family in a database of biological sequences [6]. The probability that a sequence $S$ is homologous to a family modeled by a profile HMM $\lambda$ is determined by finding the sequence of states of $\lambda$ that produces $S$ with highest probability $P(S|\lambda)$. This score measures the similarity between $S$ and the family modeled, and if it is significant, $S$ is classified as member of the family.

2.1 HMMER

HMMER [3] is a set of tools widely adopted for the analysis of biological sequences. One of its most important tools is hmmsearch, which finds biological sequences that are
Algorithm 1 Viterbi algorithm

Input: Profile HMM with length $Q$, emission probabilities $P_{em}$, transition probabilities $P_{tr}$, sequence $S$ with length $L$

Output: Score of the best alignment of $S$ with the profile HMM

for $i \leftarrow 1$ to $L$ do
  for $j \leftarrow 1$ to $Q$ do
    $M[i,j] \leftarrow P_{em}(M[j],S_i) + \max \left\{ \begin{array}{l} M[i-1,j-1] + P_{tr}(M[j],M[j]) \\ I[i-1,j-1] + P_{tr}(I[j],M[j]) \\ D[i-1,j-1] + P_{tr}(D[j-1],M[j]) \end{array} \right.$
  end for
end for

Algorithm 2 MSV algorithm

Input: Profile HMM with length $Q$, emission probabilities $P_{em}$, transition probabilities $P_{tr}$, sequence $S$ with length $L$

Output: Score of the best alignment of $S$ with the profile HMM

for $i \leftarrow 1$ to $L$ do
  for $j \leftarrow 1$ to $Q$ do
    $M[i,j] \leftarrow P_{em}(M[j],S_i) + \max \left\{ \begin{array}{l} M[i-1,j-1] + P_{tr}(M[j],M[j]) \\ I[i-1,j-1] + P_{tr}(I[j],M[j]) \end{array} \right.$
  end for
end for

return $C[L] + P_{tr}(C,T)$
Algorithm 3 SSV algorithm

Input: Profile HMM with length $Q$.
emission probabilities $P_{em}$, transition probabilities $P_{tr}$, sequence $S$ with length $L$

Output: Score of the best alignment of $S$ with the profile HMM

for $i \leftarrow 1$ to $L$
do
  for $j \leftarrow 1$ to $Q$
do
    $M[i, j] \leftarrow P_{em}(M[j, S_i]) + M[i - 1, j - 1]$
  end for
end for

$E[i] \leftarrow \max\left\{ E[i - 1], \max_{1 \leq j \leq Q}(M[i, j]) \right\}$

end for

return $E[L]$

Figure 2(a) shows the score matrix and vectors used in the MSV algorithm and represents the data dependences for computing the scores by arrows. Although it has the same time complexity as the Viterbi algorithm, the MSV algorithm performs fewer computations and has less data dependences, due to states $I$ and $D$ removal. As a consequence, all cells in a same row of matrix $M$ can be computed in parallel, while successive rows must still be computed sequentially.

3. Related Work

Cluster-based solutions are used to improve HMMER2 performance, exploiting sequence parallelism only in Viterbi algorithm [8], [9]. The idea is that, when comparing a set of sequences to a profile HMM, there are no dependences between the score matrices corresponding to two distinct sequences, therefore the matrices can be computed in parallel.

Most FPGA (Field-Programmable Gate Array) solutions [10], [11], [12], [13] use a systolic array and implement only the HMMER2 Viterbi algorithm, eliminating state $J$ and exploiting anti-diagonal data parallelism, but decreasing similarity score accuracy. Some accelerators exploit limited sequence parallelism [11], [14], while others apply strategies to reduce the accuracy loss [12], [13]. Abbas and Derrien [15] implement a FPGA accelerator for the HMMER3 MSV and Viterbi algorithms, rewriting the recurrence equations to expose more parallelism.

ClawHMMER [16] is an implementation of only the Viterbi algorithm on GPU using the Brook language. It uses profile HMMs with only $M$, $I$, and $D$ states and exploits sequence parallelism. The sequences are sorted by length and divided into batches, in order to fit in GPU memory and provide load balance. Executing on an ATI R520, it reached a speedup of 36 compared to HMMER2 executing on Intel Pentium 4 2.8GHz.

Walters et al. [17] implements only the Viterbi algorithm on GPU, using the CUDA programming model and exploiting sequence parallelism. The sequences are sorted by length to provide load balance and the inner loop of Viterbi algorithm is unrolled. Score matrices are stored in GPU global memory and accessed with coalescence, while transition and emission probabilities are kept in constant and texture memory. Using a GeForce GTX 8800 Ultra, they reached speedups between 12 and 38.6 compared to HMMER2 executing on AMD Athlon 2.2GHz.

CuHMMER [18] also implements only the Viterbi algorithm on GPU with CUDA, exploiting sequence parallelism. 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. Using a GeForce GTX 8800, it reached speedups between 13 and 45 compared to HMMER2 executing on AMD Athlon64 X2 Dual Core processor.

Du et al. [19] implements only the Viterbi algorithm on GPU using CUDA and profile HMMs with only $M$, $I$, and $D$ states. Since the $J$ state does not exist, the feedback loop is broken and it is possible to compute $M$, $I$, and $D$ anti-diagonal cells in parallel, one anti-diagonal at a time. They implement three different approaches concerning the score matrices storage. Using a GeForce GTX 9800, they reached
speedups between 1.97 and 72.21 compared to HMMER 2 executing on Intel Dual Core 2.83GHz.

Ganesan et al. [20] implements only the Viterbi algorithm on GPU using CUDA. They iterate the Viterbi algorithm recurrences, allowing cells of the same row of $M$ and $D$ score matrices to be computed in parallel, while successive rows are computed sequentially. Using a cluster of four NVIDIA Tesla C1060, they reached a speedup of 100 compared to a serial implementation of the Viterbi algorithm executing on AMD Opteron 2.33GHz.

Quirem et al. [21] implements only the MSV algorithm on GPU using CUDA. Each sequence to be analyzed is assigned to a different block, and the threads of a block compute the cells in a same row of the score matrix in parallel. Optimizations such as asynchronous data transfer and kernel execution, and the use of pinned memory are applied. They achieved speedups between 10 and 15, executing on a NVIDIA Tesla C1060, compared to HMMER 3 executed on the host.

Li et al. [22] also implements only the MSV algorithm on GPU using CUDA. They perform coalesced memory access, fetch multiple data chunks from memory at once, keep frequently used data in registers, convert sequences symbols to numbers to simplify operations, sort the sequences by length to provide load balance, perform asynchronous transfers, and store the probabilities in the texture memory. A speculative approach for the MSV algorithm is adopted, where the outer loop is unrolled by a factor of 2 and the score $B[i]$ is computed without considering the transition $J \rightarrow B$. When the speculation fails, the sequence score is recalculated on the host. Executing on Intel Xeon E5506 with a NVIDIA Tesla C2050, they achieved speedups up to 6.5 compared to HMMER 3 with SSE executed on a single core.

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. CUPS performance results are not reported for the described GPU solutions.

To the best of our knowledge, there are not in the literature solutions implementing the SSV algorithm in GPU or other parallel computing platform, apart from the HMMER 3.1 tool suite. As a consequence, all works described in this section compared their results to previous and slower versions of HMMER, in which the fast SSV algorithm was not used.

4. GPU Solution to Sequence-Profile Comparison

We developed a host-GPU solution for the sequence-profile comparison, using C++ and CUDA. The solution receives as inputs a set of sequences to be compared and a profile HMM representing a family, and computes the similarity score of the best alignment between each sequence and the profile HMM.

4.1 Chain of Filters

Our solution implements the chain of filters shown in Figure 3. Each box represents a filter which executes an algorithm and the arrows represent the paths that a sequence being compared can take. Each sequence is compared by one or more algorithms until it is discarded or accepted, when the final score of the sequence is obtained. The acceptance criteria used after each filter are based on those of the HMMER 3.1 hmmsearch tool.

The initial SSV and MSV filters receive many more sequences than the final Viterbi filter. Therefore, our efforts to improve performance are focused on the initial filters. The SSV and MSV filters are implemented as kernels running on GPU, while the Viterbi algorithm executes on the host processor. The host also controls the path that a sequence shall take: after a sequence is compared by a filter, the host decides, based on the sequence partial score, if the sequence is discarded or forwarded to the next filter.

4.2 Parallelism Approach

Both MSV and SSV kernels were modeled in a way to exploit simultaneously task and data parallelism. We exploit task parallelism assigning distinct sequences to each CUDA block. Since the comparison of distinct sequences are independent from each other, several sequences can be compared simultaneously by executing several blocks concurrently on the GPU. On the other hand, data parallelism is exploited through the many threads of each block, which compute the cells in the same row of score matrix $M$ in parallel.

Figure 4 illustrates this approach, where an execution space is shown with a grid of $n$ blocks, each block with several threads. The distinct sequences $S_0, ..., S_{n-1}$ are assigned to the blocks, so each block performs the comparison of a sequence to the profile HMM (task parallelism). The threads of a block compute in parallel the cells in the same row of matrix $M$ corresponding to the sequence assigned to that block (data parallelism).

4.3 MSV Kernel with Optimized Reduction

The MSV kernel executes a sequential loop with $L$ iterations to compute matrix $M$ and other score vectors, where at iteration $i$, $Q$ threads compute row $i$ of $M$ in parallel, in a way that thread $j$ computes cell $M[i,j]$. After computing row $i$, a reduction operation is performed to obtain $E[i]$ as the maximum among the cells in this row. The reduction operation is performed once for each iteration of the sequential loop with $L$ iterations. By the end of all iterations, the score of the best alignment is obtained.
algorithm finds the maximum value in vector $E$, in order to obtain the score of the best alignment.

The SSV kernel of the GPU solution executes a sequential loop with $L$ iterations to compute matrix $M$, where at iteration $i$, $Q$ threads compute row $i$ of $M$ in parallel, in a way that thread $j$ computes cell $M[i,j]$. After computing row $i$, each thread $j$ compares $M[i,j]$ to the maximum value it accumulated in the previous loop iterations. Then, each thread $j$ obtains the maximum value of column $j$ of matrix $M$, in parallel to the other threads.

By the end of all iterations, one reduction operation is performed to obtain the score of the best alignment among the $Q$ maximum values accumulated. Thus, the SSV kernel produces the same result as Algorithm 3, however performing the operations in a different order than that indicated in the algorithm, so that we are able to better map the operations to the GPU hardware and, consequently, to exploit more data parallelism.

Another advantage of this modification is that the number of reduction operations performed in the SSV kernel is decreased by a factor of $L$, in comparison to the MSV kernel. Besides, the number of barrier synchronizations needed for the reduction operations is also decreased by a factor of $L$.

4.5 Optimizations

Initially we developed a basic GPU implementation for the sequence-profile comparison, with no optimizations applied. Then, we applied several optimizations to our basic GPU implementation. We evaluated the optimizations separately, in order to identify which ones provide performance gains, and then, we evaluated how they behave in combination with the others. Finally, an optimized GPU implementation was developed, containing the selected optimizations.

Our GPU solution is able to handle input cases with profile HMM model and/or sequence length longer than the maximum number of threads of the GPU used. In order to do that, we implemented both MSV and SSV kernels using the tiling technique [23], with each thread being associated to several cells, instead of only one.

Table 1 lists the main optimizations evaluated and which ones resulted in performance gains and were applied in the final optimized version of the SSV and MSV kernels of the GPU solution. Besides these optimizations, other established optimizations were also applied to both kernels, such as coalesced access to global memory, storing score structure in shared memory, keeping frequently used data in registers and reducing thread divergences.

5. Results and Discussion

The execution platform used in this work consists of a GPU NVIDIA GeForce GTX 570 connected to a host with Intel Core i7-3770S processor and 8GB RAM. We evaluated our solution using the entire UniProtKB/Swiss-Prot [1] database, composed of 540,732 sequences, with average and maximum length of 355.24 and 35,213, respectively. Ten sequence families were selected from the Pfam database [24] for our
experiments. Table 2 lists the selected families and the length of their corresponding profile HMMs.

Table 2

<table>
<thead>
<tr>
<th>Family</th>
<th>Profile HMM length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avian gp85</td>
<td>256</td>
</tr>
<tr>
<td>CABIT</td>
<td>256</td>
</tr>
<tr>
<td>DUF530</td>
<td>512</td>
</tr>
<tr>
<td>PaRep2b</td>
<td>512</td>
</tr>
<tr>
<td>Flu_PB2</td>
<td>759</td>
</tr>
<tr>
<td>Totivirus_coat</td>
<td>759</td>
</tr>
<tr>
<td>ACR_tran</td>
<td>1,021</td>
</tr>
<tr>
<td>RdRP_5</td>
<td>1,271</td>
</tr>
<tr>
<td>Bac_GDH</td>
<td>1,528</td>
</tr>
<tr>
<td>AvrE</td>
<td>1,774</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>864.80</strong></td>
</tr>
</tbody>
</table>

In our experiments, we measured the execution time necessary to compare all sequences in the UniProtKB/Swiss-Prot database to each selected family from Pfam database. The GPU solution average execution time shows an approximately one order of magnitude improvement with respect to HMMER 3.1 average execution time, when SSE2 instructions are enabled. If these vector instructions are not used, the improvement is approximately three orders of magnitude.

Table 3 shows the speedups achieved by the main optimizations listed in Table 1, the speedup they produced compared to the basic non-optimized GPU solution, for the kernels SSV and MSV. The empty fields correspond to profile HMMs with length longer than 1024. The tiling technique has not been applied to the basic non-optimized GPU solution yet, therefore this solution is not able to handle these families.

Since basically all optimizations resulted in performance gains for the SSV kernel, most of them were adopted on the final optimized GPU solution. On the other hand, for the MSV kernel, some optimizations did not result in performance gains. In most cases, the problem was related to the use of additional registers and consequently lower occupancy rates of the GPU.

Figure 5 shows the average execution time of the optimized GPU solution and HMMER 3.1 with the six different configurations, using logarithmic scale. These results represent the average running time for comparing all sequences in the UniProtKB/Swiss-Prot database to each selected family from Pfam database. The GPU solution average execution time shows an approximately one order of magnitude improvement with respect to HMMER 3.1 average execution time, when SSE2 instructions are enabled. If these vector instructions are not used, the improvement is approximately three orders of magnitude.

Table 4 shows the speedups achieved by the optimized GPU solution with respect to HMMER 3.1 with the six different configurations. The GPU solution achieved impressive speedups up to 1072.0 and 471.7 compared to the sequential single-core and quad-core configurations, respectively. Besides, the GPU solution also overcomes HMMER 3.1 with SSE2 instructions enabled, reaching speedups up to 28.8 and 8.8, for the single-core and quad-core configurations, respectively.
Table 4

<table>
<thead>
<tr>
<th>Family</th>
<th>Without SSE instructions</th>
<th>With SSE instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 core 2 cores 4 cores</td>
<td>1 core 2 cores 4 cores</td>
</tr>
<tr>
<td>Avian gp85</td>
<td>401.2 205.8 114.7</td>
<td>6.4 3.4 3.7</td>
</tr>
<tr>
<td>CABIT</td>
<td>405.7 208.1 114.9</td>
<td>6.9 3.7 3.4</td>
</tr>
<tr>
<td>DUF530</td>
<td>794.0 409.5 255.7</td>
<td>12.2 6.4 4.4</td>
</tr>
<tr>
<td>PaRep2b</td>
<td>799.6 413.0 267.5</td>
<td>12.7 6.7 4.5</td>
</tr>
<tr>
<td>Flu_PB2</td>
<td>683.0 355.5 289.5</td>
<td>11.6 6.1 3.7</td>
</tr>
<tr>
<td>Totivirus_coat</td>
<td>992.1 514.6 327.0</td>
<td>28.8 15.0 8.8</td>
</tr>
<tr>
<td>ACR_tran</td>
<td>1072.0 559.5 456.4</td>
<td>26.3 13.8 8.1</td>
</tr>
<tr>
<td>RdRP_5</td>
<td>750.5 397.3 364.5</td>
<td>12.4 6.5 3.6</td>
</tr>
<tr>
<td>Bac_GDH</td>
<td>934.9 495.0 471.7</td>
<td>18.7 9.7 5.6</td>
</tr>
<tr>
<td>AvrE</td>
<td>860.8 455.1 419.4</td>
<td>24.4 12.7 7.6</td>
</tr>
</tbody>
</table>

Another observation is the higher speedups obtained by the families ACR_tran and Bac_GDH. The explanation lies in the lengths of their profile HMMs, 1,021 and 1,528, respectively. The profile HMM length is directly associated with the amount of threads used in the GPU solution and the lengths of these families, combined with the tiling technique applied, provide better GPU occupancy rates.

We also report the performance using the throughput measure CUPS (Cell Updates per Second), which indicates how many cells of the dynamic programming matrices are computed in one second. Table 5 shows the average and maximum GCUPS (10^9 CUPS) achieved by the GPU solution and HMMPER3.1 (for the six configurations), for comparing the entire UniProtKB/Swiss-Prot database to all selected families.

Table 5

<table>
<thead>
<tr>
<th>GCUPS</th>
<th>HMMER 3.1 w/o SSE</th>
<th>HMMER 3.1 w/ SSE</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 core 2 cores 4 cores</td>
<td>1 core 2 cores 4 cores</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>0.38 0.73 1.02</td>
<td>20.35 38.75 57.47</td>
<td>286.12</td>
</tr>
<tr>
<td>Maximum</td>
<td>0.40 0.79 1.41</td>
<td>26.27 49.98 82.30</td>
<td>372.06</td>
</tr>
</tbody>
</table>

6. Conclusion

We developed a high-performance GPU solution to the sequence-profile comparison problem, and applied several optimizations such as memory optimizations, padding, loop unrolling, multiple streams and computation and transfers overlapping, vectorized access, and tiling.

We performed a comprehensive performance evaluation using a modest GPU and a large and representative biological data set. The GPU solution achieved impressive speedups up to 1072.0 and 471.7 compared to the HMMER 3.1 tool running on single-core and quad-core computers, respectively. Besides, the GPU solution also overcomes HMMPER 3.1 with SSE2 instructions enabled, reaching speedups up to 28.8 and 8.8, for the single-core and quad-core configurations, respectively.

The GPU solution produced the maximum of 372.06 GCUPS, while HMMPER 3.1 executing on a quad-core computer with SSE2 instructions enabled produced the maximum of 82.30 GCUPS. Using a more powerful GPU, our solution would achieve an even better performance, since less tiling would be necessary. We can also conclude that, by modeling a problem properly and applying optimizations targeting the platform, GPUs can reach high performances.

To the best of our knowledge, this is the first GPU-based system proposed for the acceleration of the sequence-profile comparison, which implements the SSV algorithm (introduced in HMMPER 3.1), besides the MSV algorithm. There are not in the literature solutions implementing the SSV algorithm in any parallel computing platform, apart from the HMMPER 3.1 tool suite.

References