Accelerating the Smith-Waterman Algorithm for Bio-sequence Matching on GPU

Qianghua Zhu, Fei Xia, and Guoqing Jin
Electronic Engineering College, Naval University of Engineering, Wuhan, P. R. China, 430033

Abstract—Nowadays, GPU has emerged as one promising computing platform to accelerate bio-sequence analysis applications by exploiting all kinds of parallel optimization strategies. In this paper, we take a well-known algorithm in the field of pair-wise sequence alignment and database searching, the Smith-Waterman (S-W) algorithm as an example, and demonstrate approaches that fully exploit its performance potentials on GPU platform. We propose the combination of coalesced global memory accesses, shared memory tiles, and loop unrolling. The optimization schemes release the computing power of hundreds of GPU cores completely, taking the performance increase from 0.54~0.73 GCUPS (10^9 Cell Units Per Second) of initial version to 28.23 GCUPS, over 50X of speedups.

2. Background

In the area of modern molecular biology and bioinformatics, the S-W algorithm is a well-known algorithm for performing pair-wise local sequence alignment. It has become the kernel algorithm in the process of bio-sequence matching, multiple sequence alignment, and database searching to discover similarities between sequences and to further explore the evolutionary history, critical preserved motifs, and even the details of the tertiary structure and important clues about protein functions [4].

The algorithm was first developed by Temple F. Smith and Michael S. Waterman in 1981 [5] to determine the optimal local alignment of two sequences. The bio-sequences (i.e., DNA, RNA or protein) serve as the input of the S-W algorithm, and the output is an alignments score representing the degree of similarity of the two input sequences. In the alignment process, a two-dimensional matrix $H$, as a temporary data structure, is used to store alignment scores of subsequences. Consider two sequences $S$ and $L$ of length $M$ and $N$. The two subsequences in sequences $S$ and $L$ are $S_1...S_i$ and $L_1...L_j$, respectively. The maximum similarity score of subsequence $S_1...S_i$ and $L_1...L_j$ is $F(i,j)$.

The gap-penalty scheme provides the option of gaps being introduced within the alignments. In our implementation, we consider an affine gap penalty scheme that consists of two types of penalties, the gap-open penalty $\alpha$ and the gap-extension penalty $\beta$. The computation of $H(i,j)$ for grid cell $(i,j)$ is given by the following recurrences:

\[
\begin{align*}
&\text{for } 1 \leq i \leq M, 1 \leq j \leq N \\
&H(i,0) = E(i,0) = \tilde{H}(0,j) = F(0,j) = 0; \\
&\text{for } 2 \leq i \leq M, 2 \leq j \leq N \\
&\begin{cases}
H(i,0) = \max \{ 0, E(i,j), F(i,j), \tilde{H}(i-1,j-1) \} + \text{sub}(S[i],L[j]); \\
E(i,j) = \max \{ H(i,j-1) - \alpha, E(i,j-1) - \beta \}; \\
F(i,j) = \max \{ H(i-1,j) - \alpha, E(i-1,j) - \beta \}.
\end{cases}
\end{align*}
\]

$\text{sub}$ is the character substitution cost table. We make some observations on the characteristics of the S-W algorithm. These observations suggest details of parallel
Observation 1: Inter-task parallelization
There is no data dependency in the multiple task of executing the alignment of several database sequences with a single query. Pair-wise alignment, called inter-task parallelization or coarse-grained parallelization, can be performed independently. The shared data are the query sequence and the substitution cost matrix. Multiple sequence alignment tasks can be distributed to GPU platforms to utilize computing resources efficiently. These implementations focus on database partitioning and load balance, instead of the parallelism of pair-wise sequence alignment.

Observation 2: Intra-task parallelization
Data parallelism also exists in pair-wise sequence alignment; it is called intra-task parallelization or fine-grained parallelization. Recurrence Formula 2 implied regular data dependency; that is, each cell $H(i, j)$ depends on its left neighbor $E(i, j-1)$, upper neighbor $F(i-1, j)$, $H(i-1, j)$, and upper left neighbor $H(i-1, j-1)$ in filling matrix $H$. If we fill the score matrix in a row column order in turn, the process must be executed serially. Moreover, there is strict data synchronization among adjacent anti-diagonals. However, there is no data dependency among the elements located in each anti-diagonal. Therefore, all cells along the anti-diagonal $k$ can be computed parallel from the anti-diagonals $k-2$, $k-1$, which can be arranged in a wave-front mode along the diagonal from up-left to down-right (Figure 1).

The computational and spatial complexity of pair-wise sequence alignment for two sequences of length $M$ and $N$ is $O(M \times N)$, and the computational complexity of the multi sequence alignment is $O(K \times N^2)$ for $K$ sequences with average length $N$. Bio-sequence databases are undergoing exponential growth; GenBank, for example, now stands with over 100 million sequences and 100 billion base pairs [6]. Hence, although comparing two sequences using the S-W algorithm is efficient in the classical sense, the execution time is still intolerable for pair-wise sequence alignment on the whole genome scale (more than $10^9$ bases). Due to differences in manufacture technology, hardware structure, computing resource, and clock frequency across all kinds of platforms, we use the standard measurement unit, GCUPS (10^9 Cell Updates Per Second), to measure actual computing power. The cell represents the workload for computing one element of the score matrix.

3. Optimizations on GPU
Modern GPUs are designed as programmable processors employing a large number of processor cores. It contains a processor array, which consists of a number of streaming multiprocessors (SMs) and hierarchical memory architecture for programmers to utilize. GPUs are especially well-suited to address problems that can be expressed as data-parallel computations in which the same program is executed on many data elements in parallel, by mapping data elements to parallel processing threads. Thus, to achieve reasonable parallel efficiency for GPU parallel computing, memory optimization schemes have to be adopted carefully to utilize fully the three layers of memory hierarchies: register, shared memory, and global memory [7].

Considering the computation searching of the optimal local alignment between a query sequence and a subject sequence as a task, there are two approaches to the parallel processing of sequence database searches using CUDA. The first approach, as shown in Figure 2(a), is intra-task parallelization, which assigns one task to a grid. In the approach, all threads cooperate to perform the task in parallel by calculating the alignment scores of cells within the same diagonals.

The second approach is inter-task parallelization, which assigns one task to exactly one thread. T1 thread takes task 1 to calculate the optimal alignment score between query sequence and database sequence 1, T2 thread takes task 2, and so on (Figure 2(b)). The method of intra-task parallelization, reported by Liu [8], occupies less device memory space, but suffers from frequent barrier synchronizations between GPU cores. In the following optimizations, we only consider the inter-task parallelization scheme that occupies more device memory because of the need to store the intermediate alignment results but achieves better performance than intra-task parallelization, as presented in [9, 10] by Manavski and Ligowski.
Based on inter-task parallelization, the following sections focus mainly on memory optimizations. The basic data structure of the S-W algorithm includes one two-dimension substitution cost table, one query sequence, and one sequence database consisting of a number of subject sequences. During the execution of the S-W algorithm, additional memory is required to store intermediate alignment results. To support a much larger database, the global memory is used to store the sequence database and the intermediate alignment results. The substitution cost table and query sequence are read-only, and are accessed by all threads in the grid. Therefore, we store the substitution cost table and query sequence into the texture memory and the constant memory, respectively.

### 3.1 Coalesced global memory accesses

![Fig. 3: Global memory layout and access pattern. (a) Naive implementation. (b) Interleaved implementation.](image)

Coalesced global memory accesses can always have significant improvements in performance over non-coalesced global memory accesses due to the effective usage of global memory bandwidth [7]. However, in the initial GPU version of S-W algorithm, none of the memory accesses are coalesced (Figure 3(a)). First, the database sequences loaded from a disk file are allocated a contiguous area of memory in the CPU side, where one sequence is represented by the data structure of a one-dimension character array, and the next sequence occupies the side contiguous area, as depicted in Figure 3(a) (DS 0, DS 1...DS n). Second, the database sequences area is copied directly to the GPU side global memory, where all symbols of the sequences are kept in order in the CPU side. All GPU threads begin to calculate the alignment scores from the first symbol, the second, and so on. The memory addresses generated from the same thread warp are non-contiguous. The column of non-optimization in Table 1 shows that no coalesced load or store operations occur, other than $181.12 \times 10^8$ and $161.69 \times 10^8$ times of scattered loads and stores respectively.

In the optimized version, we change the data layout of database sequences in the GPU global memory, creating coalesced memory accesses. Before the area of database sequences is copied to the GPU side global memory, we interleave the database sequences to a new area in CPU side memory. The symbols with the same index of all sequences are collected together. Thereafter, the new area is copied to the GPU side global memory (Figure 3(b)). The threads in the same warp access the global memory addresses in a contiguous way, resulting in the removal of all non-coalesced load/store. The rearranged data layout scheme helps significantly in achieving better performance. The GPU performance is increased from 0.54 to 5.79 GCUPS, over 10X of speedup.

<table>
<thead>
<tr>
<th>Memory Access</th>
<th>without opt.</th>
<th>with opt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>number $(10^8)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>coalesced store</td>
<td>161.69</td>
<td>0</td>
</tr>
<tr>
<td>coalesced load</td>
<td>0</td>
<td>11.32</td>
</tr>
<tr>
<td>performance (GCUPS)</td>
<td>GTX 280</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>GTX 470</td>
<td>0.54</td>
</tr>
</tbody>
</table>

### 3.2 Shared memory tiles to reduce global memory accesses

After coalescing the contiguous global memory accesses, we can estimate the total number of global memory accesses. Figure 4(a) shows the order in which one thread calculates the matrix, first in row, then in column order. This process, which involves scanning the matrix from left to right and then from top to bottom, requires $2 \times M \times N$ global memory transactions for the $M \times N$ matrix. Calculating $H(i,j)$ involves two global memory accesses, one for loading $H(i - 1, j)$, $F(i - 1, j)$, and the other for storing $H(i,j)$, $F(i,j)$.

We utilize tiles in the shared memory layer to reduce global memory accesses. The whole matrix computation is...
The increase in tile size decreases the number of global memory accesses to amortize the global latency cost, but the tile size is strictly limited by the size of the shared memory and the number of threads because each thread occupies a tile area in the shared memory. As shown in Table 2, for GPU GTX 280, the largest tile size only reaches 6, with over 60% of performance gain, and for GTX 470, only two elements in one tile brings nearly 2X speedups, and the largest tile size can be 10 with over 3X speedups. The decrease in the total number of global memory accesses with the increase in tile size supports above performance gains.

3.3 Register-level optimization with loop unrolling

In the version of tiled GPU S-W algorithm, an additional innermost loop has to be introduced to organize the tile calculation. This loop has a small body and constant iteration count. The tiling scheme reduces the number of global memory accesses at the expense of additional shared memory accesses, branch instructions, and address calculations. When the threads within a warp diverge via data-dependent conditional branches, the warp has to execute each branch path serially, causing severely performance bottlenecks [7].

The best performance can be achieved by unrolling the loop completely and removing all innermost loop branches, induction variable increments, and inner loop address calculation instructions. Experiment results show that the loop unrolling scheme greatly decreases branch instructions (Table 3). Finally, with the help of all optimization schemes, the most powerful performance on the GTX 470 platform reaches 28.23 GCUPS, over 50X speedups compared with the initial GPU version without any optimization schemes.

4. Result and Discussion

4.1 Environment and Test Methods

Our prototype system for performance evaluation consists of a host PC and a GPU card. The host is equipped with an Intel Q9400 Quad CPU, 2GB memory, and ASUS P5Q Dulex motherboard [P45 chipset running Windows XP SP3 with Visual Studio 2008 development environment (Visual C++ Compiler 15.00.30729.01)]. We use Redhat enterprise Linux 5.4 operating system with GCC 4.1.2 compiler for testing the Xeon CPU platform. We choose two commercial graphics cards, Geforce GTX280 and GTX470 with CUDA toolkit 3.1, as our GPU experimental platforms. The original S-W source code is derived from the kernel of the ClustalW toolkit 3.1, as our GPU experimental platforms. The original S-W source code is derived from the kernel of the ClustalW [11], the famous program for multi-sequence alignment application.

Table 4 shows the experimental results of the S-W algorithm for protein database searching application on CPU and GPU platforms. For each kind of platform, we list the performance measurements of two chips with different manufacturing technologies.

![Fig. 4: Memory layout and access pattern. (a) Without tiling. (b) With tiling.](image-url)
Table 4: Results on CPU and GPU platforms with different manufacturing technologies.

<table>
<thead>
<tr>
<th>Device Chip</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development Year</td>
<td>2007</td>
<td>2008</td>
</tr>
<tr>
<td>Chip Technology</td>
<td>65</td>
<td>45</td>
</tr>
<tr>
<td>Core Number</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>1600</td>
<td>2660</td>
</tr>
<tr>
<td>Peak Memory Bandwidth</td>
<td>3.2 GB/s</td>
<td>10.6 GB/s</td>
</tr>
<tr>
<td>Cache Capacity (KB)</td>
<td>1,024</td>
<td>6,400</td>
</tr>
<tr>
<td>GCUPS (average)</td>
<td>0.35</td>
<td>2.18</td>
</tr>
</tbody>
</table>

Table 5: Compare the average performance (GCUPS) with related works.

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Approach</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpern [12]</td>
<td>Intel Paragon i860</td>
<td>Single thread, SIMD</td>
</tr>
<tr>
<td>Wozniak [13]</td>
<td>Sun Ultra SPARC 167MHz</td>
<td>Single thread, SIMD</td>
</tr>
<tr>
<td>Rognes [14]</td>
<td>Intel Pentium III 500MHz</td>
<td>Single thread, SIMD</td>
</tr>
<tr>
<td>Jacob [15]</td>
<td>Intel Pentium 4 2.8GHz</td>
<td>Single thread, SIMD</td>
</tr>
<tr>
<td>Ours</td>
<td>Intel Q9400 Quad 2.66GHz</td>
<td>Multi-thread, SIMD</td>
</tr>
<tr>
<td>GPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manavski [9]</td>
<td>GeForce GTX 8800</td>
<td>Global memory optimization</td>
</tr>
<tr>
<td>Ours</td>
<td>GeForce GTX 470</td>
<td>Global, shared memory and register opt.</td>
</tr>
</tbody>
</table>

4.2 Compared to CPU Implementation

4.2.1 GCUPS Performance Comparison

We implement the S-W algorithm with the affine gap penalty model for protein database search application on CPU and GPU platforms with 65 nm and 45 nm manufacturing technology, respectively. For each computing platform, we test the average performance (GCUPS) of the S-W algorithm with different optimization grades. As shown in Figure 5, the horizontal axis is the average performance represented by GCUPS, and the bar with different colors and letters represent different optimization grades. The right part with black oblique lines in each bar is the performance improvement on the new generation 45 nm manufacture technology as compared to 65 nm computing platforms using the same optimization scheme.

The naive S-W version running on the state-of-the-art multi-core CPU platform is only 0.03 GCUPS. The performance is improved steadily by adopting different optimization strategies, including compiler auto options, single-thread SIMD, and multi-thread SIMD. The performance reaches 2.18 GCUPS, over 70X speedups, using multi-thread SIMD on Intel Q9400 Quad CPU. On the GPU platform, the performance of the naive version on Geforce GTX 470 without any optimization is just 0.73 GCUPS, lower than that of the multi-thread SIMD implementation on the Q9400 Quad CPU. However, the final performance is increased by nearly 40 times, reaching 28.23 GCUPS, on condition of the highest optimization effort.

4.2.2 Misunderstandings on GPU and CPU comparison

We find three misunderstandings on performance comparison between GPU and CPU implementations.

(1) Optimized version GPU vs. naive version CPU.

Form Figure 5, we observe that the GPU implementation with register optimization shows a factor of more than 900X speedup over the naive version running on CPU. However, compared to the optimized version with multi-thread SIMD with loop unrolling, the speedup factor is only by 12X.

(2) New manufacturing technology GPU vs. old manufacturing technology CPU.

If we compare the performance of the S-W algorithm tested on GPUs with 45 nm manufacture technology to that on 65 nm dual-core CPU, a factor of more than 80X speedup can be achieved. However, the performance on the GPU platform is only improved by 12X compared to the CPU with the same 45 nm technology.
(3) Optimized version on new GPU platform vs. naive version on old CPU platform.

The most unfair comparison is that of the performance of the optimized version on the new generation GPU platform to the naive version running on the old CPU. If we adopt this measure approach, the GPU implementation would show a speedup factor of more than 1000X over the origin CPU version without optimization. But this is beyond scientific evaluation.

Conclusively, the performance of the GPU is superior to the CPU version. Instead of hundreds of times speedup, GPU shows a speedup factor of 12X over the CPU when both are running the optimized version under the same manufacture technology.

5. Comparison with Related works

There are a number of efficient implementations of the S-W algorithm on GPU platforms, as listed in Table 5. Most work on GPU acceleration discussed intra-task parallelization or global memory optimization schemes separately, and none compared GPU with a fully optimized CPU version. Our work combines three levels of optimizations and reports fair comparison results. Recently, there have been several papers evaluating the performance of CPU and GPU computing platforms. Both [1] from Intel Lee and [2] form IBM reported performance comparisons between carefully tuned CPU version with optimized GPU version collected from published papers. However, the comparisons were taken from 45 nm CPUs to 65/55 nm GPUs.

6. Conclusion

This paper explored the parallel schemes on GPU platforms to accelerate the S-W algorithm for pair-wise sequence alignment. We tried various optimization schemes, including coalesced global memory accesses, shared memory tiles, and loop unfolding, and obtained over 50X speedups. The experimental results show that GPU is obviously superior to CPU. However, the performance difference does not reach 100X, only 12X on the condition of fair Comparative Study.

Acknowledgments.

We would like to thank the anonymous reviewers for their detailed revising directions and constructive comments. This work is partially sponsored by the NSFC (Hardware Acceleration and Parallelism Research for Complex Biological Sequence Analysis on Heterogeneous Computing Platform).

References


