Viability of the Parallel Prefix Algorithm for Sequence Alignment on Massively Parallel GPUs

A. Christian Dicker[1], **B. John Sibandze**[1], **C. Samuel Kelly**[1], **and D. Jens Mache**[1]
[1]Computer Science, Lewis & Clark College, Portland, Oregon, USA

Abstract— *In this paper, we compare two different methods for parallelizing the Needleman–Wunsch dynamic programming algorithm for finding the optimal alignment of two sequences: (1) computing the elements of each diagonal of the table in parallel and (2) computing the elements of each row in parallel using the parallel prefix algorithm. In 2003, the latter algorithm was shown to decrease communication between processors and provide a more uniform work distribution [1]. With the increasing prevalence of general purpose programming on graphics processing units (GPUs), there is a need to reassess the viability of the parallel prefix-based algorithm. We discuss our implementation of both algorithms on a massively parallel GPU and compare the runtimes by thread count as well as by sequence size. We find that the parallel diagonal algorithm runs faster for large sequence lengths.*

Keywords: Bioinformatics, GPU, CUDA

1. Background

1.1 The Needleman–Wunsch Algorithm

The Needleman–Wunsch algorithm uses a dynamic programming table to find an optimal alignment of two sequences (which might represent DNA sequences, English words, etc.), where an alignment is found by inserting any number of gaps into either sequence so that the lengths end up the same. The score of an alignment is found by considering the pair of symbols in each column. If the symbols match (and are not both gaps), that column receives a score of c_1; if they don't match (and neither is a gap), it receives a score of c_2; and if either of the symbols is a gap, the column receives a score of c_3. The score of the alignment is the sum of the scores of all of the columns. The optimal alignment is the one with the highest score [3]. Following related work, we use $c_1 = 1$, $c_2 = 0$, and $c_3 = -1$ in our examples and experiments.

If the sequences are $a_1a_2 \ldots a_n$ and $b_1b_2 \ldots b_m$, and our table is T, then $T[i][j]$ (for $i = 0, 1, \ldots, n$ and $j = 0, 1, \ldots, m$) is the score of an optimal alignment of the substrings $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$. (If $i = 0$ or $j = 0$, then the corresponding string is the empty string.) Using this scheme, $T[i][j]$ is the maximum of $T[i-1][j]$, $T[i-1][j-1]$, and $T[i][j-1]$, each plus the cost of moving to the current cell. That is,

$$T[i][j] = \max \begin{cases} T[i-1][j] - 1 \\ T[i-1][j-1] + f(a_i, b_j) \\ T[i][j-1] - 1, \end{cases}$$

where

$$f(a,b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{if } a \neq b. \end{cases}$$

1.2 The Parallel Diagonal Algorithm

Notice that each entry in T depends only on entries in the previous two diagonals of the table—and not on any of the entries in the same diagonal. Therefore all of the entries in a diagonal can be computed in parallel.

This observation naturally leads to the parallel-diagonal method of parallelizing the Needleman–Wunsch algorithm, where the diagonals of T are computed in sequence, with each element potentially computed by a different processor [2].

1.3 The Parallel Prefix-Based Algorithm

Given an sequence C of n values and a binary associative operation $\oplus$, the prefixes S of C are given by

$$S[i] = C[1] \oplus C[2] \oplus \cdots \oplus C[i],$$

for $1 \leq i \leq n$. In the case of the Needleman–Wunsch algorithm, we use as our binary operation the maximum function, so that the ith prefix maximum is the maximum of the first i elements of the original sequence.

The sequence of prefixes S can be computed in logarithmic time with multiple processors.

This algorithm was used as the basis of a different method of parallelizing the Needleman–Wunsch algorithm by Aluru *et al.* [1] In this method, the table is computed row-by-row. Since the entries in each row depend on entries in the previous row and the same row, it is done in multiple steps.

First, a partial solution is obtained by assigning to each entry in the row the maximum of the north and northwest entries, each plus the cost of moving to the current cell. Second, the column number is added to each entry to facilitate the computation of the parallel prefix maxima. Finally, the parallel prefix maxima are computed, and the column numbers are subtracted again from each entry, yielding the final values. More detail is given in [1].

The authors showed that this algorithm was time-optimal like the parallel diagonal algorithm while decreasing the amount of communication required between processors.

The parallel prefix algorithm allows p processors to find the prefixes of an array of n numbers in $\mathcal{O}(\frac{n}{p} + \log p)$ time, while also distributing work among the processors uniformly. [1] Therefore the time of filling out the entire table is $\mathcal{O}(\frac{n^2}{p} + n \log p)$.

1.4 Graphics Processing Units

Graphics processing units (GPUs) are accelerators that use data parallel computation to perform hundreds or thousands of operations in parallel. Due to their low power consumption and relatively low cost, they are increasing in prevalence, including being an integral part of many of the fastest supercomputers in the world.

It is for this reason that it is important to re-examine parallel algorithms invented without GPUs in mind within this new paradigm. CUDA (Compute Unified Device Architecture) is a language developed by NVIDIA that allows programmers to use NVIDIA GPUs for general purpose programming. Below we describe our attempt to re-evaluate the benefits of the parallel-prefix algorithm on an NVIDIA GPU.

2. Implementation on GPU

We implemented the two algorithms for an NVIDIA GPU in order to compare their runtimes. We will give pseudocode for each implementation below.

We looked at two implementations of each algorithm: using a single block and multiple blocks. On a single block, we only have access to a single streaming multiprocessor which contains 48 CUDA cores. When we utilize all blocks, we have access to the full 336 CUDA cores on our NVIDIA GTX 460. However, there is some added overhead due to the fact that we have to leave a kernel to synchronize between blocks. Forty-eight CUDA cores already surpasses the maximum number of cores on which Aluru was able to test the parallel prefix algorithm [1].

2.1 Parallel Diagonal

Here we start off with $A = [0]$ and $B = [-1, -1]$, the first two diagonals of T. These are copied to the GPU, along with the two sequences we are comparing (S_1 and S_2), and then the following is executed on the GPU in many different threads, each with a unique thread index (t in the code below).

Note that by using global memory on CUDA, we can execute threads in several blocks, which enables us to use more parallelism.

Algorithm 1 Pseudocode for parallel diagonal GPU kernel

Require: n, the length of the sequences
Require: t, the thread index numbered from 0 to n
Require: $diag\#$, the diagonal number from 0 to the length of the diagonal
Require: C, the diagonal to be computed
Require: B, the previous diagonal
Require: A, the diagonal prior to B Every diagonal $D \in \{A, B, C\}$ is constructed so that it contains every element of the table for which $row + column = diag\#$ and with the element in column $column$ accessible at $D[column]$.
if $column = 0$ or $row = 0$ **then**
  $C[t] \leftarrow -diag\#$
else if $i < diaglen$ **then**

$$C[t] \leftarrow \max \begin{cases} B[t] - 1 \\ B[t-1] - 1 \\ A[t-1] + f(S_1[column - 1], S_2[row - 1]) \end{cases}$$

end if
$A \leftarrow B$
$B \leftarrow C$

2.2 Parallel Prefix

We begin with the first row being filled with values 0 to $-n$ where n is the length of the sequence. This row is copied to the GPU, along with the two sequences we are comparing (S_1 and S_2), and then the following is executed on the GPU in many different threads, each with a unique thread index (t in the code below).

3. Experiments

We conducted two main experiments on our NVIDIA GTX 460. The first one compares the diagonal and parallel prefix algorithms on one block with threads equal to 2^i where $0 \leq i \leq 10$ and sequence sizes of 2^j where $8 \leq j \leq 16$. The second experiment uses multiple blocks. For testing the diagonal, we ensure that there is a thread for every element. For testing the parallel prefix, we test a number of blocks equal to 2^k where $0 \leq k \leq 12$, threads per block varying as before, and the sequence size varying as before. The purpose of varying the number of blocks and threads per block is to ensure that we are achieving the optimal configuration so that we may, later, compare the algorithms at their best.

4. Results

We compared the fastest time for each sequence size. With one block, parallel prefix is faster for sequences up to and including 4096 characters. For multiple blocks, the diagonal algorithm is faster for all sequence lengths. Overall, comparing one block to multiple blocks, the single block parallel prefix is fastest for sequences up to and including

4096 characters. From sequence sizes of 8192 and up, the diagonal algorithm on multiple blocks is fastest.

In Figure 2, you can see the speedup of the parallel prefix and parallel diagonal algorithms for one block. You will notice that the speedup does not always increase as the number of threads increases. We tested multiple configurations to find the ideal number of threads. Since the runtime increases with the number of processors, this is not necessarily equal to the largest number of simultaneously running threads on the GPU.

5. Discussion

According to our results, computing elements of a row in parallel using the parallel prefix algorithm can be faster than computing elements of a diagonal in parallel for small enough sequences.

It may seem strange that the parallel prefix algorithm would be faster on one block, when it uses fewer processors, than it is on multiple blocks. This is due to added overhead that occurs when synchronizing threads between multiple blocks, which requires leaving the kernel completely. It makes sense that for larger sequences, this added overhead becomes negligible, and in fact, for sequences of 16384 and larger, the algorithm that uses multiple blocks is faster.

It may also seem strange that the parallel prefix algorithm on a single block is faster than the diagonal algorithm on multiple blocks. This is, most likely, due to some inherent overhead from the diagonal algorithm and once again, becomes negligible at longer sequence lengths. In this case, as mentioned in our results, this sequence length is 8192.

6. Conclusions & Future Work

In this GPU age, we re-examined two different methods for parallelizing the Needleman–Wunsch algorithm for finding the optimal alignment of two sequences: parallel diagonal and parallel-prefix. Our main result is that the parallel prefix-based algorithm on an NVIDIA GTX 460, running on a single block is fastest for short sequences up to and including 4096 characters. Beyond that, the diagonal algorithm on multiple blocks is fastest.

For future work, we would like to rerun our experiments on a better GPU. We have access to an NVIDIA Tesla K20, that we hope to utilize. Also, all of our experiments were done comparing two strings of the same length, which in terms of work distribution, is the worst case for the diagonal algorithm. Therefore, experiments should be done with sequences of very different lengths.

7. Acknowledgments

This research was supported by NSF grant DUE 1044932, by the James F. and Marion L. Miller Foundation and by the John S. Rogers Science Research Program. We would like to thank Kyle Barton, Samuel Dodson, Danielle Fenske, Adam Smith, Ben Whitehead.

References

[1] Srinivas Aluru, Natsuhiko Futamura, and Kishan Mehrotra. Parallel biological sequence comparison using prefix computations. *J. Parallel Distrib. Comput.*, 63(3):264–272, 2003.

[2] David Kirk and Wen mei Hwu. Programming massively parallel processors. Elsevier, 2013.

[3] Saul Needleman and Christian Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mal. Biol.*, 48:443–453, 1970.

Algorithm 2 Pseudocode for parallel prefix GPU kernels

Require: S_1, S_2, the sequences we are comparing
Require: n, the length of the sequences
Require: p, the total number of threads (analogous to processors)
Require: t, the thread index numbered from 0 to $p-1$
Require: B, the row we are computing
Require: A, the previous row of the table
Require: r, the row number of B
Require: X, the array of partial solutions for B
Require: Y, the array that the parallel prefix max is being computed on
Require: $Maxima$,

$X[\frac{nt}{p}] \leftarrow \frac{nt}{p} + \max \begin{cases} A[\frac{nt}{p}] - 1 \\ A[\frac{nt}{p} - 1] + f(S_1[\frac{nt}{p} - 1], S_2[r-1]) \end{cases}$
for $i \leftarrow \frac{nt}{p} + 1, \frac{n(t+1)}{p} - 1$ **do**
  $X[i] \leftarrow i + \max \begin{cases} A[i] - 1 \\ A[i-1] + f(S_1[i-1], S_2[r-1]) \end{cases}$
  $X[i] \leftarrow \max(X[i], X[i-1])$
end for
$Y[t] \leftarrow X[\frac{n(t+1)}{p} - 1]$
$Maxima[t] \leftarrow X[\frac{n(t+1)}{p} - 1]$
SYNCHRONIZE
for $i \leftarrow 0, \lceil \log_2 p \rceil - 1$ **do**
  $partner\_index \leftarrow (t-1)$ XOR 2^i
  $new\_max[t] \leftarrow \max(Maxima[t], Maxima[partner\_inc$
  if $t > partner\_index$ **then**
    $Y[t] \leftarrow \max(Y[t], Maxima[partner\_index])$
  end if
  SYNCHRONIZE
  $Maxima[t] \leftarrow new\_max[t]$
  SYNCHRONIZE
end for
$tmp \leftarrow Y[t]$
for $i \leftarrow \frac{nt}{p} + 1, \frac{n(t+1)}{p} - 1$ **do**
  if $X[i] < tmp$ **then**
    $X[i] \leftarrow tmp$
  end if
  $B[col] = x[col] - col$
end for

Fig. 1: Runtime for Different Algorithms

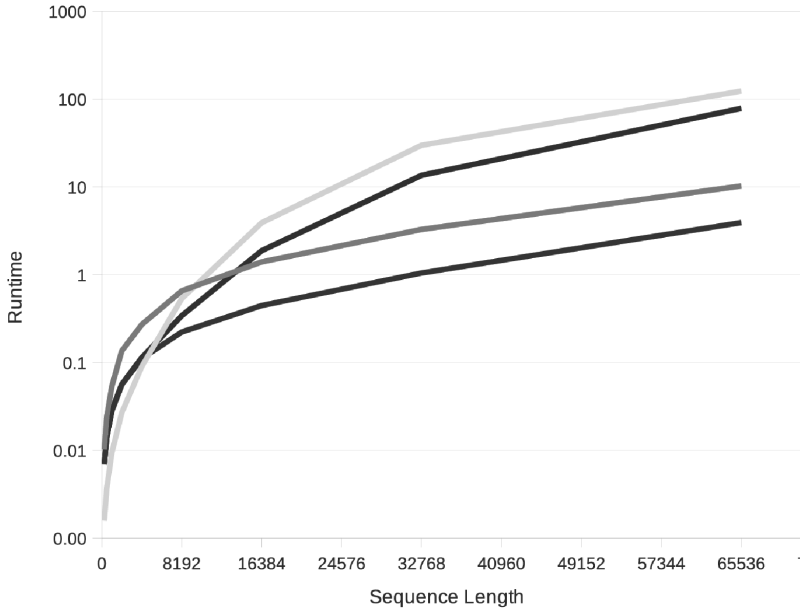

Fig. 2: Speedup for One Block