Parallel Sequence Alignment Algorithm

Jiayin Ling
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, USA
jiayinl@andrew.cmu.edu

Ruolin Zhang
Information Networking Institute
Carnegie Mellon University
Pittsburgh, USA
ruolinz@andrew.cmu.edu

I. Introduction

In this project, we implemented parallelized versions of a pairwise sequence global alignment algorithm that originated from the Needleman-Wunsch algorithm. We leveraged CUDA and OpenMP to build our two implementations and compared the performance between them. In our performance analysis, we focused on scalability of our implementations and did experiments with varied lengths of the input sequences and the number of resources provided. Finally, we include some of the directions we come up with that could be in future works.

II. Background

The Needleman-Wunsch algorithm is used in bioinformatics to align protein sequences; it is widely used for optimal global alignment of DNA sequences. Global alignment means finding a matching sequence from a potentially very long sequence. Since the algorithm scales proportionally to the product of the length of two sequences in a sequential implementation, it can get very slow for large input sequences which is a common cause for input to this algorithm. There have been efforts to speed up the algorithm sequentially, but for large inputs, it would be wise to come up with a parallel algorithm that speeds up the operation. The motivation is further promoted by the fact the algorithm involves simple computations repeated a large number of times, which can be a good candidate for CUDA. Given two sequences, the Needleman-Wunsch algorithm first constructs a scoring matrix("matrix") based on the two input sequences, then fills in the matrix with scores, and eventually traces the matrix from the bottom right to the top left to find alignment candidates.

As shown in Fig. 1, the algorithm would populate a matrix of $(M + 1) \times (N + 1)$ for a sequence A of size M and a sequence B of size N. The cells in the first row and the first column are initialized to the negative value of the corresponding index in the row/column. The computation starts at (1,1), and the score at cell (i, j) (ith row, jth column) represents how well sequence A[0:j] and sequence B[0, i] are aligned. A cell's score would be the maximum among the following:

Matrix[i-1,j-1] + MATCH_SCORE / MISMATCH_SCORE
Matrix[i,j-1] + GAP_SCORE

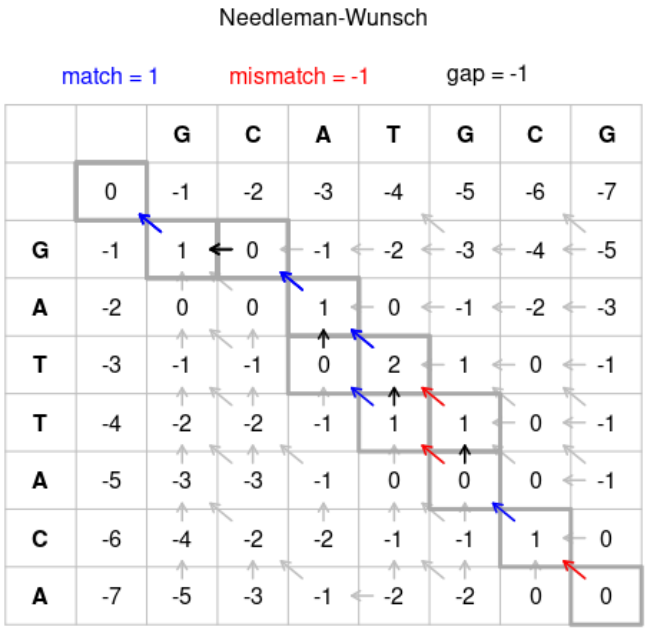

Fig. 1. Score matrix generated by the pairwise sequence alignment.Source

Matrix[i,j] + GAP_SCORE

The MATCH_SCORE, MISMATCH_SCORE, and GAP_SCORE are self-defined. If we use a positive number for MATCH_SCORE and a negative number for the others, the lower of alignment score the large the discrepancy. The tracing in Figure 1 shows one possible candidate could be:
Sequence A: GCAT-GCG
Sequence B: G-ATTACA

III. Parallel Algorithm

We are interested in parallelizing the matrix computation of the algorithm. As illustrated above, the algorithm has a fundamental dependency on nearby data. Specifically, to compute the value of every cell, we need to look at cells to the left, top left and top which could be written by other processes in a parallel algorithm. It's not ideal to seek parallelism by parallelizing on one score calculation because it requires additional space to store this score and limits the maximum

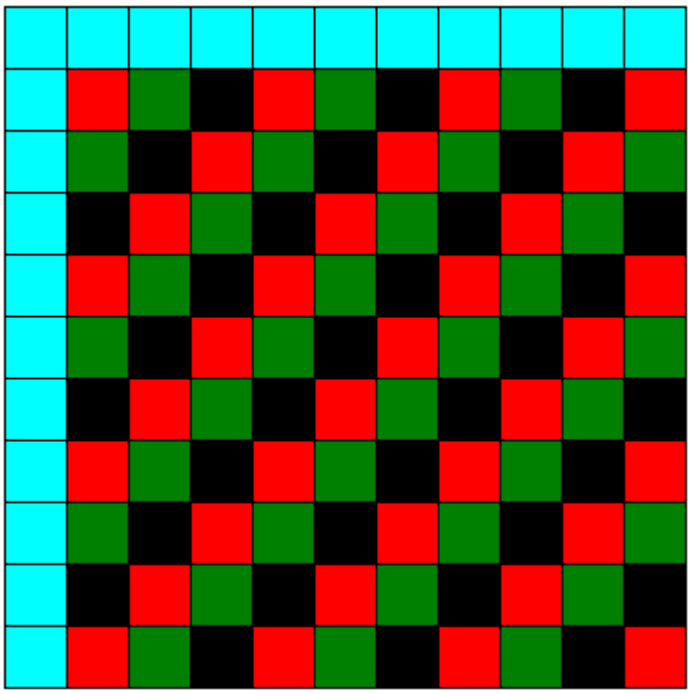

Fig. 2. Illustration of simutaneously computed cells

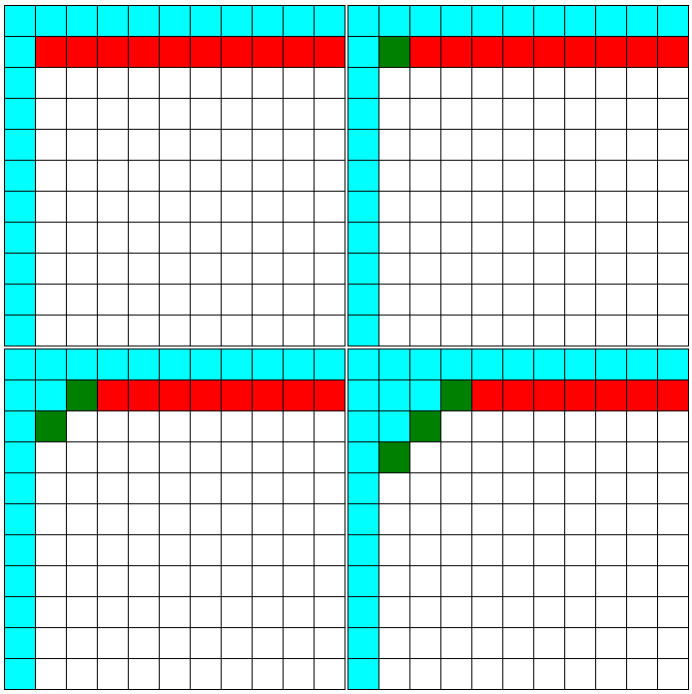

Fig. 3. CUDA kernel worker illustration

experiment problem size. Meanwhile, we only read the input sequences so there will not be any conflict. Based on the nature of data access, we can parallelize all cell computation in the second diagonal direction of the matrix. That is, for any i, j pairs, all cells in the line connecting cell (i,j) and cell(j,i). As shown in Fig. 3, the blue cells are pre-filled cells from initialization corresponding to a gap at these locations. The red, green and black cells in the same straights lines in the second-diagonal direction will be computed in parallel.

A. CUDA

In the CUDA implementation, we noted that the maximum number of cells that will be computed simultaneously will be the smaller of the two input sequences. Thus for input sizes smaller than the maximum number of threads supported by CUDA, we would only need $N = min(N1, N2)$ threads where N1 and N2 are the input sequence lengths respectively. The mapping of worker threads onto different parts of the matrix presented some challenges to us. After carefully examining the matrix, we observed that for any matrix of size $N1 \times N2$ where both N1 and N2 are smaller than the number of threads available, we can directly map the N threads to the smaller row/column of the input matrix. Thus we decided to always put the shorter sequence in the horizontal direction and the longer one in the vertical direction so $N2 \leq N1$. Now observe that for any input sequence pairs of length N1 and N2, we need $N1 + N2 - 1$ iterations to fully compute the matrix and since the size of the matrix is $N1 \times N2$ and we have N2 working threads, which means every thread will worker for exactly N1 iterations. Based on this it is not hard to see that the leftmost working thread will start by iteration 0, second one on the left will start by iteration 1 and so on. Fig. 2 shows the matrix for an input size of $10/times10$ and the blue blocks are pre-filled during initialization, all 10 workers are assigned to the horizontal blocks, then worker 0 starts to execute and moves down by one cell, then in the second iteration worker 1 joins and so on.

The issue with this implementation is that we must synchronize all the threads after each iteration to get rid of potential race conditions. For input sizes larger than the number of threads available for one block in CUDA, we cannot create more workers by increasing the grid size as we have no way of synchronization between threads from different blocks. A possible solution is to use cooperative groups from CUDA 9 where it allows synchronization between all threads from the same cooperative group that can spam multiple blocks. However we were unsuccessful in taking this approach due to the complexity of changes that needed to be made. Additionally, for multiple GPUs, there is a possibility for a cooperative group consisting of multiple GPUs. However, this requires almost completely rewriting the code to accommodate for the multi-device memory model and calling conventions which is very different from the single device case, which we do not have time for. Instead, we took the kernel and modified it so that it accepts two additional arguments where one is the current pass of computation and the other is the number of horizontal cells to compute for this pass. This allows us to use multiple kernel calls where for each pass we take care of a maximum number of 1024 columns of the matrix.

B. OPENMP

In the OpenMP implementation, the approach was similar to the CUDA implementation. Using OpenMP would also rarely achieve any parallelism if we iterate over columns or rows due to the dependency requirement of this algorithm. We instead iterate over anti-diagonals of the matrix and split the work for one diagonal to available resources(see Fig. 2). The computation of each cell in a diagonal is completely independent of others and would only be dependent on the previous diagonal. Therefore, diagonals are computed from left to right, and one diagonal should be computed only if all diagonals on its left are finished. For input sequences of size N1 and N2, this implementation would run for N1 + N2 - 1 iteration in order to compute all anti-diagonals.

```
#pragma omp parallel
    for(int i = 1; i < n1 + n2 - 1; i++) {
        #pragma omp for
        for(int j = max(1, i-n1+2);
            j < min(n2, i+1);
            j++) {
```

The above parallel region setup would synchronize at the end of each iteration of the outer loop, which matches the expected behaviors of computing one diagonal at a time. The scheduling policy used is static because the computation for each cell is complete identical. Equally distributing the iterations would less likely to cause workload imbalance in this algorithm.

IV. Result

We generated all test input sequences using a python script. The script helped us generate two random sequences of the requested lengths. Both implementations tested on the PSC machines using non-shared node.

A. machines

TABLE I
HARDWARE ON PSC

Hardware	Model	Implementation
GPU	Nvidia A100	CUDA
CPU	AMD EPYC 7742	OpenMP

B. CUDA

For the CUDA implementation, since we only have a kernel with up to 1024 threads, the total number of iterations can be calculated as $roof(N2/1024) \times (N1 + 1023) > N1N2/1024 + N2$ where $N2 > 1024$. This means when the size of the smaller matrix is larger than 1024, the scaling of the algorithm starts to deviate from that when the matrix size is smaller than 1024. When N1 and N2 are large, the number of iterations scales like N1N2, which is the same as a linear algorithm. This is a major issue with the cuda implementation with multiple kernel calls. It might be worthwhile to work on trying to synchronize more than 1024 threads with cooperative groups as stated above. The drawback is that the cooperative

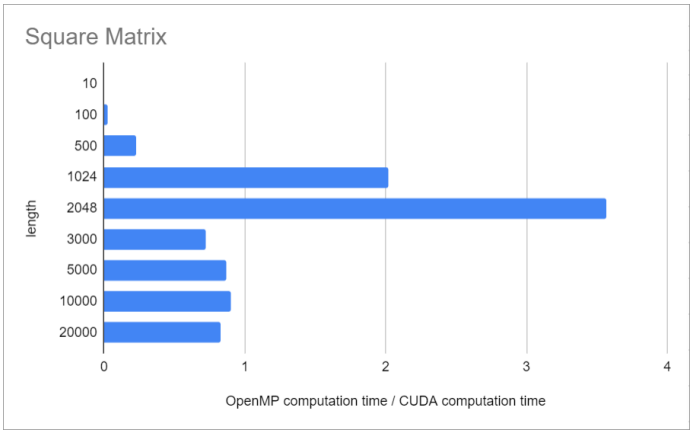

Fig. 4. CUDA speed vs 1 thread openMP for input sequences of same lengths

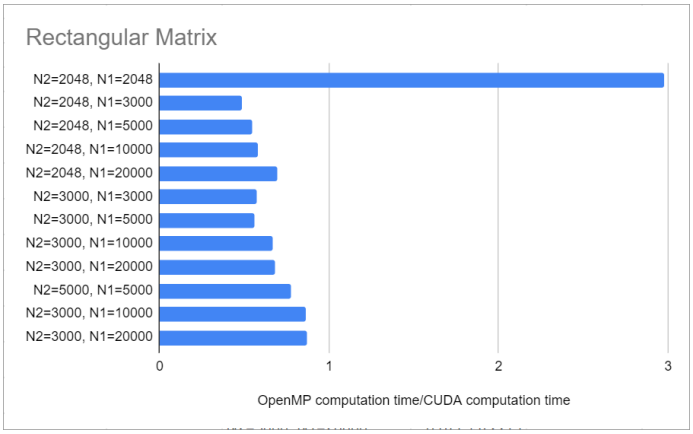

Fig. 5. CUDA speed vs 1 thread openMP for input sequences of different lengths

group is a software synchronization method, which means its overhead could be significant, and since we need to synchronize after every cycle, it is not sure if the speedup will be obvious under all circumstances. Except for this major performance obstacle, the data dependencies involved with the algorithm itself demand that we cannot make all threads work simultaneously and threads that need data from the results of other threads will not be able to start until those threads finish the operation. This causes the speedup even for small input sizes to be not proportional to the number of threads. Lastly, because of the need to synchronize on every iteration, the sum of overhead could be significant for a large number of iterations.

C. OPENMP

Although the diagonal approach brought parallelism to this highly dependent algorithm, the drawback was significant. The workload of each iteration was different because the number of cells in each diagonal was different. Especially in the beginning and at the end of the computation, there would be a lack of parallelism and resources could be idle. For example, with two long enough sequences and 128 threads

available to use, the first 128 and the last 128 diagonals would simply contain less than 128 elements and not be able to fully utilize all resources. Therefore, if the input size was small, it may achieve small/no speedup. While the computation of one cell was trivial with 3 loads and 1 store for each cell, synchronization was required for the computation of each diagonal. Hence, We expected that we needed a much larger input size so that the parallel computation benefit from increasing the thread counts would outweigh the run-time overhead.

In our experiments, we varied the input sizes and thread counts to verify the above expectation. The input sequences of equal size would populate square matrix; the input sequences of different sizes would populate rectangular matrix.

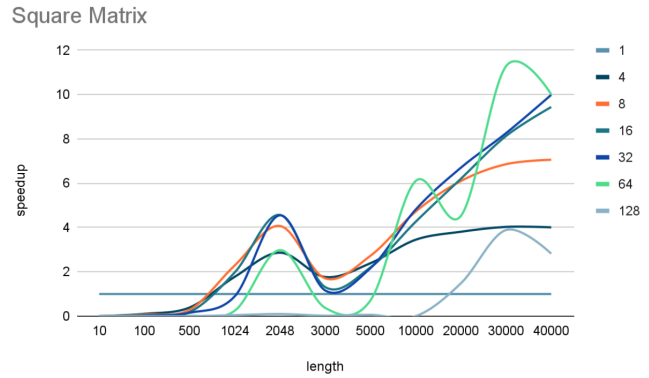

Fig. 6. OpenMP multi-thread speedup for input sequences of same length

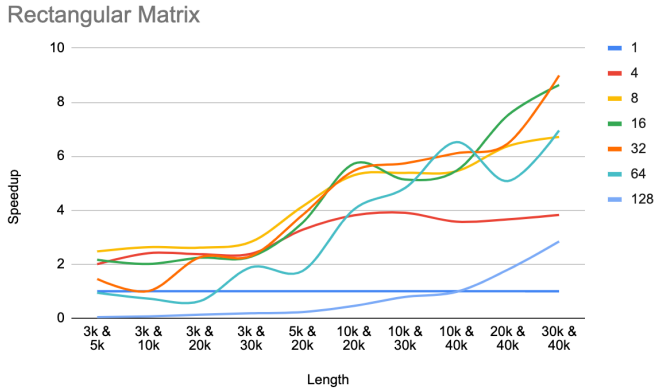

Fig. 7. OpenMP multi-thread speedup for input sequences of different lengths

In Fig. 6 sequences with $length < 1024$ received almost no speedup for all thread counts, which was expected. We could observe an increasing trend of speedup for all thread counts with increasing input sizes. For 128 threads, it only received positive speedup for $length > 20k$. To receive higher speedup, we speculated that we should further increase the input size to increase the amount of computation each thread could receive. However, due to the memory limitation of the PSC machines, we could not continue experimenting with larger problem sizes.

For both types of input, the speedup was highly associated with the input size. For input sequences that could provide large diagonal, the shape of the matrix had no significant impact on the level of parallelism.

V. Work Distribution

Jiayin is responsible for implementing, verifying and testing of CUDA verision of the program and ruolin worked on those of the OpenMP verision. Then we combined the work and wrote the report together. So the work distribution is 50-50.

References

[1] Needleman–Wunsch algorithm. (2022, April 7). In Wikipedia. https://en.wikipedia.org/wiki/Needleman-Wunsch_algorithm

[2] Biopython. Biopython/pairwise2.py (2022, May, 1) In GitHub. https://github.com/biopython/biopython/blob/master/Bio/pairwise2.py