Abstract

This study presents an optimized DNA sequence alignment algorithm focusing on the traceback phase of the Smith-Waterman algorithm. We introduce a novel design spliting a score matrix into smaller score grids to efficiently manage memory resources. Based on experiments conducted with real-world sequencing data, our design exhibited approximately an 80-fold performance improvement compared to the baseline. Our design offers a scalable and memory-efficient solution to seed extension kernel.

Keywords: Optimization in Traceback, GPU Acceleration, DNA Sequence Alignment

Table of Contents

Chapter 1

Introduction

DNA sequence alignment is a crucial computational technique in bioinformatics that aims to identify similarities and differences among DNA sequences, revealing regions of similarity, dissimilarity, and potential evolutionary relationships. The alignment process involves comparing individual bases in a sequence using dynamic programming (DP). Utilizing graphics processing units (GPUs) for DNA sequence alignment brings substantial advantages due to their inherent parallel processing capabilities, enabling simultaneous execution of multiple calculations. This parallelism significantly expedites the alignment process compared to traditional central processing units (CPUs).

Dealing with large datasets, especially with the rise of high-throughput sequencing technologies, is a common challenge in DNA sequence alignment. GPUs efficiently handle parallel processing of vast amounts of data, ensuring high-throughput capabilities essential for timely analysis of large genomic datasets.

However, a significant challenge arises during the exhaustive calculation of the score matrix and the subsequent traceback phase. The conventional approach necessitates computing scores for all matrix positions, resulting in a substantial demand for storage. The sheer volume of score matrices poses challenges for efficient shared memory utilization, limiting the optimization of memory resources and highlighting a crucial hurdle in the current methodology.

To address this challenge, we introduce a novel approach. Instead of storing the entire score matrix, we partition it into smaller units known as score grids. Only boundary scores

of score grids are stored, significantly reducing the data storage requirements. During the traceback phase, we implement an efficient strategy by dynamically calculating internal values of the score grids along the traceback route, minimizing overall storage demands and reducing the number of computations required. The analysis of three parameters, batchSize, scoreGridSize, and threadBlockSize, revealed that increasing batchSize significantly enhances performance. Additionally, an examination of the relationships between these parameters and the sizes of global memory and shared memory provided insights into effective methods for increasing batchSize. We implemented a baseline that stores all the necessary direction information for traceback, and compared its performance with our kernel. The results showed a substantial performance improvement, confirming the effective utilization of shared memory. This leads to more efficient and scalable DNA sequence alignment processes.

Chapter 2

Background

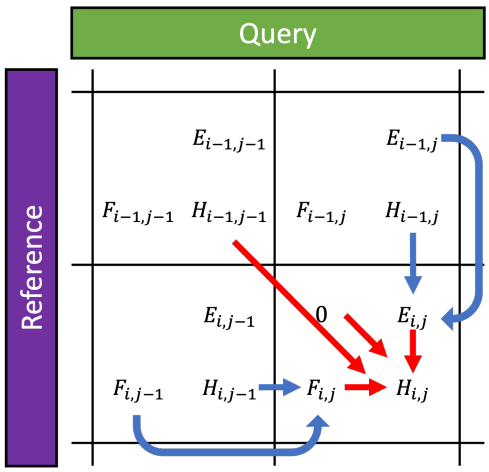
2.1 Seed Extension

Seed extension focuses on potential sequence similarities by identifying short, exact matches or "seeds." Extending these seeds enhances sensitivity to subtle similarities, reducing the search space for alignments and improving computational efficiency. Often performed by the Smith-Waterman algorithm, it makes the alignment process more manageable, especially for large genomic datasets.

2.2 Smith-Waterman Alogorithm

The Smith-Waterman algorithm is a DP-based method for local sequence alignment, designed to identify similarities in biological sequences [1]. It begins by initializing a score matrix, where the left side represents the reference sequence, and the upper side represents the query sequence as shown in Fig 2.1a. This matrix forms the foundation for subsequent dynamic programming steps. We use three matrices for an affine gap model to handle insertions and deletions in DNA sequences. The scoring matrix, denoted as $H(i, j)$, is used to store the scores of the best alignments ending at position (i, j) in the input sequences. The gap extension matrix, denoted as $E(i, j)$, is used to keep track of gap extension penalties. It represents the best score of alignments with a gap extension ending at position (i, j). The gap opening matrix, denoted as $F(i, j)$, is used to store the best scores of alignments with

a gap opening ending at position (i, j). An affine gap model follows the formula shown in Fig 2.1b.



(a) Calculations in DP Matrix.

$$H(i,j) = \max \begin{cases} 0 \\ H(i-1,j-1) + \mathrm{S}(a_i, b_j) \\ E(i,j) \\ F(i,j) \end{cases}$$

$$E(i,j) = \max \begin{cases} E(i,j-1) - \alpha \\ H(i,j-1) - \beta \end{cases}$$

$$F(i,j) = \max \begin{cases} F(i-1,j) - \alpha \\ H(i-1,j) - \beta \end{cases}$$

α : gap extend penalty, β : gap open penalty

(b) A formula for an Affine Gap Model.

Figure 2.1: An explanation of Smith-Waterman Algorithm.

The score function $S(i, j)$ returns a match score if i-th nucleotide of the reference and j-th nucleotide of the query match, and a mismatch penalty otherwise. Following the filling phase, the traceback phase identifies the optimal local alignment, as illustrated in Fig 2.2.

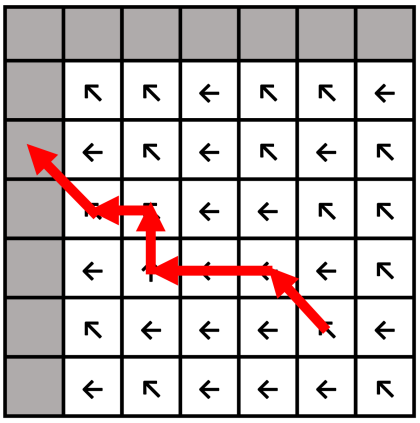


Figure 2.2: A traceback phase in Smith-Waterman Algorithm.

Starting from the highest score cell of the matrix, the algorithm traces the path back, revealing the aligned region of the sequences. For example, if the direction is left, it implies an

insertion in the alignment, indicating that a gap has been introduced in the sequence being aligned. Conversely, if the direction is up, it implies a deletion in the alignment, indicating that a gap has been introduced in the other sequence being aligned.

Chapter 3

Design

3.1 Baseline

The baseline design for this study leverages SALoBa as the primary method for filling DP matrices, chosen for its efficiency and proven effectiveness in sequence alignment tasks [2]. SALoBa is particularly selected for its capability to address memory inefficiency and workload imbalance challenges, making it well-suited for optimizing sequence alignment algorithms.

In SALoBa, only the part responsible for calculating the score matrix is implemented. Therefore, we additionally implemented a traceback kernel for the baseline and made additional work accordingly. During the filling phase, when calculating the values of the score matrix, we can determine the direction from which each value originates. We store all these direction values in global memory. To achieve this, additional memory must be allocated to store the direction, and we calculate the required size of this memory based on the lengths of the query and reference. This prevents the allocation of unnecessary memory. In the subsequent traceback phase, we retrieve the previously stored direction from global memory. Starting from the cell with the highest score, we move in the direction specified by that cell's direction (left, up, diag) until we reach the starting point of the query or reference. During each movement in the direction, we store the corresponding nucleotide or '-' for gaps in the result string. We employed inter-query parallelism, where each CUDA thread handles the traceback for one base pair.

3.2 Score Grid Based Traceback

To enhance the efficient utilization of memory resources, we introduced the concept of a score grid. We divided the score matrix into multiple square-shaped score grids, each with a side length of scoreGridSize. Utilizing this approach, we implemented each phase of the Smith-Waterman algorithm as follows.

3.2.1 Filling Phase

In the filling phase, unlike the baseline, we do not store all directions in global memory. Instead, we only store the scores corresponding to the boundary of the score grid depicted as yellow grids of Traceback phase in Fig 3.2. The essential data needed during the traceback phase is not the score but the direction. However, we store the scores because storing only the directions makes it impossible to calculate the correct values within the score grid later. By saving only a portion of the data from the score matrix, the required memory size decreases proportional to the scoreGridSize, reducing the overall memory footprint. We allocate memory to store the rows and columns of the score grid, and if the current cell spans across the respective row or column, we store its value to global memory.

3.2.2 Traceback Phase

During the traceback phase, the kernel initiates its operation from the score grid that contains the highest score, shown as orange cell in ① in Fig 3.2. Then it computes scores and fills the traceback direction matrix inside the grid. The directions are stored in thread-local memory. Due to the relatively small size of scoreGridSize compared to the total length of the string, one row of the score grid can be stored in shared memory. The process involves loading the 0th row from global memory to shared memory and subsequently calculating each row as it moves downward. To compute a single cell, the scores from left, upper, and diagonal cells are necessary. The score for upper cell is retrieved from shared memory, while the scores for left and diagonal cells are managed through thread-local memory. Therefore, to traceback a single base pair, each thread requires shared memory space capable of storing scores of length scoreGridSize, along with thread-local memory for storing left and up scores.

Additionally, thread-local memory is utilized for storing directions, requiring a memory size of $scoreGridSize * scoreGridSize$. The calculation of the score grid involves updating left and diag values as the algorithm moves one cell to the right within a row. The up value is updated with the current cell's score using a rolling window method, as visually demonstrated in Fig 3.1. The red cells represent the current computation being performed, while the green cells denote thread-local memory, and the purple cells represent shared memory. After filling an entire row, the process proceeds to the next row, and so on. This approach enables efficient operations by consistently leveraging a small shared memory.

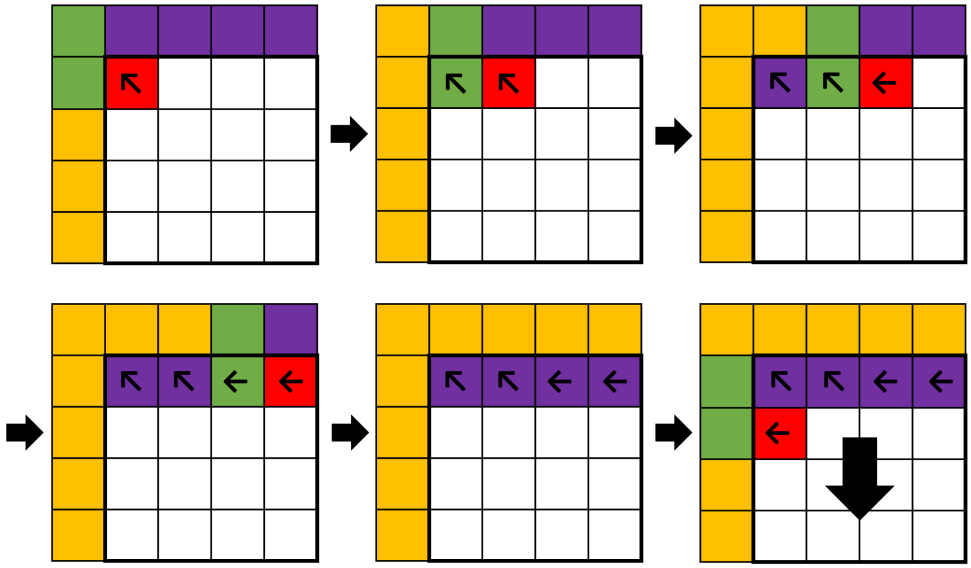


Figure 3.1: Rolling Window Method used in Score Grid Filling.

Once the traceback within a score grid is complete, the process seamlessly transitions to an adjacent score grid. If the endpoint of the traceback is at the top-left corner of the current score grid, the process moves diagonally to the neighboring score grid. If it is along the left edge, the process shifts to the score grid on the left, and if along the top edge, it transitions to the score grid above. Likewise, in the example in Fig 3.2, the traceback within the score grid ① ends on the left side, progressing to the adjacent score grid ② on the left. In ②, where the traceback within the score grid ends at the top-left, it transitions diagonally to the adjacent score grid ③. When transitioning to the left or above score grid, there can be an optimization in the calculation of directions during score grid filling, calculating only the necessary rectangular part of the grid, shown as blue cells in ② of Traceback part in

Fig 3.2. Here, there is no need to perform filling for the entire score grid because the cell coming from the previous score grid belongs to a row in the middle of the current score grid. Therefore, the direction data of the row below is not necessary for the traceback, and a similar approach can be applied when transitioning to the score grid above, helping reduce the computational workload. This iterative process of score grid filling (shown in 2-1 of Fig 3.2) and score grid traceback (shown in 2-2 of Fig 3.2) continues until reaching the starting point of the query or reference, that are, the leftmost or the uppermost cells of score matrix.

The reason this approach leads to performance improvement lies in the fact that not all values in the score matrix are utilized during traceback. Since determining which values will not be used before traceback is challenging, we saved the minimum required data, such as boundary scores of the score grid. Additionally, the use of score grids facilitates the utilization of shared memory, resulting in a performance improvement of traceback process.

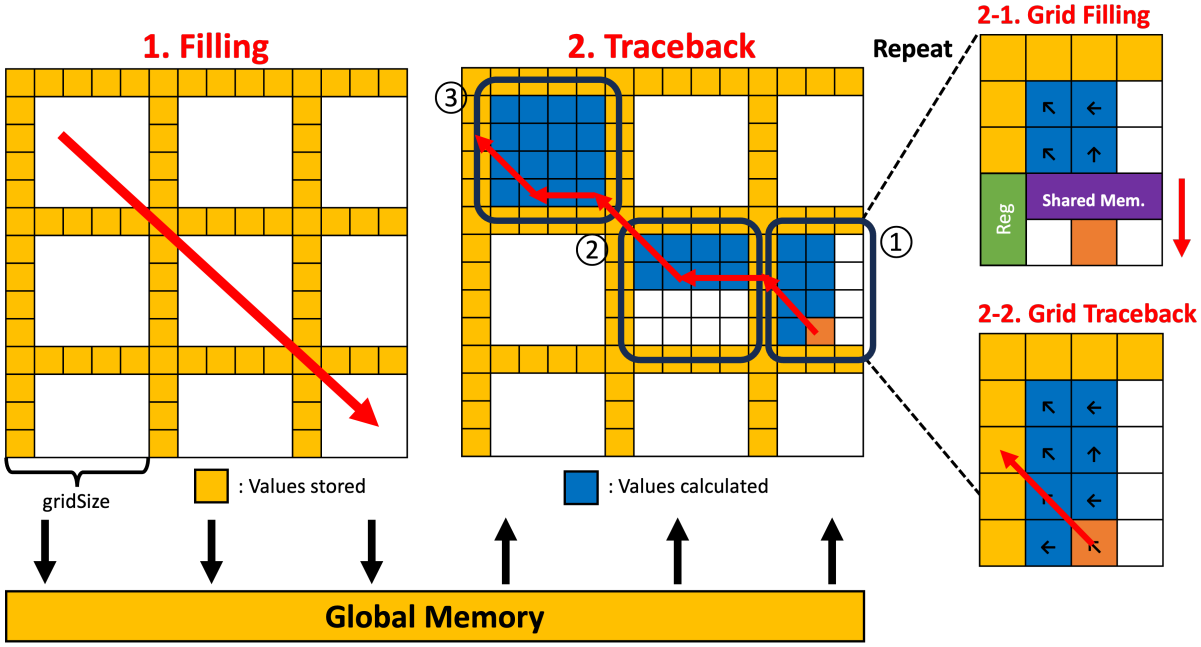


Figure 3.2: Overall Structure of Score Grid Based Kernel.

Chapter 4

Evaluation

4.1 Experimental Setup

The experiments were conducted on an RTX 4090 GPU, providing a robust platform for evaluating the performance of both the baseline and the proposed design in DNA sequence alignment. The RTX 4090, known for its high computational capabilities and parallel processing prowess, serves as an ideal hardware environment for bioinformatics applications, particularly those involving intensive computations like DNA sequence alignment.

4.2 Parameter Analysis

We first analyzed the following parameters by varying them:

batchSize : Determines the number of sequence pairs processed in a single iteration.

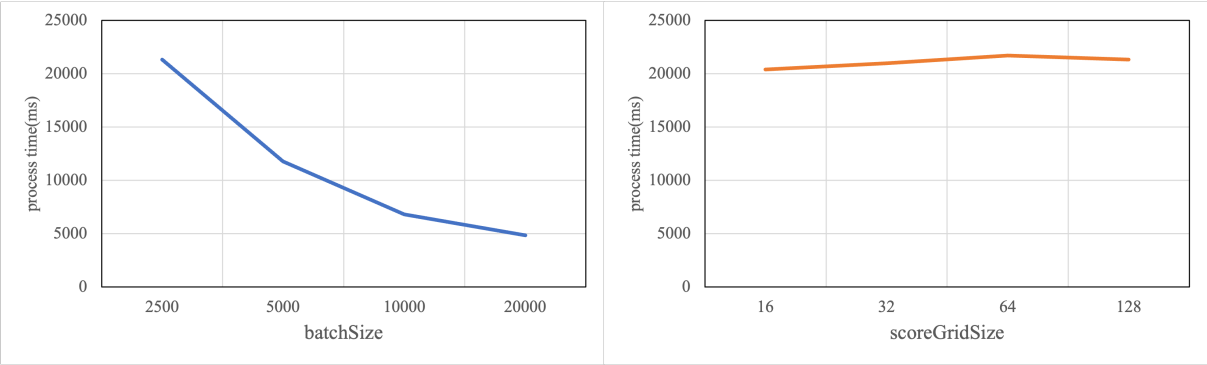
scoreGridSize : Defines the size of the score grid, which partitions the score matrix.

threadBlockSize : Specifies the number of CUDA threads within a single thread block.

While keeping other parameters fixed, the results obtained by varying batchSize and scoreGridSize are depicted in Fig 4.1. In Fig 4.1a, it is observed that as batchSize increases, the process time significantly decreases. This is likely due to the effective utilization of Stream-

ing Multiprocessors (SMs). The RTX 4090 has 144 SMs. Since the traceback kernel operates in an inter-query manner, when executing the kernel, batchSize/threadBlockSize thread blocks are generated. Typically, a thread block has a size that is a multiple of 32, which is the number of threads in a warp. Therefore, even with a minimum threadBlockSize of 32, to assign at least one thread block to each SM, batchSize of 4608 or more is required. The performance continues to improve as batchSize increases up to 20000, as latency within the SM, such as memory access time, can be hidden by context switching to another thread block.

Additionally, from observing Fig 4.1b, it can be seen that as the scoreGridSize increases, the execution time slightly increases. This is attributed to the growing computational load of unnecessary direction operations during traceback. However, the overall performance variation with respect to scoreGridSize is considerably smaller than the impact of batchSize, allowing for optimal performance by adjusting batchSize effectively.



(a) Varied BatchSize. (b) Varied ScoreGridSize.

Figure 4.1: Process Time Measurements with Varied Parameters.

Through the analysis of the relationship between parameters to find an effective batchSize, the following constraints were identified.

$$f\left(\frac{\text{batchSize}}{\text{scoreGridSize}}\right) \lesssim (\text{global memory size})$$

$$g(\text{threadBlockSize} \times \text{scoreGridSize}) \lesssim (\text{shared memory size})$$

These relationships highlight the dependencies between the parameters and certain mem-

ory constraints during the analysis.

4.3 Real-World Data Experiments

For our comparative analysis between the baseline and our design, we selected diverse datasets, including HG002, HG002 ONT, and HG005 as shown in Table 4.1. It shows the size of the dataset in base pairs (bp), the number of query and reference nucleotides, the sum of both, and the maximum length of a single sequence. These datasets feature sequences of varying sizes, providing a comprehensive evaluation across different scenarios. Leveraging datasets with diverse characteristics ensures a more robust assessment of the performance and adaptability of our design.

Table 4.1: Experiment Datasets

Data	Size (bp)	Query	Ref	Overall	Max
HG002 ONT	78,249	69M	83M	153M	19,537
HG002	180,957	278M	365M	644M	9,975
HG005	133,857	166M	201M	367M	11,626

Despite optimizing the baseline for efficient placement of the direction matrix in global memory, the baseline had limitations in scaling up batchSize due to the need to store a significant amount of data. In an effort to use the largest possible batchSize within a feasible range, we conducted measurements with a batchSize of 1000 and a threadBlockSize of 32. For our kernel, we used a batchSize of 20000, scoreGridSize of 128, and a threadBlockSize of 64 for measurements.

As shown in Fig 4.2, our kernel exhibited more than an 80x performance improvement compared to the baseline. This result can be attributed to the relatively loose memory constraints, allowing for a larger batchSize and better utilization of numerous SMs. Additionally, the exclusion of unnecessary computations led to a significant reduction in the overall workload, contributing to the performance enhancement.

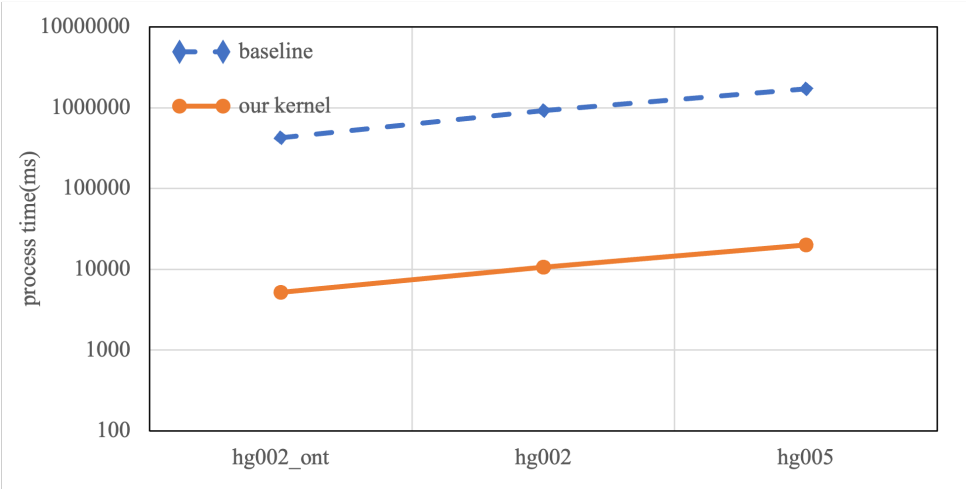


Figure 4.2: Comparison of Baseline and Our Kernel. The y axis is in log scale for visibility.

Chapter 5

Conclusion

In summary, our research focused on optimizing the traceback phase of DNA sequence alignment, a critical task in bioinformatics. The conventional approach, based on the Smith-Waterman algorithm, encounters challenges related to excessive memory requirements and inefficient shared memory utilization during traceback. To address these issues, we proposed a novel design that introduces the concept of score grids.

In conclusion, our optimized DNA sequence alignment algorithm showcases a promising approach to address the challenges associated with memory efficiency, offering enhanced performance and scalability. The results of this research provide a more efficient solution for processing large genomic datasets.

Bibliography

[1] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences", *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981. DOI: 10.1016/0022-2836(81)90087-5.

[2] S. Park, H. Kim, T. Ahmad, *et al.*, "Saloba: Maximizing data locality and workload balance for fast sequence alignment on gpus", in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, May 2022, pp. 728–738.