**A Fast Implementation of Matrix Chain Multiplication using GPU-Pipeline Algorithm**

**Abstract**

In this paper, using the pipeline implementation of Dynamic Programming (DP) on GPU, we demonstrate its utility in solving complex dynamic programming problems such as a matrix-chain multiplication problem on GPU. Serial CPU version of matrix-chain multiplication has a complexity of O(n3) when utilizing bottom-up DP implementation, n is the number of matrices. The solution is obtained by populating the DP table of size n × n wherein each cell is populated using O(n) steps. Previous works parallelize the O(n) step computation using parallel prefix computation leading to O(log n) step computation. This leads to overall O(n2 log n) complexity with n threads. We solve the matrix-chain multiplication problem on GPU in a pipeline fashion, wherein each GPU core populates the DP solution table are partially in parallel. This leads to a much better time complexity of O(n2) using n threads.

**Availability:** Latest code along with results are available at <https://github.com/ARBasharat/matrix_chain_multiplication_GPU_Pipeline.git>

**Introduction**

In matrix-chain multiplication problem, we are given N matrices, and aim is to compute the product matrix. Since matrix multiplication holds associative property, this task can be accomplished in several number of ways by selecting different combinations. Each combination of matrix multiplication can lead to varying cost. Therefore, it is highly recommended that we establish correct parenthesis of the matrices so that the cost of multiplication is minimum.

Since each combination of matrix multiplication will provide us with the correct solution, this makes it an optimization problem. If we try to compute the correct combination of parenthesis for multiplication using brute force method, the problem becomes an exponential problem in n, which makes it a poor strategy. If we carefully observe the nature of the problem, we find that the problem follow the optimal substructure property which makes this problem an inherent dynamic programming problem.

Over the years, several techniques have been used for optimizing the dynamic programming implementation. One such method parallelize the computation of each element on n threads by utilizing parallel prefix algorithm leading to O(n2log n) complexity for matrix-chain multiplication problem. In this work, we will utilize GPU pipeline algorithm for computing the matrix-chain multiplication. Each GPU cores computes partial solution in parallel at one time. The pipeline algorithm, utilizing n threads, computes one output value per one computational step and constructs the solution table with O(n2) complexity. This provides a significant speed-up over CPU implementation which has O(n3) complexity. Furthermore, this exhibits the utility of the pipeline implementation in solving other dynamic programming problems such as spectral alignment, Manhattan sales man problem, rod cutting etc. in a much efficient manner.

**Algorithm and Methods**

Matrix-chain multiplication problem is traditionally solved by dynamic programming algorithm wherein we populate a two-dimensional solution table of triangular shape (Figure 1).

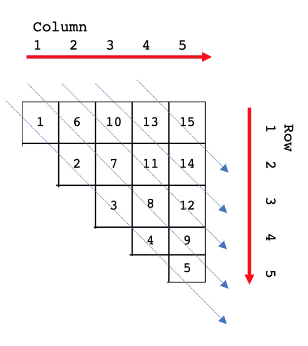
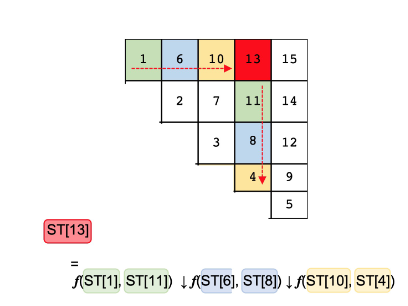
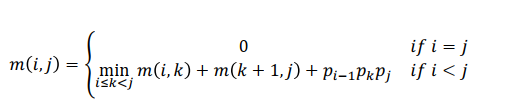
 

Figure . Dynamic programming table of MCM

The content of each element is computed based on a function defined below.



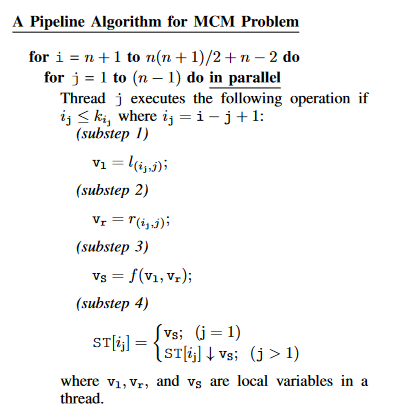
In fugure, downwards arrow represent the minimum operation and f computes the number of multiplication based on the matrix dimensions.

For GPU implementation, we assume the original DP solution table of diagonal shape is already mapped to a one dimensional solution table ST appropriately. For that, line up all the elements of the two-dimensional table are transformed into a linear array ST. The number of elements in the solution table (ST) is n(n + 1)/2, n is number of input matrices. First n elements of array ST are initialized and correspond to the elements located in the diagonal line of the original DP matrix.

Each computation for ST[i] be represented as



It should be noted that here ki will differ from others and n−1 is the largest value. In substep 1, 2, and 3, the computation of f(l(i,j), r(i,j)) is executed, and in substep 4, that obtained value is used for the computation by ↓ and the result is stored to ST[i] Figure 2.



An example with 5 input matrices has been shown below.



**Software design and Implementation:**

The code has been implemented using CUDA C++. The serial version has also been implemented to compare the results.

**Results:**

The pipeline implementation had a time complexity of O(n2) as compared to O(n3) of the serial version. Due to some confusions (computation of l and r in the substep 1 and 2) in the paper. I unfortunately could not complete the implementation. I have built the structure of the implementation.