HPC Mini-Project

Dr. D.Y. Patil Pratishthan's

D.Y. Patil College of Engineering, Akurdi, Pune-44

Department of Computer Engineering

Laboratory Practice V

PROJECT TITLE: Implement Non-Serial Polyadic Dynamic Programming with GPU Parallelization

Sr No	Name of Students	Roll No
1.	Aditya Sadakal	BECO2324A001
2.	Tushar Dhobale	BECO2324A014
3.	Swapnil Bonde	BECO2324A018

Class: BE COMPUTER ENGINEERING SEM-II

Div.: A

Academic Year: 2023-24

[illegible]

Signature of Guide

(Mrs. Dhanashree Phalke)

Problem Statement:

The problem involves implementing Non-Serial Polyadic Dynamic Programming, which requires performing multiple tensor dot products on arrays. The objective is to accelerate the computation using GPU parallelization to improve performance.

Objectives:

- Implement Non-Serial Polyadic Dynamic Programming algorithm.
- Leverage GPU parallelization for faster computation.
- Compare the performance of GPU-accelerated implementation with CPU-based implementation.

Software Requirements:

- Python programming language
- CuPy library (for GPU-accelerated computation)
- NumPy library (for array operations)
- Google Colab (for running GPU-accelerated code)

Hardware Requirements:

- Computer with GPU support (NVIDIA GPU recommended)
- Internet connection for accessing Google Colab.

Theory:

Non-serial polyadic dynamic programming (NSPD) is an optimization technique used in computer science to solve problems that involve multiple variables or dimensions. This approach extends the concept of dynamic programming to handle problems with more than two dimensions. Here's an overview of the theory behind NSPD:

Dynamic Programming (DP):

- Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems and solving each subproblem only once.
- It stores the solutions to subproblems in a table (usually implemented as an array or matrix) and reuses these solutions to solve larger subproblems or the original problem.

- The key idea of dynamic programming is to avoid redundant computations by storing and reusing intermediate results.

Polyadic Dynamic Programming:

- Polyadic dynamic programming extends the concept of dynamic programming to problems with multiple variables or dimensions.
- Instead of optimizing a function with respect to a single variable (as in traditional dynamic programming), polyadic dynamic programming optimizes a function with respect to multiple variables simultaneously.
- It involves defining a multi-dimensional table to store intermediate results, where each dimension corresponds to a different variable.
- The computation of values in the table typically involves nested loops or recursive calls to fill in the entries.

Non-Serial Polyadic Dynamic Programming (NSPD):

- NSPD further generalizes polyadic dynamic programming by allowing the computation of table entries in a non-serial (non-linear) order.
- In traditional dynamic programming, table entries are computed in a serial order, where each entry depends only on previously computed entries.
- NSPD relaxes this constraint and allows for more flexible computation strategies, such as parallel computation or exploration of alternative computation paths.
- This flexibility can lead to more efficient algorithms for solving certain types of problems, especially those with complex dependencies or non-linear relationships between variables.

Applications:

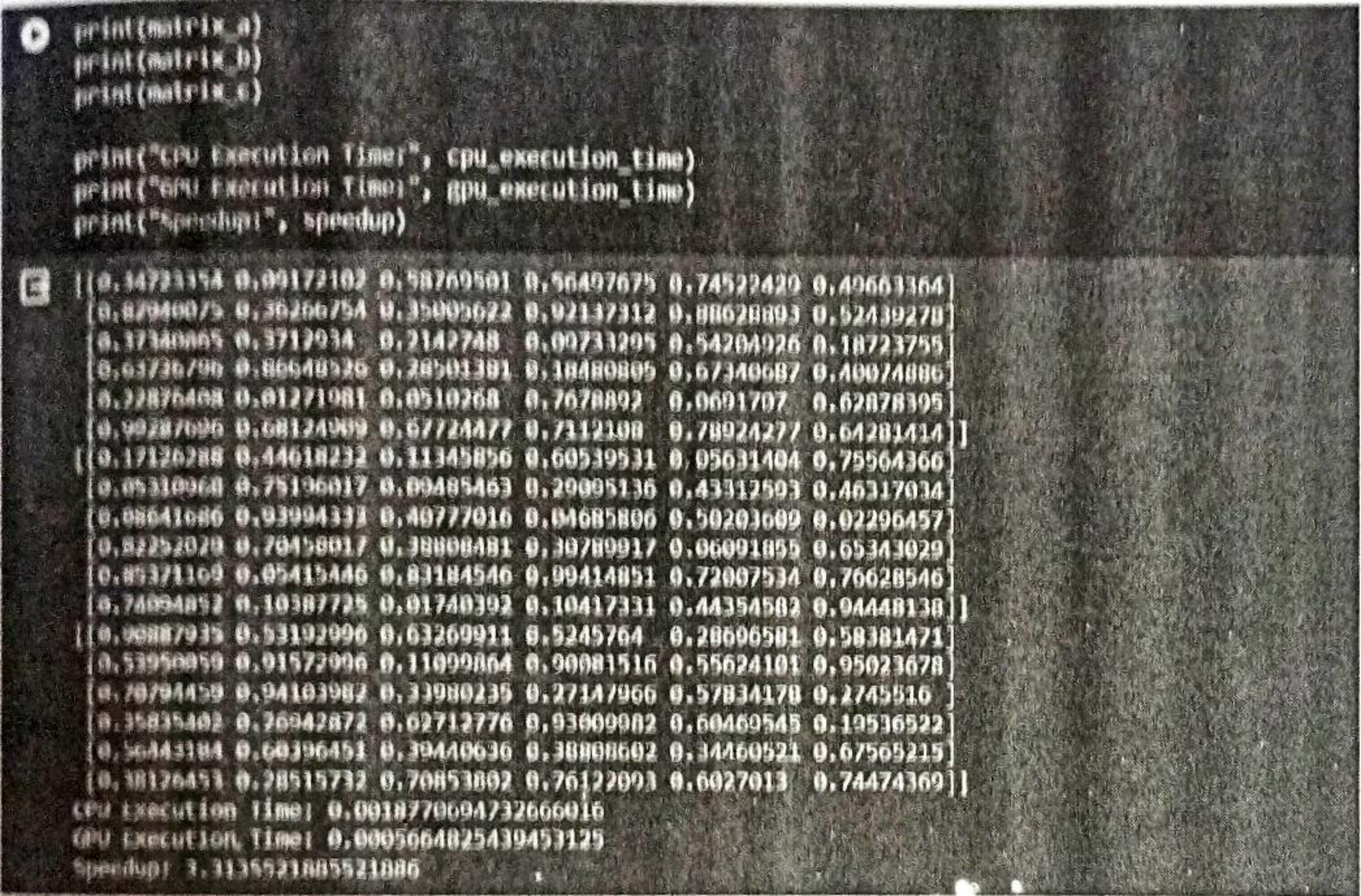
- NSPD algorithms are used in various fields such as optimization, machine learning, computational biology, and graph algorithms.
- They are particularly useful for problems involving large-scale optimization or search over high-dimensional spaces.
- Examples of problems that can be tackled using NSPD include sequence alignment, graph traversal, combinatorial optimization, and machine learning model training.

Source Code:

```
import cupy as cp

def non_serial_polyadic_dynamic_programming(matrix_a, matrix_b, matrix_c):

    # Convert input matrices to CuPy arrays

    a_gpu = cp.array(matrix_a)
    b_gpu = cp.array(matrix_b)
    c_gpu = cp.array(matrix_c)
    # Perform dynamic programming calculations
    result_gpu = cp.tensordot(a_gpu, b_gpu, axes=0)  # First polyadic product
    result_gpu = cp.tensordot(result_gpu, c_gpu, axes=0)  # Second polyadic product

    # Convert result back to NumPy array
    result = cp.asnumpy(result_gpu)

    return result

# Example usage
matrix_a = cp.random.rand(10,10)  # Example matrix A
matrix_b = cp.random.rand(10,10)  # Example matrix B
matrix_c = cp.random.rand(10,10)  # Example matrix C

import time

# Function to measure CPU execution time
def measure_cpu_execution_time(func, *args):
    start_time = time.time()
    func(*args)
    end_time = time.time()
    return end_time - start_time

# Function to measure GPU execution time
def measure_gpu_execution_time(func, *args):
    cp.cuda.Stream.null.synchronize()  # Synchronize with GPU operations
```

```
start_time = time.time()

func(*args)

cp.cuda.Stream.null.synchronize() # Synchronize with GPU operations

end_time = time.time()

return end_time - start_time

# Measure CPU execution time

cpu_execution_time =
measure_cpu_execution_time(non_serial_polyadic_dynamic_programming, matrix_a,
matrix_b, matrix_c)

# Measure GPU execution time

gpu_execution_time =
measure_gpu_execution_time(non_serial_polyadic_dynamic_programming, matrix_a,
matrix_b, matrix_c)

# Calculate speedup

speedup = cpu_execution_time / gpu_execution_time

print("CPU Execution Time:", cpu_execution_time)

print("GPU Execution Time:", gpu_execution_time)

print("Speedup:", speedup)
```

Output:



```
print(matrix_a)
print(matrix_b)
print(matrix_c)

print("CPU Execution Time:", cpu_execution_time)
print("GPU Execution Time:", gpu_execution_time)
print("Speedup:", speedup)
```

```
[[0.34723354 0.09172102 0.58769501 0.56497675 0.74522420 0.49663364]
 [0.87940075 0.36266754 0.35905622 0.92137312 0.88628893 0.52439278]
 [0.37340005 0.3712934  0.2142748  0.00733295 0.54204926 0.18723755]
 [0.63736790 0.86648526 0.28503381 0.18480805 0.67340687 0.40074886]
 [0.22876408 0.01271981 0.0510268  0.7678892  0.0691707  0.62878395]
 [0.99287696 0.68124909 0.67724477 0.7112108  0.78924277 0.64281414]]
[[0.17126288 0.44618232 0.11345856 0.60539531 0.05631404 0.75564366]
 [0.05310968 0.75196017 0.09485463 0.29095136 0.43317593 0.46317034]
 [0.08641086 0.93994331 0.40777016 0.04685806 0.50203609 0.02296457]
 [0.82252020 0.70458017 0.38808481 0.30789917 0.06091855 0.65343029]
 [0.85371109 0.05415446 0.83184546 0.99414851 0.72007534 0.76628546]
 [0.74094857 0.10387725 0.01740392 0.10417331 0.44354582 0.94448138]]
[[0.90887935 0.53192996 0.63269911 0.5245764  0.28696581 0.58381471]
 [0.53950959 0.91572996 0.11099864 0.90081516 0.55624101 0.95023678]
 [0.70794459 0.94103982 0.33980235 0.27147966 0.57834178 0.2745516 ]
 [0.35835402 0.26942872 0.62712776 0.93009982 0.60469545 0.19536522]
 [0.56443184 0.60396451 0.39440636 0.38808602 0.34460521 0.67565215]
 [0.38126451 0.28515732 0.70853802 0.76122093 0.6027013  0.74474369]]
CPU Execution Time: 0.0018770694732666016
GPU Execution Time: 0.0005664825439453125
Speedup: 3.3136521885521886
```