Project Report: Photon Correlation for Two Qubit Using 1 Dimensional Detector

Anuj Rajan Lalla
B22AI061

Suralkar Pranav Nanasaheb
B22ME066

Vaibhav Gupta
B22CS058

May 3, 2024

Abstract

This report details our approach to solving the problem of photon correlation for two qubit using a one-dimensional detector. It covers the initial and updated approaches to vector operations and their impacts on computational performance and scalability.

1 Introduction

The project aims to manipulate and analyze photon data using computational techniques to study the photon correlation for a two-qubit system. The core challenge was managing the computational complexity and improving the performance of these calculations.

2 Problem Statement

The problem involves subtracting two input vectors of the same dimension element-wise, rotating one vector, repeating this step until the size of the vector minus one times, and concatenating these results to form a resultant vector with dimensions of size(vector) x size(vector).

3 Initial Approach

Code Overview

1. Library Imports

- **numpy**: Used for numerical operations with arrays.
- **pandas**: Utilized for data manipulation and reading from CSV files.
- **matplotlib.pyplot**: Employed for plotting graphs.
- **time**: Used to track the execution time of parts of the code.

2. Function Definition - vector_operations

- **Parameters**:
  - **vec1, vec2**: Input vectors.
- **Process**:
  - Checks the lengths of **vec1** and **vec2**, and pads the shorter vector with zeros to match the length of the longer vector.
  - Initializes a result vector with the difference of **vec1** and **vec2**.
  - Performs left rotations on **vec1** and computes the differences for the first half of iterations.
  - Performs right rotations on **vec2** and computes the differences for the second half of iterations.
  - Each rotation's difference is appended to the result vector.

3. Main Execution Loop

- **CSV File Handling**:
  - Specifies the path to the CSV file `DC.csv`.
  - Defines a range of vector sizes (**values_range**) from 100 to 2000 with a step of 100 to test different input sizes.
- **Data Loading and Execution**:
  - Loads data from `DC.csv`, limiting the read to the first two columns and the number of rows specified by **values_range**.
  - Converts the read columns to NumPy arrays **vec1** and **vec2**.
  - For each size in **values_range**, executes the **vector_operations** function, measures the execution time, and stores it in **times**.
- **Performance Measurement**:
  - Plots the execution times against the vector sizes using Matplotlib to visualize the computational efficiency and scalability of the function.

4. Graph Plotting

- Plots the computational times against the number of values using a line graph.
- Marks each data point with an 'o'.
- Labels the x-axis as “Number of Values” and the y-axis as “Computational Time (seconds)”.
- Adds a title and a grid to the plot for better readability and presentation.

5. Results Visualization

- The final graph displays how computational time varies with the size of the input data, providing a visual representation of the performance scalability and efficiency of the chunked processing method.

3.1 Purpose

The initial approach aimed to analyze how the computational time is affected by the complexity of these manipulations as the vector size increases.

3.2 Visualization

Computational times were plotted against the number of vector elements to visualize performance trends.

4 Drawbacks of Initial Approach

- **Quadratic Complexity**: The computational time increased quadratically with the number of values.
- **Scalability Issues**: The approach showed poor scalability, especially for large vector sizes.

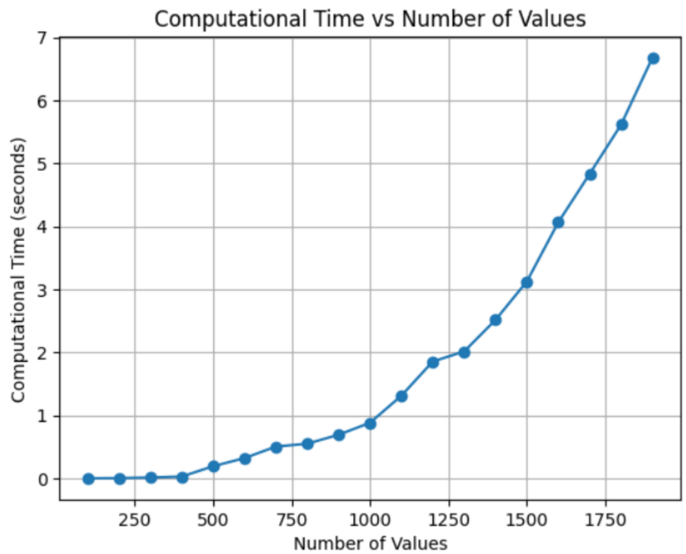

Figure 1: Time vs sample

5 Updated Approach

Code Overview

1. Library Imports

- **numpy**: For numerical operations with arrays.
- **pandas**: For data manipulation and reading from CSV files.
- **matplotlib.pyplot**: For plotting graphs.
- **time**: For tracking the execution time of code.

2. Function Definition - rotate_and_operate_chunked

Parameters

- **vec1, vec2**: Input vectors.
- **chunk_size**: Specifies the number of rows to process at a time.
- **verbose**: Boolean flag to enable progress messages during execution.

Process

- Determines the total number of operations required (n^2, where n is the length of $vec1$).
- Initializes an empty array **result_vector** of size n^2 using `float32` to reduce memory consumption.
- Processes the vectors in chunks:
  - For each chunk, computes the differences after rotating $vec1$ progressively by one element per iteration.
  - Stores the result in a temporary array **temp_result**.
  - Flattens **temp_result** and places the results in the correct segment of **result_vector**.

3. Main Execution Loop

CSV File Handling

- Specifies the path to the CSV file `DC.csv`.
- Defines a range of values (**values_range**) to process in each iteration.

Data Loading and Conversion

- Loads data from the specified CSV file using **pandas**, limiting the read to the first two columns.
- Converts the data to `float32` to align with the array type used in the function.

Performance Measurement

- Measures the time to execute the **rotate_and_operate_chunked** function for each vector size.
- Records start and end times.
- Stores execution times in the list **times**.

- Optionally captures the first 10 elements of the result for verification (**results** list).

4. Graph Plotting

- Plots computational times against the number of values using a line graph.
- Marks each data point with an 'o'.
- Labels the x-axis as "Number of Values" and the y-axis as "Computational Time (seconds)."
- Adds a title and a grid for better readability.

5. Results Visualization

The final graph displays how computational time varies with the size of the input data, providing a visual representation of the performance scalability and efficiency of the chunked processing method.

5.1 Results

Significant reduction in the computational time with the updated approach, though converting the result vector to a CSV file and storing in ascending order introduced overhead.

6 Conclusion

The updated approach effectively optimized both time and memory usage, addressing the major scalability and performance issues found in the initial methods.

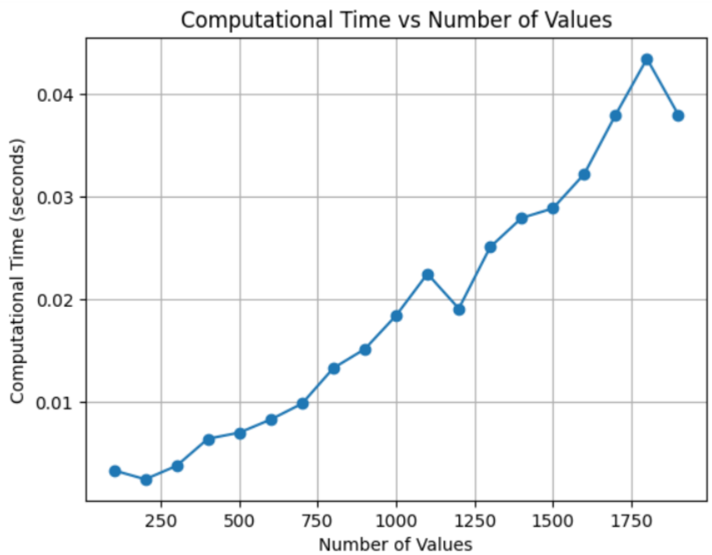

Figure 2: Time vs sample

7 Appendices

7.1 GitHub Repository

Link to our GitHub repository for this project: `https://github.com/anuj-122/DC_Project`