# Clustermatch

Introduction

The Clustermatch Correlation Coefficient (CCC) is a Python library designed to identify both linear and nonlinear relationships between datasets. Unlike traditional correlation coefficients such as Pearson or Spearman, which focus on linear or monotonic relationships, CCC uses clustering to detect more complex patterns in data. It works by dividing numerical variables into clusters based on quantiles and treating categorical variables as natural clusters, then measuring the similarity of these clusters using the Adjusted Rand Index (ARI). This approach enables CCC to reveal intricate relationships, including nonlinear patterns while being computationally efficient and adaptable to both numerical and categorical data.

Our group aimed to propose a version of the CCC library that leverages GPU parallelization to accelerate the computation process and enhance overall efficiency.

Libraries for GPU parallelization in Python

Several libraries enable GPU parallelization in Python, with most being CUDA-based. Examples include Numba, PyCUDA, and libraries tailored for specific tasks, such as CuPy, which focuses on accelerating NumPy operations.

We selected Numba for the following reasons:

1. It does not require altering the existing data structure, which CuPY requires.
2. Its coding syntax closely resembles CUDA in C
3. It offers more comprehensive documentation compared to PyCUDA

Profiling

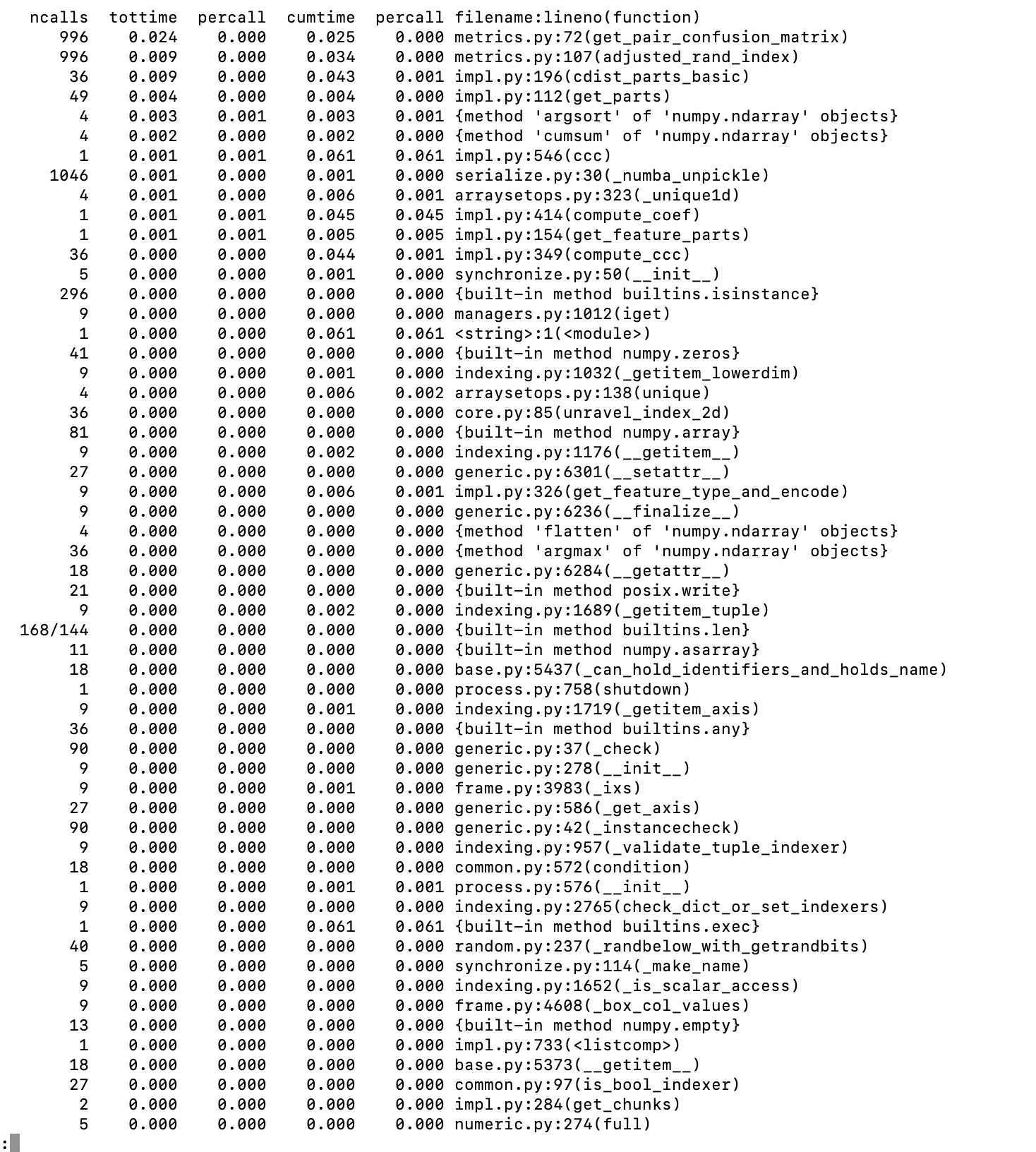
We began by profiling the existing code to identify functions with high parallelization potential. To accomplish this, we used the %prun command while executing the ccc function on the [Titanic dataset](https://www.kaggle.com/c/titanic/data). The resulting profile is as follows

Figure 1: Profiling result

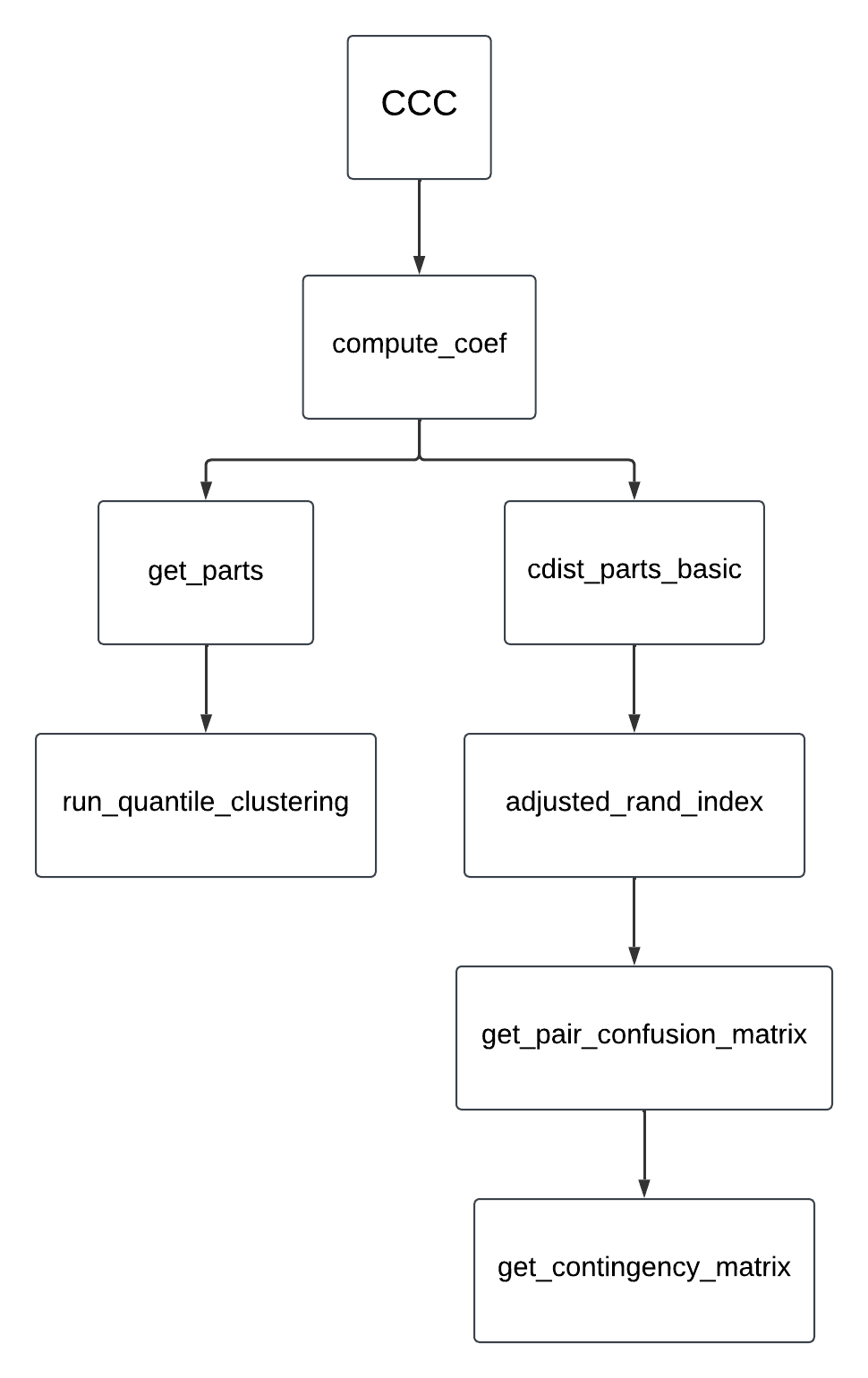
The most time consuming functions were get\_pair\_confusion\_matrix(), adjusted\_rand\_index(), cdist\_parts\_basic(), get\_parts(), ccc() and compute\_coef().

Figure 2: Function hierarchy for the functions discussed above.

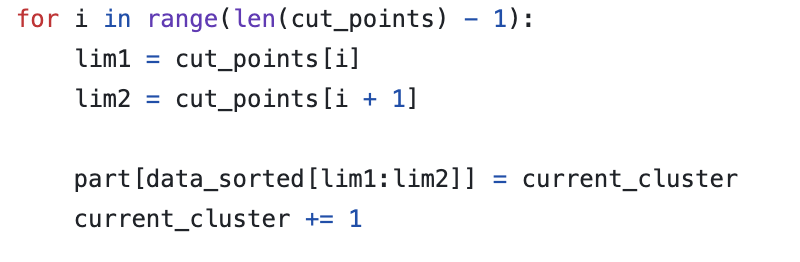
The cdist\_parts\_basic() function already employs a CPU-parallelized solution using an executor, while the get\_parts() function, designed to compute data partitions based on a given range of cluster values, relies on clustering results for specific cluster values (k). Due to its relatively low computational complexity and lightweight array operations, get\_parts() is more efficiently executed on the CPU, where parallelization can be managed without the overhead of GPU acceleration.

Figure 3: Core code for the run\_quantile\_clustering() function

So, we focused our code implementation on get\_pair\_confusion\_matrix() and adjusted\_rand\_index(), which are extensions of the sklearn library's pair\_confusion\_matrix and adjusted\_rand\_score function. get\_pair\_confusion\_matrix() generates a matrix that illustrates pairwise comparison of clustering assignments, while adjusted\_rand\_index() computes the Adjusted Rand Index to evaluate the similarity between two clusters.

# Code Implementation

[Link to the full implementation](https://github.com/quinnhyx/ccc/pull/2/files)

As mentioned earlier, we focused our parallelization efforts on functions in metrics.py, which includes get\_contingency\_matrix(), get\_pair\_confusion\_matrix() and adjusted\_rand\_index(). Specifically, we parallelized code that carries out matrix operations.

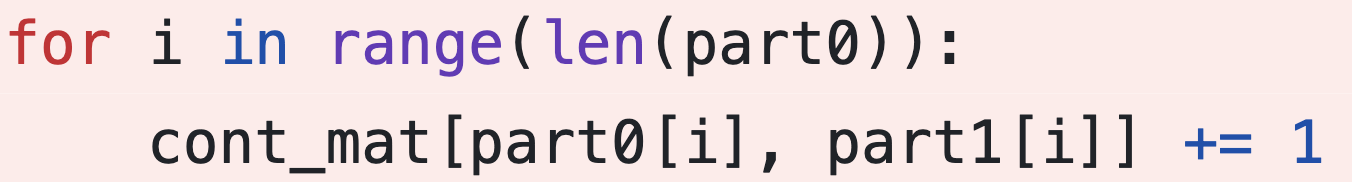
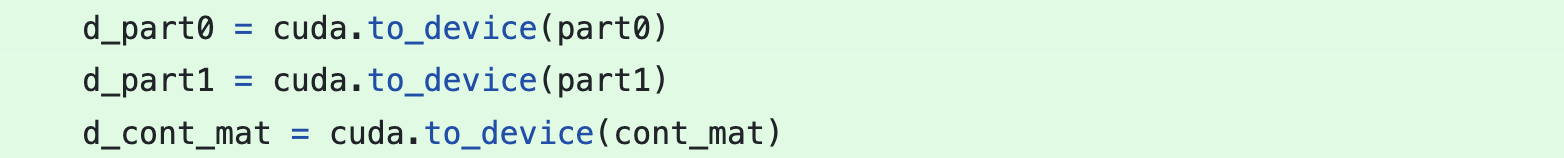
## Part 1: get\_contingency\_matrix()

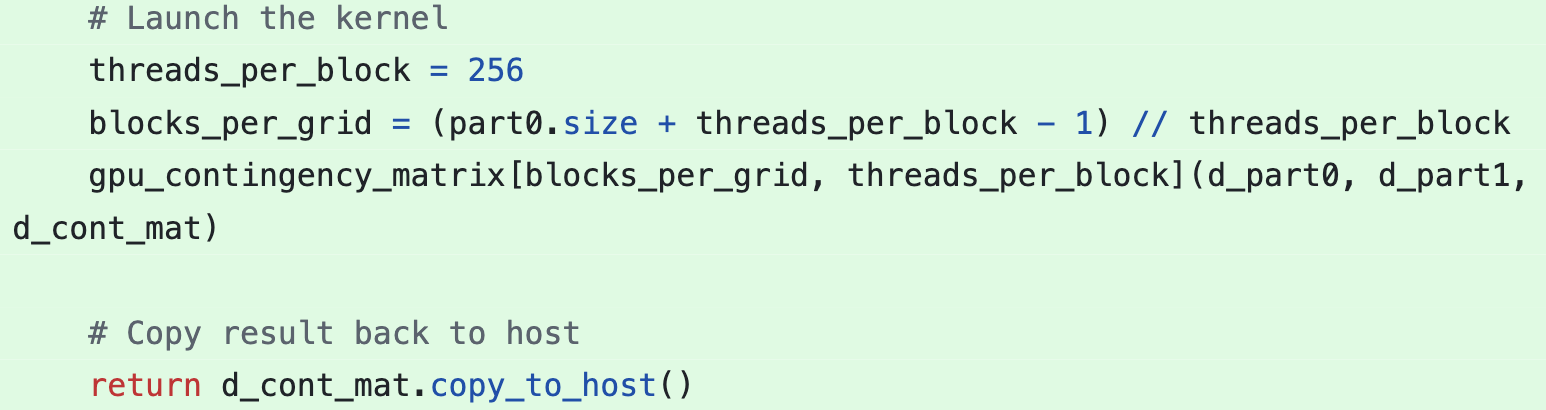
Figure 4: Original Implementation for get\_contingency\_matrix()

get\_contingency\_matrix() takes two inputs, part0 and part1, which are both 1d integer arrays that represent a partition with cluster assignments for some number of objects. get\_contingency\_matrix() demonstrates how these objects are categorized to clusters by generating a contingency matrix with k0 (number of clusters in part0) rows and k1 (number of clusters in part1) columns, where each cell (i, j) illustrates the count of objects that are grouped to cluster i in part0 and cluster j in part1.

Below lists parts of the function that we parallelized:

### 1.1 Counting the number of objects categorized to clusters in two partitions

Before: After:



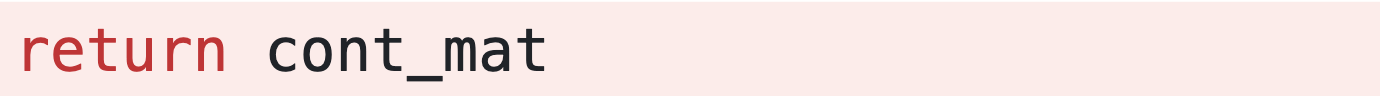


Figure 5: Parallelized Implementation Part 1

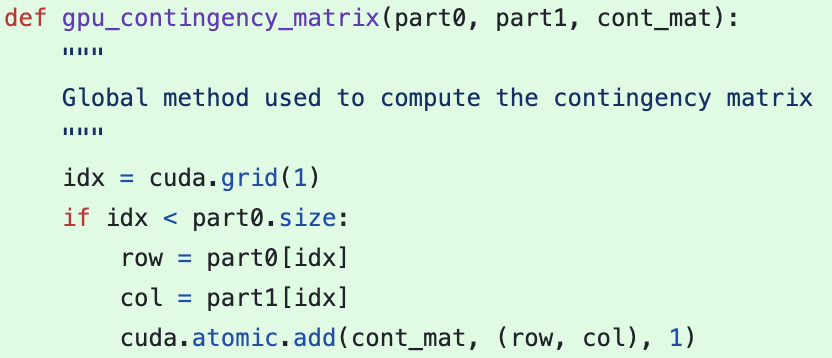
To run the original for loop that counts the number of objects grouped to clusters in part0 and part1 in parallel, we first copied part0, part1, and the initialized contingency matrix, cont\_mat, to the GPU. Then, we launched kernels with (part0.size + 256 - 1) // 256 blocks and 256 threads per block, where the implementation for the global method is demonstrated in Figure 6. 

Figure 6: Implementation for the global method gpu\_contingency\_matrix()

In the global method gpu\_contingency\_matrix(), we first obtained a global thread index idx, and for all idx’s less than part0.size, we called atomic.add(), which adds one to cell (part0[idx], part1[idx]) of cont\_mat, the contingency matrix.

We eventually copied the matrix back to the host after the kernel launch and returned it directly.

## Part 2: get\_pair\_confusion\_matrix()

Figure 7: Original implementation for get\_pair\_confusion\_matrix()

Like get\_contingency\_matrix(), get\_pair\_confusion\_matrix() also takes part0 and part1 as its input, yet it produces a confusion matrix. To be more specific, this function considers pairs of objects and assigns them to one of the following categories:

1. True negative: if the two objects are in different clusters in both partitions
2. False negative: if the two objects are in the same cluster in part0 but not in part1
3. True positive: if the two objects are in the same cluster in both partitions
4. False positive: if the two objects are in the same cluster in part1 but not in part0

get\_pair\_confusion\_matrix() will eventually return a confusion matrix, where the count of true negatives is in cell (0,0); the count of false negatives is in (1,0); the count of true positives is in (1,1); and the count of false positives is in (0,1).

Below lists parts of the code that we parallelized:

### 2.1 Calculate the sum of each row in a matrix

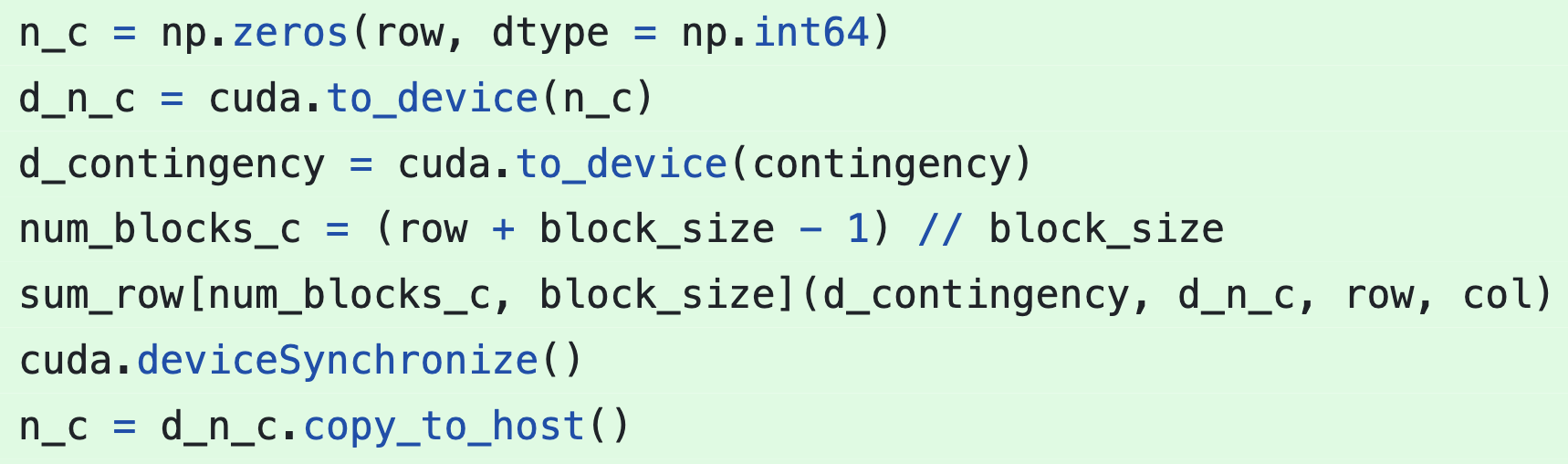
Before: After:

Figure 8: Parallelized Implementation Part 2

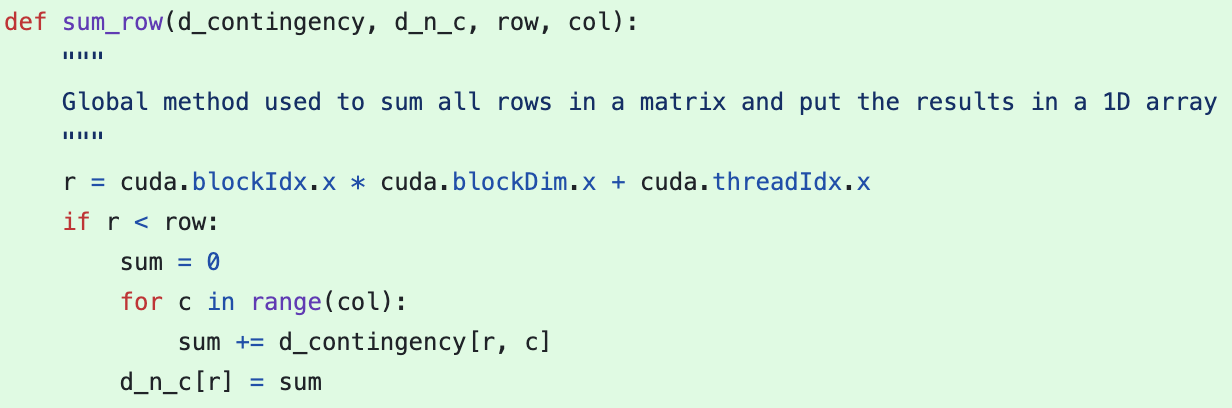
To parallelize the line in the original implementation which sums each row of the matrix contingency and stores the results in a 1D array using numpy, we first initialized an empty array n\_c with the length equivalent to the number of matrix rows (row) on the host. Then, we transferred this array, together with contingency, to the GPU’s device memory and launched kernels with (row + block\_size - 1) // block\_size blocks and block\_size = 128 threads per block. The implementation for the global method is shown in Figure 9.

Figure 9: Implementation for the global method sum\_row()

In the global method sum\_row(), we computed a global thread index, r , looped over all columns in the row corresponding to the index r , and took the sum. We eventually wrote each row’s sum into the corresponding index of d\_n\_c.

After the kernel launch, we called deviceSynchronize() to let the CPU synchronize all kernels and then transferred the results stored in d\_n\_c into n\_c.

### 2.2 Calculate the sum of each column in a matrix

This part is identical to the former part, but with the row and column swapped.

### 2.3 Calculate the sum of a squared matrix

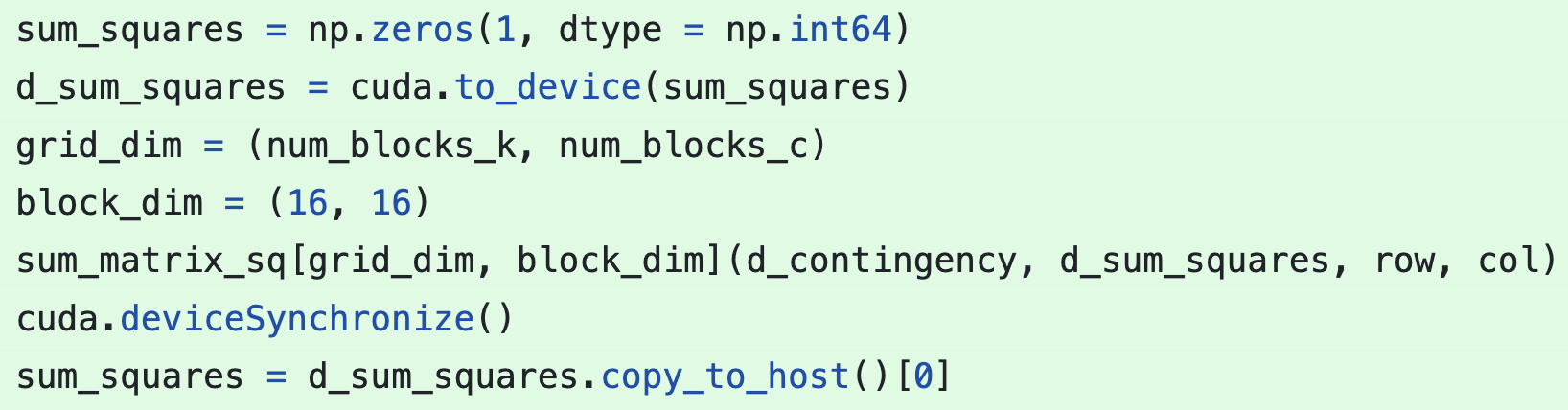
Before: After:

Figure 10: Parallelized Implementation Part 3

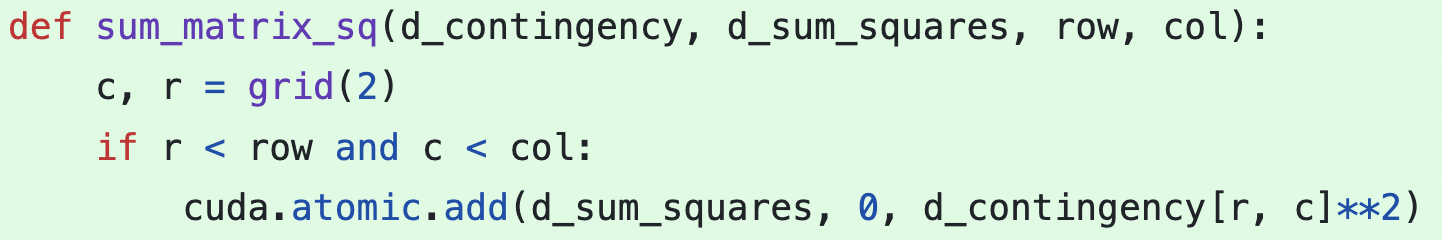
To run the original code that squares and sums the entire contingency matrix in parallel, we first created an array sum\_squares of length one and copied it to the GPU. The reason why we created an array rather than an integer was because of the syntax of atomic.add(), a built-in function that was later used in the global method. We then launched kernels with a block dimension of (16, 16) and a grid dimension of (num\_blocks\_k, num\_blocks\_c), where num\_blocks\_k = (col + block\_size - 1) // block\_size and num\_blocks\_c = (row + block\_size - 1) // block\_size. The implementation for the global method is demonstrated in Figure 11.

Figure 11: Implementation for the global method sum\_matrix\_sq()

In the global method sum\_matrix\_sq(), we obtained two global thread indices, c as well as r , and called atomic.add(), which squares the element at row r and column c of the d\_contingency matrix and adds it to the d\_sum\_squares array at index 0 .

After the kernel launch, we called deviceSynchronize() and transferred the final sum in d\_sum\_squares into sum\_squares.

### 2.4 Calculate the dot product of a matrix and an array

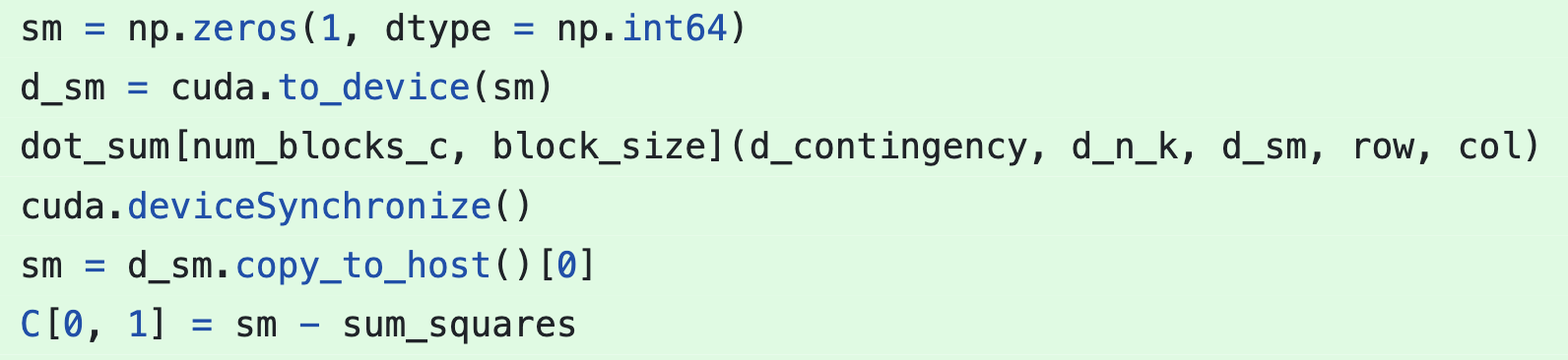
Before: After:

Figure 12: Parallelized Implementation Part 4

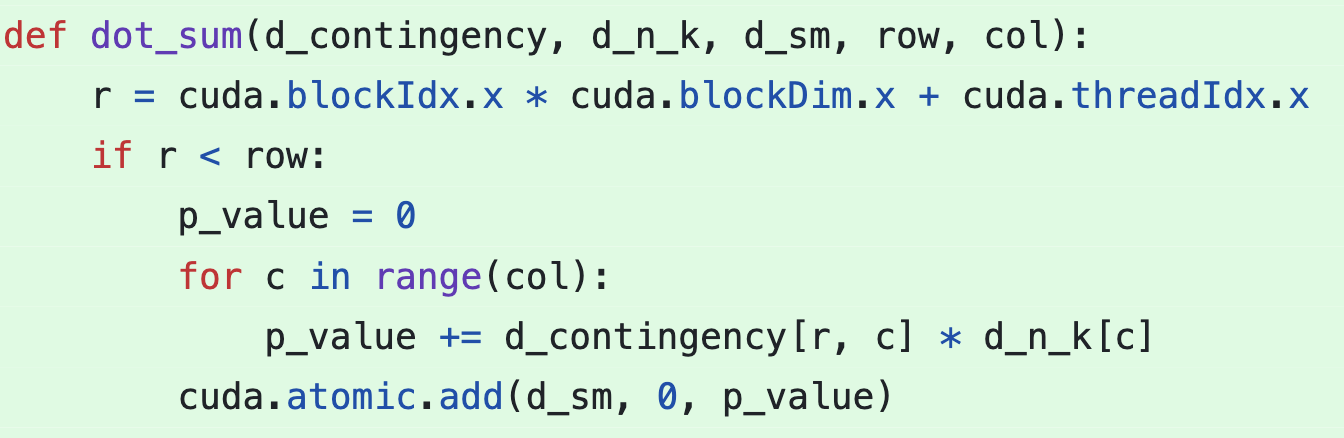
To parallelize the code that computes and then sums the dot product between the contingency matrix and n\_k, the array that stores the sum of each column of the contingency matrix, we initialized an empty array sm of length one on the CPU and transferred it to the GPU. Next, we launched kernels with num\_blocks\_c = (row + block\_size - 1) // block\_size blocks and block\_size = 128 threads per block. The implementation for the global method is shown in Figure 13.

Figure 13: Implementation for the global method dot\_sum()

In the global method dot\_sum(), we obtained a global thread index r and computed the dot product between row r of the d\_contingency matrix and the d\_n\_k array. Eventually, we took the sum of all such dot products using atomic.add() and stored them at d\_sm[0].

After launching kernels and calling deviceSynchronize(), we copied the dot product sum at d\_sm[0] back to the host and computed the number of false positives like the original code did.

Results

The performance of the contingency matrix, confusion matrix, and adjusted Rand index (ARI) was evaluated across various dataset sizes and computational implementations (CPU vs. GPU). To better capture the differences in performance between the two implementations, we used randomly generated datasets ranging in size from 1,000 to 10,000,000 elements, as the Titanic dataset was not sufficiently large to reveal significant performance disparities. Execution times were measured in milliseconds for clearer visualization of runtime differences. The profiling results are summarized below.

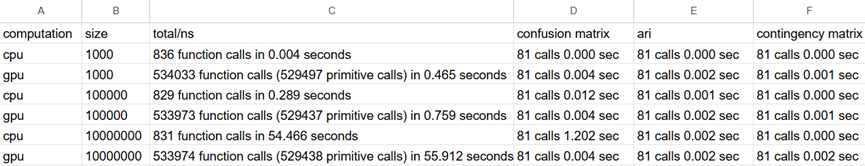


Figure 15: Profiling Result for CPU and GPU Computation

Contingency Matrix

**CPU Performance**: The CPU consistently outperformed the GPU across all dataset sizes for the contingency matrix function. The CPU implementation exhibited negligible execution times across all datasets (including the smallest size of 1,000 elements). Notably, the runtime of the contingency matrix function is not explicitly shown in the profiling figure for all three datasets. Based on this omission and the fact that other methods in the table report execution times of 0.000 seconds, we infer that the CPU execution time is faster than the methods displayed. As a result, we estimate the execution time to be approximately 0.000 seconds. We guess that the near-zero execution time is likely due to the function's low computational complexity or a lack of recorded calls during the profiling period.

**GPU Performance**: In contrast, the GPU implementation exhibited relatively slower execution times compared to the CPU. However, the GPU's execution time remained consistent across all dataset sizes, with a per-call time of approximately 0.012–0.025 ms. This slight delay is likely attributed to the overhead associated with the cuda.atomic.add() function and memory allocation processes, which may require additional time to execute.

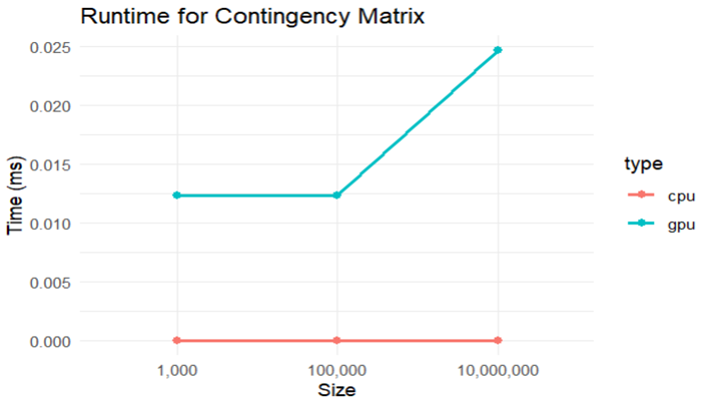


Figure 16: Runtime of Contingency Matrix

Confusion Matrix

**CPU Performance:** The CPU implementation outperformed the GPU for small datasets, demonstrating faster execution times at sizes like 1,000 elements. However, as the dataset size increased, the CPU's execution time grew significantly, highlighting its limitations with scalability. For the largest dataset size tested (10,000,000 elements), the CPU's per-call execution time peaked at 14.84 ms, which suggests that the CPU struggles to handle the increased computational demands for larger datasets.

**GPU Performance:** The GPU implementation demonstrated low and consistent execution times across all dataset sizes. It maintained a nearly constant per-call execution time. Although the GPU was slower than the CPU for small datasets, it surpassed the CPU's performance at a dataset size of 100,000 elements. While the exact size at which the GPU begins to outperform the CPU was not determined, the results clearly highlight the GPU's superior scalability for larger datasets, making it a more effective choice for large-scale computations.

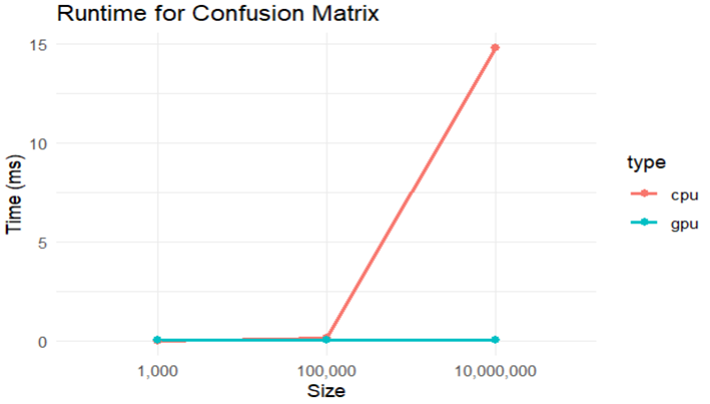


Figure 17: Runtime for Confusion Matrix

Adjusted Rand Index (ARI)

**CPU Performance:** The CPU implementation outperformed the GPU for small datasets (e.g., 1,000 elements). However, as the dataset size increased, the CPU runtime grew significantly, eventually matching the runtime of the GPU implementation for larger datasets. Interestingly, the per-call runtime of ARI was observed to be faster than that of the confusion matrix, despite the ARI computation calling the confusion matrix function with every execution.

**GPU Performance:** The GPU implementation demonstrated consistent and efficient performance across all dataset sizes, with an average per-call time ranging from 0.024 to 0.025 ms. This steady performance highlights the effectiveness of GPU parallelization in handling the ARI computation, regardless of dataset size.

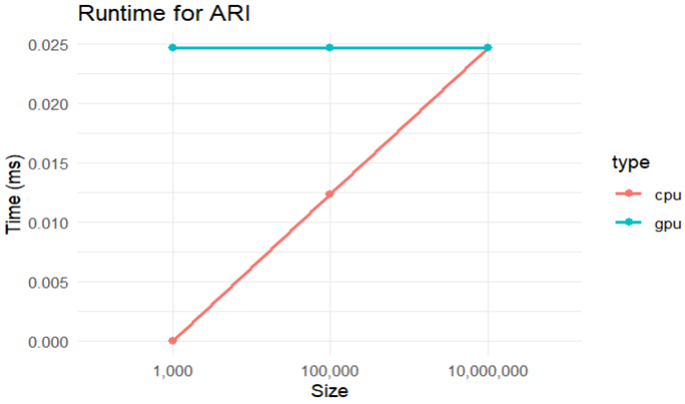


Figure 18: Runtime for ARI

Comparative Observations

The GPU demonstrated clear performance advantages for all three metrics, particularly for larger dataset sizes. While the CPU performed adequately for smaller datasets, its execution times increased exponentially with the dataset size for the confusion matrix and ARI computations. The profiling results highlight the suitability of GPU implementations for large-scale datasets, as they provided consistent performance across all metrics with minimal computational overhead.

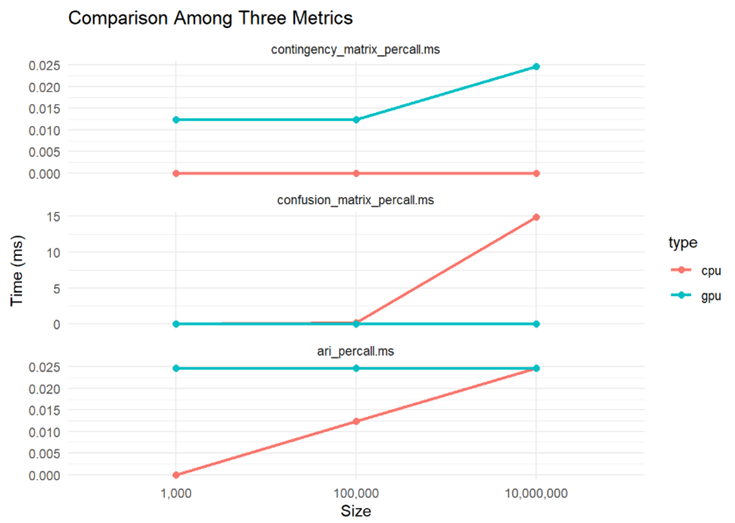


Figure 19: Comparison Among Three Metrics

# 

# 

# Future Possibilities

Using the Numba library, we successfully implemented a GPU-parallelized version of Clustermatch. However, due to memory exchange overhead, the CPU version remains faster for smaller datasets and get\_pair\_confusion\_matrix(). This allows us to alternate between the GPU and CPU versions to determine the optimal approach for a given dataset as a possible improvement for future versions.

Additionally, as discussed earlier, there is potential to enhance performance by further parallelizing the calls on the CPU instead of the GPU, which was a key focus of this course. This represents a possible area for future improvement.

# Conclusion

To put it in a nutshell, our group implemented a GPU version of the Clustermatch Correlation Coefficient (CCC) library by GPU parallelizing the matrix operation when calculating ARI, allowing greater efficiency when computing bigger data size.

# 

# 

# References

“Basics of CuPy — CuPy 13.3.0 Documentation.” \*CuPy – NumPy & SciPy for GPU — CuPy 13.3.0 Documentation\*, https://docs.cupy.dev/en/stable/user\_guide/basic.html. Accessed 15 Dec. 2024.

greenelab. “GitHub - Greenelab/Ccc.” \*GitHub\*, https://github.com/greenelab/ccc/tree/main.

Pividori, Milton, et al. “An Efficient, Not-Only-Linear Correlation Coefficient Based on Clustering.” \*Cell Systems\*, no. 9, Elsevier BV, Sept. 2024, pp. 854-868.e3. \*Crossref\*, doi:10.1016/j.cels.2024.08.005.

“Writing CUDA-Python — Numba 0.13.4 Documentation.” *Numba: A High Performance Python Compiler*, https://numba.pydata.org/numba-doc/0.13.4/CUDAJit.html#. Accessed 15 Dec. 2024.