LUT School of Engineering Sciences
Computer Vision and Pattern Recognition
BM40A1401 - GPU Computing
Responsible teachers: Prof. Lasse LENSU, and Dr. Henri PETROW

Machine Learning on GPU

Practical Assignment Report

Wednesday 2nd April, 2025
Nada Rahali - 000789952 - MLP
Tanjuma Haque - 002417756 - KNN

Contents

List of Figures

1 Introduction

The project tackles a classification problem by using two machine learning models namely *Multi-Layer Perceptron (MLP)* along with *k-Nearest Neighbors (kNN)* on *CPU* and *GPU* platforms. These models function to classify data samples through their features yet their performance will be measured on both *CPU* and *GPU* components.

The kNN model received implementation on both CPU and GPU platforms where GPU ran through a custom CUDA kernel which exploited parallelized computation. The *MLP* model received implementation through PyTorch before testing followed on both *CPU* and *GPU* to facilitate a valid comparison.

To analyze classification accuracy and runtime performance the evaluation of the *kNN* model included various k value tests. This report outlines the method behind designing the models, together with their training procedures, testing details and presentation of achieved results. This report includes discussions about preprocessing steps together with model accuracy assessments and time efficiency analyses supported by confusion matrices and additional depictions that show overall system performance.

2 Problem Statement

The goal of this project is to develop and evaluate classification models on a given dataset. The dataset consists of seven numerical features and a class label. The classification task is to predict the correct class label based on the feature values.

Two machine learning models are implemented: *k-Nearest Neighbors (kNN)* and a simple *Multilayer Perceptron (MLP)*. The *kNN* model is implemented from scratch using a custom *CUDA* kernel for *GPU* execution, while the *MLP* model is built using *PyTorch*. Both are evaluated across *CPU* and *GPU* to measure performance differences. The project focuses on realizing a high-performance implementation of the *kNN* method and show the advantage of using a *GPU*. It also focuses on realizing as-simple-as possible *MLP* to solve the classification problem well enough and compare the performances of *CPU* and *GPU* implementations.

2.1 Dataset Overview

The dataset utilized in this project is `MLoGPU_data3_train.csv`, which consists of the following:

- 4000 samples
- 7 numeric features
- Multiclass labels ranging from 0 to 6

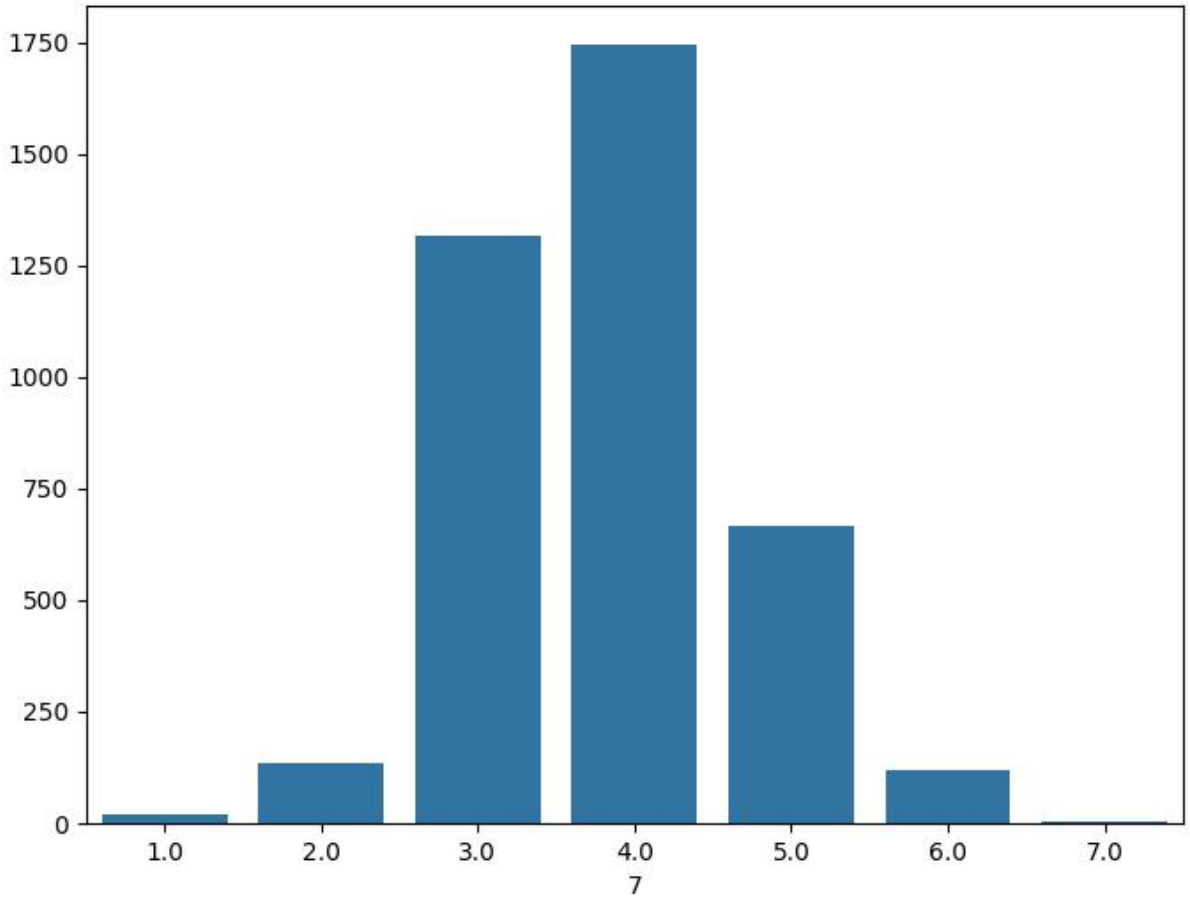

Figure 1: Class distribution

The dataset is heavily imbalanced, with classes 2 and 3 (3 and 4 on the graph) having the highest number of samples. Additionally, there is significant feature overlap between certain classes, which leads to confusion during classification. This challenge is reflected in the confusion matrices discussed later in the report.

2.2 Data Preprocessing Pipeline

To ensure optimal learning and fair comparison across models and hardware, the following preprocessing steps were applied. The dataset was not overly preprocessed, as the primary objective of this project is to compare *CPU* and *GPU* implementations rather than to fine-tune model performance.

1. **Feature Normalization using Min-Max Scaling**
    - Each feature was scaled to the range $[0, 1]$ using `MinMaxScaler`.
    - This was done to prevent features with large ranges from dominating the distance metrics used in algorithms like k-NN.
    - Min-Max scaling is especially suitable here because the dataset is numeric and non-zero, and keeping values within $[0, 1]$ ensures fairness in model behavior.
2. **Casting Labels to Integer**
    - The label column was explicitly cast to `int` type.
    - This prevents type mismatch issues during classification and GPU operations.
    - Both CuPy and PyTorch require labels to be in integer format for correct prediction handling.
3. **Train-Test Split using Stratified Sampling**
    - The dataset was split into 80% training and 20% testing using `train_test_split`.
    - `stratify=y` was used to ensure all classes are proportionally represented in both training and test sets.
    - `random_state=42` was used for reproducibility of results.
4. **Index Resetting**
    - After splitting, indices of the resulting data frames were reset.
    - This step ensures clean data structures and avoids potential mismatches during further processing.

3 k-Nearest Neighbours (k-NN)

3.1 Implementation

3.1.1 CPU Implementation

The CPU implementation was developed from scratch in Python, without using built-in `scikit-learn` classifiers. The implementation pattern was adapted from a solution to an assignment in the Pattern Recognition and Machine Learning course.

- **Distance Calculation:** The squared Euclidean distance formula $\sum(x_i - y_i)^2$ was used to compute distances from each test sample to all training samples. The square root was omitted since ranking by squared values yields the same nearest neighbors, thus saving computation time.
- **Prediction Logic:** After calculating all distances, the k smallest distances were selected. The corresponding training labels were then used to predict the test sample label via majority voting.
- **Tie-breaking:** If two or more classes had the same highest vote count, the function recursively called itself with $k - 1$ until the tie was broken or only one neighbor remained.
- **Parameters Used:**
  - `k`: Number of nearest neighbors (varied from 1 to 24)
  - `distance_metric`: Fixed to squared Euclidean

This implementation intentionally avoids external libraries such as `scikit-learn` in order to enable a fair, logical comparison to the GPU-based solution that uses equivalent logic.

3.1.2 GPU Implementation (CUDA Kernel)

The GPU version was implemented using a custom CUDA kernel named `compute_knn_predict`, written in C and executed using CuPy's `RawKernel` interface.

- **Threading Strategy:** One thread is launched per test sample, allowing each thread to independently calculate the predicted label in parallel.

- **Memory Management:**
  - Test sample data, distances, and indices are stored in local memory.
  - Constants such as `MAX_FEATURES`, `MAX_K`, and `MAX_CLASSES` are used to limit memory usage and maintain safe kernel execution within GPU hardware constraints.
- **Distance Calculation:** Each thread computes the squared Euclidean distance from its test sample to all training samples.
- **Top-k Sorting:** As distances are calculated, each thread maintains a sorted list of the k smallest distances using a manual insertion-based sorting technique.
- **Prediction:**
  - Once the k nearest neighbors are found, their class labels are tallied into counters.
  - The class with the highest count is selected as the final predicted label for the sample.
- **Parameters Used:**
  - `k`: Same as CPU version, varied from 1 to 24
  - `MAX_FEATURES = 7`: Number of input features in the dataset
  - `MAX_K = 30`: Chosen to allow flexible experimentation without exceeding memory limits
  - `MAX_CLASSES = 7`: Known number of unique class labels (0–6)

3.1.3 Design Rationale

- **Distance Metric Choice:** Squared Euclidean distance was selected instead of Manhattan distance due to the nature of the dataset. All features were continuous and normalized to the range $[0, 1]$, making Euclidean distance a more suitable measure of geometric similarity. Removing the square root further improved efficiency while preserving rank order.
- **GPU Design Philosophy:** A CUDA kernel was customized to give full manual control over memory and to exploit GPU parallelism across test samples.
- **Manual Top-k Sorting:** Instead of computing and storing all distances then sorting globally, a fixed-length insertion approach was used. This reduces memory usage and improves kernel performance.

Classification accuracy and runtime were measured, printed, and visualized for both CPU and GPU models to allow fair performance comparison.

3.2 `compute_knn_predict()` Cuda Kernel

A custom CUDA kernel named `compute_knn_predict` was implemented to utilize parallel computation in k-Nearest Neighbors (kNN) on the GPU. The kernel launches one GPU thread per test sample, allowing each thread to compute the predicted label for its allocated sample independently. This setup enables all test sample predictions to be performed in parallel.

Each thread first loads the feature values of its test sample into local memory. It then computes the squared Euclidean distance from the test sample to every training sample. During this process, it maintains a running list of the k smallest distances along with their corresponding training indices. The list is kept sorted through manual insertion as new distances are evaluated, enabling easier optimization and memory control.

Once the k nearest training samples are identified, the thread counts the frequency of each class label among them. The most frequently occurring label is selected as the predicted class for that test sample and is written to the output array.

This GPU-based approach exploits thread-level parallelism to classify multiple test samples simultaneously, leading to significantly faster runtime compared to a CPU-based solution. The kernel design also includes boundary checks and predefined memory limits (e.g., `MAX_FEATURES`, `MAX_K`, `MAX_CLASSES`) to ensure that it operates safely within the constraints of GPU hardware.

```
compute_knn_predict = cp.RawKernel(r'''
extern "C" __global__ void compute_knn_predict(
    const float *X_train, const float *X_test, const int *y_train,
    int *predicted_labels, int train_data_size, int test_data_size,
int num_features, int k)
{
    int test_index = blockIdx.x * blockDim.x + threadIdx.x; //getting
the index of the current thread (for one test sample)

    if (test_index >= test_data_size) return; //if the thread is out
of bound then exit

    //setting maximum limits for allocating memory
    #define MAX_FEATURES 7
    #define MAX_K 30
    #define MAX_CLASSES 7

    //if feature count or value of k exceeds the set limits, we exit
to avoid crashes
    if (num_features > MAX_FEATURES || k > MAX_K) return;

    //allocating local memory for storing the current test sample
    float test_sample[MAX_FEATURES];

    //arrays for tracking top-k distances and their associated
indices
    float top_k_distances[MAX_K];
    int top_k_indices[MAX_K];

    // as first step, we are loading test sample's features into
local memory
    for (int f = 0; f < num_features; f++) {
        test_sample[f] = X_test[test_index * num_features + f];
    }

    // initializing top-k array with values as max float value so any
lower distance can replace it
    for (int i = 0; i < k; i++) {
        top_k_distances[i] = 3.402823e+38;  // max float value
        top_k_indices[i] = -1;  //placing an invalid index to act as
a placeholder since no neighbour found yet
    }
```

Figure 2: `compute_knn_predict()` Kernel – Part 1

```
 // looping over all train samples for computing distances from test
sample and compute top_k values
    for (int train_index = 0; train_index < train_data_size;
train_index++) {

        float distance = 0.0; // for holding the sum of squared
differences
        //computing euclidean distances between current test sample
and training sample
        for (int f = 0; f < num_features; f++) {
            float diff = X_train[train_index * num_features + f] -
test_sample[f];
            distance += (diff * diff);
        }

        // if the distance is small enough, we insert the distance
and index in the top-k lists
        if (distance < top_k_distances[k - 1]) {
            int insert_position = k - 1; //starting from the last
position in the top-k list

            //shifting higher distances to the right to keep list
sorted to make the new insertion
            while (insert_position > 0 && distance <
top_k_distances[insert_position - 1]) {
                top_k_distances[insert_position] =
top_k_distances[insert_position - 1]; //moving larger distance to the
right by 1 step
                top_k_indices[insert_position] =
top_k_indices[insert_position - 1]; //moving the related index as
well similarly
                insert_position--; //moving left for continuing to
find where to insert
            }
            //inserting the new distance and its associated training
sample index in the correct position in the lists
            top_k_distances[insert_position] = distance;
            top_k_indices[insert_position] = train_index;
        }
    }
```

Figure 3: `compute_knn_predict()` Kernel – Part 2

```
 // counting all class labels from the top-k neighbours we found
    int class_votes[MAX_CLASSES] = {0}; //for counting frequency of
each class
    for (int i = 0; i < k; i++) {
        int label = y_train[top_k_indices[i]]; //get the label of the
i-th neighbour
        //making sure label is within range and then, incrementing
label's count
        if (label >= 0 && label < MAX_CLASSES) {
            class_votes[label]++;
        }
    }

    // finally, finding the label with the highest count for final
prediction

    // variables to keep track of the highest count and its label
    int max_votes = 0;
    int predicted_label = -1;

    //looping through for all classes to find the class with highest
count
    for (int c = 0; c < MAX_CLASSES; c++) {
        if (class_votes[c] > max_votes) {
            max_votes = class_votes[c];
            predicted_label = c;
        }
    }

    predicted_labels[test_index] = predicted_label; //saving the
predicted label for current test sample in the output array
}
''', 'compute_knn_predict')
```

Figure 4: `compute_knn_predict()` Kernel – Part 3

3.3 Results and Performance Analysis

1. Summary Statistics

The summary gives a quick overview of the most important numbers:

- **Best Accuracy (CPU and GPU)**: Both achieve the very same best accuracy of 0.52625 when $k = 1$. That is expected, since for $k = 1$, the algorithm is just choosing the nearest neighbor, and since both CPU and GPU are applying the same logic and data usage, the predictions will naturally be the same.
- **Average Accuracy**: Slightly better on GPU (0.5071) than on CPU (0.5069), and just enough to conclude that the GPU variant performs comparably in terms of quality.
- **Average Time Taken**: There is a significant distinction here – GPU only takes 0.0059 seconds, while the CPU takes 0.4995 seconds on average. The reason for this is that the GPU performs tasks in parallel, while the CPU processes the samples sequentially.
- **Average Speedup Factor**: GPU is 84.39x faster, a tremendous win. That's a strong reason to use the GPU for predictions when speed is a concern, especially for larger datasets.

The use of average accuracy and time across all k values is used to minimize the effect of outliers or fluctuations as much as possible, giving a more even and accurate view of the model's performance in general, on both GPU and CPU.

```
Best KNN Accuracy on GPU:  0.52625  with k =  1
Best KNN Accuracy on CPU:  0.52625  with k =  1

Time Performance Comparison:
GPU kNN Average Accuracy: 0.5071, Time: 0.0059 seconds
CPU kNN Average Accuracy: 0.5069, Time: 0.4995 seconds

 GPU Average Speedup Over CPU: 84.39x faster
```

Figure 5: `Summary results`

2. GPU Speedup Over CPU vs k

- Here, we observe how the speedup factor changes with values of k.
- The trend shows that GPU is significantly faster when k is small, even yielding speedups of over 170x.
- With increasing k, the speedup factor gradually reduces to approximately 30–40x. That is intuitive — for large k, there is more logic (like comparisons and sorting) per thread, so the GPU advantage becomes less.
- Nonetheless, even the smallest speedup is huge compared to CPU — GPU is still far more efficient.

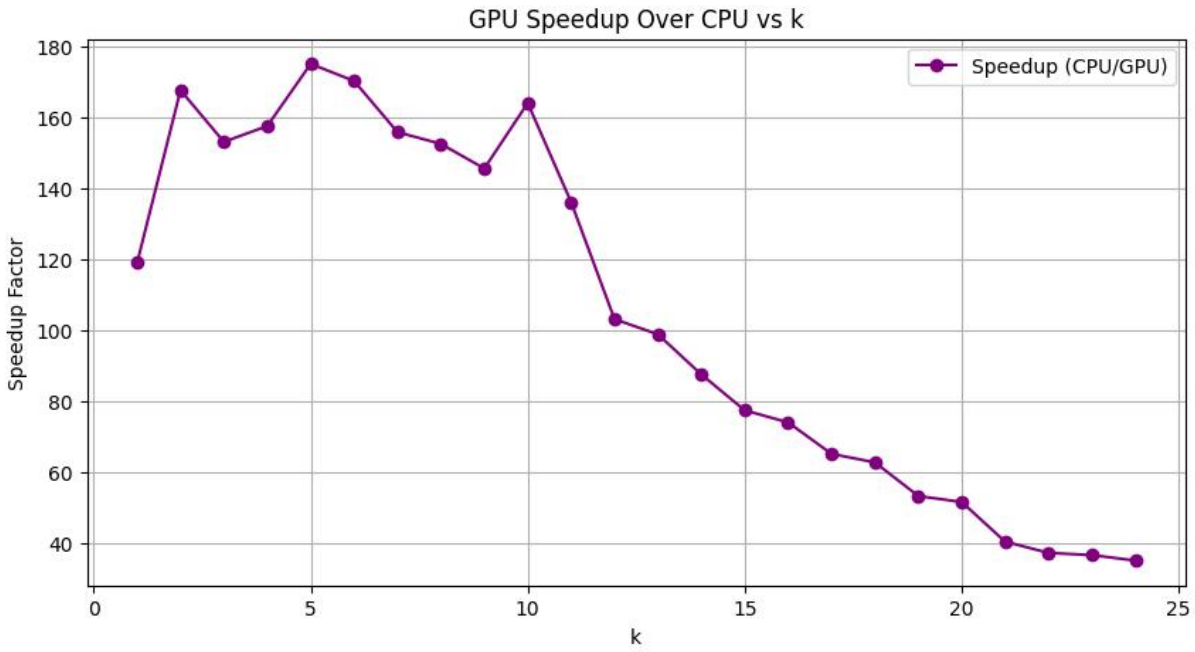

Figure 6: `GPU Speedup Over CPU vs` k

3. Computation Time vs k

- This plot indicates real times in seconds for GPU and CPU.
- GPU time remains almost flat across all k values, staying around 0.005–0.01 seconds.
- CPU time increases moderately with k, portraying that CPU is impacted more by how complex the k loop and distance calculations are.
- This GPU flatness even demonstrates how effectively parallelism accommodates additional work — the value of k isn't weighing it down as it does with the CPU.

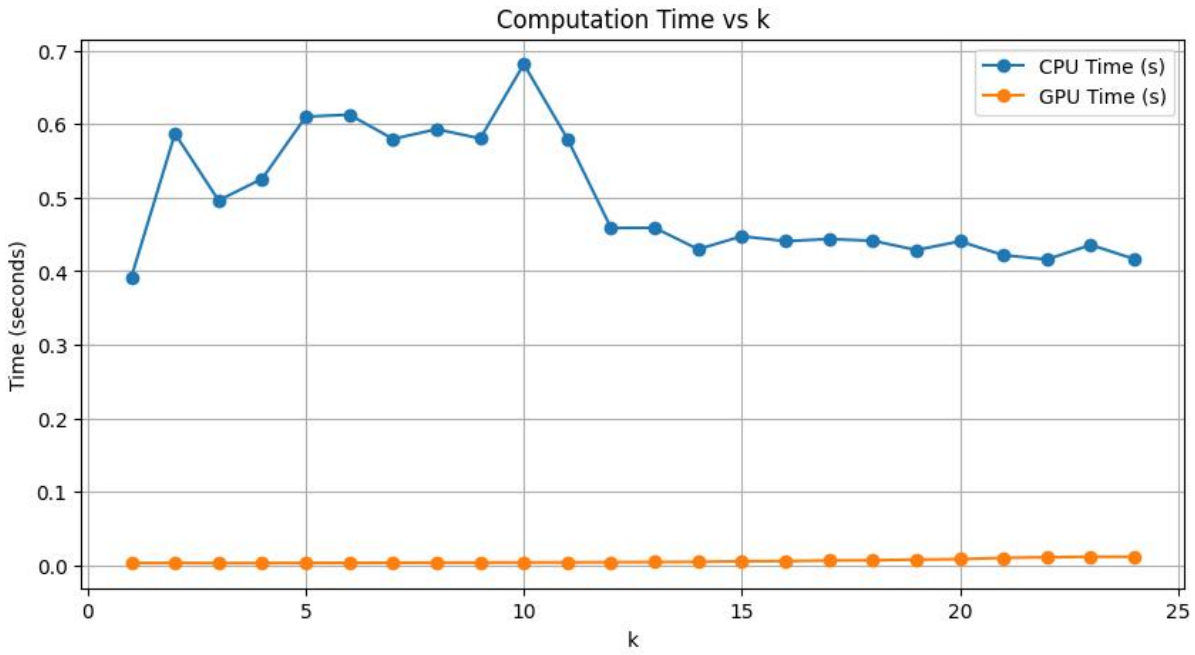

Figure 7: `Computation Time vs` k

4. Confusion Matrices

- The two matrices are almost the same, which is more reasonable since the predictions are almost the same due to the same accuracy.
- The models perform best on classes 2 and 3, which also happened to be the largest classes in the data set.
- There is a lot of misclassification between class 2, 3, and 4, and this suggests little overlap between features — especially because classes tend to confuse with each other.
- Very few of the samples of the minority classes (e.g., class 0, 1, 6) are being predicted correctly — some get zero predictions at all. This is an indication of the class imbalance in the dataset.

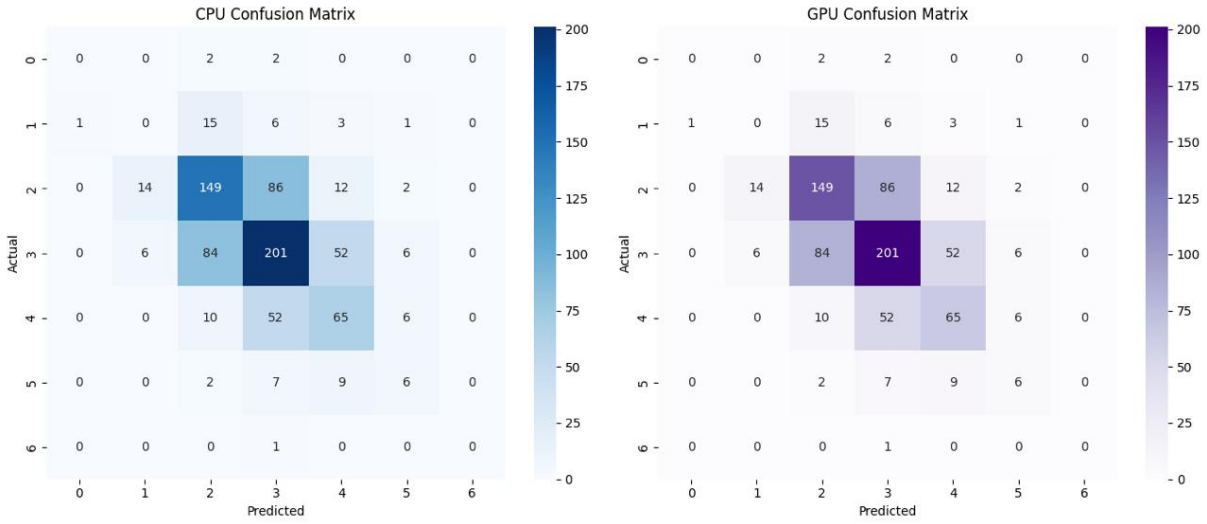

Figure 8: `KNN Confusion Matrices`

5. Classification Report

- Precision, recall, and F1-scores are all the highest for classes 3 and 4 — the majority classes. That's in alignment with the confusion matrix.
- Zero recall and precision for certain classes (1, 6, 7), which means they were never predicted correctly.
- Weighted average accuracy is approximately 0.52, which is not great but understandable considering the imbalanced dataset and similarity in features.
- Finally, the report is identical for both CPU and GPU — this is added evidence that the GPU kernel is logically accurate and functionally the same as the CPU version.

	CPU				GPU			\
	precision	recall	f1-score	support	precision	recall	f1-score	
1	0.000	0.000	0.000	4.000	0.000	0.000	0.000	
2	0.000	0.000	0.000	26.000	0.000	0.000	0.000	
3	0.569	0.567	0.568	263.000	0.569	0.567	0.568	
4	0.566	0.576	0.571	349.000	0.566	0.576	0.571	
5	0.461	0.489	0.474	133.000	0.461	0.489	0.474	
6	0.286	0.250	0.267	24.000	0.286	0.250	0.267	
7	0.000	0.000	0.000	1.000	0.000	0.000	0.000	
accuracy	0.526	0.526	0.526	0.526	0.526	0.526	0.526	
macro avg	0.269	0.269	0.269	800.000	0.269	0.269	0.269	
weighted avg	0.519	0.526	0.523	800.000	0.519	0.526	0.523	

Figure 9: `KNN Classification Report`

6. Accuracy vs k

- This shows how accuracy changes with different values of k.
- There is a huge fluctuation, especially in the first few values of k. The accuracy starts to settle down after around $k = 10$.
- Interestingly, $k = 1$ works best. Greater values of k in general reduce the impact of noise, but in this case, feature values of different classes overlap significantly, especially for dominant classes. Thus, when more neighbors are taken into account (larger k), it results in more uncertainty — primarily due to neighboring incorrect classes. So, accuracy becomes worse even with $k = 1$ instead of improving.

- The curves for CPU as well as GPU start saturating after $k = 15$, signifying the boundaries of model decision are not becoming better using more neighbors anymore. Instead, it's just smoothing over noising or vague points — so providing stabilized albeit suboptimal performance.

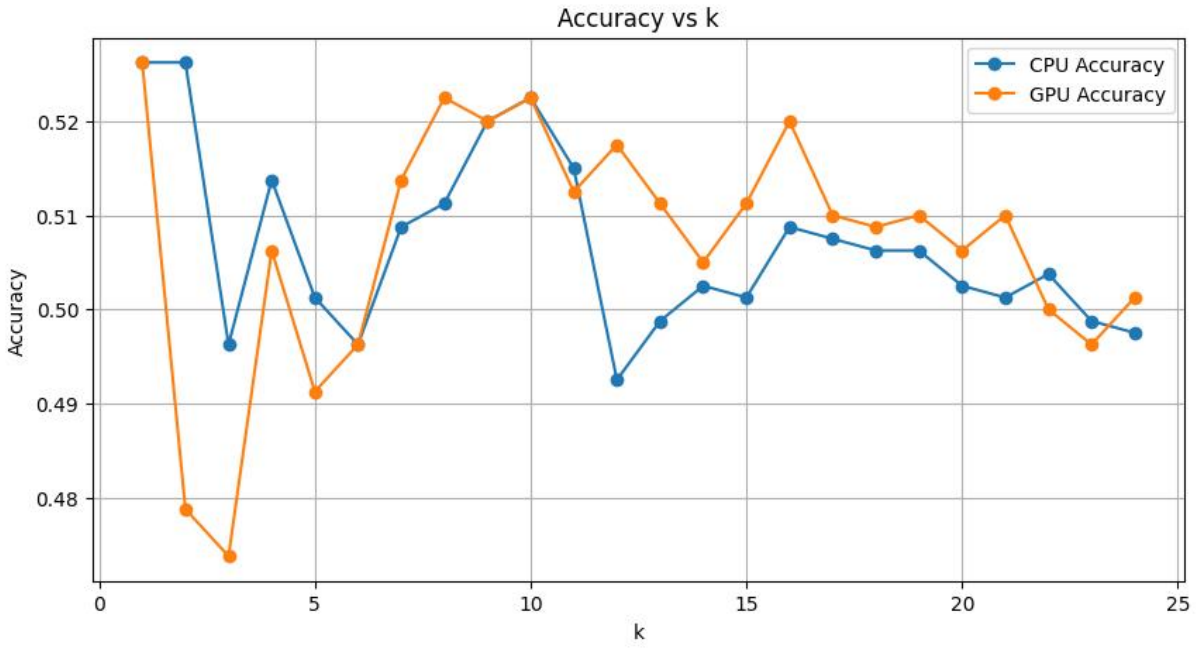

Figure 10: `Accuracy vs` k

Overall, both CPU and GPU versions of kNN produce the same results, which confirms the accuracy of the GPU kernel. The GPU does have a big performance gain, especially for small k, because of its parallel processing. While the classification accuracy remains limited due to class imbalance and feature distribution overlap, the project demonstrates how even a basic machine learning algorithm like kNN can be enhanced through GPU acceleration. Future work could explore better class imbalance management or experimentation on more separable datasets to further improve performance.

4 Multilayer Perceptron (MLP)

4.1 Implementation

4.1.1 Model Architecture and Parameters

The network consists of three linear layers:

- **Input Layer**: Accepts 7 normalized numerical features.
- **Hidden Layers**: Two hidden layers, each containing 64 neurons and using ReLU activation for non-linearity.
- **Output Layer**: Maps the 64-dimensional representation to 7 output neurons, corresponding to the number of target classes.

All weight matrices (`W1`, `W2`, `W3`) were initialized using a normal distribution scaled by 0.1, and biases (`b1`, `b2`, `b3`) were initialized to zeros. These parameters were defined as `torch.nn.Parameter` objects, making them compatible with PyTorch's optimization utilities.

4.1.2 Forward Pass

The forward propagation function performs the following sequence of operations:

1. Compute $x \cdot W1^T + b1$, apply ReLU.
2. Compute $x \cdot W2^T + b2$, apply ReLU.
3. Compute $x \cdot W3^T + b3$, return raw output logits.

4.1.3 Training Procedure

The training process was executed for 100 epochs using the Adam optimizer with a learning rate of 0.001. The `CrossEntropyLoss` function was used to measure classification error. For each batch in the training data:

- A forward pass was executed to get predictions.
- The loss was computed between predictions and ground truth.
- Gradients were calculated and used to update parameters via backpropagation.

The total training duration was recorded to compare CPU and GPU performance.

4.1.4 Evaluation Procedure

During evaluation:

- Gradient tracking was disabled with `torch.no_grad()` to reduce memory and computation overhead.
- The model output was computed for each batch and predictions were taken as the index of the maximum logit using `torch.max()`.
- Accuracy and inference time were calculated and reported.

4.1.5 Device Handling

The implementation automatically detects GPU availability using `torch.cuda.is_available()`. Based on the result, all model parameters and tensors are transferred to either CPU or CUDA device accordingly. This ensures consistent behavior and enables runtime and accuracy comparisons between devices.

4.1.6 Design Rationale

- **Manual Initialization:** The network parameters were explicitly defined instead of using built-in layers to allow full transparency over model structure and learning dynamics.
- **Three-Layer Structure:** A simple MLP architecture with two hidden layers was chosen to keep the implementation interpretable while still being deep enough to model class boundaries.
- **Choice of Hidden Units:** Each hidden layer has 64 neurons—a balanced choice between underfitting and overfitting, providing enough capacity for the 7-class classification task.
- **ReLU Activation:** ReLU was selected due to its simplicity, and proven effectiveness in deep learning models.
- **CrossEntropyLoss:** This is the standard loss function for multi-class classification tasks and is compatible with softmax output interpretation.
- **Adam Optimizer:** Selected for its adaptive learning rate and generally good convergence properties without the need for extensive tuning.
- **Direct Comparison Intent:** All architectural choices were aligned with the objective of comparing CPU and GPU runtime and performance under identical training conditions.

The goal of this implementation was to keep the structure simple and clear while making sure both CPU and GPU versions work the same way. This helps compare their performance fairly and understand how the training behaves on different devices.

4.2 Results and Performance Analysis

1. Summary Statistics The model reached the following test accuracies:

- CPU: 0.5225
- GPU: 0.5387

```
SUMMARY:
CPU Training Time:     9.9540 sec
GPU Training Time:     10.3894 sec
GPU is faster than CPU for training by: 0.96x

CPU Inference Time:    0.0101 sec
GPU Inference Time:    0.0118 sec
GPU is faster than CPU for inference by: 0.86x

Final Test Accuracy (CPU):  0.5225
Final Test Accuracy (GPU):  0.5387
```

Figure 11: `Summary Report`

The GPU model performed slightly better. Since the architecture and training steps were the same, the improvement is likely due to small differences in computation speed and stability when using GPU.

2. Timing Analysis

- **Training Time:**
  - CPU: 9.9540 seconds
  - GPU: 10.3894 seconds

  GPU was slower in training. This can happen when the model is small, and the overhead of moving data and running operations on GPU cancels out the benefit of parallel processing.

- **Inference Time:**
  - CPU: 0.0101 seconds
  - GPU: 0.0118 seconds

  Similar to training, *GPU* was slightly slower in inference too, for the same reason that the model is small and not demanding enough to show *GPU* speed gains.

3. Confusion Matrices

The confusion matrices from CPU and GPU runs look almost the same. This confirms that both models are working in a similar way.

- Class 3 and class 2 have the most correct predictions. These are the largest classes in the dataset.
- Small classes like 0, 1, 5, and 6 were rarely predicted correctly.
- There is a lot of confusion between classes 2, 3, and 4. This shows that their feature values might be overlapping.
- These observations are in alignment with those made with *kNN*.

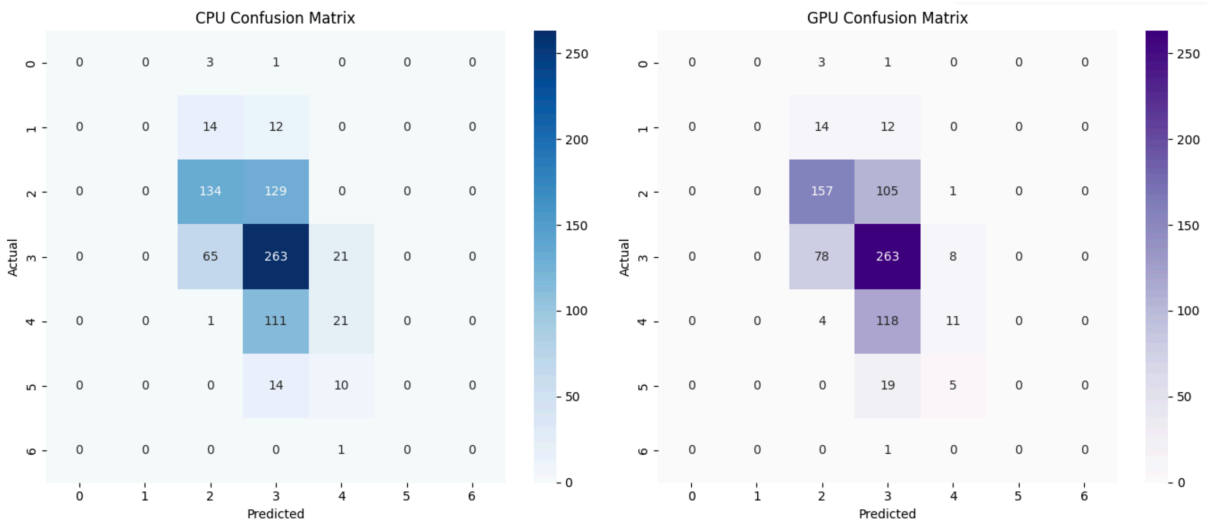

Figure 12: `MLP Confusion Matrices`

4. Classification Report

- Precision, recall, and F1-score are highest for classes 2 and 3.
- Some classes have 0 scores, showing the model did not predict them at all.
- Weighted average accuracy is around 0.52 for both devices, meaning the performance is similar.

	CPU				GPU			\
	precision	recall	f1-score	support	precision	recall	f1-score	
1	0.000	0.000	0.000	4.000	0.000	0.000	0.000	
2	0.000	0.000	0.000	26.000	0.000	0.000	0.000	
3	0.569	0.567	0.568	263.000	0.569	0.567	0.568	
4	0.566	0.576	0.571	349.000	0.566	0.576	0.571	
5	0.461	0.489	0.474	133.000	0.461	0.489	0.474	
6	0.286	0.250	0.267	24.000	0.286	0.250	0.267	
7	0.000	0.000	0.000	1.000	0.000	0.000	0.000	
accuracy	0.526	0.526	0.526	0.526	0.526	0.526	0.526	
macro avg	0.269	0.269	0.269	800.000	0.269	0.269	0.269	
weighted avg	0.519	0.526	0.523	800.000	0.519	0.526	0.523	

Figure 13: `MLP Classification Report`

Overall, both the CPU and GPU versions of the MLP model gave similar results in terms of accuracy. The GPU model reached slightly higher accuracy, but it was also a bit slower in training and inference. This is likely because the model is small, and the GPU overhead reduces the benefit of parallelism. The model performed well on the larger classes, but it struggled with the smaller ones. This happened because the dataset was imbalanced, and there was overlap between the features of some classes, which made it more difficult for the model to separate them correctly.

5 Discussion

5.1 Limitations

1. **GPU and Memory Management Constraints**

   - **Limitation:** The CUDA kernel uses fixed memory restrictions such as `MAX_K`, `MAX_CLASSES`, and `MAX_FEATURES` to ensure safe execution on the GPU. However, these hard-coded limits reduce the flexibility of the implementation. For example, if a higher k value or more output classes are needed, the kernel would have to be manually adjusted and recompiled.
   - **Effect:** These limits restrict the kernel's scalability when dealing with more complex or larger datasets. In such cases, higher neighbor counts or more output classes may be necessary for improved accuracy, which this implementation currently cannot handle without manual intervention.

2. **Training and Inference Time Comparisons**

   - **Limitation:** While GPU acceleration clearly improves the runtime for kNN predictions, the benefit for the MLP model is less pronounced due to the small size of the dataset and the relatively shallow model. On a larger dataset or a deeper MLP architecture, the GPU advantage would likely be more significant.
   - **Effect:** Although it is not a critical issue for this project, the current MLP setup does not fully showcase the GPU's potential. Future work involving more data or more complex models could better highlight the GPU's capabilities in accelerating MLP training and inference.

5.2 Challenges Faced

Some of the theoretical and technical issues encountered during the duration of this project are as follows:

- **Class Imbalance in the Dataset**
  The dataset was highly imbalanced, with some classes—particularly classes 3 and 4—dominating the label distribution. This made overall classification accuracy difficult to achieve, as the models skewed toward the majority classes and often ignored the under-represented ones. We saw this clearly in the confusion matrices and classification reports, where precision and recall for the minority classes were close to zero. This imbalance forced us to focus more intensely on other parts of the model design and evaluation.

- **Maximizing Accuracy Despite Overlapping Features**
  A large amount of feature overlap was present between several classes, especially 2, 3, and 4. This resulted in higher misclassification rates when using larger k values in KNN, as more neighbors ended up introducing noise instead of signal. Determining the optimal k value and understanding the impact of feature distribution required multiple rounds of testing and visual analysis.

- **GPU Optimization**
  To demonstrate GPU acceleration for KNN, we followed a step-by-step optimization strategy using a custom CUDA kernel. We began by offloading distance calculations, then added top-k sorting to minimize data transfer, and finally incorporated majority voting—completing a fully parallel prediction pipeline. At each stage, we tested runtime improvements. This progressive refinement significantly reduced runtime and effectively highlighted the GPU's performance advantage.

- **Forcing Fair Runtime Comparison**
  Another subtle challenge was ensuring that the runtime comparisons between CPU and GPU were valid. It's easy to accidentally time extra operations in one version but not the other. To avoid this, we were intentional about keeping the timed sections equivalent. For example, in `knn_gpu()`, we made sure the `y_pred` array conversion from CuPy to NumPy was done outside the function, just like in the CPU version. This helped isolate only the prediction time.

- **Report and Visualization Planning**
  With all the variations, graphs, and experiments we generated, one of the more subtle but important challenges was structuring our results in a clear, digestible way. We had to decide which plots provided the most insight—such as accuracy vs k, GPU speedup, and confusion matrices—and how to organize them to support our analysis without overwhelming the reader with redundancy.

6 Conclusion

In this final project, we succeeded in showcasing the usefulness of parallel computing in accelerating machine learning tasks, specifically k-nearest neighbors (kNN). For the multilayer perceptron (MLP), GPU performance was observed and compared with that of the CPU. The kNN implementation was developed progressively—starting with GPU-based distance computation and later integrating top-k sorting and majority voting into the custom CUDA kernel. This led to a fully parallel prediction pipeline and a significant increase in speed—exceeding 80x faster than the CPU version in some instances—while maintaining similar accuracy.

On the other hand, the MLP model, built manually in PyTorch using low-level modules, showed comparable inference and training times between GPU and CPU due to the small dataset and the relatively simple architecture. Nonetheless, this exercise provided valuable hands-on experience with GPU acceleration, memory management, and fair benchmarking practices.

As discussed earlier, we faced and resolved several challenges during the project, including class imbalance and the development of an optimized GPU-based kNN pipeline. Additionally, the manual implementation of the MLP offered a deeper understanding of the behavior, benefits, and limitations of both models. In summary, we were able to explore how the kNN pipeline can benefit from GPU optimization and how vital it is to make thoughtful design choices when both data and computational resources are limited.

References

[1] cppreference.com. Numeric limits - cppreference.com. https://en.cppreference.com/w/c/types/limits, 2025. Accessed: 2025-04-02.