CSC14120 – PARALLEL PROGRAMMING

FINAL PROJECT

1. Introduction

1.1 Problem Statement

Feature engineering is a fundamental challenge in machine learning: how do we automatically discover good representations of data that capture its underlying structure? In this project, you will implement an **Autoencoder-based unsupervised feature learning system** for image classification on the **CIFAR-10 dataset**.

Traditional supervised learning trains a model end-to-end using labelled examples. However, autoencoders take a different approach: they learn meaningful representations by attempting to reconstruct the input data itself, without requiring any labels during the feature learning phase. This unsupervised pre-training can discover features that are often more robust and generalizable than those learned through direct supervision alone.

Your task is to build and optimize a complete two-stage pipeline:

Stage 1 - Unsupervised Feature Learning:

- Train a convolutional autoencoder to reconstruct CIFAR-10 images
- The autoencoder learns to encode 32×32×3 images into an 8,192-dimensional feature representation that captures meaningful visual patterns
- No labels are used during this training phase
- The network learns features that capture important visual patterns (edges, textures, shapes)

Stage 2 - Supervised Classification:

- Extract features from the trained encoder for all images
- Train an SVM classifier on these learned features with class labels
- Evaluate classification performance on the test set

The primary focus is on **implementing and optimizing the autoencoder training and inference in CUDA**, while using existing libraries (LIBSVM) for the SVM classifier. Through systematic optimization, you will accelerate the autoencoder from taking hours on CPU to completing in seconds on GPU.

1.2 CIFAR-10 Dataset

CIFAR-10 is one of the most widely used benchmark datasets in computer vision and machine learning research. It provides a challenging yet computationally manageable image classification task.

Dataset Specifications:

- **Image size:** 32×32 pixels (RGB)
- **10 classes:** airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck
- **Training set:** 50,000 images (5,000 per class)
- **Test set:** 10,000 images (1,000 per class)
- **Total images:** 60,000
- **Format:** Binary files with uint8 pixel values

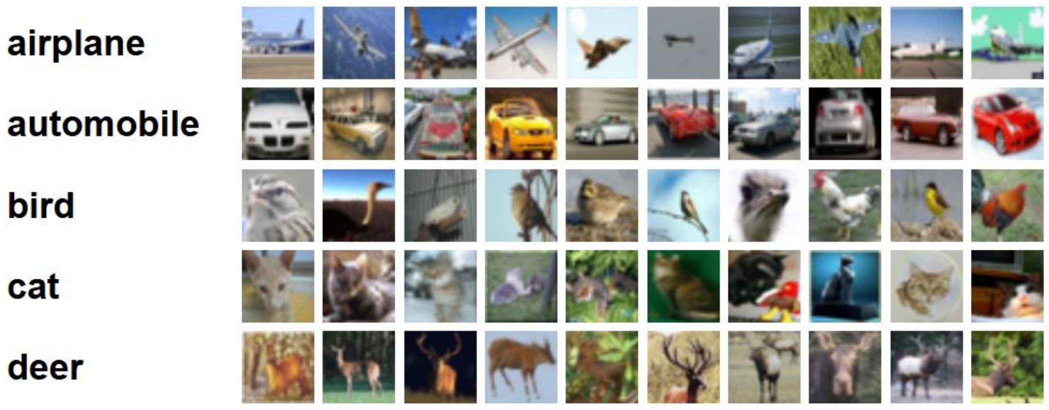

Figure 1 - Examples from CIFAR-10 Dataset

Dataset Organization for This Project:

- **Autoencoder training:** All 50,000 training images (labels ignored)
- **SVM training:** Same 50,000 images with labels (using extracted features)
- **Evaluation:** 10,000 test images with labels

Dataset Link: https://www.cs.toronto.edu/~kriz/cifar.html

1.3 Expected Performance Targets:

Metric	Target
Autoencoder training time	<10 minutes
Feature extraction time	<20 seconds for all 60K images
Test classification accuracy	60-65%

Metric	**Target**
GPU speedup over CPU	>20x

2. Background knowledge

2.1 Autoencoders for Unsupervised Feature Learning

What is an Autoencoder?

An autoencoder is a neural network trained to reconstruct its input, forcing the network to learn a compressed, meaningful representation in the process. It consists of two parts:

- **Encoder:** Compresses the input into a low-dimensional latent representation (feature vector)
- **Decoder:** Reconstructs the original input from the latent representation

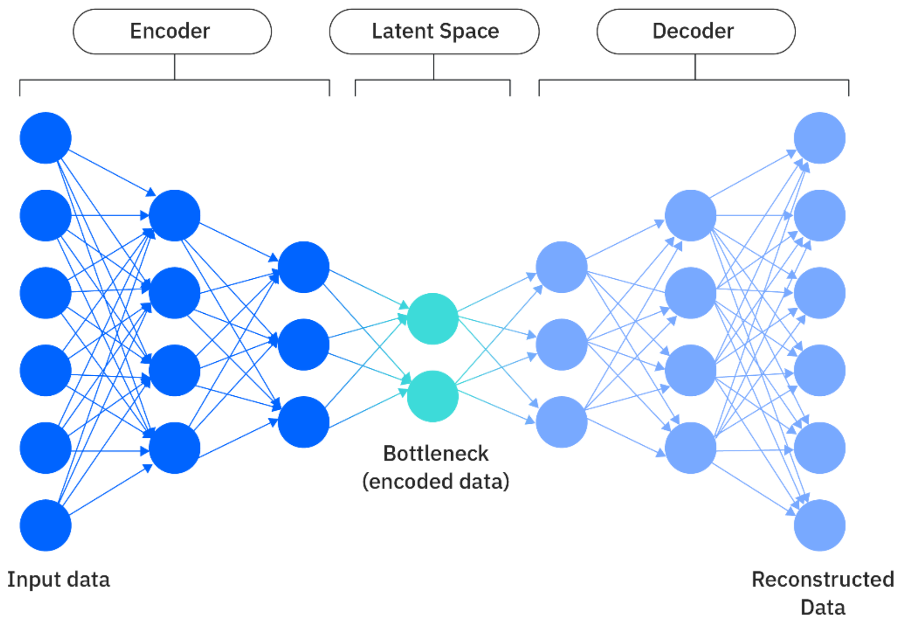

Figure 2 - The architecture of a standard autoencoder neural network.

Training Objective:

Loss = MSE(Input, Reconstructed_Output)

$= (1/N) * \Sigma(x - \text{decoder}(\text{encoder}(x)))^2$

The network learns to minimize reconstruction error, forcing the encoder to capture essential information about the input

Key Concepts:

1. **Bottleneck Layer (Latent Space):**
   - The smallest layer in the middle
   - Forces compression and feature extraction
   - In this project: (6, 6, 128) = 4,608 dimensions
2. **Symmetric Architecture:**
   - Decoder mirrors the encoder
   - Up sampling operations reverse the down sampling
   - Helps in reconstruction quality
3. **Feature Extraction:**
   - After training, we discard the decoder
   - Use only the encoder to extract features
   - These features feed into the SVM classifier

Recommended Reading Materials:

- **Foundational Papers:**

Hinton, G. E., & Salakhutdinov, R. R. (2006). "Reducing the Dimensionality of Data with Neural Networks." *Science*, 313(5786), 504-507.

Link: https://www.cs.toronto.edu/~hinton/science.pdf

- **Book Chapters:**

Goodfellow, I., Bengio, Y., & Courville, A. "Deep Learning" (2016)

Chapter 14: Autoencoders

Link: https://www.deeplearningbook.org/contents/autoencoders.html

- **Video guide**: Autoencoders and Representation Learning

Link: https://www.youtube.com/watch?v=R3DNKE3zKFk

- **Reference source code:**

Link: https://github.com/turkdogan/autoencoder

Link: https://github.com/tbennun/cudnn-training

- **CNN references:** Since your autoencoder uses convolutional layers, knowledge of CNNs is required

- ✓ This video "How Convolutional Neural Networks work" give a very good explanation with respect to CNNs. You can also check other good materials about Neural Network here from the same authors.
- ✓ This Standford cheatsheet give an excellent explain on how CNNs works
- ✓ This "Dive into deep learning" book also give through explain on CNN.

2.2 Support Vector Machine (SVM) for Classification

Your Role with SVM: You do NOT need to implement SVM from scratch. Use existing optimized libraries.

Recommended SVM Libraries:

1. **LIBSVM (Primary Recommendation)**
    - Most popular SVM library
    - Easy to use C/C++ interface
    - Supports multi-class classification
    - RBF kernel built-in
    - Link: https://www.csie.ntu.edu.tw/~cjlin/libsvm/
    - Documentation: https://github.com/cjlin1/libsvm
2. **LIBLINEAR** (Alternative for Linear SVM)
    - Faster for linear kernels
    - Good if you want to experiment
    - Link: https://www.csie.ntu.edu.tw/~cjlin/liblinear/
3. **ThunderSVM** (GPU-Accelerated)
    - GPU implementation of SVM
    - Optional: Could explore for extra credit
    - Link: https://github.com/Xtra-Computing/thundersvm

3. Project Pipeline

The project follows a clear 5-step pipeline:
Step 1: Load CIFAR-10 Images

- ✓ 50,000 training images (for autoencoder)
- ✓ 10,000 test images
- ✓ Preprocess: normalize to [0,1]

Step 2: Train Autoencoder (Your CUDA Code)

- ✓ Use all 50k images (unsupervised training)
- ✓ Ignore labels during autoencoder training
- ✓ CNN-based autoencoder architecture (see Section 4)
- ✓ Train to minimize reconstruction loss
- ✓ Save encoder weights

Step 3: Extract Features (Your CUDA Code)

- ✓ Load trained encoder weights
- ✓ Run encoder forward pass (no decoder)
- ✓ train_features: (50000, 8192)
- ✓ test_features: (10000, 8192)

Step 4: Train SVM (Library)

- ✓ Input: train_features + labels
- ✓ Kernel: RBF (Radial Basis Function)
- ✓ Hyperparameters: C=10, gamma=auto
- ✓ Output: trained SVM model
- ✓ *(Can use other SVM setup as you see fit)*

Step 5: Evaluate

- ✓ Predict on test_features using SVM
- ✓ Calculate accuracy, confusion matrix
- ✓ Expected accuracy: 60-65%
- ✓ Compare with baseline methods

Key Points:

- ✓ Your focus: Steps 2 and 3 (autoencoder implementation in CUDA)
- ✓ Using libraries: Step 4 (SVM training)
- ✓ Evaluation: Step 5 (measure success)

4. Network Architecture

4.1 Architecture Overview

INPUT: (32, 32, 3) ➔ ENCODER (compress) ➔ LATENT: (8, 8, 128) = 8,192 features ➔ DECODER (reconstruct) ➔ OUTPUT: (32, 32, 3)

4.2 Detailed Architecture Specification

ENCODER (Down sampling Path)

```
INPUT: (32, 32, 3)
    ↓
```

```
Conv2D(256 filters, 3×3 kernel, padding=1, stride=1) + ReLU
    → (32, 32, 256)
    ↓
MaxPool2D(2×2, stride=2)
    → (16, 16, 256)
    ↓
Conv2D(128 filters, 3×3 kernel, padding=1, stride=1) + ReLU
    → (16, 16, 128)
    ↓
MaxPool2D(2×2, stride=2)
    → (8, 8, 128)
    ↓
LATENT REPRESENTATION: (8, 8, 128) = 8192 dimensions
```

DECODER (Upsampling Path - Mirror of Encoder):

```
LATENT: (8, 8, 128)
    ↓
Conv2D(128 filters, 3×3 kernel, padding=1, stride=1) + ReLU
    → (8, 8, 128)
    ↓
UpSample2D(2×2) [Nearest neighbor or bilinear]
    → (16, 16, 128)
    ↓
Conv2D(256 filters, 3×3 kernel, padding=1, stride=1) + ReLU
    → (16, 16, 256)
    ↓
UpSample2D(2×2) [Nearest neighbor or bilinear]
    → (32, 32, 256)
    ↓
Conv2D(3 filters, 3×3 kernel, padding=1, stride=1) [No activation]
    → (32, 32, 3)
    ↓
OUTPUT: (32, 32, 3)
```

Or keras setup for reference:

```
input_size = (32, 32, 3)

input_image = Input(shape=input_size)

# Encoder
x = Conv2D(256, (3, 3), activation='relu', padding='same')(input_image)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same', name='encoded_layer')(x)

# Decoder
x = Conv2D(128, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
```

```
decoded = Conv2D(3, (3, 3), padding='same')(x)
```

Parameter summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 256)       7168
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 256)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 128)       295040
_________________________________________________________________
encoded_layer (MaxPooling2D) (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 128)         147584
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 256)       295168
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 32, 32, 3)         6915
=================================================================
Total params: 751,875
Trainable params: 751,875
Non-trainable params: 0
```

5. Implementation Guideline

You will implement this project in at least **4 progressive phases**, focusing on CUDA optimization for the autoencoder training and inference.

Phase 1: CPU Baseline & Data Pipeline

Objective: Set up the project infrastructure and create a working CPU baseline.

What to Implement:

1.1 Data Loading and Preprocessing

- Create a CIFAR10 Dataset class to handle data loading
- Read CIFAR-10 binary files (5 training batches + 1 test batch)
- Parse the binary format: 1 byte label + 3,072 bytes image per record
- Convert uint8 pixel values [0, 255] to float [0, 1] for normalization
- Implement batch generation for training

- Add data shuffling capability
- Organize train images (50,000), test images (10,000), and their labels in memory

1.2 CPU Neural Network Layers

Implement CPU versions of all necessary operations:

- **Convolution (Conv2D):** Apply 3×3 kernels with padding and stride
- **ReLU Activation:** Element-wise max(0, x) operation
- **Max Pooling:** 2×2 pooling to downsample by half
- **Upsampling:** Nearest neighbor interpolation to double spatial dimensions
- **MSE Loss:** Mean squared error between output and target for reconstruction

1.3 Autoencoder

- Create an Autoencoder class to encapsulate the network
- Allocate memory for all weight matrices and biases (5 conv layers total)
- Implement weight initialization
- Allocate memory for intermediate activations between layers
- Implement forward pass: sequentially apply encoder layers, then decoder layers
- Implement backward pass:
- Implement feature extraction method: run encoder only and return latent representation
- Add weight saving/loading functionality for model persistence

1.4 Training

- Set hyperparameters: batch size (32), epochs (20), learning rate (0.001)
- Loop over epochs and batches
- For each batch: forward pass → compute loss → backward pass → update weights
- Track and display training loss
- Measure time per epoch
- Save trained model weights after completion

Deliverables:

- Working data loading and preprocessing pipeline

- Complete CPU implementation of all layers
- Baseline performance measurements

Phase 2: Naive GPU Implementation

Objective: Port all operations to GPU with basic parallelization.

What to Implement:

2.1 GPU Memory Management

- Create GPUAutoencoder class for GPU operations
- Allocate device memory for all weight matrices
- Allocate device memory for all activation buffers
- Allocate device memory for gradients
- Implement functions to copy weights between host and device
- Implement proper memory cleanup (cudaFree)

2.2 Naive GPU Kernels

Design basic GPU kernels where each thread handles simple work:

Convolution Kernel:

- Each thread computes one output pixel
- Thread performs nested loops over kernel and input channels
- Uses global memory for all reads/writes
- Handles boundary conditions with padding

ReLU Kernel:

- Each thread processes one element
- Simple in-place operation: x = max(0, x)

MaxPooling Kernel:

- Each thread computes one output element
- Finds maximum value in 2×2 input window
- Writes result to output

Upsampling Kernel:

- Each thread computes one output pixel

- Maps output coordinates back to input coordinates (divide by 2)
- Copies input value to output (nearest neighbour)

MSE Loss Kernel:

- Parallel reduction to sum squared differences
- Use shared memory for partial sums
- Use atomicAdd for final accumulation

2.3 GPU Forward Pass

- Copy input batch from host to device
- Launch kernels sequentially for each layer
- Synchronize after kernel launches
- Copy final output back to host
- Compute loss value

2.4 GPU Backward Pass

Design gradient computation kernels:

- Implement gradient of each layer type (conv, relu, maxpool, upsample)
- Backpropagate errors from output to input
- Compute gradients with respect to weights
- Implement simple SGD weight update: weight -= learning_rate × gradient

2.5 GPU Training Loop

- Set larger batch size for GPU (64 instead of 32)
- For each batch: copy to device → forward → compute loss → backward → update
- Use GPU timer for accurate timing
- Display progress and loss values
- Save trained weights

Deliverables:

- All layers successfully ported to GPU
- Working GPU training loop

- Forward and backward passes implemented
- Correctness verification against CPU results

Phase 3: Advanced Optimization

Objective: Optimize memory access patterns and apply advanced techniques for maximum performance (May have several versions here)

Optimization Ideas (You don't need to implement all; this list is just a suggestion)

Category 1: Memory Optimization

1. Shared Memory Tiling for Convolution

Load input tiles cooperatively into shared memory to reduce global memory accesses. Each thread block processes a tile, reusing loaded data across multiple threads.

2. Convert to Matrix Multiplication

Transform convolution into matrix multiplication, then leverage optimized via matrix multiplication.

3. Memory Coalescing Optimization

Ensure threads in a warp access consecutive memory address for maximum bandwidth. Reorganize data access patterns and consider changing memory layout

4. Constant Memory for Small Weights

Store small, frequently accessed data (biases, small kernel weights) in constant memory for fast broadcast access. All threads in a warp can read the same value in a single instruction.

5. Pinned (Page-Locked) Memory

Use cudaMallocHost for pinned memory on CPU side to enable faster asynchronous transfers. Reduces H2D and D2H transfer time compared to pageable memory.

6. Unified Memory Management

Use CUDA Unified Memory to simplify memory management and allow on-demand data migration. System automatically handles data movement between CPU and GPU.

7. Memory Pool/Reuse Strategy

Reallocate all necessary buffers once and reuse them across batches/epochs. Eliminates repeated cudaMalloc/cudaFree overhead which can be significant.

Category 2: Kernel-Level Optimization

8. Kernel Fusion (Conv + ReLU + Bias)

Combine multiple operations into single kernel to eliminate intermediate global memory writes/reads. Reduces memory bandwidth requirements and improves arithmetic intensity.

9. Block-Level Fusion (Entire Encoder/Decoder)

Fuse entire encoder or decoder paths into single mega-kernel using shared memory for intermediates. Dramatically reduces global memory traffic but requires careful shared memory management.

10. Loop Unrolling

Unroll loops in convolution, pooling, and other kernels to reduce loop overhead. Compiler can often optimize better with unrolled loops, especially for small kernel sizes.

11. Vectorized Memory Access (float4)

Load/store 4 floats at once using float4 for better memory bandwidth utilization. Particularly effective when processing channels or batch dimensions.

12. Optimized Thread Block Dimensions

Tune block sizes (e.g., 16×16, 32×8) based on profiling to maximize occupancy and minimize wasted threads. Different layers may benefit from different configurations.

13. Mixed Precision Training (FP16/FP32)

Use FP16 for forward pass and most computations, FP32 for weight updates and accumulators. Reduces memory bandwidth.

Category 3: Parallelism & Concurrency

14. Gradient Checkpointing

Store only subset of activations during forward pass, recompute others during backward pass. Trades computation for memory, enabling larger batch sizes.

15. Multi-Stream Pipeline

Use multiple CUDA streams to overlap H2D transfer, computation, and D2H transfer. Can hide transfer latency by processing next batch while current batch trains.

16. Batched Operations

Process multiple images in parallel rather than one at a time. Amortizes kernel launch overhead and improves GPU occupancy.

Phase 4: SVM Integration and analysed result

Objective: Complete the pipeline and thoroughly evaluate performance.

6. Project Report

Report Format and Structure

Submission Format

Your project report must be submitted as a **Notebook (.ipynb)** that can run on **Google Colab**.

Section 1: Problem Description

What to Include:

1. **Problem Statement**
   - Clearly define the image classification task
   - Describe the motivation for GPU acceleration
2. **CIFAR-10 Dataset Overview**
   - Dataset specifications (size, classes, split)
   - Show sample images from each class (use visualization)
   - Explain data preprocessing steps (normalization, format)
3. **Autoencoder Architecture**
   - Describe the network architecture with a diagram
   - Specify layer dimensions and transformations
   - Explain the encoder-decoder structure and latent representation
   - Include architecture visualization
4. **Project Objectives**
   - Performance goals (training time, speedup targets, accuracy)
   - Technical learning objectives
   - Success criteria

Section 2: Implementation Phases

For each phase, provide a structured description following this template:

Phase 2.1: CPU Baseline Implementation

Objectives:

- What you aimed to achieve in this phase
- Why this phase is necessary

Implementation Details:

- **Data Pipeline:** How you loaded and pre-processed CIFAR-10 data
- **Layer Implementations:** Brief description of each layer (Conv2D, ReLU, MaxPool, Upsample)
- **Training Loop:** How you structured the training process
- **Key Code Snippets:** Show 2-3 critical functions (e.g., convolution function signature and main loop structure)

Results:

- Training time per epoch and total training time
- Final reconstruction loss
- Sample reconstructed images (show original vs reconstructed)
- Memory usage

Key Takeaways:

- What did you learn about the algorithm?
- What insights guided your GPU implementation?

Phase 2.2: GPU Basic Implementation

Objectives:

- Port CPU code to GPU with basic parallelization
- Verify correctness of GPU kernels
- Establish baseline GPU performance

Implementation Details:

- **Parallelization Strategy:** How you mapped operations to GPU threads
- **Kernel Designs:**
    - Convolution kernel: thread-to-output mapping
    - Pooling kernel: how threads handle 2×2 windows
    - Other kernels (ReLU, upsampling)

- **Memory Management:** Device memory allocation strategy
- **Key Code Snippets:** Show kernel signatures and launch configurations

Results:

- Training time per epoch and total training time
- Speedup over CPU baseline (include table and chart)
- GPU memory usage
- Verification that outputs match CPU (show error metrics)

Profiling Analysis:

- Basic profiling results (time spent in each kernel type)
- Memory bandwidth utilization (if measured)
- Initial bottleneck identification

Key Takeaways:

- What was surprisingly fast or slow?
- Where do you see optimization opportunities?

Phase 2.3: GPU Optimized Implementation - Version 1

Optimization Focus: (e.g., Memory Optimization)

Objectives:

- What specific optimization(s) you targeted
- Expected performance improvement

Implementation Details:

- **Optimization Technique(s) Applied:**
    - Detailed explanation of the optimization (e.g., shared memory tiling)
    - Why this optimization should help
    - Implementation approach
- **Key Code Snippets:** Show the optimized kernel or key changes

Results:

- Training time comparison with previous version

- Speedup over previous phase (incremental and cumulative)
- Performance metrics (bandwidth utilization, occupancy)
- Profiling comparison: before vs after

Analysis:

- Why did this optimization work (or not work as expected)?
- What did profiling reveal?
- What's the next bottleneck?

Key Takeaways:

- Lessons learned from this optimization
- Applicability to other problems

Phase 2.4: GPU Optimized Implementation - Version 2 (if applicable)

Optimization Focus: (e.g., Kernel Fusion and Advanced Techniques)

Follow the same structure as Version 1:

- Objectives
- Implementation Details
- Results
- Analysis
- Key Takeaways

Note: You may have multiple optimization versions. Create a separate subsection for each major optimization iteration.

Phase 2.5: SVM Integration

Objectives:

- Extract features using trained encoder
- Train SVM classifier on learned features
- Evaluate end-to-end classification performance

Implementation Details:

- **Feature Extraction:** How you extracted 1,024-dim features from encoder
- **LIBSVM Integration:** How you interfaced with LIBSVM
- **Hyperparameter Selection:** SVM parameters chosen (C, gamma, kernel)
- **Key Code Snippets:** Feature extraction and SVM training code

Results:

- Feature extraction time (50K train + 10K test)
- SVM training time
- Classification accuracy on test set
- Per-class accuracy breakdown (table)
- Confusion matrix (visualization)
- Comparison with baseline methods (if available)

Analysis:

- Which classes are easiest/hardest to classify?
- What does the confusion matrix reveal?
- How does accuracy compare to expectations?

Key Takeaways:

- Quality of learned features
- Effectiveness of two-stage approach

Section 3: Comprehensive Performance Analysis

3.1 Performance Comparison Across All Phases

Create comprehensive comparison including:

Table Format:

Phase	Training Time	Speedup (vs CPU)	Incremental Speedup	Memory Usage	Key Optimization
CPU Baseline	1800s	1.0×	-	-	-

Phase	**Training Time**	**Speedup (vs CPU)**	**Incremental Speedup**	**Memory Usage**	**Key Optimization**
GPU Basic	180s	10.0×	10.0×	2.1 GB	Parallelization
GPU Opt v1	45s	40.0×	4.0×	2.3 GB	Shared memory
GPU Opt v2	25s	72.0×	1.8×	2.5 GB	Kernel Fusion + Streams

Visualization Requirements:

- Bar chart showing training time across phases
- Line graph showing cumulative speedup

Section 4: Lessons Learned and Challenges Overcome

4.1 Key Technical Insights

What you have learn about: CUDA Programming, Deep Learning,Performance Optimization

4.2 Major Challenges and Solutions

Present 2-3 significant challenges using this format:

- ✓ Challenge 1: [Brief Title]
- ✓ Problem: One sentence describing the issue.
- ✓ Solution: 1-2 sentences explaining how you solved it.
- ✓ Lesson: One sentence on what you learned.

Section 5: Conclusion and Future Work

5.1 Project Summary

- Recap of what was accomplished
- Final performance metrics summary table
- Achievement of original objectives

5.2 Key Achievements

Highlight your best results:

- Maximum speedup achieved
- Classification accuracy

- Most successful optimization
- Technical skills mastered

5.3 Limitations

Honestly discuss:

- Current performance bottlenecks
- Accuracy limitations
- Implementation constraints

5.4 Future Improvements

7. Project Deliverable

Submission Requirements

You must submit the following items:

1. Team Plan and Work Distribution

- Document detailing each team member's responsibilities
- Task breakdown and timeline
- Contribution percentage for each member

2. Project Report

- **Jupyter Notebook (.ipynb)** as specified in Report Requirements
- Must be executable on Google Colab
- Export to PDF for archival purposes

3. Source Code Package

- **All source code files** (.cpp, .cu, .h, header files, etc.)
- **README.md** with:
  - Setup instructions (dependencies, CUDA version, libraries)
  - Compilation commands
  - Execution instructions with example commands
  - Hardware requirements (GPU model, memory)
  - Expected outputs

- **Trained model weights** (if file size permits, otherwise provide download link)

4. Presentation Video (15-20 minutes)

- **Upload to YouTube** with **Unlisted** visibility
- **Include link** in your notebook report and README file
- **DO NOT upload video to Moodle** (file size restrictions)

Video Content Should Cover:

- Problem overview and approach (2-3 min)
- Live demonstration of running code (5-7 min)
- Performance results and analysis (5-7 min)
- Key optimizations and lessons learned (3-5 min)