Image Processing Library for MNIST Digit Recognition

Naveen Meena

April 2024

Abstract: This project develops an image processing library for recognizing MNIST dataset digits, utilizing C++ and CUDA to balance performance and flexibility. It features the LeNet-5 CNN architecture, renowned for its computer vision efficacy, making it ideal for digit recognition tasks. The library includes essential operations like convolution, ReLU and tanh activations, and max and average pooling, alongside softmax and sigmoid for probability distributions. These operations, implemented in C++, are optimized through CUDA kernels to leverage GPU capabilities, significantly speeding up data processing during neural network training and inference. With pre-trained weights, the LeNet-5 model is evaluated on the MNIST test dataset, showcasing the library's effectiveness in digit classification. The project also employs CUDA streams to enhance throughput, demonstrating considerable efficiency gains by concurrently processing images.

1 Introduction

This project aims to enhance digit recognition in computer vision by developing an image processing library tailored for the MNIST dataset, which includes handwritten digits from 0 to 9. Central to this initiative is the implementation of the LeNet-5 architecture, a pioneering Convolutional Neural Network (CNN) by Yann LeCun et al., renowned for its effectiveness in deep learning applications for vision. LeNet-5's design, featuring convolutional, subsampling, and fully connected layers, is well-suited for identifying patterns within the MNIST dataset. By integrating the computational strengths of C++ and the parallel processing capabilities of CUDA, the project seeks to optimize performance for MNIST digit recognition. The selection of LeNet-5 is based on its historical significance and proven effectiveness in the field, aiming to leverage its architecture to advance current image recognition capabilities. This effort is not only about achieving high accuracy in digit classification but also about contributing to the evolution of image recognition technologies. Through detailed documentation of the development process, encountered challenges, and optimization strategies, the project intends to offer a comprehensive framework for future advancements in computer vision research and applications.

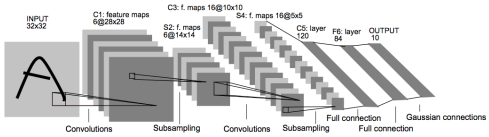

Figure 1: LeNet architecture

2 CUDA Kernels

The project adopts a GPU-accelerated approach to digit recognition using the CUDA programming model. CUDA allows for direct access to the GPU's virtual instruction set and parallel computational elements, making it ideal for deep learning applications.

Parallel Execution Strategy for Convolution

```
__global__ void conv_cuda(const float*
    input, const float* weights, float*
    output, int in_channel, int
    out_channel, int isize, int ksize) {
    // Kernel implementation
}
```

Block Configuration

We define our block dimensions as '(16, 16, 1)', creating blocks that house 256 threads. This arrangement is instrumental in facilitating each thread to independently calculate a distinct element of the output, thus, leveraging parallel computation effectively.

Grid Configuration

The grid is systematically arranged to mirror the dimensions of the output feature map, set as '(2, 2, 20)' in our example. This configuration encapsulates:

- Two blocks in the x-direction to span the output feature map's width,
- Two blocks in the y-direction for the output feature map's height,
- Twenty blocks in the z-direction, each processing one of the 20 output channels.

```
dim3 c1_block(16, 16, 1);
dim3 c1_grid((24 + c1_block.x - 1) /
    c1_block.x,(24 + c1_block.y - 1) /
    c1_block.y,20);
conv_cuda<<<c1_grid, c1_block>>>(c1_input
    , d_conv1, c1_output, 1, 20, 28, 5);
```

Parallel Execution Strategy for MaxPooling

The MaxPooling CUDA kernel leverages GPU parallelism to efficiently process pooling operations across CNN layers. Here we outline the strategic block and grid configurations that enable this:

```
__global__ void MaxPooling(const float*
    input, float* output, int in_channel,
    int isize, int ksize, int stride) {
    int res = (isize - ksize) / stride +
        1;
    int x = blockIdx.x * blockDim.x +
        threadIdx.x;
    int y = blockIdx.y * blockDim.y +
        threadIdx.y;
    int channel = blockIdx.z * blockDim.z
         + threadIdx.z;
    if (x < res && y < res && channel <
        in_channel) {
        // Pooling operation
    }
}
```

Output Dimensions Management

```
// Blocks configured for output width and
     height
dim3 p1_block(16, 16, 1); // Threads per
    block
```

Blocks are arranged in the x and y directions to cover the entire output feature map's width and height. This ensures that each thread is responsible for a unique portion of the output, maximizing the efficiency of the parallel computation.

Channel-wise Parallelism

```
// Grid configuration spans all input
    channels
dim3 p1_grid((res + p1_block.x - 1) /
    p1_block.x,(res + p1_block.y - 1) /
    p1_block.y,in_channel); // Ensuring
    parallelism across channels
```

The z-dimension in our grid configuration corresponds to the input channels, allowing the kernel to process multiple channels in parallel. This design ensures that each thread in a block can simultaneously work on different channels of the input data, significantly speeding up the pooling operation.

MaxPooling Kernel Execution

```
dim3 p1_grid((12 + p1_block.x - 1) /
    p1_block.x,(12 + p1_block.y - 1) /
    p1_block.y,20);
MaxPooling<<<p1_grid, p1_block>>>(
    c1_output, p1_output, 20, 24, 2, 2);
```

By invoking the 'MaxPooling' kernel with our carefully configured blocks and grids, we achieve an efficient parallel reduction of spatial dimensions while preserving critical features, a fundamental operation in CNNs for tasks such as image recognition.

Parallel Execution Strategy for Fully connected Convolution

```
__global__ void fconv_cudaR(const float*
    input, const float* weights,float*
    output, int in_channel, int
    out_channel,  int isize) {
    // Kernel execution strategy
}
```

```
int f1_block = 256; // Defines the number
     of threads per block
```

This configuration exploits the GPU's parallel execution capability, allowing us to process multiple neurons simultaneously.

Grid Configuration

To ensure that every neuron in the fully connected layer is processed, we calculate the required num-

ber of blocks in the grid based on the total number of neurons:

```
int f1_grid = (500 + f1_block - 1) /
    f1_block; // Determines the number of
    blocks needed
dim3 threads1(f1_block);
dim3 blocks1(f1_grid);
```

This grid configuration ensures that each neuron's output is computed in parallel, significantly reducing computation time.

Kernel Invocation

The kernel is invoked with the calculated configuration to perform the computation for the fully connected layer as demonstrated below:

```
int f1_block = 256; // Number of threads
    per block
int f1_grid = (500 + f1_block - 1) /
    f1_block; // Number of blocks in the
    grid
dim3 threads1(f1_block);
dim3 blocks1(f1_grid);
fconv_cudaR<<<blocks1, threads1>>>(
    p2_output, d_fc1, f1_output, 50, 500,
    4);
```