GPU-Accelerated Neural Network Library in CUDA C

Anh Vu
CSC-213: Operating Systems & Parallel Algorithms

Inspirations

- Worked with neural networks (NN) before but never messed around under the hood
- Parallelization with GPU has proven instrumental to creating fast NNs
    - Popular NN libraries all rely on cuDNN library by NVIDIA
- Parallelization with GPU seems very interesting from what we learned!

Project Goals

To create a library in CUDA C that:

- Allows for customizable NNs
- Utilizes GPU for speed when possible
- Has a simple and user-friendly API

Class Concepts

1. Parallelism with Threads
2. Thread Synchronization
3. File and File Systems

Concepts 1 & 2: Parallelism with Threads & Thread Synchronization

Performing *Single-precision GEneral Matrix Multiply* (SGEMMs)

- Time complexity is determined by all dimensions of both matrices
- Fits the *Single Instruction, Multiple Data* (SIMD) model
- Is used at multiple places in a neural network
    - Forward-propagation & Back-propagation

=> Parallelize!

```c
/**
 * Performs a matrix multiplication on the CPU
 *
 * @param A First matrix (m x p)
 * @param B Second matrix (p x n)
 * @param result Result matrix (m x n)
 */
void cpu__matrix_multiply(float *A, float *B, float *result, size_t rows_A,
                          size_t inner_dim, size_t cols_B) {
  // Select row in A
  for (int r = 0; r < rows_A; r++) {
    // Select col in B
    for (int c = 0; c < cols_B; c++) {
      // Combine each index in selected row and col
      float sum = 0.0f;
      for (int k = 0; k < inner_dim; k++) {
        sum += A[r * inner_dim + k] * B[k * cols_B + c];
      }

      // Store cell result
      result[r * cols_B + c] = sum;
    }
  }
}
```

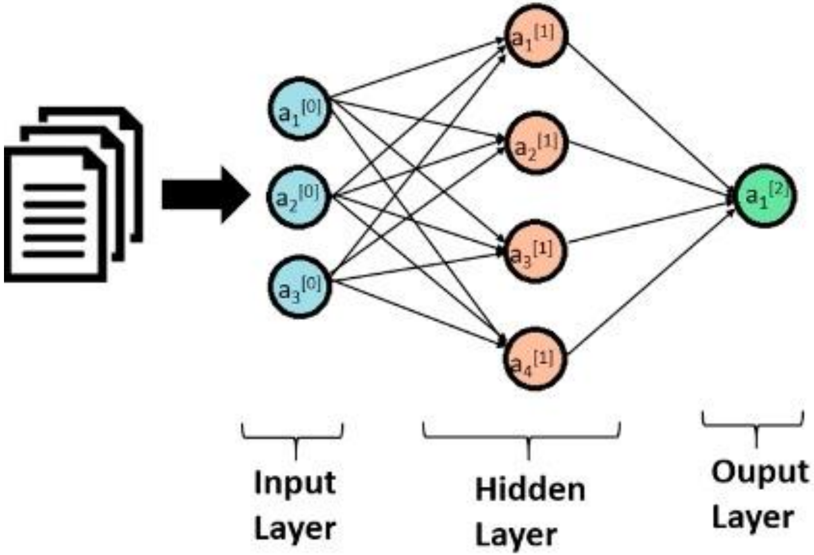
Naive CPU implementation:

$O(m \cdot n \cdot p)$

```
/**
 * Kernel for matrix multiplication
 *
 * @param A First matrix (m x p)
 * @param B Second matrix (p x n)
 * @param result Result matrix (m x n)
 */
__global__ void __kernel_matrix_multiply(float *A, float *B, float *result,
                                          size_t rows_A, size_t inner_dim,
                                          size_t cols_B) {
  // Get row and column of thread in result matrix
  size_t row = blockIdx.y * blockDim.y + threadIdx.y;
  size_t col = blockIdx.x * blockDim.x + threadIdx.x;

  // Check boundaries
  if (row >= rows_A || col >= cols_B)
    return;

  // Calculate cell
  float sum = 0.0f;
  for (int k = 0; k < inner_dim; k++) {
    sum += A[row * inner_dim + k] * B[k * cols_B + col];
  }

  // Store result
  result[row * cols_B + col] = sum;
}
```

GPU implementation:
$m \cdot n$ parallel threads of
$O(p)$

SGEMMs in Neural Networks



$a_1^{[1]}$ = activation_function($W_{11}^{[1]} * a_1^{[0]} + W_{12}^{[1]} * a_2^{[0]} + W_{13}^{[1]} * a_3^{[1]} + B1$)

$a_2^{[1]}$ = activation_function($W_{21}^{[1]} * a_1^{[1]} + W_{22}^{[1]} * a_2^{[1]} + W_{23}^{[1]} * a_3^{[1]} + B1$)

Concepts 3: File and File Systems

Saving and loading a network is an important feature of any NN library

- Saving your results to present somewhere
- Transfer learning

What needs to be stored?

- Network architecture
- Trained weights & biases

WIP!

```
// ===== Network Architecture =====
num_layers num_inputs num_outputs num_epochs learning_rate loss_func

// ===== Layer Architecture & Data =====
// Layer 1
num_neurons(n) prev_layer_dim(m) activation_func
w1_a w1_b w1_c … w1_m
w2_a w2_b w2_c … w2_m
…
wn_a wn_b wn_c … wn_m
b1 b2 b3 … bn

// Layer 2
…
…
…
```

DEMO

Limitations & Future Steps

- Math.
  - Taking derivatives of derivatives of derivatives is hard
  - Floating point precision error
- Library API
  - Allowing for a lot of customization means a lot of generalizable code
- Memory bottleneck
  - A huge bottleneck for GPU-related computations is the cost of copying and accessing memory from the CPU