Accelerating Softmax-Based Classification on MNIST Using CUDA

Sina Hakimzadeh
Raha Rahmanian

February 3, 2025

1 Project Description

1.1 Objective

The primary goal of this project is to implement and optimize the Softmax classification algorithm for the MNIST handwritten digits dataset using CUDA. By leveraging GPU parallelism, the project aims to efficiently compute the Softmax function, gradients, and parameter updates, significantly improving performance compared to traditional CPU-based implementations.

1.2 Background

Softmax classification is a fundamental method for multi-class classification tasks. It calculates the probability distribution over multiple classes and assigns the highest-probability class to a given input. The training process involves computationally intensive operations such as matrix multiplications, exponentiations, and gradient computations, making it an ideal candidate for GPU acceleration.

CUDA, with its massively parallel processing capabilities, provides an efficient way to optimize these computations. By implementing the Softmax classifier on a GPU, this project seeks to explore performance improvements in terms of both execution time and efficiency.

1.3 Scope of Work

1. **Data Preparation**
    - Load and preprocess the MNIST dataset into a flattened feature matrix.
    - Normalize input data to ensure numerical stability.
    - Transfer the dataset to GPU memory for efficient processing.
2. **Softmax Model Implementation**
    - Implement the forward pass to compute class probabilities.
    - Compute the loss function.
    - Implement gradient descent to update model parameters:
        - Compute gradients of the loss with respect to the weights.
        - Update weights using the computed gradients.
3. **Optimization**
    - Design and implement CUDA kernels for matrix multiplication and Softmax computation.
    - Experiment with different thread-block configurations to optimize performance.
4. **Evaluation**
    - Compare the accuracy and runtime of the GPU-based implementation against a CPU baseline.

1.4 Significance

This project provides hands-on experience with CUDA programming while demonstrating the efficiency gains of GPU acceleration in machine learning. By optimizing a fundamental algorithm like Softmax classification, the project highlights the advantages of parallel computing in deep learning applications.

2 Performance Comparison: C vs Python for Computational Tasks

2.1 Low-Level Memory Management in C

- **Direct Memory Access:** In C, you have direct control over memory allocation and access, which allows for more efficient handling of data. For example, when allocating memory with `malloc`, the program allocates only the exact amount of memory needed, and there's no overhead of interpreting data as in Python. This leads to faster access times and less overhead.
- **Cache Efficiency:** The C code is often more cache-efficient, as it explicitly manipulates memory and can better align data structures to the CPU cache. The code in C handles arrays in a more contiguous block format, reducing cache misses, which speeds up the computation.

2.2 Compiling vs. Interpreting

- **C Code Compilation:** C is a compiled language, meaning that your code is directly translated into machine code specific to the target architecture. This leads to much faster execution times because the CPU can execute the instructions directly.
- **Python Interpretation:** Python, on the other hand, is an interpreted language. Python code is first translated into bytecode and then interpreted by the Python interpreter. This adds overhead at runtime, making Python slower for compute-heavy tasks like matrix multiplications and gradient calculations.

2.3 Loop Optimization in C

- **Optimized Loops:** In the C code, loops like those for calculating the logits or softmax function are more efficiently compiled. The compiler can apply low-level optimizations like loop unrolling, which can make these operations run faster.
- **Python's Global Interpreter Lock (GIL):** Python's GIL prevents true parallel execution in multi-threaded scenarios. Even though the code may use multiple threads in Python, the GIL means that only one thread executes at a time, which can limit performance for CPU-bound tasks.

2.4 Vectorization and SIMD (Single Instruction, Multiple Data)

- **C with SIMD:** In C, it's easier to take advantage of CPU-specific optimizations like SIMD instructions, which allow the CPU to process multiple data points in parallel within a single instruction. Many C compilers (e.g., GCC) have optimizations that use SIMD operations automatically, making numerical calculations significantly faster.
- **Python with NumPy/Vectorization:** While Python's libraries like NumPy can use similar optimizations, they still rely on Python's interpreter to execute the logic, which introduces additional overhead. Even though NumPy is backed by C and can use vectorization, its performance often lags behind C due to Python's intrinsic overhead.

2.5 Data Structure Management

- **C Arrays:** The data structures in C are simple, static arrays that are straightforward to optimize and access. C compilers are better equipped to manage these efficiently.
- **Python Lists:** In Python, lists are dynamic arrays, which means they can grow and shrink in size. This flexibility introduces overhead and makes them slower when performing many iterations over large datasets. Python's lists are also more general-purpose, not optimized for the types of numerical operations you're performing.

2.6 Math Libraries

- **Math in C:** The C math library (`math.h`) is highly optimized for performance, and functions like `expf` and `logf` are implemented in low-level, highly efficient code that works directly with floating-point operations.

- **Math in Python:** Python's math library functions are not as optimized, and often they are wrappers around lower-level implementations (e.g., from C libraries). While libraries like NumPy and SciPy offer optimized implementations, the overhead of Python's interpreter still remains.

3 Comparing python implementation using NUMPY library with raw c implementation:

3.1 Low-Level Optimizations

- **Highly Optimized C Back-End:** Although we are using Python, NumPy operations are actually implemented in C. The core functionality (such as matrix operations, broadcasting, and linear algebra operations) is heavily optimized using C and other low-level libraries like BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage).
- **Memory Layout:** NumPy arrays are stored in contiguous blocks of memory, which allows efficient access and use of CPU caches, leading to better performance for operations on large datasets. This efficient memory access is similar to what you would get in a well-optimized C program but abstracted away from you.

3.2 Parallelism and Vectorization

- **Multithreading:** Many of NumPy's core operations are multithreaded. This means that NumPy can automatically use multiple CPU cores when performing operations like matrix multiplication or summation. This is often handled through libraries like OpenBLAS or Intel MKL (Math Kernel Library) that perform automatic parallelization, which is something you would need to explicitly manage in your C code.
- **SIMD Instructions:** NumPy leverages SIMD (Single Instruction, Multiple Data) instructions through optimized libraries, allowing the CPU to process multiple data points in parallel during mathematical operations. This makes NumPy operations faster than traditional loops in C, especially when performing element-wise operations like matrix multiplications, softmax, etc.

4 Vectorized Operations

- **NumPy Vectorization:** NumPy uses vectorization, which allows entire operations on arrays to be performed without explicit loops. For example, in your code, the `np.dot()` function for matrix multiplication and the `softmax()` function are all vectorized, meaning the operations are done on whole arrays at once. This is more efficient than writing explicit loops to handle each element one by one in C.
- **Broadcasting in NumPy:** Broadcasting in NumPy allows arrays of different shapes to be automatically aligned for mathematical operations, eliminating the need for extra looping logic.

4.1 Use of External Libraries

- **Optimized External Libraries:** NumPy often integrates with optimized external libraries like Intel MKL, OpenBLAS, or ATLAS, which offer highly optimized implementations of mathematical operations. These libraries can take advantage of CPU-specific optimizations such as cache optimization, parallelization, and SIMD instructions that are hard to implement by hand in C without external help.

4.2 Development Time vs. Efficiency

- **Efficiency vs. Development Time:** Writing efficient C code that takes advantage of all of the above features (like vectorization and multithreading) is difficult and time-consuming. With NumPy, you get all these optimizations automatically, which allows you to focus on implementing the higher-level logic. In contrast, in C, you would need to manually handle memory management, multithreading, and SIMD optimizations to reach a similar level of performance.

Program	Execution Time (Total)	User Time	System Time	CPU Usage	Epochs	Inference Results
C Code	35.13s	0.31s	73%	48.234%	10	Inference result0: 5, result1: 0, result2: 4, result3: 1
Normal Python	242.60s	1.81s	74%	5:26.86	1	Predicted label: 3, label: 0, label: 4, label: 1
Optimized Python (NumPy)	5.21s	1.08s	55%	11.295s	10	Predicted label: 5, label: 0, label: 4, label: 1

Table 1: Comparing Results: Execution Times and Inference Results for C Code, Normal Python, and Optimized Python (NumPy) Implementations

5 softmax algorithm for parallel computation on GPU

All traditional algorithms we are familiar with are designed to execute sequentially and in a specific order. However, CUDA and parallel programming require a different approach to achieve their goals, such as creating concurrency and optimizing performance through efficient coding techniques.

The algorithm we used is batch gradient descent, which trains a model by processing a batch of data at once. This approach has significant potential for parallel programming due to its ability to compute large amounts of data simultaneously, resulting in better occupancy. Additionally, batch gradient descent naturally helps to reduce noise in the training set, leading to more stable and accurate learning.

We store each flattened image in a two-dimensional matrix with a size of `BATCH_SIZE×IMG_SIZE`.

Batch Gradient Descent Algorithm:

- **Compute Logits:** To compute the logits for each image, we multiply each row of the image matrix by the corresponding row of the weights matrix, summing the results, and then store the value in the corresponding element of the logits matrix.
- **Compute Probabilities:** Using the softmax function, we evaluate the predicted class for each image. This step involves exponential calculations to determine the model's predicted class. A matrix is then created to store the probabilities for each class across all images.
- **Compute Gradient and Update Parameters:** Next, we use the gradient of the cross-entropy function to calculate the necessary changes for the weights and biases, as described in Formula 1.

  For the cross-entropy loss, L, with respect to the predicted probabilities $\hat{y}$ and the true labels y, the gradient with respect to the weights W and biases b is:

  $$L = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

  Where:

  - N is the number of samples,
  - y_i is the true label for sample i,
  - $\hat{y}_i$ is the predicted probability for sample i.

  The gradient with respect to the weights is:

  $$\frac{\partial L}{\partial W} = \frac{1}{N} X^T (\hat{y} - y)$$

  And the gradient with respect to the biases is:

  $$\frac{\partial L}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)$$

  Where:

  - X is the matrix of input features (size $N \times D$, where D is the number of features),
  - $\hat{y}$ is the vector of predicted probabilities (size $N \times C$, where C is the number of classes),
  - y is the vector of true labels (size $N \times C$).

5.1 Why it different in parallel

Due to the advantages of parallel programming and the computational power of CUDA, we must adopt an approach that efficiently utilizes matrices for operations. To achieve this goal, it is sometimes necessary to transpose matrices to ensure coalesced access to global memory, which enhances memory efficiency and overall performance.

6 Optimization Review

Initially, we execute our kernel, assuming it is optimized to some extent, and obtain results that evolve over time. The first set of results is as follows:

Program	**Execution Time (Total)**	**User Time**	**System Time**	**Epochs**	**Accuracy**
C Code	35.13s	0.31s	73%	10	N/A
CUDA Version 1	24.89s	20.45s	4.44s	10	81.2%

Table 2: Comparison of Execution Times and Accuracy for C Code and CUDA Version 1

The program is now much faster and more efficient for training a model. But by getting deep in data performance everything changed.

6.1 Memory

By analyzing the code more closely, it appears that for each `epoch`, we only need to copy a batch of data to the GPU and process it. This approach improves the program's ability to utilize streaming efficiently. However, when dealing with multiple `epochs`, we encounter a performance issue: each batch is copied multiple times and replaced with new data in every iteration. This redundancy significantly degrades the program's performance.

Program	**Execution Time (Total)**	**User Time**	**System Time**	**Epochs**	**Accuracy**
C Code	35.13s	0.31s	73%	10	N/A
CUDA Version 1	24.89s	20.45s	4.44s	10	81.2%
CUDA Version 2	19.89s	17.38s	2.51s	10	81.2%

Table 3: Comparison of Execution Times and Accuracy for C Code, CUDA Version 1, and CUDA Version 2

6.2 Thread block Configuration

The logits computation kernel utilizes `COMPUTE_Z_BLOCK_SIZE` to configure GPU thread blocks. When its value is set to 800, which is a multiple of 32 to fully utilize warp resources, the program achieves an accuracy of approximately 81.2%. However, increasing this value to 1024 improves accuracy to 84.09%. This indicates that the kernel reduction is influenced by the thread block size, impacting the overall model performance.

Program	**Execution Time (Total)**	**User Time**	**System Time**	**Epochs**	**Accuracy**
C Code	35.13s	0.31s	73%	10	N/A
CUDA Version 1	24.89s	20.45s	4.44s	10	81.2%
CUDA Version 2	19.79s	17.33s	2.51s	10	81.2%
CUDA Version 3	19.81s	17.38s	2.53s	10	84.09%

Table 4: Comparison of Execution Times and Accuracy for C Code, CUDA Version 1, CUDA Version 2, and CUDA Version 3

6.2.1 Why does it happen?

This issue occurs due to the nature of the reduction process. Reduction operates by repeatedly dividing by 2, which is only fully effective when applied to numbers that are powers of two. Since 800 is not a power of two, some data points are not correctly summed, leading to a decrease in accuracy.

6.3 Rolling or Unrolling

While using direct access to global memory, it is much more efficient to apply the unrolling technique for the reduction operation. This approach minimizes the overhead of checking loop conditions and reduces warp divergence. By unrolling the first 8 elements, we can maximize memory bandwidth utilization because the entire 32-byte data fetched into the L2 cache per thread block is fully utilized. Additionally, when the reduction reaches 32 elements or fewer, all threads operate within a single warp, eliminating the need for synchronization. However, is it beneficial to use unrolling when the entire dataset resides in shared memory? The answer is no, because when performing binary reduction, we use strides that are multiples of 32 for accessing data in L1 cache. This results in severe bank conflicts when accessing memory with a stride of 2, significantly degrading performance. Instead of improving efficiency, unrolling in this case can cause excessive contention in shared memory, leading to inefficient memory access patterns and reduced throughput. To overcome this issue, we need to adopt a more efficient approach—using a loop to access data in a way that avoids bank conflicts. By restructuring the memory access pattern, we can ensure that shared memory operations are efficiently executed without unnecessary contention.

Currently, the main bottleneck in the CUDA application is the `update_weights` function, which accounts for approximately 83% of the total GPU processing time. Addressing this issue and optimizing weight updates can significantly improve overall performance, leading to a more efficient execution of the program.

Program	Execution Time (Total)	User Time	System Time	Epochs	Accuracy
C Code	35.13s	0.31s	73%	10	N/A
CUDA Version 1	24.89s	20.45s	4.44s	10	81.2%
CUDA Version 2	19.79s	17.33s	2.51s	10	81.2%
CUDA Version 3	19.81s	17.38s	2.53s	10	84.09%
CUDA Version 4	10.623s	5.292s	5.277s	10	84.09

Table 5: Comparison of Execution Times and Accuracy for all versions

7 Getting More Accuracy

By adjusting certain parameters, the accuracy can improve significantly, as they directly impact model convergence and generalization.

7.1 Changing Batch Size

Batch Size	Accuracy (%)	Execution Time (Total)	User Time	System Time
16	86.48	39.109s	19.794s	19.263s
32	85.76	20.031s	10.109s	9.875s
64	84.09	10.623s	5.292s	5.277s
128	81.85	5.983s	2.982s	2.960s

Table 6: Batch Size vs. Accuracy and Execution Time on GPU using 10 epochs

By decreasing the batch size, the accuracy improves, but the execution time also increases. This happens because a smaller batch size leads to lower GPU occupancy, meaning fewer active threads are utilized at any given time. As a result, more computational resources remain idle, leading to inefficient hardware utilization and longer overall execution times. The improvement in accuracy occurs because a smaller batch size allows the model to train on more detailed variations within the data. With smaller batches, the model updates its weights more frequently, capturing finer details that might otherwise be lost in larger batch sizes. On the other hand, a large batch size tends to smooth out gradients and can act as a filter, potentially suppressing useful details and variations in the data. While this can lead to more stable training, it may also reduce the model's ability to generalize effectively, impacting overall accuracy.

7.2 Changing Learning Rate

Learning Rate	Accuracy (%)
0.01	75.12
0.05	81.79
0.1	84.09
0.2	85.70
0.5	86.72
1.0	86.44

Table 7: Effect of Learning Rate on Accuracy (Batch Size = 64, Epochs = 10)

The table indicates that increasing the learning rate to 0.5 leads to improved accuracy. This is because our data is normalized between 0 and 1, and a small learning rate slows down the weight updates, making convergence inefficient. When the learning rate is too low, the model struggles to make significant progress in optimizing the loss function, which negatively impacts accuracy. By increasing the learning rate, we allow the model to update its parameters more effectively, leading to better performance. However, an excessively high learning rate could cause instability, so finding the right balance is crucial.

7.3 Different Batch Size For One Training

I ran the program using two different batch sizes, 128 and 64, randomly switching between them during training. However, the results were not promising, and the model achieved an accuracy of only 70.01%. This suggests that inconsistent batch sizes may have disrupted the learning process, potentially affecting gradient updates and overall convergence stability.

7.4 More epochs

Epochs	Accuracy (%)	Execution Time (Total)	User Time	System Time
1	71.83	1.739s	0.881s	0.809s
5	85.62	5.646s	2.784s	2.825s
10	86.72	10.735s	5.365s	5.311s
30	86.33	30.674s	15.490s	15.141s
100	86.11	1m41.352s	51.348s	49.964s

Table 8: Effect of Epochs on Accuracy and Execution Time

The results in Table 8 show that increasing epochs improves accuracy but with diminishing returns. Accuracy jumps from 71.83% (1 epoch) to 86.72% (10 epochs), but further training beyond this point provides minimal gains, with 30 and 100 epochs showing slight declines, likely due to overfitting.

Execution time scales linearly with epochs, increasing from 1.739s (1 epoch) to 1m41.352s (100 epochs). This highlights a trade-off: while more epochs enhance learning initially, excessive training significantly increases computational cost with little accuracy gain. 10 epochs appear to be the optimal balance between performance and efficiency.