HPC PROJECT

IBRAHIM SHAHID 22i-0873
AHMED FASEEH JANJUA 22i-1259
RAJA SAIF 22i-1353

INTRODUCTION:
The neural network for MNIST classification's baseline implementation (V1) is fully written in C and runs on a CPU. CUDA was used to develop GPU-based versions (V2, V3, V4) in order to speed up the training and inference time. This section provides an analysis of the speedup attained, implementation details, and the tactics employed to improve performance.

V1 IMPLEMENTATION:
Implementation is entirely sequential, nestled loops are used to carry out feedforward and backpropagation on the CPU. During training, all 60,000 training samples are iterated across three epochs.
PROBLEMS:

- Lack of parallelism results in a lengthy training period.
- Ineffective for vectorized calculations.
- Only makes use of one CPU core; as a result, processor speed determines performance.

OBSERVED TIME:

```
bscs-22i-0873@FHPC:~/project/src/V1$ make
gcc -Wall -O2 -o nn.exe nn.c -lm
./nn.exe
MNIST Neural Network

Epoch 1 - Loss: 0.2671 - Train Accuracy: 91.90% - Time: 7.406s
Epoch 2 - Loss: 0.1065 - Train Accuracy: 96.81% - Time: 7.415s
Epoch 3 - Loss: 0.0749 - Train Accuracy: 97.82% - Time: 7.614s
Total training time: 22.435s
Test Accuracy: 96.85%
bscs-22i-0873@FHPC:~/project/src/V1$
```

V2 IMPLEMENTATION:
GOAL:
Utilize GPU cores by parallelizing the main bottlenecks in V1 with CUDA.

STRATEGIES USED:

- Parallelizing Feedforward & Backward Pass:
  In parallel, Forward Kernel 1 and Kernel 2 determine the output and hidden layer activations.
  A thread is associated with each neuron computation.
  __global__ CUDA kernels were used for:
  Dot product operations at the layer level.
  Softmax and ReLU activations.
- Parallel Backpropagation:
  To calculate gradients and update weights and biases, several CUDA kernels were developed.
  Updates such as bias correction and weight delta are carried out in various threads.
- GPU Memory Utilization:
  2D weight matrices that have been flattened for GPU memory efficiency.
  The cudaMalloc device memory is used for gradients, inputs, weights, and outputs.
  Data was sent between the host and the device using cudaMemcpy.

- Thread Configuration:
  Blocks and grids are configured to ensure maximum parallel execution per layer dimension

LIMITATIONS:

This implementation is still primitive; neither shared memory nor memory access optimization have been included. For async execution, there is no stream overlap or batch processing. The CPU is used to calculate Softmax (kernel launched with <<<1, 1>>>).

OBSERVED TIME:

```
nvcc -O2 -o nn_v2 nn.cu
./nn_v2
MNIST Neural Network - GPU Implementation (V2)

Using GPU: NVIDIA GeForce RTX 3080
Epoch 1 - Loss: 0.2694 - Train Accuracy: 91.85% - Time: 13.835s
Epoch 2 - Loss: 0.1064 - Train Accuracy: 96.87% - Time: 13.938s
Epoch 3 - Loss: 0.0731 - Train Accuracy: 97.89% - Time: 13.886s
Total training time: 41.660s
Test Accuracy: 96.70%
bscs-22i-0873@FHPC:~/project/src/V2$
```

V3 IMPLEMENTATION:

GOAL:

Address the main drawbacks of the GPU implementation (V2), particularly the absence of batch processing and asynchronous operations, to greatly increase performance. Implement adjustments to cut down on overhead and make greater use of the GPU's parallel architecture.

STRATEGIES USED:

- Batch Processing:
  Process several training samples at once with a single kernel launch by grouping them (`MAX_BATCH_SIZE = 256`).
- Asynchronous Execution (CUDA Streams):
  Use several CUDA streams ({NUM_STREAMS = 4`) to overlap kernel operations with data transfers (Host <-> Device).
- Optimized Batch Kernels:
  To work effectively with data batches, CUDA kernels (`forwardLayer`, `batchRelu`, `compute Error`, `updateWeights*`, `updateBiases`) were redesigned.
  All network parameters and intermediate batch data (inputs, activations, gradients) are stored on the GPU as contiguous 1D arrays (`d_W1`, `d_inputs`, etc.) using the flattened device memory model.
- Host-Based Softmax:
  After transferring pre-activation outputs from the GPU, compute the final Softmax activation on the CPU.
- Xavier Initialization:
  For possibly improved training convergence, use the Xavier/Glorot weight initialization.
- GPU Event Timers:
  To measure GPU execution time precisely, regardless of CPU overhead, use `cudaEvent`.

LIMITATIONS:

Host Softmax Overhead:

Latency is introduced and the asynchronous flow is briefly interrupted inside each batch by the transfer of pre-softmax outputs to the host, stream synchronization, CPU computation, and result transfer back to the device. A Softmax that is entirely GPU-based might be quicker.

Accuracy Variation:

V1/V2 (96%+) is higher than the achieved accuracy (92.70%). Different hyperparameters ({LEARNING_RATE`, `HIDDEN_SIZE`) and the altered training dynamics brought about by batching and initialization are the main causes of this. For the accuracy to be on par with or better than before, hyperparameter adjustment would be necessary.

Memory Optimizations:

While employing flattened arrays is a good idea, sophisticated methods such as utilizing shared memory within kernels to maximize data reuse—particularly in gradient calculations or matrix multiplications—are not used.

Kernel Fusion:

Although the present stream overlap probably mitigates much of this, it is possible to fuse some sequential kernel launches (such as layer calculation followed by activation) into single kernels to reduce launch overhead.

OBSERVED TIME:

```
Starting training (V3 modified for accuracy)...
Epoch 1 - Avg Loss: 0.6447 - Train Accuracy: 83.45% - Time: 0.120s
Epoch 2 - Avg Loss: 0.3330 - Train Accuracy: 90.62% - Time: 0.118s
Epoch 3 - Avg Loss: 0.2867 - Train Accuracy: 91.81% - Time: 0.118s
Total GPU processing time: 0.357s
Total training time (CPU): 0.357s

Evaluating model...
Test Accuracy: 92.70% (9270 / 10000)
GPU evaluation time: 0.018s
Total evaluation time (CPU): 0.018s

Cleaning up...
Done.
```

V4 IMPLEMENTATION:

GOAL:

Achieve the highest possible performance by leveraging specialized hardware units, specifically NVIDIA Tensor Cores, available on modern GPUs (Compute Capability 7.0+). This involves integrating the highly optimized cuBLAS library and employing a mixed-precision approach to maximize computational throughput beyond the custom kernel optimizations of V3.

STRATEGIES USED:

- Use cuBLAS to Take Advantage of Tensor Cores:
  Calls to cublasGemmEx were used in place of the custom CUDA kernels for matrix multiplication (forwardLayer*Kernel in V3). cublasGemmEx was set up to use float (FP32) precision computation/accumulation and half (FP16) precision inputs, specifically focusing on Tensor Core routes through the CUBLAS_TENSOR_OP_MATH mode.
- Implement Mixed-Precision Computing:
  In order to optimize Tensor Core throughput and minimize memory bandwidth, FP16 (half) is utilized for the primary weight matrices (d_W*_half) and activations (d_inputs_half, d_hiddens_half) supplied as input to cublasGemmEx.
  In order to preserve numerical range and accuracy during backpropagation, FP32 (float) is used to store intermediate activations post-bias/ReLU (d_hiddens_float), accumulate results within cublasGemmEx, and calculate and store gradients (d_d_outputs_float, d_d_hiddens_float).

  FP64 (double): Preserves greater accuracy during training iterations by being kept for the final weight updates and the master copies of the weights and biases (d_W*, d_b*). In order to maintain consistency with V3, it is also utilized to send pre/post-softmax values to/from the host for the host-based Softmax computation.
- Explicit Precision Conversion Kernels:
  In order to manage the required data type conversions between double, float, and half on the device GPU buffers at various stages of the computation, specific CUDA kernels

(doubleToHalfKernel, floatToHalfKernel, halfToFloatKernel, floatToDoubleKernel, etc.) were introduced.

- Modify Kernels for Varying Accuracy:
  To properly read from and write to the new float and half precision buffers, the current bias application, activation (ReLU), gradient computation (compute*Error*Kernel), and weight/bias update kernels (update*Kernel) were modified. This allowed them to still interact with the double precision master weights/biases or host-transfer buffers in a suitable manner.
- Retain V3 Optimizations:
  To increase GPU utilization, Batch Processing (MAX_BATCH_SIZE) was continued.
  While the host Softmax continues to serve as a synchronization point, asynchronous execution was maintained using numerous CUDA streams (NUM_STREAMS) to overlap computation and data transfers where feasible.
  For maybe improved convergence, Xavier/Glorot Initialization was maintained.
  For precise GPU-side timing measurement, GPU Event Timers (cudaEvent) were utilized.

LIMITATIONS:

Precision Conversion Overhead:

Although Tensor Cores greatly speed up GEMM operations, the additional kernels for FP16, FP32, and FP64 conversion impose a minor computational overhead that wasn't there in V3.

Memory Optimizations:

Memory access patterns rely on common global memory access optimizations; sophisticated methods such as the use of shared memory within custom kernels (especially for gradient accumulation or update stages, though less pertinent for cuBLAS GEMM itself) were not included.

OBSERVED TIME:

```
Starting training (V4 Tensor Core Enabled)...
Epoch 1: 51200 / 60000 samples processed.
Epoch 1 - Avg Loss: 0.4444 - Train Accuracy: 86.63% - Time: 0.118s
Epoch 2: 51200 / 60000 samples processed.
Epoch 2 - Avg Loss: 0.1515 - Train Accuracy: 95.43% - Time: 0.091s
Epoch 3: 51200 / 60000 samples processed.
Epoch 3 - Avg Loss: 0.1090 - Train Accuracy: 96.73% - Time: 0.090s

Total GPU processing time (measured by CUDA events): 0.307s
Total training wall time (CPU measured): 0.299s

Evaluating model on test data (V4)...

Test Accuracy: 96.41% (9641 / 10000)
GPU evaluation time (measured by CUDA events): 0.020s
Total evaluation wall time (CPU measured): 0.015s

Cleaning up host and device resources...
Freeing network resources...
Network resources freed.
Host data freed.

Execution Finished Successfully.
bscs-22i-0873@FHPC:~/project/src/V4$
```

SPEED UP ACHIEVED:

VERSION	TIME	SPEEDUP
V2	**41.660 sec**	**0.54x with V1**
V3	**0.357 sec**	**116.69x with V2**
V4	**0.307 sec**	**1.16x with V3**

GITHUB REPOSITORY LINK:
https://github.com/ibrahim10102002/Neural-network-acceleration.git