Project 4 Report

Compilation Instructions:

sh test.sh

The script compiles Softmax and Neural Network classifiers with and without OpenACC using both **g++** and **pgc++**, organizes the resulting executables in the "build" directory, removes temporary files, and potentially submits a batch job using "sbatch.sh"

Softmax Compilation:

Uses **g++** to compile two C++ files with optimization (**-O2**) into an executable named "softmax" in the "build" directory.
Uses **pgc++** with OpenACC directives to compile three C++ files into an executable named "softmax_openacc" in the "build" directory.

Neural Network Compilation:

Uses **g++** to compile three C++ files with optimization (**-O2**) into an executable named "nn" in the "build" directory.
Uses **pgc++** with OpenACC directives to compile three C++ files into an executable named "nn_openacc" in the "build" directory.

Execution Instructions:

sh sbatch.sh

The script essentially runs Softmax and Neural Network classifiers in both sequential and OpenACC-parallelized versions. It uses Slurm directives to specify job parameters such as the number of nodes, tasks per node, and GPU allocation. The results are logged to the "Project4-Results.txt" file.

Softmax Execution:

Runs the Softmax classifier sequentially (**softmax** executable) on the specified datasets using one CPU task.
Runs the Softmax classifier with OpenACC directives (**softmax_openacc** executable) on the specified datasets using one GPU.

Neural Network Execution:

Runs the Neural Network classifier sequentially (**nn** executable) on the specified datasets using one CPU task.
Runs the Neural Network classifier with OpenACC directives (**nn_openacc** executable) on the specified datasets using one GPU.

Task1: Train MNIST with softmax regression

These functions encapsulate the training process for a softmax regression model, handling details such as batch processing, one-hot encoding, softmax normalization, gradient computation, parameter updating, and evaluation on training and test datasets. The **train_softmax** function orchestrates multiple epochs and provides insights into the model's performance over time.

softmax_regression_epoch_cpp Function

Purpose:

This function performs a single training epoch for a softmax regression model using stochastic gradient descent (SGD).

Key Steps:

Batch Processing:

Iterates through the dataset in batches.
Each batch is processed independently.

One-Hot Encoding:

Converts the label vector **y** into a one-hot matrix **Y**.

Logits Computation:

Computes logits by performing a matrix multiplication (**matrix_dot**) of input data **X** and weight matrix **theta**.

Softmax Normalization:

Applies the softmax function to normalize the logits.

Gradient Computation:

Computes gradients using the softmax derivative.

Parameter Update:

Updates the weight matrix **theta** using the computed gradients.

Memory Cleanup:

Frees dynamically allocated memory.

train_softmax Function

Purpose:

This function fully trains a softmax classifier over multiple epochs.

Key Steps:

Initialization:

Allocates memory for the weight matrix **theta**.

Initializes **theta** with zeros.

Allocates memory for result matrices for the training and test sets.

Epoch Training Loop:

Iterates through a specified number of training epochs.

Calls **softmax_regression_epoch_cpp** for each epoch to perform training.

Evaluates the model on the training and test datasets.

Performance Metrics:

Computes and prints the mean loss and error for each epoch.

Execution Time Measurement:

Measures and prints the total execution time.

Memory Cleanup:

Frees dynamically allocated memory.

Task2: Accelerate softmax with OpenACC

OpenACC directives are employed to parallelize and optimize critical sections of the softmax regression training process, resulting in improved execution time efficiency, particularly when executed on GPUs.

The **softmax_regression_epoch_openacc** function utilizes OpenACC directives to parallelize and optimize the training of a softmax regression model, specifically for execution on accelerators like GPUs. The key aspects of OpenACC's contribution to execution time efficiency include:

Loop Parallelization:

OpenACC directives are used to parallelize loops, enabling concurrent execution of iterations and taking advantage of the parallel processing capabilities of GPUs.

Data Management:

OpenACC directives for data movement (**#pragma acc enter data**, **#pragma acc exit data**) facilitate efficient transfer of data between the CPU and GPU, optimizing overall memory access.

Data Parallelism:

The **parallel loop** directive identifies and exploits data parallelism, distributing computation across multiple threads or processing units for increased performance.

Independent Loop Optimization:

The **#pragma acc parallel loop independent** directive signifies that loop iterations are independent, allowing the compiler to optimize for parallel execution.

GPU Acceleration:

OpenACC enables GPU acceleration for compute-intensive tasks, offloading matrix operations and gradient computations to the GPU for parallel processing.

Task3: Train MNIST with neural network

These functions implement the training and evaluation processes for a two-layer neural network, emphasizing simplicity and clarity in the code structure. The training involves forward and backward passes through the network, updating weights with SGD. The evaluation assesses the network's performance on training and testing datasets.

nn_epoch_cpp Function

Purpose:

Performs a single epoch of Stochastic Gradient Descent (SGD) for a two-layer neural network. The network consists of a ReLU activation followed by a linear layer without bias terms.

Key Steps:

Batch Processing:

Iterates through the dataset in batches without shuffling.

Forward Pass:

Computes the dot product of input (**X**) and the first layer weights (**W1**).
Applies ReLU activation element-wise.
Computes the dot product of the ReLU output and the second layer weights (**W2**).

Softmax Activation:

Applies softmax activation to the second layer output.

One-Hot Encoding:

Converts class labels (**y**) to one-hot encoding.

Backward Pass (Gradient Descent):

Computes gradients for the weights (**W1** and **W2**) using backpropagation.
Updates weights using the calculated gradients and the learning rate.

Memory Cleanup:

Releases memory for temporary variables.

evaluate_nn Function

Purpose:

Performs a forward pass through the neural network to evaluate its performance on a dataset.

Key Steps:

Forward Pass:

Computes the dot product of input (**X**) and the first layer weights (**W1**).
Applies ReLU activation element-wise.
Computes the dot product of the ReLU output and the second layer weights (**W2**).

Softmax Activation:

Applies softmax activation to the second layer output.

Result Storage:

Copies the result to the output array.

Memory Cleanup:

Releases memory for temporary variables.

train_nn Function

Purpose:

Fully trains a neural network over multiple epochs using the provided training and testing datasets.

Key Steps:

Initialization:

Allocates memory for first and second layer weights (**W1** and **W2**).
Initializes weights with normal distribution and scales them.

Epoch Training Loop:

Iterates through a specified number of training epochs.
Calls **nn_epoch_cpp** for each epoch to perform training.

Evaluation:

Evaluates the trained model on both training and testing datasets after each epoch.

Performance Metrics:

Computes and prints mean loss and error for each epoch.

Execution Time Measurement:

Measures and prints the total execution time.

Memory Cleanup:

Releases memory for weights and result arrays.

Task4: Accelerate neural network with OpenACC

The OpenACC directives in this code aim to optimize memory usage, minimize data transfer overhead, and parallelize computationally intensive tasks. The overall goal is to improve the efficiency of neural network training, especially when executed on parallel architectures like GPUs.

Data Movement:

The **#pragma acc data** directives are used to manage data movement between the CPU and GPU memory. It ensures that relevant data, such as matrices used in neural network computations, is efficiently transferred and available on the GPU when needed.

Parallelization of Loops:

The **#pragma acc parallel loop** directives are applied to loops where parallel execution can be exploited. For example, the loops involving element-wise operations like ReLU activation and element-wise multiplication are parallelized. This allows multiple iterations of the loop to be processed simultaneously, leveraging the parallel processing capabilities of GPUs or multicore CPUs.

Collapse and Independent Clauses:

The **collapse(2)** clause in **#pragma acc parallel loop** is used to collapse nested loops into a single parallel loop. This enhances parallelism, especially in scenarios with nested loops.

The **independent** clause in **#pragma acc parallel loop** indicates that loop iterations are independent. This allows the compiler to optimize the parallel execution further.

Memory Optimization:

The **copyin** and **copyout** clauses in **#pragma acc data** directives are used to control the copying of data between CPU and GPU memory explicitly. This helps minimize unnecessary data transfers, reducing overhead and optimizing memory usage on the GPU.

Parallelization of Matrix Operations:

Functions like **matrix_dot_openacc** and **matrix_softmax_normalize_openacc** are explicitly parallelized using OpenACC directives. These functions involve computationally intensive matrix operations commonly found in neural network training.

Explicit Memory Management:

The **enter data** and **exit data** directives are used to explicitly manage GPU memory. They specify which data needs to be present on the GPU during computation and when to transfer it back to the CPU.

Experiment results

Softmax Sequential	softmax OpenACC	NN Sequential	NN OpenACC
9658ms	4115ms	996392ms	105046ms

Experiment results

Task1: Train MNIST with softmax regression

Softmax Sequential
Training softmax regression

Epoch	Train Loss	Train Err	Test Loss	Test Err
0	0.35134	0.10182	0.33588	0.09400
1	0.32142	0.09268	0.31086	0.08730
2	0.30802	0.08795	0.30097	0.08550
3	0.29987	0.08532	0.29558	0.08370
4	0.29415	0.08323	0.29215	0.08230
5	0.28980	0.08182	0.28973	0.08090
6	0.28633	0.08085	0.28793	0.08080
7	0.28345	0.07997	0.28651	0.08040
8	0.28099	0.07923	0.28537	0.08010
9	0.27886	0.07847	0.28442	0.07970

Execution Time: 9658 milliseconds

Performance counter stats for 'sh sbatch.sh':

```
    20,682,403      instructions:u          #    1.03  insn per cycle          (73.77%)
    20,100,595      cpu-cycles:u                                                 (75.62%)
       172,918      cache-references:u                                           (69.60%)
        47,492      cache-misses:u         #   27.465 % of all cache refs      (79.73%)
     2,690,565      branch-instructions:u                                        (79.57%)
        95,503      branch-misses:u        #    3.55% of all branches          (58.52%)
       311,702      bus-cycles:u                                                 (58.56%)

   10.121149064 seconds time elapsed
   0.015037000 seconds user
   0.017046000 seconds sys
```

Observation:
- The Softmax Sequential training run has an instruction per cycle (IPC) of 1.03.
- Cache misses account for 27.47% of all cache references.

Analysis:
- The IPC suggests relatively good instruction efficiency.
- Cache misses could be optimized to enhance data access efficiency.
- Overall, the training run exhibits decent performance, but further improvements in cache utilization could be explored.

Task2: Accelerate softmax with OpenACC

Softmax OpenACC
Training softmax regression (GPU)

Epoch	Train Loss	Train Err	Test Loss	Test Err
0	0.35127	0.10177	0.33582	0.09400
1	0.32137	0.09270	0.31083	0.08720
2	0.30799	0.08795	0.30096	0.08560
3	0.29984	0.08530	0.29558	0.08370
4	0.29412	0.08327	0.29214	0.08230
5	0.28978	0.08182	0.28973	0.08090
6	0.28631	0.08080	0.28793	0.08070
7	0.28343	0.07998	0.28652	0.08040
8	0.28097	0.07927	0.28537	0.08010
9	0.27884	0.07845	0.28442	0.07960

Execution Time: 4115 milliseconds

Performance counter stats for 'sh sbatch.sh':

```
    17,007,053      instructions:u          #    1.17  insn per cycle          (73.01%)
    14,573,163      cpu-cycles:u                                                 (73.13%)
       189,850      cache-references:u                                           (72.08%)
        61,460      cache-misses:u         #   32.373 % of all cache refs      (74.78%)
     4,157,000      branch-instructions:u                                        (79.42%)
       114,992      branch-misses:u        #    2.77% of all branches          (56.76%)
       344,842      bus-cycles:u                                                 (56.67%)

   4.707307364 seconds time elapsed
   0.016887000 seconds user
   0.023087000 seconds sys
```

Observation:
- Softmax OpenACC training run has an instruction per cycle (IPC) of 1.17.
- Cache misses account for 32.37% of all cache references.

Analysis:
- IPC indicates good instruction efficiency.
- Cache misses are relatively high, indicating potential for optimization in data access.
- The overall performance is decent, but tuning cache utilization may further enhance efficiency.
- Performing data copyin and copyout for data management introduces a lot of overhead and time consumption that affects execution time, some functions with openacc are not as efficient.

Task3: Train MNIST with neural network

NN Sequential
Training two layer neural network w/ 400 hidden units

Epoch	Train Loss	Train Err	Test Loss	Test Err
0	0.13443	0.04015	0.14269	0.04250
1	0.09679	0.02998	0.11609	0.03690
2	0.07349	0.02193	0.10037	0.03160
3	0.05823	0.01703	0.09075	0.02820
4	0.04699	0.01288	0.08386	0.02570
5	0.03920	0.01042	0.07927	0.02500
6	0.03307	0.00825	0.07584	0.02450
7	0.02827	0.00673	0.07364	0.02400
8	0.02458	0.00565	0.07204	0.02320
9	0.02178	0.00480	0.07120	0.02260
10	0.01889	0.00380	0.06969	0.02180
11	0.01681	0.00323	0.06898	0.02180
12	0.01513	0.00265	0.06855	0.02140
13	0.01369	0.00213	0.06825	0.02110
14	0.01234	0.00160	0.06763	0.02110
15	0.01119	0.00118	0.06736	0.02080
16	0.01024	0.00092	0.06709	0.02040
17	0.00942	0.00075	0.06687	0.02040
18	0.00860	0.00058	0.06652	0.02020
19	0.00797	0.00043	0.06638	0.02020

Execution Time: 996392 milliseconds

Performance counter stats for 'sh sbatch.sh':

18,413,016	instructions:u	# 1.27 insn per cycle	(73.74%)
14,484,426	cpu-cycles:u		(72.95%)
169,913	cache-references:u		(68.50%)
81,953	cache-misses:u	# 48.232 % of all cache refs	(65.18%)
4,227,178	branch-instructions:u		(78.20%)
114,069	branch-misses:u	# 2.70% of all branches	(68.28%)
400,754	bus-cycles:u		(61.29%)

996.880360942 seconds time elapsed
0.023877000 seconds user
0.020015000 seconds sys

Observation:

- NN Sequential training run has an instruction per cycle (IPC) of 1.27.
- Cache misses account for 48.23% of all cache references.

Analysis:

- IPC indicates reasonable instruction efficiency.
- A high cache miss rate suggests potential for optimization in memory access patterns.
- Overall, the performance could benefit from further tuning, particularly in reducing cache misses.

Task4: Accelerate neural network with OpenACC

NN OpenACC
Training two layer neural network w/ 400 hidden units (GPU)

Epoch	Train Loss	Train Err	Test Loss	Test Err
0	0.13465	0.04023	0.14293	0.04240
1	0.09619	0.03008	0.11522	0.03650
2	0.07466	0.02253	0.10136	0.03140
3	0.05790	0.01678	0.09014	0.02820
4	0.04698	0.01298	0.08338	0.02660
5	0.03882	0.01010	0.07877	0.02500
6	0.03248	0.00805	0.07555	0.02480
7	0.02844	0.00680	0.07388	0.02360
8	0.02429	0.00550	0.07189	0.02300
9	0.02124	0.00462	0.07065	0.02240
10	0.01862	0.00372	0.06975	0.02210
11	0.01665	0.00315	0.06924	0.02150
12	0.01485	0.00257	0.06841	0.02140
13	0.01349	0.00207	0.06811	0.02110
14	0.01225	0.00167	0.06776	0.02090
15	0.01116	0.00128	0.06744	0.02060
16	0.01024	0.00098	0.06722	0.02030
17	0.00936	0.00073	0.06691	0.02030
18	0.00869	0.00073	0.06686	0.02010
19	0.00796	0.00053	0.06658	0.01980

Execution Time: 105046 milliseconds

Performance counter stats for 'sh sbatch.sh':

```
     15,455,285      instructions:u            #    0.90  insn per cycle          (69.95%)
     17,135,847      cpu-cycles:u                                                  (74.73%)
        218,105      cache-references:u                                            (78.25%)
        102,137      cache-misses:u            #   46.829 % of all cache refs     (77.66%)
      4,132,187      branch-instructions:u                                         (73.92%)
        107,666      branch-misses:u           #    2.61% of all branches         (53.19%)
        324,480      bus-cycles:u                                                  (55.39%)

   106.751030625 seconds time elapsed

     0.022792000 seconds user
     0.016773000 seconds sys
```

Observation:

- NN OpenACC training run has an instruction per cycle (IPC) of 0.90.
- Cache misses account for 46.83% of all cache references.

Analysis:

- IPC indicates suboptimal instruction efficiency.
- High cache misses suggest room for improvement in data access patterns.
- The overall performance is subpar; optimizing cache usage could significantly enhance efficiency.
- Performing data copyin and copyout for data management introduces a lot of overhead and time consumption that affects execution time, some functions with openacc are not as efficient.