CSC4005 Project 4

ID:120090712

How to execute the program:

Depress the zip file, the depressed folder is "120090712". Upload into the cluster. Then, in the cluster, use following commands to complie and subbmit the tasks:

```
cd /path/to/120090712
bash ./test.sh
```

Then, waite for the results, the results will be in the file "Project4_results.txt" under the "path/to/120090712" directory. (If there is any problem when executing, maybe you need to change the pwd command in sbatc.sh to the file ptah manually)

How did I design and implement each algorithm:

Task1: Train MNIST with softmax regression

```
In designing and implementing the algorithm, I created C++ functions for
fundamental matrix operations, including matrix multiplication, transposition,
element-wise subtraction, and scalar operations. These operations serve as
building blocks for the softmax regression algorithm.

The softmax_regression_epoch_cpp function conducts a single epoch of training
using stochastic gradient descent. It calculates logits, applies softmax
normalization, converts labels to one-hot encoding, computes gradients, and
updates the parameter matrix (theta). The process is repeated for multiple epochs
in the train_softmax function, which also monitors and displays training and
testing loss, as well as error rates.

The overall approach emphasizes modularity and efficiency, utilizing optimized
nested loops for matrix operations and employing standard C++ libraries for memory
management and performance measurement.
```

Task2: Accelerate softmax with OpenACC

```
In designing and implementing the algorithm, I utilized OpenACC directives to
parallelize key sections of the code for efficient execution on parallel
architectures. The matrix operations, including dot multiplication, transposition,
element-wise subtraction, and scalar operations, were adapted to leverage
OpenACC's parallelization capabilities. The softmax regression epoch training
```

function was modified to integrate OpenACC directives, ensuring parallel execution during the training process. Additionally, the training function was extended to monitor and display the execution time. The overall approach focused on optimizing performance while maintaining a modular code structure for clarity.

Task3: Train MNIST with neural network

In designing and implementing the algorithm, I structured a neural network training process for image classification using the MNIST dataset. I organized the code into modular functions, specifically the nn_epoch_cpp and train_nn functions. Within the training epoch function, I applied a two-layer neural network architecture with ReLU activation. The forward pass involved matrix operations such as dot products and softmax normalization. For the backward pass, I calculated gradients and updated the weights using a specified learning rate.

I allocated memory for intermediate variables and gradients, ensuring efficient memory usage. Random initialization of weights was incorporated, and weight scaling was applied for better convergence. The training process was organized into epochs, and during each epoch, I processed data in batches to enhance efficiency.

To evaluate the model's performance, I monitored and displayed the training and testing loss, as well as classification errors, at each epoch. The code also included timing measurements to assess the execution duration. Overall, I aimed for a clear, concise, and modular design to facilitate understanding and modification of the neural network training algorithm.

Task4: Accelerate neural network with OpenACC

In designing and implementing the OpenACC-accelerated neural network training algorithm, I extended the existing CPU-based neural network code to leverage the parallel processing capabilities of GPUs. I focused on optimizing matrix operations, which are fundamental to neural network computations, using OpenACC directives.

I introduced parallelization directives, such as #pragma acc parallel loop, to distribute the computation tasks across the GPU cores efficiently. This allowed me to accelerate critical sections, including matrix multiplication, softmax normalization, and gradient updates, by offloading them to the GPU.

I utilized specific OpenACC constructs, such as data directives, to manage data transfers between the CPU and GPU memory spaces, optimizing memory usage during parallel execution. Additionally, I applied atomic updates and parallel loops to ensure data consistency and efficient parallelization in the presence of shared resources.

To enable OpenACC acceleration for the entire neural network training process, I

```
systematically integrated the OpenACC directives into key functions, like
nn_epoch_openacc and train_nn_openacc, ensuring a balanced distribution of
workload between the CPU and GPU.

By incorporating these optimizations, I aimed to enhance the overall performance
of the neural network training algorithm, allowing it to efficiently exploit the
parallel processing capabilities of GPUs for faster convergence during training.
```

Experiment results and numerical analysis

```
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: 7545 milliseconds

Softmax OpenACC
Training softmax regression (GPU)
| 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: 6510 milliseconds

NN Sequential
Training two layer neural network w/ 400 hidden units
| Epoch | Train Loss | Train Err | Test Loss | Test Err |
|     0 |    0.13365 |   0.03995 |   0.14203 |  0.04290 |
|     1 |    0.09597 |   0.02967 |   0.11524 |  0.03610 |
|     2 |    0.07262 |   0.02187 |   0.10002 |  0.03100 |
|     3 |    0.05818 |   0.01702 |   0.09115 |  0.02800 |
|     4 |    0.04759 |   0.01313 |   0.08488 |  0.02620 |
|     5 |    0.03931 |   0.01023 |   0.08012 |  0.02530 |
```

```
|     6 |    0.03313 |   0.00828 |   0.07670 |  0.02470 |
|     7 |    0.02857 |   0.00700 |   0.07464 |  0.02380 |
|     8 |    0.02475 |   0.00592 |   0.07267 |  0.02300 |
|     9 |    0.02146 |   0.00483 |   0.07099 |  0.02250 |
|    10 |    0.01911 |   0.00388 |   0.07011 |  0.02200 |
|    11 |    0.01702 |   0.00322 |   0.06922 |  0.02170 |
|    12 |    0.01513 |   0.00253 |   0.06850 |  0.02130 |
|    13 |    0.01368 |   0.00218 |   0.06807 |  0.02100 |
|    14 |    0.01243 |   0.00168 |   0.06768 |  0.02080 |
|    15 |    0.01121 |   0.00122 |   0.06732 |  0.02020 |
|    16 |    0.01035 |   0.00093 |   0.06711 |  0.01990 |
|    17 |    0.00944 |   0.00078 |   0.06695 |  0.02020 |
|    18 |    0.00870 |   0.00065 |   0.06667 |  0.01990 |
|    19 |    0.00802 |   0.00055 |   0.06642 |  0.01960 |
Execution Time: 270200 milliseconds

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 |    0.09652 |   0.03020 |   0.11593 |  0.03700 |
|     2 |    0.07355 |   0.02223 |   0.10058 |  0.03170 |
|     3 |    0.05826 |   0.01717 |   0.09062 |  0.02800 |
|     4 |    0.04670 |   0.01307 |   0.08331 |  0.02640 |
|     5 |    0.03870 |   0.01008 |   0.07891 |  0.02530 |
|     6 |    0.03272 |   0.00807 |   0.07585 |  0.02440 |
|     7 |    0.02817 |   0.00680 |   0.07348 |  0.02430 |
|     8 |    0.02437 |   0.00548 |   0.07190 |  0.02310 |
|     9 |    0.02132 |   0.00462 |   0.07103 |  0.02220 |
|    10 |    0.01883 |   0.00372 |   0.06990 |  0.02190 |
|    11 |    0.01668 |   0.00307 |   0.06897 |  0.02180 |
|    12 |    0.01504 |   0.00258 |   0.06857 |  0.02130 |
|    13 |    0.01365 |   0.00200 |   0.06813 |  0.02160 |
|    14 |    0.01214 |   0.00160 |   0.06770 |  0.02110 |
|    15 |    0.01108 |   0.00113 |   0.06731 |  0.02080 |
|    16 |    0.01024 |   0.00097 |   0.06727 |  0.02080 |
|    17 |    0.00937 |   0.00080 |   0.06699 |  0.02050 |
|    18 |    0.00850 |   0.00063 |   0.06657 |  0.02020 |
|    19 |    0.00796 |   0.00055 |   0.06663 |  0.02000 |
Execution Time: 120938 milliseconds
```

Speedup and efficiency

```
Softmax:"Speedup:1.16","efficiency:116%", assuming I am comparing a single GPU or
core.
NN:"Speedup:2.23","efficiency:223%", assuming I am comparing a single GPU or core.
```

Kinds of optimizations I have tried

COMPELLING

I use O3 instead of O3, O3 aims to improve the efficiency of parallel computing compared to O2. It achieves this by applying advanced optimization techniques specifically targeted at parallel code. These optimizations include automatic vectorization, loop unrolling, and software pipelining. Loop unrolling is a technique where the loop iterations are reduced by executing multiple iterations in a single loop iteration. This reduces the overhead of loop control and improves the efficiency of parallel execution.

Locality

The number of operands of matrix multiplication is O (n^3), but due to the discontinuity of data in different rows, it will increase the number of cache misses, and this will waste a lot of time. My code has adjusted the order of the loops. After I made the adjustments, my matrix multiplication will continuously take data from the same row during each multiplication, thereby reducing the number of cache misses.

Openacc

I applied OpenACC directives to parallelize key computations in the softmax regression algorithm, optimizing matrix operations for GPU acceleration. However, upon further reflection, I recognized a potential for additional improvement in the form of reducing the frequency of "data present" events. Currently, the code involves multiple transfers of data between the CPU and GPU, and I acknowledge that minimizing these transfers is crucial for further optimization.

To address this, I plan to explore ways to consolidate and streamline data transfers, potentially by optimizing memory usage and minimizing unnecessary movements of data between the CPU and GPU. This optimization strategy aims to enhance the efficiency of the algorithm by mitigating the impact of data transfer overhead.

What have I found from the experiment results

During the experiment, I observed that despite the introduction of OpenACC to parallelize the softmax regression algorithm, the reduction in execution time wasn't as substantial as anticipated. I hypothesize that this discrepancy may be attributed to the overhead associated with frequent data transfers between the CPU and GPU. This led me to realize the importance of minimizing "data present" events, prompting further consideration for optimizing data transfer strategies in future iterations of the code.

Profiling OpenACC with nsys

NN_Openacc profiling:

```
 Time(%)  Total Time (ns)  Instances    Average    Minimum     Maximum
StdDev                                                 Name
```

```
71.7   53,005,842,848     24,080  2,201,239.3   69,664  1,155,528,765
33,058,216.8  matrix_dot_openacc_12_gpu(float const*, float const*, float*,
unsigned long, unsigned long, unsigne…
27.4   20,261,396,091     24,000    844,224.8   32,160      3,043,635
810,250.5  matrix_dot_trans_openacc_36_gpu(float const*, float const*, float*,
unsigned long, unsigned long, u…
 0.2      125,050,710     12,000     10,420.9    9,695         19,008
431.1  matrix_softmax_normalize_openacc_113_gpu(float*, unsigned long, unsigned
long)
 0.2      111,030,006     24,000      4,626.3    1,887         15,841
2,674.1  matrix_div_scalar_openacc_101_gpu(float*, float, unsigned long, unsigned
long)
 0.1      108,658,626     36,000      3,018.3    1,695          9,376
1,742.6  matrix_minus_openacc_77_gpu(float*, float const*, unsigned long, unsigned
long)
 0.1       84,730,441     24,000      3,530.4    1,727         13,472
1,752.2  matrix_mul_scalar_openacc_89_gpu(float*, float, unsigned long, unsigned
long)
 0.1       70,316,656     24,000      2,929.9    1,599         12,000
1,270.1  matrix_dot_trans_openacc_31_gpu(float const*, float const*, float*,
unsigned long, unsigned long, u…
 0.1       61,554,017     12,000      5,129.5    4,896         13,663
216.9  matrix_trans_dot_openacc_60_gpu(float const*, float const*, float*,
unsigned long, unsigned long, u…
 0.1       49,257,115     24,080      2,045.6    1,567        234,399
6,784.8  matrix_dot_openacc_7_gpu(float const*, float const*, float*, unsigned
long, unsigned long, unsigned…
 0.0       27,504,189     12,000      2,292.0    2,207          3,712
83.4  matrix_mul_openacc_270_gpu(float*, float const*, unsigned long)
 0.0       24,123,040     12,000      2,010.3    1,919          9,503
102.3  matrix_trans_dot_openacc_55_gpu(float const*, float const*, float*,
unsigned long, unsigned long, u…
 0.0       21,771,410     12,000      1,814.3    1,727          3,104
74.4  vector_to_one_hot_matrix_openacc_137_gpu(unsigned char const*, float*,
unsigned long, unsigned long)
 0.0       20,012,273     12,000      1,667.7    1,599          9,568
139.5  vector_to_one_hot_matrix_openacc_132_gpu(unsigned char const*, float*,
unsigned long, unsigned long)
```

Softmax_Opencc profiling:

```
 Time(%)  Total Time (ns)  Instances   Average   Minimum   Maximum      StdDev
Name
```

```
77.1    1,810,723,812      6,020  300,784.7  174,559  43,915,517  1,851,937.8
matrix_dot_openacc_12_gpu(float const*, float const*, float*, unsigned long,
unsigned long, unsigne…
16.5      386,798,360      6,000   64,466.4   57,375     112,927      9,444.2
matrix_dot_trans_openacc_36_gpu(float const*, float const*, float*, unsigned long,
unsigned long, u…
 2.7       62,323,279      6,000   10,387.2    9,599      18,944      1,757.5
matrix_softmax_normalize_openacc_113_gpu(float*, unsigned long, unsigned long)
 0.9       22,067,025     12,000    1,838.9    1,695       3,233        224.6
matrix_minus_openacc_77_gpu(float*, float const*, unsigned long, unsigned long)
 0.6       13,220,842      6,000    2,203.5    2,047       3,776        280.8
matrix_div_scalar_openacc_101_gpu(float*, float, unsigned long, unsigned long)
 0.5       11,114,523      6,000    1,852.4    1,759       3,136        220.9
matrix_mul_scalar_openacc_89_gpu(float*, float, unsigned long, unsigned long)
 0.5       11,076,203      6,000    1,846.0    1,727       3,040        222.1
vector_to_one_hot_matrix_openacc_137_gpu(unsigned char const*, float*, unsigned
long, unsigned long)
 0.4       10,344,788      6,000    1,724.1    1,631       2,816        201.3
matrix_dot_trans_openacc_31_gpu(float const*, float const*, float*, unsigned long,
unsigned long, u…
 0.4       10,222,387      6,020    1,698.1    1,599       6,720        284.0
matrix_dot_openacc_7_gpu(float const*, float const*, float*, unsigned long,
unsigned long, unsigned…
 0.4       10,118,474      6,000    1,686.4    1,599       2,752        195.0
vector_to_one_hot_matrix_openacc_132_gpu(unsigned char const*, float*, unsigned
long, unsigned long)
```