Project 4

Parallel Programming with Machine Learning

This project weights 12.5% for your final grade (4 Projects for 50%)

Release Date:

November 21th，2023 (Beijing Time, UTC+08:00)

Deadline:

11:59 P.M., December 8th, 2023 (Beijing Time, UTC+08:00)

TA/USTF In Charge of This Assignment

Mr. Xue Haonan for Implementation (haonanxue@link.cuhk.edu.cn)

Mr. Zhong Yebin for Implementation (yebinzhong@link.cuhk.edu.cn)

Mr. Chen Weibin for Writing Grading criteria (220019062@link.cuhk.edu.cn)

Prologue

In this project, you will have the opportunity to gain insight and practice in using OpenACC to accelerate machine learning algorithms. Specifically, you will be accelerating softmax regression and neural networks (NNs).

First, you will need to understand the basic principles and algorithms of softmax regression and neural networks. Then, you will work with OpenACC, a programming model for parallel computing that makes it easier for you to optimize your code to run on GPUs, thereby greatly increasing the speed of computation.

This assignment will help you understand the importance of parallel computing in machine learning, especially when working with large-scale data and complex models. You will learn how to effectively utilize hardware resources to improve the performance and efficiency of machine learning algorithms.

REMIND: Please start ASAP to avoid the peak period of cluster job submission.

Task0: Setup

Download the dataset from BB. Unzip `dataset.zip` to folder `project4`.

The structure of working directory should look like below:

```
$ tree .
.
├── build
│   ├── nn
│   ├── nn_openacc
│   ├── softmax
│   └── softmax_openacc
├── dataset
│   ├── testing
│   │   ├── t10k-images.idx3-ubyte
│   │   └── t10k-labels.idx1-ubyte
│   └── training
│       ├── train-images.idx3-ubyte
│       └── train-labels.idx1-ubyte
├── README.md
├── sbatch.sh
├── src
│   ├── nn_classifier.cpp
│   ├── nn_classifier_openacc.cpp
│   ├── simple_ml_ext.cpp
│   ├── simple_ml_ext.hpp
│   ├── simple_ml_openacc.cpp
│   ├── simple_ml_openacc.hpp
│   ├── softmax_classifier.cpp
│   └── softmax_classifier_openacc.cpp
└── test.sh

5 directories, 20 files
```

Task1: Train MNIST with softmax regression

Softmax regression (or multinomial logistic regression) is an extension of logistic regression that can handle multiclass classification problems. Softmax Regression, also known as Multinomial Logistic Regression, is an extension of Logistic Regression to the multi-class problem. The mathematical expression is as follows:

Suppose we have an input vector $x \in R^n$, and we want to classify it into one of K different classes.

For each class j, we have a weight vector $w_j \in R^n$ and a bias term b_j.

We can compute the unnormalized log probabilities o_j for x belonging to class j as follows:

$$z_j = x^T \theta_j + b_j$$

This gives us an output vector $\theta \in R^K$, where each element o_j represents the unnormalized log probability of x belonging to class j.

We can then convert these unnormalized log probabilities into probabilities using the softmax function. The softmax function is defined as follows:

$$p(y = j|x) = \frac{e^{z_j}}{\Sigma_{k=1}^K e^{z_k}}$$

This gives us a probability vector $p \in R^K$, where each element p_j represents the probability of x belonging to class j. This is the mathematical expression of softmax regression. It maps an input vector x to a probability vector p, where each element p_j represents the probability of x belonging to class j.

This process is also known as softmax classification. In practice, we usually choose the class with the highest probability as the predicted class.

For a multi-class output that can take on values $y \in \{1, \ldots, k\}$, the softmax loss takes as input a vector of logits $z \in \mathbb{R}^k$, the true class $y \in \{1, \ldots, k\}$ returns a loss defined by:

$$\ell_{\text{softmax}}(z, y) = \log \sum_{i=1}^k \exp z_i - z_y$$

Softmax gradient descent optimization algorithm: we need to compute the gradients of the loss function with respect to the weights and biases, and then update the weights and biases. We can also write this in the more compact notation we discussed in class. Namely, if we let $X \in \mathbb{R}^{m \times n}$ denote a design matrix of some m inputs (either the entire dataset or a minibatch), $y \in \{1, \ldots, k\}^m$ a corresponding vector of labels, and overloading ℓ_{softmax} to refer to the average softmax loss, then

$$\nabla_\Theta \ell_{\text{softmax}}(X\Theta, y) = \frac{1}{m} X^T (Z - I_y)$$

where

$$Z = normalize(\exp(X\Theta)) \quad (\text{normalization applied row-wise})$$

denotes the matrix of logits, and $I_y \in \mathbb{R}^{m \times k}$ represents a concatenation of one-hot bases for the labels in y.

Here is the given training code in Python:

```
def softmax_regression_epoch(X, y, theta, lr=0.1, batch=100):
    for i in range(0, X.shape[0], batch):
        X_b = X[i : i + batch]
        h_X_exp = np.exp(np.dot(X_b, theta))
        Z = h_X_exp / np.sum(h_X_exp, axis=1)[:, None]
        Y = np.zeros(Z.shape, np.float32)
        Y[np.arange(y[i : i + batch].size), y[i : i + batch]] = 1
        gradients = np.dot(X_b.T, Z - Y) / batch * lr
        theta -= gradients
```

Note that for "real" implementation of softmax loss you would want to scale the logits to prevent numerical overflow, but we won't worry about that here (the rest of the assignment will work fine even if you don't worry about this).

There are some functions inside the softmax function that you also need to fill in the details:

Function Declaration	What does the function do
`matrix_dot(A, B, C, m, n, k)`	perform a matrix multiplication operation between matrices A and B , and the result is stored in matrix C
`matrix_softmax_normalize(A, m, n)`	apply the softmax activation function to the matrix A
`vector_to_one_hot_matrix(y, Y, m, n)`	convert a vector y into a one-hot encoded matrix Y with dimensions $m \times n$
`matrix_minus(A, B, m, n)`	perform element-wise subtraction between matrices A and B, with the result stored in matrix A
`matrix_dot_trans(A, B, C, n, m, k)`	perform a matrix multiplication between the transpose of A and B, with the result stored in matrix C
`matrix_div_scalar(A, scalar, m, n)`	divides all elements of matrix A by the scalar value $scalar$
`matrix_mul_scalar(A, scalar, m, n)`	multiply all elements of matrix A by the scalar value $scalar$

Function Declaration	What does the function do
`matrix_minus(A, B, m, n)`	subtracts matrix A from matrix B element-wise, , with the result stored in matrix A
`softmax_regression_epoch_cpp(X, y, theta, m, n, k, lr, batch)`	train $theta$ of softmax regression for 1 epoch
`train_softmax(train_data, test_data, num_classes, epochs, lr, batch)`	train a softmax classifier

In the implementation, you are allowed to define your variables and functions to facilitate your programming.

The outcome is like below:

```
| 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 |    0.28981 |   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 |    0.28100 |   0.07923 |   0.28537 |  0.08010 |
|     9 |    0.27887 |   0.07847 |   0.28442 |  0.07970 |
```

Task2: Accelerate softmax with OpenACC

You need to accelerate the `train_softmax` function and the functions inside the `softmax_regression_epoch_cpp` function with OpenACC.
Hint: You can accelerate the program by applying OpenACC to each function.

Task3: Train MNIST with neural network

The inference and training process of a neural network can be described by the following formulas:

1. **Forward Propagation (Inference)**
   The forward propagation process of a neural network can be described by the following formula, where $a^{(l)}$ is the activation value of the l th layer $W^{(l)}$ is the weight of the l th layer, $b^{(l)}$ is the bias of the l th layer, and f is the activation function:

$$a^{(l)} = f(W^{(l)}a^{(l-1)} + b^{(l)})$$

This process starts from the input layer, through the calculation of each layer's weights and biases, as well as the activation function, and finally obtains the predicted value of the output layer.

2. **Backward Propagation (Training)**
   The training process of a neural network mainly updates the weights and biases through the backpropagation algorithm. First, we need to define a loss function L to measure the gap between the predicted value and the true value. Then, we update the weights and biases by calculating the gradient of the loss function for the weights and biases:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial W^{(l)}}$$

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial b^{(l)}}$$

Here, $\frac{\partial L}{\partial a^{(l)}}$ can be propagated from the next layer to the previous layer through the chain rule. Finally, we use the gradient descent method to update the weights and biases:

$$W^{(l)} = W^{(l)} - \alpha \frac{\partial L}{\partial W^{(l)}}$$

$$b^{(l)} = b^{(l)} - \alpha \frac{\partial L}{\partial b^{(l)}}$$

Here, α is the learning rate, which controls the step size of the update.

In this project, we are going to implement a 2-layer NN with SGD.

$$Z = W_2^T ReLU(W_1^T x)$$

where $W_1 \in \mathbb{R}^{n \times d}$ and $W_2 \in \mathbb{R}^{d \times k}$ represent the weights of the network (which has a d-dimensional hidden unit), and where $z \in \mathbb{R}^k$ represents the logits output by the network. We again use the softmax / cross-entropy loss, meaning that we want to solve the optimization problem.

Using the chain rule, we can derive the backpropagation updates for this network (we'll briefly cover these in class, on 9/8, but also provide the final form here for ease of implementation). Specifically, let

$$Z_1 \in \mathbb{R}^{m \times d} = \mathrm{ReLU}(XW_1)$$

$$G_2 \in \mathbb{R}^{m \times k} = normalize(\exp(Z_1 W_2)) - I_y$$

$$G_1 \in \mathbb{R}^{m \times d} = 1\{Z_1 > 0\} \circ (G_2 W_2^T)$$

where $1\{Z_1 > 0\}$ is a binary matrix with entries equal to zero or one depending on whether each term in Z_1 is strictly positive and where $\circ$ denotes elementwise multiplication. Then the gradients of the objective are given by:

$$\nabla_{W_1} \ell_{\text{softmax}}(\text{ReLU}(XW_1)W_2, y) = \frac{1}{m} X^T G_1$$

$$\nabla_{W_2} \ell_{\text{softmax}}(\text{ReLU}(XW_1)W_2, y) = \frac{1}{m} Z_1^T G_2$$

Here is the given training code in Python:

```
def nn_epoch(X, y, W1, W2, lr=0.1, batch=100):
    for i in range(0, X.shape[0], batch):
        X_b = X[i : i + batch]
        Z1 = np.maximum(0, np.dot(X_b, W1))
        h_Z1_exp = np.exp(np.dot(Z1, W2))
        Z2 = h_Z1_exp / np.sum(h_Z1_exp, axis=1)[:, None]
        Y = np.zeros(Z2.shape, np.float32)
        Y[np.arange(y[i : i + batch].size), y[i : i + batch]] = 1
        G1 = np.dot(Z2 - Y, W2.T) * (Z1 > 0)
        W1_l = np.dot(X_b.T, G1) / batch * lr
        W2_l = np.dot(Z1.T, Z2 - Y) / batch * lr
        W1 -= W1_l
        W2 -= W2_l
```

There are some new functions inside the NN function that you also need to fill in the details:

Function Declaration	What does the function do
`matrix_trans_dot(A, B, C, m, n, k)`	perform a matrix multiplication between A and the transpose of B, with the result stored in matrix C
`matrix_mul(A, B, size)`	multiply matrix A from matrix B element-wise, with the result stored in matrix A
`nn_epoch_cpp(X, y, W1, W2, m, n, l, k, lr, batch)`	train the 2-layer NN for 1 epoch
`train_nn(train_data, test_data, num_classes, hidden_dim, epochs, lr, batch)`	train a 2-layer NN classifier

The outcome is like below:

```
| Epoch | Train Loss | Train Err | Test Loss | Test Err |
|     0 |    0.13466 |   0.04023 |   0.14293 |  0.04240 |
```

```
|     1 |    0.09653 |   0.03020 |   0.11593 |  0.03700 |
|     2 |    0.07351 |   0.02227 |   0.10043 |  0.03170 |
|     3 |    0.05862 |   0.01715 |   0.09091 |  0.02880 |
|     4 |    0.04677 |   0.01298 |   0.08348 |  0.02650 |
|     5 |    0.03878 |   0.01015 |   0.07878 |  0.02490 |
|     6 |    0.03281 |   0.00822 |   0.07595 |  0.02470 |
|     7 |    0.02796 |   0.00672 |   0.07341 |  0.02390 |
|     8 |    0.02452 |   0.00558 |   0.07204 |  0.02280 |
|     9 |    0.02133 |   0.00453 |   0.07076 |  0.02240 |
|    10 |    0.01880 |   0.00365 |   0.07004 |  0.02200 |
|    11 |    0.01675 |   0.00320 |   0.06925 |  0.02190 |
|    12 |    0.01510 |   0.00265 |   0.06867 |  0.02190 |
|    13 |    0.01345 |   0.00203 |   0.06821 |  0.02150 |
|    14 |    0.01217 |   0.00150 |   0.06793 |  0.02080 |
|    15 |    0.01136 |   0.00128 |   0.06787 |  0.02100 |
|    16 |    0.01010 |   0.00098 |   0.06725 |  0.02060 |
|    17 |    0.00949 |   0.00090 |   0.06736 |  0.02050 |
|    18 |    0.00860 |   0.00068 |   0.06690 |  0.02020 |
|    19 |    0.00793 |   0.00050 |   0.06666 |  0.02030 |
```

Task4: Accelerate neural network with OpenACC

You need to accelerate the `train_nn` function and the functions inside the `nn_epoch_cpp` function with OpenACC.

Since the calculating precisions on CPU and GPU platforms are different, there is a tiny gap between the outcome of sequential and OpenACC programs. Here is the sample output of OpenACC:

```
| Epoch | Train Loss | Train Err | Test Loss | Test Err |
|     0 |    0.13466 |   0.04023 |   0.14293 |  0.04240 |
|     1 |    0.09699 |   0.03037 |   0.11628 |  0.03700 |
|     2 |    0.07349 |   0.02233 |   0.10028 |  0.03230 |
|     3 |    0.05790 |   0.01675 |   0.09053 |  0.02800 |
|     4 |    0.04668 |   0.01280 |   0.08374 |  0.02650 |
|     5 |    0.03846 |   0.01003 |   0.07861 |  0.02520 |
|     6 |    0.03255 |   0.00810 |   0.07542 |  0.02420 |
|     7 |    0.02800 |   0.00678 |   0.07333 |  0.02410 |
|     8 |    0.02444 |   0.00548 |   0.07163 |  0.02350 |
|     9 |    0.02127 |   0.00447 |   0.07054 |  0.02290 |
|    10 |    0.01869 |   0.00365 |   0.06941 |  0.02230 |
```

```
|    11 |    0.01683 |   0.00318 |   0.06875 |  0.02200 |
|    12 |    0.01501 |   0.00252 |   0.06818 |  0.02120 |
|    13 |    0.01352 |   0.00200 |   0.06757 |  0.02080 |
|    14 |    0.01241 |   0.00172 |   0.06769 |  0.02070 |
|    15 |    0.01116 |   0.00120 |   0.06712 |  0.02050 |
|    16 |    0.01014 |   0.00098 |   0.06664 |  0.02010 |
|    17 |    0.00948 |   0.00088 |   0.06664 |  0.02030 |
|    18 |    0.00856 |   0.00067 |   0.06628 |  0.01980 |
|    19 |    0.00815 |   0.00057 |   0.06644 |  0.01970 |
```

Hint: You can accelerate the program by applying OpenACC to each function.

Extra Credit: Extend Neural Network to Convolutional Neural Network with OpenACC

You need to implement and accelerate the `train_cnn` function and the functions inside the `cnn_epoch_cpp` function with OpenACC. You can use any hyperparameters and filters as you like. Note that your performance of CNN should be better in accuracy than the previous 2-layer NN.

Hint: You can accelerate the program by applying OpenACC to each function. Filters in static when compiling may help a lot in time performance.

How to Execute the Program

Execute the bash script.

```
bash ./test.sh
```

Baseline

Softmax Sequential	softmax OpenACC	NN Sequential	NN OpenACC
9767 ms	1066 ms	683586 ms	68563 ms

NOTICE: the outcome of the classifier in training (including loss and error) should be the same as the sample outcome number by number.

Requirements & Grading Policy

- **Machine Learning (50%)**
    - Task1: Train MNIST with softmax regression (10%)

- Task2: Accelerate softmax with OpenACC (20%)
- Task3: Train MNIST with neural network (10%)
- Task4: Accelerate neural network with OpenACC (10%)

Your programs should be able to compile & execute to get the expected computation result to get the full grade in this part.

- **Performance of Your Program (30%)**
  - 7.5% for each Task

  Try your best to do optimization on your parallel programs for higher speedup. If your programs show similar performance to the baseline performance, then you can get the full mark for this part. Points will be deducted if your parallel programs perform poorly while no justification can be found in the report.
- **One Report in PDF (20%, No Page Limit)**
  - **Regular Report (10%)**
    The report does not have to be very long and beautiful to help you get a good grade, but you need to include what you have done and what you have learned in this project. The following components should be included in the report:
    - How to compile and execute your program to get the expected output on the cluster.
    - Explain clearly how you designed and implemented each algorithm
    - Show the experiment results you get, and do some numerical analysis, such as calculating the speedup and efficiency, demonstrated with tables and figures.
    - What kinds of optimizations have you tried to speed up your parallel program, and how do they work?
    - Any interesting discoveries you found during the experiment?
  - **Profiling OpenACC with nsys (10%)**
    You are required to practice profiling OpenACC programs with nsys as we explained in the Instruction of profiling tools with perf and nsys. The command line profiling of nsys is mandatory while the GUI Nsight System is optional.
- **Extra Credits (10%)**
  - Implement CNN (5%)
  - Accelerate CNN with OpenACC (5%)

  Extra optimizations or interesting discoveries in the first three tasks may also earn you some extra credits.

The Extra Credit Policy

According to the professor, the extra credits in this project cannot be added to other projects to make them full marks. The credits are the honor you received from the professor and the teaching staff, and the professor may help raise you to a higher grade level if you are at the boundary of two grade levels and he thinks you deserve a better grade with your extra credits. For example, if you are among the top students with B+ grade, and get enough extra credits, the professor may raise you to A- grade. Furthermore, the professor will invite a few students with high extra credits to have dinner with him.

Grading Policy for Late Submission

1. late submission for less than 10 minutes after the DDL is tolerated for possible issues during submission.
2. 10 Points deduction for each day after the DDL (11 minutes late will be considered as one day, so be careful)
3. Zero points if you submitted your project late for more than two days
   If you have some special reasons for late submission, please send an email to the professor and cc TA Liu Yuxuan.

File Structure to Submit on BlackBoard

```
<Your StudentID>.pdf  # Report
<Your StudentID>.zip  # Codes
├── sbatch.sh
├── src
│   ├── nn_classifier.cpp
│   ├── nn_classifier_openacc.cpp
│   ├── simple_ml_ext.cpp
│   ├── simple_ml_ext.hpp
│   ├── simple_ml_openacc.cpp
│   ├── simple_ml_openacc.hpp
│   ├── softmax_classifier.cpp
│   └── softmax_classifier_openacc.cpp
└── test.sh

5 directories, 20 files
```