CSC4005 2023 Fall Project 4 Report

Tianci Hou 121090184

1 Parallel Programming with Machine Learning

1.1 Set up

Environment: Refer to course GitHub page.

Type	Compilation flags
Sequential	`-O2`
OpenACC	`-acc`

(a) Compilation Flags

```
$ cd path/to/project
$ bash test.sh
```

(b) Build Commands

Figure 1: Compilation and Build Information

1.2 Overall performance

Softmax Sequential	Softmax OpenACC	NN Sequential	NN OpenACC
9175 ms	916 ms	629244 ms	5807 ms

Table 1: Overall performance.

1.3 Speed up statistic

Softmax Sequential	Softmax OpenACC	NN Sequential	NN OpenACC
6.45%	16.38%	8.64%	1080.70%

Table 2: Speed up compared with baseline

Softmax OpenACC	NN OpenACC
901.64%	10735.96%

Table 3: Speed up compared with sequential approach

1.4 Correctness

My output will be a little different than TA's. Here is a summary of the absolute error.

Softmax: epoch 0-7: $err = 0$; epoch 8-9: $err \leq 10^{-5}$.

NN: epoch 0-4: $err \leq 10^{-5}$; epoch 5-19: $err \leq 5 \times 10^{-4}$

2 Task1: Train MNIST with softmax regression

MNIST is a dataset that recognizes individual numbers, and in the following we will all use the image vector to represent a vector of input matrices.

2.1 Solution

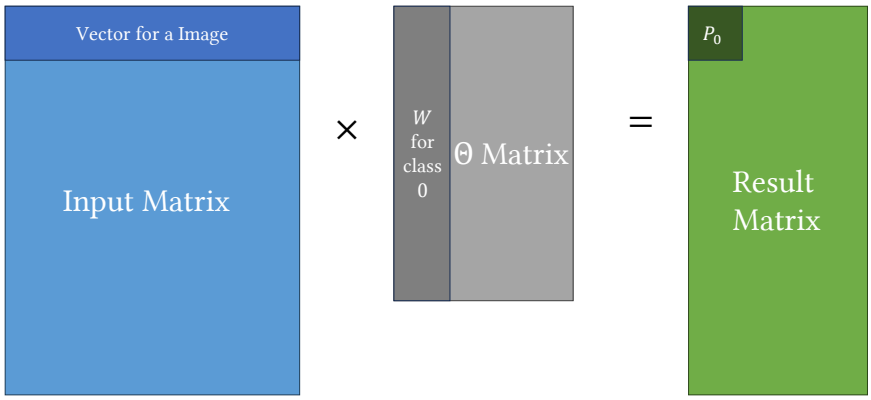

Figure 2: Diagram of task 1

Implement the functions described on GitHub and write the equivalent C++ code.

Python code for Task 1:

```
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
```

Our aim is to finally obtain the theta matrix and then obtain the final result by matrix multiplication. At the same time we do not design thresholds, i.e., the one with the highest probability is our final classification.

2.2 Bottleneck analysis

If we do not apply any tricks and use the simplest i,j,k order for matrix multiplication and `exp` for calculating the powers of e. It ran 1600 ms, so it cannot beat baseline.

After analyzing, the main bottleneck comes from matrix multiplication. After using the techniques in my Project 2 (Transport + Loop Unrolling), we will reach to 9200 ms.

Another trick is that we can use `expf` instead of `exp` since we are not too sensitive to precision.

3 Task2: Accelerate softmax with OpenACC

3.1 Solution

I have already written code for matrix multiplication accelerated with OpenACC in the bonus section of project 2, so I use it directly.

Meanwhile, for other functions like `matrix_softmax_normalize_openacc`, we can use reduce to speed up the operation since we need to get the sum of a `exp` matrix i.e., `Z` in the python code.

3.2 Trick 1: Loop unrolling

In my naive solution in the bonus section of project 2, I didn't use loop unrolling. For this project, I try to unroll loop for it and find that it indeed reduce time, from 1010 ms to 920 ms.

3.3 Trick 2: Copy once, use all the time

This trick is very intuitive, we only copy in `train_softmax_openacc`, all other functions do not perform the copy operation.

3.4 Trick 3: Create temporally arrays directly on device

Notice that we need a lot of temporally arrays in the `softmax_regression_epoch_openacc` function, such as gradient, which we can create directly on the device using the create command to save time.

3.5 Profiling results

Time(%)	Total Time (ns)	Average	Name
66.2	681,770,146	12,592.5	cuStreamSynchronize
31.0	318,935,695	5,895.3	cuLaunchKernel
2.5	25,947,632	25,947,632.0	cuMemHostAlloc
0.1	1,133,503	1,133,503.0	cuMemAllocHost_v2
0.1	1,072,622	82,509.4	cuMemAlloc_v2

Table 4: Overall profiling results of task 2

Time(%)	Total Time (ns)	Average	Name
69.5	441,755,510	73,381.3	matrix_dot_openacc_9_gpu
9.5	60,570,344	10,095.1	matrix_dot_trans_openacc_40_gpu
5.0	31,950,558	2,662.5	matrix_minus_openacc_102_gpu
4.3	27,437,010	4,572.8	matrix_softmax_normalize_openacc_138_gpu
3.0	19,076,667	3,179.4	matrix_div_scalar_openacc_126_gpu

Table 5: Profiling results of task 2 for specific functions

From this we can see that after our improvements, the time spent on COPY does not account for the majority of the overall time. Meanwhile matrix multiplication is still the bottleneck of this task.

3.6 Interesting findings: order of operations for assignment statements

Suppose we now have a loop like this:

```
for (int i = 0; i < n; ++i)
    C += A[i];
```

We have two ways of unrolling:

```
for (int i = 0; i < n; i += 4) {
    C += A[i];
    C += A[i + 1];
```

```
    C += A[i + 2];
    C += A[i + 3];
}
```

and

```
for (int i = 0; i < n; i += 4) {
    C += A[i] + A[i + 1] + A[i + 2] + A[i + 3];
}
```

The results of these two approaches on floating point numbers are not consistent.

For the first one, the compiler doesn't directly swap the order of operations because they both depend on C.

But for the second one, since we're using concatenation, it's kind of an undefined behavior for C++ to prioritize which piece of the concatenated operation goes first and the compiler will rearrange it. In other words, it might combine A[i+1] and A[i+2] first.

So the results will be different.

4 Task3: Train MNIST with neural network

4.1 Solution

Implement the functions described on GitHub and write the equivalent C++ code.

Python code for Task 3:

```
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
```

4.2 Trick: use `mask` to replace `mul`

Let's rethink the below line shown in python code.

```
G1 = np.dot(Z2 - Y, W2.T) * (Z1 > 0)
```

We can use `mask` to replace `mul` here. Since the logic behind this line is when `Z1` is positive, we keep `(Z2-Y).dot(W2.T)`, otherwise, we set it to 0.

```
tmp = np.dot(Z2 - Y, W2.T)
# Enumerating i j
if Z1[i][j] > 0: G1[i][j] = tmp[i][j]
else: G1[i][j] = 0
```

Using the above trick obviously saves computation complexity, since it is always easier to determine who is bigger or smaller than 0 than it is to multiply.

5 Task4: Accelerate neural network with OpenACC

5.1 Solution

Use the functions implemented in task 2 and change the functions in task 3 into OpenACC code, we can easily solve this task.

5.2 Trick: `batch/lr`

In the previous part, I just follow the sample python code and translate them into C++ code. So when calculating the gradients. I first `div batch` then `mul lr`.

However, in practice, `lr` is smaller than 1 but not too small. So we can first calculating `batch/lr` since it will not overflow and `div` them. So we saved the computing complexity for it.

5.3 Profiling results

Time(%)	Total Time (ns)	Average	Name
79.3	4,436,748,589	24,599.3	cuStreamSynchronize
20.1	1,126,217,193	6,247.0	cuLaunchKernel
0.5	26,889,898	26,889,898.0	cuMemAlloc_v2

Table 6: Overall profiling results of task 4

Time(%)	Total Time (ns)	Average	Name
63.7	2,780,857,984	115,484.1	matrix_dot_openacc_9_gpu
26.0	1,136,446,578	47,351.9	matrix_dot_trans_openacc_40_gpu
2.6	113,854,429	4,743.9	matrix_div_scalar_openacc_126_gpu
2.6	112,046,094	3,112.4	matrix_minus_openacc_102_gpu
1.3	56,930,402	4,744.2	matrix_trans_dot_openacc_71_gpu
0.9	39,888,431	3,313.0	matrix_relu_openacc_331_gpu
0.3	11,324,561	283,114.0	mean_softmax_loss_openacc_272_gpu

Table 7: Profiling results of task 4 for specific functions

We can notice that matrix operations are still the largest part of the running time. But for a single operation, mean softmax loss is really slow. This is due to its large inputs combined with frequent internal exp and log operations.

So one idea for improvement is that we try to minimize this operation, maybe we can reduce the size of the value passed by the functions by some mathematical distortions, for example, we calculate all the minimum values first, subtract it from both exp and log incoming values, and then add it externally with the aim of reducing the value passed by the functions.

Another noteworthy point is that my implementation is more than 10x faster than baseline, I'm not sure how baseline is implemented but rather suspect that there are frequent copy operations that cause this.

It can be noted that Memalloc still has a certain percentage because of the high number of epochs, and if don't use the copy once trick, then the runtime should be larger.