SHANGHAI JIAO TONG UNIVERSITY

CS 075

High Perfomance Computing

Final Projects

Yiyiyimu

July 5, 2019

Contents

List of Figures

List of Tables

1 Introduction to SJTU π supercomputer

SJTU $\boldsymbol{\pi}$ supercomputer, which was founded in April 2013, is No.1 supercomputer among China universities in when built and still the biggest GPU cluster among China universities and it's is designed to provide technical support for large-scale scientific and engineering computing needs in SJTU.

$\boldsymbol{\pi}$ is a heterogeneous high-performance computing system consisting of CPU + GPU + FAT nodes with a theoretical peak performance of 343 TFLOPS (CPU 135 TFLOPS + GPU 208 TFLOPS). $\boldsymbol{\pi}$ is also equipped with 3 PB storage system whose bandwidth could reach 13GB/s and 100 MB/s per thread and 56G/s infiniband network with ¡2μsend-to-end delay. The cluster has 435 nodes, including 332 CPU nodes, 69 GPU nodes, 20 fat nodes, 6 storage nodes, and 8 management login nodes. The number of CPU cores is 7000, the memory is 30TB, and the aggregate storage capacity is 5PB. Use 100 pieces of NVIDIA Kepler K20 GPU, 10 K40 GPUs, 24 K80 GPUs, 4 Pascal P100 GPUs, 80 400GB SSD high-speed hard drives, and FDR 56Gbps Infiniband network high-speed interconnect between nodes.

$\boldsymbol{\pi}$ servers more than 140 research groups, covering all STEM schools in SJTU and provides more than 20 million corehours per years (500x bigger than a workstation). Highlight applications includes airplane noise analysis, material genomes, deeplearning-based speech recognition, rice sequencing, plasma physics and etc al.

$\boldsymbol{\pi}$ 2.0 supercomputer was built two months ago in April 2019, which is the first supercomputer to adopt new generation of Intel Cascade Lake Gold Edition processor in China, supplemented by the industry's advanced computing network Intel Omni Path storage system, to meet the full line rate and non-blocking communication requirements in the computing and storage process which could not only carry computing tasks in a temporary computing test environment, and also cope with the demand for large amounts of small file concurrency.

After the upgrade, the peak speed of the floating point calculation of $\boldsymbol{\pi}$ 2.0 supercomputer could exceed 2 PFlops. According to the 52^{nd} TOP500 ranking of the global supercomputer announced in November 2018, $\boldsymbol{\pi}$ 2.0 could rank in top 100. Compared to $\boldsymbol{\pi}$, the performance of VLPL-S nodes is improved by 4.5 times(single node) and 4.7 times(four nodes), and storage performance is also improved by 5 times which reaches 15GB/s.

2 OpenMP

2.1 Background

OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. Compared to Message Passing Interface (MPI), OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes.

Openmp codes that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed. Each thread has an id attached to it which can be obtained using a function. The thread id is an integer, and the master thread has an id of 0. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions.

The OpenMP functions are included in a header file labelled omp.h in C. and the format of most of the constructs is *#pragma omp construct [clause [clause]...]*

2.1.1 Motivation

OpenMP standardizes last 20 years of SMP practice, so it could be a good idea to start parallel computing with OpenMP to calculate the value of π.

Besides the basic requirement, the project also finishes extra tasks including false sharing and different compilers.

2.1.2 False sharing

In symmetric multiprocessor systems, each processor has a local cache. The memory system must guarantee cache coherence while false sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces an update, which would hurt performance.

2.1.3 Different compilers

In this project, two compilers were implemented: GNU Compilaer Collection(GCC) and Intel C++ Compiler(ICC). The difference bewteen them includes GCC is open sourced while ICC is not and GCC could work cross platform. But commonly speaking, ICC has better perfomance than GCC.

2.2 Experiment setups

Besides following the steps on

2.2.1 π calculation

To calculate the value of π, using the knowledge for calculating the circle, we know

$$\int_0^1 dx \frac{4}{1+x^2} = \int_0^{\pi/4} \frac{d\theta}{\cos^2\theta} \frac{4}{1+\tan^2\theta} = \int_0^{\pi/4} 4d\theta = \pi \tag{1}$$

For calculation, we discretize the original equation to get

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \Delta \cong \pi \tag{2}$$

So sample code to calculate π could be

```
void main() {
        long num_steps = 100000000;
        int i; double step,x,sum=0.0,pi;
        step = 1.0/num_steps;
        for (i=0; inum_steps; i++) {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        pi = sum*step;
}
```

So the task is to code for with and avoiding false sharing separately, and use gcc and icc to compile and run.

2.2.2 Resolution of false sharing

Implementing *#pragma omp parallel private(i)* to replace *#pragma omp parallel* to store partial sums into a private scalar. And implementing *#pragma omp critical* before combining local sums to make sure: 1. every thread is done and 2. only one thread combines the local sums into a single global sum.

2.3 Performance Analysis

Two codes can both get the true value of π, while the time consuming and speedup are far from each other.

As we can find in given tables and figures, gcc with false sharing failed to get any speedup with multi threads. On contrary, it takes even more time(around 1.2 times) to finish running. But icc with false sharing didn't occur the same problem. I failed to find possible reasons some others offer, and I guess the reason may be some optimization of icc that could avoid false sharing without specific codes.

Table 1: Comparison of time comsuming in different setup of OpenMP in multiple threads

	gcc		icc	
	With	Avoid	With	Avoid
1	1.634	1.62	0.359	0.388
2	1.917	0.85	0.238	0.214
3	2.004	0.558	0.115	0.112
4	2.141	0.407	0.086	0.087
5	2.315	0.321	0.071	0.069
6	2.000	0.268	0.058	0.057
7	2.230	0.229	0.050	0.050
8	2.080	0.200	0.043	0.043

Table 2: Comparison of speedup in different setup of OpenMP in multiple threads

	gcc		icc	
	With	Avoid	With	Avoid
1	1.000	1.000	1.000	1.000
2	0.852	1.905	1.508	1.813
3	0.815	2.903	3.121	3.464
4	0.763	3.980	4.174	4.459
5	0.705	5.046	5.056	5.623
6	0.817	6.044	6.189	6.807
7	0.732	7.074	7.18	7.760
8	0.785	8.100	8.348	9.023

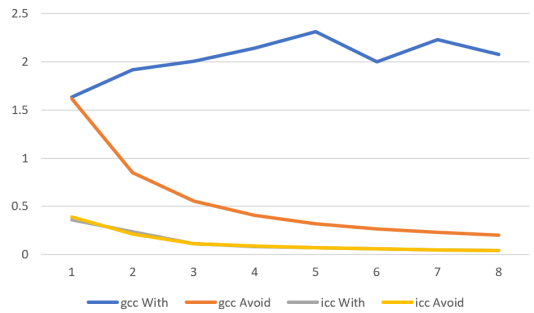

Figure 1: Comparison of time comsuming

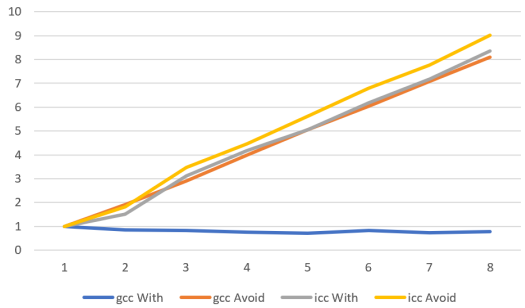

Figure 2: Comparison of speedup

Without considering that, we can find using icc is around 5 times faster than gcc, when specificly avoid false sharing, and from single threads to 8 threads. For icc itself, there is not much difference between with or without false sharing, so it kind of prove that icc itself avoid false sharing.

For speedup, not considering gcc with false sharing, we can find the other three has almost the same pattern of speedup, that *Speedup* $\cong$ *Number of threads*, which accords with Amdahl's Law.

2.4 Summary

1. Using gcc with false sharing could not get speedup with multiple threads.
2. Icc is around 5 times faster than gcc.
3. For who got speedup almost have the same pattern of speedup.

3 MPI

3.1 Background

As talked in OpenMP part, Message Passing Interface (MPI) is used for parallelism between nodes while OpenMP is used for parallelism within a (multi-core) node. MPI is a communication protocol for programming parallel computers. Both point-to-point and collective communication are supported. MPI "is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation." MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few language-specific features. MPI programs always work with processes, but programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or mpiexec.

3.1.1 Motivation

This project is used to compare with OpenMP to find the advantage of MPI, so the task is still calculating the value of π.

Besides the basic requirement, the project also finishes extra tasks including different compilers and different nodes, because MPI could perform on multiple nodes.

3.1.2 Experiment setups

The calculation of π and the difference of compilers is talked in last part. So just following the steps in document of pi instructions.

3.2 Performance Analysis

Due to there is not much difference between the result of using gcc or icc with single node, so multiple nodes were only implemented in icc.

Table 3: Comparison of time comsuming in different setup of MPI in multiple threads and nodes

	gcc	icc		
		1	2	4
1	0.8295	0.8309	0.6882	0.7439
2	0.4522	0.4259	0.4652	0.4582
4	0.1801	0.2162	0.2288	0.2512
8	0.1102	0.1088	0.1130	0.1201

Table 4: Comparison of speedup in different setup of MPI in multiple threads and nodes

	gcc	icc		
		1	2	4
1	1.00	1.00	1.00	1.00
2	1.83	1.95	1.48	1.62
4	4.61	3.84	3.01	2.96
8	7.53	7.64	6.09	6.19

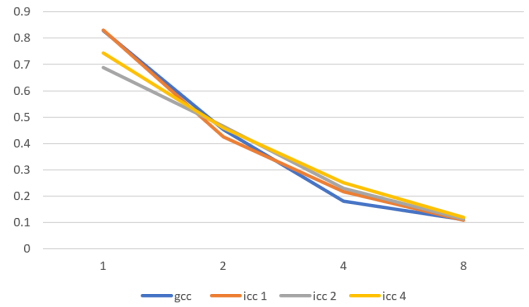

Figure 3: Comparison of time comsuming

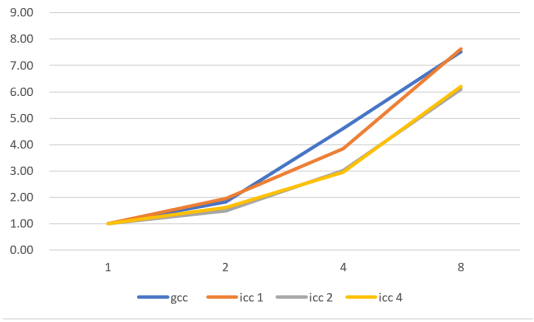

Figure 4: Comparison of speedup

For single node with multiple threads, we can find the result, both time consuming and speedup are around the same between gcc and icc. Focus on time consuming, we can find multiple nodes outperforms single node at single thread, but perform worse when multiple threads were used. One possible reason is that communication between multiple threads and multiple nodes would take more time than it saves with parallel computing.

And the same result could be deducted in speedup table and figure that single node in gcc and icc have the same speedup ratio with 8 threads, which is higher than the other two setups, due to its higher time comsuming in single thread. And icc with both 2 and 4 nodes have the same speedup ratio at different threads, which prove the time consuming between multiple nodes.

3.3 Summary

1. With single node, not much difference in result was found between gcc and icc.
2. Multiple nodes outperforms single node at single thread, but perform worse when multiple threads were used.
3. Icc with multiple nodes have the same speedup ratio.

4 CUDA

4.1 Background

CUDA is a parallel computing platform and API model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled GPU for general purpose processing — an approach termed GPGPU (General-Purpose computing on Graphics Processing Units), not only graphic computing. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

And for this task, library cuBLAS – CUDA Basic Linear Algebra Subroutines library is used to calculate matrix dot multiplication.

4.1.1 Matrix-matrix multiplication

Let's say we have two matrices, A and B. Assume that A is a n×m matrix and B is a m×w matrix. The result of the multiplication A∗B is a n×w matrix, which we call M. That is, the number of rows in the resulting matrix equals the number of rows of the first matrix A and the number of columns of the second matrix B.

Let's take the cell 1,1 (first row, first column) of M. The number inside it after the operation M=A∗B is the sum of all the element-wise multiplications of the numbers in A, row 1, with the numbers in B, column 1. That is, in the cell i,j of M we have the sum of the element-wise multiplication of all the numbers in the i-th row in A and the j-th column in B.

We could easily find this is perfect for parallel computing, due to each cell of M do not effect each other. But time consuming in this circumstance is still not affordable, so we need GPU to use its power.

4.2 Motivation

As a specialized computer processor, GPU addresses the demands of real-time high-resolution 3D graphics compute-intensive tasks. GPUs had evolved into highly parallel multi-core systems allowing very efficient manipulation of large blocks of data. This design is more effective than general-purpose CPU for algorithms in situations where processing large blocks of data is done in parallel. Especially in deep learning, where matrix especially sparse matrix multiplication is highly time consuming, so there is highly need for us to learn a language that could runnning on GPU.

Besides the basic requirement, the project also finishes extra tasks including shared memory, blocking and not square matrix cultiplication.

4.3 Experiment setups

4.3.1 Shared memory

IN GPU, shared memory is on-chip and is much faster than local and global memory. Shared memory latency is roughly 100x lower than uncached global memory latency. Threads can access data in shared memory loaded from global memory by other threads within the same

thread block. Memory access can be controlled by thread synchronization to avoid race condition (__syncthreads). Shared memory can be uses as user-managed data caches and high parallel data reductions. For variables that use static shared memory, we could easily add '__shared__' before normal definition and __syncthreads() to ensure all threads have completed before continue.

4.3.2 Block

CUDA uses blocks and threads to provide data parallelism. CUDA creates multiple blocks and each block has multiple threads. Each thread calls the same kernel to process a section of the data. The GPU chip is organized as a collection of multiprocessors, with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple multiprocessors. For this project, the calculation is not based on each cell but on each block to speed up the calculation.

4.4 Performance Analysis

Blocks is used in both two tasks. For testing the effect of shared memory, two matrixs are 256*256 are randomly generated and two tasks use the same matrixs.

Table 5: Speedup of CUDA with effect of shared memory

	Shared	Not
CPU/s	96.30	101.4
GPU/s	1.831	0.969
speedup	52.8	101.4

As we can find in result, CUDA could significantly speed up the calculation of matrix-matrix multiplication for both square matrix and not square matrix calculation, but the result also surprising shows shared memory failed to improve speedup. Due to data in shared memory would be reused, it's hard to figure out the reason of this result.

Shared memory is implemented in square matrix multiplication, and not squared matrix multiplication didn't use it. So when experiment with matrix size, no shared memory were used for square matrix multiplication.

Table 6: Speedup of CUDA with effect of matrix size

	256*256	256*200
CPU/s	95.82	79.06
GPU/s	0.972	1.074
speedup	98.5	73.5

The result shows square matrix multiplication got higher speedup, just as expected.

4.5 Summary

1. CUDA could significantly speed up the calculation of matrix-matrix multiplication.

2. Shared memory failed to improve speedup
3. Square matrix multiplication got higher speedup

5 Pytorch CNN

5.1 Background

In deep learning, a convolutional neural network (CNN) is a class of deep neural networks which use convolution to reduce size and extract features. CNN are regularized versions of multilayer perceptrons. Multilayer perceptrons usually refer to fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to overfitting data. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. However, CNN take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNN are on the lower extreme.

CNN were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage. They have applications in image and video recognition, recommender systems, image classification, medical image analysis, and natural language processing.

5.2 Motivation

It's the era of Artificial Intelligence, of cource we need to get some basic ideas of CNN.

Besides the basic requirement, the project also finishes extra tasks including comparison between GPU and CPU, and different hyperparameters.

5.3 Experiment setups

MNIST is a famous database of handwritten digits, and usually used for a entrance instances in many books about deep learning. It has a training set of 60,000 examples, and a test set of 10,000 examples. Every example of MNIST datasets is a photo of handwritten number ranging from 0 to 9. The size of an example is 28x28, and every example digit will be located in the center of an image.

The MNIST handwritten digits recognition is a typical task of classification. The implemented model should take a feature vector of 28x28 as input and output an integer label ranging from 0 to 9. And the reason to choose MNIST classification as task is because it could be solved by both CNN and RNN, so we could somewhat get some idea on how they performs on simple image classification problem (although RNN is originally not made for this..).

A simple CNN model is built as:
(conv1): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
(conv2): Conv2d(20, 50, kernel_size=(5, 5), stride=(1, 1))

(fc1): Linear(in_features=800, out_features=500, bias=True)
(fc2): Linear(in_features=500, out_features=10, bias=True)

Activation function is relu function and max pooling is implemented after each convolutional layer.

Both GPU and CPU version is running on k80 node.

5.4 Performance Analysis

Table 7: Comparison of constant learning rate chosen

epoch	learning rate		
	0.1	0.01	0.001
1	0.9851	0.9455	0.6033
2	0.9898	0.9672	0.7920
3	0.9893	0.9708	0.8698
4	0.9892	0.9799	0.8932
5	0.9882	0.9825	0.9075
6	0.9919	0.9850	0.9211

Table 8: Speedup of GPU

CPU/s	398
GPU/s	62.4
Speedup	6.386

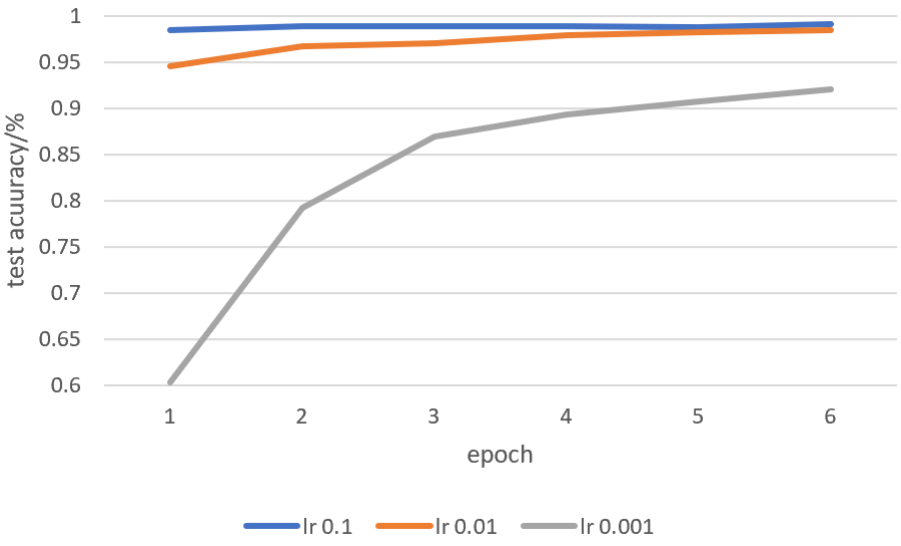

Figure 5: Result of CNN

As we can find in table 8, GPU could significantly accelarate the calculation time in CNN, for about 6 times faster.

For hyperparameters, there is nothing much to adjust in CNN, so learning rate with Stochastic Gradient Descent (SGD) optimizer is chosen to give results that have larger differences. And the result actually prove this. As we could find in table 7 and figure 5, the smaller learning rate, the lower test accuracy, especially at first epoch.

The reason is simple. Learning rate is just like the length of each step. While the goal is to find global mininum point, shorter each step, harder and more time needed to get to get to global minunum point, although the direction is the same. As a result, methods used these day is apply other self-adaptive optimizer like Adam optimizer, or use larger learning rate at beginning, when not much progress could been make, reduce learning rate and keep learning.

5.5 Summary

1. Running on GPU could be 6 times faster than CPU.
2. Smaller learning rate, lower the test accuracy at beginning, especially at first epoch.

6 Pytorch RNN

6.1 Background

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition.

The term "recurrent neural network" is used indiscriminately to refer to two broad classes of networks with a similar general structure, where one is finite impulse and the other is infinite impulse. Both classes of networks exhibit temporal dynamic behavior. A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that can not be unrolled.

Both finite impulse and infinite impulse recurrent networks can have additional stored state, and the storage can be under direct control by the neural network. The storage can also be replaced by another network or graph, if that incorporates time delays or has feedback loops. Such controlled states are referred to as gated state or gated memory, and are part of long short-term memory networks (LSTMs) and gated recurrent units.

6.2 Motivation

It's the era of Artificial Intelligence, of cource we also need to get some basic ideas of RNN.

Besides the basic requirement, the project also finishes extra tasks including comparison between GPU and CPU, and different hyperparameters.

6.3 Experiment setups

MNIST is introduced in last part. And a simple RNN model with lstm layer is built as:
(rnn): LSTM(28, 64, batch_first=True)
(out): Linear(in_features=64, out_features=10, bias=True)

Both GPU and CPU version is running on k80 node.

6.4 Performance Analysis

As we can find in table 10, GPU could also significantly accelarate the calculation time in RNN, for about 4 times faster. But it's less than CNN which is around 6 times faster. Possible reason is RNN is actually not made for image classification so the gap between CPU and GPU could be narrower.

For hyperparameters, there is also nothing much to adjust in RNN, so learning rate with Stochastic Gradient Descent (SGD) optimizer is chosen to give results that have larger differences. And the result actually prove this. As we could find in table 9 and figure 6, the smaller

Table 9: Comparison of constant learning rate chosen

epoch	learning rate		
	0.1	0.01	0.001
1	0.19	0.12	0.09
2	0.46	0.12	0.09
3	0.67	0.12	0.09
4	0.88	0.13	0.09
5	0.93	0.14	0.09
6	0.92	0.15	0.09

Table 10: Speedup of GPU

CPU/s	189
GPU/s	43.6
Speedup	4.339

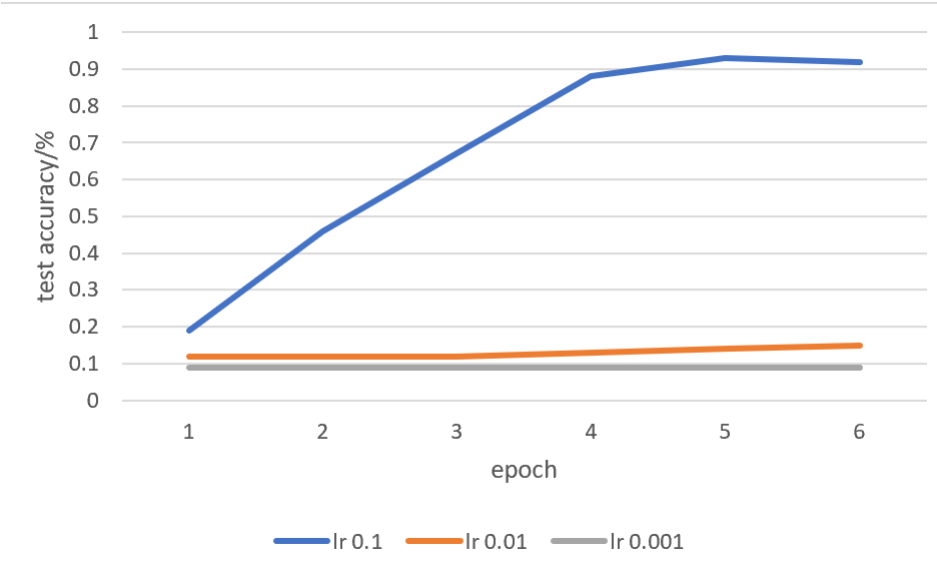

Figure 6: Result of CNN

learning rate, the lower test accuracy at beginning. Compared with CNN, we can find if we use 0.01 or 0.001 as learning rate at beginning, the model even could not learn much things, compared to 0.1 as learning rate.

The reason is the same. Learning rate is just like the length of each step. While the goal is to find global mininum point, shorter each step, harder and more time needed to get to get to global minunum point, although the direction is the same. As a result, methods used these day is apply other self-adaptive optimizer like Adam optimizer, or use larger learning rate at beginning, when not much progress could been make, reduce learning rate and keep learning.

6.5 Summary

1. Running on GPU could be 4 times faster than CPU.
2. Smaller learning rate, lower the test accuracy at beginning, or even cannot learn anything.