GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Practice Report for Systementwicklung in einer forschungsbezogenen Projektarbeit

Exploration for Distributed Learning Design with Golang

Silin Zhao

MatrNr: 21569349

Supervisor: Prof. Julian Kunkel, Sadegh Keshtkar

Georg-August-Universität Göttingen
Institute of Computer Science

April 2, 2024

Abstract

With the rapid development of Artificial Intelligence (AI), the requisition of computing resources stimulates people for powerful training tools. Compare to accelerate the training process on a single host machine, distributed learning offers us an innovative way to deal with huge data volumes.

In this report, we are going to start with the introduction of the concepts of distributed learning, specifically from the traditional Scalable AI design to advanced technologies for gradient updating. And then we will introduce a real-time asynchronous communication mechanism for updating networks in distributed systems and illustrate how this mechanism can be implemented in our model.

Following, we will focus on GPU utilization for the Golang package, Gorgonia. There will be many efforts needed to ensure the compatibility of the model we build with Gorgonia. At first, a network model will be implemented with Gorgonia, and the utilization of GPUs for our model will be enabled. The key part is the configuration of the environment for training our model, and some package limitations will be discussed. We are going to study the differences between with and without singularity for training our model when we launch our model with Slurm in the Emmy cluster. About a fine-turning part of our model is out of the content of our report. Also, the implementation of distributed learning with our model and comparison with other languages can be further investigated for our study.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- ☑ Not at all
- ☐ In brainstorming Found 4 items, similar to cluster.
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☐ Other, namely: -

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC High Performance Computing

AI Artificial Intelligence

HDFS Hadoop Distributed File System

BD Big Data

MPI Massage Passing Interface

NCCL NVIDIA Collective Communications Library

1 Introduction

1.1 Background

Since AI has completely dominated the game of Go in 2010, people start to utilize AI for convenience, such as for day-to-day industry activities. Since Big Data (BD) has also made rapid progress, the reachable data for model training has become more and more generous. Based on those, we are now focusing on the question that how to contribute in such scenario. This report is going to explore the answer of this question in a combination search area of High Performance Computing (HPC) and AI.

In order to accelerate the training process of an AI model, many attempts have been proposed in hardware and software architecture. On the hardware side, people are trying to improve the performance of GPU acceleration with different designs, while on the software side, with the newly invested infrastructure of the HPC cluster, people have already realized that distributed learning can accelerate the training process significantly through parallelism. In our report, we will focus only on the software side, namely, distributed learning.

For distributed learning, we encounter more challenges with distributed learning compared to training our model on a single, powerful host machine. As the first step, we have to decide how to distribute the model we are going to train. For a big model, we can use model parallel, where multiple machines collaborate to deploy the model. So the model can be distributed across different machines, each machine handles its specific part .In each iteration, the model trains the output from the previous part in its respective machine, and provides its output for the next part. In one complete iteration, these parts are executed sequentially cross different machines.

Another more popular distributed learning model is called data parallel, which is designed for data separation on different machines. In this way, our model will be completely launched in different machines independently, and the input data will be equally separated for each machine. As a reference, some hyper-models can combine those two parallelisms, i.e., model parallel and data parallel, for a powerful model with enormous training data. But for our report, we only focus on the data parallel model. We are going to illustrate such a scenario that each machine launches exactly one complete model with separated data. So in this report, we refer to models and machines as having the same meaning.

1.2 The Goals of our Report

Currently, the most famous AI frameworks are almost based on Python, which is a dynamic, high-level programming language. But because of the lack of natural support of concurrency, Python is not highly motivated for high performance application in HPC. Meanwhile, there are some new languages, such as Golang and Julia, which are more effective and suitable for HPC. In our report, we will illustrate the performance of Golang for our topic. Our model is based on the Golang's package Gorgonia[1], it also supports GPU acceleration. The purpose of our work is to present the capability of building deep learning models with GPU acceleration using Golang. We are going to implement a famous deep learning model, ResNet, with Gorgonia and discuss the design for distributed learning with the model we build. Another aim is to explore the performance difference

[1] `https://github.com/gorgonia/gorgonia`

for with and without singularity of our model, also illustrate the performance difference with the comparison to ResNet in Python.

1.3 Report's architecture

This report contains **three parts**: an overview of Scalable AI development with an introduction of gradient updating, an introduction of Golang with its environment configuration and the main part. The main part is about the implementation of a deep learning model with Golang and the illustration of the performance of our model and further discussion, also the exploration for distributed learning model with our model.

In Chapter 2, we will discuss the background of Scalable AI and their corresponding tools for implementation. We will have an overview of currently research about the performance of those implementations and try to analyze the advantages and disadvantages of those implementations. For more benefits, we can see people are trying to improve performance with many technologies, such as cloud computing, parallel computing, and GPU hardware. There is also a significance and critical component for each implementation, namely, how the weights from different machines will be upgraded, here some designs will be introduced. Also we are going to present a new design that is totally asynchronous for weight updating in distributed system in real-time.

Next we will discuss the advantages of Golang in Chapter 3, which will be used for building our deep learning model. Compare to other program languages, we will discuss the benefits of using Golang. A natural support of parallelism will help us for the communication between different threads within a node, such as data sharing with channels during the training of a model. The second topic in this chapter is about gradient aggregation, which is enabled by the Massage Passing Interface (MPI) communication in distributed system, whether asynchronous or synchronous. The last part is the environment configuration. Because of the incompatibility of some packages between local host machine and the computation nodes in Emmy cluster, we can't directly transfer the binary code from local machine to the cluster and run it with Slurm. In order to run our model in cluster, we can use singularity to create an isolated environment. Singularity maps the GPU from the host machine to the isolated environment, so that we can use the GPU to train our model in a compatible environment. Alternatively we can run our model without singularity in cluster. However, a special configuration will be required and will be discussed in detail.

In Chapter 4 we start with the introduction of the model we are going to implement, in order to be able to compare the performance with other programming languages, such as Python, we are going to implement a famous model, namely, ResNet [He+15]. The second important part of this chapter is about how to run the model in cluster. As we want to train our model in two above-mentioned environments, the result and discussion with be presented for both cases. For the key part in Chapter 4, we are going to compare the performance of the ResNet model we implement in Golang with Python. Lastly, how can we build a distributed learning model based on the model we implement in Golang will be discussed in detail. Here we will summarise the preparation in our report for constructing a distributed learning model with Golang. As conclusion we summarise our report in Chapter 5.

2 From Scalable AI to Distributed learning

In this section we start with the topic of Scalable AI at first, and then we discuss the concepts of distributed learning and its benefits later. In this way, an overview of traditional scalable machine learning will be presented firstly. Then we will detail how the weights can be upgraded after the gradient aggregation with two difference architectures, i.e., MapReduce and MPI. Next we will review some research results from Scalable AI to distributed learning. After Horovod is introduced as an innovative algorithm for gradient updating, we present a new asynchronous real-time distributed learning design, and indicate that how this new design can be implemented for our distributed learning model later.

Scalable AI is designed for models that can train with neither small volumes nor enormous amounts of data. With the growing ecosystem of BD, we have more and more reachable data for AI models. How those data can be elastically fed into machine learning model is the key research area of Scalable AI. Meanwhile people always try to reconstruct and optimize the architectures of Scalable AI.

Scalable AI is fully connected to data science. According to the available data volume, scalable AI can elastically adjust its demand for data volume. Compared to the data loading to the model for single machine training, data parallel model in scalable AI will be more elastic and scalable. Depending on the number of parallel-launched models, the input data can be equally divided and fed to each model simultaneously. Because our input data may have different storage and sources, infrastructure of BD offers us many convenient usage scenarios, such as batch mode, real-time models, and hybird model for data processing. In our report we use neural networks, while for other machine learning algorithms, such as SVM or K-Means, it's normally not necessary to consider the model parallel because of its complexity during scaling.

A comprehensive review article[GSJ16] illustrates the scenario in detail for scalable AI. Based on BD infrastructure, a scalable machine learning algorithm can be applied for many application. By using MapReduce, the authors summarized the picture for the implementation of supervised learning, unsupervised learning, and semi-supervised learning with parallel computing.

2.1 MapReduce and MPI

When we inspect the detail of the critical gradient aggregation part of Scalable AI and distributed learning, we have to enable the communication between different models. We can categorize the gradient aggregation into two directions: MapReduce and MPI. At first, we review the concept of the MapReduce mechanisms, and then some corresponding implementations will be presented for demonstration.

MapReduce is a data processing framework for BD analysis with parallel acceleration. In the BD playground, people require such a tool to process enormous volumes of data in distributed file system, such as Hadoop Distributed File System (HDFS). The following steps describe how MapReduce works in general.

1. Mapper: Each mapper is based on one worker node individually and concurrently, and this step starts with the extraction of input data from the file system. Depending

on the implementation of the file system, each mapper only receives a subset of input data and only needs to process that subset of data. As the result, the output of each mapper will be presented in a set of key-value pairs, waiting for further processing.

2. Shuffler: The key-value pairs will be shuffled and even sorted according to their keys. This step is still executed individually at each worker node.

3. Reducer: Comparing to the other two steps above, this step needs the outputs from all worker nodes as input and launches the user-defined function. In the function, the key-value pairs can be aggregated or filtered as defined, and the result is formed together as key-value pairs.

Despite the different implementations of MapReduce, the main work flow is similar. Hadoop has a rice ecosystem, but due to its performance, people are more likely to use Spark.

Another implementation in our report for communication between different machines is MPI. Our models are started simultaneously by MPI on different machines. Each machine has its own unique index, i.e., rank. In the concept of Scalable AI and distributed learning, the weights from different models, which are launched by different machines, are passed through MPI for aggregation and broadcasting. We will introduce some research cases for this implementation as following.

2.2 Related researches for gradients updating

We are going to overview some research cases for gradient updates. At this moment, we do not specify the difference between Scalable AI and distributed learning. The following overviews are about to describe our research case in different directions.

1. **Back propagation neural network with MapReduce** This article[LXL17] implements a 3-layer neural network model for precision and efficiency performance tests with three different platforms, Hadoop, HaLoop, and Spark. Compared to Hadoop, HaLoop can facilitate the expression of iteration with its own programming interface and API implementations for loop control and caching. So we can understand that there is slightly more effective and better performance for HaLoop than Hadoop. Meanwhile, Spark shows a significant advantage for its memory computation.

2. **Neural network with MapReduce on cloud computing** Recently, studies show that cloud computing can offer Scalable AI a suitable solution for its specific demand, i.e., elastic computation resources[Mun23]. This article also provides an overview of the benefits of cloud-based AI architecture from many aspects, such as load balancing, communication strategies, and cloud tools for serverless deployment.

3. **Distributed Synchronous Stochastic Gradient Descent** A Facebook research group published this article[ZX16] for illustrating the weights updating in a distributed system. They declared that since we should nearly linearly scale the learning rate with respect to batch size, a gradual warm-up for the learning rate at the beginning of training for large mini-batch sizes is better than a constant warm-up. They demonstrated in a parallel acceleration research case, the time per epoch drops from 16 minutes to 30 seconds when the mini-batch size varies from 256 to 11264.

4. **Distributed learning Frameworks** Deep learning framework Tensorflow[Aba+16] was developed by Google Brain for a wide range of AI tasks. Tensorflow offers users specific APIs for distributed learning, but enormous customized code has to be investigated, and parameter servers can be used for data sharing within a parallel process. While in article[Dea+12] the authors introduced another framework, i.e., DistBelief. This framework is special for parallel distributed model training because of the support for data parallelism and model parallelism. As for gradient aggregation, DistBelief offers us two mechanisms for different specifications: Downpour SDG and Sandblaster L-BFGS. Both methods use a centralized shared parameter server and can tolerate the speed differences and failures of their distributed replica models. Meanwhile, it is specifically load-balancing implemented for pushing gradients to the centralized parameter server so that the fast models can carry on their next iteration and the gradients of slow models will be sent every few completed portions.

5. **Parallel SGD** [Mam+18] describes an overview of a combination of parameter servers and MPI for synchronous and asynchronous gradient aggregation. The authors use Key-Value-Store to maintain the shared weights with parameter server. As the increasing of the worker number of parallel SGD communication, synchronous SGD, asynchronous SGD, and asynchronous elastic-SGD presented significant variance in performance. The authors presented that asynchronized elastic-SGD reduces the pressure of internet collection and memory. The key difference compared to asynchronized SGD is that asynchronized elastic-SGD groups the gradients from certain iterations for average before pushing and pulling.

6. **CUDA-Aware S-Caffe** GPU-based HPC clusters have been applied to DL model training because of their significant benefits for acceleration. This article[Awa+17] presents a special modified S-Caffe for scale-up within one node with multiple GPUs and scale-out within multiple nodes with multiple GPUs. The authors modified Caffe in many aspects, such as the MPI communication buffer, which has been enlarged to hundreds of megabytes. Compared to other DL frameworks that use parameter servers, S-Caffe implements a reduction tree to avoid the bottleneck of a single GPU with very large buffer aggregation. In this article, the data transferring between GPU and CPU is minimized with the CUDA-Aware API, which enables data collection and broadcasting between GPU and GPU (SC-O). Based on direct GPU-GPU communication, in order to save the idle time between serialized computation and communication, S-Caffe is re-designed to maximize the overlap between computation and communication phases with the non-blocking collective operations (NBC) of MPI-3. Instead of a coarse-grained data communication in a single operation, researchers divide the gradient communication into multi-stages for overlap. Non-blocking broadcasting is called at beginning and returns immediately, and the wait operation is executed at an appropriate time so that neither too soon for poor communication-computation overlap nor too late for degradation of performance (SC-OB). The last approach based on CUDA-Aware in this article is about the helper control thread, called SC-OBR. With the help of this extra thread, researchers can maximize the overlap of the gradients in the communication of the previous layer with the computation of the next layer. As conclusion authors pointed out, with MPI-3 non-blocking operations, SC-OB is about 15 percent better than SC-O, and SC-OBR achieves about 20 percent improvement in time performance

compared to GoogLeNet with 160 GPUs.

2.3 Horovod with Ring-Reducing

In order to accelerate the gradient updating phase, people are always looking for more effective methods for gradient aggregation. As mentioned in article[Awa+17], parameter server and reduction tree are both fully discussed. But in this subsection, we are going to display a ring-reducing algorithm for gradient aggregation, as in [SB18]. The original idea is from a blog of Baidu: *Bringing HPC Techniques to Deep Learning*. As in Figure 1 we illustrate the ring-reducing mechanism with 5 GPUs as an example. After the backward error propagation, each GPU has its own gradients to update. Before the update, we want to accumulate the gradients from 5 GPUs and average them for each GPU for the next iteration. Comparing to the parameter server and reduction tree, ring-reducing can accumulate the gradients from GPUs and update the weights in GPUs in place. There, we list the steps for gradient updating in detail with ring-reducing.

1. Data preparation: Depends on the number of GPUs, the gradients in each GPU will be divided into parts for the same size at first, as in Figure 1, 5 partitions of gradients in each GPU for 5 GPUs.

2. Ring Aggregation: simultaneously, different partitions of different GPUs will be accumulated in the corresponding partition of the next GPU. Similarity, the last GPU will send its partition to the first GPU for accumulation. This operation will be executed for the same partition on the rest GPUs until the same partition on all GPUs has been accumulated. Now we have a fully accumulated gradient partition on different GPUs, and then they will be averaged by dividing the number of GPUs for the next step.

3. Ring Updating: The averaged partition of each GPU will be copied to the next GPU in the corresponding partition simultaneously. This update process works similar as the previous step, i.e., ring aggregation, until all partitions have been updated by averaged gradients.

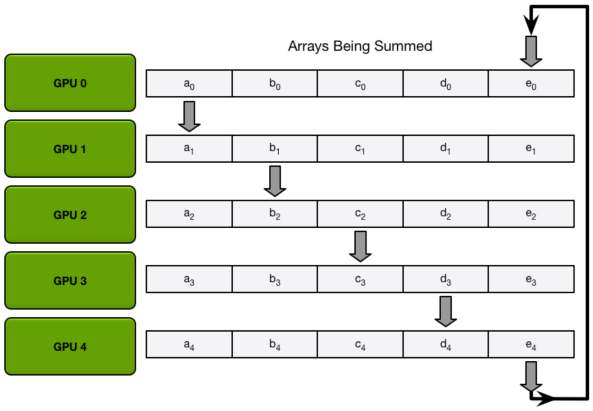

Figure 1: *The ring reducing algorithm for gradient aggregation.*

After the above three steps, all GPUs can start their next iteration with updated gradients. In [SB18] the authors from Uber released a for all Tensorflow version compatible

stand-alone Python package *Horovod*, which is implemented with NVIDIA Collective Communications Library (NCCL). In this way, the three steps we mentioned above for Ring-Reducing can be executed from GPU to GPU.

As the last part we want to point out, even with ring-allreduce in CUDA-aware, we can minimize the time and space consumption of gradient updates in distributed systems, but this design still has some drawbacks. First, all GPUs have to be ready for ring-allreduce operation, which means those operations are not asynchronous. For distributed systems, we should expect that synchronous operations will have a potential idle time. The second difficulty is that ring-all reduction requires more effort for multiple GPUs in multiple nodes, because this requires GPU-GPU communication internal nodes and between nodes, which require MPI-OpenMP architecture.

2.4 New approach for mode updating

We are going to explore a new gradient update mechanism for distributed learning with a parameter server that is totally asynchronous and failure-tolerant for accidents in distributed system. In order to reduce the complexity of our report, we only use MPI for nodes communication, such nodes only launch one GPU for model training.

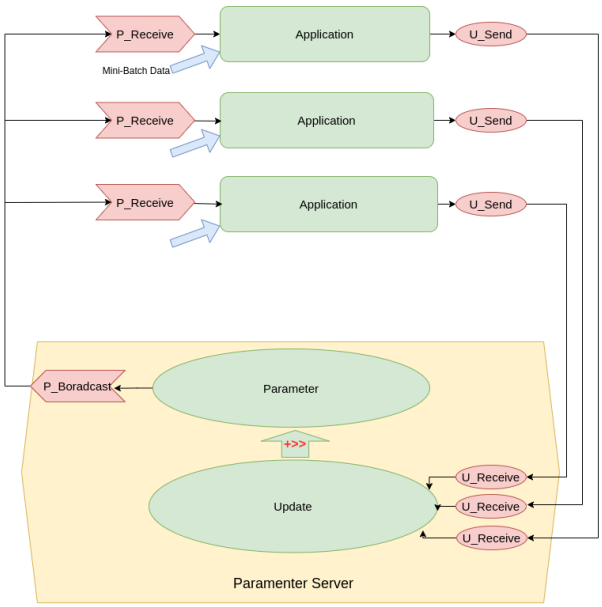

Figure 2: *The illustration of real-time asynchronous gradient updating in distributed system.*

In Figure 2 we can see that this design is divided into 2 parts, i.e., a distributed client and a parameter server. Each distributed client has three threads, a receiving thread, a sending thread, and an application thread. The parameter server also has three threads: a receiving thread, a broadcast thread, and an updating thread. We will list the details of this approach below.

1. **Distributed Client:** This part is illustrated in the top of Figure 2, for simplicity we only show three clients for discuss.

   - Receiving part
     This part is indicated in the left of each distributed client in Figure 2 with red arrow-form mark, which is denoted as *P_Receive*. Receiving part is only

responsible for receiving the new broadcast weights from parameter server and update the weights from broadcasting in shared memory for application part. Totally independent from other parts, the receiving of new weights runs immediately if it receives a signal from broadcasting operation of parameter server.

- Application part
  The application part trains our model with GPU. Before an epoch starts, application thread requires the new training data, which will not be discussed here, and the newest weights from receiving part. By the way we can expect that there is no idle time between epochs so that we can maximumly utilize GPU. So the next epoch can be started with new data and newest weights from receiving thread immediately.
- Sending part
  After every epoch has been done, sending thread will copy the locally updated weights from application thread, and send the weights to parameter server.

2. **Parameter Server** Parameter server has been marked in Figure 2 as the bottom part with yellow background, which also contains the following 3 parts.

   - Receiving operation
     Receiving operation in parameter server receives the weights from the sending thread in distributed clients.
   - Updating Operation
     After the parameter server receives the weights of any distributed client, they will be accumulated to the old weights, a bit-shift to left for average to replace the old weights in parameter server.
   - Broadcast operation
     The last operation in parameter server broadcasts the weights back to receiving part in all distributed clients.

For our approach the same initialization has been broadcast to all models from parameter server at first, and then the model training is started with their corresponding data in the distributed clients. Each epoch starts with the newest weights from the receiving part and training data and ends with updating the weights with gradients. Each distributed client is totally asynchronous and failure-tolerant with respect to other distributed clients. While on the parameter server, we have three operations, which waits for new weights from any distributed client and executes the accumulation, bit-shifting for average and broadcasting.

For the implementation of this approach, we can start with the parameter server. Since they are only addition operations and bit-shift for average, we can expect that the bottleneck should be the data receiving and broadcasting. In the distributed client, we also have 3 parts that need to be implemented. We can use different threads for each part so that the weights can be managed by shared memory. In this way, we can fully utilize GPUs for training in application threads, and we can expect that there will always be new weights available in shared memory. During the model training, the receiving thread updates the weights again and again from the parameter server and waits for the application thread.

By this design, what the model learned from the previous epoch has been sent to parameter server. But for next epoch, those information possibly still not be included in

the new weights from receiving part. This doesn't bother our model, because those information has been already sent to parameter server, receiving part in each distributed client should be able to reach them immediately. Besides, this design is perfect for federated learning. The distributed client can be distributed devices over the world. The receiving part of those distributed devices pull always the newest weights from parameter server.

Because we have discussed CUDA-aware MPI communication before, we can also imagine that we can deploy a GPU on the parameter server, so this approach can also benefit from GPU-GPU communication. Meanwhile, the updating operations in the parameter server are considered to be very suitable for GPU instruction.

Lastly, let us view the drawbacks of this approach. Since we are looking for totally asynchronous communication in distributed learning, all distributed clients should update their weights directly from the parameter server. Once the parameter server broadcasts the new weights to every clients, the client has to update its weights again and again until the application thread uses them, enormous communication has been wasted.

3 Golang

With the development of Big Data, the mostly famous AI frameworks have been rapidly governed by Python because of their simplicity and convenience. Also, because of its powerful REPL tool, Python can offer user a fine-sculpture playground to explore the code execution. But as for the playground of HPC, Python is absolutely not the best choice for its performance. Python do have MPI package for distributed system communication, but it still needs to call the MPI API with C++ API. Also for GPU support, Python still need to call C++ API for CUDA, such as Tensorflow or Pytorch.

Golang is a statically typed, compiled programming language created by Google. It's famous for its simplicity, efficiency, and strong support for concurrent programming. We are going to explore the benefit of replace Python with Golang for AI task. Since our goal is to establish the base for Golang to distributed learning, let us go over some of its features in greater depth.

3.1 Goroutine and channel

As a totally new-designed language, Golang has been assigned many new features, such as simplicity, efficient compilation, etc. But we are more likely to be interested in its support for concurrent programming by nature. Goroutine is designed for launching a thread only by calling the keyword `go` at the beginning of function definition. Compared to multi-threads programming in C or C++, Goroutine starts a new thread much more elegantly, just like calling a normal function. If the system has multiple cores, Goroutine can assign the called thread to other cores for balance automatically.

Similarly, in the C-like language, we use shared memory with a mutual exclusion lock for multiple threads communication. Golang uses a channel for sending the data between threads. Based on the combination of Goroutine and channel, Golang offers us a terrific platform for concurrent programming.

3.2 MPI and CGO

MPI[2] is designed for massage passing in a distributed system, in which each node has its own memory address space. In practice, each process, which is called by MPI, can be treated as an individual running process, and those processes can parallelly run on multiple cores of one host machine, or multiple nodes in a supercomputer cluster [Zho20]. The communication between those processes can be classified into two difference types, namely, non-collective (point-to-point, such as sending and receiving), and collective (all-to-all or all-to-point, such as reducing).

Each process has its own rank, which is the identifier of those processes. For point-to-point communication, its destination and provenance process should be identified by calling. Those codes can only be executed in a code block, which we specially identify. While collective directives do not need to be in such a special code block, which means all processes should be able to execute this instruction, for example, broadcasting and reducing.

Because our MPI is implemented with C language, in order to be compatible with this requirement, we have to use the CGO package. This is because of a special feature of Golang, i.e., Golang is naturally compatible with C language. This package allow us to execute C language code within Golang. So as one of the dependencies for our report, MPI package for Golang through CGO has been already available here [3].

3.3 Golang with GPU

Since the acceleration of GPUs has been applied to AI model training, the benefit of this has become more and more apparent. Compared to the universality of the CPU for general purposes, a specifically optimized GPU can benefit those steps for our module training, such as the arithmetical operations for model forward training and error backward propagation, through all neural network layers. In article[GSJ16] the utilization of GPU is not well considered. No matter which implementation of MapReduce is used for Scalable AI, the use of GPU may not be suitable for this scenario. This is the limitation of traditional Scalable AI. In order to have a better performance, in our report, we will consider GPU utilization as an important and necessary factor for further discussion.

Fortunately, with the rapidly development of Golang, Gorgonia developed *cu* package for GPUs acceleration. Compared to other AI frameworks, which use Python, Gorgonia only need to add the instruction *-tags='cuda'* for compiling. For more description please refer to Appendix A.2.

Before we move our journey to next level, let me briefly describe the GPU environment in cluster and local host machine. GPU environment includes GPU programming and compiling tool, i.e., CUDA toolkit, and execution tool, i.e., CUDA driver. CUDA toolkit can be manually installed in local host machine and it is fundamental required for Golang compilation for GPU-version binary code. However this tool is not provided in any cluster, the failed attempt for installing CUDA toolkit in cluster can be found in Appendix A.3. So we can only transfer the well compiled GPU-version binary code to cluster. While CUDA driver is very generous. In Emmy cluster we have multiple versions can be loaded from package management tools, such as Spark. Besides in local host machine, CUDA driver is already available if CUDA toolkit is well installed. For more detail please refer

[2] `https://www.open-mpi.org/`

[3] `https://pkg.go.dev/github.com/cpmech/gosl/mpi`

to Appendix A.1.

3.4 Environment configuration

In this subsection we are going to start with environment configuration, which contains many attempts and failures. We will also discuss the GPU utilization of our model and singularity support. For our report we are able to use the Emmy cluster in Göttingen, which is hosted by GWDG and GPU-available[4]. Because of the static compilation of Golang, we expect that we can compile the Golang code for binary code in local host machine at first, which is GPU boosted. And then the GPU-version binary code should be transferred to cluster and launched by Slurm commands in cluster.

For environment configuration we are going to explore 2 directions, i.e., with singularity and without singularity. With singularity we can profit a well isolated environment for code execution. In this way we can launch our model in such a isolated environment in different machines in cluster. The second direction, i.e., without singularity, is also a terrific option for our purpose. Unlike the dynamical program language Python, after the static compilation of Golang, we only need to run the GPU-version binary code in cluster. Basically we can train our model just like launching normal HPC tasks. Compared to with singularity, this option saves enormous cost for running a singularity in each machine. All performance is only used for model training.

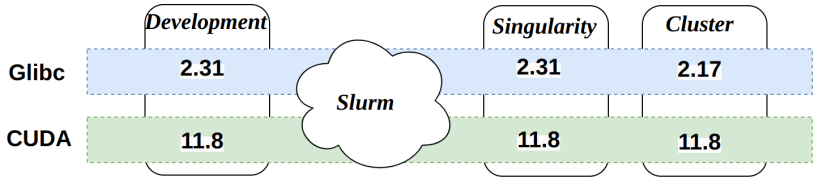

Figure 3: *The illustration of the environment of how we can execute the code with singularity in cluster. The development environment is Ubuntu 20 with latest CUDA version of 11. The singularity in cluster can offer us a appropriate Glibc version according to our development environment.*

Our first architecture is that we compile our Golang code in development environment to GPU-version binary code and then transfer the compiled binary code to cluster for execution with singularity. With this design a very serious problem has been solved implicit, i.e., Glibc version incompatible for compilation and execution. The Glibc version for execution has to be equal or higher than the Glibc version of compilation. As shown in Figure 3, in our development environment we have Glibc version of 2.31, but in cluster Emmy for execution Glibc package is 2.17. So we need singularity to construct a isolated environment for Glibc 2.31, so that the compiled binary code can be executed in cluster. For a detail description of Glibc in different frameworks please refer to Appendix C. For this design, we have to declare that we have two MPI schema, outside of singularity and inside of singularity. Please see a script for launching Slurm with singularity in Listing 2 of in Appendix B.

The second architecture is without singularity, because of the above mentioned incompatibility, we have to specifically deal with the Glibc version problem. There are five attempts as following.

[4]For our project we use A100 GPU with 80 GB.

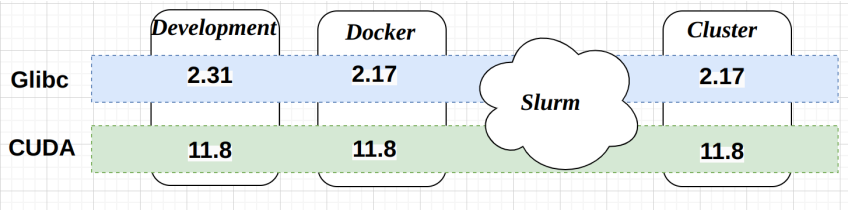

Figure 4: *The illustration of the environment of how we can execute the code without singularity in cluster. The development environment is still Ubuntu 20 with latest CUDA version of 11. The docker in our development environment can offer us a appropriate Glibc version for code compilation according to the Glibc version in Cluster.*

1. Checking other cluster for Glibc friendly version: Beside Emmy cluster in NLRN, GWDG also hosts another cluster, which is called SCC. Unlike the nodes in Emmy cluster has operation system of *CentOS 7.9*, the nodes of SCC use *Scientific Linux release 7.9 (Nitrogen)*, which has also Glibc version of 2.17.

2. Using execution environment in cluster to compile the code: This attempt is not possible, because it require CUDA toolkit to be installed in cluster. For more detail please refer Appendix A.3.

3. Installing Glibc 2.17 package: If we can successfully install Glibc 2.17, we can use *go build -ldflags* to specify Glibc version. But after cloning the source code [5], the installation is complex and failed for lacking of multiple dependencies. As a briefly description for those failures, *Autoconf* is needed for version of 2.53. Even the new installation of *Autoconf* from source code of this version is done, this complain still remains. Another problem is about *gcc*, Glibc 2.17 source code installation requires *gcc* 4.6, which is too old for our development environment. This attempt has been given up for so many difficulties.

4. Installing operation system for glibc 2.17: As in the Table 1 of Appendix C, *CentOS 7.9* has Glibc of version 2.17, and this is also the operation system of the nodes in Emmy, so this attempt is install *Centos 7.9* in local host machine. The installation of *Centos 7.9* is straightforward, but the installation of CUDA toolkit complains a lot.

   (a) Firstly disable *enouveau*

   (b) And then update the *kernel-devel* and *kernel-headers* dependencies

   (c) Last step is to close all *X-server* applications.

   After all those steps are done, we can only install the CUDA toolkit with *runfile*. But this attempt works unstable, because after I updated some packages for CUDA toolkit 11.4, it started to complain that the latest CUDA driver is needed. Weather it is stable for CUDA toolkit 11.8, more investigations is necessary.

5. Create Glibc 2.17 with docker image: The last attempt is more elegant and convenient, just like we make a isolated execution environment with singularity for code execution, we can also create a isolated environment for compilation. Docker image can help us exactly in this case. Such a docker image should based on *CentOS 7.9* as

[5] `https://github.com/apc-llc/glibc-2.17.git`

we mentioned above. Am 10th. November 2013, NVIDIA released many docker images in DockerHub, one of them names *nvidia/cuda:11.5.2-cudnn8-devel-centos7*, this is exactly what we need. Because we still want MPI dependence and Golang environment in our docker image, a customized *Dockerfile* file can be found in Listing 4 in Appendix B for our requirement.

As in Figure 4, we use *Centos 7.9* based docker image to compile our model in development environment, and then transfer our binary code to Emmy cluster. The training process can now be launched as usual HPC task in cluster. Just like in Figure 3, also in Figure 4 we can see that the dependencies are compatible at both sides of Slurm, which stands for compilation and execution. A Slurm script to run our model can be found as Listing 3 in Appendix B.

4 Distributed learning with Golang

In this chapter we are going to start our main part of our report, firstly we will explore the implementation of a famous AI architecture, i.e., Residual Network. And then we will try to train our model in Emmy for with singularity and without singularity, and illustrate their performances.

4.1 ResNet

Before the transformer model has been proposed for AI [Ash17], deep learning models are all built on neural networks for different architectures. Up to now transformer model has achieved the-stats-of-art performance for NPL and image classification[Kol+20]. Since Gorgonia [6] does not support the transformer model, we are going to implement the best model before the transformer model, i.e., the ResNet network.

ResNet stands for residual network, which is specially designed for deep neural networks because of gradients varnish and explosion. Gradient explosion and vanishing are the common problems for neural network training. Gradient explosion can be caused by some improper hyper-parameter, such as learning rate. As for gradient vanishing, some processes can mitigate this problem, such as normalization. However as the layer increases, gradient vanishing still remains. Residual block can avoid information loss by adding the original input data again to the output data of that block, this is displayed in Figure 5. Before the active function layer we can see that the output data $F(x)$ has been accumulated to original input data x. But the shape of input data and output data might be different. We can use 1×1 convolutional kernel to adjust the shape of the original input data.

How to implement ResNet? Tensorflow and Pytorch call corresponding C++ API of *libtorch* and offer user a very convenient function. We will overview the implementation of Golang with this design at first, and then describe how to build ResNet from scratch, without C++ API of *libtorch*.

[6] `https://github.com/gorgonia/gorgonia`

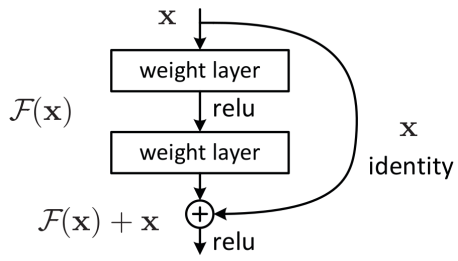

Figure 5: *The illustration of residual block in ResNet, this image is original from [He+15]*

4.2 Gotorch

Just like Pytorch, Gotorch [7] is not active any more since 2020, so the official support of CUDA version is only 10.1 an 10.2. Therefore we have to switch operating system to Ubuntu 18 for CUDA 10.2 according to the compatibility. However for customization we can try to modify *build.sh* file in *cgotorch* folder for different *libtorch* version. This file is responsible for downloading *libtorch* dependence and offering us their API through Golang wrapper functions. This step is the core component of Gotorch, which delivers the possible functions from *libtorch* for Golang. As configuration, the *libtorch* dependence and generated Golang libraries *libcgotorch* should be included in *LD_LIBRARY_PATH* for compiling and execution, and this will become a unsolvable problem in cluster. Besides CUDA toolkit development environment and *Opencv* as dependencies should also be guaranteed for Gotorch. In cluster we have to configure the following dependencies,

1. *Opencv*: We need to compile *opencv* from source code with *-prefix*, and then copy the installation folder to cluster and add the address to *LD_LIBRARY_PATH*.
2. *libcgotorch*: This is the generated Golang libraries in configuration step, also need to be transferred to cluster and added to *LD_LIBRARY_PATH*.
3. CUDA driver: We also need to load CUDA module as dependence.

Even all above mentioned configuration has been guaranteed, it still complains for dependence missing, such as *libdc1394.so.25* and *libgtk-3.so*. This is because of the different kernel versions and dependencies of different operation system for our case, namely, *Centos 7.9* in cluster and *Ubuntu 18* in local host machine. Up to this step, we identify that this architecture is excluded. We need to execute our model without dependencies in cluster. For further investigation, weather with isolated environment, such as docker or singularity can solve this problem, the further investigation is needed.

4.3 Gorgonia

In this subsection we start to construct our model from scratch with Golang, namely, we discard *libtorch* dependence for CUDA calling. Let's have a briefly overview of Gorgonia.

Gorgonia is an AI framework based on Golang, which is similar to other famous frameworks such as Tensorflow or Pytorch. Similar to Pytorch, Gorgonia provides us also

[7] `https://github.com/wangkuiyi/gotorch`

automatic differentiation tools for error propagation. Analogically to Numpy, Gorgonia implement a powerful tensor library[8] that enables multidimensional tensor operations. Meanwhile, GPU support is also provided by Gorgonia[9]. This package is totally in accord with the requirements of our report. Besides for GPU support, user has to explicate the tag for compiling in Gorgonia with *-tags='cuda'*. The CPU operations will be maximally replaced by GPU operations according to the offered functions in *cu* package.

Similarly to Pytorch, the network in Gorgonia is presented with a graph. In this graph, we use nodes for variables such as input data and layer weights. The operation combines the nodes for new node states. All those nodes and operations have been managed by a machine, which can launch the whole graph for training and prediction. We are going to implement ResNet with Gorgonia. For such a deep learning network, we need fully connected layer, convolutional layer, pooling layer, normalization layer, and active function. Fortunately, all those operations have been implemented in Gorgonia.

Even Gorgonia offers us all such layers, but we have to manually configure the weights for some layers, such as convolutional layer and normalization layer. This makes Gorgonia significantly different than Tensorflow or Pytorch. Therefore we need more efforts to organize the code, we call this step as *weights immigration*. As for benefit, this design can help us to build our model into a distributed learning system. After each epoch of training, weights updating in each model will be exactly holed by this manually configuration part. This is the key part for build a distributed learning system. For more details please refer the implementation part following.

4.4 Model implementation

As mentioned above, more investigation will be needed to construct ResNet with Gorgonia. Therefore we want to introduce another package that is call *GoDL* [10], it is fully based on the top of Gorgonia. Even this package is also inactive for more than 2 years, but we are fully inspired by its organization to implement ResNet. *GoDL* has successful implemented *TabNet* and *VGG16*. Now *VGGFace2* is in progress, but *ResNet* is not started. Our implementation can be treated as contribution for this package.

As designed in *GoDL*, our model and the blocks in our model will be treated as *Module* type, which must have function *Name* and *Forward* to be implemented as methods. Based on the configuration of model and block, function *Forward* will decide how the model should be trained.

Before we display the whole process of implementation of ResNet, at first let's briefly overview the architecture of ResNet as shown in Figure 6,

The ResNet series (ResNet18, ResNet34, ResNet50, ResNet101 and ResNet152) has similar structure, but the last 3 designs, i.e., ResNet50, ResNet101 and ResNet152 are more likely to be grouped as the same architecture with different size and we are going to implement this architecture. This architecture contains the following 3 parts.

1. The beginning convolutional part: This part contains one convolutional layer and max pooling layer, seeing Listing 7 in Appendix D.

2. The main part: This part contains 4 blocks, the number of total layers, i.e., stage, in each block is determined by *resnetSerie* in configuration file. Also in each block

[8] https://github.com/gorgonia/tensor
[9] https://github.com/gorgonia/cu
[10] https://github.com/dcu/godl

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 6: *The illustration of the architecture for ResNet serial, this image is original from [He+15]*

the first stage is always with `downsample` for residual connection, and the other stages are considered only to be with convolutional layers and normalization layers. Seeing Listing 8 in Appendix D.

3. The fully connected layer part: As the last part of ResNet, there is only two components, `GlobalAvgPool` and fully connected layer to output as in Listing 9 in Appendix D. But the implementation of `GlobalAvgPool` is still not fully correctly implemented, we use normal max-pooling layer with kernel size of 7×7 for our case for the same functionality.

No matter for the whole model or the stage component, which is denoted as block in code, we specify their workflow in `Forward` method. But in the initialization phase we have to manually configure their weights as Listing 10 in Appendix D (`weights immigration`). This slot is exposed from Gorgonia on purpose. So that we can manually update the weights after weights aggregation and broadcasting for distributed learning system.

4.5 Model training and performance

In this subsection we are going to demonstrate the performance of our model. But the content of the manifestation of distributed learning for ResNet in Golang with real-time gradient updating will not be included. Because as the content limitation of our report, we do not have any investigation for that, and this report is basically designed as the preparation for it.

For training data we use ImageNet100 [11], which is the subset of ImageNet, about 17 GB. The data label has only 100 classes. In the demonstration of our report, all training data has been loaded before training start, the next epoch trains the same data as before. Because in our model we still do not have stream data processing function, such as `DataLoader` in Pytorch. For simplicity we have to load all data at once.

As in the configuration file, Listing 6 in Appendix D, We list a few attributes for explaining. For our environment we can only load a small subset of the training dataset[12]. it's 325 examples, about 40 Megabytes, some machines in the Emmy cluster may take an incredible long time before they start to train if we increase the amount of this training

[11] `https://www.kaggle.com/datasets/ambityga/imagenet100`

[12] It's `1/400` randomly selected examples from the training dataset in all 100 classes

data to be loaded at once. Data loading will be significantly improved for later distributed learning implementation for larger training data. `resnetSerie` indicate that we use different model for training, 1, 2, and 3 respectively stands for ResNet50, ResNet101 and ResNet151. We execute the validation after each training epoch, and launch our model for 6 epochs.

Listing 1: *A snippet of configuation file for ResNet model*

```
...
epoches=6
validateEvery=1
resnetSerie=1  # 1 for ResNet50, 2 for ResNet101, 3 for ResNet152
...
```

For the demonstration for the performance of ResNet, we are going to explore the performance of our model in CPU vs. GPU at first. And then the performance for with vs. without singularity will be displayed with respect to different batch-size. At last, we are going to illustrate the performance comparison of our model in Golang with that in Python.

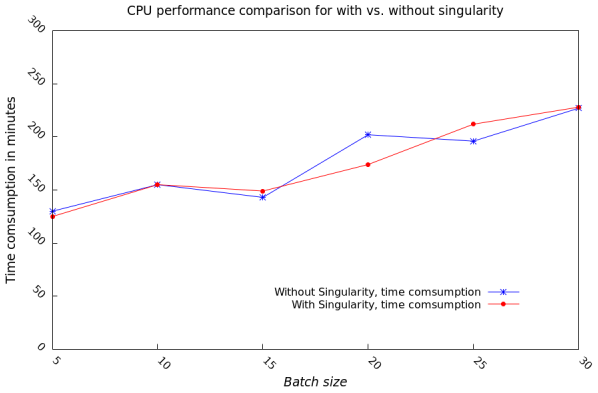

Figure 7: *The illustration of the time consumption of training our ResNet model in CPU for with singularity(red) vs. without singularity(blue), with respect of increasing batch-size from 5 to 30.*

In Figure 7 we present the CPU time consumption for training our model with singularity(red) vs. without singularity(blue). Our batch-size has been increased from 5 to 30, since we can also see that the time consumption is also increased for both cases. It indicates no big performance difference for both cases as the time consumption has alternately increased. Those performances are measured in minutes. Since we already discovered the tendency and the training is extremely heavy on CPU, we stopped our exploration up to batch-size of 30. Since we do see a mark-able difference between with singularity and without singularity in CPU training, we can estimate that two different operation systems, i.e., `Ubuntu 20.04` and `Centos 7.9` have about the same CPU instruction performances for our model.

In the left of Figure 8 we illustrate the time consumption comparison of CPU and GPU with respect of different batch-size for with singularity. With the increasing for batch-size, the time consumption of our model in CPU is increasing, while the time consumption of GPU stays almost the same. Up to batch-size of 30, the time consumption in CPU is about 5 times more than in GPU. Meanwhile the dashed line with dots indicates that the

GPU consumption is from 3 GB linearly increased to 28 GB with the increasing of batch size from 5 to 50.

While in the right of Figure 8 we displayed the same performance index for without singularity. We see also the continuous increasing of time consumption when model runs on CPU. Similarly, the time consumption for the model on the GPU stays almost the same, and almost linearly increasing GPU consumption occurs as the batch size increases.

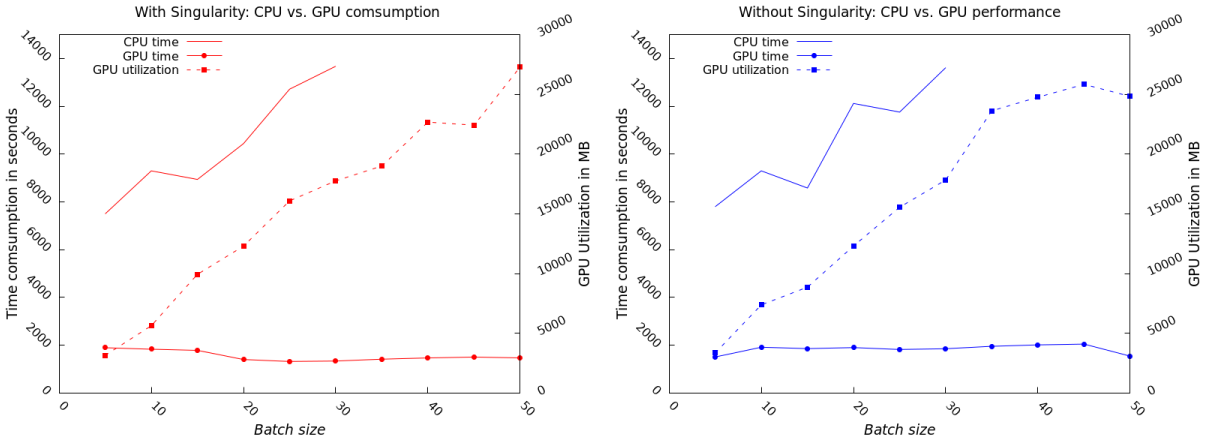

Figure 8: *The illustration of the performance for running our model on GPU with the increasing of batch-size, also the CPU performance is presented for comparison. Red for with singularity and blue for without singularity*

In Figure 9 we can see a significant difference of GPU time consumption for with vs. without singularity. For the same model, in our case, RestNet50, the GPU consumption with singularity(red) is less than without singularity(blue) for batch size bigger than 30. Moreover, the time consumption for with singularity(red) is also less than without singularity(blue).

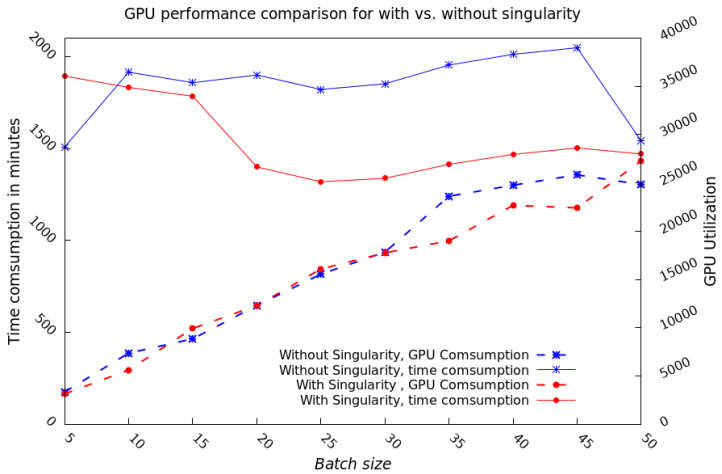

Figure 9: *The illustration of GPU consumption for with singularity(red) and without singularity(blue). Solid line stands for time consumption in minutes on the left side, and dashed line stands for GPU consumption on the right side.*

Let's take a glance about the performance difference between with and without singularity. The vast majority of reasons is about the different operation systems. With singularity we use *`Ubuntu 20.04`* which has Linux kernel version of 5.4, while without

singularity we use *Centos 7.9* which has Linux kernel version of 3.9. We can image that kernel version of 5.4 include performance improvements and dependencies optimization. Special for our ResNet model training with CUDA, newer kernel version provide us certainly better GPU utilization for GPU. This is the reason why without singularity to train our ResNet mode needs more GPU but has a worse time consumption performance.

As our model has been successful implemented and trained in HPC cluster, a performance comparison with other famous framework should be included. We are going to compare the time consumption of our model in Golang with the same mode in Pytorch. A detail description of the model training with Pytorch in HPC cluster can be found in Appendix E.

In Figure 10 we present the time consumption for training ResNet model in Golang for with and without singularity and ResNet model in Python(violet line). We listed all values in seconds for those three cases with respect of the increasing on batch size. We also use logarithm to scale the y axis because of the enormous difference. The results show that with python the time consumption is about only about *1/30* comparing to ResNet in Golang without singularity about *1/20* for with singularity. This result is also accord with the results of with singularity from V100 GPU in Emmy cluster as illustrated in Figure 11[13]. Even there are some small fluctuation for Python, but the performance domination of Python over Golang is obviously. Meanwhile a small insight should be noted, the time consumption for Golang stays almost the same, but it increased slightly for Python.

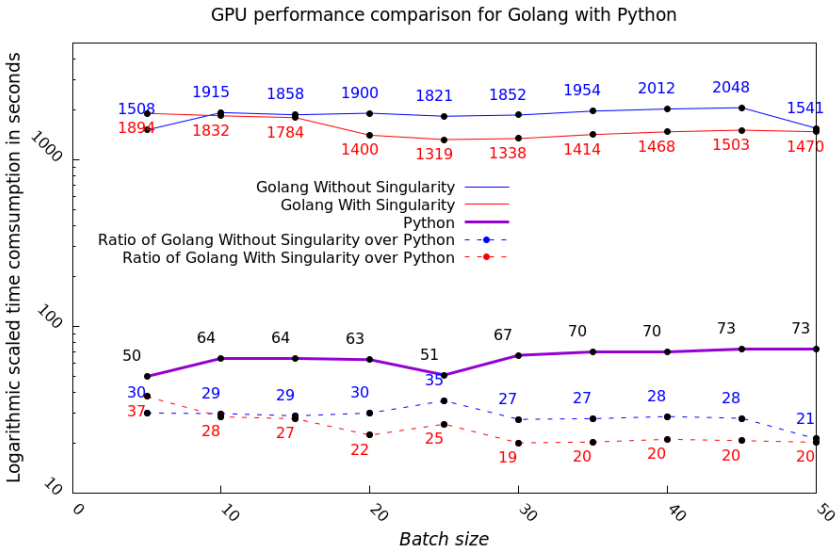

Figure 10: *The illustration of time consumption for ResNet50 in Golang and in Python, and the y axis is the logarithmic scale of 10 in seconds.*

As the last part of this section we want to discuss is the reason of the significant performance difference between Golang and Python. The first reason we can declare that Python has abundant package in its ecosystem for performance optimization, such as for data processing. Secondly, as the C/C++ API of Pytorch has been well parallelly implemented, even not in distributed system, Pytorch can accelerate the training process with multiple cores. The GPU consumption is much more large than Golang. We do not present this index in our report because the exactly number of GPU consumption

[13]V100's VRAM has only 32 GB, so our batch size can only be extended up to 20.

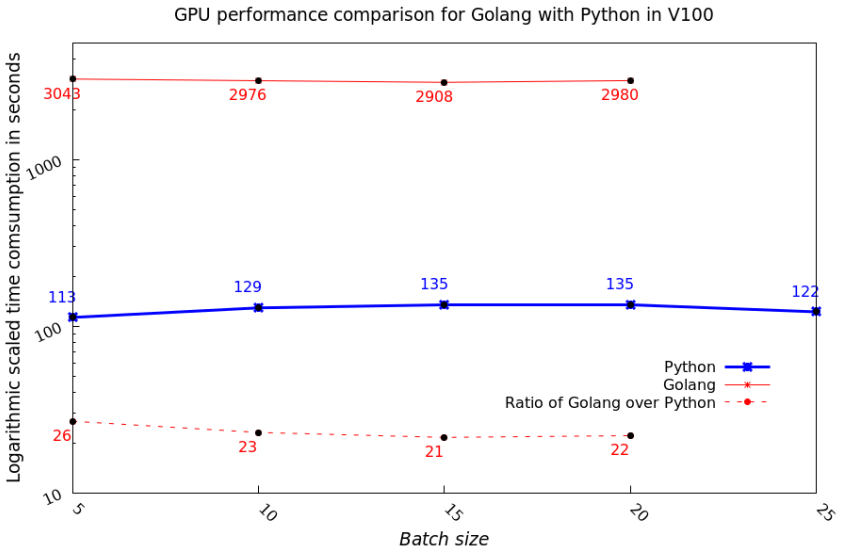

Figure 11: *The same illustration of time consumption for ResNet50 with singularity in Golang and in Python on V100 GPU.*

fluctuates very much(from 20 GB to 60 GB). The third possible reason is that the CUDA instructions for Pytorch are better optimized than Golang.

4.6 The design of distributed learning

This subsection we want to explore the design of distributed learning model based on our implemented ResNet model in Golang. After the data has been equally divided for each model, we are going to launch the training process with all models. In the first epoch, the parameter server generate the initial weights and broadcast them to each distributed client. As illustrated in Figure 2, the application part starts to train our model with corresponding data. Once the whole epoch has been completed, the trained weights will be sent to parameter server by sending part, and the next epoch can be started with new weights from receiving part.

This is the general implementation design, we are going to specify two aspects here, namely, data loading and parallelism.

As a critical component of our distributed learning model, how to assemble training data needs to be careful designed. Our purpose is to train our model on large datasets. Even the whole dataset can be equally divided for each machine in distributed system, the size of subset is still very large. We need to arrange that how this subset can be fed into each epoch. Processing data stream-likely is a great idea. We do not need to load all data at once, by stream-like dataflow we can release the memory pressure. In Pythorch these tools has been well realized, but in Golang these has to be implemented by ourselves.

The other important theme is parallelism, as in Figure 2 we can implement the three parts in distributed client parallel for training, or serialized. For this step we have to trade-off the benefit from parallelism and the consumption for mutex with shared variables. Thanks to the special design of Gorgonia for weights assignment, as the new weights for next epoch has been assigned to model as step *`weights immigration`* described, we can construct distributed learning as intuitive and effective.

The description in this subsection is still not implemented, the further implementation in this direction will be contributed to my master thesis.

5 Conclusion

This report is designed to explore the preparation of distributed learning with Golang. We introduced the background of this research topic in Chapter 1. With the increasing reachable data, distributed learning becomes a very fascinating playground for HPC. For simplicity we only talk about data parallelism. The goal of this report is to propose a architecture for distributed learning. In order to find an alternative for deep learning, we choose Golang as our program language, instead of Python.

For the first part in Chapter 2, we started with the introduction about Scalable AI. This research topic is stimulated by the growing of BD, many framework has been developed for handing the increasing data volume. In order to extract the insight of the data, many machine learning methods has been applied in this research area. Based on the development of BD, Scalable AI was implemented by BD frameworks at first, such as MapReduce tools. Later we saw that the research activities have been moved to cloud computing and distributed learning in HPC area, and we reviewed some research activities.

The next topic is about weights updating after gradient aggregation for neural network. In distributed system a parameter server is generally needed for gradient aggregation and broadcasting. Many architectures have been purposed for reducing the time between computation and communication, such as reduction tree. But as innovative design, such as Horovod, we don't need parameter server, but the drawbacks is also non-negligible. As the last part of this section we proposed a new real-time weights updating mechanism. It's totally asynchronous and failure tolerant in distributed system. Also this design is suitable for federated learning system cross many devices in different places.

In the Chapter 3 we claimed the advantages of Golang over Python for our report. From performance to dependencies, Golang illustrates us a new playground for this research area. No matter for MPI or CUDA libraries, CGO offers us the corresponding wrapper function for those APIs. Golang compilation provides us direct binary file for deployment, which is perfect for HPC tasks in cluster.

We had a deep discussion about the environment configuration for how to execute the binary file from Golang in cluster. The two directions, i.e., with and without singularity as in Figure 3 and Figure 4, both have their own appropriate use cases. In order to create a compatible environment for compilation and execution, we can compile our code depending on the requirements of the nodes in cluster. In our report, we create such a compilation environment with docker. After we implemented our model in Golang, we compile our code in this docker, so that the binary code can be directly executed in cluster like normally HPC task. On the other hand we can compile our mode as usual in development environment, and create such a isolated environment with singularity in cluster.

After we configured our environment, we come to the main part of our report in section 4. A short overview of ResNet model has been presented. Similarly to Pytorch and Tensorflow, we also explore the possibility that building our model based on *libtorch*. But the configuration of execution environment in cluster has been failed. So we decide to build our model from scratch with Gorgonia.

Next we presented the package dependencies for automatic differentiation, multidimensional tensor operations and CUDA compilation in Golang. The main component of the implementation for ResNet serial has been displayed in Listing 8 in Appendix D.

We only implemented the last 3 versions of ResNet serial, ResNet50, ResNet101 and ResNet151. For the simplicity, our results are only from ResNet50.

After the implementation of ResNet we presented the performances of our ResNet50 model. The time consumption for running our model in GPU and in CPU illustrate us a significantly and more and more bigger performance difference. With and without singularity, we see a slightly difference of GPU utilization, Without singularity our model utilized a little more GPU, while their time consumption are more than with singularity. Meanwhile a linearly increasing of GPU utilization as the batch-size growing is displayed for both cases. As the performance comparison with other famous framework, we illustrate the huge difference between ResNet in Golang and in Python.

As the last part, we discuss the design for distributed learning system with our model. Because of the *weights immigration* step in Gorgonia, we should be able to build our distributed learning system intuitive.

As summarizes for our report, we explored the possibility of building a distributed learning model with Golang. The corresponding model implementation and environment configuration have been done, and performances is also presented. For further development we need to immigrate MPI for weights update at *weights immigration*, add extra threads for data loading with Goroutine. After the distributed learning model with our ResNet model has been accomplished, we are excited to see the comparison of performance with other famous frameworks for distributed learning tasks.

References

[Aba+16] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning.* 2016. arXiv: 1605.08695 [cs.DC].

[Ash17] Noam Shazeer Ashish Vaswani etc. "Attention Is All You Need". In: *arXiv:1706.03752v5* (2017).

[Awa+17] Ammar Ahmad Awan et al. "S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters". In: vol. 52. Jan. 2017, pp. 193–205. DOI: 10.1145/3018743.3018769.

[Dea+12] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[GSJ16] Preeti Gupta, Arun Sharma, and Rajni Jindal. "Scalable machine-learning algorithms for big data analytics: a comprehensive review: Scalable machine-learning algorithms for big data analytics". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 6 (Sept. 2016). DOI: 10.1002/widm.1194.

[He+15] Kaiming He et al. *Deep Residual Learning for Image Recognition.* 2015. arXiv: 1512.03385 [cs.CV].

[Kol+20] Alexander Kolesnikov et al. *Big Transfer (BiT): General Visual Representation Learning.* 2020. arXiv: 1912.11370 [cs.CV].

[LXL17] Yang Liu, Lixiong Xu, and Maozhen Li. "The Parallelization of Back Propagation Neural Network in MapReduce and Spark". In: *International Journal of Parallel Programming* 45.4 (Aug. 2017), pp. 760–779. ISSN: 1573-7640. DOI: 10.1007/s10766-016-0401-1. URL: https://doi.org/10.1007/s10766-016-0401-1.

[Mam+18] Amith R Mamidala et al. *MXNET-MPI: Embedding MPI parallelism in Parameter Server Task Model for scaling Deep Learning.* 2018. arXiv: 1801.03855 [cs.DC].

[Mun23] Neelesh Mungoli. *Scalable, Distributed AI Frameworks: Leveraging Cloud Computing for Enhanced Deep Learning Performance and Efficiency.* 2023. arXiv: 2304.13738 [cs.LG].

[SB18] Alexander Sergeev and Mike Del Balso. "Horovod: Fast and Easy Distributed Deep Learning in Tensorflow". In: *CoRR* (2018). arXiv: 1802.05799 [cs.LG]. URL: http://arxiv.org/abs/1802.05799v3.

[Zho20] Schneider R. Zhou H. Gracia J. "MPI Collectives for Multi-core Clusters: Optimized Performance of the Hybrid MPI+MPI Parallel Codes." In: *arXiv* (2020).

[ZX16] Hai-jun Zhang and Nan-feng Xiao. “Parallel implementation of multilayered neural networks based on Map-Reduce on cloud computing clusters”. In: *Soft Computing* 20.4 (Apr. 2016), pp. 1471–1483. ISSN: 1433-7479. DOI: 10.1007/s00500-015-1599-3. URL: https://doi.org/10.1007/s00500-015-1599-3.

A CUDA support

A.1 CUDA Environment

For this report we use CUDA toolkit 11.8 as our development environment, and this is the last version of CUDA 11. For Gorgonia, CUDA toolkit 11.4 is the lowest compatible version. If the CUDA toolkit version is too old, by compiling we will receive the error that such function is not defined. But for different version of CUDA toolkit, we may have to face the problem that we have to switch the operating system. Such as for Gotorch which requires CUDA 10.1 or 10.2, We have to install Ubuntu18.

A.2 Compiling Gorgonia with CUDA

This appendix is about how Gorgonia can be compiled with CUDA libraries. The first step is to check if CUDA is available for Gorgonia with *cudatest* in *cu* package. The second step is to generate the wrapper function with *cudagen*. A go file *cudamodules.go* willl be automatically generated at project root directory.

A.3 CUDA Toolkit in cluster

We want to create a GPU compiler environment in cluster. No matter in Emmy or in SCC, we are facing the same problem. CUDA execution environment in cluster can be created by loading the corresponding CUDA module with spack. But CUDA development environment in cluster is impossible. We try to install it in login nodes. However such nodes do not have GPU support, the complain for *libcuda.so can not be found* remains always. Also *ld -lcuda -verbose* explicit show us that CUDA can't be found.

B Slurm script and Dockerfile

Listing 2: *Slurm bash file with singularity*

```
#!/bin/bash

#SBATCH -J silin_resNet_go
#SBATCH -N 2
#SBATCH -p grete
#SBATCH -G 2
#SBATCH --tasks-per-node 1
#SBATCH -t 2:00:00


module load singularity
module load openmpi
module load cuda/11.8
module load cudnn

export ASSUME_NO_MOVING_GC_UNSAFE_RISK_IT_WITH=go1.21

srun --mpi=pmix -n 2 singularity exec --nv
& --bind /home/nimichdu/RestNet/:/home/nimichdu/RestNet
& /home/nimichdu/RestNet/myProject/deepcam.sif
& /home/nimichdu/RestNet/myProject/restNet
```

Listing 3: *Slurm bash file without singularity*

```
#!/bin/bash

#SBATCH -J silin_resNet_go
#SBATCH -N 2
#SBATCH -p grete
#SBATCH -G 2
```

```
#SBATCH --tasks-per-node 1
#SBATCH -t 2:00:00


module load singularity
module load openmpi
module load cuda/11.8
module load cudnn

export ASSUME_NO_MOVING_GC_UNSAFE_RISK_IT_WITH=go1.21

srun --mpi=pmix -n 2 restNet
```

Listing 4: *Dockerfile of compilation environment in local hostmachine*

```
FROM nvidia/cuda:11.5.2-cudnn8-devel-centos7

RUN yum install -y wget
RUN wget https://go.dev/dl/go1.21.4.linux-amd64.tar.gz
RUN tar -C /usr/local -xzf go1.21.4.linux-amd64.tar.gz
ENV PATH=$PATH:/usr/local/go/bin
ENV GOPATH=/home/si/go
RUN rm go1.21.4.linux-amd64.tar.gz


RUN  echo "Installing Open MPI"
RUN  yum install -y perl
RUN  mkdir -p /opt/ompi
RUN  mkdir -p /tmp/ompi
RUN  wget https://download.open-mpi.org/release/open-mpi/v5.0/openmpi-5.0.0.tar.gz
RUN  tar -C /tmp/ompi -xzf openmpi-5.0.0.tar.gz
RUN  cd /tmp/ompi/openmpi-5.0.0
RUN  ./configure --prefix=/opt/ompi && make -j8 install
ENV  PATH=/opt/ompi/bin:$PATH
ENV  LD_LIBRARY_PATH=/opt/ompi/lib:$LD_LIBRARY_PATH
```

C Glibc Version

We list the Glibc version information for all possible frameworks, With command: *ldd -version*.

host	Operation system	Glibc version
localhost	*Ubuntu 22.04*	2.35
localhost	*Ubuntu 20.04*	2.31
localhost	*Ubuntu 18*	2.27
localhost	*Ubuntu 16*	2.17
localhost	*CentOS 7.9*	2.17
singularity deepcam	*Ubuntu 20.04*	2.31
nvidia/cuda:11.5.2-cudnn8-devel-centos7	*Centos 7.9*	2.17
Emmy in HLRN	*Centos 7.9*	2.17
SCC in GWDG	*Scientific Linux release 7.9*	2.17

Table 1: *Glibc version list of different operation systems*

Our singularity image is called *deepcam*, its link is from my supervisor, Sadegh Keshtkar. This singularity already immigrate MPI and CUDA dependencies. Because we only need to launch our model with singularity in cluster, a Golang environment is not needed.

Listing 5: *The source of singularity image*

```
singularity pull deepcam.sif docker://nvcr.io/nvidia/pytorch:22.08-py3
```

D Snippets of ResNet model Implementation with Golang

Listing 6: *The complete configuation file of ResNet model*

```
epoches=6
batchSize=4
usedistence=100
validateEvery=2
learningrate=5e-4
imageSizeLength=224
imageSizeWidth=224
resnetSerie=1  # 1 for ResNet50, 2 for ResNet101, 3 for ResNet152
dataPath=/home/si/data/ImageNet100/
```

Listing 7: *The convolutional part of ResNet model implementation*

```
result := Conv2d(m.model, Conv2dOpts{
  InputDimension:  3,
  OutputDimension: 64,
  KernelSize:      tensor.Shape{7, 7},
  Pad:             []int{3, 3},
  Stride:          []int{2, 2},
  WeightsName:     "/layer0/conv1/7x7",
}).Forward(x)

result = BatchNorm2d(m.model, BatchNormOpts{
  InputSize: result[0].Shape()[1],
  ScaleName: "/layer0/bn/gamma",
  BiasName:  "/layer0/bn/beta",
}).Forward(result[0])

x = gorgonia.Must(gorgonia.Rectify(result[0]))
x = gorgonia.Must(gorgonia.MaxPool2D(x, tensor.Shape{3, 3}, []int{1, 1}, []int{2, 2}))
```

Listing 8: *The main part of ResNet model implementation*

```
layers := []godl.Module{}
for layer, number := range opts.structure {
  stride := []int{1, 1}
  if layer != 0 {
    stride = []int{2, 2}
  }

  block_downsample := RestBlock(m, RestBlockOpts{
    InputDimension:  opts.inputchannel,
    OutputDimension: opts.intermediachannel[layer],
    Layer:           layer,
    Stride:          stride,
    Stage:           0,
  })
  layers = append(layers, block_downsample)

  opts.inputchannel = opts.intermediachannel[layer] * 4
  for stage := 0; stage < number-1; stage++ {
    block := Block(m, BlockOpts{
      InputDimension:  opts.inputchannel,
      OutputDimension: opts.intermediachannel[layer],
      Layer:           layer,
      Stage:           stage + 1,
    })
    layers = append(layers, block)
  }
}
```

Listing 9: *The fully connected layer of ResNet model implementation*

```
layers = append(layers,
Linear(m, LinearOpts{
  InputDimension:  2048,
  OutputDimension: 100,
  WithBias:        opts.WithBias,
  Activation:      gorgonia.Rectify,
  Dropout:         0.0,
  WeightsInit:     opts.WeightsInit,
  BiasInit:        opts.BiasInit,
  WeightsName:     "/fc1/fc1_W:0",
  BiasName:        "/fc1/fc1_b:0",
  FixedWeights:    fixedWeights,
}),
```

Listing 10: *The weights initination of ResNet model*

```
func Block(m *godl.Model, opts BlockOpts) *BlockModule {
```

```
    opts.setDefaults()
    lt := godl.AddLayer("RestNet.Block")

    ConvBlockName := fmt.Sprintf("Convulution:%d_%d", opts.Layer, opts.Stage)
    w1 := m.AddWeights(
            lt,
            tensor.Shape{opts.OutputDimension, opts.InputDimension, 1, 1},
            godl.NewWeightsOpts{
                InitFN:     opts.WeightsInit1x1,
                UniqueName: ConvBlockName + "_1",
                Fixed:      opts.FixedWeights,
                }
            )
    w2 := m.AddWeights(
            lt,
            tensor.Shape{opts.OutputDimension, opts.OutputDimension, 3, 3},
            godl.NewWeightsOpts{
                InitFN:     opts.WeightsInit3x3,
                UniqueName: ConvBlockName + "_2",
                Fixed:      opts.FixedWeights,
                }
            )
    w3 := m.AddWeights(
            lt,
            tensor.Shape{opts.OutputDimension * 4, opts.OutputDimension, 1, 1},
            godl.NewWeightsOpts{
                InitFN:     opts.WeightsInit1x1,
                UniqueName: ConvBlockName + "_3",
                Fixed:      opts.FixedWeights,
                }
            )

    BatchNormName := fmt.Sprintf("Normalize:%d_%d", opts.Layer, opts.Stage)
    bn1 := BatchNorm2d(m, BatchNormOpts{
      InputSize: opts.OutputDimension,
      ScaleName: BatchNormName + "_1/gamma",
      BiasName:  BatchNormName + "_1/beta",
    })
    bn2 := BatchNorm2d(m, BatchNormOpts{
      InputSize: opts.OutputDimension,
      ScaleName: BatchNormName + "_2/gamma",
      BiasName:  BatchNormName + "_2/beta",
    })
    bn3 := BatchNorm2d(m, BatchNormOpts{
      InputSize: opts.OutputDimension * 4,
      ScaleName: BatchNormName + "_3/gamma",
      BiasName:  BatchNormName + "_3/beta",
    })

    return &BlockModule{
      model:     m,
      layer:     lt,
      opts:      opts,
      weight:    []*godl.Node{w1, w2, w3},
      bns:       []*BatchNormModule{bn1, bn2, bn3},
      expansion: 4,
    }
}
```

E ResNet model with Python in cluster

As Pythorch has been very implemented for Deep Learning model in python ecosystem, we can easily test it in cluster for its performance.

1. Dataset: We take only the 1/40 examples from the training and validation sets from ImageNet100. DataLoader will take care of the data preparation.

2. Model implementation in python: for better comparison we do not use the ResNet model in libraries, but the implementation from scratch with Pytorch. Since there are many references for the implementation, we used the code snippets from internet[14].

3. Configuration extraction: we create a new virtual environment for minimal necessary python dependencies installation. And then we extract the dependencies to a *txt* file, and upload the file with python code into cluster.

[14] https://github.com/danielchyeh/ImageNet-100-Pytorch/blob/main/main_IN100.py

4. Environment Configuration: In cluster, at first we activate the *conda* module, and then we create a virtual environment and activate it. In the virtual environment we need to install the dependencies with the *txt* file.

5. Execution: after all dependencies has been installed, we can allocate a GPU compute node with *srun -p grete -pty -n 1 -c 64 -t 24:00:00 -G A100:2 bash*[15]. After the compute resource has been successfully allocated, we will be redirected to the interactive terminal environment of the compute node. In this terminal environment we are able to launch the training process.

[15]The description of this command please refer to `https://slurm.schedmd.com/srun.html`