Num Worker Tuner: An Automated Spawn Parameter Tuner for Multi-Processing DataLoaders

DoangJoo Synn*, JongKook Kim*[1]
*Dept. of Electrical Engineering, Korea University

Abstract

In training a deep learning model, it is crucial to tune various hyperparameters and gain speed and accuracy. While hyperparameters that mathematically induce convergence impact training speed, system parameters that affect host-to-device transfer are also crucial. Therefore, it is important to properly tune and select parameters that influence the data loader as a system parameter in overall time acceleration. We propose an automated framework called Num Worker Tuner (NWT) to address this problem. This method finds the appropriate number of multi-processing subprocesses through the search space and accelerates the learning through the number of subprocesses. Furthermore, this method allows memory efficiency and speed-up by tuning the system-dependent parameter, the number of multi-process spawns.

1. Introduction

Modern computing architectures use multi-cores and multi-GPUs, revealing the characteristics of a distributed system within one system. Due to this characteristic, various distributed system topologies are widely used to train deep learning models. For example, in multi-node learning, large models are trained using thousands of nodes. Thus, various technics such as learning large-sized batches and large-sized models were introduced [1].

In this process, modern deep learning frameworks actively utilize data loaders through multi-process or multi-thread spawn in serving data to devices. (e.g. PyTorch [2], Tensorflow [3]) A multi-stream data loader controls the number of data loader processes and serves data to multiple nodes through a dataloader node. As a default, the dataloader is implemented by loading and allocating data to the main process. However, in a distributed deep learning framework, performing speed optimization through an appropriate number of data loader processes is not only affected by the dataset but also the number of CPU cores and number of CPU cores as system-dependent parameters. In addition, it is affected by various factors such as clock speed, RAM size, GPU FLOPS, and GRAM size. Loading the data existing on the disk may significantly correlate on disk I/O depending on the type of data. Memory I/O affects the speed at which loaded data is put into memory. With a large number of multi-process spawns, each dataloader process will take up more than the proper amount of memory because it is holding the dataset. Therefore, memory efficiency is degraded.

This paper proposes the Num Worker Tuner (NWT). NWT provides a way to find an appropriate number of dataloaders through the search space and utilize them as machine-dependent parameters. Through this method, it is possible to automate and control the multi-process spawn of the data loader as a parameter, and it is possible to obtain memory efficiency and speed simultaneously.

2. Related Works

2.1 Multi-Processing in DataLoader

Due to the language design, multithreading in Python is extremely cumbersome and not recommended. The Global Interpreter Lock (GIL) prevents fully parallelizing Python code between threads within the Python process. [4] Multi-Process dataloader sets the num_workers argument to a positive integer and provides an easy switch to perform multi-process data loads to avoid blocking computation with data loads.

In general, the dataloader spawns a num_workers worker process whenever its iterator is created. Each worker receives a set of initialization arguments with a location for data fetching. This means that dataset access with disk I/O, transform (e.g., collate function) is executed in the worker process. Finally, the worker terminates when the iterator is garbage collected or the task is done.

2.2 Automatic Memory Pinning

Data transfer from CPU to GPU uses hardware-dependent APIs. Currently, hardware vendors provide various techniques to optimize data transfer to GPU. For example, one of the prominent vendors, NVIDIA, optimizes host-to-device data transfer with Pinned Memory. [5]

However, at this point, GPU does not have direct access to data in pageable host memory. Therefore, when the data transfer is executed from pageable host memory to GPU memory, the CUDA driver first allocates a temporary page lock or pinned host array and copies it. Hence, it allows data transfer from host to device memory as a fixed array.

[1] Corresponding author

Pinned memory is used as a staging area for transfers from the device to the host. Programmers can directly allocate host arrays in pinned memory to avoid transfer costs between pageable and pinned host arrays. Pinned Memory data transfer can be enabled through CUDA C/C++ or CUDA Python API, and various frameworks support this.

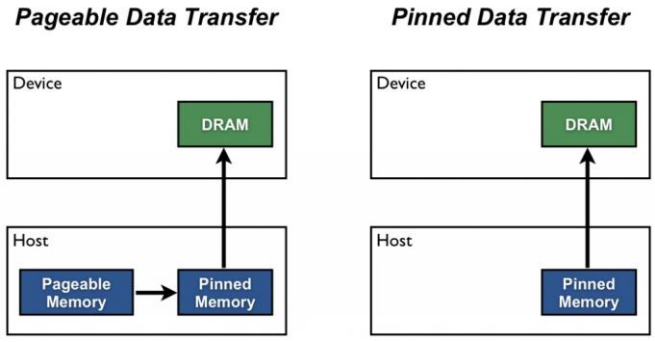

Fig 1. Pageable Data Transfer versus Pinned Data Transfer

3. Num Worker Tuner (NWT)

Num Worker Tuner (NWT) is a framework that finds the appropriate number of multiprocessing subprocesses for dataloaders through greedy search, considering the characteristics of the dataset and hardware dependencies. Thus, it holds the same designated result for a single dataset combination and hardware dependency. Furthermore, even if there are various datasets, the result can be re-used if the transmitted size is similar.

3.1 Parameters

NWT calls up to N-1 search spaces for subprocess in proportion to the number of CPUs (N). Thus, if N processes are allocated to CPU cores, context switching between the processes will degrade the overall performance. Also, NWT ignores differences lower than the alpha value in the minimum execution time based on performance. Thus, insignificant performance gains can be controlled through alpha and track the best memory optimization. Finally, the iteration parameter performs a data transfer loop based on pre-defined options. The default value for iteration is 100, which measures the transfer time and converts it to an average used to measure performance.

Parameter	Role	Default
alpha	Ignoring differences within a certain range (seconds)	0.5
N	Maximum value of search space	#cores
iteration	Number of loops to accumulate	100

Table 1. Parameters to Tune for NWT

3.2 NWT

NWT starts with the main process, spawns subprocesses up to N, and sequentially measures the data transfer cost. Then, it iterates by time spent, tracking best times. At this time, NWT tracks the memory usage of the entire system. Finally, based on the measurement results, the number of subprocesses of the dataloader with minimum overhead is determined and used as a system-wide parameter.

Algorithm 1 Num Worker Tuner (NWT)

```
# Parameters for NWT
N ←number of CPU cores
iter ← 100
alpha ← 0.5

# Search Best Time with corresponding N
i, j, result ← 0
best ← inf
while i ≠ N do
  cumm ← 0
  while j ≠ iter do
    cumm = cumm + MemCpyTime (host to device)
    j ← j + 1
  end while
  normalized ← cumm/iter
  if normalized < best − alpha then
    best ← normalized
    result ← i
  end if
  i ← i + 1
end while
```

Fig 2. Algorithm for NWT

4. Experiments

4.1 Experimental Setup

The experiment was performed on the Ubuntu 20.04 operating system with AMD Ryzen Threadripper 3970X 32-Core Processor for CPU, DDR5 128GB for RAM, and 4x Nvidia Geforce RTX A100. We used CUDA (Compute Unified Device Architecture) 11.1, a GPU parallel computing framework, and PyTorch 1.11.1 nightly. All the experiments were performed with the GPU Pinned Memory option turned on as a default.

We used CIFAR-10 as the dataset for the experiment. CIFAR-10 is an image classification dataset with dimensions of 60000x32x32 and was verified using the standard data batch size 128 recommended for MobileNetV2 and ResNet through CIFAR10 in PyTorch.

We used traced malloc to track memory usage. Traced malloc emits current and peak usage of the memory for specified bound of the code. Since the current usage from traced malloc does not reflect the memory usage removed by garbage collection, the memory usage was tracked through the peak memory usage.

4.2 Experimental Result

# of Subprocesses	Data Transfer Time (Normalized, second)	Memory Usage (MB)
0 (default)	14.25	2757.84
14(optimal)	1.49	2025.32
16(time best)	1.41	2055.37
32	1.55	2275.36
63	2.21	2773.60

Table 2. Experimental Result over number of processes (selected)

Table 2 shows the memory usage and data transfer time for the number of subprocesses in the CIFAR10 dataset. In

this case, using the normalization factor is for the calculation with accumulated value during 100 iterations. Thus, the normalization value is expressed as the time required per actual data transfer.

When data exists only in the main process, num of workers 0, data fetching is performed in the same process which the dataLoader is initialized. Therefore, computing may be blocked in data loading by Python's Global Interpreter Lock (GIL). However, this mode may be preferred if the resources used to share data between processes (e.g., shared memory, file descriptors) are limited or if the entire dataset is small and fully loaded into memory. Due to this property, the result shows that memory usage for num_worker 0 is more significant than that of the multi-processing dataloader.

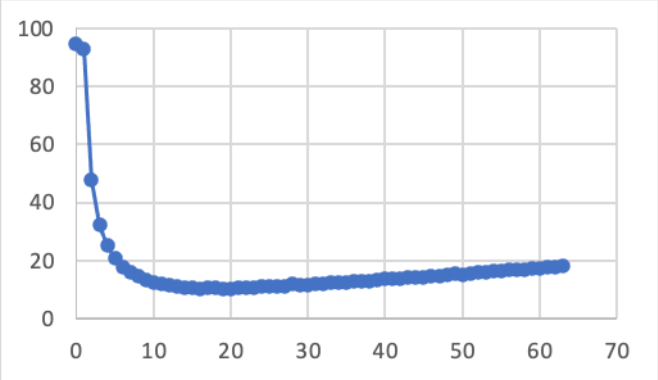

Fig 2. Data Transfer Time over
the number of Dataloader Subprocesses

As the number of processes increases, the result shows that it becomes slower when the number of workers increases than the time best of 16 through inter-process communication overhead, including overhead on the operating system. Accordingly, it is crucial to find out the specific number of processes. Also, based on the AMD Ryzen Threadripper 3970X 32-Core Processor in the experimental environment, simply allocating 16 processes means allocating 17 processes, including the main process. Therefore, the experimental results show that picking the appropriate number of processes based on the number of cores may not be suitable.

Fig 3. shows the result on memory usage. It can be derived from the normalization and alpha (default 0.5) that the difference between N 14 and 16 is less than 0.005 per epoch. Therefore, it is possible to get the optimal value that appropriately responds to the linearly increasing memory usage without significantly affecting the data loading speed. From the experimental results, it takes 1.49 seconds for 14 and 1.41 seconds for 16 subprocesses. From the given alpha, the difference between 14 and 16 subprocesses is insignificant at this time. Furthermore, memory usage is linearly increasing when a multiprocessing dataloader is a number greater than 2. Consequently, the NWT can derive 14 as the optimal value for subprocesses spawn and pass it on to the training algorithm.

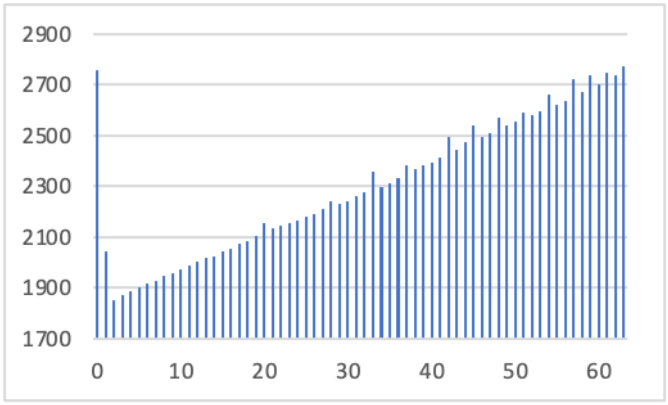

Fig 3. Memory Usage(MB) over Number of Dataloader Subprocesses

5. Conclusion

In this paper, we propose NWT(Num Worker Tuner), an automated framework to tune the number of subprocesses spawns for multi-processing dataloader.

NWT searches the optimal value for high-speed data transfer from CPU to GPU with corresponding data sizes and serves them as a system-wide parameter.

NWT benefits several situations, such as 1) training a large model on distributed frameworks, 2) training with a higher speed across the single machine with multi-GPUs with corresponding dataloader processes.

Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2016R1D1A1B04933156) This work was supported in part by the National Re- search Foundation of Korea (NRF) through the Basic Science Research Program funded by the Ministry of Education under Grant 2014R1A1A2059527, and in part by the Information Technology Research Center (ITRC), Ministry of Science and ICT (MSIT), South Korea, through a Support Program under Grant IITP-2020-2018-0-01433, supervised by the Institute for Infor- mation and Communications Technology Promotion (IITP)

Reference

[1] You, Y., Gitman, I., & Ginsburg, B. (2017). Large Batch Training of Convolutional Networks. arXiv: Computer Vision and Pattern Recognition.

[2] PyTorch “https://pytorch.org/docs/stable/data.html”

[3] Tensorflow “https://www.tensorflow.org/api_docs/python/tf/data”

[4] Python Global Interpreter Lock “https://docs.python.org/3/library/multiprocessing.html”

[5] NVIDIA, Optimize Data Transfers in CUDA C/C++ “https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpp/optimize-data-transfers/profile.cu”