Comparative Analysis of Runtime and Memory Optimization with ResNet152

Yoshimasa Iwano*
SFSU

ABSTRACT

This study explores the optimization of memory usage and runtime in training deep neural networks using PyTorch, with a focus on model parallelism and optimized data loading. Experiments were conducted using the CIFAR-100 dataset and the ResNet152 model, varying batch sizes and GPU counts to evaluate their impact on performance. The findings reveal significant improvements in runtime and memory efficiency with increased batch sizes and parallel processing, suggesting effective strategies for optimizing deep learning training in high-performance computing environments.

1 INTRODUCTION

Training large-scale deep neural networks is a computationally intensive task, often constrained by memory limitations and prolonged runtimes. This study addresses the challenge of optimizing memory usage and runtime in the training process, focusing on the PyTorch framework. The efficient training of neural networks is crucial for advancing research and applications in various domains, necessitating a comprehensive understanding of the factors influencing training performance.

To investigate this challenge, the study utilizes the CIFAR-100 dataset and the ResNet152 model, employing techniques like model parallelism and optimized data loading. By systematically altering batch sizes and GPU counts, the research aims to identify configurations that optimize memory usage and reduce training time. This approach provides a nuanced understanding of how different training parameters affect the performance of deep learning models.

The results of the study highlight that larger batch sizes and increased GPU counts lead to substantial improvements in runtime and memory efficiency. Specifically, the use of parallel processing techniques significantly reduces the number of kernel calls and optimizes GPU memory usage. These findings offer valuable insights for practitioners and researchers in the field, presenting practical strategies for enhancing the efficiency of neural network training in high-performance computing settings.

2 RELATED WORK

Recent research has focused on optimizing deep learning training in multi-GPU environments, addressing challenges such as scaling inefficiencies and memory utilization. Kourtis et al. [1] introduced Crossbow, a system enabling efficient training on multi-GPU servers with small batch sizes. Their approach, leveraging synchronous model averaging (SMA) and auto-tuning learners, demonstrated significant improvements in hardware efficiency and training time.

Complementing this, the work by Rajbhandari et al. [2] delved into optimizing multi-GPU parallelization strategies. They explored a hybrid of data and model parallelism to counteract the inefficiencies of large-scale data parallel training. Their analytical framework identifies the crossover point for effective parallelization, leading to notable speedups in training times for various deep learning models.

*email:yiwano@sfsu.edu

Both studies provide crucial insights into the efficient utilization of multi-GPU systems for deep learning training, aligning with our focus on optimizing memory usage and runtime in PyTorch-based training environments.

3 IMPLEMENTATION

3.1 Environment Setup

To enable parallel processing in PyTorch, necessary libraries such as `torch.distributed` and `torchvision` are imported. The environment variables `RANK` and `WORLD_SIZE` are set to define the rank of each process and the total number of processes in the distributed environment, respectively. The main script initializes the distributed process group and parses command-line arguments for configurations such as batch size and data size factor.

```
# main.py
import torch.distributed as dist
def main():
    # initialize the process group
    rank = int(os.environ['RANK'])
    world_size = int(os.environ['WORLD_SIZE'])
    dist.init_process_group("nccl", rank,
    world_size)
    // followed by data loading and training
```

3.2 Data Preparation and Distributed Sampling

The CIFAR100 dataset is used for training. To handle this dataset in a distributed manner, `DistributedSampler` is employed, which partitions the dataset among the different processes. This ensures that each GPU works on a unique subset of the data, reducing data redundancy and improving parallel processing efficiency.

```
# data_preparation.py
from torch.utils.data import DataLoader, Subset,
    DistributedSampler

def get_train_loader():
    // transform, load CIFAR-100 data, and create
    subsets (omitted)

    # Using DistributedSampler to partition the
    dataset for distributed training
    sampler = DistributedSampler(trainset,
    num_replicas, rank)
    trainloader = DataLoader(trainset, batch_size,
     sampler, pin_memory=True)

    return trainloader
```

3.3 Model Setup for Distributed Training

The model is initialized using the ResNet152 architecture. To adapt this model for distributed training, it is wrapped with `DistributedDataParallel`. This wrapper distributes the model across the available GPUs, allowing parallel computation of gradients and synchronized updates.

```
# model_setup.py
from torch.nn.parallel import
    DistributedDataParallel
import torch.distributed as dist

def setup_model():
    device = torch.device(f"cuda:{rank}")

    // initialize ResNet152 model and send it to
    GPU (omitted)

    # using DistributedDataParallel to distribute
    the model across GPUs
    model = DistributedDataParallel(model,
    device_ids=[rank])
    return model
```

3.4 Training Process

The training is conducted over multiple epochs. Each process independently loads a batch of data and performs forward and backward passes. The gradients are then synchronized across all GPUs to update the model weights. Profiling tools provided by PyTorch are utilized to monitor the performance and memory usage during training, enabling an understanding of the efficiency of the parallelization.

```
# train.py
import torch.optim as optim
from torch.profiler import profile,
    ProfilerActivity
def train_model():
    // Setup model, optimizer, and loss function (
    omitted)

    with profile(activities=[ProfilerActivity.CPU,
     ProfilerActivity.CUDA], profile_memory=True)
    as prof: // add this line to profile
        // training loop (omitted)
```

4 Results

4.1 Computational Platform and Software Environment

The experiments were conducted on the *Perlmutter* supercomputer located at NERSC. The focus of our study was the GPU capabilities of Perlmutter, which is equipped with four NVIDIA A100 (Ampere) GPUs. Each GPU is connected to the CPUs via PCIe 4.0, ensuring high-speed data transfer. The system also features a PCIe 4.0 NIC-CPU connection and 4 HPE Slingshot 11 NICs.

Each of the NVIDIA A100 GPUs has 40 GB of High Bandwidth Memory (HBM) with a memory bandwidth of 1555.2 GB/s. The CPU nodes, on the other hand, are equipped with a single AMD EPYC 7763 (Milan) CPU, featuring 64 cores per CPU and a memory bandwidth of 204.8 GB/s. Additionally, each GPU pair is interconnected with 12 third-generation NVLink links, with each link providing a bandwidth of 25 GB/s in each direction. The nodes are complemented with 256 GB of DDR4 DRAM, ensuring substantial memory capacity for compute-intensive tasks.

The software environment for our experiments included the gcc compiler version 11.2.0 20210728, provided by Cray Inc. This setup allowed for the optimized compilation of our codes, leveraging the advanced hardware capabilities of the Perlmutter system. Detailed information regarding the computational setup and software environment was sourced from the NERSC Perlmutter Architecture and Native compilers on NERSC systems documentation.

4.2 Methodology

4.2.1 Datasets, Model, and Optimizer

Our experiments utilized the CIFAR-100 dataset, a standard benchmark in machine learning for image classification tasks. The model of choice was ResNet152, a deep convolutional neural network known for its efficiency and performance in image recognition. For optimization, we employed Stochastic Gradient Descent (SGD) with a learning rate of 0.001 and a momentum of 0.9, defined in PyTorch as follows:

```
optimizer = optim.SGD(model.parameters(), lr
    =0.001, momentum=0.9)
```

4.2.2 Performance Test Parameters

To thoroughly evaluate the model's performance, we conducted tests across a range of configurations, varying in data size, batch size, and GPU count. This approach was designed to assess the impact of these parameters on the training efficiency and effectiveness of the model.

Data Size Variations: The experiments were conducted using three different data sizes to understand the scalability and memory efficiency of the model. The sizes tested were:

- The full CIFAR-100 dataset.
- Half of the CIFAR-100 dataset.
- One-tenth of the CIFAR-100 dataset.

Batch Size Variations: Batch size is a crucial factor that affects the speed and stability of the training process. We tested four different batch sizes to observe their impact on the model's performance:

- 128
- 512
- 1024
- 2048

GPU Utilization: The model was also tested across different GPU configurations to examine how parallel processing affects training time and efficiency. The configurations included:

- Single GPU (1 GPU)
- Dual GPUs (2 GPUs)
- Quadruple GPUs (4 GPUs)

Each combination of data size, batch size, and GPU count was systematically tested. For instance, the model was trained on the full dataset with a batch size of 128 using 1 GPU, then 2 GPUs, and finally 4 GPUs. This process was repeated for each batch size and data size combination, providing a comprehensive set of results to analyze the performance under various training conditions.

4.2.3 Performance Metrics

Three primary performance metrics were considered:

1. **Runtime**: The total time taken for the training process regarding to CPU and GPU.
2. **Number of Calls**: The count of kernel calls, such as for convolution operations.
3. **Memory Transfer**: The maximum CPU and GPU memory transfer during the training process.

4.2.4 Measuring Metrics

Metrics were collected using the PyTorch Profiler, which provides detailed insights into CPU and GPU time, the number of operations, and memory usage. The profiler was configured to monitor both CPU and GPU activities and track memory usage. The snippet below demonstrates the profiling process: p

```
with profile(activities=[ProfilerActivity.CPU,
            ProfilerActivity.CUDA],
            profile_memory=True) as prof:
    # Training loop
    # ...
```

The results from the profiler, including total operations, maximum CPU and GPU memory usage, and the number of calls for significant operations, were analyzed to understand the impact of varying batch sizes, data sizes, and GPU counts on the model's performance.

4.3 Runtime Performance

The runtime performance was analyzed based on both CPU and CUDA (GPU) times across different configurations of batch sizes and GPU counts, for three dataset sizes: full, half, and one-tenth of CIFAR-100.

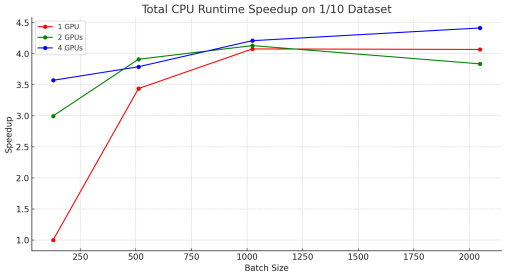

Figure 1: CPU Runtime for one-tenth of the CIFAR-100 dataset, showing the relationship between batch size, number of GPUs, and CPU execution time.

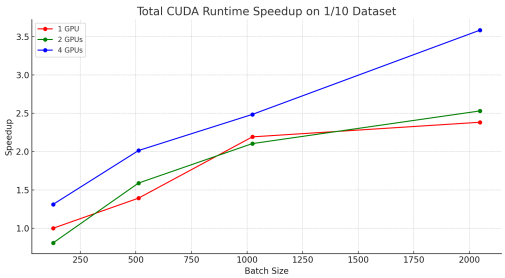

Figure 2: CUDA Runtime for one-tenth of the CIFAR-100 dataset, illustrating the impact of batch size and GPU count on GPU execution time.

The analysis of runtime performance across different dataset sizes reveals distinct trends in CPU and CUDA runtimes in response to changes in batch size and GPU count. While an overall reduction in runtime with increased batch size and GPU count was observed, the impact of these parameters varied depending on the size of the dataset.

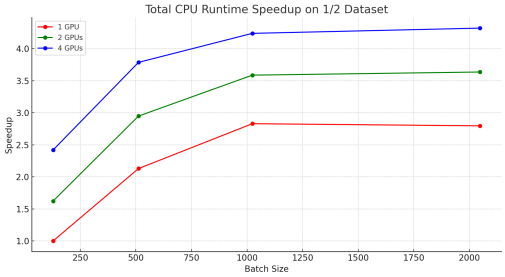

Figure 3: CPU Runtime for half of the CIFAR-100 dataset, indicating how the CPU time varies with different batch sizes and GPU configurations.

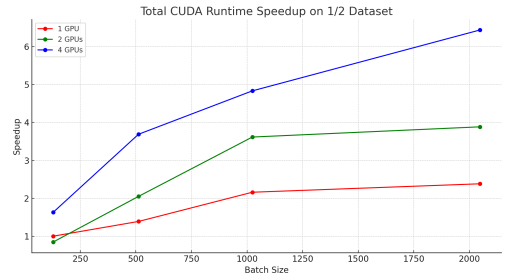

Figure 4: CUDA Runtime for half of the CIFAR-100 dataset, demonstrating the effect of batch size and number of GPUs on CUDA processing time.

4.3.1 One-Tenth of the Dataset

For the one-tenth dataset size, where each dataset contains approximately 6000 images, the reduction in runtime with increased batch sizes and GPU counts was less pronounced. This can be attributed to the smaller dataset size, which limits the efficiency gains from parallel processing. Particularly with a batch size of 2048, the interaction per epoch is reduced to around three times, diminishing the advantages of using multiple GPUs.

4.3.2 Half and Full Dataset

In contrast, for the half and full datasets, a more substantial decrease in runtime was observed with increasing batch sizes and GPU counts. Notably, in smaller batch sizes, an increase in batch size had a more significant impact on reducing runtime. However, in larger batch sizes, the increase in GPU count became more influential in decreasing runtime. This trend suggests that while larger batch sizes effectively reduce the number of iterations per epoch, further runtime reductions in these scenarios are achieved more efficiently through increased parallelism provided by additional GPUs.

The study underscores the importance of balancing batch size and GPU count to optimize runtime performance in deep learning model training. For smaller datasets, like one-tenth of CIFAR-100, the advantage of multiple GPUs is limited due to reduced interaction per epoch. In contrast, for larger datasets, leveraging both larger batch sizes and more GPUs is crucial for achieving significant reductions in both CPU and CUDA runtimes.

4.4 Number of Calls Performance

The number of kernel calls, such as those for convolution operations in the neural network, is a crucial metric in understanding the computational complexity and efficiency of the model. This subsection presents the analysis of the total number of calls for each

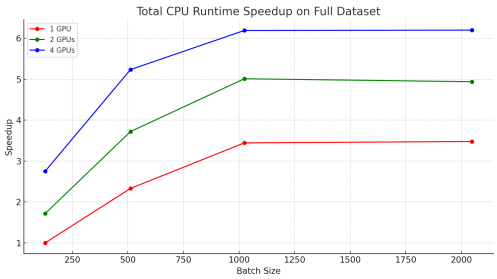

Figure 5: CPU Runtime for the full CIFAR-100 dataset, showing how CPU time changes with varying batch sizes and GPU counts.

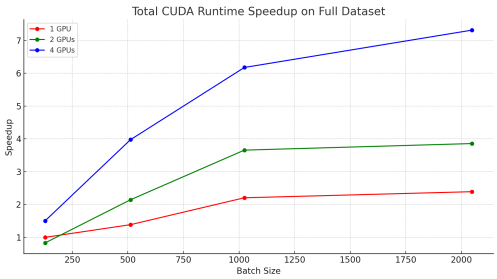

Figure 6: CUDA Runtime for the full CIFAR-100 dataset, highlighting the relationship between batch size, GPU count, and CUDA execution time.

configuration, across three dataset sizes: full, half, and one-tenth of CIFAR-100.

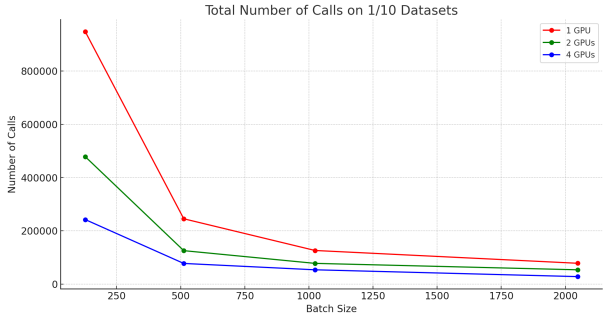

Figure 7: Total number of calls for one-tenth of the CIFAR-100 dataset, illustrating the impact of batch size and GPU count on the frequency of kernel calls.

Across all dataset sizes, an increase in batch size and the number of GPUs used generally led to a decrease in the total number of kernel calls. This pattern highlights the importance of optimizing these parameters to reduce computational overhead and improve the efficiency of training deep learning models.

4.5 Memory Performance

The memory performance was evaluated by analyzing the maximum memory transfers for both CPU and GPU. This analysis provides insights into the memory efficiency of the model under different configurations of batch sizes and GPU counts, for three dataset sizes: full, half, and one-tenth of CIFAR-100.

The memory performance analysis across all dataset sizes indicates that while the CPU memory transfer remains relatively unaffected by batch size and GPU count, the GPU memory transfer shows a substantial decrease with increased parallelism and larger batch sizes. This pattern suggests effective utilization of GPU memory in larger and more complex training configurations.

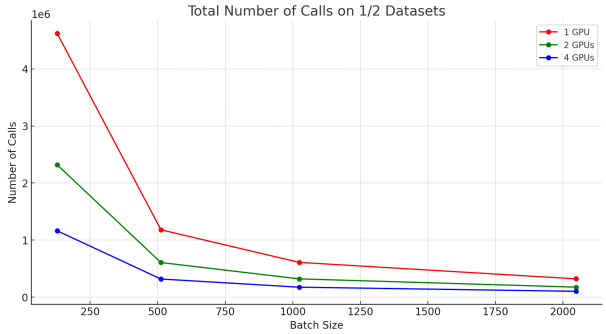

Figure 8: Total number of calls for half of the CIFAR-100 dataset, demonstrating how batch size and number of GPUs affect the model's computational demands.

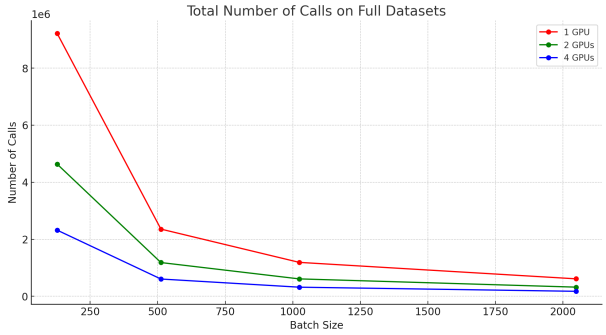

Figure 9: Total number of calls for the full CIFAR-100 dataset, showing the relationship between computational complexity and varying batch sizes and GPU counts.

References

[1] e. a. Kourtis. Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers. *arXiv preprint arXiv:1901.02244*, 2019.

[2] e. a. Rajbhandari. Optimizing multi-gpu parallelization strategies for deep learning training. *arXiv preprint arXiv:1907.13257*, 2019.

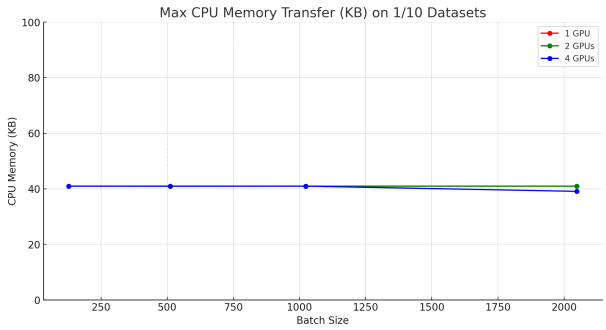

Figure 10: Maximum CPU memory transfer for one-tenth of the CIFAR-100 dataset, showing the impact of batch size and GPU count on CPU memory usage.

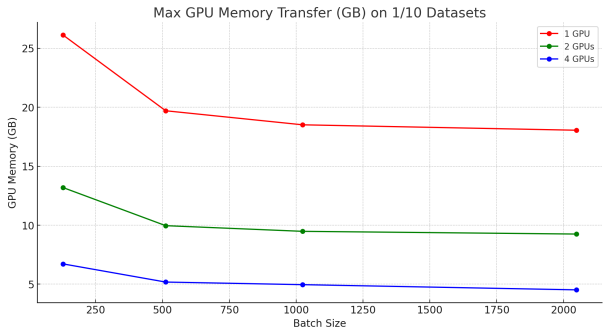

Figure 11: Maximum GPU memory transfer for one-tenth of the CIFAR-100 dataset, illustrating how memory usage varies with different batch sizes and GPU configurations.

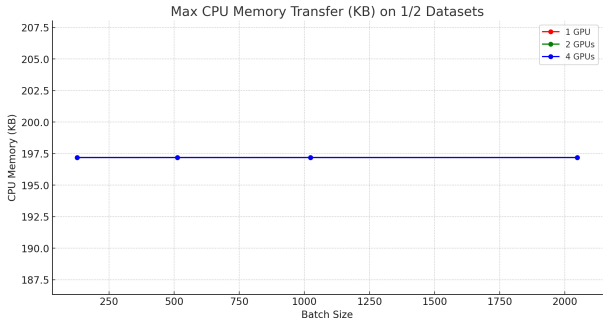

Figure 12: Maximum CPU memory transfer for half of the CIFAR-100 dataset, demonstrating the CPU memory requirements under various configurations.

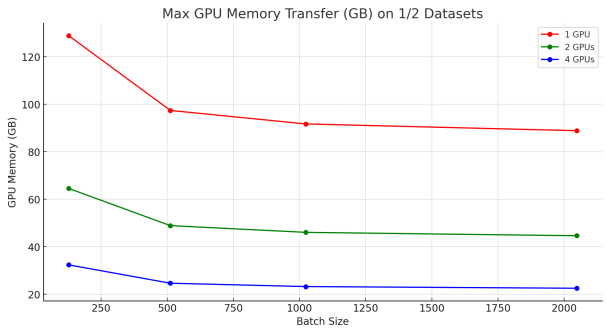

Figure 13: Maximum GPU memory transfer for half of the CIFAR-100 dataset, showing the relationship between GPU memory usage and batch size/GPU count.

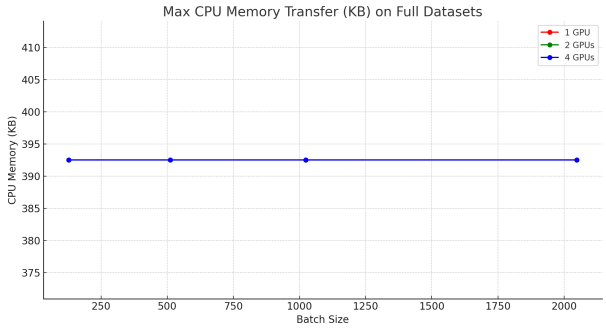

Figure 14: Maximum CPU memory transfer for the full CIFAR-100 dataset, indicating how CPU memory transfer varies with batch sizes and GPU counts.

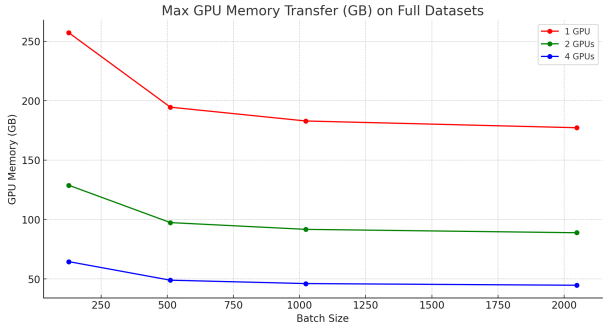

Figure 15: Maximum GPU memory transfer for the full CIFAR-100 dataset, highlighting the effect of batch size and GPU count on GPU memory utilization.