Harjot Mangat

EECS 245 – Parallel Computing

Final Project Report

1. INTRODUCTION

Pytorch is an open-source deep learning framework. Pytorch allows tasks to be run on either the CPU or GPU. The ability to transfer data onto the GPU can allow for extreme parallelism. Pytorch implements this ability using CUDA, Nvidia's parallel computing platform that allows processing to be run on the GPU. This accelerates many deep learning tasks that would normally take a very long time to run on the CPU.

The function to transfer data to the device is the *.to(device)* function in Pytorch. This function works with the tensor or with entire models. Pytorch's core is in C++, so when this *.to(device)* function is called at the python level, there is a function in the C level *to_impl()* that handles the arguments from *.to(device)*. Once the device argument is determined to be 'cuda', it is established that the tensor operation or model should be moved on the GPU. All the data from the *.to(device)* function is handled through *CUDACachingAllocator()* for allocating in device memory for operation[1].

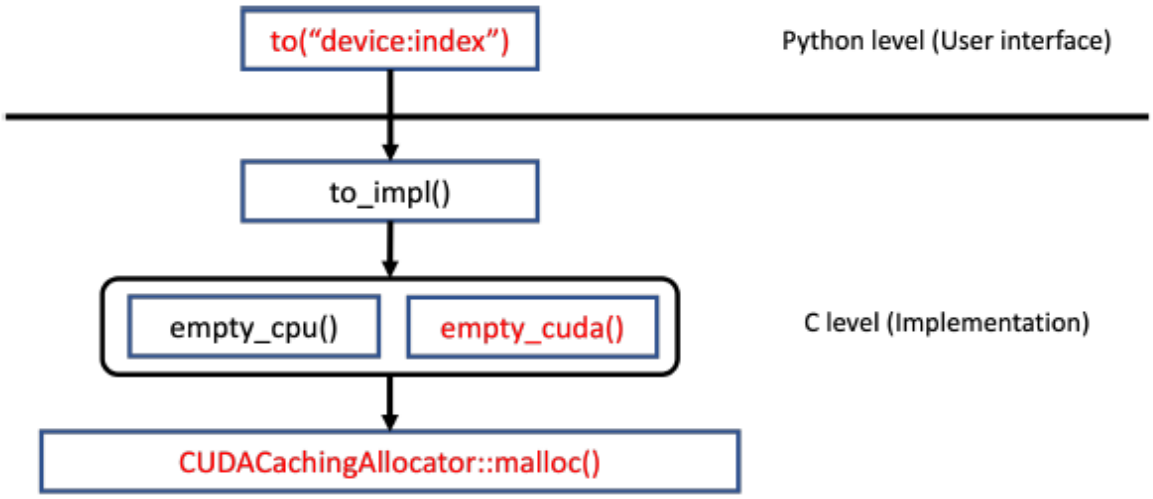


Figure 1 Memory Allocation from .to(device) function [1]

2. PYTORCH & CUDA

Initially, I assumed that Pytorch used CUDA python or some similar implementation to allow for the use of GPU acceleration. After doing more research, I learned that much of Pytorch is written in C++ for high performance [1]. The core library is *libtorch,* written in native C++, and it has the data structures for tensors, GPU and CPU operators, and gradient formulas for built-in functions. The core library allows Pytorch operators to be executed entirely in a multithreaded evaluator without having to hold the python global interpreter lock [2]. This allows the user to have the simplicity of writing in Python while the program can still leverage the high performance of C.

The use of C++ allows the integration of CUDA to be very simple. Previously, I discussed the *.to(device)* function that Pytorch has for sending tensor operations and models to the GPU and allocating memory on the GPU for these operations. The CUDA stream mechanism allows Pytorch to queue kernel invocations on the GPU. With the use of both CPU and GPU, Pytorch can allow the execution of Python code on the CPU and tensor operators on the GPU at the same time. This mechanism is nearly invisible to the user while still allowing for high performance even with the high overhead of Python [2].

3. PYTORCH PERFORMANCE

Next, I tried to look for any papers that attempted to measure the performance of Pytorch while using the CUDA features. The first paper was, “PyTorch: An Imperative Style, High-Performance Deep Learning Library” by Paszke et al. [2]. This paper contains a benchmark comparing Pytorch’s performance to other deep learning frameworks, such as CNTK, MXNet, TensorFlow, Chainer, and PaddlePaddle. Table 1 shows how the performance of Pytorch is regularly high compared to the other frameworks.

Framework	*Throughput (higher is better)*					
	AlexNet	VGG-19	ResNet-50	MobileNet	GNMTv2	NCF
Chainer	778 ± 15	N/A	**219** ± 1	N/A	N/A	N/A
CNTK	845 ± 8	84 ± 3	210 ± 1	N/A	N/A	N/A
MXNet	**1554** ± 22	113 ± 1	218 ± 2	444 ± 2	N/A	N/A
PaddlePaddle	933 ± 123	112 ± 2	192 ± 4	**557** ± 24	N/A	N/A
TensorFlow	1422 ± 27	66 ± 2	200 ± 1	216 ± 15	$9631 \pm 1.3\%$	$4.8e6 \pm 2.9\%$
PyTorch	1547 ± 316	**119** ± 1	212 ± 2	463 ± 17	**15512** $\pm 4.8\%$	**5.4e6** $\pm 3.4\%$

Table 1: Training speed for 6 models using 32bit floats. Throughput is measured in images per second for the AlexNet, VGG-19, ResNet-50, and MobileNet models, in tokens per second for the GNMTv2 model, and in samples per second for the NCF model. The fastest speed for each model is shown in bold.

We can see how the throughput on VGG-19, GNMTv2, and NCF models is highest in PyTorch. Also, in the models where the Pytorch framework is not the highest, it is still close to the performance of the top performing framework.

The next paper, “Efficient Use of GPU Memory for Large-Scale Deep Learning Model Training” by Choi, H and Lee, J mainly focuses on an implementation of CUDA unified memory into Pytorch. The focus of the paper is different from what I was looking for, so I could only find one performance analysis that included the default CUDA implementation for Pytorch. The performance analysis consisted of the original CUDA implementation, and four of their implementations for unified memory with different advice (PL – Preferred Location, RM – Read mostly, NA – no memory advice) [1]. They measured the overhead of each unified memory approach compared to the original CUDA implementation with no unified memory. The results can be seen in figure 2, although this cannot tell us much about the CUDA implementation’s performance versus no use of CUDA.

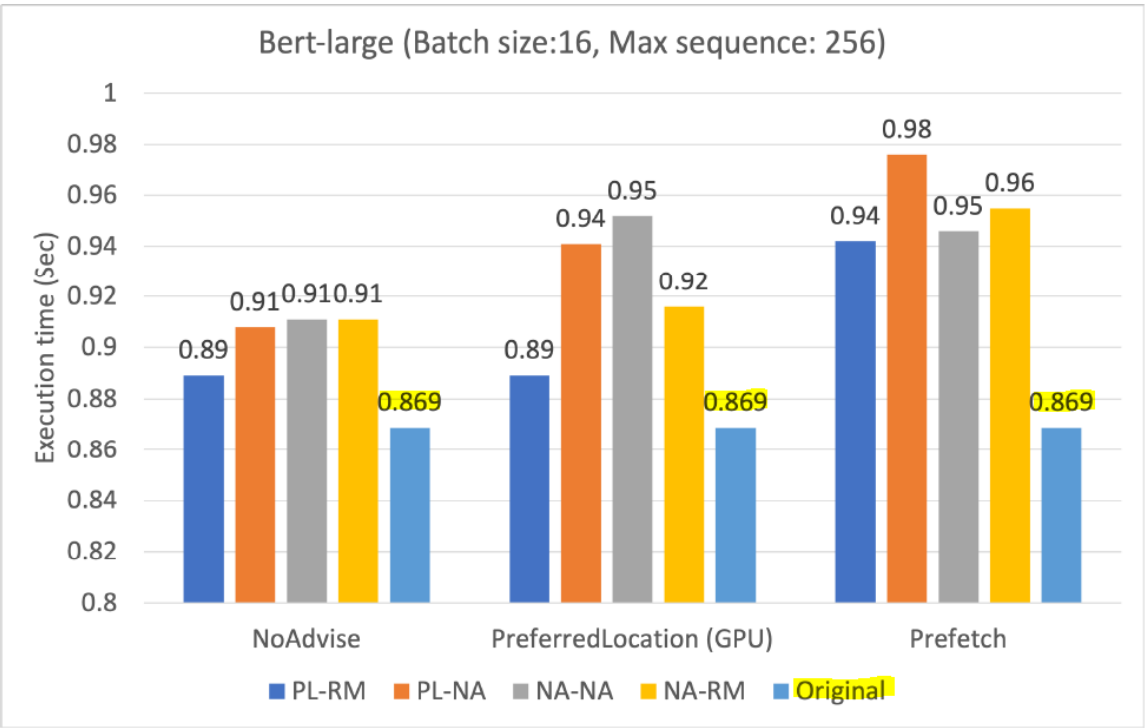


Figure 2. Execution time when the data size is smaller than GPU memory size. (Note that the y-axis of this graph starts with 0.8.)

Since I was not able to find many papers that really dive into benchmarking Pytorch while running with CUDA versus without CUDA, I will try to do some benchmarks myself in the next section.

4. BENCHMARK

For this benchmark, I used a very simple neural network model trained on a very common dataset known as MNIST. This MNIST dataset contains 70,000 28x28 grayscale images of handwritten numbers. The model itself consists of two convolutional layers and two linear layers, with ReLU activation function, SGD optimizer function, and negative log likelihood loss (nll_loss) as the loss function [3].

During benchmarking, I used Pytorch's included benchmarking utility, torch.utils.benchmark with its Timer function. On the first benchmark, I trained the model for 1 epoch on CPU. This was run twice; the first run used all threads on the CPU and the second run used one thread on the CPU. The next benchmark used pytorch's .to(device) function to allocate the model and the

dataset to the GPU for computation. This GPU benchmark was also run two times, the first time with all CPU threads and the second time with 1 CPU thread.

Device	Time(sec)
CPU(all threads)	14.78
CPU(single thread)	24.08
GPU(all CPU threads)	8.78
GPU(single CPU thread)	9.53

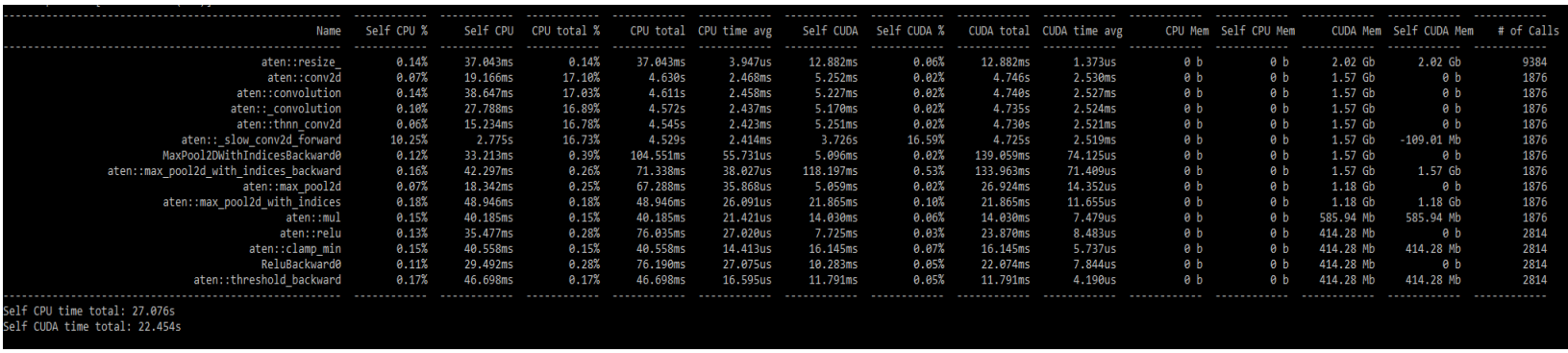
Table 2 Network training time for 1 epoch

By looking at table 2, we can gain several insights. When looking at the two CPU model trainings, we can see that the performance scales with the increase in CPU threads, the model trained faster with all threads than it did with a single thread. For the model on the GPU, increasing the number of CPU threads did very little for the model, since most of the computation has been moved to the GPU. We can also see the performance increase from using the GPU with pytorch's CUDA implementation versus running the model on a CPU. This difference in performance comes from the increased parallelism offered by GPUs compared to CPUs. The improvement in performance difference should continue this trend as the size of the model increases.

5. PROFILING

I have looked into profiling the performance of the Pytorch model that I had made and benchmarked in the previous section. I saw that Pytorch offers profiling abilities in its profiler library. This library allows profiling on the CPU, GPU, Memory, and other items called 'activities'. I used this profiler to run a profile on one training iteration of my program. I profiled the CPU, GPU, and memory in this instance. The figure is sorted by CUDA memory usage, but

it could be sorted by any of the identifiers in the figure. I notice that the profiler does add a lot of overhead to the overall time of the execution.



Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	CPU Mem	Self CPU Mem	CUDA Mem	Self CUDA Mem	# of Calls
aten::resize_	0.14%	37.043ms	0.14%	37.043ms	3.947us	12.882ms	0.06%	12.882ms	1.373us	0 b	0 b	2.02 Gb	2.02 Gb	9384
aten::conv2d	0.07%	19.166ms	17.10%	4.630s	2.468ms	5.252ms	0.02%	4.746s	2.530ms	0 b	0 b	1.57 Gb	0 b	1876
aten::convolution	0.14%	38.647ms	17.03%	4.611s	2.458ms	5.227ms	0.02%	4.740s	2.527ms	0 b	0 b	1.57 Gb	0 b	1876
aten::_convolution	0.10%	27.788ms	16.89%	4.572s	2.437ms	5.170ms	0.02%	4.735s	2.524ms	0 b	0 b	1.57 Gb	0 b	1876
aten::thnn_conv2d	0.06%	15.234ms	16.78%	4.545s	2.423ms	5.251ms	0.02%	4.730s	2.521ms	0 b	0 b	1.57 Gb	0 b	1876
aten::_slow_conv2d_forward	10.25%	2.775s	16.73%	4.529s	2.414ms	3.726s	16.59%	4.725s	2.519ms	0 b	0 b	1.57 Gb	-109.01 Mb	1876
MaxPool2DWithIndicesBackward0	0.12%	33.213ms	0.39%	104.551ms	55.731us	5.096ms	0.02%	139.059ms	74.125us	0 b	0 b	1.57 Gb	0 b	1876
aten::max_pool2d_with_indices_backward	0.16%	42.297ms	0.26%	71.338ms	38.027us	118.197ms	0.53%	133.963ms	71.409us	0 b	0 b	1.57 Gb	1.57 Gb	1876
aten::max_pool2d	0.07%	18.342ms	0.25%	67.288ms	35.868us	5.059ms	0.02%	26.924ms	14.352us	0 b	0 b	1.18 Gb	0 b	1876
aten::max_pool2d_with_indices	0.18%	48.946ms	0.18%	48.946ms	26.091us	21.865ms	0.10%	21.865ms	11.655us	0 b	0 b	1.18 Gb	1.18 Gb	1876
aten::mul	0.15%	40.185ms	0.15%	40.185ms	21.421us	14.030ms	0.06%	14.030ms	7.479us	0 b	0 b	585.94 Mb	585.94 Mb	1876
aten::relu	0.13%	35.477ms	0.28%	76.035ms	27.020us	7.725ms	0.03%	23.870ms	8.483us	0 b	0 b	414.28 Mb	0 b	2814
aten::clamp_min	0.15%	40.558ms	0.15%	40.558ms	14.413us	16.145ms	0.07%	16.145ms	5.737us	0 b	0 b	414.28 Mb	414.28 Mb	2814
ReluBackward0	0.11%	29.492ms	0.28%	76.190ms	27.075us	10.283ms	0.05%	22.074ms	7.844us	0 b	0 b	414.28 Mb	0 b	2814
aten::threshold_backward	0.17%	46.698ms	0.17%	46.698ms	16.595us	11.791ms	0.05%	11.791ms	4.190us	0 b	0 b	414.28 Mb	414.28 Mb	2814

Self CPU time total: 27.076s
Self CUDA time total: 22.454s

Figure 3 Profiling results ranked by CUDA memory usage

The results from figure 3 show that the highest CUDA memory usage is by the ‘resize_’ process, followed by a few convolution computations. This figure is filtered by the top 15 most CUDA memory intensive processes. The total time for both CPU and CUDA is a lot longer than it would take without the profiling, but this profiling is insightful.

The ability to profile the model can allow for insights into where bottlenecks could be in the system and potential areas for optimization. In the case of my small example model that was profiled, we could look at optimizing the ‘resize_’ and other high CUDA memory usage operations to help with model performance. In addition, we could reprofile the model and sort by the operations that use the most total CUDA time, most CPU time, or any other issue that we would be interested in.

6. CONCLUSION

In conclusion, Pytorch has a good implementation of CUDA where it leverages the performance of C++ for executing model and tensor operations, while still keeping the simplicity of python for the user. Performance benchmarks show that Pytorch has a high throughput, and my own benchmarks show that CUDA is working correctly at accelerating training time versus using CPUs. The profiling utility is very helpful at showing several different aspects of system usage to help with optimization and detecting bottlenecks.

Works Cited

[1] Choi, H., & Lee, J. (2021). Efficient Use of GPU Memory for Large-Scale Deep Learning Model Training. *Applied Sciences*. Retrieved from https://doi.org/10.3390/app112110377

[2] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., . . . Chintala, S. (2019). Pytorch: An Imperative Style, High-Performance Deep Learning Library. *NIPS'19: Proceedings of the 33rd Internation Conference on Neural Information Processing Systems*, (pp. 8026-8037). Vancouver. Retrieved from https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[3] Koehler, G. (2020, February 17). *MNIST Handwriten Digit Recognition in PyTorch*. Retrieved from nextjournal.com: https://nextjournal.com/gkoehler/pytorch-mnist

[4] Pytorch Benchmark. (n.d.). Retrieved from pytorch.org: https://pytorch.org/tutorials/recipes/recipes/benchmark.html