Sparse Parallel PyTorch on Multicore

Alexander Miller[1], Ari Khaytser[1], Jiajing Chen[1], and Jinhao Pang[1] *

Computer Science Department, New York University, New York NY 10012, USA
{ahm9968,ajk745,jc12020,jp6862}@nyu.edu

Abstract. PyTorch[1] is a scientific computing package extensively used for deep learning research and applications. Nevertheless, its focus on GPUs rather than CPUs can result in challenges deploying to lower-end devices and performance discrepancies between the two platforms. To address this issue, we propose enhancing the CPU parts of a PyTorch pipeline for improved CPU performance on CPU-only platforms like embedded systems, CPU-only cloud platforms, or personal laptops. Our approach combines state-of-the-art model pruning which retain most of a model's accuracy with custom C++ code for accelerating the pruned model performance using OpenMP parallelization. These models still run slower than the original dense computations which rely on highly-optimized matrix multiplication libraries and vector instructions on underlying hardware. However, this tradeoff comes with the benefit of reducing memory usage by nearly the full pruning rate; for example, with a pruning rate of 99.5%, these models reduce the memory required to run the model by 98% or more.

1 Introduction

Model developers typically make use of specialized hardware to train neural networks, such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs). However, after training these networks, a model developer may want to deploy a trained model on low-resource, power-constrained, accelerator-free platforms to perform inference tasks. However, PyTorch does not offer substantial CPU-targeted solutions for accelerating these process. We focus on a simple type of network which forms the backbone of many more complex ones: feedforward neural networks, sometimes known as a multilayer perceptron. These kinds of networks are present in both transformer networks (after multi-head attention) as well as in some kinds of residual convolutional networks (i.e. where 1x1 convolutions are present).

We achieve the following contributions in this paper:

1. We apply a pruning technique following [2], who show that neural networks tend to be overparameterized and that most of the performance of the model can be extracted from a small number of weights, e.g. 90% or higher accuracy with only 2% of the model weights. We use code from [3] to implement the pruning algorithm, keeping only the highest few percentage of weights with the largest magnitude.

2. The pruning techniques above only zero-out weights: they do not result in runtime or memory improvements for the end user. So, we implement several sparse matrix-vector algorithms to compare their performance. By default, these algorithms use dramatically less memory but by default are not competitive with highly optimized dense matrix computations, even with high pruning rates.

3. We use OpenMP to accelerate the sparse computations in order to get back closer to dense performance (again while using MUCH less memory). We explore the differences in these techniques and the resulting performance.

PyTorch provides a developer-friendly C++ extension framework[4] that allows for customized functions written in C++ to be integrated back into the PyTorch framework, which we use to optimize performance.

The remainder of this paper is organized as follows: Section II introduces related works about PyTorch Parallelization. Section III describes our general idea and model in detail. Section IV shows our details on CIMS deployment and experimental settings. Section V describes extensive experiments to validate the effectiveness of our approach. Discussion for our conclusion is given in section VI.

* Alphabetical by first name.

2 Literature Survey

Parallelism has been widely used in both academia and industry to improve the performance of PyTorch implementation. However, these approaches often focus on optimizing specific algorithms for particular application scenarios, which may not be suitable for other use cases. For instance, Lienen et al. proposed an Ordinary Differential Equations(ODE) solver that can solve ODEs in parallel independently[5]. Laporte et al. introduced a highly parallelizable application of reimagining photonic circuits on PyTorch[6]. Shao et al. presented PyChain to build a Kaldi automatic speech recognition toolkit[7]. Although these approaches have shown promising results in their respective domains, they may not be scalable or adaptable to different hardware platforms, which can lead to expensive application re-engineering, porting, and hardware platform switching[8].

To address this issue, we propose a novel research perspective that focuses on improving it at the kernel level rather than a user-level acceleration. To maximize the performance of PyTorch-based models generally, we start from the most common computational unit, matrix-vector multiplication due to the following two reasons: 1) matrix multiplication is inherently parallelizable and there is much research about how to optimize the matrix multiplication algorithms. For instance, Strassen's Algorithm[9] and Cannon's Algorithm[10] apply a divide-and-conquer idea in matrix multiplication to reduce arithmetic operations; Blocked Matrix Multiplication[11] divides matrices into smaller blocks to increase data locality and reduce cache misses; Winograd's Algorithm[12] speeds up convolutional neural networks by transforming the convolution into a set of matrix multiplications; Fast Fourier Transform (FFT)[13] reduced the repetitive computation by exploiting the symmetry and periodicity of the input arguments. These algorithms have different strengths and weaknesses and are often used in combination with each other to achieve the best performance for a given application.

While we wouldn't expect to be able to outperform these models on their own, recent works have shown that intelligently pruning a neural network can often maintain most of its performance. However, these techniques don't come with changes to the underlying computation algorithm that can actually take advantage of the smaller network. [2] in particular noticed that by removing the lowest magnitude weights from a network, more than 99.5% of the weights of some networks could be removed and still get accuracy above 90% on simple benchmarks.

3 Proposed Idea

Targeting a more general and scalable solution to improve the performance of PyTorch-based models compared to the existing approaches, we proposed a project that aims to optimize the matrix-vector multiplication algorithms for PyTorch-based models, which can potentially improve the performance of various machine learning applications. Inspired by the idea that matrix multiplication can be accelerated on the input matrices' characteristics, we propose using sparse matrix algorithms, which are specialized in leveraging the sparsity to accelerate the multiplication process in networks with many zero weights after the pruning algorithm has zeroed most of them out.

We choose to implement both CSR and COO formats in order to compare their performance.

To be more specific, the pseudo code for our CSR algorithm show as follow:

Algorithm 1 Pseudo code for CSR Conversion Algorithm

Require: m, n, $Tensor$
Ensure: ROW_INDEX, COL_INDEX, V
$num\_of\_value \leftarrow 0$
for $i \leftarrow 1$ to m **do**
  $ROW\_INDEX[i] \leftarrow num\_of\_value$
  for $j \leftarrow 1$ to n **do**
    $value \leftarrow$ MATRIX(i, j)
    if $value \neq 0.0$ **then**
      $V[num\_of\_value] \leftarrow value$
      $COL\_INDEX[num\_of\_value] \leftarrow j$
      $num\_of\_value \leftarrow num\_of\_value + 1$
    end if
  end for
end for
$ROW\_INDEX[m+1] \leftarrow num\_of\_value$

Algorithm 2 Pseudo code for CSR multiplication Algorithm

Require: m, ROW_INDEX, COL_INDEX, V, x
Ensure: $result$
for $i \leftarrow 0$ **to** $m-1$ **in parallel do**
  $sum \leftarrow 0$
  for $j \leftarrow$ ROW_INDEX[i] **to** ROW_INDEX[$i+1$]-1 **do**
    $col \leftarrow$ COL_INDEX[j]
    $sum \leftarrow sum + V[j] \times x[col]$
  end for
  $result[i] \leftarrow sum$
end for

The purpose of this function is to optimize the multiplication of a dense matrix with a tensor (vector) by combining two previously implemented functions into one. The original functions included a conversion of a tensor to Compressed Sparse Row (CSR) format, which returned a triple consisting of three arrays, and a multiplication of a matrix in CSR format with a tensor, which returned a tensor.

The algorithm for the optimized function includes two main steps. The first step involves converting the dense matrix to CSR format, which is accomplished by creating three arrays: V, which stores the non-zero values in the matrix; CO_INDEX, which stores the column indices of the non-zero values; and ROW_INDEX, which stores the number of non-zero values between row 0 and row i-1.

The algorithm's second step is multiplying the matrix with a vector. For example, given a matrix of the form $\begin{bmatrix}0 & 1 & 3 & 0\end{bmatrix}$ and a vector of the form $\begin{bmatrix}1 & 1\end{bmatrix}$, the CSR format would have "V" equal to $[1, 3]$, COL_INDEX equal to $[1, 0]$, and ROW_INDEX equal to $[0, 1, 2]$. These values indicate that row -1 has no items, row 0 has 1 item, and rows 0 to 1 have 2 items in total.

To perform the multiplication, the algorithm loops through the matrix rows, retrieves the value from V, and uses its corresponding column index to find the value in the vector, and then multiplies the two values. Combining the conversion and multiplication functions into one, the optimized function can directly return the resulting tensor without returning the triple of arrays from the CSR conversion.

We also implemented the Coordinate (COO) algorithm to represent the pruned matrices, testing if they will provide an improvement in multi-threaded performance over CSR. Similarly to CSR, COO greatly reduces the memory usage of sparse matrices and should allow for significant improvements in the performance of the matrix multiplication algorithm.

Algorithm 3 Pseudo code for COO multiplication Algorithm

Require: nnz, ROW_INDEX, COL_INDEX, V, x
Ensure: $result$
for $i \leftarrow 0$ **to** $nnz - 1$ **in parallel do**
    $col \leftarrow$ COL_INDEX[i]
    $row \leftarrow$ ROW_INDEX[i]
    $val \leftarrow$ V[i]
    $sum \leftarrow val \times x[col]$
    $result[row] \leftarrow result[i] + sum$
end for

A dense matrix is converted to COO by creating arrays: V, which stores the non-zero values in the matrix; CO_INDEX, which stores the column indices of the non-zero values; and ROW_INDEX, which stores the row indices of the non-zero values. Multiplication is done by iterating with one pass through all 3 of these arrays, getting each non-zero element and its position in the array, and multiplying it with the corresponding element in the vector, and adding the result to the corresponding row index in the output vector. As with the CSR multiplication algorithm, we directly return the resulting tensor without returning the triple of arrays from the COO conversion.

Because the elements in COO are stored element-wise, that is, each element in each of the 3 arrays that make up the COO representation correspond to one element in the represented matrix, it may be easier to parallelize these elements evenly between different threads, achieving better load balancing between threads and stronger multi-threaded performance.

4 Experimental Setup

In this section, we describe the hardware and software used for our experiments, as well as the experimental settings and evaluation metrics employed.

4.1 Hardware and Software

We conducted our experiments on a single snappy node 1 of a high-performance computing cluster on Courant Institute of Mathematical Sciences(CIMS) server. The node has two Intel Xeon E5-2680 processors, each with 20 physical cores running at 2.80GHz, and 128GB of DDR4 memory. We do not configure any GPU on it to build up a CPU-only running environment. All experiments were conducted on CentOS 7, using PyTorch 2.0.0+cpu with Python 3.9 and gcc 9.2.

4.2 Experimental Settings

We evaluated our proposed method on Multi-Layer Perceptrons (MLP). The MLP model consists of a sequence of a modifiable number of linear layers with ReLu activation. All of the hidden layers between the input and output linear layers have the same customizable dimension.

To ensure that the output of the accelerated models matches that of the original PyTorch model, we set the initial weights to be the same in both models before parallel execution. This was accomplished by overwriting the weights of all models with the same randomly generated weights.

We also allow for a choice of the number of threads to use for the parallel versions of the models. These threads will automatically be load balanced for the matrix multiplication during the forward pass of the model.

We use a pruning rate of 99.5% in all of our experiments. While this seems very high, [2] show that models can retain 90% or more accuracy on some tasks with this level of pruning. We do not expect the sparse models to be able to compete with the level of optimization that dense matrix multiplication has achieved unless the pruning rate is quite high.

4.3 Evaluation Metrics

We measure the parameter count of each of the models to measure memory usage. Models that will have most of their weights pruned and use a sparse representation should have significantly fewer parameters than the models which represent all of their weights with a dense matrix.

We also measure the total running time to complete a set number of forward passes through the model with equivalent input data. This will allow us to compare the speed of the different models, and how efficient the parallel models are compared to the single-threaded versions.

We have a model running fully on the PyTorch library, which we will use as a control to verify correctness of the other models. We will then have one running with dense computations through the C++ plugin (the "fused" calls into C++ to do the matrix-vector multiply and relu in a single call within C++, the "full forward" version also iterates through the layers of the network in C++), single-threaded models running the C++ models with the CSR and COO sparse algorithm, and multi-threaded versions of each of those sparse models.

Functional Correctness Given the same input data and initialized weights, we compare the output matrices from the PyTorch version and CPP extension, respectively. If any value in the corresponding position matches its counterpart, then we know the multi-threading enhanced implementation is functionally correct.

5 Experiments and Analysis

We ran each experiment 100 times and report the model sizes and total timings.

Model	**Size**	**Timing(s)**
Dense Python	8652800	0.877
Dense CPP primitives	8652800	0.757
Dense CPP full forward	8652800	0.698
Sparse CSR Model 1 thread	92677	153.317
Sparse COO Model 1 thread	129792	133.312
Sparse CSR Model 32 threads	92677	20.867
Sparse COO Model 32 threads	129792	15.123

Table 1. Run-times of each algorithm for the wide, shallow model: 3 layers, 2048 hidden features.

Model	**Size**	**Timing(s)**
Dense Python	426240	0.126
Dense CPP primitives	426240	0.085
Dense CPP full forward	426240	0.070
Sparse CSR Model 1 thread	6063	9.363
Sparse COO Model 1 thread	6393	6.817
Sparse CSR Model 32 threads	6063	1.890
Sparse COO Model 32 threads	6393	1.303

Table 2. Run-times of each algorithm for the medium model: 7 layers, 256 hidden features.

Model	Size	Timing(s)
Dense Python	8652800	0.272
Dense CPP primitives	393344	0.165
Dense CPP full forward	393344	0.142
Sparse CSR Model 1 thread	5901	10.013
Sparse COO Model 1 thread	7032	6.184
Sparse CSR Model 32 threads	5901	3.117
Sparse COO Model 32 threads	7032	2.051

Table 3. Run-times of each algorithm for the deep, narrow model: 24 layers, 128 hidden features.

5.1 Python vs C++

We see some small speedups by calling into C++ rather than writing our inference loops in pure python PyTorch code. This makes sense as it reduces the number of times that the python runtime has to call into C++ by combining multiple underlying computational library calls with each call into C++.

5.2 Dense vs Sparse

We see significant slowdowns moving from dense computation to single-threaded sparse computation. This makes sense as the dense libraries have been highly optimized and can take advantage of vector computation instructions on the CPUs which dramatically speed up the runtime. Without multithreading, our sparse algorithm is far too costly to run.

However, we are able to speed up our networks by splitting the computation across threads. The speedup varies based on the shape of the network. For the shallow, dense network, we see 7-10x speedup with 32 threads. For the deep, narrow network (more layers, fewer parameters per layer) we see smaller speedups of 3-5x. This makes sense because there is less time spent within the parallelized computational block and more time passing vectors from one layer to the next. Anecdotally, these speedups were even higher when available computational resources were less congested.

With a pruning rate of 99.5%, depending on the size of the model, we are able to reduce the total number of parameters required for the model by 98-99% using the sparse matrix formats. This reduction in memory usage is a huge advantage for settings where one wants to deploy on memory-limited devices.

CSR vs COO It is noticeable that throughout all tests, COO consistently achieves faster single-thread performance, and COO has a better speedup when multi-threaded. The CSR algorithm appears to reduce more of the parameters from the dense matrix compared to COO, yet still performs worse in sequential execution than COO. This may be down to how values are stored as vectors in our COO implementation, providing better cache coherence and removing the overhead of using tensors, as we do with CSR. Further, we split the load among threads for CSR by row, whereas with COO, we split the load among threads by elements. The rows are not guaranteed, and usually do not, have the same number of elements as each other. Consequentially, we appear to achieve significantly better load balancing with the COO multiplication algorithm, and reach a better speedup as a result.

In summary, CSR allows us to save slightly more memory by reducing the total number of model parameters further than COO. On the other hand, COO achieves better single-threaded speed and reaches a greater speedup as we add more threads than CSR. Therefore, CSR is more optimal when memory efficiency is prioritized, while COO should be preferred when some memory efficiency can be sacrificed for additional performance.

6 Conclusions

With neural networks becoming increasingly powerful, it will become important to find ways to deploy them on the devices that we use day-to-day. We may tolerate some applications taking a while to compute, but if the models cannot be compressed to smaller memory footprints than exist on the devices then they won't even have the chance to try. Our approach prunes larger neural networks, reducing memory usage by nearly the full pruning rate. Existing literature shows that depending on

the application, these models can retain 90% or higher performance on their task even after this pruning. By using OpenMP parallelization to get up to a 10x speedup with a sparse matrix-vector multiplication algorithm, these smaller networks can still run reasonably fast.

Future work would explore alternative sparse matrix formats as well as cover different types of neural network architectures.

References

1. Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
2. Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, March 2019. arXiv:1803.03635 [cs].
3. Michela Paganini. Pruning Tutorial — PyTorch Tutorials 2.0.0+cu117 documentation.
4. Custom C++ and CUDA Extensions — PyTorch Tutorials 2.0.0+cu117 documentation.
5. Marten Lienen and Stephan Günnemann. torchode: A Parallel ODE Solver for PyTorch, January 2023. arXiv:2210.12375 [cs, math].
6. Floris Laporte, Joni Dambre, and Peter Bienstman. Highly parallel simulation and optimization of photonic circuits in time and frequency domain based on the deep-learning framework PyTorch. *Scientific Reports*, 9(1):5918, April 2019. Number: 1 Publisher: Nature Publishing Group.
7. Yiwen Shao, Yiming Wang, Daniel Povey, and Sanjeev Khudanpur. PyChain: A Fully Parallelized PyTorch Implementation of LF-MMI for End-to-End ASR, May 2020. arXiv:2005.09824 [cs, eess].
8. William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs, July 2022. arXiv:2207.00257 [cs].
9. Steven Huss-Lederman, Elaine M. Jacobson, Anna Tsao, Thomas Turnbull, and Jeremy R. Johnson. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, Supercomputing '96, pages 32–es, USA, November 1996. IEEE Computer Society.
10. Hyuk-Jae Lee, James P. Robertson, and José A. B. Fortes. Generalized Cannon's algorithm for parallel matrix multiplication. In *Proceedings of the 11th international conference on Supercomputing - ICS '97*, pages 44–51, Vienna, Austria, 1997. ACM Press.
11. Vipul Gupta, Shusen Wang, Thomas Courtade, and Kannan Ramchandran. OverSketch: Approximate Matrix Multiplication for the Cloud. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 298–304, December 2018.
12. Andris Ambainis, Yuval Filmus, and François Le Gall. Fast Matrix Multiplication: Limitations of the Coppersmith-Winograd Method. In *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*, STOC '15, pages 585–593, New York, NY, USA, June 2015. Association for Computing Machinery.
13. Paul Heckbert. Fourier Transforms and the Fast Fourier Transform (FFT) Algorithm.