Lowering Memory in the Batch Normalization Layer to Try to Speed Up Training

Eric Eng (epe24)

July 24, 2024

Abstract

In past work, practictioners showed was how to use recomputation to limit HBM storage. Doing this sacrifices compute for memory, but we can end up speeding up our operations if they are memory bound. This led to the question, can we alter some of our algorithms to sacrifice compute and improve on memory, leading to an overall speedup? We focused on the batch normalization layer. We wanted to know if recomputing the normalized data instead of storing it on HBM could influence peak memory in neural networks, and speed up training time. When we ran this over a 1D ResNet, we found that we were able to lower the maximum memory of the architecture during training, but it did not affect training time. Because we were able to lower total memory, it is possible that if we distributed over multiple machines, we could potentially see a speedup.

Background

Recomputation is a technique first explored in the paper ”Training deep nets with sublinear memory cost [1]”. In the paper, they chose to recompute intermediate activations during the backward pass rather than storing them during the forward pass. This approach significantly reduces the memory requirements for training deep neural networks.

Many papers have built off of this paper and used recomputation in effective ways. The paper that inspired this project was the ”FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness”[2] paper. In the FlashAttention paper, the authors used tiling, recomputation, and kernel fusion to speed up the attention process. In this project, we wanted to see how we could try similar techniques to speed up other memory-bound algorithm. We chose the batch normalization layer [4]. The batch normalization layer is a layer that takes the mean and variance across a feature and then scales and shifts data, usually after passed through a neural layer. It was originally created to address internal covariate shift, which was eventually disproven, but has stayed in machine learning for how well it works. CUDA has already written the batch normalization layer in one kernel, and there did not seem to be an obvious way to incorporate tiling into the layer. So we focused on recomputation. In the FlashAttention paper, the authors showed how storing fewer values on high-bandwidth memory (HBM) could speed up the attention layer by using recomputation.

Introduction

To introduce the problem, when we put data through a batch normalization layer, we store the data on HBM three separate times. First, we store the original data, then we store the normalized data, and finally we store the output data. We need to store all of these copies for the backwards pass to compute the gradients with respect to each part. The idea behind this project was to not store the intermediate normalized data and see if it could lower the memory of a network. If it lowered the memory, we wondered if it could speed up training in the process. This is not guaranteed because we are sacrificing memory for compute. This is where the recomputation comes in. To make up for not storing our normalized data for the backwards pass, we have to recompute the normalized data when we're calculating our gradients. In the experiment section, we talk about how we did it. We tried over a smaller network and a larger one.

Gradient Derivation

Because we was rewriting the backward pass, we had to figure out how to take the gradients within the batch norm so the training process still worked. So first we went through and wrote out the gradients for the 2d case, when we're looking at data with the shape (batch_size, features). These gradients were fairly straightforward.

Given:

- input: Input to the batch normalization layer.
- mean: Mean of the input.
- var: Variance of the input.
- γ: Scale parameter.
- β: Shift parameter (not involved in this specific gradient).
- eps: A small constant for numerical stability.
- grad_output: Gradient of the loss function with respect to the output of the batch normalization layer.

Normalized Input:

$$\text{input\_normalized} = \frac{\text{input} - \text{mean}}{\sqrt{\text{var} + \text{eps}}}$$

Output of Batch Normalization:

$$\text{output} = \gamma \times \text{input\_normalized} + \beta$$

Backward Pass of the Input With Respect to the Loss
1. Gradient through output to normalized input:

$$\frac{\partial L}{\partial \text{input\_normalized}} = \text{grad\_output} \times \gamma$$

2. Gradient of normalized input with respect to input:

$$\frac{\partial \text{input\_normalized}}{\partial \text{input}} = \frac{1}{\sqrt{\text{var} + \text{eps}}}$$

$$\frac{\partial L}{\partial \text{input}} = \frac{\partial L}{\partial \text{input\_normalized}} \times \frac{\partial \text{input\_normalized}}{\partial \text{input}} = \text{grad\_output} \times \gamma \times \frac{1}{\sqrt{\text{var} + \text{eps}}}$$

3. Account for the mean and variance:

$$\frac{\partial L}{\partial \text{mean}} = -\sum \left(\text{grad\_output} \times \gamma \times \frac{1}{\sqrt{\text{var} + \text{eps}}} \right)$$

$$\frac{\partial L}{\partial \text{var}} = -\frac{1}{2} \sum \left(\text{grad\_output} \times \gamma \times (\text{input} - \text{mean}) \times (\text{var} + \text{eps})^{-\frac{3}{2}} \right)$$

Using the derivatives of the mean and variance with respect to the input:

$$\frac{\partial \text{mean}}{\partial \text{input}} = \frac{1}{N}$$

$$\frac{\partial \text{var}}{\partial \text{input}} = \frac{2}{N}(\text{input} - \text{mean})$$

Combine them:

$$\frac{\partial L}{\partial \text{input}} + = \frac{\partial L}{\partial \text{mean}} \times \frac{\partial \text{mean}}{\partial \text{input}} + \frac{\partial L}{\partial \text{var}} \times \frac{\partial \text{var}}{\partial \text{input}}$$

Backward Pass of γ With Respect to the Loss

$$\frac{\partial L}{\partial \gamma} = \sum (\text{grad\_output} \times \text{input\_normalized})$$

Backward Pass of β With Respect to the Loss

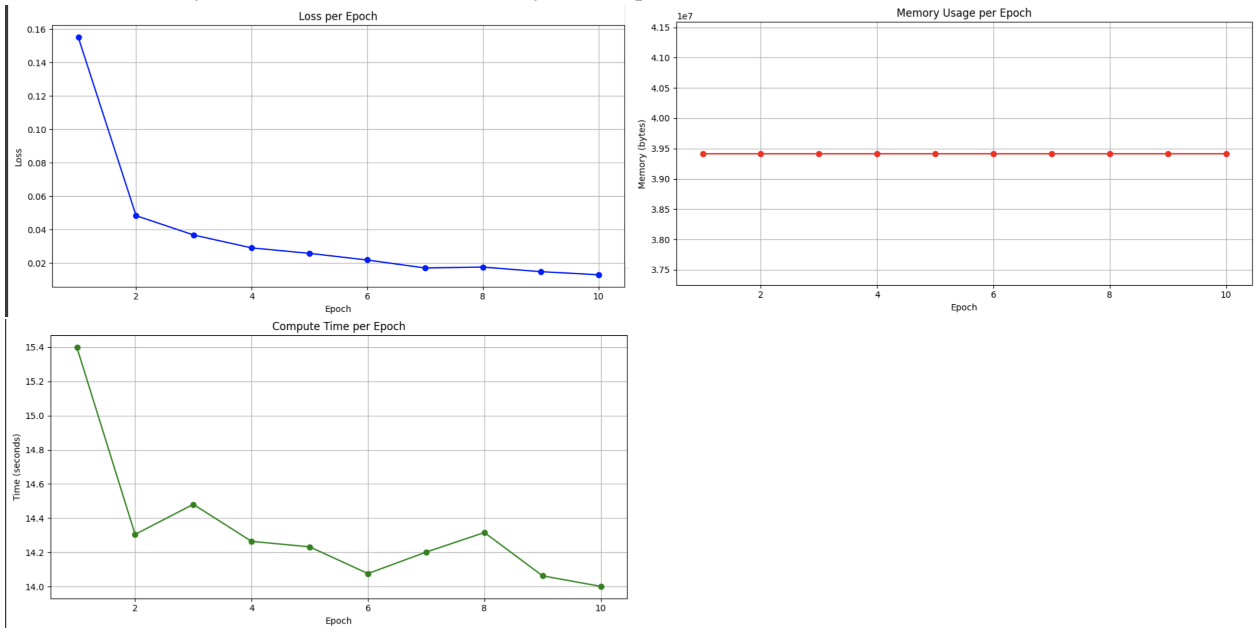
$$\frac{\partial L}{\partial \beta} = \sum (\text{grad\_output})$$

After calculating, we implemented a basic check over networks to check that the gradient values were the same. They were over multiple steps. Also, after training, the losses converged to about the same values. Each time we ran a network over an entire dataset, there were small differences due to randomness.

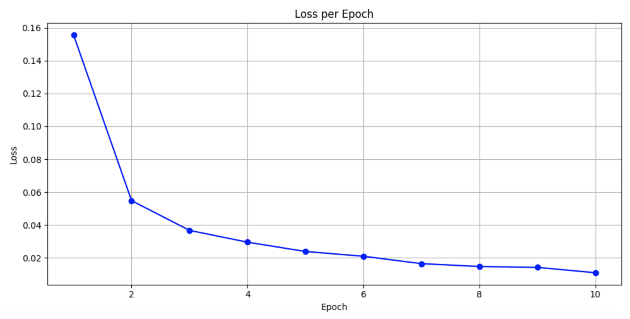
Experiments

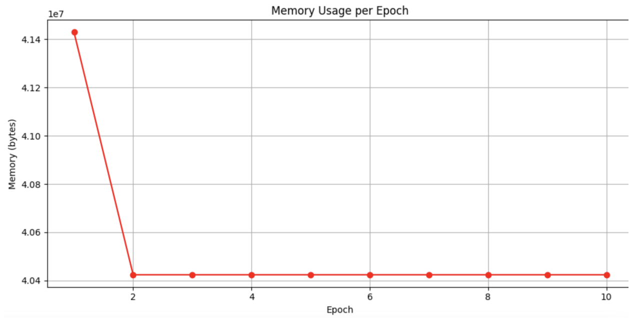
Originally, we tried comparing to the Pytorch implemeted batch normalization layer. Unfortunately, because of how optimized the CUDA kernel for the batch normalization was, this was very difficult. We could not isolate the affect of recomputation, because the cuDNN kernel is not public. So we ran everything over the Pytorch version, our version without recomputation, and our version with recomputation. Also, his is all implemented over BatchNorm1D for the simplicity of gradients. A similar process could also be done for the more popular BatchNorm2D. Finally, because we wanted to overwrite the backwards pass in addition to the forwards pass, we did it with torch.autograd.

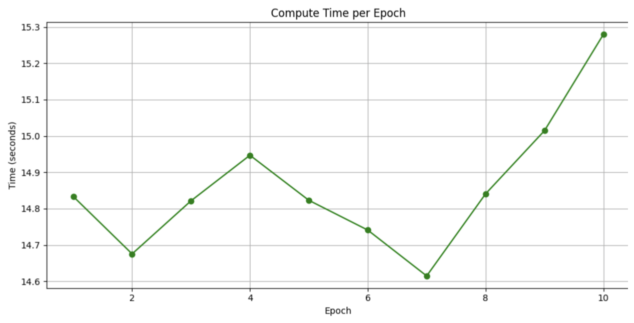
For the experiment, we wanted to test over two cases. First, we wanted to test over a smaller scale. We chose the LeNet [5]. We trained it for 10 epochs over the MNIST dataset, with the Adam optimizer, a learning rate of 0.001, and a 64 batch size. Firstly, here are the results with the Pytorch impelmented version:



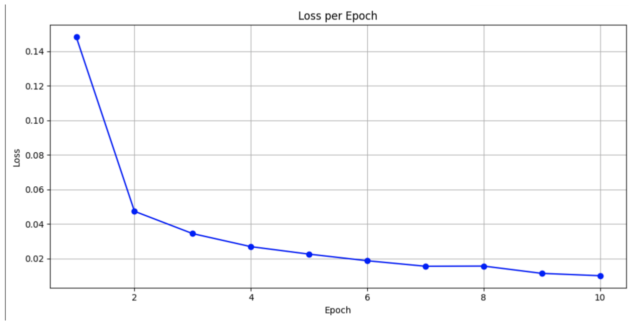
Next, we ran this with the same hyperparameters over our implementation without recomputation. Here are the results:

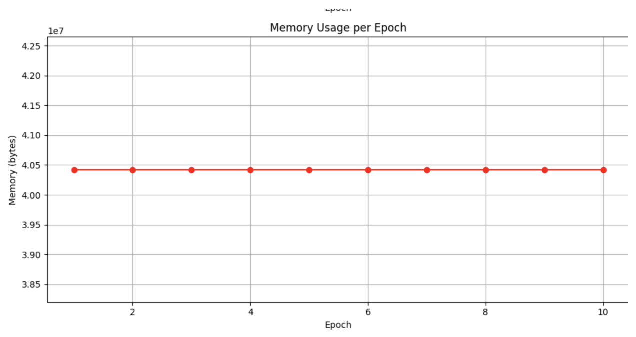


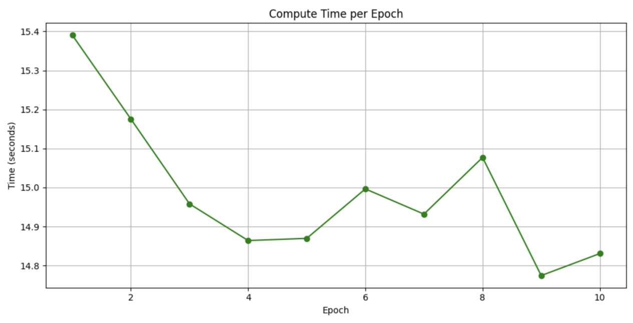




Finally, we tried it with our implementation with recomputation over the same hyperparameters. Here are the results :





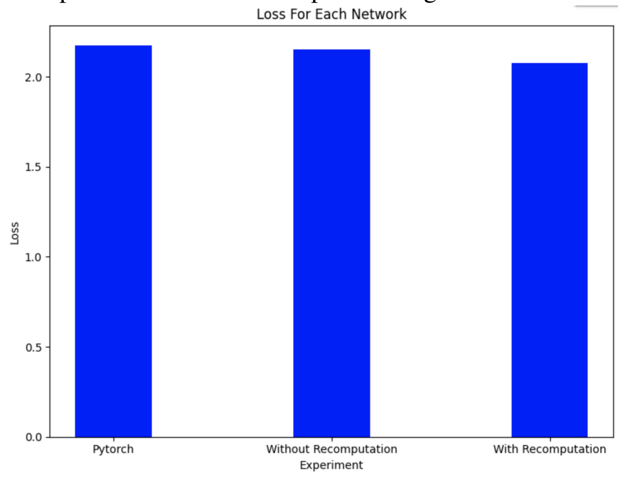


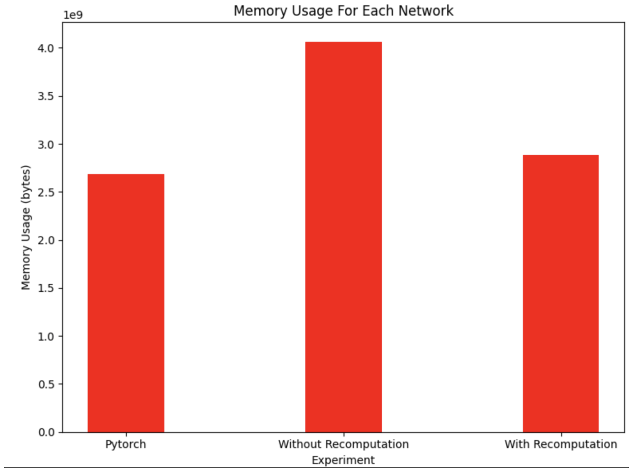
These results were fairly consistent across multiple iterations. The loss graphs were similar across all three iterations. The best results were the Pytorch implementation, and in this case the with recomputation and without recomputation versions had fairly similar results.

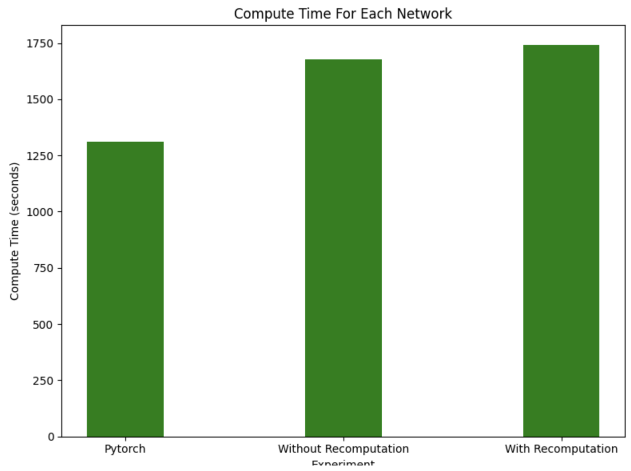
Now we want to move on to our second experiment. In this experiment, we used an adapted ResNet [3] with our 1d batch normalization layer on the SPEECHCOMMANDS dataset in torchaudio. We wanted to test the idea out on a larger scale to see how performance changed. Since this case is more memory intensive, we were hoping that it would highlight the difference more than in the LeNet case.

In reality, it was much more informative than the LeNet version. In this case, we used Adam optimizer with learning rate 0.001, with batch size 32. Also, all of the training was done on a NVIDIA Tesla T4 GPU.

Unfortuately, each iteration took about 30 minutes to run. Because of that, we would lose access to Colab GPUs due to memory usage. So we were only able to run one epoch per algorithm per run. We chose to do this instead of doing 3 epochs for each algorithm, because every time we reset our GPU we got a different baseline for maximum bytes. So to try to standardize, we did one epoch on each 3 time. Here are the results from the median gap between the recomputation and non recomputation algorithms.







We can see that there is a significant memory reduction in this case. It shines through more than the LeNet case because the SPEECHCOMMANDS dataset is more memory intensive, and the ResNet uses substantially more batch normalization layers than the implemented LeNet. Unfortunately, this did not lead to a direct speedup in training. This is likely because the added compute in the backwards pass balanced out the memory cost saved.

Conclusions and Future Work

Summing up the results, in our tests to see how recomputation affects the batch normalization layer, we barely saw any changes over our LeNet. But, when we implemented our ResNet, we saw a significant difference in peak memory usage, but this did not translate to training time.

The first obvious follow up would be to implement the entire batch normalization kernel and do the recomputation in one kernel, so we could see an exact comparison to the Pytorch version. A second follow up we could do would be to fully shard a model and see if this memory reduction could hypothetically speed up training. It would be interesting to see if we could parallelize our extra compute across enough GPUs, so that this reduction in memory could translate better to training time.

Link To Code

https://colab.research.google.com/drive/1MfHrTuY0JyyZaeSftgMIhk953ZO2Haj2#scrollTo=-jPkaSzDRHNZ

References

[1] Tianqi Chen, Bing Xu, Chiyuan Zhang, Carlos Guestrin. *Training deep nets with sublinear memory cost*. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1725–1734, 2016.

[2] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, Christopher Ré. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. arXiv preprint arXiv:2205.14135, 2022.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Deep Residual Learning for Image Recognition*. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[4] Sergey Ioffe, Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. In Proceedings of the 32nd International Conference on Machine Learning, pages 448–456, 2015.

[5] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner. *Gradient-based learning applied to document recognition*. Proceedings of the IEEE, 86(11):2278–2324, 1998.