Deep Learning Systems Lab 1

Bayard Walsh

October 26 2023

1 Experiments

I trained NanoGPT on my personal computer (Intel Core i7 processor) with no GPU, and I chose the following optimizations: Batch Size reduction, Gradient Accumulation with micro batches, SGD Learner Optimizer, Mixed Precision, and Full Precision bits. For each of these 5 optimization strategies, I varied the following three model configurations for training: the number of attention heads, number of layers, and the dimension of the embedding. These are the baselines used: batchsize=12, dtype ='float32', nlayer = 4, nhead = 4, nembd = 128, optimizer= AdamW, gradient accumulator= No. For each configuration I doubled the value (nhead = 8, nlayer = 8, nembd = 256). When reducing batch size, I used batchsize=6. Activation Check pointing is used on all training runs. I used the python memray library [Blo23] and two time.time() calls from the python time library to record memory and time metrics, with performance given by the NanoGPT model after 2000 iterations.

2 Expectations

Because of the overall larger increase in embedding size compared to the increase in heads and layers, I expect that the model will train significantly slower in those instances, because of the extra computation needed to train in the far larger vector space. According to a paper that expands on the BERT model [al20b], the authors decreased the embedding dimension while maintaining the same performance, so I believe the increase in performance will be marginal, especially because the NanoGPT already has an embedding size equal to BERT (128) before the increase. For increasing heads performance, based on the findings in the BERT paper [al19], I believe that more heads will decrease the error rate, though on a small scale because of the size of the model. For optimizations, Mixed and Full precision bits should perform similarly in terms of run time and memory usage, I predict the gain from the expanded magnitude of `bfloat16` will be negligible, and expect the decreased fractional precision in `bfloat16` will cause a slight drop off in performance. I believe that the Gradient Accumulator will have the longest run time because of the computation needed to combine

mini-batches, though memory usage will decrease (even without the benefit of multiple GPUs). [Lam21]

3 Results

Surprisingly, `bfloat16` and `float16` had the same error rate over all three configurations, implying that for NanoGPT the 3 bits of difference between the two data types made no impact on the model, although the full precision bits had a slightly faster runtime across all optimizations. In terms of performance relative to configurations, doubling heads was universally better than doubling layers, and had a faster run time, better performance, and less memory used in all 5 optimization cases. This aligns with one of the discoveries on performance mentioned in the paper on Scaling Laws [al20a]. The paper tests models up to 207 layers and finds that while the number of layers increases the model performance, at 6 layers this benefit plateaued, and longer, skinnier models were less effective. Therefore training at 8 layers at the cost of fewer heads and parameters indicates that NanoGPT has likely reached this point of stagnation from increased layers. Along with worse performance across the board, doubling the number of layers also vastly increased the memory usage compared to the other 10 cases. This is due to all the added memory consumption of maintaining more weights with double the layer size. Surprisingly, batch size reduction caused a decrease in run time and an increase in error, having the worst performance across all tests. This directly contradicts the expected behavior of Batch Size reduction. A larger batch size causing an increase in run time is understandable, because I'm not benefiting from any of the parallelism that I would have with separate GPU training co-currently, and the decrease in performance is likely due to a lack of convergence from such a small batch size. Because batch size reduction can often decrease error rate, the limit of this idea is testing on a batch size of 1 for theoretically ideal performance, however, this results in a more jagged path to optimization, because the model has more noise and needs to update the weights more. Therefore considering I trained on 2000 iterations, the batch size of 6 was likely too small to converge. The authors of a paper focused on small batch size training[Dom18] found a "sweet spot" in the batch size of 32 for their experiments, with a decrease in performance for both larger and smaller size batches. Therefore, on this scale batch size reduction had the opposite effect as mentioned in the lecture, because I trained with far less than the sweet spot size, and didn't have enough iterations to reach convergence. Additionally, while the model was training the per iteration error rate varied far more than other training configurations, indicating the impact of extra noise brought by a smaller batch size. Also, as micro batching was a separate optimization from batch size reduction, the smaller batch size didn't benefit from this technique which is often paired with small batch size training. While memory, run time, and loss are weighted differently based on the needs and resources of a given model designer, which NanoGPT configuration performed the 'best', as an equally weighted combination of the three metrics? To determine this I

applied the following heuristic equation and ranked the 15 tests *(see table page 5 for results)*.

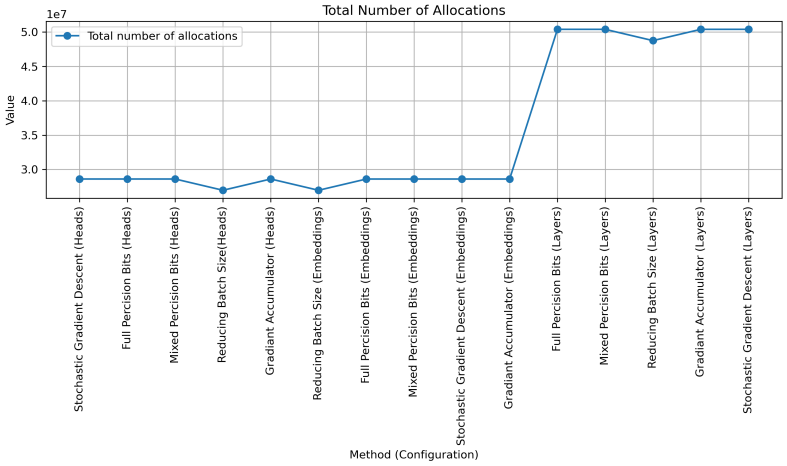
$$\text{Rank} = 1 \times \text{Accuracy} - \text{RuntimeMax} \times \frac{1}{\text{Runtime}} - \text{MemoryMax} \times \frac{1}{\text{Memory}} \quad (1)$$

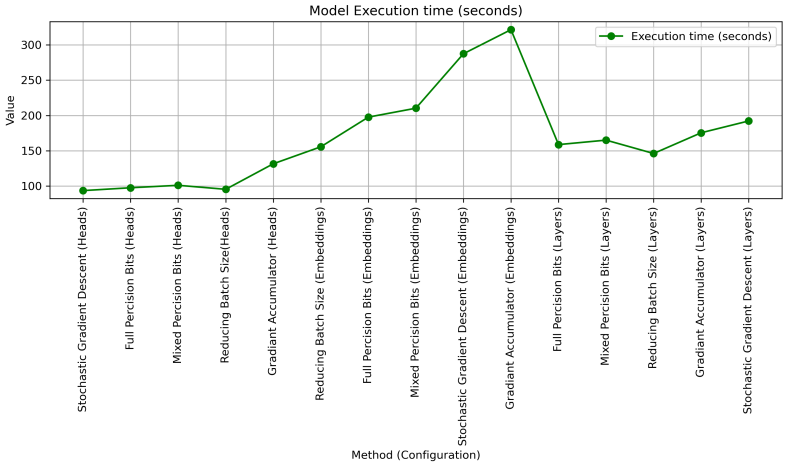
4 Major Sources of Error

A major source of error was recording the experiments during different sittings, and some of the baseline tests appeared slightly off because the CPU performance wasn't consistent, so having a designated processor for training would be ideal for further experimentation. Other sources of error include rounding in the returned performance metric, as the model only returned 4 digits of accuracy. Also, using memray with the Python training script could have impacted the run time by running an additional program co-currently.

5 Improving the Experiment

An area where I could have improved the results of the experiment was the amount that I increased the configurations by, specifically the embedding size. For each case I doubled the baseline amount to draw out a correlation, however in the case of embedding it strongly skewed the output of the data for run time. While I intended to keep the impact of each section proportional, adding 4 heads has less of an impact on compute power than increasing the embedding size by 128. Note that the Scaling Law paper mentioned that the overall size of the model is a better indicator of performance than the shape of models [al20a], so by unintentionally increasing the overall size of the model, an overall increase in performance will skew the proportional impact of increasing embedding size. A way to remedy this would be to estimate a baseline resource bound, transferable between computation and memory, and then estimate the equivalent amount of this resource used by increasing a proportional number of heads and embedding size, and measure performance from there (a similar approach was adopted by the Scaling Law paper [al20a] where the authors hold total non-embedding parameter count N fixed). This would lead to a less skewed model design, and better inform which optimizations are ideal. Another simple way to improve the experiment is scale- for example, all increased embedding size instances (except batch reduction) had a loss rate of exactly 1.4374 (though the configuration had vastly different run times, with Full precision bits having the fastest at 197.67 seconds). This alignment in error indicates that the scale in the NanoGPT model was not large enough for these approaches to show the more granular changes in performance. Furthermore, as batch size reduction had the opposite effect on performance and run time, clearly more computation was needed to effectively measure the optimization.


Total Number of Allocations
1e7
Total number of allocations
Value
5.0
4.5
4.0
3.5
3.0
Stochastic Gradient Descent (Heads)
Full Percision Bits (Heads)
Mixed Percision Bits (Heads)
Reducing Batch Size(Heads)
Gradiant Accumulator (Heads)
Reducing Batch Size (Embeddings)
Full Percision Bits (Embeddings)
Mixed Percision Bits (Embeddings)
Stochastic Gradient Descent (Embeddings)
Gradiant Accumulator (Embeddings)
Full Percision Bits (Layers)
Mixed Percision Bits (Layers)
Reducing Batch Size (Layers)
Gradiant Accumulator (Layers)
Stochastic Gradient Descent (Layers)
Method (Configuration)


Model Execution time (seconds)
Execution time (seconds)
Value
300
250
200
150
100
Stochastic Gradient Descent (Heads)
Full Percision Bits (Heads)
Mixed Percision Bits (Heads)
Reducing Batch Size(Heads)
Gradiant Accumulator (Heads)
Reducing Batch Size (Embeddings)
Full Percision Bits (Embeddings)
Mixed Percision Bits (Embeddings)
Stochastic Gradient Descent (Embeddings)
Gradiant Accumulator (Embeddings)
Full Percision Bits (Layers)
Mixed Percision Bits (Layers)
Reducing Batch Size (Layers)
Gradiant Accumulator (Layers)
Stochastic Gradient Descent (Layers)
Method (Configuration)

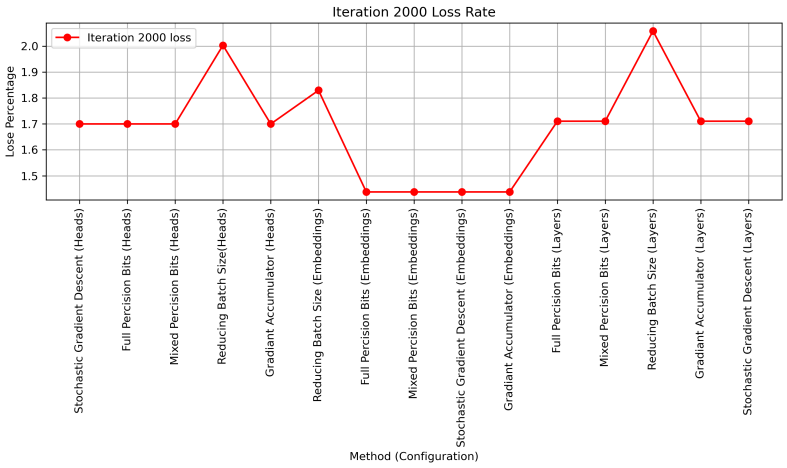

Iteration 2000 Loss Rate
Iteration 2000 loss
Lose Percentage
2.0
1.9
1.8
1.7
1.6
1.5
Stochastic Gradient Descent (Heads)
Full Percision Bits (Heads)
Mixed Percision Bits (Heads)
Reducing Batch Size(Heads)
Gradiant Accumulator (Heads)
Reducing Batch Size (Embeddings)
Full Percision Bits (Embeddings)
Mixed Percision Bits (Embeddings)
Stochastic Gradient Descent (Embeddings)
Gradiant Accumulator (Embeddings)
Full Percision Bits (Layers)
Mixed Percision Bits (Layers)
Reducing Batch Size (Layers)
Gradiant Accumulator (Layers)
Stochastic Gradient Descent (Layers)
Method (Configuration)

Table 1: Methods Performance (ranked by equation)

Method	Allocations	Time (s)	Loss (Iter 2000)
Stochastic Gradient Descent (Heads)	28616229	93.58	1.7000
Full Precision Bits (Heads)	28615639	97.63	1.7000
Mixed Precision Bits (Heads)	28615717	101.10	1.7000
Reducing Batch Size (Heads)	26977732	95.43	2.0038
Gradient Accumulator (Heads)	28615643	131.56	1.7000
Reducing Batch Size (Embeddings)	26977752	155.64	1.8298
Full Precision Bits (Embeddings)	28612316	197.67	1.4374
Mixed Precision Bits (Embeddings)	28612401	210.30	1.4374
Stochastic Gradient Descent (Embeddings)	28612910	287.43	1.4374
Gradient Accumulator (Embeddings)	28612316	321.56	1.4374
Full Precision Bits (Layers)	50383451	158.76	1.7106
Mixed Precision Bits (Layers)	50383527	165.09	1.7106
Reducing Batch Size (Layers)	48742794	146.17	2.0589
Gradient Accumulator (Layers)	50383456	175.50	1.7106
Stochastic Gradient Descent (Layers)	50384042	192.25	1.7106

References

[Dom18] Carlo Luschi Dominic Masters. *Revisiting Small Batch Training for Deep Neural Networks*. 2018. arXiv: 1804.07612 [cs.CL].

[al19] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[al20a] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.0836 [cs.CL].

[al20b] Zhenzhong Lan et al. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2020. arXiv: 1909.11942 [cs.CL].

[Lam21] Joel Lamy-Poirier. *Layered gradient accumulation and modular pipeline parallelism: fast and efficient training of large language models*. 2021. arXiv: 2106.02679 [cs.CL].

[Blo23] Bloomberg. *memry*. https://github.com/bloomberg/memray. 2023.