Predicting Peak VRAM Usage in Local Fine-tuning of Large Language Models

Scott Salisbury
Department of Computer Science and Engineering
Ohio State University
Columbus OH, USA
salisbury.100@osu.edu

***Abstract*—This work explores where peaks in VRAM usage occur during local fine-tuning of LLM's on personal computers' GPUs and presents a tool to aid in fully utilizing GPU memory capacity with low risk of out-of-memory crash. It uses techniques from the QLoRA paper [1] as well as gradient checkpointing to make fine-tuning a multi-billion parameter LLM on a gaming GPU possible, and then leverages Pytorch tools for profiling and visualizing GPU memory usage. This uncovers that there are at least 7 different types of peak/bottleneck points in the fine-tuning of one single model (Google's Gemma [2] 2b; the 7b model variant adds an 8th type). This and other experimental results highlight how complex a process LLM fine-tuning is, and indicate that it may be infeasible to make a predictor like this without doing substantial experimentation and special code for each supported model. It also highlights how Pytorch's coarse-grained VRAM allocation approach makes VRAM bottlenecks markedly worse even as it reduces memory fragmentation concerns. Finally, this work delivers a predictor for VRAM usage of local fine-tuning runs of 2b and 7b members of the Gemma model family.**

***Index Terms*—Graphics processing units, Natural language processing, Deep learning**

I. Introduction

Fine-tuning an LLM locally (on a personal computer's GPU) can be desirable in some cases for privacy or cost reasons (relative to cloud fine-tuning), especially when fine-tuning on personal documents. Quantization and parameter-efficient fine-tuning (PEFT) libraries make this feasible even for models with parameter counts in the single-digit-billions (as long as one has a gaming GPU with at least 6-8GB of VRAM). However, such fine-tuning runs are still quite prone to crashing from running out of VRAM, which leads to time-wasting trial and error. The motivation for this work is to reduce time spent on trial-and-error by predicting which fine-tuning configurations will maximally utilize the GPU's memory capacity without exceeding it and then present those options to the user.

The original plan was to derive precise rules from model architecture details and first principles, then create a predictor which could easily be generalized to many different models. In practice, experiments with Pytorch's VRAM profiling/visualization tools have made it clear that things are too complicated for that. Pytorch performance optimizations make GPU memory allocations far more coarse-grained than expected, which makes it harder to reliably predict which fine-tuning configurations will be safe and also makes it far harder to determine what exactly is wrong with the assumptions of the predictor's calculation rules. More importantly, experiments with Pytorch's VRAM profiler/visualizations showed that even a single model can have over half a dozen different *types* of places in the training process which could be a global peak/bottleneck in VRAM usage, depending on various things like the relative size of 'sequence length' vs 'non-frozen parameter count' or 'is LoRA being applied to the MLP blocks'. As a result, this paper will describe a fairly narrowly-applicable VRAM usage predictor and discuss the experimental findings about VRAM usage peaks which informed its development.

II. Related Work

Parameter-efficient fine-tuning (PEFT). In 2021, Hu et al [3] introduced Low Rank Adaptation (LoRA), which is still a widely used PEFT technique today. It freezes the base model weights and adding low-rank adapter matrices whose weights can be fine-tuned. This allows effective fine-tuning with smaller dataset s and with less strain on GPU memory (e.g. no need to track optimizer states, gradients, etc. for frozen weights).

Quantization of frozen weights. Last year, Dettmers et al [1] introduced several innovations to allow LoRA-fine-tuning of large models on relatively weaker GPU's. Most importantly, they discovered how to quantize the frozen weights to just 4 bits per weight (instead of 16-32 bits) during fine-tuning and inference. They also invented the NormalFloat4 data type (for less lossy quantization), the technique of double quantization to reduce the memory overhead of the quantization constants, and paged optimizers to make it so that optimizer states don't contribute as much to GPU-memory-usage spikes. This work employs all of those techniques (as implemented in version 0.43.1 of the bitsandbytes OSS library)

Gemma family of models. Earlier this year, Google DeepMind released the Gemma family of open-weights models [2], consisting of a 2B parameter model and a 7B parameter model. This was very useful because it provided a non-outdated model with a parameter count in the *low*-single-digit billions (possible to fine-tune on even 4GiB of VRAM) and also a related model that's closer to the smallest size for Llama models (i.e. 7B or 8B).

III. Design

The original design for the predictor relied on the assumptions that 1) there existed straightforward formulae for calculating things like "GPU memory occupied by the LoRA parameters", "GPU memory occupied by optimizer states", "maximum size of build-up of activations tensors", or "maximum size of build-up of gradients tensors" from the model's details and a given set of peft configuration choices; and 2) that there would be one bottleneck point in the training process where all of those categories of tensors (and other categories of tensors with calculateable overall sizes) would be present in the GPU memory simultaneously. The second assumption was completely wrong and the first is only sometimes right.

As a result, experimental work motivated by getting some empirical grounding for things like 'how does activation memory build-up actually behave when using this particular library's implementation of gradient checkpointing' (for the purpose of making sure the predictor was well-tuned) turned into a more general exploration of the dynamics of GPU memory usage during LLM fine-tuning.

However, the report-generator design (where the predictor would be used to identify promising configurations that would be reported to the user) was implemented. It sweeps over possible combinations of choices for the supported training configurations (admittedly, the set of supported training configuration options is smaller than originally planned). It uses early-stopping logic to avoid wasting time checking configurations that are strictly more VRAM-intensive than configurations that were already found to be nonviable. Finally, it presents the user with the configurations that come closest to fully utilizing the GPU's memory capacity (at peak/bottleneck points) without exceeding it.

IV. Experimental Findings About Types of VRAM-Peak/Bottleneck Points during LLM Fine-tuning

Peak candidate A happens once, before any mini-batches can occur, but the other 7 recur in each mini-batch.

1) **A- Quantization Peak**: This occurs near the end of the initial process of quantizing the frozen weights of the base model. Its size depends only on the choice of model and the quantization level (only 4bit is currently supported, but 8bit quantization might later be added)
2) **B- Forward pass through highest layer**: During the forward pass through the highest layer, the build-up of long-lived activation tensors reaches its highest point. On top of that, large and short-lived 'input' tensors are created as part of performing that forward pass. This peak can be relatively higher (compared to other peak candidates) if sequence length is low.
3) **C- Central Activations Spike**: After the end of the forward pass, there is a spike of 2 or more large stacked 'activation' tensors (on top of the built up long-lived activation tensors). This can become very large when batch size and/or sequence length are increased
4) **D- Central Autograd Spike**: After the end of the forward pass and the Central Activations Spike, there is a spike of 2 or more large stacked 'autograd_detail' tensors (on top of one of the large short-lived activation tensors from the previous peak as well as the built up long-lived activation tensors). This can also become very large when batch size and/or sequence length are increased
5) **E- Spike in middle of backward pass through highest layer**: At the start of the backward pass, the build-up of long-lived 'activation' tensors is mostly still present. That, combined with the large 'temporary' tensors which are created during the backward pass through a layer, can create a high peak
6) **F- Final large spike in backward pass through highest layer**: If LoRA is applied to the MLP blocks and the LoRA-r is high enough, the final large 'temporary' tensor spike during a layer's backward-pass may actually be a higher peak than the one from the middle spike.
7) **G- Spike in middle of backward pass through lowest layer**: Near the end of the backward pass, there will have been a large build-up of gradient tensors. That, combined with the large 'temporary' tensors which are created during the backward pass through a layer, can create a high peak.
8) **H- Final large spike in backward pass through lowest layer**: If LoRA is applied to the MLP blocks and the LoRA-r is high enough, the final large 'temporary' tensor spike during a layer's backward-pass may actually be a higher peak than the one from the middle spike
9) **I- Autograd spike at end of minibatch**: If LoRA was applied to the token embedding matrix, there will be a spike from 2 'autograd_detail' tensors being created on top of the gradient-tensors-build-up after the end of the backward pass. If LoRA-r is high enough, this can actually be a taller peak than the peaks in the final part of the backward pass.

V. Evaluation

Only three of the peaks (C, D, and G) are initially supported because the rest were discovered too late for sufficient data to be collated about them.

The calculation rules for the 3 supported peaks were tested against values of LoRA-r and batch-size which had never been used when gathering data to define/tune those calculation rules:

There is also a version of the results table in the associated ”os final project notes” Excel file (in the ”Evaluation” sheet), which includes auto-calculated error margins for each prediction.

VI. Issues you encountered

A. *Getting the profiler working*

. Firstly, there were delays involved in getting to the point where I could do local training runs that used Pytorch's profiler to visualize VRAM usage. I installed CUDA/etc. on my laptop, then found that Pytorch's memory profiler couldn't work if it was running in a Windows OS. Then I installed WSL (Windows Subsystem for Linux) on my laptop(16GB of main RAM), set up the Python virtual environment inside it, and tried again- I was then able to do training runs and benefit from a crude form of GPU memory profiling in Pytorch, but the main functionality (an interactive and highly-detailed visualization of VRAM usage over time) wasn't working (Pytorch would try to generate that visualization for 1-3 hours and then crash). My best guess is that generating the interactive visualization is highly main-RAM-intensive, which caused the extremely long runtimes (thrashing) and eventual crashes. The reason I think that is that, after installing CUDA and WSL on my desktop (64GB of main RAM) and setting up the appropriate Python virtual environment, the profiler created the high-value visualizations in under 10 minutes.

B. *Coarse-grained memory allocation*

One major source of confusion was that the 'bitsandbytes' library (for quantizing/compressing the frozen weights of the base model) allocated GPU memory in somewhat confusing ways. It would allocate a very large number of moderately-sized tensors for quantized versions of the base model's weight matrices, but it also allocates a massive single tensor (e.g. 2.0Gib exactly for Gemma 2b, or 2.9GiB for Gemma 7b) whose purpose is not immediately clear. That is, the moderately-sized tensors add up to an amount of memory equal to roughly half the model's parameter count, which would naively seem to be sufficient for 4bit quantizing of the weights (i.e. 1/2 byte per weight).

Similarly (although to a less extreme degree), Pytorch generally allocates VRAM in larger increments than are strictly necessary, as part of performance optimizations to reduce memory management overhead and memory fragmentation. This made the experimental analysis somewhat trickier.

C. *Under-appreciated complexity of training process dynamics*

I radically underestimated just how complex a basic supervised-fine-tuning training process is. There were over half a dozen possible peak/bottleneck points, and memory-usage categories like ”activations” or ”gradients” contribute to each one to a different degree. Even worse, some potential peak points only sometimes even show up in the training process (e.g. peak candidate type 'I' only appearing if LoRA was applied to the token embedding matrix). Relatedly, in most cases, the 4th of 6 spikes in the backward-pass through a particular layer will be the tallest, but in some conditions involving high LoRA-r (and LoRA being applied to the MLP blocks), the last spike will actually be the tallest within a layer's backward pass. This is what led to the splits between

Model	LoRA r	LoRA component	Batch Size	Predicted C	Actual C	Predicted D	Actual D	Predicted G	Actual G
Gemma 2b	128	Embedding	1	3326757376	3259954688	3330724608	3262995456	3652933488	3454763520
Gemma 2b	128	MLP	1	3892890112	3637440000	3896857344	3640480768	4596815728	4323248640
Gemma 2b	2	Attention	8	3226831872	3212931072	3226569728	3204477952	3339308928	3339152896
Gemma 2b	2	MLP	8	3245190144	3231289344	3244928000	3222836224	3366408064	3365534208
Gemma 7b	64	Embedding	1	7252210176	7249413632	7256161024	7250791424	7764124800	7695238656
Gemma 7b	64	Attention	1	7361425920	7388611072	7365376768	7389988864	8012506240	8032621056
Gemma 7b	2	Attention	8	7171085312	7287743488	7170692096	7279159296	7539089408	7669301760
Gemma 7b	2	MLP	8	7207556096	7324214272	7207162880	7315630080	7590338560	7720861184

TABLE I

RESULTS FOR PREDICTORS OF DIFFERENT PEAKS OF VRAM USAGE

peak candidates E/F and G/H. It's caused by additional gradient tensors being allocated at 3 points partway through that layer's backward pass (but only if LoRA was applied to the MLP blocks!), where those additional gradient tensors could (with high LoRA-r) be larger than the 'temporary' tensors which were deallocated between the 4th and 6th spikes.

Moreover, the relative sizes of things like "spikes of 'input' tensors during each layer of the forward pass", "build up of activations tensors by the middle of the mini-batch", "build up of gradients tensors by the time the lowest layer has been handled by the backwards pass", etc. vary wildly as you change things like sequence-length, batch size, lora-r, which model blocks lora is applied to, etc.

While paged optimizers (from [1]) are a smart optimization to reduce the severity of VRAM bottlenecks, they in practice make it much harder to develop or test a predictor/calculation-rule for how much GPU memory will be occupied by optimizer state. Sometimes a training run with far more trainable parameters than the previous run will nevertheless have a smaller optimizer state footprint in GPU memory, or vice versa. There were even three occasions where the profiler/visualizer insisted that there was literally zero footprint for optimizer states in the GPU memory (with LoRA r of 48 and LoRA adapters being added to only one of the 3 types of targets- embedding, attention, or MLP; see Gemma 7B scenarios 10-12 in experimental data). That was especially perplexing because the "only apply LoRA to one type of target" scenarios for Gemma 2B still showed optimizer state memory usage in the profiler's visualization.

This has been a very humbling (as well as educational) experience. Discovering and grappling with these curious patterns in the training process has shown me how little real/deep understanding I have of modern ML libraries like Pytorch. Without a strong familiarity with these libraries, I haven't been able to even make testable guesses about what is going on. That was not helped by the fact that, while the libraries are open-source, the stack traces from the visualizer do less than one might expect to narrow down exactly which line in which library's code is ultimately responsible for a given tensor being allocated in VRAM. They would usually end with something unhelpful like "built-in method ¡name¿ of ¡insert non-specific class name here¿" or have a line number that just pointed to the start of a method rather than the actual line within the method's body.

VII. CONCLUSION

To sum up, the goal of this project is not generally viable. It requires far too much experimentation and tuning of calculation rules for a given model family (e.g. Gemma 1.0), and it would probably be very difficult or impossible to make these rules generalize across very different model types. However, it has brought attention to some interesting details about the variety of different ways that an ML training run can end in an ”out of (GPU) memory” error, which might possibly be useful for tuning ML libraries to be less likely to suffer such failures.

VIII. Acknowledgements

In closing, I want to thank Professor Ting Zhu for allowing me to attempt this overly ambitious project. While in hindsight my original proposal was wildly under-estimating the complexity and likely unforeseen problems that this project would involve, working on it has still been extremely educational and broadened/deepened my understanding of LLM fine-tuning.

References

[1] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. (2023). “QLoRA: Efficient Finetuning of Quantized LLMs” unpublished (Arxiv eprint 2305.14314)

[2] Gemma Team, Google DeepMind. (2024). “Gemma: Open Models Based on Gemini Research and Technology” unpublished (corporate technical report)

[3] E. Hu et al. (2021) “LoRA: Low-Rank Adaptation of Large Language Models” unpublished (Arxiv eprint 2106.09685)