A fine-grained runtime CUDA memory profiler for large models

Bo Qiang, Jasper Bucher, Odin Zhang, Yanjing Li
Paul G. Allen School of Computer Science & Engineering
Seattle, Washington, USA
{bqiang,jbutch,odinz,yanjing}@cs.washington.edu

Keywords

Dynamic Testing, Memory Profiling, Machine Learning, Graphics Processing Unit

1 Introduction

Machine learning models have the potential to transform scientific endeavors by uncovering insights from extensive datasets that were previously inaccessible. [1] According to the scaling law of machine learning [2], larger models training on larger datasets will always gain much better performance.

In large-scale projects, computational resources, especially GPU memory are typically pushed to their limits. However, out-of-memory (OOM) errors can still arise due to various factors, including GPU memory leaks, variations in floating-point precision, differences in forward-pass behavior with respect to tokenization, or dynamically growing input sizes during autoregressive inference. These errors are particularly challenging to diagnose because the exact line at which a program fails is not necessarily the primary contributor to excessive memory consumption. Throughout this paper, all subsequent references to 'OOM' will specifically pertain to GPU out-of-memory situations. As a result, identifying problematic inputs during both training and inference is often a labor-intensive process, typically involving exhaustive ablation studies or manually excluding anomalous samples one by one. These approaches are neither scalable nor effective in helping ML practitioners gain deeper insights into their model architectures.

Efficiently profiling GPU memory usage at a fine-grained level and attributing it to specific code blocks or neural network layers is therefore crucial for optimizing deep learning workflows. However, existing tools either offer CUDA-level logging that lacks detailed, user-friendly insights into memory consumption for specific Python code segments and variables, or function on PyTorch tensors, making it difficult to analyze model parameters. None of the previous works are widely used by ML developers for debugging out-of-memory errors and optimizing resource allocation in large-scale models. We introduce FineProfiler , a Python based tool that generates fine-grained GPU memory usage profiles for PyTorch-based models, offering precise tracking of memory allocation and usage across different operations. In this paper, we develop a lightweight CUDA memory profiler for tracking the consumption of PyTorch-based modules. Our approach dynamically tests the memory trace of large models and we demonstrate its applicability to architectures of varying levels of complexity. We perform experiments and analyses to demonstrate both the efficiency of our system and the user-friendly interface designed for developers.

In order to further evaluate FineProfiler , we will present a representative case study focusing on one of the leading foundation models in the field of 'AI for Science', AlphaFold 3. The occurrence of out-of-memory (OOM) errors in this model is influenced by several factors related to the hierarchical structured inputs. Notably, when processing inputs that include non-standard residue types, the dimensionality of the input vectors can increase significantly. Additionally, we have observed memory leaks when certain modules are invoked exclusively for specific input data. Our approach clarifies which architectural components can be downsized or pruned altogether, based on an assessed trade-off between memory usage and performance. This profiler helps identify that a significant amount of memory is consumed by the trunk of the network. By reducing the size of this part of the architecture, we find reduced error rates on spurious samples as well as improved parameter efficiency. We provide our open-source implementation of the profiler at github.com/YanjingLiLi/Torch_mempro.

2 Related work

To profile CUDA memory usage in machine learning models, developers have created multiple tools. Below we summarize existing software capable of tracing memory consumption.

NVIDIA Nsight Systems. [3] This is a performance analysis tool that is seamlessly integrated into the CUDA developer tool suite. Nsight Systems provides developers with time and memory consumption for every CUDA-level function call, *e.g.*, convolution network forwards. A summary of I/O, CPU, GPU memory, and GPU utilization is presented in a visualization panel. However, most ML researchers are now working with ML frameworks written in Python, where multiple neural network layers utilize the same CUDA API. Therefore, error tracing is hard for Nsight Systems, because we cannot interpret the location of bugs from a CUDA-level error message which could be raised from multiple places in the neural network. In contrast, our method solves this by building the profiler purely on Python and integrating it into a popular ML framework, Pytorch.

PyTorch's Built-in Profiler. [5] PyTorch is currently the most popular machine learning framework, according to a survey conducted by Stack Overflow[4]. A profiler operates in the background to track memory usage by calling the function `torch.profiler.profile()`. Compared to Nsight Systems, this profiler can save the memory occupation time curve as a snapshot, allowing for visualization of memory usage based on specific memory addresses. However,

ACM 2025, Seattle, WA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

establishing connections between variables that encompass multiple vectors and gradients distributed across various GPUs poses a significant challenge. In contrast to this profiler, our method tracks variable memory by segmenting the code into hierarchical graphs that represent the architecture of the neural network. This approach can facilitate direct analysis for machine learning researchers. Methodologically, our method can be viewed as a machine learning objective layer built upon the built-in profiler in PyTorch.

Third-party package pytorch_memlab. [6] This is a GitHub repository with over 1k stars, developed by Kaiyu Shi from Shanghai Jiaotong University. It provides line-by-line analysis for the pytorch `nn.Module` class, which are blocks of neural networks including their parameters and other run-time variables. This profiler measures variable memory usage by logging the data type and shape of every intermediate tensor and its gradients. However, two significant challenges arise. First, calculating the memory usage of *torch.tensor* only provides a lower bound for CUDA memory usage, as it overlooks critical components such as optimizer states and model parameters. Additionally, because this profiler is run on CPUs, it requires frequently transferring tensors back and forth between the CPU and GPU, which negatively impacts efficiency. To overcome this challenge, instead of manually calculating every tensor shape, we will keep variables on the GPU and log the memory allocation using torch-based API.

Weights and Biases (WandB). [7] WandB is a popular, lightweight logging tool for AI developers, primarily designed for the convenient tracking of training metrics. It is well-known for its ease of implementation and its user-friendly web-based API. The framework includes various GPU tracing utilities that monitor aspects such as process time spent accessing memory, as well as GPU allocation and utilization. While WandB can be useful for detecting potential memory leaks, its API does not provide the capability to diagnose specific neural modules that may be contributing to excessive memory consumption.

In conclusion, although various tools have been developed, there is currently no profiler that can be directly applied to large ML models without extensive log analysis.

3 Methodology

3.1 Overview of FineProfiler

In this section, we introduce an overview of our fine-grained CUDA memory profiler (FineProfiler). Currently, neural networks written in Python are coded as follows:

```
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(2, 1)

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        return x
```

Machine learning researchers will define more complex model architectures within the `nn.Module`. Our framework primarily consists of two components: a json file generator that log memory usage and a user-friendly visualizer that outputs HTML files. When users intend to track memory usage for the forward path of the model, they only need to wrap the forward call of the model with a Python decorator `memory_tracker`. In Python, a decorator is a function that modifies the behavior of another function or method without changing its code. It is commonly used for logging, profiling, authentication, and performance monitoring. In our context, a decorator is designed to automatically track CUDA memory usage during model execution in PyTorch. It wraps a function (typically the model's forward pass for inference mode and backward pass for train mode) and logs GPU memory allocation and deallocation at both the code line level and variable level.

```
# Apply memory tracing on model inference
@memory_tracker #wrap with the decorator
def inference_model(x):
    #moving model into GPU
    model = SimpleNN().to("cuda")
    output = model(x)
    return output

# Apply memory tracing on model training
@memory_tracker #wrap with the decorator
def train_model(output, label, optimizer):
    #moving model into GPU
    model = SimpleNN().to("cuda")
    output = model(x)
    loss = loss_func(output, label)
    loss.backward()
    optimizer.step()
```

Users simply need to apply the decorator to their model and train it iteratively, with each iteration comprising a forward and backward pass. A raw JSON file will be generated, capturing detailed CUDA memory logs for every line of code. A parser then processes this JSON file, filtering out low-level details unrelated to debugging. The refined JSON is subsequently mapped to the model's hierarchical structure, constructed using our novel model architecture builder. Ultimately, a structured JSON output file is produced, detailing memory usage per line of code and per variable. Additionally, an HTML visualization is generated, offering an intuitive representation of memory consumption across different code segments. The workflow is visualized in Figure 1.

Our main contributions are: (1) Providing an unbiased CUDA memory profiler gives full analysis from both line-of-code level and variable level, which the former is not supported by any previous works; (2) We enable memory tracking for both the forward pass and gradient backpropagation, while the latter is not supported by MemLab. (3) We present memory profiling results in both JSON and an interactive HTML page.

3.2 Design and Implementation

3.2.1 Per-variable memory track. The memory of variables is tracked by iterating through all named parameters in a PyTorch model for each line:

```
    mem_bytes = param.element_size() * param.nelement()
    print(f"{name}: {mem_bytes / 1024**2:.3f} MB")
```

The memory usage of each parameter is computed by multiplying its **element size** (the number of bytes each element occupies) with the **total number of elements** (obtained using `nelement()`).

This approach provides insights into the lower-bound memory consumption of each tensor involved in the current line-of-code, which is crucial for optimizing memory usage, particularly when training large models on GPUs.

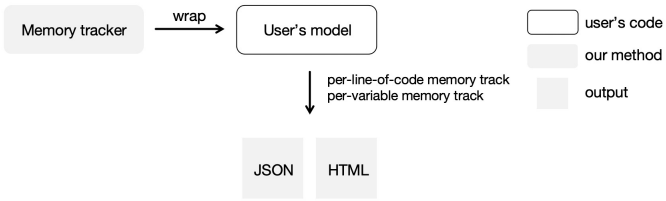

Figure 1: Proposed Workflow of FineProfiler: Gray blocks represent components executed or generated by our method, while the white block represents the user's model. Users can wrap the memory tracker directly on their customized model. During execution, the memory tracker monitors memory usage for each line of code and tensor operation, summarizing the results in a JSON file and an HTML page as output.

3.2.2 *Per-line-of-code memory track.*

Forward Pytorch decorator. The decorator used as a memory tracker is the core component of FineProfiler , represented in Figure 1. It is responsible for tracking CUDA memory consumption. Unlike existing tools such as PyTorch-MemLab, which focuses only on variable usage, or Nsight System only on overall usage, the forward pytorch decorator logs memory allocation and deallocation for every line of python code. This dual-level approach allows users to diagnose CUDA OOM errors more precisely by distinguishing whether memory spikes are caused by large input data, memory-intensive operations, or inefficient tensor management.

The Python decorator wraps the model's forward pass and captures memory usage while injecting dynamic hooks into the execution flow. The decorator intercepts function calls, allowing it to track memory usage without modifying the original code. Dynamic hooks are injected both at the Python execution level and at the PyTorch tensor level. At the Python execution level, our decorator hooks into each executed line of code using `sys.settrace()`, enabling it to track per-line memory usage. At the PyTorch tensor level, it hooks into PyTorch's autograd system by leveraging `sys.settrace()`, which monitors tensor operations within the neural network layers.

To capture memory usage at the Python execution level, our decorator dynamically injects hooks using `sys.settrace()`, allowing it to intercept execution at each line and extract file names and line numbers from `frame.f_code.co_filename` and `frame.f_lineno`. GPU memory usage is recorded before and after execution using `torch.cuda.memory_allocated()` and `torch.cuda. memory_reserved()`. This enables FineProfiler to log per-line memory changes, detect sudden spikes, and identify inefficient memory usage patterns that may indicate memory leaks. These delta memory usages are linked with specific lines of code and variables. By capturing execution at the Python level, users can pinpoint the exact line of code responsible for excessive memory consumption.

For variable-level tracking, the system captures memory allocation and deallocation events associated with tensors. Each tensor is uniquely identified using id(tensor), and its memory footprint is computed through attributes such as `tensor.element_size()` and `tensor.nelement()`. To monitor tensor transformations and track memory usage across different operations, the system employs PyTorch hooks, including `register_forward_hook`, to attribute memory usage to specific layers in a neural network.

The decorator plays a key role in this process by automatically wrapping user-defined models and injecting the necessary instrumentation code. Instead of defining new custom hooks, the decorator captures function calls, performs additional tracking logic, and then forwards execution to the original module. This approach allows the system to dynamically attach hooks at runtime without modifying the underlying model code. The collected data is stored in structured logs, enabling an in-depth analysis of how each variable contributes to overall memory consumption.

Backward Pytorch decorator. In machine learning, neural networks are usually trained by a forward pass to get the loss and then backpropagate the gradient through model layers to update all parameters. Therefore, tracking the CUDA memory usage for backpropagation is also important to detect errors in code.

In our implementation, we monitor CUDA memory usage by assessing memory consumption before and after executing the module, paralleling the method used during the forward pass. However, unlike the forward decorator, it is impossible to analyze the backward pass line by line due to its operation on the underlying computational graph instead of explicitly defining all execution. To overcome this, we employ the `register_full_backward_hook` for each submodule of the complete network, allowing us to capture and log memory usage across all modules. This approach enables us to generate comprehensive logs for each submodule and neural layer, facilitating an in-depth analysis of memory consumption during the backward pass.

3.2.3 *Outputs processing.*

Log parser. Both the forward decorator and the backward decorator run recursively to capture the fine-grained details of the neural networks. This implies that for a `nn.Linear()` layer, the logging operations are decomposed into weight matrix multiplication and bias calculation, resulting in redundancy for debugging purposes.

To address this issue, we implement a log parser that processes the JSON file by removing all low-level call logs. A Python script automatically identifies directory paths containing third-party PyTorch-level packages and removes any corresponding logs from the raw JSON output while preserving the correct hierarchical structure. After this pruning process, the refined JSON file is passed to the builder for further processing.

Model architecture builder. The key challenge in using other CUDA memory tracking tools is how to interpret the logs that could be useful for real-life machine learning debugging. Nowadays, neural networks are getting bigger and bigger, both on the scale of the number of parameters and the complexity of these designed architectures. Rather than analyzing code line by line, Python programs are structured as a hierarchical tree within the minds of machine learning engineers.

In order to make the logs interpretable, we implement a forward and backward decorator, as well as a model architecture builder that parses any PyTorch-based code into a tree-based structure. This structure recursively collects all computations in the forward

pass of every `nn.Module`. Each node in the tree represents a specific computation, and we enhance this representation by adding the corresponding file name and line numbers as metadata for each child node. This comprehensive logging mechanism facilitates better debugging and understanding of the model's architecture and its execution flow during training and inference.

3.2.4 Visualization. The visualization component aims to offer an efficient, fine-grained, and user-readable solution for deep learning users working with large-scale models. By integrating detailed memory tracking at both the code and variable levels, detailed output generation, and multi-GPU awareness, this tool will address critical challenges in memory optimization and help developers more effectively diagnose and mitigate out-of-memory errors. Our tool will generate structured output in JSON format, capturing memory usage statistics for each code block and variable. To be specific, JSON output takes the hierarchical form of the neural network where each key represents a chunk of code that is sequentially executed during the forward pass. In the JSON output for the chunk of code, memory usage, and variables are logged. This JSON output will enable easy integration into existing debugging pipelines, allowing developers to systematically analyze memory trends and identify bottlenecks.

Additionally, an HTML page will be generated where each variable / layer is represented as a button. When the user clicks on a button, it will display the memory usage information for each line that involves the selected item.

4 Experiment

In the experimental evaluation section, we first conducted a fundamental functional test on a toy model to determine whether our approach could deliver line-by-line CUDA memory logging and organize this data module by module. Next, we performed experiments to assess the performance of our system, FineProfiler , and compared it with other well-known tools. Finally, we applied our method to real-world debugging scenarios in AlphaFold 3.

4.1 Functional test

4.1.1 Line-Specific Detection. We first examine the profiler's capacity for line-specific memory attribution. An MLP with multiple layers is constructed, and its hidden dimensions are progressively increased. During training, the profiler logs memory allocation, linking each allocation to the exact line of code responsible (e.g., weight initialization or forward-pass operations). Through this experiment, we verify whether the profiler pinpoints the specific lines causing memory surges, enabling developers to optimize particular code segments more efficiently.

```
Refined Line-specific tracing
{
    "file": "model/MLP.py",
    "line": 25,
    "function": "forward",
    "delta_allocated_bytes": 4000256,
    "delta_reserved_bytes": 0,
    "delta_allocated_gb": 0.0037255287170410156,
    "delta_reserved_gb": 0.0
}
...
```

4.1.2 Module-Specific Detection. Next, we investigate module-specific memory tracking using the same MLP setup. Instead of attributing allocations to individual lines, the profiler aggregates memory usage at the module level, providing a consolidated view of each layer's resource consumption. This module-level analysis highlights which parts of the network (e.g., certain dense layers) demand the most GPU memory. By identifying these high-consumption modules, developers can strategically apply optimization techniques (e.g., reduced hidden dimension size or selective gradient checkpointing) where they will have the greatest impact.

This module-level logging is enabled not only for forward calls of the neural network but also for tracking CUDA memory consumption during the backpropagation of gradients, which is essential for parameter optimization. To our knowledge, FineProfiler is the first tool that offers fine-grained logging for model training.

```
{
"name": "Main_model",
"children": [
    {
        "name": "layer1",
        "children": [
            {
                "name": "linear",
                "children": [],
                "delta_allocated_gb": 0.01,
                "delta_reserved_gb": 0.02
            },
            {
                "name": "relu",
                "children": [],
                "delta_allocated_gb": 0.01,
                "delta_reserved_gb": 0.0
            }
        ]
    },
    {
        "name": "layer2",
        "children": [
            {
                "name": "linear",
                "children": [],
                "delta_allocated_gb": 0.01,
                "delta_reserved_gb": 0.02
            },
            {
                "name": "relu",
                "children": [],
                "delta_allocated_gb": 0.0,
                "delta_reserved_gb": 0.0
            }
        ]
    },
    {
        "name": "fc",
        "children": [],
```

```
            "delta_allocated_gb": 0.0,
            "delta_reserved_gb": 0.0
        }
    ]
}
```


4.1.3 Actionable Suggestions. The practical advantage of our profiler is its ability to generate actionable suggestions. Specifically, we leverage the profiler logs to compute the peak GPU memory usage during the entire backward pass of the network, then convert that value in relation to the current batch size. By dividing the total available GPU memory by the per-sample memory footprint, we obtain an **estimated maximum feasible batch size**.

```
>> python script/mlp_experiment.py
Maximum batch size: 207240
>> python script/transformer_experiment.py
Maximum batch size: 467
```

As shown above, running our profiler on a simple MLP architecture yields a maximum batch size of **207,240**, while a more complex Transformer model caps at a feasible batch size of **467**. This estimation addresses a common challenge in deep learning development: practitioners typically discover the maximum batch size through a trial-and-error process (e.g., testing batch sizes 2, 4, 8, etc.) until an out-of-memory (OOM) error arises. Our approach, however, streamlines the process by systematically analyzing the memory profiling logs to pinpoint exactly how much memory is consumed at peak allocation. Not only does this save considerable time, but it also ensures that GPU resources are utilized as efficiently as possible—maximizing batch sizes without risking OOM errors.

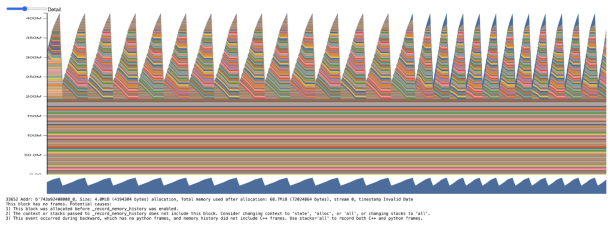

Figure 2: UI of Pytorch built-in profiler

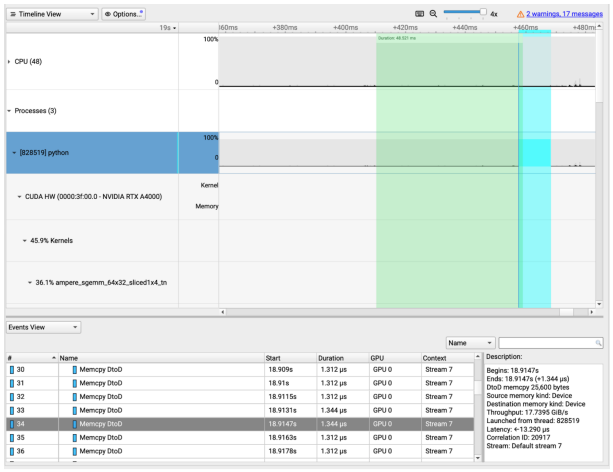

Figure 3: UI of NVIDIA Nsight system

Figure 4: UI of our FineProfiler

4.2 UI Analysis

In this section, we provide an analysis of the current popular CUDA memory tools and discuss the advantages of our visualization webpage.

As shown in Figure 3, the built-in PyTorch profiler provides a plot of CUDA memory usage over time. Memory allocations are logged and color-coded according to the corresponding CUDA calls. This feature is particularly useful for identifying memory leaks and investigating which iterations cause abnormal memory behavior. However, this visualization is hard to interpret for Python debugging and model architecture analysis, especially for large models or huge batch sizes that are non-trivial to fit into a single GPU for one forward pass. Nsight system provides more fine-grained low-level logging as in Figure 3. Besides CUDA memory logging, it also gives information about CPU usage and CPU-GPU information exchanges.

A visualization of the demo is presented in Figure 4. In addition to logging basic information about peak memory usage during the entire training iteration and inference pipeline, the webpage features a detailed line-by-line delta memory visualization. To identify abnormal memory behavior or bottlenecks, users can first examine peak CUDA memory by reviewing the JSON output from our memory summary, which is organized by module. They can then click on the corresponding code lines on the webpage to examine the tensor memory size and additional memory allocations at each computational step.

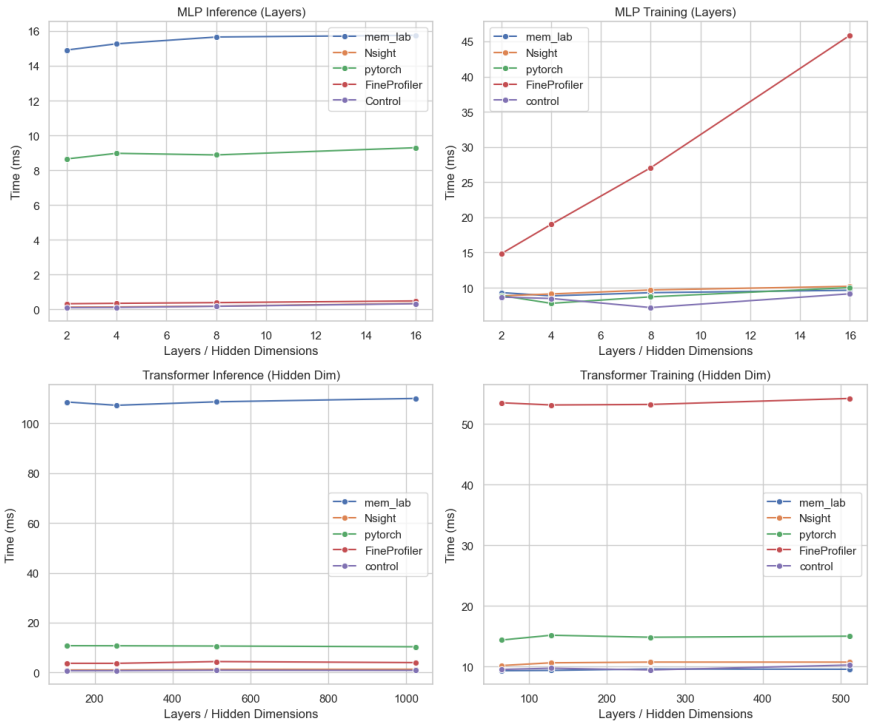

Figure 5: Runtime of different cuda profiler for MLP / Transformer experiment

4.3 Experiment on runtime

While all profilers can offer valuable insights to aid ML developers in debugging their models, these methods may also introduce additional runtime overhead that could adversely affect their practical applications. Therefore, in this section, we benchmark the running time for well-known CUDA memory profilers. Experiments are conducted on MLPs of different numbers of layers and vanilla transformers of different sizes of hidden dimensions. Runtimes are evaluated for both inference and training processes. During inference, the automatic gradient calculation is disabled for PyTorch models. For each training iteration, randomly generated data is input for the forward pass to compute the loss, followed by one step of backpropagation and optimization. We run 100 iterations for all MLP experiments and 10 iterations for all Transformer experiments. The results are illustrated in Figure. 5. 'control' stands for the blank experiments without any profiling.

In inference mode, both MemLab and the built-in PyTorch profiler introduce significantly more overhead in running time compared to the Nsight Systems and our method. However, this overhead does not scale with the number of layers or the size of the hidden dimensions. For larger models, the additional running time incurred by the built-in PyTorch profiler and the blank experiment remains consistent. We conclude that this overhead is primarily due to the process of saving snapshots at the beginning and end of the program rather than from injecting additional logging into the model itself. However, MemLab still significantly slows down model inference. On the other hand, our method incurs only a marginal additional runtime compared to the native NVIDIA profiler, demonstrating its efficiency.

For model training, it is clear that our approach, denoted as FineProfiler , may introduce some latency in the backpropagation gradient computation due to the recursive logging mechanism. The upper-right figure illustrates that the running time scales linearly with the number of layers in the neural network. However, as demonstrated in our Transformer experiments, the running time of our method does not exhibit scaling with the size of the hidden dimensions. These findings indicate that while our method may hinder training efficiency—especially when developers partition neural networks into multiple nested modules—it remains unique in its capability to offer fine-grained logging during training. This level of detailed logging is crucial for optimizing model performance and debugging, setting our method apart from others in the field.

4.4 Peak memory detection

It is essential for the profiler to accurately identify memory bottlenecks in the code. Consequently, we analyze the output logs from Memlab and the built-in profiler of PyTorch, comparing their peak memory usage with that of our method. We run all methods on a 12-layer transformer encoder-decoder network and the results are shown in Figure. 6. It is obvious that FineProfiler and the built-in profiler of PyTorch give similar results while Memlab underestimate the memory consumption of the transformer model. This is because Memlab only tracks the memory for training through

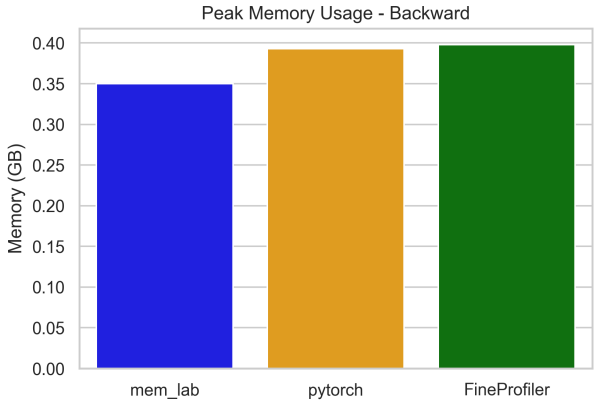

Figure 6: Peak memory profiled by different methods for a 12-layer transformer

tensor sizes and tensor gradients. The gradient stored in the parameters and optimizers takes around 0.05GB in the experiment which only FineProfiler and the built-in profiler of PyTorch are able to detect.

4.5 Application to development of the AlphaFold3 architecture

In this evaluation, we aim to demonstrate that our profiler can be applied to large, complex models, and serve to aid developers on complex real-world tasks. In particular, we are interested in an machine learning architecture design problem related to AlphaFold3. AlphaFold3 is a 300M parameter model trained to predict the structure of a protein given its amino acid sequence. The architecture of AF3 is complex, but can be viewed as containing two major components: (i) the *trunk* which processes the amino acid sequence, and (ii) the *diffusion module* which generates the full structure given the outputs of the trunk. The trunk serves to steer the final generation, similar to how a prompt to a large language model guides how it generates the answer.

For our application, we want to use a model that has already learned some things and apply it to a new job: creating proteins from scratch, without needing any initial sequence input or guidance. Normally, this involves the trunk, which helps in generating initial guesses. However, for this task, the trunk isn't needed, and it's actually holding back the model's performance because it's taking up resources that could be used more effectively.

We aim to use FineProfiler to understand the effect of removing the trunk. And to use our program to understand how relinquished resources can be reallocated to accelerate training and thereby boost performance.

Using the FineProfiler , we can provide a rough measure of architectural complexity. Upon removal of the trunk component, we found a nearly ~ 4-fold decrease in the number of submodules called on the forward pass (3393 in the base model versus 900 in the variant without the trunk–the 'A14' model). This significant reduction in the architectural complexity upon removing the trunk suggests the model would be substantially simpler and therefore faster. Interestingly, however, we found that our initial implementation did not significantly accelerate training.

We investigate in Figure 7 the causes of this effect by profiling the memory consumption of both models during the backward-pass. We see that, where a significant proportion of memory was allocated to the trunk (top of Figure 7), there is now still a large activation outside of the diffusion module.

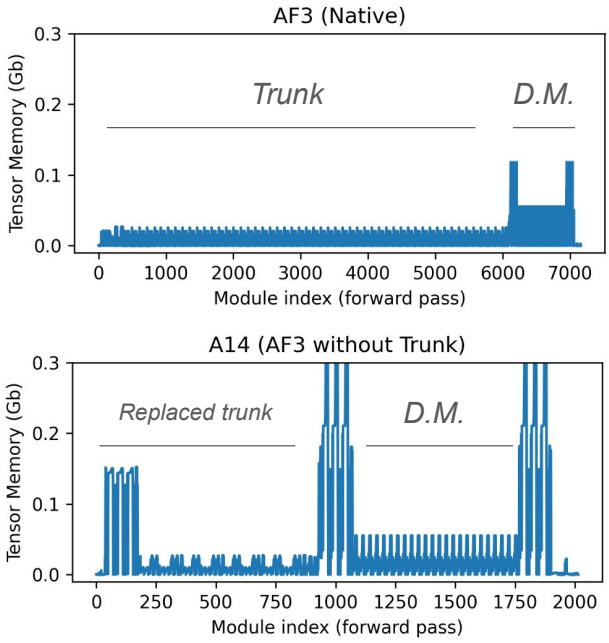

Figure 7: Memory profiles for AF3 architecture (with trunk, top) and 'A14' (AF3 without the trunk, bottom).

To investigate the memory effects of the replaced trunk modules, we carry out an ablation of the memory consumption associated with the replaced trunk components in Figure 8. We see that during the backward pass, the memory consumption of the A14 (initial implementation) is comparable to that of the original AF3, providing a potential explanation as to why the training time is approximately the same after removing the trunk. Finally, we see that the forward pass is largely consumed by the trunk components, suggesting an alternative module is likely most appropriate for the A14 variant of AF3.

We found after training the new variant with the removed additional components, that training time was reduced by 35%. This efficiency increase is attributable to the decreased memory allocation during the backward pass, and was previously undetectable using only the forward pass (see figure 8). Other conventional methods of memory tracing must rely on the forward pass only and therefore could not be used to guide development in the same manner.

Overall, this shows our profiler can be used to aid in real-world problems. We note there are additional effects other than the ones described here which likely increase memory consumption of the ablated model associated with.

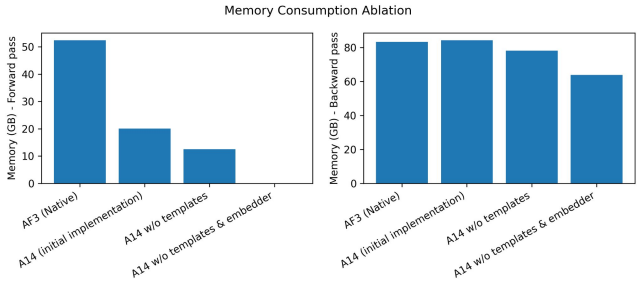

Figure 8: Memory consumption by model variants.

5 Discussion

The increasing emphasis on inference-time scalability has highlighted the importance of ensuring model robustness not only during training but also throughout the inference process. In this work, we introduced FineProfiler, a lightweight, Python-based CUDA memory profiler for large-scale PyTorch models. Unlike existing profilers that operate at the CUDA or framework level, FineProfiler offers fine-grained insights at both the code-line and variable level using Python decorators and dynamic tracing mechanisms. Our tool not only facilitates accurate memory attribution during both the forward and backward passes, but also produces structured, interpretable outputs in both JSON and HTML formats. Through comprehensive experiments and real-world applications such as AlphaFold3, we demonstrated that FineProfiler can effectively detect memory bottlenecks, guide architectural decisions, and suggest optimization strategies—ultimately improving model reliability and resource efficiency. On this task, we showed that our framework offers insights by monitoring memory consumption of the backward pass which are not available from the existing literature. By offering an intuitive debugging interface and maintaining minimal runtime overhead, FineProfiler addresses a critical gap in the deep learning development ecosystem and is a promising tool for optimizing deep learning workflows at scale.

Acknowledgments

We thank the members of the Institute for Protein Design; Rohith Krishna, Nathaniel Corley and Woody Ahern for their insightful thoughts during the planning of this project.

References

[1] Michael I Jordan and Tom M Mitchell. 2015. Machine learning: Trends, perspectives, and prospects. *Science* 349, 6245 (2015), 255–260.
[2] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).
[3] NVIDIA Corporation. [n. d.]. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems. Accessed: 2025-02-05.
[4] Stack Overflow. 2024. Stack Overflow Developer Survey 2024: Technology. https://survey.stackoverflow.co/2024/technology Accessed: 2024-10-02.
[5] PyTorch. 2023. PyTorch Profiler Recipe. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html. Accessed: 2025-02-05.
[6] Stonesjtu. 2025. pytorch_memlab. https://github.com/Stonesjtu/pytorch_memlab Accessed: [Insert Date Here].
[7] Weights and Biases. 2025. Weights and Biases. https://wandb.ai/site/. Accessed: 2023-10-05.

A Methods

(Appendix)

Received 24 January 2024