Final Project

CS265, Edwin Tembo, Spring 2024

Abstract:

Activation Checkpointing is a widely used method to address GPU memory underutilization or to increase the batch size during training, given a limited GPU memory budget. This process works by discarding tensors from GPU memory after their last use in the forward pass and recomputing or swapping them back into GPU memory when they are needed for Gradient computation in the backward pass. While swapping can result in much lower peak allocated memory throughout a training loop, it also comes with a swap time overhead that lengthens the training time. Recomputation of tensors can also be used to either with swapping or as a standalone optimization. In this project swapping was used for profiling and selecting recomputation candidates. However, for the purposes of this project, execution of the optimized graph only used the recomputation method.

This discussion will explain the process of creating a graph-based profiler using the Pytorch FX library, then running an activation checkpointing algorithm based on the MuTwo Algorithm [1], by recomputing and executing an optimized torch FX graph.

Existing Solutions:

Some exiting solutions include *torch.utils.checkpoint* from Pytorch, Amazon SageMaker's distributed parallel library, FairScale, DeepSpeed from Microsoft, among others. This project takes a more in-depth approach by directly using a Pytorch FX graph to accomplish activation checkpointing, based on the mu2 paper [1]. This approach selects tensors based on their inactive time (the time between the last forward use and first backward use of the tensor) and the size of the memory allocated to each tensor on GPU memory, and the approximate time it takes to recompute the tensor. This will be discussed in more detail in the methodology section. See Appendix A for pseudocode.

Methodology:

Graph Profiling: During the graph profiling stage, a Torch FX graph was created based on Torch Benchmark [2] models. Data was collected for each node of the graph, including node_execution_time, allocated_gpu_mem, swap_in_time,swap_out_time, tensor_size, last_fw_access_node, first_bw_access_node and tensor swap_times. See Appendix A for the Graph Profiling Algorithm pseudocode. Garbage Collection was also added to remove any obsolete objects after the execution of each node. This stage also categorized appropriate nodes into Activations, or intermediate nodes that would be candidate for recomputation in the activation checkpointing algorithm.

Activation Checkpointing: After statistics were gathered, intermediate nodes were selected based on the inactive time and the recomputation ratio, which is the tensor memory size divided by its approximate time required for recomputation (See Appendix A). Because this value was compared to inactive time, which runs on a different scale than the recomputation ratio. The comparison was based on standardized values using an SciKitLearn standard scaler. If the standardized inactive time was greater than the recomputation ratio, then the tensor was deemed a good candidate for recomputation, otherwise it was not included in the recomputation list.

Graph Recomputation and Profiling: Once the tensors were selected, Torch FX functions were used to create new subgraphs for each selected node. The nodes of each new subgraphs were assigned new names and placed just before the original node's first backward use. The original node was then removed from the first backward access node's inputs after the new node is added as a new input. This means that when the new graph runs, the original node will no longer have a use in backpropagation and will be deleted after its last use in the forward pass, freeing up GPU memory to allow the possibility of training with larger batch sizes. A file containing the statistics of the highest batch size successfully profiled using swaps can be used for graph rewriting.

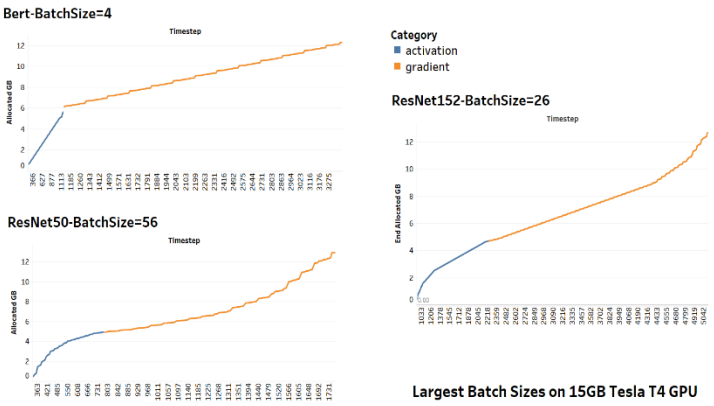

Figure 1. NVIDIA T4 15GB

Evaluation: Preliminary evaluations were conducted on a Google Colab machine with a NVIDIA T4 GPU, running on 15.0 GB of GPU memory and 51 GPU CPU RAM. It became apparent that the garbage collection mechanism was not performing the required cleanup. Preliminary results are shown in the results section, as a point of reference. Once the garbage collection issue was resolved, experiments were moved to a Google Colab machine with a NIVIDIA A100 GPU, with 40GB of GPU Memory and 83.5GB CPU RAM. The evaluation was based on BERT and ResNet50 Models from the Torch Benchmark repository [2].

Preliminary Results: Using a T4 GPU, without garbage collection, the maximum possible batch size for the BERT model was only 4 and ResNet50 was 56. These results are included to showcase the importance of relieving GPU memory of tensors and other objects that could accumulate, causing peak memory to rise very quickly, resulting in an out of memory error. In Figure 1., the built-in garbage collector was not running as anticipated, resulting in extremely high memory usage at lower batch sizes. The evaluation results going forward will be based on the pseudocode provided in Appendix A. The code included utilities to drop every object on its last use to simulate the garbage collection process in a regular Pytorch training loop.

A100 GPU Results:

ResNet50: This model showed very high gains in memory allocation when swapping and running the recomputed graph. The maximum possible batch size before any optimizations was 480. Although the actual maximum possible batch size was not explored, swapping and recomputation were able to achieve higher batch sizes (484, 532) than 480 at just under half and just over half the maximum GPU capacity respectively. See Figure 5. for more details.

Bert: The structure of Bert is such that there is a burst of very large tensors (2GB+) created just before and just after the beginning of backpropagation. Because of this, the implementation in this paper was unable to get any higher batch sizes than the 48 used without any optimizations. See Figure 4 for more details for memory and batch size statistics, and Figure 2. for tensor size data.

Execution Time Tradeoff: As anticipated, swapping takes almost up to 2.5+ times the average execution time per node than recomputation and running with no optimization. However, it also shows the most reduction in peak memory allocation for the ResNet50 model. See Figure 2. for more details.

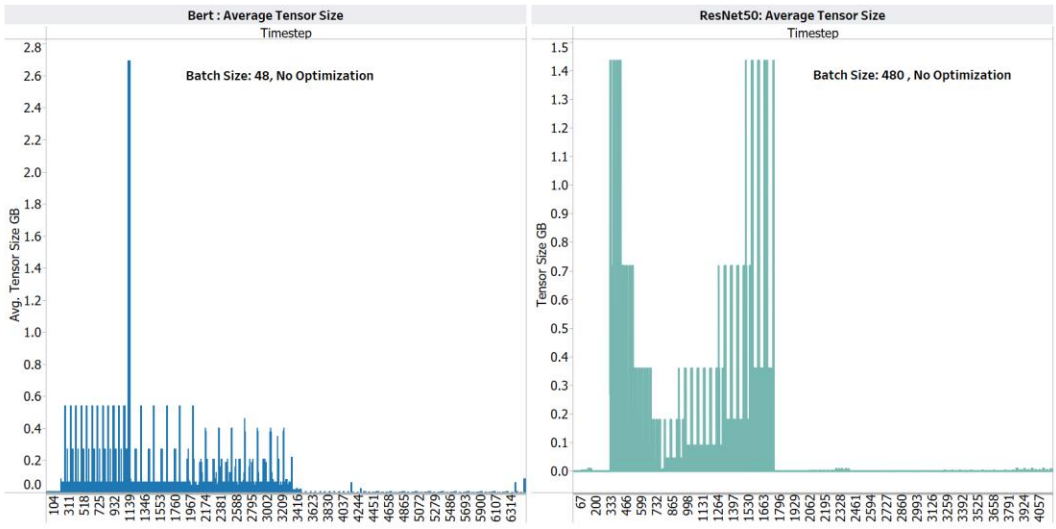

Figure 2. Tensor Sizes

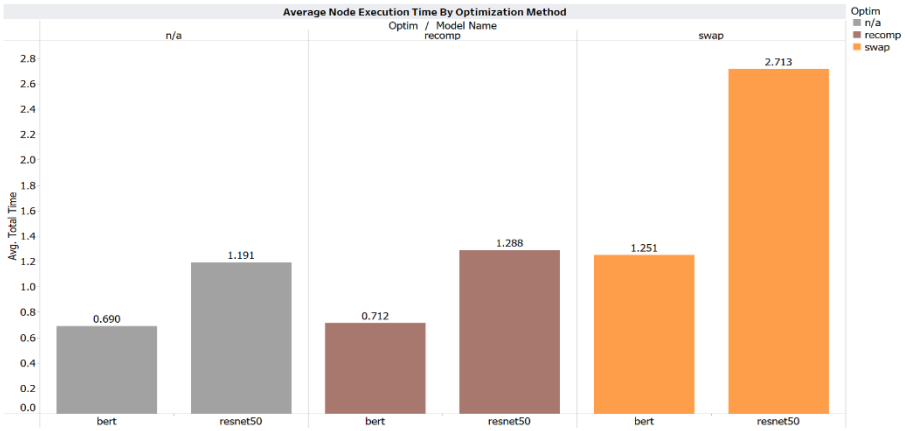

Figure 3. Execution Time Tradeoff

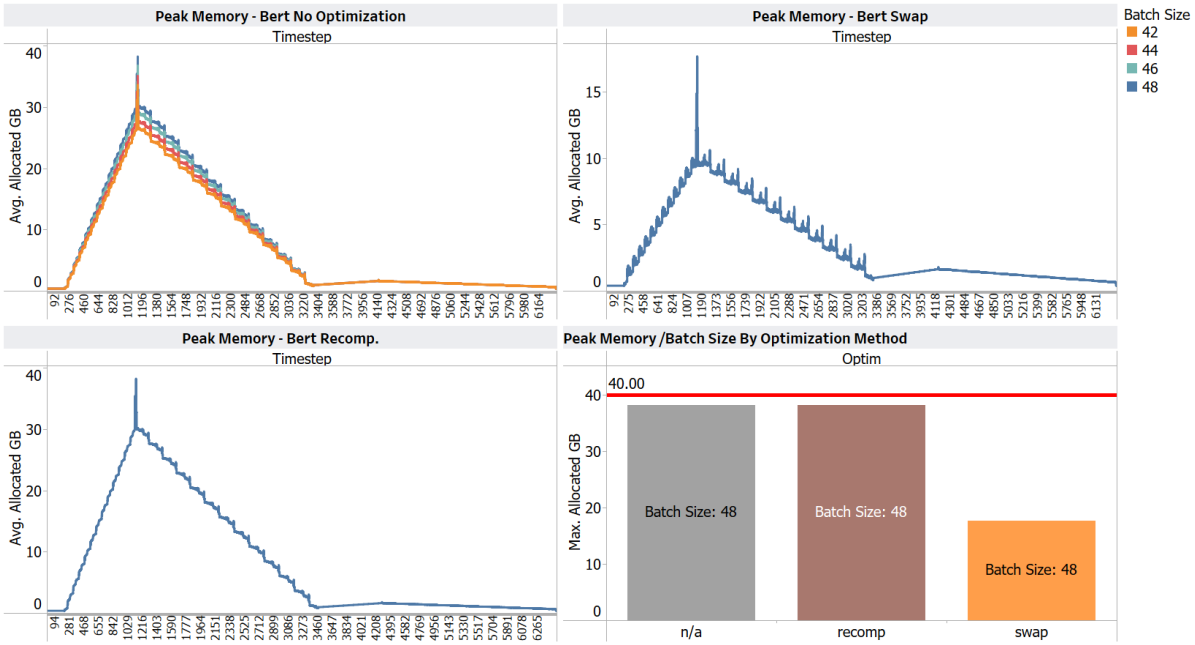

Figure 4: Bert Results

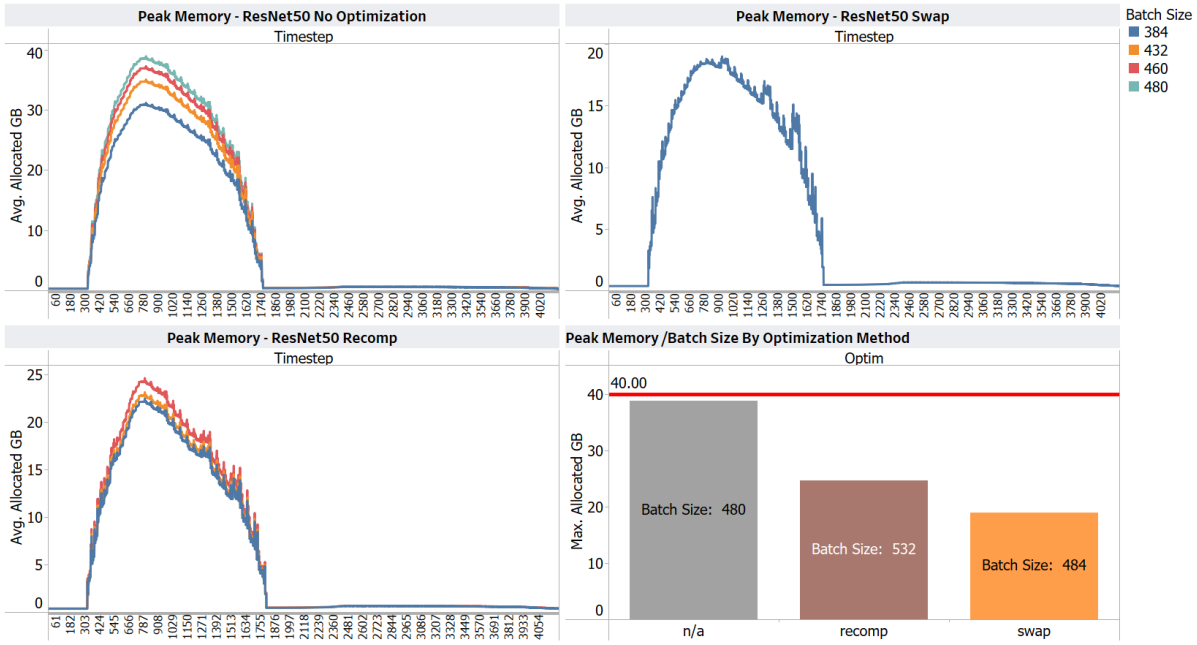

Figure 5: ResNet50 Results

***Note**: For the ResNet50 model with batch size of 480 and 484, with a swap or no optimization, all the statistics are collected and saved. However, an OOM error may occur at the very end during object deletion/garbage collection. These results have been included here. A batch size of 472 may be used to get past this error if necessary. The output of the swap profiling at batch size 484 was used for the graph rewrite represented in the charts above.*

Conclusion:

This project shows how activation checkpointing can be a useful tool for the reduction of peak GPU memory. However, it also depends on the model's architecture as sone models my benefit more than others. The algorithm used for candidate selection also has an impact on the outcome and may have to be formulated differently for different models.

References:

[1] μ-TWO: 3× Faster Multi-Model Training with Orchestration and Memory Optimization Sanket Purandare, Abdul Wasay, Stratos Idreos

[2] Torch Benchmark, https://github.com/pytorch/benchmark

APPENDIX A

Graph Profiler:

Input ->Graph_Module, run_swap, run_recomp:

```
For graph in Graph_Module:
   For node in Graph:
      last_fw_access_nodes, first_bw_access_nodes,last_bw_access_nodes,node_types= ComputeNodeAtrrs(node)
   For node in graph:
      allocated_gpu_memory, node_execution_time = GatherStatistics(Node)

      IF run_swap
        IF is_forward_pass:
           swap_out_time = GatherSwapStatistics(node_tensor)
      ELSE:
         swap_time_in   = GatherSwapStatistics(tensor)

      IF is_forward_pass:
        FOR l_node in node.last_fw_access_nodes
          IF count(l_node.first_bw_access_node) == 0 :
            last_use_nodes.append( l_node)

      IF is_backward_pass:
        FOR l_node in node.last_fw_access_nodes:
           last_use_nodes.append( l_node)

    GarbageCleaning(node, last_use_nodes)
    SaveStatistics( node_execution_time, allocated_gpu_mem, swap_in_time,swap_out_time, tensor_size)

 SaveAllStats()
SaveAllStatsAndAttrs()
```

Activation Checkpointing:

Input -> TensorSwapStatsFile:

```
recomp_nodes = []
intermediate_nodes = node_types == 'Activation'

For node in intermediate_nodes:
  normRecompRatio ComputeStandardizedRecompRatio( tensor_size, node_execution_time)
  inactiveTime = CUMULATIVE_SUM(node.last_bw_access_node +1 , node.first_bw_access_node - 1)
 normInactiveTime = ComputeNornalizedInactiveTime(inactive_time)

 IF normInactiveTime >    normRecompRatio:
   recomp_nodes.append(node)
```

Graph Recomputation:

Input -> Graph_Module, recomp_nodes:

```
 new_graph = Graph_module.graph
recomp_graph_nodes = []
For node in recomp_nodes:
   required_inputs = nodes.inputs
  recomp_graph_nodes.append({node: reversed(required_inputs)

For node in reversed(recomp_graph_nodes):
    new_node = new_graph.create_new_node(node, node.required_inputs)
   new_graph.place_before( node.first_bw_access, new_node)
   new_graph.add_input(node.first_bw_access_node, new_node)
  new_graph.remove(node.first_bw_access_node, node)
```