Briefing: Reducing Activation Recomputation in Large Transformer Models

Executive Summary

Training large-scale transformer models is a primary computational challenge in modern AI, fundamentally constrained by the memory required to store activations for backpropagation. The standard solution, known as activation recomputation or gradient checkpointing, circumvents memory limits by re-computing activations instead of storing them, but at a significant cost of a 30-40% increase in execution time.

This document synthesizes findings from a paper by NVIDIA researchers that introduces two novel, simple techniques—**Sequence Parallelism** and **Selective Activation Recomputation**—that collectively and almost entirely eliminate this computational overhead. Sequence Parallelism extends tensor parallelism to distribute previously replicated activation memory across devices. Selective Activation Recomputation intelligently recomputes only the most memory-intensive yet computationally inexpensive parts of a transformer layer.

The combined impact of these methods is substantial:

• **5x Reduction in Activation Memory:** The techniques drastically reduce the memory footprint required for activations.

• **Over 90% Overhead Recovery:** The methods recover more than 90% of the execution time overhead introduced by traditional, full activation recomputation.

• **~30% Throughput Increase:** End-to-end training throughput is increased by 29% to 32% across models ranging from 22 billion to one trillion parameters. When training a 530B parameter model on 2240 NVIDIA A100 GPUs, this approach achieves a Model Flops Utilization (MFU) of 54.2%, representing a 29% speedup compared to the 42.1% MFU achieved using standard recomputation. The techniques are slated for implementation in NVIDIA's Megatron-LM and NeMo-Megatron frameworks.

1. The Activation Memory Bottleneck

As transformer models scale towards trillions of parameters, distributing the model's parameters, optimizer state, and activations across multiple GPU devices becomes necessary. While model parallelism techniques like tensor and pipeline parallelism help distribute parameters, the memory required to store activations remains a critical bottleneck. Activations are intermediate tensors created during the forward pass that are essential for calculating gradients during the backward pass.

For large models, the required activation memory significantly exceeds the capacity of modern accelerators. For example, baseline configurations for models ranging from 22 billion to 1 trillion parameters all require more than the 80GB of memory available on an NVIDIA A100 GPU.

The conventional method to manage this memory pressure is **full activation recomputation**, where most activations are discarded during the forward pass and recomputed as needed during the backward pass. While effective at saving memory, this introduces a severe performance penalty, with observed execution time overheads of 30-40%.

2. Core Optimization Techniques

The paper presents two primary techniques to alleviate activation memory pressure, thereby reducing the need for costly recomputation.

2.1. Sequence Parallelism

Standard tensor parallelism, as implemented in Megatron-LM, parallelizes the computationally intensive attention and MLP blocks of a transformer layer. However, it leaves other components—specifically layer-norms and dropouts—replicated across all GPUs in the tensor parallel group. These replicated operations do not require significant compute but consume a considerable amount of activation memory.

Sequence Parallelism addresses this limitation by partitioning these non-tensor-parallel regions along the sequence dimension.
• **Mechanism:** It introduces communication collectives (`all-gather` in the forward pass, `reduce-scatter` in the backward pass, and vice versa) that act as converters between sequence-parallel and tensor-parallel regions.

• **Efficiency:** Crucially, these new communications are combined with existing tensor parallelism operators, ensuring that sequence parallelism does not introduce any additional communication overhead. The total communication bandwidth used remains the same as in standard tensor parallelism.

• **Impact:** By enabling the distribution of all activations within a transformer layer, sequence parallelism reduces the required memory by a factor of the tensor parallel size (`t`). This is a significant improvement over standard tensor parallelism, where a substantial portion of activation memory (`10sbh` in the derived formula) remains un-parallelized.

2.2. Selective Activation Recomputation

Rather than applying recomputation to entire transformer layers, this technique selectively targets parts of each layer that offer the best trade-off between memory savings and recomputation cost.

• **Core Insight:** The operations within the self-attention block responsible for the attention matrix computation (e.g., `QKT` matrix multiply, softmax) have large input sizes and thus generate large activations, but they are not computationally expensive (low FLOPs per input element).

• **Targeted Operations:** The technique specifically checkpoints and recomputes the activations related to QKT matrix multiplication, softmax, softmax dropout, and attention over values. These activations correspond to the 5as/h term in the paper's memory formula.

• **Quantified Benefit:**
  ◦ For a GPT-3 scale model (175B), these targeted activations account for **70%** of the total activation memory, but recomputing them adds only **2.7%** in FLOPs overhead.
  ◦ For an MT-NLG scale model (530B), they account for **65%** of activation memory with only **1.6%** FLOPs overhead.
By only recomputing these specific, high-impact activations, this method drastically reduces the computational penalty while still achieving substantial memory savings.

3. Performance and Impact Analysis
The combination of sequence parallelism and selective activation recomputation yields significant improvements in both memory efficiency and execution speed.

3.1. Memory Reduction

The proposed techniques deliver substantial reductions in activation memory requirements. When used together, they achieve a **5x reduction** in memory compared to a baseline using only tensor parallelism, bringing the memory footprint to just ~20% of the baseline. This is only slightly more than the ~10% achieved by full recomputation but is accomplished with a fraction of the computational cost.

Technique	Activation Memory Per Transformer Layer (in bytes)
No Parallelism	sbh (34 + 5as/h)
Tensor Parallel (Baseline)	sbh (10 + 24/t + 5as/ht)
Tensor + Sequence Parallel	sbh/t (34 + 5as/ht)
Tensor Parallel + Selective Recompute	sbh (10 + 24/t)
Tensor + Sequence Parallel + Selective Recompute (Present Work)	**sbh (34/t)**
Full Activation Recomputation	sbh (2)

Variable Definitions: s=sequence length, b=microbatch size, h=hidden size, a=attention heads, t=tensor parallel size.

3.2. Execution Speed and Overhead

The primary benefit of the proposed methods is the dramatic reduction in the time penalty associated with activation recomputation.

• **Overhead Reduction:** The techniques combined recover over **90%** of the compute overhead from full activation recomputation.
  ◦ For a 22B model, the combined forward/backward pass overhead drops from **39%** with full recomputation to just **4%**.

◦ For 530B and 1T parameter models, the overhead is reduced to a mere **2%**, compared to 36% for full recomputation.

3.3. End-to-End Throughput

These component-level improvements translate directly into faster end-to-end training. The combined approach results in a throughput increase of **29.0% to 32.1%** across all tested model configurations. This leads to higher hardware utilization and shorter training times.

4. Evaluated Model Configurations

The evaluations were performed on four GPT-style model configurations, with tensor parallelism fixed at 8-way. No data parallelism was used in these specific evaluations to isolate the impact of the proposed techniques.

5x Less Memory, 30% Faster: The Two Simple Tricks Speeding Up AI Training

AI models are growing at an astonishing rate, scaling from millions to billions, and now even trillions, of parameters. This relentless growth has unlocked incredible capabilities, but it begs a fundamental question: how do we actually *train* these colossal models without hitting a computational wall? The primary obstacle is GPU memory. The sheer size of these models, along with the data required to train them, can easily overwhelm even the most powerful hardware available today.

For years, the AI community had been forced to make a painful bargain. To solve the memory problem, engineers relied on a technique called "full activation recomputation"—a deal with the devil that traded precious training time for vital memory. Instead of storing massive temporary datasets generated during training, they would discard them and then re-calculate them from scratch when needed. This trick saved huge amounts of memory, but it came with a steep penalty: this redundant computation slowed down the entire training process by a staggering 30-40%.

This created a frustrating paradox where researchers had to choose between saving memory and saving time. But what if that trade-off was unnecessary?

Researchers at NVIDIA have demonstrated two surprisingly simple techniques that break this long-standing compromise. By rethinking how data is managed and recomputed, they managed to dramatically reduce memory usage *and* speed up training, proving you can have the best of both worlds.

Takeaway 1: The Real Memory Hog Isn't What You Think

When thinking about the memory needed to train a giant AI model, everyone assumed the model's parameters—the billions of weights it learns—were the main problem. While these are significant, the real culprit was hiding in the process itself, in the form of temporary data known as "activations."

In technical terms, an activation is "any tensor that is created in the forward pass and is necessary for gradient computation during back-propagation." In other words, they are the intermediate results the model must remember to figure out how to learn from its mistakes during the "backward pass." The problem is, for truly massive models, the amount of memory needed to store these activations can be astronomical.

As data from the research shows, this isn't a minor issue. For models ranging from 22 billion to 1 trillion parameters, the memory required for activations vastly exceeds the 80GB capacity of a top-tier NVIDIA A100 GPU. This makes it physically impossible to train these models using standard methods. This insight is critical: any solution that focuses only on shrinking the model's parameters is missing the biggest part of the engineering challenge.

Takeaway 2: A New Dimension of Parallelism

To train a single model that is too big for one GPU, researchers use "tensor parallelism," a technique that splits a single layer of the model across multiple GPUs. While this successfully split the most computationally heavy parts, the researchers identified a subtle but critical flaw. Certain operations, like layer-norms and dropouts, were left untouched. This meant a significant chunk of activation memory was wastefully replicated on every single GPU, creating a new memory wall that parallelism couldn't break.

The clever solution was "Sequence Parallelism." The researchers' key insight was that for these specific, un-split operations, the calculations are independent along the sequence dimension. Imagine a long sentence being processed: what happens to the first half is independent of what happens to the second half. This insight allowed them to split the sentence itself across GPUs—GPU A handles the first 1024 tokens, GPU B handles the next 1024—for just these parts of the calculation, dramatically reducing replicated memory.

The most impressive part was how they avoided a performance penalty. Adding a new dimension of parallelism would normally require extra communication between GPUs, slowing things down. But they designed the new communication steps (an all-gather and a reduce-scatter) to be combined with existing ones. As the paper notes, "a ring all-reduce is composed of two steps: a reduce-scatter followed by an all-gather." This elegant engineering trick meant they could save a huge amount of memory with, as the paper states, "no communication overhead."

Takeaway 3: Not All Calculations Are Worth Re-doing

The old method of "full activation recomputation" was a sledgehammer: to save memory, it simply re-calculated almost an entire transformer layer. This worked, but it came with that 30-40% speed penalty. The researchers' new idea was a scalpel.

They proposed a surgical approach called "selective activation recomputation." Instead of recomputing everything, they analyzed a transformer layer to identify which specific parts took up the most activation memory while being computationally cheap to re-do. Their analysis of a GPT-3-style model was striking: the specific attention operations they targeted accounted for a staggering **70% of activation memory**, yet recomputing them added a mere **2.7% FLOPs**

overhead. By only recomputing this small, high-impact portion, they could get most of the memory savings of the old method while incurring only a fraction of the performance cost.
...we propose to checkpoint and recompute only parts of each transformer layer that take up a considerable amount of memory but are not computationally expensive to recompute...

The Result: A 5x Memory Reduction and 90% Less Overhead

By combining these two elegant ideas—Sequence Parallelism and Selective Recomputation—the researchers achieved dramatic improvements across the board. The new approach is not just a marginal improvement; it represents a fundamental step forward in training efficiency.
The top-line results speak for themselves:
• **5x reduction** in the memory required to store activations.
• **Over 90% reduction** in execution time overhead. The painful 30-40% speed penalty caused by recomputation was almost entirely eliminated, with the new overhead dropping to a mere 2-4%.
• **~30% faster** end-to-end training. For instance, a 530-billion parameter model trained 29.7% faster than with the old method.

In practice, these savings directly translate into shorter training times and lower computational costs. This enables researchers and engineers to experiment more rapidly, iterate on new ideas, and ultimately build more powerful and capable AI models faster than ever before.

Conclusion: Rethinking Old Bottlenecks

The story of this research is a powerful reminder that sometimes the biggest breakthroughs come not from more powerful hardware, but from smarter software. For years, the trade-off between memory and speed was accepted as a necessary evil in the world of large-scale AI. By questioning that fundamental assumption, researchers developed two novel, yet conceptually simple, techniques that solved a long-standing bottleneck.

Study Guide: Reducing Activation Recomputation in Large Transformer Models

This guide is designed to review the core concepts, technical innovations, and experimental results presented in the paper "Reducing Activation Recomputation in Large Transformer Models." It provides a structured approach to understanding the challenges of training large-scale AI models and the novel solutions proposed to enhance efficiency.

Quiz: Short-Answer Questions

Instructions: Answer the following questions in 2-3 sentences based on the information provided in the source context.

1. What is the primary problem that "activation recomputation" is designed to solve in training large transformer models, and what is its main drawback?

2. Explain the concept of "sequence parallelism" and how it complements "tensor parallelism" to reduce activation memory.

3. What is "selective activation recomputation," and why is it more efficient than "full activation recomputation"?

4. According to the paper, why does pipeline parallelism not uniformly reduce the memory required for activations across all stages? Which stage experiences the most memory pressure?

5. The paper introduces conjugate operators g and $\bar{g}$ for sequence parallelism. What operations do they perform in the forward and backward passes, and why do they not introduce communication overhead compared to tensor parallelism alone?

6. Quantitatively, by what factor do sequence parallelism and selective activation recomputation combined reduce activation memory, and by how much do they reduce the execution time overhead from recomputation?

7. In "selective activation recomputation," which specific parts of the self-attention block are targeted for recomputation, and what is the rationale for choosing them?

8. What is Model FLOPs Utilization (MFU), and how does the approach presented in the paper improve it for large models like the 530B parameter configuration?

9. How does tensor parallelism alone fall short in reducing activation memory, leading to the need for sequence parallelism?

10. Based on Figure 1, what is the key observation regarding the memory requirements for baseline model training versus the "present work" for models from 22B to 1T parameters, relative to the NVIDIA A100 GPU's memory capacity?

Answer Key

1. Activation recomputation solves the problem of memory capacity constraints by not storing most activations, instead recomputing them as needed for the backward pass. Its main drawback is the significant computational penalty, as it adds a redundant forward pass that can increase execution time by 30-40%.

2. Sequence parallelism is a technique that partitions activations along the sequence dimension in regions of a transformer layer—such as layer-norms and dropouts—that are not parallelized by tensor parallelism. It complements tensor parallelism by eliminating the redundant storage of these activations across the tensor parallel group, further reducing overall memory usage.

3. Selective activation recomputation is a technique where only specific, memory-intensive but computationally inexpensive parts of each transformer layer are recomputed. This is more efficient than recomputing the full layer because it saves a large percentage of activation memory (e.g., 65-70%) while introducing a very small FLOPs overhead (e.g., 1.6-2.7%).

4. Pipeline parallelism does not uniformly reduce activation memory because efficient schedules require storing activations for several microbatches to keep the pipeline full and reduce idle time (the "pipeline bubble"). The first stage of the pipeline experiences the most memory pressure because it must store activations for `p` microbatches, where `p` is the pipeline parallel size.

5. In the forward pass, `g` is an all-gather and `ḡ` is a reduce-scatter; their roles are reversed in the backward pass. They do not introduce communication overhead because a ring all-reduce operation (used in tensor parallelism) is composed of a reduce-scatter followed by an all-gather, meaning the communication bandwidth used is the same.

6. Combined, the two techniques provide a 5x reduction in activation memory. They also recover over 90% of the compute overhead introduced by full activation recomputation, reducing the overall execution time overhead to as low as 2-4% for very large models.

7. The technique targets operations within the self-attention block after the Q, K, and V values are calculated: the `QKT` matrix multiply, softmax, softmax dropout, and the attention over `V`. They are chosen because they have large input sizes, and thus large activations, but a very low number of floating-point operations (FLOPs) per input element.

8. MFU is the number of floating-point operations required for a single forward/backward pass (Model FLOPs) divided by the iteration time and the accelerator's peak theoretical FLOPs per second. The paper's approach improves MFU by drastically reducing iteration time; for the 530B model, it increased MFU from an implied 42.1% to 54.2%, a 29% improvement in throughput.

9. Tensor parallelism successfully parallelizes the computationally intensive attention and MLP blocks but leaves operations like layer-norms and their subsequent dropouts replicated across the tensor parallel group. These replicated operations do not require much compute but demand a considerable amount of activation memory, which sequence parallelism addresses by partitioning them.

10. Figure 1 clearly shows that for all model sizes from 22B to 1T, the baseline memory requirement significantly exceeds the 80GB capacity of an NVIDIA A100 GPU. The "present work" successfully reduces the activation memory component so that the total memory requirement for each configuration fits within the 80GB limit.

Essay Questions

Instructions: The following questions are designed for a deeper, essay-format exploration of the paper's topics. Formulate comprehensive answers by synthesizing information from across the source document.

1. Describe the memory bottleneck in training large transformer models, focusing on the role of activations. Analyze how the three main types of parallelism discussed in the paper—tensor, pipeline, and the newly proposed sequence parallelism—interact and contribute to managing this bottleneck.

2. The paper presents a formula-driven analysis of activation memory. Trace the derivation of the "Activations memory per layer" formula from its baseline form (Equation 1) through the application of tensor parallelism (Equation 2) and finally to the combined tensor and sequence parallelism (Equation 4). Explain the significance of each term's reduction.

3. Compare and contrast "full activation recomputation" with the proposed "selective activation recomputation." Discuss the trade-offs between memory savings and computational overhead for both methods, referencing the quantitative data provided for models like GPT-3 and MT-NLG.

4. Evaluate the effectiveness of the proposed techniques using the experimental results presented in Section 6. Discuss the impact on memory usage (Figure 7), execution time per layer (Table 4, Figure 8), and end-to-end throughput and Model FLOPs Utilization (Table 5).

5. The paper states its techniques are "complementary" to other methods like partitioning data across data parallel ranks or offloading to CPU memory. Based on the "Related Work" section, elaborate on how the paper's model parallelism optimizations differ from these data parallelism-based approaches and why the authors focus exclusively on the former for large-scale training.

Glossary of Key Terms

Term	Definition
Activation Recomputation	The standard approach to alleviate memory pressure where, instead of storing activations for backpropagation, they are recomputed as necessary to calculate gradients. Also called "gradient checkpointing."
Activations	Any tensor created in the forward pass that is necessary for gradient computation during backpropagation. This excludes model parameters and optimizer state but includes items like dropout masks.
All-gather	A communication collective where each participating process gathers data from all other processes, resulting in each process having a complete copy of the data. Used by the g operator in the forward pass of sequence parallelism.
All-reduce	A communication collective where data from all participating processes is combined via a reduction operation (e.g., sum), and the final result is distributed back to all processes. Used in tensor parallelism.

Full Activation Recomputation	A specific method of activation recomputation where activations are checkpointed (stored) at transformer layer boundaries, and the rest of the necessary activations within the layer are recomputed during the backward pass.
Hardware FLOPs Utilization (HFU)	The number of floating point operations actually performed on the hardware per iteration, divided by the iteration time and the accelerator's theoretical peak FLOPs per second. This accounts for operations from recomputation.
Microbatch	A smaller subdivision of a training data batch, used in pipeline parallelism to enable overlapping of computation across different stages of the model.
Model FLOPs Utilization (MFU)	The number of floating point operations required to perform a single forward and backward pass for a given model, divided by the iteration time and the accelerator's theoretical peak FLOPs per second. It is an implementation-independent measure of efficiency.
Model Parallelism	A set of techniques required to distribute model parameters, activations, and optimizer state across multiple devices when a model is too large to fit into a single device's memory.
MLP Block	The Multi-Layer Perceptron block within a transformer layer. It typically consists of two linear layers that increase the hidden size (e.g., to $4h$) and then reduce it back to the original size (h).
Pipeline Bubble	The idle time or underutilization of devices that occurs at the beginning and end of a training iteration in a pipeline parallel setup. Efficient schedules aim to minimize this bubble by keeping devices busy.
Pipeline Parallelism	A form of model parallelism where the model's layers are split and distributed across different devices, which form stages in a pipeline.
Reduce-scatter	A communication collective where data from all processes is combined via a reduction operation, and the resulting data is then scattered (partitioned) among the processes. Used by the $\bar{g}$ operator in the forward pass of sequence parallelism.
Selective Activation Recomputation	A novel technique that checkpoints and recomputes only specific parts of each transformer layer that are memory-intensive but not computationally expensive, such as parts of the self-attention mechanism.
Self-Attention Block	A core component of a transformer layer responsible for calculating attention scores. It includes Query (Q), Key (K), and Value (V) projections, matrix multiplications, softmax, and dropout.
Sequence Parallelism	A novel technique that partitions activations along the sequence dimension in the non-tensor-parallel regions of a transformer layer. It works in conjunction with tensor parallelism to reduce activation memory without additional communication overhead.
Tensor Parallelism	A form of model parallelism where the parameters and computations of individual layers (e.g., large matrix multiplications) are distributed across a group of devices.
Transformer Layer	The fundamental building block of a transformer model, replicated multiple times. Each layer typically consists of a self-attention block and an MLP block, with layer normalization and residual connections.