Jain et al., “Checkmate: Breaking the Memory Wall With Optimal Tensor Rematerialization” (2020)
Recompute activations instead of storing them

Gradient Checkpointing, Chen et al. (2016)
  • Pick segments to recompute in backward pass
  • $O(\sqrt{N})$ memory for $O(N)$ extra ops
  • Many later segmenting approaches

Checkmate, Jain et al. (2020)
  • Rematerialize individual values
  • ILP for optimal(!) planning
Checkpointing: Trade Time for Space

- Recompute activations instead of storing them
- **Gradient Checkpointing**, Chen et al. (2016)
  - Pick segments to recompute in backward pass
  - $O(\sqrt{N})$ memory for $O(N)$ extra ops
  - Many later segmenting approaches
- **Checkmate**, Jain et al. (2020)
  - Rematerialize individual values
  - ILP for optimal(!) planning
• Recompute activations instead of storing them
• Gradient Checkpointing, Chen et al. (2016)
  • Pick segments to recomputed in backward pass
  • $O(\sqrt{N})$ memory for $O(N)$ extra ops
  • Many later segmenting approaches
• Checkmate, Jain et al. (2020)
  • Rematerialize individual values
  • ILP for optimal(!) planning
Past approaches plan checkpoints in advance
- Require static knowledge of the model
- Planning can be expensive, limits applications

Our contributions:
- **Static planning is unnecessary for checkpointing**
- Still achieve good compute-memory tradeoffs
Dynamic Tensor Rematerialization

- Cache-like approach: A runtime system
  - No static information necessary
  - Greedily allocate, evict and recompute as needed
  - Collects metadata to guide heuristics
  - Operates at a high level of abstraction
- Still competitive with static planning!
Rematerializing on the Fly

Execution trace when computing $t_7$

Current operation: $\text{PerformOp}(op_7, [t_5, t_6])$
Current operation: \( \text{PerformOp}(\text{op}_7, [t_5, t_6]) \)
Problem: Need to compute $t_7$ but $t_5$ is evicted

Current operation: Rematerialize($t_5$)
Rematerializing on the Fly

$\mathbf{t_3}$ is present, but no room for result

Current operation: $\text{PerformOp}(op_5, [t_3])$
Rematerializing on the Fly

The heuristic is free to pick $t_2$

MEMORY BUDGET: 4

Current operation: PerformEviction()
Rematerializing on the Fly

Now we can recompute $t_5$

Current operation: $\text{AllocateBuffer}(t_5.\text{size}); \text{op}_5(t_3)$
Rematerializing on the Fly

Our arguments are back—but still no room for $t_7$!

Current operation: AllocateBuffer($t_7$.size)
Rematerializing on the Fly

Don’t need $t_3$ right now, so we can evict

Current operation: PerformEviction()
Rematerializing on the Fly

Current operation: \( op_7(t_5, t_6) \)

Now we can proceed

MEMORY BUDGET: 4

\( \bigcirc \ = \text{IN MEMORY} \)
Rematerializing on the Fly

MEMORY BUDGET: 4
☐ = IN MEMORY
**AllocateBuffer(size):** Allocate if enough room, else evict until there is

**PerformEviction():** Heuristic chooses a tensor to evict

**Rematerialize(t):** Recompute t by replaying its parent op (PerformOp)

**PerformOp(op, args):**
- Rematerialize evicted arguments
- Make room for result
- Update metadata
What Do Heuristics Look Like?

- Dynamic prediction of which tensor is least valuable
- Useful metadata, easy to track:
  - Cost $c(t)$: Avoid recomputing expensive tensors
  - Staleness $s(t)$: Recently used $\implies$ likely to be used soon
  - Memory $m(t)$: Large tensors are most profitable to evict
- Resulting policy: minimize $h(t) = \frac{c(t)}{(m(t) \cdot s(t))}$
- Others: LRU $\left(\frac{1}{s(t)}\right)$ and largest-first $\left(\frac{1}{m(t)}\right)$
Reasoning About Tensor Cost

• True cost of a rematerialization includes recursive calls
• Recursively computing exact cost is expensive!
• We approximate evicted components via union-find
  • Keep a running sum for union-find components
  • When tensor rematerialized, map to a new component
  • Leaves “phantom connections” but is fast
Performance on $N$-layer linear feedforward network:

- $\Omega(\sqrt{N})$ memory and $O(N)$ operations
- Same bound as Chen et al. (2016)
- No advance knowledge of model!
Proof (Sketch) in Pictures

Reduced (compute-memory), $2\sqrt{n}$ memory (n=128 layers)
Proof (Sketch) in Pictures

Reduced (compute-memory), $2\sqrt{n}$ memory (n=128 layers)

Horizontal lines: Checkpoints!

Triangles: Recomputing segments
Proof (Sketch) in Pictures

Reduced (compute-memory), $2\sqrt{n}$ memory ($n$=128 layers)

Full proof shows:
- Checkpoints are evenly spaced
- At most constant segment cost
Proof (Sketch) in Pictures

Reduced (compute-memory), $2\sqrt{n}$ memory (n=128 layers)

Also a “no-free-lunch” proof:

- Adversarial input exists for every heuristic
- Hence our empirical exploration
Comparison of Heuristics (Simulated)
Comparison of Heuristics (Simulated)

Overhead: Additional operator cost

Budget: Fraction of memory needed without checkpointing
Comparison of Heuristics (Simulated)

Heuristics of varying complexity

Heuristics using more metadata hit lower budgets
Similar trend holds across all models examined!
Comparison of Heuristics (Simulated)

Similar trend holds across all models examined!
Comparison Against Static Techniques

Simulated comparison via the Checkmate MLSys 2020 artifact
Comparison Against Static Techniques

Neck-and-neck with optimal! But runs in milliseconds

Simulated comparison via the Checkmate ML Sys 2020 artifact
Thin wrapper over tensor operators, core logic a few hundred LOC
Prototype Implementation in PyTorch

Overhead due to naively looping through tensors

Thin wrapper over tensor operators, core logic a few hundred LOC
Strange profiling behavior, perhaps due to Python reflection—but the prototype still worked.

Thin wrapper over tensor operators, core logic a few hundred LOC.
• Encouraging initial results
• Many possible avenues of future work
  • Distributed settings: DTR per GPU?
  • Combining DTR with swapping
  • Tighter integration into the memory manager
  • Learning heuristics, learn from past batches
• Check out the simulator and prototype!
  https://github.com/uwsamp1/dtr-prototype