CHECKMATE: BREAKING THE MEMORY WALL WITH OPTIMAL TENSOR REMATERIALIZATION

Paras Jain [*1] **Ajay Jain** [*1] **Aniruddha Nrusimha** [1]
Amir Gholami [1] **Pieter Abbeel** [1] **Kurt Keutzer** [1] **Ion Stoica** [1] **Joseph E. Gonzalez** [1]

ABSTRACT

We formalize the problem of trading-off DNN training time and memory requirements as the *tensor rematerialization* optimization problem, a generalization of prior checkpointing strategies. We introduce Checkmate, a system that solves for optimal rematerialization schedules in reasonable times (under an hour) using off-the-shelf MILP solvers or near-optimal schedules with an approximation algorithm, then uses these schedules to accelerate millions of training iterations. Our method scales to complex, realistic architectures and is hardware-aware through the use of accelerator-specific, profile-based cost models. In addition to reducing training cost, Checkmate enables real-world networks to be trained with up to 5.1× larger input sizes. Checkmate is an open-source project, available at `https://github.com/parasj/checkmate`.

1 INTRODUCTION

Deep learning training workloads demand large amounts of high bandwidth memory. Researchers are pushing the memory capacity limits of hardware accelerators such as GPUs by training neural networks on high-resolution images (Dong et al., 2016; Kim et al., 2016; Tai et al., 2017), 3D point-clouds (Chen et al., 2017; Yang et al., 2018), and long natural language sequences (Vaswani et al., 2017; Devlin et al., 2018; Child et al., 2019). In these applications, training memory usage is dominated by the intermediate activation tensors needed for backpropagation (Figure 3).

The limited availability of high bandwidth on-device memory creates a *memory wall* that stifles exploration of novel architectures. Across applications, authors of state-of-the-art models cite memory as a limiting factor in deep neural network (DNN) design (Krizhevsky et al., 2012; He et al., 2016; Chen et al., 2016a; Gomez et al., 2017; Pohlen et al., 2017; Child et al., 2019; Liu et al., 2019; Dai et al., 2019).

As there is insufficient RAM to cache all activation tensors for backpropagation, some select tensors can be discarded during forward evaluation. When a discarded tensor is necessary as a dependency for gradient calculation, the tensor can be *rematerialized*. As illustrated in Figure 1, rematerializing values allows a large DNN to fit within memory at the expense of additional computation.

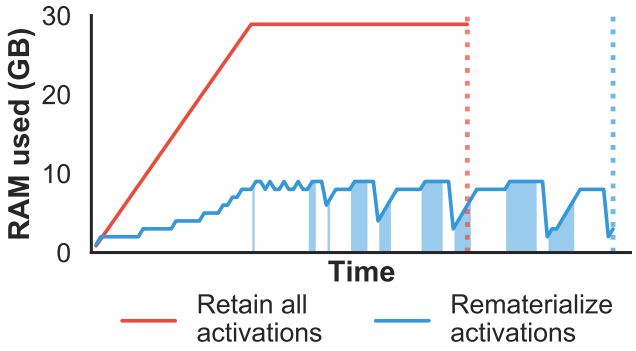

Figure 1. This 32-layer deep neural network requires 30GB of memory during training in order to cache forward pass activations for the backward pass. Freeing certain activations early and rematerializing them later reduces memory requirements by 21GB at the cost of a modest runtime increase. Rematerialized layers are denoted as shaded blue regions. We present Checkmate, a system to rematerialize large neural networks *optimally*. Checkmate is hardware-aware, memory-aware and supports arbitrary DAGs.

Griewank & Walther (2000) and Chen et al. (2016b) present heuristics for rematerialization when the forward pass forms a linear graph, or path graph. They refer to the problem as checkpointing. However, their approaches cannot be applied generally to nonlinear DNN structures such as residual connections, and rely on the strong assumption that all nodes in the graph have the same cost. Prior work also assumes that gradients may never be rematerialized. These assumptions limit the efficiency and generality of prior approaches.

Our work formalizes tensor rematerialization as a constrained optimization problem. Using off-the-shelf numerical solvers, we are able to discover optimal rematerializa-

[*]Equal contribution [1]Department of EECS, UC Berkeley. Correspondence to: Paras Jain <parasj@berkeley.edu>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020.

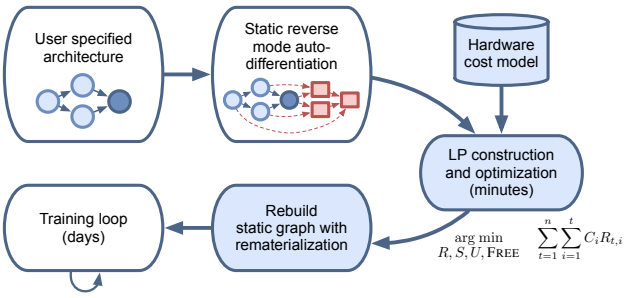

Figure 2. Overview of the Checkmate system.

tion strategies for arbitrary deep neural networks in TensorFlow with non-uniform computation and memory costs. We demonstrate that optimal rematerialization allows larger batch sizes and substantially reduced memory usage with minimal computational overhead across a range of image classification and semantic segmentation architectures. As a consequence, our approach allows researchers to easily explore larger models, at larger batch sizes, on more complex signals with minimal computation overhead.

In particular, the contributions of this work include:

- a formalization of the rematerialization problem as a mixed integer linear program with a substantially more flexible search space than prior work, in Section 4.7.
- a fast approximation algorithm based on two-phase deterministic LP rounding, in Section 5.
- Checkmate, a system implemented in TensorFlow that enables training models with up to 5.1× larger input sizes than prior art at minimal overhead.

2 Motivation

Memory consumption during training consists of (a) intermediate features, or activations, whose size depends on input dimensions and (b) parameters and their gradients whose size depends on weight dimensions. Given that inputs are often several order of magnitude larger than kernels, most memory is used by features, as demonstrated in Figure 3.

Frameworks such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2017; 2019) store all activations during the forward pass. Gradients are backpropagated from the loss node, and each activation is freed after its gradient has been calculated. In Figure 1, we compare this memory intensive policy and a rematerialization strategy for a real neural network. Memory usage is significantly reduced by deallocating some activations in the forward pass and recomputing them in the backward pass. Our goal is fit an arbitrary network within our memory budget while incurring the minimal additional runtime penalty from recomputation.

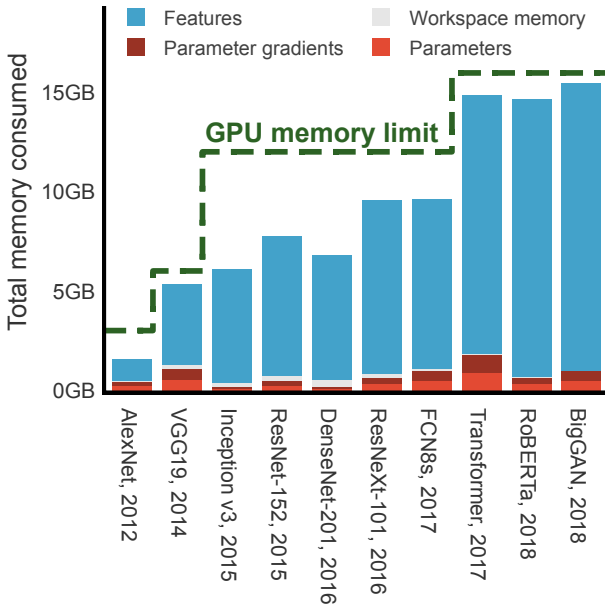

Figure 3. Memory consumed by activations far outweigh parameters for popular model architectures. Moreover, advances in GPU DRAM capacity are quickly utilized by researchers; the dashed line notes the memory limit of the GPU used to train each model.

Most prior work assumes networks have linear graphs. For example, Chen et al. (2016b) divides the computation into $\sqrt{n}$ segments, each with $\sqrt{n}$ nodes. Each segment endpoint is stored during the forward pass. During the backward pass, segments are recomputed in reverse order at $O(n)$ cost.

Linear graph assumptions limit applicability of prior work. For example, the popular ResNet50 (He et al., 2016) requires each residual block to be treated as a single node, leading to inefficient solutions. For other networks with larger skip connection (*e.g.*, U-Net (Ronneberger et al., 2015)), the vast majority of the graph is incompatible.

Prior work also assumes all layers are equally expensive to recompute.In the VGG19 (Simonyan & Zisserman, 2014) architecture, the largest layer is *seven orders of magnitude* more expensive than the smallest layer.

Our work makes few assumptions on neural network graphs. We explore a solution space that allows for (a) arbitrary graphs with several inputs and outputs for each node, (b) variable memory costs across layers and (c) variable computation costs for each layer (such as FLOPs or profiled runtimes). We constrain solutions to simply be *correct* (a node's dependencies must be materialized before it can be evaluated) and *within the RAM budget* (at any point during execution, resident tensors must fit into RAM).

To find solutions to this generalized problem, we find solutions that minimize the amount of time it takes to perform

a single training iteration, subject to the correctness and memory constraints outlined above. We project schedules into space and time, allowing us to cast the objective as a linear expression. This problem can then be solved using off-the-shelf mixed integer linear program solvers such as GLPK or COIN-OR Branch-and-Cut (Forrest et al., 2019). An optimal solution to the MILP will minimize the amount of additional compute cost within the memory budget.

3 Related Work

We categorize related work as checkpointing, reversible networks, distributed computation, and activation compression.

Checkpointing and rematerialization Chen et al. (2016b) propose a heuristic for checkpointing idealized unit-cost linear n-layer graphs with $O(\sqrt{n})$ memory usage. Griewank & Walther (2000) checkpoint similar linear unit-cost graphs with $O(\log n)$ memory usage and prove optimality for linear chain graphs with unit per-node cost and memory. In practice, DNN layers vary significantly in memory usage and computational cost (Sze et al., 2017), so these heuristics are not optimal in practice. Chen et al. (2016b) also develop a greedy algorithm that checkpoints layers of a network in roughly memory equal segments, with a hyperparameter b for the size of such segments. Still, neither procedure is cost-aware nor deallocates checkpoints when possible. Gruslys et al. (2016) develop a dynamic programming algorithm for checkpoint selection in unrolled recurrent neural network training, exploiting their linear forward graphs. Feng & Huang (2018) provide a dynamic program to select checkpoints that partition branching networks but ignore layer costs and memory usage. Siskind & Pearlmutter (2018a) develop a divide-and-conquer strategy in programs. Beaumont et al. (2019) use dynamic programming for checkpoint selection in a specific architecture with joining sub-networks.

Intermediate value recomputation is also common in register allocation. Compiler backends lower an intermediate representation of code to an architecture-specific executable binary. During lowering, an abstract *static single assignment* (SSA) graph of values and operations (Rosen et al., 1988; Cytron et al., 1991) is concretized by mapping values to a finite number of registers. If insufficient registers are available for an SSA form computation graph, values are *spilled* to main memory by storing and later loading the value. Register allocation has been formulated as graph coloring problem (Chaitin et al., 1981), integer program (Goodwin & Wilken, 1996; Lozano et al., 2018), and network flow (Koes & Goldstein, 2006).

Register allocators may recompute constants and values with register-resident dependencies if the cost of doing so is less than the cost of a spill (Chaitin et al., 1981; Briggs et al., 1992; Punjani, 2004). While similar to our setup, register rematerialization is limited to exceptional values that can be recomputed in a single instruction with dependencies already in registers. For example, memory offset computations can be cheaply recomputed, and loads of constants can be statically resolved. In contrast, Checkmate can recompute entire subgraphs of the program's data-flow.

During the evaluation of a single kernel, GPUs spill per-thread registers to a thread-local region of global memory (*i.e.* local memory) (Micikevicius, 2011; NVIDIA, 2017). NN training executes DAGs of kernels and stores intermediate values in shared global memory. This produces a high range of value sizes, from 4 byte floats to gigabyte tensors, whereas CPU and GPU registers range from 1 to 64 bytes. Our problem of interkernel memory scheduling thus differs in scale from the classical problem of register allocation within a kernel or program. Rematerialization is more appropriate than copying values out of core as the cost of spilling values from global GPU memory to main memory (RAM) is substantial (Micikevicius, 2011; Jain et al., 2018), though possible (Meng et al., 2017).

Reversible Networks Gomez et al. (2017) propose a reversible (approximately invertible) residual DNN architecture, where intermediate temporary values can be recomputed from values derived *later* in the standard forward computation. Reversibility allows forward pass activations to be recomputed during the backward pass rather than stored, similar to gradient checkpointing. Bulo et al. (2018) replace only ReLU and batch normalization layers with invertible variants, reconstructing their inputs during the backward pass, reducing memory usage up to 50%. However, this approach has a limit to memory savings, and does not support a range of budgets. Reversibility is not yet widely used to save memory, but is a promising complementary approach.

Distributed computation An orthogonal approach to address the limited memory problem is distributed-memory computations and gradient accumulation. However, model parallelism requires access to additional expensive compute accelerators, fast networks, and non-trivial partitioning of model state to balance communication and computation (Gholami et al., 2018; Jia et al., 2018b; McCandlish et al.). Gradient accumulation enables larger batch sizes by computing the gradients in sub-batches across a minibatch. However, gradient accumulation often degrades performance as batch normalization performs poorly on small minibatch sizes (Wu & He, 2018; Ioffe & Szegedy, 2015).

Activation compression In some DNN applications, it is possible to process compressed representations with minimal accuracy loss. Gueguen et al. (2018) classify discrete cosine transforms of JPEG images rather than raw images. Jain et al. (2018) quantizes activations, cutting memory usage in half. Compression reduces memory usage by a constant factor, but reduces accuracy. Our approach is math-

METHOD	DESCRIPTION	GENERAL GRAPHS	COST AWARE	MEMORY AWARE
Checkpoint all (Ideal)	No rematerialization. Default in deep learning frameworks.	√	×	×
Griewank et al. $\log n$	Griewank & Walther (2000) REVOLVE procedure	×	×	×
Chen et al. $\sqrt{n}$	Chen et al. (2016b) checkpointing heuristic	×	×	×
Chen et al. greedy	Chen et al. (2016b), with search over parameter b	×	×	∼
AP $\sqrt{n}$	Chen et al. $\sqrt{n}$ on articulation points + optimal R solve	∼	×	×
AP greedy	Chen et al. greedy on articulation points + optimal R solve	∼	×	∼
Linearized $\sqrt{n}$	Chen et al. $\sqrt{n}$ on topological sort + optimal R solve	√	×	×
Linearized greedy	Chen et al. greedy on topological sort + optimal R solve	√	×	∼
Checkmate ILP	Our ILP as formulated in Section 4	√	√	√
Checkmate approx.	Our LP rounding approximation algorithm (Section 5)	√	√	√

Table 1. Rematerialization baselines and our extensions to make them applicable to non-linear architectures

ematically equivalent and incurs no accuracy penalty.

4 OPTIMAL REMATERIALIZATION

In this section, we develop an optimal solver that schedules computation and garbage collection during the evaluation of general data-flow graphs including those used in neural network training. Our proposed scheduler minimizes computation or execution time while guaranteeing that the schedule will not exceed device memory limitations. The rematerialization problem is formulated as a mixed integer linear program (MILP) that can be solved with standard commercial or open-source solvers.

4.1 Problem definition

A computation or data-flow graph $G = (V, E)$ is a directed acyclic graph with n nodes $V = \{v_1, \ldots, v_n\}$ that represent operations yielding values (*e.g.* tensors). Edges represent dependencies between operators, such as layer inputs in a neural network. Nodes are numbered according to a topological order, such that operation v_j may only depend on the results of operations $v_{i<j}$.

Each operator's output takes M_v memory to store and costs C_v to compute from its inputs. We wish to find the terminal node v_n with peak memory consumption under a memory budget, M_{budget}, and minimum total cost of computation.

4.2 Representing a schedule

We represent a schedule as a series of nodes being saved or (re)computed. We unroll the execution of the network into T stages and only allow a node to be computed once per stage. $S_{t,i} \in \{0, 1\}$ indicates that the result of operation i should be retained in memory at stage $t-1$ until stage t. We also define $R_{t,i} \in \{0, 1\}$ be a binary variable reflecting whether operation i is recomputed at time step t.

Our representation generalizes checkpointing (Griewank & Walther, 2000; Chen et al., 2016b; Gruslys et al., 2016; Siskind & Pearlmutter, 2018b; Feng & Huang, 2018), as values can be retained and deallocated many times, but comes at the cost of $O(Tn)$ decision variables.

To trade-off the number of decision variables and schedule flexibility, we limit T to $T = n$. This allows for $O(n^2)$ operations and constant memory in linear graphs.

4.3 Scheduling with ample memory

First, consider neural network evaluation on a processor with ample memory. Even without a memory constraint, our solver must ensure that checkpointed and computed operations have dependencies resident in memory. Minimizing the total cost of computation across stages with dependency constraints yields objective (1a):

$$\underset{R,S}{\arg\min} \quad \sum_{t=1}^{n}\sum_{i=1}^{t} C_i R_{t,i} \tag{1a}$$

subject to

$$R_{t,j} \leq R_{t,i} + S_{t,i} \qquad \forall t\, \forall (v_i, v_j) \in E, \tag{1b}$$

$$S_{t,i} \leq R_{t-1,i} + S_{t-1,i} \qquad \forall t \geq 2\, \forall i, \tag{1c}$$

$$\textstyle\sum_i S_{1,i} = 0, \tag{1d}$$

$$\textstyle\sum_t R_{t,n} \geq 1, \tag{1e}$$

$$R_{t,i}, S_{t,i} \in \{0, 1\} \qquad \forall t\, \forall i \tag{1f}$$

Constraints ensure feasibility and completion. Constraint (1b) and (1c) ensure that an operation is computed in stage t only if all dependencies are available. To cover the edge case of the first stage, constraint (1d) specifies that no values are initially in memory. Finally, covering constraint (1e) ensures that the last node in the topological order is computed at some point in the schedule so that training progresses.

4.4 Constraining memory utilization

To constrain memory usage, we introduce memory accounting variables $U_{t,k} \in \mathbb{R}_+$ into the ILP. Let $U_{t,k}$ denote the

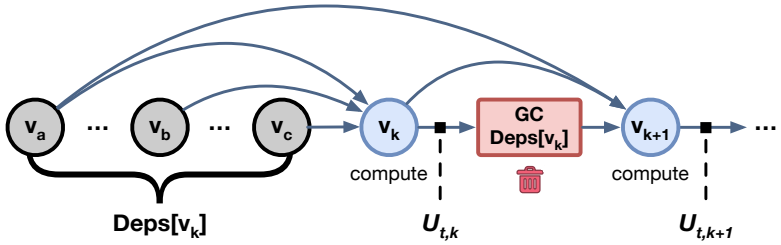

Figure 4. Dependencies of v_k can only be garbage collected after it is evaluated. $U_{t,k}$ measures the memory used after evaluating v_k and before deallocating its dependencies. v_b and v_c may be deallocated during garbage collection, but v_a may not due to a forward edge.

memory used just after computing node v_k in stage t. $U_{t,k}$ is defined recursively in terms of auxiliary binary variables $\text{FREE}_{t,i,k}$ for $(v_i, v_k) \in E$, which specifies whether node v_i may be deallocated in stage t after evaluating node v_k.

We assume that (1) network inputs and parameters are always resident in memory and (2) enough space is allocated for gradients of the loss with respect to parameters.[1] Parameter gradients are typically small, the same size as the parameters themselves. Additionally, at the beginning of a stage, all checkpointed values are resident in memory. Hence, we initialize the recurrence,

$$U_{t,0} = \underbrace{M_{\text{input}} + 2M_{\text{param}}}_{\text{Constant overhead}} + \sum_{i=1}^{n} \underbrace{M_i S_{t,i}}_{\text{Checkpoints}} \tag{2}$$

Suppose $U_{t,k}$ bytes of memory are in use after evaluating v_k. Before evaluating v_{k+1}, v_k and dependencies (parents) of v_k may be deallocated if there are no future uses. Then, an output tensor for the result of v_{k+1} is allocated, consuming memory M_{k+1}. The timeline is depicted in Figure 4, yielding recurrence (3):

$$U_{t,k+1} = U_{t,k} - \text{mem\_freed}_t(v_k) + R_{t,k+1} M_{k+1}, \tag{3}$$

where $\text{mem\_freed}_t(v_k)$ is the amount of memory freed by deallocating v_k and its parents at stage t. Let

$$\begin{aligned} \text{DEPS}[k] &= \{i : (v_i, v_k) \in E\}, \text{ and} \\ \text{USERS}[i] &= \{j : (v_i, v_j) \in E\} \end{aligned}$$

denote parents and children of a node, respectively. Then, in terms of auxiliary variable $\text{FREE}_{t,i,k}$, for $(v_i, v_k) \in E$,

$$\text{mem\_freed}_t(v_k) = \sum_{\substack{i \in \text{DEPS}[k] \\ \cup \{k\}}} M_i * \text{FREE}_{t,i,k}, \text{ and} \tag{4}$$

$$\text{FREE}_{t,i,k} = R_{t,k} * \underbrace{(1 - S_{t+1,i})}_{\text{Not checkpoint}} \prod_{\substack{j \in \text{USERS}[i] \\ j > k}} \underbrace{(1 - R_{t,j})}_{\text{Not dep.}} \tag{5}$$

[1] While gradients can be deleted after updating parameters, we reserve constant space since many parameter optimizers such as SGD with momentum maintain gradient statistics.

The second factor in (5) ensures that M_i bytes are freed only if v_i is not checkpointed for the next stage. The final factors ensure that $\text{FREE}_{t,i,k} = 0$ if any child of v_i is computed in the stage, since then v_i needs to be retained for later use. Multiplying by $R_{t,k}$ in (5) ensures that values are only freed at most once per stage according to Theorem 4.1,

Theorem 4.1 (No double deallocation)**.** If (5) holds for all $(v_i, v_k) \in E$, then $\sum_{k \in \text{USERS}[i]} \text{FREE}_{t,i,k} \leq 1 \; \forall t, i$.

Proof. Assume for the sake of contradiction that $\exists k_1, k_2 \in \text{USERS}[i]$ such that $\text{FREE}_{t,i,k_1} = \text{FREE}_{t,i,k_2} = 1$. By the first factor in (5), we must have $R_{t,k_1} = R_{t,k_2} = 1$. Assume without loss of generality that $k_2 > k_1$. By the final factor in (5), we have $\text{FREE}_{t,i,k_1} \leq 1 - R_{t,k_2} = 0$, which is a contradiction. □

4.5 Linear reformulation of memory constraint

While the recurrence (2-3) defining U is linear, the right hand size of (5) is a polynomial. To express FREE in our ILP, it must be defined via linear constraints. We rely on Lemma 4.1 and 4.2 to reformulate (5) into a tractable form.

Lemma 4.1 (Linear Reformulation of Binary Polynomial)**.** *If $x_1, \ldots, x_n \in \{0, 1\}$, then*

$$\prod_{i=1}^{n} x_i = \begin{cases} 1 & \sum_{i=1}^{n} (1 - x_i) = 0 \\ 0 & \textit{otherwise} \end{cases}$$

Proof. If all $x_1, \ldots, x_n = 1$, then $\sum_{i=1}^{n} (1 - x_i) = 0$ and we have $\Pi_{i=1}^{n} x_i = 1$. If otherwise any $x_j = 0$, then we have $\Pi_{i=1}^{n} x_i = 0$, as desired. This can also be seen as an application of De Morgan's laws for boolean arithmetic. □

Lemma 4.2 (Linear Reformulation of Indicator Constraints)**.** *Given $0 \leq y \leq \kappa$ where y is integral and κ is a constant upper bound on y, then*

$$x = \begin{cases} 1 & y = 0 \\ 0 & \textit{otherwise} \end{cases}$$

if and only if $x \in \{0, 1\}$ and $(1 - x) \leq y \leq \kappa(1 - x)$.

Proof. For the forward direction, first note that by construction, $x \in \{0,1\}$. If $y = 0$ and $x = 1$, then $(1-x) = 0 \leq y \leq 0 = \kappa(1-x)$. Similarly, if $y \geq 1$ and $x = 0$, then $1 \leq y \leq \kappa$, which is true since $0 \leq y \leq \kappa$ and y is integral. The converse holds similarly. □

To reformulate Constraint 5, let num_hazards(t,i,k) be the number of zero factors on the RHS of the constraint. This is a linear function of the decision variables,

$$\text{num\_hazards}(t,i,k) = (1-R_{t,k}) + S_{t+1,i} + \sum_{\substack{j \in \text{USERS}[i] \\ j>k}} R_{t,j}$$

Applying Lemma 4.1 to the polynomial constraint, we have,

$$\text{FREE}_{t,i,k} = \begin{cases} 1 & \text{num\_hazards}(t,i,k) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

By Lemma 4.2, if κ is the maximum value that num_hazards(t,i,k) can assume, the following constraints are equivalent to (6),

$$\text{FREE}_{t,i,k} \in \{0,1\} \tag{7a}$$
$$1 - \text{FREE}_{t,i,k} \leq \text{num\_hazards}(t,i,k) \tag{7b}$$
$$\kappa(1 - \text{FREE}_{t,i,k}) \geq \text{num\_hazards}(t,i,k) \tag{7c}$$

4.6 Tractability via frontier-advancing stages

Fixing the execution order of nodes in the graph can improve the running time of the algorithm. In eager-execution frameworks such as PyTorch, the order is given by user code and operations are executed serially. Separating ordering and allocation is common in compiler design, and both LLVM (Lattner, 2002) and GCC (Olesen, 2011) have separate instruction scheduling and register allocation passes.

Any topological order of the nodes is a possible execution order. Given a topological order, such as the one introduced in Section 4.1, we partition the schedule into frontier-advancing stages such that node v_i is evaluated for the first time in stage i. We replace constraints (1d, 1e) that ensure the last node is computed with stricter constraints (8a-8c),

$$R_{i,i} = 1 \ \ \forall i \quad \text{(frontier-advancing partitions)} \tag{8a}$$
$$\textstyle\sum_{i \geq t} S_{t,i} = 0 \quad \text{(lower tri., no initial checkpoints)} \tag{8b}$$
$$\textstyle\sum_{i > t} R_{t,i} = 0 \quad \text{(lower triangular)} \tag{8c}$$

This reduces the feasible set, constraining the search space and improving running time. For an 8 layer ($n = 17$) linear graph neural network with unit C_i, M_i at a memory budget of 4, Gurobi optimizes the unpartitioned MILP in 9.4 hours and the partitioned MILP in 0.23 seconds to the same objective. In Appendix A, we analyze the integrality gap of both forms of the problem to understand the speedup.

4.7 Complete Integer Linear Program formulation

The complete memory constrained MILP follows in (9), with $O(|V||E|)$ variables and constraints.

$$\begin{aligned} \underset{R,S,U,\text{FREE}}{\arg\min} \quad & \sum_{t=1}^{n} \sum_{i=1}^{t} C_i R_{t,i} \\ \text{subject to} \quad & (1b), (1c), (1f), (2), (3), \\ & (7a), (7b), (7c), (8a), (8b), (8c), \\ & U_{t,k} \leq M_{\text{budget}} \end{aligned} \tag{9}$$

4.8 Constraints implied by optimality

Problem 9 can be simplified by removing constraints implied by optimality of a solution. $\text{FREE}_{t,k,k} = 1$ only if operation k is spuriously evaluated with no uses of the result. Hence, the solver can set $R_{t,k} = 0$ to reduce cost. We eliminate $|V|^2$ variables $\text{FREE}_{t,k,k}$, assumed to be 0, by modifying (4) to only sum over $i \in \text{DEPS}[k]$. These variables can be computed inexpensively after solving.

4.9 Generating an execution plan

Given a feasible solution to (9), (R, S, U, FREE), Algorithm 1 generates an execution plan via a row major scan of R and S with deallocations determined by FREE. An execution plan is a program $P = (s_1, \ldots, s_k)$ with k statements. When statement `%r = compute v` is interpreted, operation v is evaluated. The symbol `%r` denotes a virtual register used to track the resulting value. Statement `deallocate %r` marks the value tracked by virtual register `%r` for garbage collection.

The execution plan generated by Algorithm 1 is further optimized by moving deallocations earlier in the plan when possible. Spurious checkpoints that are unused in a stage can be deallocated at the start of the stage rather than during the stage. Still, this code motion is unnecessary for feasibility as the solver guarantees that the unoptimized schedule will not exceed the desired memory budget.

The execution plan can either be interpreted during training, or encoded as a static computation graph. In this work, we generate a static graph $G' = (V', E')$ from the plan, which is executed by a numerical machine learning framework. See Section 6.2 for implementation details.

4.10 Cost model

To estimate the runtime of a training iteration under a rematerialization plan, we apply an additive cost model (1a), incurring cost C_i when node v_i is evaluated. Costs are determined prior to MILP construction by profiling network layers on target hardware with random inputs across a range of batch sizes and input shapes, and exclude static graph construction and input generation time. As neural network

Algorithm 1 Generate execution plan

Input: graph $G = (V, E)$, feasible (R, S, FREE)
Output: execution plan $P = (s_1, \ldots, s_k)$
Initialize $\text{REGS}[1 \ldots |V|] = -1$, $r = 0$, $P = ()$.
for $t = 1$ **to** $|V|$ **do**
  for $k = 1$ **to** $|V|$ **do**
    if $R_{t,k}$ **then**
      // Materialize v_k
      add `%r = compute v_k` to P
      $\text{REGS}[k] = r$
      $r = r + 1$
    end if
    // Free v_k *and dependencies*
    for $i \in \text{DEPS}[k] \cup \{k\}$ **do**
      if $\text{FREE}_{t,i,k}$ **then**
        add `deallocate %REGS[i]` to P
      end if
    end for
  end for
end for
return P

operations consist of dense numerical kernels such as matrix multiplication, these runtimes are low variance and largely independent of the specific input data (Jia et al., 2018a; Sivathanu et al., 2019). However, forward pass time per batch item decreases with increasing batch size due to improved data parallelism (Canziani et al., 2016), so it is important to compute costs with appropriate input dimensions.

The memory consumption of each value in the data-flow graph is computed statically as input and output sizes are known. Values are dense, multi-dimensional tensors stored at 4 byte floating point precision. The computed consumption M_i is used to construct memory constraints (2-3).

5 APPROXIMATION

Many of our benchmark problem instances are tractable to solve using off-the-shelf integer linear program solvers, with practical solve times ranging from seconds to an hour. ILP results in this paper are obtained with a 1 hour time limit on a computer with at least 24 cores. Relative to training time, *e.g.* 21 days for the BERT model (Devlin et al., 2018), solving the ILP adds less than a percent of runtime overhead.

While COTS solvers such as COIN-OR (Forrest et al., 2019) leverage methods like branch-and-bound to aggressively prune the decision space, they can take superpolynomial time in the worst-case and solving ILPs is NP-hard in general. In the worst-case, for neural network architectures with hundreds of layers, it is not feasible to solve the rematerialization problem via our ILP. An instance of the VGG16 architecture (Simonyan & Zisserman, 2014) takes seconds to solve. For DenseNet161 (Huang et al., 2017), no feasible solution was found within one day.

For many classical NP-hard problems, approximation algorithms give solutions close to optimal with polynomial runtime. We review a linear program that produces fractional solutions in polynomial time in Section 5.1. Using the fractional solutions, we present a two-phase rounding algorithm in Section 5.2 that rounds a subset of the decision variables, then finds a minimum cost, feasible setting of the remaining variables to find near-optimal integral solutions.

5.1 Relaxing integrality constraints

By relaxing integrality constraints (1f), the problem becomes trivial to solve as it is a linear program over continuous variables. It is well known that an LP is solvable in polynomial time via Karmarkar's algorithm (Karmarkar, 1984) or barrier methods (Nesterov & Nemirovskii, 1994). With relaxation $R, S, \text{FREE} \in [0, 1]$, the objective (1a) defines a lower-bound for the cost of the optimal integral solution.

Rounding is a common approach to find approximate integral solutions given the result of an LP relaxation. For example, one can achieve a $\frac{3}{4}$-approximation for MAX SAT (Yannakakis, 1994) via a simple combination of randomized rounding ($\mathbf{Pr}\left[x_i^{\text{int}} = 1\right] = x_i^*$) and deterministic rounding ($x_i^{\text{int}} = 1$ if $x_i^* \geq p$, where commonly $p = 0.5$).

We attempt to round the fractional solution R^*, S^* using these two strategies, and then apply Algorithm 1 to $R^{\text{int}}, S^{\text{int}}$. However, direct application of deterministic rounding returns infeasible results: the rounded solution violates constraints. Randomized rounding may show more promise as a single relaxed solution can be used to sample many integral solutions, some of which are hopefully feasible. Unfortunately, using randomized rounding with the LP relaxation for VGG16 at a $4\times$ smaller budget than default, we could not find a single feasible solution out of 50,000 samples.

5.2 A two-phase rounding strategy

To find feasible solutions, we introduce *two-phase rounding*, detailed in Algorithm 2. Two-phase rounding is applicable when a subset of variables can be solved in polynomial time given the remaining variables. Our approximation algorithm only rounds the checkpoint matrix S^*. Given S^*, we solve for the conditionally optimal binary computation matrix R^{int} by setting as few values to 1 as possible. Algorithm 2 begins with an all-zero matrix $R^{\text{int}} = 0$, then iteratively corrects violated correctness constraints.

Note that during any of the above steps, once we set some $R_{i,j}^{\text{int}} = 1$, the variable is never changed. Algorithm 2 corrects constraints in a particular order so that constraints that are satisfied will continue to be satisfied as other violated constraints are corrected. The matrix R^{int} generated by this

Algorithm 2 Two-phase rounding

Input: Fractional checkpoint matrix S^* from LP
Output: Binary S^{int}, R^{int}, FREE
Round S^* deterministically: $S^{\text{int}}_{t,i} \leftarrow \mathbb{1}[S^*_{t,i} > 0.5]$
$R^{\text{int}} \leftarrow \mathbf{I}_n$ thereby satisfying (8a)
while $\exists t \geq 2, i \in [n]$ such that $S^{\text{int}}_{t,i} > R^{\text{int}}_{t-1,i} + S^{\text{int}}_{t-1,i}$ *i.e.* (1c) violated **do**
  Compute v_i to materialize checkpoint: $R^{\text{int}}_{t-1,i} \leftarrow 1$
end while
while $\exists t \geq 1, (i,j) \in E$ such that $R^{\text{int}}_{t,j} > R^{\text{int}}_{t,i} + S^{\text{int}}_{t,i}$ *i.e.* (1b) violated **do**
  Compute v_i as temporary for dependency: $R^{\text{int}}_{t,i} \leftarrow 1$
end while
Evaluate FREE by simulating execution
return S^{int}, R^{int}, FREE

rounding scheme will be optimal up to the choice of S^{int} as every entry in R^{int} is set to 1 if and only if it is necessary to satisfy a constraint. In implementation, we detect and correct violations of (1b) in reverse topological order for each stage, scanning $R^{\text{int}}, S^{\text{int}}$ matrices from right to left.

5.3 Memory budget feasibility

Since we approximate S by rounding the fractional solution, $S^{\text{int}}, R^{\text{int}}$ can be infeasible by the budget constraint $U_{t,k} \leq M_{\text{budget}}$. While the fractional solution may come under the budget and two-phase rounding preserves correctness constraints, the rounding procedure makes no attempt to maintain budget feasibility. Therefore, we leave an allowance on the total memory budget constraint ($U_{t,k} \leq (1-\epsilon)M_{\text{budget}}$). We empirically find $\epsilon = 0.1$ to work well.

6 EVALUATION

In this section, we investigate the impact of tensor rematerialization on the cost and memory usage of DNN training. We study the following experimental questions: (1) *What is the trade-off between memory usage and computational overhead when using rematerialization?* (2) *Are large inputs practical with rematerialization?* and (3) *How well can we approximate the optimal rematerialization policy?*

We compare our proposed solver against baseline heuristics on representative image classification and high resolution semantic segmentation models including VGG16, VGG19, ResNet50, MobileNet, U-Net and FCN with VGG layers, and SegNet. As prior work is largely limited to linear graphs, we propose novel extensions where necessary for comparison. Results show that optimal rematerialization allows significantly lower computational overhead than baselines at all memory budgets, and lower memory usage than previously possible. As a consequence, optimal rematerialization allows training with larger input sizes than previously possible, up to 5.1× higher batch sizes on the same accelerator. Finally, we find that our two-phase rounding approximation algorithm finds near-optimal solutions in polynomial time.

6.1 Baselines and generalizations

Table 1 summarizes baseline rematerialization strategies. The nominal evaluation strategy stores all features generated during the forward pass for use during the backward pass—this is the default in frameworks such as TensorFlow. Hence, every layer is computed once. We refer to this baseline as *Checkpoint all*, an ideal approach given ample memory.

On the linear graph architectures, such as VGG16 and MobileNet (v1), we directly apply prior work from Griewank & Walther (2000) and Chen et al. (2016b), baselines referred to as *Griewank and Walther* $\log n$, *Chen et al.* $\sqrt{n}$ and *Chen et al. greedy*. To build a tradeoff curve for computation versus memory budget, we search over the segment size hyperparameter b in the greedy strategy. However, these baselines cannot be used for modern architectures with residual connections. For a fair comparison, we extend the $\sqrt{n}$ and greedy algorithms to apply to general computation graphs with residual connections or branching structure (*e.g.* ResNet50 and U-Net).

Chen et al. (2016b) suggests manually annotating good checkpointing candidates in a computation graph. For the first extensions, denoted by *AP* $\sqrt{n}$ and *AP greedy*, we automatically identify *articulation points*, or *cut vertices*, vertices that disconnect the forward pass DAG, and use these as candidates. The heuristics then select a subset of these candidates, and we work backwards from the checkpoints to identify which nodes require recomputation.

Still, some networks have few articulation points, including U-Net. We also extend heuristics by treating the original graph as a linear network, with nodes connected in topological order, again backing out the minimal recomputations from the selected checkpoints. These extensions are referred to as *Linearized* $\sqrt{n}$ and *Linearized greedy*.

Sections B.1 and B.2 provide more details on our generalizations. Note that all proposed generalizations exactly reproduce the original heuristics on linear networks.

6.2 Evaluation setup

Checkmate is implemented in Tensorflow 2.0 (Abadi et al., 2016), accepting user-defined models expressed via the high-level Keras interface. We extract the forward and backward computation graph, then construct and solve optimization problem (9) with the Gurobi mathematical programming library as an integer linear program. Finally, Checkmate translates solutions into execution plans and constructs a new static training graph. Together, these components form

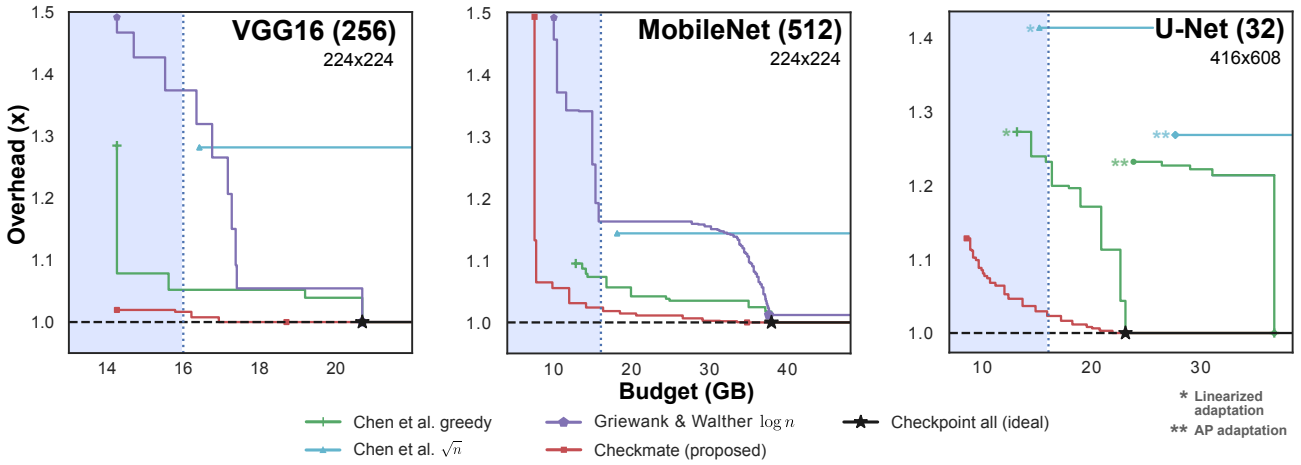

Figure 5. Computational overhead versus memory budget for (a) VGG16 image classification NN (Simonyan & Zisserman, 2014), (b) MobileNet image classification NN, and (c) the U-Net semantic segmentation NN (Ronneberger et al., 2015). Overhead is with respect to the best possible strategy without a memory restriction based on a profile-based cost model of a single NVIDIA V100 GPU. For U-Net (c), at the 16 GB V100 memory budget, we achieve a $1.20\times$ speedup over the best baseline—linearized greedy—and a $1.38\times$ speedup over the next best—linearized $\sqrt{n}$. **Takeaway:** our model- and hardware-aware solver produces in-budget solutions with the lowest overhead on linear networks (a-b), and dramatically lowers memory consumption *and* overhead on complex architectures (c).

the Checkmate system, illustrated in Figure 2.

To accelerate problem construction, decision variables R and S are expressed as lower triangular matrices, as are accounting variables U. FREE is represented as a $|V| \times |E|$ matrix. Except for our maximum batch size experiments, solutions are generated with a user-configurable time limit of 3600 seconds, though the majority of problems solve within minutes. Problems with exceptionally large batch sizes or heavily constrained memory budgets may reach this time limit while the solver attempts to prove that the problem is infeasible. The cost of a solution is measured with a profile-based cost model and compared to the (perhaps unachievable) cost with no recomputation (Section 4.10).

The feasible set of our optimal ILP formulation is a strict superset of baseline heuristics. We implement baselines as a static policy for the decision variable S and then solve for the lowest-cost recomputation schedule using a similar procedure to that described in Algorithm 2.

6.3 What is the trade-off between memory usage and computational overhead?

Figure 5 compares rematerialization strategies on VGG-16, MobileNet, and U-Net. The y-axis shows the computational overhead of checkpointing in terms of time as compared to baseline. The time is computed by profiling each individual layer of the network. The x-axis shows the total memory budget required to run each model with the specified batch size, computed for single precision training. Except for the $\sqrt{n}$ heuristics, each rematerialization algorithm has a knob to trade-off the amount of recomputation and memory usage, where a smaller memory budget leads to higher overhead.

Takeaways: For all three DNNs, Checkmate produces clearly faster execution plans as compared to algorithms proposed by Chen et al. (2016b) and Griewank & Walther (2000) – over $1.2\times$ faster than the next best on U-Net at the NVIDIA V100 memory budget. Our framework allows training a U-Net at a batch size of 32 images per GPU with less than 10% higher overhead. This would require 23 GB of memory without rematerialization, or with the original baselines without our generalizations.

6.4 Are large inputs practical with rematerialization?

The maximum batch size enabled by different rematerialization strategies is shown in Figure 6. The y-axis shows the theoretical maximum batch size we could feasibly train with bounded compute cost. This is calculated by enforcing that the total cost must be less than the cost of performing just one additional forward pass. That is, in Figure 6 the cost is at most an additional forward pass higher, *if* the specified batch size would have fit in GPU memory. We reformulate Problem (9) to maximize a batch size variable $B \in \mathbb{N}$ subject to modified memory constraints that use $B * M_i$ in place of M_i and subject to an additional cost constraint,

$$\sum_{t=1}^{n}\sum_{i=1}^{t} C_i R_{t,i} \leq 2 \sum_{v_i \in G_{\text{fwd}}} C_i + \sum_{v_i \in G_{\text{bwd}}} C_i. \qquad (10)$$

The modified integer program has quadratic constraints, and is difficult to solve. We set a time limit of one day for the experiment, but Gurobi may be unable to reach optimality within that limit. Figure 6 then provides a lower bound on

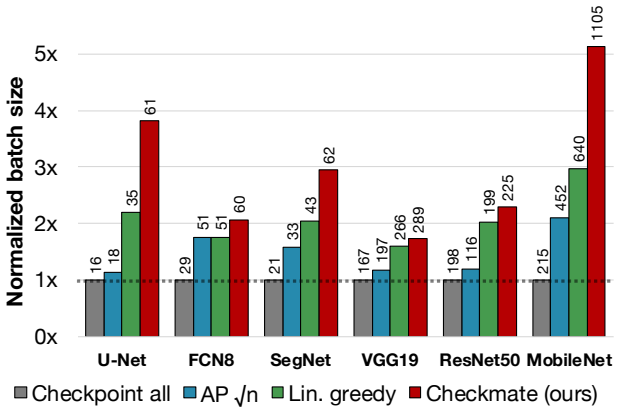

Figure 6. Maximum batch size possible on a single NVIDIA V100 GPU when using different generalized rematerialization strategies with at most a single extra forward pass. We enable increasing batch size by up to $5.1\times$ over the current practice of caching all activations (on MobileNet), and up to $1.73\times$ over the best checkpointing scheme (on U-Net).

the maximum batch size that Checkmate can achieve.

For fair comparison on the non-linear graphs used in U-Net, FCN, and ResNet, we use the AP $\sqrt{n}$ and linearized greedy baseline generalizations described in Section 6.1. Let $M_{\text{fixed}} = 2M_{\text{param}}$, as in (2) and let $M_{@1}$ be the memory a baseline strategy uses at batch size 1. The maximum baseline batch size is estimated with (11), where the minimization is taken with respect to hyperparameters, if any.

$$\max B = \left\lfloor \frac{16 \text{ GB} - M_{\text{fixed}}}{\min M_{@1} - M_{\text{fixed}}} \right\rfloor \tag{11}$$

Costs are measured in FLOPs, determined statically. U-Net, FCN8 and SegNet semantic segmentation networks use a resolution of 416×608, and classification networks ResNet50, VGG19 and MobileNet use resolution 224×224.

Takeaways: We can increase the batch size of U-Net to 61 at a high resolution, an unprecedented result. For many tasks such as semantic segmentation, where U-Net is commonly used, it is not possible to use batch sizes greater than 16, depending on resolution. This is sub-optimal for batch normalization layers, and being able to increase the batch size by $3.8\times$ (61 vs 16 for a representative resolution) is quite significant. Orthogonal approaches to achieve this include model parallelism and distributed memory batch normalization which can be significantly more difficult to implement and have high communication costs. Furthermore, for MobileNet, Checkmate allows a batch size of 1105 which is $1.73\times$ higher than the best baseline solution, a greedy heuristic, and $5.1\times$ common practice, checkpointing all activations. The same schedules can also be used to increase image resolution rather than batch size.

	Chen $\sqrt{n}$	Chen greedy	Griewank $\log n$	Two-phase LP rounding
MobileNet	$1.14\times$	$1.07\times$	$7.07\times$	**1.06**$\times$
VGG16	$1.28\times$	$1.06\times$	$1.44\times$	**1.01**$\times$
VGG19	$1.54\times$	$1.39\times$	$1.75\times$	**1.00**$\times$
U-Net	$1.27\times$	$1.23\times$	-	**1.03**$\times$
ResNet50	$1.20\times$	$1.25\times$	-	**1.05**$\times$

Table 2. Approximation ratios for baseline heuristics and our LP rounding strategy. Results are given as the geometric mean speedup of the optimal ILP across feasible budgets.

6.5 How well can we approximate the optimal rematerialization policy?

To understand how well our LP rounding strategy (Section 5) approximates the ILP, we measure the ratio $\text{COST}_{\text{approx}}/\text{COST}_{\text{opt}}$, *i.e.* the speedup of the optimal schedule, in FLOPs. As in Section 6.3, we solve each strategy at a range of memory budgets, then compute the geometric mean of the ratio across budgets. The aggregated ratio is used because some budgets are feasible via the ILP but not via the approximations. Table 6 shows results. The two-phase deterministic rounding approach has approximation factors close to optimal, at most $1.06\times$ for all tested architectures.

7 Conclusions

One of the main challenges when training large neural networks is the limited capacity of high-bandwidth memory on accelerators such as GPUs and TPUs. This has created a memory wall that limits the size of the models that can be trained. The bottleneck for state-of-the-art model development is now memory rather than data and compute availability, and we expect this trend to worsen in the future.

To address this challenge, we proposed a novel rematerialization algorithm which allows large models to be trained with limited available memory. Our method does not make the strong assumptions required in prior work, supporting general non-linear computation graphs such as residual networks and capturing the impact of non-uniform memory usage and computation cost throughout the graph with a hardware-aware, profile-guided cost model. We presented an ILP formulation for the problem, implemented the Checkmate system for optimal rematerialization in TensorFlow, and tested the proposed system on a range of neural network models. In evaluation, we find that optimal rematerialization has minimal computational overhead at a wide range of memory budgets and showed that Checkmate enables practitioners to train high-resolution models with significantly larger batch sizes. Finally, a novel two-phase rounding strategy closely approximates the optimal solver.

ACKNOWLEDGEMENTS

We would like to thank Barna Saha and Laurent El Ghaoui for guidance on approximation, Mong H. Ng for help in evaluation, and the paper and artifact reviewers for helpful suggestions. In addition to NSF CISE Expeditions Award CCF-1730628, this work is supported by gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, NVIDIA, Scotiabank, Splunk and VMware. This work is also supported by the NSF GRFP under Grant No. DGE-1752814. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. March 2016.

Beaumont, O., Herrmann, J., Pallez, G., and Shilova, A. Optimal memory-aware backpropagation of deep join networks. Research Report RR-9273, Inria, May 2019.

Briggs, P., Cooper, K. D., and Torczon, L. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pp. 311–321, New York, NY, USA, 1992.

Brock, A., Donahue, J., and Simonyan, K. Large scale GAN training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*, 2018.

Bulo, S. R., Porzi, L., and Kontschieder, P. In-place Activated BatchNorm for Memory-Optimized Training of DNNs. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5639–5647. IEEE, June 2018.

Canziani, A., Paszke, A., and Culurciello, E. An Analysis of Deep Neural Network Models for Practical Applications. May 2016. arXiv: 1605.07678.

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., and Yuille, A. L. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. June 2016a. arXiv: 1606.00915.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training Deep Nets with Sublinear Memory Cost. April 2016b. arXiv: 1604.06174.

Chen, X., Ma, H., Wan, J., Li, B., and Xia, T. Multi-view 3D Object Detection Network for Autonomous Driving. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6526–6534. IEEE, 2017.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating Long Sequences with Sparse Transformers. April 2019. arXiv: 1904.10509.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., and Salakhutdinov, R. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. January 2019. arXiv: 1901.02860.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. October 2018. arXiv: 1810.04805.

Dong, C., Loy, C. C., He, K., and Tang, X. Image super-resolution using deep convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(2):295–307, Feb 2016.

Feng, J. and Huang, D. Cutting Down Training Memory by Re-fowarding. July 2018.

Forrest, J. J., Vigerske, S., Ralphs, T., Santos, H. G., Hafer, L., Kristjansson, B., Fasano, J., Straver, E., Lubin, M., rlougee, jpgoncal1, Gassmann, H. I., and Saltzman, M. COIN-OR Branch-and-Cut solver, June 2019.

Gholami, A., Azad, A., Jin, P., Keutzer, K., and Buluc, A. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pp. 77–86. ACM, 2018.

GLPK. GNU Project - Free Software Foundation (FSF).

Gomez, A. N., Ren, M., Urtasun, R., and Grosse, R. B. The Reversible Residual Network: Backpropagation Without Storing Activations. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and

Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 2214–2224. Curran Associates, Inc., 2017.

Goodwin, D. W. and Wilken, K. D. Optimal and Near-optimal Global Register Allocation Using 0–1 Integer Programming. *Software: Practice and Experience*, 26(8): 929–965, 1996.

Griewank, A. and Walther, A. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, March 2000.

Gruslys, A., Munos, R., Danihelka, I., Lanctot, M., and Graves, A. Memory-efficient Backpropagation Through Time. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pp. 4132–4140, USA, June 2016. Curran Associates Inc.

Gueguen, L., Sergeev, A., Kadlec, B., Liu, R., and Yosinski, J. Faster Neural Networks Straight from JPEG. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 3933–3944. Curran Associates, Inc., 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Holder, L. Graph Algorithms: Applications, 2008.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Ioffe, S. and Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *International Conference on Machine Learning*, February 2015.

Jain, A., Phanishayee, A., Mars, J., Tang, L., and Pekhimenko, G. Gist: Efficient Data Encoding for Deep Neural Network Training. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pp. 776–789, Piscataway, NJ, USA, 2018. IEEE Press.

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks. In *International Conference on Machine Learning*, pp. 2274–2283, July 2018a.

Jia, Z., Zaharia, M., and Aiken, A. Beyond Data and Model Parallelism for Deep Neural Networks. *SysML Conference*, pp. 13, Feb. 2018b.

Karmarkar, N. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pp. 302–311. ACM, 1984.

Kim, J., Lee, J. K., and Lee, K. M. Accurate image super-resolution using very deep convolutional networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1646–1654, June 2016. doi: 10.1109/CVPR.2016.182.

Koes, D. R. and Goldstein, S. C. A Global Progressive Register Allocator. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pp. 204–215, New York, NY, USA, 2006. ACM. event-place: Ottawa, Ontario, Canada.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012.

Lattner, C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. RoBERTa: A Robustly Optimized BERT Pretraining Approach. July 2019. arXiv: 1907.11692.

Long, J., Shelhamer, E., and Darrell, T. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440, 2015.

Lozano, R. C., Carlsson, M., Blindell, G. H., and Schulte, C. Combinatorial Register Allocation and Instruction Scheduling. April 2018. arXiv: 1804.02452.

McCandlish, S., Kaplan, J., Amodei, D., and Team, O. D. An Empirical Model of Large-Batch Training. arXiv: 1812.06162.

Meng, C., Sun, M., Yang, J., Qiu, M., and Gu, Y. Training Deeper Models by GPU Memory Optimization on TensorFlow. pp. 8, December 2017.

Micikevicius, P. Local Memory and Register Spilling, 2011.

Nakata, I. On Compiling Algorithms for Arithmetic Expressions. *Commun. ACM*, 10(8):492–494, August 1967. ISSN 0001-0782. doi: 10.1145/363534.363549. URL http://doi.acm.org/10.1145/363534.363549.

Nesterov, Y. and Nemirovskii, A. *Interior-point polynomial algorithms in convex programming*, volume 13. Siam, 1994.

NVIDIA. NVIDIA Tesla V100 GPU Architecture, August 2017. URL https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

Olesen, J. S. Register Allocation in LLVM 3.0, November 2011.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop*, 2017.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d' Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Pohlen, T., Hermans, A., Mathias, M., and Leibe, B. Full-resolution residual networks for semantic segmentation in street scenes. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, 2017.

Punjani, M. Register Rematerialization in GCC. In *GCC Developers' Summit*, volume 2004. Citeseer, 2004.

Ronneberger, O., Fischer, P., and Brox, T. U-Net: Convolutional Networks for Biomedical Image Segmentation. In Navab, N., Hornegger, J., Wells, W. M., and Frangi, A. F. (eds.), *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, Lecture Notes in Computer Science, pp. 234–241. Springer International Publishing, 2015. ISBN 978-3-319-24574-4.

Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pp. 12–27, New York, NY, USA, 1988. ACM.

Sethi, R. Complete Register Allocation Problems. pp. 14, April 1973.

Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. September 2014. arXiv: 1409.1556.

Siskind, J. M. and Pearlmutter, B. A. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6):1288–1330, 2018a. doi: 10.1080/10556788.2018.1459621.

Siskind, J. M. and Pearlmutter, B. A. Divide-and-Conquer Checkpointing for Arbitrary Programs with No User Annotation. *Optimization Methods and Software*, 33(4-6): 1288–1330, November 2018b.

Sivathanu, M., Chugh, T., Singapuram, S. S., and Zhou, L. Astra: Exploiting Predictability to Optimize Deep Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '19*, pp. 909–923, Providence, RI, USA, 2019. ACM Press.

Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

Szegedy, C., Wei Liu, Yangqing Jia, Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, June 2015. doi: 10.1109/CVPR.2015.7298594.

Tai, Y., Yang, J., and Liu, X. Image super-resolution via deep recursive residual network. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2790–2798, July 2017. doi: 10.1109/CVPR.2017.298.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is All you Need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 5998–6008. Curran Associates, Inc., 2017.

Wu, Y. and He, K. Group Normalization. pp. 3–19, 2018.

Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500, 2017.

Yang, B., Liang, M., and Urtasun, R. HDNET: Exploiting HD Maps for 3D Object Detection. pp. 10, 2018.

Yannakakis, M. On the approximation of maximum satisfiability. *Journal of Algorithms*, 17(3):475–502, 1994.

A INTEGRALITY GAP

To understand why the partitioned variant of the MILP (Section 4.2) is faster to solve via branch-and-bound, we can measure the integrality gap for particular problem instances. The integrality gap is the maximum ratio between the optimal value of the ILP and its relaxation, defined as follows:

$$IG = \max_I \frac{\text{COST}_{int}}{\text{COST}_{frac}},$$

where COST_{int} and COST_{frac} are the optimal value the ILP and that of its relaxation, respectively. $I = (G, C, M, M_{\text{budget}})$ describes a problem instance. As our ILP is a minimization problem, $\text{COST}_{int} \geq \text{COST}_{frac}$ for all I, and $IG \geq 1$. While it is not possible to measure the ratio between the ILP and LP solutions for all problem instances, the ratio for any particular problem instance gives a lower bound on the integrality gap.

For the 8-layer linear neural network graph discussed in Section 4.2, frontier-advancement reduces the integrality gap from 21.56 to 1.18, *i.e.* the LP relaxation is significantly tighter. In branch-and-bound algorithms for ILP optimization, a subset of feasible solutions can be pruned if the LP relaxation over the subset yields an objective higher than the best integer solution found thus far. With a tight LP relaxation, this condition for pruning is often met, so fewer solutions need to be enumerated.

B GENERALIZATIONS OF PRIOR WORK

B.1 AP $\sqrt{n}$ and AP greedy

We identify Articulation Points (AP) in the undirected form of the forward pass data-flow graph as candidates for checkpointing. Articulation points are vertices that increase the number of connected components (*e.g.* disconnect) the graph if removed, and can be identified in time $O(V + E)$ via a modified DFS traversal (Holder, 2008). An articulation point v_a is a good candidate for checkpointing as subsequent vertices in the topological order have no dependencies on vertices before v_a in the order. DNN computation graphs are connected, so each intermediate tensor can be reconstructed from a single articulation point earlier in the topological order, or the input if there is no such AP. APs include the input and output nodes of residual blocks in ResNet, but not vertices inside blocks. We apply Chen's heuristics to checkpoint a subset of these candidates, then solve for the optimal recomputation plan R to restore correctness. Solving for R ensures that the dependencies of a node are in memory when it is computed.

We could find R by solving the optimization problem (9) with additional constraints on S that encode the heuristically selected checkpoints. However, as S is given, the optimization is solvable in $O(|V||E|)$ via a graph traversal per row of R that fills in entries when a needed value is not in memory by the same process described in Section 5.2.

B.2 Linearized $\sqrt{n}$ and Linearized greedy

The forward graph of the DNN $G_{\text{fwd}} = (V_{\text{fwd}}, E_{\text{fwd}})$ can be treated as a linear graph $G_{\text{lin}} = (V_{\text{fwd}}, E_{\text{lin}})$ with edges connecting consecutive vertices in a topological order:

$$E_{\text{lin}} = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{L-1}, v_L)\}$$

While G_{lin} does not properly encode data dependencies, it is a linear graph that baselines can analyze. To extend a baseline, we apply it to G_{lin}, generate checkpoint matrix S from the resulting checkpoint set, and find the optimal R as with the AP baselines.

C HARDNESS OF REMATERIALIZATION

Sethi (1973) reduced 3-SAT to a decision problem based on register allocation in straight line programs, with no recomputation permitted. Such programs can be represented by result-rooted Directed Ayclic Graphs (DAGs), with nodes corresponding to operations and edges labeled by values. In Sethi's graphs, the desired results are the roots of the DAG. If a program has no common subexpressions, *i.e.* the graph forms a tree, optimal allocation is possible via a linear time tree traversal (Nakata, 1967). However, Sethi's reduction shows a register allocation decision problem in the general case—whether a result-rooted DAG can be computed with fewer than k registers without recomputation—is NP-complete.

The decision problem characterizes computation of a DAG as a sequence of four possible moves of stones, or registers, on the nodes of the graph, analogous to statements discussed in Section 4.9. The valid moves are to (1) place a register at a leaf, computing it, or (2) pick up a register from a node. Also, if there are registers at all children of a node x, then it is valid to (3) place a register at x, computing it, or (4) move a stone to x from one of the children of x, computing x. The register allocation problem reduces to the following no-overhead rematerialization decision problem (RP-DEC):

Definition C.1. (RP-DEC): Given result-terminated data-flow DAG $G = (V, E)$ corresponding to a program, with unit cost to compute each node and unit memory for the results of each node, does there exist an execution plan that evaluates the leaf (terminal) node $t \in V$ with maximum memory usage b at cost at most $|V|$?

RP-DEC is decidable by solving the memory-constrained form of Problem 1 with sufficient stages, then checking if the returned execution plan has cost at most $|V|$. RP-DEC closely resembles Sethi's decision problem, differing only in subtleties. The register allocation DAG is rooted at the desired result t whereas a data-flow graph terminates at the

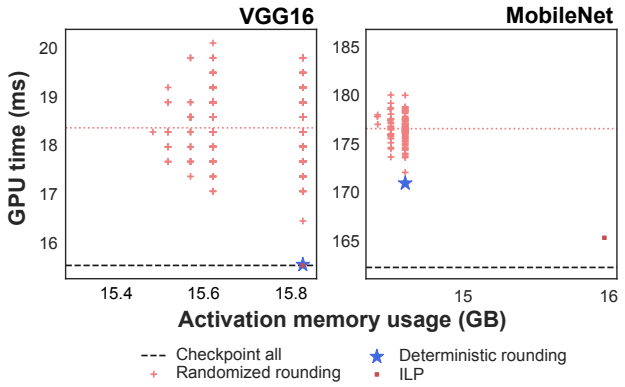

Figure 7. Comparison of the two-phase LP rounding approximation with randomized rounding of S^* and deterministic rounding of S^* on different models. We compare memory usage and computational cost (objective), in milliseconds according to profile-based cost model. The average of the randomized rounding costs is shown as a dotted line.

result. Second, register-based computations can be in place, *e.g.* a summation $a + b$ may be written to the same location as either of the operands. In neural network computation graphs, we cannot perform all computations in place, so we did not make this assumption. To reduce Sethi's decision problem to RP-DEC, given result-rooted DAG G, construct result-terminated G' by reversing all edges. Then, if Sethi's instance allows for at most k registers, allow for a memory budget of $b = k + 1$ bytes: one byte to temporarily write outputs of operations that would have been written in place.

Despite hardness of register allocation, Goodwin & Wilken (1996) observe that a 0-1 integer program for optimal allocation under an instruction schedule has empirical complexity $O(n^{2.5})$, polynomial in the number of constraints. Similarly, Section 6 shows that the frontier-advancing, constrained optimization problem (9) is tractable for many networks.

D Comparison of Approximations

In Section 5, we discussed an approximation strategy based on rounding the LP relaxation, evaluated with deterministic rounding in Section 6.5. Figure 7 compares schedules produced by our proposed two-phase rounding strategy when the S^* matrix from the LP relaxation is rounded with a randomized and a deterministic approach. While two-phase randomized rounding of S^* offers a range of feasible solutions, two-phase deterministic rounding produces consistently lower cost schedules. While appropriate for VGG16, for MobileNet, our budget allowance $\epsilon = 0.1$ is overly conservative as schedules use less memory than the 16 GB budget. A search procedure over $\epsilon \in [0, 1]$ could be used to produce more efficient schedules.

E Artifact Reproducibility Instructions

Checkmate is a Python package that computes memory-efficient schedules for evaluating neural network dataflow graphs created by the backpropagation algorithm. To save memory, the package deletes and rematerializes intermediate values via recomputation. The schedule with minimum recomputation for a given memory budget is chosen by solving an integer linear program. Find the software for the artifact and documentation at https://github.com/parasj/checkmate/tree/mlsys20_artifact.

E.1 Artifact check-list (meta-information)

- **Algorithm**: Integer linear programming (Gurobi 9.0)
- **Model**: Code included in setup and public, including neural network architectures VGG16, VGG19, U-Net, MobileNet, SegNet, FCN, ResNet50. Trained weights not required.
- **Run-time environment**: Ubuntu 18.04.3 LTS
- **Hardware**: 2x Intel E5-2670 CPUs, 256GB DDR4 RAM
- **Execution**: Runtime varies, 1m to 24hr
- **Metrics**: Computational overhead (slowdown based on cost model), maximum supported batch size
- **Output**: Plot of memory budget vs overhead. Console output of maximum supported batch size
- **Experiments**: Commands provided in README.md for Gurobi installation and running experiment Python scripts
- **How much disk space required?**: 1 GB
- **Publicly available?**: Yes. https://github.com/parasj/checkmate/tree/mlsys20_artifact. Archived at https://zenodo.org/badge/latestdoi/209406827.
- **Code licenses**: Apache 2.0 licensed