TRANSCENDING RUNTIME-MEMORY TRADEOFFS IN CHECKPOINTING BY BEING FUSION AWARE

Shangdi Yu [1 2] **Horace He** [3]

ABSTRACT

Gradient checkpointing is an optimization that reduces the memory footprint by re-computing some operations instead of saving their activations. Previous works on checkpointing have viewed this as a tradeoff between peak memory and performance. However, we argue that this framing does not account for a key aspect of modern deep learning systems – operator fusion. In this work, we demonstrate that with a fusion aware checkpointing algorithm, we can transcend the runtime-memory tradeoffs of traditional checkpointing and improve both memory **and** runtime simultaneously. We evaluate our algorithm on a wide range of standard neural network models as well as some novel patterns. We achieve a geomean of 12% throughput improvement over an existing compiled baseline, and the maximum batch size that can be attained is up to 1.75 times larger on standard models. In novel patterns, we achieve up to a 10x improvement, with by a 5x reduction in peak memory.

1 INTRODUCTION

Deep neural networks (DNNs) have been shown effective to solve problem in many fields such as computer vision (He et al., 2016; Krizhevsky et al., 2017), natural language processing (Vaswani et al., 2017; Devlin et al., 2019; Child et al., 2019; Radford et al., 2019), and recommendation systems (Yi et al., 2019; Zhang et al., 2019). As deep neural networks (DNNs) get larger, improving execution time and reducing the peak memory usage have become priorities. One standard approach for reducing peak memory usage is gradient checkpointing. Standard autograd (with backpropagation) requires saving significant amount of data (commonly known as "activations") to compute the backwards pass. Gradient checkpointing is a technique that reduces activations saved by avoiding saving some activations, and recomputing them during the backwards pass.

As we reduce our peak memory by performing more computation, previous works (Chen et al., 2016; Siskind & Pearlmutter, 2018; Feng & Huang, 2018; Griewank & Walther, 2000; Lanctot et al., 2022; Kirisame et al., 2021; Jain et al., 2020; Kumar et al., 2019; Beaumont et al., 2021) on checkpointing have primarily viewed checkpointing as a technique for trading off a slower runtime for less peak memory.

In this paper, we argue that this framing misses a key component of modern deep learning systems and hardware — operator fusion. ***Operator fusion*** (or kernel fusion) is a key optimization in many state-of-the-art DNN execution frameworks, such as PyTorch (Paszke et al., 2019) or TVM (Chen et al., 2018). It combines multiple GPU kernels into a single GPU kernel, which allows for memory-bandwidth costs to be eliminated. In the presence of operator fusion, it is no longer true that extra computation necessarily leads to a slower runtime. In fact, in many cases, recomputation can actually lead to **improved** runtime when combined with operator fusion.

We present a checkpointing strategy that takes operator fusion into account and demonstrate that it can lead to improvements in **both** runtime and memory over no checkpointing. To see how this is possible, consider the following function, which is also in Figure 1.

```
def f(x):
    return x.tanh().tanh()
```

In naive automatic differentiation, the forward pass computes the outputs of f and saves all tensors that will be used in the backward pass. The backward pass reads those saved tensors and computes the gradient of x. In total, the forward pass would save both $x.tanh()$ and $x.tanh().tanh()$ for the backward pass, which incurs a total of two memory write from the forward pass and two memory reads from the backward pass.

However, we can reduce these memory accesses by simply reading x in the backwards pass and recomputing $x.tanh()$ and $x.tanh().tanh()$ while fusing them into other operations in the backwards pass. Since reading and writing to

[1]EECS, MIT, Cambridge, MA, USA [2]Work done while at Meta Inc. [3]Meta Inc., Menlo Park, CA, USA. Correspondence to: Horace He <chilli@fb.com>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023.

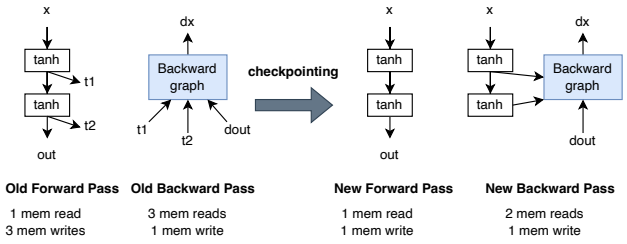

Figure 1. The forward and backward pass of naive automatic differentiation and naive checkpointing.

the global memory takes much more cycles than computing tanh (He, 2022), gradient checkpointing will improve the execution time. Since we are saving less activations, peak memory will also improve. Although in this simple example, the optimal strategy is to recompute everything in the forwards pass, in practice, recomputing everything (i.e. naive gradient checkpointing) is often not beneficial or correct. For example, some computations like matrix multiplications can be too computationally expensive, even if fused, while other computations might not be fusible at all. As a results, we need more sophisticated methods to choose the checkpointing strategy. More motivating examples are given in Section 2.

In Section 3, we describe our algorithm and show that it gives the optimal memory bandwidth under some assumptions. In Section 4, we show our experimental results. In Section 5, we discuss some related works.

In summary, our contributions are:

- We demonstrate that in some cases, gradient checkpointing can improve both peak memory and runtime.
- We design a fusion-aware checkpointing algorithm that leverages this fact to perform well on generic models, and prove that this algorithm is optimal with some assumptions on the cost model.
- We implement our algorithm in the PyTorch framework, and evaluated our algorithm on various DNN models, as well as some novel patterns. We achieve a geomean of 12% throughput improvement over an existing compiled baseline, and the maximum batch size that can be attained is up to 1.75 times larger on standard models. In novel patterns, we achieve up to a 10x improvement, with by a 5x reduction in peak memory.

2 PRELIMINARIES

2.1 Runtime of Neural Networks

The execution time of a DNN model can be decomposed into three components: ***compute time***, ***memory time***, and everything else. Compute time is time spent on a GPU computing floating point operations, and the efficiency of it is commonly measured by number of floating point operations per second (FLOPS). Memory time is the time spent on transferring tensors *within* a GPU, e.g. from CUDA global memory to CUDA shared memory, and the efficiency of it is commonly measured by bytes per second. If an operation spends more time on memory-bandwidth than computing, it is a ***memory-bandwidth-bound*** operation. Such operators are common on modern GPUs. Here we give an example comparison of compute time versus memory time on an NVIDIA A100 GPU (NVIDIA, 2020), which has 1.5 terabytes per second of global memory bandwidth, and 19.5 teraFLOPS on vector operations. So the **time to read from and write back a 4-byte floating point number to the global memory is 104 times slower than doing a unary operation on it** (He, 2022). Memory-bandwidth cost is important because data movement can be a key bottleneck during training. For example, Ivanov et al. (2021) found that over a third of the runtime in a BERT training iteration is spent in memory-bandwidth-bound operators.

Fortunately, the memory time of such operations can be reduced by operation fusion, which performs several computations at once without reading or writing intermediate values to global memory. Operator fusion is an commonly used optimization in deep learning and there exists compilers that can perform fusions such as NVFuser (Sarofeen et al., 2022), TorchInductor (Ansel, 2022), and XLA (Sabne, 2020). Due to the memory-bandwidth-bound nature of many operations, fusion leads to some surprising consequences. For example, computing two operators (under the presence of fusion) may take nearly the same time as computing one of them. For example, on an A100 GPU, computing a single cosine operation takes 62.36ms while computing two fused cosine operations takes 62.91ms. Since computation can be cheap under the presence of fusion, gradient checkpointing may actually **decrease runtime**, despite performing more operations.

2.2 Gradient Checkpointing

Gradient checkpointing is a standard technique for reducing the peak memory of automatic differentiation (AD). Gradient checkpointing works by recomputing the intermediate values of a neural network (which would ordinarily be stored in the forwards pass) during the backwards pass.

This is typically framed as a tradeoff between runtime and

memory usage (Beaumont et al., 2020; Chen et al., 2016; Feng & Huang, 2018; Kusumoto et al., 2019). However, this framing misses a crucial aspect of modern deep learning frameworks — operator fusion. In the presence of operator fusion, we are unable to model runtime solely through each operator's runtime individually.

To see an example of where this simplified performance model goes wrong, let's look at an example of an activation function. For example, let's benchmark the cumulative time of the forwards and backwards pass of the GeLU (Hendrycks & Gimpel, 2016) activation function. GeLU(x) = $x\Phi(x)$, where Φ the standard Gaussian cumulative distribution function. The benchmark results are summarized in Table 1. With an input of size 2^{25}, running without any checkpointing results in a runtime of 1.33ms. However, if we apply naive gradient checkpointing (i.e. recomputing the forwards pass in the backwards pass), we achieve a runtime of 0.5ms!

This may be surprising, as we're recomputing fifteen operators in the backwards pass. However, if we look at the memory accesses, we see that without gradient checkpointing, the forwards pass reads a single tensor and writes out five, while the backwards pass reads six tensors and outputs one. This is a total of 13 memory accesses. On the other hand, with gradient checkpointing, the forwards pass reads a single tensor and writes out one, while the backwards pass reads two tensors and writes out one. This is a total of 5 memory accesses.

Note that in this case, checkpointing with fusion is the best of both worlds — not only does it achieve the lowest runtime, we also save less activations than without checkpointing. We note that this is an optimization very similar to the manual combination of gradient checkpointing and fusion of GeLU that is performed manually in the Megatron-LM codebase (Shoeybi et al., 2019). This motivates us to look at gradient checkpointing not as a tradeoff between memory and runtime, but as an optimization to improve **both** memory and runtime.

Table 1. Results of running the forward and backward pass of the GeLU activation function with an input of size 2^{25}. "Ops" is the number of operations in the forward and backward pass computation. "Mem" is the number of reading and writing memory accesses to the DRAM. "Acts." is short for activations. "Checkp." is for whether the naive gradient checkpointing is applied.

Checkp.	Acts. (MB)	Ops	Fusion Mem	Fusion Time	No Fusion Mem	No Fusion Time
✓	671	47	5	**0.5**	98	10.1 ms
X	134	34	13	1.33	74	7.3 ms

2.3 Examples of Minimizing Runtime through Smarter Gradient Checkpointing

Although naive gradient checkpointing can sometimes reduce runtime when combined with operator fusion, it's often not so easy. There are many situations where neither naive AD nor naive gradient checkpointing will provide the optimal result.

Partially re-compute forward pass. Sometimes, we might only want to re-compute some nodes of the forward graph instead of all nodes. Consider this function:

```
def f1(a, b, c, d):
    x = a + b + c + d
    return x.cos().cos()
```

Naive AD would end up saving x and $x.cos()$, resulting in 2 writes from the forwards pass and 2 reads in the backwards pass. On the other hand, naive gradient checkpointing would save the 4 inputs, resulting in no writes from the forwards pass (as the inputs are already materialized) but 4 reads in the backwards pass.

However, neither of these naive approaches are optimal. Instead, we should save x, which is sufficient to compute the backwards pass, and results in merely one write from the forwards pass and one read in the backwards pass.

In practice, this results in a 21.6% performance improvement over either of the naive approaches. This aligns with the total number of memory accesses – the naive approaches require 11 memory accesses while the optimal approach requires 9.

Avoiding re-computation of some operators. Moreover, there are some operators that should not be recomputed. For example, nodes that are computationally intensive (matmuls) or involve randomness. Consider this function, which is similar to dropout:

```
def f2(x):
  rand_like = torch.rand_like(x)
  mask = rand_like < 0.5
  return x * mask
```

Here we need to make sure that we don't recompute `rand_like`. One possibility is that we simply save the output of random operations (i.e. `rand_like`). In this case, however, we miss an optimization opportunity. Instead of saving `rand_like` (a FloatTensor), we could save `mask` (a BoolTensor), and save 3 bytes per element. In this function, performing this optimization saves 25% of memory and 25% of runtime.

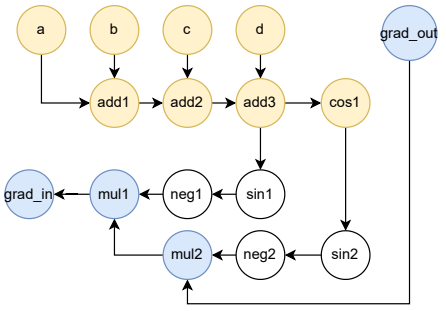

Figure 2. The data-flow graph of naive gradient checkpointing for $f1$. Yellow nodes are nodes in the forward pass. Blue nodes are nodes that must be in the backward pass. The white nodes can either be computed in the forward pass or backward pass.

2.4 Data-flow Graph

A computation graph or ***data-flow graph*** (Reed et al., 2022) $G = (V, E)$ is a DAG. Each node is an operation that has input arguments and output values; and each directed edge (u, v) represents a dependency of v on u, i.e. u is one of v's arguments. We call v a ***user*** of node u. The nodes without any incoming edge are called ***placeholder nodes***, and these are the inputs to the graph. We give an example of the data-flow graph of naive gradient checkpointing for $f1$ in Figure 2. Nodes ***`grad_out`*** and ***`grad_in`*** are the input and output of the backward pass, respectively. Nodes a, b, c, d are the inputs to the forward pass. Yellow nodes are nodes in the forward pass. For example, node `add3` produce $x = a + b + c + d$. Blue nodes are nodes that must be in the backward pass. The white nodes can either be computed in the forward pass or backward pass. For example, `sin1` can be computed either in the forward pass or backward pass because it does not require `grad_out`.

2.5 Minimum s-t Cut Problem

The minimum s-t cut problem (Cormen et al., 2022) takes a graph $G' = (V', E')$ and two nodes $s, t \in V'$ as inputs. Each edge in E' has a capacity. The problem finds a partition of the nodes V' into S ($s \in S, t \notin S$) and $T = V' \setminus S$ that minimizes the sum of the capacity of edges going from S to T. Let S be the ***source partition*** and T be the ***sink partition***. This partition is call an ***s-t cut***, and a ***cut-set*** of a cut is the set of edges that connect the source part of the cut to the sink part. The ***capacity*** of an s-t cut is the sum of the capacities of the edges in its cut-set. So the minimum s-t cut problem determine S and T such that the capacity of the s-t cut is minimal.

3 Minimum Cut Checkpointing

As shown in the previous section, in some cases, gradient checkpointing can improve performance when combined with operator fusion. However, in many cases, applying naive gradient checkpointing is insufficient to achieve runtime improvements. Moreover, there are there are many diverse situations in which we might wish to apply this kind of optimization. This motivates us to come up with a principled approach for checkpointing that allows us to cover all of these cases.

In this section we describe our minimum cut (min-cut) checkpointing algorithm, which intelligently chooses which nodes to save from the forwards pass, and thus which nodes to recompute in the backwards pass. This allows us to improve on **both** memory saved as well as execution time. We also show that under some assumptions, our algorithm optimally minimizes the memory-bandwidth of reading and writing to the activations. In Section 3.7, we show that our algorithm can also be extended to compute recomputation between any two fusion groups.

3.1 Assumptions

First, we note that although our algorithm generally reduces both memory usage and execution time (compared to no checkpointing), our primary goal is to reduce the **execution time** of our model. In addition, there are several assumptions we make in our cost model:

1. All operators can be classified into memory-bandwidth bound operators and compute-bound operators.
2. All memory-bandwidth bound operators are free to compute when fused with another operator. However, if a memory-bandwidth bound operator is not fusible with some of its users and must be materialized in the backward pass, it is not free to re-compute anymore, because it must be materialized anyway and re-computing it can cause extra reads and writes in the backward pass. So we ***ban*** operators that are not fusible with some of its users in the backward pass from re-computation.
3. When minimizing execution time, we never wish to recompute a compute-bound operator, such as matrix multiplication, convolution, and layer norm. These nodes are ***banned*** from recomputation.
4. Chains of fusible operators are still fusible.

We find that these assumptions are generally true. He (2022) shows that we need to fuse 50 multiplications into one kernel before there is any appreciable difference in runtime. Moreover, with compute-bound operators, operator fusion has a negligible affect on their runtime, and so computing them multiple times is rarely worth it for reducing runtime (Sabne, 2020; Rotem et al., 2018).

Now we show that re-computing a node not fusible with some of its users in the backward pass does not save any

memory, so we should avoid recomputing them by banning them as described above. Let the set of nodes computable from the forwards pass be V_f.

Lemma 3.1. *Recomputing a node i not fusible with some of its users in $V \setminus V_f$ cannot avoid the cost of reading and writing i.*

Proof. Suppose i is not fusible with its user j and $j \in V \setminus V_f$. If i is not re-computed, we have to pay for reading and writing i between the forward and backward pass because j must be in the backward pass.

If we re-compute i, we still have to pay for reading and writing i in the backward pass because i and j are not fusible. □

3.2 Problem Setup

We will start with the data-flow graph of the backwards pass with naive checkpointing, which we call the **joint graph**. Note that this graph is nearly a superset of both the forwards and backwards graphs from naive AD – the only exception are the nodes in the forwards pass which do not need to save any activations for the backwards pass.

We are allowed to save any valid set of nodes computable from the forwards pass in order to minimize the runtime of our backwards pass. We will call this set V_{saved} and the set of nodes computable from the forwards pass V_f. There are three requirements for V_{saved} to be ***valid***. The first is that `grad_in` must be computable from $V_{saved} \cup$ `grad_out`. The second is that V_{saved} must be computable from the inputs to the forwards pass. Finally, the banned nodes cannot be recomputed. V_{saved} becomes the input to our backwards pass, and also corresponds to what is generally called ***saved activations***.

Depending on the nodes saved, this formulation may recover both naive AD as well as naive gradient checkpointing. For example, in Fig 2, if we saved `add3` and `cos1`, we would be saving the same activations as naive AD (and avoid any recomputation). On the other hand, if we saved `a`, `b`, `c`, and `d`, then we would be saving the same activations as naive checkpointing. However, as discussed in Section 2.3, for this case the optimal solution is to only save `add3`.

We note that choosing the inputs to our backwards pass uniquely defines not only our backwards pass, but also our forwards pass. Our backwards pass is all nodes computable from V_{saved}, while our forwards pass simply outputs V_{saved}.

3.3 Minimum Cut

Our algorithm is outlined in Algorithm 1. The input is the joint graph $G = (V, E)$, and the set of nodes computable from the forwards pass $V_f \subset V$. We want to use $V_{\text{saved}} \subseteq V_f$ as inputs to the backward pass and we want to optimize the memory bandwidth while improving the execution time. We will create an auxiliary graph G' such that solving the min-cut problem on G' gives us the optimal V_{saved}.

The weighted auxiliary graph G' will encode the data dependency of G, the memory cost of each node, and the restriction of banning certain nodes from recomputation as described in Section 3.1. To avoid confusion, we will call V in G ***nodes***, and V' in G' ***vertices***. The graph G' has source and sink vertices s, t. For each node $i \in V_f$, we also create two vertices i_{in} and i_{out}. Let these two nodes be i's ***in-vertex*** and ***out-vertex***, respectively. For each node $i \in V \setminus V_f$, we only create i_{in}. In the min-cut partition, the partition with the source vertex s is called the ***source partition***, and the other partition is called the ***sink partition***. If i_{in} is in the source partition and i_{out} is in the sink partition, then i is in the ***cut set nodes***. The cut set nodes are returned as V_{saved}.

The edges of G' are created like below.

- Nodes not in V_f have an edge from their in-vertices to the sink with infinite capacity. Since these nodes cannot be saved because they are not computable from the inputs to the forward pass, we add these infinite-capacity edges to avoid putting their in-vertices into the source partition.
- The input nodes to the forward pass have an edge between their in-vertices and source with infinite capacity. So these nodes must be in the source partition.
- If a node is in V_f and is banned from recomputation, we add an edge between the source and its in-vertex with infinite capacity. So these nodes must be in the source partition.
- For each node $i \in V_f$, we create an edge with infinite capacity from its out-vertex to its users' in-vertices such that we don't have edges between vertices of different nodes across the cut. We also create an edge between i_{in} and i_{out}. *The capacity of this edge is the cost of materializing this node*, which is the cost of backward pass reading it and the potentially cost of forward pass writing it. The edge capacity is determined like below and the pseudocode is presented in Algorithm 2.

Edge capacity. We first use a heuristic to re-weight the nodes' capacities, which we found to improve the performance in practice and will be described in more details in the next subsection. Let this re-weighted node weight be w. We let the edge capacity be w if the node must be materialized in the forward pass. Let these **materialized nodes** be V_m. Nodes are materialized if they are input nodes or outputs to the forward pass or itself or any of its users is

not fusible. In those cases, the node must be materialized, and we only need to pay for an additional reading cost if we choose to save this activation and let the backward pass read it. Otherwise, for nodes that are not materialized in the forward pass, we let the edge capacity be $2w$ because we need to pay additional reading and writing costs.

Lemma 3.2. *The cut set of the minimum s-t cut only contains edges connecting an in-vertex and an out-vertex, i.e. the (i_{in}, i_{out}) edges.*

Proof. There always exists a finite-capacity cut in G'. We can let all the vertices corresponding to nodes in $V \setminus V_f$ and all out-vertices incident to them be in the sink partition with the sink vertex, and all other vertices be in the source partition. This corresponds to not recomputing anything from the forward pass.

Now we show all edges across this cut have finite capacity. From Algorithm 1, we can see there are four types of edges with infinite capacity, and we will show that none of them can exist across the cut. For the first type (Line 6), vertices for nodes in $V \setminus V_f$ have infinite-capacity edges to sink, but all those nodes are already in the sink partition with the sink vertex. For the second and third type (Line 9 and 11), sink partition does not contain in-vertices of nodes in V_f. The only other infinite-capacity edges (Line 14) are between out-vertices of nodes and in-vertices of their users. If we have an edge $(i_{\text{out}}, j_{\text{in}})$ across the partitions, then j_{in} is in the sink partition and j must be in $V \setminus V_f$ because only out-vertices can be in the sink partition without corresponding to nodes in V_f. However, by construction i_{out}, which is incident to j_{in} must be in the sink partition as well. This a contradiction, so we cannot have such edge.

Since there exists a finite-capacity cut, the minimum cut must have finite capacity. The only finite-capacity edges are the $(i_{\text{in}}, i_{\text{out}})$ edges. So the cut set of the minimum s-t cut only contains the $(i_{\text{in}}, i_{\text{out}})$ edges.

□

By Lemma 3.2, the edges across the source partition and the sink partition are the $(i_{\text{in}}, i_{\text{out}})$ edges. So we can let V_{saved} to be the cut set nodes, i.e. the set of i such that i_{in} is in source partition and i_{out} is in sink partition. The capacity of any finite s-t cut with cut set nodes V_{saved} is $\sum_{i \in V_m \cap V_{\text{saved}}} \text{weight}(i) + \sum_{i \in V_{\text{saved}} \setminus V_m} 2 * \text{weight}(i)$.

3.4 Heuristics to improve performance

There are cases when the simplifications we described in Section 3.1 don't hold true, so we add the following heuristics to make our algorithm work better in practice.

First, we re-weight the nodes to bias towards saving nodes closer to the backwards pass, primarily as a tiebreak mechanism. If node A and B are the same sizes, we can compute B from A, but we only need node B, there's no reason to save A, although our min-cut algorithm would naively weight them the same.

Algorithm 1 Min-cut Checkpointing

```
1:  input: G = (V, E), V_f
2:  V' = {s, t} ∪ {i_in, i_out | i ∈ V_f} ∪ {i_in | i ∈ V \ V_f},
    E' = {}
3:  G' = (V', E')
4:  for i ∈ V do
5:      if i ∉ V_f then
6:          G'.edge( i_in, t, ∞)
7:          Continue
8:      if i is an input to forward pass then
9:          G'.edge(s, i_in, ∞)
10:     if ban-recomputation(i) then
11:         G'.edge(s, i_in, ∞)
12:     G'.edge(i_in, i_out, get-capacity(i))
13:     for j ∈ i.users do
14:         G'.edge(i_out, j_in, ∞)
15: V_saved = minimum-cut(G', s, t)
16: Return V_saved
```

Algorithm 2 Get Capacity

```
function is_materialized(i)
    if i is input then
        Return True
    Return not (node and all node.users are fusible)
function get_capacity(i)
    w = re-weight(i.capacity)
    if is_materialized(i) then
        Return w
    else
        Return w * 2
```

Second, in addition to the operators mentioned in Section 3.1, we ban recomputation of the following operations: reduction operations where the output of the reduction is significantly smaller (we choose 4 times smaller in our experiments), and operations that involve randomness.

In the case that the shapes are static, we only need to run this algorithm once at the beginning of training. If shapes change throughout training, however, it's possible that a different cut would be optimal for different shapes. Luckily, we observe that the vast majority of dynamic shapes do not affect the min-cut significantly. This is since most of the decisions the algorithm makes involve simply saving less activations, activations with a smaller data type, or activations that are drastically smaller (such as the output of a reduction). Thus, we can simply substitute an arbitrary shape and obtain most of the benefits (in practice, we simply peek at the underlying shape of the first inputs we see). Note that all possible cuts are "valid", they only differ in their performance characteristics.

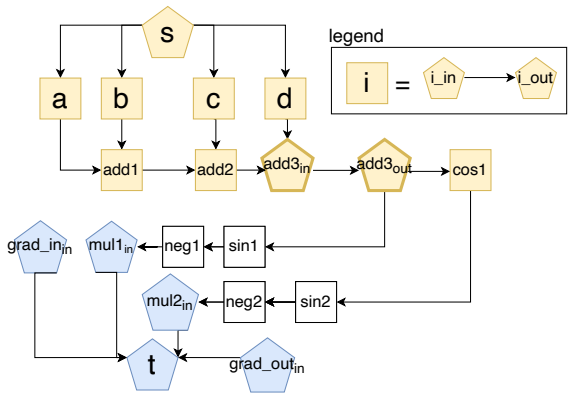

Figure 3. The auxiliary graph G' of the data-flow graph G in Figure 2. The pentagons are the vertices in G' and the squares represent an in-vertex, an out-vertex, and the edge between them. The yellow vertices and white vertices correspond to nodes in V_f. The cut set node of the min-cut in G' is $\{\texttt{add3}\}$ and the vertices of this node are bolded.

3.5 Example

In Figure 3, we give the auxiliary graph G' of the data-glow graph G in Figure 2. The pentagons are the vertices in G' and the squares represent an in-vertex, an out-vertex, and the edge between them. The yellow vertices and white vertices correspond to nodes in V_f. The cut set node of the min-cut in G' is $\{\texttt{add3}\}$ and the vertices of this node are bolded. For the optimal strategy where only `add3` is saved, we would put $\{\texttt{source}, \texttt{add3}_{\text{in}}\} \cup \{i_{\text{in}}, i_{\text{out}}\}$ for $i \in [\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{add1}, \texttt{add2}]$ in the source partition, and other nodes in the sink partition.

The naive AD corresponds to putting $\{\texttt{source}\} \cup \{i_{\text{in}}\}$ for $i \in [\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}]$ in the source partition, and other nodes in the sink partition.

The naive gradient checkpointing corresponds to putting $\{\texttt{source}, \texttt{add3}_{\text{in}}, \texttt{cos1}_{\text{in}}\} \cup \{i_{\text{in}}, i_{\text{out}}\}$ for $i \in [\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{add1}, \texttt{add2}]$ in the source partition, and other nodes in the sink partition.

3.6 Optimality

In this section, we show the optimality of our algorithm.

Lemma 3.3. *For any valid set V_{saved}, we can find a finite s-t cut such that V_{saved} is the cut set nodes.*

Proof. Let V_B be the set of nodes used to compute `grad_in` from $V_{saved} \cup \texttt{grad\_out}$. V_B must exist because V_{saved} is valid. Let the in-vertices of V_{saved} be in the source partition and the out-vertices of V_{saved} be in the sink partition. Let the vertices corresponds to V_B be in the sink partition. Let the rest of the vertices be in the source partition.

Now we show that the cut set of this cut is $\{(i_{\text{in}}, i_{\text{out}}) | i \in V_{\text{saved}}\}$. The infinite-capacity edge (i_{in}, t) added on Line 6 cannot be in the cut set because $i \notin V_f$, so i must be in V_B and i_{in} is in the sink partition.

The infinite-capacity edge (s, i_{in}) added on Line 9 cannot be in the cut set because a forward pass input node has no incoming edge and so it cannot be computed from other nodes and thus cannot be in V_B.

Since a valid V_{saved} requires that banned nodes are not recomputed, the banned nodes cannot be part of V_B, so the edges added on Line 11 cannot be in the cut set.

We also cannot have edge $(i_{\text{in}}, i_{\text{out}})$ on Line 12 from the source partition to sink partition for $i \notin V_{\text{saved}}$ because vertices of nodes not in V_{saved} are all in the same partition by construction.

Finally, we cannot have edge $(i_{\text{out}}, j_{\text{in}})$ from the source partition to sink partition for any i, j because if i_{out} is in the source partition and j_{in} is in the sink partition, nodes i and j are both not in V_{saved}. Moreover, node j should be in V_B. However, since i is an input of j, i should also be in V_B and i_{out} should be in the sink partition, which is a contradiction.

□

Lemma 3.4. *For any finite s-t cut, its cut set nodes V_{saved} is valid.*

Proof. First, we show `grad_in` is computable from $V_{saved} \cup \texttt{grad\_out}$. If `grad_in` is not computable, then there must exist a path from a forward pass input to `grad_in` that contains no node in V_{saved}. This means there's also a path from s to t without any vertex corresponding to nodes in V_{saved}. However, this is impossible because there must be some finite-capacity edge going from the source partition to the sink partition in the path.

Second, we show V_{saved} is computable from the inputs to the forwards pass. By construction, $V_{saved} \subseteq V_f$, and V_f is the set of nodes computable from the forward pass by definition.

Finally, we show V_{saved} would not recompute banned node, because we added the infinite-capacity edges on Line 11. Since the cut has finite capacity, the in-vertices of any banned node must be in the source partition. If a banned node i is in V_{saved}, it is not recomputed. If node i is not in V_{saved}, then the i_{out} is also in the source partition with i_{in}. Since the sink t is in the sink vertex, there must exist some finite-capacity edge $(j_{\text{in}}, j_{\text{out}})$ such that $j \in V_{\text{saved}}$ on every path from i_{out} to t. So i does not need to recomputed because `grad_in` can be computed from j without i.

□

Lemma 3.5. *Let V_{saved} be the cut set nodes of some cut. Without the re-weighting heuristics, the capacity of the cut*

is the same as the cost of writings $V_{saved} \setminus V_m$ in the forward pass plus the cost of reading V_{saved} in the backward pass.

Proof. The capacity of any finite s-t cut with cut set nodes V_{saved} is $\sum_{i \in V_m \cap V_{\text{saved}}} \text{weight}(i) + \sum_{i \in V_{\text{saved}} \setminus V_m} 2 * \text{weight}(i)$ by construction of our graph.

The cost of writing non-materialized nodes in V_{saved} in the forward pass is $W_f = \sum_{i \in V_{\text{saved}} \setminus V_m} \text{weight}(i)$. The cost of reading V_{saved} in the backward pass is $W_b = \sum_{i \in V_{\text{saved}}} \text{weight}(i)$.

So the total cost is $\sum_{i \in V_{\text{saved}} \setminus V_m} 2 * \text{weight}(i) + \sum_{i \in V_{\text{saved}} \cap V_m} \text{weight}(i)$, which is the same as capacity of the cut. □

Theorem 3.6. *The checkpointing strategy found by the min-cut algorithm is the checkpointing strategy with the lowest memory cost in our cost model among all valid subsets of forward pass nodes. Specifically, it minimizes the cost of reads and writes between the forward and backward passes plus the cost of extra reads and writes in the backwards pass caused by some nodes being recomputed in the backward pass.*

Proof. We've shown that we can create a valid set of saved activations from any finite-capacity partition (Lemma 3.4), and we can also create a finite-capacity partition from any valid set of saved activations (Lemma 3.3). So there is a 1-to-1 relationship between valid checkpointing plans in G and finite-capacity s-t cut in G'. Also by Lemma 3.5, the cut capacity is same as the cost of writings $V_{\text{saved}} \setminus V_m$ in the forward pass plus the cost of reading V_{saved} in the backward pass. Moreover, by Lemma 3.2, the minimum s-t cut must have finite capacity. So the partitioning plan found by the min-cut algorithm is the partitioning plan with the lowest cost of writings $V_{\text{saved}} \setminus V_m$ in the forward pass plus the cost of reading V_{saved} in the backward pass.

The forward pass has three disjoint sets of nodes to write: 1) the non-fusible nodes, 2) the fusible output nodes, and 3) the nodes in V_{saved} excluding the first two types and the inputs to the forward pass. The inputs do not need to be written because they are already in the DRAM before the forward pass. We see that the third set is exactly $V_{\text{saved}} \setminus V_m$, and the first two sets have to be written in any checkpointing strategy. So minimizing cost of writings $V_{\text{saved}} \setminus V_m$ in the forward pass plus the cost of reading V_{saved} in the backward pass minimizes the memory cost of reading and writing between the forward and backward pass.

Extra reads and writes in the backwards pass can only be caused by recomputing nodes materialized in the backward pass, but we've banned those nodes from recomputation, so the total cost is equal to the cost of reads and writes between the forward and backward passes. We've shown in Lemma 3.1 that recomputing these nodes does not decrease the total cost.

Thus, the partitioning plan found by the min-cut algorithm is the partitioning plan with the lowest memory cost. □

3.7 Checkpointing between Fusion Groups

Let the operators within a fused kernel be a ***fusion group***. We note that this algorithm can be extended to a heuristic for computing a checkpointing plan between any two fusion groups. The heuristic works on pairs of fusion groups $(\mathcal{A}, \mathcal{B})$ that are ***touching***, meaning fusion group $\mathcal{A}$'s output is directly used by fusion group $\mathcal{B}$. Instead of saving activation from the forward pass, it can be used to save activation from $\mathcal{A}$ and reduce the reading and writing cost between fusion groups by recomputing some fusible nodes in $\mathcal{A}$. Similarly, we use a min-cut algorithm to determine which nodes should be saved. Let G be the joint graph of $\mathcal{A}$ and $\mathcal{B}$, and let V_f be the nodes in $\mathcal{A}$. One slight change to make is that if a node in $\mathcal{A}$ is used by any node not in $\mathcal{A}$ and $\mathcal{B}$, it is considered to be in V_m because it must be materialized for other nodes to read.

To see that re-computation between fusion groups are helpful, we give an example in Figure 4. In the figure, the white nodes are input and output nodes; the green nodes are fusible nodes; and the red nodes are non-fusible nodes. The orange boxes are the fusion groups formed. The left graph is the original graph, and we see that fusion group 1 has 1 input and 3 outputs while fusion group 2 has 3 inputs and 1 output. This means that there are 4 memory reads and 4 memory writes in total. In the right graph, we recompute b and c from fusion group 1 in fusion group 2. Now, fusion group 1 has 1 input and 1 output while fusion group 2 has 2 inputs and 1 output. This means that there are 3 memory reads and 2 memory writes in total. We *reduce 2 writes and 1 read*, which means less memory bandwidth. If the green nodes are all cheap computation nodes like point-wise operations, we also reduce execution time.

In this example, the whole fusion group 1 is recomputed in fusion group 2, which is the optimal transformation when all nodes have the same input and output sizes, but this is not always the case. If node b is a reduce operator like SUM and a is a very large tensor, then the optimal solution is to only recompute c in fusion group 2.

4 EVALUATION

4.1 Evaluation Setup

Our algorithm is implemented in PyTorch (Paszke et al., 2019) on top of AOTAutograd (Horace He, 2021). The dataflow graphs we work with are torch.fx graphs (Reed et al., 2022), which are produced by AOTAutograd. To perform

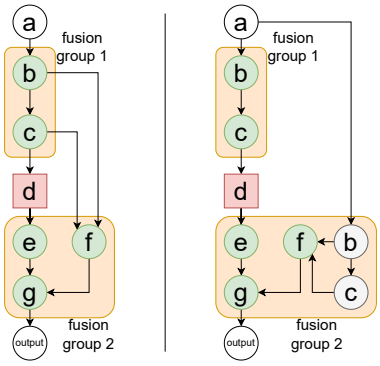

Figure 4. An example of data-flow graph and recomputation of b and c.

Table 2. Benchmark models and the maximum batch size that can be achieved by each method. Table column 'full' represents `full-checkpoint` and 'none' represents `no-checkpoint`.

dataset	full	min-cut	none
densenet121 (Huang et al., 2017)	2880	512	384
hf_Albert (Lan et al., 2020)	1088	64	48
hf_GPT2 (Radford et al., 2019)	480	56	32
mobilenet_v2 (Sandler et al., 2018)	5504	768	512
resnet18 (He et al., 2016)	5504	2560	1536
timm_efficientnet (Tan & Le, 2019)	5120	768	512
timm_regnet (Xu et al., 2022)	5120	768	512
timm_resnest (Zhang et al., 2020)	4096	896	768
timm_vovnet (Lee & Park, 2020)	4352	768	512
vgg16 (Simonyan & Zisserman, 2014)	4096	512	512

operator fusion, we leverage compilers built into PyTorch – specifically TorchInductor (Ansel, 2022). We used the push-relabel algorithm in NetworkX (Hagberg et al., 2008) to compute the minimum cut.

We evaluate our algorithm on an AWS p4d.24xlarge machine with 8 A100-40GB GPUs, although all of our benchmarks run with only a single GPU. For each experiment, we use the median of the results of 50 runs.

4.2 Throughput

We evaluate our methods on a variety of models using TorchBench (Constable et al., 2020). As our approach improves both memory and runtime, we evaluate the peak throughput we can achieve with each method, varying the batch size. For example, if a method has no latency improvements for a fixed batch size, it may still lead to throughput improvements by allowing us to run with a bigger batch size and make more effective use of our hardware. We run all models in automatic mixed precision and in CUDA.

We measure 3 different settings.

- ***`no-checkpoint`***: Models are run **with** a compiler but **no** checkpointing.
- ***`full-checkpoint`***: Models are run **with** a compiler and a standard checkpointing setup, where we choose only to save the inputs at every layer.
- ***`min-cut`***: Models are run **with** a compiler and our checkpointing approach.

Note that **the same compiler** is used for all methods, so the performance differences among them can be attributed to our checkpointing scheme.

We present the results in Figure 5. Across all models benchmarked, `min-cut` results in improved throughput compared to both `no-checkpoint` and `full-checkpoint`.

Compared to `no-checkpoint`, `min-cut` generally improves for two reason. The first is that we're simply reducing the memory bandwidth, as presented above. The second is that we're able to reduce the number of activations saved, thus allowing us to increase our batch size.

However, the throughput of `full-checkpoint` shows that simply increasing batch size through the standard checkpointing approach is not sufficient to improve throughput. Although `full-checkpoint` is able to drastically reduce peak memory and run with batch sizes significantly beyond what `min-cut` or `no-checkpoint` can, it comes at the cost of recomputing the entire forwards pass again. This additional recomputation results in the lowest throughput on all of the models tested.

Zooming in on some particular models, we see a wide variety in choices of what's being rematerialized. On `hf_Albert`, for example, there is a reimplementation of GeLU, and we automatically perform the optimization previously described in Section 2. On `timm_regnet`, there are several dtype conversions (such as from float16 to float32) where we are able to avoid saving both a float16 and a float32 version of an activation.

4.3 Latency

To verify that our performance improvements are not solely coming from an increased batch size, we also run `min-cut` with the same batch size that `no-checkpoint` is run with. This allows us to verify that our approach not only improves **throughput** but also **latency**.

We see that for the majority of models we see that `min-cut-nc-bs` has improved performance over `no-checkpoint`, with a geomean improvement of 10%. Thus, even just from the perspective of latency, `min-cut` improves upon `no-checkpoint`. And as we see from Table 2, `min-cut` increases the max batch size we are able to run with.

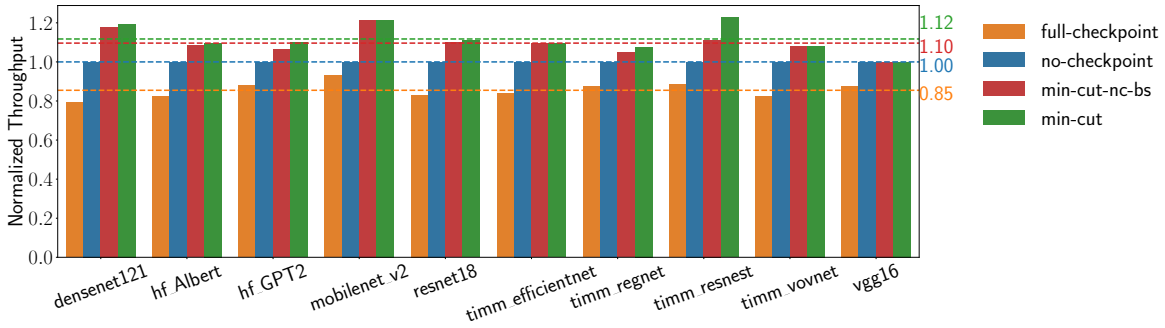

Figure 5. Normalized throughput of all methods. The throughput is normalized with respect to the `no-checkpoint` method. A higher bar means more throughput. `min-cut-nc-bs` is the same implementation as `min-cut` but uses the same batch size as the maximum batch size of `no-checkpoint`. The dotted lines are the geometric means of the methods respectively.

4.4 Runtime of our Algorithm

Although the theoretical complexity of min-cut is $O(|V|^2\sqrt{|E|})$, the runtime in practice is much faster than that.

For example, on our largest graph, `hf_GPT2`, with over 5000 nodes, our algorithm finishes in 0.48 seconds. This is despite the fact that we use NetworkX, which is written completely in Python. Preliminary experiments with a C++ implementation of min-cut suggest that we could get another order of magnitude reduction of runtime by simply porting the algorithm into C++.

4.5 Benchmark on Non-Standard Models

Although our algorithm finds speedups from standard models, in many cases, frameworks like PyTorch hand-write what tensors are saved and what tensors are recomputed. For example, with the GeLU operator in Section 2.2, PyTorch has an hand-written derivative formula for it that minimizes the memory saved [1].

One of the advantages of our algorithm is that it rediscovers many of these hand-written rules automatically. However, we can also see the benefits of our approach when applied to unconventional architectures. With unconventional architectures, there are often many more optimizations for smart checkpointing that would be hand-optimized if the architecture was standard.

For example, if we apply our algorithm to Daulton et al. (2020), a bayesian optimization model with a novel acquisition function, we see much larger speedups and memory savings. The results are summarized in Table 3. For `no-checkpoint` we see execution times of 1.13 seconds with a memory usage of 5.3 GB. However, when applying our algorithm, we achieve an execution time of 0.158 seconds with a peak memory usage of 0.946 GB.

The reason for such a drastic performance and memory improvement is the existence of large intermediate values. Naive autograd will save these values, forcing the compiler to load and store to them. However, our algorithm will automatically avoid saving large intermediates when they can be cheaply computed from smaller values. Thus, we store 5.2 GB of activations without our algorithm, and only 0.83 GB with our algorithm (the rest of the peak memory usage is from the parameters).

Table 3. Results of running the forward and backward pass of Daulton et al. (2020). The best memory and time are bolded.

Method	Peak Mem. (GB)	Time (sec)
`min-cut`	**0.946**	**0.158**
`no-checkpoint`	5.3	1.13

If this were a standard model, then it's likely PyTorch would have provided a custom operator and derivative formula that avoids saving this large intermediate value. However, due to the novel nature of Daulton et al. (2020), our approach is able to discover the optimization, demonstrating one of the advantages of compiler-based approaches.

5 Related Work

There has been many works on checkpointing and intermediate value recomputation in register allocation (Chaitin et al., 1981; Briggs et al., 1992; Punjani, 2004), reverse-mode automatic differentiation (AD) (Griewank, 1994; Griewank & Walther, 1997; 2000; Hascoet & Pascual, 2013; Siskind & Pearlmutter, 2018; Moses et al., 2021) and in deep learning (Beaumont et al., 2020; Chen et al., 2016; Feng & Huang, 2018; Kusumoto et al., 2019).

In register allocation, an allocator may recompute values if there are not enough registers for assignment and doing so is cheaper than storing the value and then loading it back from the memory. 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).

Reverse-mode AD can produce a large "tape" to store in-

[1] https://github.com/pytorch/pytorch/blob/master/tools/autograd/derivatives.yaml#L1919

termediate values and this leads to large memory footprints. Checkpointing methods are used to recompute segments of the tape to reduce the number of values saved. Recently, Moses et al. (2021) proposes an automatic differentiation tool called Enzyme to generate gradients of LLVM code. They also leverage a min-cut algorithm to determine what values they should read from cache as opposed to recomputing. However, Enzyme operates at a substantially lower level of representation from our work (LLVM IR as opposed to an operator graph), and so has a substantially different set of performance characteristics and concerns. For example, Enzyme is not able to take advantage of a fusing compiler like XLA or TorchInductor.

Chen et al. (2016) extends the checkpointing methods in reverse-mode AD to deep learning and divides the data-flow graphs into segments for recomputation during backpropagation. Chen et al. (2016) assumes that all nodes in the graph have the same cost and that the forward pass has a linear data flow graph. These assumptions limits the algorithm's ability to reflect the true computation. Kumar et al. (2019) considers the problem where given a computation graph as an input, construct a schedule that minimizes peak memory via tree decomposition. Jain et al. (2020) formalizes the problem of trading-off DNN training time and memory requirements as the tensor rematerialization optimization problem, and solves for the optimal solution using a mixed integer linear program or approximates using a linear program. Zheng et al. (2020) reduces the GPU memory footprint used for training by recomputing the feature maps of the attention and RNN layers. Kirisame et al. (2021) proposed a greedy online heuristic algorithm for checkpointing. It enables training under a limited memory budget. Beaumont et al. (2021) combines rematerialization and offloading to optimize training throughput of linearized DNNs under memory constraints. Zhang et al. (2022) used intermediate data recomputation to optimize graph neural networks.

Our checkpointing algorithm differs from prior work because our primary goal is to improve execution time while reducing the memory footprint, while these works focus on managing the execution vs. memory footprint tradeoff. Moreover, these works do not take operator fusion into consideration. As we present in Section 2, this insight is what allows us to avoid these tradeoffs and strictly improve both memory and runtime. Although these works may be able to speed up training overall by enabling a larger batch size, to the best of our knowledge, we are the first work to demonstrate that performance can be improved **even with the same batch size**.

6 Conclusion

We demonstrate that with a fusion-aware gradient checkpointing algorithm, we can achieve the best of both worlds and improve both memory and runtime. More generally, we argue that optimizations on deep learning models need to take into account the practical details of the hardware and frameworks.

References

Ansel, J. Torchinductor: a pytorch-native compiler with define-by-run ir and symbolic shapes. 2022. URL https://dev-discuss.pytorch.org/t/torchinductor-a-pytorch-native-compiler-with-define-by-run-ir-and-symbolic-shapes/747.

Beaumont, O., Herrmann, J., Pallez, G., and Shilova, A. Optimal memory-aware backpropagation of deep join networks. *Philosophical Transactions of the Royal Society A*, 378(2166):20190049, 2020.

Beaumont, O., Eyraud-Dubois, L., and Shilova, A. Efficient combination of rematerialization and offloading for training dnns. *Advances in Neural Information Processing Systems*, 34:23844–23857, 2021.

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.

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, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

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

Constable, W., Zhao, X., Bittorf, V., Christoffersen, E., Robie, T., Han, E., Wu, P., Korovaiko, N., Ansel, J., Reblitz-Richardson, O., and Chintala, S. TorchBench: A collection of open source benchmarks for PyTorch

performance and usability evaluation, 9 2020. URL https://github.com/pytorch/benchmark.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2022.

Daulton, S., Balandat, M., and Bakshy, E. Differentiable expected hypervolume improvement for parallel multi-objective bayesian optimization. *Advances in Neural Information Processing Systems*, 33:9851–9864, 2020.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Feng, J. and Huang, D. Cutting down training memory by re-fowarding. 2018.

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. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1, 04 1994. doi: 10.1080/10556789208805505.

Griewank, A. and Walther, A. Treeverse: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Software*, 26:200–0, 1997.

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 (TOMS)*, 26(1):19–45, 2000.

Hagberg, A., Swart, P., and S Chult, D. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

Hascoet, L. and Pascual, V. The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3), May 2013. ISSN 0098-3500. doi: 10.1145/2450153.2450158. URL https://doi.org/10.1145/2450153.2450158.

He, H. Making deep learning go brrrr from first principles. 2022. URL https://horace.io/brrr_intro.html.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Hendrycks, D. and Gimpel, K. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Horace He, R. Z. functorch: Jax-like composable function transforms for pytorch. https://github.com/pytorch/functorch, 2021.

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.

Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., and Hoefler, T. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Keutzer, K., Stoica, I., and Gonzalez, J. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *MLSys*, 2020. URL https://proceedings.mlsys.org/book/320.pdf.

Kirisame, M., Lyubomirsky, S., Haan, A., Brennan, J., He, M., Roesch, J., Chen, T., and Tatlock, Z. Dynamic tensor rematerialization. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Vfs_2RnOD0H.

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. *Communications of the ACM*, 60(6):84–90, 2017.

Kumar, R., Purohit, M., Svitkina, Z., Vee, E., and Wang, J. Efficient rematerialization for deep networks. *Advances in Neural Information Processing Systems*, 32, 2019.

Kusumoto, M., Inoue, T., Watanabe, G., Akiba, T., and Koyama, M. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. *Advances in Neural Information Processing Systems*, 32, 2019.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. ALBERT: A lite BERT for self-supervised

learning of language representations. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=H1eA7AEtvS.

Lanctot, M., Gruslys, A., Danihelka, I., and Munos, R. Memory-efficient backpropagation through time, 2022. US Patent 11,256,990.

Lee, Y. and Park, J. Centermask: Real-time anchor-free instance segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 13906–13915, 2020.

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

Moses, W. S., Churavy, V., Paehler, L., Hückelheim, J., Narayanan, S. H. K., Schanen, M., and Doerfert, J. Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2021.

NVIDIA. Nvidia tesla v100 gpu architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

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

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

Reed, J., DeVito, Z., He, H., Ussery, A., and Ansel, J. torch.fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems*, 4:638–651, 2022.

Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

Sabne, A. Xla : Compiling machine learning for peak performance, 2020.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

Sarofeen, C., Bialecki, P., Jiang, J., Stephano, K., Kozuki, M., Vaidya, N., and Bekman, S. Introducing nvfuser, a deep learning compiler for pytorch. 2022. URL https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

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, 2018.

Tan, M. and Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pp. 6105–6114. PMLR, 2019.

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.

Xu, J., Pan, Y., Pan, X., Hoi, S., Yi, Z., and Xu, Z. Regnet: self-regulated network for image classification. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

Yi, X., Yang, J., Hong, L., Cheng, D. Z., Heldt, L., Kumthekar, A., Zhao, Z., Wei, L., and Chi, E. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *Proceedings of the 13th ACM Conference on Recommender Systems*, pp. 269–277, 2019.

Zhang, H., Wu, C., Zhang, Z., Zhu, Y., Lin, H., Zhang, Z., Sun, Y., He, T., Mueller, J., Manmatha, R., Li, M., and Smola, A. Resnest: Split-attention networks, 2020.

Zhang, H., Yu, Z., Dai, G., Huang, G., Ding, Y., Xie, Y., and Wang, Y. Understanding gnn computational graph: A coordinated computation, io, and memory perspective. In Marculescu, D., Chi, Y., and Wu, C.

(eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 467–484, 2022. URL https://proceedings.mlsys.org/paper/2022/file/9a1158154dfa42caddbd0694a4e9bdc8-Paper.pdf.

Zhang, S., Yao, L., Sun, A., and Tay, Y. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)*, 52(1):1–38, 2019.

Zheng, B., Vijaykumar, N., and Pekhimenko, G. Echo: Compiler-based gpu memory footprint reduction for lstm rnn training. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1089–1102. IEEE, 2020.