Echo: Compiler-based GPU Memory Footprint Reduction for LSTM RNN Training

Bojian Zheng[†,⋆], Nandita Vijaykumar[†,§], Gennady Pekhimenko[†,⋆]
[†]*University of Toronto,* [⋆]*Vector Institute,* [§]*Intel*
{bojian, nandita, pekhimenko}@cs.toronto.edu

***Abstract*—The Long-Short-Term-Memory Recurrent Neural Networks (LSTM RNNs) are a popular class of machine learning models for analyzing sequential data. Their training on modern GPUs, however, is limited by the GPU memory capacity. Our profiling results of the LSTM RNN-based Neural Machine Translation (NMT) model reveal that feature maps of the attention and RNN layers form the memory bottleneck, and runtime is unevenly distributed across different layers when training on GPUs. Based on these two observations, we propose to recompute the feature maps of the attention and RNN layers rather than stashing them persistently in the GPU memory.**

While the idea of feature map recomputation has been considered before, existing solutions fail to deliver satisfactory footprint reduction, as they do not address two key challenges. For each feature map recomputation to be efficient, its effect on (1) the total memory footprint, and (2) the total execution time has to be carefully estimated. To this end, we propose *Echo*, a new compiler-based optimization scheme that addresses the first challenge with a practical mechanism that estimates the memory benefits of recomputation over the entire computation graph, and the second challenge by non-conservatively estimating the recomputation runtime overhead leveraging layer specifics. *Echo* reduces the GPU memory footprint *automatically and transparently* without any changes required to the training source code, and is effective for models beyond LSTM RNNs.

We evaluate *Echo* on numerous state-of-the-art machine learning workloads, including NMT, DeepSpeech2, Transformer, and ResNet, on real systems with modern GPUs and observe footprint reduction ratios of 1.89x on average and 3.13x maximum. Such reduction can be converted into faster training with a larger batch size, savings in GPU energy consumption (e.g., training with one GPU as fast as with four), and/or an increase in the maximum number of layers under the same GPU memory budget. *Echo* is open-sourced as a part of the MXNet 2.0 framework.[1]

***Index Terms*—DNN Training, GPU Memory Footprint Reduction, LSTM RNN**

I. Introduction

LSTM [1] RNNs form an important class of machine learning models for analyzing sequential data, having applications in language modeling [2], [3], machine translation [4]–[7], and speech recognition [8], [9]. In these tasks, LSTM RNNs are trained over large amounts of data samples (e.g., sentences or audio files) to capture their inherent temporal dependencies. Despite their importance, LSTM RNN training tends to be less efficient on modern GPUs compared to other types of deep neural networks (DNNs) such as Convolutional Neural Networks (CNNs) [10], [11]. One of the main reasons for such inefficiency is the high GPU memory consumption of LSTM RNNs that limits the maximum training batch size, which, in turn, limits the GPU compute utilization because of the small amount of available data parallelism [12].

There have been numerous works [13]–[16] that propose techniques for memory footprint reduction in DNNs, but these works, unfortunately, have limited applicability for LSTM RNN training. Specifically, prior works that propose efficient compression techniques for inference (e.g., [13], [14]) focus on weights rather than feature maps (which consume the majority of the overall memory in DNN training [15], [16]). Prior works such as vDNN [15] and Gist [16] that attempt to reduce footprint in CNN training cannot be directly applied to LSTM RNNs as they either lead to (i) high runtime overhead for many small vector layers used in LSTM RNNs, or (ii) limited applicability, as LSTM RNNs use $\tanh$/$\mathrm{sigmoid}$, rather than ReLU activations, resulting in almost no opportunities for the data encodings proposed in *Gist* [16].

To better understand the reasons that lead to the high GPU memory consumption in LSTM RNNs, we perform a detailed breakdown analysis of the GPU memory consumption (and also complimentary runtime analysis) during the training of the state-of-the-art LSTM RNN-based NMT model [4], [7], and observe that (i) the feature maps of the attention and RNN layers consume most of the GPU memory (*feature maps* are the data entries saved in the forward pass to compute the gradients during the backward pass), and (ii) the runtime is unevenly distributed across different layers (fully-connected layers dominate the runtime while other layers are relatively lightweight). From these two observations, we adopt the idea of *selective recomputation* [17]–[21], where we can leverage the low computational cost of non-fully-connected layers to *recompute* their feature maps during the backward pass, rather than stashing them in the GPU memory.

Although the idea of feature map recomputation has been explored before in prior works [17]–[19], they fail to deliver satisfactory footprint reduction in the case of LSTM RNN training and other state-of-the-art training workloads (as we will show in Sections III-D and VI). These previous proposals are ineffective in the LSTM RNN context as a result of not addressing two important challenges:

(1) *Accurately estimating footprint reduction.* While recomputation obviates the need to store feature maps of some layers

Proceeding of the 47th Annual International Symposium on Computer Architecture, Worldwide, 2020.

[1] https://issues.apache.org/jira/projects/MXNET/issues/MXNET-1450

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00092

(or group of layers), it needs to *additionally* stash some data entries that are needed to recompute these (group of) layers as new feature maps. Hence, a practical recomputation strategy should involve the comparison between the feature maps that are released and the ones that are newly allocated, but such a comparison is far from being trivial and is overlooked by prior works [17]. First, when making a decision on whether an operator should be recomputed, we should focus not only on the storage allocations that are *local* to this operator, but also on the *global effects* of these allocations, and take into account any potential *reuse* across different operators within the same computation graph. Second, we need a practical mechanism to estimate the recomputation benefits over the entire graph. Naïve implementation has combinatorial runtime complexity, which can be impractical in the LSTM RNN context given that the number of the operator nodes in the graph is usually huge (e.g., around 15.9K in large NMT models [22]).

(2) *Non-conservatively estimating runtime overhead.* Recomputation always comes with runtime overhead (as feature maps have to be recomputed), and hence potential targets for recomputation have to be carefully selected. Naïve implementations simply exclude any compute-heavy layers (e.g., convolutions, fully-connected layers) to keep the recomputation overhead low [17]. Such an approach is too conservative and leads to limited applicability in the LSTM RNN context. We, however, notice that if *layer specifics* are taken into account, certain layers that are otherwise filtered out can be amenable to recomputation with low overhead. For example, the fully-connected layers do not need recomputation as their gradients' computation does not need their outputs. Other examples of layer specifics include binarization [16] for ReLU activations and dropout layers [23]. Those layers require special handling so that we would not miss opportunities for footprint reduction and still avoid significant runtime overhead.

To effectively address these challenges and enable practical recomputation in LSTM RNN training, we propose *Echo*, a new compiler-based optimization scheme. *Echo* employs two key ideas to achieve this goal. To address the first challenge, *Echo* makes footprint reduction estimation practical by partitioning the whole computation graph into smaller subgraphs to restrict the scope (and hence reduce the complexity) of the footprint reduction estimation. Compute-heavy layers form the natural boundaries for partition, since they are not recomputed and therefore out of the estimation scope. *Echo* then analyzes each small subgraph *independently* and makes accurate footprint reduction estimation for recomputation. To address the second challenge, *Echo* infers the data dependencies of the gradient operators. *Only if* the gradient computation requires the forward operators' outputs will the forward operators' runtime be added as a part of the recomputation overhead estimate.

Our major contributions can be summarized as follows:

(1) We present a detailed breakdown and analysis of how the GPU memory is consumed and where the runtime is spent in NMT training. Our profiling reveals that the feature maps of the attention and RNN layers form the memory bottleneck and the runtime is unevenly distributed across different layers, which motivates us to adopt the idea of selective recomputation to reduce the GPU memory footprint.

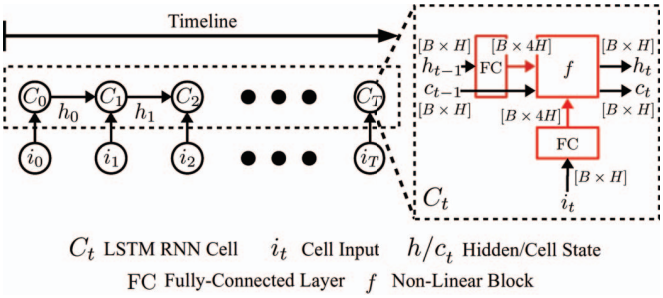


Fig. 1: Left: A single-layer LSTM RNN that scans through an input sequence. Right: Zoom-in view of one LSTM cell. Both diagrams are simplified for clarity.

(2) We address the key challenges in making selective recomputation practical by carefully estimating the footprint reduction and runtime overhead, therefore significantly outperforming prior works in both aspects.

(3) We implement our ideas in a new graph optimization pass, *Echo*, that is open-sourced in MXNet [24] NNVM [25], a state-of-the-art machine learning framework graph compiler. *Echo* reduces the GPU memory footprint *transparently and automatically* for numerous machine learning models without any changes to the training source code (even beyond LSTM RNNs). As Section VI will show, we additionally implement hand-tuned CUDA kernels that perform recomputation more efficiently but those kernels take us several weeks to develop for every specific model, whereas *Echo* is effective for all the models we have examined in a fully automatic way.

(4) We evaluate *Echo* in a state-of-the-art machine learning framework (MXNet [24]) on four state-of-the-art machine learning models [12], [26] used for machine translation (NMT [4], [7], Transformer [27]), speech recognition (DeepSpeech2 [28]), and image classification (ResNet [29]), and observe GPU memory footprint reduction ratio of $3.13\times$, $1.56\times$, $1.59\times$, and $2.13\times$ correspondingly. On the NMT model, we demonstrate that this reduction can be converted into training to the same quality with a larger batch size $1.28\times$ faster or training with one GPU as fast as with four, and on Transformer and ResNet models, we further show that our approach can help increase the maximum number of layers by $1.83\times$ and $4.0\times$ respectively while using the same GPU memory budget.

II. Background

This section gives a short overview of LSTM RNNs and their applications for machine translation tasks. For simplicity, we hide the algorithmic details that are not relevant to this work and focus on the tensor shape transformations across different layers that matter most in memory allocations. **Bold text** in this section will be used as examples in Section IV.

Figure 1 shows a simplified view of a single-layer LSTM RNN that reads through an input sequence $i_{1\sim T}$, where the annotations in square brackets denote the tensor shapes. The layer has T LSTM cells $C_{1\sim T}$, where T is the input sequence length (e.g., in the context of machine translation, T denotes

the number of words per sentence). Each cell C_t receives three inputs from two directions: (1) from the input i_t of the current time step, and (2) from the hidden and cell state of the previous cell. All inputs are of dimension $[B \times H]$, where B is the batch size and H is the hidden dimension. Both i_t and h_t need to go through a fully-connected layer, defined by Equation (1):

$$Y = XW^T + b, W : [4H \times H], b : [4H] \quad (1)$$

where X, Y, W, b are input, output, weight, and bias respectively. The weight and bias are shared across the timeline.

After their hidden dimension has become $4\times$ larger by Equation 1, i_t and h_t, together with c_t, enter the non-linear block f that consists of slicing and element-wise operations. The output of the LSTM cell is the hidden and cell state of the current time step, both of which are of dimension $[B \times H]$. **The sum of the non-linear block's input sizes $[9 \times B \times H]$ is greater than its output sizes $[B \times H]$, which will become an example in Section IV-A (Figure 7).**

The NMT model [4], [7] is the state-of-the-art LSTM RNN-based model used for machine translation. It has three major blocks, namely the encoder, decoder, and attention (Figure 2). In the encoder, source sentences of the training dataset are batched into a tensor of shape $[B \times T]$. In the embedding layer, each word in a sentence is encoded into a hidden state of dimension H. The result is sent to the LSTM RNN as an input (see Figure 1). The hidden states of the LSTM cell at all time steps (each of which is of shape $[B \times H]$) are concatenated together into the source hidden state ($[B \times T \times H]$). The encoder passes its internal hidden states to the decoder, which decodes the target sentences one word at a time into a hidden state h_t ($[B \times H]$), also known as a *query*.

The query and encoder hidden state are given to the attention layer, where they go through the following procedures to generate the attention hidden state a_t (see Figure 2):

❶ A scoring function compares the query with the encoder hidden state, generating the attention scores ($[B \times H]$), which is used to determine the attention weights α_{ts} ($[B \times H]$). **The encoder hidden state H_s is reused across all the time steps, which will become an example in Section IV-A (Figure 8).**

❷ A context vector c_t ($[B \times H]$) is computed as the attention weights-weighted average of the encoder hidden state.

❸ The query and context vector are concatenated together to generate the attention hidden state a_t ($[B \times H]$), which is sent to the next decoder time step for the next word in sequence.

This process continues until the maximum sequence length is reached. The decoded sequence is then sent to the output layer to evaluate the training loss. In training, the loss is further propagated back through the network to compute the gradients that are used to update the model weights. By the nature of the backpropagation algorithm [30], some data entries, denoted as feature maps, have to be stashed in memory during the forward pass to compute the gradients [16].

III. Motivation

A. Why does GPU memory footprint matter?

There are two major benefits of GPU memory footprint reduction. First comes from boosting the training performance

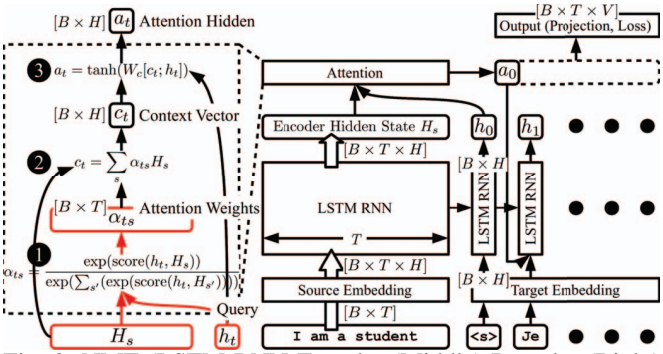
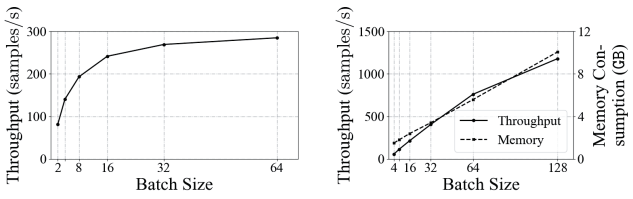


Fig. 2: NMT (LSTM RNN Encoder (Middle)-Decoder (Right) with Attention (Left))

by using larger training batch size,[2] and second from allowing to train wider and deeper models with the same GPU resources.

Increase Training Throughput with Larger Batch Size. We compare the training throughput between ResNet-50 [29] (CNN-based model used for image classification) and NMT [4], [7] (LSTM RNN-based model) with respect to their training batch size. Figure 3a shows the correlation between training throughput (measured as samples/second) and batch size for ResNet-50 (detailed methodology in Section VI-A). We notice that the training throughput saturates as the batch size increases. Our previous work [12] on benchmarking DNN Training reveals that the reason is because the GPU compute units have been almost fully utilized (starting from the batch size of 32) and therefore further increasing the batch size yields little benefit on the training throughput. However, the story is different in LSTM RNNs. Figure 3b shows a similar graph for NMT. We observe that the training throughput increases linearly with the batch size, but such increase stops when the model hits the GPU memory capacity wall on a modern 11 GB RTX 2080 Ti GPU [31] at the batch size of 128, and cannot increase any further. From the comparison, we draw the conclusion that, **in LSTM RNN-based model, performance is limited by the GPU memory capacity**, and hence this justifies why footprint reduction techniques can further increase the training throughput for such models (as we will show in Section VI-B).



(a) ResNet-50 (b) NMT

Fig. 3: (a) Training Throughput of ResNet-50 versus Batch Size (b) Training Throughput and GPU Memory Usage of NMT versus Batch Size (using one RTX 2080 Ti GPU)

Wider and Deeper Models. Although models such as ResNet might not always be able to benefit from the footprint reduction to achieve performance gains, they can still benefit indirectly by becoming **wider and deeper** while using the same

[2]Although one might argue that large training batch size might hurt convergence, in Section VI-B we show actual training curves that increase convergence speed to the same quality when training with larger batch size.

GPU memory budget. For example, data encoding approach such as the one used in our prior work *Gist* can increase the maximum number of layers in ResNet from 851 to 1202 with a batch size of 16 [16]. As deep learning models grow larger, recent years have seen models that cannot fit into a single GPU even with a batch size of 1 [32]. These models can become practical if efficient footprint reduction techniques are applied.

B. Memory Consumption Breakdown

To understand the reason behind NMT's large GPU memory footprint, we develop a GPU memory profiler[3] to show the detailed breakdown. We categorize the memory consumption in two orthogonal ways: (1) by layer types (e.g., RNN), and (2) by data structures. Major data structures include: feature maps, weights, and workspace:

(1) **Feature Maps**: Each layer needs memory for its own input and output variables. If any of those variables are needed in the backward pass to compute the gradients, it is stashed persistently in memory as feature maps, while those that are not can be released back to the storage pool. For example, consider the $\tanh$ activation function $Y = \tanh(X)$. Since we have $Y' = 1 - \tanh^2(X)$, the value of $\tanh(X)$ needs to be stored for the gradient computation in the backward pass.

(2) **Weights**: Layers such as fully-connected layers (Equation 1) have parameters W, B that are optimized as training progresses. In the following text, we use the term *Weights* as a generic term that includes W, B, plus their respective gradients and optimizer states which are used to do weight updates.

(3) **Workspace** is the scratchpad of a layer to compute the results. When a layer completes its own forward or backward pass, its workspace, if previously requested, can be freed.

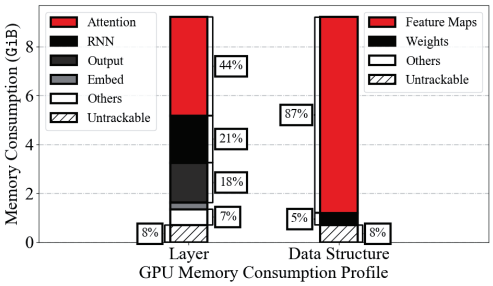


Fig. 4: Memory Consumption Breakdown by Layer Types (Left) and Data Structures (Right)

C. Runtime Breakdown

To motivate the use of selective recomputation approach, we do a runtime profile analysis that shows the runtime distribution across different layers. Figure 5 illustrates the NMT runtime breakdown on one training iteration. The profile is obtained from the *NVProf* tool [33]. We observe that **the runtime is unevenly distributed across different layers**, with 50% going into the fully-connected layers and the other 50% into many small compute kernels. The longest kernel of the latter runs for only 5 ms (the *mshadow* bar represents the tensor library backend of MXNet and consists of multiple CUDA kernels).

[3]The GPU memory profiler has been integrated as a part of MXNet [24]: https://issues.apache.org/jira/projects/MXNET/issues/MXNET-1404

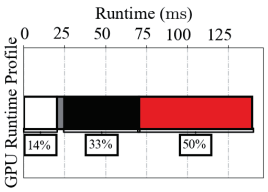


Fig. 5: Runtime Breakdown by Layer Types

D. Selective Recomputation

Given that the GPU memory consumption limits training performance and that the execution time is distributed unevenly across different layers, selective recomputation [17]–[19] can be a viable option to trade runtime with memory capacity. To illustrate the key idea behind selective recomputation, consider the computation graph in Figure 6, where we have a sequence of n operator nodes. Assume that the gradient node i' on the backward pass has data dependency on the output edge of its corresponding forward node i. The dependency is shown as an edge pointing from the forward to the backward pass (❶), which is marked as the feature map that has to be stashed. Therefore, the memory allocated for those edges cannot be released back to the storage pool, resulting in four persistent storage units by the time the forward pass completes (❷). If recomputation is used, those four dependency edges can be replaced with only one edge on the input to Node #1 (❸). This releases storage pressure as the inputs to Node #2 $\sim$ 4 can now be taken by their respective outputs (and hence do not need to be stashed anymore), but it comes with the cost of having to redo the forward computation when the backward pass starts (❹). This comes with runtime overhead, but such overhead can be controlled if the recomputed nodes (shown in gray in Figure 6b) are restricted to those that are computationally cheap. Hence, selective recomputation has the potential to reduce the memory footprint at small runtime cost.

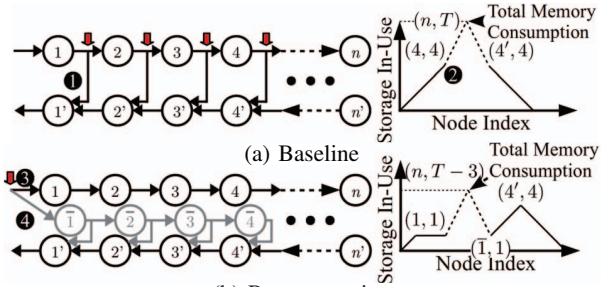


Fig. 6: A Computation Graph with Recomputation Applied (Red Arrows denote Persistent Feature Maps Storage)

In practice, however, we observe that the current state-of-the-art recomputation approach has limited benefits on the NMT training. Table I compares the training performance and GPU memory footprint between training with and without selective recomputation, where the recomputation implementation is based on the prior work [17]. We observe that recomputation causes the performance to drop by 17%, and it can only give a footprint reduction of 26% in return, which does not provide

enough memory space to increase the performance by, for example, increasing the batch size.

	Baseline	Chen et al. [17]
Avg. Throughput (samples/s)	1192	983
GPU Memory Footprint (GB)	10.0	7.4

TABLE I: NMT [22] Training with & without Recomputation

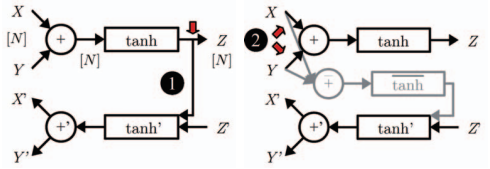
IV. Key Ideas

The major reason for the ineffectiveness of the state-of-the-art implementations of the recomputation approach is because they fail to adequately address two important challenges:

A. Footprint Reduction Estimation

Challenge #1. Accurately estimating footprint reduction. The effect of each operator recomputation on the total memory footprint needs to be carefully estimated. We observe from the red arrows in Figure 6b that although recomputation removes the feature maps on the outputs of Node $\#1 \sim 4$, it also brings in a new data entry which is the input to Node $\#1$ that now has to be stashed. In real models, it is possible for the latter to be larger than the former combined.

Consider the concrete example in Figure 7a, where X and Y, both being 1D arrays of shape $[N]$, are elementwise-added first before passing through a $\tanh$ activation function. The output Z is also of shape $[N]$. This example is a simplified version of the LSTM cell described in Section II (Figure 1).



(a) Baseline (b) Recomputation

Fig. 7: $Z = \tanh(X + Y)$

In the baseline, the framework only stashes Z as the feature map (❶), because it is the only value that is needed to compute the gradient in the backward pass. With the $+$ and $\tanh$ listed as recomputed in Figure 7b, the backward dependency is brought forward from Z to X and Y (❷), similar to Figure 6b. However, the only memory saving is from the feature map Z, while the new ones (X and Y) need new allocations, doubling the amount of persistent storage required from N to $2N$.

We therefore conclude that a practical recomputation strategy should involve the comparison between the feature maps that are released and those that are newly allocated. However, such a comparison is far from being trivial and is overlooked by prior works [17]. Although a simple comparison between the inputs size and outputs suffices to do the job in the example above, it only considers the storage allocations that are *local* to each operator and ignores the *global impact* of those allocations as it fails to consider the *storage reuse* across different operators within the same computation graph.

Consider another example in Figure 8, where we have T tensors of shape $[N]$ added with the same tensor of shape $[T \times N]$ by broadcasting input values and passing through the $\tanh$ activation function. The example is a simplified version of the attention scoring function introduced in Section II Figure 2. If we restrain the analysis scope within the dashed box and naïvely compare the inputs size versus outputs, we will arrive at the same conclusion as Figure 7 that recomputation is not needed. However, with recomputation the total feature map size can be reduced from $T^2 \times N$ to $T \times 2N$ and the key reason is because the storage of $[T \times N]$ is shared by multiple operators. Therefore, we conclude that a global graph analysis is needed to take *reuse* effects into account when using recomputation.

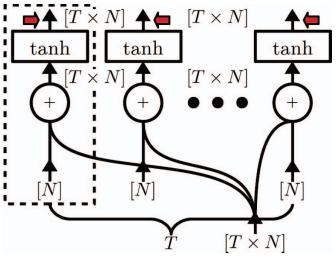


Fig. 8: T Tensors of Shape $[N]$ Added with $[T \times N]$ by Broadcasting Input Values and Passing Through the $\tanh$ Activation Function

A simple but impractical solution can be an entire computation graph traversal to evaluate the memory benefits of recomputing for every operator in the graph. This solution has combinatorial runtime complexity, which is challenging in the LSTM RNN context because the number of operators in the graph is usually huge (e.g., around 15.9K in large NMT models [22]). To address this challenge, we propose to partition the whole computation graph into small subgraphs to restrict the scope (and hence reduce the complexity) of the footprint reduction analysis. Compute-heavy layers (e.g., fully-connected layers) form the natural boundaries. This is because the goal of the footprint reduction analysis is to estimate the effect of *recomputation* on the total memory footprint. Since compute-heavy layers are never recomputed to avoid significant runtime overhead (as we mention in Section III-D), they are excluded from the scope of analysis and hence can serve as a good place to partition. After the partitioning, each subgraph captures any *reuse* across tensor edges in the computation graph. We then analyze each small subgraph *independently* and accurately estimate potential footprint reduction from recomputation.

In Section V-A, we show examples how footprint reduction estimation can help to (i) avoid pathological cases such as Figure 7 where recomputation increases the memory footprint, and (ii) reduce the recomputation runtime overhead.

B. Runtime Overhead Estimation

Challenge #2. Non-conservatively estimating runtime overhead. The effect of every operator recomputation on the total execution time needs to be carefully estimated taking layer specifics into consideration. From Figure 6b, we observe that recomputation always comes with runtime overhead, because feature maps have to be recomputed in the backward pass. Therefore, in practical implementations recomputation is always done selectively. One naïve way to do this is to simply exclude all the feature maps of the compute-heavy operators (e.g., convolutions, fully-connected) from being recomputed to keep the runtime overhead low [17]. This approach, however, is too conservative and leads to limited applicability in the LSTM

RNN context. A common example is the fully-connected layer that we have listed in Section II Equation 1:

$$Y = XW^T \Rightarrow \frac{dE}{dX} = \frac{dE}{dY}W, \frac{dE}{dW} = \frac{dE}{dY}^T X \quad (2)$$

where E is the training loss that has to be optimized.

We observe that the gradients of the fully-connected layer only have data dependency on X and W (but not Y), both of which are the *inputs* to the fully-connected layer. This implies that to recompute the feature maps of the fully-connected layer one does not have to recompute the layer itself (as the output Y is not needed). Such a property is layer-specific. There are also other types of layer-specific properties that can enable efficient footprint reduction. For example, the feature maps of ReLU activations [16] and $\mathrm{dropout}$ layers [23] can be stored in 1-bit binary format. Those layers require special handling so that we do not miss opportunities for footprint reduction and still avoid significant runtime overhead.

To address this challenge, we design a non-conservative runtime overhead estimator that infers the layer specifics before estimating the recomputation overhead. In the context of fully-connected layers, the idea is that the gradient computation of these layers only need the inputs rather than the outputs. The estimator's goal is therefore to distinguish between those two cases, and *only if* the feature maps of an operator are part of its outputs will the operator's runtime be added as a part of the overhead estimate. The key challenge for the estimator design lies in its generality. Although hard coding the layer specifics is possible, it is not a scalable solution given the number of layers supported in a machine learning framework [24]. Hence, we leverage the dataflow analysis approach to obtain such information, as we show in Section V-B.

C. Automatic and Transparent Compiler Pass Design

Machine learning programmers always have the option to do recomputation manually by writing hand-tuned GPU kernels [34], [35]. As Section VI will show, manual recomputation has the advantage of doing extra optimizations such as kernel fusion that could greatly reduce the recomputation overhead (and even improve runtime due to fewer memory accesses).

Despite its merits, manual recomputation has the following major shortcomings: (i) it is hard to pinpoint the correct places where recomputation should be done, especially in an LSTM RNN model with around 15.9K operators [22], and (ii) it requires nontrivial knowledge of GPU programming, machine learning algorithms, and framework system integration. This places significant burden on the programmer. We first added the recomputation optimization manually to the NMT model and spent more than two weeks seeking recomputation opportunities and testing the manual implementations in MXNet [24]. This non-trivial effort motivated us to push for an *automatic and transparent* method to perform this optimization (and we will show the performance comparison of manual versus automatic version in Section VI-B). An automatic and transparent recomputation scheme should reduce the GPU memory footprint without any changes needed to the training source code, while guaranteeing that (i) the recomputation impact on training performance is minimal and (ii) the GPU memory footprint is never worse than that in the baseline.

To this end, we present *Echo*, a compiler-based optimization scheme that can reduce the GPU memory footprint automatically and transparently. *Echo* addresses the two key challenges for accurate footprint reduction and non-conservative runtime overhead estimation in a computation graph compiler middle-end without domain-specific knowledge of the graph structures. We present the implementation details of *Echo* in Section V.

V. Implementation Details

We integrate *Echo* in NNVM [25], which is the computation graph compiler for MXNet [24], the state-of-the-art machine learning framework. Its compilation workflow is illustrated in Figure 9. The input to the workflow is a computation graph that consists of operator nodes of the forward pass. The shapes and data types of those nodes' data entries are unknown. NNVM starts by inserting the gradient operators into the computation graph, and then applying the pre-registered gradient functions to each node in the graph. If recomputation has been enabled, NNVM will additionally insert the recomputation nodes as in Figure 6. After the full computation graph with forward and backward operator nodes has been formed, NNVM infers the shapes and data types of each operator node's data entries from the input arguments (usually the training data). Finally, NNVM plans memory allocations by assigning virtual storage IDs to all the data entries. Those IDs will then be used to materialize the memory space of the computation graph nodes.

$$\textbf{Gradient} \rightarrow \textbf{InferShape} \ \& \ \textbf{Type} \rightarrow \textbf{PlanMemory}$$

Fig. 9: MXNet NNVM Compilation Workflow (in Sequence)

A. Footprint Reduction Estimation

We observe that with the current workflow, it is impossible to accurately estimate the footprint reduction, because critical information such as the *shape* and *data type* of each tensor edge is only available after the recomputation algorithm has been executed. We know from the example in Section IV-A (Figures 7 and 8) that this information is required to accurately estimate the footprint reduction. We therefore start by changing the compilation workflow in Figure 9 to Figure 10, so that more information is available by the time *Echo* is executed.

$$\textbf{Gradient} \rightarrow \textbf{InferShape} \ \& \ \textbf{Type} \rightarrow \textbf{EdgeUseRef} \rightarrow \textbf{\textit{Echo}} \rightarrow \textbf{InferShape} \ \& \ \textbf{Type} \rightarrow \textbf{AllocateMemory}$$

Fig. 10: Adjusted NNVM Compilation Workflow (The first *Gradient* pass does not perform recomputation)

With the new workflow, allocation-relevant information (e.g., shape and data type) is now available prior to *Echo*, which gives it the opportunity to accurately estimate the footprint reduction brought by recomputation using a bidirectional dataflow analysis. Algorithm 1 shows *Echo*'s high-level workflow. *Echo* first performs a backward pass (line 6-10) to obtain all the possible targets for recomputation, and partitions the entire graph at the compute-heavy layers to avoid significant overhead in case these layers are included in recomputation. It then performs a

Algorithm 1: *Echo*'s High-Level Workflow

```
Input : Computation Graph G
1  worklist H = [G.OutputNodes];
2  while !H.empty() do
3      h = H.pop();
4      if h ∈ G.InputPlaceholders then continue;
5      subgraph S = {h}, worklist W = [h.InputNodes];
       /* 1. Backward: Expand the subgraph S backward until blocked
          by compute-heavy layers                                  */
6      while !W.empty() do
7          w = W.pop();
8          if w ∈ G.InputPlaceholders then continue;
9          if w.op ∈ ComputeHeavyOps then
               H.append(w); continue;
10         S.insert(w); W.append(w.InputNodes);
11     create the recomputation path of S (Figure 11c);
       /* 2. Forward: Traverse through the subgraph S in topological
          order and estimate the footprint effect ∀recomputation   */
12     for s ∈ S.TopologicalOrderView do
13         if s.op ∈ ComputeHeavyOps then
14             create a dummy gradient node gs′;
15             GradFunc[s.op](s, gs′);
16             create a node s̄ copying s on the recomputation path;
17             for e ∈ s.InputEdges ∧ e ∈ gs′.InputEdges do
18                 link ē to s̄ and gs (Figure 14d);
19             continue;
20         if s.op ∈ BinarizableOps then insert encode and decode
             subroutines between s and its gradient node gs; continue;
21         edges set AllocEdges = RelEdges = {};
22         for e ∈ s.InputEdges do
23             for n ∈ S ∪ S.GradientGraph that
                 references e as input do
24                 if n ∈ S then
25                     AllocEdges.insert(n.OutputEdges);
26                     RelEdges.insert(n.InputEdges);
27                 else AllocEdges.insert(e);
28         for e ∈ AllocEdges do Alloc += e.Size;
29         for e ∈ RelEdges do Rel += e.Size;
30         if Rel ≥ Alloc then remove s̄ from the recomputation
             path (Figure 12a and Figure 12b);
```

forward pass (line 12-30) to remove recomputations that do not reduce memory footprint. The forward pass reduces the recomputation runtime overhead, and also guarantees that the total memory consumption after recomputation is performed will never increase compared with the baseline.

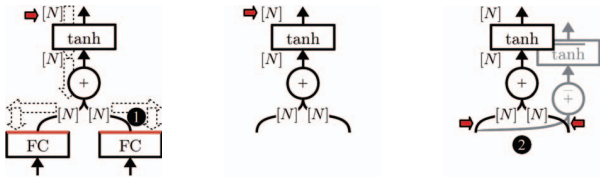
Figure 11 illustrates an example similar to Figure 7, with two fully-connected layers added before the elementwise-add operator. The backward pass, shown in Figure 11a, starts from the top (i.e., output) of the graph, propagates backward along the dashed edges, and then stops at the fully-connected layers (❶), because these two layers do not belong to the possible targets for recomputation as being too compute-heavy (see the runtime profile in Section III-C, Figure 5).

After the backward pass, *Echo* takes the nodes and edges that the backward pass has processed (Figure 11b), which form a partitioned subgraph of the original computation graph, and assumes that all the operator nodes within this subgraph can be recomputed. This is illustrated in Figure 11c as a graph shown in gray that mirrors the subgraph. The mirrored graph is the *recomputation path*, similar to the gray segments in Figure 6 and 7. Due to the recomputation path, the feature maps that are originally at the output edge of tanh in Figure 11a, are now placed at the beginning of the recomputation path (❷), as we have shown in Section III-D, Figure 6.

After the recomputation path has been formed, a followup forward pass removes recomputations that cannot reduce memory footprint. A node can be removed from the recomputation path if, by removing that node, the storage released from its inputs is greater than or equal to that allocated for its outputs.

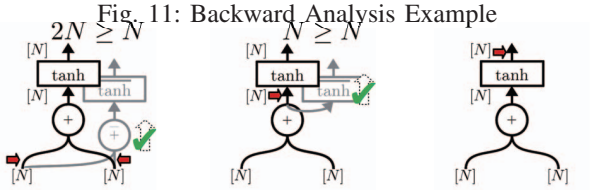
Based on the previous example, we consider to remove the + and tanh nodes from the recomputation path in succession. Figure 12a shows that the removal of + is successful, because its inputs size ($N+N$) is greater than its output (N). Similarly, Figure 12b shows that the removal of tanh is also successful, because its input size (N) is equal to its output size (N). After the removals, the final graph (Figure 12c) does not have any recomputation. This indicates that recomputation is unable to reduce memory footprint in this specific example (as we have seen in Section IV-A), but *Echo* is still able to preserve both the same performance and memory footprint as the baseline. This is in stark contrast to prior works [17]–[19] that introduce unnecessary recomputation which doubles the feature maps storage in this example (shown in Section IV-A, Figure 7b).

After the forward pass, the backward pass resumes at the inputs of the fully-connected layers in Figure 11a for the next partitioned subgraph. This backward-forward loop continues until all the inputs of the whole graph are reached. As one might notice, since the next subgraph continues at the places where the previous subgraph stops, each subgraph is *disjoint*. This property keeps the runtime complexity of Algorithm 1 reasonable (less than 300 ms for the models in Section VI in the hardware environment listed in Section VI-A).



(a) Backward Analysis Flow (b) Partitioned Subgraph (c) Recomputation Path

Fig. 11: Backward Analysis Example



(a) Forward on Elementwise Add (b) Forward on tanh Activation (c) Final Graph

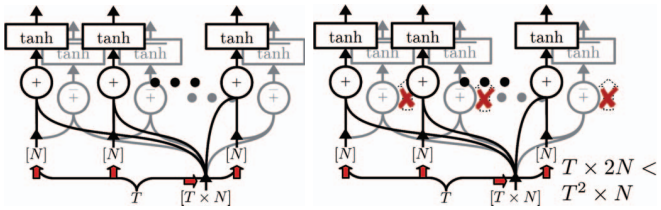
Fig. 12: Forward Analysis Example (Figure 11 Continued)

As we have discussed in the example in Section IV-A (Figure 8), the comparison between the storage released and allocated requires more than just the comparison between input versus output sizes that only reflects allocations that are *local* to each operator. For accurate footprint reduction estimation, *Echo* considers the *global effect* (i.e., *reuse*) of storage allocations within each subgraph *independently*, because each subgraph is disjoint and hence there is no reuse across subgraphs. *Echo* abstracts the *reuse* using the *use references* on each tensor edge, as it represents the number of times a particular tensor

(and hence the storage allocated to that tensor) is *reused*. The tensor edge use references come from the *EdgeUseRef* pass (see Figure 10), where the compiler traverses through the whole computation graph and, for each tensor, records the number of times it is referenced by different operators.

Figure 13 shows how *Echo* leverages the use references information in its analysis, using the example in Section IV-A (Figure 8), where we have T tensors of shape $[N]$ added with the same tensor of shape $[T \times N]$ by broadcasting inputs values and passing through the $\tanh$ activation function. Although the backward pass in this example (shown in Figure 13a) is similar to that in Figure 11c, where all operators are listed as recomputed, the forward pass is different. This is because the tensor $[T \times N]$ is used by T different operators, and trying to remove any of those operators on the recomputation path will cause all the other operators to be removed as well, since they all need the tensor $[T \times N]$ to be stashed as feature maps to do recomputation. This is illustrated as T parallel arrows on the recomputation path in Figure 13a. *Echo* therefore needs to compare the storage allocated for *all* T operators' inputs with their outputs. It then notices that, since the storage allocated for $[T \times N]$ is shared by all the T operators, the total inputs storage size ($T \times N + T \times N = T \times 2N$) is smaller than the outputs ($T \times T \times N = T^2 \times N$) when T is large enough (usually T is in the range of $50 \sim 100$ [4], [22]). *Echo* therefore stops trimming the recomputation path right at the beginning, leaving all operators marked as recomputed. This indeed leads to the optimal result, as the total feature maps storage in the final graph is $T \times 2N$, which is an order of magnitude smaller than storage required in the baseline ($T^2 \times N$).

We conclude that, compared with prior works [17]–[19] that recompute every layer that is not compute-heavy. *Echo* uses rigorous dataflow analysis to avoid pathological cases where recomputation is not needed while preserving those where it is beneficial. This explains why *Echo keeps overhead minimal and never increases the overall memory footprint*, as our results in Section VI-C1 will show.

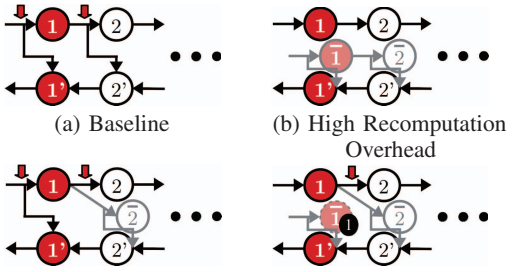


(a) Recomputation Path (b) Final Graph

Fig. 13: Analysis Example based on Figure 8

B. Runtime Overhead Estimation

As we show in Section IV-B, the gradients of the fully-connected layers only need their inputs. Layers such as convolutions and batched dot product also have similar property. Prior works [17] simply skip all the compute-heavy layers (Figure 14c). In contrast, *Echo* deems these layers as potentially good targets for recomputation by performing non-conservative runtime overhead estimation (Algorithm 1 line 13-18). *Echo* starts by inferring the data dependencies of the gradient operator that are specific to different types of layers. It does so by (i) creating a dummy gradient operator node, (ii) applying the gradient function to the (gradient, forward) operator tuple, and (iii) analyzing the data dependencies of the gradient operator on the forward operator. If the dependencies do not include the outputs of the forward operator, this implies that recomputation in this case does not require to recompute the operator itself, and hence its runtime is *not* added to the total runtime overhead when recomputing feature maps. If this is the case, *Echo* creates a "dead" node on the recomputation path (❶) that forwards the output edge of the previous recomputation node to the gradient node. However, the node itself is disconnected from the next node on the recomputation path. This implies that the node's outputs are never referenced by any other nodes, making the node effectively *dead* and thereby avoiding any unnecessary recomputation. Such an approach releases the feature maps on the input edges of the compute-heavy nodes but does not lead to situations with huge runtime overhead (as in Figure 14b), and can therefore further reduce the GPU memory footprint.



(a) Baseline (b) High Recomputation Overhead

(c) Prior Works [17] No Footprint Reduction (d) Low-Overhead Footprint Reduction

Fig. 14: Recomputation Strategies for Compute-Heavy Layers (shown in Red) whose Gradients only need Their Inputs

In addition to leveraging the layer specific gradient dependencies, *Echo* also uses layer specific encoding to binarize the feature maps of the $\mathrm{dropout}$ and ReLU layers [16], [23] (Algorithm 1 line 18). It encodes the dropout feature maps to 1-bit in the forward pass and decodes them back to 32-bit in the backward pass. Such layer specific optimizations give more footprint reduction with small runtime overhead.

In summary, we have demonstrated how *Echo* benefits from layer specific information and uses non-conservative runtime overhead estimation to reduce the memory footprint with low runtime overhead, as our results in Section VI will show.

VI. Evaluation

A. Methodology

Infrastructure. Our major compute platform is a single machine with 32-core AMD EPYC 7371 [36] and 4 NVIDIA RTX 2080 Ti GPUs [31] connected via PCIe v3 [37] (detailed specifications in Table II), installed with CUDA 10.0 [38], cuDNN 7.6.3 [39], and MXNet v0.12.1 [40]. All the experiments in this paper are conducted in this platform, except for the hardware sensitivity study in Figure 19.

Applications. We evaluate *Echo* by training the Sockeye [22] NMT toolkit on the IWSLT15 English-Vietnamese (small) [48] and WMT16 English-German (large) [49] datasets, using the

CPU	Memory
32-core AMD EPYC 7371 [36]	128 GB DDR4 [41]
32-core Intel Xeon E5-2686 v4 [42]	244 GB DDR4 [41]
GPU (Generation)	
NVIDIA RTX 2080 Ti [31] (Turing [43])	11 GB GDDR6 [44]
NVIDIA Tesla V100 [45] (Volta [46])	16 GB HBM2 [47]

TABLE II: CPU/GPU Specifications

hyperparameter settings from Zhu et al. [12] for the single-GPU experiments on the small dataset and Hieber et al. [22] for the multi-GPU experiments on the large dataset. In addition to NMT, we also show results on other state-of-the-art machine learning models [12], [26] shown in Table III.

Model	Dataset	Application
DeepSpeech2 [28]	LibriSpeech [50]	speech recognition
Transformer [27]	WMT16 EN-DE [49]	translation
ResNet [29]	ImageNet [51]	image classification

TABLE III: Models Evaluated in addition to NMT

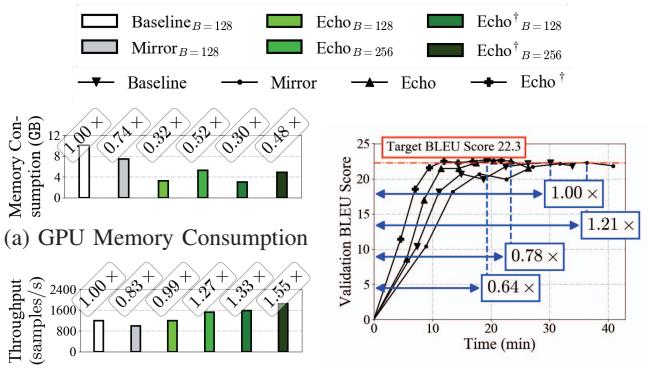
Baselines. We compare our fully automated and transparent approach, *Echo*, with two baselines: the baseline system without recomputation, which we refer to as *Baseline*, and the state-of-the-art implementation of recomputation by Chen et al. [17], which we refer to as *Mirror*.

For the experiments on NMT [4], [7], we use the superscript (†) to denote our hand-tuned implementation. We leverage the information provided by *Echo* to pinpoint the places where recomputation is beneficial and manually fuse the operators that are on the recomputation path into a single operator (i.e., Node #$\overline{1} \sim \overline{4}$ in Figure 6b are fused together as a single CUDA kernel). As we discussed in Section IV-C, this requires non-trivial effort and expertise in CUDA programming, machine learning algorithms, and framework system integration, but it shows potential benefits that recomputation can provide.

Metrics. We show the (1) GPU memory consumption and (2) throughput as the key metrics. We also show the training curves in the NMT experiments, and power/energy consumption on the GPUs in the multi-GPU experiments. The training curves are expanded using the CPU wall clock time. Those curves use BLEU score [52] to quantify the machine translation quality, where a higher BLEU score means better translation quality and a BLEU score that is greater than 20 is considered strong [22], [52]–[54]. As training progresses, we periodically query the memory and power usage of the GPUs using the *nvidia-smi* tool [55] and approximate the GPU energy consumption as power over time. The throughput is reported as the average of the throughput numbers given by the MXNet speedometer [56], which measures throughput by dividing the number of training samples by the CPU wall clock time.

B. Machine Translation Results with NMT Model

1) English-Vietnamese: Figure 15a and 15b illustrate the comparison of the GPU memory usage and the training throughput under different batch sizes on the English-Vietnamese translation task (the subscript B denotes the batch size used). We observe that *Echo* can achieve 3.13× footprint reduction ratio over *Baseline* and 2.31× over *Mirror* under the same



(a) GPU Memory Consumption

(b) Throughput

(c) Validation Curve BLEU Score

Fig. 15: (a) GPU Memory Consumption, (b) Throughput, and (c) Validation Curve BLEU Score compared between *Baseline*, *Mirror*, and *Echo*$^{(\dagger)}$ on English-Vietnamese NMT

batch size of 128 (35.4% of the reduction comes from more aggressive recomputation due to non-conservative runtime overhead estimation (Section IV-B)). We also notice that *Echo* only has 1% runtime overhead, which is 18× less than *Mirror*.

Because *Baseline*$_{B=128}$ and *Mirror*$_{B=128}$ consume around 10.0 GB and 7.4 GB of GPU memory respectively when the batch size is 128, their batch size can no longer be doubled (otherwise the GPU will run out of memory). The situation is different for *Echo*. Since it only consumes around 3.0 GB of memory, *Echo*'s training batch size can be further increased to 256, producing the training curve *Echo*$_{B=256}$ in Figure 15c. By training the NMT model with larger batches, we increase the throughput by 1.27× and converge to the same validation BLEU score 1.28× faster than the baseline. The reason why the achieved speedup 1.28× is smaller than those shown in Figure 3b is two-fold: (1) Similar to the ResNet model [29] (Figure 3a), throughput can saturate at large batch size, as the compute resource utilization increases with the batch size. (2) The recomputation overhead lessens the performance benefits of having a larger batch size. However, such negative impact can be mitigated by the hand-tuned implementation *Echo*†. As one might notice, at the same batch size of 128, *Echo*† (manually optimized version of *Echo*) reduces the GPU memory footprint by 3× while increasing, rather than decreasing, the throughput by 33%. The reason for the increase is because the overhead of the recomputation is so small that it is outweighed by the benefits of kernel fusion. Such benefits include reduction in (i) the `cudaLaunch` overhead and (ii) the number of GPU memory accesses [57]. As Figure 15c illustrates, after doubling the batch size, *Echo*† improves the speed of convergence further by 1.56× compared with the baseline.

Figure 16 illustrates the memory consumption breakdown comparison between *Baseline*, *Mirror*, and *Echo*. We observe that *Echo* is able to reduce the memory consumption on layers such as attention and RNN by 81.4% and 68.6% respectively when compared with the baseline and *Mirror*. The reason is because it accurately estimates the footprint reduction (Section IV-A) and hence can leverage more optimal recomputation paths that lead to lower memory footprint.

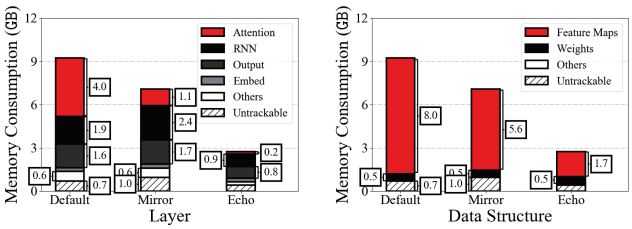


Fig. 16: NMT GPU Memory Consumption Breakdown Comparison between *Baseline*, *Mirror* and *Echo* ($B = 128$)

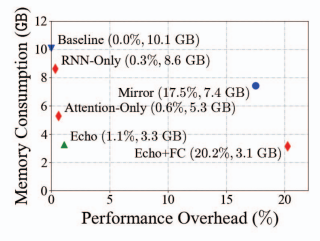


Fig. 17: Trade-Off between the Memory Consumption and Performance Overhead ($(x\%$, y GB$)$ stands for $x\%$ overhead over the baseline with y GB memory consumption)

To study whether doing more recomputation can enable more aggressive footprint reduction (or alternatively, doing less recomputation can reduce the performance overhead while still preserving most of the footprint reduction benefits), we selectively enable/disable the recomputations for certain types of layers (more fine-grain per-layer trade-off is difficult due to the large number of operator nodes in the model (e.g., around 15.9K in large NMT models [22])). Figure 17 shows the trade-off between the aggressiveness of the recomputation and the resulting performance overhead, which is measured as the throughput degradation over the baseline. We observe that solely enabling the recomputation of RNN or attention layers can only deliver 21.1% and 70.2% of *Echo*'s footprint reduction respectively. On the other hand, aggressively enabling the recomputation of fully-connected layers (FC) on top of *Echo* can lead to a minor footprint benefit of 1.8%, with a drastic 20.2% performance overhead. We conclude that out of all the configurations examined, *Echo* finds the sweet spot in selecting what layers to recompute to maximize the footprint reduction with acceptable overhead.

2) English-German: Multi-GPU training is a common way to reduce the training time [7], [8], however, in multi-GPU training, communication can potentially become a bottleneck. Moreover, power and energy is now a primary concern as GPU cards such as RTX 2080 Ti is known for being power-hungry (having a TDP at around 250 W) [31].

Figure 18a and 18c show the memory usage under different batch sizes and device settings on the English-German translation task (the superscript Dev denotes the number of GPUs used, and if multiple GPUs are used, the memory usages are aggregated up across all GPUs). We observe from Figure 18a that *Baseline* already has a memory consumption of more than 8 GB on a single GPU when the batch size is 16, meaning that we need to use 4 GPUs to train on a batch size of 64. However, with *Echo* we can train on just a single GPU with a batch of 64, and even 128, as the memory consumption (9.6 GB) can still fit in the memory capacity of one RTX 2080 Ti card [31].

Figure 18b and 18d show the throughput comparison between different implementations. We observe that although *Echo* is 14% behind *Baseline* when the batch size is 16, *Echo* running on one GPU outperforms the baseline on four GPUs by 1.35×, both using the maximum training batch size. The reason for the low scalability of the multi-GPU baseline (2.14×) is two-fold: (1) the nature of the translation model that limits the scalability [58], [59] and (2) relatively low bandwidth of the PCIe [37] interconnect. Although NVLink-enhanced compute systems such as the ones in Amazon EC2 p3.8xlarge instance [60] (32-core Intel Xeon E5-2686 v4 [42] and 4 NVIDIA Tesla V100 GPUs [45] connected via NVLink [61], detailed specifications in Table II) can be used to boost the scalability up to 3.40×, such systems can be as much as 6× more expensive [62], [63]). Moreover, even in this hardware setup, we observe that *Echo* running on one GPU achieves nearly the same performance with the *Baseline* running on four GPUs, as Figure 19 shows.

We further observe that *Echo* can significantly reduce the power and energy consumption on the GPUs. In Figure 20a we show the validation BLEU score curve in one training epoch expanded by the CPU wall clock time and Figure 20b the averaged power and accumulated energy consumption of all the GPUs. After one epoch, all implementations reach a BLEU score of 28.0 on the validation dataset, however, $\textit{Echo}_{B=128}^{\text{Dev}=1}$ completes the epoch 1.35× faster than $\textit{Baseline}_{B=64}^{\text{Dev}=4}$. It also saves 65% of the energy consumption on the GPUs, because it requires only one GPU to train on this large batch size.

On the other hand, although *Mirror* can halve the number of GPUs required for training at a batch size of 64, it cannot further squeeze the training model onto a single GPU, let along further doubling the batch size. Hence, it is 31% behind *Echo* in convergence speed and consumes 2× more energy on the GPUs to complete the training.

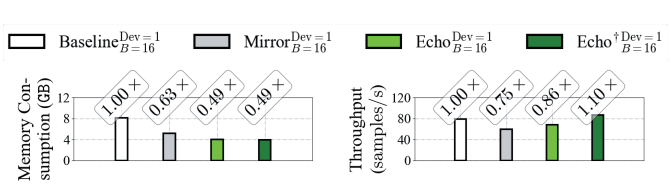
In summary, we have shown the benefit of *Echo* on the GPU memory footprint and how to convert such benefit into faster convergence speed and/or lower GPU energy consumption. In fact, we have also implicitly shown how one could train larger and deeper model with *Echo*, as the model for English-German translation has 2× more layers than that for English-Vietnamese [12], [22], but with *Echo* they can both train properly on a single GPU under a batch size of 128. We conclude that *Echo* can not only provide performance gain and energy consumption reduction, but also allow us to train deeper models with the same amount of GPU resources.

C. Echo Generality Across Machine Learning Models

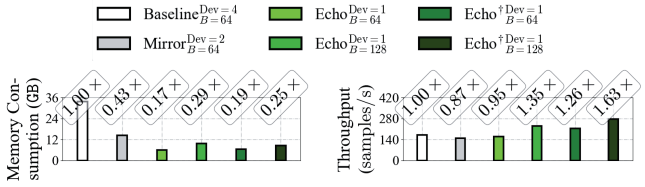
Since *Echo*'s key ideas are independent of the structure of the computation graph, it is potentially applicable to any machine learning models. In this section, we evaluate *Echo* on four state-of-the-art (SOTA) machine learning models.

1) DeepSpeech2: Figure 22(1) shows the comparison on the GPU memory consumption and training throughput by training the DeepSpeech2 (DS2) model [28], which is the SOTA model for speech recognition [12], [26]. We use 40K audio samples from the LibriSpeech [50] dataset for training.

Figure 21 shows the GPU memory consumption breakdown of DS2 [28] with a batch size of 8. We notice that 40% of



(a) GPU Memory Consumption (b) Throughput



(c) GPU Memory Consumption (d) Throughput

Fig. 18: (a, c) GPU Memory Consumption and (b, d) Throughput compared between *Baseline*, *Mirror*, and *Echo*(†) on English-German NMT (a-b: Single-GPU, $b = 16$, c-d: Multi-GPU, $B = 64/128$)

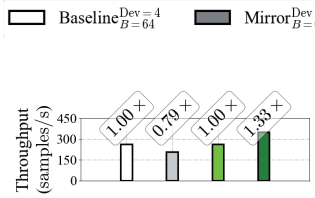
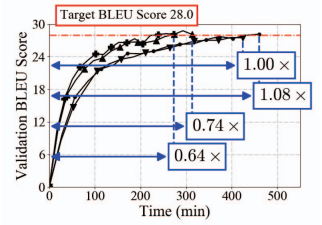
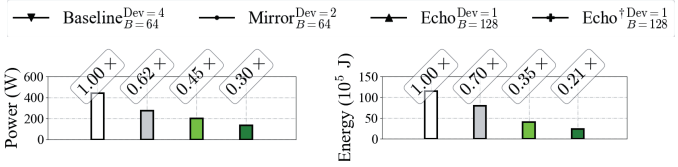


Fig. 19: Throughput Comparison on 4 Tesla V100s connected via NVLink



(a) Validation Curve BLEU Score



(b) Power & Energy Consumption

Fig. 20: (a) Validation Curve BLEU Score and (b) Power & Energy Consumption compared between *Baseline*, *Mirror*, and *Echo*(†) on English-German NMT

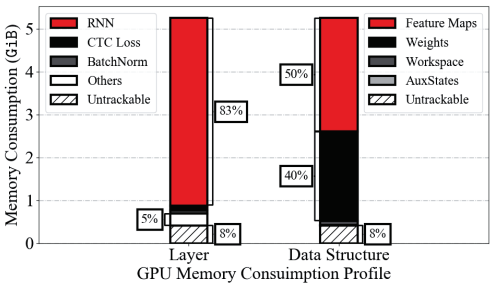


Fig. 21: DS2 Memory Consumption Breakdown by Layer Types (Left) and Data Structures (Right) ($B = 8$)

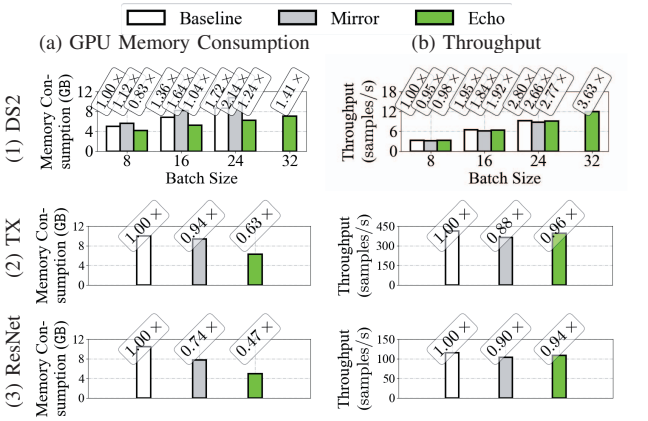


Fig. 22: (a) GPU Memory Consumption and (b) Throughput compared between *Baseline*, *Mirror*, and *Echo* on DS2 (1), Transformer (TX, 2), and ResNet (3)

the GPU memory allocated is for weights, making feature maps less important under the small batch size. This matches our observation from Figure 22(1a) that when the batch size is small, *Echo* cannot provide significant footprint reduction over the baseline. However, as the batch size increases from 8 to 24, the GPU memory footprint with *Echo* increases much slower compared with the baseline and *Mirror*, because relative proportion of feature maps increases and the latter two cannot adequately address this increase. Thereby, *Echo* does not hit the GPU memory capacity wall even under a batch size of 32, allowing the training throughput to further scale up, as is illustrated in Figure 22(1b). The recomputation runtime overhead of *Echo* is within 2% of the baseline under the same batch size, giving it the opportunity to compensate for the performance loss by increasing the batch size. As the rightmost bar of Figure 22(1b) shows, the throughput of $Echo_{B=32}$ is 3.6× that of $Baseline_{B=8}$ and 1.3× that of $Baseline_{B=24}$ (the best throughput in the baseline).

We further observe from Figure 22(1a) that *Mirror* consumes more (rather than less) GPU memory than the baseline. This is because it fails to accurately estimate the footprint reduction effects after recomputation (Challenge #1, Section IV-A). It also has 5× more runtime overhead compared to *Echo*.

2) Transformer: Both the NMT and DS2 are RNN-based models [4], [7], [28]. To demonstrate *Echo*'s generality beyond RNNs, we evaluate the effect of *Echo* on the Transformer model [27], which is the state-of-the-art model for machine translation [12], [26] and does not have a RNN component in it. Figure 22(2) shows that *Echo* achieves a footprint reduction ratio of 1.59× over the baseline with 3.0× less overhead than *Mirror*, where 38% of the footprint reduction over the baseline comes from layer specific knowledge that the feature maps of the dropout layer can be binarized [16] (Section IV-B).

3) ResNet: All the previous models belong to the domain of sequence-to-sequence learning. To show *Echo*'s generality in other domains, we further evaluate the effect of *Echo* on the ResNet-152 model, which is the state-of-the-art model for image classification [12], [26]. Figure 22(3) shows that *Echo* achieves a footprint ratio of 2.13× over the baseline and 1.57× over *Mirror* with 1.67× less overhead than *Mirror*. Figure 23 shows the memory consumption breakdown comparison between *Baseline*, *Mirror*, and *Echo*. We observe that *Echo* reduces the footprint of compute-heavy layers (i.e., Conv) by 58.3% when compared with *Mirror*, because the non-conservative runtime overhead estimation (Section IV-B)

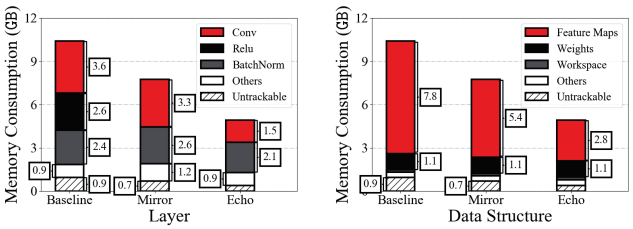


Fig. 23: ResNet-152 GPU Memory Consumption Breakdown Comparison between *Baseline*, *Mirror*, and *Echo* ($B = 32$)

offers more footprint reduction opportunities on those layers.

Although models such as Transformer and ResNet might not always be able to directly benefit from the footprint reduction by achieving performance gains, they can still benefit indirectly by becoming deeper under the same GPU memory budget. Table IV shows the maximum number of Transformer and ResNet layers that we can run on one RTX 2080 Ti [31]. We choose the number of layers specified in Hieber et al. [22] and He et al. [29] and pick the maximum batch sizes that can fit into the 11 GB GPU memory budget in the baseline. As we experiment on *Mirror*, *Echo* while keeping the batch sizes fixed, we observe that *Echo* increases the maximum number of layers by $1.83\times$ and $4.0\times$ on Transformer and ResNet respectively. Such increase in depth aligns with the recent trends in deep learning that have growing demand for more layers [51], [64].

Model	Baseline	Mirror	Echo
Transformer	6	6	11
ResNet	50	101	200

TABLE IV: Maximum Number of Layers one RTX 2080 Ti

In summary, by evaluating *Echo* on diverse workloads, we have shown the generality of the ideas behind *Echo*. We also conclude that *Echo* is able to efficiently and effectively reduce the memory footprint for RNNs [4], [7], [28], Transformer [27], and CNNs (ResNet [29]) with very low overhead. Such footprint benefits can be converted into performance improvements and/or an increase in the number of layers that can be executed while using the same batch size and GPU memory budget.

VII. Related Work

In this work, we present *Echo*, a compiler-based optimization scheme that automatically and transparently reduces the GPU memory footprint used for training across diverse machine learning models without any changes needed to the training source code. *Echo* finds and addresses two key challenges of selective recomputation that are missing in prior works [17]–[19]. Compared with other prior works, *Echo* (i) focuses on training rather than inference [13], [14], [65]–[72], and (ii) requires no domain-specific knowledge of the computation graph structures and/or manual efforts to hand-write CUDA kernels [16], [34], [35], and (iii) makes no changes to the underlying training algorithm [73], [74]. It is largely orthogonal to prior works that speed up training using model/pipelined parallelism [75], [76] and networking optimizations [77], [78].

Selective Recomputation. In parallel with our work [35], TWRemat [20] and Checkmate [21] also leverage the selective recomputation idea but using different approaches. While The former transforms the computation graph into a tree and solves the recomputation problem recursively by decomposing the tree, the latter formulates the problem as a constrained optimization problem and solves it using integer linear programming. These proposals are fundamentally different from *Echo*'s dataflow analysis approach.

Memory Virtualization. vDNN [15], cDMA [79], and MC-DLA [80] fit large neural networks in the device memory by virtualizing the memory usage using the host-side or the device-side memory nodes. The idea of virtualization can be used jointly with that of selective recomputation, and it has been shown in Capuchin [81] that combining the two ideas achieves the same amount of footprint reduction as the sole virtualization approach but with better performance (7%).

Data Encoding/Compression for CNNs. Our prior work, Gist [16], [82] proposes several lossless memory compression technique in the context of CNNs. One of the techniques requires the use of ReLU activations. Unfortunately, LSTM RNNs mostly use tanh and sigmoid, and hence Gist encodings become mostly inapplicable in the LSTM RNN context. There have been numerous efficient footprint reduction techniques that target inference [13], [14], [65]–[72]. These works focus on reducing the footprint of model weights. However, in training, weights are frequently updated, so it is challenging to apply these ideas in the training context. Furthermore, as is shown in Section III-B (Figure 4), weights are not the major memory consumer in NMT training.

Machine Learning Compilers. *Echo* is a compiler-based optimization that reduces the GPU memory footprint of DNN training tasks. There are other works on machine learning compilers that propose new programming paradigm and/or runtime performance improvements, which include TVM [83], Relay [84], Latte [85], TensorFlow XLA [86], Glow [87], and TensorComprehensions [88]. These are all possible platforms to which *Echo* could be added.

Algorithmic Innovations. Unlike *Echo*, there have also been works that address the inefficiency of LSTM RNN training from an algorithmic perspective. For example, BPPSA [73] re-formulates the backpropagation of neural networks as a parallel scan operation and improves the runtime complexity of the backward pass from $\Theta(n)$ to $\Theta(\log(n))$ (where n is the number of compute devices). Such approaches can be used in conjunction with *Echo* for even better performance.

VIII. Conclusion

In this paper, we propose *Echo*, an open-sourced compiler-based optimization scheme that reduces memory footprint automatically and transparently. On four state-of-the-art machine learning models, *Echo* reduces the GPU memory footprint by $1.89\times$ on average and $3.13\times$ maximum with marginal runtime overhead. System researchers, machine learning practitioners can all benefit from *Echo*, as it speeds up training convergence, reduces the GPU energy consumption, and allows for larger and deeper models. We hope that *Echo* would become a platform

for further research on memory footprint optimizations and efficient system design for key machine learning applications.

IX. Acknowledgement

We want to thank Xiaodan Tan, Qiongsi Wu (University of Toronto), Haibin Lin, Sheng Zha (Amazon Web Services) for their constructive feedback during the development of this work. This work was supported in part by the NSERC Discovery grant, the Canada Foundation for Innovation JELF grant, the Connaught Fund, the Huawei grants, the Province of Ontario, the Government of Canada through CIFAR, and sponsors of the Vector Institute (www.vectorinstitute.ai/#partners).

References

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Compututation*, vol. 9, pp. 1735–1780, Nov. 1997.

[2] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *CoRR*, vol. abs/1409.2329, 2014.

[3] M. Sundermeyer, R. Schlüter, and H. Ney, "LSTM neural networks for language modeling," in *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[4] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[5] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems 27* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 3104–3112, Curran Associates, Inc., 2014.

[6] D. Britz, A. Goldie, T. Luong, and Q. Le, "Massive exploration of neural machine translation architectures," *ArXiv e-prints*, Mar. 2017.

[7] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016.

[8] A. Graves, A. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) 2013*, pp. 6645–6649, IEEE, 2013.

[9] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks," in *International Conference on Machine Learning*, pp. 1764–1772, 2014.

[10] T. Lei, Y. Zhang, and Y. Artzi, "Training RNNs as fast as CNNs," *CoRR*, vol. abs/1709.02755, 2017.

[11] J. Bradbury, S. Merity, C. Xiong, and R. Socher, "Quasi-recurrent neural networks," *International Conference on Learning Representations (ICLR 2017)*, 2017.

[12] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Benchmarking and analyzing deep neural network training," in *IEEE International Symposium on Workload Characterization (IISWC) 2018*, Aug. 2018.

[13] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," *SIGARCH Computer Architecture News*, vol. 44, pp. 243–254, June 2016.

[14] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *International Conference on Learning Representations (ICLR 2016)*, 2016.

[15] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 18:1–18:13, IEEE Press, 2016.

[16] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *Proceeding of the 45st Annual International Symposium on Computer Architecture*, ISCA 2018, pp. 776–789, 2018.

[17] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *ArXiv e-prints*, 2016.

[18] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, "Memory-efficient backpropagation through time," *CoRR*, vol. abs/1606.03401, 2016.

[19] J. Martens and I. Sutskever, "Training deep and recurrent networks with hessian-free optimization," in *Neural networks: Tricks of the trade*, pp. 479–535, Springer, 2012.

[20] R. Kumar, M. Purohit, Z. Svitkina, E. Vee, and J. Wang, "Efficient rematerialization for deep networks," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 15172–15181, Curran Associates, Inc., 2019.

[21] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica, "Breaking the memory wall with optimal tensor rematerialization," in *Proceedings of Machine Learning and Systems 2020*, pp. 497–511, 2020.

[22] F. Hieber, T. Domhan, M. Denkowski, D. Vilar, A. Sokolov, A. Clifton, and M. Post, "Sockeye: A toolkit for neural machine translation," *CoRR*, vol. abs/1712.05690, 2017.

[23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, Jan. 2014.

[24] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[25] DMLC, "NNVM: Build deep learning system by parts," 2017.

[26] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, D. Kang, D. Kanter, N. Kumar, J. Liao, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. Janapa Reddi, T. Robie, T. St John, C.-J. Wu, L. Xu, C. Young, and M. Zaharia, "MLPerf training benchmark," in *Proceedings of Machine Learning and Systems 2020*, pp. 336–349, 2020.

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

[28] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. H. Engel, L. Fan, C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu, "Deep speech 2: End-to-end speech recognition in english and mandarin," *CoRR*, vol. abs/1512.02595, 2015.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1," ch. Learning Internal Representations by Error Propagation, pp. 318–362, Cambridge, MA, USA: MIT Press, 1986.

[31] NVIDIA, "Geforce rtx 2080 ti," 2018.

[32] Google Research, "BERT #out-of-memory issues," 2018.

[33] NVIDIA, "Profiler user's guide v10.0," 2018.

[34] C. Meng, M. Sun, J. Yang, M. Qiu, and Y. Gu, "Training deeper models by GPU memory optimization on tensorflow," in *Proceedings of Machine Learning Systems Workshop in NIPS*, 2017.

[35] B. Zheng, A. Tiwari, N. Vijaykumar, and G. Pekhimenko, "EcoRNN: Efficient computing of LSTM RNN training on gpus," *CoRR*, vol. abs/1805.08899, 2018.

[36] AMD, "AMD EPYC™7371," 2019.

[37] Wikipedia, "PCI Express 3.0," 2011.

[38] NVIDIA, "CUDA toolkit documentation v10.0," 2018.

[39] NVIDIA, "cuDNN library developer guide v7.6.3," 2019.

[40] Apache MXNet, "MXNet ver. 0.12.1," 2017.

[41] JEDEC, "Main memory: DDR4 and DDR5 SDRAM," 2017.

[42] Intel, "Xeon E5-2686 v4 - Intel," 2016.

[43] NVIDIA, "NVIDIA Turing GPU architecture whitepaper," 2018.

[44] JEDEC, "Graphics double data rate 6 (GDDR6) SGRAM standard," 2018.

[45] NVIDIA, "Tesla V100 data center GPU," 2017.

[46] NVIDIA, "NVIDIA Tesla V100 GPU architecture," 2017.

[47] JEDEC, "High bandwidth memory (HBM) DRAM," 2020.

[48] C. D. Manning, M.-T. Luong, A. See, and H. Pham, "Neural machine translation," 2018.

[49] WMT16, "ACL 2016 first conference on machine translation," 2016.

[50] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5206–5210, IEEE, 2015.

[51] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 248–255, IEEE, 2009.

[52] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, (Stroudsburg, PA, USA), pp. 311–318, Association for Computational Linguistics, 2002.

[53] M. Luong, E. Brevdo, and R. Zhao, "Neural machine translation (seq2seq) tutorial," 2017.

[54] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-source toolkit for neural machine translation," *CoRR*, vol. abs/1701.02810, 2017.

[55] NVIDIA, "NVIDIA system management interface program," 2018.

[56] Apache MXNet, "Speedometer," 2018.

[57] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*. Cambridge, MA, USA: MIT Press, 1990.

[58] M. Luong, E. Brevdo, and R. Zhao, "Neural machine translation (seq2seq) tutorial #multi- GPU training," 2017.

[59] M. Luong, E. Brevdo, and R. Zhao, "Neural machine translation (seq2seq) tutorial #WMT german-english," 2017.

[60] Amazon Web Services, "Amazon EC2 P3 instance product details," 2019.

[61] NVIDIA, "NVLink," 2019.

[62] Lambda Labs, "Deep learning workstation for 2019 - 4x GPUs: Customize — Lambda Quad," 2019.

[63] Lambda Labs, "Tesla V100 server - 4 GPUs or 8 GPUs + NVLink — Lambda Hyperplane," 2019.

[64] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green AI," *CoRR*, vol. abs/1907.10597, 2019.

[65] A. Jain, P. Hill, S.-C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 41:1–41:13, IEEE Press, 2016.

[66] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 1–13, IEEE Press, 2016.

[67] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 269–284, ACM, 2014.

[68] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 367–379, IEEE Press, 2016.

[69] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 19:1–19:12, IEEE Press, 2016.

[70] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA 2017, (New York, NY, USA), pp. 27–40, ACM, 2017.

[71] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 267–278, IEEE Press, 2016.

[72] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "ScaleDeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 13–26, ACM, 2017.

[73] S. Wang, Y. Bai, and G. Pekhimenko, "BPPSA: Scaling back-propagation by parallel scan algorithm," in *Proceedings of Machine Learning and Systems 2020*, pp. 451–469, 2020.

[74] M. MacKay, P. Vicol, J. Ba, and R. B. Grosse, "Reversible recurrent neural networks," in *Advances in Neural Information Processing Systems 31* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), pp. 9042–9053, Curran Associates, Inc., 2018.

[75] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, (New York, NY, USA), p. 1–15, Association for Computing Machinery, 2019.

[76] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen, "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 103–112, Curran Associates, Inc., 2019.

[77] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," in *Proceedings of Machine Learning and Systems 2019*, pp. 132–145, 2019.

[78] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, (New York, NY, USA), p. 16–29, Association for Computing Machinery, 2019.

[79] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pp. 78–91, IEEE Computer Society, 2018.

[80] Y. Kwon and M. Rhu, "Beyond the memory wall: A case for memory-centric hpc system for deep learning," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, p. 148–161, IEEE Press, 2018.

[81] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based GPU memory management for deep learning," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 891–905, Association for Computing Machinery, 2020.

[82] A. Phanishayee, G. Pekhimenko, and A. Jain, "Efficient data encoding for deep neural network training," Nov. 14 2019. US Patent App. 16/024,311.

[83] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "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.

[84] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock, "Relay: A new ir for machine learning frameworks," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, (New York, NY, USA), pp. 58–68, ACM, 2018.

[85] L. Truong, R. Barik, E. Totoni, H. Liu, C. Markley, A. Fox, and T. Shpeisman, "Latte: A language, compiler, and runtime for elegant and efficient deep neural networks," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), pp. 209–223, ACM, 2016.

[86] TensorFlow, "XLA overview," 2020.

[87] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph lowering compiler techniques for neural networks," *CoRR*, vol. abs/1805.00907, 2018.

[88] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, vol. abs/1802.04730, 2018.