BPPSA: Scaling Back-propagation by Parallel Scan Algorithm

by

Shang Wang

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

BPPSA: Scaling Back-propagation by Parallel Scan Algorithm

Shang Wang
Master of Science
Graduate Department of Computer Science
University of Toronto
2020

In an era when the performance of a single compute device plateaus, software must be designed to scale on massively parallel systems for better runtime performance. However, in the context of training deep learning models, the popular back-propagation (BP) algorithm imposes a *strong sequential dependency* in the process of gradient computation. Under model parallelism, BP takes $\Theta(n)$ steps to complete which hinders its scalability on parallel systems (n represents the number of compute devices into which a model is partitioned).

In this work, in order to improve the scalability of BP, we reformulate BP into a *scan* operation which is a primitive that performs an in-order aggregation on a sequence of values and returns the partial result at each step. We can then scale such reformulation of BP on parallel systems by our modified version of the Blelloch scan algorithm which theoretically takes $\Theta(\log n)$ steps. We evaluate our approach on a vanilla Recurrent Neural Network (RNN) training with synthetic datasets and a RNN with Gated Recurrent Units (GRU) training with the IRMAS dataset, and demonstrate up to 2.75× speedup on the overall training time and 108× speedup on the backward pass. We also demonstrate that the retraining of pruned networks can be a practical use case of our method.

Acknowledgements

This work is published with the same title in the Third Conference on Machine Learning and Systems (*MLSys 2020*) [70]. I want to thank my co-author, Yifan Bai, for her indispensable contributions. I also want to thank my advisor and co-author, Gennady Pekhimenko, for his invaluable guidance and support during the development of both this work towards its eventual publication, as well as me towards a (competent, hopefully) researcher.

I want to thank Xiaodan (Serina) Tan, James Gleeson, Geoffrey Yu, Roger Grosse, Jimmy Ba, Andrew Pelegris, Bojian Zheng, Kazem Cheshmi and Maryam Mehri Dehnavi for their constructive feedback during the development of this work.

I'm forever grateful to my parents, Guojin Shen and Wenqi Wang, and my fiancée, Diantong Yang. Without their unbounded love and support, I would not have been able to achieve what I have achieved now. I also want to thank all my friends (too many to enumerate) who helped me countless times when I needed it the most.

Contents

1 Introduction **1**

2 Background and Motivation **3**
2.1 Problem Formulation . . . 3
2.2 Prior Works . . . 3
2.3 Definition of the Scan Operation . . . 4

3 Proposed Method: BPPSA **5**
3.1 Back-propagation as a Scan Operation . . . 5
3.2 Scaling Back-propagation with the Blelloch Scan Algorithm . . . 6
3.3 Jacobian Matrices in Sparse Format . . . 6
3.4 Generating Jacobian Matrix in CSR Analytically . . . 7
3.5 Convergence . . . 8
3.6 Complexity Analysis . . . 9

4 Methodology **10**
4.1 RNN End-to-end Benchmark . . . 10
4.2 GRU End-to-end Benchmark . . . 12
4.3 Pruned VGG-11 Micro-benchmark . . . 13

5 Evaluation **14**
5.1 RNN End-to-end Benchmark . . . 14
5.2 GRU End-to-end Benchmark . . . 16
5.3 Pruned VGG-11 Micro-benchmark . . . 16

6 Conclusion **18**

Bibliography **19**

A Artifact Appendix **25**
A.1 Abstract . . . 25
A.2 Artifact Check-list (Meta-information) . . . 25
A.3 Description . . . 26
A.3.1 Hardware Dependencies . . . 26
A.3.2 Software Dependencies . . . 26

A.3.3 Datasets . . . 26
A.3.4 Models . . . 26
A.4 Installation . . . 27
A.5 Experiment Workflow . . . 27
A.6 Evaluation and Expected Result . . . 27
A.7 Experiment Customization . . . 28
A.8 Methodology . . . 28

B Space Complexity of GPipe **29**

C Affect of PipeDream's Staleness on Adam **30**

D Sparse Jacobian Generation Routines **32**
D.1 Convolution . . . 32
D.2 ReLU . . . 32
D.3 Max-pooling . . . 32

E Overhead Analysis of the GRU End-to-end Benchmark **36**

F GRU Training Curve **38**

G Additional Hardware Sensitivity Results **39**

Chapter 1

Introduction

The training of deep learning models demands more and more compute resources [4] as the models become more powerful and complex with an increasing number of layers in recent years [40, 66, 65, 30, 34]. For example, ResNet can have more than a thousand layers [31], and ResNet-152 takes days to train on eight state-of-the-art GPUs [20]. Now that the performance of a single compute device plateaus [24, 6], training has to be designed to scale on massively parallel systems.

Data parallelism [63] is the most popular way to scale training by partitioning the training data among multiple devices, where each device contains a full replica of the model. As the number of devices increases, data parallelism faces the trade-off between synchronization cost in synchronous parameter updates and staleness in asynchronous parameter updates [9]. Furthermore, recent studies demonstrate the scaling limit of data parallelism even when assuming perfect implementations and zero synchronization cost [63]. Lastly, data parallelism cannot be applied when a model does not fit into one device due to memory constraints (e.g., caused by deep network architecture, large batch size, or high input resolution [59, 72]).

Model parallelism [39, 35, 64, 48] is another approach to distributed training which partitions a model and distributes its parts among devices. It covers a wide spectrum of training deep learning models where data parallelism does not suffice. Naïve training under model parallelism does not scale well with the number of devices due to under-utilization of the hardware resources, since *at most one* device can be utilized at any given point in time [48]. To address the aforementioned issue, prior works on pipeline parallelism, including PipeDream [48] and GPipe [35], propose pipelining across devices for better resource utilization; however, as the number of layers and devices increases, pipeline parallelism still faces the trade-off between resource utilization in synchronous parameter updates and staleness in asynchronous parameter updates [48]. Moreover, to fully fill the pipeline with useful computation, each device needs to store the activations at the partition boundaries for all mini-batches that enter the pipeline. Therefore, the maximum number of devices that pipeline parallelism can support is limited by the memory capacity of a single device.

Algorithmically, the fundamental reason for this scalability limitation observed from prior works is that the back-propagation (BP) algorithm [60] imposes a *strong sequential dependency* between layers during the gradient computation. Since computing systems evolve to have more and more parallel nodes [24, 6], in this work, we aim at exploring the following question: *How can BP scale efficiently when the number of layers and devices keeps increasing into the foreseeable future?*

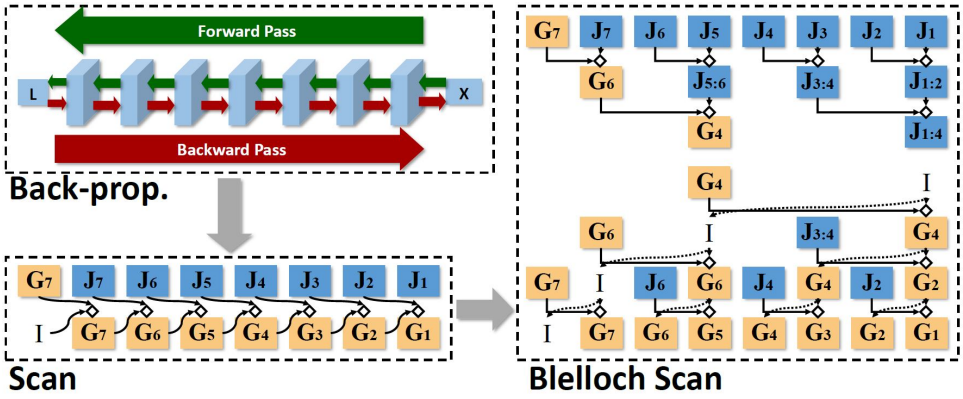


Figure 1.1: BP as a *scan* operation, scaled by our modified version of the *Blelloch scan* algorithm.

To answer this question, we utilize a primitive operation called *Scan* [13] that performs an in-order aggregation on a sequence of values and returns the partial result at each step. Parallel algorithms [33, 13] have been developed to scale the scan operation on massively parallel systems. We observe that BP is mathematically similar to a scan operation on the transposed Jacobian matrix [71] of each layer and the gradient vector of the output from the last layer. Inspired by this key observation, we *restructure* the strong sequential dependency of BP, and present a new method to scale **B**ack-**p**ropagation by **P**arallel **S**can **A**lgorithm (**BPPSA**). Our major contributions are summarized below.

• We reformulate BP as a *scan* operation and modify the *Blelloch scan* algorithm [13] to efficiently scale BP in a parallel computing environment. Our method has a theoretical step complexity[1] of $\Theta(\log n)$, where n represents the number of devices into which a model is partitioned, compared to $\Theta(n)$ of the naïve implementation of model parallelism. Moreover, our algorithm does not have the theoretical scalability limit by the memory capacity of a single device as pipeline parallelism does. As an example, Figure 1.1 shows how BP for training a network composed of 7 layers (blue cubes) can be reformulated into a scan operation on the transposed Jacobian matrices (blue squares) of this network and the final gradient vector (yellow squares), as well as how this scan operation can be scaled by BPPSA.

• Generating, storing and processing full Jacobian matrices are usually considered to be prohibitively expensive. However, we observe that the Jacobians of many types of layers (e.g., convolution, activation, max-pooling) can be extremely sparse where we can leverage sparse matrix format [61] to reduce the runtime and storage costs; more importantly, the positions of input-independent zeros in this case are deterministic, which leads to potentially more optimized implementations of sparse matrix libraries. Based on these observations, we develop routines to efficiently generate sparse transposed Jacobians for various operators.

• As a proof of concept, we evaluate BPPSA on a vanilla Recurrent Neural Network (RNN) [23] training with synthetic datasets, as well as a RNN with Gated Recurrent Units (GRU) [19] training with the IRMAS dataset [14]. Our method achieves a maximum 2.75× speedup in terms of the overall (end-to-end) training time, and up to 108× speedup on the backward pass, compared to the baseline BP approach which under-utilizes the GPU. Moreover, we demonstrate that the retraining of pruned networks [29, 62, 32] (e.g., pruned VGG-11 [65]) can also be a practical use case of BPPSA.

[1] Step complexity (detailed in Section 3.6) quantifies the runtime of a parallel algorithm.

Chapter 2

Background and Motivation

2.1 Problem Formulation

We conceptualize a deep learning model as a vector function f composed of sub-functions $\vec{x}_i = f_i(\vec{x}_{i-1}\,;\,\vec{\theta}_i)$:

$$f(.\,;\,\vec{\theta}_1, ..., \vec{\theta}_n) = f_1(.\,;\,\vec{\theta}_1) \circ ... \circ f_n(.\,;\,\vec{\theta}_n) \tag{2.1}$$

where $\vec{\theta}_i, i \in \{1, ..., n\}$ are the parameters of the model. The model is evaluated by an objective function $l(f(\vec{x}_0\,;\,\vec{\theta}_i, i \in \{1, ..., n\}))$, where $\vec{x}_0$ is the initial input to the model. Figure 2.1 visualizes a convolutional neural network conceptualized in this formulation.

To train the model f, a first-order optimizer requires the gradients $\nabla_{\vec{\theta}_i} l$, which are derived from the gradients $\nabla_{\vec{x}_i} l$:

$$[\nabla_{\vec{\theta}_1} l, ..., \nabla_{\vec{\theta}_n} l] \leftarrow [(\frac{\partial \vec{x}_1}{\partial \vec{\theta}_1})^T \nabla_{\vec{x}_1} l, ..., (\frac{\partial \vec{x}_n}{\partial \vec{\theta}_n})^T \nabla_{\vec{x}_n} l] \tag{2.2}$$

where $\frac{\partial \vec{x}_i}{\partial \vec{\theta}_i}$ is the Jacobian matrix of the output $\vec{x}_i$ of f_i to its parameters $\vec{\theta}_i$. To derive $\nabla_{\vec{x}_i} l$ given $\nabla_{\vec{x}_n} l$, BP [60] solves the following recursive equation, from $i = n-1$ to $i = 1$:

$$\nabla_{\vec{x}_i} l \leftarrow (\frac{\partial \vec{x}_{i+1}}{\partial \vec{x}_i})^T \nabla_{\vec{x}_{i+1}} l, \forall i \in \{n-1, ..., 1\} \tag{2.3}$$

where $\frac{\partial \vec{x}_{i+1}}{\partial \vec{x}_i}$ is the Jacobian matrix of the output $\vec{x}_{i+1}$ of f_{i+1} to its input $\vec{x}_i$. Equation 2.2 itself does not have dependency along i; therefore, the computation of $\nabla_{\vec{\theta}_i} l$ can be parallelized if $\nabla_{\vec{x}_i} l$ are available. However, Equation 2.3 imposes a *strong sequential dependency* along i where the computation of $\nabla_{\vec{x}_i} l$ can not begin until the computation of $\nabla_{\vec{x}_{i+1}} l$ finishes, and therefore, hinders the scalability when multiple workers (defined as instances of execution; e.g., a core in a multi-core CPU) are available.

2.2 Prior Works

To increase the utilization of hardware resources in model parallelism, prior works, e.g., PipeDream [48] and GPipe [35], propose to pipeline the computation in the forward and backward passes across devices. However, these solutions are not "silver bullets" to scalability due to the following reasons.

First, both PipeDream [48] and GPipe [35] require storing activations and/or multiple versions of

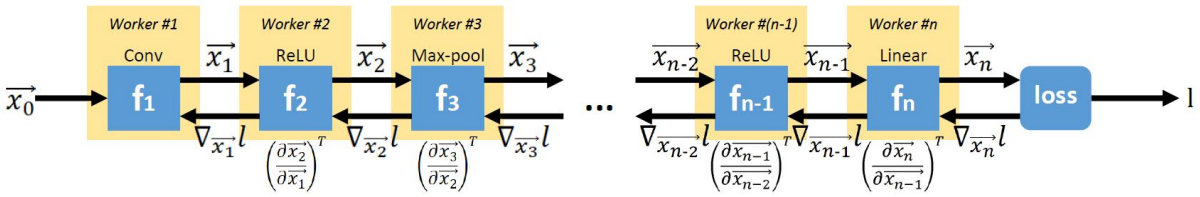


Figure 2.1: A visualization of the formulation in Section 2.1 on convolutional neural networks. Different parts of the model can be distributed to different devices (workers).

weights for all batches that enter the pipeline. Although GPipe's re-materialization [17] can mitigate memory usage, the theoretical per-device space complexity grows linearly with the length of the pipeline (i.e., the number of devices).[1] Thus, the maximum number of devices that pipeline parallelism can support is limited by the memory capacity of a single device (e.g., the GPU global memory), and such memory capacity is not expected to grow significantly in the foreseeable future [46].

Second, if the parameter updates are partially asynchronous as proposed in PipeDream [48], the resulting staleness may effect the convergence for adaptive optimizers such as Adam [37] (Appendix C). If the gradient updates are fully synchronized as proposed in GPipe [35], the "bubble of idleness" between the forward and backward passes increases linearly with the length of the pipeline, which linearly reduces the hardware utilization and leads to diminishing returns.

Our approach fundamentally differs from these key prior works [48, 35] in the following ways. First, instead of following the dependency of BP, we reformulate BP so that scaling is achieved via the Blelloch scan algorithm [13] which is designed for parallelism. Second, the original BP is reconstructed exactly without introducing new sources of errors (e.g., staleness); therefore, our method is agnostic to the exact first-order optimizer being used. Third, our approach becomes more advantageous as the number of devices increases, instead of diminishing returns or hitting scalability limits due to linear per-device space complexity.

2.3 Definition of the Scan Operation

For a *binary* and *associative* operator $\oplus$ with an identity value I, the *exclusive scan* (a.k.a., *prescan*) on an input array $[a_0, a_1, a_2, ..., a_{n-1}]$ produces an output array $[I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, ..., a_0 \oplus ... \oplus a_{n-2}]$ [13]. Parallel scan algorithms have been developed due to the importance of the scan operation and the need to leverage the computing power of emerging parallel hardware systems [33, 13].

[1] Appendix B includes a detailed space complexity analysis.

Chapter 3

Proposed Method: BPPSA

3.1 Back-propagation as a Scan Operation

We define a *binary*, *associative*, and *non-commutative* operator $A \diamond B = BA$, whose identity value is the identity matrix I, where A can be either a matrix or a vector and B is a matrix. Using operator $\diamond$, we can reformulate Equation 2.3 as calculation of the following array:

$$[\nabla_{\vec{x}_n} l, \nabla_{\vec{x}_n} l \diamond (\frac{\partial \vec{x}_n}{\partial \vec{x}_{n-1}})^T, \nabla_{\vec{x}_n} l \diamond (\frac{\partial \vec{x}_n}{\partial \vec{x}_{n-1}})^T \diamond (\frac{\partial \vec{x}_{n-1}}{\partial \vec{x}_{n-2}})^T, ..., \nabla_{\vec{x}_n} l \diamond (\frac{\partial \vec{x}_n}{\partial \vec{x}_{n-1}})^T \diamond ... \diamond (\frac{\partial \vec{x}_2}{\partial \vec{x}_1})^T] \tag{3.1}$$

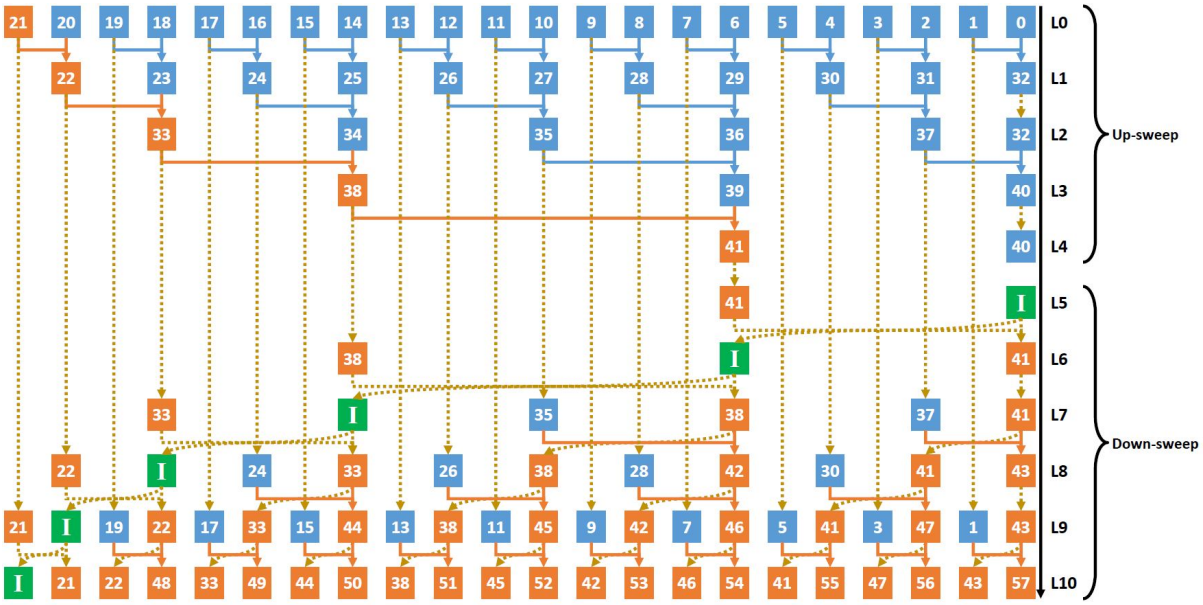
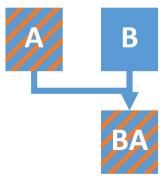


Figure 3.1: Applying our algorithm on the convolutional layers of VGG-11 [65]. Blue, orange, and green squares represent transposed Jacobian matrices, gradient vectors, and *symbolic* identity matrices respectively. Blue solid lines, orange solid lines, and yellow dash lines represent matrix-matrix multiplications, matrix-vector multiplications, and *logical* data movements (that do not always have to be performed explicitly) respectively.

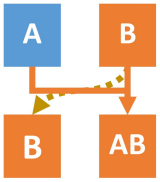
Equation 3.1 can be interpreted as an *exclusive scan* operation of $\diamond$ on the following input array:

$$[\nabla_{\vec{x}_n} l, (\frac{\partial \vec{x}_n}{\partial \vec{x}_{n-1}})^T, (\frac{\partial \vec{x}_{n-1}}{\partial \vec{x}_{n-2}})^T, ..., (\frac{\partial \vec{x}_2}{\partial \vec{x}_1})^T, (\frac{\partial \vec{x}_1}{\partial \vec{x}_0})^T] \tag{3.2}$$

3.2 Scaling Back-propagation with the Blelloch Scan Algorithm



(a) Up-sweep: $B \leftarrow A \diamond B$.



(b) Down-sweep: $A, B \leftarrow B, B \diamond A$.

Figure 3.2: Visualizations of the primitive operations performed in the up-sweep and the down-sweep phases.

We parallelize the computation of Equation 3.1 on multiple workers with the Blelloch scan algorithm [13], formally described in Algorithm 1. The algorithm contains two phases: *up-sweep* and *down-sweep*. As an example, Figure 3.1 visualizes this algorithm applied on the convolutional layers of VGG-11 [65] with levels L0-L4 as the up-sweep and levels L5-L10 as the down-sweep. Only the up-sweep phase contains matrix-matrix multiplications. Due to the *non-commutative* property of the operator $\diamond$, we have to reverse the order of operands for $\diamond$ during the down-sweep phase. This modification is reflected on line 13 of Algorithm 1 and visualized in Figure 3.2b.

Algorithm 1 Modified Blelloch Scan Algorithm

Input: $a = [\nabla_{\vec{x}_n} l, (\frac{\partial \vec{x}_n}{\partial \vec{x}_{n-1}})^T, ..., (\frac{\partial \vec{x}_1}{\partial \vec{x}_0})^T]$ ▷ Input array of Equation 3.2
Output: $a = [I, \nabla_{\vec{x}_n} l, ..., \nabla_{\vec{x}_1} l]$ ▷ $\nabla_{\vec{x}_i} l$ for Equation 2.2; computed **in-place**
1: **for** $d \leftarrow 0$ to $\lceil \log(n+1) \rceil - 2$ **do** ▷ Up-sweep Phase
2: **for all** $i \leftarrow 0$ to $(n - 2^d)$ by 2^{d+1} **do in parallel**
3: $(l, r) \leftarrow (i + 2^d - 1, \min(i + 2^{d+1} - 1, n))$
4: $a[r] \leftarrow a[l] \diamond a[r]$
5: **end for**
6: **end for**
7: $a[n] \leftarrow I$
8: **for** $d \leftarrow \lceil \log(n+1) \rceil - 1$ to 0 **do** ▷ Down-sweep Phase
9: **for all** $i \leftarrow 0$ to $(n - 2^d)$ by 2^{d+1} **do in parallel**
10: $(l, r) \leftarrow (i + 2^d - 1, \min(i + 2^{d+1} - 1, n))$
11: $T \leftarrow a[l]$
12: $a[l] \leftarrow a[r]$
13: $a[r] \leftarrow a[r] \diamond T$ ▷ **Modification:** Reverse the operands of $\diamond$.
14: **end for**
15: **end for**

3.3 Jacobian Matrices in Sparse Format

A full Jacobian matrix $\frac{\partial \vec{x}_{i+1}}{\partial \vec{x}_i}$ of $f_{i+1}(.\,; \vec{\theta}_{i+1})$ can be too expensive to generate, store, and process. In fact, the Jacobian of the first convolution operator in VGG-11 [65] processing a 32×32 image occupies 768 MB of memory if stored as a full matrix, which is prohibitively large. Fortunately, Jacobian matrices

Table 3.1: The sparsity expressions of guaranteed zeros for various operators.

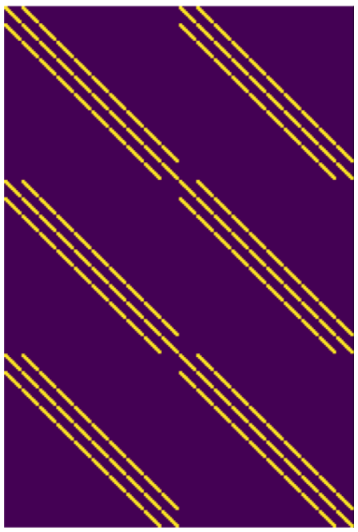
Operator	Filter/Kernel Size	Input Size	Output Size	Sparsity	Examples †	Analytical Generation Speedup §
Convolution	$c_o \times c_i \times h_f \times w_f$	$c_i \times h_i \times w_i$	$c_o \times h_o \times w_o$	$1 - \frac{h_f w_f}{h_i w_i}$ ‡	0.99157	$8.3 \times 10^3\times$
ReLU	N/A	$c \times h \times w$	$c \times h \times w$	$1 - \frac{1}{chw}$	0.99998	$1.2 \times 10^6\times$
Max-pooling	$h_f \times w_f$	$c_i \times h_i \times w_i$	$c_o \times h_o \times w_o$	$1 - \frac{h_f w_f}{c_i h_i w_i}$	0.99994	$1.5 \times 10^5\times$

† The examples of sparsity for the first convolution, ReLU and max-pooling operators of VGG-11 [65] operating on 32×32 images are shown in the second last column of the table.
‡ Approximation when h_i and w_i are much greater than the padding size.
§ Over generating the transposed Jacobian through PyTorch's Autograd [55] one column at a time; measured on a Ryzen Threadripper 1950X [3] machine; averaged across 1000 trials.

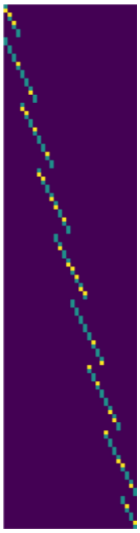
of major operators (such as convolution, ReLU, and max-pooling) are usually extremely sparse as shown in Figure 3.3. In comparison, representing the data contained in the same Jacobian of the aforementioned convolution operator in the Compressed Sparse Row (CSR) [61] format shrinks the memory consumption down to only 6.5 MB. We can observe that there are two reasons for zeros to appear in an operator's Jacobian: *guaranteed zeros* that are input ($\vec{x}_0$) invariant (e.g., zeros that are not on the diagonal of ReLU's Jacobian) and related to the model's architecture; and *possible zeros* that depend on the input (e.g., zeros on the diagonal of ReLU's Jacobian). For any operator, the positions of guaranteed zeros (named as the *sparsity pattern* for brevity) in the Jacobian is *deterministic* with the model architecture and known *ahead of training time*. Thus, mapping non-zero elements in the input matrices to each non-zero element in the product matrix (e.g., calculating the number of non-zeros and index merging in CSR matrix-matrix multiplication [42]) can be performed prior to training and removed from a generic sparse matrix multiplication routine (e.g., cuSPARSE [50]) to achieve significantly better performance during the training phase. As an example, the second last column of Table 3.1 shows the extremely *high sparsity* of *guaranteed zeros* (defined as the fraction over all elements in a matrix) for various operators in VGG-11 [65]. In our implementation, the transposed Jacobian matrices are represented in the CSR format since it is the most straightforward and commonly used sparse matrix format; however, any other sparse matrix format can be used as an alternative, including a potentially more efficient customized sparse matrix format that utilizes the deterministic property of the current sparsity pattern, which we leave to investigate as part of our future work.

3.4 Generating Jacobian Matrix in CSR Analytically

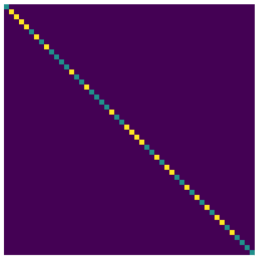
To practically generate the Jacobian for an operator, instead of generating one column at a time either numerically [44] or via automatic differentiation [55, 67], we develop analytical routines to generate the transposed Jacobian directly into the CSR format. Appendix D demonstrates such analytical routines in detail for the convolution, ReLU and max-pooling operators. As proofs of concept for the potential performance benefits, the last column of Table 3.1 shows the speedup on analytical generation of the transposed Jacobians for the aforementioned operators in VGG-11 [65]. As part of our future work to build a mature framework with automatic differentiation capability that performs training via BPPSA, we aim to provide a library that implements a "sparse transposed Jacobian operator" (replacing the backward operator in the case of cuDNN [51]) for each forward operator.



(a) Convolution

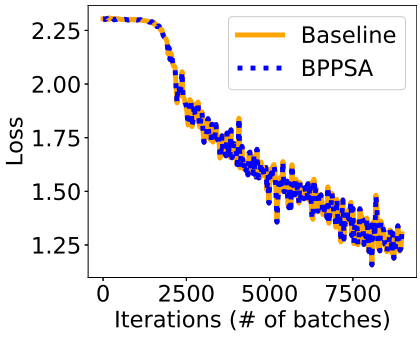


(b) Max-pooling

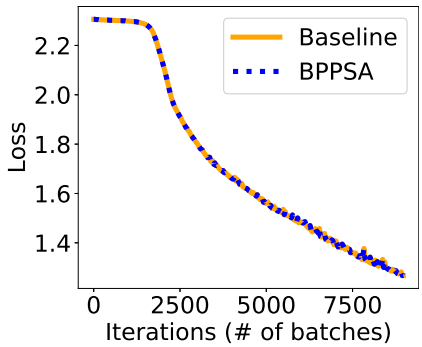


(c) ReLU

Figure 3.3: Transposed Jacobians for various operators. Yellow, cyan and purple dots represents locations of non-zero elements, possible zeros and guaranteed zeros in the matrix.



(a) Training loss per iteration.



(b) Test loss per iteration.

Figure 3.4: Training and test loss per iteration for training LeNet-5 on CIFAR-10. *Baseline* represents training via the PyTorch Autograd, while *BPPSA* represents our method.

3.5 Convergence

Theoretically, our algorithm is a reconstruction of BP instead of an approximation, and hence, expected to reproduce the exact same outputs. However, in practice, numerical differences could be introduced due to the change in the order of matrix multiplications. We apply our algorithm to train LeNet-5 [43] on CIFAR-10 [38] to demonstrate that such numerical differences would not affect model convergence. We use a mini-batch size of 256 and the SGD [58] optimizer with a learning rate of 0.001 and a momentum of 0.9. We seed the experiments with the same constant. Figure 3.4 shows that the orange lines overlap with the blue lines for both training and test losses, which means our algorithm has negligible impact on the convergence compared to the original BP.

3.6 Complexity Analysis

Runtime Complexity We leverage the following definitions to quantify the complexity of a parallel algorithm: (1) *step complexity* (S) which evaluates the minimum number of steps to finish the execution on the critical path (end-to-end) given the number of parallel workers; (2) *per-step complexity* (P) which evaluates the runtime of a single step; and (3) *work complexity* (W) which evaluates the number of total steps executed by all workers. For brevity, we refer to performing the scan operation serially as *linear scan*, which is essentially emulating BP by using the transposed Jacobian and multiplying it with the gradient (as shown in Equation 2.3) explicitly. Assuming the system can be conceptualized as a parallel random-access machine (PRAM) [41], the number of workers is p and the size of the input array in Equation 3.2 is $n+1$, the step and work complexity of our algorithm can be derived as:

$$S_{Blelloch}(n) = \begin{cases} \Theta(\log n) & p > n \\ \Theta(n/p + \log p) & \text{otherwise} \end{cases} \tag{3.3}$$

$$W_{Blelloch}(n) = \Theta(n) \tag{3.4}$$

compared to $S_{Linear}(n) = \Theta(n), W_{Linear}(n) = \Theta(n)$ of the linear scan (which has the same step and work complexity as BP). Therefore, in an ideal scenario where there is an unbounded number of workers with unit per-step complexity, our algorithm reduces the runtime of BP from $\boldsymbol{\Theta(n)}$ to $\boldsymbol{\Theta(\log n)}$. If, however, we consider the difference in per-step complexity between our algorithm ($P_{Blelloch}$) and the baseline (P_{Linear}) due to runtime difference between matrix-matrix and matrix-vector multiplications, our algorithm has a runtime of $\Theta(\log n)P_{Blelloch}$ compared to $\Theta(n)P_{Linear}$ in the baseline. There are two approaches to make our algorithm achieve a lower runtime and better scaling than the baseline. First, we can reduce $P_{Blelloch}$, which is reflected in leveraging the sparsity in the transposed Jacobian as analyzed in Section 4.3 and Section 5.3. Second, without lowering $P_{Blelloch}$, our algorithm can still outperform the baseline if $P_{Blelloch}/P_{Linear} < \Theta(n/\log n)$. This can occur when $n/\log n$ grows larger than the dimension of $\vec{x}_i$. The performance benefit of such case is demonstrated in Section 4.1 and Section 5.1.

Space Complexity Assuming space of storing a transposed Jacobian matrix is bounded by M_{Jacob} and storing $\vec{x}_i$ is bounded by $M_{\vec{x}}$ (note that $M_{Jacob} \ll O({M_{\vec{x}}}^2)$ due to sparse matrix formats; both M_{Jacob} and $M_{\vec{x}}$ are not functions of p), in our method, each worker requires the space of $M_{Blelloch}(n) = \Theta(max(\frac{n}{p}, 1))M_{Jacob}$ which reduces as p increases until a constant M_{Jacob}, comparing to $M_{Pipeline} = \Theta(\frac{n}{p} + p)M_{\vec{x}}$ for pipeline parallelism which increases linearly as p increases. Therefore, our method does not have the limitation of scalability on p, as long as each worker has the memory capacity of at least M_{Jacob}.

Chapter 4

Methodology

Although training deep learning models on thousands of devices has been proven feasible in the industry [45, 28], setting up an experiment for such a large number of devices would require a data center of GPUs and re-implementing/optimizing our entire experiment framework, which requires both monetary and engineering resources out of reach for a typical academic research group. Thus, we set up small-scale experiments that can reflect the large-scale workloads to demonstrate the potential performance benefits of our method.

Environment Setup Our experiments are performed on two platforms with RTX 2070 [52] and RTX 2080Ti [53] respectively (both are Turing architecture GPUs) whose specifications are listed in Table 4.1[1].

Baselines We evaluate our method against *PyTorch Autograd* [55] with cuDNN backend [51] which is a widely adopted and state-of-the-art implementation of BP.

Metrics We use three metrics to quantify the results from our evaluations: (1) *wall-clock time* which measures the system-wide actual time spent on a process, (2) *speedup* which is the ratio of the wall-clock time spent on the baseline over our method, and (3) *FLOP* which represents the number of floating-point operations executed.

We leverage three types of benchmarks to empirically evaluate BPPSA: (1) an end-to-end benchmark of a vanilla RNN training on synthesized datasets to demonstrate the scalability benefits of BPPSA on long sequential dependency; (2) an end-to-end benchmark of a GRU training on the IRMAS dataset [14] to demonstrate the potential of BPPSA on a more realistic workload; and (3) a micro-benchmark of a pruned VGG-11 [65] to evaluate the feasibility of using sparse matrix format to reduce the per-step complexity of BPPSA.

4.1 RNN End-to-end Benchmark

We set up experiments of training an RNN [23] on sequential data, which is a classical example of workloads where the runtime performance (in terms of the wall-clock time) is limited due to the *strong sequential dependency*. The large number of operators n is modeled through a large sequence length T. The large number of workers p is reflected in the total number of CUDA threads that can be executed

[1] Appendix G includes results on V100 [54].

Table 4.1: Specifications of our experiment platforms.

GPU	RTX 2070	RTX 2080Ti
Number of Streaming Multiprocessors (SMs)	36	68
NVIDIA GPU Driver	430.50	440.33.01
CUDA [49]	10.0.130	10.0.130
cuDNN [18]	7.5.1	7.6.2
PyTorch [55]	1.1.0	1.2.0
CPU	Ryzen Threadripper 1950X [3]	EPYC 7601 [2]
Host Memory	32GB, 2400MHz	128GB, 2133MHz
Linux Kernel [68]	4.15.0-76	4.19.49

concurrently in all SMs of a single GPU, which we model through the fraction of GPU per sample (derived as one over the mini-batch size B).

Datasets We synthesize the datasets $X = \{(x^{(k)}, c^{(k)})\}$ of 32000 training samples (i.e., $k \in \{0, 31999\}$) for the task of *bitstream classification*. Each sample consists of a class label $c^{(k)}$ where $c^{(k)} \in \{0, ..., 9\}$ and a bitstream $x^{(k)}$ where the value $x_t^{(k)}$ at each time step $t \in \{0, ..., T-1\}$ is sampled from the Bernoulli distribution [11, 25]:

$$x_t^{(k)} \sim Bernoulli(0.05 + c^{(k)} \times 0.1) \tag{4.1}$$

Equivalently, each bitstream $x^{(k)}$ can be viewed as a binomial experiment [11, 25] of class $c^{(k)}$. The objective of this task is to classify each bitstream $x^{(k)}$ into its corresponding class $c^{(k)}$ correctly. We synthesize eight datasets with different T, where T increases up to 30000. In reality, long sequences of input can often be found in audio signals such as speech [47, 8, 7] or music [12, 10].

Model We leverage a vanilla RNN [23] (described in Equation 4.2) to solve the aforementioned task, since RNN is an intuitive, yet classical, deep learning model and often used to process sequential data:

$$\vec{h}_t^{(k)} = tanh(W_{ih} x_t^{(k)} + \vec{b}_{ih} + W_{hh} \vec{h}_{t-1}^{(k)} + \vec{b}_{hh}) \tag{4.2}$$

where $\vec{h}_t^{(k)}, \vec{b}_{ih}, \vec{b}_{hh} \in \mathbb{R}^{20}$. The output classes are predicted via the softmax function [16] applied on a linear transformation to the last hidden states $\vec{h}_{T-1}^{(k)}$. The cross entropy [27] is used as the loss function which is optimized in training via the Adam optimizer [37] with the learning rate of 1×10^{-5}. The computation of $\nabla_{\vec{h}_t^{(k)}} l$ during the backward pass carries the *strong sequential dependency* which is the target for acceleration via BPPSA.

Implementation We implement our modified version of the Blelloch scan algorithm as two custom CUDA kernels for the up-sweep and down-sweep phases respectively, along with a few other CUDA kernels for the preparation of the input transposed Jacobian matrices. Each level during the up-/down-sweep is associated with a separate CUDA kernel launch (in the same CUDA stream); therefore, synchronization is ensured between two consecutive levels. Each thread block is responsible for the $\diamond$ operation (i.e. multiplication in reverse) of two matrices as well as moving the intermediate results, and the shared memory is leveraged for caching input and output matrices. Our custom CUDA kernels are integrated into the Python front-end where the RNN and the training procedure are defined through PyTorch's Custom C++ and CUDA Extensions [26]. For the forward pass and the baseline of PyTorch Autograd

Table 4.2: MFCC configurations and the resulting feature sizes (represented as the number of frames F multiplied by the number of coefficients C) for the S, M and L sets.

Set Name	S	M	L
MFCC Coefficients	20	13	7
FFT Window Length	4096	2048	1024
Hop Length	512	256	128
Resulting Input Features ($F \times C$)	259×38	517×24	1034×12

[55], we simply plug in the PyTorch's `RNN` module [57] which calls into the cuDNN's RNN implementations (`cudnnRNNForwardTraining` and `cudnnRNNBackwardData`) [51]; therefore, our baseline is already much faster than implementing RNN in Python using PyTorch's `RNNCell` module [57] due to GEMM streaming and kernel fusions [5].

4.2 GRU End-to-end Benchmark

To extend the aforementioned RNN end-to-end benchmark to a more realistic setting, we evaluate the runtime performance of training a GRU [19] on the IRMAS [14] dataset for the task of *instrument classification* based on audio signals.

Datasets We preprocess the IRMAS dataset and compute the mel-frequency cepstral coefficients (MFCC) [21] for each waveform audio sample via LibROSA's [15] MFCC implementation. With different MFCC configurations as listed in Table 4.2, the preprocessing results in three sets (S, M and L), reflecting the trade-off between the temporal and frequency resolutions. For all samples, we normalize the values of each coefficient across the frames to have zero mean and unit variance. We remove the first coefficient because it only represents the average power of the audio signal.

Model Since instrument classification is a more complex task than the synthetic workloads in Section 4.1, a GRU [19] (described in Equations 4.3) is used in this set of experiments.

$$\begin{aligned}
\vec{r}_t &= \sigma(W_{ir}\vec{x}_t + \vec{b}_{ir} + W_{hr}\vec{h}_{t-1} + \vec{b}_{hr}) \\
\vec{z}_t &= \sigma(W_{iz}\vec{x}_t + \vec{b}_{iz} + W_{hz}\vec{h}_{t-1} + \vec{b}_{hz}) \\
\vec{n}_t &= tanh(W_{in}\vec{x}_t + \vec{b}_{in} + \vec{r}_t \circ (W_{hn}\vec{h}_{t-1} + \vec{b}_{hn})) \\
\vec{h}_t &= (1 - \vec{z}_t) \circ \vec{n}_t + \vec{z}_t \circ \vec{h}_{t-1}
\end{aligned} \tag{4.3}$$

where $\vec{h}_t \in \mathbb{R}^{20}$, $t \in \{0, ..., F-1\}$ and $\vec{x}_t \in \mathbb{R}^C$. Since cuDNN's GRU implementation [5] is closed source, we are unable to generate the transposed Jacobians efficiently (Appendix E), which leads to significant *overhead* in the forward pass. However, such overhead could potentially be reduced if cuDNN's source code becomes publicly available. Other settings are the same as Section 4.1 with the exception of a 3×10^{-4} learning rate.

Implementation We directly use PyTorch's `GRU` module [57] which calls into the cuDNN's GRU implementations (`cudnnRNNForwardTraining` and `cudnnRNNBackwardData` with `CUDNN_GRU`) [51]. We reuse the same CUDA implementation of the Blelloch scan algorithm as in Section 4.1.

4.3 Pruned VGG-11 Micro-benchmark

Despite the recent advances in network pruning algorithms [29, 62, 32], there is no existing widely adopted software or hardware platform that can exploit performance benefits from pruning, as most techniques are evaluated through "masking simulation" which leads to the same (if not worse) runtime and memory usage. In contrast, in this work, we discover that the *retraining of pruned networks* could benefit from BPPSA due to the following reason: Since the values in the Jacobian of a convolution operator only depend on the filter weights (Appendix D.1), pruning the weights can lead to a higher sparsity in the Jacobian, which then reduces the per-step complexity of sparse matrix-matrix multiplications.

To evaluate the feasibility of leveraging the sparsity in the transposed Jacobian of each operator, we set up a benchmark with VGG-11 [65]: training on CIFAR-10 [38], pruning away 97% of the weights in all convolution and linear operators using the technique proposed by See et al. [62], and retraining the pruned network. We choose this pruning percentage so that a similar validation accuracy is reached (90.1% v.s. 88.9%) after retraining for the same number of epochs (100) as training. We then apply BPPSA on the convolutional layers of VGG-11 to compute Equation 2.3.

Since the sparsity pattern of the transposed Jacobian can be determined ahead of training time from the model architecture (as we show in Section 3.3), existing sparse matrix libraries which target generic cases are sub-optimal for our method. For example, cuSPARSE [50] calculates the number of non-zeros in the product matrix and merges the indices of the input matrices before it can perform the multiplication. Such preparations do not need to repeat across iterations in BPPSA's case and could be performed ahead of time due to the deterministic nature of the sparsity pattern. This, in turn, saves considerable amount of execution time. Therefore, due to the lack of a fair implementation, we perform the evaluation by calculating the FLOPs needed for each step in our method and the baseline implementations through *static analysis*.

Chapter 5

Evaluation

In this section, we present the results from the RNN end-to-end benchmark (Section 4.1), the GRU end-to-end benchmark (Section 4.2) and the pruned VGG-11 micro-benchmark (Section 4.3).

5.1 RNN End-to-end Benchmark

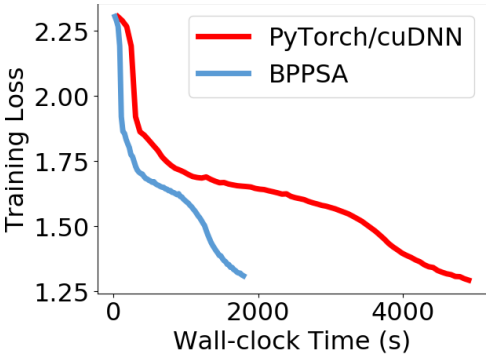
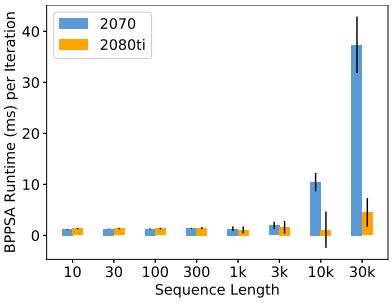


Figure 5.1: Training loss across wall-clock time when the RNN is trained via BPPSA (blue curve) and the PyTorch Autograd baseline with cuDNN's RNN backend (red curve).

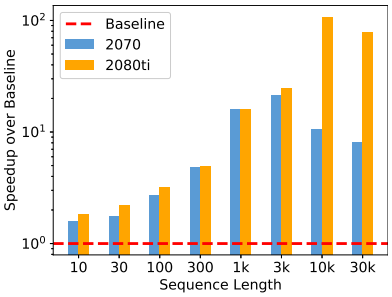
Figure 5.1 shows the training curves of loss values with respect to wall-clock time when we train the RNN for 80 epochs on the RTX 2070 GPU with the mini-batch size $B = 16$ and the sequence length $T = 1000$. This experiment can be viewed as the simplest mechanism to process sequential data such as audio signals. We observe that the blue curve (BPPSA) is roughly equivalent to the red curve (PyTorch/cuDNN baseline) scaled down by 63% along the horizontal (time) axis. We conclude that, in this setting, training the RNN through BPPSA reconstructs the original BP algorithm while achieving a 2.73× speedup on the overall training time and 16× on the BP runtime.

Sensitivity Analysis We measure the performance variation as the sequence length T and the fraction of GPU per sample ($1/B$) vary, since those two parameters represent the total number of operators n and the number of workers p respectively — key variables in the theoretical runtime of our method. To estimate the speedups, we measure the wall-clock time of training via BPPSA for a single epoch, and take the average of 20 measurements from different epochs. We then compare against training via the PyTorch/cuDNN baseline measured in the same way. We can also derive the backward pass runtime by measuring the wall-clock time of the training procedure without actually performing the backward pass, and subtracting from the total runtime (including the overhead of preparing the input transposed Jacobians).

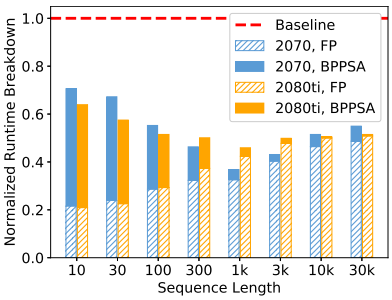
Figure 5.2a, Figure 5.2b and Figure 5.2c show how changing the sequence length T affects the backward pass and overall training time. We make three observations from these figures. First, our method scales as n increases when n is relatively in the same range as p. Second, when n increases to be



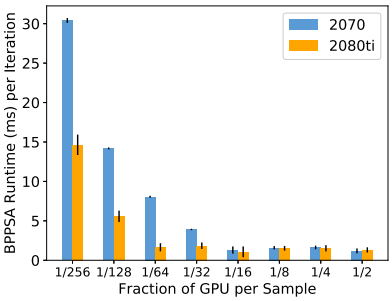
(a) The BPPSA runtime per iteration as the sequence length T increases.



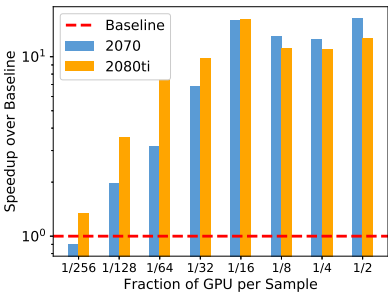
(b) The backward pass speedup as the sequence length T increases.



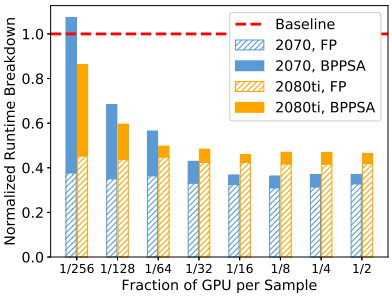
(c) The runtime breakdown as the sequence length T increases.



(d) The BPPSA runtime per iteration as the fraction of GPU per sample ($1/B$) increases.



(e) The backward pass speedup as the fraction of GPU per sample ($1/B$) increases.



(f) The runtime breakdown as the fraction of GPU per sample ($1/B$) increases.

Figure 5.2: We report the BPPSA backward pass latency per iteration (Figure 5.2a and Figure 5.2d), the backward pass speedups (Figure 5.2b and Figure 5.2e) of BPPSA over the baseline, as well as the runtime (normalized by the baseline) breakdowns to demonstrate the overall speedups (Figure 5.2c and Figure 5.2f). The fraction of GPU per sample (which reflects the number of workers p) is computed as one over the batch size B. *FP* refers to the forward pass. The standard deviations of the BPPSA latency are reported as black lines in Figure 5.2a and Figure 5.2d.

much larger than p, the performance starts to be bounded by p. Third, even in the range of overly large n, our method still achieves better utilization on massively parallel hardware than the baseline.

Figure 5.2d, Figure 5.2e and Figure 5.2f show how changing the fraction of GPU per sample ($1/B$) affects the backward pass and overall training time. We can conclude that BPPSA scales as the "effective" number of workers p per sample increases (equivalently, as the batch size B decreases, since the total number of SMs in the GPU is constant). In reality, determining the appropriate mini-batch size can be nontrivial: training with large batch can lead to "generalization gap" [36], while training with small batch would under-utilize the hardware resources and lead to longer training time. Here, BPPSA can be viewed as offering an alternative to train with smaller mini-batch while utilizing the hardware resources more efficiently than BP.

By comparing the speedup in Figure 5.2b and Figure 5.2e between RTX 2070 and RTX 2080Ti (RTX 2080Ti has a higher number of SMs than RTX 2070; 68 vs. 36 [53, 52]), we can observe that: (1) BPPSA achieves its maximum speedup at a higher sequence length on RTX 2080Ti than RTX 2070; (2) as the batch size B increases, the speedup of BPPSA on RTX 2080Ti drops at a slower rate than RTX 2070. These two observations, together with Figure 5.2a and Figure 5.2d where the BPPSA latency per iteration on RTX 2080Ti is lower than RTX 2070, are consistent with the aforementioned conclusions

regarding the performance variation with the number of workers p. We can observe a maximum of 108× backward pass speedup on RTX 2080Ti and a maximum of 2.75× overall speedup on RTX 2070 (the highest backward pass speedup might not lead to the highest overall speedup due to different forward pass runtime on which BPPSA has no impact).

5.2 GRU End-to-end Benchmark

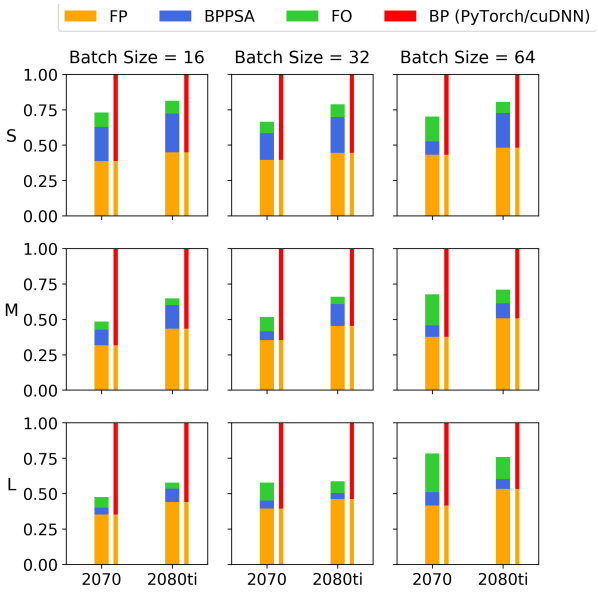


Figure 5.3: The runtime breakdowns in the GRU end-to-end benchmark as the dataset type (S, M, L) and batch sizes B vary. *FP* represents the forward pass; *FO* represents the forward pass overhead of computing the transposed Jacobians; *BP* represents the BP baseline; and *BPPSA* represents the backward pass via BPPSA. The measurements are normalized by the total runtime of the baseline (FP + BP).

We include the training curves of loss values with respect to the wall-clock time when we train the GRU with the preprocessed datasets in Appendix F. They leads to the same conclusions as the ones in Section 5.1.

Sensitivity Analysis To perform an analysis similar to Section 5.2, we only need to vary the batch size B since the preprocessed dataset type (S, M, L) already reflects the sequence length T. However, since the overhead of computing the transposed Jacobians during the forward pass cannot be neglected (as mentioned in Section 4.2), to achieve a deeper understanding of the performance variation, we demonstrate the runtime breakdowns among the forward pass, the backward pass and the overhead. We can derive the overhead by taking the difference in the runtime of the training procedures without actually performing the backward pass between BPPSA and the PyTorch/cuDNN baseline. The measurements are averaged across 100 epochs.

Figure 5.3 shows how the sequence length T and batch size B affect the runtimes of the forward pass, the backward pass and the overhead. We make two observations from this figure. First, our method achieves a higher speedup on the backward pass as T increases (changing the preprocessed dataset from S to L), which reinforces the observation from Section 5.1 that our method scales well as the total number of operators n increases. Second, since the maximum sequence length (1034) is not as extreme as in Section 5.1, the backward pass runtime of BPPSA is less affected than the overhead by B and the GPU model, which means n is still within the same range as the number of workers p in this set of experiments. The maximum overall speedup and backward pass speedup (excluding the overhead) are 2.36× and 13.4× respectively.

5.3 Pruned VGG-11 Micro-benchmark

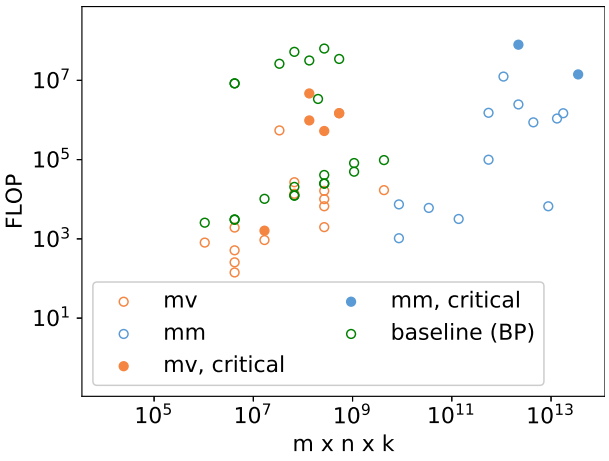


Figure 5.4: Measuring FLOP for each step when retraining pruned VGG-11 on CIFAR-10. *mv* and *mm* represent matrix-vector and matrix-matrix multiplications in BPPSA respectively. *critical* indicates that the step is on the critical path. The x-axis represents the theoretical runtime complexity of the step **if** the transposed Jacobian were not encoded in a sparse format. The green circles represent the FLOP estimated for each "gradient operator" in the BP baseline.

Since the sparsity of the product matrix might reduce after each multiplication, the per-step complexity might increase as the up-sweep phase progresses into deeper levels. Fortunately, we can adopt BPPSA to balance the number of levels in the up-/down-sweep phases according to the sparsity of the products on each level to achieve an overall speedup. Specifically, in this experiment, BPPSA performs the up-sweep from L0 to L2 (consistent with the notations in Figure 3.1), calculates the partial results that are needed for the down-sweep phase through linear scan, and then performs the down-sweep from L7 to L10.

Assuming the sparse transposed Jacobian matrices are encoded in the CSR format, Figure 5.4 shows the calculated FLOP of each step in BPPSA and each "gradient operator" in the baseline (BP) for re-training pruned VGG-11 on CIFAR-10. We observe that the green circles (baseline) have similar expected performance as the other circles (BPPSA). Thus, we can conclude that exploiting the sparsity in the transposed Jacobian is an efficient strategy that reduces the per-step complexity of our method $P_{Blelloch}$ to a level similar with the baseline P_{Linear}. This strategy makes the overall scalability to be "ensured" algorithmically.

Chapter 6

Conclusion

In this work, we explore a novel direction to scale BP by challenging its fundamental limitation of the *strong sequential dependency*. We reformulate BP into a *scan* operation which is scaled by our modified version of the Blelloch scan algorithm. Our proposed algorithm, BPPSA, achieves a logarithmic runtime complexity rather than linear. In addition, BPPSA has a constant per-device space complexity; hence, its scalability is not limited by the memory capacity of each device. In our detailed evaluations, we demonstrate that overall speedup can be already achieved in two important use cases. First, for the case where there is a long dependency in BP, we evaluate BPPSA on two different sets of benchmarks: (1) training a RNN with synthetic datasets where our method achieves up to 2.75× speedup on the overall (end-to-end) training time and 108× speedup on the backward pass alone; and (2) training a GRU with the IRMAS dataset [14] where our method achieves up to 2.36× overall speedup and 13.4× speedup on the backward pass alone. Second, we can reduce the per-step complexity by leveraging the sparsity in the Jacobian itself. To this end, we develop efficient routines to generate the transposed Jacobian in the CSR format, and demonstrate that the retraining of pruned networks can potentially benefit from BPPSA (as we show for a pruned VGG-11 benchmark when re-training on the CIFAR-10 dataset). We hope that our work will inspire radically new ideas and designs to improve distributed DNN training beyond the existing theoretical framework.

Bibliography

[1] Amazon. Amazon ec2 p3 instances. `https://aws.amazon.com/ec2/instance-types/p3/`, 2019. Accessed: 2019-08-28.

[2] AMD. Epyc 7601 dual amd server processors. `https://www.amd.com/en/products/cpu/amd-epyc-7601`, 2019. Accessed: 2019-08-28.

[3] AMD. Ryzen threadripper 1950x processor. `https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-1950x`, 2019. Accessed: 2019-08-28.

[4] Dario Amodei, Danny Hernandez, Girish Sastry, Jack Clark, Greg Brockman, and Ilya Sutskever. Ai and compute. `https://openai.com/blog/ai-and-compute/`, 2018. Accessed: 2019-08-28.

[5] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. Optimizing performance of recurrent neural networks on gpus. *CoRR*, abs/1604.01946, 2016.

[6] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. Mcm-gpu: Multi-chip-module gpus for continued performance scalability. *ACM SIGARCH Computer Architecture News*, 45(2):320–332, 2017.

[7] Jon Barker, Shinji Watanabe, Emmanuel Vincent, and Jan Trmal. The fifth 'chime' speech separation and recognition challenge: Dataset, task and baselines. *CoRR*, abs/1803.10609, 2018.

[8] Timo Baumann, Arne Köhn, and Felix Hennig. The spoken wikipedia corpus collection: Harvesting, alignment and an application to hyperlistening. *Language Resources and Evaluation*, Jan 2018.

[9] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *CoRR*, abs/1802.09941, 2018.

[10] Kirell Benzi, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. FMA: A dataset for music analysis. *CoRR*, abs/1612.01840, 2016.

[11] J. Bernoulli. *Jacobi Bernoulli, ... Ars conjectandi, opus posthumum. Accedit Tractatus de seriebus infinitis, et epistola Gallice scripta De ludo pilae reticularis.* impensis Thurnisiorum, fratrum, 1713.

[12] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[13] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[14] J.J. Bosch, Jordi Janer, Ferdinand Fuhrmann, and Perfecto Herrera. A comparison of sound segregation techniques for predominant instrument recognition in musical audio signals. *Proceedings of the 13th International Society for Music Information Retrieval Conference, ISMIR 2012*, pages 559–564, 01 2012.

[15] Brian McFee, Colin Raffel, Dawen Liang, Daniel P.W. Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. librosa: Audio and Music Signal Analysis in Python. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 18 – 24, 2015.

[16] John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Françoise Fogelman Soulié and Jeanny Hérault, editors, *Neurocomputing*, pages 227–236, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[17] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[18] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[19] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[20] Cody A. Coleman, Deepak Narayanan, Daniel Kang, Tian Jiao Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chester Beatty Re, and Matei A. Zaharia. Dawnbench : An end-to-end deep learning benchmark and competition. 2017.

[21] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(4):357–366, August 1980.

[22] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009.

[23] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[24] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376. ACM, 2011.

[25] M.J. Evans and J.S. Rosenthal. *Probability and Statistics: The Science of Uncertainty*. W. H. Freeman, 2009.

[26] Peter Goldsborough. Custom c++ and cuda extensions. `https://pytorch.org/tutorials/advanced/cpp_extension.html`, 2019. Accessed: 2019-08-28.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[28] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[29] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1135–1143. 2015.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016.

[32] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. *CoRR*, abs/1707.06168, 2017.

[33] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[34] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[35] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965, 2018.

[36] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *CoRR*, abs/1609.04836, 2016.

[37] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

[38] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[39] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, 2012.

[41] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95 – 132, 1990.

[42] Rakshith Kunchum, Ankur Chaudhry, Aravind Sukumaran-Rajam, Qingpeng Niu, Israt Nisa, and P. Sadayappan. On improving performance of sparse matrix-matrix multiplication on gpus. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 14:1–14:11, 2017.

[43] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[44] John Mahaffy. Numerical evaluation of jacobians - personal.psu.edu. `http://www.personal.psu.edu/jhm/ME540/lectures/NumJacobian.html`, 2019. Accessed: 2019-08-28.

[45] MLPerf. Mlperf training v0.6 results. `https://mlperf.org/training-results-0-6/`, 2019. Accessed: 2019-08-28.

[46] Onur Mutlu. Memory scaling: A systems architecture perspective. pages 21–25, 05 2013.

[47] Arsha Nagrani, Joon Son Chung, and Andrew Zisserman. Voxceleb: a large-scale speaker identification dataset. *CoRR*, abs/1706.08612, 2017.

[48] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Granger, Phil Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *SOSP 2019*, October 2019.

[49] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[50] NVIDIA. cusparse :: Cuda toolkit documentation, 2018. [Accessed: 2018-11-06].

[51] NVIDIA. cudnn developer guide :: Deep learning sdk documentation. `https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html`, 2019. Accessed: 2019-08-28.

[52] NVIDIA. Geforce rtx 2070 graphics card | nvidia. `https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2070/`, 2019. Accessed: 2019-08-28.

[53] NVIDIA. Geforce rtx 2080 ti graphics card | nvidia. `https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/`, 2019. Accessed: 2019-08-28.

[54] NVIDIA. Nvidia v100 tensor core gpu. `https://www.nvidia.com/en-us/data-center/v100/`, 2019. Accessed: 2019-08-28.

[55] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[56] PyTorch. Torch.nn. `https://pytorch.org/docs/stable/nn.html#torch.nn.MaxPool2d`, 2019. Accessed: 2019-05-23.

[57] PyTorch. torch.nn — pytorch master documentation. `https://pytorch.org/docs/stable/nn.html`, 2019. Accessed: 2019-08-28.

[58] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Netw.*, 12(1):145–151, January 1999.

[59] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. Virtualizing deep neural networks for memory-efficient neural network design. *CoRR*, abs/1602.08124, 2016.

[60] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, 1988.

[61] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[62] Abigail See, Minh-Thang Luong, and Christopher D. Manning. Compression of neural machine translation models via pruning. *CoRR*, abs/1606.09274, 2016.

[63] Christopher J. Shallue, Jaehoon Lee, Joseph M. Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. *CoRR*, abs/1811.03600, 2018.

[64] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10414–10423. Curran Associates, Inc., 2018.

[65] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

[66] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[67] PyTorch Forums. How to compute jacobian matrix in pytorch? `https://discuss.pytorch.org/t/how-to-compute-jacobian-matrix-in-pytorch/14968`, 2019. Accessed: 2019-08-28.

[68] Linus Torvalds. Linux kernel source tree. `https://github.com/torvalds/linux`, 2019. Accessed: 2019-08-28.

[69] Gaël Varoquaux, Emmanuelle Gouillart, and Olav Vahtras. Compressed sparse row format (csr), 2019. Accessed: 2019-05-23.

[70] Shang Wang, Yifan Bai, and Gennady Pekhimenko. Bppsa: Scaling back-propagation by parallel scan algorithm. In *Proceedings of Machine Learning and Systems*, volume 2, pages 451–469, 2020.

[71] Eric W. Weisstein. "jacobian." from mathworld–a wolfram web resource. `http://mathworld.wolfram.com/Jacobian.html`, 2019. Accessed: 2019-08-28.

[72] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization, IISWC 2018*, pages 88–100, 2018.

Summary of Appendices

To keep the main text concise for the audience, several important details are moved to the appendices below that include the following content:

- Appendix A describes our open-sourced artifact and explains how to reproduce all major experiments in this work.
- Appendix B performs an analysis on the space complexity for one of the key prior works, GPipe [35]. Appendix C describes our initial attempts to analyze PipeDream's [48] behavior on VGG-16 with the Adam optimizer (instead of a vanilla SGD). We use these two appendices to support our arguments in Section 2.2.
- Appendix D lists the routines that we developed to generate the transposed Jacobians for various operators directly into the CSR format. This is the complementary material for Section 3.4.
- Appendix E shows how to derive the transposed Jacobians for GRU, and demonstrates the source of the overhead in the forward pass for our GRU end-to-end benchmark (Section 4.2).
- Appendix F includes the training curves for our GRU end-to-end benchmark, which serves as a complementary material to Section 5.2.
- Appendix G reports the additional hardware sensitivity results on the Volta-based V100 GPU [54] to validate BPPSA potential across different GPU generations.

Appendix A

Artifact Appendix

A.1 Abstract

We provide the source code, scripts and data that corresponds to Section 4 as our artifact to reproduce the results in Section 5 and Table 3.1. We require an x86-64 based machine with at least one NVIDIA GPU to evaluate the artifact, and NVIDIA Container Toolkit is the only software dependency to prepare. After the installation, the entire workflow (from building the needed Docker image to plotting the final results) is automated by a single `workflow.sh` script. Although the the exact numerical results produced by the artifact might vary across hardware platforms, the general trends and conclusions should be similar to the results reported in this paper.

A.2 Artifact Check-list (Meta-information)

- **Algorithm:** Back-propagation by Parallel Scan Algorithm (**BPPSA**)
- **Program:** RNN and GRU end-to-end benchmarks (Section 4.1, 4.2); a VGG-11 micro-benchmark Section 4.3. All benchmarks are public, included, and automated.
- **Compilation:** GCC 7.4.0 and CUDA 10.0 are recommended, included, and tested, although other versions of GCC and CUDA might work as well.
- **Binary:** Scripts included to build binaries from the source code.
- **Data set:** The synthetic datasets (Section 4.1) and IRMAS are included; approximately 4.5 GB in total.
- **Run-time environment:** The main software dependency is NVIDIA Container Toolkit (`https://github.com/NVIDIA/nvidia-docker`) which dictates the OS requirements. We recommend and tested on Ubuntu 18.04.
- **Hardware:** An x86-64 based machine with at least one NVIDIA GPU and internet access. No SUDO access needed.
- **Run-time state:** No contentions on hardware resources (CPU, GPU, RAM, PCIe) with other processes.
- **Execution:** Around 57 hours in total on the RTX 2080Ti platform listed in Table 4.1.
- **Metrics:** Wall-clock time, speedup, and FLOP (Section 4).
- **Output:** Figures that are similar to Figure 5.1, 5.2, 5.3, 5.4 and F.1; Text that contains speedups similar to the last column of Table 3.1.

- **Experiments:** A single script is provided that automates the entire workflow.
- **How much disk space required (approximately)?:** Approximately a total of 19.7 GB is needed.
- **How much time is needed to prepare workflow (approximately)?:** Around one hour to install NVIDIA Container Toolkit with its dependencies.
- **How much time is needed to complete experiments (approximately)?:** Refer to the **Execution** part above.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT.
- **Data licenses (if publicly available)?:** MIT.
- **Workflow framework used?:** No.
- **Archived (provide DOI)?:** 10.5281/zenodo.3605368

A.3 Description

The source code is publicly available on GitHub (`https://github.com/UofT-EcoSystem/BPPSA-open`) and Zenodo (`https://doi.org/10.5281/zenodo.3605368`). The source code and scripts only require 37.9 kB of disk space. However, the `workflow.sh` script builds a 7.7 GB Docker image, then downloads and unzips 12 GB of data.

A.3.1 Hardware Dependencies

The hardware specifications used are listed in Table 4.1. In general, an x86-64 based machine with at least one NVIDIA GPU and internet access is required.

A.3.2 Software Dependencies

Although it is possible to run the experiments natively on the host machine (and, in fact, this is how our RTX 2070 platform was set up), we do not recommend this approach since installing the dependencies can be tedious, non-portable, and unsafe (due to the SUDO access requirements). Instead, we package all of the original dependencies into a Docker image which can be built natively by the `workflow.sh` script. Therefore, our artifact only requires NVIDIA Container Toolkit (`https://github.com/NVIDIA/nvidia-docker`). We recommend and tested on Ubuntu 18.04, however, it is possible to evaluate the artifact on other Linux distributions that NVIDIA Container Toolkit supports as well.

A.3.3 Datasets

The `workflow.sh` script downloads all the required datasets automatically.

A.3.4 Models

The RNN (Section 4.1) and GRU (Section 4.2) are included. The transposed Jacobians of VGG-11 are downloaded by the `workflow.sh` script.

A.4 Installation

Assuming the hardware listed in Section A.3.1 is available, the following steps are needed to perform the installation:

1. Clone the project by `git clone https://github.com/UofT-EcoSystem/BPPSA-open.git`.
2. Install a NVIDIA GPU driver that is compatible with the GPU, the CUDA version (10.0 recommended) and the NVIDIA Container Toolkit.
3. Install Docker Engine - Community (`https://docs.docker.com/install/`), then configure the `docker` group to use Docker as a non-root user. `https://docs.docker.com/install/linux/linux-postinstall/`).
4. Install NVIDIA Container Toolkit (`https://github.com/NVIDIA/nvidia-docker`).

We provide the `install.sh` script as a reference to the above steps 2 to 4.

A.5 Experiment Workflow

We provide the `workflow.sh` script that automates the entire workflow consisting of the following stages:

1. Build the Docker image used across experiments.
2. Download and unzip the synthetic datasets (Section 4.1) and IRMAS.
3. Execute the RNN (Section 4.1) and GRU (Section 4.2) end-to-end benchmarks.
4. Plot the results for the RNN and GRU end-to-end benchmarks.
5. Evaluate the speedups for sparse transposed Jacobian generation (Section 3.4).
6. Download the sparse transposed Jacobians of a regular and pruned VGG-11.
7. Execute the VGG-11 micro-benchmark (Section 4.3) and plot the results.

After the installation in Section A.4, the user only need to run the command "`./workflow.sh`" in the project root directory, which takes around 57 hours on our reference plarform with the RTX 2080Ti GPU (Table 4.1).

A.6 Evaluation and Expected Result

After `./workflow.sh` finishes, a `results/` directory is created to contain the following results:

- `fig_5.1.png` corresponding to Figure 5.1.
- `fig_5.2_a.png`, `fig_5.2_b.png`, `fig_5.2_c.png`, `fig_5.2_d.png`, `fig_5.2_e.png` and `fig_5.2_f.png` corresponding to Figure 5.2.
- `fig_5.3.png` corresponding to Figure 5.3.

- `fig_5.4.png` corresponding to Figure 5.4.
- `fig_F.1.png` corresponding to Figure F.1.
- `table_3.1_last_column.txt` corresponding to the last column of Table 3.1.

The exact numerical results might vary across hardware platforms, but the general trends should be similar to the results presented in this paper where we conducted the experiments on platforms with the RTX 2070 and 2080Ti GPUs (Table 4.1). In addition, the speedups of BPPSA over BP should be easily observable in the RNN and GRU end-to-end benchmarks.

A.7 Experiment Customization

Each stage of the workflow can be turned off independently by commenting out the corresponding lines in `workflow.sh`. The software environment can be customized by modifying `docker/Dockerfile` and rebuilding the Docker image. The parameters of the RNN and GRU end-to-end benchmarks can be customized by modifying the `code/rnn_grid_run.sh` and `code/gru_grid_run.sh` scripts which are launched by `workflow.sh` through Docker containers.

A.8 Methodology

Submission, reviewing and badging methodology:

- `http://cTuning.org/ae/submission-20200102.html`
- `http://cTuning.org/ae/reviewing-20200102.html`
- `https://www.acm.org/publications/policies/artifact-review-badging`

Appendix B

Space Complexity of GPipe

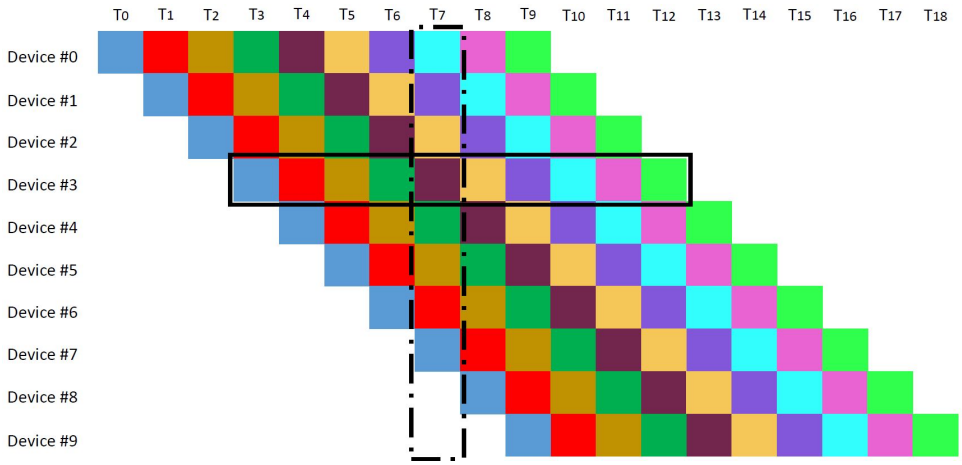


Figure B.1: Timing diagram of the forward pass when distributing a model via pipeline parallelism. Each color represents an individual batch.

Using the notations consistent with GPipe [35], with re-materialization enabled, each device reserves $\Theta(L/K)$ space for re-computing the intermediate activations of each sample in a "micro-batch", where L and K are the length of the network and the number of devices in the pipeline correspondingly. As we show in Figure B.1, to fully fill the pipeline with useful computation, the number of "micro-batches" entering the pipeline (the solid black box) should be equal to the length of the pipeline (the dashed black box); thus, each device needs to store at least $\Theta(K)$ activations at the partition boundary for each sample, resulting in a $\Theta(L/K + K)$ per-device space complexity.

Appendix C

Affect of PipeDream's Staleness on Adam

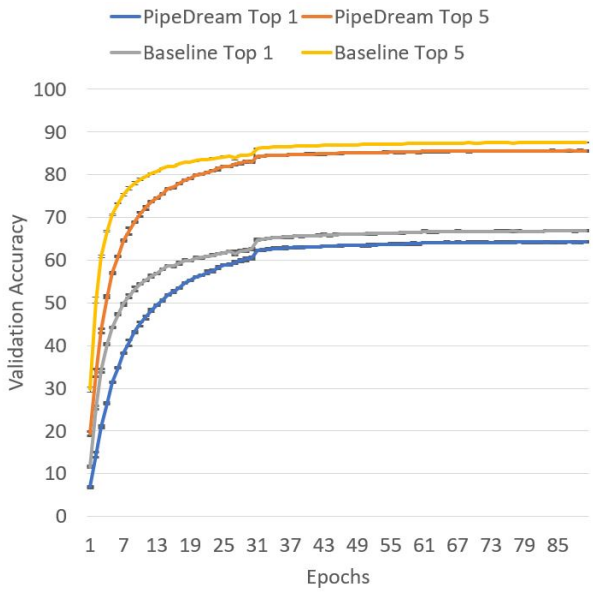


Figure C.1: Top-1 and Top-5 validation accuracy across epochs for both PipeDream and the baseline. We report both the mean (as curves) and the range (as error bars).

Using the source code from `https://github.com/msr-fiddle/pipedream`, we reproduce PipeDream's results on VGG-16 with the same settings except the following:

- 4 RTX 2080Ti GPUs (instead of 16 V100 GPUs).
- Mini-batch size of 32 (instead of 64).
- Adam optimizer with the learning rate of 0.00003 and zero weight decay (instead of SGD with the learning rate of 0.01 and the weight decay of 0.0005).
- 90 epochs in total (instead of 60).
- Step decay learning rate schedule (instead of polynomial decay).

For the baseline, we use the source code from `https://github.com/pytorch/examples/tree/master/imagenet` (which is a plain VGG implementation used as one of PyTorch's official examples for ImageNet [22]) and the same aforementioned settings except using one GPU (instead of four). We choose these settings for the following purpose: (1) to fit in the hardware resources available to us; (2) to match with a widely adopted baseline; and (3) to use the Adam optimizer instead of SGD. We run both experiments three times and record the Top-1 and Top-5 validation accuracy across epochs. We present our results in Figure C.1.

We observe a 2.6% top-1 and 1.9% top-5 accuracy loss from PipeDream compared to the baseline. Combining with the observation that the error bars are negligible (i.e., negligible variance across runs), we conclude that at least in this case, PipeDream is not fully equivalent to the baseline for some adaptive optimizers (e.g., Adam), which differs from the optimizer-oblivious property of our work. It is important to emphasize that we **do not** imply that PipeDream will always have a negative impact on the convergence. A deeper analysis is needed on a much greater space of hyper-parameters to understand how general is such an effect on convergence with PipeDream that is beyond the scope of our work.

Appendix D

Sparse Jacobian Generation Routines

D.1 Convolution

Algorithm 2, Algorithm 3 and Algorithm 4 show how to generate the CSR `indptr`, `indices` and `data` arrays [69] respectively for the transposed Jacobian of a convolution operator that has a 3×3 filter and padding size of 1.[1]

Algorithm 2 Compute the CSR `indptr` array for the transposed Jacobian of a 3×3 convolution.

Input: input channels c_i, output channels c_o, input height h_i, input width w_i
Output: `indptr` $\leftarrow$ `malloc`$(c_i h_i w_i + 1)$

for all $i \leftarrow 0$ to $(c_i h_i w_i)$ **do in parallel**
  $a \leftarrow \lfloor i/(h_i w_i) \rfloor$
  $b \leftarrow i \bmod (h_i w_i)$
  if $b \leqslant w_i$ **then**
    `indptr`$[i] \leftarrow ac_o(3w_i(3h_i - 2)) + 6c_o b$
  else if $b \leqslant w_i(h_i - 1)$ **then**
    `indptr`$[i] \leftarrow ac_o(3w_i(3h_i - 2)) + 6c_o w_i + 9c_o(b - w_i)$
  else
    `indptr`$[i] \leftarrow ac_o(3w_i(3h_i - 2)) + 6c_o w_i + 9c_o(w_i(h_i - 2))$
      $+ 6c_o(b - w_i(h_i - 1))$
  end if
end for

D.2 ReLU

Our methods of generating the CSR `indptr`, `indices` and `data` arrays [69] for the transposed Jacobian of a ReLU operator are formally described in Algorithm 5, Algorithm 6 and Algorithm 7 respectively.

D.3 Max-pooling

Assuming the stride size and the window size are the same, and we can access a tensor (named as `pool_indices` for brevity) which specifies the indices of the elements in the input tensor that are "pooled"

[1] Although we assume a specific configuration of the convolution operator here, deriving a generic routine is doable.

Algorithm 3 Compute the CSR `indices` array for the transposed Jacobian of a 3×3 convolution.

Input: input channels c_i, output channels c_o, input height h_i, input width w_i, `indptr` computed from Algorithm 2
Output: `indices` $\leftarrow$ `malloc`$(3w_i(3h_i - 2)c_ic_o)$

for all $i \leftarrow 0$ to $(c_ih_iw_i - 1)$ **do in parallel**
  $r \leftarrow i \bmod (h_iw_i)$
  `base` $\leftarrow$ `malloc`$(9c_o)$
  for all $j \leftarrow 0$ to $(c_o - 1)$ **do in parallel**
    for all $k \leftarrow 0$ to 2 **do in parallel**
      `base`$[9j + 3k : 9j + 3(k + 1)] \leftarrow ($
        $[-1, 0, 1] + (jh_i + k - 1)w_i + r) \bmod (c_oh_iw_i)$
    end for
  end for
  if $r < w_i$ or $r \geqslant w_i(h_i - 1)$ **then**
    `row` $\leftarrow$ `malloc`$(6c_o)$
    (`left`, `right`) $\leftarrow (3, 9)$ if $r < w_i$; $(0, 6)$ otherwise
    for all $j \leftarrow 0$ to $(c_o - 1)$ **do in parallel**
      `row`$[6j : 6j + 6] \leftarrow$ `base`$[9j +$ `left` $: 9j +$ `right`$]$
    end for
  else
    `row` $\leftarrow$ `base`
  end if
  `indices`$[$`indptr`$[i] :$ `indptr`$[i + 1]] \leftarrow$ `sorted`(`row`)
end for

for the output tensor (documented in [56]), our methods of generating the CSR `indptr`, `indices` and `data` arrays [69] are formally described in Algorithm 8, Algorithm 9 and Algorithm 10 respectively.

Algorithm 4 Compute the CSR `data` array for the transposed Jacobian of a 3×3 convolution.

Input: input channels c_i, output channels c_o, input height h_i, input width w_i, filter `weights`, `indptr` computed from Algorithm 2
Output: `data` $\leftarrow$ `malloc`$(3w_i(3h_i - 2)c_ic_o)$

for all $i \leftarrow 0$ to $(c_ih_iw_i - 1)$ **do in parallel**
  $r \leftarrow i \bmod (h_iw_i)$
  $m \leftarrow \lfloor i/(h_iw_i) \rfloor$
  `range` $\leftarrow$(`1::-1`) if $(r < w_i)$;
    (`2:0:-1`) if $(r \geqslant w_i(h_i - 1))$;
    (`2::-1`) otherwise
  `data`[`indptr`$[i] :$ `indptr`$[i + 1]] \leftarrow$ `flatten`(
    `weights`$[:, m,$ `range`$, ::-1])$
  Fix corner cases when $(i \bmod w_i) = 0$ or $(i \bmod w_i) = (w_i - 1)$.
end for

Algorithm 5 Compute the CSR `indptr` array for the transposed Jacobian of ReLU.

Input: size d of the (flattened) input tensor x
Output: `indptr` $\leftarrow$ `malloc`$(d + 1)$

for all $i \leftarrow 0$ to d **do in parallel**
  `indptr`$[i] \leftarrow i$
end for

Algorithm 6 Compute the CSR `indices` array for the transposed Jacobian of ReLU.

Input: size d of the (flattened) input tensor x
Output: `indices` $\leftarrow$ `malloc`(d)

for all $i \leftarrow 0$ to $(d - 1)$ **do in parallel**
  `indices` $\leftarrow i$
end for

Algorithm 7 Compute the CSR `data` array for the transposed Jacobian of ReLU.

Input: the (flattened) input tensor x, and its size d
Output: `data` $\leftarrow$ `malloc`(d)

for all $i \leftarrow 0$ to $(d - 1)$ **do in parallel**
  if $x[i] > 0$ **then**
    `data`$[i] \leftarrow 1$
  else
    `data`$[i] \leftarrow 0$
  end if
end for

Algorithm 8 Compute the CSR `indptr` array for the transposed Jacobian of max-pooling.

Input: `pool_indices`, input height h_i, input width w_i, output height h_o, output width w_o, output channels c_o
Output: `indptr` $\leftarrow$ `malloc`$(c_o h_i w_i + 1)$, `mapping` $\leftarrow$ `malloc`$(c_o h_i w_i)$

for all $i \leftarrow 0$ to $c_o h_i w_i - 1$ **do in parallel**
    `mapping`$[i] \leftarrow -1$
end for
for all $c \leftarrow 0$ to $c_o - 1$ **do in parallel**
    for all $h \leftarrow 0$ to $h_o - 1$ **do in parallel**
        for all $w \leftarrow 0$ to $w_o - 1$ **do in parallel**
            $i \leftarrow ch_i w_i +$ `pool_indices`$[c, h, w]$
            $j \leftarrow (ch_o + h)w_o + w$
            `mapping`$[i] \leftarrow j$
        end for
    end for
end for
`ptr` $\leftarrow 0$
for $i \leftarrow 0$ to $(c_o h_i w_i - 1)$ **do**
    `indptr`$[i] \leftarrow$ `ptr`
    if `mapping`$[i] \neq -1$ **then**
        `ptr` $\leftarrow$ `ptr` $+ 1$
    end if
end for
`indptr`$[-1] \leftarrow$ `ptr`

Algorithm 9 Compute the CSR `indices` array for the transposed Jacobian of max-pooling.

Input: `mapping` computed from Algorithm 8, input height h_i, input width w_i, output height h_o, output width w_o, output channels c_o
Output: `indices` $\leftarrow$ `malloc`$(c_o h_o w_o)$

`indices_ptr` $\leftarrow 0$
for $i \leftarrow 0$ to $(c_o h_i w_i - 1)$ **do**
    if `mapping`$[i] = -1$ **then**
        continue
    end if
    `indices`[`indices_ptr`] $\leftarrow$ `mapping`$[i]$
    `indices_ptr` $\leftarrow$ `indices_ptr` $+ 1$
end for

Algorithm 10 Compute the CSR `data` array for the transposed Jacobian of max-pooling.

Input: output channels c_o, output height h_o, output width w_o
Output: `data` $\leftarrow$ `malloc`$(c_o h_o w_o)$

for all $i \leftarrow 0$ to $(c_o h_o w_o - 1)$ **do in parallel**
    `data` $\leftarrow 1$
end for

Appendix E

Overhead Analysis of the GRU End-to-end Benchmark

We can rewrite the GPU in Equation 4.3 into the following form:

$$
\begin{aligned}
&\vec{R}_t = W_{ir}\vec{x}_t + \vec{b}_{ir} + W_{hr}\vec{h}_{t-1} + \vec{b}_{hr} \\
&\vec{Z}_t = W_{iz}\vec{x}_t + \vec{b}_{iz} + W_{hz}\vec{h}_{t-1} + \vec{b}_{hz} \\
&\vec{M}_t = W_{hn}\vec{h}_{t-1} + \vec{b}_{hn}, \quad \vec{N}_t = W_{in}\vec{x}_t + \vec{b}_{in} + \vec{r}_t \circ \vec{M}_t \\
&\vec{r}_t = \sigma(\vec{R}_t), \quad \vec{z}_t = \sigma(\vec{Z}_t), \quad \vec{n}_t = tanh(\vec{N}_t) \\
&\vec{h}_t = (1 - \vec{z}_t) \circ \vec{n}_t + \vec{z}_t \circ \vec{h}_{t-1}
\end{aligned}
\tag{E.1}
$$

Given the GRU expressed in the above form, the transposed Jacobian between consecutive hidden

states $\frac{\partial \vec{h}_t}{\partial \vec{h}_{t-1}}$ can be computed analytically:

$$
\begin{aligned}
J_1 &= (\frac{\partial \vec{R}_t}{\partial \vec{h}_{t-1}})^T = W_{hr}^T \\
\vec{j}_2 &= Diag((\frac{\partial \vec{r}_t}{\partial \vec{R}_t})^T) = \vec{r}_t \circ (1 - \vec{r}_t) \\
\vec{j}_3 &= Diag((\frac{\partial \vec{N}_t}{\partial \vec{r}_t})^T) = \vec{M}_t \\
J_4 &= (\frac{\partial \vec{M}_t}{\partial \vec{h}_{t-1}})^T = W_{hn}^T \\
\vec{j}_5 &= Diag((\frac{\partial \vec{N}_t}{\partial \vec{M}_t})^T) = \vec{r}_t \\
\vec{j}_6 &= Diag((\frac{\partial \vec{n}_t}{\partial \vec{N}_t})^T) = 1 - \vec{n}_t \circ \vec{n}_t \\
\vec{j}_7 &= Diag((\frac{\partial \vec{h}_t}{\partial \vec{n}_t})^T) = 1 - \vec{z}_t \\
J_8 &= (\frac{\partial \vec{Z}_t}{\partial \vec{h}_{t-1}})^T = W_{hz}^T \\
\vec{j}_9 &= Diag((\frac{\partial \vec{z}_t}{\partial \vec{Z}_t})^T) = \vec{z}_t \circ (1 - \vec{z}_t) \\
\vec{j}_{10} &= Diag((\frac{\partial \vec{h}_t}{\partial \vec{z}_t})^T) = \vec{h}_{t-1} - \vec{n}_t \\
J_{11} &= (\frac{\partial \vec{h}_t}{\partial \vec{h}_{t-1}})^T_{\text{direct}} = I \circ \vec{z} \\
\frac{\partial \vec{h}_t}{\partial \vec{h}_{t-1}} &= (J_1 \circ (\vec{j}_2 \circ \vec{j}_3)^T + J_4 \circ \vec{j}_5^T) \circ (\vec{j}_6 \circ \vec{j}_7)^T + J_8 \circ (\vec{j}_9 \circ \vec{j}_{10})^T + J_{11}
\end{aligned}
\tag{E.2}
$$

where $Diag(.)$ represents taking the diagonal of a square matrix, and $\circ$ represents the broadcasting element-wise (Hadamard) product. Since cuDNN's GRU implementation [5] is closed source, we cannot access the values of the gates ($\vec{r}_t$, $\vec{z}_t$, $\vec{n}_t$). Therefore, we have to recompute the gates (but in a more parallelized way) during the forward pass in order to compute $\frac{\partial \vec{h}_t}{\partial \vec{h}_{t-1}}$ as shown in Equations E.2. This engineering challenge results in significant *overhead* during the forward pass in our experiments, however, can potentially be resolved if cuDNN's source code were publicly available and modifiable.

Appendix F

GRU Training Curve

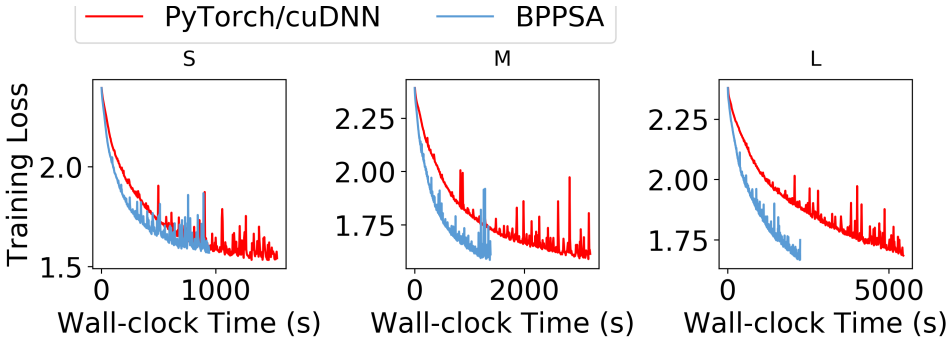


Figure F.1: Training loss across wall-clock time when the GRU is trained via BPPSA (blue curve) and the PyTorch Autograd baseline with cuDNN's RNN backend (red curve).

Figure F.1 shows the training curves of loss values with respect to wall-clock time when we train the GRU with the (S, M, L) preprocessed datasets for 400 epochs on the RTX 2070 GPU when the mini-batch size B is 16. We observe that the blue curve (BPPSA), if horizontally-scaled, maintains a similar shape as the red curve (PyTorch/cuDNN baseline), which reinforces our observation in Section 5.1 that BPPSA reconstructs the original back-propagation algorithm but achieves a shorter training time.

Appendix G

Additional Hardware Sensitivity Results

To further validate BPPSA on a different GPU architecture, we repeat the experiments in Section 4.1 and Section 4.2 on an NVIDIA V100 (Volta architecture) [54] through an AWS p3.2xlarge instance [1] with the same software stack as our RTX 2080Ti platform (in Table 4.1). The results are summarized in:

- Table G.1 and Table G.2 for the RNN end-to-end benchmark (Section 4.1). We can derive the backward pass runtime of the *baseline* by subtracting the "Forward Pass Only" column from the "Baseline" column for each row (T or $1/B$); while we can also derive the backward pass runtime of *our* method by subtracting the "Forward Pass Only" column from the "BPPSA" column for each row (T or $1/B$).
- Table G.3 for the GRU end-to-end benchmark (Section 4.2). We can derive the backward pass runtime of the *baseline* by subtracting the "*FP*" row from the "Baseline" row for each column (batch size); while we can also derive the backward pass runtime of *our method* by subtracting the "*FP* + Overhead" row from the "BPPSA" row for each column (batch size).

From the aforementioned results, we can observe trends in both benchmarks similar to the ones in the results from RTX2070 and RTX2080Ti (Section 5.1 and Section 5.2).

Table G.1: The wall-clock time (s) for running a single epoch of the RNN end-to-end benchmark (Section 4.1) as the sequence length T increases.

Sequence Length (T)	Forward Pass Only	Baseline	BPPSA
10	1.57 ± 0.01	4.49 ± 0.04	4.24 ± 0.04
30	2.13 ± 0.01	5.5 ± 0.06	4.91 ± 0.06
100	3.82 ± 0.02	8.87 ± 0.05	6.64 ± 0.08
300	8.79 ± 0.03	18.49 ± 0.1	11.71 ± 0.07
1000	25.86 ± 0.12	53.33 ± 0.39	29.29 ± 0.18
3000	75.33 ± 0.36	157.11 ± 0.58	79.48 ± 0.45
10000	250.28 ± 0.32	510.13 ± 1.11	265.02 ± 0.64
30000	748.99 ± 0.71	1557.94 ± 6.53	764.31 ± 0.63

Table G.2: The wall-clock time (s) for running a single epoch of the RNN end-to-end benchmark (Section 4.1) as the fraction of GPU per sample ($1/B$) increases.

Fraction of GPU per Sample ($1/B$)	Forward Pass Only	Baseline	BPPSA
$1/256$	2.13 ± 0.02	4.02 ± 0.06	3.72 ± 0.02
$1/128$	3.87 ± 0.02	7.96 ± 0.11	5.49 ± 0.01
$1/64$	7.51 ± 0.06	14.77 ± 0.2	9.27 ± 0.02
$1/32$	13.62 ± 0.11	29.51 ± 0.34	15.0 ± 0.12
$1/16$	25.86 ± 0.12	53.33 ± 0.39	29.29 ± 0.18
$1/8$	51.29 ± 0.18	102.77 ± 0.61	56.37 ± 0.35
$1/4$	100.55 ± 0.25	209.67 ± 0.66	112.23 ± 0.47
$1/2$	200.65 ± 0.24	409.33 ± 0.67	227.13 ± 0.74

Table G.3: The wall-clock time (s) of running one epoch in the GRU end-to-end benchmark (Section 4.2) as the dataset type (S, M, L) and the batch size B varies. "FP" represents running the forward pass only; "FP + Overhead" represents running the forward pass with GRU's Jacobian generation overhead; "Baseline" represents training normally via the BP baseline; "BPPSA" represents training via our method.

		Batch Size $B = 16$	Batch Size $B = 32$	Batch Size $B = 64$
S	FP	1.66 ± 0.01	0.91 ± 0.01	0.52 ± 0.0
	Baseline	3.71 ± 0.02	1.9 ± 0.01	1.0 ± 0.0
	FP + Overhead	2.1 ± 0.01	1.09 ± 0.01	0.62 ± 0.0
	BPPSA	2.94 ± 0.02	1.48 ± 0.01	0.82 ± 0.01
M	FP	3.21 ± 0.01	1.73 ± 0.01	0.86 ± 0.01
	Baseline	6.19 ± 0.04	3.3 ± 0.02	1.82 ± 0.02
	FP + Overhead	3.52 ± 0.02	1.83 ± 0.01	1.18 ± 0.01
	BPPSA	4.2 ± 0.03	2.22 ± 0.01	1.2 ± 0.01
L	FP	5.78 ± 0.03	3.04 ± 0.02	1.64 ± 0.01
	Baseline	11.43 ± 0.09	6.09 ± 0.07	3.19 ± 0.04
	FP + Overhead	6.06 ± 0.04	3.23 ± 0.02	2.13 ± 0.01
	BPPSA	6.91 ± 0.05	3.57 ± 0.04	2.33 ± 0.01