Scaling Back-Propagation by Parallel Scan Algorithm

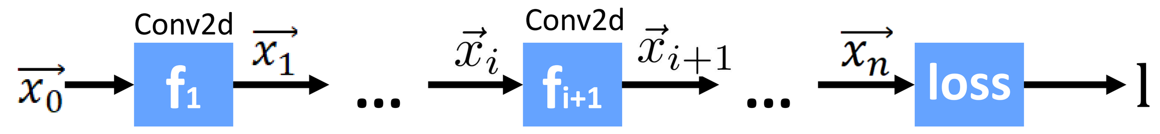
Shang Wang[1,2] | Yifan Bai[1] | Gennady Pekhimenko[1,2]

1 Computer Science, University of Toronto

2 Vector Institute

Back-Propagation Algorithm

Conceptualize a deep learning model as:



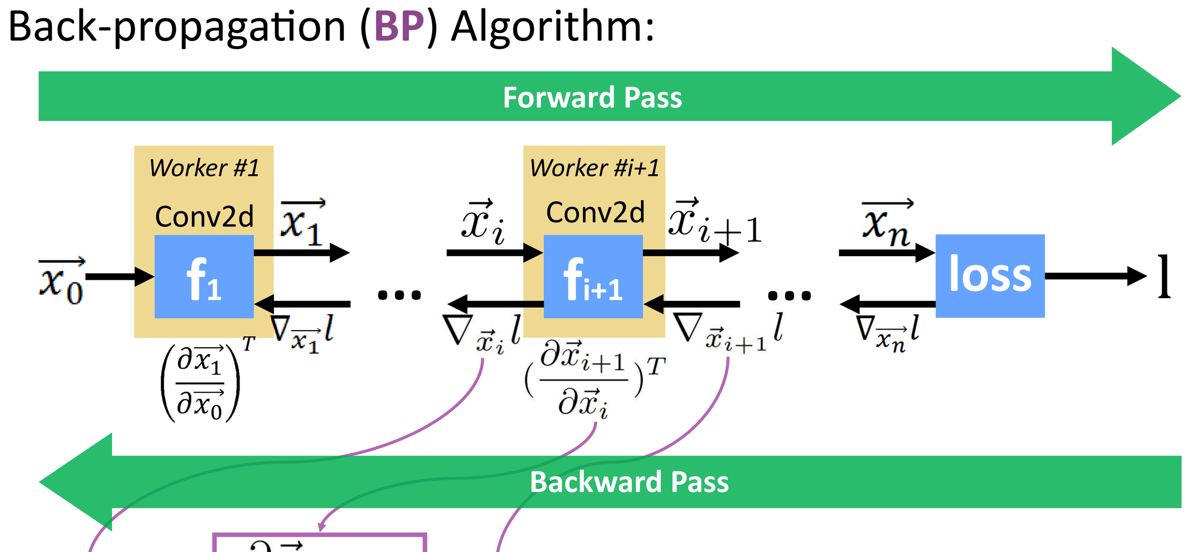
$$f(.;\vec{\theta}_1,...,\vec{\theta}_n) = f_1(.;\vec{\theta}_1) \circ ... \circ f_n(.;\vec{\theta}_n)$$

Parameter updates need:

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

Could be parallelized if all **inputs** are available.

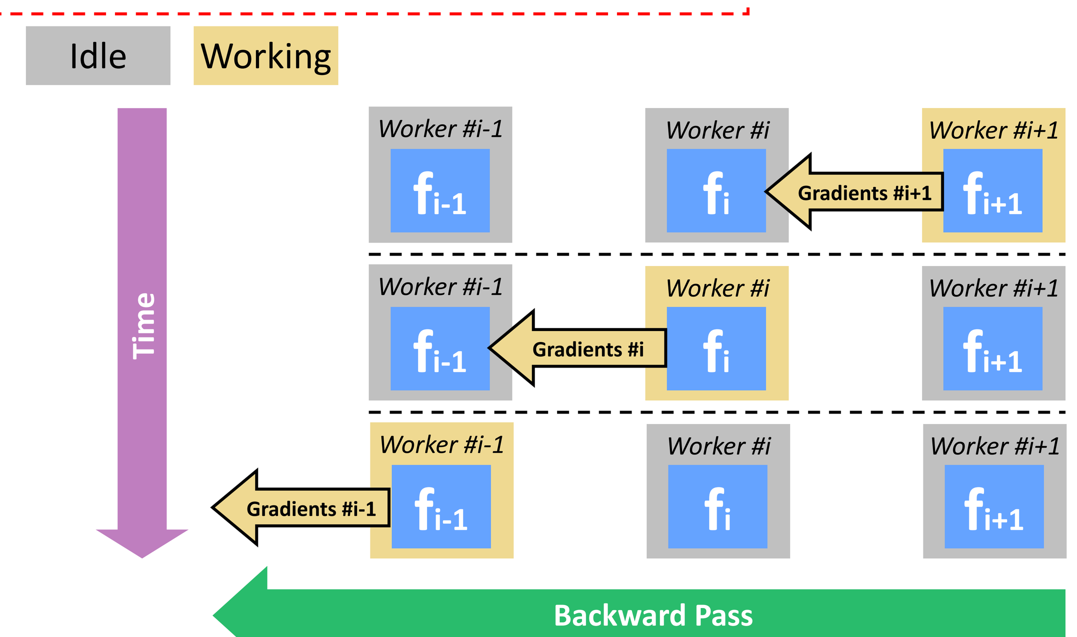
Back-propagation (**BP**) Algorithm:



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

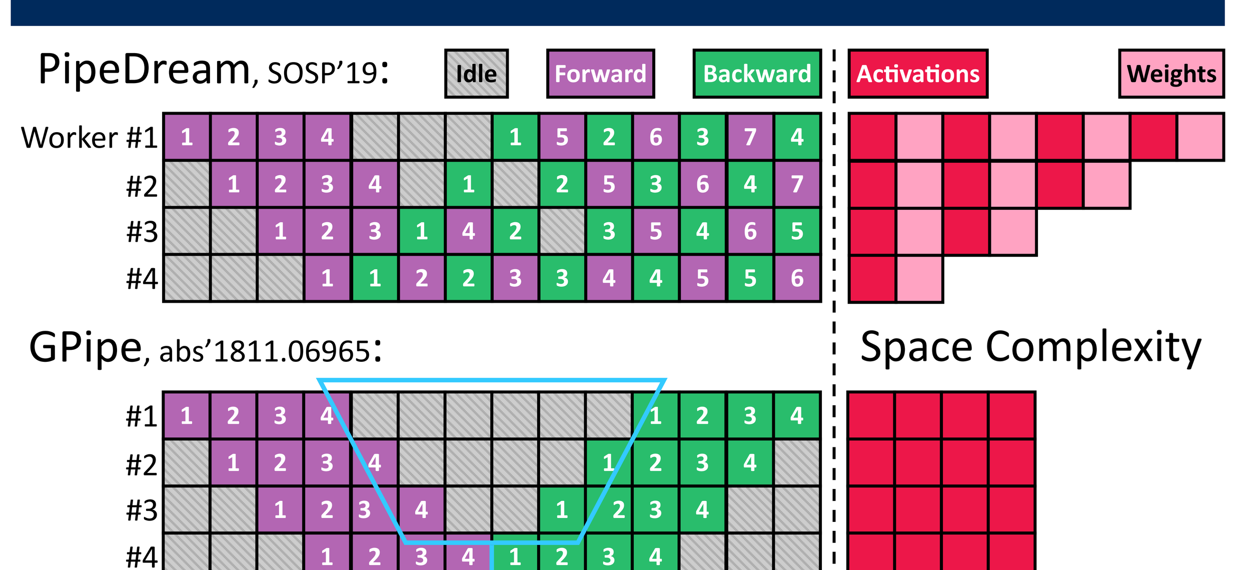
Strong Sequential Dependency along **i**

Limitation of **BP** on parallel systems:

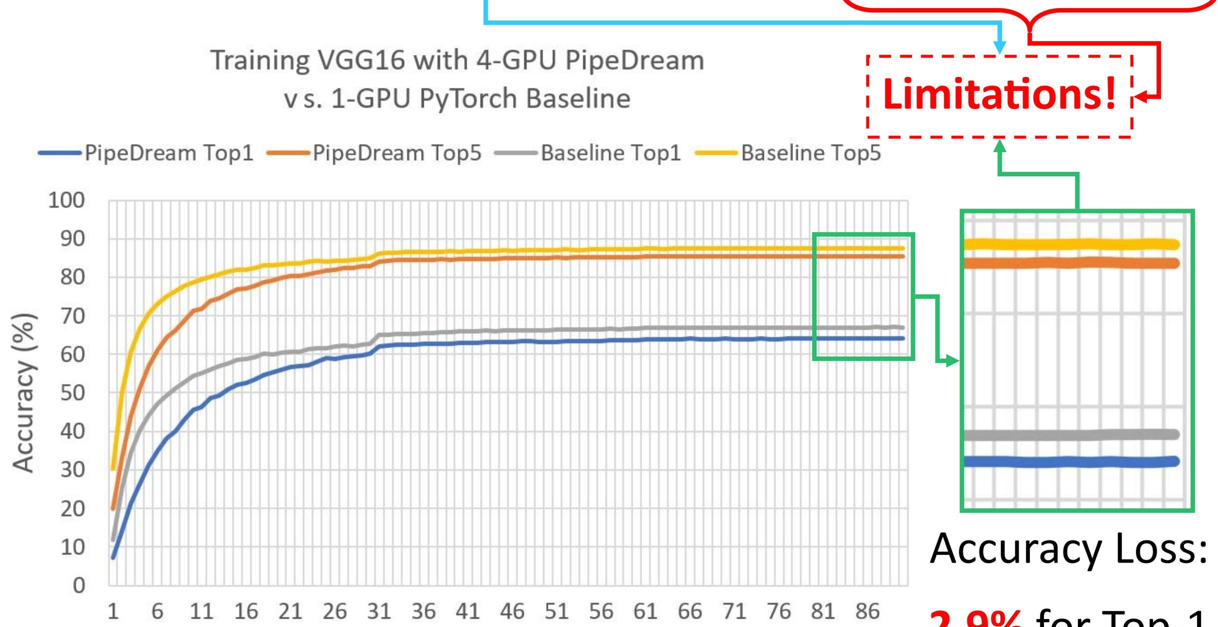


Step Complexity: **Θ(n)**

Prior Works: Pipeline Parallelism



Limitations!

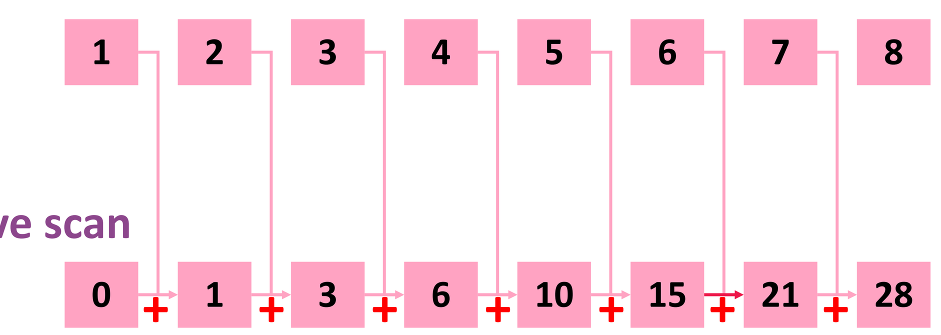


Accuracy Loss: **2.9%** for Top-1

Scan Explained by an Example

For a **binary**, **associative** operator (example: **+**), given the input array:

The **exclusive scan** produces:



Parallel scan algorithms (**Blelloch scan**) were developed to compute scan on parallel systems.

BP as a Scan Operation

Matrix multiplication is also **binary** and **associative**!

Define a **binary**, **associative**, **non-commutative** operator:

$$A \diamond B = BA$$

We can reformulate BP as calculating:

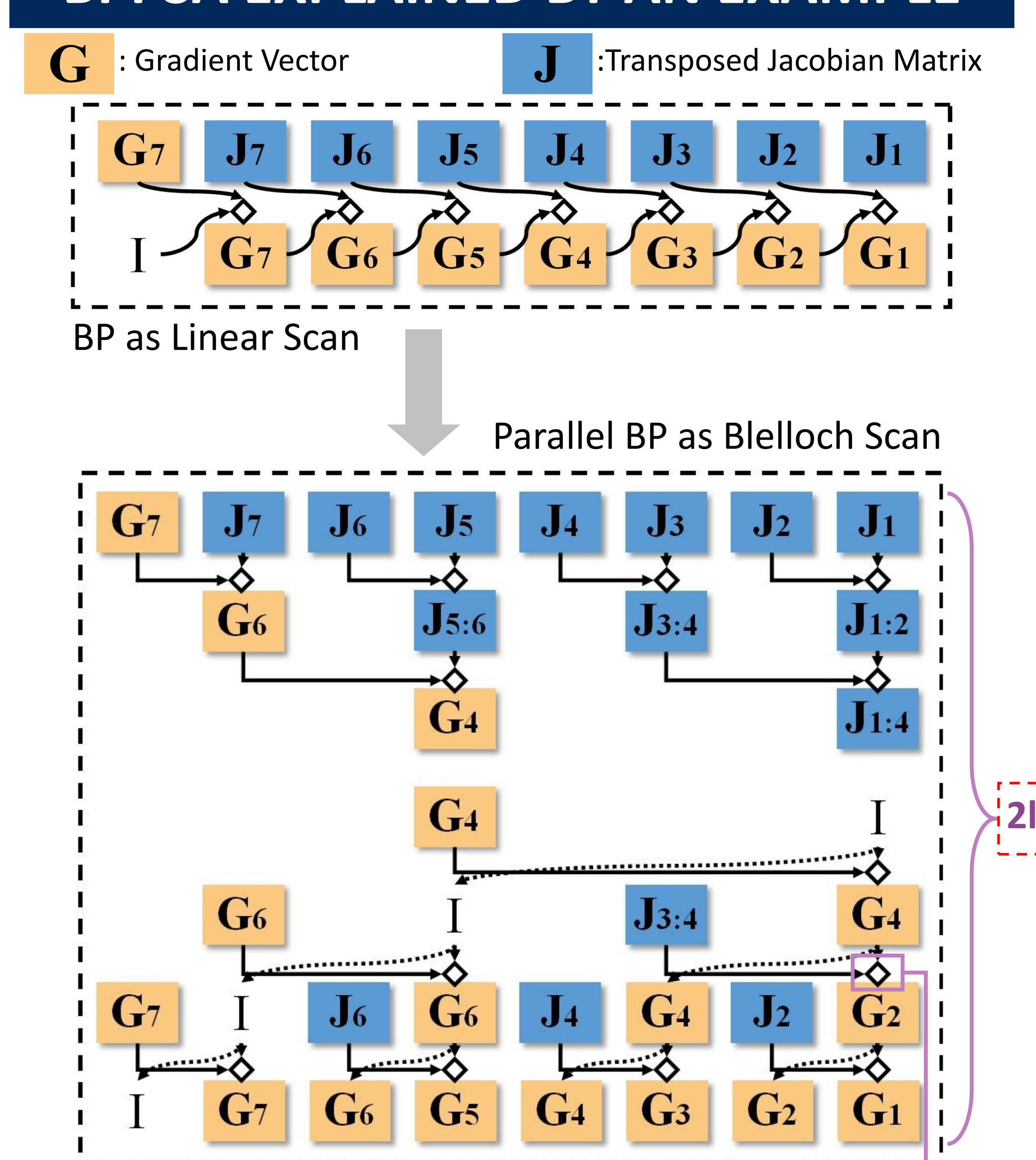
$$[\nabla_{\vec{x}_n} l, \nabla_{\vec{x}_n} l \diamond (\frac{\partial \vec{x}_n}{\partial \vec{x}_{n-1}})^T, \nabla_{\vec{x}_n} l \diamond (\frac{\partial \vec{x}_n}{\partial \vec{x}_{n-1}})^T \diamond (\frac{\partial \vec{x}_{n-1}}{\partial \vec{x}_{n-2}})^T, ..., \nabla_{\vec{x}_n} l \diamond (\frac{\partial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

Which is an **exclusive scan** on the input array:

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

Blelloch scan can be used to **scale BP on parallel systems!**

BPPSA Explained by an Example

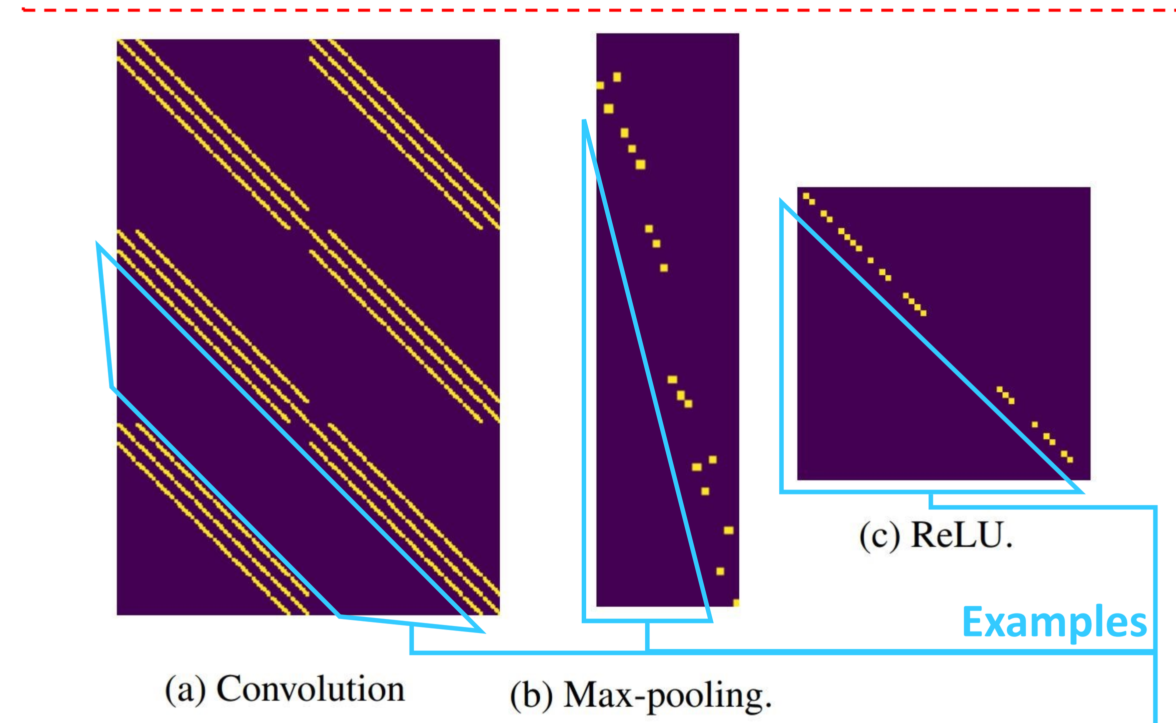


Operands **swapped** for **non-commutativity**!

Insight: Sparsity in the Jacobians

A full Jacobian can be prohibitively large to handle.

However, the Jacobians of major operators can be **extremely sparse**:

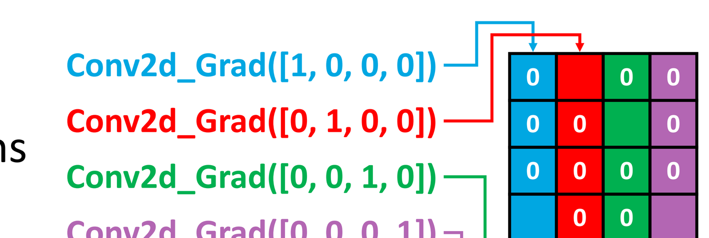


(a) Convolution (b) Max-pooling (c) ReLU.

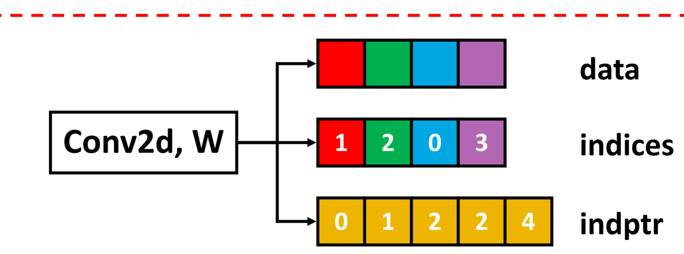
Guaranteed zeros: **deterministic**, known **ahead of time**.

Could be used for better SpGEMM performance!

Therefore, **instead of** calculating the Jacobians column-wise:



Generate **directly** into **Compressed Sparse Row (CSR)**:



First three ops of VGG-11 on CIFAR-10	Convolution	ReLU	Max Pooling
Sparsity	0.99157	0.99998	0.99994
Generation Speedup	8.3×10^3 X	1.2×10^6 X	1.5×10^5 X

Complexity Analysis

n: length of the model; **p**: # of workers.

S: Step complexity—# of steps to finish execution.

W: Work complexity—# of total steps by all workers.

C: Per-step complexity—Runtime of a single step.

M: Space complexity

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

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

$$M_{Blelloch}(n) = \Theta(max(\frac{n}{p}, 1)) M_{Jacob}$$

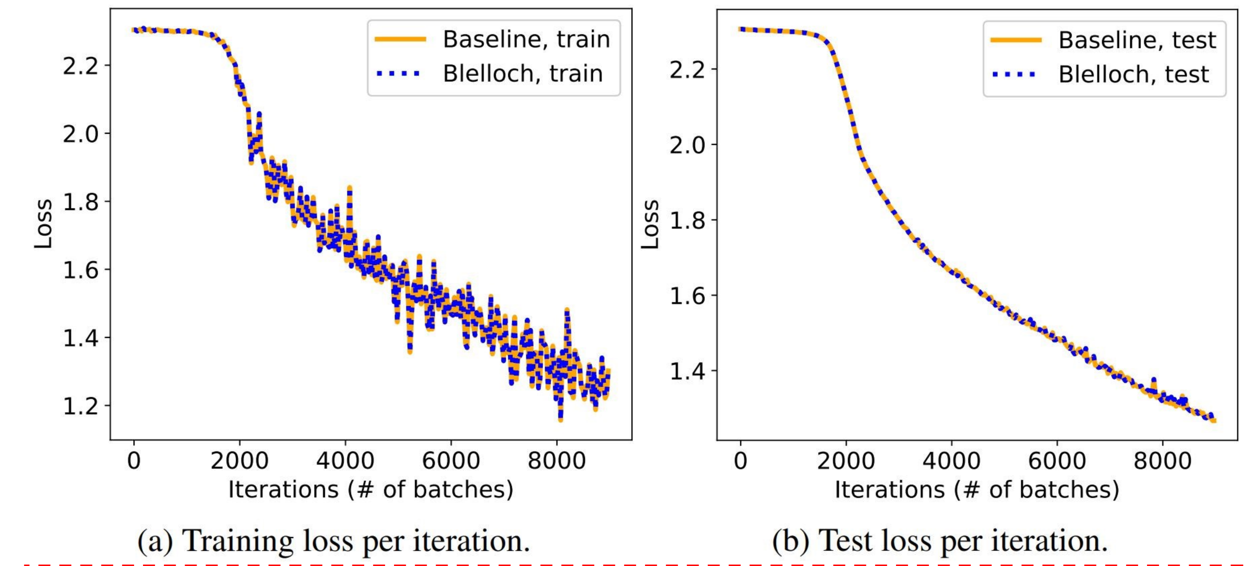
Break-even: $\frac{C_{\text{BPPSA}}}{C_{\text{Baseline}}} < \Theta(\frac{n}{\log n})$

1. Reduce **C**: SpGEMM
2. Large **n**: deep network, long sequential dependency

Convergence / Numerical Stability

Training LeNet-5 on CIFAR-10 (baseline: PyTorch Autograd).

The purple dash lines overlap with the yellow solid lines:



(a) Training loss per iteration. (b) Test loss per iteration.

The **original BP** is **re-constructed exactly**!

Performance Evaluation

Model—**RNN**: $\vec{h}_t^{(k)} = tanh(W_{ih}x_t^{(k)} + \vec{b}_{ih} + W_{hh}\vec{h}_{t-1}^{(k)} + \vec{b}_{hh})$

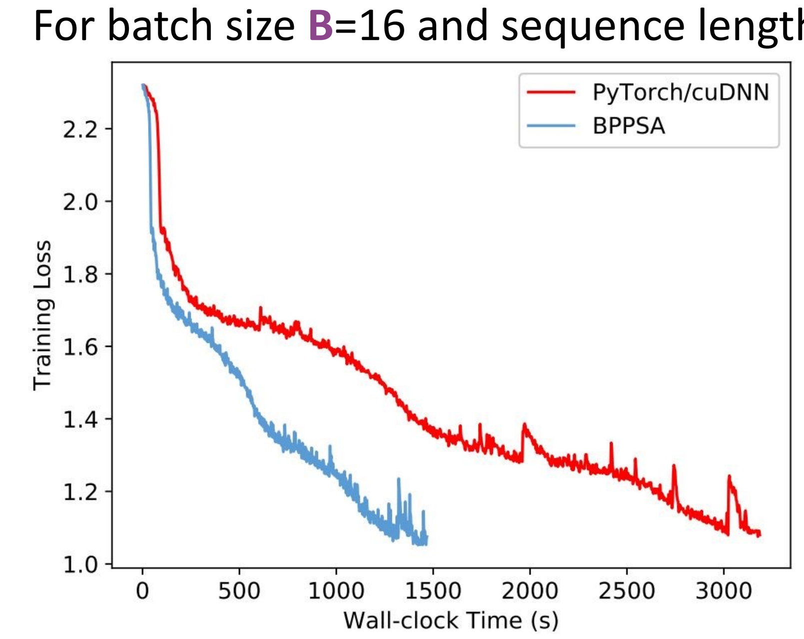
Task—**Classify Bitstream**: $x_t^{(k)} \sim Bernoulli(0.05 + c^{(k)} \times 0.1)$

Baseline: **cuDNN**'s **cudnnRNNBackwardData**

Implementation: **custom CUDA kernels** with PyTorch

Hardware: **RTX 2070**, **RTX 2080Ti** (Turing architecture GPUs)

For batch size **B**=16 and sequence length **T**=1000 on 2070:



2.17X speedup on the **overall** training time.

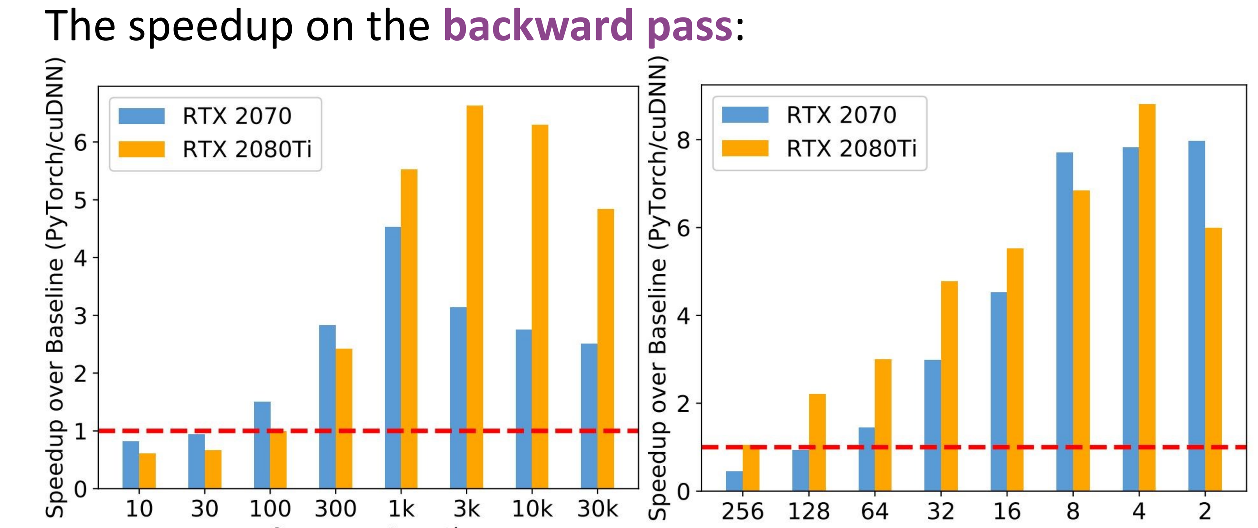
4.53X speedup on the **backward pass** runtime.

Sensitivity Analysis:

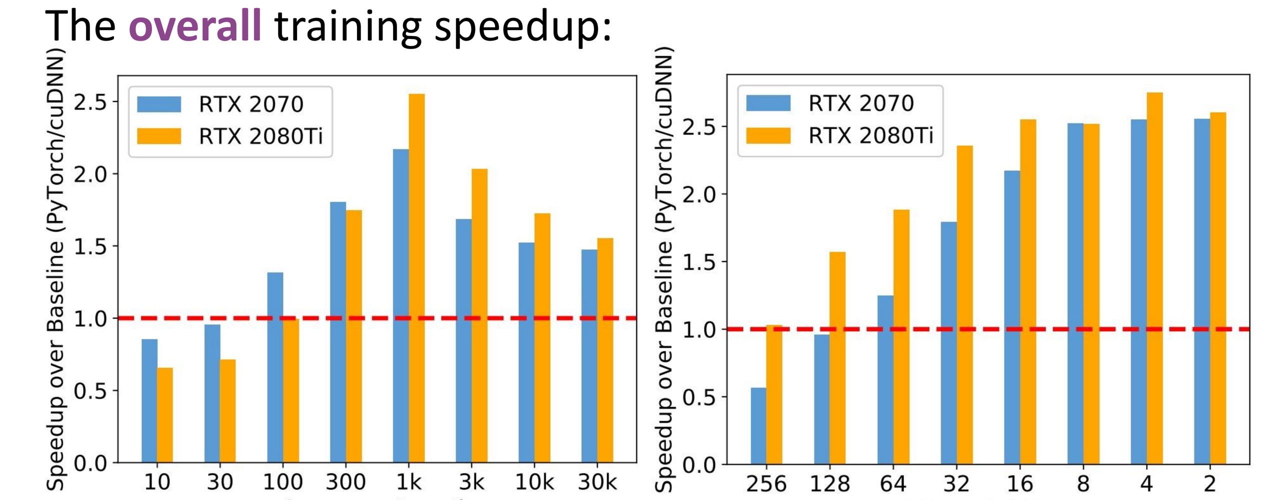
Sequence length (**T**): reflects the model length **n**.

Mini-batch size (**B**): reflects the number of (per-sample) workers **p**.

The speedup on the **backward pass**:



The **overall** training speedup:



1. BPPSA scales with **n** when **n** is in the same range as **p**. When **n** >> **p**, the performance starts to be bounded by **p**.

2. BPPSA scales with **p**.

3. Since #SMs(2080Ti) > #SMs(2070), 2080Ti achieves the maximum speedup at a higher **T** than 2070. As **B** increases, the speedup on 2080Ti drops at a slower rate than 2070.