Scaling Back-Propagation by Parallel Scan Algorithm

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

1 Computer Science UNIVERSITY OF TORONTO

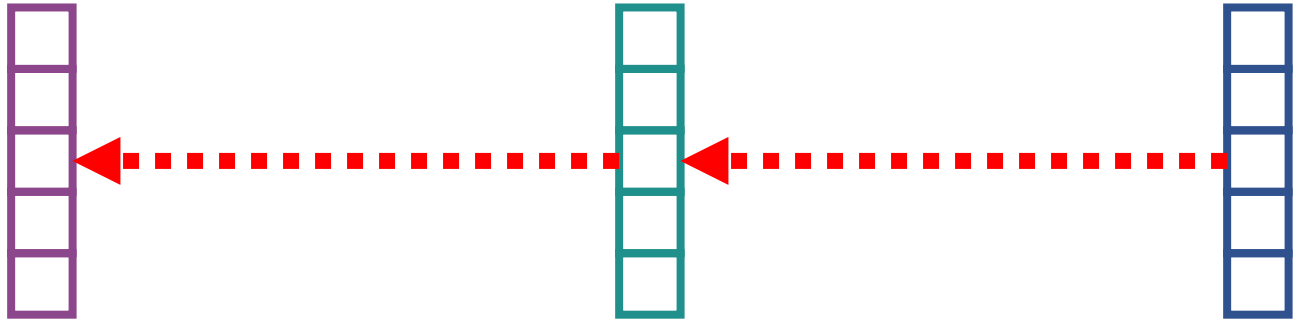
2 VECTOR INSTITUTE

The original PPTX file can be downloaded from here.

Executive Summary

The **back-propagation (BP)** algorithm is **popularly used** in training deep learning (DL) models and **implemented in many** DL frameworks (e.g., PyTorch and TensorFlow).

Problem: BP imposes a **strong sequential dependency** along layers during the gradient computations.



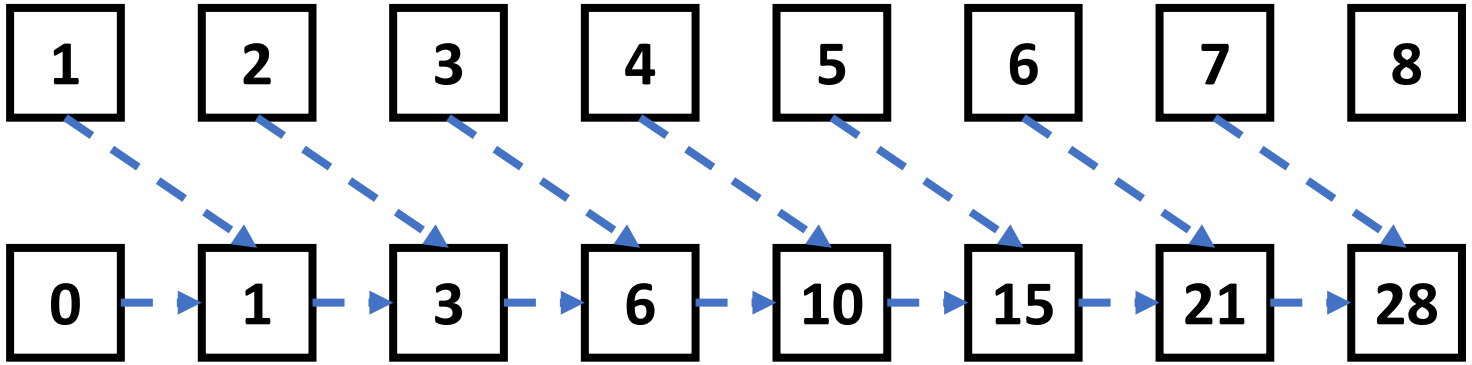
Executive Summary

The **back-propagation (BP)** algorithm is **popularly used** in training deep learning (DL) models and **implemented in many** DL frameworks (e.g., PyTorch and TensorFlow).

Problem: BP imposes a **strong sequential dependency** along layers during the gradient computations.

Key idea: We propose scaling **BP** by **P**arallel **S**can **A**lgorithm (**BPPSA**)**:**

- Reformulate BP into a **scan** operation.



Executive Summary

The **back-propagation (BP)** algorithm is **popularly used** in training deep learning (DL) models and **implemented in many** DL frameworks (e.g., PyTorch and TensorFlow).

Problem: BP imposes a **strong sequential dependency** along layers during the gradient computations.

Key idea: We propose scaling **BP** by **P**arallel **S**can **A**lgorithm (**BPPSA**)**:**

- Reformulate BP into a **scan** operation.
- Scaled by a customized parallel algorithm.

Key Results: **Θ(log n)** vs. **Θ(n)** steps on parallel systems.

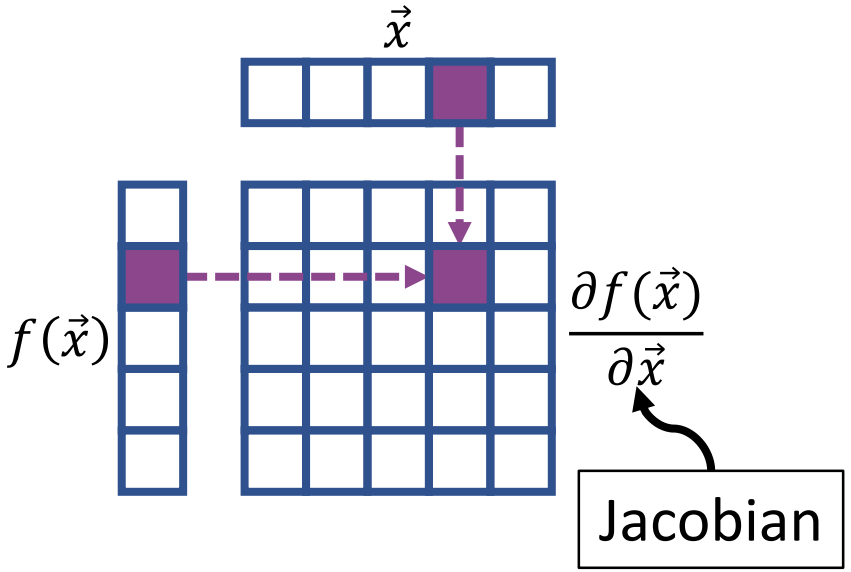
Up to **108×** backward pass speedup (→ **2.17×** overall speedup).

Back-propagation[1] (BP) Everywhere

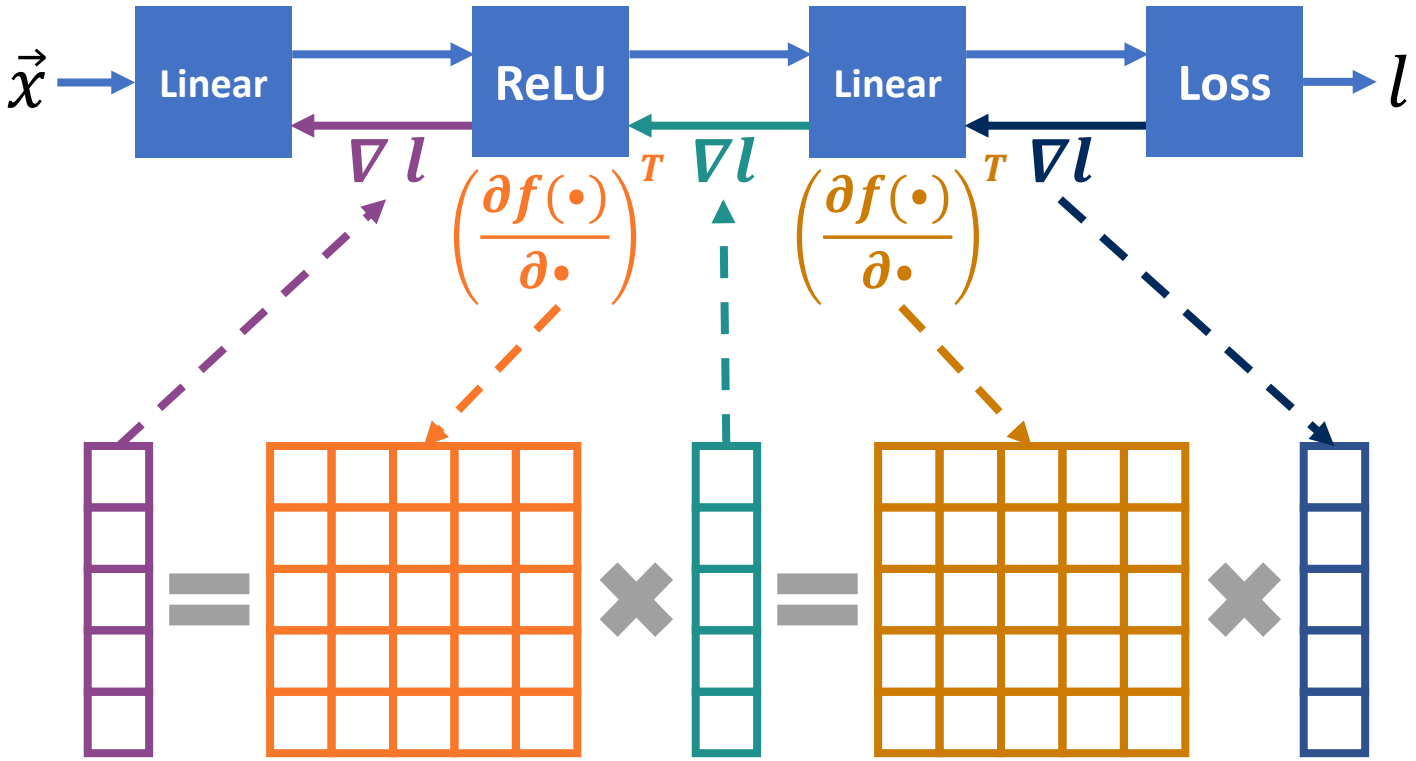
PyTorch theano Caffe mxnet Chainer Microsoft Cognitive Toolkit aka.ms/CognitiveToolkit Microsoft TensorFlow

[1]Rumelhart et al. “Learning representations by back-propagating errors.”, Nature (1986)

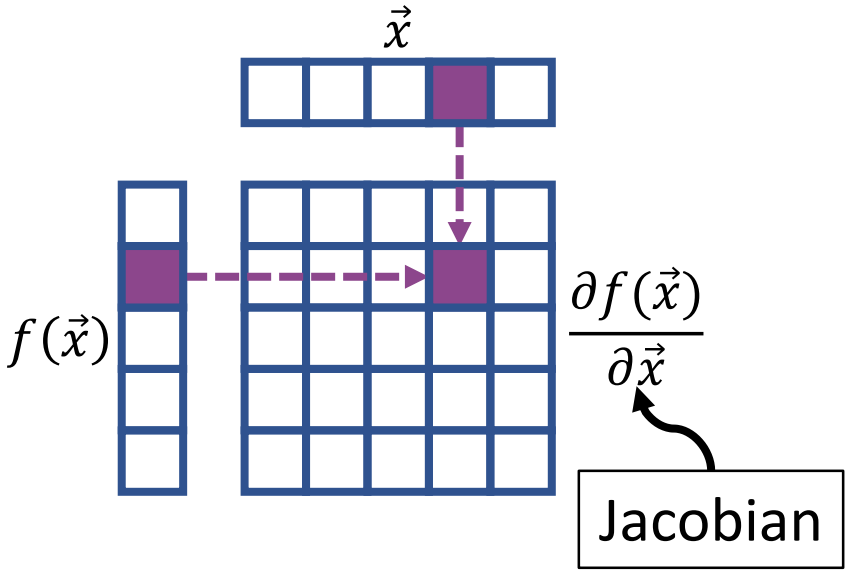
BP's Strong Sequential Dependency



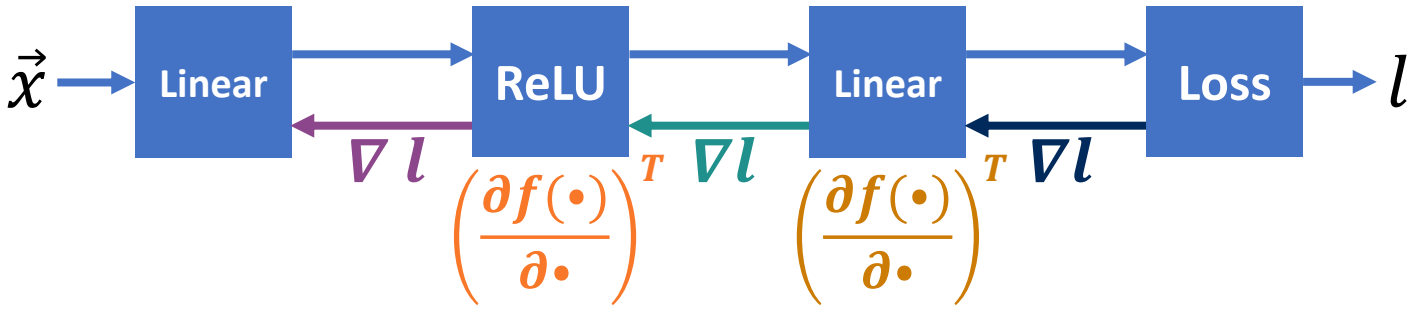
$$\nabla_{\vec{x}} l = \left(\frac{\partial f(\vec{x})}{\partial \vec{x}}\right)^T \nabla_{f(\vec{x})} l$$

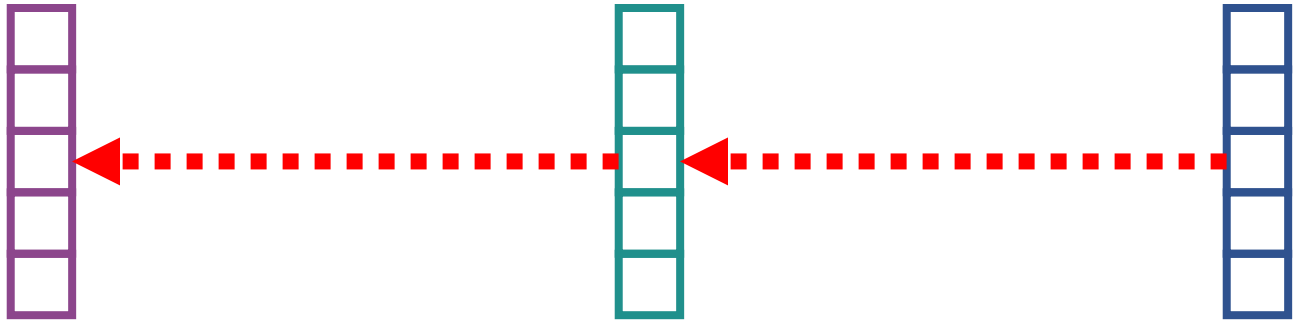


BP's Strong Sequential Dependency



$$\nabla_{\vec{x}} l = \left(\frac{\partial f(\vec{x})}{\partial \vec{x}}\right)^T \nabla_{f(\vec{x})} l$$





Strong Sequential Dependency along layers.

Data Parallel Training

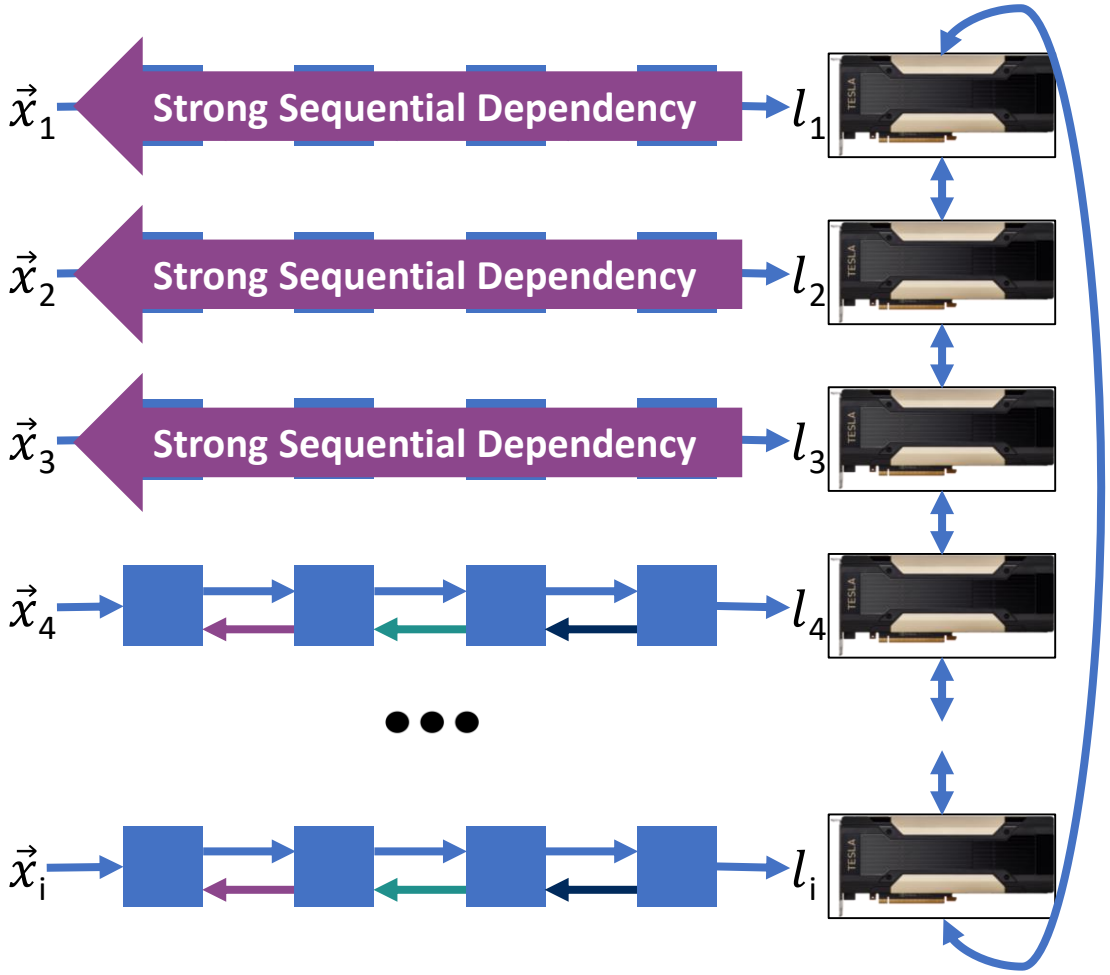
Respects BP's strong sequential dependency.

Conceptually **simple**, **widely used**.

Effectively increases the batch size:

- **Generalization gap**[1]
- **Batch size scaling limit**[2]

Constraint: The model **must** fit in one device.



[1]Keskar, Nitish Shirish et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." ICLR (2017)
[2]Shallue, Christopher J. et al. "Measuring the Effects of Data Parallelism on Neural Network Training." Journal of Machine Learning Research 20 (2019)

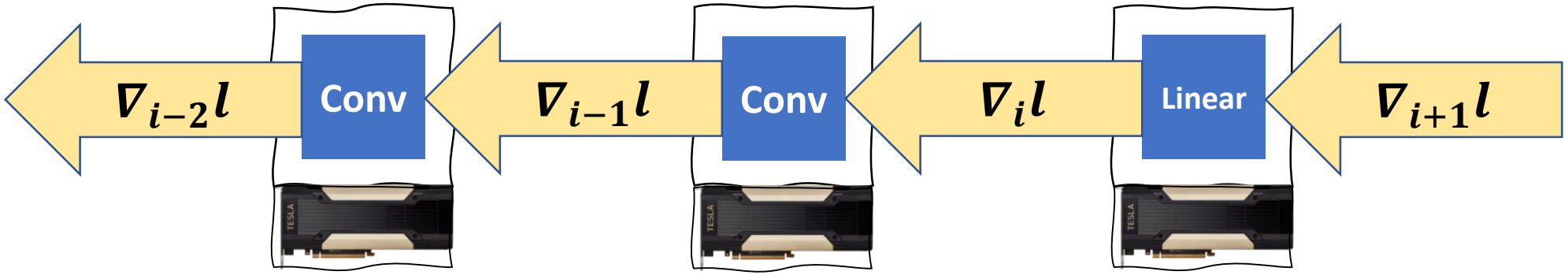
Model Parallel Training

Used when the model cannot fit in one device.

BP's strong sequential dependency **limits scalability**.

Prior works on **pipeline parallel training**[1,2] to mitigate such problem, but have their own limitations:

- **Linear** per-device space complexity.
- Trade-off between "**bubble of idleness**" vs. potential **convergence affect**.



[1]Harlap, Aaron et al. "PipeDream: Fast and Efficient Pipeline Parallel DNN Training." SOSP (2019)
[2]Huang, Yanping et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism." NeurIPS (2019)

Rethinking BP from an Algorithm Perspective

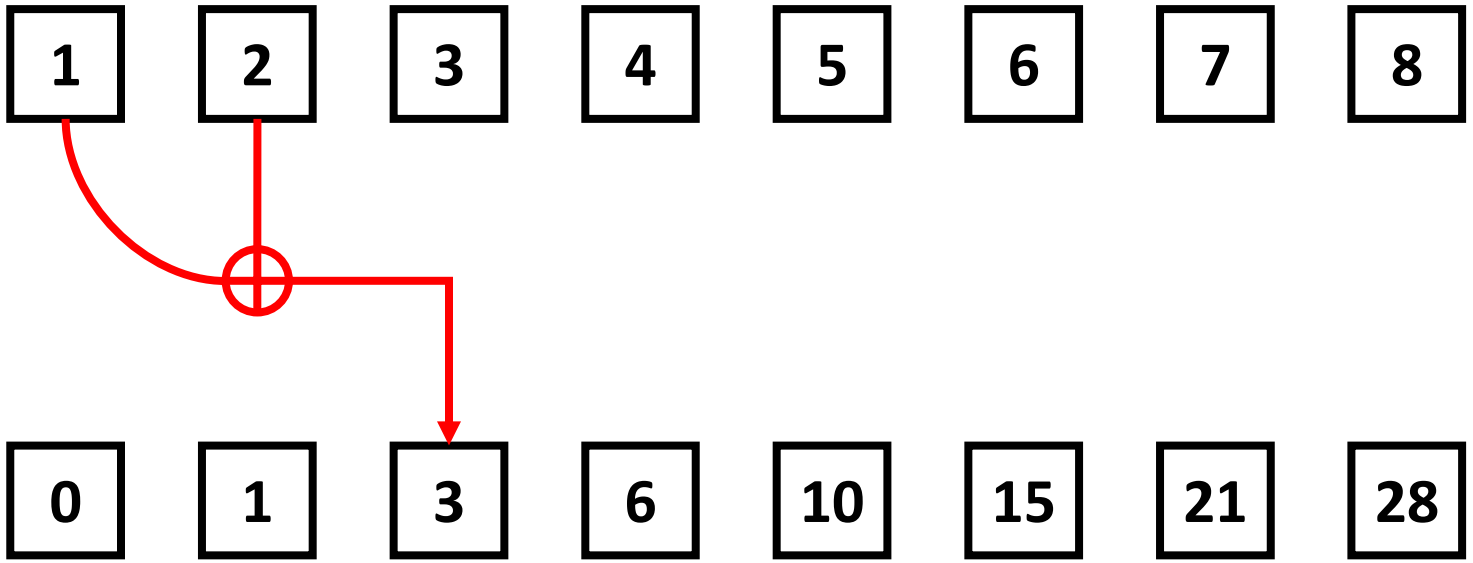
Rethinking BP from an Algorithm Perspective

- Problems with strong sequential dependency were studied in the past (80’), but in a much simpler context.

- We propose scaling **B**ack-**P**ropagation by **P**arallel **S**can **A**lgorithm (**BPPSA**):
  - Reformulate BP as a **scan** operation.
  - Scale BP by a **customized Blelloch Scan** algorithm.
  - Leverage **sparsity** in the Jacobians.

What is a Scan[1] Operation?

Binary, **associative** operator: **+** Identity: **0**

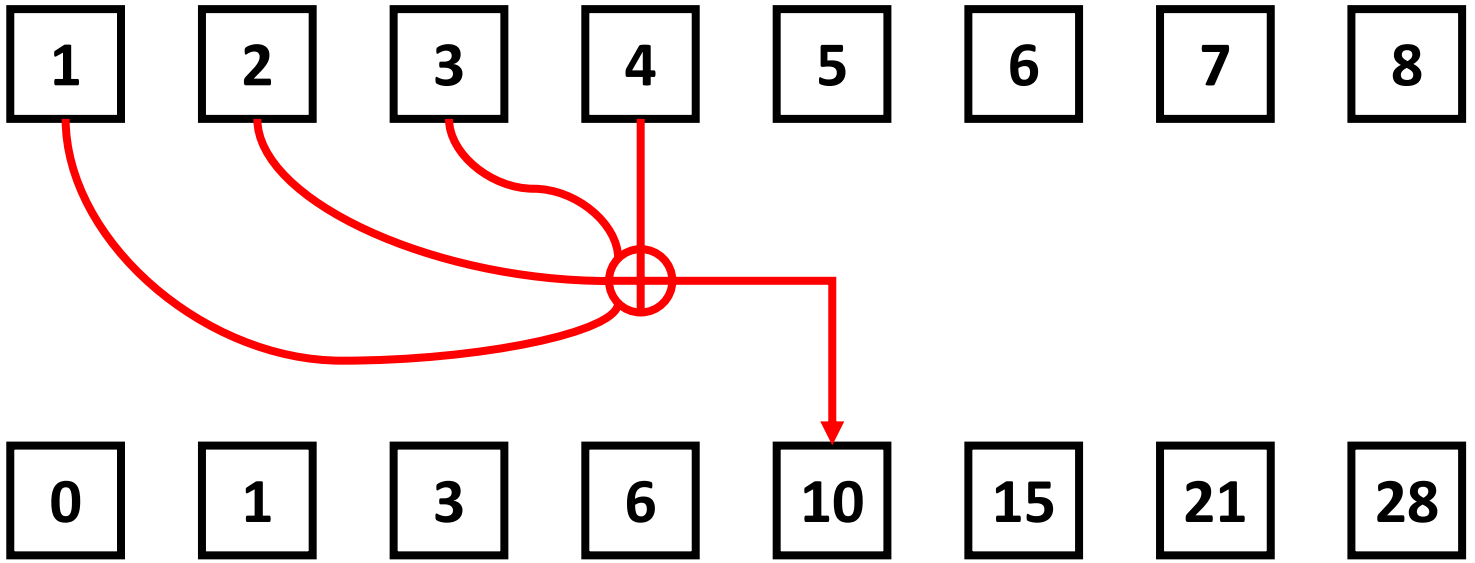


Compute partial reductions at each step of the sequence.

[1]Blelloch, Guy E. ”Prefix sums and their applications”. Technical Report (1990)

What is a Scan[1] Operation?

Binary, **associative** operator: **+** Identity: **0**



Compute partial reductions at each step of the sequence.

[1]Blelloch, Guy E. ”Prefix sums and their applications”. Technical Report (1990)

Linear Scan

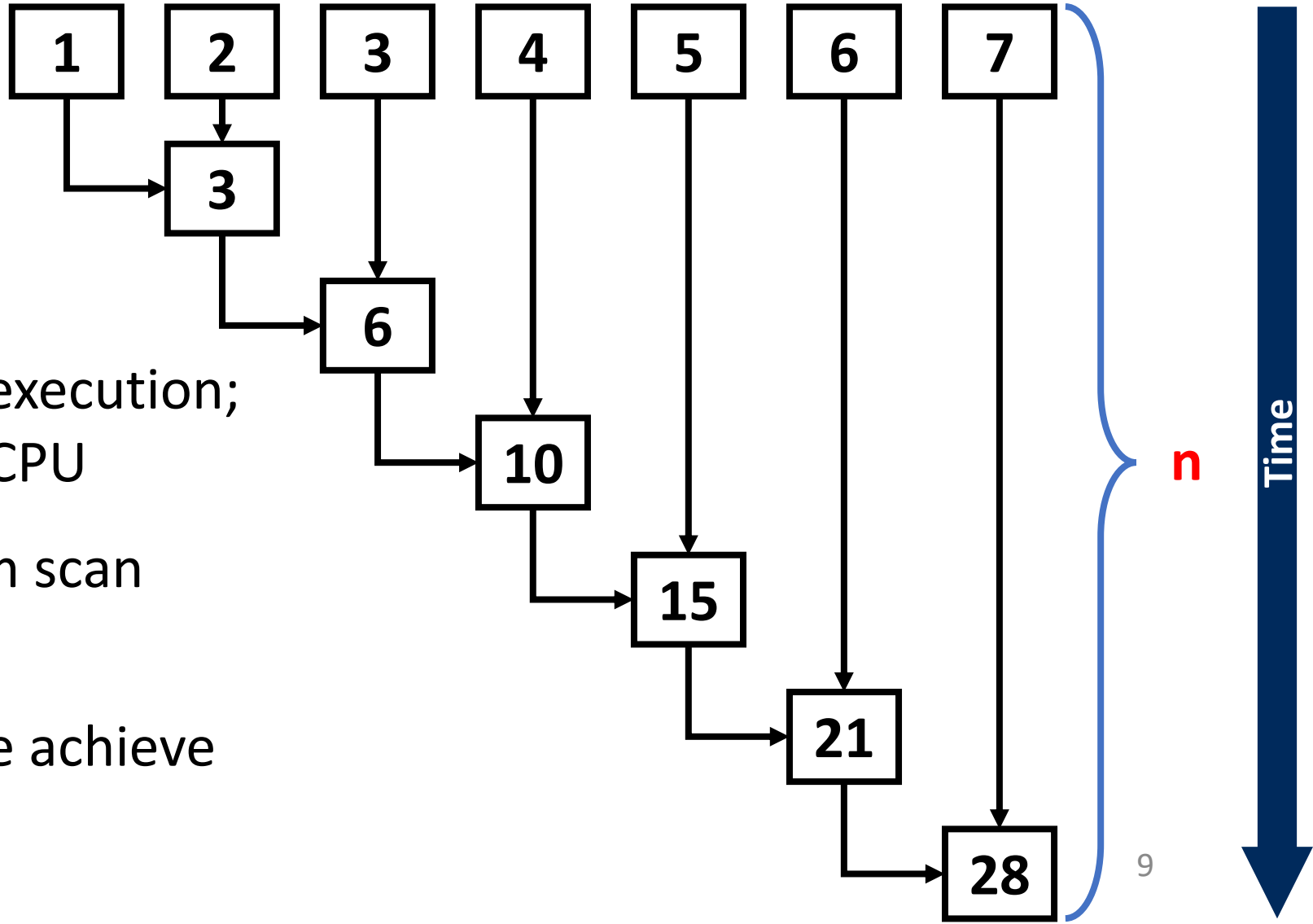
Step: executing the operator once.

Number of Elements (**n**)

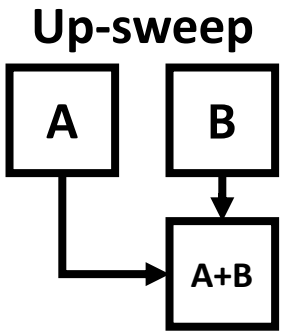
Worker (**p**): an instance of execution; e.g., a core in a multi-core CPU

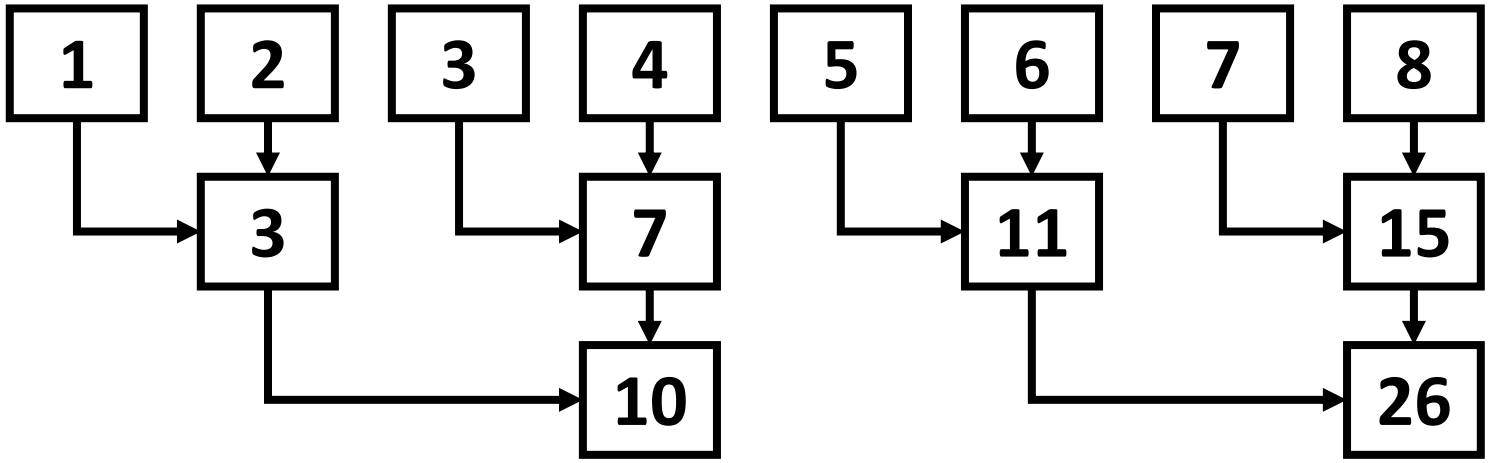
On a single worker: perform scan linearly; takes **n** steps.

With more workers: Can we achieve **sublinear** steps?



Blelloch Scan: ① Up-sweep Phase

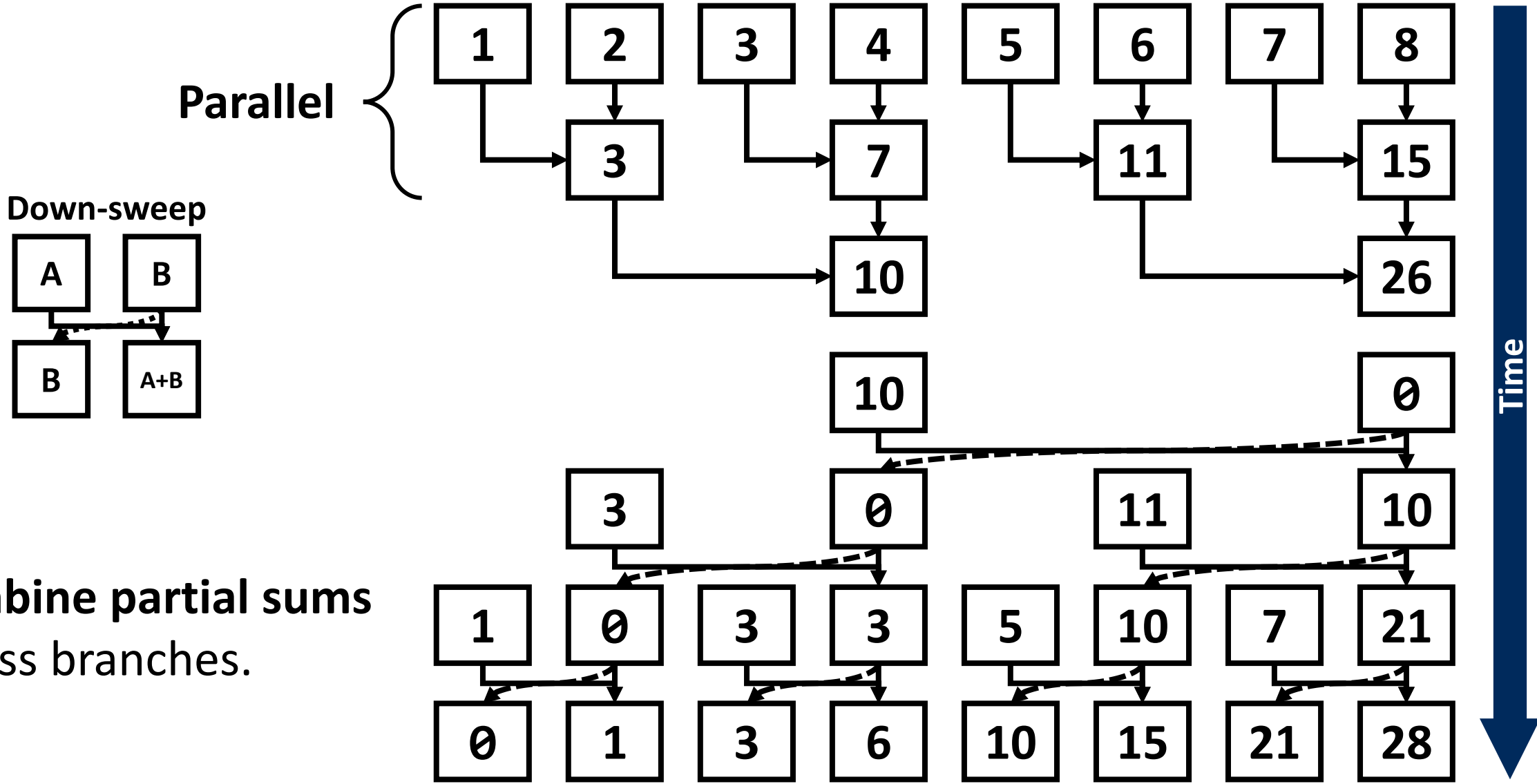




Time

Compute partial sums
via a **reduction tree**.

Blelloch Scan: ② Down-sweep Phase

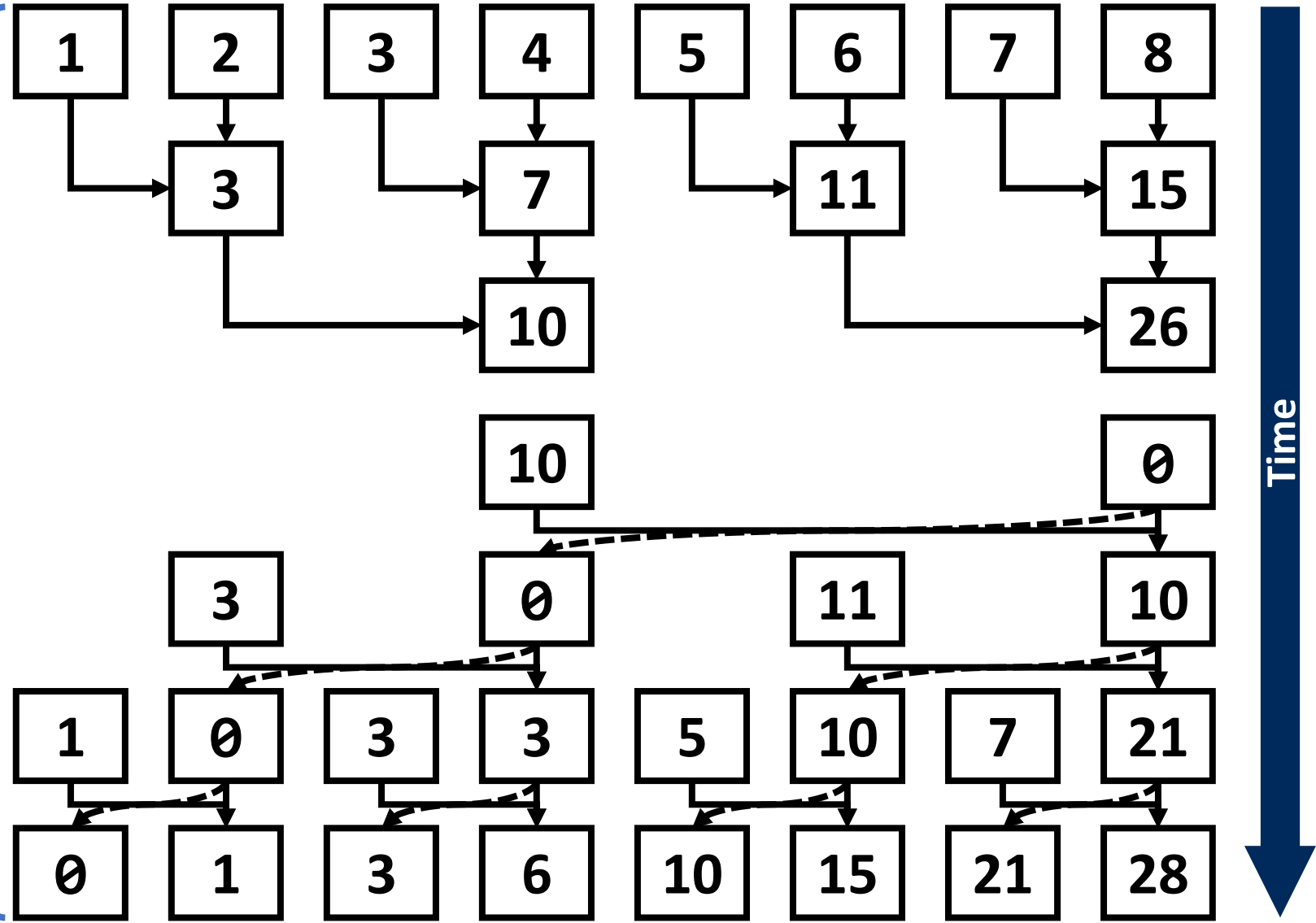


Combine partial sums across branches.

Blelloch Scan: Efficiency

Logarithmic steps along the critical path.

2logn



Reformulate BP as a Scan Operation

Binary, **associative** operator: **+** Identity: **0**

Input sequence: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Exclusive scan: | 0 | 1 | 3 | 6 | 10 | 15 | 21 | 28 |

Key Insight: **matrix multiplication** in **BP** is also **binary** & **associative**!

Reformulate BP as a Scan Operation

$$G_i = \nabla_{\vec{x}_i} l$$

$$J_{i+1} = \left(\frac{\partial \vec{x}_{i+1}}{\partial \vec{x}_i}\right)^T$$

Binary, **associative** operator: **A ◊ B = BA** Identity: **I**

Input sequence: G_7 J_7 J_6 J_5 J_4 J_3 J_2 J_1

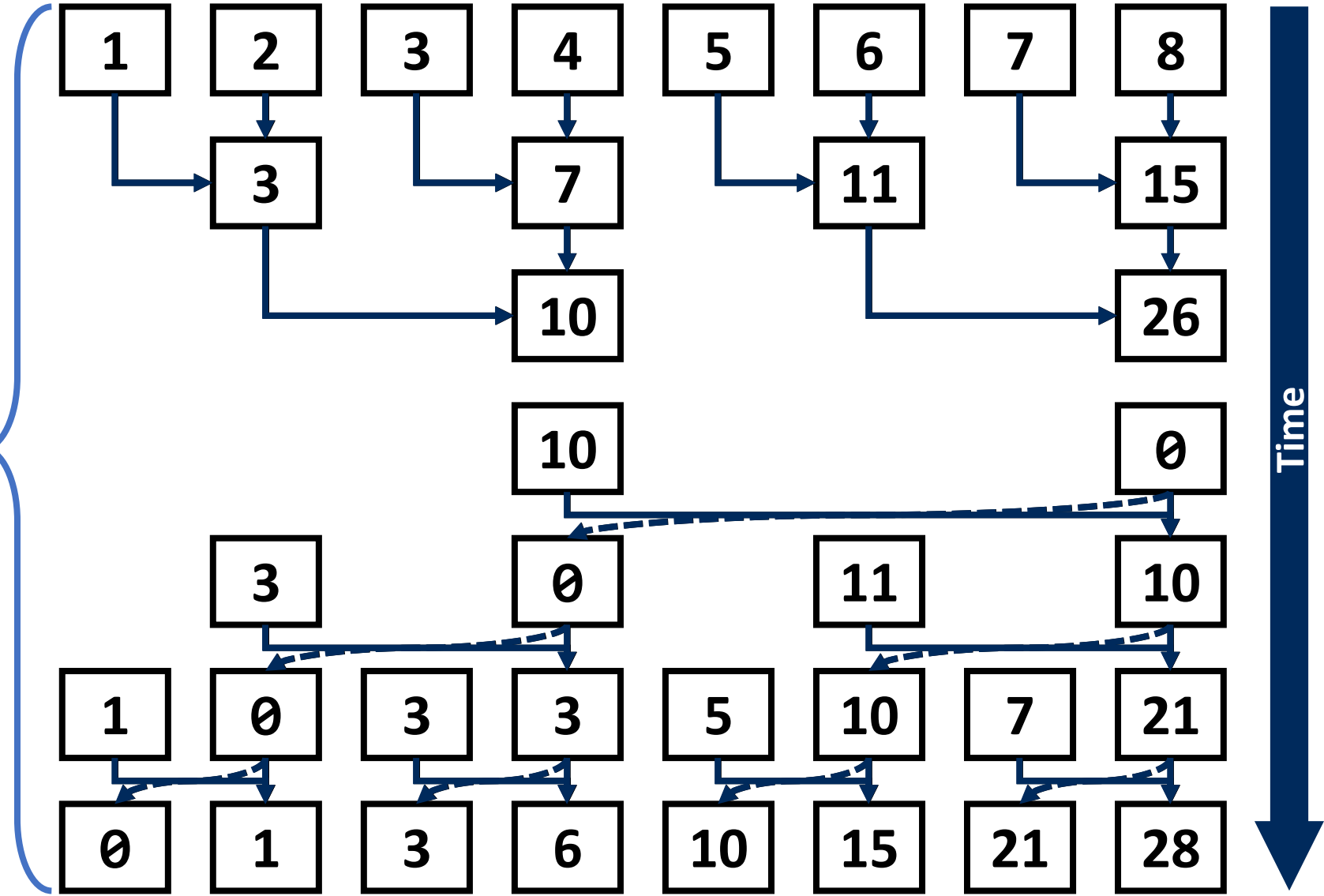
Exclusive scan: **I** G_7 G_6 G_5 G_4 G_3 G_2 G_1

Key Insight: **matrix multiplication** in **BP** is also **binary** & **associative**!

Scale BP by Blelloch Scan

Logarithmic steps along the critical path!

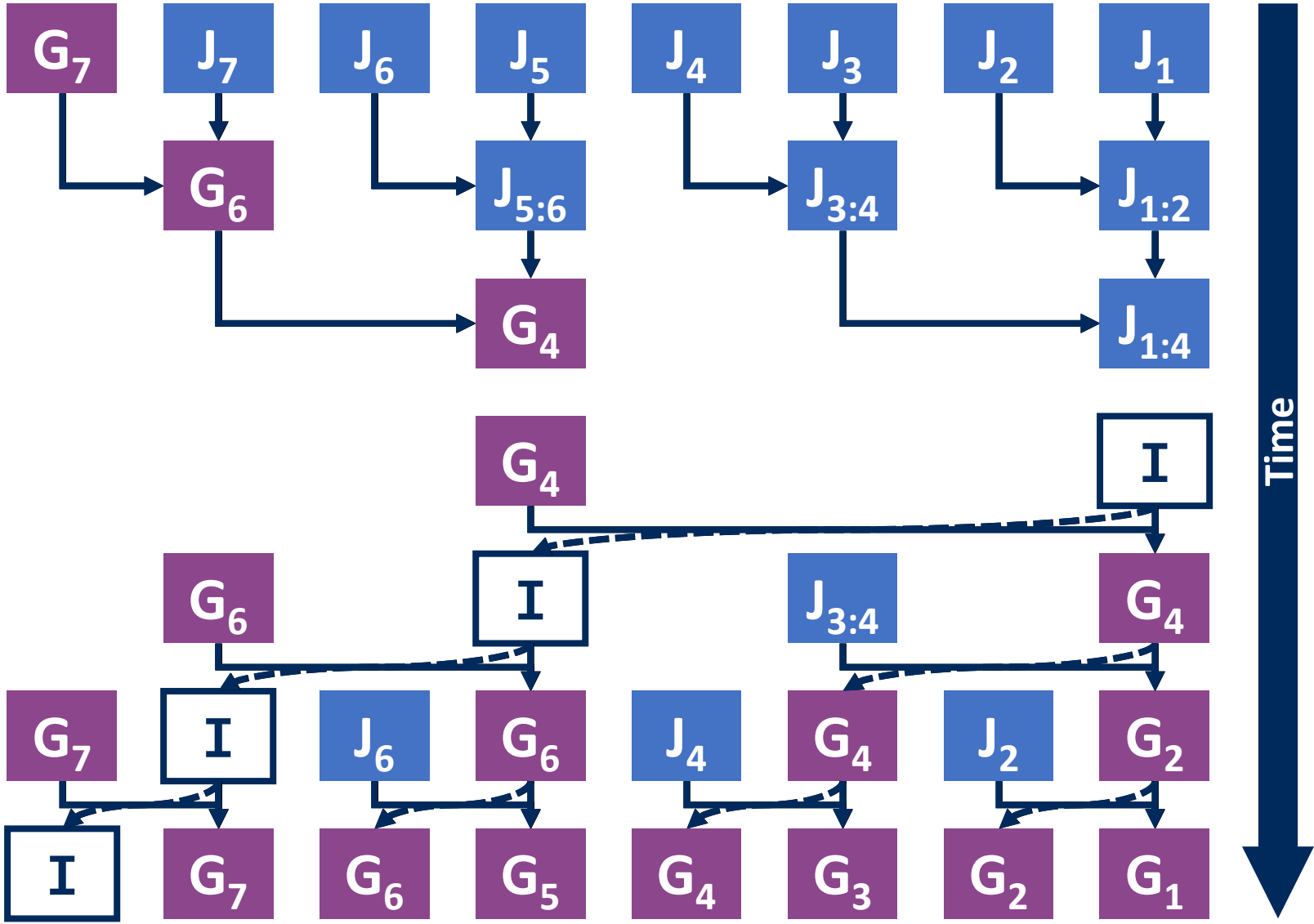
2logn



Scale BP by Blelloch Scan

Logarithmic steps along the critical path!

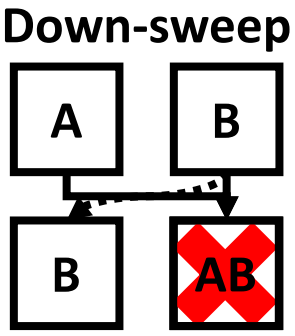
2logn



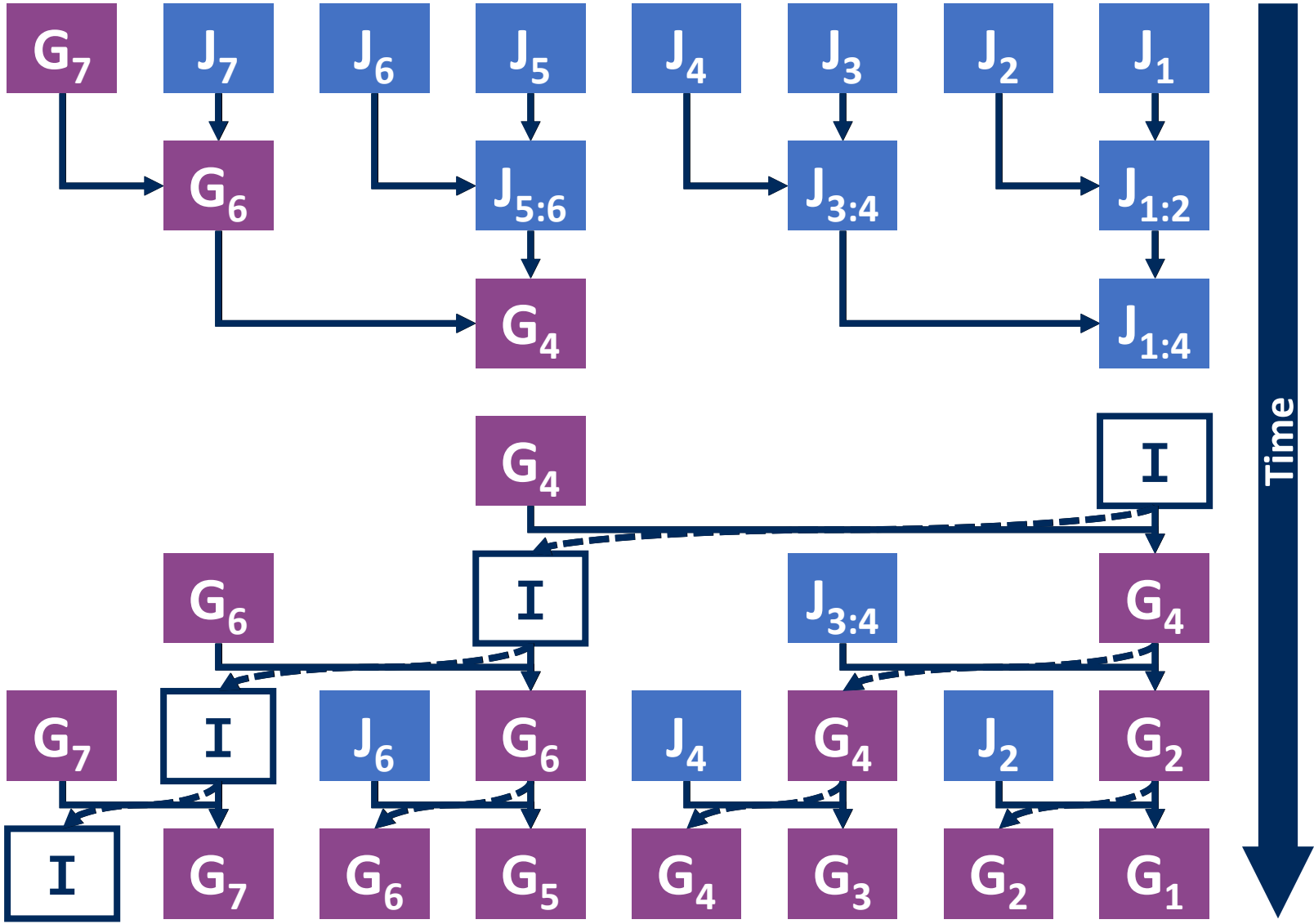
Scale BP by Blelloch Scan

Logarithmic steps along the critical path!

2logn



Matrix multiplications are **noncommutative**.

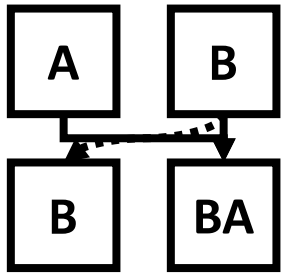


Scale BP by Blelloch Scan

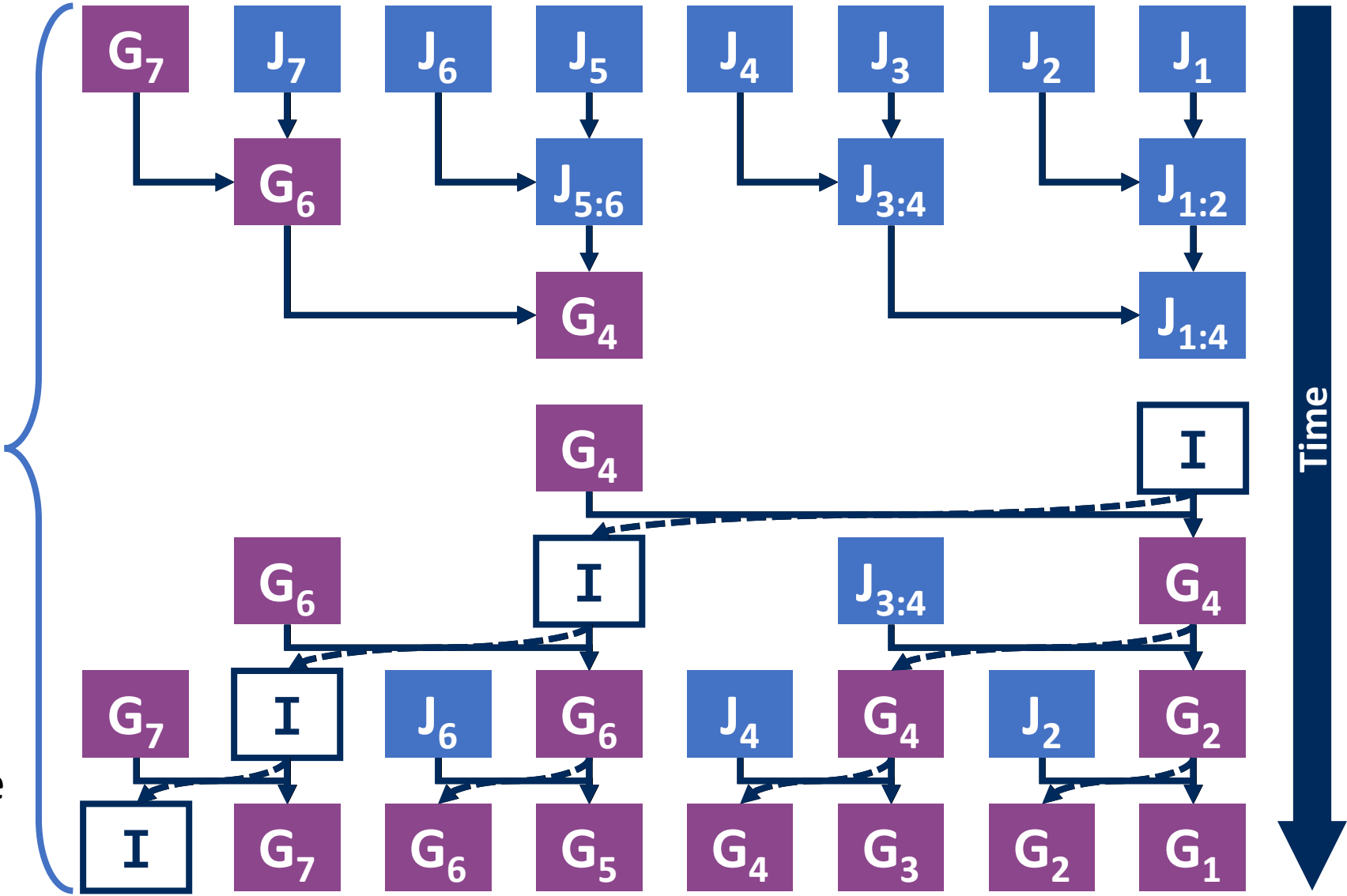
Logarithmic steps along the critical path!

2logn

Down-sweep



Matrix multiplications are **noncommutative**.



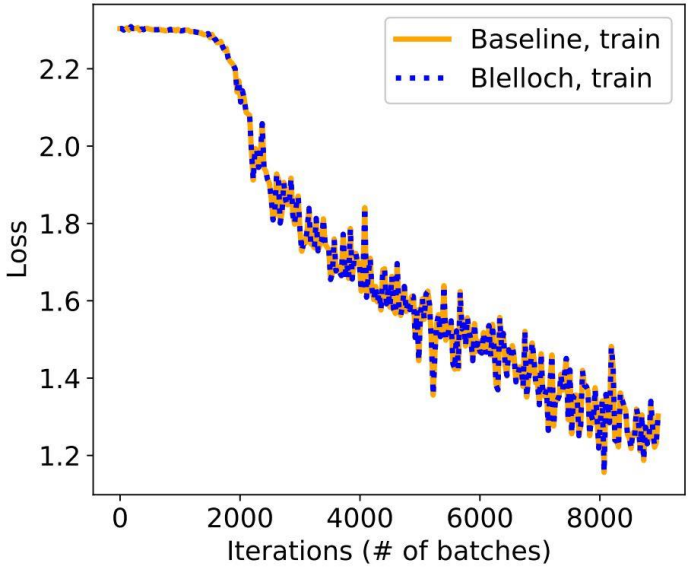
Reconstructs the Original BP Exactly

Our method produces gradients **mathematically equivalent** to BP.

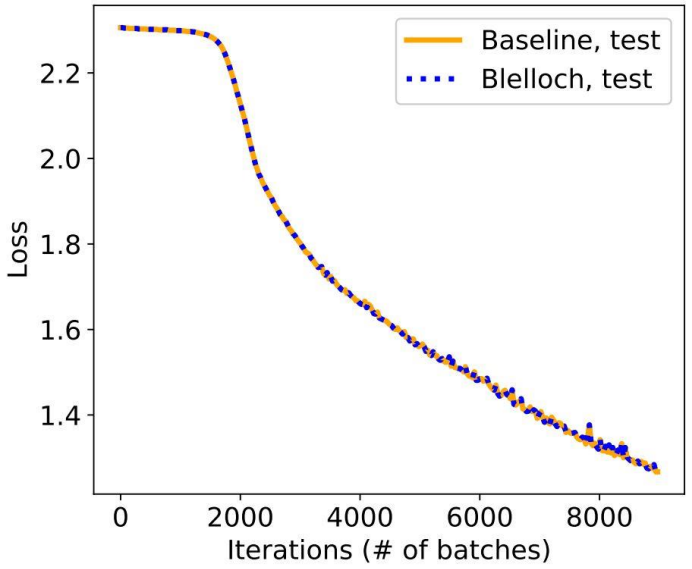
The Jacobians are multiplied in a different order → numerical differences.

Empirically show that such differences do not effect convergence.

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



(a) Training loss per iteration.

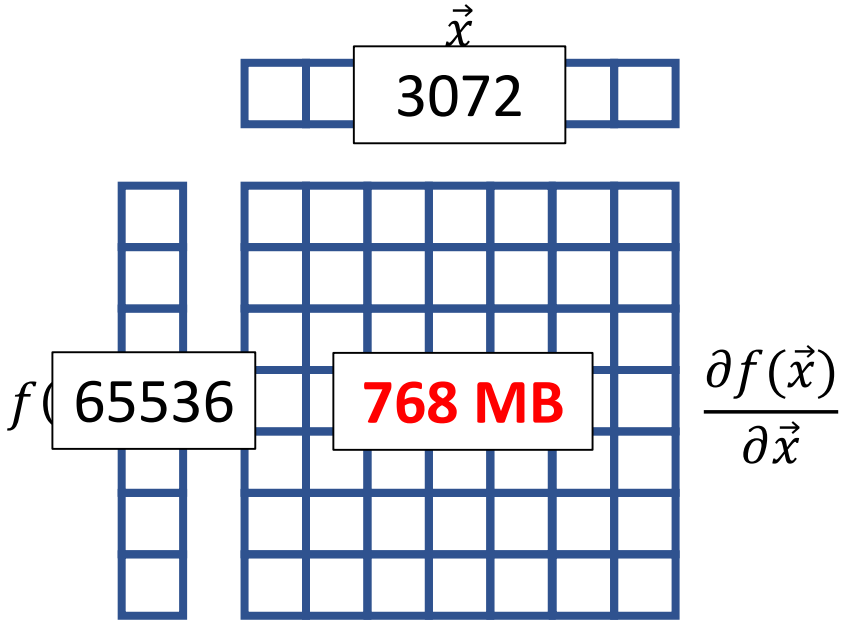


(b) Test loss per iteration.

Jacobians are Memory & Compute Hungry

A full Jacobian can be prohibitively expensive to handle.

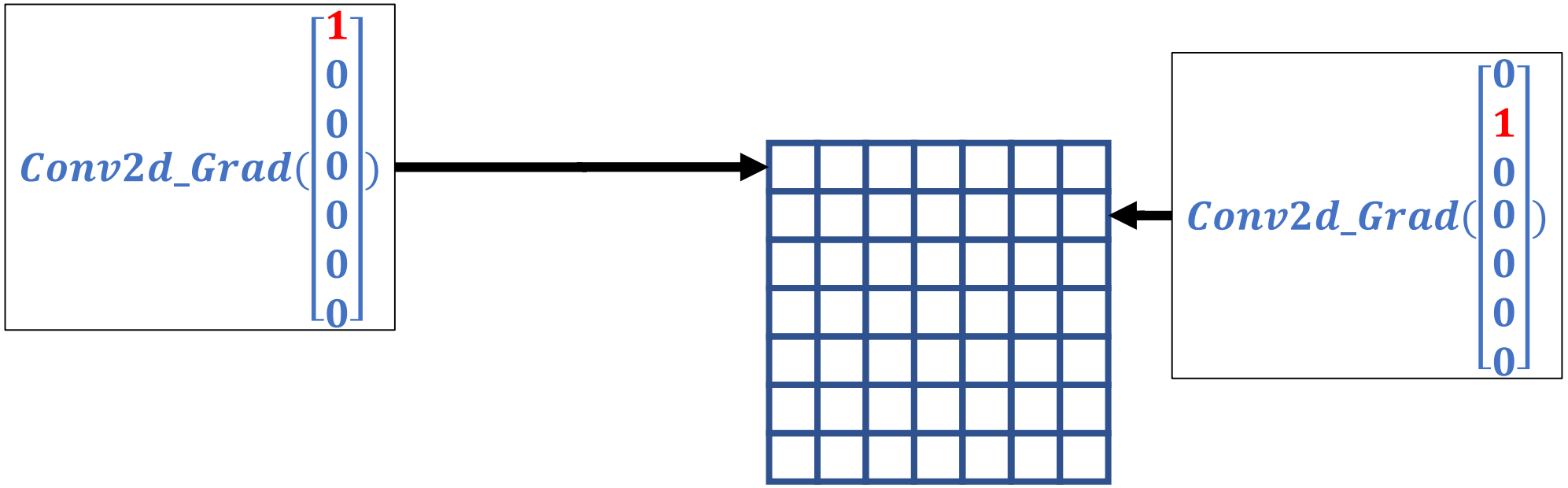
- e.g., 1st convolution in VGG-11 on CIFAR-10 images occupy **768 MB** of memory.



Jacobians are Memory & Compute Hungry

A full Jacobian can be prohibitively expensive to handle.

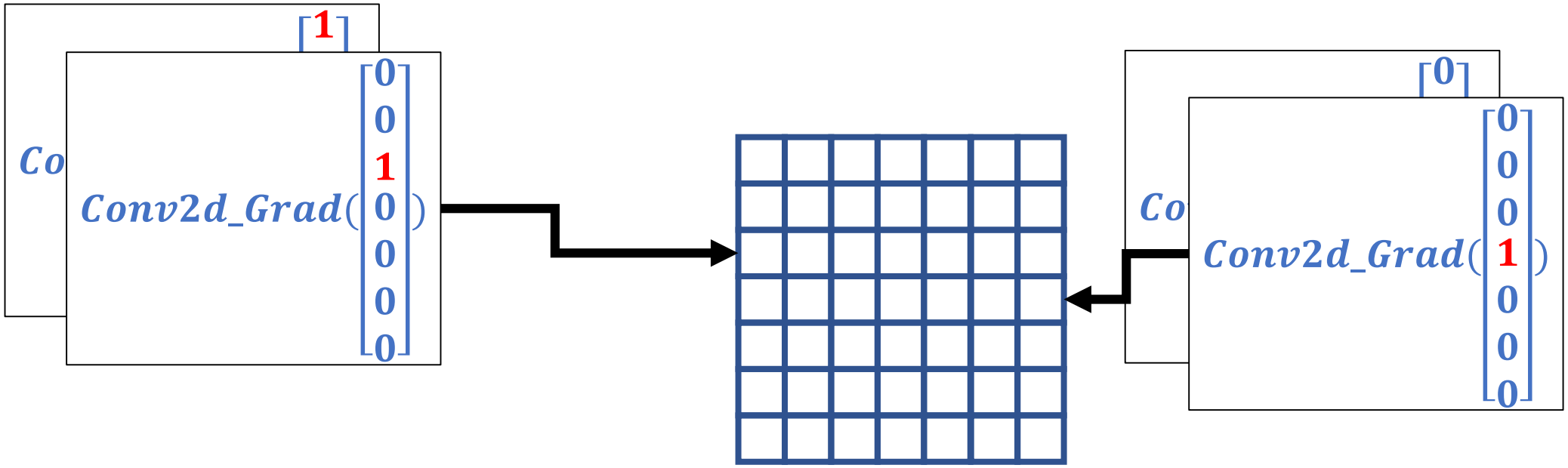
- e.g., 1st convolution in VGG-11 on CIFAR-10 images occupy **768 MB** of memory.
- Generated one row at a time by passing basis vectors into `Op_Grad()` (the VJP function).



Jacobians are Memory & Compute Hungry

A full Jacobian can be prohibitively expensive to handle.

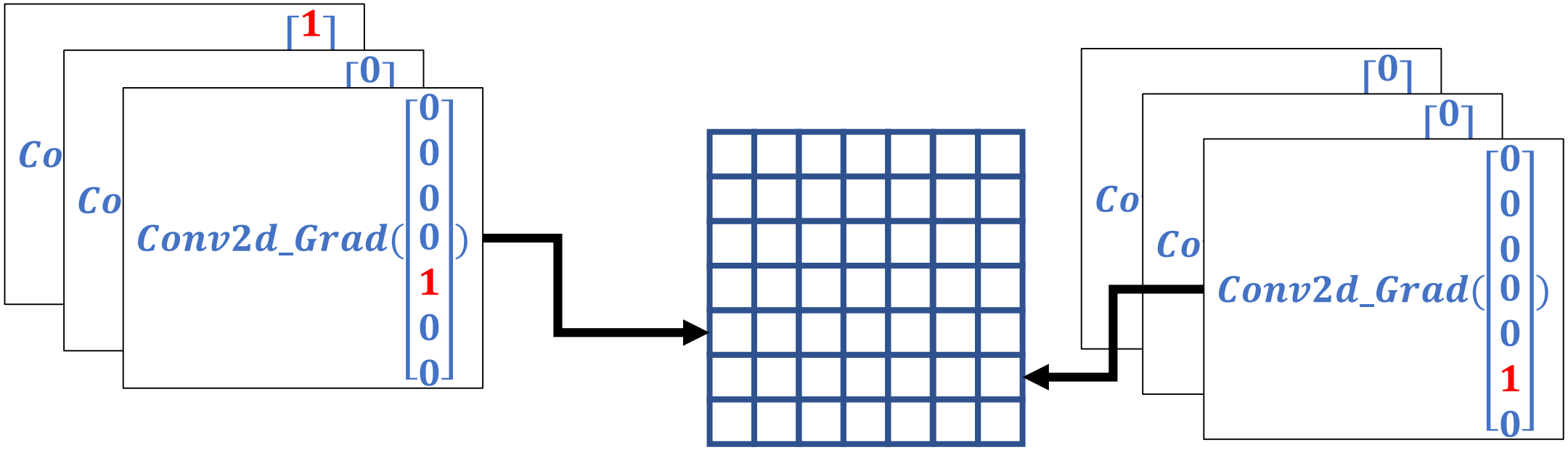
- e.g., 1st convolution in VGG-11 on CIFAR-10 images occupy **768 MB** of memory.
- Generated one row at a time by passing basis vectors into `Op_Grad()` (the VJP function).



Jacobians are Memory & Compute Hungry

A full Jacobian can be prohibitively expensive to handle.

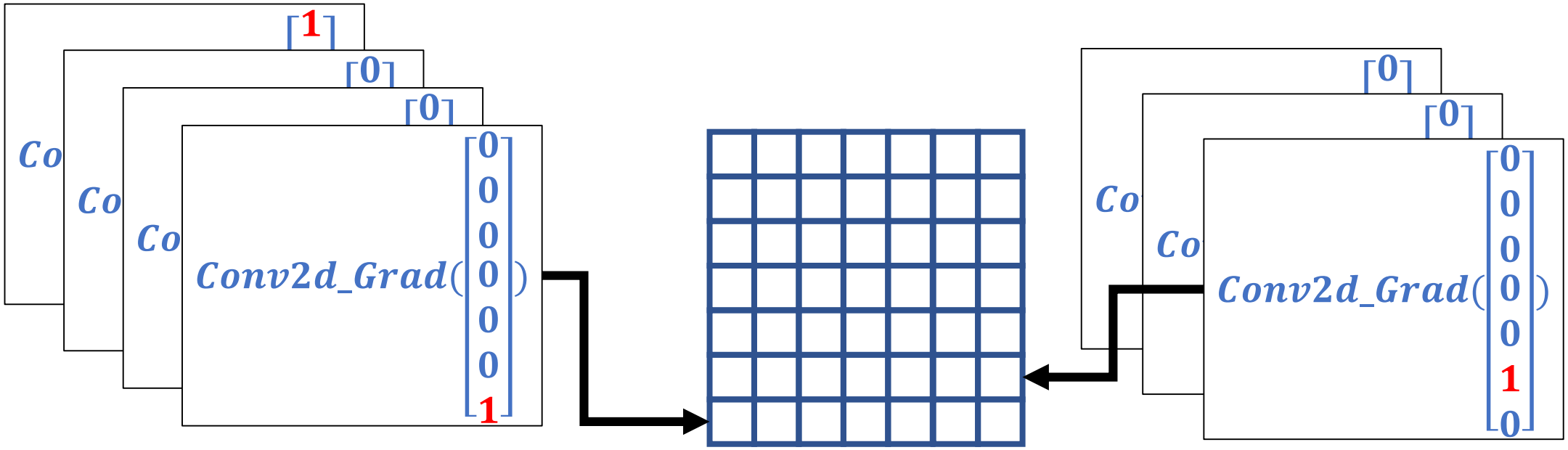
- e.g., 1st convolution in VGG-11 on CIFAR-10 images occupy **768 MB** of memory.
- Generated one row at a time by passing basis vectors into `Op_Grad()` (the VJP function).



Jacobians are Memory & Compute Hungry

A full Jacobian can be prohibitively expensive to handle.

- e.g., 1st convolution in VGG-11 on CIFAR-10 images occupy **768 MB** of memory.
- Generated one row at a time by passing basis vectors into `Op_Grad()` (the VJP function).



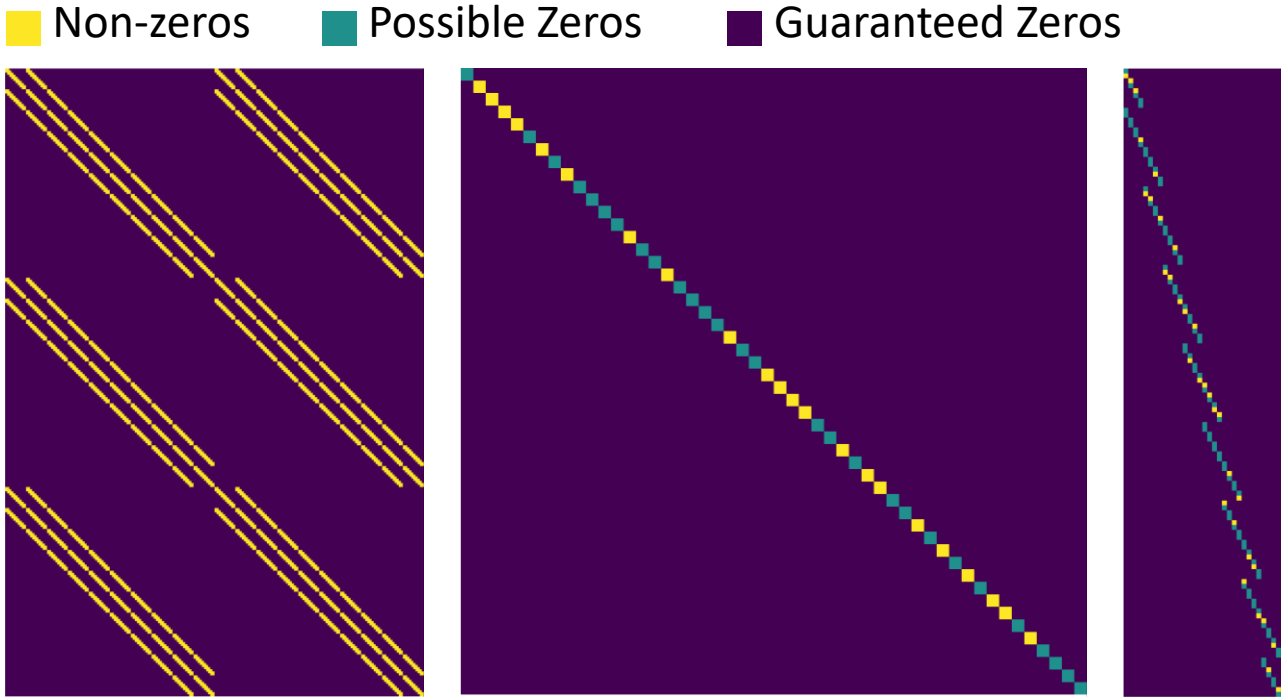
Jacobians are Memory & Compute Hungry

A full Jacobian can be prohibitively expensive to handle.

- e.g., 1st convolution in VGG-11 on CIFAR-10 images occupy **768 MB** of memory.
- Generated one row at a time by passing basis vectors into `Op_Grad()` (the VJP function).

Conventional ML algorithms avoid using Jacobians directly (including BP).

The Jacobians of Many Operators are Sparse



Guaranteed zeros:

Known **ahead of training time**.

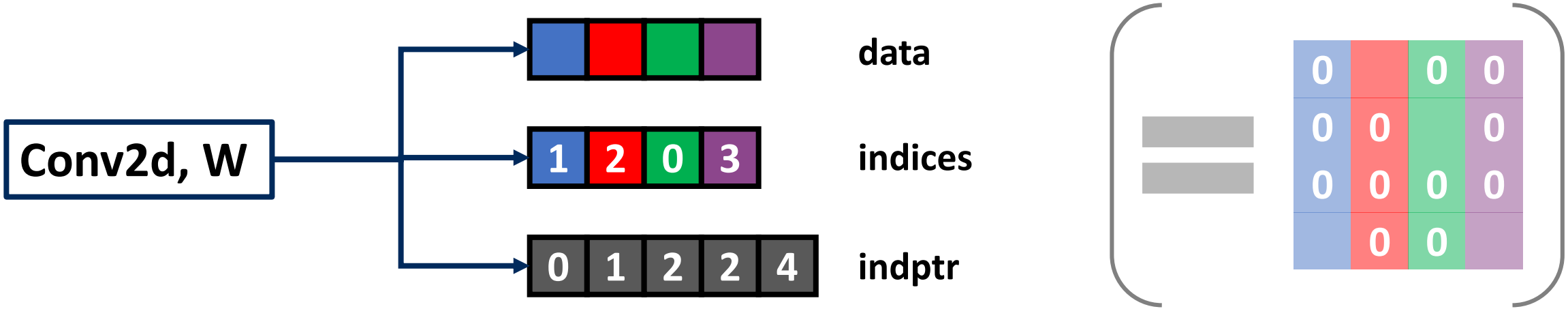
Deterministic pattern.

Potentially **better** SpGEMM performance.

First three ops of VGG-11 on CIFAR-10	Convolution	ReLU	Max Pooling
Sparsity	0.**99**157	0.**99**998	0.**99**994

Fast Sparse Jacobians Generation

Therefore, instead of calculating the Jacobians row-wise, generate **directly** into **Compressed Sparse Row (CSR)**:



First three ops of VGG-11 on CIFAR-10	Convolution	ReLU	Max Pooling
Jacobian Calculation Speedup	8.3×10^3 x	1.2×10^6 x	1.5×10^5 x

Complexity Analysis

Runtime:

BPPSA BP

C_{BPPSA} $\Theta(\log n)$ vs. C_{BP} $\Theta(n)$

Per-step Complexity (**C**): runtime of each step.

Complexity Analysis

Per-step Complexity (**C**): runtime of each step.

Runtime:

BPPSA		BP
C_{BPPSA} $\Theta(\log n)$	vs.	C_{BP} $\Theta(n)$

Performance benefits:

1. Large **n**: deep network, long sequential dependency.

Complexity Analysis

Per-step Complexity (**C**): runtime of each step.

Runtime:

BPPSA		BP
C_{BPPSA} $\Theta(\log n)$	vs.	C_{BP} $\Theta(n)$

Performance benefits:

1. Large **n**: deep network, long sequential dependency.
2. Reducing per-step complexity: SpGEMM.

Complexity Analysis

Runtime:

BPPSA BP

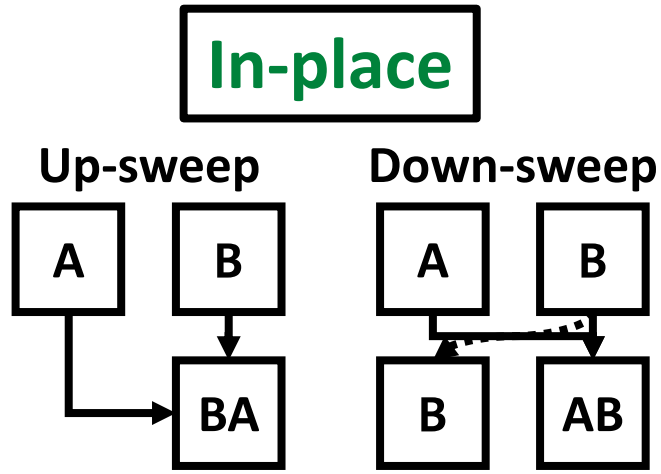
C_{BPPSA} $\Theta(\log n)$ vs. C_{BP} $\Theta(n)$

Per-step Complexity (**C**): runtime of each step.

Performance benefits:

1. Large **n**: deep network, long sequential dependency.
2. Reducing per-step complexity: SpGEMM.

Constant per-device space complexity!

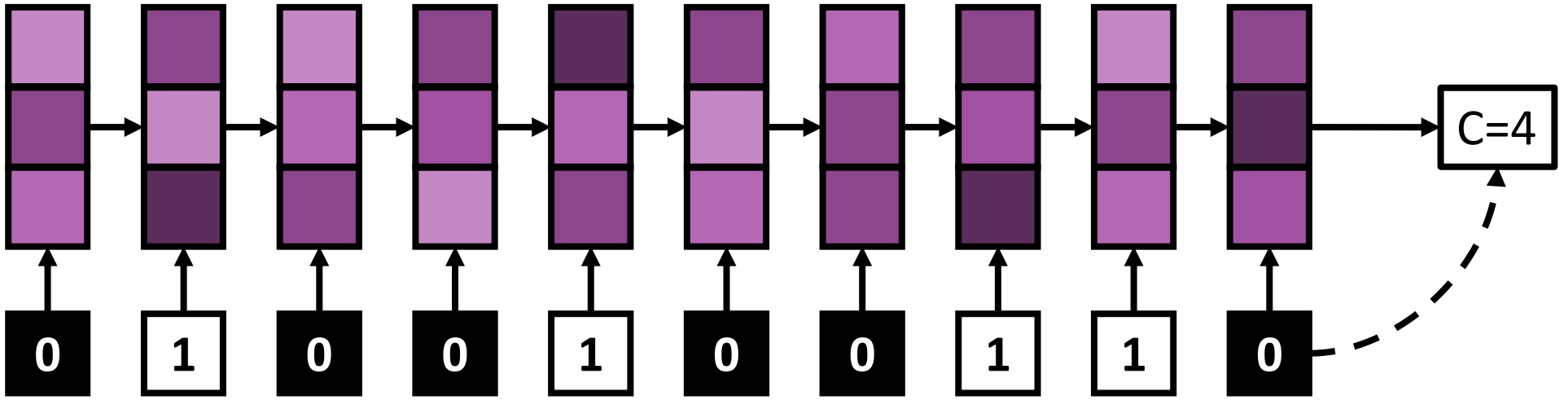


Methodology: Benchmark

Model: RNN

Task: Bitstream Classification

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



$$x_t^{(k)} \sim Bernoulli(0.05 + C^{(k)} \times 0.1)$$

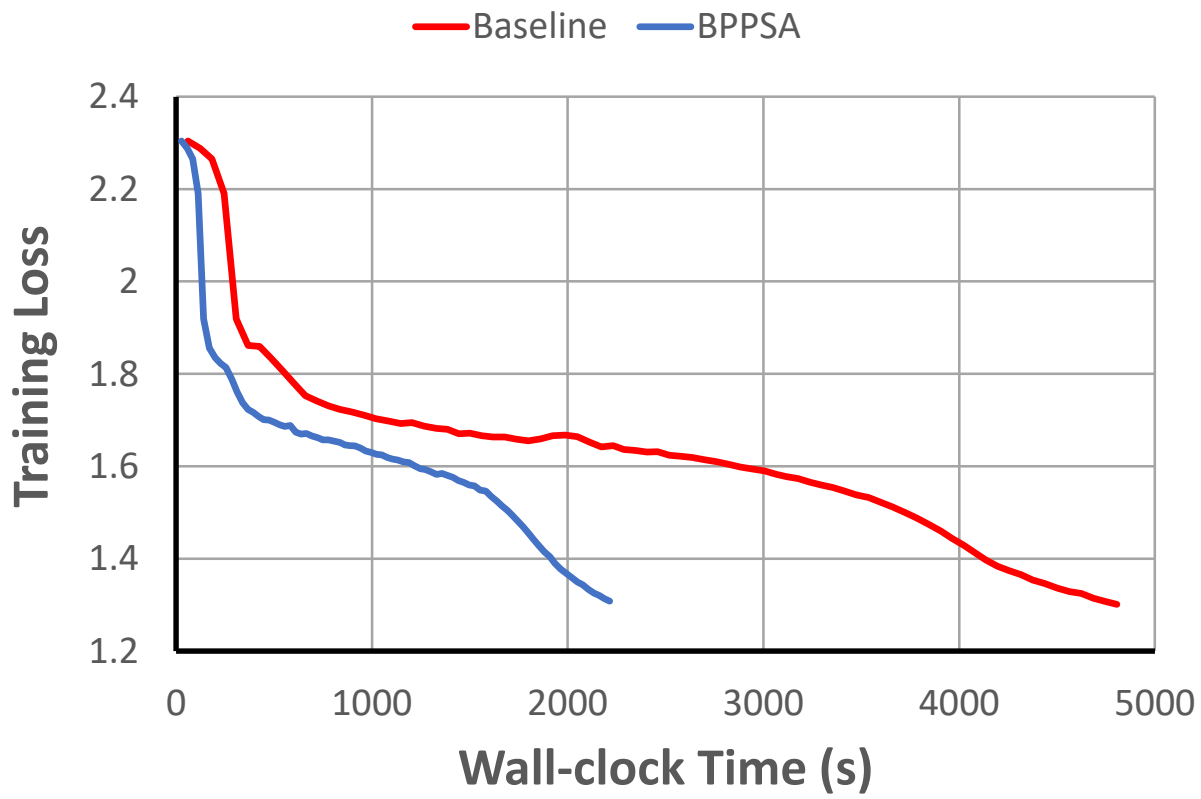
Methodology: Environment

		RTX 2070	RTX 2080 Ti
Hardware:			
Baseline:	cuDNN	7.5.1	7.6.2
	PyTorch	1.1	1.2

Implementation: custom CUDA 10 kernels.

End-to-end Training Speedup

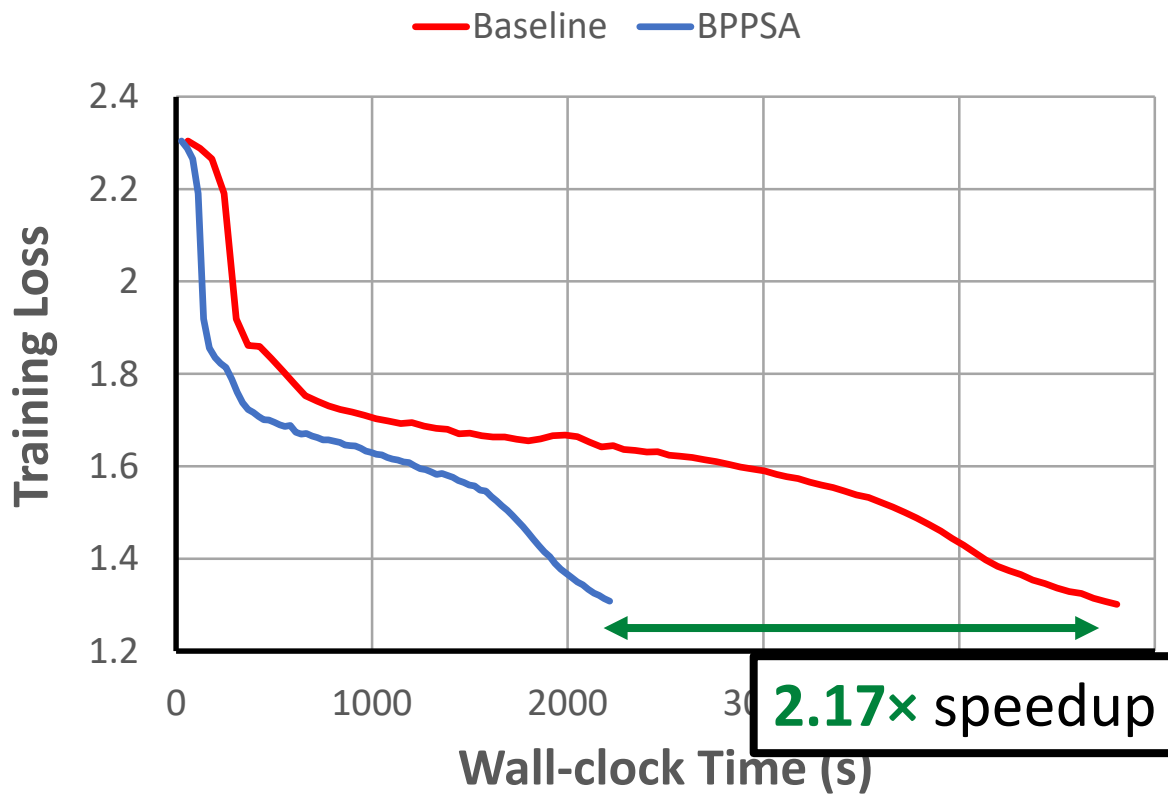
Training curve of BPPSA v.s. the baseline
when batch size **B**=16, sequence length **T**=1000:



Numerical differences do **not** effect convergence.

End-to-end Training Speedup

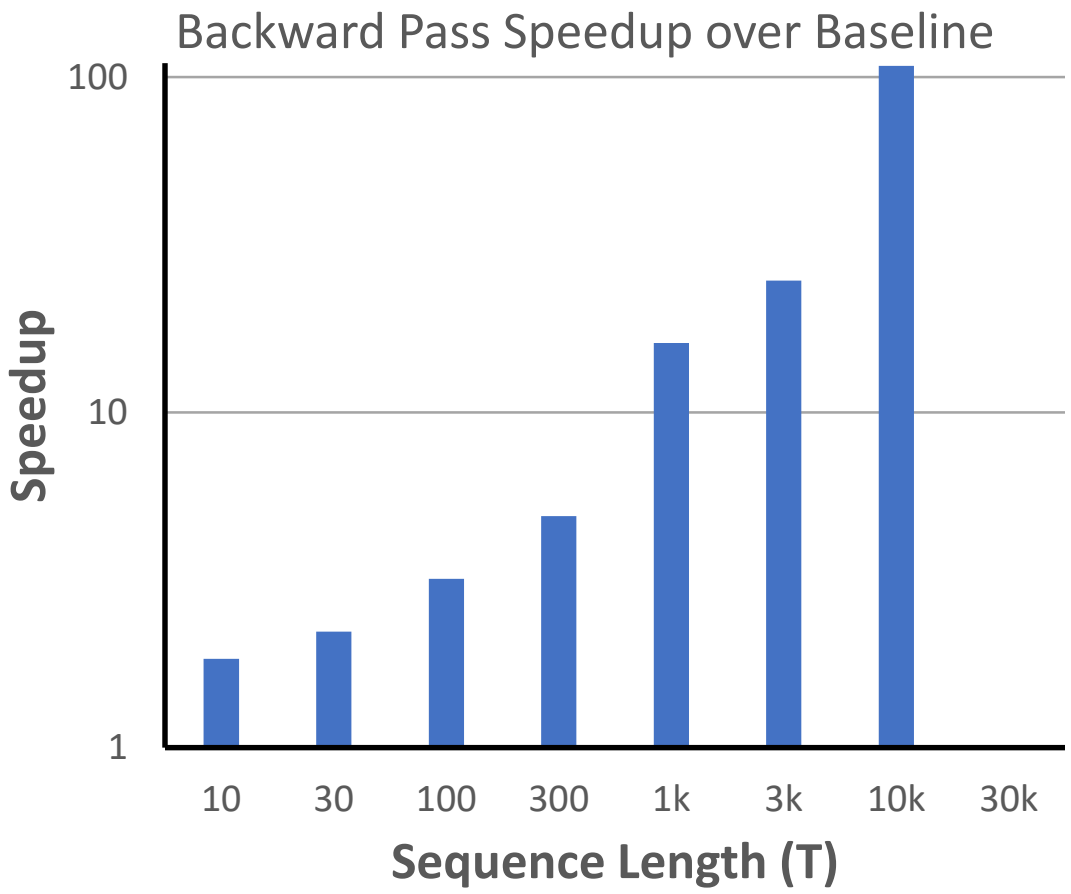
Training curve of BPPSA v.s. the baseline
when batch size **B**=16, sequence length **T**=1000:



Numerical differences do **not** effect convergence.

2.17× speedup on the overall training time.

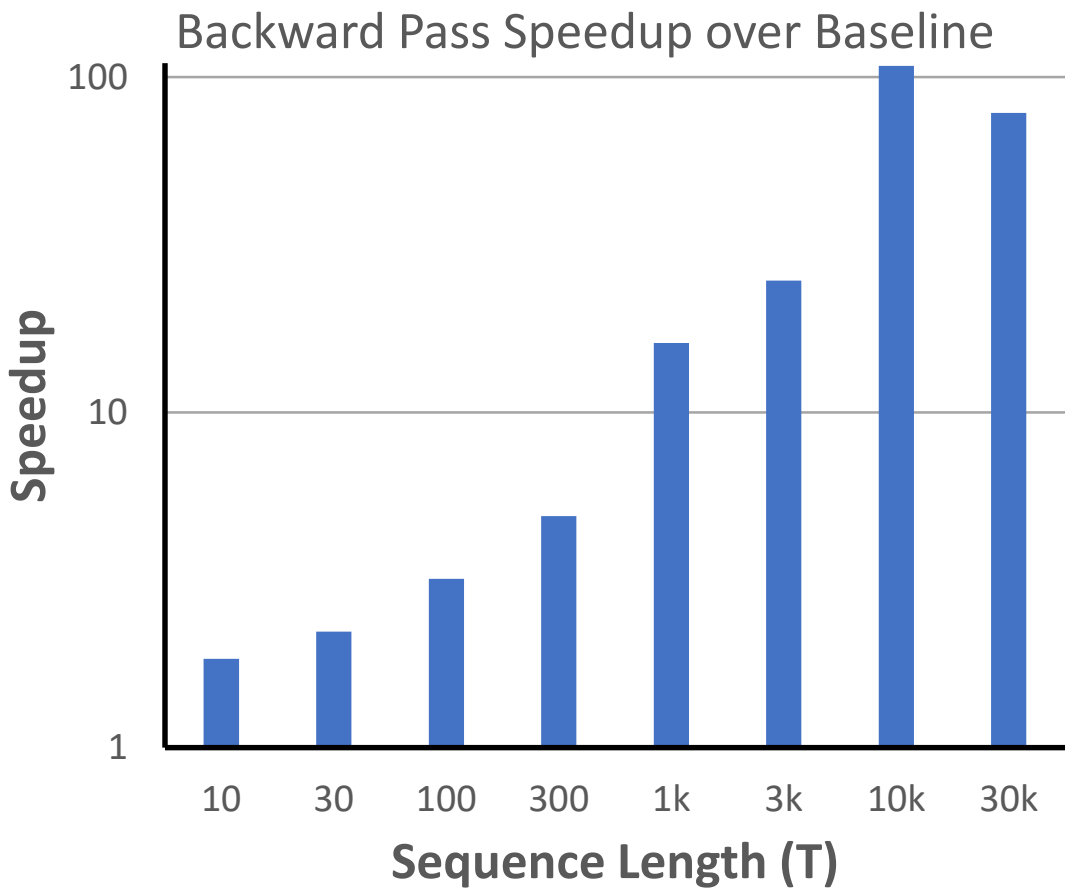
Sensitivity Analysis: Model Length



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

BPPSA **scales** with the model length (**n**);

Sensitivity Analysis: Model Length

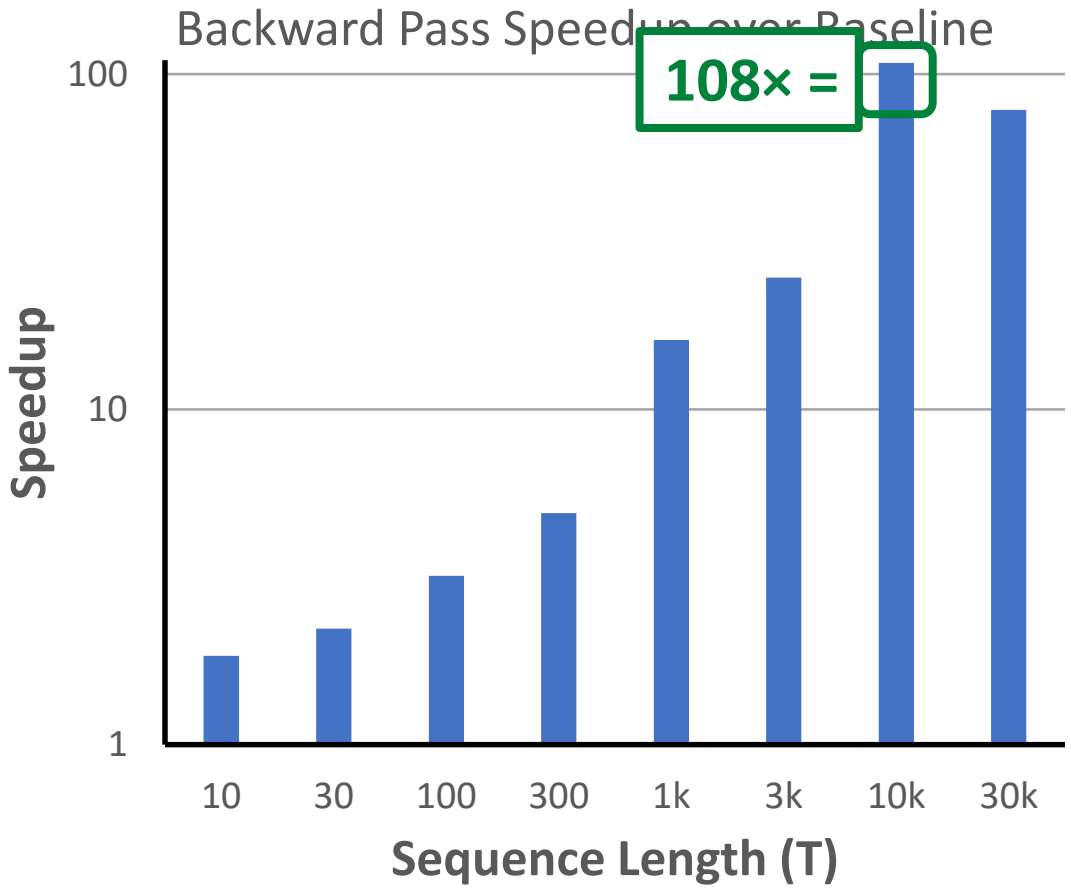


Sequence length (**T**) reflects the model length **n**.

BPPSA **scales** with the model length (**n**);

until being bounded by the number of workers (**p**).

Sensitivity Analysis: Model Length

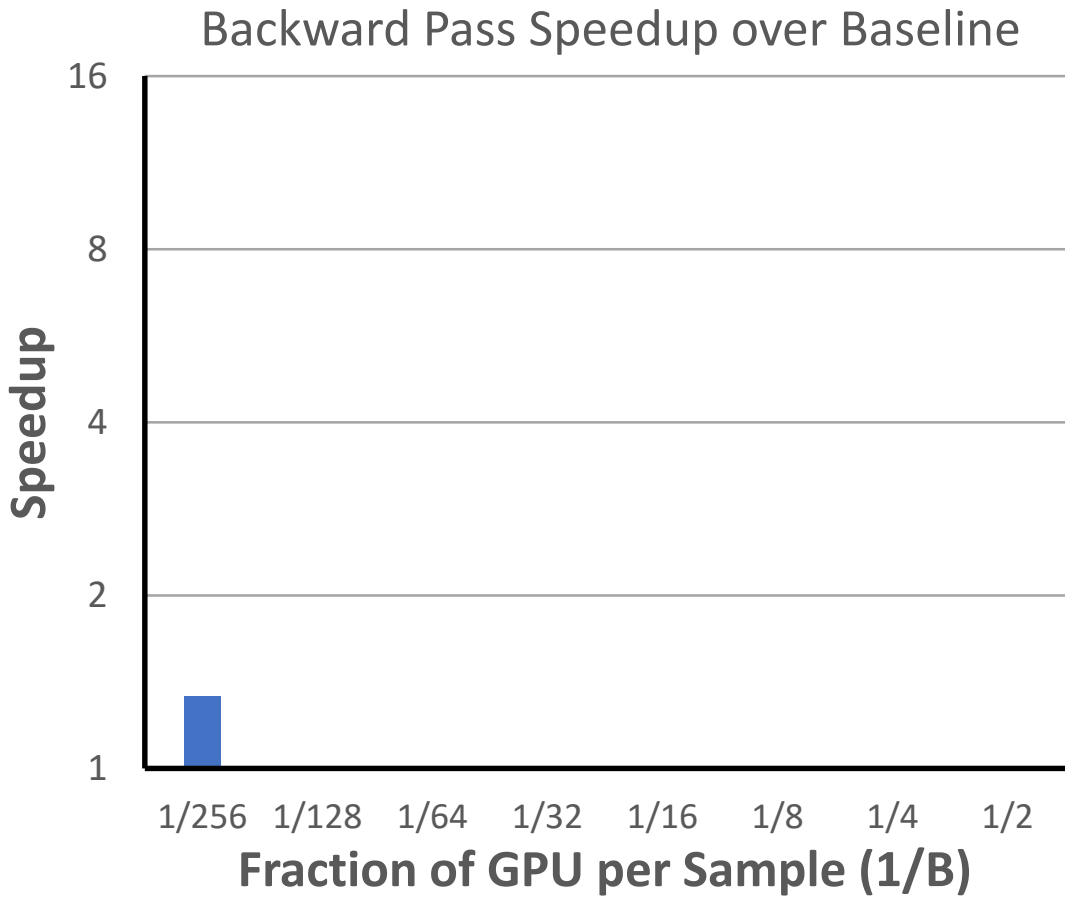


Sequence length (**T**) reflects the model length **n**.

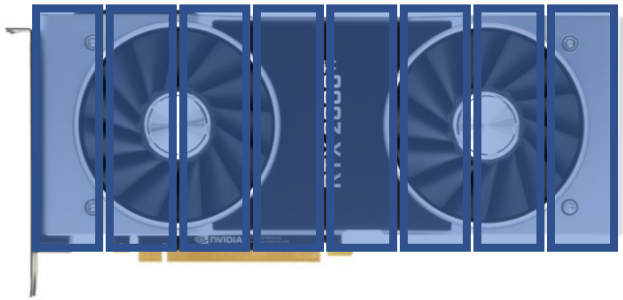
BPPSA **scales** with the model length (**n**);

until being bounded by the number of workers (**p**).

Sensitivity Analysis: Number of Workers

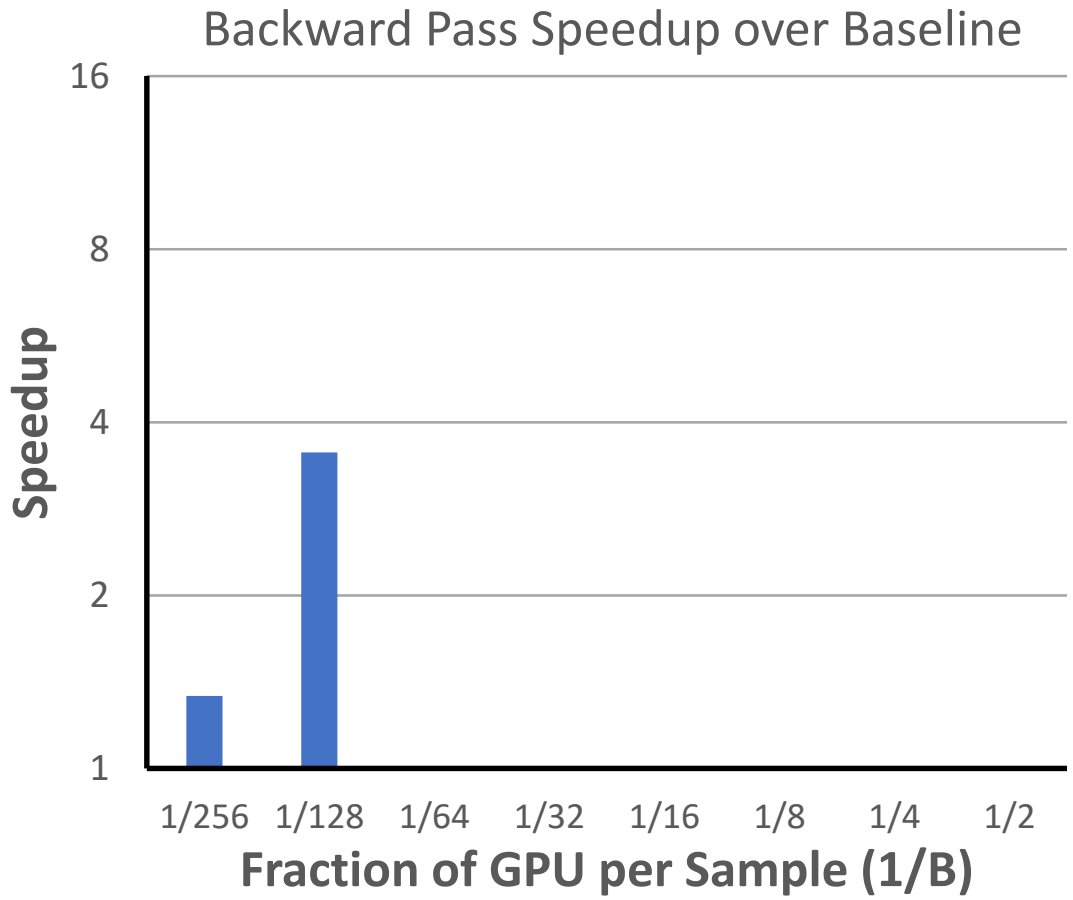


Fraction of GPU per sample (**1/B**) reflects the number of workers **p**.

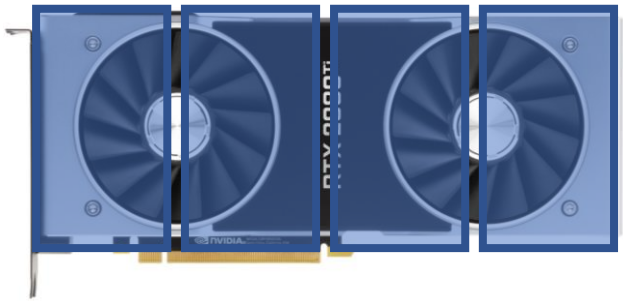


BPPSA **scales** with the number of workers (**p**).

Sensitivity Analysis: Number of Workers

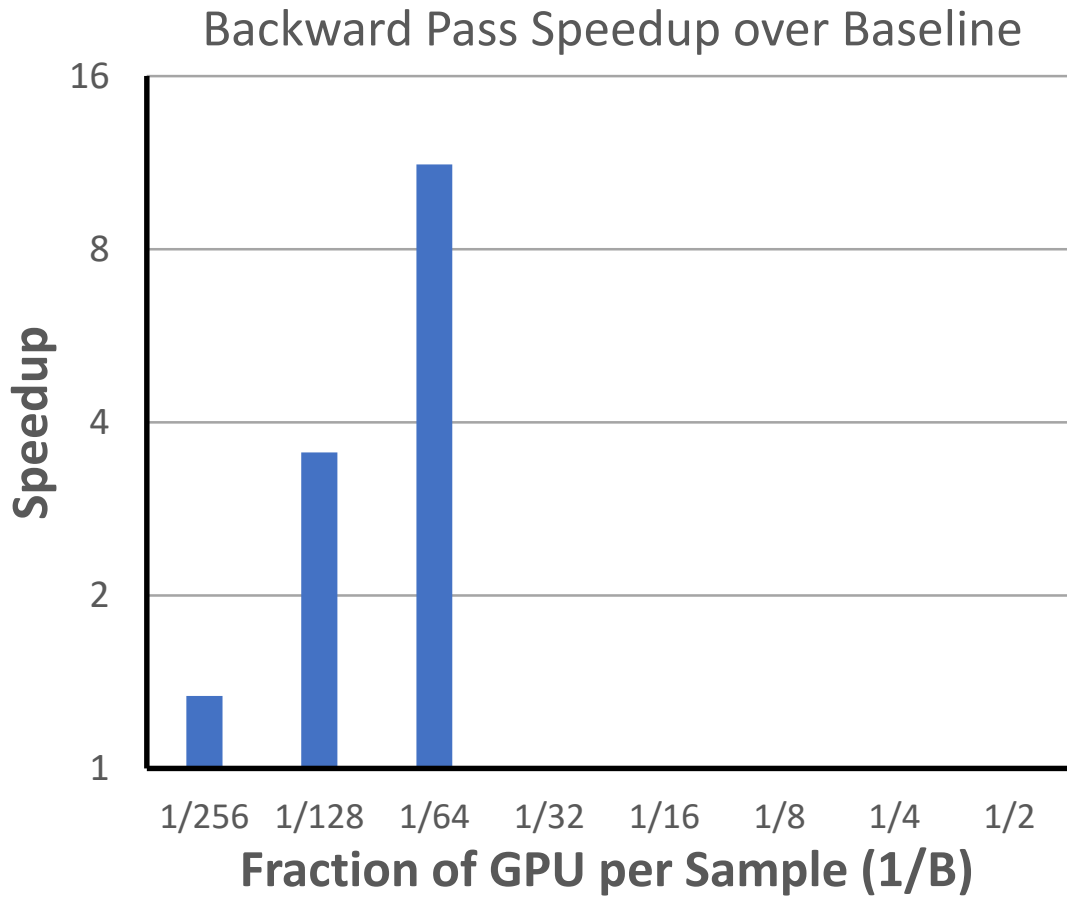


Fraction of GPU per sample (**1/B**) reflects the number of workers **p**.

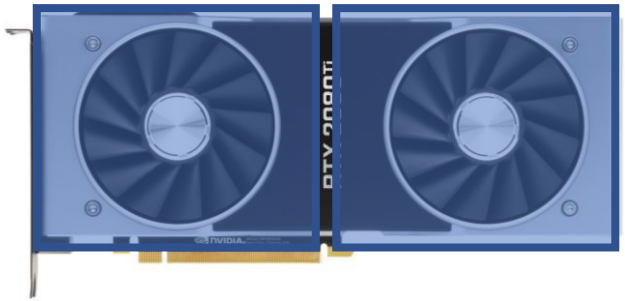


BPPSA **scales** with the number of workers (**p**).

Sensitivity Analysis: Number of Workers

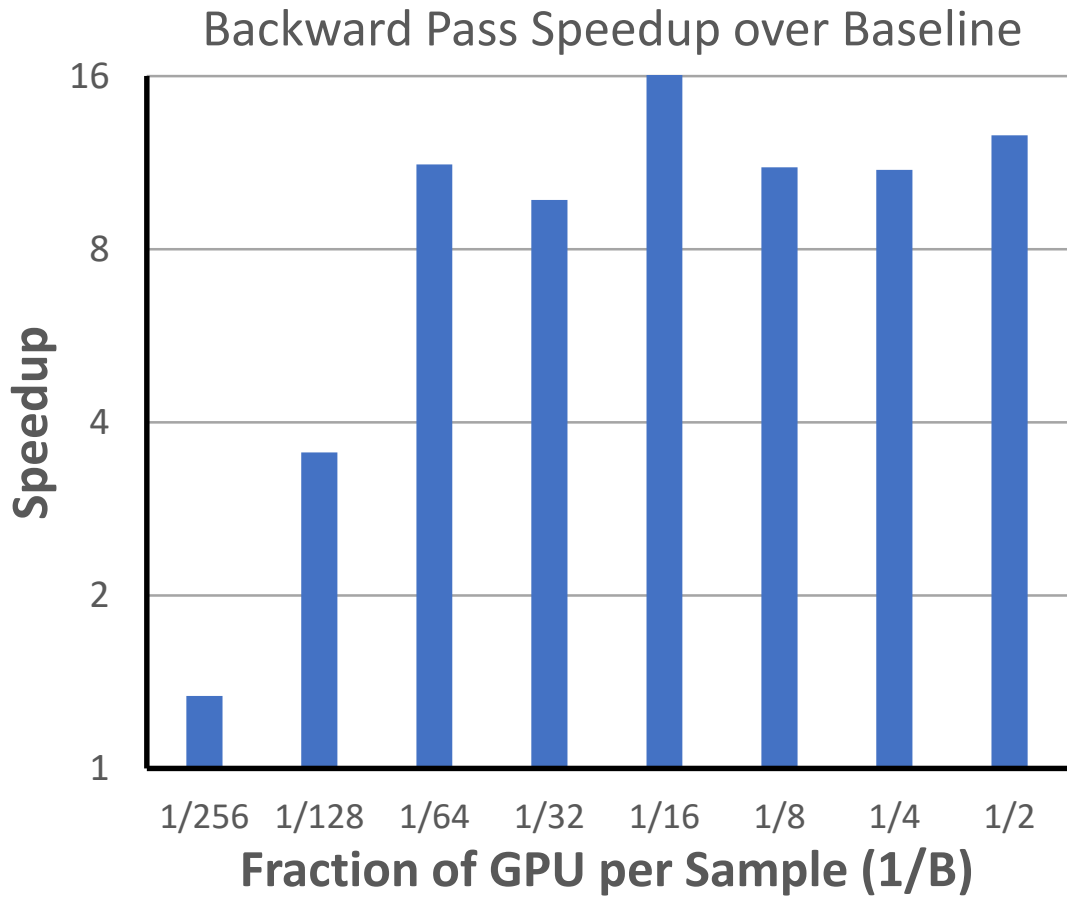


Fraction of GPU per sample (**1/B**) reflects the number of workers **p**.

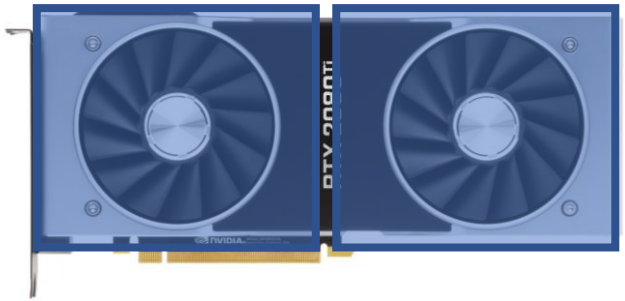


BPPSA **scales** with the number of workers (**p**).

Sensitivity Analysis: Number of Workers



Fraction of GPU per sample (**1/B**) reflects the number of workers **p**.

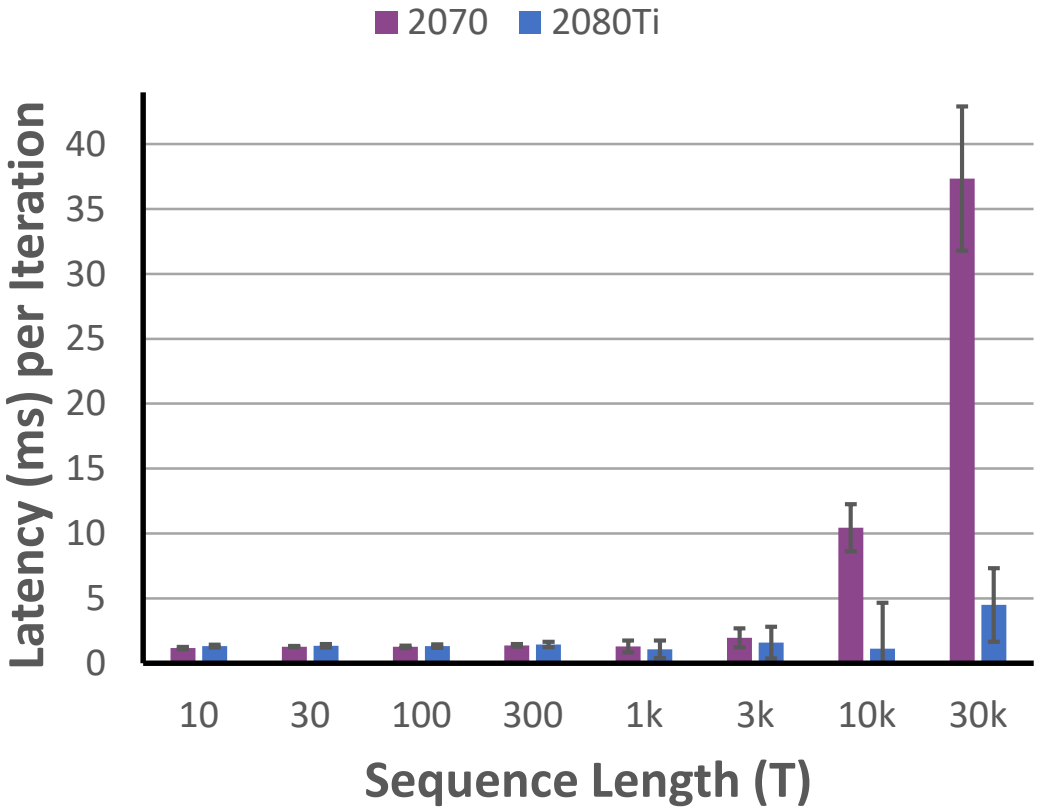


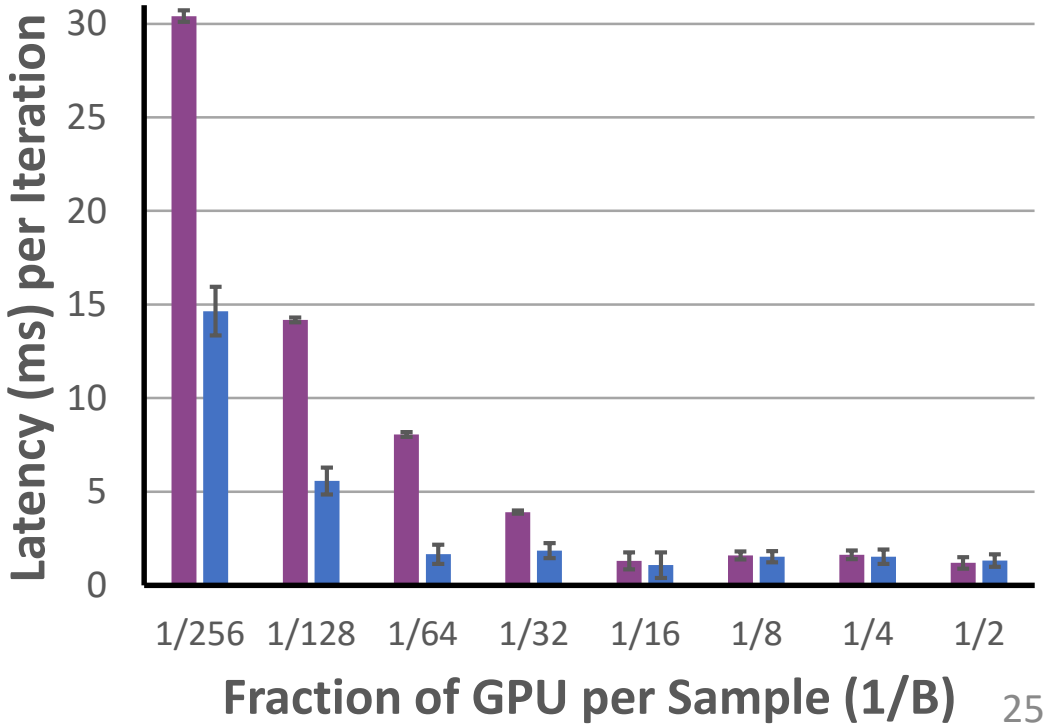
BPPSA **scales** with the number of workers (**p**).

Sensitivity Analysis: 2070 v.s. 2080Ti

#SMs(2070) < #SMs(2080Ti)
→ Latency(2070) > Latency(2080Ti)

SM: Streaming Multiprocessor;
i.e., “Parallel Cores”.





More Results in the Paper

- End-to-end benchmarks of GRU training on IRMAS.
  - A more realistic version of the RNN results.
- Pruned VGG-11 retraining on CIFAR-10.
  - Microbenchmark via FLOP measurements.
  - Evaluate the effectiveness of leveraging the Jacobians' sparsity in CNNs.

Conclusion

BP imposes a **strong sequential dependency** among layers during the gradient computations, limiting its scalability on parallel systems.

We propose scaling **B**ack-**P**ropagation by **P**arallel **S**can **A**lgorithm (**BPPSA**):

- Reformulate BP as a **scan** operation.
- Scale by a **customized Blelloch scan** algorithm.
- Leverage **sparsity** in the Jacobians.

Key Results: **Θ(log n)** vs. **Θ(n)** steps on parallel systems.

Up to **108×** speedup on the backward pass (→ **2.17×** overall speedup).