THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
RELAXED SYSTEM LAB

Nvidia GPU Computation and Communication

COMP6211J

Binhang Yuan

RELAXED
SYSTEM LAB

GPU Architecture

ANNOUNCING NVIDIA A100 PCIE

Greatest Generational Leap - 20X Volta

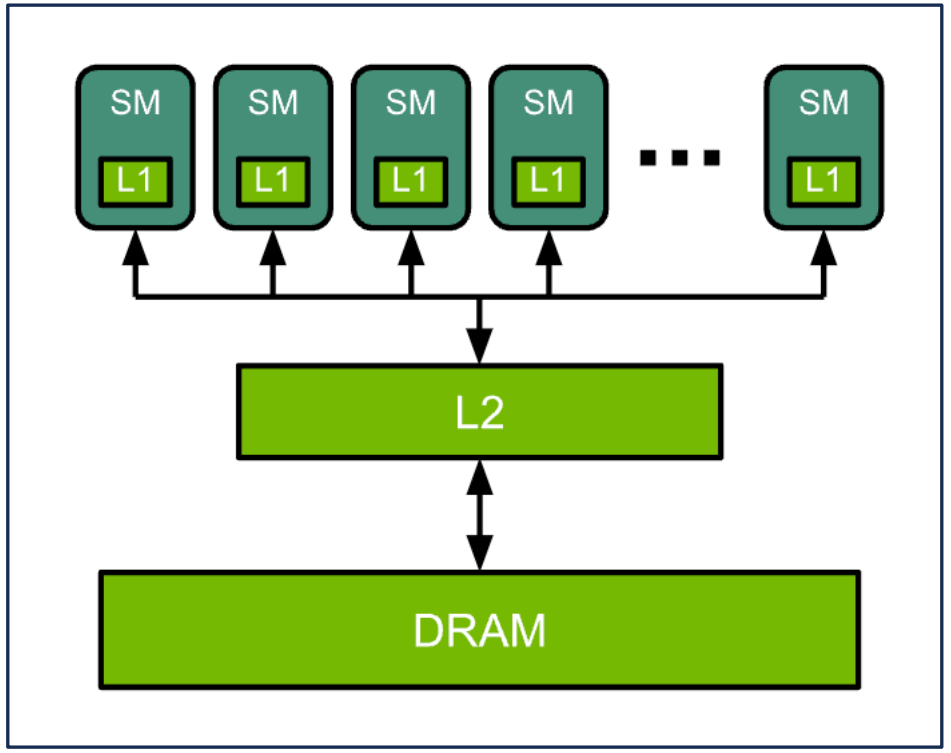
	Peak		Vs Volta
FP32 TRAINING	312	TFLOPS	20X
INT8 INFERENCE	1,248	TOPS	20X
FP64 HPC	19.5	TFLOPS	2.5X
MULTI INSTANCE GPU			7X GPUs

54B XTOR | 826mm2 | TSMC 7N | 40GB Samsung HBM2

GPU Architecture

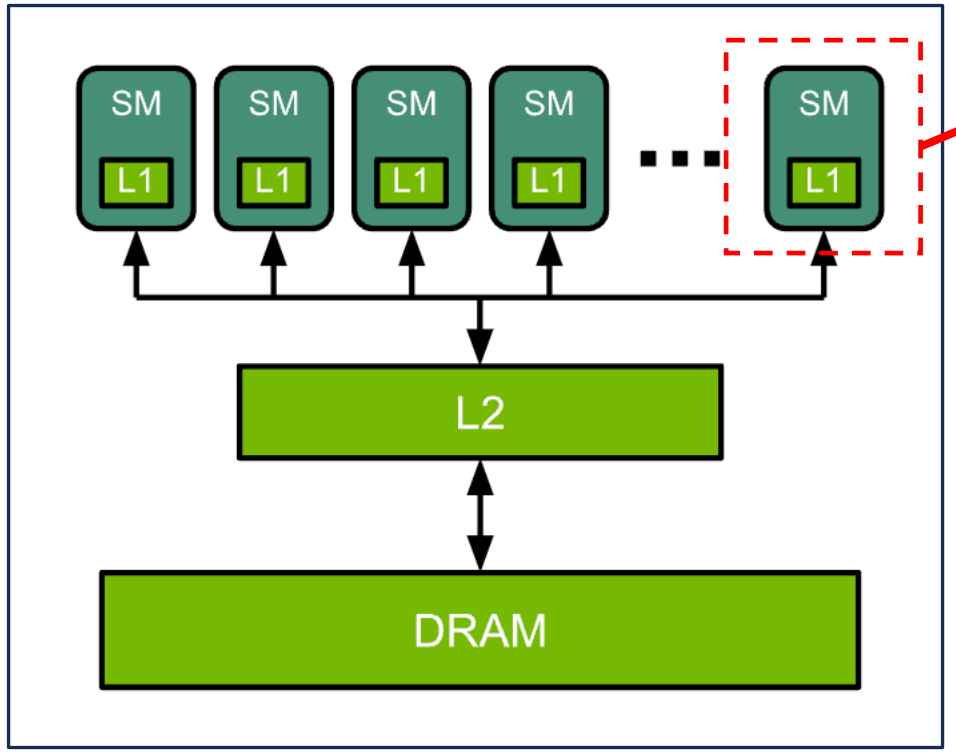
RELAXED SYSTEM LAB

- The GPU is a highly parallel processor architecture, including processing elements and a memory hierarchy.
- The memory hierarchy:
  - L0, L1 cache in Streaming Multiprocessors (SMs);
  - On-chip L2 cache;
  - High bandwidth DRAM (HBM).
- Arithmetic and other instructions are executed by the SMs.
- Data and code are accessed from DRAM via the L2 cache.



Ampere GPU Architecture

RELAXED SYSTEM LAB



108 SM in a A100 GPU

Ampere GPU SM

RELAXED SYSTEM LAB

- In Ampere GPU, SM contains **four** processing blocks that share an L1 cache for data caching.
- Each processing block has:
  - 1 Warp scheduler (where the maximum number of thread blocks per SM is 32);
  - 16 INT32 CUDA cores;
  - 16 FP32 CUDA cores;
  - 8 FP64 CUDA cores;
  - 8 Load/Store cores;
  - 1 SFU core (special function units: e.g., sin, cos)
  - 1 **Tensor core** for matrix multiplication;
  - 1 16K 32-bit register file.

A100 GPU Memory Hierarchy

RELAXED SYSTEM LAB

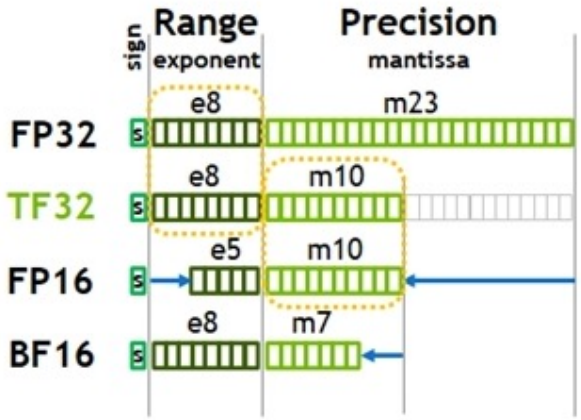
- Size:
  - L1 cache: 192 KB per SM;
  - L2 cache: 40 MB
  - HBM: 80 GB
- Accessibility:
  - The L2 cache is unified, shared by all SMs, and set aside for data and instructions.
  - The L1 instruction cache is private to a single streaming multiprocessor.
  - The L0 instruction cache is private to a single streaming multiprocessor subprocessing block.

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

A100 GPU Tensor Core Computation

RELAXED SYSTEM LAB

- Multiply-add is the most frequent operation in modern neural networks. This is known as the fused multiply-add (FMA) operation.
- Includes one multiply operation and one add operation, counted as two float operations.
- A100 GPU has 1.41 GHz clock rate.
- The Ampere A100 GPU Tensor Cores multiply-add operations per clock:



Ampere A100 GPU FMA per clock on a SM					
FP64	TF32	FP16	INT8	INT4	INT1
64	512	1024	2048	4096	16384

A100 GPU Specifications

RELAXED SYSTEM LAB

Ampere A100 GPU FMA per clock on a SM					
FP64	TF32	FP16	INT8	INT4	INT1
64	512	**1024**	2048	4096	16384

A100 GPU Specs	
Tensor core Float 32 (TF32)	156 TFLOPS
Tensor core Float 16 (FP16)	**312 TFLOPS**
Tensor core Int 8 (INT8)	624 TOPS
GPU Memory	80 GB
GPU Memory Bandwidth	2039 GB/s

$$1024 \times 2 \times 1.41 \times 10^{9} \times 108 = 312 \times 10^{12}$$

Tensor Cores

RELAXED
SYSTEM LAB

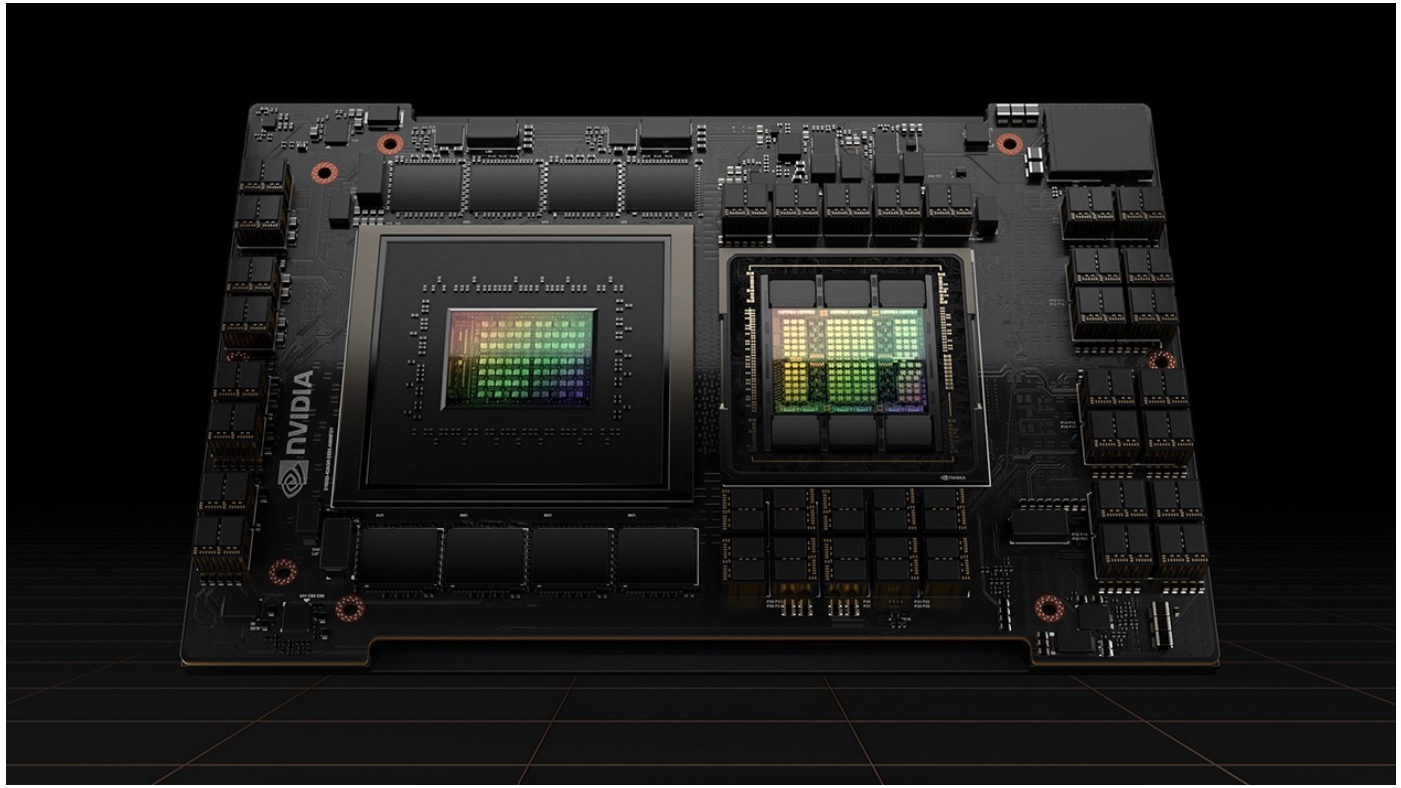
- Tensor Cores were introduced in the NVIDIA Volta™ GPU architecture to accelerate matrix multiply and accumulate operations for machine learning and scientific applications.
- These instructions operate on small matrix blocks:
  - For example, 16×16 blocks in A100 GPUs.
- Tensor Cores can compute and accumulate products with higher precision than the inputs:
  - During training with FP16 inputs, Tensor Cores can compute products without loss of precision and accumulate in FP32.

H100 GPU

RELAXED
SYSTEM LAB

SPECIFICATIONS

	H100 SXM	H100 PCIe
FP64	34 TFLOPS	26 TFLOPS
FP64 Tensor Core	67 TFLOPS	51 TFLOPS
FP32	67 TFLOPS	51 TFLOPS
TF32 Tensor Core	989 TFLOPS*	756 TFLOPS*
BFLOAT16 Tensor Core	1,979 TFLOPS*	1,513 TFLOPS*
FP16 Tensor Core	1,979 TFLOPS*	1,513 TFLOPS*
FP8 Tensor Core	3,958 TFLOPS*	3,026 TFLOPS*
INT8 Tensor Core	3,958 TOPS*	3,026 TOPS*
GPU memory	80GB	80GB
GPU memory bandwidth	3.35TB/s	2TB/s

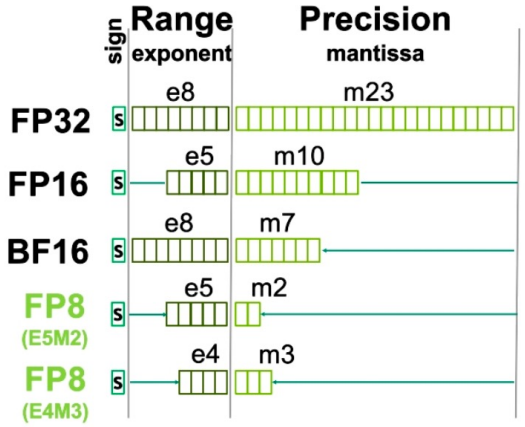


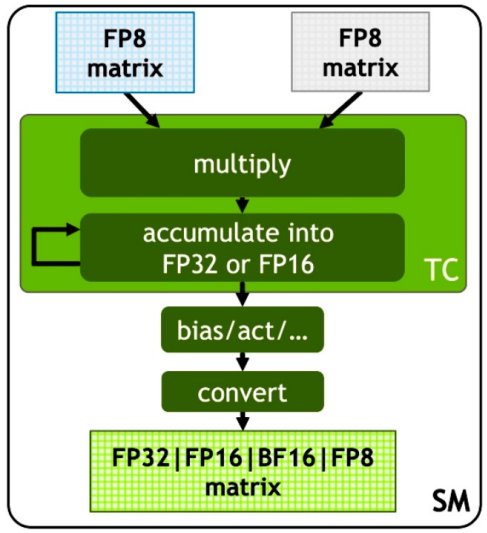
* Shown with sparsity. Specifications 1/2 lower without sparsity.

H100 GPU - Highlights

RELAXED SYSTEM LAB

- New fourth-generation Tensor Cores are up to 6x faster chip-to-chip compared to A100 (the third-generation), including per-SM speedup, additional SM count, and higher clocks of H100.
- Hopper FP8 Data Format:
  - Add FP8 Tensor Cores to accelerate both AI training and inference;
  - Two new FP8 inputs types:
    - E4M3 with 4 exponent bits, 3 mantissa bits, and 1 sign bit
    - E5M2 with 5 exponent bits, 2 mantissa bits, and 1 sign bit.
  - Support multiple accumulator and output types.





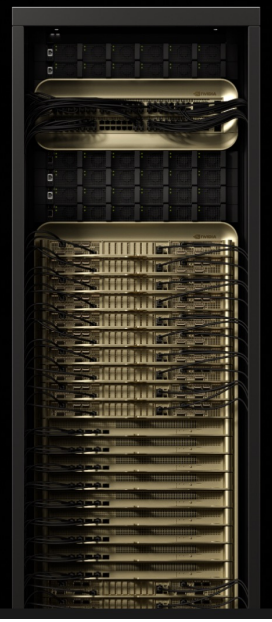
GB200 in 2025

RELAXED
SYSTEM LAB

NVIDIA GB200 NVL72

Powering the new era of computing.

Read Datasheet



Technical Specifications[1]		
	GB200 NVL72	**HGX B200**
Blackwell GPUs | Grace CPUs	72 | 36	8 | 0
CPU Cores	2,592 Arm Neoverse V2 Cores	-
Total FP4 Tensor Core	1,440 PFLOPS	144 PFLOPS
Total FP8/FP6 Tensor Core	720 PFLOPS	72 PFLOPS
Total Fast Memory	Up to 30TB	Up to 1.4TB
Total Memory Bandwidth	Up to 576TB/s	Up to 62TB/s
Total NVLink Bandwidth	130TB/s	14.4TB/s
	Individual Blackwell GPU Specifications	
FP4 Tensor Core	20 PFLOPS	18 PFLOPS
FP8/FP6 Tensor Core	10 PFLOPS	9 PFLOPS
INT8 Tensor Core	10 POPS	9 POPS
FP16/BF16 Tensor Core	5 PFLOPS	4.5 PFLOPS
TF32 Tensor Core	2.5 PFLOPS	2.2 PFLOPS
FP32	80 TFLOPS	75 TFLOPS
FP64/FP64 Tensor Core	40 TFLOPS	37 TFLOPS
GPU Memory | Bandwidth	186GB HBM3E | 8TB/s	180GB HBM3E | 7.7TB/s
Multi-Instance GPU (MIG)	7	
Decompression Engine	Yes	
Decoders	7 NVDEC[2] 7 nvJPEG	
Max Thermal Design Power (TDP)	Configurable up to 1,200W	Configurable up to 1,000W
Interconnect	5th Generation NVLink: 1.8TB/s PCIe Gen5: 128GB/s	
Server Options	NVIDIA GB200 NVL72 partner and NVIDIA-Certified Systems™ with 72 GPUs	NVIDIA HGX B200 partner and NVIDIA-Certified Systems with 8 GPUs

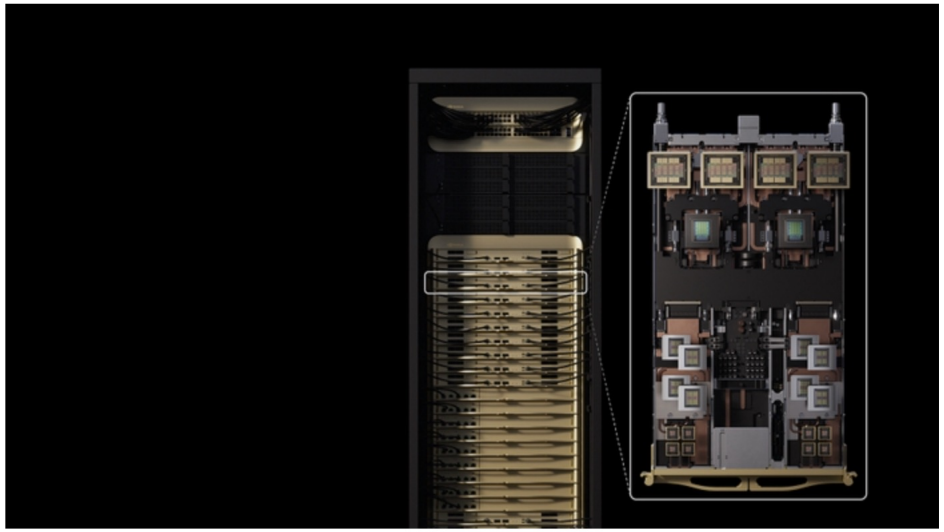
1. All Tensor Core numbers except FP64 with sparsity.
2. Supported formats provide these speed-ups over H100 Tensor Core GPUs: 2X H.264, 1.25X HEVC, 1.25X VP9. AV1 support is new to Blackwell GPUs

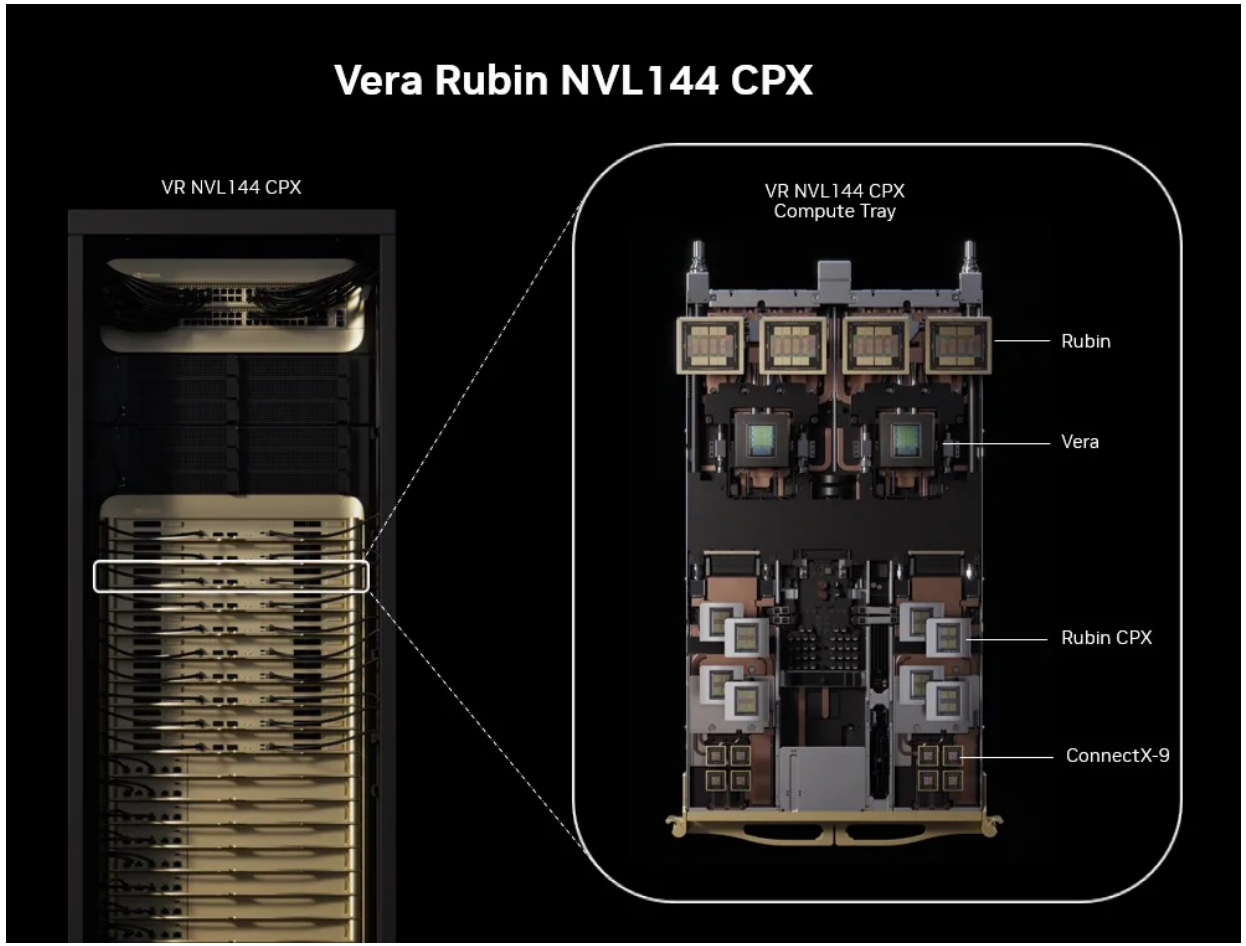
Yesterday ...

RELAXED
SYSTEM LAB

NVIDIA Unveils Rubin CPX: A New Class of GPU Designed for Massive-Context Inference

September 9, 2025





GPU Execution Model

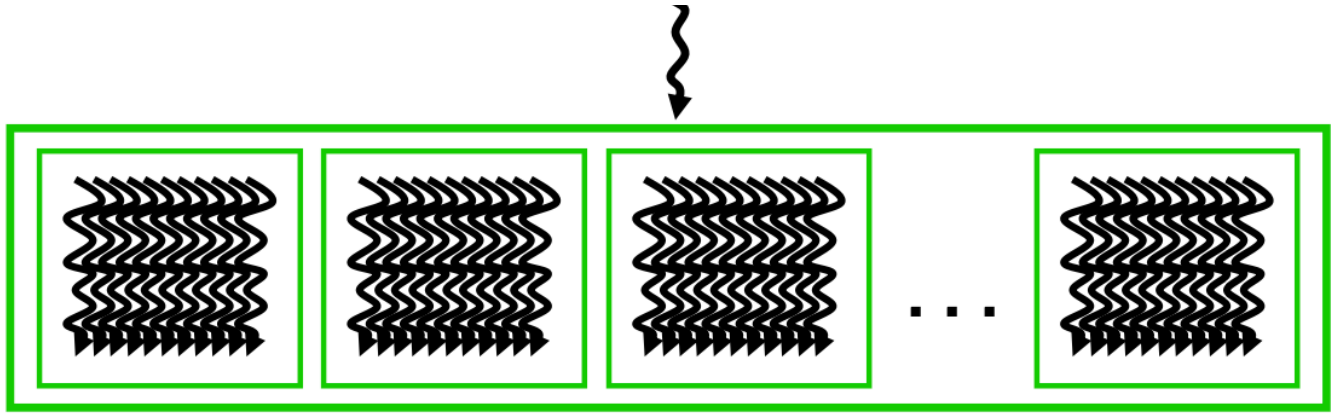
3-level Thread Hierarchy

RELAXED SYSTEM LAB

- GPUs execute functions using a 3-level hierarchy of threads.
  - Each invocation of a CUDA kernel creates a new **grid**, which consists of multiple **blocks**. Each block consists of up to 1024 individual **threads**.
- GPUs hide dependent instruction latency by switching to the execution of other threads.
  - The number of threads needed to utilize a GPU effectively is much higher than the number of cores or instruction pipelines.

Host: launch the function

Device: Parallel Kernel



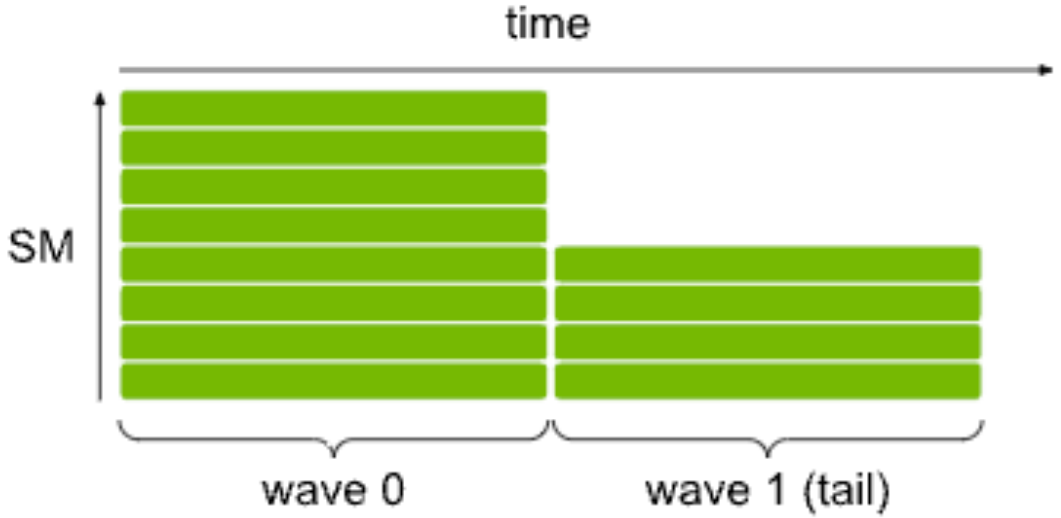
3-level Thread Hierarchy

- GPUs have many SMs, each of which has pipelines for executing many threads and enables its threads to communicate via shared memory and synchronization.
- At runtime, a thread block is placed on an SM for execution, enabling all threads in a thread block to communicate and synchronize efficiently.
- Launching a function with a single thread block would only give work to a single SM; to fully utilize a GPU with multiple SMs, one needs to launch many thread blocks.
- Since an SM can execute multiple thread blocks concurrently, typically, one wants the number of thread blocks to be several times higher than the number of SMs.

3-level Thread Hierarchy

RELAXED
SYSTEM LAB

- Minimize the "tail" effect: at the end of a function execution, only a few active thread blocks remain.
- We use the term wave to refer to a set of thread blocks that run concurrently.
- It is most efficient to launch functions that execute in several waves of thread blocks - a smaller percentage of time is spent in the tail wave, minimizing the tail effect and thus the need to do anything about it.
- For the higher-end GPUs, typically only launches with fewer than 300 thread blocks should be examined for tail effects.



Utilization of an 8-SM GPU when 12 thread blocks with an occupancy of 1 block/SM at a time are launched for execution. The blocks execute in 2 waves, the first wave utilizes 100% of the GPU, while the 2nd wave utilizes only 50%.

Understanding Performance

Overview

RELAXED
SYSTEM LAB

- The performance of a function on a given processor is limited by one of the following three factors:
  - Memory bandwidth;
  - Math bandwidth;
  - Latency.
- Consider a simplified model where a function:
  - Read its input from memory;
  - Perform math operations;
  - Write its output to memory.

Modeling the Cost

RELAXED
SYSTEM LAB

- T_{mem} time is spent in accessing memory;
- T_{math} time is spent performing math operations.
- If we further assume that memory and math portions of different threads can be overlapped;
- The total time for the function is $\max(T_{mem}, T_{math})$.
- The longer of the two times demonstrates what limits performance:
  - If math time is longer, we say that a function is *math limited*;
  - If memory time is longer then it is *memory limited*.

Arithmetic Intensity

- How much time is spent in memory or math operations depends on both the algorithm and its implementation, as well as the processor's bandwidths.
- Memory time is equal to the number of bytes accessed in memory divided by the processor's memory bandwidth.
- Math time is equal to the number of operations divided by the processor's math bandwidth.

Arithmetic Intensity

RELAXED
SYSTEM LAB

- Thus, on a given processor a given algorithm is math limited if:
  - $T_{math} > T_{mem}$
  - $\frac{\#op}{BW_{math}} > \frac{\#bytes}{BW_{mem}}$

- By simple algebra, the inequality can be rearranged to:
  - $\frac{\#op}{\#bytes} > \frac{BW_{math}}{BW_{mem}}$

- The left-hand side: the algorithm's *arithmetic intensity*.
- The right-hand side: *ops:byte ratio*.

Arithmetic Intensity

RELAXED
SYSTEM LAB

- Arithmetic intensity: the ratio of algorithm implementation operations and the number of bytes accessed.
- Ops:byte ratio: the ratio of a processor's math and memory bandwidths.
- Thus, an algorithm is math limited on a given processor if the algorithm's arithmetic intensity is higher than the processor's ops:byte ratio.
- Conversely, an algorithm is memory limited if its arithmetic intensity is lower than the processor's ops:byte ratio.

Arithmetic Intensity

RELAXED SYSTEM LAB

- Compare the algorithm's arithmetic intensity to the ops:byte ratio on an NVIDIA Volta V100 GPU.
  - V100 has a peak math rate of 125 FP16 Tensor TFLOPS;
  - An off-chip memory bandwidth of approx. 900 GB/s
  - An on-chip L2 bandwidth of 3.1 TB/s;
- So it has a ops:byte ratio between 40 and 139, depending on the source of an operation's data (on-chip or off-chip memory).

Operation	Arithmetic Intensity	limited by
Linear layer (4096 outputs, 1024 inputs, batch size 512)	315 FLOPS/B	arithmetic
Linear layer (4096 outputs, 1024 inputs, batch size 1)	1 FLOPS/B	memory
Max pooling with 3x3 window and unit stride	2.25 FLOPS/B	memory
ReLU activation	0.25 FLOPS/B	memory
Layer normalization	10 FLOPS/B	memory

Arithmetic Intensity

RELAXED
SYSTEM LAB

- Note that this type of analysis is a simplification, as we're counting only the algorithmic operations used.
- In practice, functions also contain instructions for operations not explicitly expressed in the algorithm, such as:
  - Memory access instructions;
  - Address calculation instructions;
  - Control flow instructions, and so on.

Limited by Lantency

- The arithmetic intensity and ops:byte ratio analysis assumes that a workload is sufficiently large to saturate a given processor's math and memory pipelines.
- However, if the workload is not large enough, or does not have sufficient parallelism, the processor will be under-utilized and performance will be limited by latency.
- For example:
  - Consider the launch of a single thread that will access 16 bytes and perform 16000 math operations.
  - While the arithmetic intensity is 1000 FLOPS/B and the execution should be math-limited on a V100 GPU, creating only a single thread grossly under-utilizes the GPU, leaving nearly all of its math pipelines and execution resources idle.

DNN Operation Categories

Elementwise Operations

RELAXED
SYSTEM LAB

- Elementwise operations may be unary or binary operations;
- The key is that layers in this category perform mathematical operations on each element independently of all other elements in the tensor.
- For example:
  - A ReLU layer returns max(0, x) for each x in the input tensor.
  - The element-wise addition of two tensors computes each output sum value independently of other sums.
- Layers in this category include most non-linearities (sigmoid, tanh, etc.), scale, bias, add, and others.
- These layers tend to be ***memory-limited***, as they perform few operations per byte accessed.

Reduction Operations

RELAXED
SYSTEM LAB

- Reduction operations produce values computed over a range of input tensor values.
- For example:
  - Pooling layers compute values over some neighbourhoods in the input tensor.
  - Batch normalization computes the mean and standard deviation over a tensor before using them in operations for each output element.
  - SoftMax also falls into the reduction category.
- Typical reduction operations have a low arithmetic intensity and thus ***are memory limited***.

Dot-Product Operations

RELAXED
SYSTEM LAB

- Operations in this category can be expressed as dot-products of elements from two tensors, usually a weight (learned parameter) tensor and an activation tensor.
- Examples:
  - Fully-connected layers are naturally expressed as matrix-vector and matrix-matrix multiplies.
  - Convolutions can also be expressed as collections of dot-products - one vector is the set of parameters for a given filter, and the other is an "unrolled" activation region to which that filter is being applied.
- Operations in the dot-product category can be math-limited if the corresponding matrices are large enough.
- However, for the smaller sizes, these operations end up being memory-limited. For example, a fully-connected layer applied to a single vector (a tensor for a mini-batch of size 1)) is memory limited.

Dot-Product Operations: Matrix Multiplication

RELAXED SYSTEM LAB

- Compute $C = AB$ suppose:
  - A is an $M \times K$ matix; (M rows and K columns)
  - B is an $K \times N$ matix;
  - C is an $M \times N$ matix;
- A total of $M \times N \times K$ fused multiply-adds (FMAs) are needed to compute the product. so a total of $2 \times M \times N \times K$ flops are required.
- The total number of byte scan in FP16: $2(M \times K + K \times N + M \times N)$
- Arithmetic intensity= $\frac{M \times N \times K}{(M \times K + K \times N + M \times N)}$.

RELAXED
SYSTEM LAB

NCCL

Collective Communication

RELAXED
SYSTEM LAB

- **Process group**: all communication is within a group of processes, and the collective communication is over all of the processes in that group.
- **World**: defines all of the processes for the parallel job.
- **World size**: number of processes in the world.
- **Rank**: the unique process ID in the world.

NCCL Overview

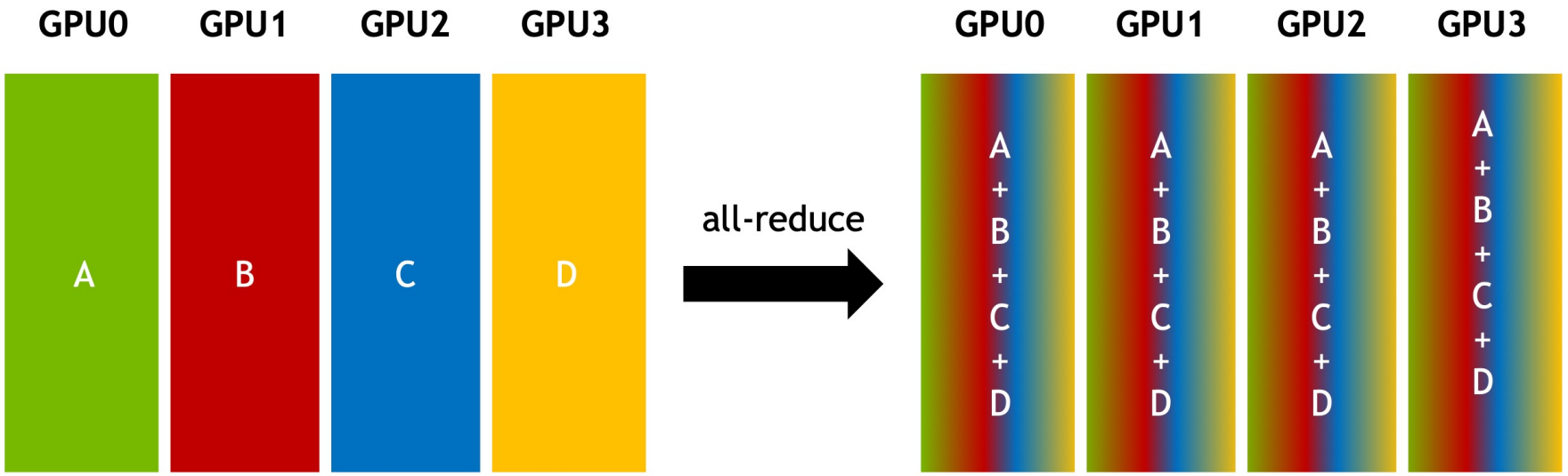
RELAXED SYSTEM LAB

- Optimized collective communication library from Nvidia to enable high-performance communication between CUDA devices.
- NCCL Implements:
  - **AllReduce**;
  - **Broadcast;**
  - **Reduce;**
  - **AllGather;**
  - **Scatter;**
  - **Gather;**
  - **ReduceScatter;**
  - **AlltoAll.**
- Easy to integrate into DL framework (e.g., PyTorch).

AllReduce

RELAXED SYSTEM LAB

- The **AllReduce** operation performs reductions on data (for example, sum, min, max) across devices and writes the result in the receive buffers of **every** rank.



AllReduce in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.all_reduce(*tensor*, *op=<RedOpType.SUM: 0>*, *group=None*, *async_op=False*) [SOURCE]

Reduces the tensor data across all machines in a way that all get the final result.

After the call `tensor` is going to be bitwise identical in all processes.

Complex tensors are supported.

Parameters

- **tensor** (*Tensor*) – Input and output of the collective. The function operates in-place.
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op

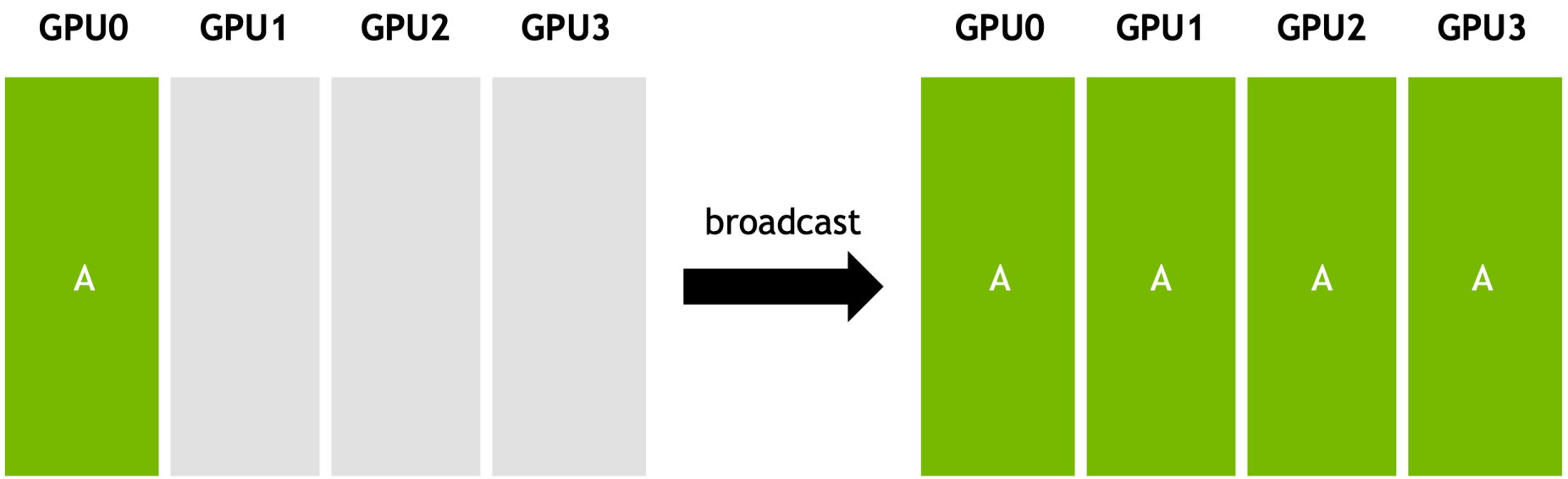
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

Broadcast

RELAXED
SYSTEM LAB

- The **Broadcast** operation copies an *N*-element buffer on the root rank to all ranks.



Broadcast in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.broadcast(*tensor*, *src*, *group=None*, *async_op=False*) [SOURCE]

Broadcasts the tensor to the whole group.

`tensor` must have the same number of elements in all processes participating in the collective.

Parameters

- **tensor** (*Tensor*) – Data to be sent if `src` is the rank of current process, and tensor to be used to save received data otherwise.
- **src** (*int*) – Source rank.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op

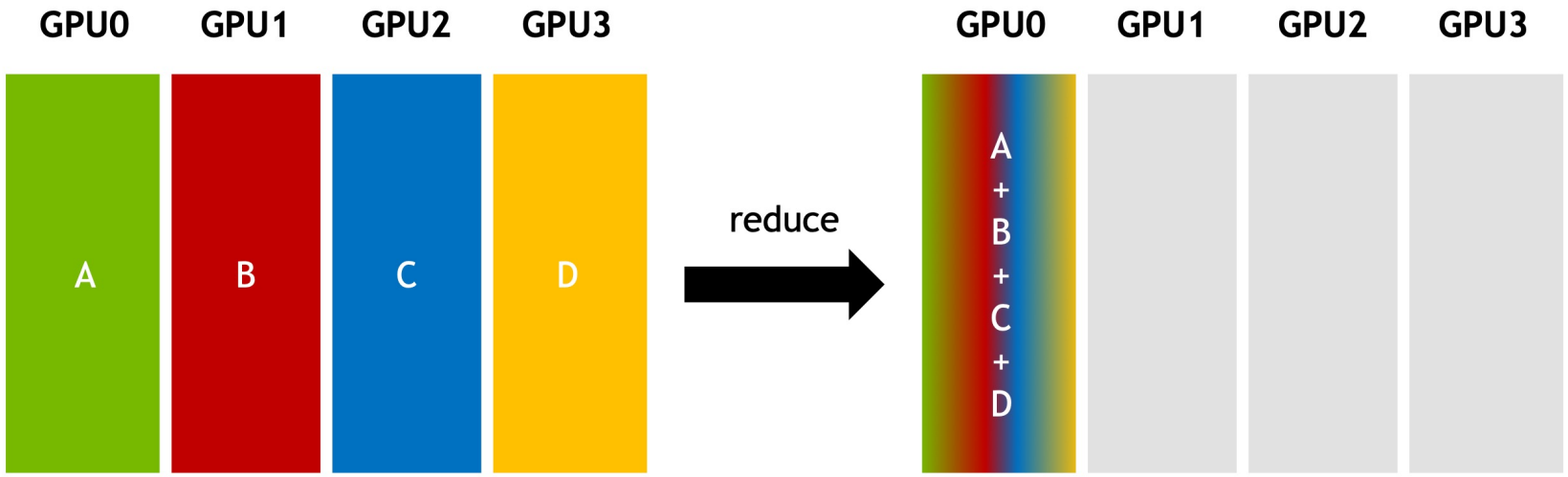
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

Reduce

RELAXED SYSTEM LAB

- The **Reduce** operation is performing the same operation as **AllReduce**, but writes the result only in the receive buffers of a specified root rank.



Reduce in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.reduce(*tensor*, *dst*, *op=<RedOpType.SUM: 0>*, *group=None*, *async_op=False*) [SOURCE]

Reduces the tensor data across all machines.

Only the process with rank `dst` is going to receive the final result.

Parameters

- **tensor** (*Tensor*) – Input and output of the collective. The function operates in-place.
- **dst** (*int*) – Destination rank
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op

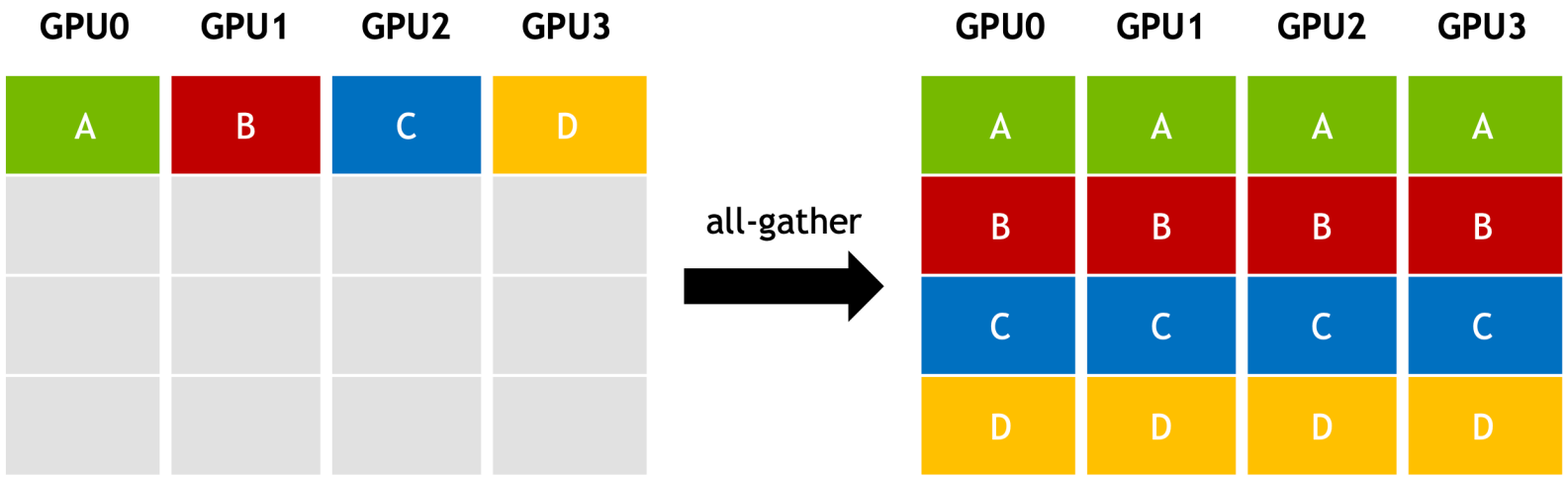
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

AllGather

RELAXED SYSTEM LAB

- The **AllGather** operation gathers N values from k ranks into an output of size kN, and distributes that result to all ranks.
- The output is ordered by rank index. The **AllGather** operation is therefore impacted by a different rank or device mapping.



AllGather in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.all_gather(*tensor_list*, *tensor*, *group=None*, *async_op=False*) [SOURCE]

Gathers tensors from the whole group in a list.

Complex tensors are supported.

Parameters

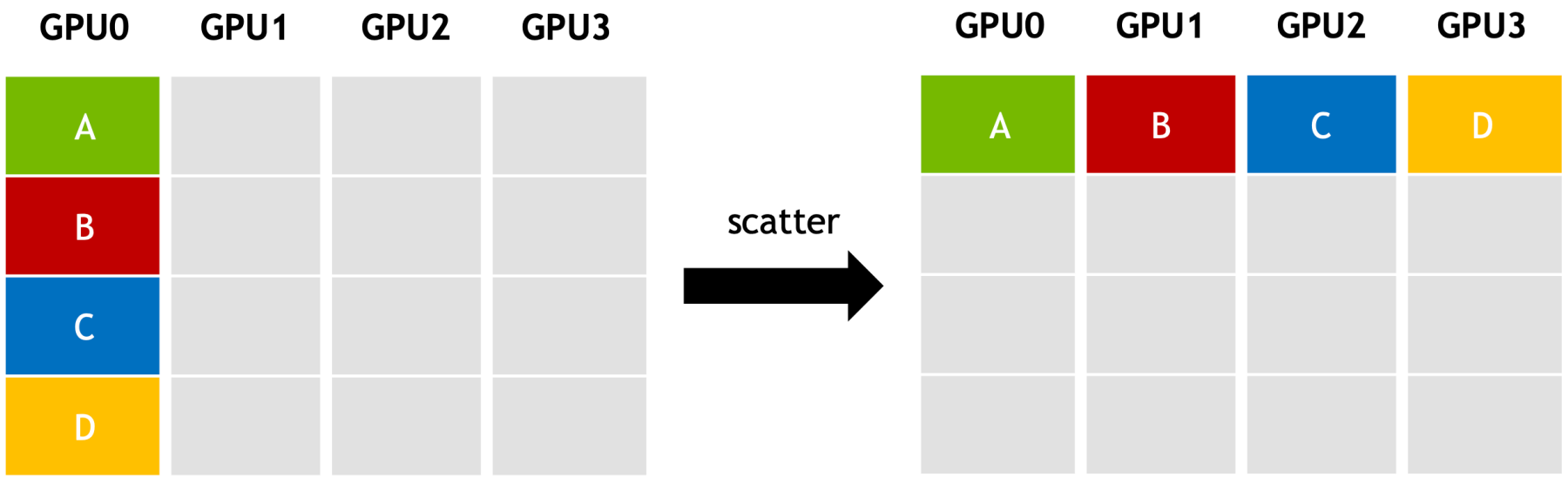
- **tensor_list** (*list*[*Tensor*]) – Output list. It should contain correctly-sized tensors to be used for output of the collective.
- **tensor** (*Tensor*) – Tensor to be broadcast from current process.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

Scatter

- **Scatter** is a collective routine similar to **Broadcast**.
- **Scatter** involves a designated root process sending data to all processes.
- **Broadcast** sends the same piece of data to all processes while **Scatter** sends chunks of an array to different processes.



Scatter in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.scatter(*tensor*, *scatter_list=None*, *src=0*, *group=None*, *async_op=False*) [SOURCE]

Scatters a list of tensors to all processes in a group.

Each process will receive exactly one tensor and store its data in the `tensor` argument.

Complex tensors are supported.

Parameters

- **tensor** (*Tensor*) – Output tensor.
- **scatter_list** (*list*[*Tensor*]) – List of tensors to scatter (default is None, must be specified on the source rank)
- **src** (*int*) – Source rank (default is 0)
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op

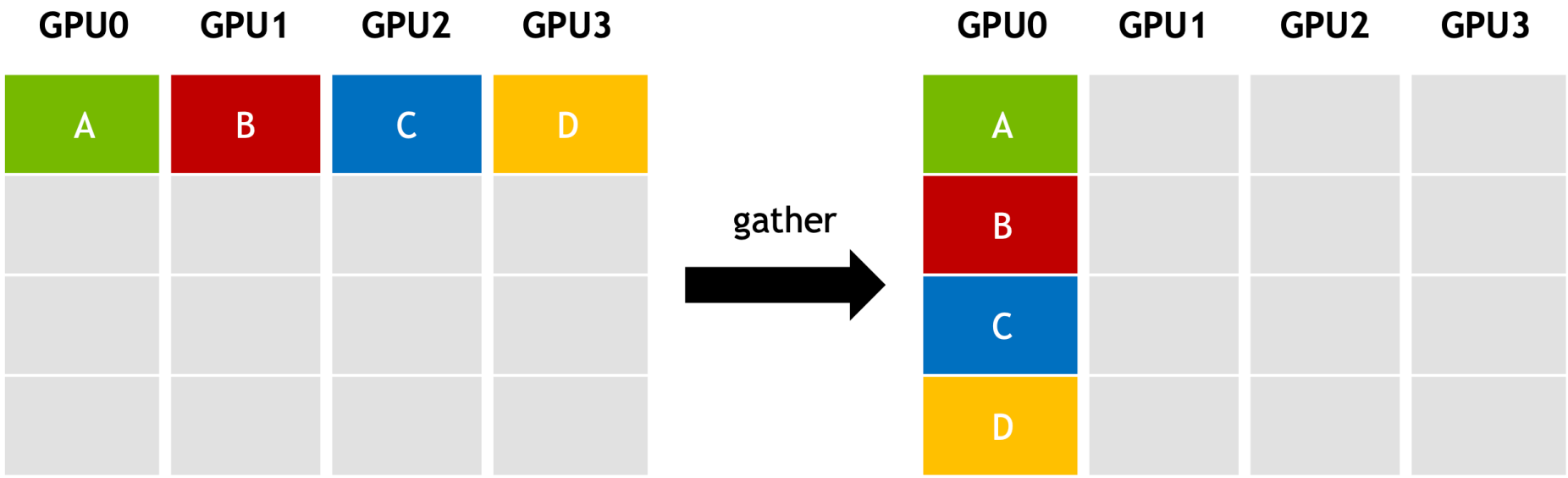
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

Gather

RELAXED
SYSTEM LAB

- Instead of spreading elements from one process to many processes, **Gather** takes elements from many processes and gathers them to one single process.



Gather in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.gather(*tensor*, *gather_list=None*, *dst=0*, *group=None*, *async_op=False*) [SOURCE]

Gathers a list of tensors in a single process.

Parameters

- **tensor** (*Tensor*) – Input tensor.
- **gather_list** (*list*[*Tensor*], *optional*) – List of appropriately-sized tensors to use for gathered data (default is None, must be specified on the destination rank)
- **dst** (*int*, *optional*) – Destination rank (default is 0)
- **group** (*ProcessGroup*, *optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool*, *optional*) – Whether this op should be an async op

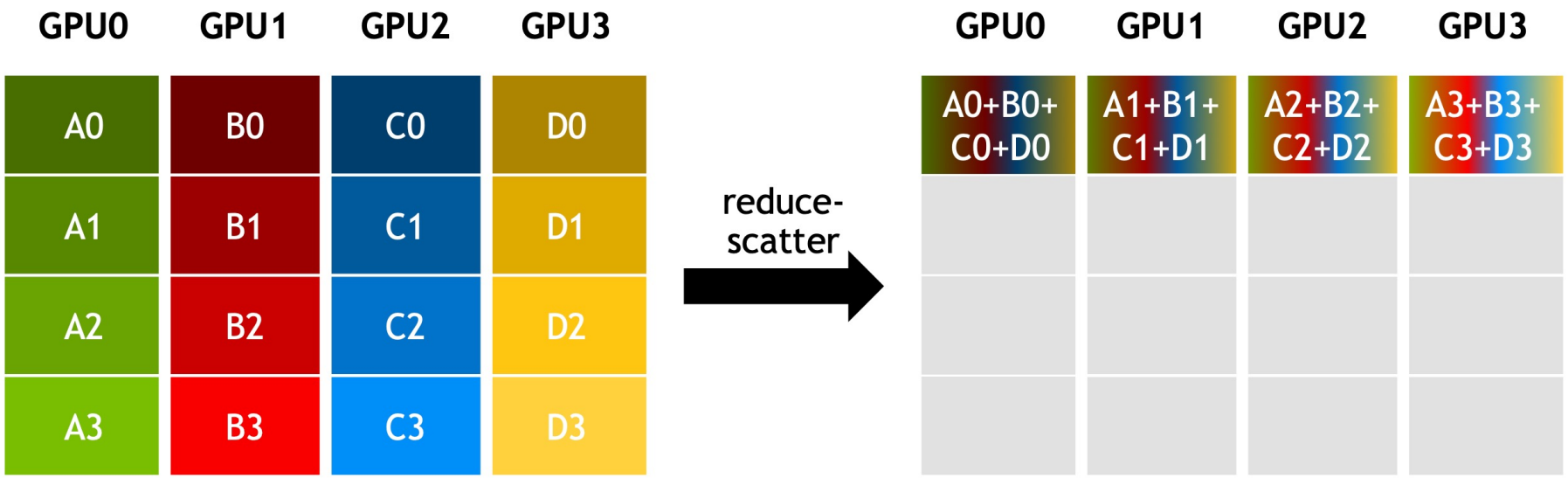
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

ReduceScatter

RELAXED SYSTEM LAB

- The **ReduceScatter** operation performs the same operation as the Reduce operation, except the result is scattered in equal blocks among ranks, each rank getting a chunk of data based on its rank index.



ReduceScatter

RELAXED
SYSTEM LAB

torch.distributed.reduce_scatter(*output*, *input_list*, *op=<RedOpType.SUM: 0>*, *group=None*, *async_op=False*) [SOURCE]

Reduces, then scatters a list of tensors to all processes in a group.

Parameters

- **output** (*Tensor*) – Output tensor.
- **input_list** (*list*[*Tensor*]) – List of tensors to reduce and scatter.
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op.

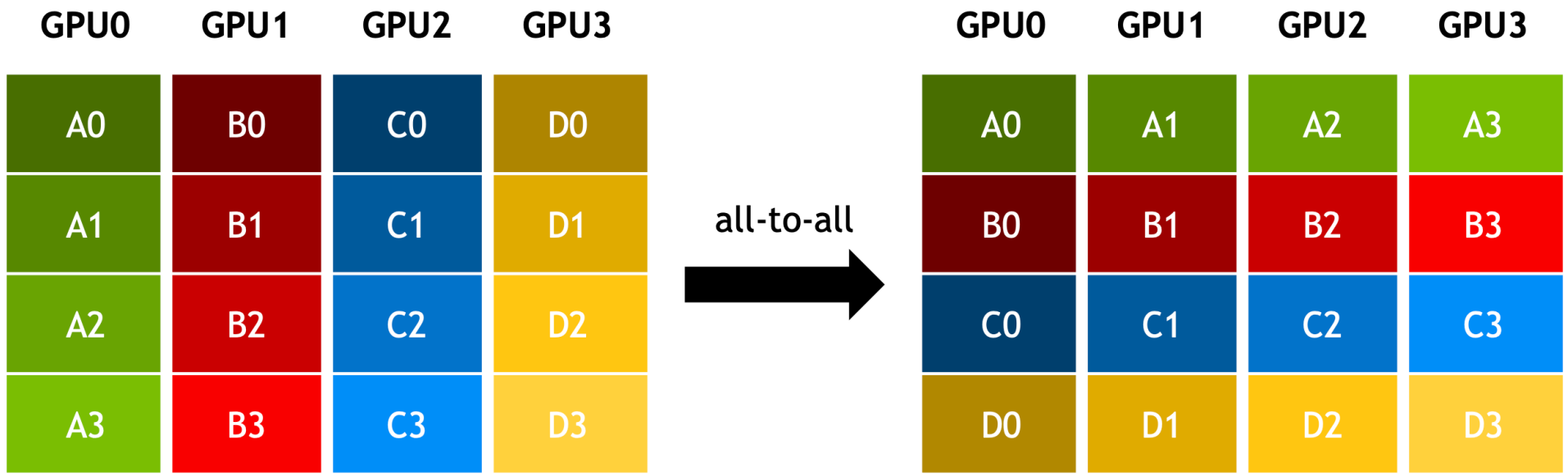
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group.

AlltoAll

RELAXED
SYSTEM LAB

- Scatter/Gather distinct messages from each participant to every other.



AlltoAll in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.all_to_all(*output_tensor_list*, *input_tensor_list*, *group=None*, *async_op=False*) [SOURCE]

Scatters list of input tensors to all processes in a group and return gathered list of tensors in output list.

Complex tensors are supported.

Parameters

- **output_tensor_list** (*list*[*Tensor*]) – List of tensors to be gathered one per rank.
- **input_tensor_list** (*list*[*Tensor*]) – List of tensors to scatter one per rank.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op.

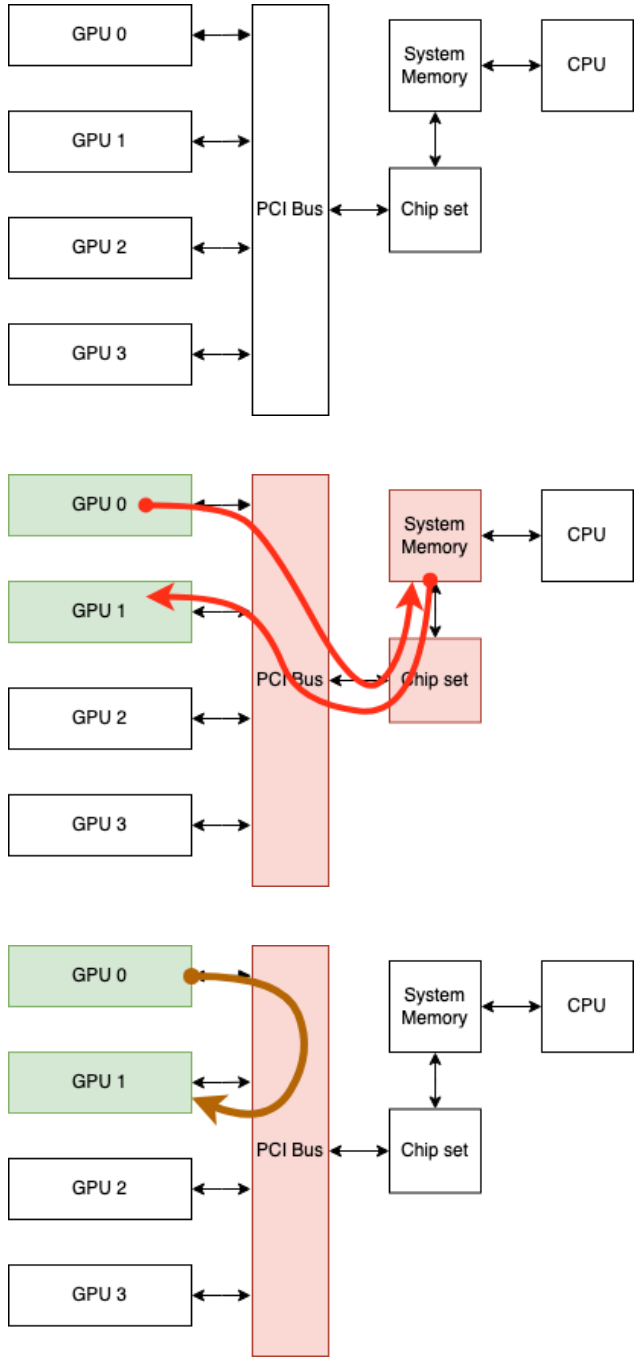
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group.

NVLink and NVSwitch

Base System PCIe

- GPUs connected to a motherboard via the PCIe;
- If there is **no peer access**, the data would first be copied from GPU 0 to the system memory, and then copied from the system memory to GPU 1.
- If the system provides **peer access** over PCIe, then the data would only be copied once. Note that the data still flows over a relatively slow PCIe interface.

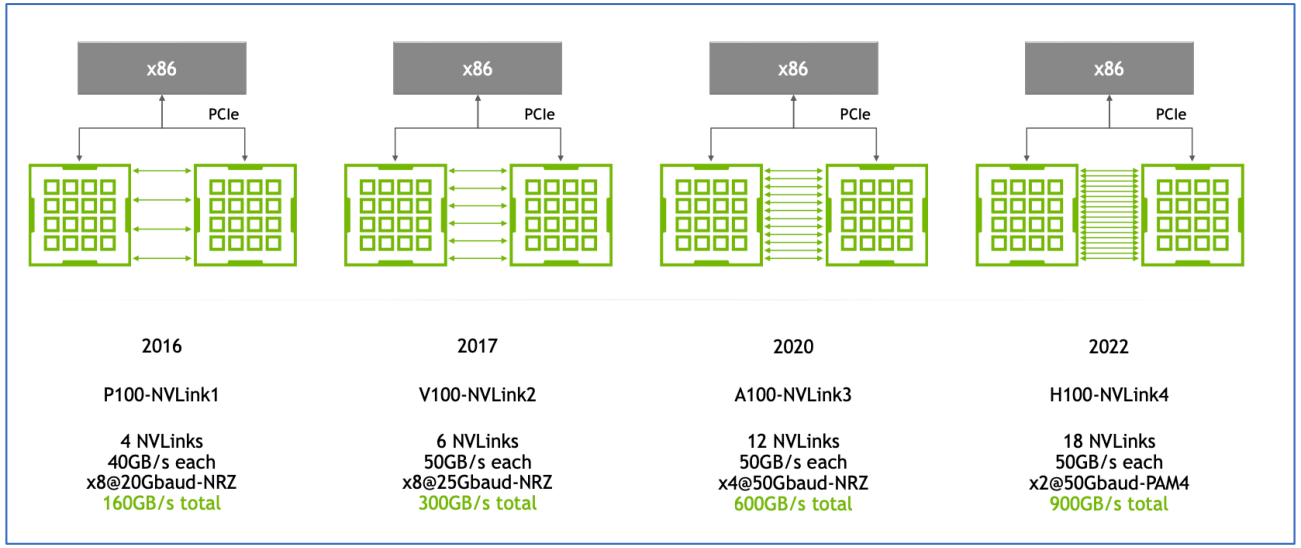


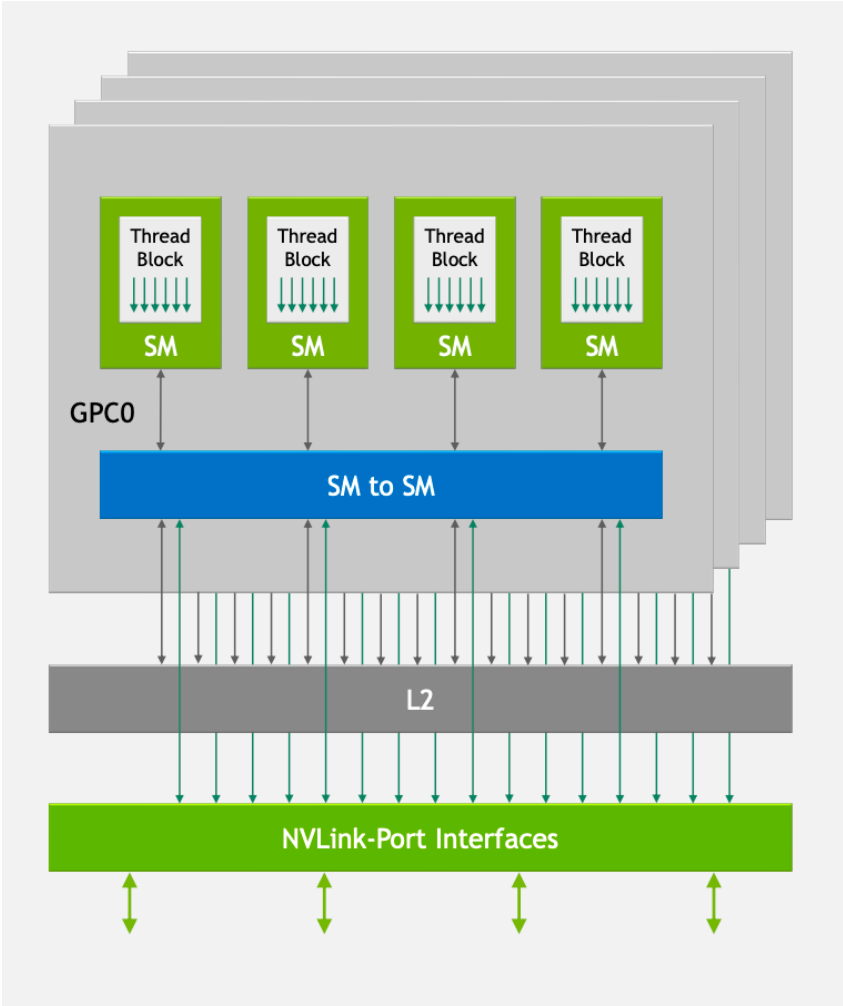
RELAXED SYSTEM LAB

NVlink

RELAXED SYSTEM LAB

- Much faster connection:
  - 100 Gbps-per-lane (NVLink4) vs 32Gbps-per-lane (PCIe Gen5);
  - Multiple NVLinks can be "ganged" to realize higher aggregate lane counts.

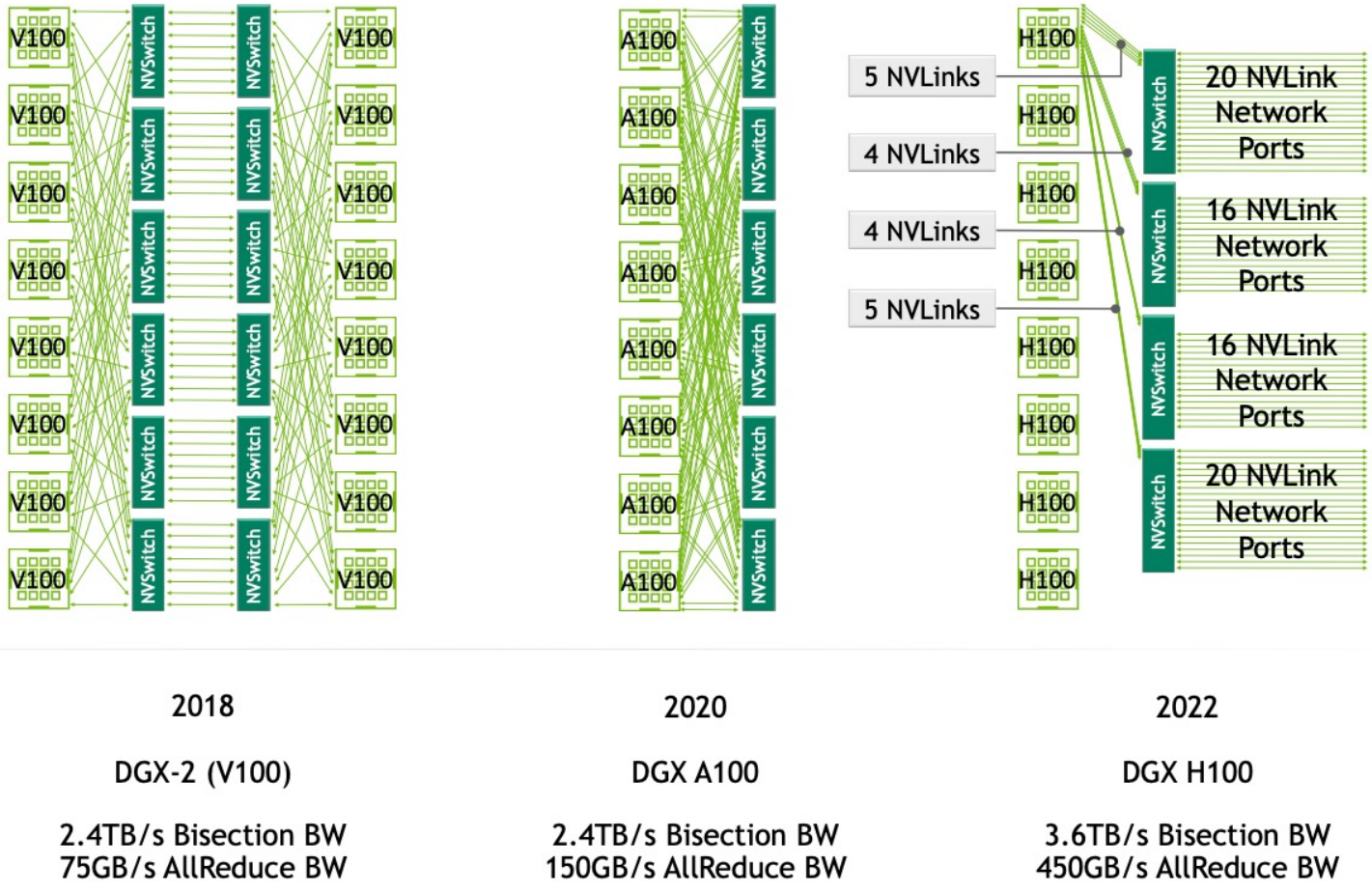




NVSwitch

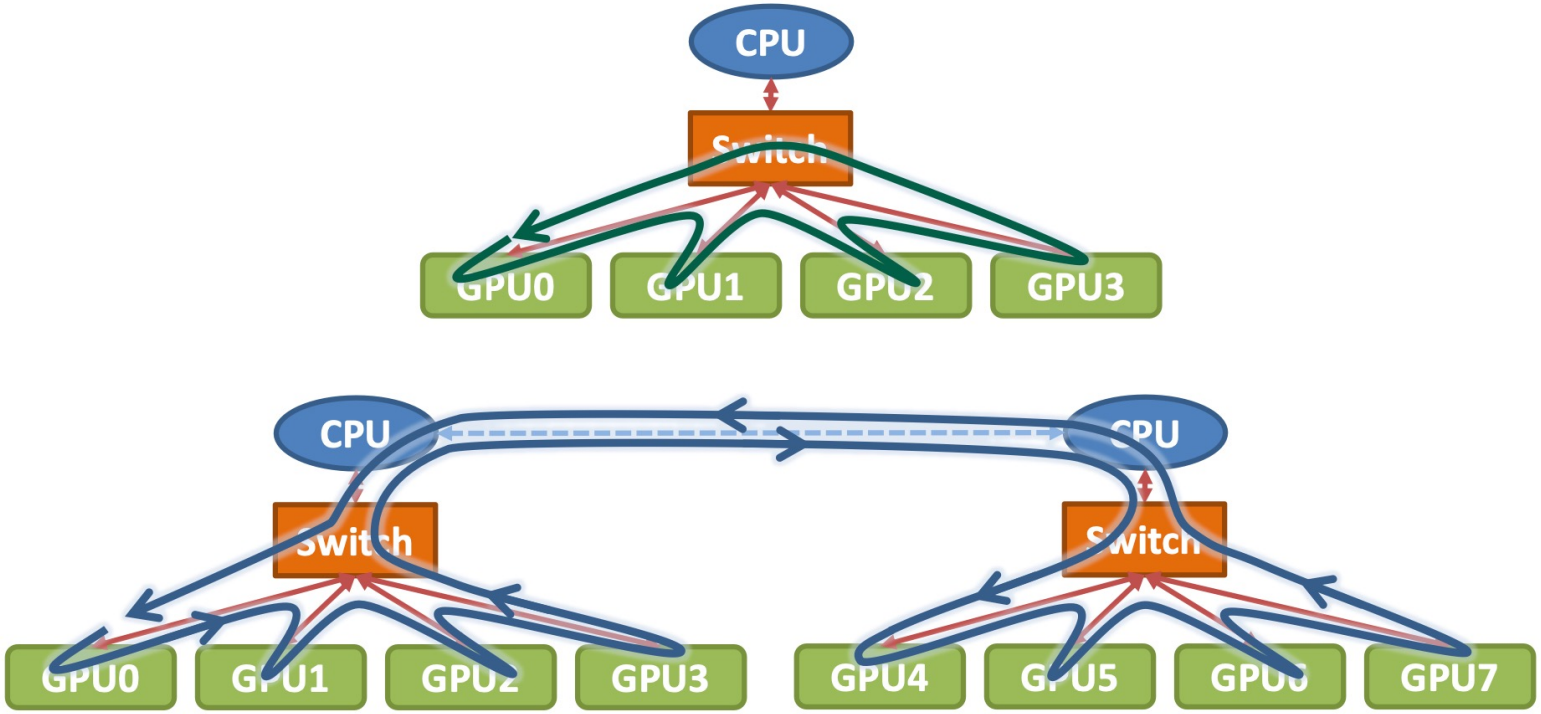
RELAXED SYSTEM LAB

- NVSwitch is an NVLink switch chip;
- NVSwitch facilitate s seamless, high-bandwidth communication between multiple GPUs by interconnecting GPUs through NVLink interfaces.



NCCL Implementation & Optimization

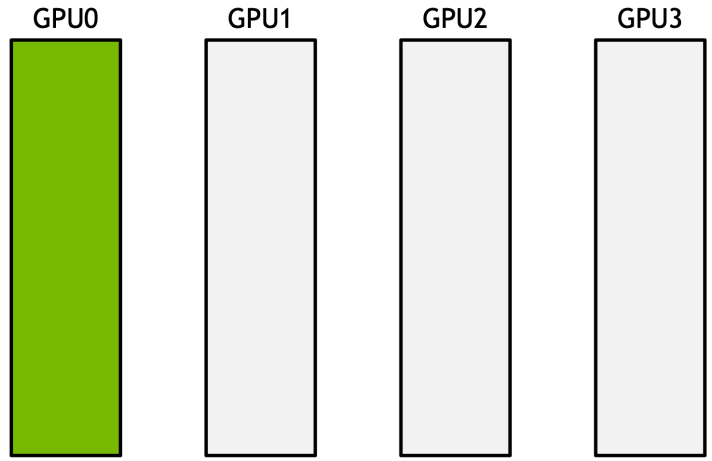
NCCL Implementation



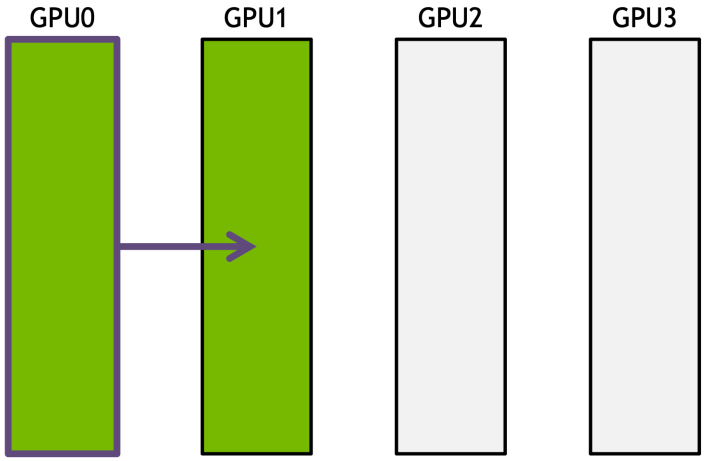
NCCL uses ***rings*** to move data across all GPUs and perform reductions.

Naïve Implementation of Broadcast

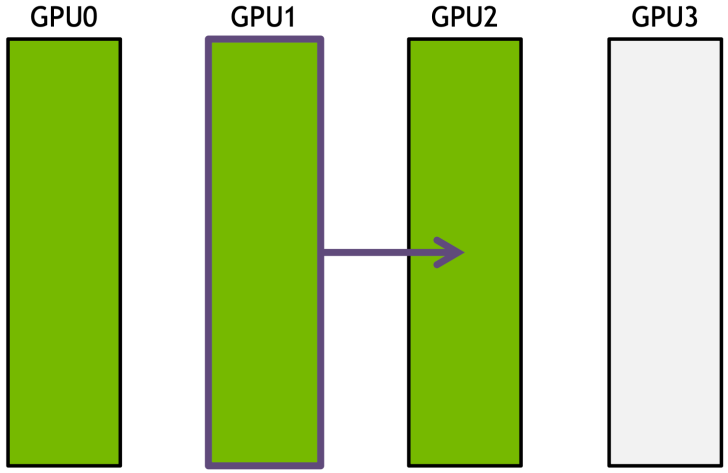
RELAXED SYSTEM LAB



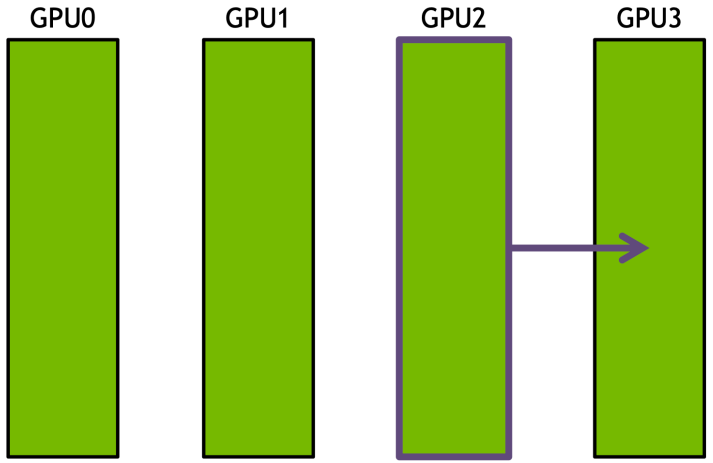
Step 0



Step 1



Step 2

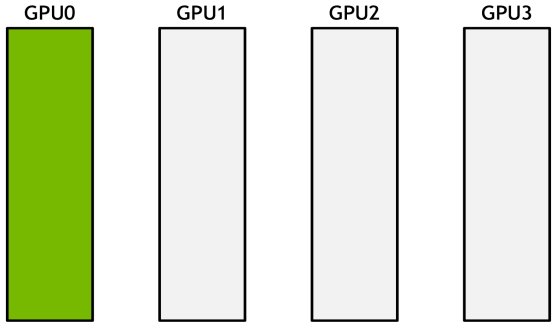


Step 3

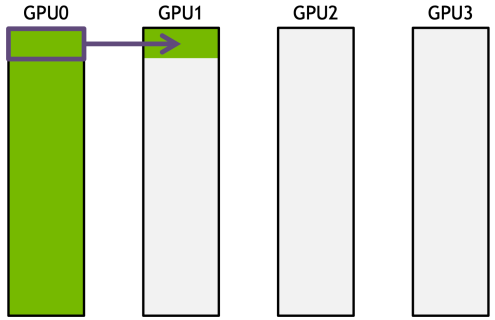
- N: bytes to broadcast
- B: bandwidth of each link
- k: number of GPUs
- Total time: $\mathrm{T} = \frac{(k-1)N}{B}$.

Optimized Implementation of Broadcast

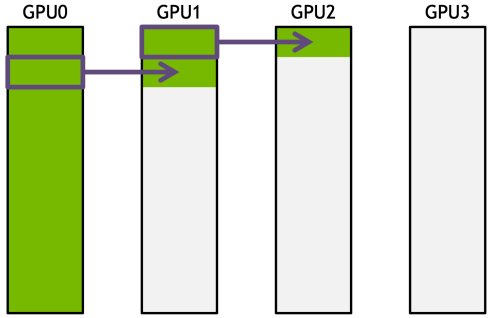
RELAXED SYSTEM LAB



Step 0

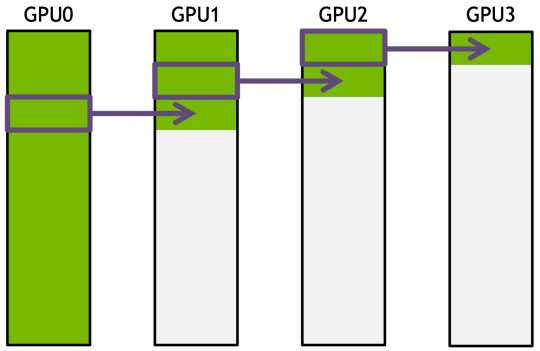


Step 1

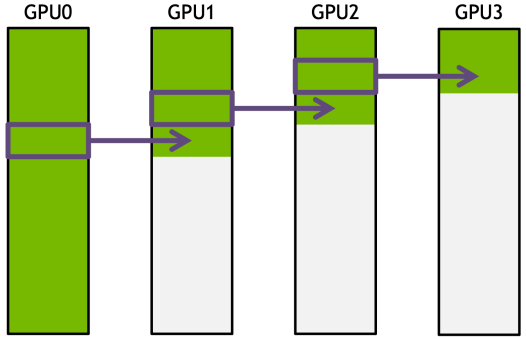


Step 2

- N: bytes to broadcast
- B: bandwidth of each link
- k: number of GPUs
- Split data to s message.
- Each step $t = \frac{N}{SB}$
- Total time: $\frac{(S+k-2)N}{SB} \rightarrow \frac{N}{B}$.

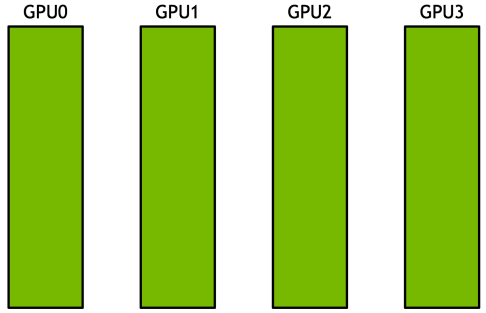


Step 3



Step 4

……



By the end

Ring based AllReduce

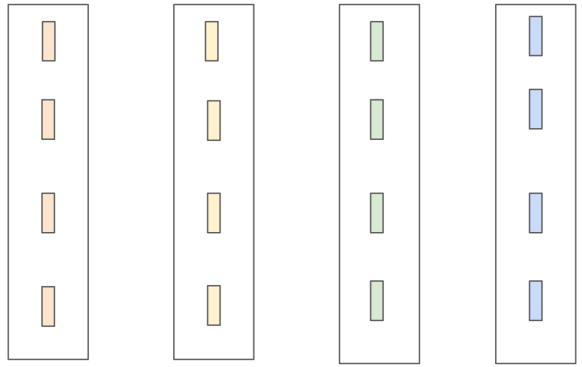
RELAXED
SYSTEM LAB

- In ring based AllReduce, we assume:
  - N: bytes to aggregate
  - B: bandwidth of each link
  - k: number of GPUs
  - The original tensor is equally split into k chunks.
- Ring based AllReduce implementation has two phases:
  - Reduction phrase (Aggregation phrase);
  - AllGather Phrase.
  - Total time: $\frac{2(k-1)N}{kB}$.

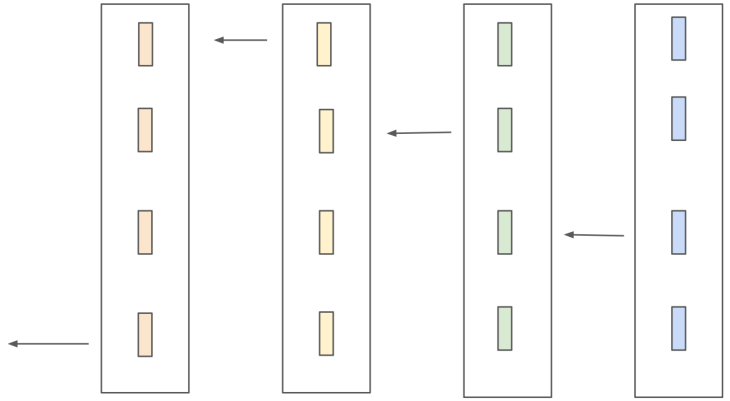
Ring based AllReduce- Reduction Phrase

RELAXED SYSTEM LAB

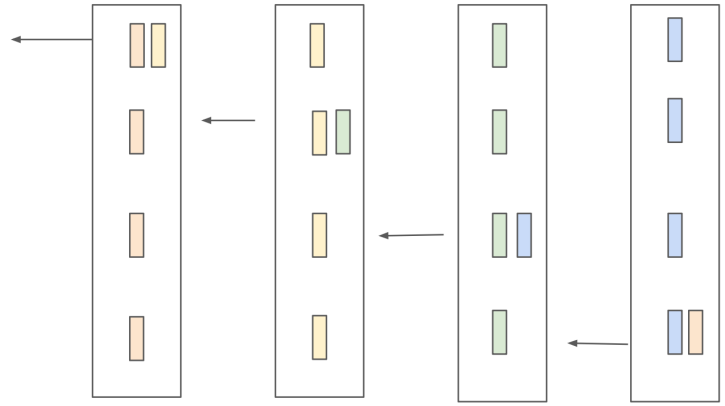
In this visualization, two or more blocks mean the aggregation results of two or more blocks by the same shape as the original block.



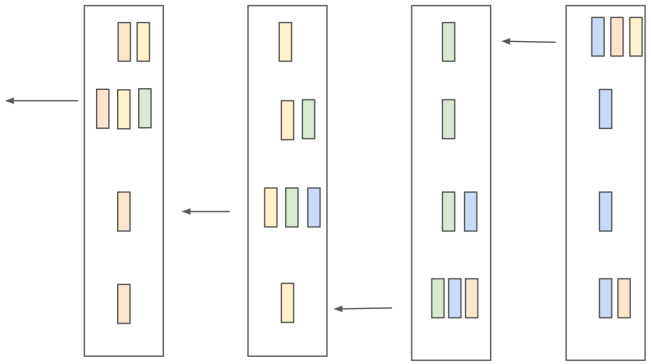
Initial State



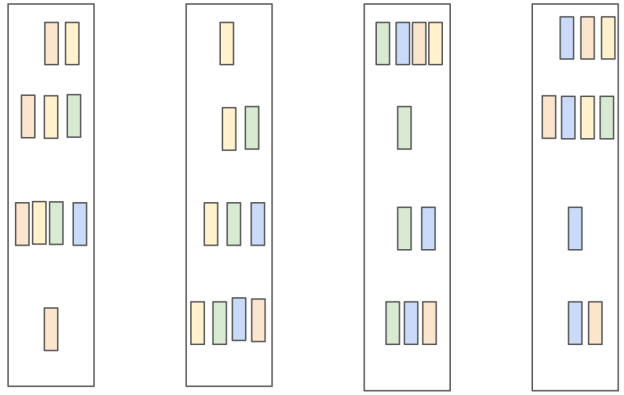
Step 0



Step 1



Step 2



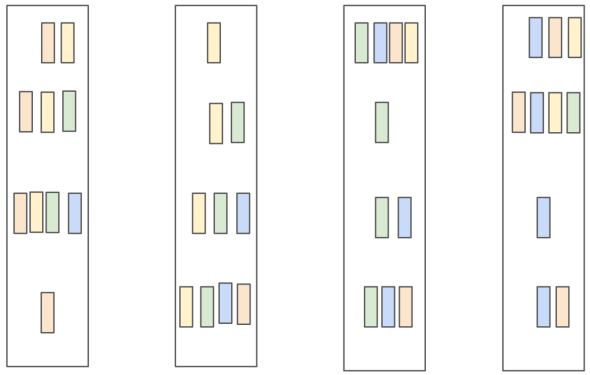
Step 3

- N: bytes to aggregate
- B: bandwidth of each link
- k: number of GPUs
- Time in reduction phase: $\frac{(k-1)N}{kB}$.

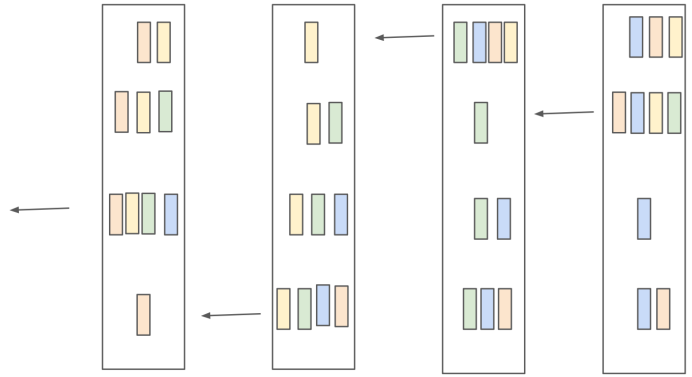
Ring based AllReduce

RELAXED SYSTEM LAB

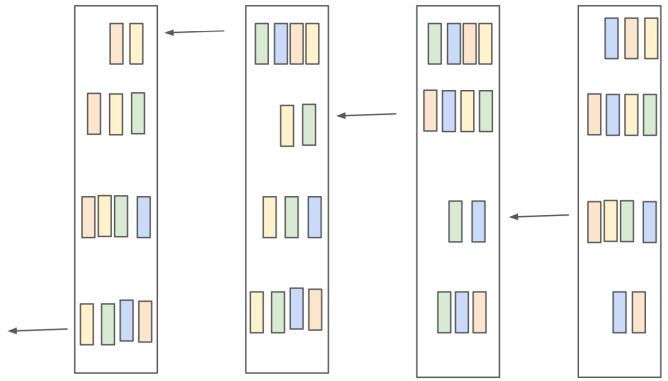
In this visualization, two or more blocks mean the aggregation results of two or more blocks by the same shape as the original block.



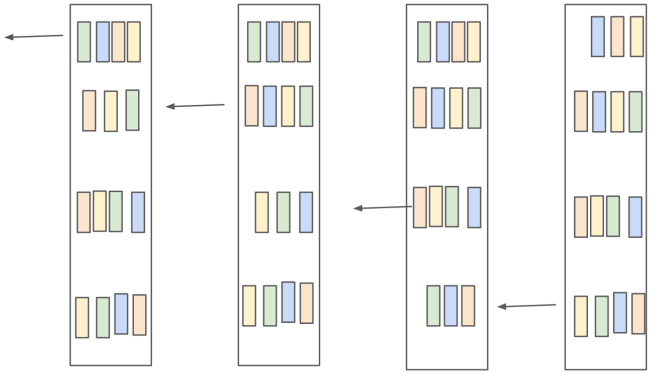
Initial State



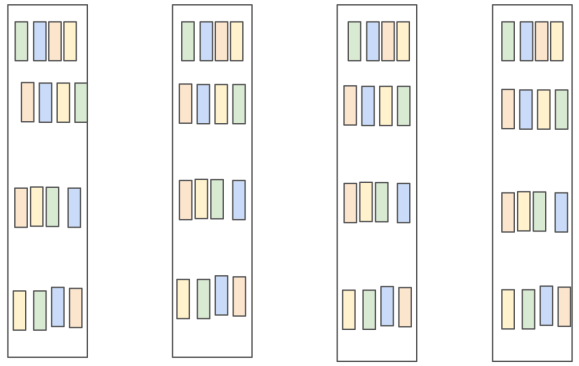
Step 0



Step 1



Step 2



Step 3

- N: bytes to broadcast
- B: bandwidth of each link
- k: number of GPUs
- Time in AllGather phase: $\frac{(k-1)N}{kB}$.

NCCL Practice in PyTorch

PyTorch Distributed NCCL Backend

RELAXED SYSTEM LAB

- PyTorch Distributed Process Groups:
  - Process groups (PG) take care of communications across processes. It is up to users to decide how to place processes, e.g., on the same machine or across machines. PG exposes a set of communication APIs, e.g., send, recv, and the collective communication operators.
- Process Group Backends:
  - Gloo: for distributed CPU training;
  - NCCL: for distributed GPU training;
  - MPI.

Backend	`gloo`		`mpi`		`nccl`	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✘	✓	?	✘	✓
recv	✓	✘	✓	?	✘	✓
broadcast	✓	✓	✓	?	✘	✓
all_reduce	✓	✓	✓	?	✘	✓
reduce	✓	✘	✓	?	✘	✓
all_gather	✓	✘	✓	?	✘	✓
gather	✓	✘	✓	?	✘	✓
scatter	✓	✘	✓	?	✘	✓
reduce_scatter	✘	✘	✘	✘	✘	✓
all_to_all	✘	✘	✓	?	✘	✓
barrier	✓	✘	✓	?	✘	✓

Initialize the Process Group.

RELAXED SYSTEM LAB

- The PyTorch distributed package needs to be initialized using the torch.distributed.init_process_group() function before calling any other methods.
- The process will block until all processes have joined.
- By default collectives operate on the default group (also called the world) and require all processes to enter the distributed function call.
- However, some workloads can benefit from more fine-grained communication.
- new_group() function can be used to create new groups, with arbitrary subsets of all processes. It returns an opaque group handle that can be given as a group argument to all collectives.

torch.distributed.init_process_group(*backend=None, init_method=None, timeout=None, world_size=-1, rank=-1, store=None, group_name='', pg_options=None*) [SOURCE]

Initialize the default distributed process group.

This will also initialize the distributed package.

There are 2 main ways to initialize a process group:

1. Specify `store`, `rank`, and `world_size` explicitly.
2. Specify `init_method` (a URL string) which indicates where/how to discover peers. Optionally specify `rank` and `world_size`, or encode all required parameters in the URL and omit them.

If neither is specified, `init_method` is assumed to be "env://".

Parameters

- **backend** (*str or Backend, optional*) – The backend to use. Depending on build-time configurations, valid values include `mpi`, `gloo`, `nccl`, and `ucc`. If the backend is not provided, then both a `gloo` and `nccl` backend will be created, see notes below for how multiple backends are managed. This field can be given as a lowercase string (e.g., `"gloo"`), which can also be accessed via `Backend` attributes (e.g., `Backend.GLOO`). If using multiple processes per machine with `nccl` backend, each process must have exclusive access to every GPU it uses, as sharing GPUs between processes can result in deadlocks. `ucc` backend is experimental.
- **init_method** (*str, optional*) – URL specifying how to initialize the process group. Default is "env://" if no `init_method` or `store` is specified. Mutually exclusive with `store`.
- **world_size** (*int, optional*) – Number of processes participating in the job. Required if `store` is specified.
- **rank** (*int, optional*) – Rank of the current process (it should be a number between 0 and `world_size`-1). Required if `store` is specified.
- **store** (*Store, optional*) – Key/value store accessible to all workers, used to exchange connection/address information. Mutually exclusive with `init_method`.
- **timeout** (*timedelta, optional*) – Timeout for operations executed against the process group. Default value is 10 minutes for NCCL and 30 minutes for other backends. This is the duration after which collectives will be aborted asynchronously and the process will crash. This is done since CUDA execution is async and it is no longer safe to continue executing user code since failed async NCCL operations might result in subsequent CUDA operations running on corrupted data. When TORCH_NCCL_BLOCKING_WAIT is set, the process will block and wait for this timeout.
- **group_name** (*str, optional, deprecated*) – Group name. This argument is ignored
- **pg_options** (*ProcessGroupOptions, optional*) – process group options specifying what additional options need to be passed in during the construction of specific process groups. As of now, the only options we support is `ProcessGroupNCCL.Options` for the `nccl` backend, `is_high_priority_stream` can be specified so that the nccl backend can pick up high priority cuda streams when there're compute kernels waiting.

Blocking Mode VS. Non-Blocking Mode

RELAXED SYSTEM LAB

- Blocking mode: all processes stop until the communication is completed.
- Non-blocking: the script continues its execution and the methods return a Work object upon which we can choose to wait().

Blocking Mode

```
def init_process(rank, backend='nccl'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=2)

def run(rank, size, device):
    tensor = torch.zeros(10).to(device)
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
        dist.send(tensor=tensor, dst=1)
    else:
        # Receive tensor from process 0
        dist.recv(tensor=tensor, src=0)
    print('Rank ', rank, ' has data ', tensor[0])
```

Blocking Mode VS. Non-Blocking Mode

RELAXED SYSTEM LAB

- Blocking mode: all processes stop until the communication is completed.
- Non-blocking: the script continues its execution and the methods return a Work object upon which we can choose to wait().

Non-Blocking Mode

```
def init_process(rank, backend='nccl'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=2)

def run(rank, size):
    tensor = torch.zeros(10).to(device)
    req = None
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
        req = dist.isend(tensor=tensor, dst=1)
        print('Rank 0 started sending')
    else:
        # Receive tensor from process 0
        req = dist.irecv(tensor=tensor, src=0)
        print('Rank 1 started receiving')
    # Call print('Rank ', rank, ' has data ', tensor[0]) here may present incorrect results.
    req.wait()
    print('Rank ', rank, ' has data ', tensor[0])
```

Blocking Mode VS. Non-Blocking Mode

RELAXED SYSTEM LAB

- For collective communications, you should set the **async_op** to determine whether to run it in blocking mode (**async_op =False** by default) or Non-blocking mode (**async_op =True**).

Blocking Mode

```
def init_process(rank, backend='nccl'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=4)

def run(rank, size, device):
    tensor = torch.ones(10).to(device)
    dist.all_reduce(tensor, op=dist.ReduceOp.SUM)
    print('Rank ', rank, ' has data ', tensor[0])
```

Blocking Mode VS. Non-Blocking Mode

RELAXED SYSTEM LAB

- For collective communications, you should set the **async_op** to determine whether to run it in blocking mode (**async_op =False** by default) or Non-blocking mode (**async_op =True**).

Non-Blocking Mode

```
def init_process(rank, backend='nccl'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=4)

def run(rank, size, device):
    tensor = torch.ones(10).to(device)
    handle = dist.all_reduce(tensor, op=dist.ReduceOp.SUM, async_op=True)
    # Call print('Rank ', rank, ' has data ', tensor[0]) here may present incorrect results.
    handle.wait()
    print('Rank ', rank, ' has data ', tensor[0])
```

References

- https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf
- https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
- https://www.alcf.anl.gov/sites/default/files/2021-07/ALCF_A100_20210728%5B80%5D.pdf
- https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf
- https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
- https://www.alcf.anl.gov/sites/default/files/2021-07/ALCF_A100_20210728%5B80%5D.pdf
- https://hc34.hotchips.org/assets/program/conference/day2/Network%20and%20Switches/NVSwitch%20HotChips%202022%20r5.pdf
- https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html#collective-operations
- https://pytorch.org/tutorials/intermediate/dist_tuto.html
- https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf
- https://dlsys.cs.washington.edu/pdf/lecture11.pdf
- https://docs.mstarcfd.com/KB/peer-access.html