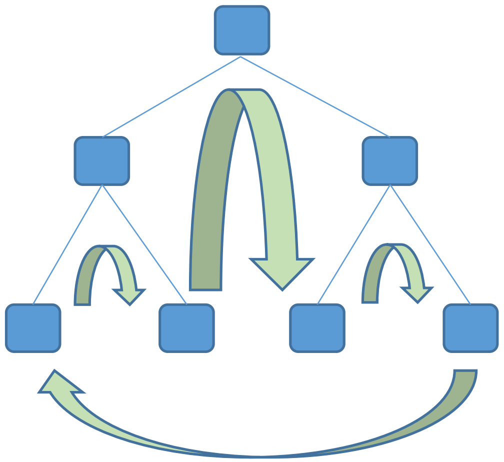
HPML 12
Rings and Things

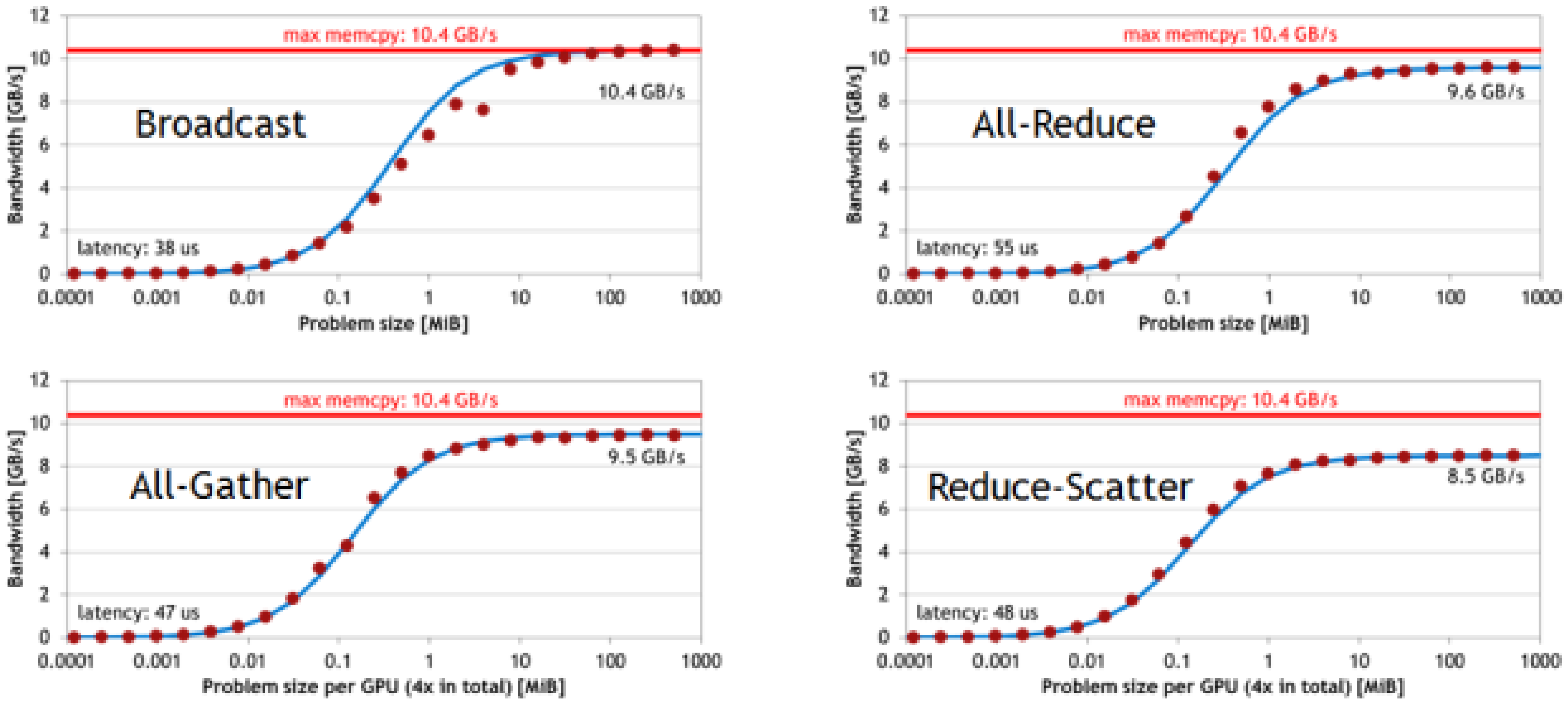
Ulrich Finkler

Ring Algorithm for Collectives

- PCIe tree or network tree topology common
  - PCIe links bidirectional
  - Ethernet links bidirectional
- Can overlay ring
- Uses both directions of all cables



NCCL



4 GeForce GTX Titan X in PCIe tree
https://devblogs.nvidia.com/fast-multi-gpu-collectives-nccl/

NCCL use pattern

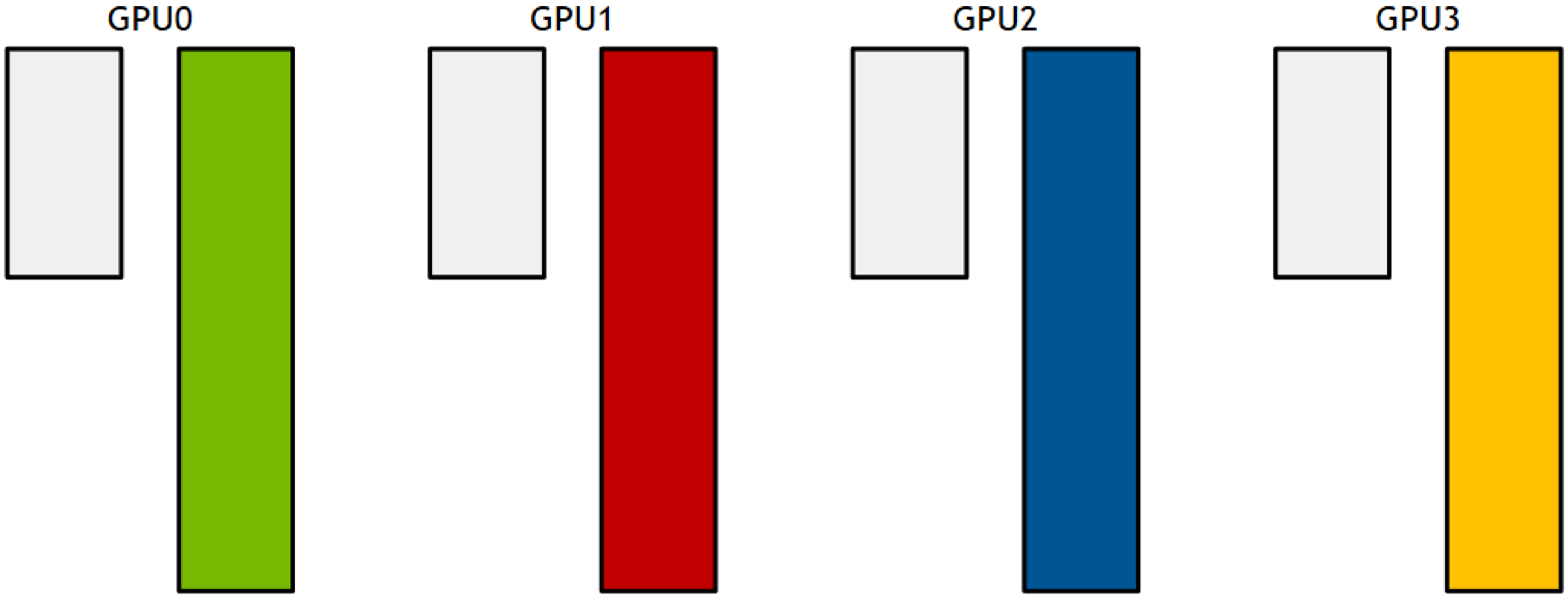
```
if (myRank == 0) ncclGetUniqueId(&id);
MPICHECK(MPI_Bcast((void *)&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD))
…

NCCLCHECK(ncclCommInitRank(&comm, nRanks, id, myRank));

cudaStream_t s;
CUDACHECK(cudaStreamCreate(&s));
NCCLCHECK(ncclAllReduce((const void*)sendbuff, (void*)recvbuff, size, ncclFloat, ncclSum, comm, s));

CUDACHECK(cudaStreamSynchronize(s));
```

https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide

ALL-REDUCE

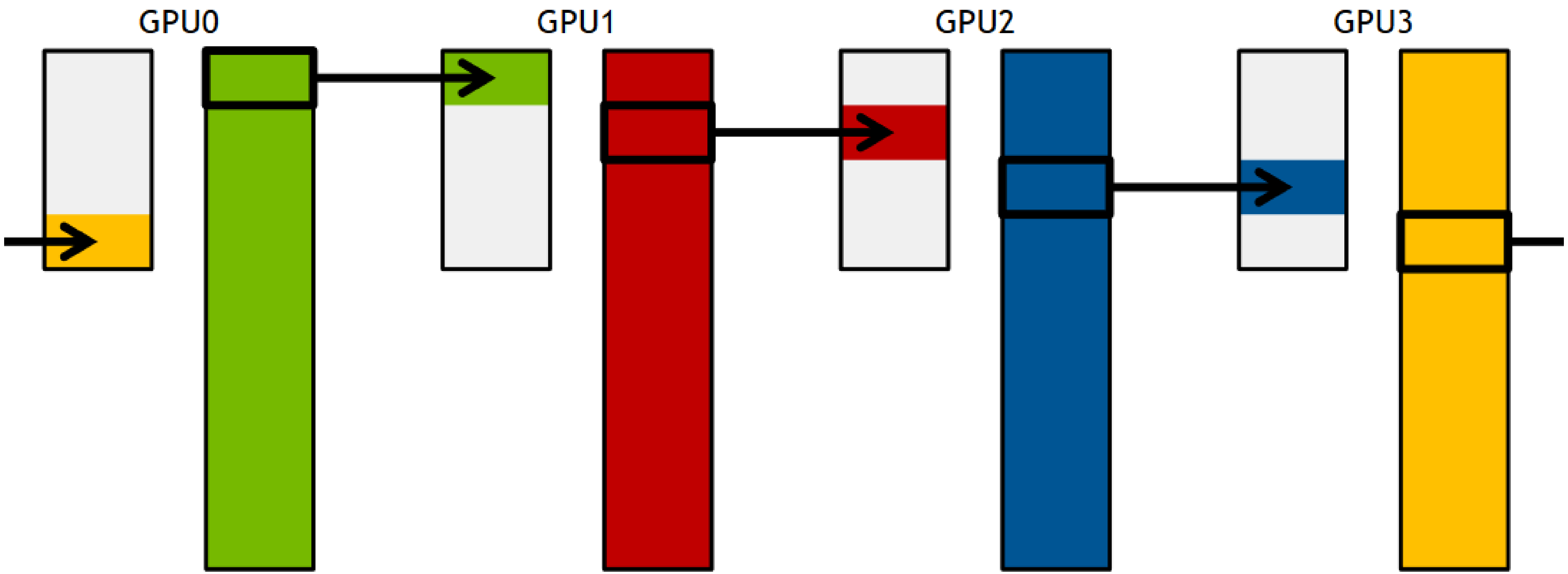
with unidirectional ring

Chunk: 1
Step:



ALL-REDUCE
with unidirectional ring

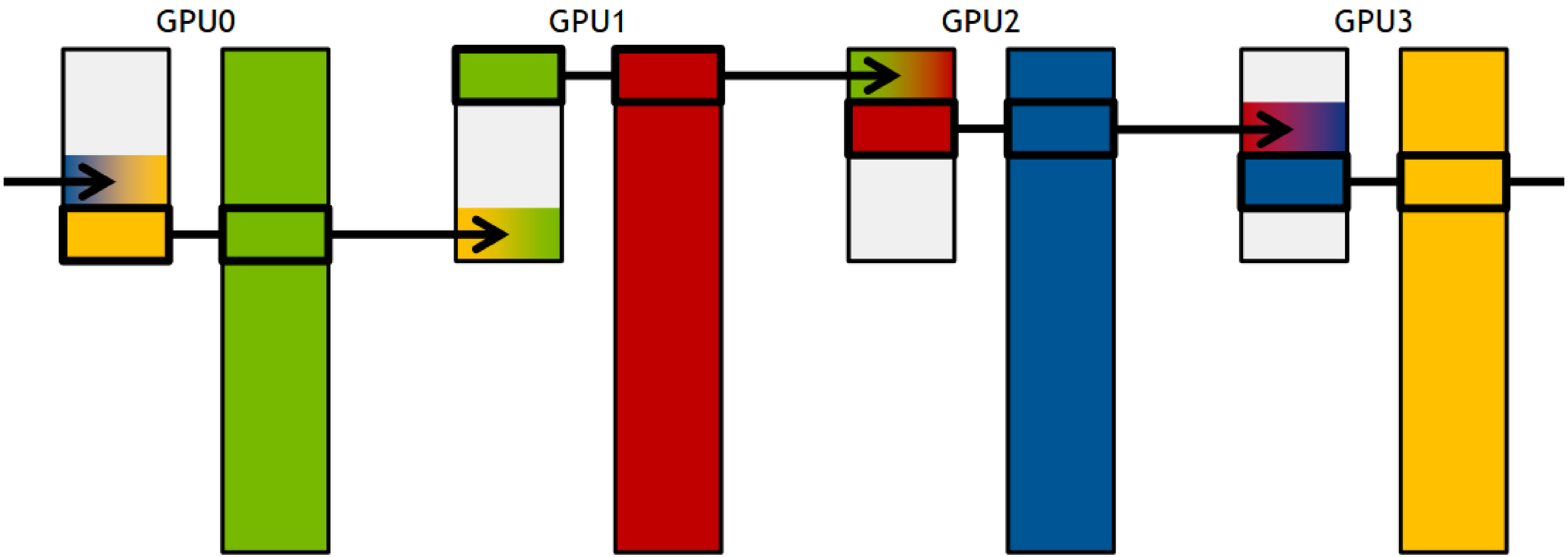
Chunk: 1
Step: 1



ALL-REDUCE

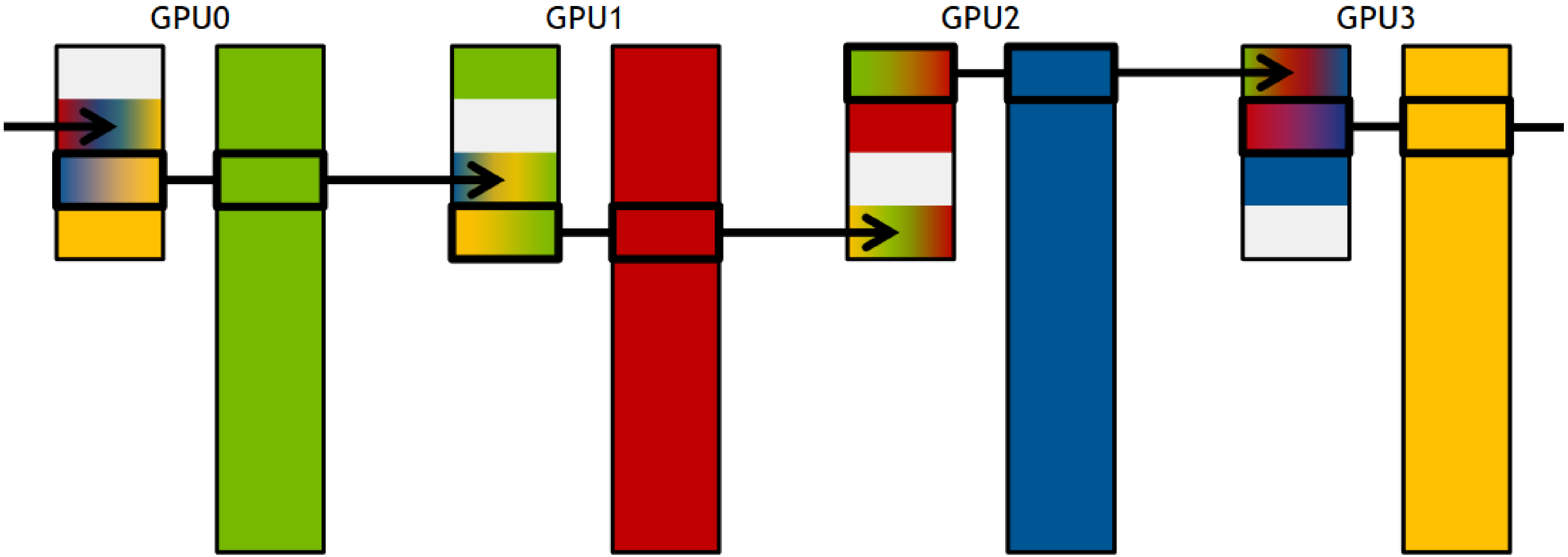
with unidirectional ring

Chunk: 1
Step: 2



ALL-REDUCE
with unidirectional ring

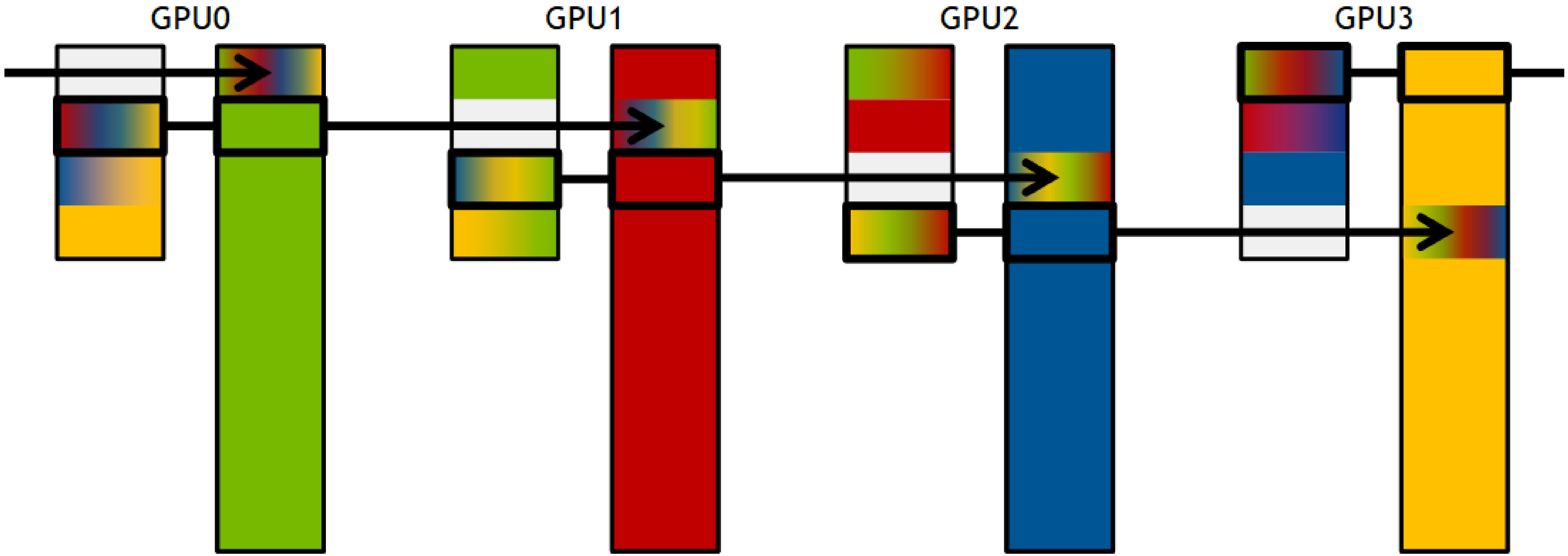
Chunk: 1
Step: 3



ALL-REDUCE

with unidirectional ring

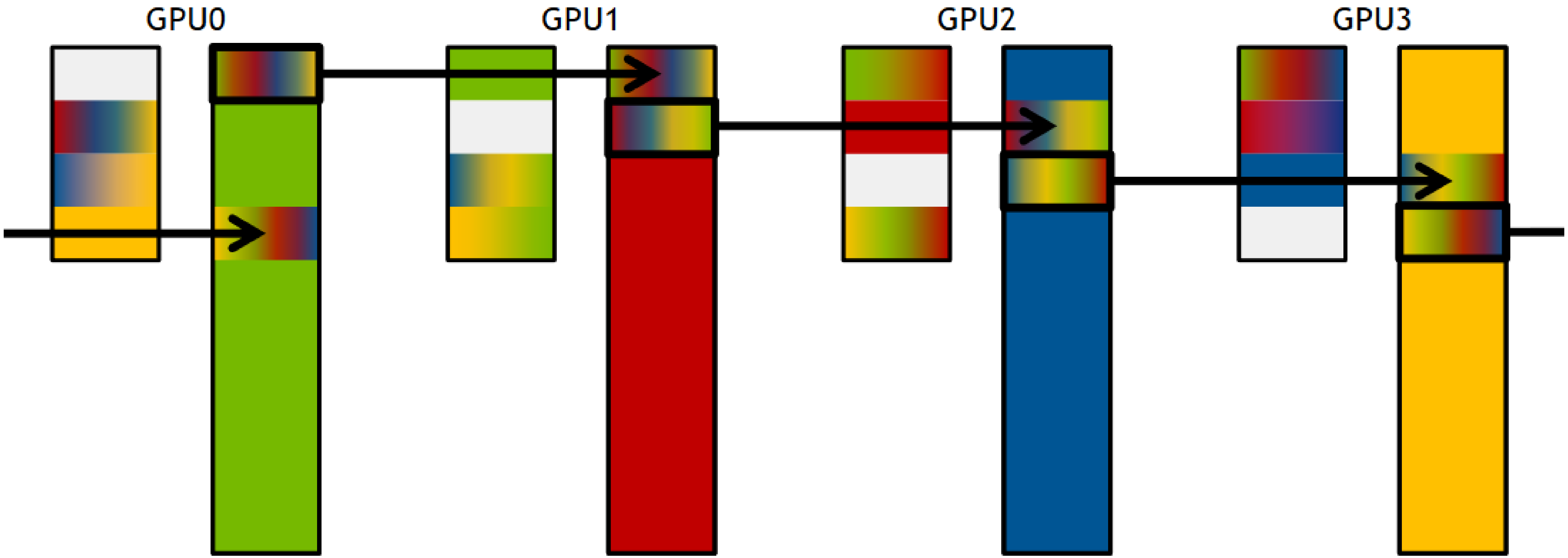
Chunk: 1
Step: 4



ALL-REDUCE

with unidirectional ring

Chunk: 1
Step: 5

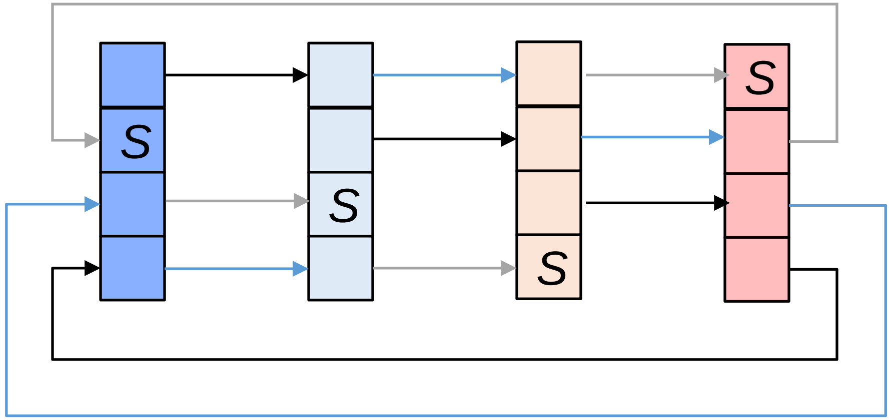


Animation from :

Cliff Woolley, Sr. Manager, Developer Technology Software,

NCCL: ACCELERATED MULTI-GPU COLLECTIVE COMMUNICATIONS

Multi GPU Learning Synchronization

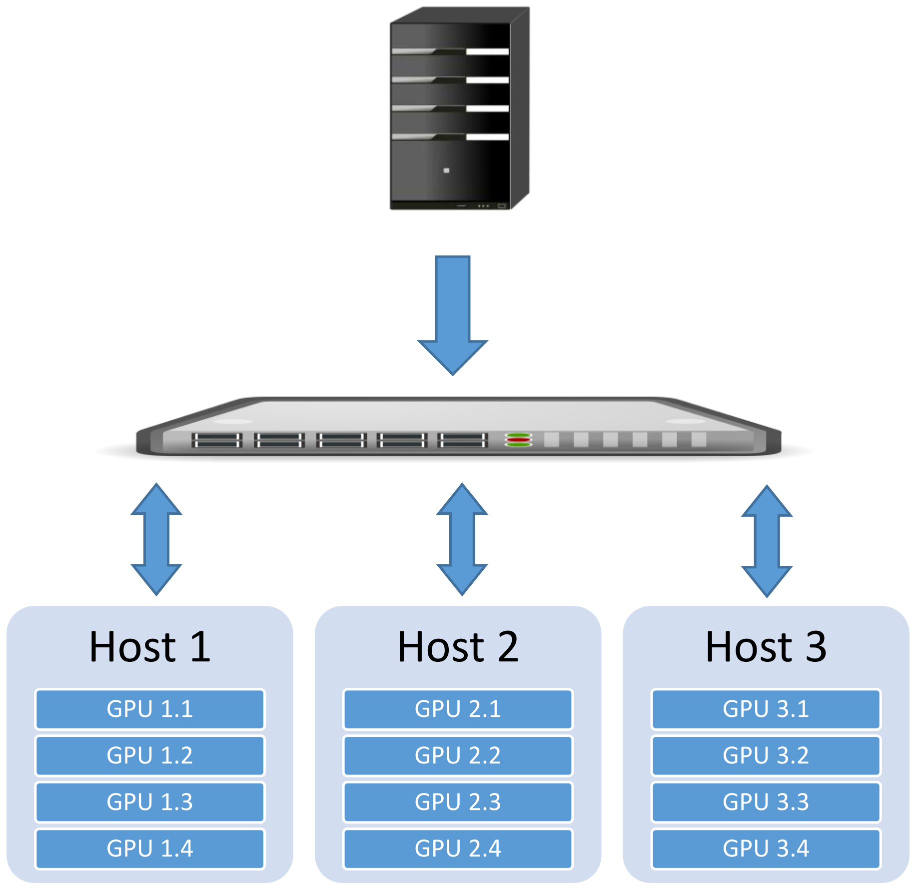


- Reduce-Scatter/All-gather
- Ring communication
- Concurrently compute sum in N-1 phases (color of arrows)
- Analogously distribute result ‘S’ to all GPUs
- Scales linear with number of GPUs

One phase: N*12GB/s throughput (PCIe Peer to peer ring)
Total data volume 2*(N-1)* |network-parameters|

- All links busy all the time
- Optimal complexity O(|network parms|)
- For large rings, many phases => sensitive to latency and variability
- Hierarchical rings for heterogenous topologies

Feeding the 'Beasts'



Storage System

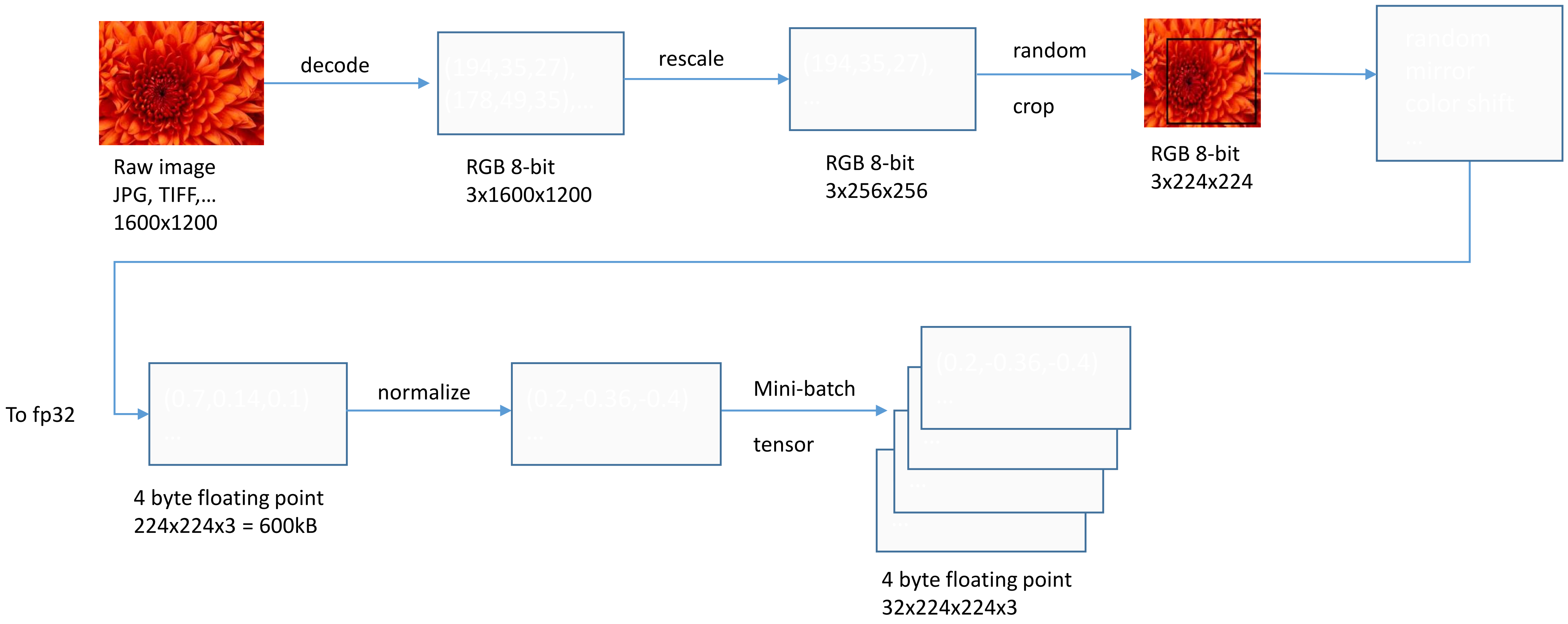
Network (IB, 40 GbitE or similar)

Hosts:
Dual Socket, 20+ physical cores
4 P100 GPUs
256 GB RAM, 2 TB local storage

Starting Point: Benchmark Setup

- Deep Learning Framework, e.g. Tensorflow, PyTorch
- Imagenet benchmark, e.g. fp32, resnet 50
- MPI-augmented training script
  - 1 Rank per GPU
  - High-performance parameter synchronization (allreduce)
  - Forward-backward pass batch size 32, single GPU < 100 ms
  - Including parameter synchronization < 150 ms per iteration
    - Time per iteration similar for e.g. 2 – 64 hosts with good network
- Throughput 32 GPUs (8 nodes) ~ 7000 images/sec
  - Effective batch size 1024 (still stable, resource efficient conversion)
  - ~ 3 minutes per epoch
- Effective setup, cost/throughput nearly constant over wide range

Feeding Training Data



Data Formats

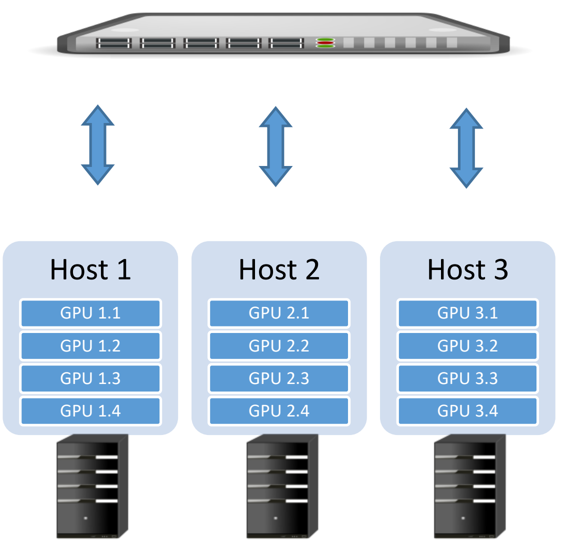
- LMDB (Caffe)
  - Dataset in one large file that is memory mapped
  - Prerandomized, sufficient for convergence in many cases
  - Decoding and scaling already performed, e.g. 256x256x3 byte RGB
  - Class identifier and image co-located
- TFRecord (Tensorflow)
  - Typically multiple large files, each with hundreds of training points
  - A file is a sequence of data + metadata pairs partitioned by separators
  - Class identifier and image co-located
- Raw (PyTorch/Torchvision)
  - Each image in separate file (usually jpeg)
  - Class index deduced from directory structure

From Benchmark to Application

- Dataset size can be multiple TB (Speech, richer image sets, …)
- Experimentation with different configurations
  - Different networks with different input tensor sizes, e.g. 299x299x3
  - Different subsets of the data
  - Different classifications
    - Sparser or denser partitioning over the dataset
    - Different semantics, e.g. walking-running/male-female
- Problems
  - Training throughput much lower than benchmark
  - Generating TFRecord files or LMDB files with different resolutions and/or class assignments takes hours, even creating symbolic links for a new directory structure can take 1 hour+
  - Each variant creates a copy of the data or creates millions of symbolic links

Step 1

- 7000 images/sec, 0.5-1 MB per image => 3.5-7 Gbyte/sec
  - Dataset >> RAM, not cached, every epoch fetches the entire dataset!
  - Exceeds the capacity of the network interface to the storage system
- Improvement
  - Use local storage on compute nodes, overlay GlusterFS
    - Storage bandwidth scales linearly with number of compute nodes
    - A small amount of additional CPU load
    - No additional hardware cost!
  - Text file to associate meta-data with locations of raw input files
    - Selecting subsets, changing class assignments by processing a text file
    - Many randomly distributed files lead to natural load balancing!
- Problem: Throughput still significantly too low!



Bottleneck 2

- Transformation from file to tensor
  - Load data element, decompress, e.g. jpeg to RBG representation
  - Data augmentation, e.g. random crop, mirror, scale, color normalization
  - Transform to floating point tensor (raw data usually quantized)
  - Random selection of images from large dataset => file access latency
- Cost
  - ~100 ms latency to fetch image, high variability
  - >= 20 ms data augmentation compute cost per image (after some tuning)
- Single host: ~150 ms to process 128 images, ~ 1 ms/image
  - V100 with fp16 hybrid computation ~ 3 X faster
  - Resnet 18 vs Resnet 50 ~ 4 X faster (¼ of compute, ¼ of data volume)
  - Compute time per iteration can drop below 20 ms!

Step 2

- Concurrent file requests to hide latency
  - T1 ~= N*B*<L>/w, e.g. 4*32*100ms/w
- Concurrent workers to parallelize computation
  - T2 ~= N*B*T/w, e.g. 4*32*20ms/w
- Symbols
  - N : number of GPUs
  - B : number of data elements per GPU and iteration
  - <L> : average latency to obtain a file
  - T: time to perform the transformation
- ~20 workers per GPU roughly enough
  - T1 ~= 12800/80 ms = 160 ms
  - T2 ~= 2650/80 ms = 32 ms
- Problem: Throughput improved, but still significantly below target

Bottleneck 3

- Bottleneck is N:1 handoff from python workers to ‘trainer’
  - Python uses processes, not threads
  - Partial serialization in the interpreter
  - => Parallel workers in Python have limited scalability
- Python extension with threads in C/C++
  - Mutex protected queue to collect tensors, pthread primitives
    - Worker has mini-batch
    - Acquires mutex, checks queue status
    - If room in queue, add mini-batch
    - Releases mutex
  - Still below target!

Python Extensions, setup.py

```
from distutils.core import setup, Extension
import os

LocDir = os.environ['PWD']

module1 = Extension(‘myext',
                    sources = [‘myext.c'],
                    library_dirs=[LocDir],
                    libraries=[‘myext'])

setup (name = ‘myext',
       version = '0.1',
       description = 'python extension',
       ext_modules = [module1])
```

Python Extension, Interface code

```
#include <Python.h>

#include <numpy/arrayobject.h>


static PyObject *myext_Nextbatch(PyObject *self, PyObject *args) {

  uint64_t hdl;

  if (!PyArg_ParseTuple(args, "k",  &hdl))      return NULL;

  float *ptr= NextBatch(hdl);                                                    // hdl identifies loader instance

  …

  npy_intp idims[4]={bsz,hght,wdth,chnls};

  PyArrayObject *pImgs = (PyArrayObject*)(PyArray_SimpleNewFromData(4, idims, NPY_FLOAT, (void*)(ptr)));

  return PyArray_Return(pImgs);

}

static PyMethodDef MyextMethods[] = {  {"Nextbatch",myext_Nextbatch,METH_VARARGS, "get next batch"}, {NULL,NULL,0,NULL} };

PyMODINIT_FUNC initmyext(void) {

…

m = Py_InitModule(“myext", MyextMethods);

…

}
```

Amdahl’s Law

$$S = \frac{1}{((1-p)+p/s)}$$

p: fraction of computation that is parallelized

s: speed up for the parallel part

$\Rightarrow$ A parallel program is never slower than a sequential one

This is ‘not entirely accurate’

Amdahl’s Law

$$S = \frac{1}{((1-p)+p/s)}$$

p: fraction of computation that is parallelized
s: speed up for the parallel part
$\Rightarrow$ A parallel program is never slower than a sequential one ??
This conclusion is ‘not entirely accurate’

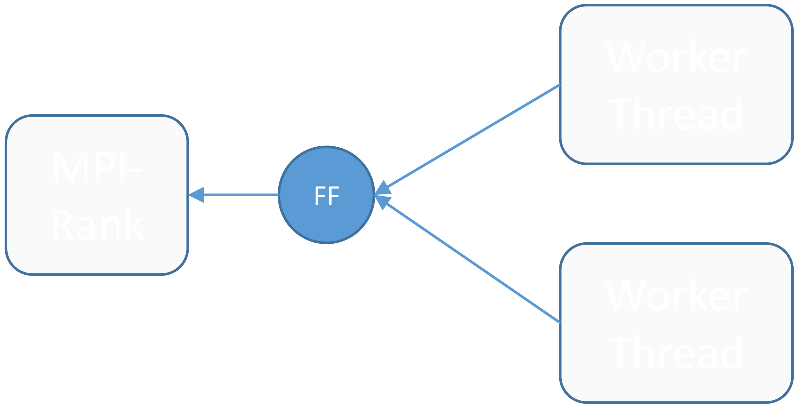
$$S = \frac{C}{\sum_i \frac{C'_i}{s_{C'i}} + \sum_j \frac{E_j}{s_{Ej}} + \sum_k \frac{I_k}{s_{Ik}}}$$

C: Cost of sequential computation
C’ >= C: Cost of computation in parallel algorithm
E : Cost of explicitly added ops (pthread_lock …)
I: Cost if implicititly added ops (coherence protocol …)
Sx: Degree of parallelization for a fraction of the ops

The synchronization operations add significant explicit compute cost with low speedup!
On 8 GPU node, V100+Res18 , ~20 ms per iteration and ~10 ms handoff overhead!

Step 3

- N:1 handoff with atomic flip-flop (‘lock-free’)
- bool __sync_bool_compare_and_swap (t *ptr, t oldv, t newv, ...)
  - If ‘*ptr’ contains ‘oldv’, set it to ‘newv’
  - Return ‘true’ if successful
  - Only one out of N concurrent attempts succeeds



FlipFlop Abstract

- A single 64-bit variable V in which the ‘trainer’ receives addresses of tensors with mini-batches
- States of V
  - V==0: The ‘flipflop’ is empty, ‘workers’ may attempt to write to it
  - V!=0: The ‘flipflop’ is loaded, ‘workers’ may not write to it
- Construct performs three tasks
  - Merges concurrent data streams from the workers into a series
  - ‘Paces’ the workers so that their throughput does not exceed that of the trainers
  - ‘Holds’ the trainer if there are no data available

The FlipFlop (flawed)

```
class FlipFlop {
  uint64_t val;

  FlipFlop(void) { val=0;}                                   // initially empty

  bool tryadd(uint64_t V) {
    return __sync_bool_compare_and_swap (&val,0,V);      // succeeds only if val is 0
  }

  uint64_t get(void) {
    uint64_t V;
    for (;;) {

      V=val;
      if (V!=0) {
          if (__sync_bool_compare_and_swap (&val,V,0))       // returns once nonzero V was acquired
            return V;
      }
      usleep(10);
    }
  } /* endmethod */
};
```

Memory Consistency

- Memory consistency establishes rules that determine when changes made to memory location X by one execution sequence (e.g. thread) become visible to another execution sequence
- Relaxed, order preserving consistency model
  - B written after A by execution sequence 1 => if A visible by execution sequence 2, then B is also visible by execution sequence 2
  - No guarantee on how long it takes for the change of V by the worker to become visible to the trainer!
- Weak consistency
  - Only a memory barrier guarantees that all writes before the barrier by ex-seq 1 are visible to ex-seq 2
  - The data referenced by the pointer V may be wrong in the view of the trainer, unless the atomic operation implementation on the target platform also acts as a full memory barrier

The FlipFlop

```
class FlipFlop {
  uint64_t val;

  FlipFlop(void) { val=0;}                                  // no value present

  bool tryadd(uint64_t V) {
    return __sync_bool_compare_and_swap (&val,0,V);      // succeeds if val is 0
  }

  uint64_t get(void) {
    uint64_t V;
    for (;;) {
      __sync_synchronize();                                 // memory barrier
      V=val;
      if (V!=0) {
          if (__sync_bool_compare_and_swap (&val,V,0))      // returns once nonzero V was acquired
            return V;
      }
      usleep(10);                                           // reducer
    }
  } /* endmethod */
};
```

Reducer: Why sleep ?

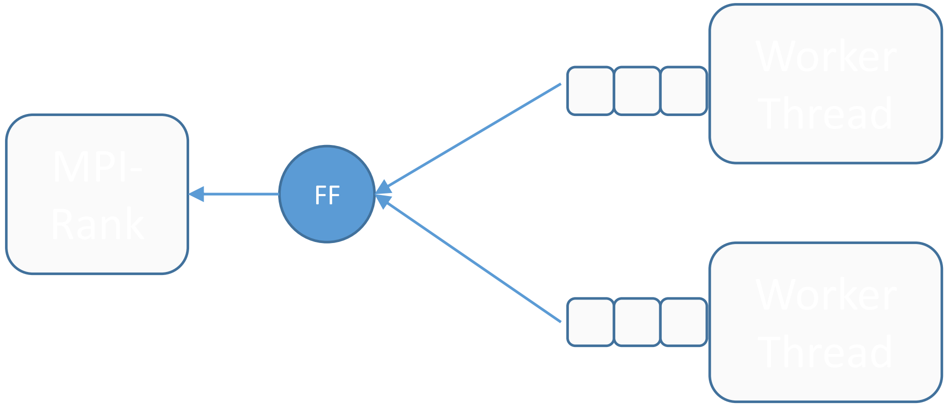
- A: Reduces CPU use of waiting workers
  - We have more threads/processes than physical cores
  - Workers, Trainer, OS processes, filesystem processes compete for resources
- B: ‘Yields’ a (virtual) core to allow another thread to be scheduled
  - Granularity of time slices for threads, spinning thread ‘holds’ the core until the time slice is completed
- C: Reduces load on the ‘atomic instruction’ logic
  - High frequency unsuccessful attempts by workers can impede the attempt of the trainer
- Optimal turnaround in the trainer dominated by the memory barrier
  - Decreasing return on investment of CPU time for spinning

Network/Filesystem Variability

- There can be long (hundreds of milliseconds, even seconds) periods in which network traffic or the distributed filesystem ‘stops’
  - OS maintenance tasks ‘hold’ a glusterfs daemon whose response is needed
  - Package failure that causes a timeout or retries on a network read
  - RAID array event, file system health check, …
- Any delay > 150 ms potentially impedes progress of ALL trainers since they have to synchronize their weights!
- Inherent buffer capacity <= number of workers

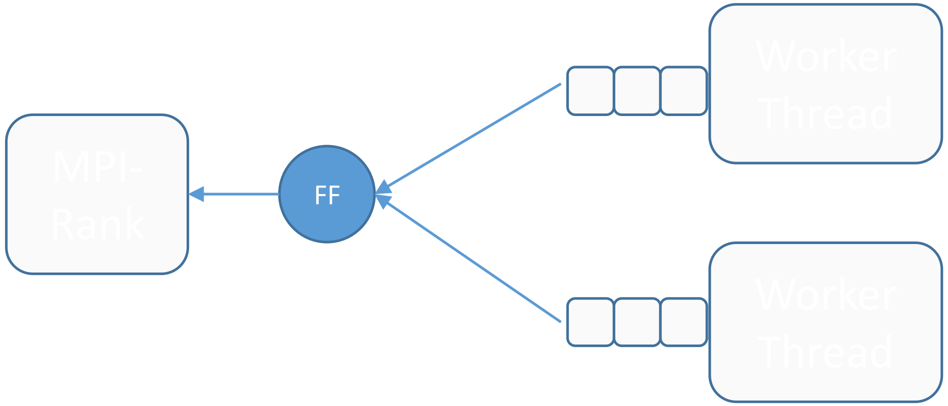
Worker Buffers

- Trainer pickup has minimal overhead => Buffer logic in workers
- Each worker has queue of fixed size, each entry is a mini-batch
- If attempt on flip-flop fails and free capacity, append to queue
- Only loop on FF if queue is full



Worker Buffers

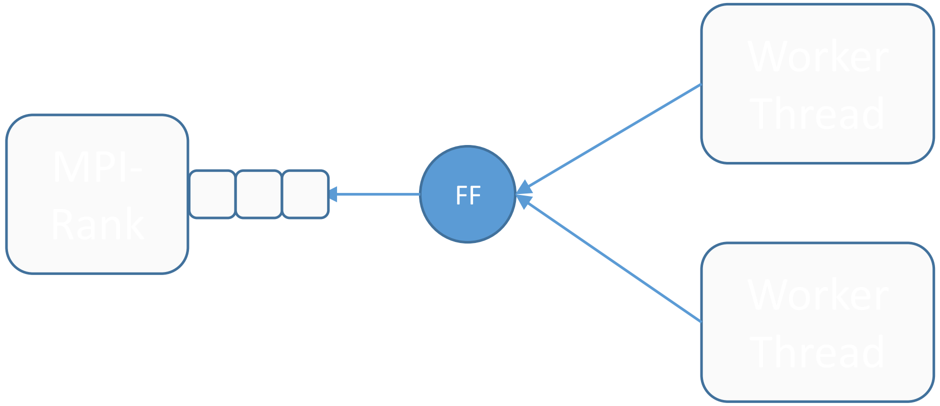
- Trainer pickup has minimal overhead => Buffer logic in workers
- Each worker has queue of fixed size, each entry is a mini-batch
- If attempt on flip-flop fails and free capacity, append to queue
- Only loop on FF if queue is full



If workers are blocked on network/filesystem, e.g. a ‘read’, they do not deliver to the flipflop
=> trainer can stall despite data ready
A filesystem ‘hold’ will affect all workers!

‘Take Two’

- Ring buffer of fixed size S in trainer
  - No allocation/deallocation, memory locality
- If n>S/2 entries
  - Try to get one entry from FF, use element from queue
- Else if n<=S/2 entries
  - Get one element from FF, try for a second



Auto-stabilizes around S/2 entries if workers provide enough throughput

This variant reaches our performance target if there are enough cores for data augmentation

TryGet

```
uint64_t tryget(void) {
   uint64_t V;
   __sync_synchronize();
   V=val;
   if (V!=0) {
     if (__sync_bool_compare_and_swap (&val,V,0))
                    return V;
   }
   return 0;
 } /* endmethod */
```

In essence ‘Get’ without the loop.

Reusing Memory

- A tensor covers many 4kB memory pages
- Memory ownership flows from worker (allocates) to trainer (deallocates)
  - One of the worst multithreaded use patterns for ‘malloc’
- Low memory locality, location of new tensor depends on ‘malloc’
  - Cache misses, TLB misses, file cache evictions potentially triggered by allocation

Flip-Flop provides unidirectional data flow, only state change to ‘empty’ flows back!

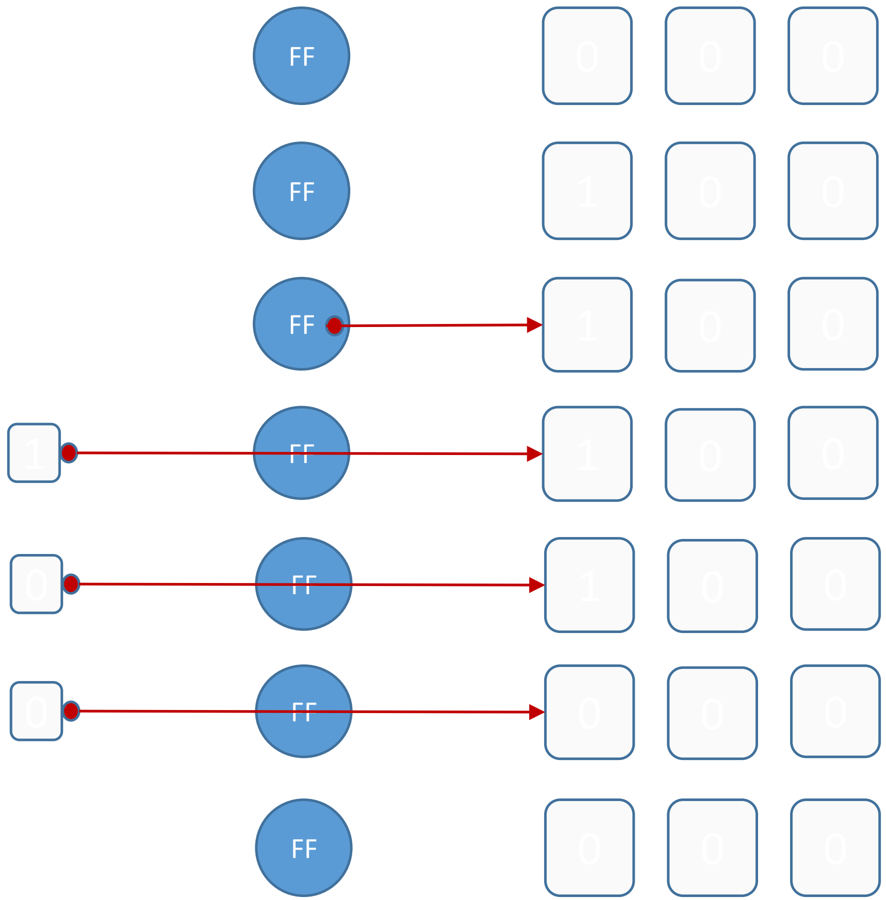
How do we return memory blocks to the workers with minimal overhead ?

Piggybacking

- Each worker initialized with a pool of M memory blocks
  - Blocks contain a prefix and a tensor
  - M status variables, initially zero (memory block unused)
- Worker
  - ‘allocates’ by selecting ‘0’ status slot, sets status to ‘1’
  - Sets prefix to address of status variable
  - Passes pointer to augmented block to trainer through FF
- Trainer
  - Receives augmented block, uses it (potentially delayed through queue)
  - Before taking the next block, writes ‘1’ to status variable of last used block

Why does it work even with weak consistency with no atomics or synchronization?

Trainer/Worker States



Only worker has view of free memory blocks

Worker prepares data, sets block to used

Worker loads address of block into flipflop

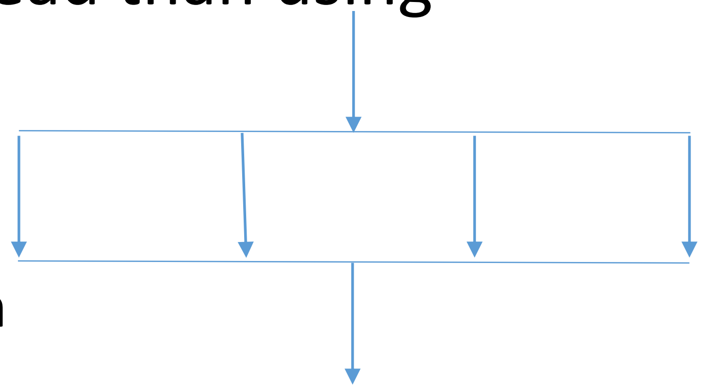
Trainer receives address, two views exist!

Trainer ‘releases’, views are different!

Sync happens, views consistent again

The Wave

- Example of another ‘lock-free’ pattern
  - E.g. in NLP processing, feature generation for text segment
- Let N threads work on common input data, produce a single set of results
- One to two orders of magnitude lower overhead than using pthread conditional variables
- Abstract
  - 0-1 flipflop to start the threads
  - Atomic counter to accumulate thread completion

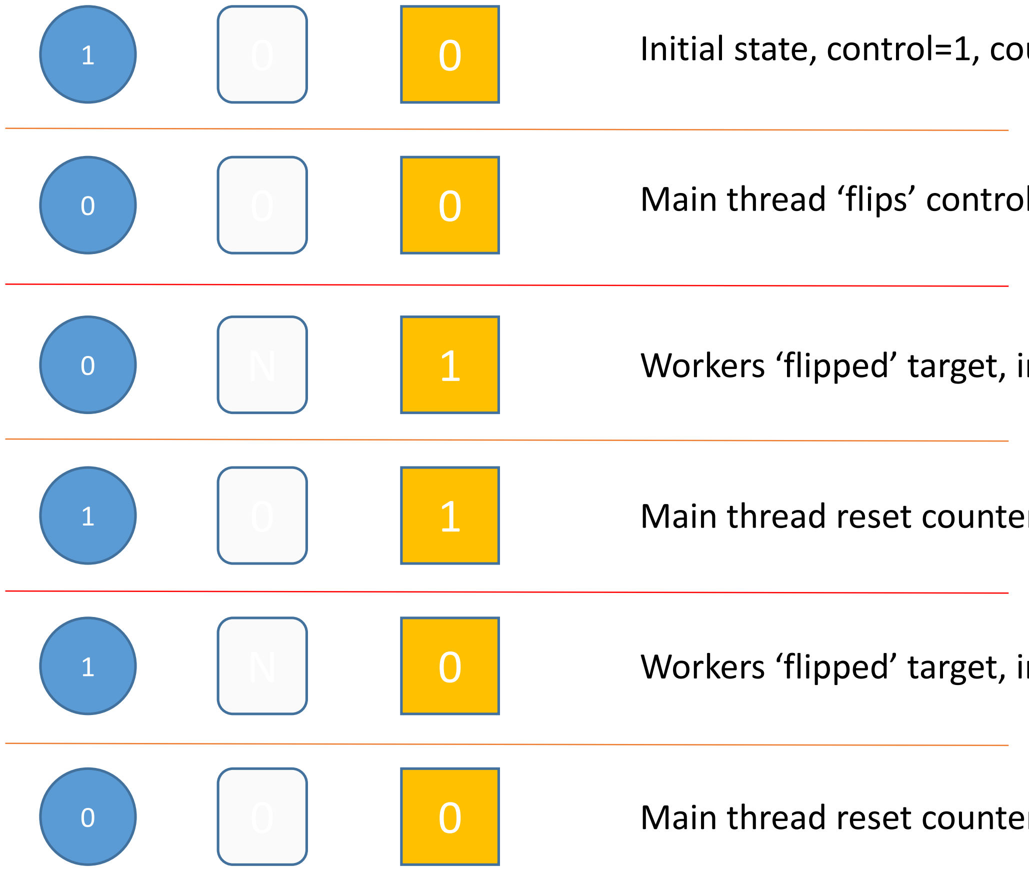


- Employs ‘__sync_fetch_and_add’
  - Acquire latest state of a variable and add a number to it

Wave Code

```
void atomicinc(int *var) {   __sync_fetch_and_add(var,1); }                  // atomic increment of variable
void pwait(int *var, int val) {
  while (true) { if ( (*var)==val) return;  sleep(…);   }                    // wait until desired state is reached (may need memory barrier)
}
void *TFunc(void *ptr) {                                                     // the threads. Ptr is where input is staged
  int tgt=0;                                                                 //  initial target state
  while (true) {
    pwait(GetCtrl(ptr),tgt);                                                 // wait for target state
    ExecJob(ptr);                                                            // perform computation
    atomicinc(GetCtrl(ptr));                                                 // increment ‘result collector’, first slot in ‘stage’
    tgt = (tgt==0) ? 1 : 0;                                                  // alternate target state
  } …
}
void DoWork(…) {
    …                                                                        // prepare the stage and memory barrier
    if (Ctrl==0) atomicinc(&Ctrl); else atomicdec(&Ctrl);                    // flip control state
    utl_pwait(&Cnt,Nthr);                                                    // pick up all the threads
    …                                                                        // memory barrier and use the results
}
```

Wave States



Initial state, control=1, counter=0,Target=0

Main thread ‘flips’ control, control matches worker target => workers start

Workers ‘flipped’ target, incremented counter => result ready

Main thread reset counter, ‘flipped’ control to match new target => workers start

Workers ‘flipped’ target, incremented counter => result ready

Main thread reset counter, ‘flipped’ control to match new target => workers start

Wave Observations

- Alternating the state of the ctrl avoids a ‘reset’ with additional sync
- Gcc-intrinsics tend to constitute a full memory barrier at least on the variable they operate on, avoids unnecessary syncs on platforms on which this is true
- Tradeoff
  - AtomicInc serializes the additions, could set independent variables
  - Independent variables serialize testing in the calling process
  - Independent variables need padding to avoid coherence issues
  - There is an ‘explicit’ overhead with speedup ‘1’ in either case!

The Moral of the Story

Working with a quantitative goal is important

- Differential analysis against a target enables identification of problems
  - Performance bottlenecks
  - Performance bugs
  - Real bugs!
    - E.g. when something is faster/better than predicted
    - In machine learning, ‘dropping’ a significant part of the training data can on specific test cases converge to a good or even better solution
- Iterative process
  - Characterize simplified case
  - Model based on unit-characterization of pieces
  - Refine model/base case as more information/functionality becomes available