THE CYPRUS INSTITUTE
RESEARCH • TECHNOLOGY • INNOVATION

Scaling up: Multi-GPU Parallelism for AI Models

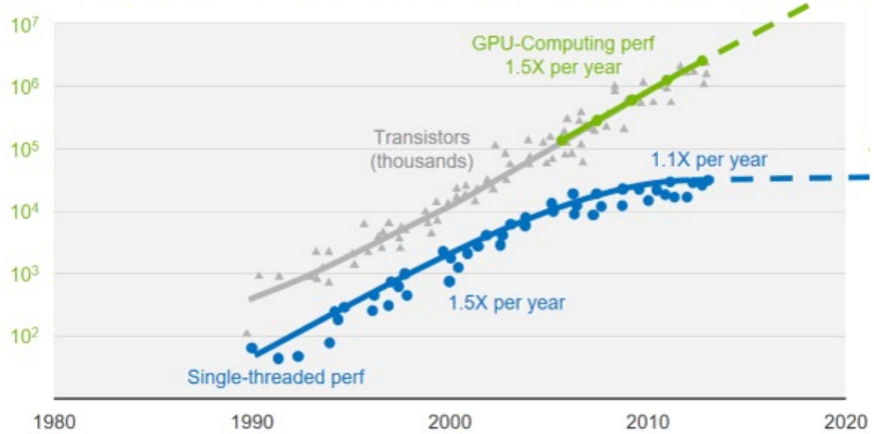
Spiros Millas

Research Engineer CaSToRC, CyI

Motivations

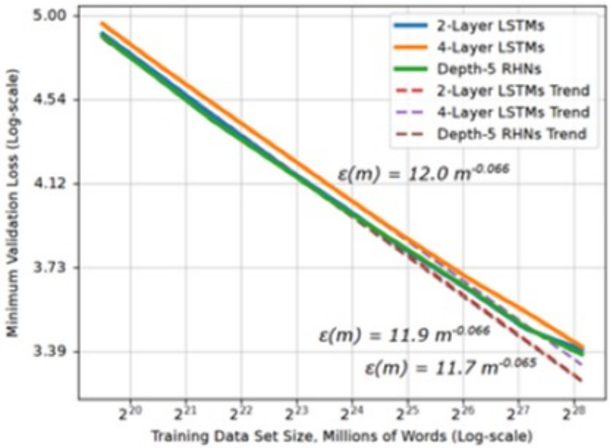
Why should we train on multiple GPUs

- CPU computational performance has not increased substantially during the last decade
- GPU performance has been exponentially increasing
- Larger and complex models are more accurate, but require much more computational power



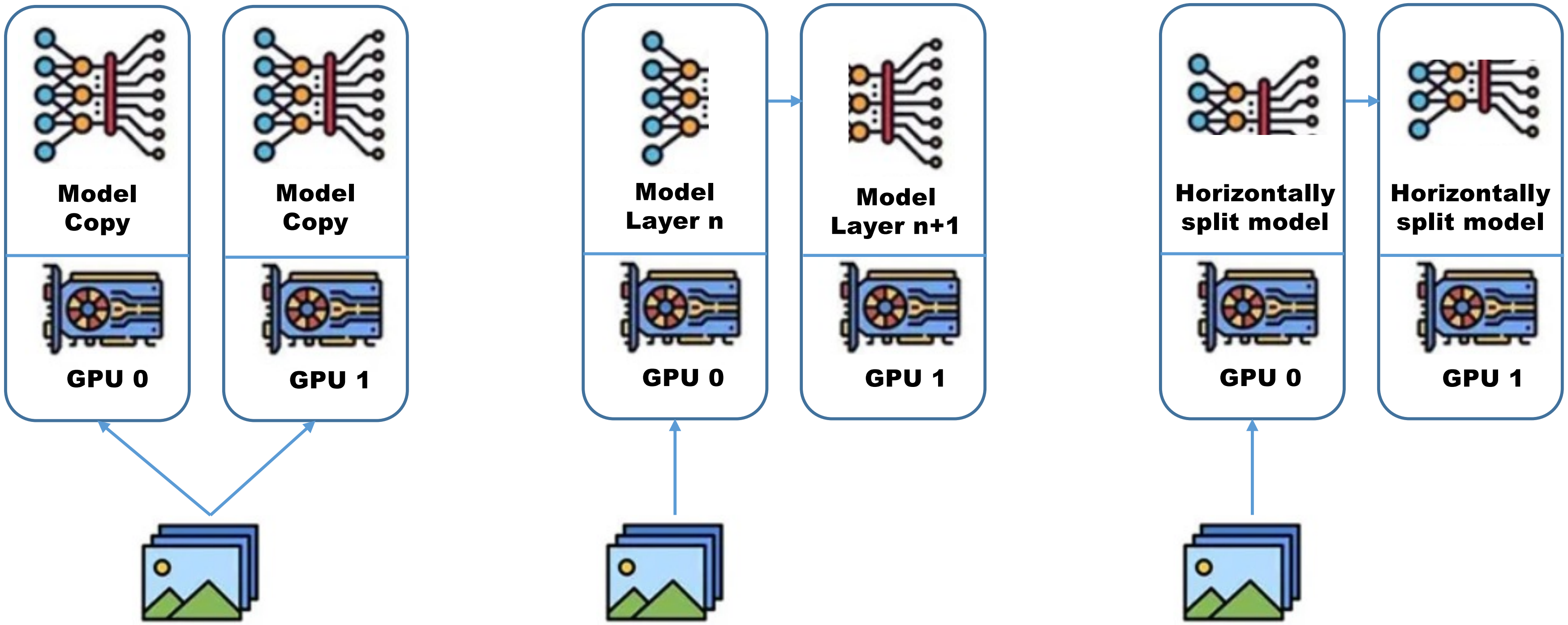
When?

- **Model is too large and Complex**:
  If your model doesn't fit into the memory of a single GPU or runs extremely slowly on just one, scaling out to multiple GPUs can help.
- **Dataset Scale**:
  Very large datasets benefit from more GPUs, as you can process more samples in parallel and speed up training convergence.



Types of Parallelism

Data Parallelism | Pipeline Parallelism | Tensor Parallelism

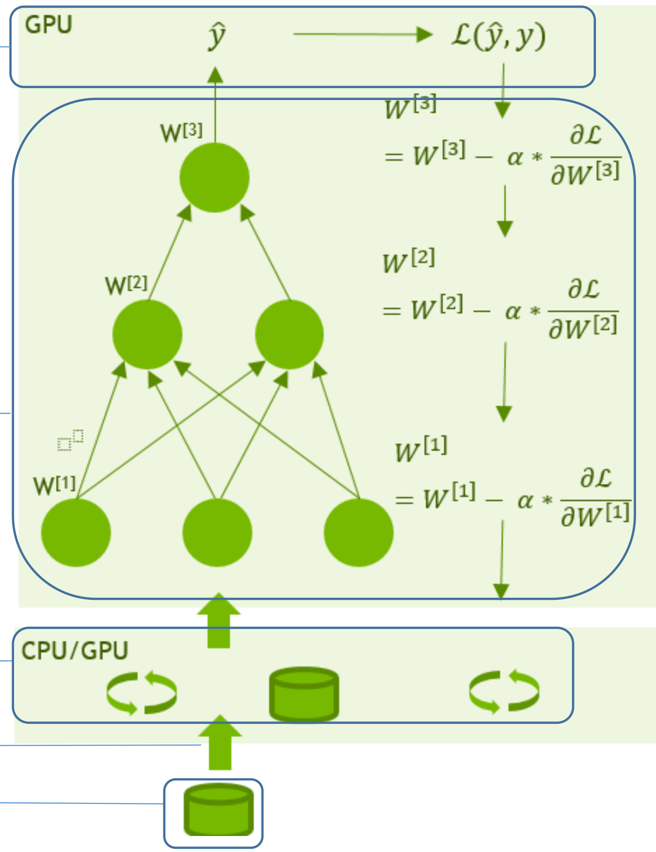


Data Parallelism

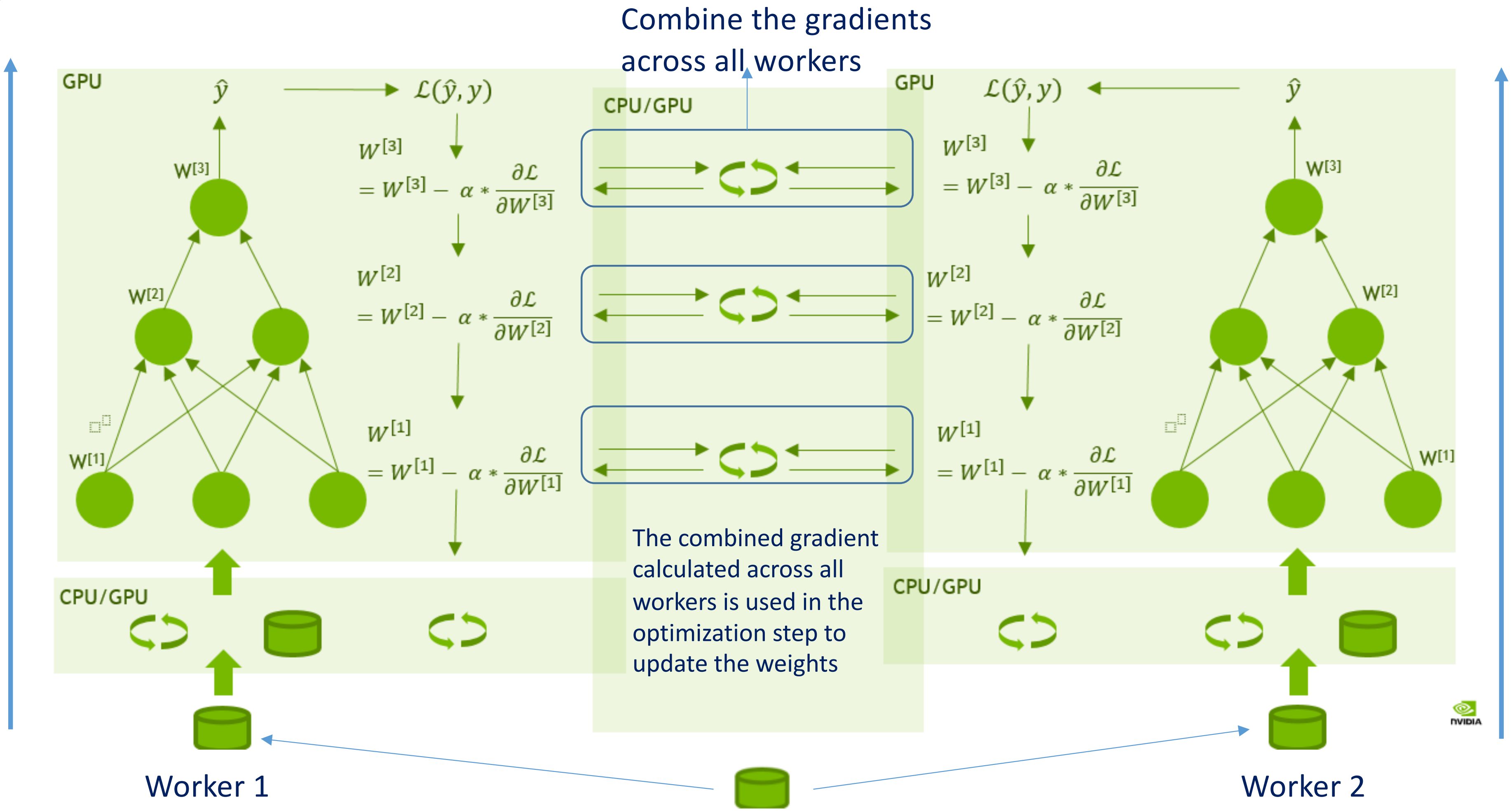
8. Execute optimization step and update the weights
7. Back-propagate through all the layers
6. Calculate the output and loss

5. Calculate the activations for all layers
4. Transport the data into a model

3. Preprocess and queue the data
2. Transport the data
1. Read the data



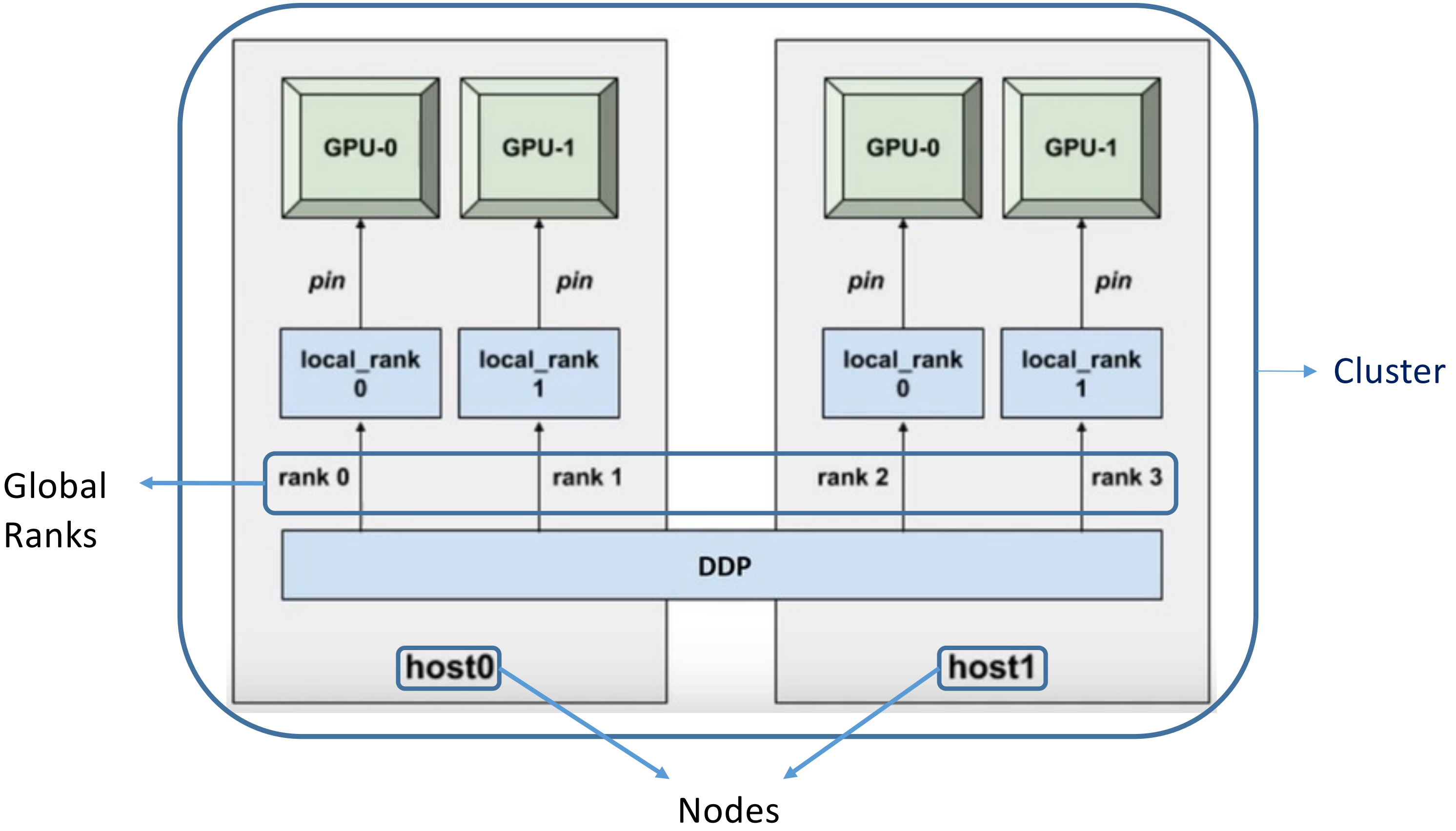
Data Parallelism



Introduction into Data Distributed Parallel (DDP)

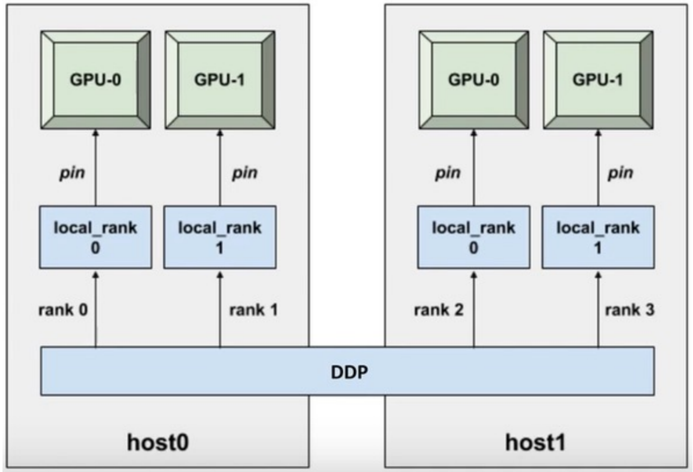
Distributed Data Parallel (DDP) in PyTorch implements data parallelism at the module level, running across multiple machines. Each process should has a single DDP instance, and DDP uses collective communications from the package to synchronize gradients and buffers.

Introduction into Data Distributed Parallel (DDP)



Introduction into Data Distributed Parallel (DDP)

```
if __name__ == '__main__':
    torch.multiprocessing.spawn(worker, nprocs=args.num_gpus, args=(args,))

def worker(local_rank, args):
    global_rank = args.node_id * args.num_gpus + local_rank
    dist.init_process_group(
    backend='nccl',
    world_size=WORLD_SIZE,
    rank=global_rank
    )

    download = True if local_rank == 0 else False
    if local_rank == 0:
        train_set = torchvision.datasets.FashionMNIST("./data", download=download, transform=
                                                      transforms.Compose([transforms.ToTensor()]))
        test_set = torchvision.datasets.FashionMNIST("./data", download=download, train=False, transform=
                                                     transforms.Compose([transforms.ToTensor()]))
    dist.barrier()

    if local_rank != 0:
        train_set = torchvision.datasets.FashionMNIST("./data", download=download, transform=
                                                      transforms.Compose([transforms.ToTensor()]))
        test_set = torchvision.datasets.FashionMNIST("./data", download=download, train=False, transform=
                                                     transforms.Compose([transforms.ToTensor()]))

   device = torch.device("cuda:" + str(local_rank) if torch.cuda.is_available() else "cpu")

   model = model.to(device)
   model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(WideResNet(num_classes)).to(device)
   model = nn.parallel.DistributedDataParallel(model, device_ids=[local_rank])
```




```
torch.distributed.all_reduce(v_accuracy, op=dist.ReduceOp.AVG)
torch.distributed.all_reduce(v_loss, op=dist.ReduceOp.AVG)
val_accuracy.append(v_accuracy)
```

Profiling Multi-GPU Applications

- Importance of profiling in Multi-GPU Training:
  - Performance Optimization: Ensures efficient utilization of multiple GPUs
  - Bottleneck Identification: Detects communication overhead, memory constraints
  - Resource management: Helps balance workloads and minimizing idle times across GPUs

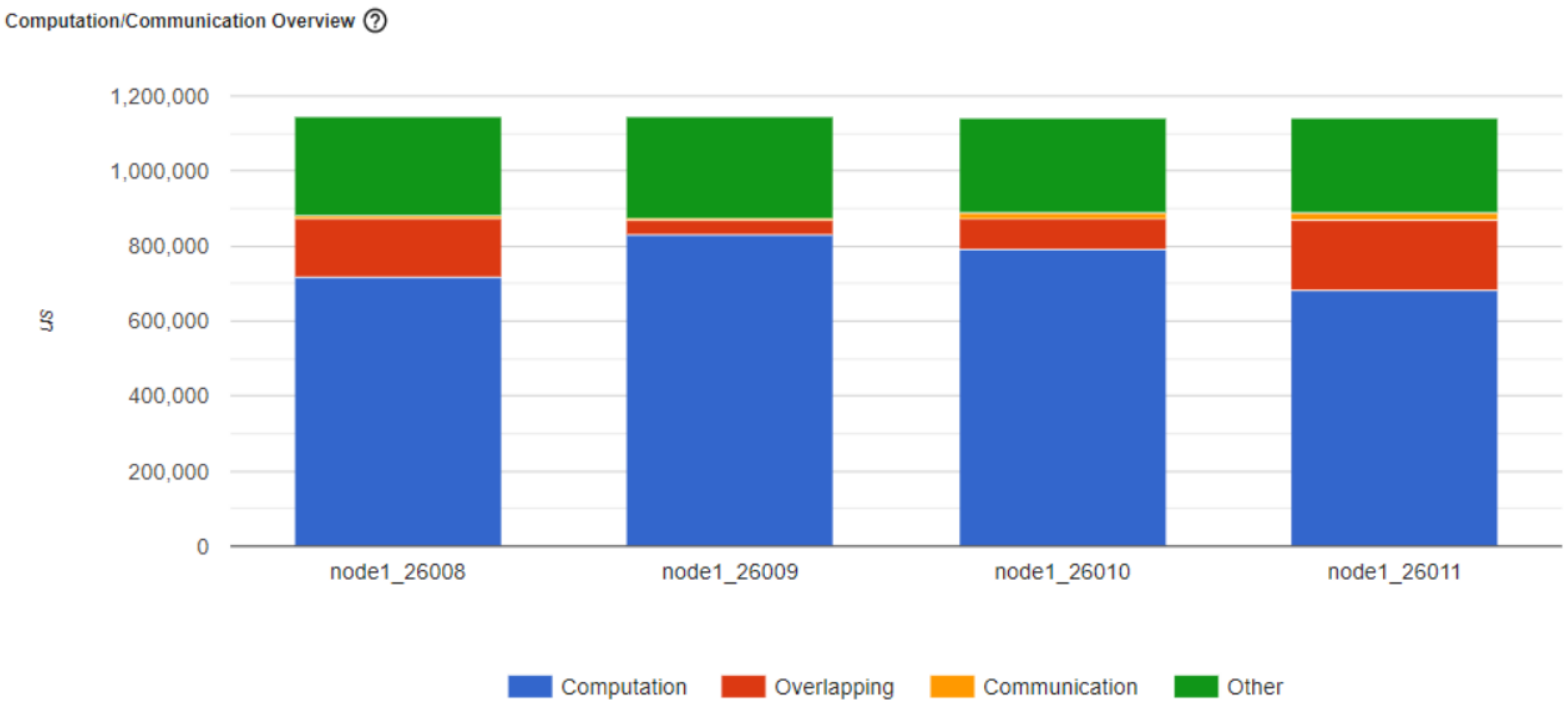
Key Profiling metrics for multi GPU

- GPU Utilization
  - Ensure GPU memory is neither overloaded or underused
  - Per GPU utilization should be roughly equal

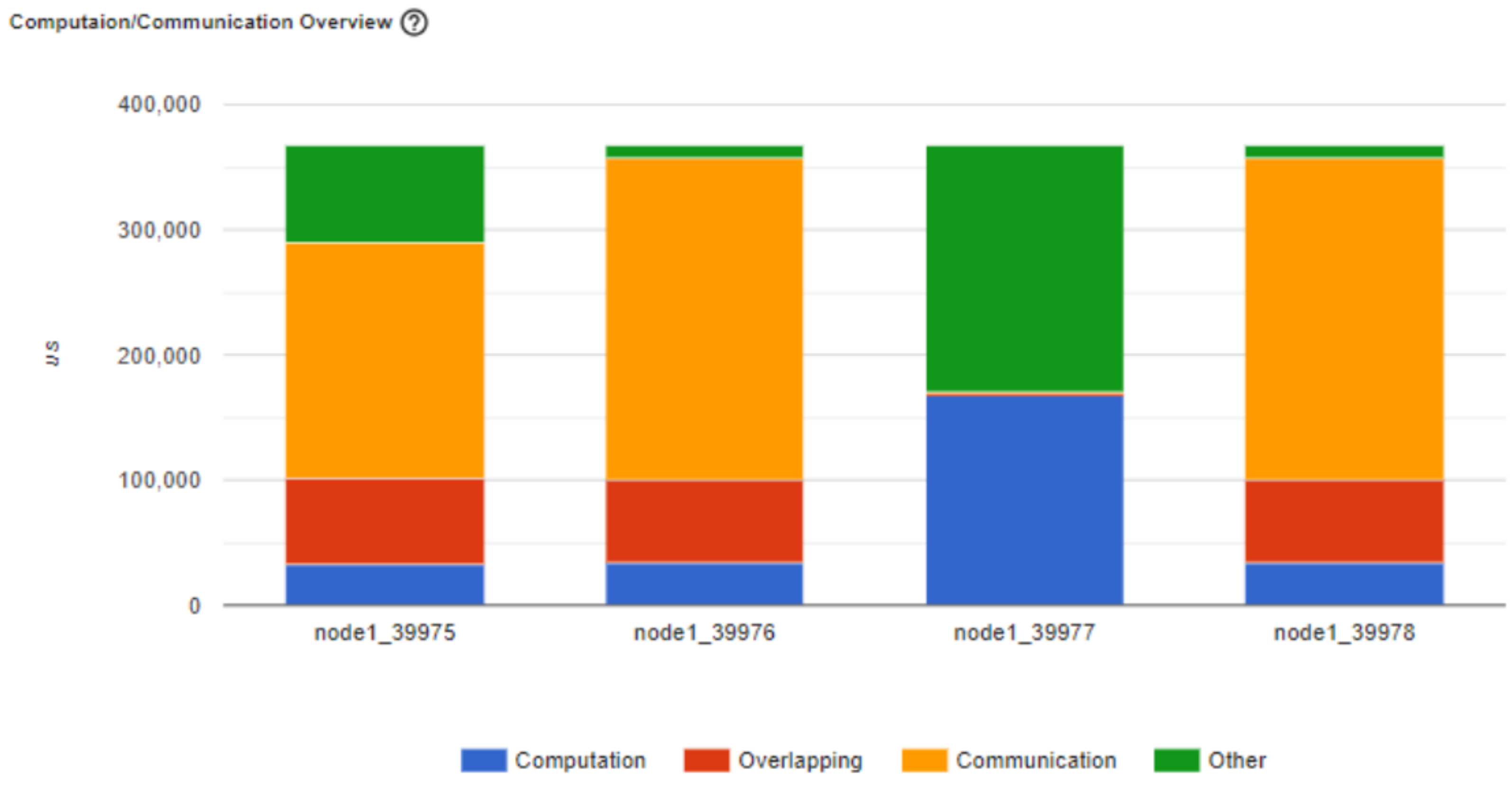
- Communication / Synchronization
  - GPU to GPU communication overhead
  - Time spent waiting for all GPUs to reach the same point in training steps
  - Time it takes to transfer data.

- Load imbalances
  - One or more GPUs performing slower or handling more data than others, creating "stragglers" that delay the entire process.
  - Results in uneven work distribution, reduced efficiency, and suboptimal scaling performance.

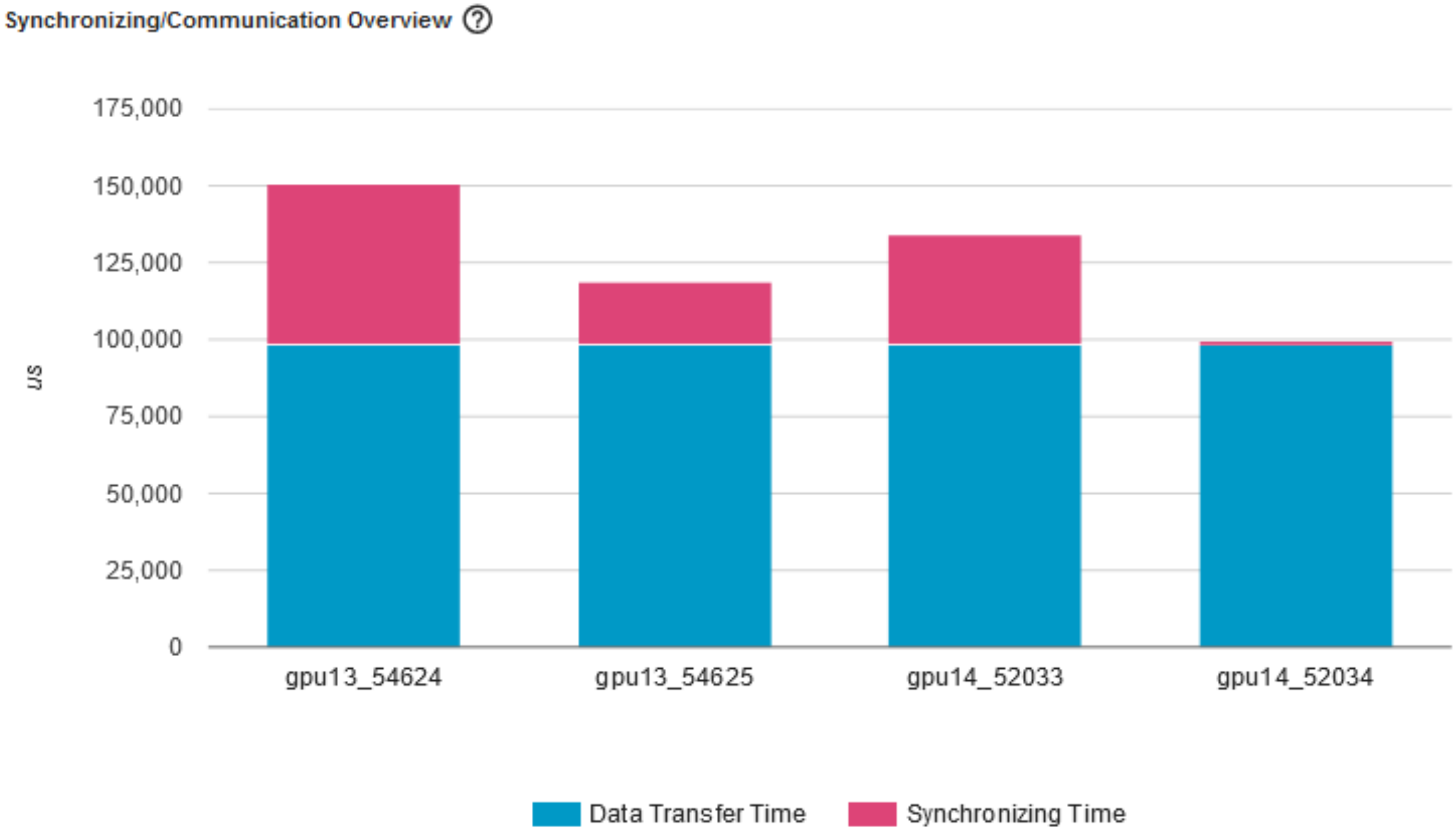
Computation/Communication



Computation/Communication



Synchronization/Data Transfer



Setting Up Distributed Training

```python
def setup_slurm_distributed():
    """Initialize distributed training for SLURM environment"""
    local_rank = int(os.environ["SLURM_LOCALID"])
    global_rank = int(os.environ["SLURM_PROCID"])
    world_size = int(os.environ["WORLD_SIZE"])

    dist.init_process_group(
        backend="nccl",
        init_method="env://",
        world_size=world_size,
        rank=global_rank
    )
    torch.cuda.set_device(local_rank)

    return local_rank, global_rank, world_size
```

Setting Up Distributed Training

```python
class EnhancedEfficientNet(nn.Module):
    def __init__(self, num_classes=100):
        super().__init__()
        self.efficientnet = efficientnet_v2_l(pretrained=False)
        self.classifier = nn.Sequential( ···
        )
        self.efficientnet.classifier = self.classifier
        self.aux_classifier = nn.Sequential( ···
        )

    def forward(self, x):
        features = self.efficientnet.features(x)
        main_out = self.efficientnet.classifier(
            self.efficientnet.avgpool(features).flatten(1)
        )
        aux_out = self.aux_classifier(features)
        return main_out + 0.3 * aux_out
```

```python
model = EnhancedEfficientNet(num_classes=args.num_classes).to(device)
model = DDP(model, device_ids=[local_rank], output_device=local_rank)
```

Setting Up Distributed Training

```python
def setup_data(batch_size, rank, world_size):
    if rank == 0:
        trainset = torchvision.datasets.CIFAR100(
            root='./data',
            train=True,
            download=True,
            transform=transform
        )
    else:
        trainset = torchvision.datasets.CIFAR100(
            root='./data',
            train=True,
            download=False,
            transform=transform
        )

    dist.barrier()
    sampler = DistributedSampler(trainset, num_replicas=world_size, rank=rank, shuffle=True)
    trainloader = DataLoader(
        trainset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=4,
        pin_memory=True,
        sampler=sampler
    )
    return trainloader
```

Setting Up Distributed Training

```
profiler = torch.profiler.profile(
    schedule=torch.profiler.schedule(
        wait=args.wait,
        warmup=args.warmup,
        active=args.active_steps,
        repeat=1
    ),
    on_trace_ready=torch.profiler.tensorboard_trace_handler(log_dir),
    record_shapes=True,
    profile_memory=True,
    with_stack=True,
    with_modules=True
)
```