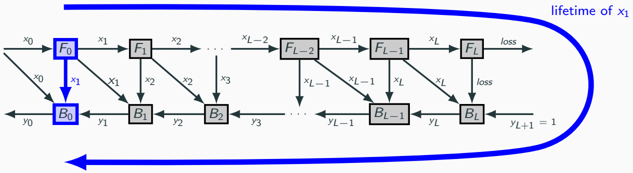
Pipelined Model Parallelism: Complexity Results and Memory Considerations
Topal Working Group – May 6, 2021

Olivier Beaumont, Lionel Eyraud-Dubois, Alena Shilova

Context

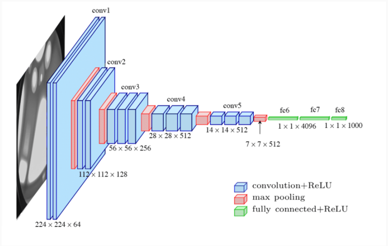
- During the first half of Alena's thesis,
  - much attention was paid to memory problems during training

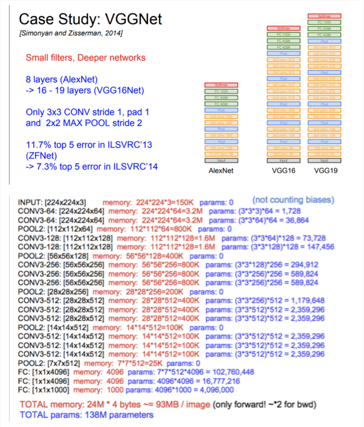


  - linearized networks (we continue to use this assumption)
  - x_1 must be kept in memory for a very long time, so
    - either we delete it and recompute it later (re-materialization)
    - or we store it in the (large) memory of the CPU (offloading)
- Question: what about **parallelism** ?
  - With Linear Algebra applications
    - overall memory is (more or less) distributed among nodes (Loomis Whitney)
  - With Training
    - **2 sources** of memory needs (network weights and activations)
    - network weights: updated after each mini-batch (say every 32 trained images)
    - activations: 2 dependences, 1 very short, 1 long (once per image and per layer)

What do networks look like ?

- Ex: VGG16:




Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13
(ZFNet)
-> 7.3% top 5 error in ILSVRC'14

AlexNet
VGG16
VGG19

INPUT: [224x224x3] memory: 224*224*3=150K params: 0 (not counting biases)
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

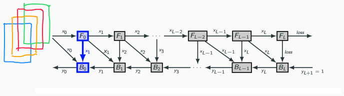
- The memory to store activations is not negligible at all
  - the number of weights increases with depth
  - the size of activations becomes smaller with depth

Data Parallelism: Mini Batch Parallelism

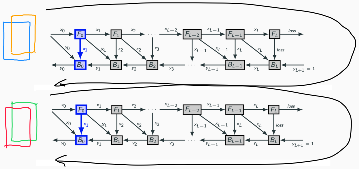
With N (identical) GPUs

- Each GPU can perform SGD with a batch size of B
- We train in parallel a batch size of size BN (B on each GPU)
- Each resource computes a gradient (size of all weights)
- MPI `Allreduce` is used to compute a global gradient
- A new set of NB images is used for training...

Sequential



Parallel



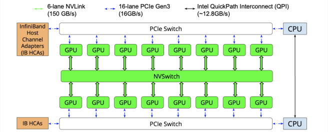
Limitations

- there is a strict barrier at the end of each `Allreduce` operation
- thus, MPI `Allreduce` is expensive when N becomes large

Data Parallelism: Does it work well ?

- the weights are communicated in collective operations
- activations (transformed images) are always kept local
- performance strongly depends on the size of the network

  the speed of the interconnect
- **YES:** DK Panda's group on Summit:



6-lane NVLink (150 GB/s)
16-lane PCIe Gen3 (16GB/s)
Intel QuickPath Interconnect (QPI) (~12.8GB/s)
InfiniBand Host Channel Adapters (IB HCAs)
PCIe Switch
CPU
GPU
NVSwitch
IB HCAs

Figure 1: Hardware configuration of NVIDIA DGX2 system with cutting-edge GPU and interconnect architectures

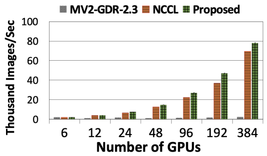
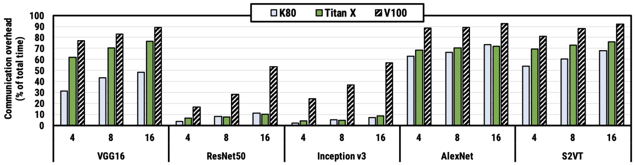


Figure 16: ResNet-50 Training using PyTorch on Summit system (6 GPUs/node)[1]

- **NO:** Pipedream group on AWS:



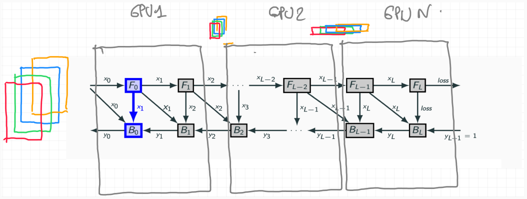
Model Parallelism

General Idea

- distribute the network itself onto several nodes (distributed memory)
- Advantages: distribute **both** weights and activations
- Each GPU
  - stores only the weights for its own layers
  - stores only the activations corresponding to these layers
- Data Parallelism: communication of network weights
- Model Parallelism: communication of **activations**

Model Parallelism: what can be expected ?

- If we assume that the graph is a chain:



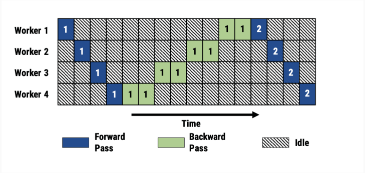
- Characteristics (homogeneous layers)
  - Work in $\Theta(2L)$
  - Critical Path in $\Theta(2L)$
  - trivial sequential solution in $\Theta(2L)$...
- There is not much to expect: LowerBound $= \max(CP, W/P)$!
- **But:** Possibility to distribute memory
- **More importantly,** possibility to use pipelining

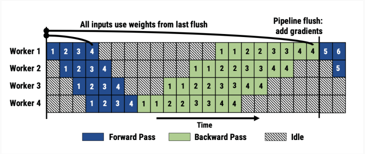
  insert one new image in the pipeline every x ms

Model Parallelism – Pipelined Versions

Source: *PipeDream: Generalized Pipeline Parallelism for DNN Training*, Deepak Narayanan et al., SOSP'19

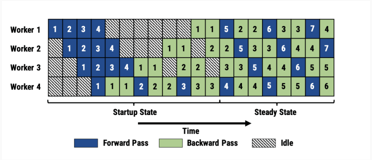
- Training task graph is very sequential by nature





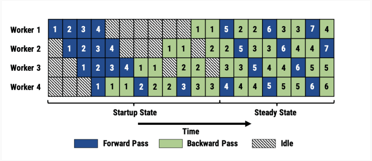
All inputs use weights from last flush
Pipeline flush: add gradients
Worker 1
Worker 2
Worker 3
Worker 4
Time
Forward Pass
Backward Pass
Idle

- How to increase resource utilization? by splitting the work into smaller pieces (micro batches) and use pipelining
- Ok, there is still a lot of idle time... keep several copies of the weights (to make consistent updates)



Model Parallelism – Pipedream

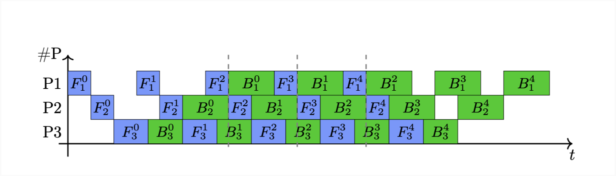
Pipedream is extremely nice, but many unclear issues



- About Memory
  - You need to keep the weights used for Forward(I_i) until Backward(I_i)
    - I_1, I_2, I_3, I_4 with the same, but I_5, I_6 and I_7 correspond to different model weights... if you update weights immediately after backward
  - You need to keep several activations simultaneously
    - B_{k-4} is performed immediately before F_k, so F_{k-4}, F_{k-3}, F_{k-2}, F_{k-1} must reside in memory at the same time
- Concerning scheduling, Pipedream says
  - Just inject several images in the pipeline (here 4)
  - alternate backward and forward (in the natural order)
  - ...and it will work !

Model Parallelism – Periodic Schedules – Memory for Models

Pipedream builds **1-periodic** schedules (1 type of each task in each period)

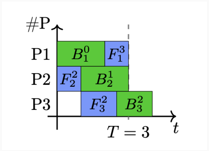


- About Memory required for the models
  - You need to keep the weights used for Forward(I_i) until Backward(I_i)
    - In fact you can always keep only 2 versions of the weights
  - For instance,
    - 1 model M0 used for F_1^0, F_1^1, F_1^2
    - accumulate gradients in B corresponding to B_1^0, B_1^1, B_1^2
    - another model M1 for F_1^3, F_1^4, F_1^5
    - update M0 after B_1^2, reset B
    - accumulate gradients in B corresponding to B_1^3, B_1^4, B_1^5
    - update M1 after B_1^5, reset B
    - and so on...
- We need to keep only a small (say 3) number of models.

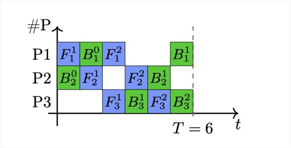
Model Parallelism – Periodic Schedules – Memory for Activations

Different types of Periodic Schedules

1-periodic:



2-periodic:



About Memory required for the activations

- For layer l, $\text{NCA}_l = \max_t \#F_l(t' < t) - \#B_l(t' < t)$
  where $\#F_l(t' < t)$ counts the number of F_l
- For periodic schedules, looking at shifts in the schedule is enough

Valid Periodic Schedule

- operations in the right order: F_i^j ends before F_i^{j+1} and B_i^j
  F_i^j ends before F_{i+1}^j
- Overall memory not exceeded (can be computed from the schedule)

Well formulated optimization problem

- When restricting the search to periodic schedules

Classical (Pipedream's) Assumptions

Two implicit assumptions

- Consider only **1-periodic** schedules (more simplicity)
- Consider only **contiguous** allocations
  - Contiguous: P_0 receives $L_0, \ldots, L_i$, P_1 receives $L_{i+1}, \ldots, L_j$,...
  - **Intuition:** more stages mean larger index shift on P_0

Questioning these assumptions

- they are rather intuitive
- enable to find easily good allocations (layers to processors) and good (1-periodic) schedules
- **Our Paper**: what influence on the quality (throughput) of the schedules?

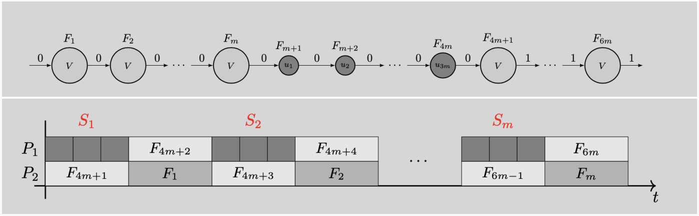
Model Parallelism – Complexity Results

General Problem: Allocation & Schedule

- **Inputs:** weights, activation sizes, processing times (F & B)
  GPU memories and target throughput T
- **Goal:** Find an allocation and a periodic schedule with throughput $\geq T$
- **NP Complete** in the strong sense

Scheduling Only Problem

- Even if the allocation of layers to GPUs is given
- and we only look for a valid periodic schedule with throughput $\geq T$
- The problem **remains** NP Complete in the strong sense



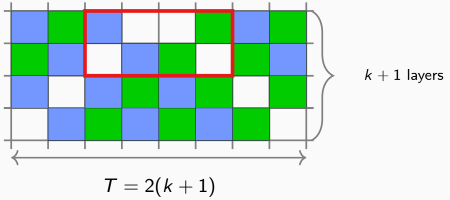
Model Parallelism – Finding Good Schedules

Positive Result: optimal 1-periodic schedule

- Given an allocation and a target throughput, it is possible to find an optimal 1-Periodic Schedule that **minimizes the memory needs**
- The algorithm is non-trivial, but computationally cheap

Negative Result: 1-periodic is not always enough

- There exists allocations for which no j-periodic schedule with $j < k$ is able to provide the same throughput as a k-periodic schedule.



- $\forall j, k$ the performance ratio is as large as $\left(1 + \frac{1}{j}\right) / \left(1 + \frac{1}{k}\right)$
- Restricting the search to 1-periodic schedules **hinders throughput**

Model Parallelism – Contiguous vs non-Contiguous Allocations

How efficient are **contiguous allocations**, where each processor is in charge of a sequence of contiguous layers?

Without memory constraints

- **Positive result:** The best *contiguous* throughput is at most **twice smaller** than the best *non-contiguous* throughput
- **Negative result:** $\forall k$, there are cases where the ratio is $2 - 1/k$

With memory constraints, only negative results

- There are cases where non-contiguous allocations are actually needed (*i.e* where contiguous allocations fail under memory constraint)
- If both non-contiguous and contiguous allocations exist, then the throughput with non-contiguous allocations can be **arbitrarily larger**.

Conclusion

Pipedream has in theory many drawbacks...

- uses **1-periodic** and **contiguous** allocations, both can hinder throughput
- worse, the solution might not fulfill memory constraints

But it is very hard to improve it!

- **Integer Linear Programming** based solution
  - with a rather complete model
  - limited to (very) small problems
- **Dynamic Programming** based solution
  - separates allocation and scheduling issues (known to be a bad idea)
  - looks for solutions in a larger class (1 proc with an arbitrary set of stages)
  - still, it is expected to improve performance

In practice, the solutions of Pipedream are

- very simple
- easy to implement at runtime (finding which task to perform next is trivial)

The complete problem turns out to be very complicated!

- **Simplifying assumptions** are needed

Perspectives

Pros and Cons of Model parallelism

- **cheap** in terms of communications
- **memory hungry**, **might be combined** with re-materialization / offloading
- **limited** in terms of expected scalability (not more GPUs than layers)
- deeper pipelines generate **large** memory needs

Should be combined with other type of parallelisms, typically Data

- Model parallelism defines groups of layers, Data Parallelism inside groups
- Collective communications take place in **smaller** groups
- Each image will pass through one GPU from each group, **can be dynamic**

Modeling both data and model parallelisms will be difficult

- It is hard the find the right simplifying assumptions

Practical solutions already exist

- Awan, Ammar Ahmad, et al. *HyPar-Flow: Exploiting MPI and Keras for Scalable Hybrid-Parallel DNN Training with TensorFlow*. International Conference on High Performance Computing. Springer, Cham, 2020.