IBM Research – Deep Learning

Large Data Flow Graphs in Limited GPU Memory

Geert Janssen
geert@us.ibm.com

Vladimir Zolotov
zolotov@us.ibm.com

Tung D. Le
tung@jp.ibm.com

Motivation & Intentions

- Deep Learning applications traditionally heavily make use of GPUs.
- A GPU has limited memory; typical Nvidia v100 GPUs have 16GB, newer ones come with 32GB.
- A host server system typically has a magnitude larger RAM memory; 512GB up to 1TB is not uncommon.
- There is a trend of using larger models for deep learning.
  Newer models have 10s to 100s of layers.
- Accuracy of the result is important for many applications.
  This requires high resolution samples and deep networks.

- Utilize GPU memory to the max.
- Be able to handle large models with high resolution samples.
- Provide an automated solution with little or no intrusion on existing training tool and flow.

Agenda for rest of talk

- Related work
- Graph modifications
- Timing analysis theory
- Candidate selection
- Serialization (with special case: single-node-at-a-time)
- Experimental results
- Conclusion/summary

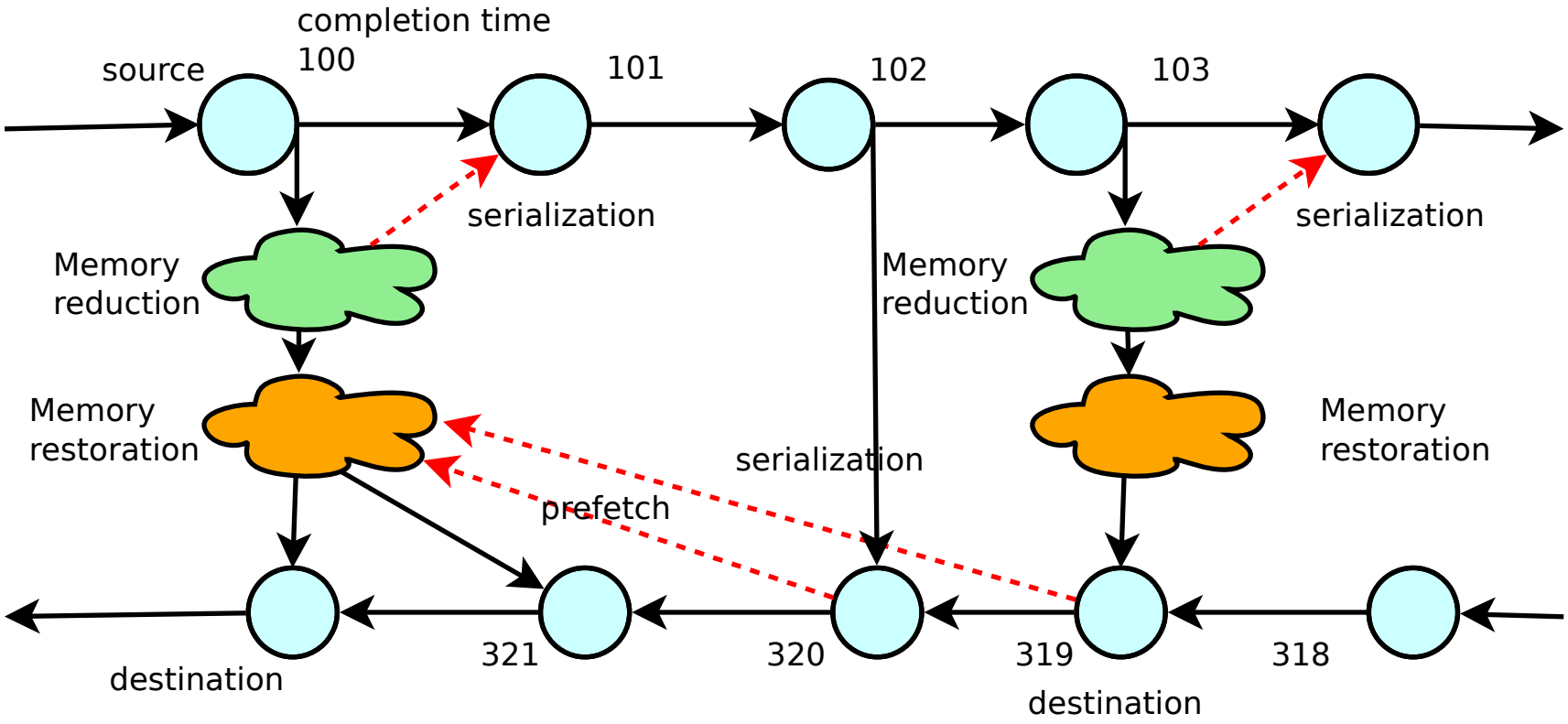
Related work

1. Resort to using a so-called Unified Memory approach.
2. Break up the model in pieces and dedicate separate devices to processing each piece. This is called model parallelism.
3. Discard tensor data when memory becomes tight and later recompute that data when needed.
4. Use the CPU's memory as a swap space for GPU memory. This is the approach we propose and follow in this paper.
   - It is not a completely new approach but we think our take on it is different and innovative.
   - In particular we apply the theory of timing analysis of digital circuits to the data flow graph to heuristically find good positions in the graph for insertion of reduction/restoration sub graphs.

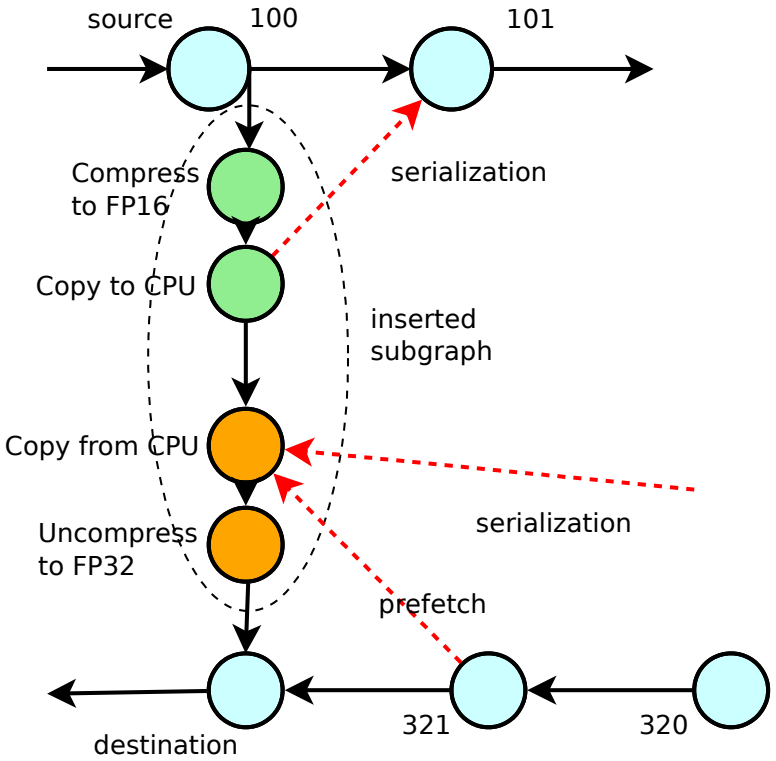
Graph modifications

- Our solution is implemented by modifications to the data-flow graph structure. We do not modify the program (TensorFlow) that does the execution of the graph.
- We employ 3 possible sub graph structures to be inserted for purpose of saving memory. Each consists of a reduction part and a complementary restoration part. We call such a combination a conservation sub graph.
    1. a swap-out, swap-in pair of nodes
    2. a compress, uncompress pair nodes
    3. a combination of 1 and 2

Memory conservation sub graph insertion

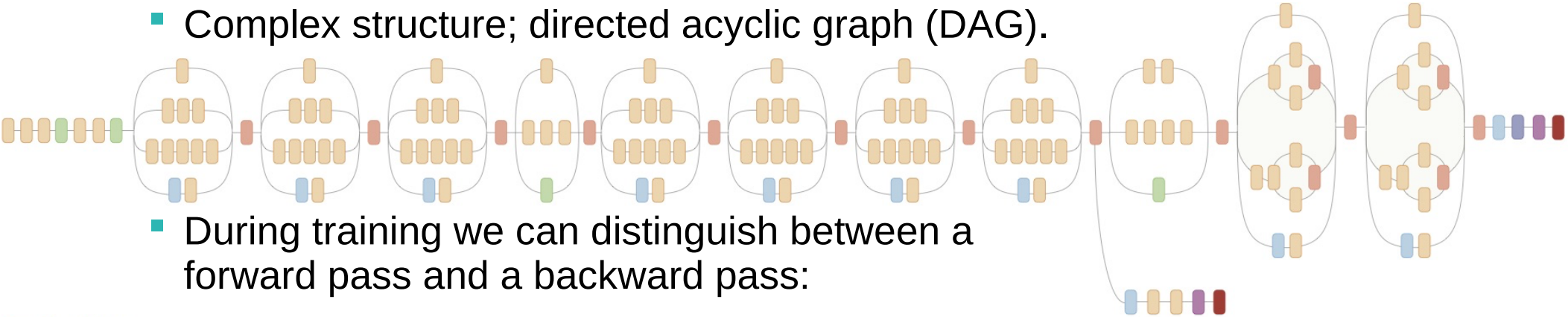


Combined swap-out and compression



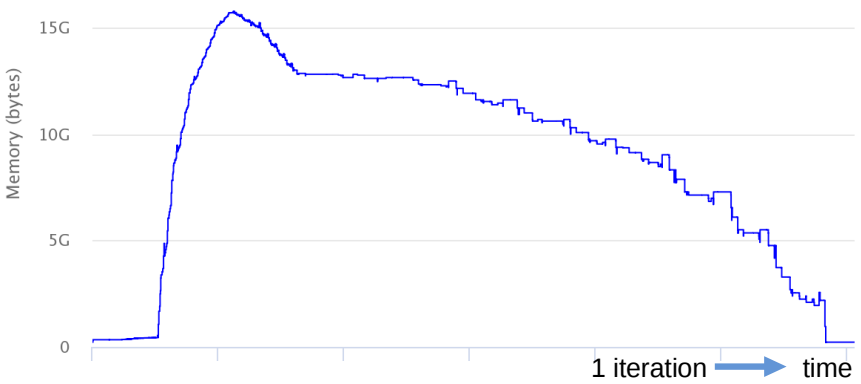
Typical neural network model/graph

- Complex structure; directed acyclic graph (DAG).

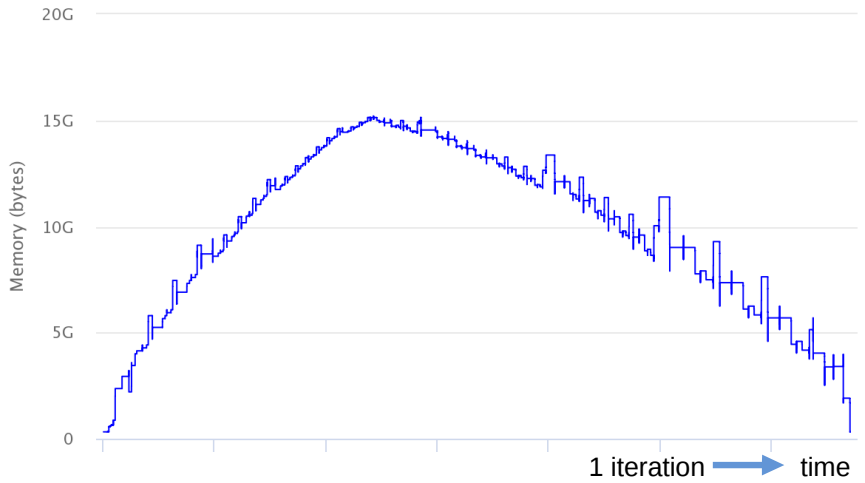


- During training we can distinguish between a forward pass and a backward pass:
- During the forward pass activation tensors are computed from inputs and weights and biases.
- During the backward pass gradients are computed and applied to update the weights and biases.
- Activation tensors computed during the forward phase are needed to compute the gradients in the backward phase.

Memory profiles: 1) resnet50 maximum batch size 191; 2) half precision compression: batch size 357



A typical trend is seen here: more and more memory is allocated as execution proceeds, until some peak is reached. Then memory is often released in reverse order: most recent allocated memory is released first. This GPU offers 16GB of memory; the chosen batch size is the maximum that will fit: the peak is over 15GB.

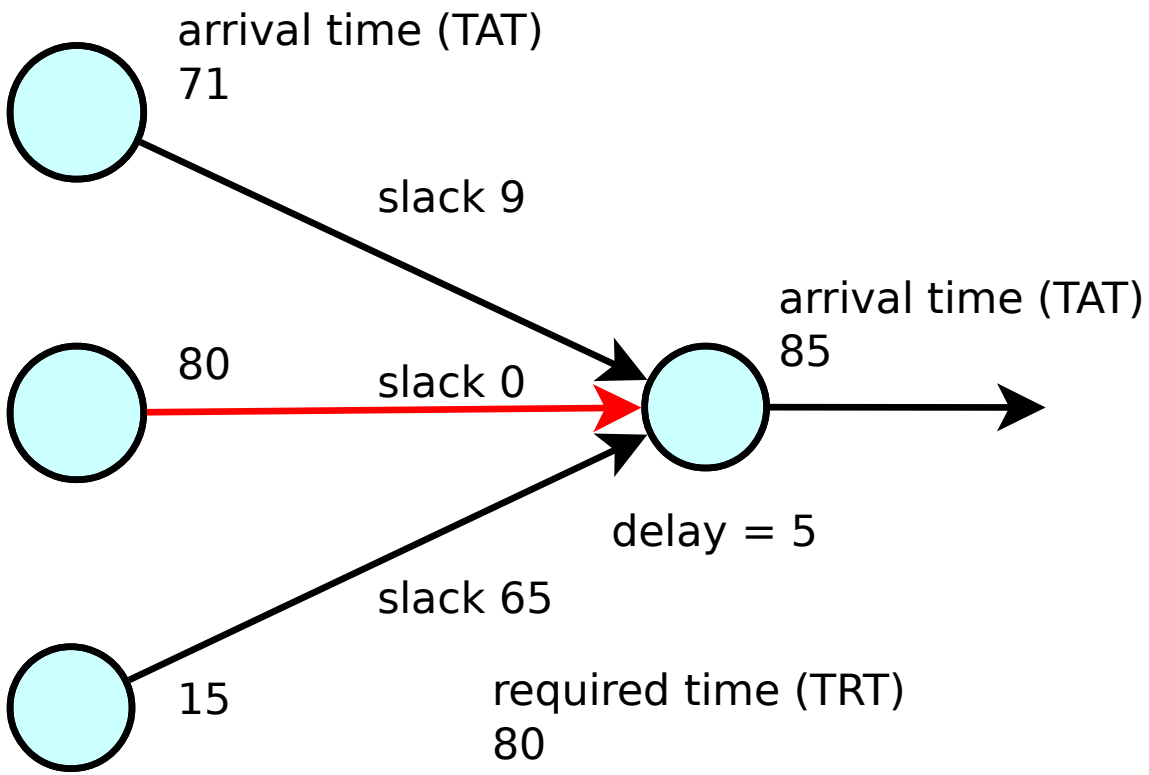


Large tensors are compressed from 32-bit floating point numbers to 16-bit floating point, thereby saving half the memory to store them. The effect is that the batch size can almost be doubled before the GPU is “full”.

Timing analysis (1)

- In digital circuit design, (static) timing analysis computes required times, arrival times and slack for all signals.
- Analogous, in a data flow graph we have operational nodes that compute output tensors given the presence of input tensors. Like signals, the tensor data is propagated through the graph via its hyperedges.
- We define the arrival time (TAT) of a node's output tensor as the time point when computation of the tensor is completed and the new value is available to other nodes for consumption. We assume that all outputs of a given node are computed at the same time.
- We define the required time (TRT) for a node's input as the time point when the tensor value at that input is expected to be available for the execution of the node. It is the same for all inputs of a given node.

Arrival Time, Required Time, Slack (in DAG)



Timing analysis (2)

- Input tensors must all be available before computation starts, and hence the TRT must not be earlier than the latest TAT. Not allowing for any wasted time, we set the TRT to the maximum of input TATs.
- It follows: *TAT(node) = TRT(node) + delay(node)*.
- It also allows us to introduce the notion of slack for any given input tensor: *slack(input) = TRT(node) – TAT(input)*.
- slack expresses the amount of time the particular tensor could be delayed without affecting the overall computation time.
- This is the principal criterion we use to select insertion points in the graph.
- Note that many different delay models can be used. We decided on a simple unit-delay model.

Candidate selection

- Tensor data that reside in GPU memory for a long time are obvious candidates for conservation. The tensor slack is an appropriate measure for the life span of the tensor data. The larger the slack value, the longer the data occupies precious GPU memory. As a secondary criterion we use the tensor size (in bytes).
- We offer a user the following options:
  - a lower-bounded threshold for the slack value
  - a lower-bounded threshold for the size
  - the maximum number of actual insertions to apply out of all candidates

Serialization (1)

- A scheduler might issue multiple node evaluations simultaneously. This implies several time-overlapping tensor memory allocations. Of course exploiting parallelism improves run-time. But if we are strapped for memory, we might agree to a trade-off.
- It is easy to enforce serial execution by adding control dependency edges. These edges do not carry data. A node with incoming control dependency cannot execute before the source node of the edge has executed.
- Mind that many neural networks tend to have a rather linear structure and hence many nodes are executed serially by default.

Serialization (2)

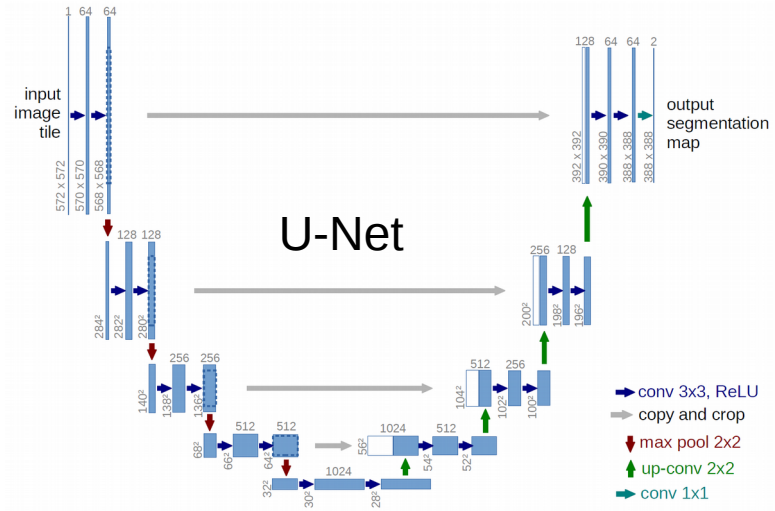
- Full serialization, i.e., using a linear order, is easy to achieve by computing any topological order and then adding control dependencies for any missing edges.
- We adopt a partial serialization solution based on some heuristics:
  1. Prevent all nodes using a swapped-out (or compressed) tensor from directly executing.
  2. Serialize all restoration actions, i.e., if possible, make sure no 2 consecutively restored tensors occupy memory at the same time.

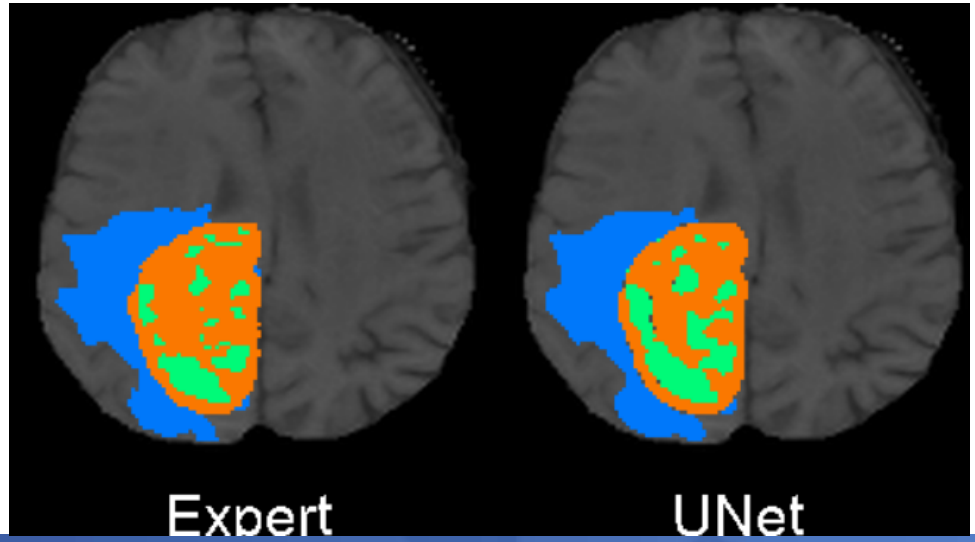
Experiments

- We don't have an array of successively larger models.
  Instead we test our approach by simply changing the batch size.
- Neural nets: 3D-Unet, ResNet50 and ResNet152.
- IBM Power8 with P100 GPUs, 16 GiB device memory and NVLink.

A really large model: 3D U-Net

- Fabian Isensee Keras/TensorFlow model
- BraTS 2017 dataset: 192 x 192 x 192 input images
- With batch size 1, already too big to fit 16GB GPU!
- Our methods raise this to batch size 4





Throughput in images/second

P8 / x86	resnet50 (v1)			
	Throughput (ImageNet images/sec)			
BatchSize	Baseline Without LMS	Swap/Identity only	HalfComp + Swapout + SwapInHalfPrec[tmp]	HalfComp only
191	245 / 235			
256		191 / 160	212 / 212	232 / 229
357		165 / 115	190 / 181	215 / 215
775		130 / 74	165 / 120	
1024		117 / 65		

TensorFlow 1.8
Nvidia P100 GPU

- Applied techniques:
  - Swapping by identity nodes placed on CPU or custom copy nodes
  - Tensor compression by FP32 → FP16
  - Combination of the above

Conclusions

- Automatically modify the data flow graph.
- 3 variants: swap-out, compress, combination.
- Partial serialization.
- Approach based on formal technique of timing analysis.
- Experiments show decent improvements.

Acknowledgments

- Youjie Li (Summer intern)
  - for implementing the custom TF operators
- Samuel Matzek
  - for advice on how to set the TensorFlow CUDA host memory limit
  - for help running 3D U-Net and access to the scripts and dataset
- Minsik Cho
  - for proving the example of compression code to our summer intern

Additional material

Introduction

- We consider training of a neural network on a GPU device.
- We choose Google's TensorFlow as the subject of our study.
- TensorFlow needs two types of input:
  the model to be trained and a labeled data set.
- We assume to have access to its model data flow graph. (Python API)
- At run-time (during training) large tensors are flowing through the data flow graph, from node to node. These are called activation tensors.
- They are the result of the evaluation of a node's function.
- TensorFlow is responsible for scheduling the order of node operations and the memory management of the activation tensors.
  Several operations might be scheduled to run concurrently.

Problem Statement

GPU memory is a limited resource when training a deep learning network. We like to train large models on large input samples that would otherwise not fit and cause out-of-memory conditions.

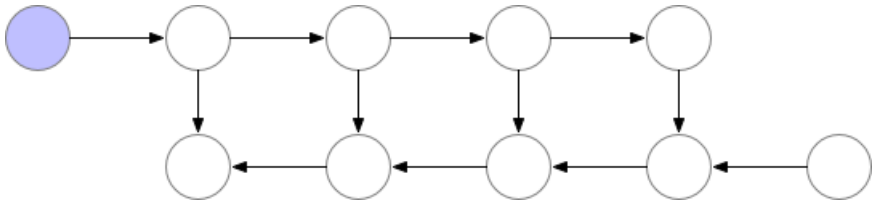
Intuition

- We intend to modify the data flow graph by inserting certain sub graph structures and additional control dependency edges that off-load activation tensor data from the GPU's memory.
- We propose what these sub graph structures comprise of and indicate where to insert them and where to add control dependencies.
- Two obvious guidelines come to mind:
  1. Consider large tensors; it doesn't make much sense to off-load small tensors since the communication overhead might become too large.
  2. Consider tensors that have a long "life time", i.e., there is a substantial time interval between the moment they are produced till the moment they are last used. We call this "life time" interval "slack".

Brief Overview of TensorFlow www.tensorflow.org

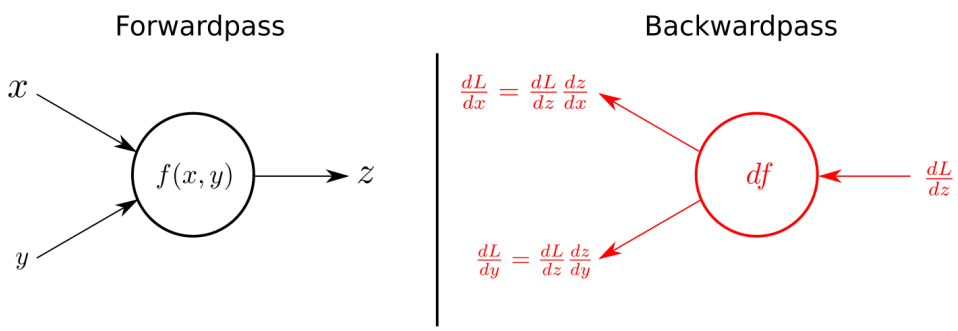
- "An open source machine learning framework for everyone"
- Very sophisticated and extensive tool
  - Many APIs at different levels
  - Many back-ends (CPU, GPU, XLA, TPU)
  - Extensive documentation, tutorials, Jupyter notebooks
  - Large, active community; many third-party contributions
- Core C++ functionality exposed through Python
- "Programming model" needs getting used to:
  - 1) Define data-flow graph; 2) Evaluate graph

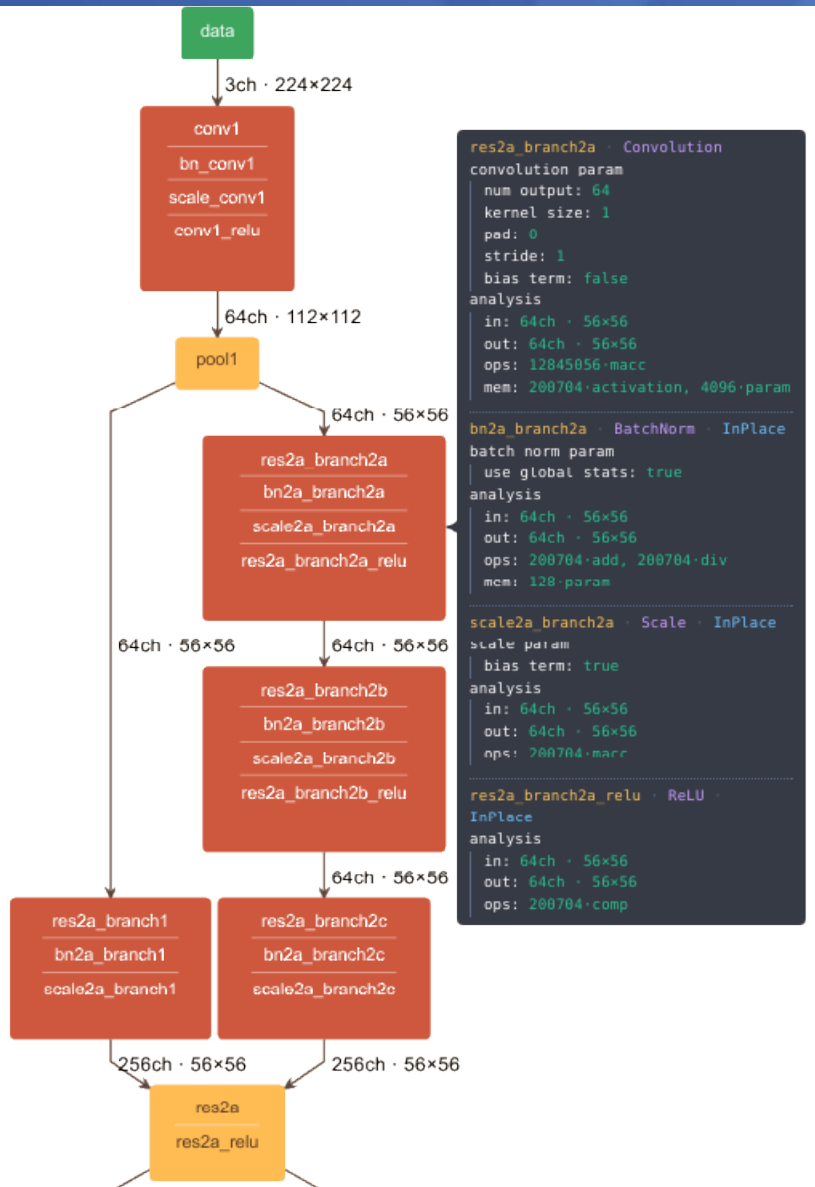
Typical forward/backward propagation of tensors



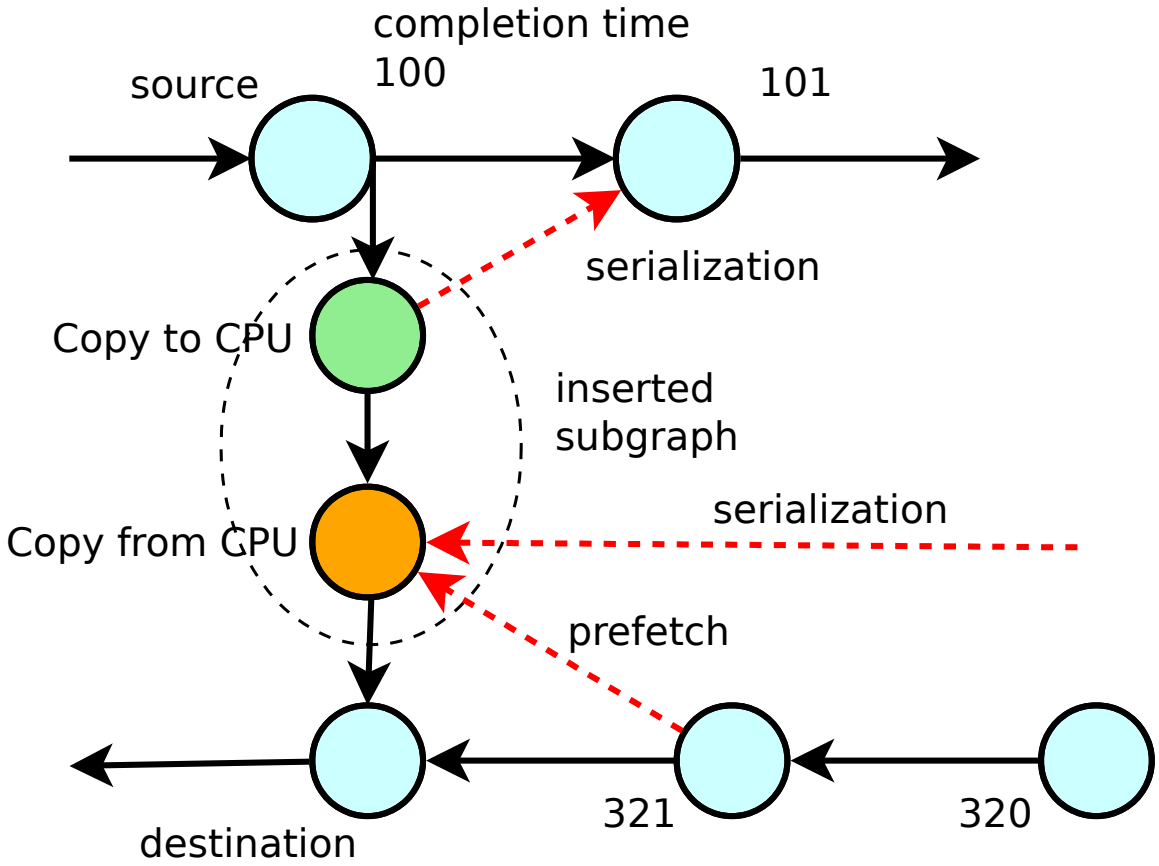
Forward: more and more tensors are allocated
Backward: gradients are computed, tensors are freed

Resnet50 structure detail

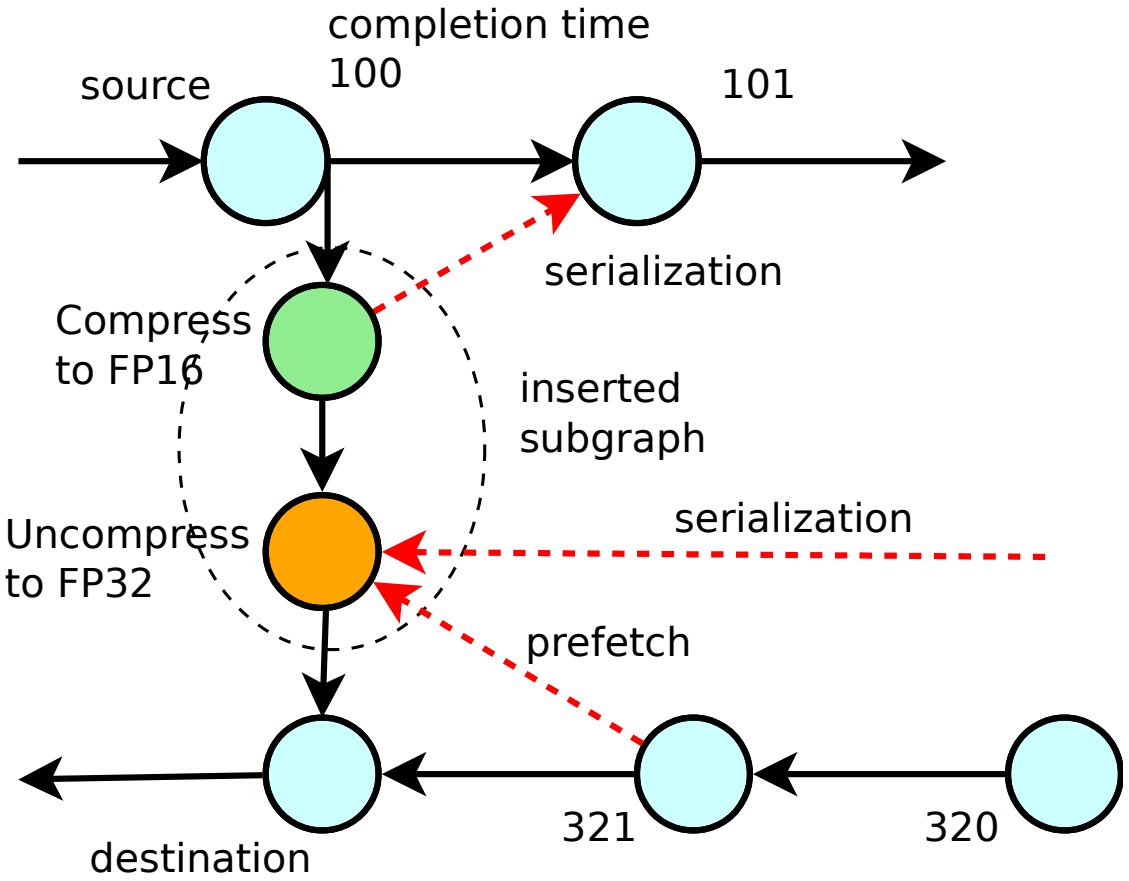




Example of a subgraph: swapout/swapin nodes



Example of a subgraph: compress/uncompress nodes



Semantics

- During a each training iteration:
  - input a sample (or minibatch)
  - compute node activations in forward phase (tensor data allocations)
  - apply loss function
  - compute gradients in backward phase (more allocations)
  - apply gradient update to weights and biases (tensor data deallocations)
- Producing a node's output requires the creation of a new tensor that needs a memory allocation. This memory can only be freed after the last consumer node of that tensor has executed.
- At any one time during a training iteration there can be many activation tensors occupying GPU memory.

Memory profile resnet50 batch size 1024; swapout/swapin