CUDA GRAPH IN TENSORFLOW

Jiajie Yao, Dec 2020

CONTENT

What’s CUDA Graph

How to Use CUDA Graph

Launch Overhead in TensorFlow

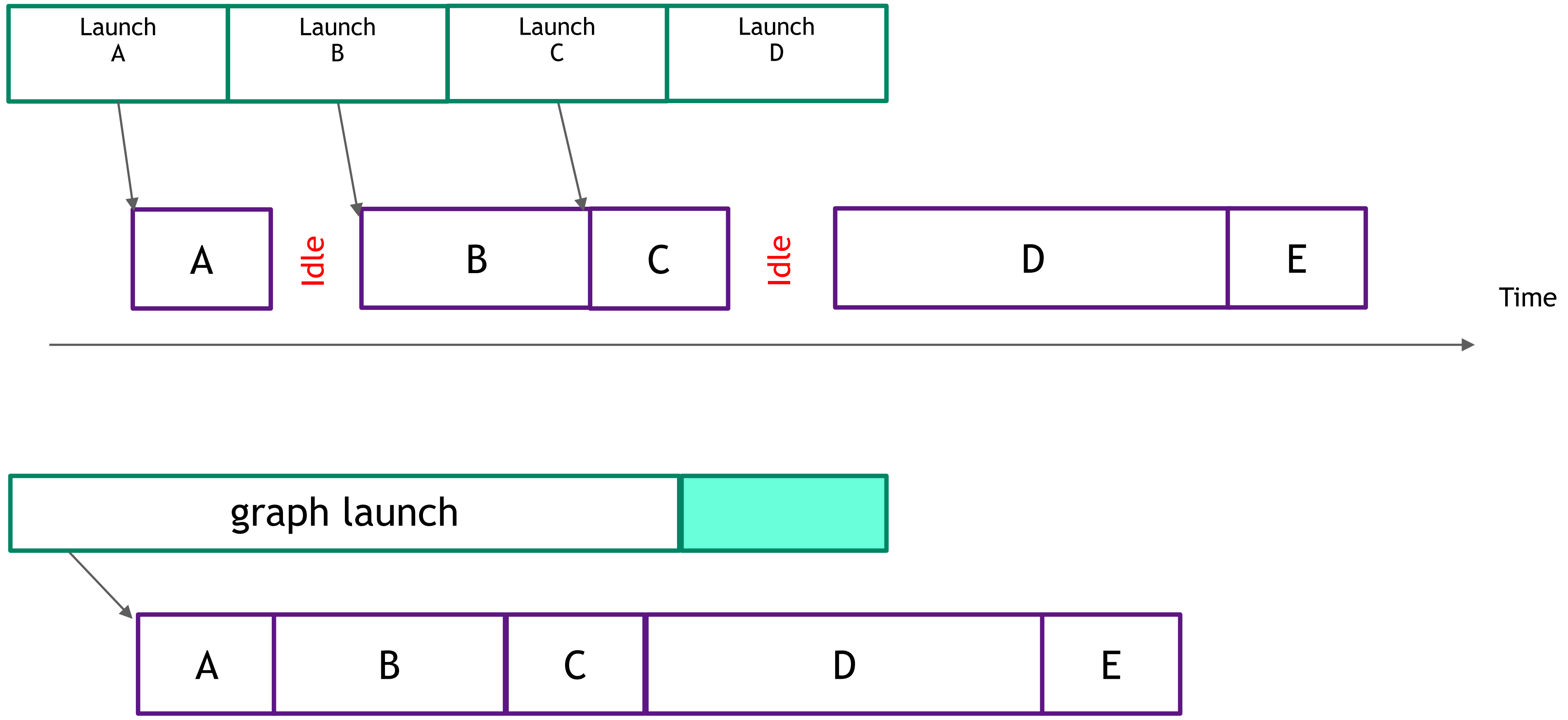
Integrate CUDA Graph into TensorFlow

Performance

WHAT'S CUDA GRAPH

What Problem CUDA Graph Solves

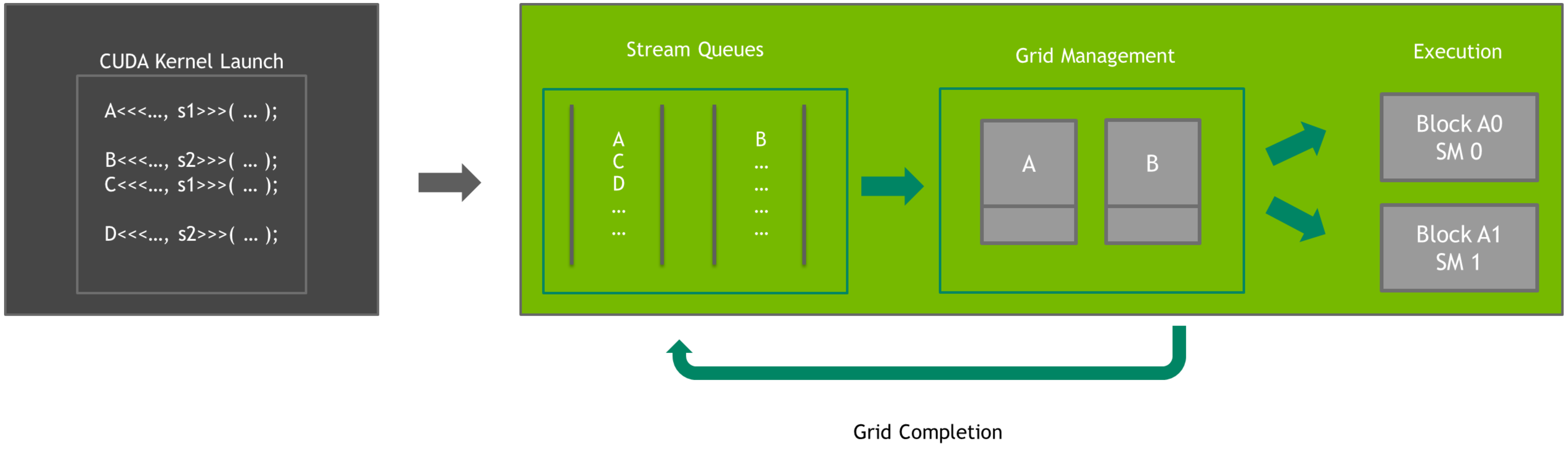
Reduce Launch Overheads



WHAT'S CUDA GRAPH

Stream Launch vs Graph Launch

Stream Launch

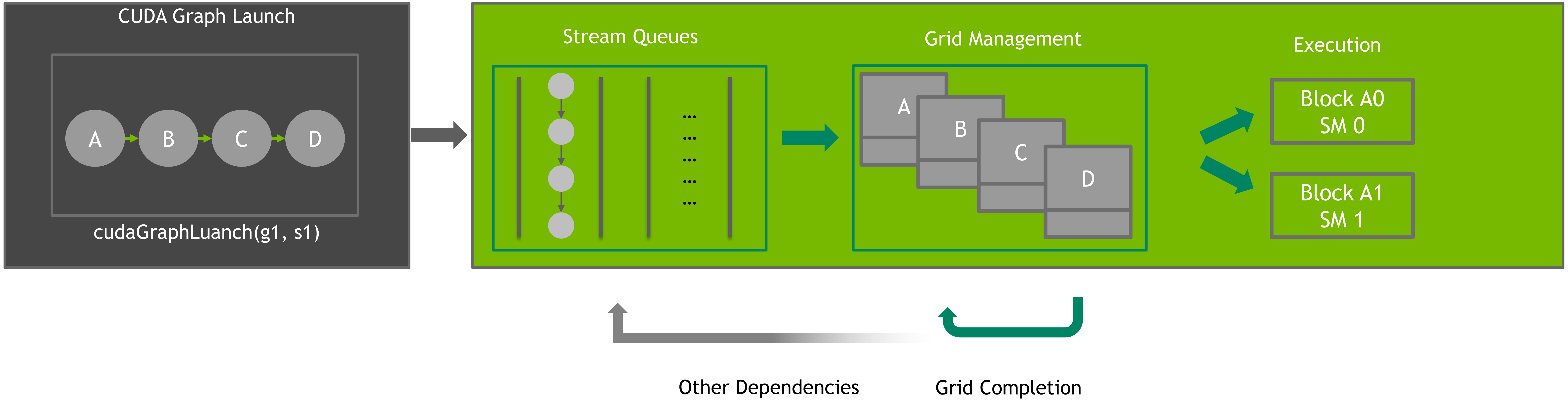


https://developer.nvidia.com/gtc/2020/video/s21760

WHAT'S CUDA GRAPH

Stream Launch vs Graph Launch

Graph Launch (Pre-Ampere)

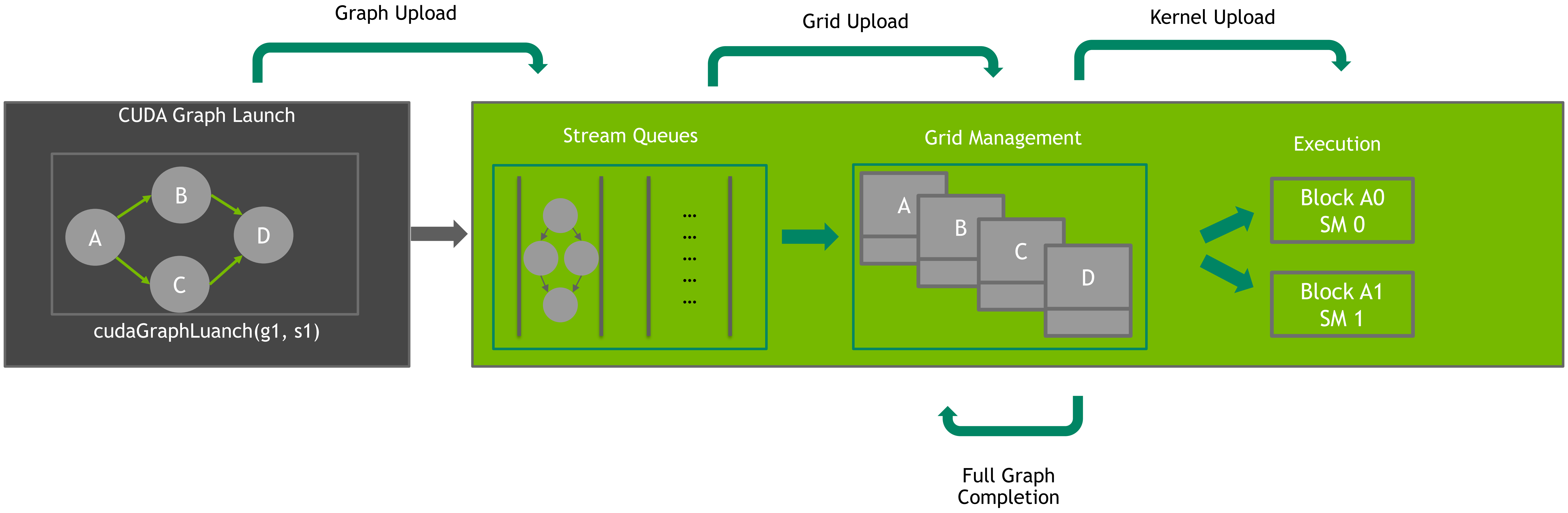


https://developer.nvidia.com/gtc/2020/video/s21760

WHAT'S CUDA GRAPH

Stream Launch vs Graph Launch

Graph Launch (Ampere)



https://developer.nvidia.com/gtc/2020/video/s21760

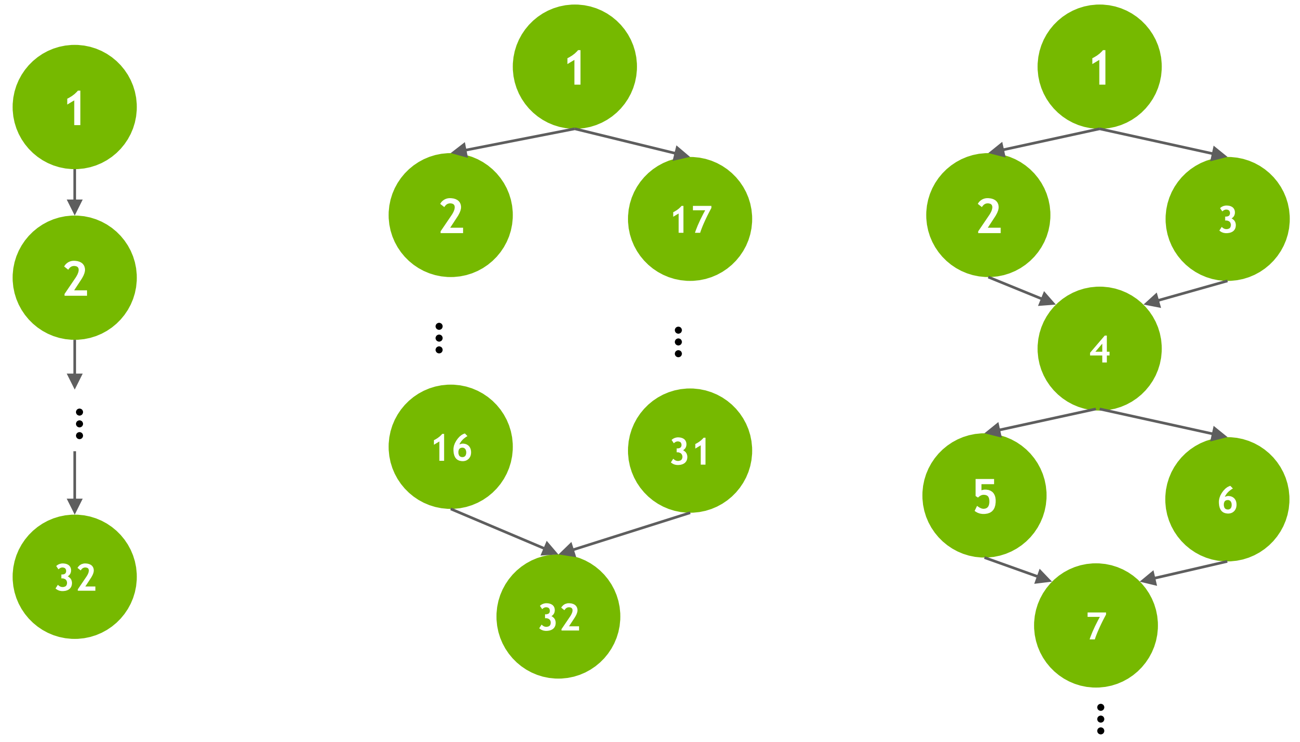
WHAT'S CUDA GRAPH

Stream Launch vs Graph Launch

Launch overhead comparison (test using empty kernel)

A100 GPU *

Graph with 32 nodes



Pattern	graph		stream		host speedup	device speedup
	host (ms)	device (ms)	host (ms)	device (ms)		
1 striaght line	4.43	28.12	65.25	60.67	14.7	2.2
2 two branches	3.17	15.47	69.25	83.46	21.8	5.4
3 fork and join	4.28	21.32	93.75	161.79	21.9	7.6

HOW TO USE CUDA GRAPH

- Define a CUDA Graph
    - Stream Capture
    - CUDA Graph API
- Instantiate a CUDA Graph
    - Call cudaGraphInstantiate(…)
- Launch the CUDA Graph executable instance
    - Call cudaGraphLaunch(…)

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html

HOW TO USE CUDA GRAPH

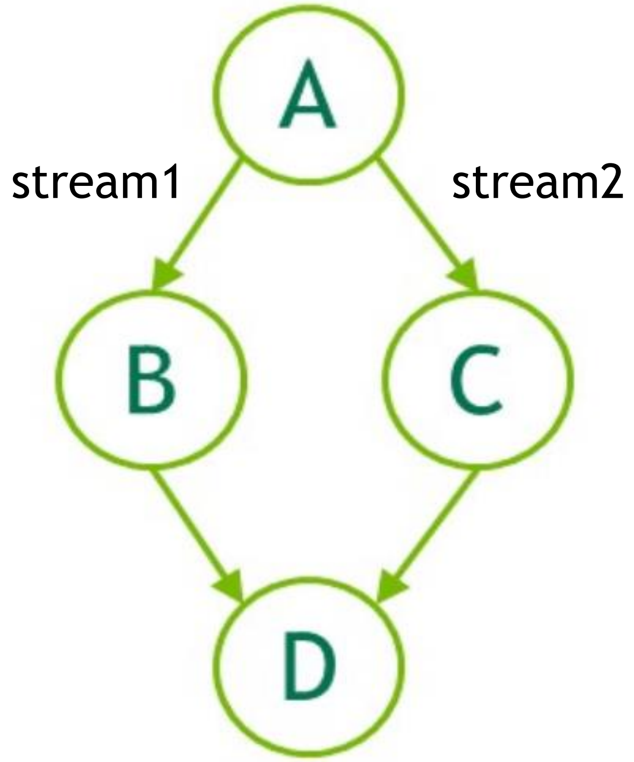
Define a CUDA Graph

Stream Capture

```
cudaStreamBeginCapture(stream1);
kernel_A<<< ..., stream1 >>>(...);

cudaEventRecord(event1, stream1);
cudaStreamWaitEvent(stream2, event1);

kernel_B<<< ..., stream1 >>>(...);
kernel_C<<< ..., stream2 >>>(...);

cudaEventRecord(event2, stream2);
cudaStreamWaitEvent(stream1, event2);

kernel_D<<< ..., stream1 >>>(...);

// End capture in the origin stream
cudaStreamEndCapture(stream1, &graph);
```

Graph APIs

```
// Create the graph - it starts out empty
cudaGraphCreate(&graph, 0);

cudaGraphAddKernelNode(&a, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&b, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&c, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&d, graph, NULL, 0, &nodeParams);

// Now set up dependencies on each node
cudaGraphAddDependencies(graph, &a, &b, 1);    // A->B
cudaGraphAddDependencies(graph, &a, &c, 1);    // A->C
cudaGraphAddDependencies(graph, &b, &d, 1);    // B->D
cudaGraphAddDependencies(graph, &c, &d, 1);    // C->D
```



HOW TO USE CUDA GRAPH

CUDA Graph Node Types

Kernel

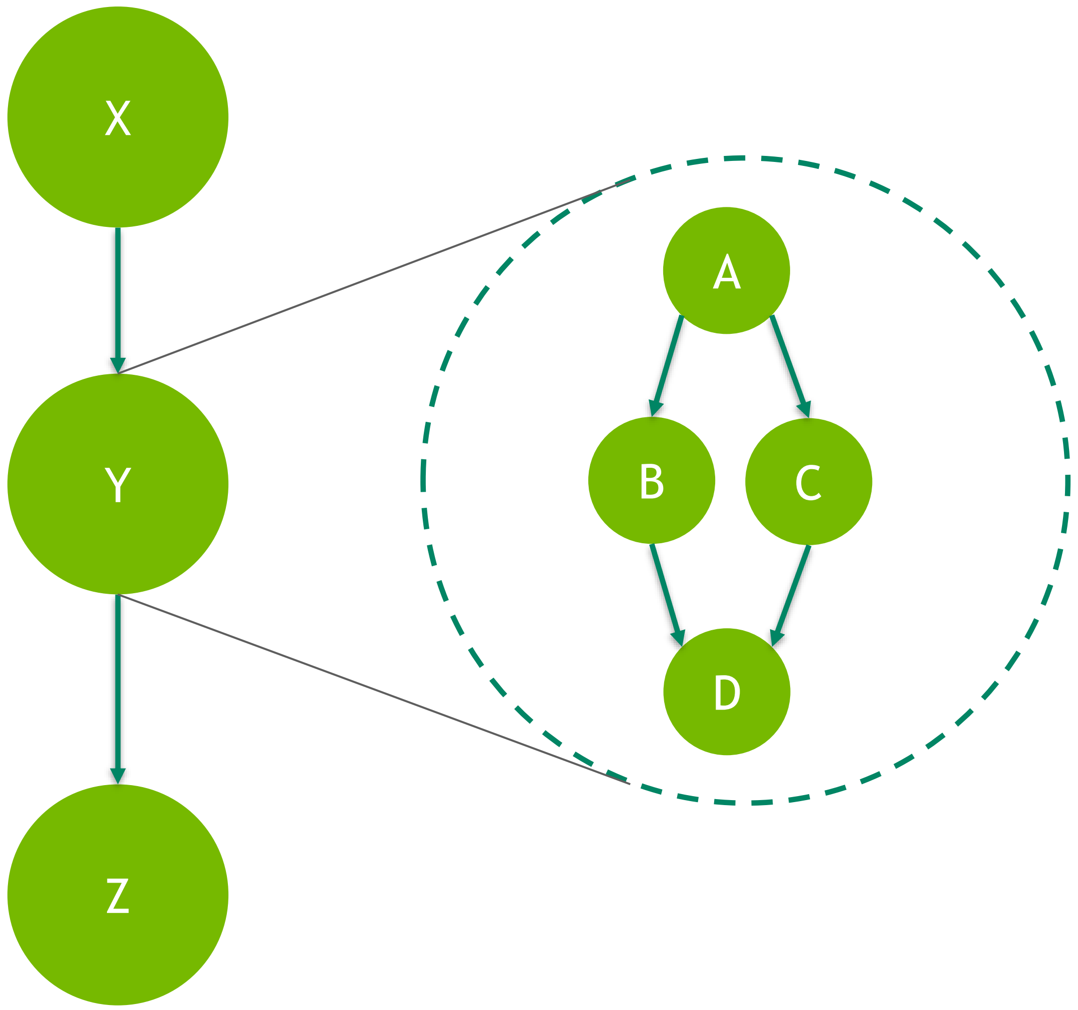
CPU function call

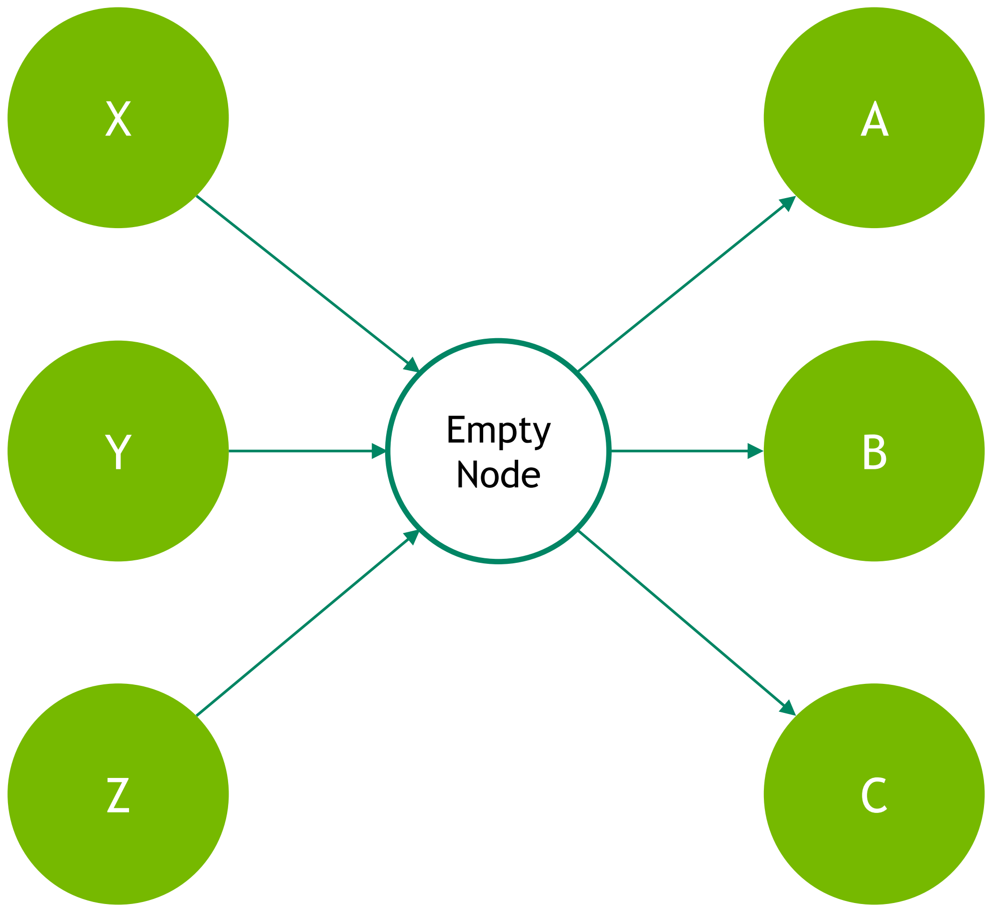
Memory copy

Memset

Empty node

Child graph





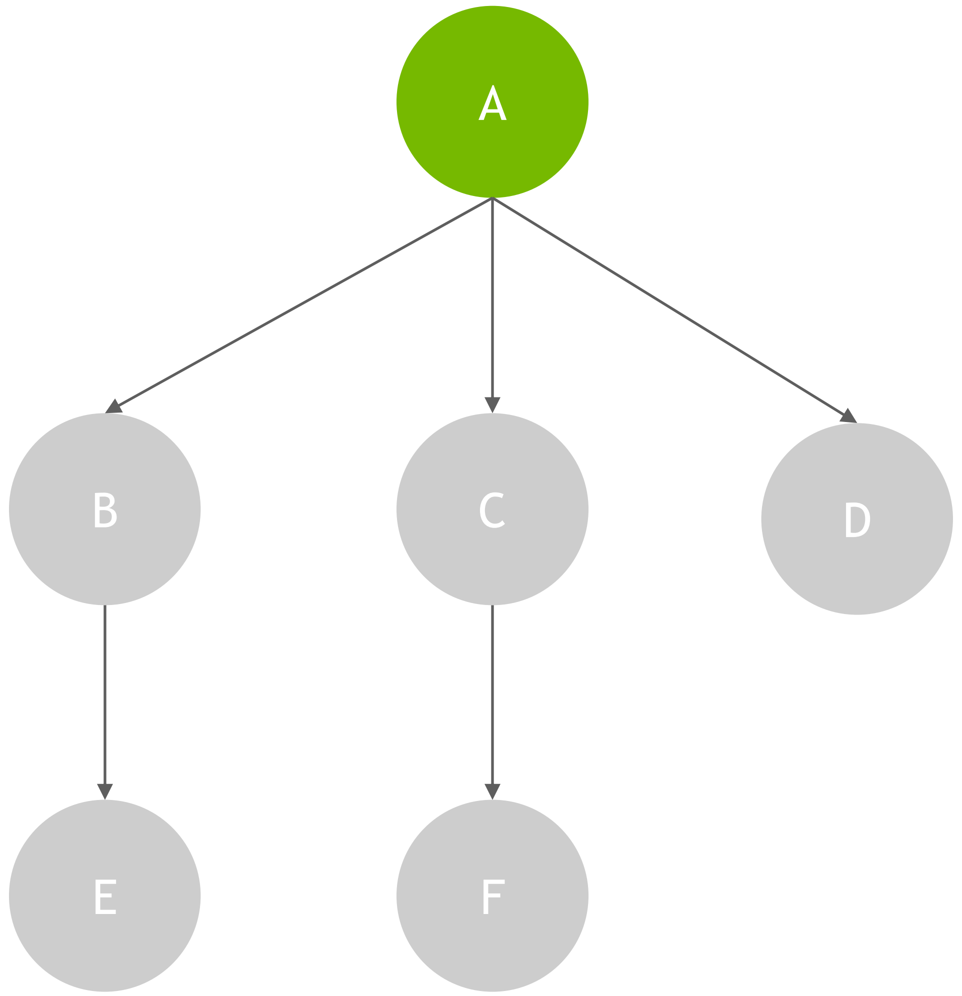
LAUNCH OVERHEAD IN TENSORFLOW

TF op Scheduling

Node A is ready to Run

ready_nodes = [A,]

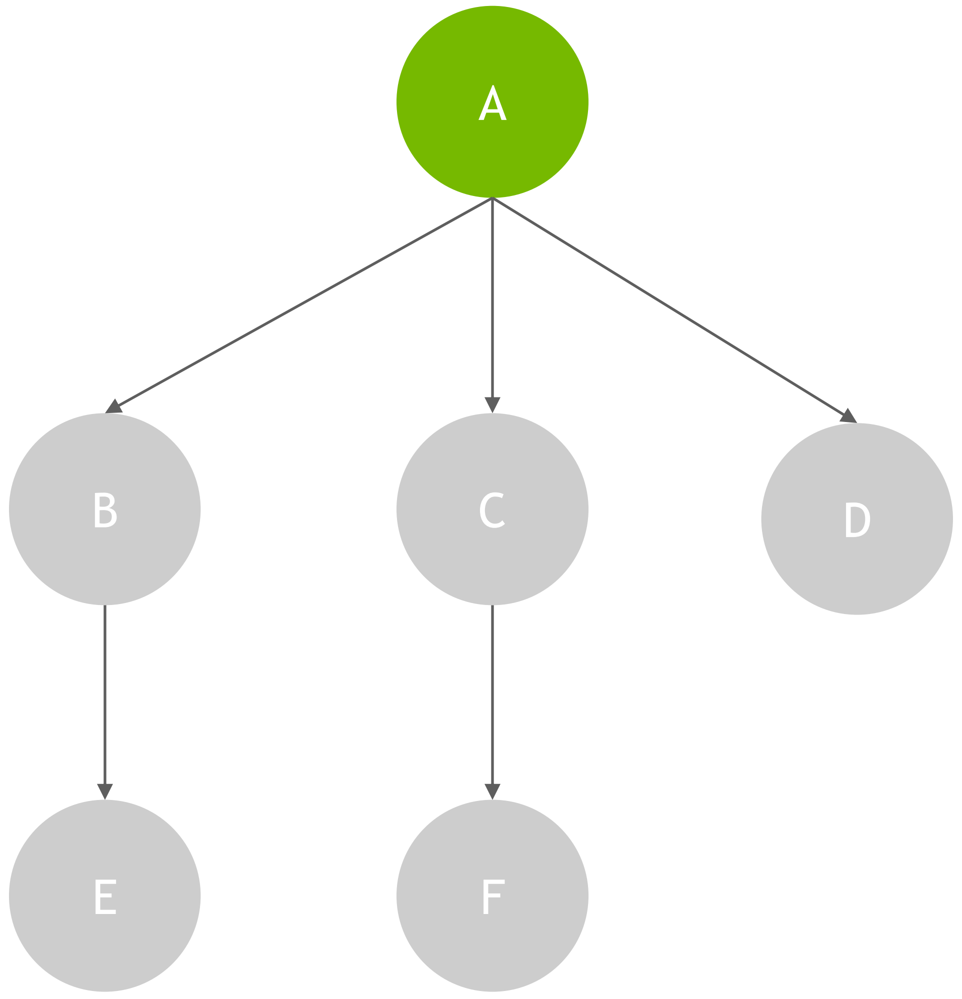
Call **ScheduleReady(read_nodes)** # [A,]



LAUNCH OVERHEAD IN TENSORFLOW

TF op Scheduling

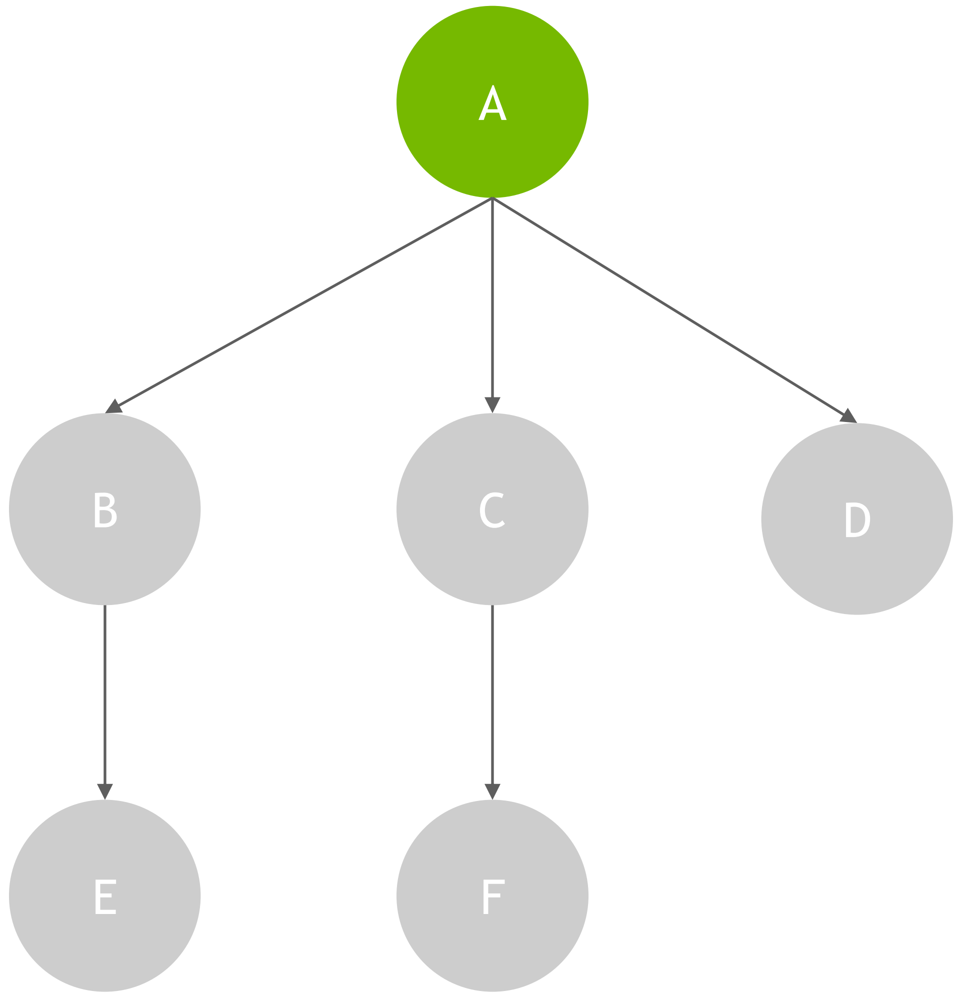
```
ScheduleReady(read_nodes):  # [A,], thread 1
    for node in read_nodes:  # [A,], thread 1
        Process (node)  # A, thread 2
```



LAUNCH OVERHEAD IN TENSORFLOW

TF op Scheduling

```
Process(node): # A
    inline_ready = [node,]  # A,
    while inline_ready no empty:
        node = pop(inline_ready)
        device.Compute(op_kernel, ctx) *
        ready_nodes = [B, C, D]
        ScheduleReady([B, C, D], inline_ready)  # thread 2
```

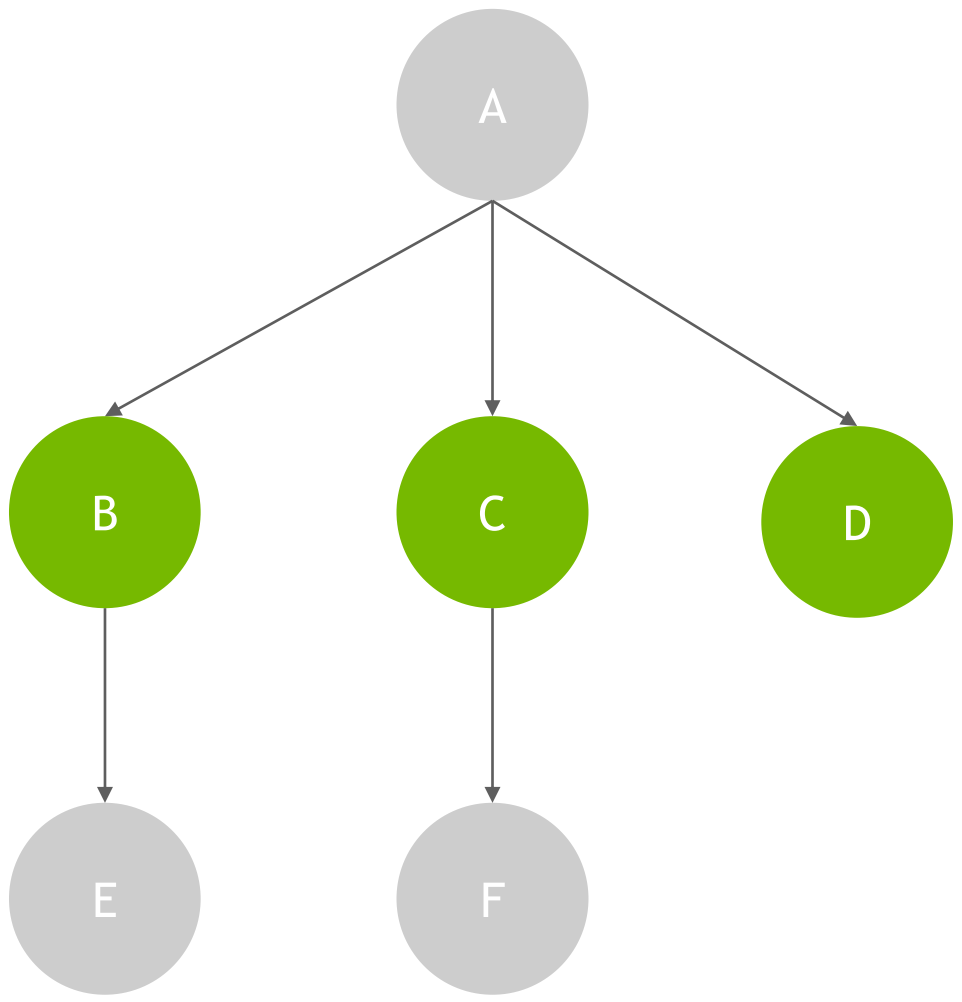


* The process of OpKernal (sync op)

LAUNCH OVERHEAD IN TENSORFLOW

TF op Scheduling

```
ScheduleReady(read_nodes, inline_ready):  # [B, C, D], thread 2
    for node in ready_nodes: # [B, C, D], thread2
        if node is expensive:
            Process (node) # thread 3,4,…
        else:
            inline_ready.push_back(node)
```



LAUNCH OVERHEAD IN TENSORFLOW

TF op Scheduling

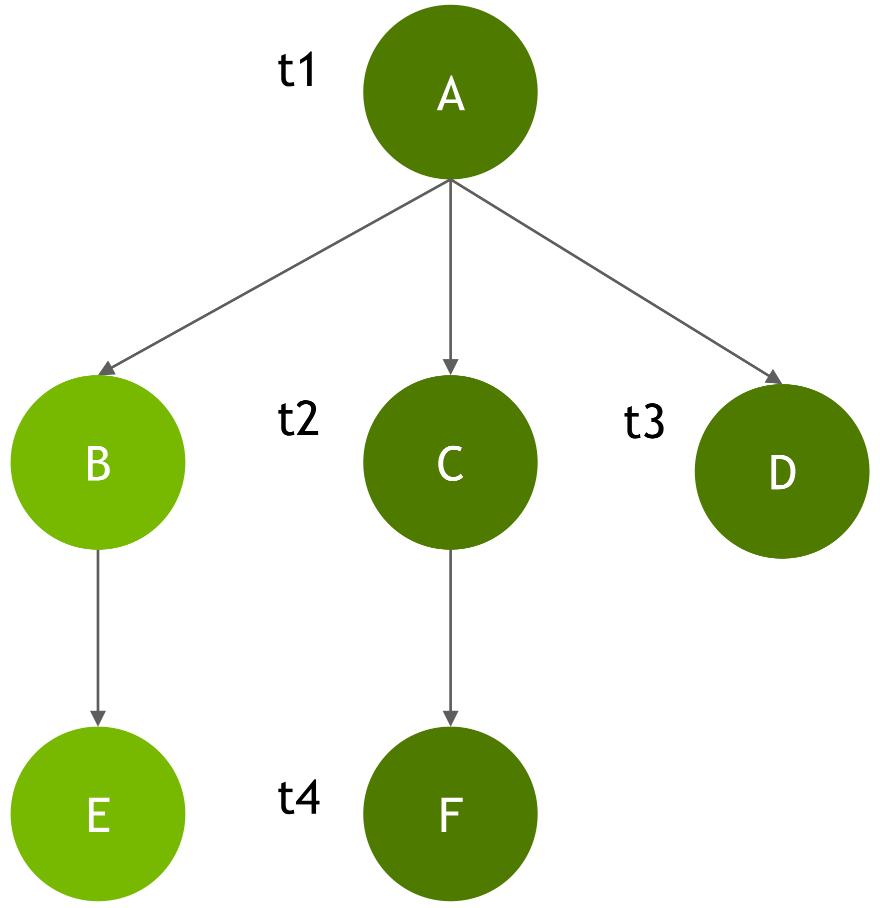
Expensive

Inexpensive

May use up to 4 threads to compute the graph

T1, T2, T3 may launch kernels simultaneously

- ❑ Multiple threads launching could significantly increase the launch overhead
- ❑ If multiple threads are executing session run, the overhead is even worse



INTEGRATE CUDA GRAPH INTO TENSORFLOW

Overview

Use only 1 stream for Compute, H2D, D2H, D2D

Disable syncs during session runs

- Syncs are not necessary, as in capture mode, all GPU operations are not executed but just recorded
- Synchronize happens during stream switch (e.g. scheduling from compute stream to copy stream)
- TF do the synchronization on GPU by building dependencies between the copy streams and the compute stream (using CUDA Events)

Memory management

- Hold the Tensors allocated during capturing
- Tensor reusing

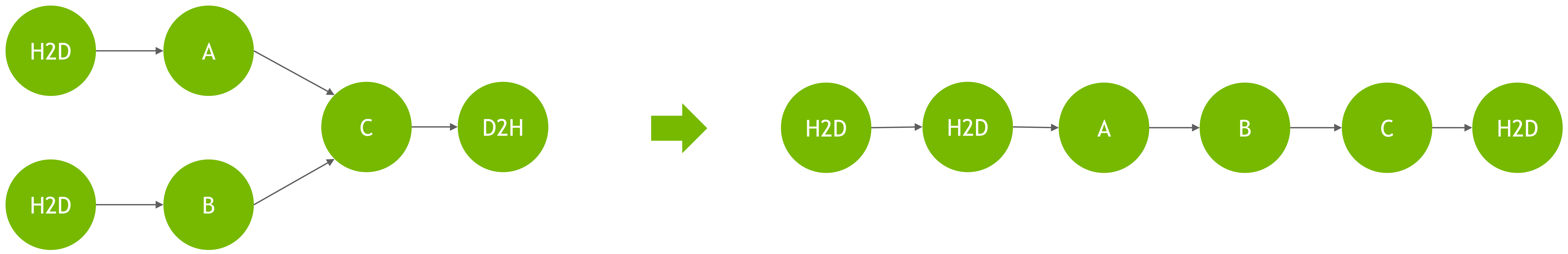
INTEGRATE CUDA GRAPH INTO TENSORFLOW

One Stream Executing

Use only 1 stream for Compute, H2D, D2H, D2D

Simplify the capture process, no need to use CUDA Events to build the dependencies

Pitfall: The original Graph flattens into a straight line, no parallelism between branches, nor between memcpy nodes and computing nodes (this issue can be solved by using multiple-graphs and multiple-streams)



INTEGRATE CUDA GRAPH INTO TENSORFLOW

One Stream Executing

Use only 1 stream for Compute, H2D, D2H, D2D

In DirectSession::EnableGraphCaptureMode(…), set single stream for GPU device.

tensorflow/core/common_runtime/gpu/gpu_device.cc

```
#ifdef GOOGLE_CUDA
// For enabling cuda-graph
void BaseGPUDevice::SetSingleStream(){
    if(stream_catpure_mode_) return;

    stream_backup_ = *stream_;

    stream_->device_to_host = stream_->host_to_device = stream_->compute;

    size_t d2d_size = stream_->device_to_device.size();
    for(size_t i = 0; i < d2d_size; i ++){
        stream_->device_to_device[i] = stream_->compute;
    }

    stream_->compute->SetStreamCaptureMode(true);

    device_context_->stream_ = stream_->compute;
    device_context_->host_to_device_stream_ = stream_->host_to_device;
    device_context_->device_to_device_stream_ = stream_->device_to_device;
    device_context_->device_to_host_stream_ = stream_->device_to_host;
}
```

```
void BaseGPUDevice::ResetStreams(){
    if(! stream_catpure_mode_) return;

    stream_->compute->SetStreamCaptureMode(false);

    *stream_ = stream_backup_;

    device_context_->stream_ = stream_->compute;
    device_context_->host_to_device_stream_ = stream_->host_to_device;
    device_context_->device_to_device_stream_ = stream_->device_to_device;
    device_context_->device_to_host_stream_ = stream_->device_to_host;
    gpu_device_info_->stream = stream_->compute;

}
```

INTEGRATE CUDA GRAPH INTO TENSORFLOW

Memory Management

Each Graph has a “Tensor Holder” object which holds the tensors (allocated during session run), so memory for subsequent CUDA Graph launch and for normal session runs are isolated

Tensor Holder is also responsible for reducing memory footprint (by reusing tensors)

tensorflow/core/common_runtime/direct_session.cc : DirectSession::RunInternal

```
……

// create tensor holder before the scheduling
TensorHolder * tensor_holder = nullptr;
if(cuda_graph_capture_mode_){
     TF_RETURN_IF_ERROR(GetTensorHolder(&tensor_holder));
}

……

#ifdef GOOGLE_CUDA
   if(cuda_graph_capture_mode_){

      ……

      // hold tensors for both CPU and GPU
      args.tensor_holder = tensor_holder;
   }
#endif

item.executor->RunAsync(args, barrier->Get());
```

tensorflow/core/common_runtime/executor.cc : ExecutorState::Process

```
……


// Synchronous computes.

OpKernelContext ctx(&params, item.num_outputs);

if(tensor_holder){
       ctx.tensor_holder = tensor_holder;
}

nodestats::SetOpStart(stats);

……
```

* *Similar changes will be applied to Async computes*

INTEGRATE CUDA GRAPH INTO TENSORFLOW

Memory Management

TF dynamically allocate output tensors via OpKernelContext

OpKernelContext will check if it can resue Tensors, and tensor holder will hold newly allocated tensors (in Capture Mode)

tensorflow/core/framework/op_kernel.cc

```
Status OpKernelContext::allocate_tensor(
    DataType type, const TensorShape& shape, Tensor* out_tensor,
    AllocatorAttributes attr, const AllocationAttributes& allocation_attr) {

  if(shape.num_elements() > 0 && tensor_holder){
      Tensor reuse_tensor = tensor_holder->FindUsableTensor(type, shape);
      if(reuse_tensor.TotalBytes() > 0){

          out_tensor->shape_ = shape;
          out_tensor->set_dtype(type);
          if(out_tensor->buf_){
              out_tensor->buf_->Unref();
          }
          out_tensor->buf_ = reuse_tensor.buf_;
          out_tensor->buf_->Ref();

          return Status::OK();
      }
  }

……
```

tensorflow/core/framework/op_kernel.cc : OpKernelContext::allocate_tensor

```
Tensor new_tensor(a, type, shape,
                  AllocationAttributes(allocation_attr.no_retry_on_failure,
                                       /* allocation_will_be_logged= */ true,
                                       allocation_attr.freed_by_func));

……

  if(tensor_holder){
      if(new_tensor.AllocatedBytes() > 0){
          tensor_holder->Add(&new_tensor);
      }
  }

  *out_tensor = std::move(new_tensor);
  return Status::OK();
```

INTEGRATE CUDA GRAPH INTO TENSORFLOW

Post Capture Process

After capturing, post process the CUDA Graph

- Validation of the Graph
    - If the graph contains extra D2H or H2D nodes (besides the input/output tensor), then it may have uncaptured CPU operations, which could make the Graph invalid
    - Each H2D node in the graph, is either corresponding to an input tensor or const* CPU tensor (given specified input shapes, generated by shape related operations on CPU, e.g. concat/shuffle of dims) which is held by Tensor Holder
    - Each D2H node in the graph, is corresponding to an output tensor
- H2D nodes removal
    - Remove the H2D nodes corresponding to input tensors, and record the (host_src --> gpu_dst) mappings for the input tensors
    - Eliminate the needs to do extra H2H (host to host) copies before launching the CUDA Graph, and user is responsible for the H2D copies of input data based on the (host_src --> gpu_dst) mapping
- Instantiate the CUDA Graph
    - Create the Executable CUDA Graph instance

* The solution can handle CPU operations which process shape data (most shape info is generated on CPU, so there will no be corresponding D2H nodes, if the input shapes do not change, we can view these info as const, and these info will be built into the launch parameters of the kernel nodes in the captured graph)

INTEGRATE CUDA GRAPH INTO TENSORFLOW

Graph Management

Graphs and Executable Graph Instances (and the corresponding tensor holders) are held by Direct Session object

Get a Graph by specifying the model name and the graph index

Specify the captured model name in DirectSession::EnableGraphCaptureModel(model_name)

After capturing, the Graph and Executable Graph Instances are appended to the vector indexed by “model_name”

DirectSession::DestroyCudaGraphs() will destroy the CUDA Graphs and corresponding tensor holders (release memory)

tensorflow/core/common_runtime/direct_session.h

```
class DirectSession : public Session {
    ….

    // holds the tensors allocated during graph capturing
    // model_name --> tensor_holders
    // for each model, multiple graphs can be captured,
    // so we can run multiple graph instances in parallel

    std::map<std::string, std::vector<TensorHolder>>        cuda_graph_tensor_holders_;
    std::map<std::string, std::vector<cudaGraph_t>>         cuda_graphs_;
    std::map<std::string, std::vector<cudaGraphExec_t>>     cuda_graph_instances_;

    ….
};
```

INTEGRATE CUDA GRAPH INTO TENSORFLOW

Workflow

Capture

Enable Capture Mode: session->EnableGraphCapture(“model_name”)

Call session->run (call multiple times to capture multiple graphs for the specific model)

Disable Capture Mode: session->DisableGraphCapture()

Repeat the above steps to capture graphs for other models

Launch

Get src->dst mappings for the H2D nodes corresponding to the input tensors

Copy real inputs to GPU (based on the src->dst mappings)

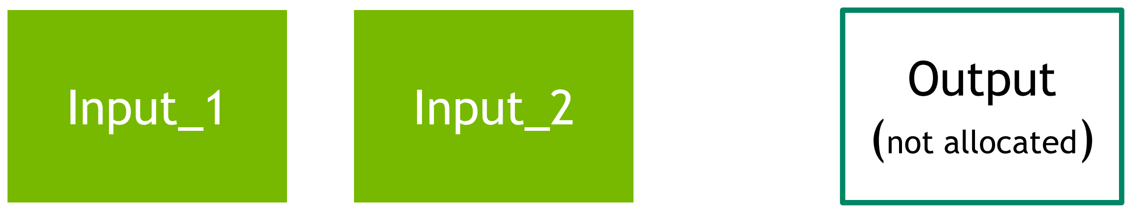
Launch graphs into different streams, session->RunCudaGraph(“model_name”, graph_idx, stream)

Do H2H copy for the results (optional)

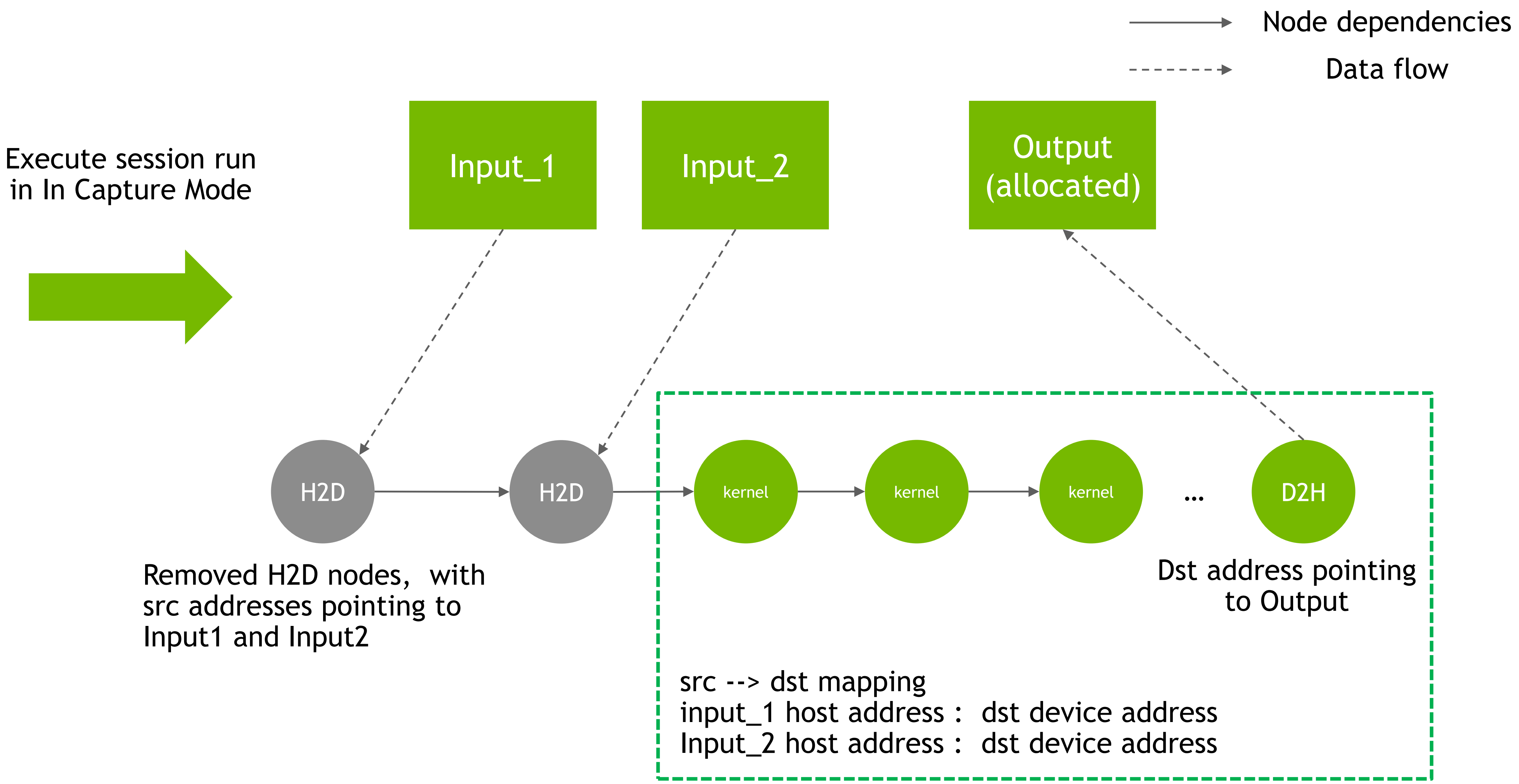
INTEGRATE CUDA GRAPH INTO TENSORFLOW

Workflow

Capture Graph



User: Prepare input tensors and output tensors, output tensors are just empty tensors for now, they will be allocated after session run



Captured CUDA Graph (with SRC --> DST mapping info)

INTEGRATE CUDA GRAPH INTO TENSORFLOW

Workflow

Launch Graph

Host

Device

New Input_1

New Input_2

Step1: User initiates H2D copies for new input1 and input2 (given the src --> dst mapping)

Captured CUDA Graph (with SRC --> Dst mapping info)

kernel → kernel → kernel ... D2H

Dst address pointing to Output

src --> dst mapping
input_1 host address : dst device address
Input_2 host address : dst device address

Step2: Launch the CUDA Graph (the CUDA graph contains the H2D node which copy results back to host)

This is the same tensor used for capturing

Output (CUDA Graph)

Step3: User does H2H copy, send to results to a specified output tensor

Output

INTEGRATE CUDA GRAPH INTO TENSORFLOW

Future Work

- How to “record” the operations on CPU – Stream Capture only records GPU operations (kernel launch and memory copy)

- How to efficiently update the parameters of the captured graphs (e.g. changing of batch size or input shapes, will change the kernels/parameters to be used)

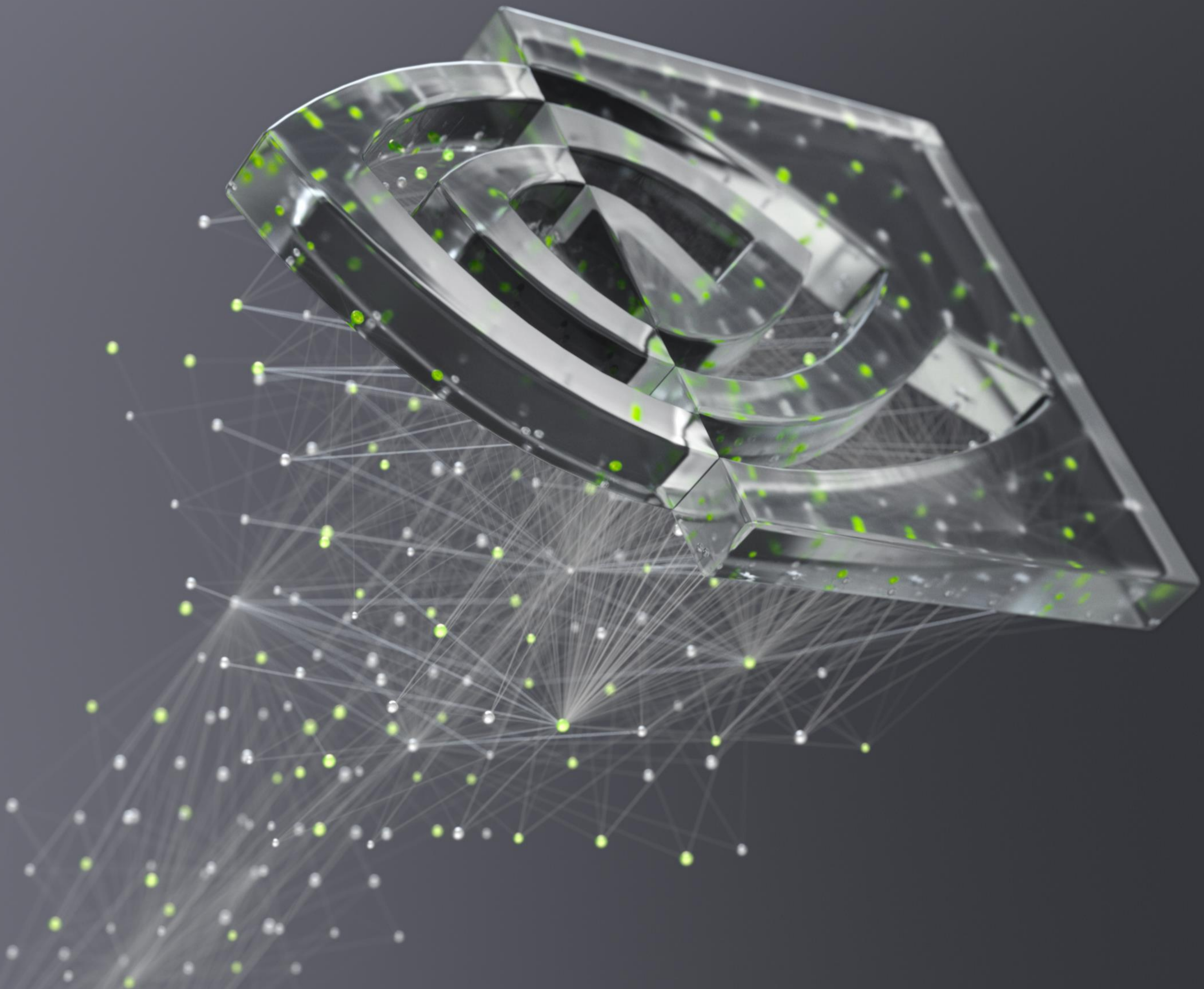
    - Direct updating of parameters of nodes in the CUDA Graph executable instance is almost impossible when use libs as the kernel signatures are unknown

PERFORMANCE

An MLP model

batch size	TF(3,3)	TF(1,3)	TF(*,3)	TF(1,8)	TF(3,1)	TF(1,1)	TF(*,3)	CUDA Graph	
64	1035	1174	415	1116	514	882	441	4155	(3.5X)
200	617	1020	403	1046	413	704	419	2979	(2.8X)
800	369	664	361	672	327	453	355	1184	(1.7X)
1600	314	432	308	433	301	358	287	727	(1.6X)

- QPS was tested for this model (Queries per second)
- Tested on Nvidia-T4 GPU PCIE 16G
- TF(M, N) : means that for each session, there are M threads used for launching GPU kernel, and there are N threads call session runs simultaneously
  By default, TF use all available threads on the system to schedule operations (and do kernel launches), TF(*, N) represents the default setting
  we can set the number of threads used for kernel launching (or more precisely, for processing GPU operations) by exporting following env variables:
  $ export TF_GPU_THREAD_MODE=gpu_shared # or gpu_private, gpu_shared – all GPU devices share the thread pool, gpu_private – each device uses its own pool
  $ export TF_GPU_THREAD_COUNT=M
- In the CUDA graph case, 3 graphs and 3 CUDA streams are used
- The speed up data is calculated by comparing the QPS got from CUDA Graph launch and the best QPS got from TF



NVIDIA