CUDA GRAPH IN TENSORFLOW

Jiajie Yao, Chenlu Li, Apr 2021

CONTENT

What’s CUDA Graph

How to Use CUDA Graph

What happens in TF session run

Integrate CUDA Graph into TensorFlow

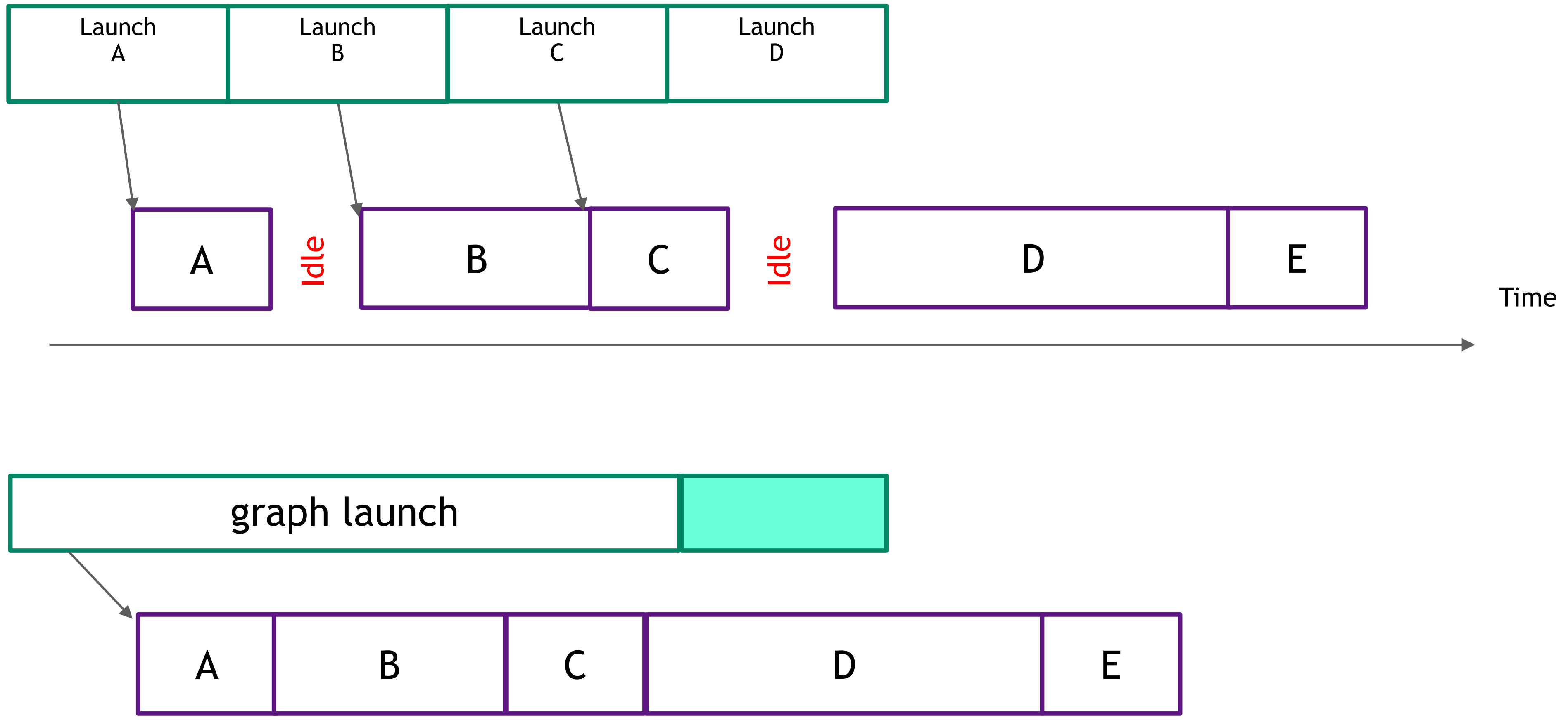
Use case: Alibaba Search Recommendation System

CONTENT

WHAT'S CUDA GRAPH

What Problem CUDA Graph Solves

Reduce Launch Overheads



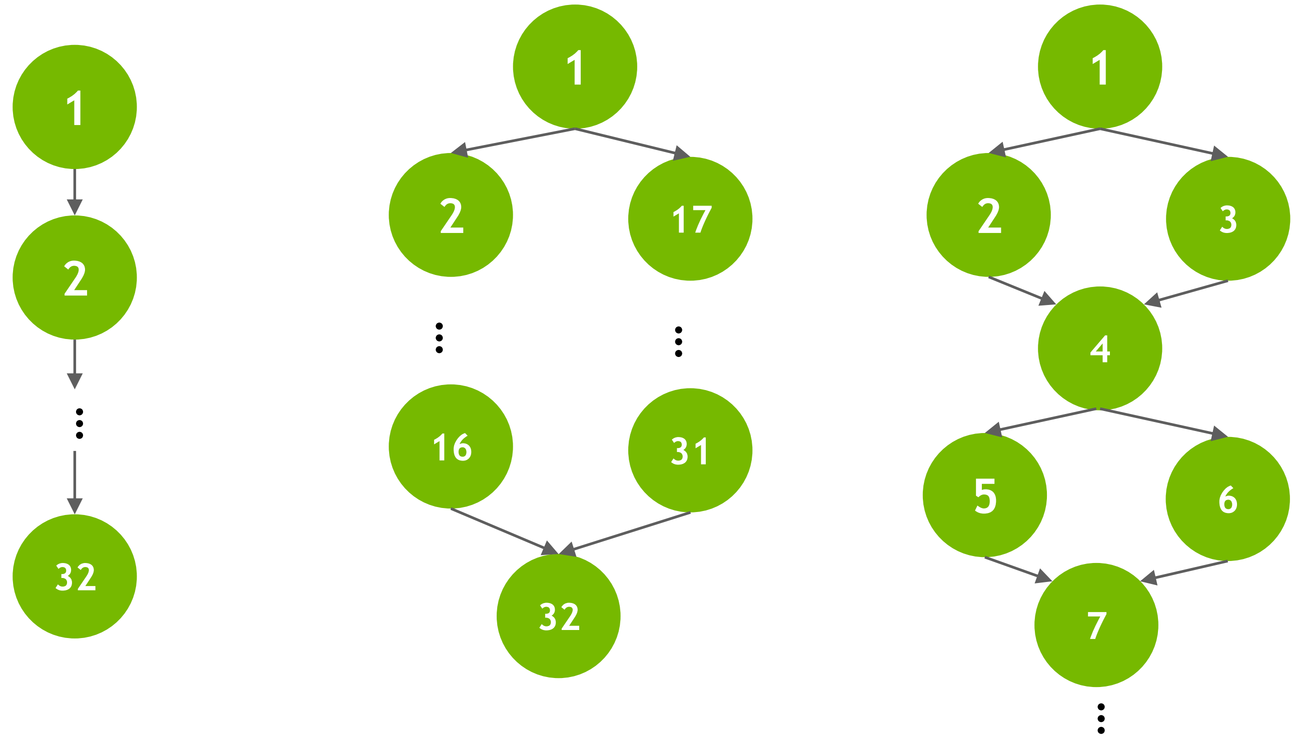
WHAT'S CUDA GRAPH

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

CONTENT

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

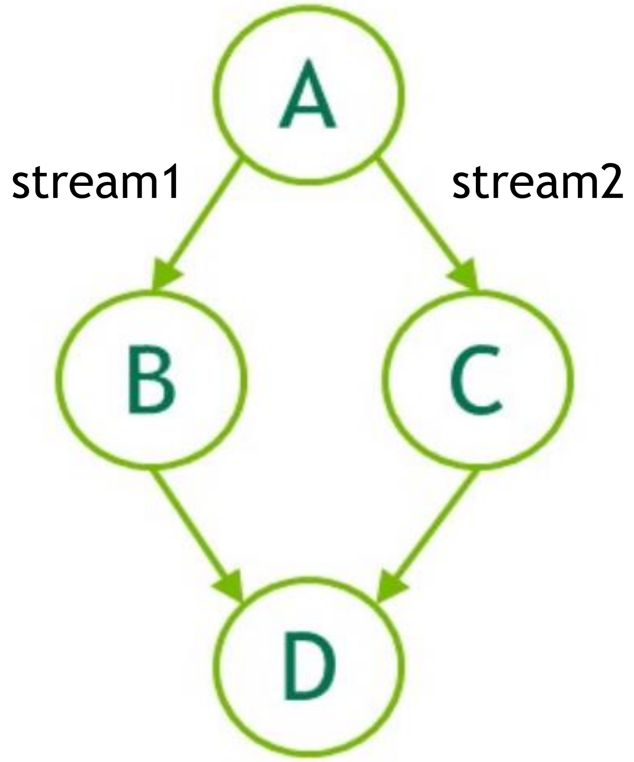
Define a CUDA Graph

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

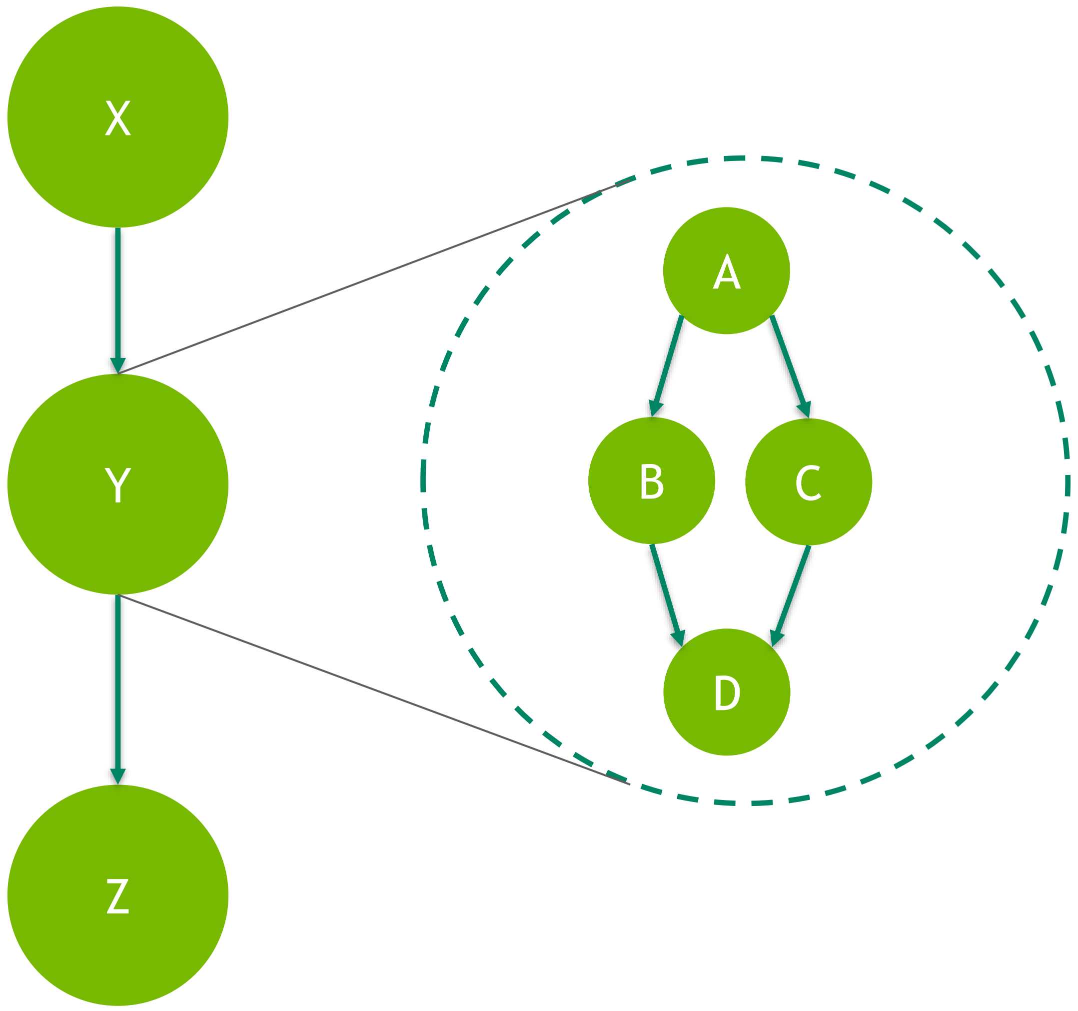
CPU function call

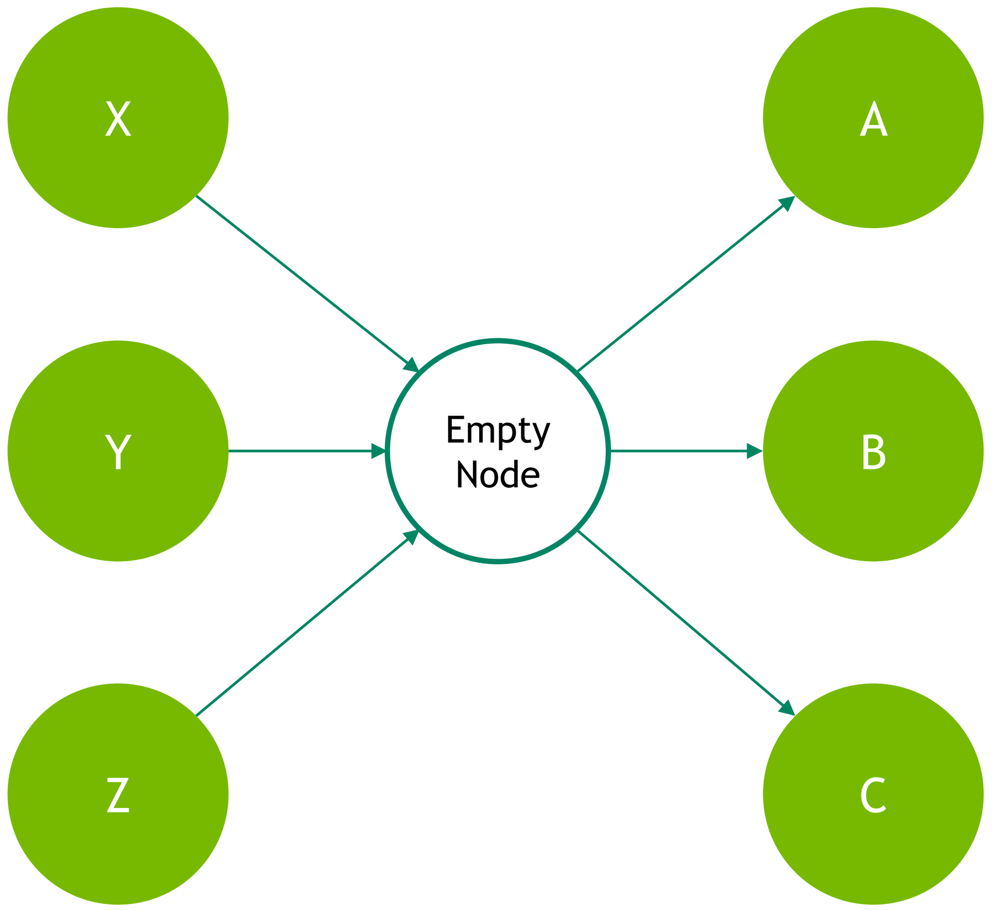
Memory copy

Memset

Empty node

Child graph





CONTENT

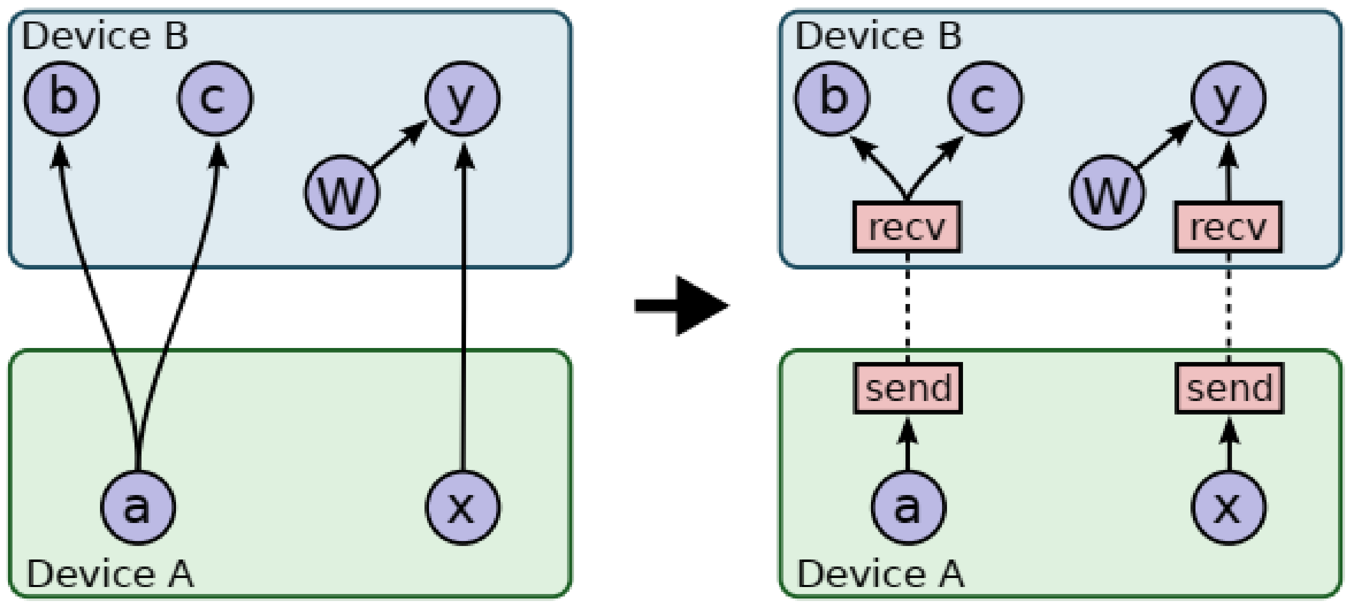
WHAT HAPPENS IN SESSION.RUN

Steps in session.run

1) Graph Optimizing (**grappler**) and op placement

3) Send/Recv nodes (**rendezvous**) nodes inserting

3) Create executors for each device, and start the scheduling

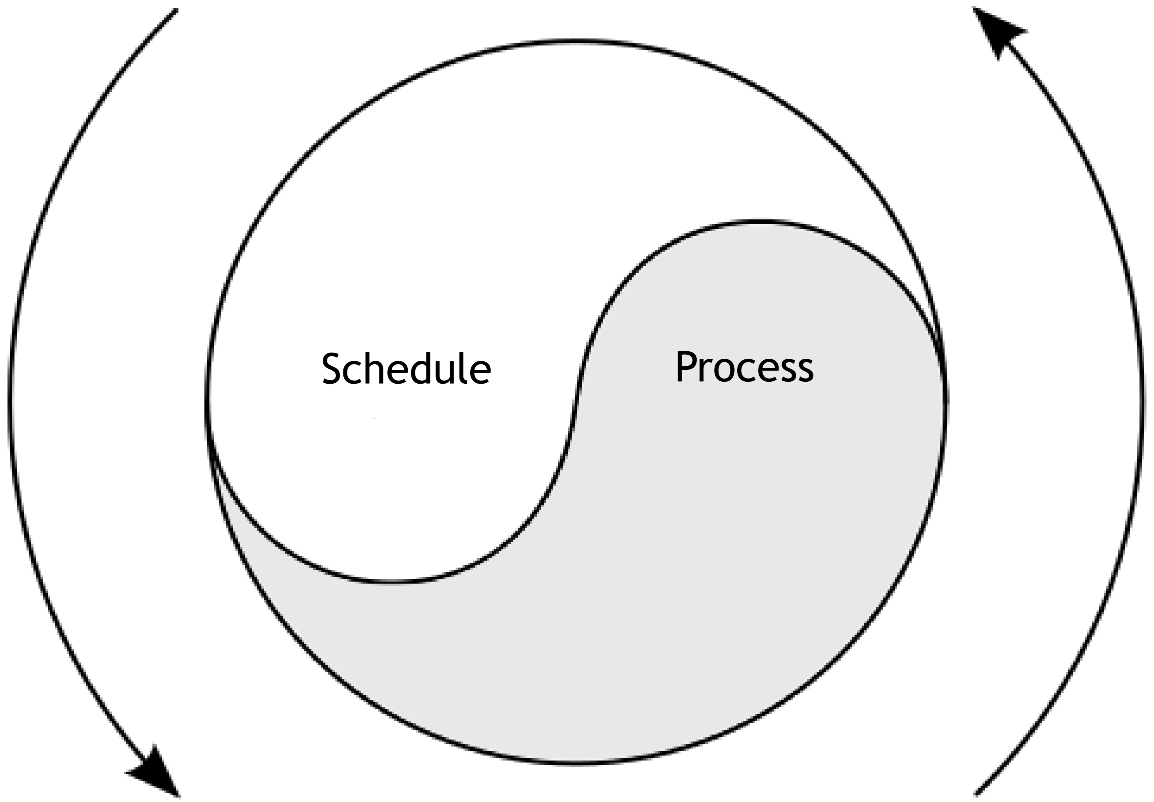


WHAT HAPPENS IN SESSION.RUN

How Are Operations Scheduled

Schedule – Process Loop

Schedule
Send “ready nodes” to Process



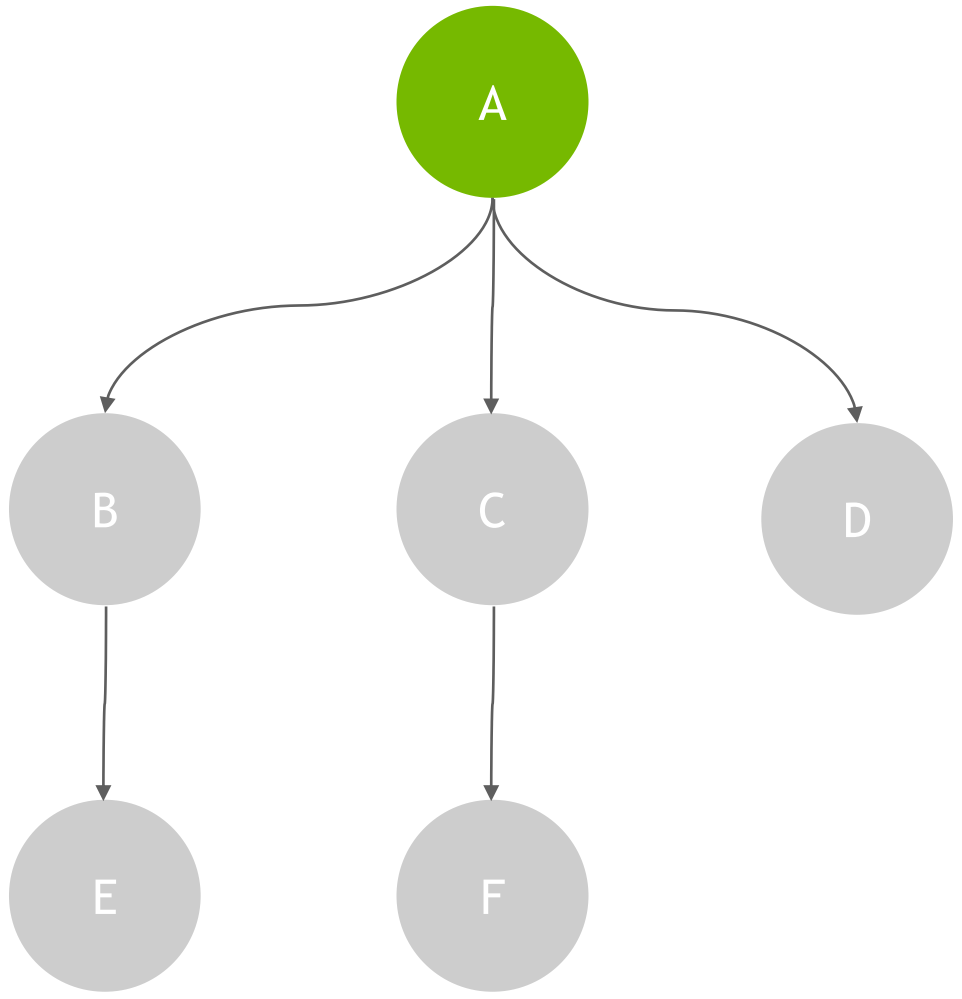
Process
Compute a node, generate new “ready nodes”

WHAT HAPPENS IN SESSION.RUN

How Are Operations Scheduled

ready_nodes = [A,]

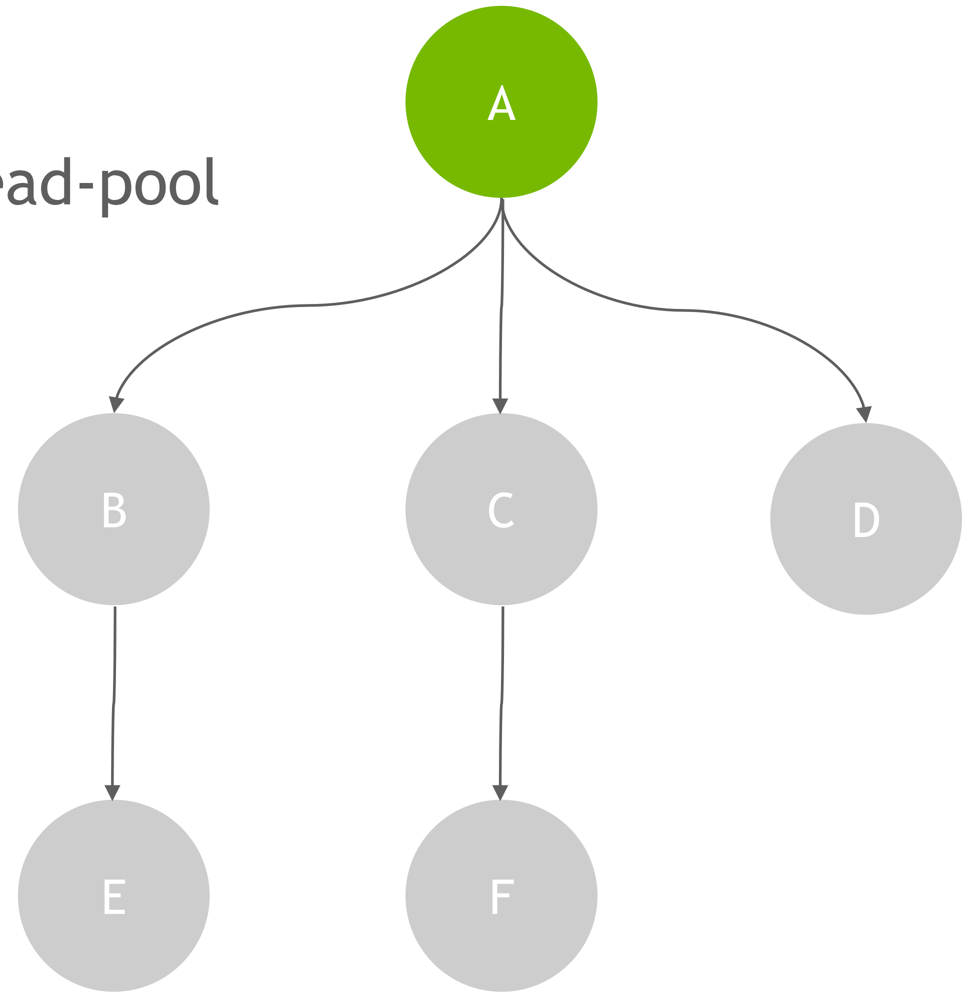
ScheduleReady(read_nodes)



WHAT HAPPENS IN SESSION.RUN

How Are Operations Scheduled

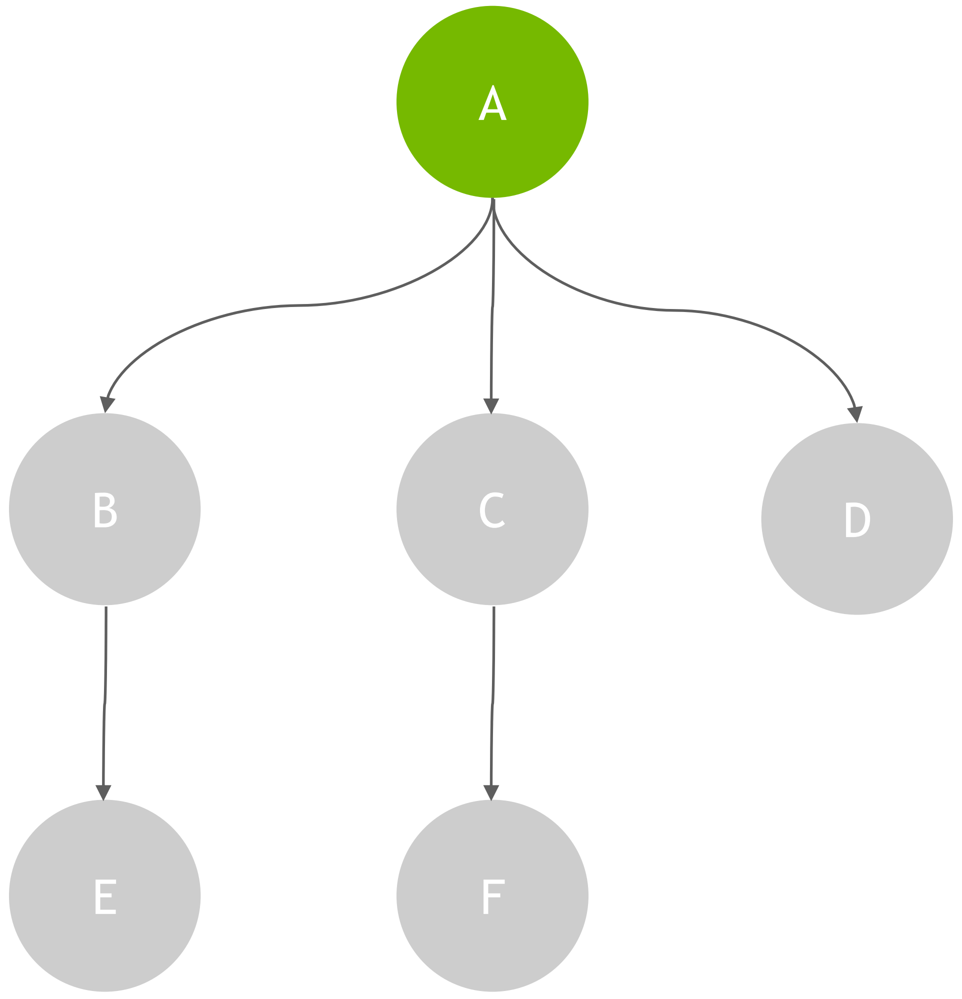
```
ScheduleReady(read_nodes):          # [A,]
    for node in read_nodes:         # [A,]
        Process (node)              # A, in a new thread got from thread-pool
```



WHAT HAPPENS IN SESSION.RUN

How Are Operations Scheduled

```
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)
```

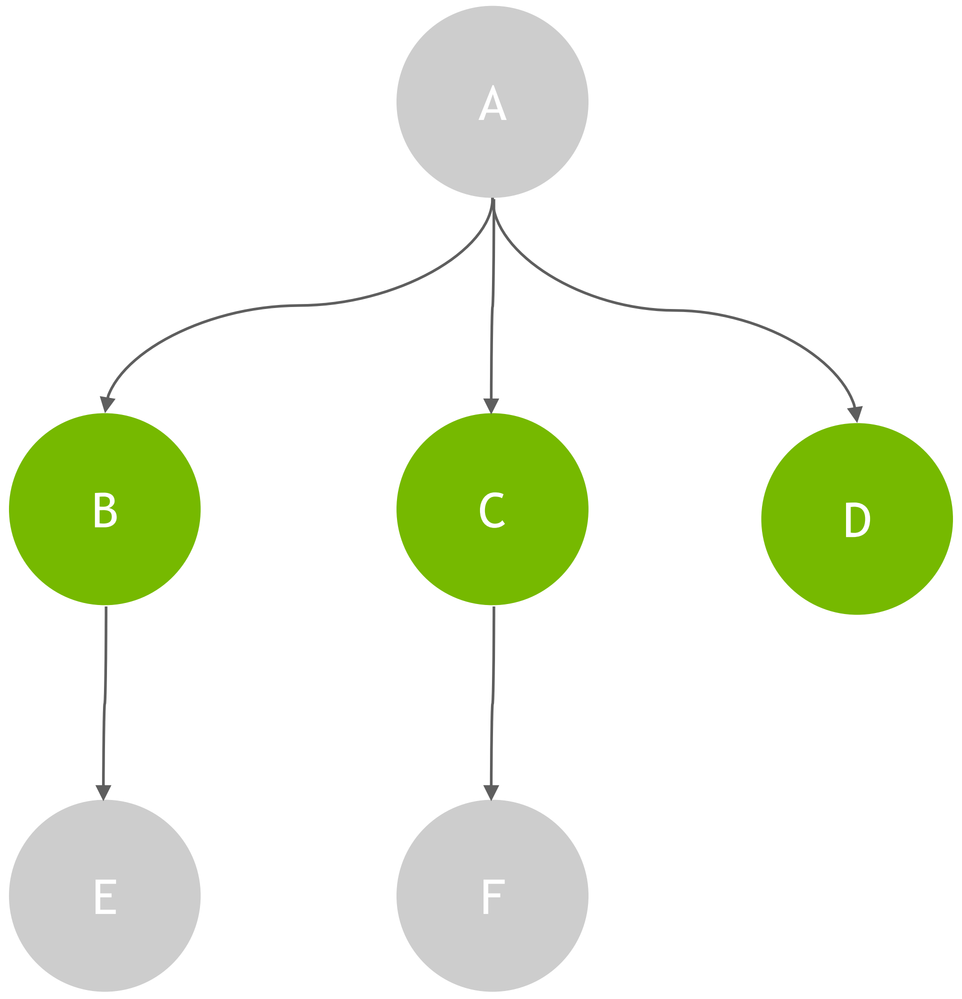


* The process of OpKernal (sync op)

WHAT HAPPENS IN SESSION.RUN

How Are Operations Scheduled

```
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)
```

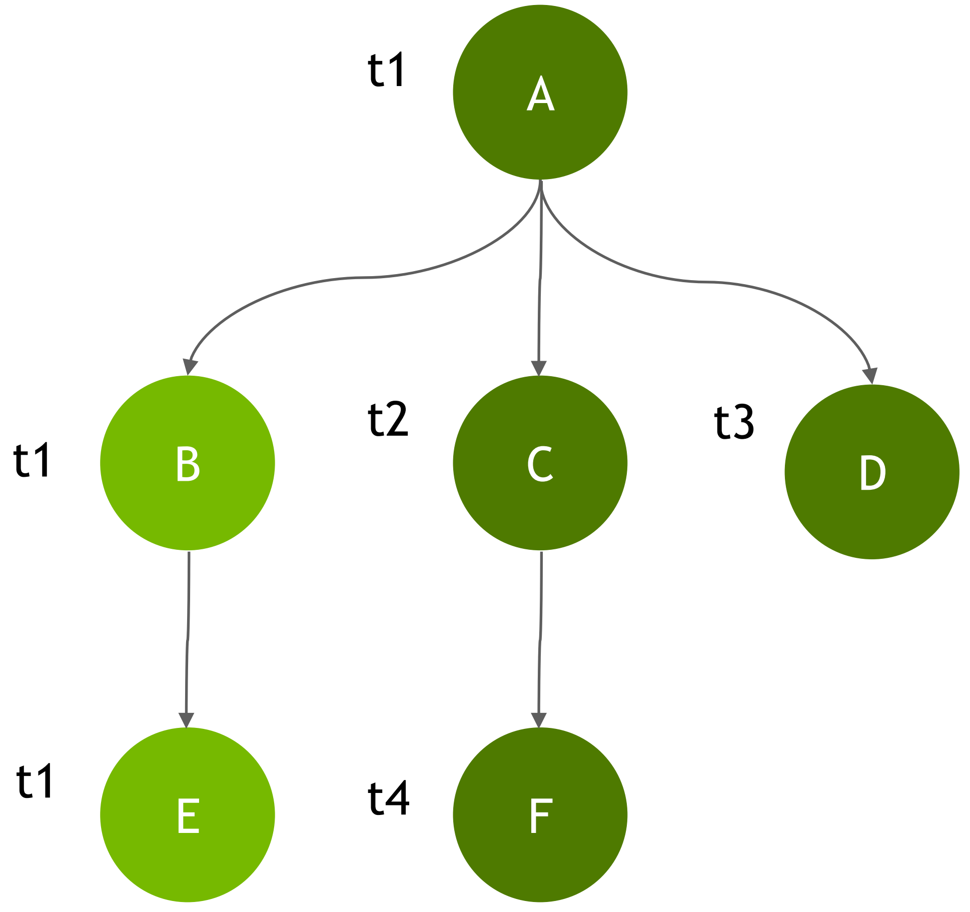


WHAT HAPPENS IN SESSION.RUN

How Operations Are Scheduled

Expensive

Inexpensive



Most CPU ops are expensive

GPU ops are inexpensive

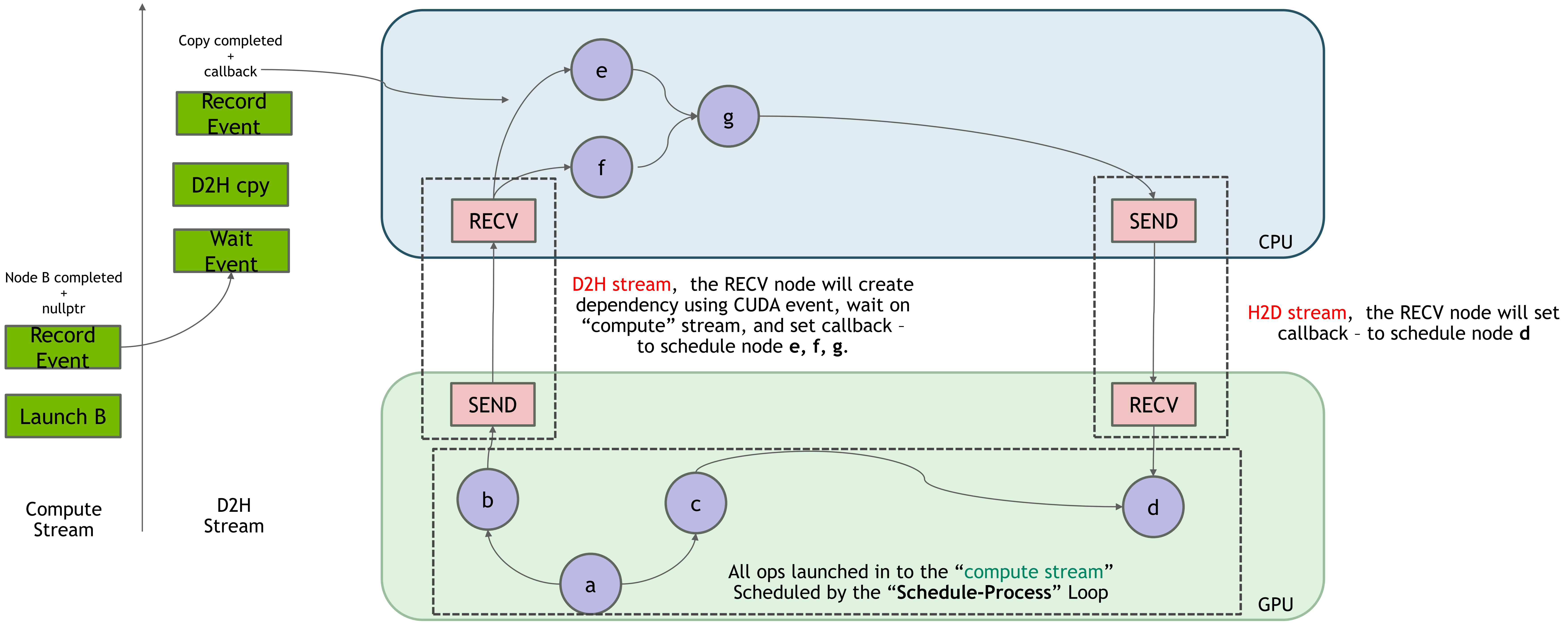
OpKernels executing on GPU tie very few resources on the CPU where the scheduler runs

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

WHAT HAPPENS IN SESSION.RUN

How TF Use CUDA Streams

A Group of Streams (1 for compute, 1 for h2d, 1 for d2h, several for d2d)



WHAT HAPPENS IN SESSION.RUN

CUDA Events Management

CUDA Events are used to create dependencies between streams

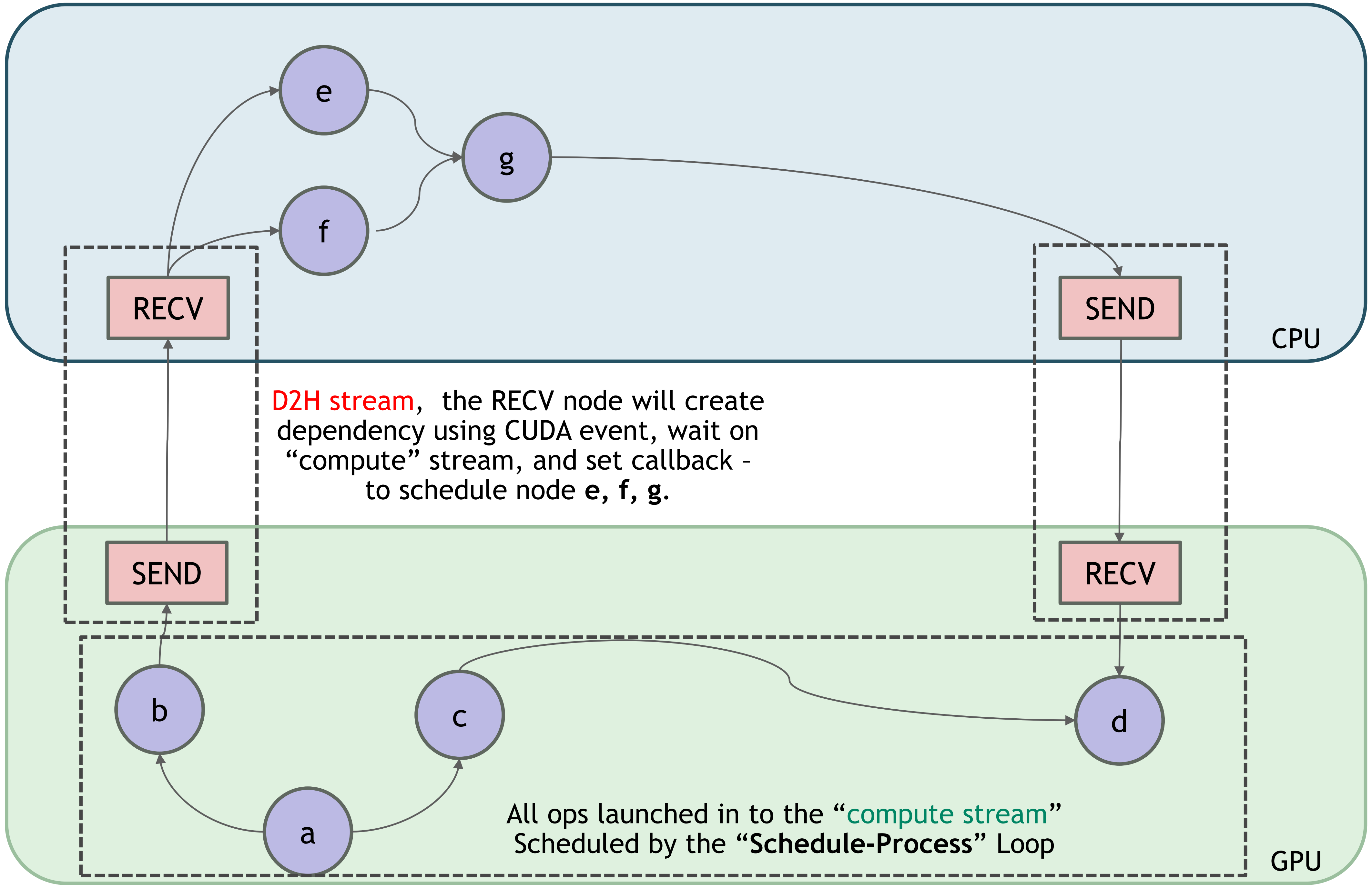
TF reuses events if possible

Maintain two lists of events:

- Free Events (not used)
- In-use Events

When need to record an Event, get one from free event list, or create a new one (append to in-use lists)

There is a thread periodically check the in-use event list, move completed events to free event list.



WHAT HAPPENS IN SESSION.RUN

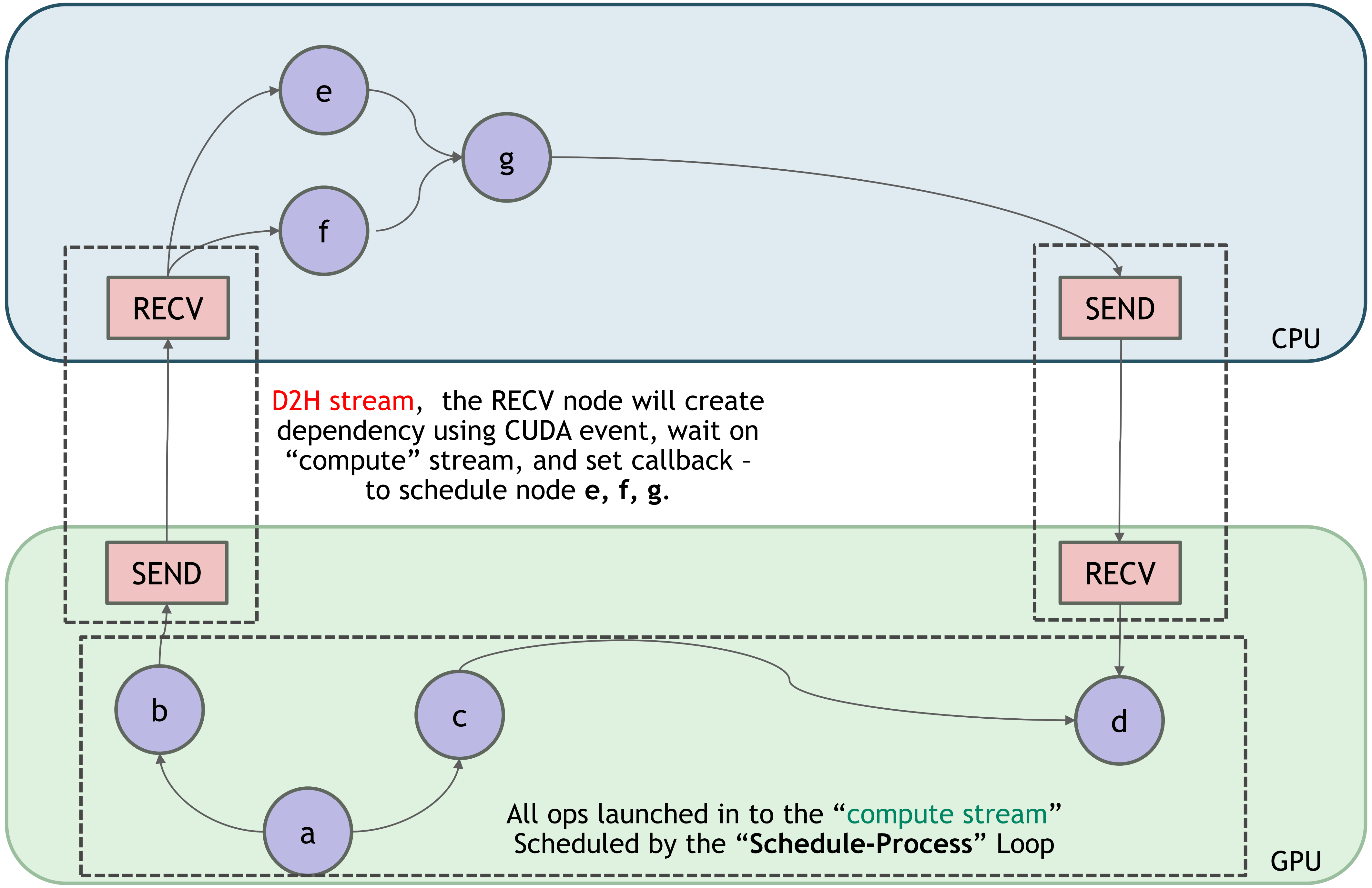
Stream Sync & Context Sync in TF

Stream Sync

- After scheduling is done, TF will sync all streams

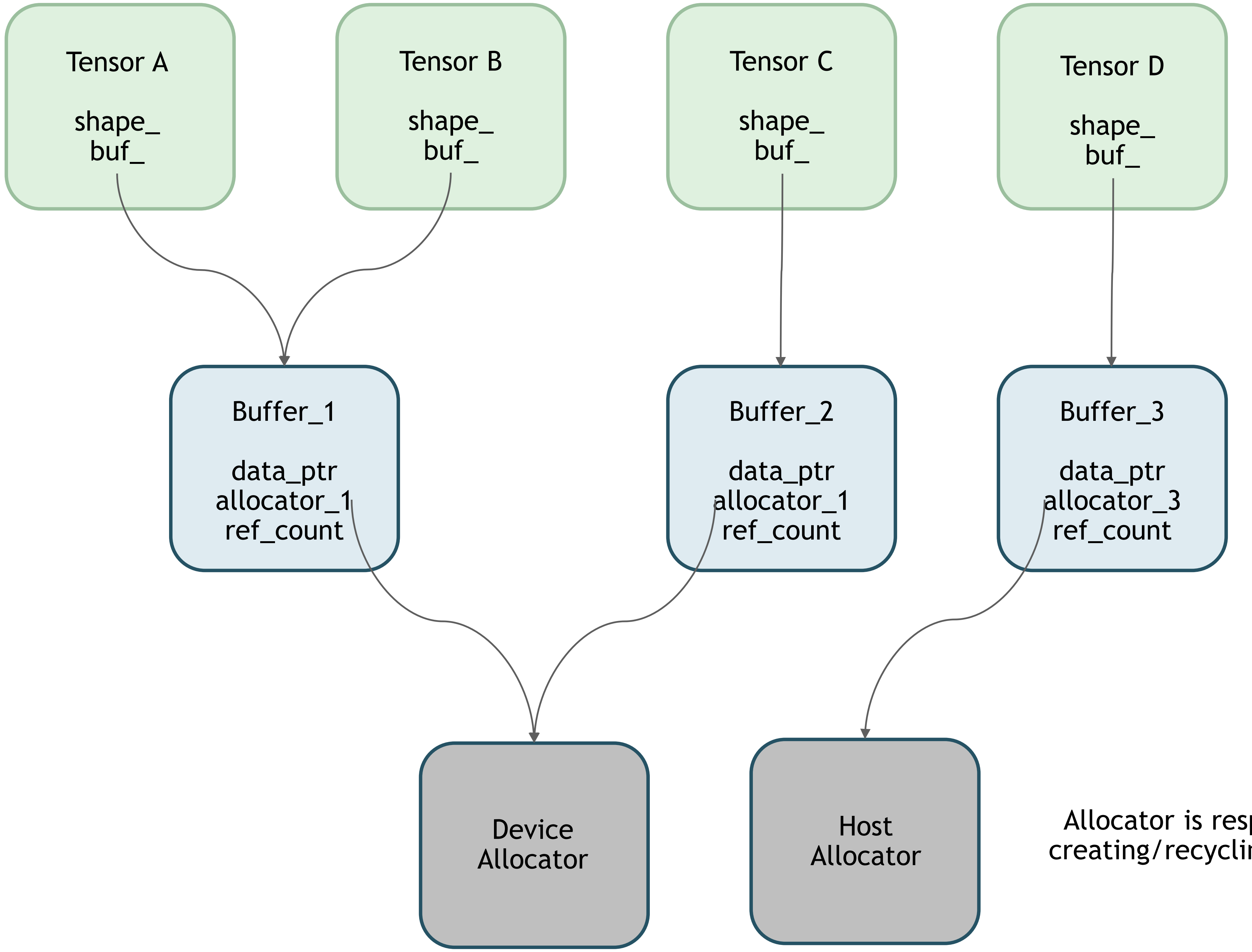
Ctx Sync (For debug)

- If “sync on every op” is set.
- If “sync on every driver call” is set.



WHAT HAPPENS IN SESSION.RUN

How does TF Manage Memory



Allocator is responsible for creating/recycling the buffer

WHAT HAPPENS IN SESSION.RUN

How does TF Manage Memory

Tensors are dynamically Allocated/Released

Tensor allocation in Conv op

```
template <typename Device, typename T>
class Conv2DOp : public BinaryOp<T> {

 public:
  explicit Conv2DOp(OpKernelConstruction* context) : BinaryOp<T>(context) { … }


  void Compute(OpKernelContext* context) override {

        …
        const Tensor& input = context->input(0);
        const Tensor& filter = context->input(1);
        …

        Tensor* output = nullptr;
        OP_REQUIRES_OK(context, context->allocate_output(0, out_shape, &output));
        ---

        auto* stream = context->op_device_context()->stream();
        ---

        // no need to do stream synchronization ! TF handles the sync and dependencies automatically
        }
```

```
Process(node):

    inline_ready = [node,]

    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)
```

CONTENT

INTEGRATE CUDA GRAPH INTO TENSORFLOW

Overview

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

- Simplify the capture
- No need to create extra CUDA Events for capturing

Disable syncs during session runs

- Syncs are not necessary, as in capture mode, all GPU operations are not executed but just recorded

Memory management

- Hold the Tensors allocated during capturing
- Tensor reusing

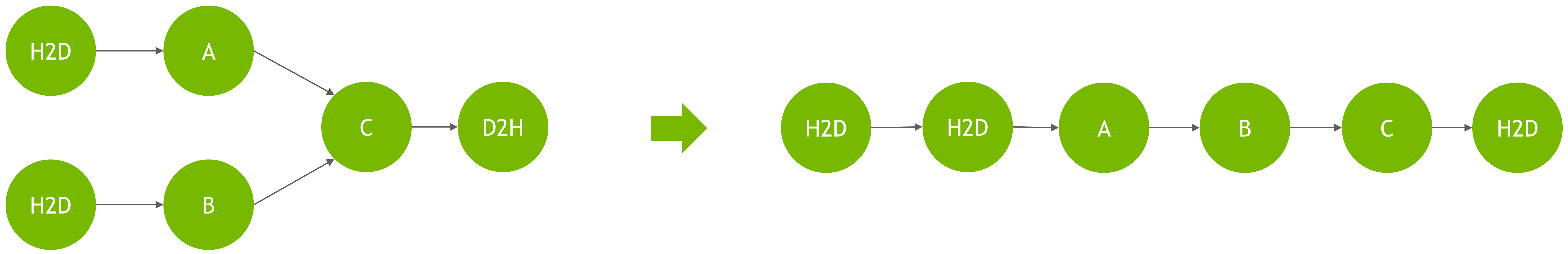
INTEGRATE CUDA GRAPH INTO TENSORFLOW

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 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;
}
```

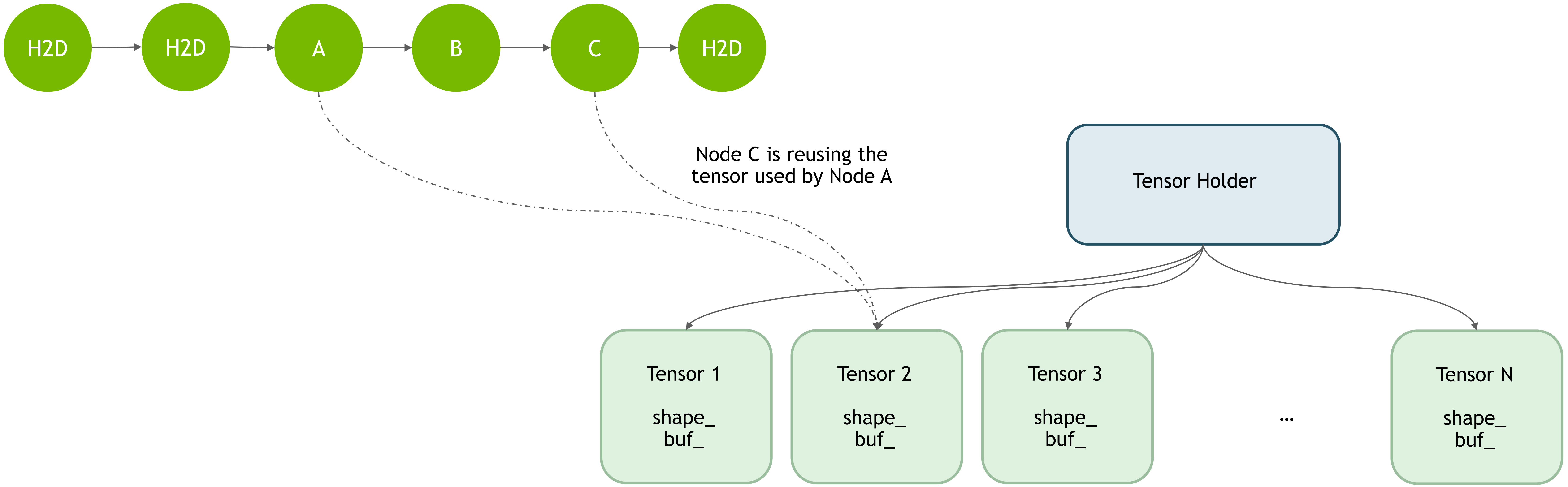
```
void BaseGPUDevice::ResetStreams(){
    if(! stream_catpure_mode_) return;

    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)



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)

The reusing is not global optimal

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 H2D/D2H nodes
    - ❑ Condition 1: Each H2D node in the graph, is either corresponding to an input tensor or const CPU tensor which is held by Tensor Holder
    - ❑ Condition 2: Each D2H node in the graph, is corresponding to an output tensor
    - ❑ If either condition 1 or condition 2 fails, the graph is invalid (there may be uncaptured CPU operations)
- ❑ 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

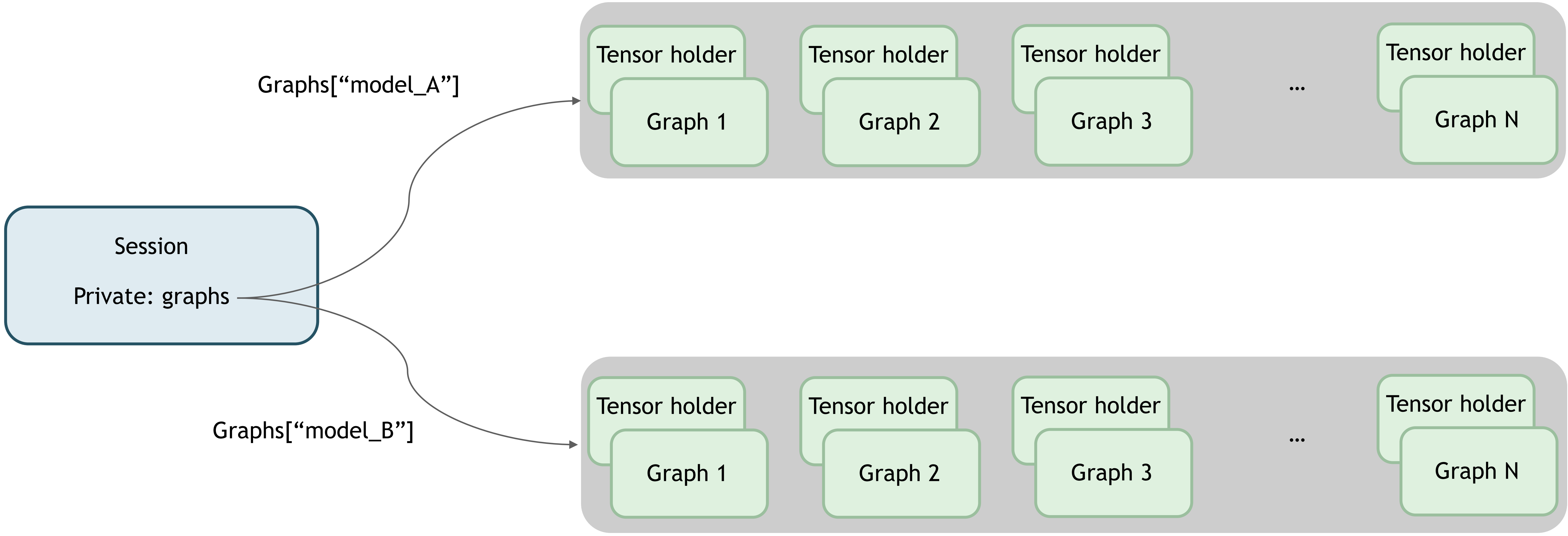
INTEGRATE CUDA GRAPH INTO TENSORFLOW

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's name and the graph index

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



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)

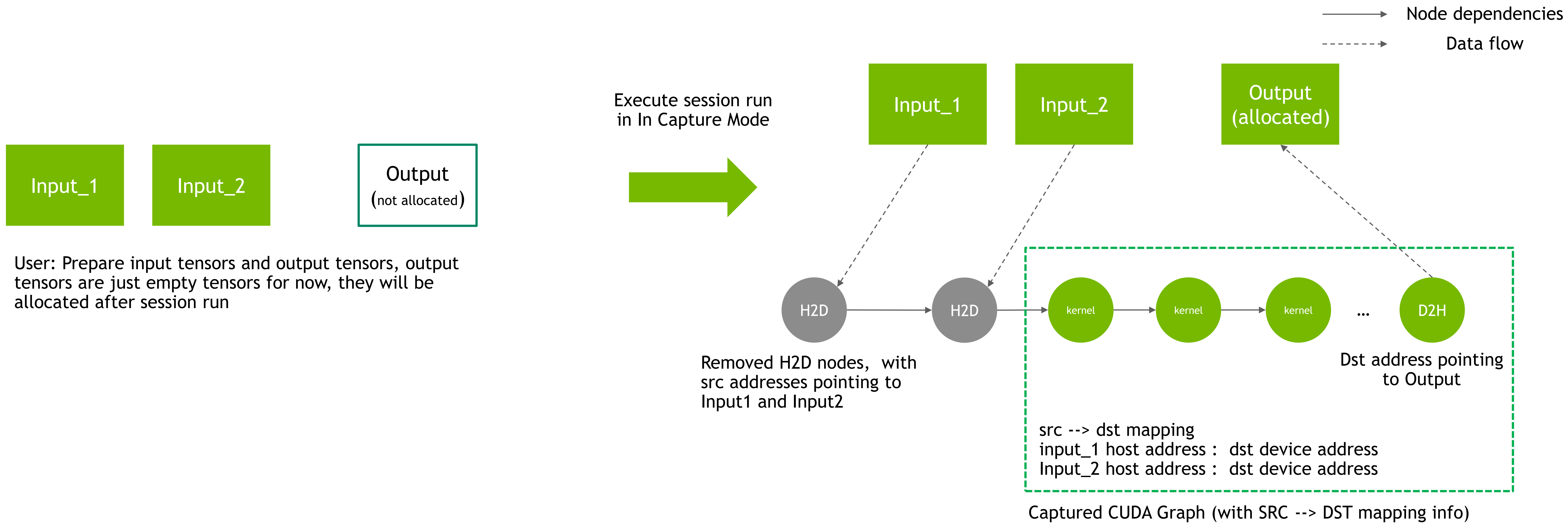
Launch graphs into different streams, session->RunCudaGraph("model_name", graph_idx, stream)

Do H2H copy for the results (optional)

INTEGRATE CUDA GRAPH INTO TENSORFLOW

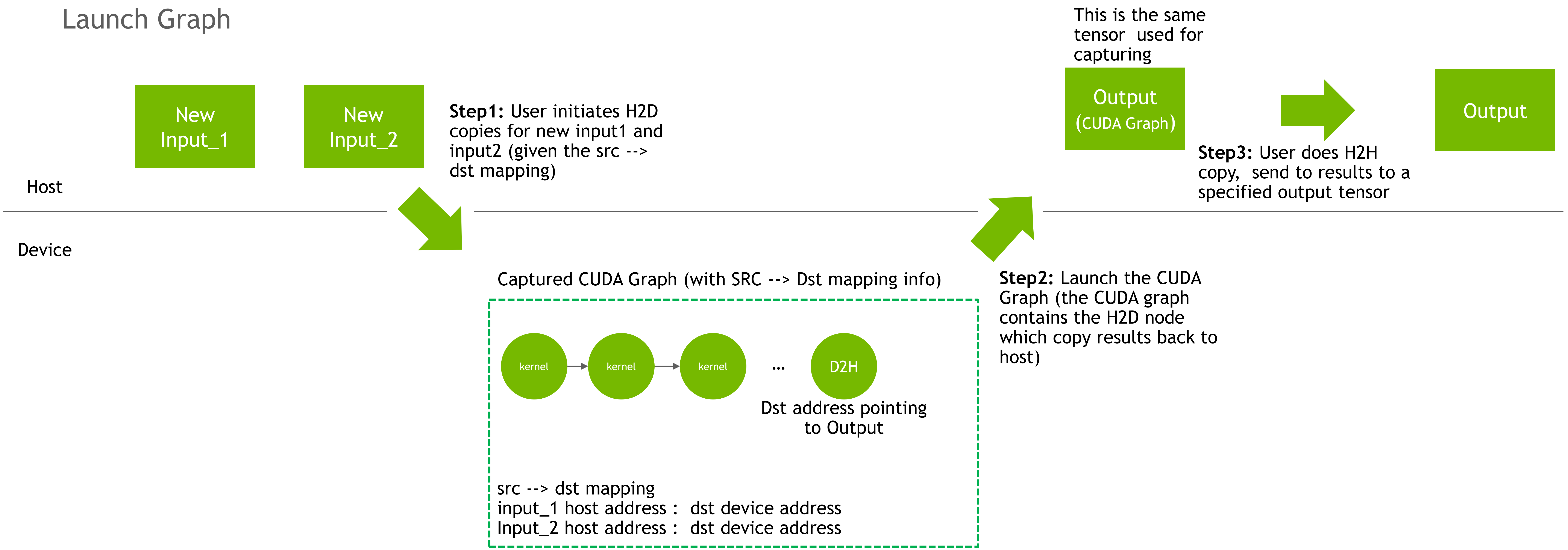
Workflow

Capture Graph



INTEGRATE CUDA GRAPH INTO TENSORFLOW

Workflow



CONTENT

What’s CUDA Graph

How to Use CUDA Graph

What happens in TF session run

Integrate CUDA Graph into TensorFlow

Use case: Alibaba Search Recommendation System

CONTENT

USE CASE: ALIBABA SEARCH RECOMMENDATION SYSTEM

Recommendation Network

Sparse Part

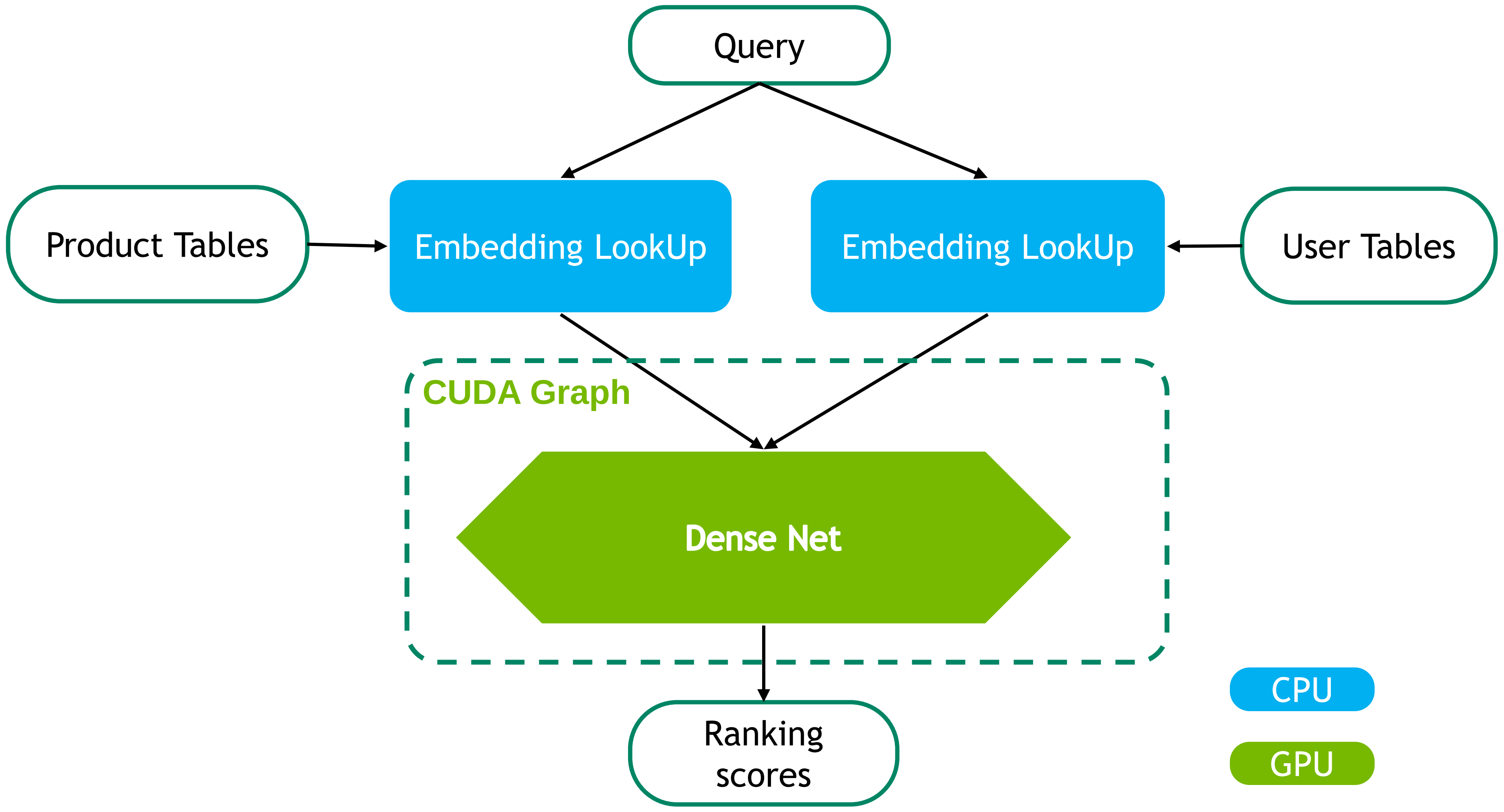
- Embedding lookup

Dense Part

- MLP or MultiHeadAttention ……
- Consist of dense computations
- Suffer op launch overhead problem

How:

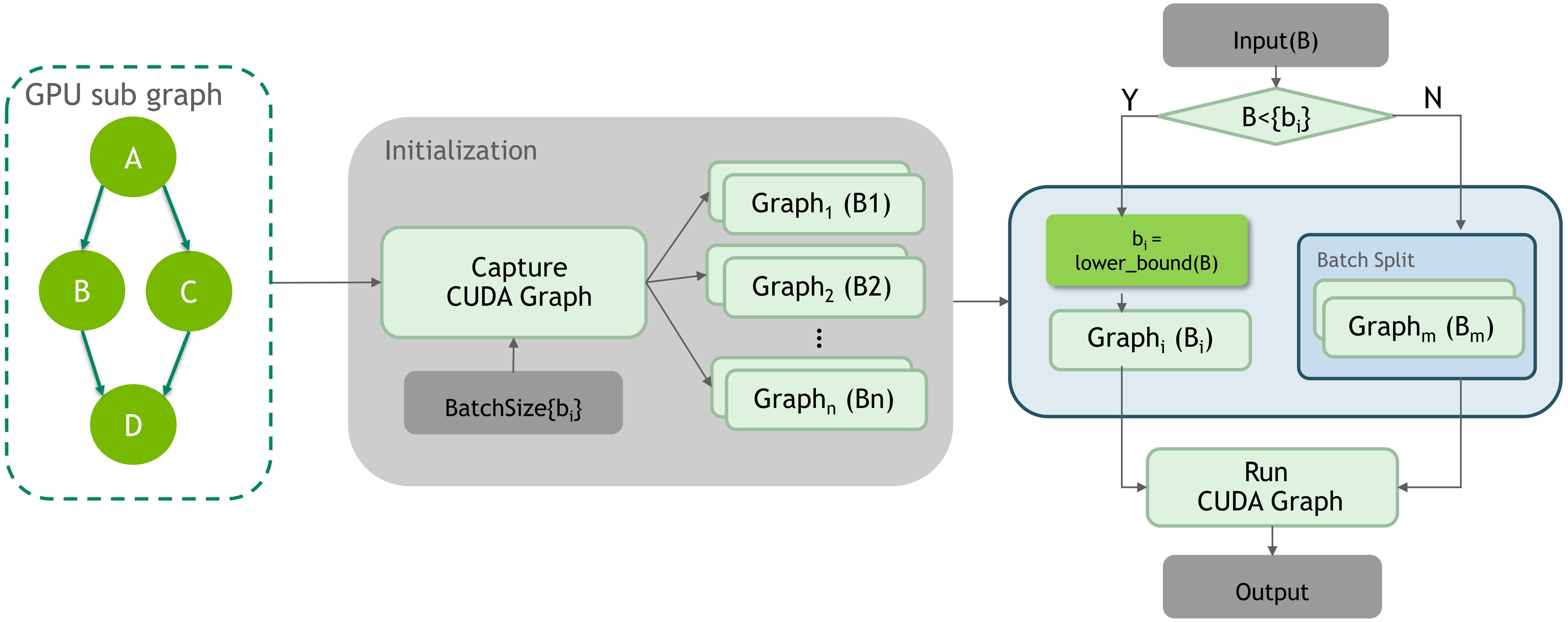
- ❑ Place sparse part on CPU
- ❑ Place dense part on GPU
- ❑ Optimize dense part with CUDA Graph



USE CASE: ALIBABA SEARCH RECOMMENDATION SYSTEM

Workflow

- **Divide model:** Divide model into CPU sub graph and GPU sub graph (CUDA Graph)
- **Initialization (Offline model):** Capture all GPU activities and stored as graph in GPU memory(N batchsize, M streams → M * N graphs)
- **Execution (Online model):** Select graph and run in CUDA Graph model



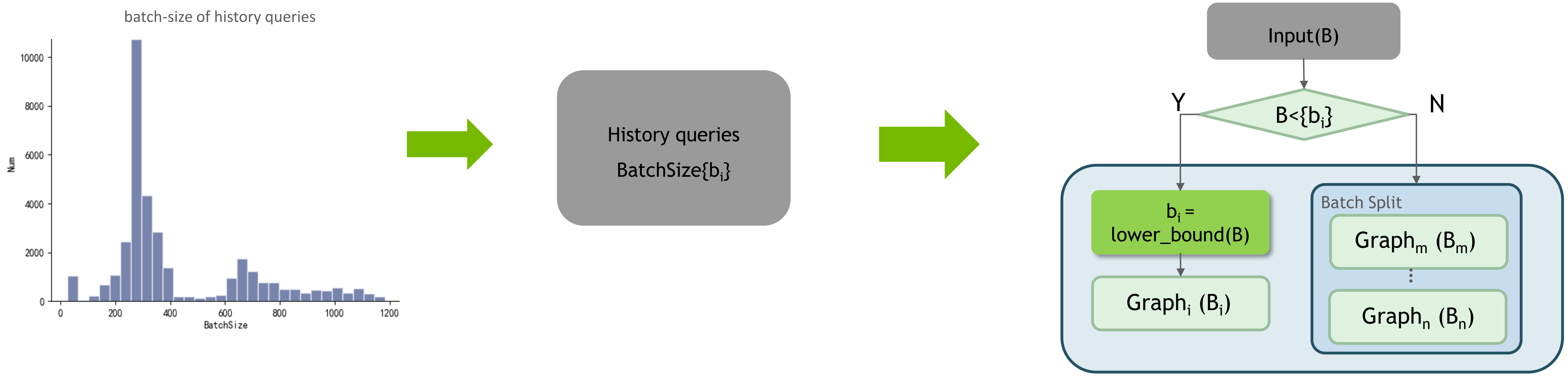
USE CASE: ALIBABA SEARCH RECOMMENDATION SYSTEM

Dynamic Memory Management

- Support dynamic memory management

- According to the distribution of batch-size of history queries, selects N batch-sizes ({Bi}={B1, B2 … Bn}) as typical batch-sizes.

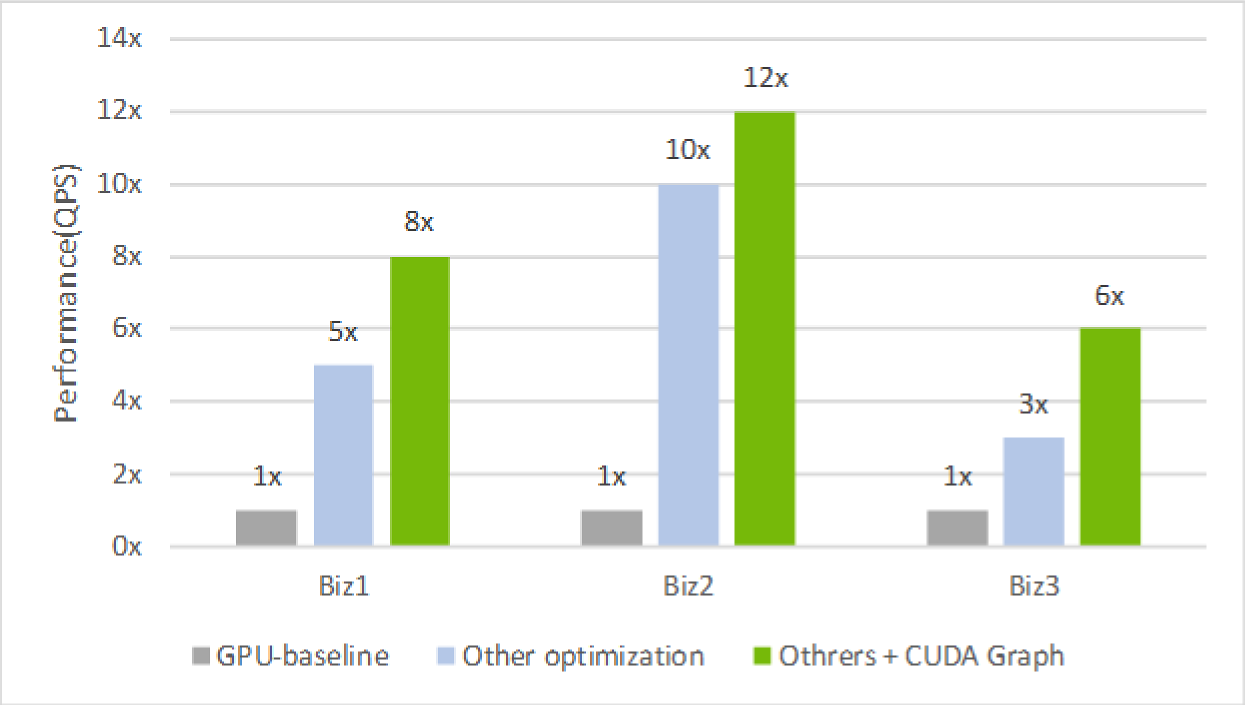
  eg. [300, 400, 800, 1200]
- Select one graph (padding) or select multiple graphs (batch split)

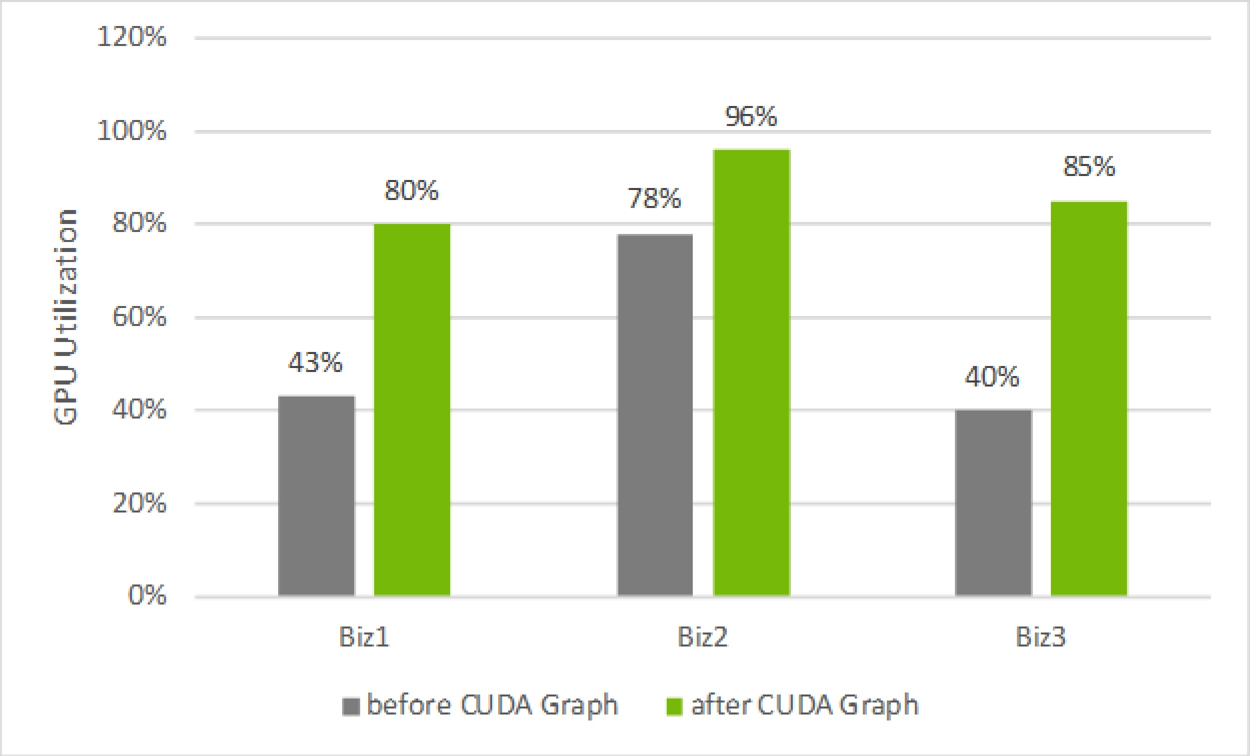


NVIDIA

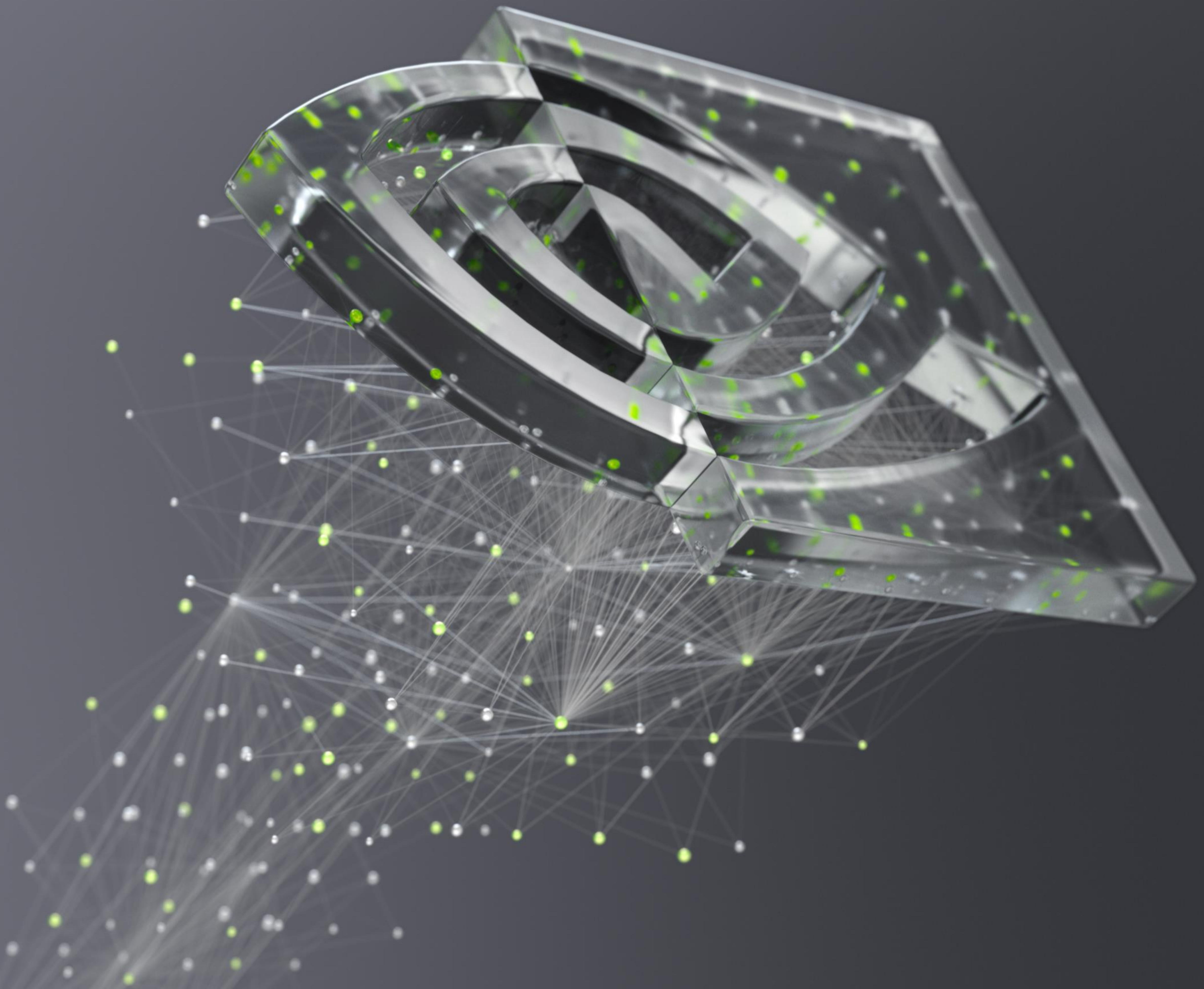
USE CASE: ALIBABA SEARCH RECOMMENDATION SYSTEM

Experimental Results





- QPS(Query per second): 50% improvement on average
- Rt(Response time): 10% reduction on average
- GPU utilization : >= 80%



NVIDIA