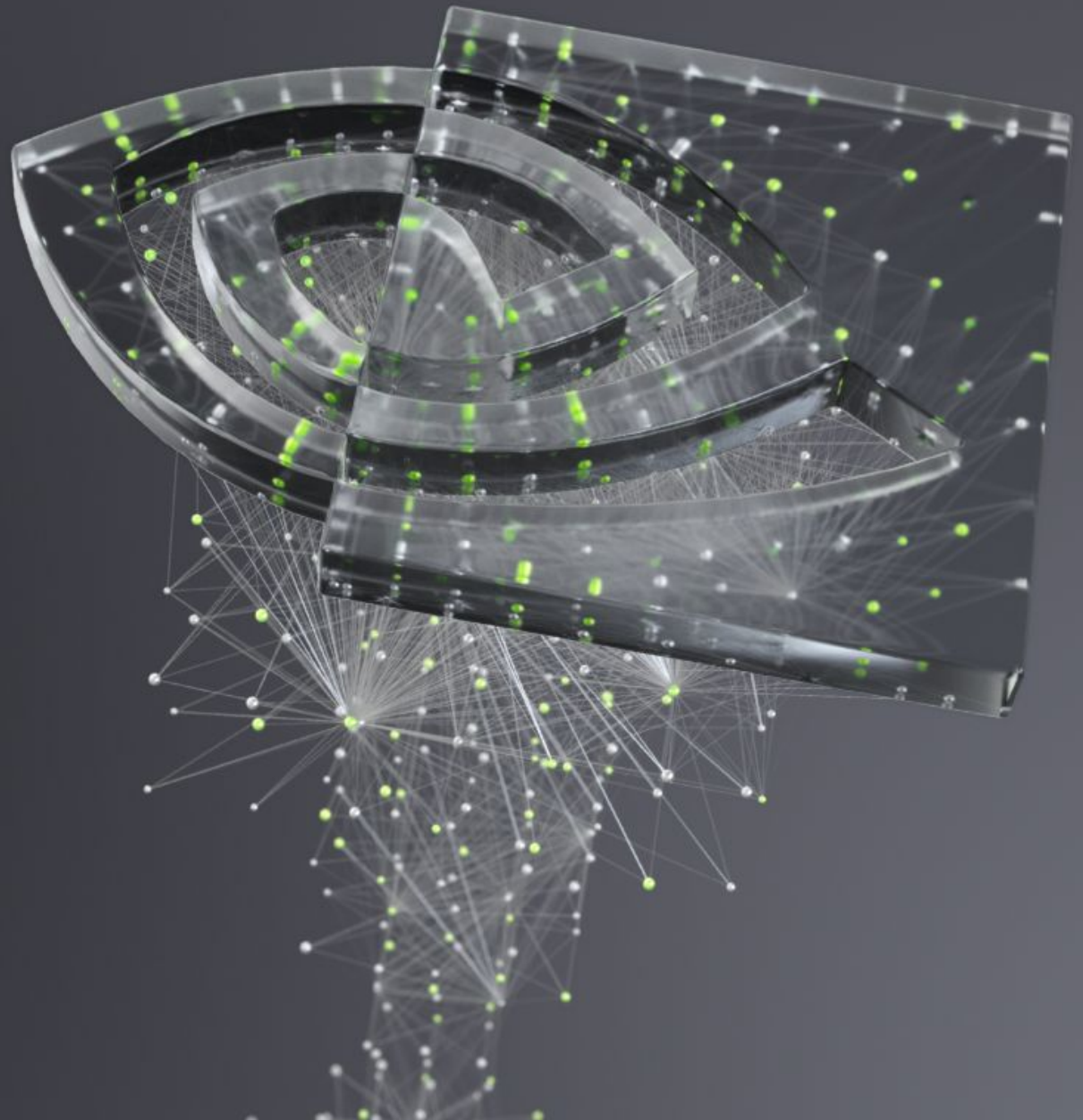
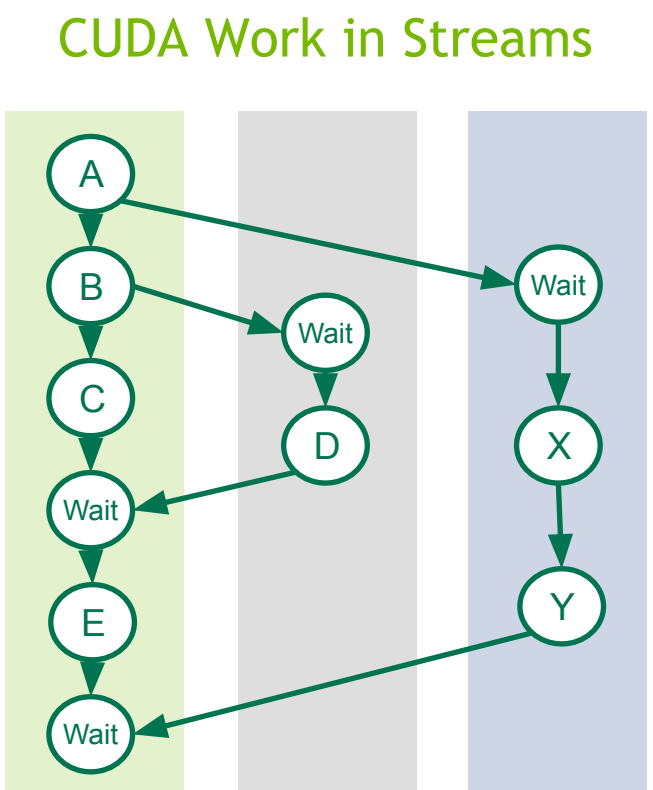


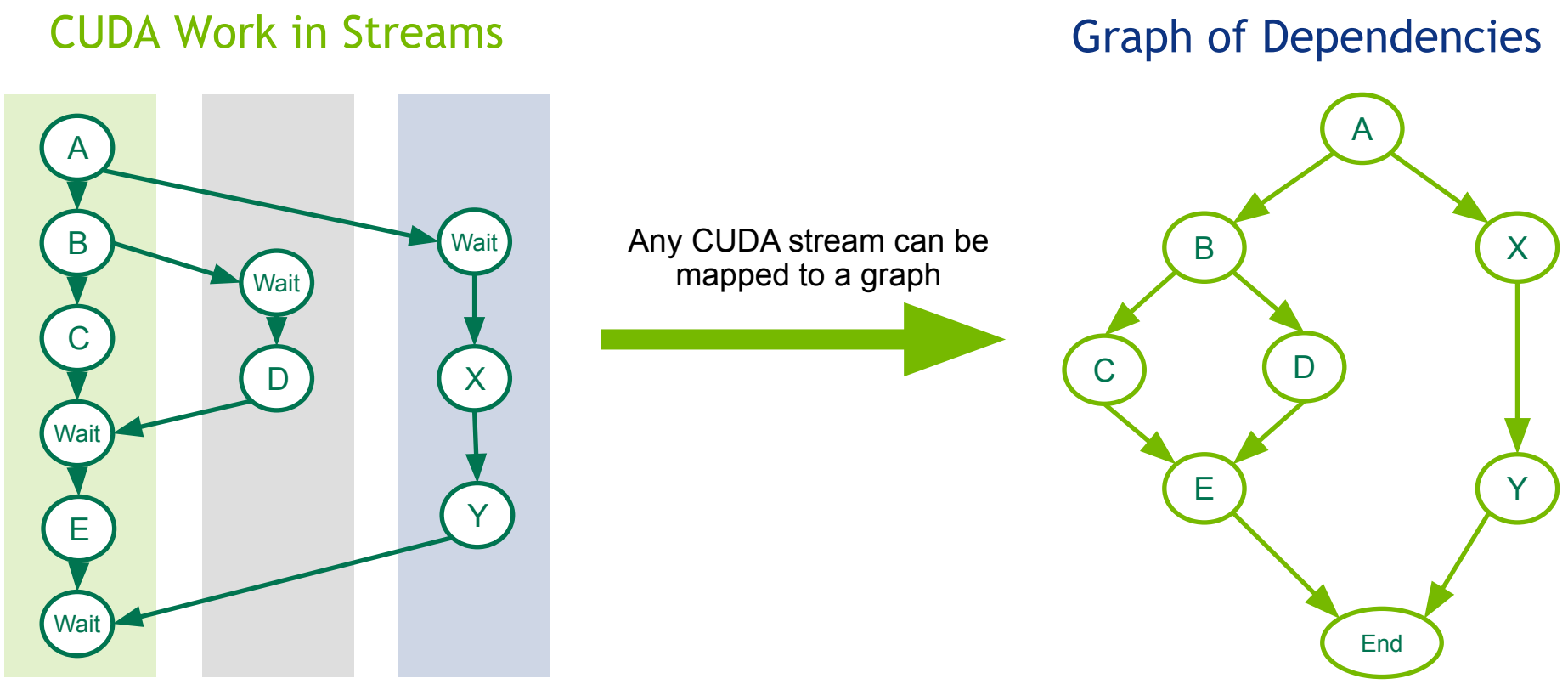
CUDA Graphs

NVIDIA Corporation

ALL CUDA WORK FORMS A GRAPH



ALL CUDA WORK FORMS A GRAPH



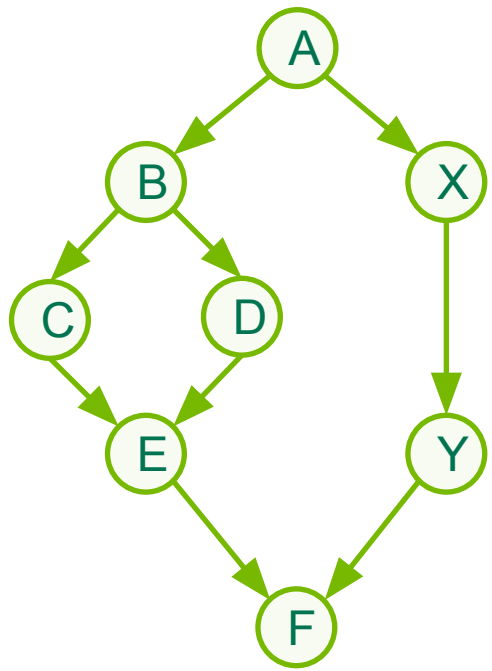
DEFINITION OF A CUDA GRAPH

A graph node is any **asynchronous** CUDA operation

Sequence of operations, connected by dependencies.

Operations are one of:

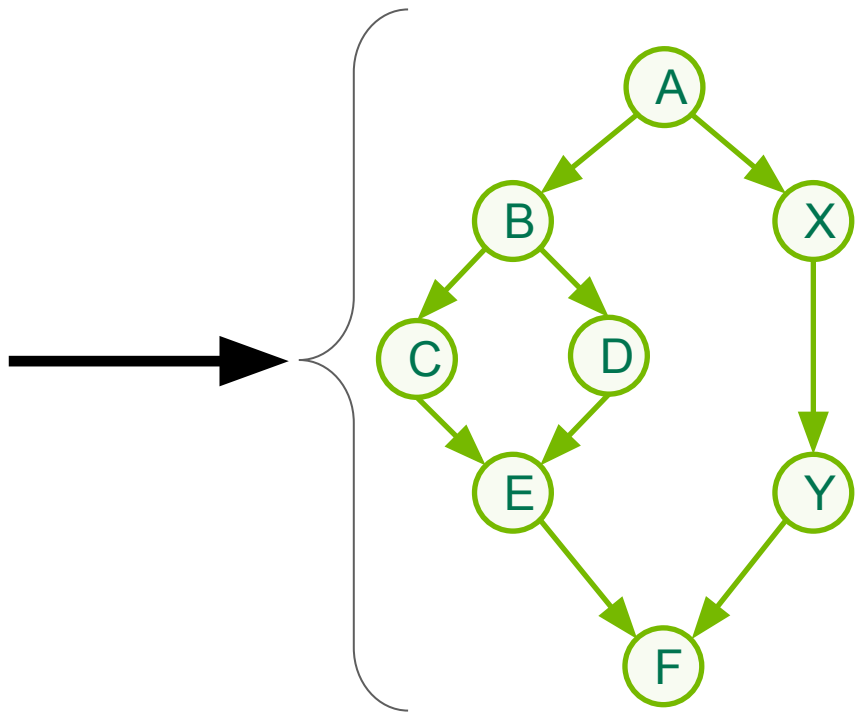
Kernel Launch	CUDA kernel running on GPU
CPU Function Call	Callback function on CPU
Memcopy/Memset	GPU data management
Memory Alloc/Free	Inline memory allocation
Sub-Graph	Graphs are hierarchical



NEW EXECUTION MECHANISM

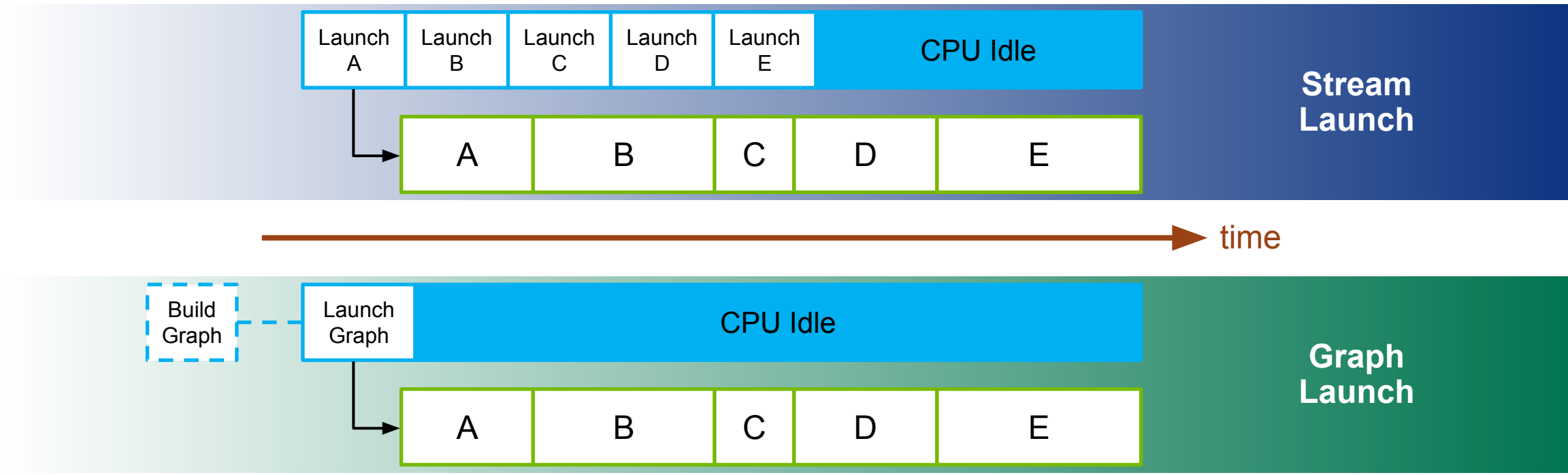
Graphs Can Be Generated Once Then Launched Repeatedly

```
for(int i=0; i<1000; i++) {
    launch_graph( G );
}
```



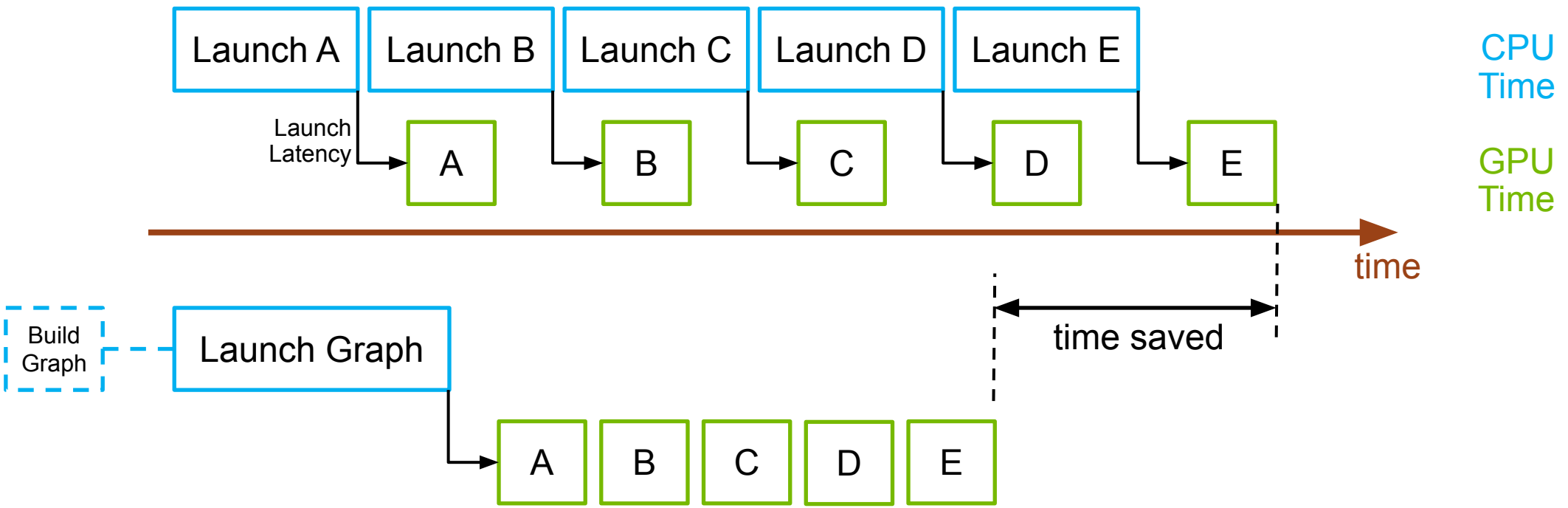
FREE UP CPU RESOURCES

Release CPU Time For Lower Power, or Running Other Work



LAUNCH OVERHEAD REDUCTION

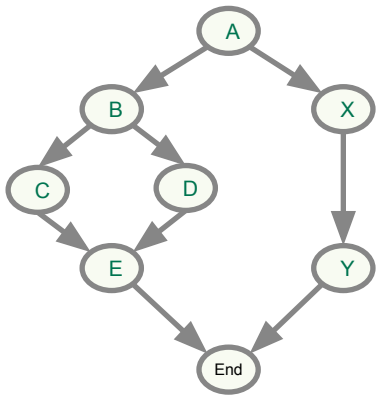
Graph launch submits all work at once, reducing CPU cost



When kernel runtime is short, execution time is dominated by CPU launch cost

THREE-STAGE EXECUTION MODEL

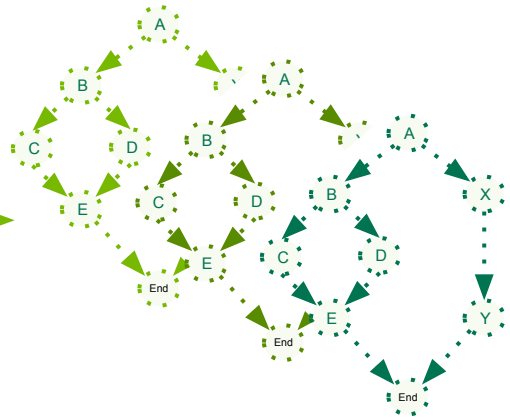
Define



Single Graph “Template”

Created in host code,
or loaded from disk,
or built up from libraries

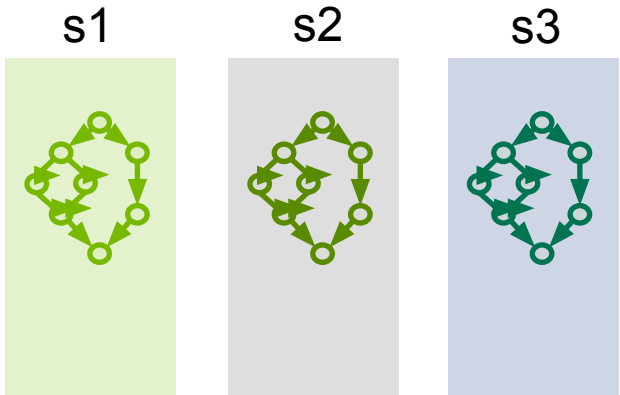
Instantiate



Multiple “Executable Graphs”

Snapshot of template
Sets up & initializes GPU
execution structures
(create once, run many times)

Execute

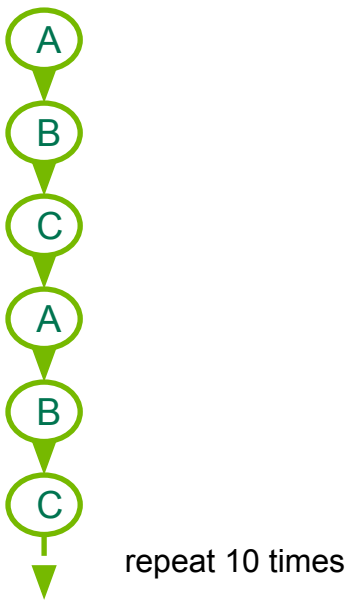


Executable Graphs
Running in CUDA Streams

Concurrency in graph
is not limited by stream
(see later)

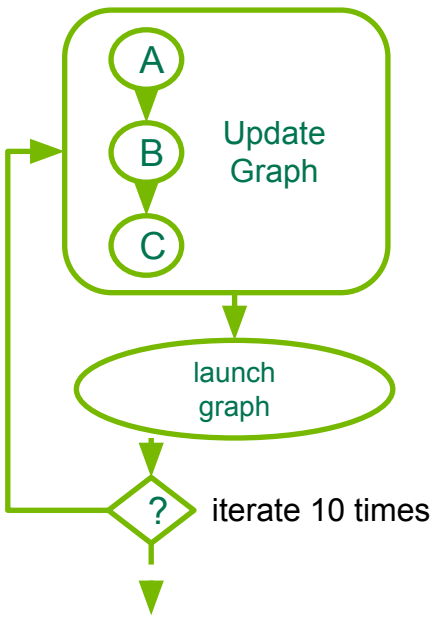
MODIFYING GRAPHS IN-PLACE

Stream Launch



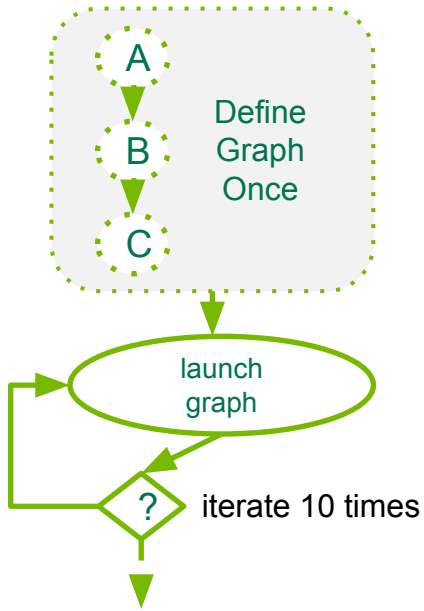
Parameters: **may** change
Topology: **may** change

Graph Update



Parameters: **may** change
Topology: **may not** change

Graph Re-Launch



Parameters: **may not** change
Topology: **may not** change

PROGRAMMING MODEL

ASYNCHRONOUS OPERATIONS ONLY

Typically Shows Up During Stream Capture

Stream Capture

- Very convenient way of creating a graph from existing library calls (see later slide)
- Records operations without actually launching a kernel
- Library must call an API to tell if kernels are being captured instead of launched

Problem if library calls *cudaStreamSynchronize()* or any other synchronous operation.

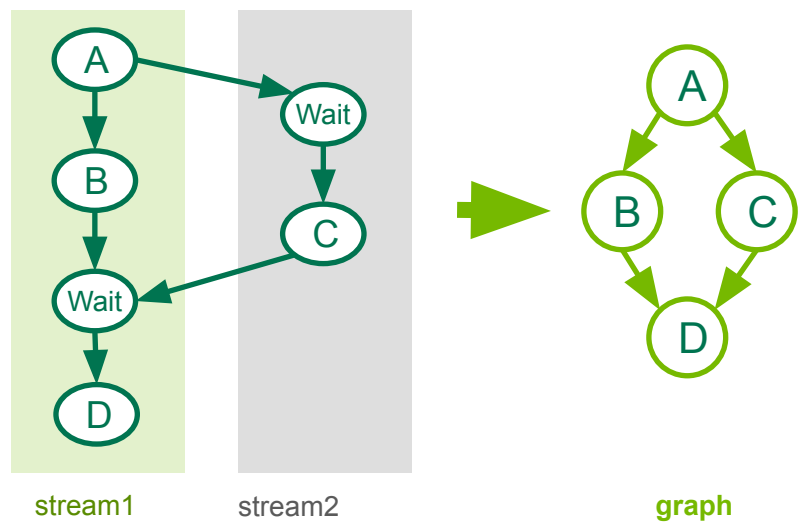
Capture is not launching anything so synchronize cannot wait for anything.

Capture operation fails.

CAPTURE CUDA STREAM WORK INTO A GRAPH

Construct a graph from normal CUDA stream syntax

```
// Start by initiating stream capture
cudaStreamBeginCapture(&stream1);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```

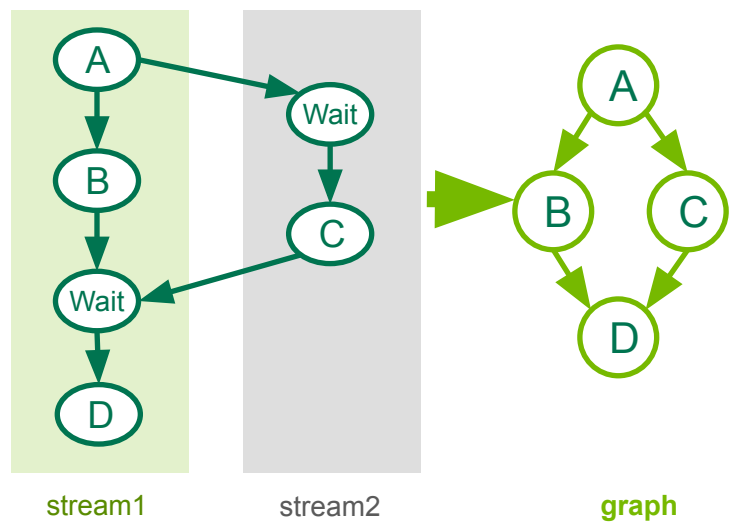


CAPTURE CUDA STREAM WORK INTO A GRAPH

Construct a graph from normal CUDA stream syntax

```
// Start by initiating stream capture
cudaStreamBeginCapture(&stream1);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
      cudaStreamWaitEvent(str
            eam2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
      cudaStreamWaitEvent(str
            eam1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```

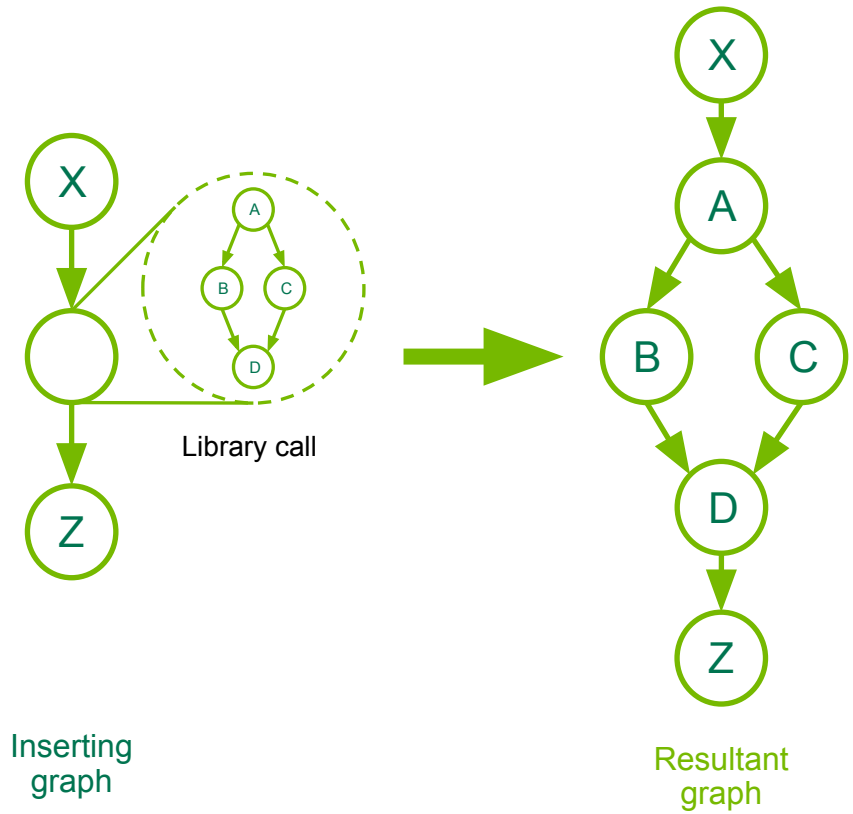
Capture follows inter-stream dependencies to create forks & joins



CAPTURE EXTERNAL WORK

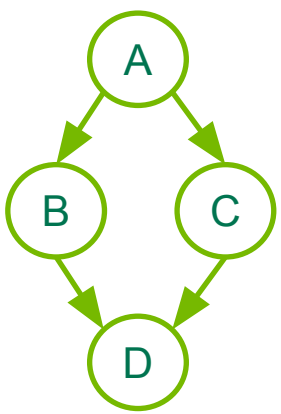
Stream Capture

```
// Start by initiating stream capture
cudaStreamBeginCapture(&stream);

// Captures my kernel launches and inside library calls
X<<< ..., stream >>>();
libraryCall(stream);        // Launches A, B, C, D
Z<<< ..., stream >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream, &graph);
```

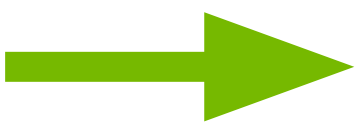


CREATE GRAPHS DIRECTLY

Map Graph-Based Workflows Directly Into CUDA



Graph from framework



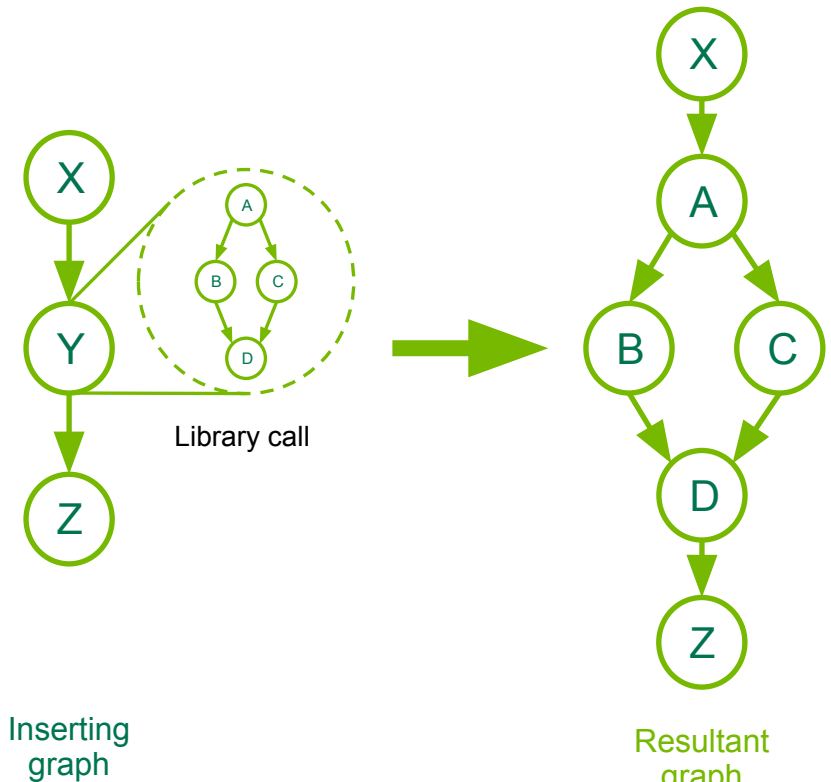
```
// Define graph of work + dependencies
cudaGraphCreate(&graph);

cudaGraphAddNode(graph, kernel_a, {}, ...);
cudaGraphAddNode(graph, kernel_b, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_c, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_d, { kernel_b, kernel_c }, ...);

// Instantiate graph and apply optimizations
cudaGraphInstantiate(&instance, graph);

// Launch executable graph 100 times
for(int i=0; i<100; i++)
      cudaGraphLaunch(instance, stream);
```

(Full list of API calls in the CUDA Docs)

COMBINING GRAPH & STREAM WORK

Capturing Streams Into An Existing Graph

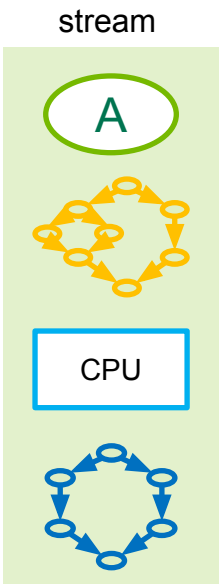
```
// Create root node of graph via explicit API
cudaGraphAddNode(main_graph, X, {}, ...);

// Capture the library call into a subgraph
cudaStreamBeginCapture(&stream);
libraryCall(stream);          // Launches A, B, C, D
cudaStreamEndCapture(stream, &library_graph);

// Insert the subgraph into main_graph as node "Y"
cudaGraphAddChildGraphNode(Y, main_graph, { X } ... libraryGrpah);

// Continue building main graph via explicit API
cudaGraphAddNode(main_graph, Z, { Y }, ...);
```



GRAPH EXECUTION SEMANTICS

Order Graph Work With Other Non-Graph CUDA Work

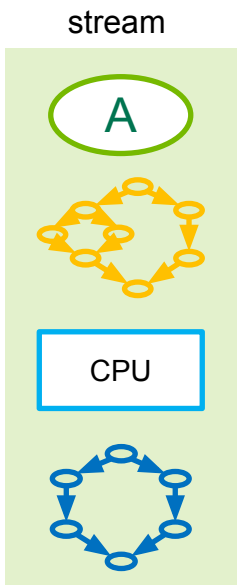
```
launchWork(cudaGraphExec_t i1, cudaGraphExec_t i2,
           CPU_Func cpu, cudaStream_t stream) {

    A <<< 256, 256, 0, stream >>>();        // Kernel launch
    cudaGraphLaunch(i1, stream);            // Graph launch
    cudaStreamAddCallback(stream, cpu);     // CPU callback
    cudaGraphLaunch(i2, stream);            // Graph launch

    cudaStreamSynchronize(stream);
}
```



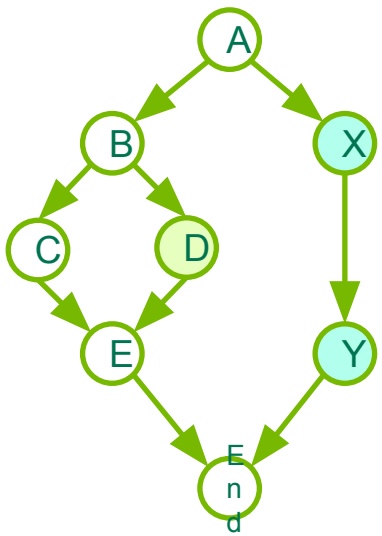
If you can put it in a CUDA stream, you can run it together with a graph

GRAPHS IGNORE STREAM SERIALIZATION RULES

Launch Stream Is Used Only For Ordering With Other Work



Branches in graph still execute concurrently even though graph is launched into a stream

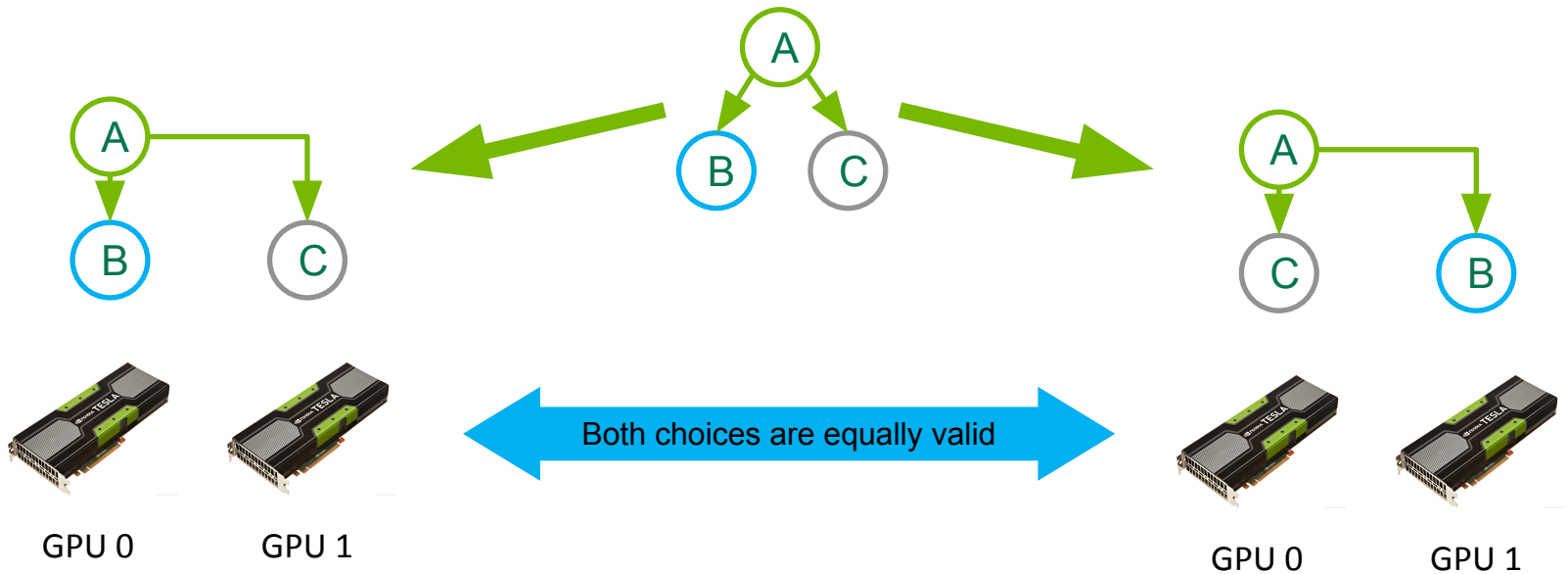


WHAT CAN YOU NOT DO WITH IT?

NO AUTOMATIC PLACEMENT

User Must Define Execution Location For Each Node

If fork in graph can run on 2 GPUs,
how do we pick what runs where?



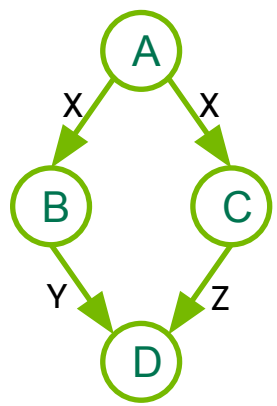
Best choice may depend on data locality – unknown at execution layer

EXECUTION DEPENDENCIES

CUDA Dependencies are Execution Dependencies

Data dependency graph definition

Task	Inputs	Outputs
A	none	X
B	X	Y
C	X	Z
D	Y, Z	none



Execution dependency graph definition

```
cudaGraphAddNode(graph, A, {}, ...);
cudaGraphAddNode(graph, B, { A }, ...);
cudaGraphAddNode(graph, C, { A }, ...);
cudaGraphAddNode(graph, D, { B, C }, ...);
```

All data dependencies can trivially be mapped to execution dependencies, but
Not all execution dependencies can be mapped to data dependencies

WHAT CAN YOU DO WITH IT?

RAPID RE-ISSUE OF WORK

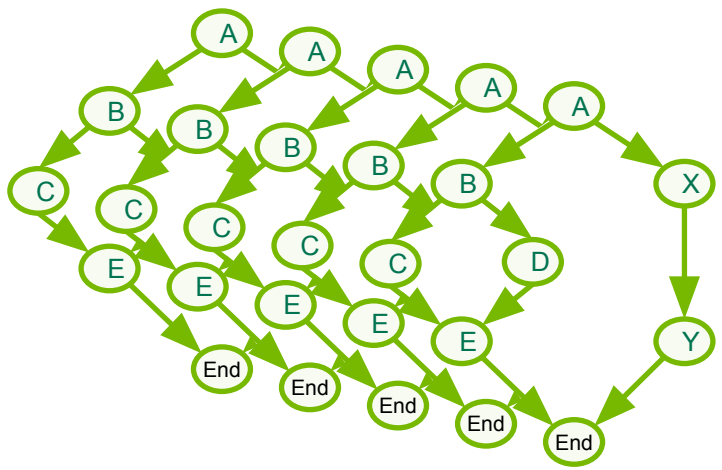
Graphs Can Be Generated Once And Executed Repeatedly

Cost of graph instantiation

≈

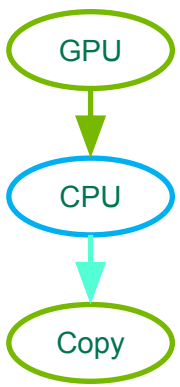
Cost of normal launch

```
for(int i=0; i<5; i++) {
    launch_graph( G );
}
```



HETEROGENEOUS NODE TYPES

Graph Nodes Include GPU Work, CPU Work and Data Movement



Heterogeneous Execution

Data management **may** be optimized transparently

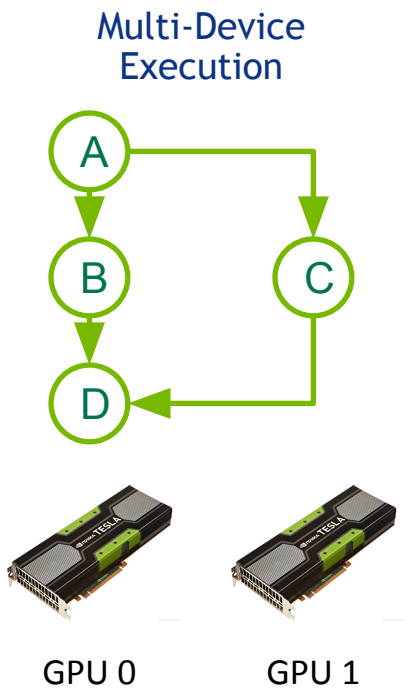
- Prefetching
- Read duplication
- Subdivision to finer granularity

Optimize for **bandwidth** and **latency** of memory access

Optimize for bandwidth of **interconnect** (PCI, QPI, NVLink)

CROSS-DEVICE DEPENDENCIES

CUDA Can Sync Multiple GPUs



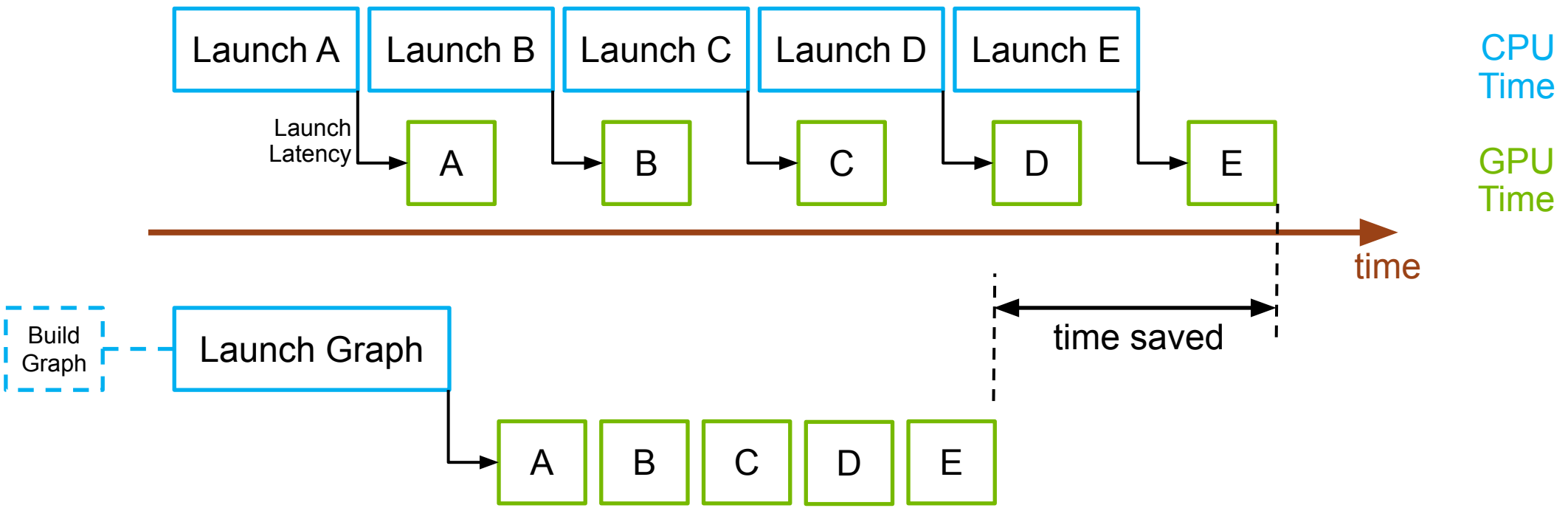
CUDA is closest to the O/S and the hardware

- Can optimize **multi-device** dependencies
- Can optimize **heterogeneous** dependencies
- Especially if executing Graphs

EXECUTION DETAILS

LAUNCH OVERHEAD REDUCTION

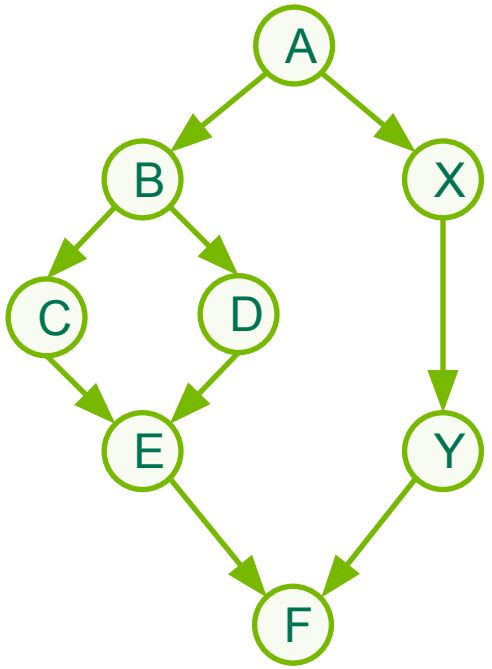
Reducing System Overheads Around Short-Running Kernels



When kernel runtime is short, execution time is dominated by CPU launch cost

TAKEAWAYS

- Cuda Graphs
  - Efficient way to express dependency
- Performance Optimization
  - Launch latency



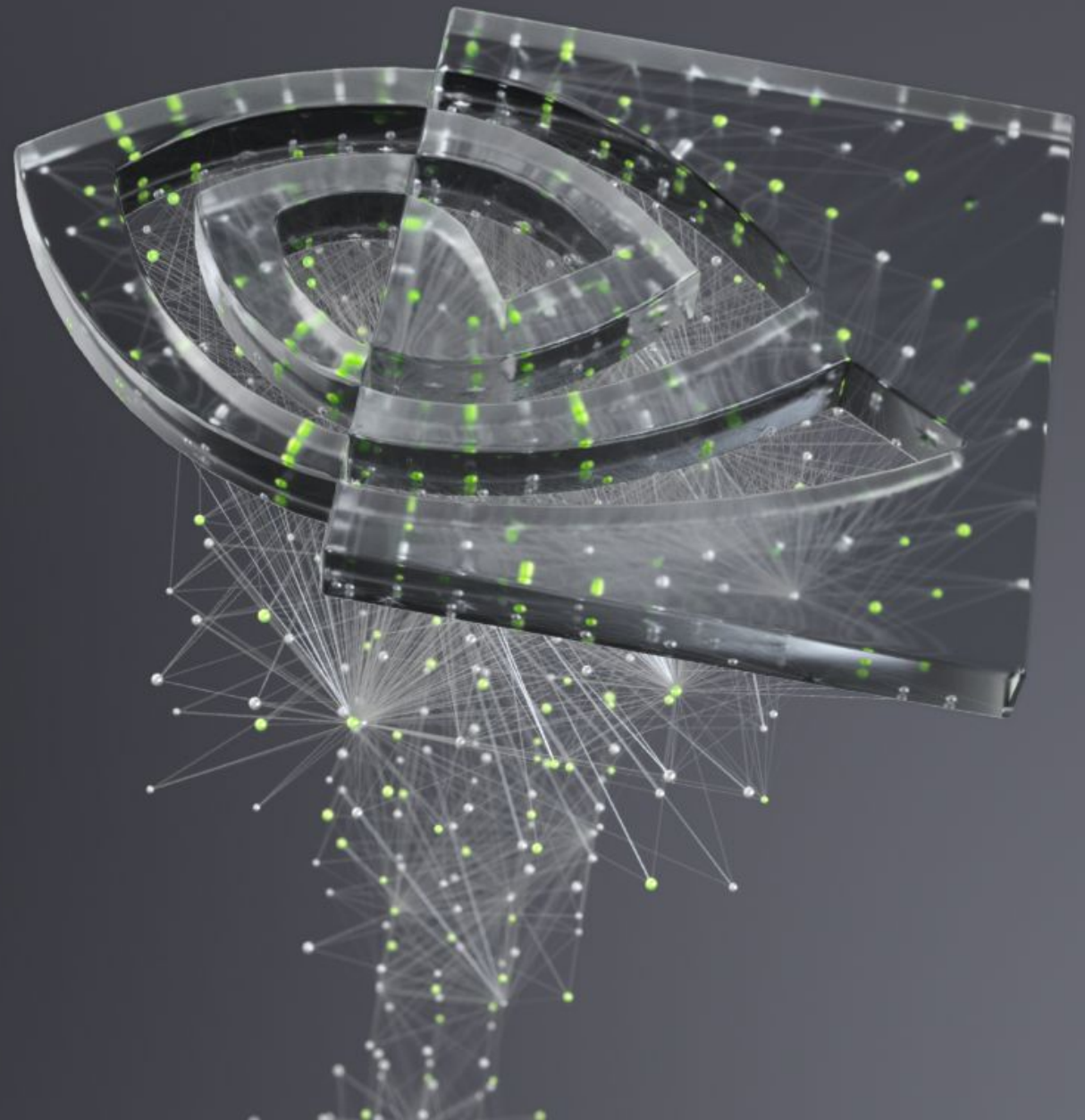
FURTHER STUDY

- Effortless CUDA Graphs GTC Spring 2021 talk
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32082/
- Cuda Memory Nodes
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#graph-memory-nodes
- Cuda Graphs API Documentation:
  - https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html

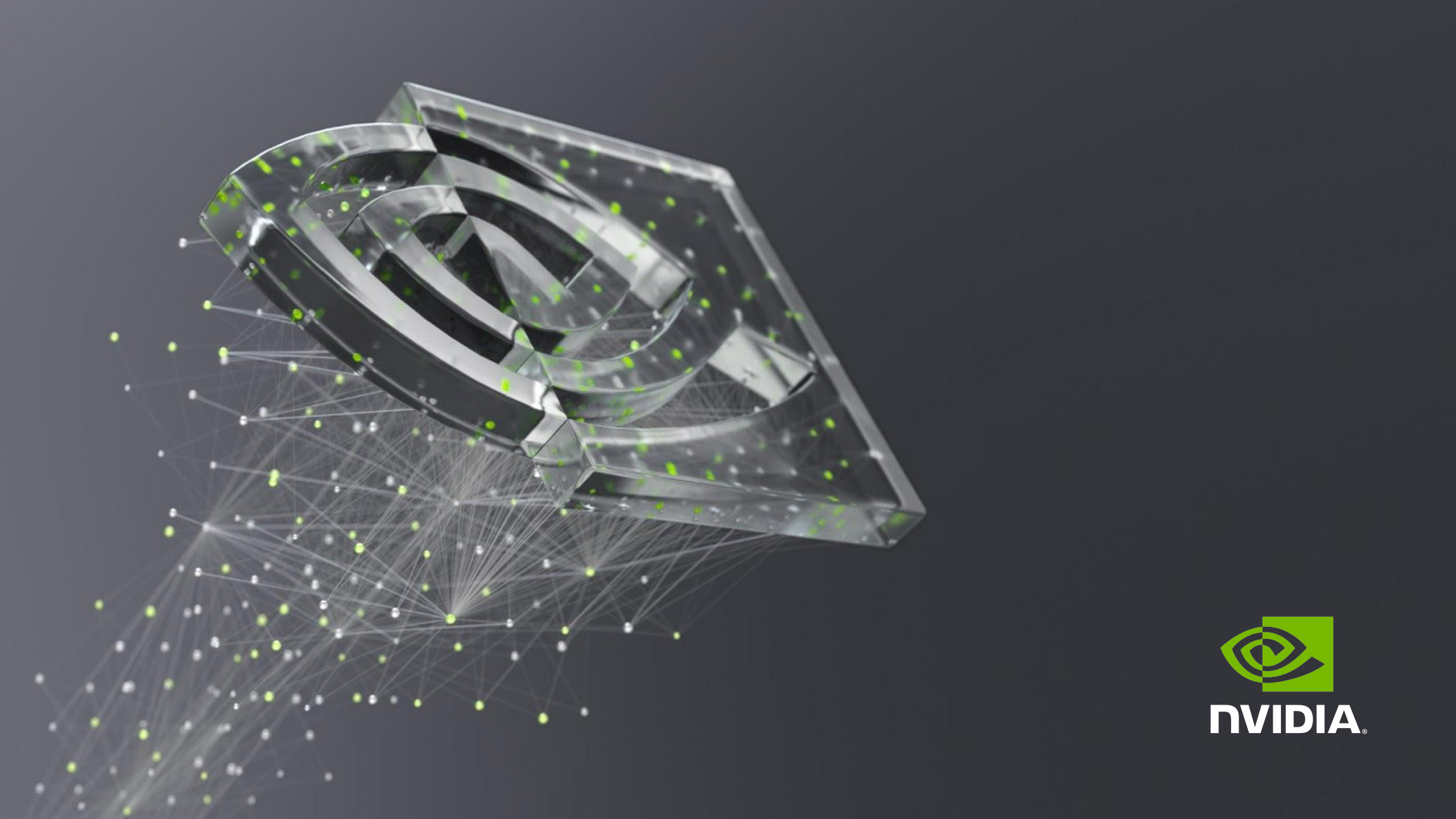
HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw13/README.md

- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming



QUESTIONS?


NVIDIA