CUDA Graphs 101

Sally Stevenson, Senior System SW Engineer | GTC 2023/March 22, 2023

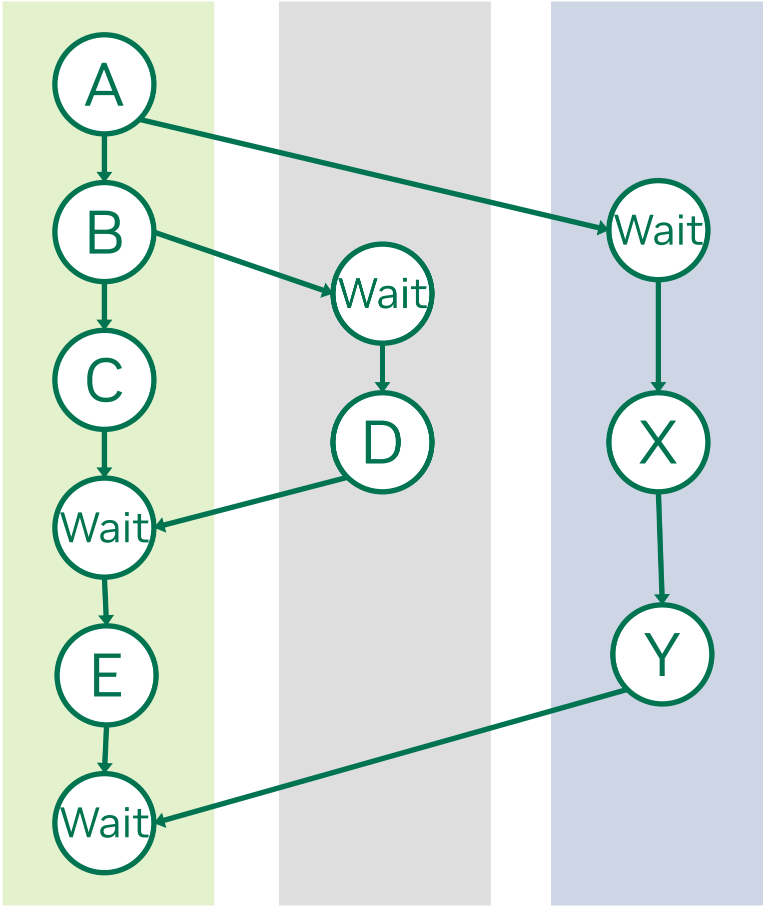
Lesson Plan

- What is CUDA graphs?
- Programming model overview
- Performance tips & tricks
- What’s new in CUDA graphs
- CUDA device graph launch

WHAT IS CUDA GRAPHS?

Speeding Up Work Launch And Execution

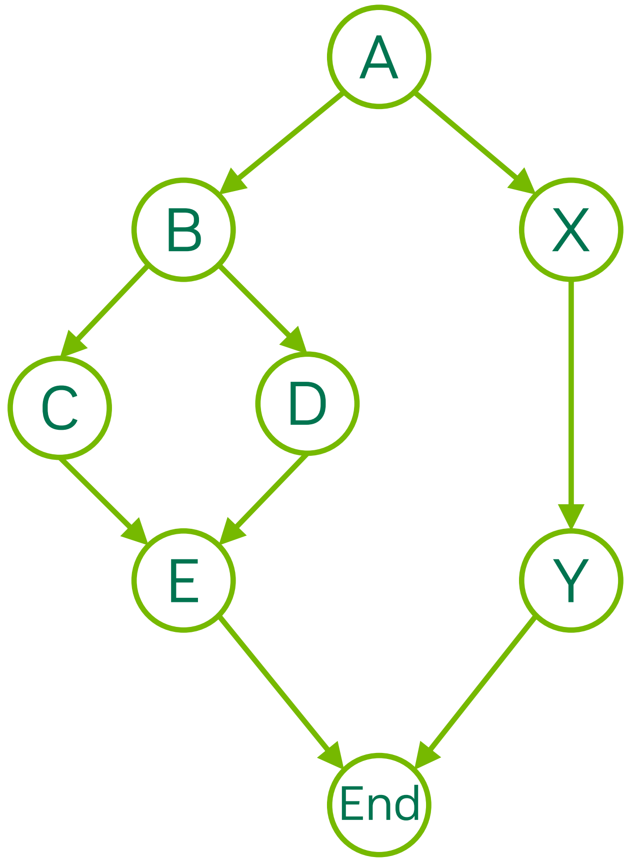
CUDA Work in Streams



Dispatched immediately

Any CUDA stream can be mapped to a graph

Graph of Dependencies



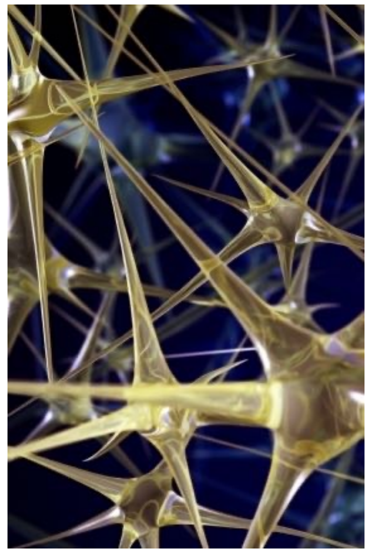
Dispatched after the workflow is fully defined

CUDA GRAPHS

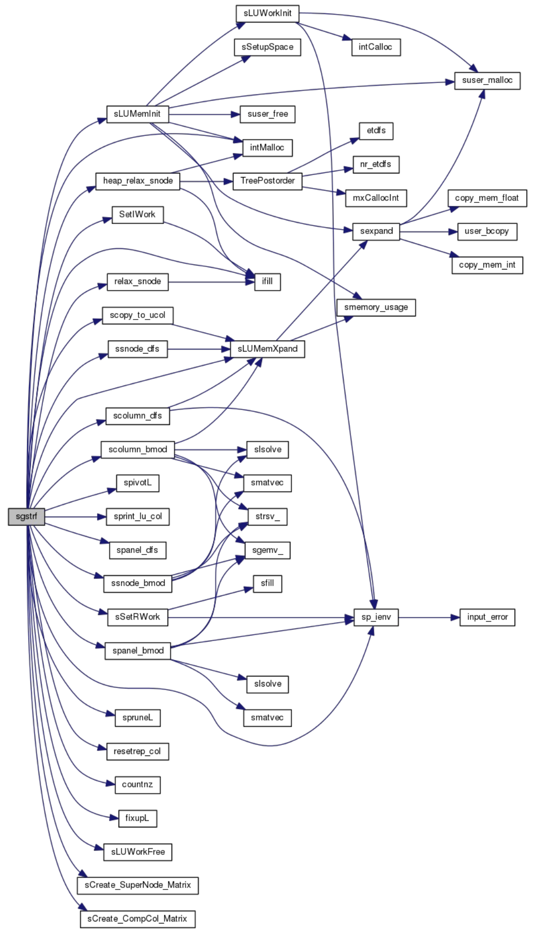
Execution Optimization When Workflow is Known Up-Front

```
// Basic function to test primality.
bool IsPrime( size_t n)
{
	if (n == 2) return true;
	if ((n == 1) || ((n % 2) == 0)) return false;
	size_t iters = (unsigned int)sqrt( (double)n);
	for (size_t i = 3; i <= iters; i+=2) if (n % i == 0) return false;
	return true;
}

// Compute primes from 1 to 100,000,000.
size_t ComputePrimes()
{
	size_t Primes = 0;
	for ( size_t Start = 1; Start <= 100000000; ++Start)
	{
		if ( IsPrime( Start) )
		{
			++Primes;
		}
	}
	return Primes;
}
```

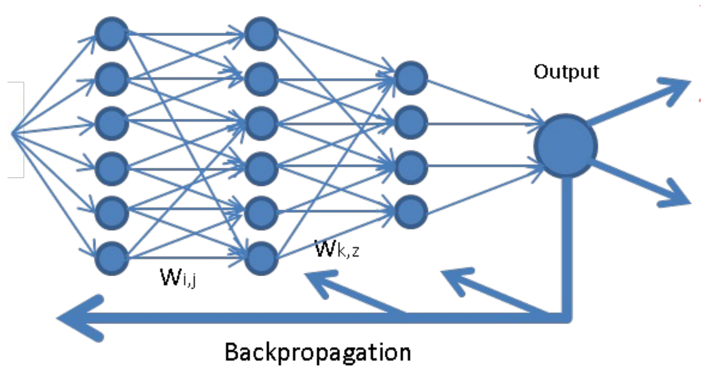
Loop & Function offload



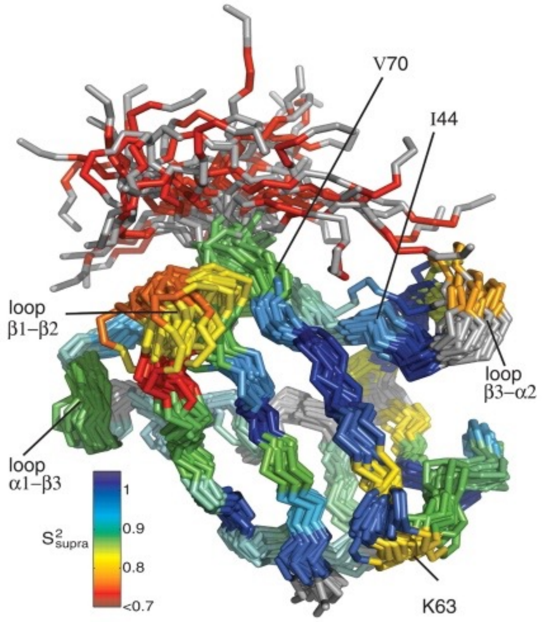
DL Inference



Linear Algebra



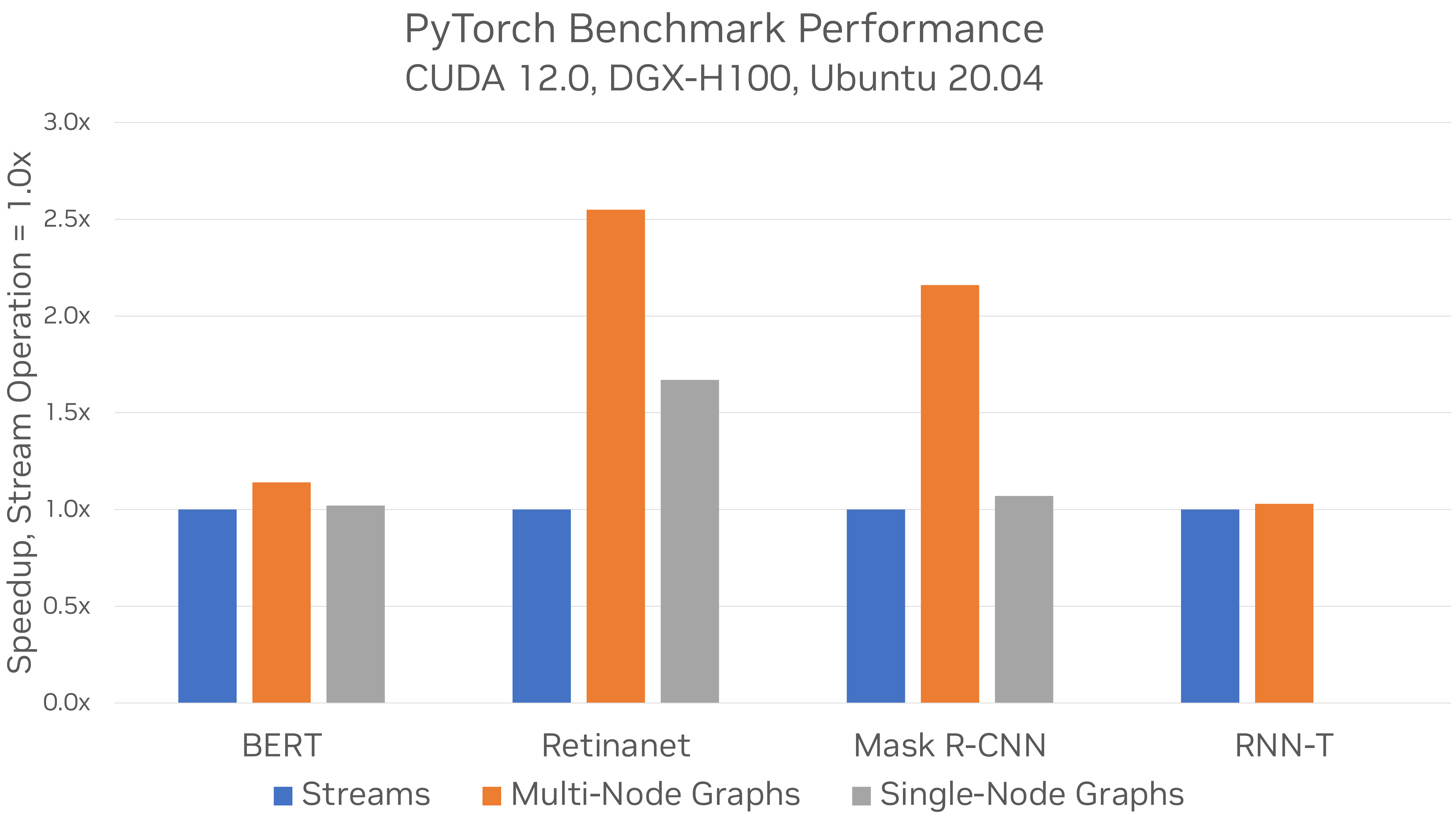
Deep Neural Network Training



HPC Simulation

PERFORMANCE IMPACT: DEEP LEARNING

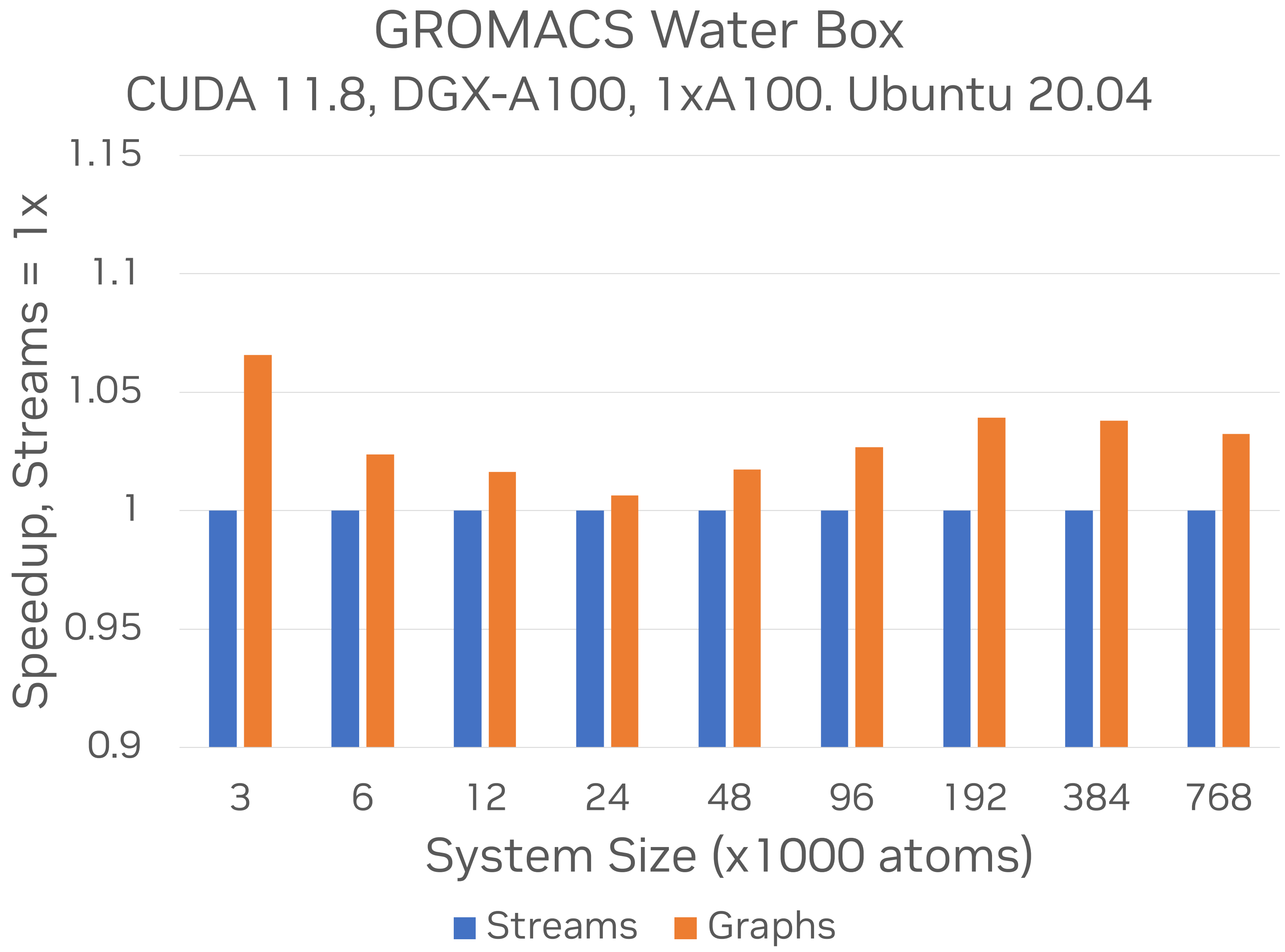
PyTorch – Machine Learning Framework



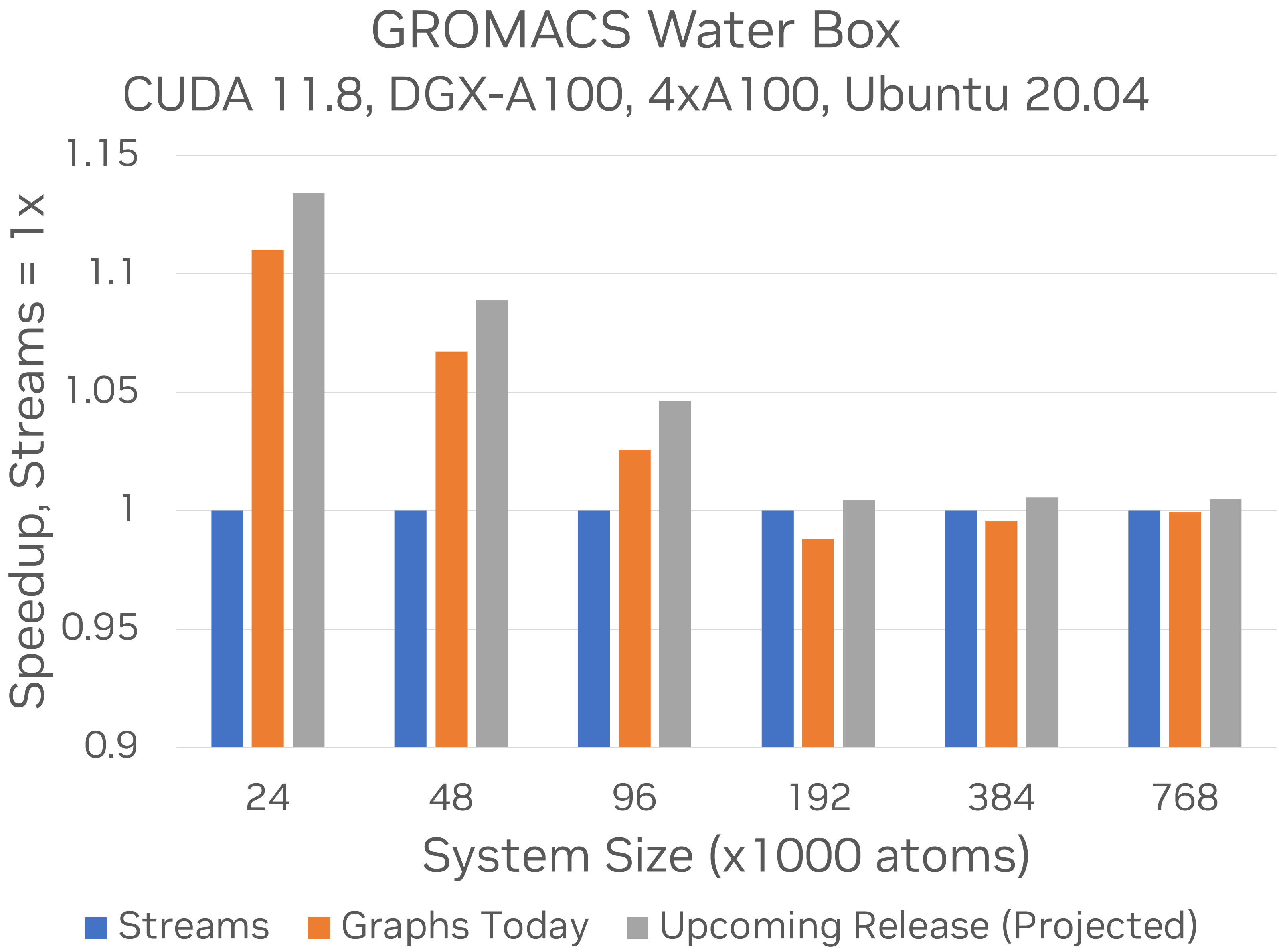
PERFORMANCE IMPACT: HPC

GROMACS – Molecular Dynamics Simulation

Single-GPU:

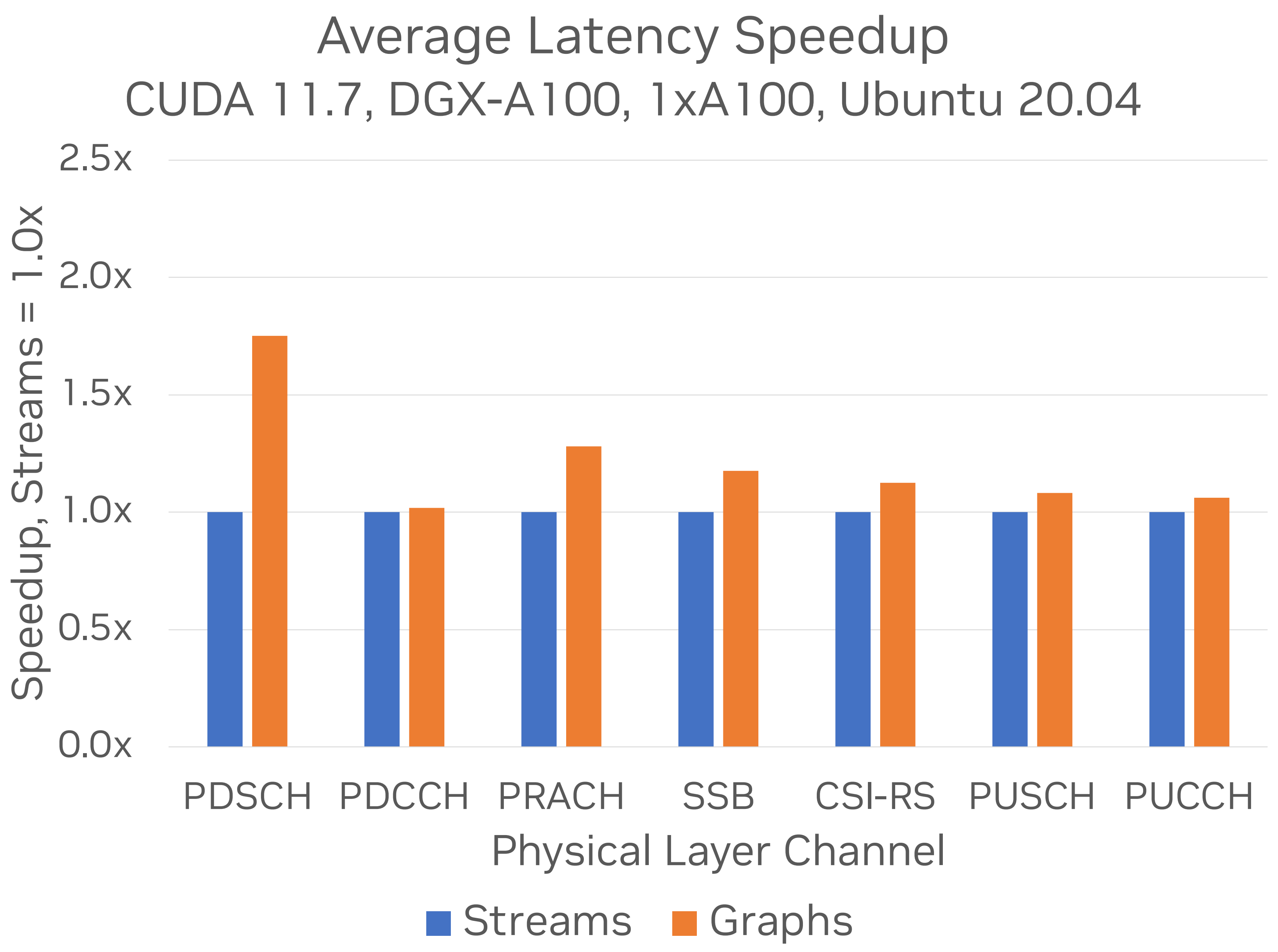


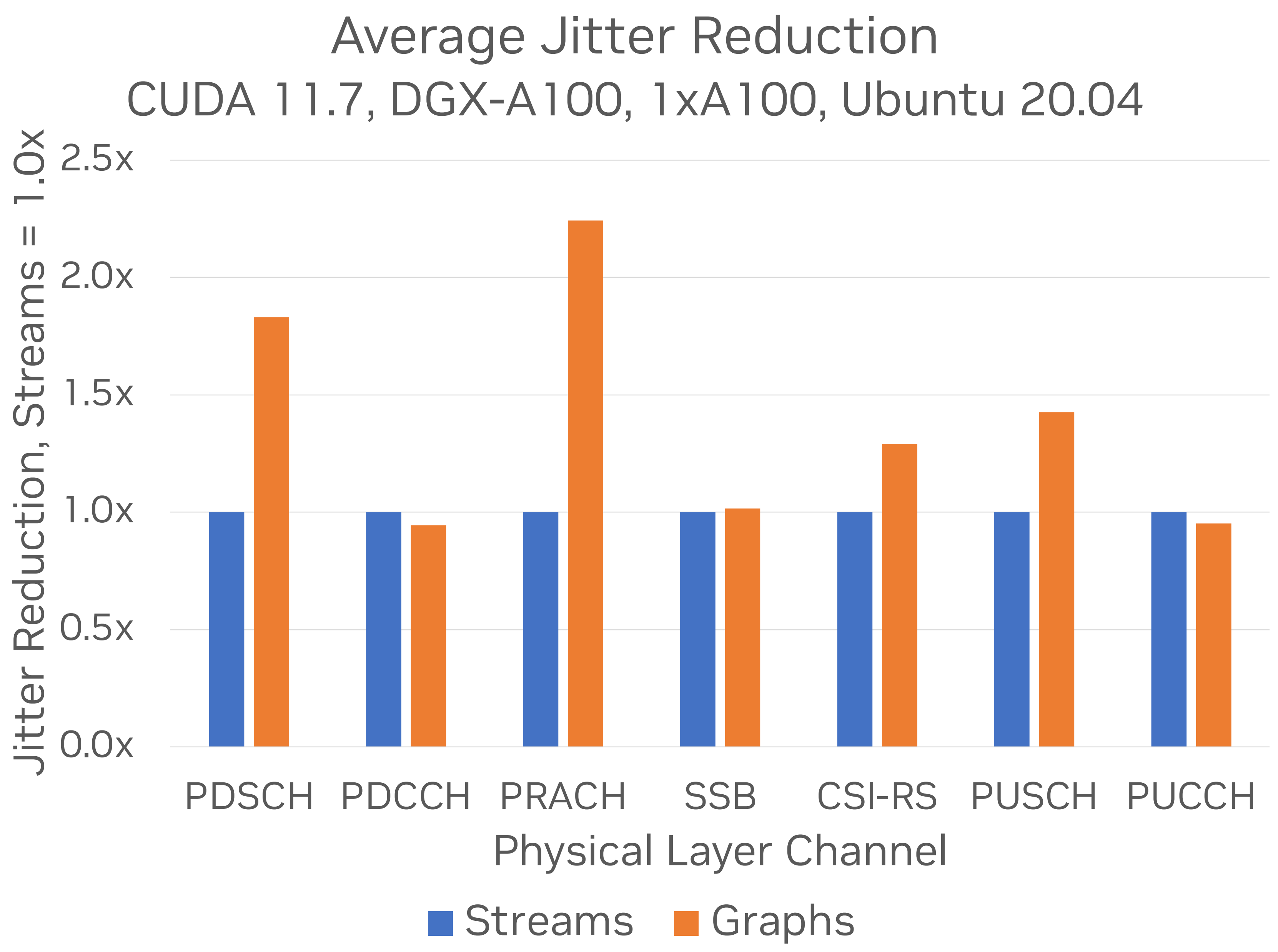
Multi-GPU:



PERFORMANCE IMPACT: COMMS

Aerial – Networking & Communications Framework





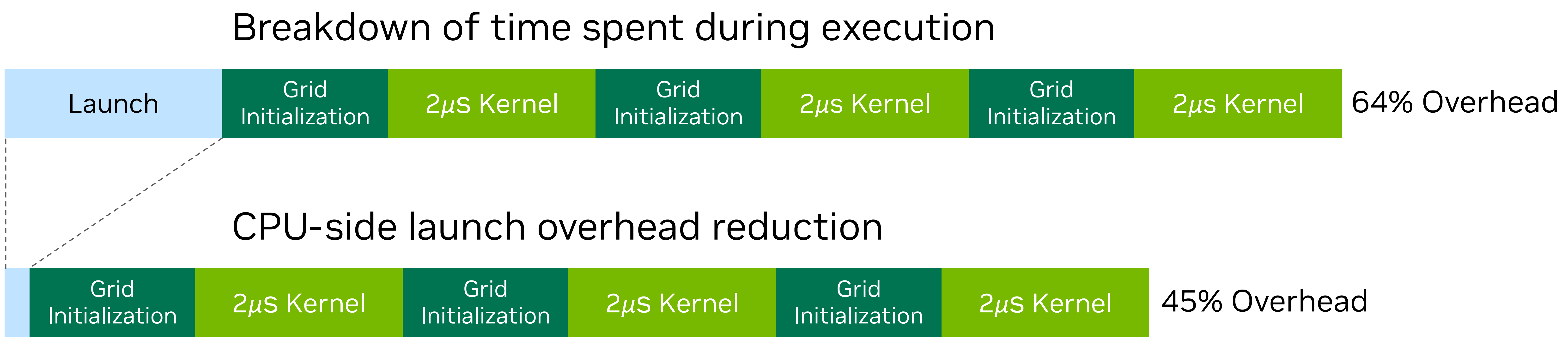
WHERE IS PERFORMANCE COMING FROM?

Reducing System Overheads Around Short-Running Kernels

Breakdown of time spent during execution

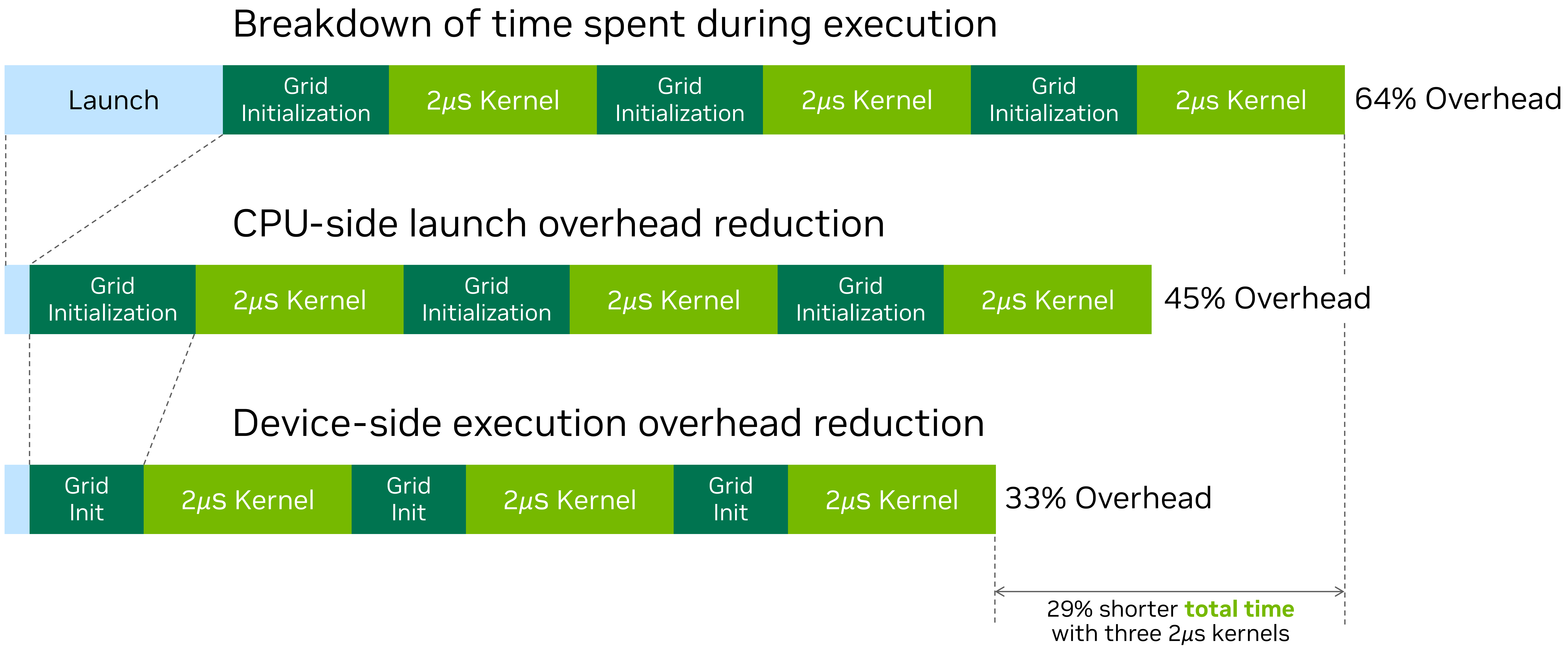
WHERE IS PERFORMANCE COMING FROM?

Reducing System Overheads Around Short-Running Kernels



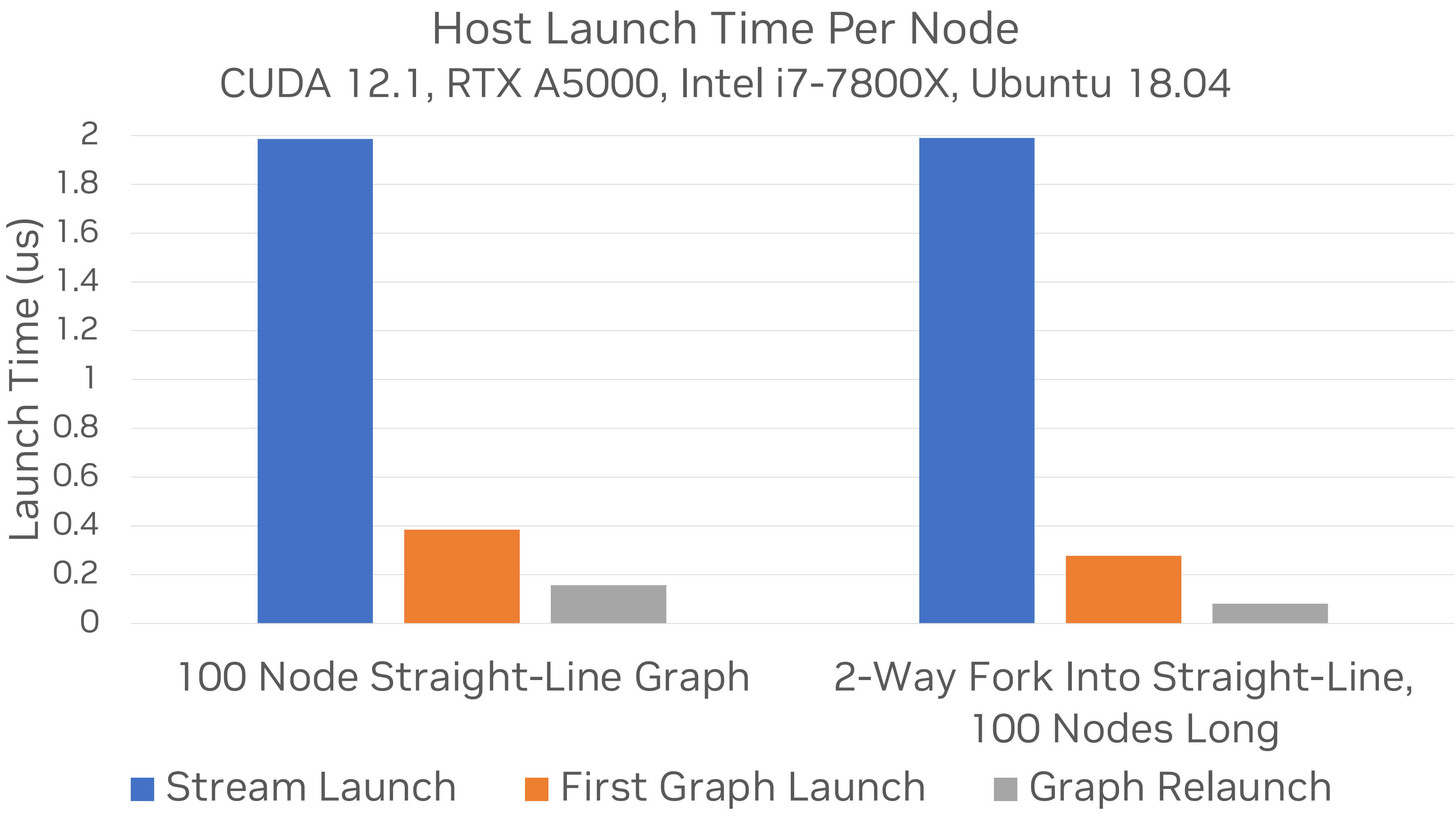
WHERE IS PERFORMANCE COMING FROM?

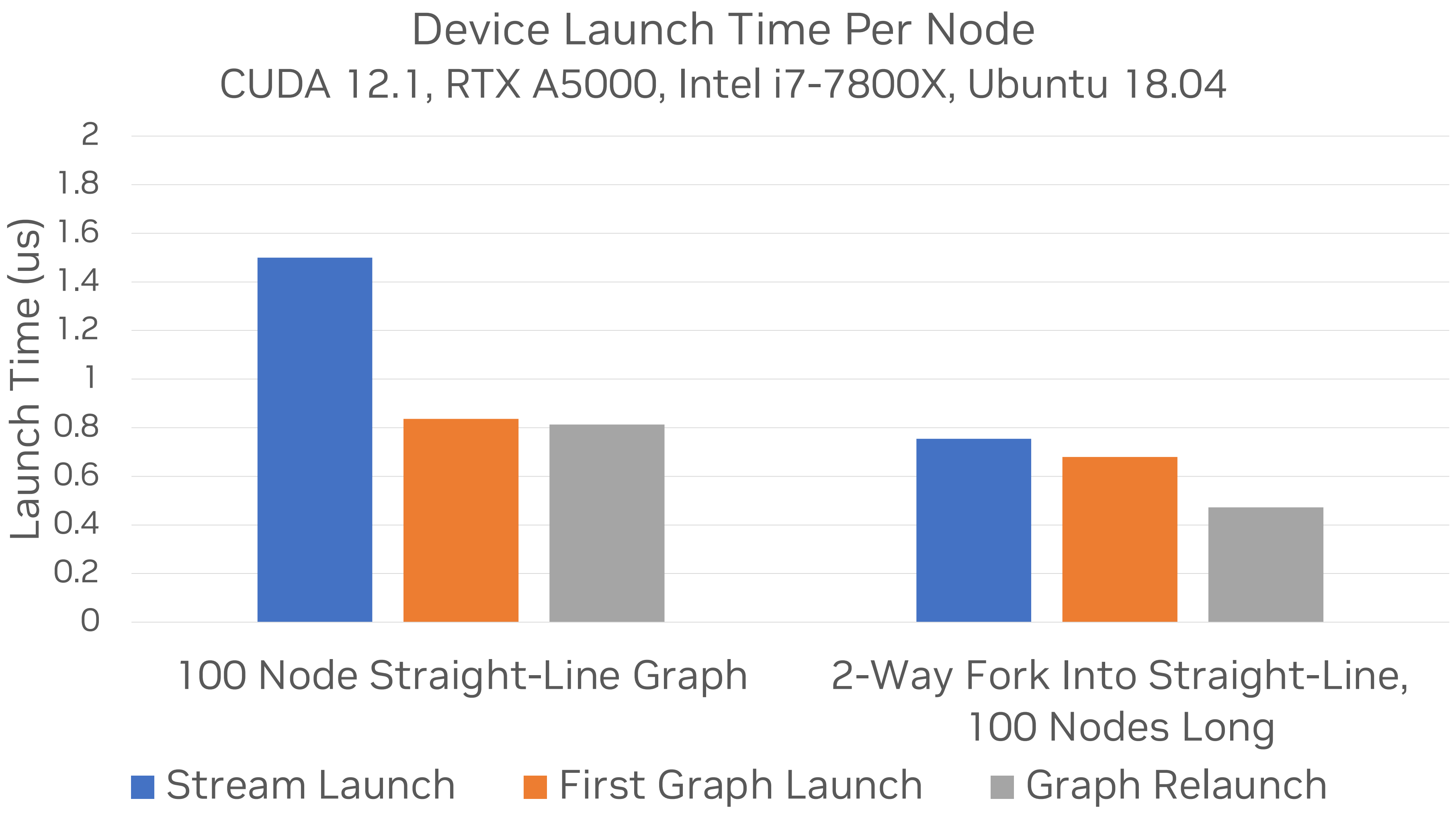
Reducing System Overheads Around Short-Running Kernels



TWO SEPARATE OPTIMIZATIONS

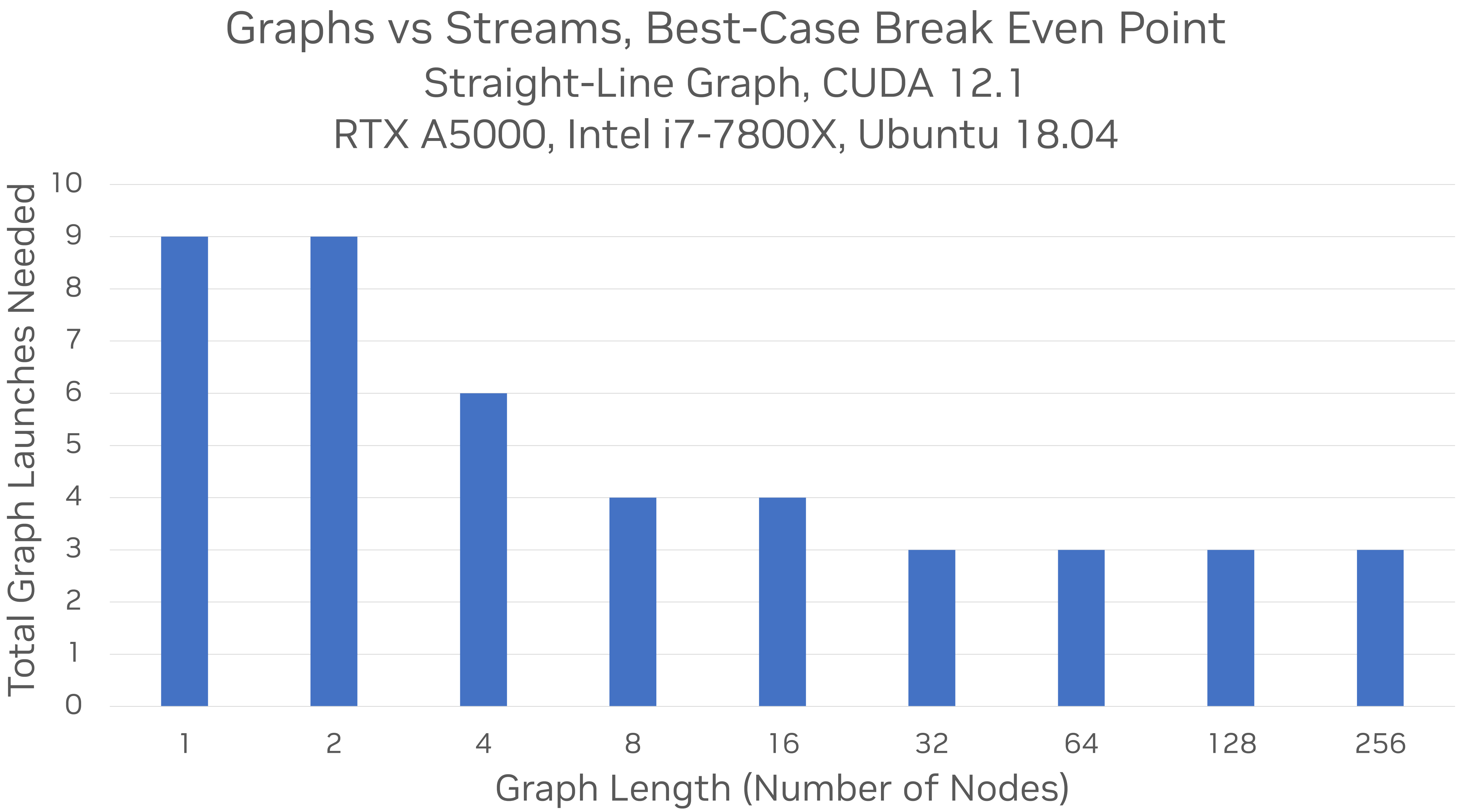
Measuring Launch Cost Separately From Grid Initialization





GRAPHS BENEFIT FROM RE-USE

CUDA Graphs Should Be Used For Repeatable Workloads

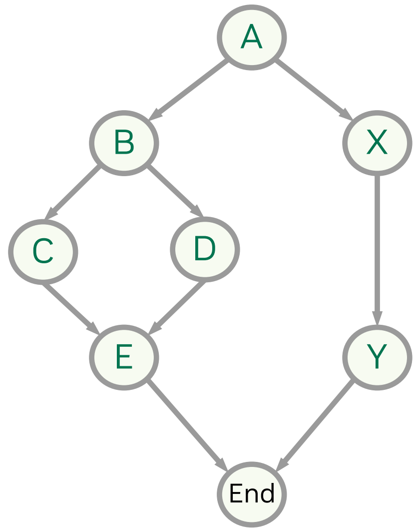


Most graphs need at least 3-4 launches to be faster than streams

THREE-STAGE EXECUTION MODEL

Minimizes Execution Overheads – Pre-Initialize As Much As Possible

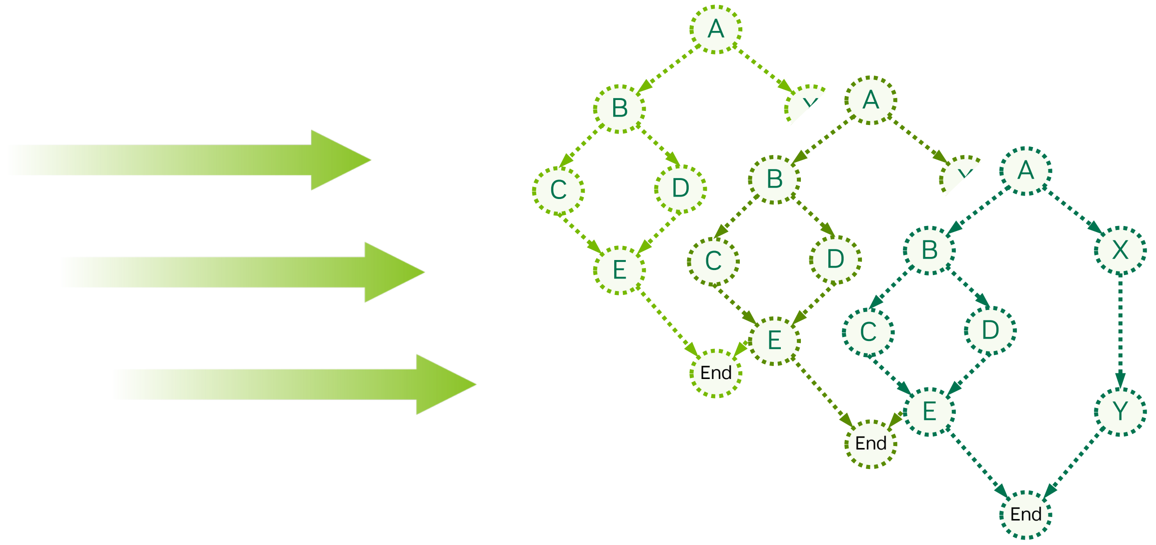
1. Define



Single Graph “Template”

Created in host code
or built up from libraries

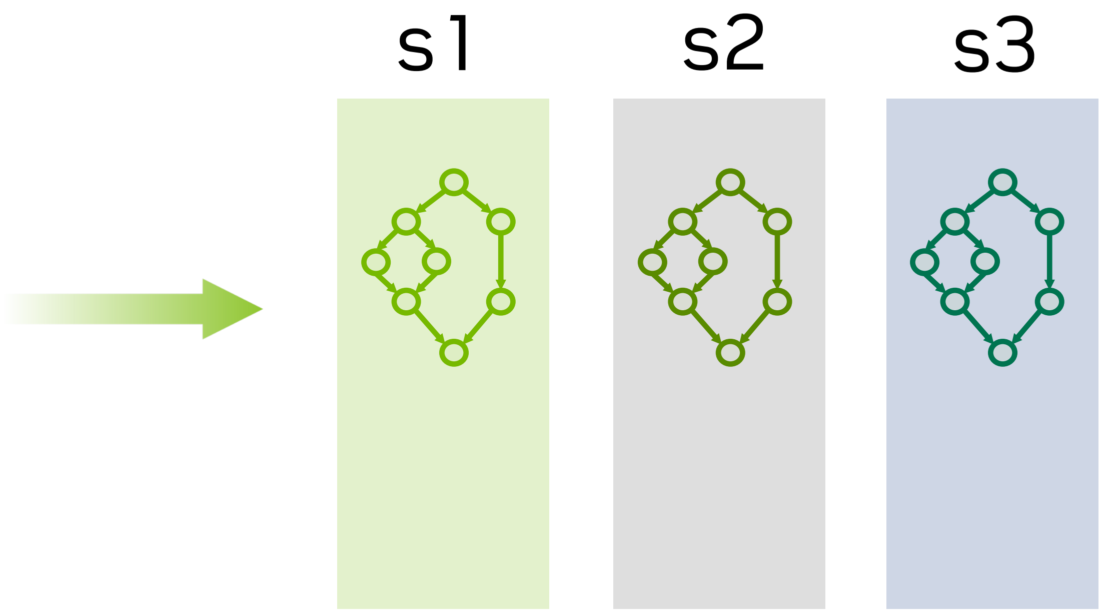
2. Instantiate



Multiple
“Executable Graphs”

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

3. Execute



Executable Graphs
Running in CUDA Streams

Concurrency in graph
is not limited by stream

WHAT OPERATIONS CAN A GRAPH NODE DO?

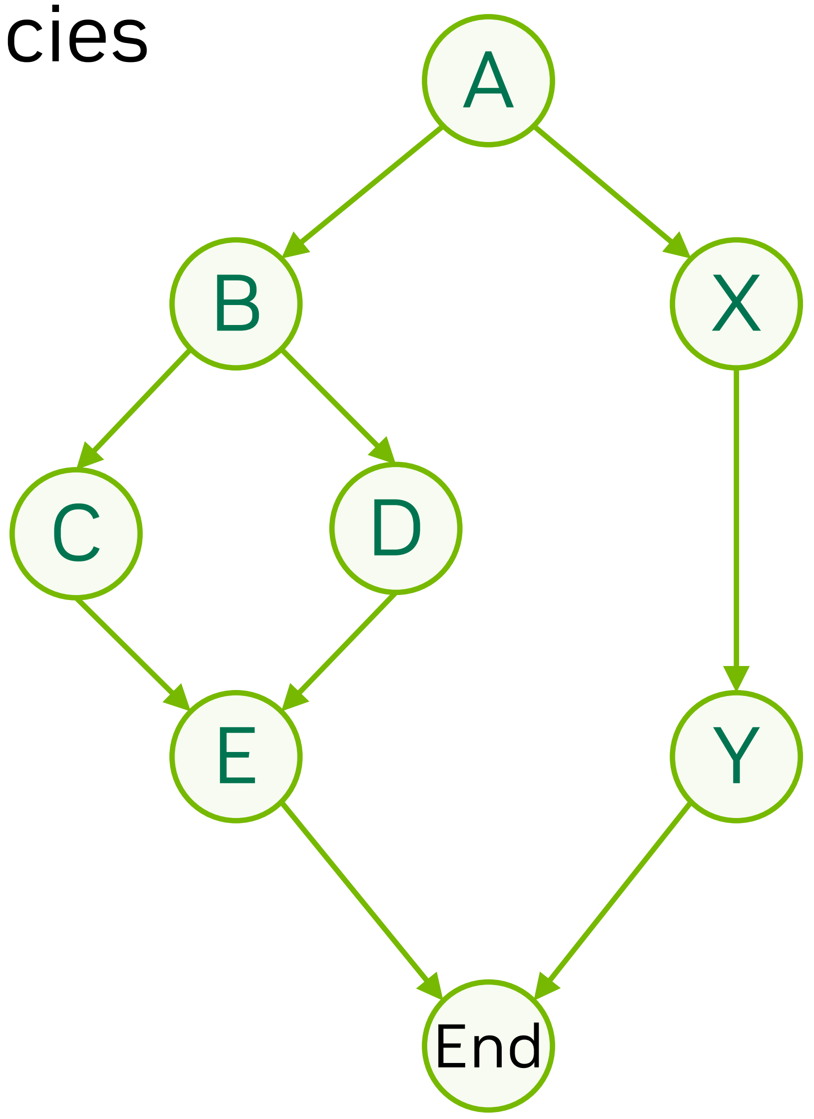
Everything You Would Expect

Graph: Sequence of operations (nodes), connected by dependencies

Nodes within a graph can span multiple devices

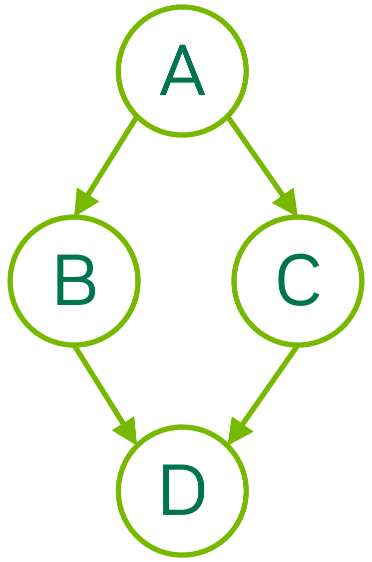
Node operations are one of:

Kernel Launch	CUDA kernel running on GPU
CPU Function Call	Callback function on CPU
Memcopy/Memset	GPU data management
Memory Alloc/Free	Memory management
External Dependency	External semaphores/events
Child Graph	Graphs are hierarchical



CREATE GRAPHS DIRECTLY

Map Graph-Based Workflows Directly Into CUDA



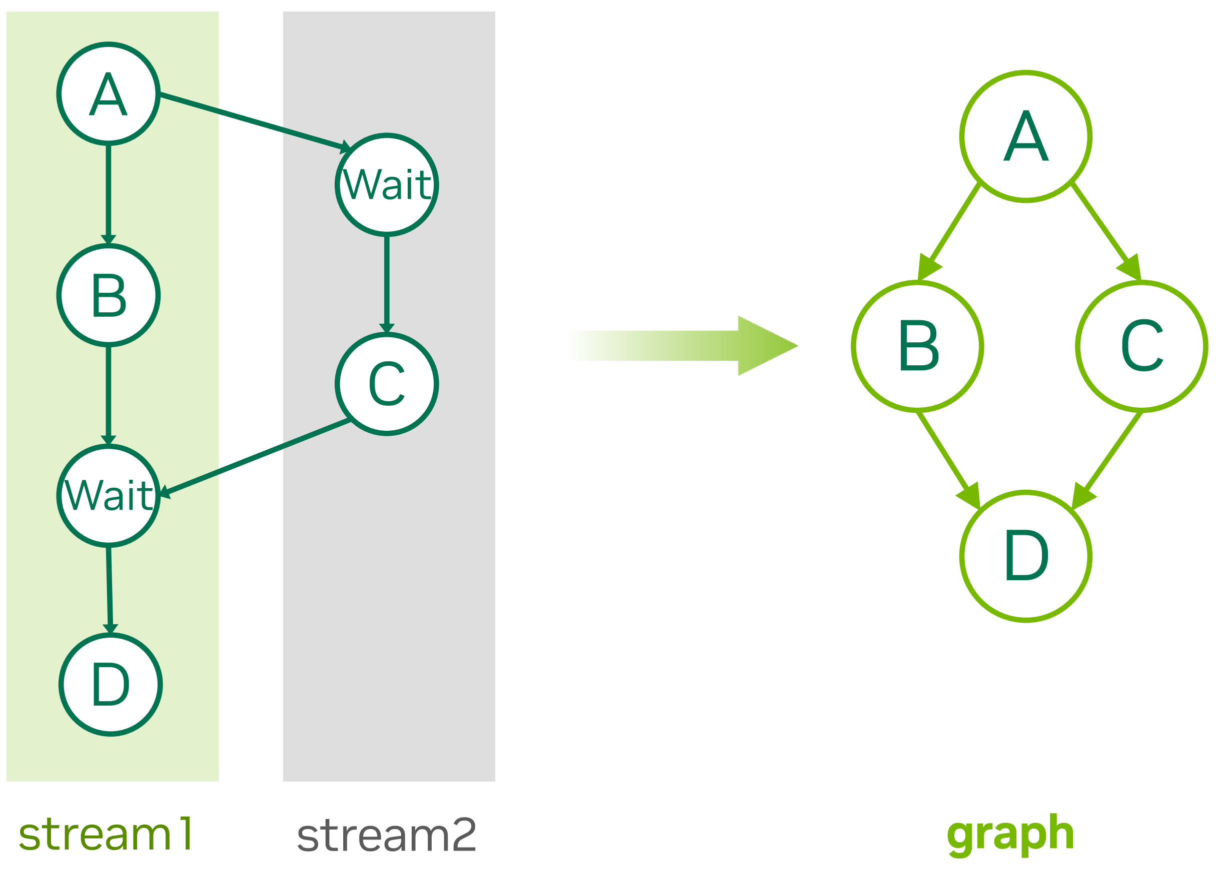
Graph from framework

```
// 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(&graphExec, graph);

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

STREAM CAPTURE

Reap the Benefits of Graphs Without Rewriting Your Code

```
// Start by initating 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);
```

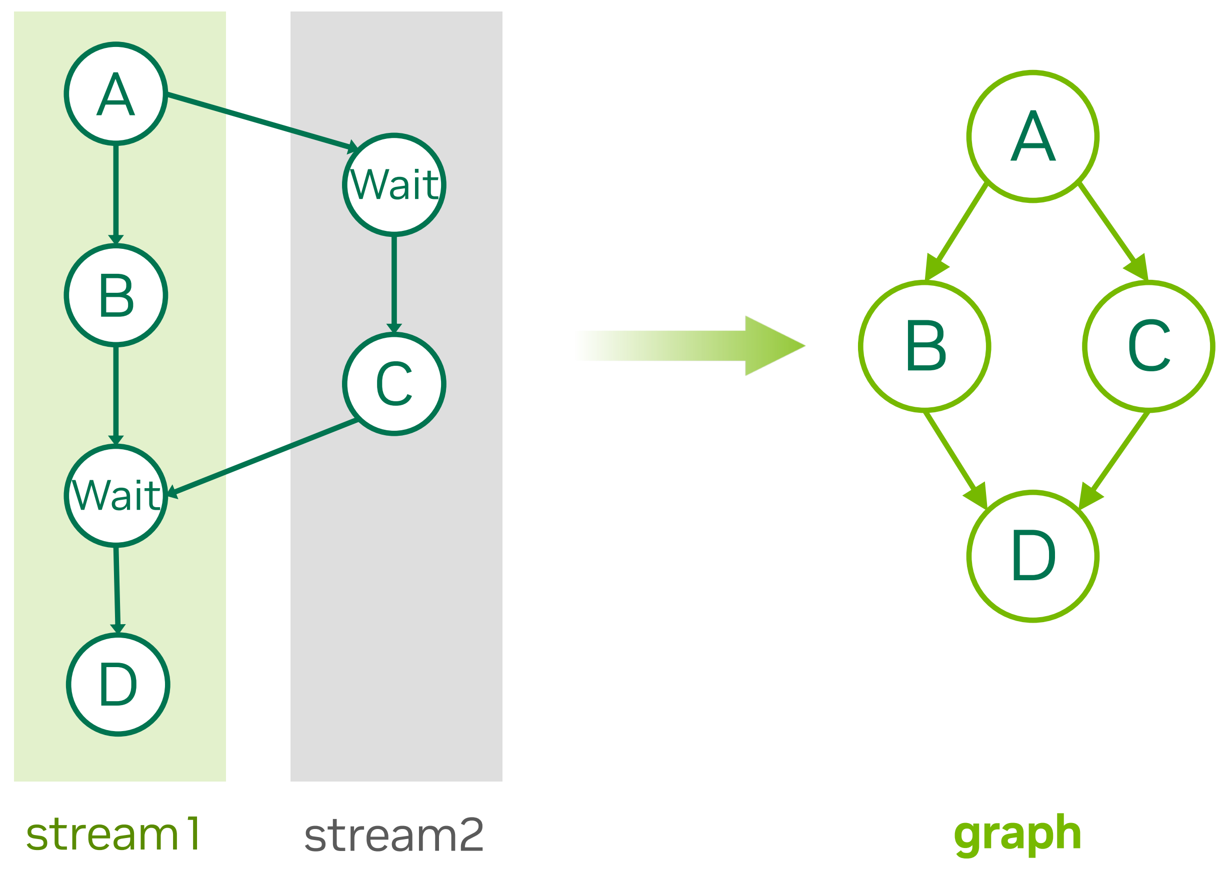


STREAM CAPTURE

Reap the Benefits of Graphs Without Rewriting Your Code

```
// Start by initating 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);
```

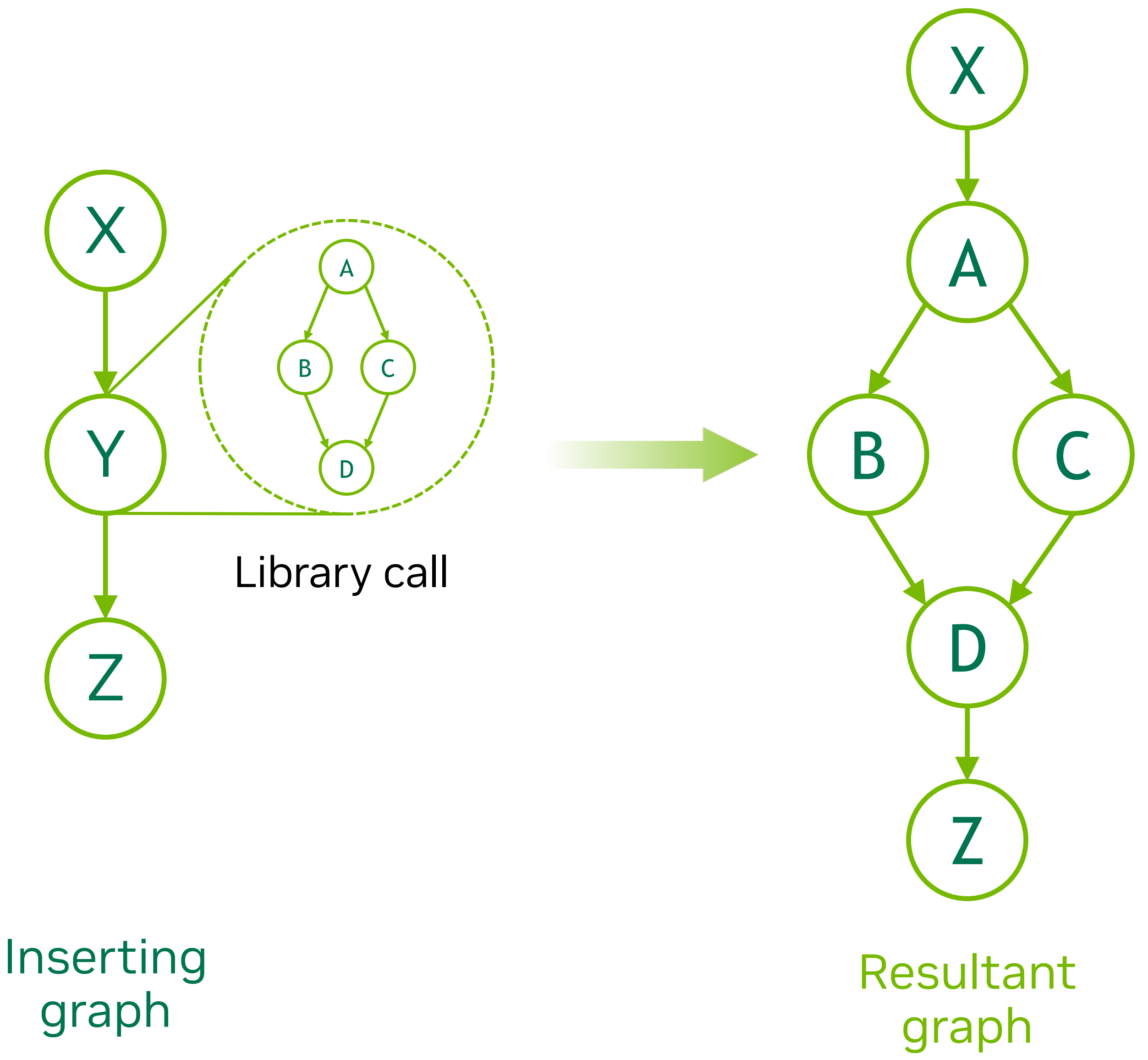
Capture follows inter-stream dependencies to create forks & joins



COMBINING GRAPH & STREAM WORK

Capturing Library Calls to Add 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”
cudaGraphAddSubgraphNode(main_graph, Y, library_graph, { X });

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



STREAM CAPTURE IN PRACTICE

Only Fully Asynchronous Sequences Can Be Captured

Original Code

```
cudaMallocHost(...);

cudaMalloc(...);

cudaMemcpy(...);

cudaDeviceSynchronize();

hostLogic(...);

libraryCall(cudaStreamDefault);

cudaFree(...);

cudaFreeHost(...);
```

With Capture

```
cudaStreamBeginCapture(streamDefault, ...);

cudaMallocHost(...);

cudaMalloc(...);

cudaMemcpy(...);

cudaDeviceSynchronize();

hostLogic(...);

libraryCall(cudaStreamDefault);

cudaFree(...);

cudaFreeHost(...);

cudaStreamEndCapture(streamDefault, &graph);
```

This code will <u>not</u> “just work” with capture!

For many applications, stream capture requires some adjustments…

STREAM CAPTURE IN PRACTICE

Only Fully Asynchronous Sequences Can Be Captured

Original Code

```
cudaMallocHost(...);

cudaMalloc(...);

cudaMemcpy(...);

cudaDeviceSynchronize();

hostLogic(...);

libraryCall(cudaStreamDefault);

cudaFree(...);

cudaFreeHost(...);
```

With Capture

```
cudaStreamBeginCapture(streamDefault, ...);

cudaMallocHost(...);

cudaMalloc(...);

cudaMemcpy(...);

cudaDeviceSynchronize();

hostLogic(...);

libraryCall(cudaStreamDefault);

cudaFree(...);

cudaFreeHost(...);

cudaStreamEndCapture(streamDefault, &graph);
```

How would one go about adapting this code for capture?

STREAM CAPTURE LIMITATIONS

Some Operations Cannot Be Captured

Original Code

```
cudaMallocHost(...);
cudaMalloc(...);
cudaMemcpy(...);
cudaDeviceSynchronize();
hostLogic(...);
libraryCall(cudaStreamDefault);
cudaFree(...);
cudaFreeHost(...);
```

With Capture

```
cudaStreamBeginCapture(stream, ...);
cudaMallocHost(...);
cudaMalloc(...);
cudaMemcpy(...);
cudaDeviceSynchronize();
hostLogic(...);
libraryCall(stream);
cudaFree(...);
cudaFreeHost(...);
cudaStreamEndCapture(stream, &graph);
```

The default ("null") stream cannot be captured

STREAM CAPTURE LIMITATIONS

Some Operations Cannot Be Captured

Original Code

```
cudaMallocHost(...);
cudaMalloc(...);
cudaMemcpy(...);
cudaDeviceSynchronize();
hostLogic(...);
libraryCall(cudaStreamDefault);
cudaFree(...);
cudaFreeHost(...);
```

With Capture

```
cudaStreamBeginCapture(stream, ...);
cudaMallocHost(...);
cudaMallocAsync(..., stream);
cudaMemcpyAsync(..., stream);
cudaDeviceSynchronize();
hostLogic(...);
libraryCall(stream);
cudaFreeAsync(..., stream);
cudaFreeHost(...);
cudaStreamEndCapture(stream, &graph);
```

Synchronous calls cannot be captured

STREAM CAPTURE LIMITATIONS

Some Operations Cannot Be Captured

Original Code

```
cudaMallocHost(...);
cudaMalloc(...);
cudaMemcpy(...);
cudaDeviceSynchronize();
hostLogic(...);
libraryCall(cudaStreamDefault);
cudaFree(...);
cudaFreeHost(...);
```

With Capture

```
cudaMallocHost(...);
cudaStreamBeginCapture(stream, ...);
cudaMallocAsync(..., stream);
cudaMemcpyAsync(..., stream);
cudaDeviceSynchronize();
hostLogic(...);
libraryCall(stream);
cudaFreeAsync(..., stream);
cudaStreamEndCapture(stream, &graph);
cudaFreeHost(...);
```

Synchronous calls cannot be captured
(Calls with no asynchronous equivalent must occur outside the capture)

STREAM CAPTURE LIMITATIONS

Some Operations Cannot Be Captured

Original Code

```
cudaMallocHost(...);
cudaMalloc(...);
cudaMemcpy(...);
cudaDeviceSynchronize();
hostLogic(...);
libraryCall(cudaStreamDefault);
cudaFree(...);
cudaFreeHost(...);
```

With Capture

```
cudaMallocHost(...);
cudaStreamBeginCapture(stream, ...);
cudaMallocAsync(..., stream);
cudaMemcpyAsync(..., stream);
cudaDeviceSynchronize();
hostLogic(...);
libraryCall(stream);
cudaFreeAsync(..., stream);
cudaStreamEndCapture(stream, &graph);
cudaFreeHost(...);
```

Stream capture cannot synchronize

GRAPHS IN PRACTICE

Am I Done Once My Code Is Capturable?

Capturable Code

```
cudaMallocHost(...);

cudaStreamBeginCapture(stream, ...);

cudaMallocAsync(..., stream);

cudaMemcpyAsync(..., stream);

hostLogic(...);

libraryCall(stream);

cudaFreeAsync(..., stream);

cudaStreamEndCapture(stream, &graph);

cudaFreeHost(...);

// Instantiate & launch the graph
```

So, is this code graphs-ready now?

GRAPH ADOPTION TIPS

Avoid Common Pitfalls

Capturable Code

```
cudaMallocHost(...);
cudaStreamBeginCapture(stream, ...);
cudaMallocAsync(..., stream);
cudaMemcpyAsync(..., stream);
hostLogic(...);
libraryCall(stream);
cudaFreeAsync(..., stream);
cudaStreamEndCapture(stream, &graph);
cudaFreeHost(...);
// Instantiate & launch the graph
```

Graphs-Ready Code

```
cudaMallocHost(...);
cudaStreamBeginCapture(stream, ...);
cudaMallocAsync(..., stream);
cudaMemcpyAsync(..., stream);
hostLogic(...);
libraryCall(stream);
cudaFreeAsync(..., stream);
cudaStreamEndCapture(stream, &graph);
cudaFreeHost(...);
// Instantiate & launch the graph
```

Tip #1: Put your memory management into the capture via cudaMalloc/FreeAsync

GRAPH ADOPTION TIPS

Avoid Common Pitfalls

Capturable Code

```
cudaMallocHost(...);

cudaStreamBeginCapture(stream, ...);

cudaMallocAsync(..., stream);

cudaMemcpyAsync(..., stream);

hostLogic(...);

libraryCall(stream);

cudaFreeAsync(..., stream);

cudaStreamEndCapture(stream, &graph);

cudaFreeHost(...);

// Instantiate & launch the graph
```

Graphs-Ready Code

```
cudaMallocHost(...);

cudaStreamBeginCapture(stream, ...);

cudaMallocAsync(..., stream);

cudaMemcpyAsync(..., stream);

hostLogic(...);

libraryCall(stream);

cudaFreeAsync(..., stream);

cudaStreamEndCapture(stream, &graph);

// Instantiate & launch the graph

cudaFreeHost(...);
```

Tip #1.5: For other allocations – keep the memory around, or it cannot be accessed on launch

GRAPH ADOPTION TIPS

Avoid Common Pitfalls

Capturable Code

```
cudaMallocHost(...);
cudaStreamBeginCapture(stream, ...);
cudaMallocAsync(..., stream);
cudaMemcpyAsync(..., stream);
hostLogic(...);
libraryCall(stream);
cudaFreeAsync(..., stream);
cudaStreamEndCapture(stream, &graph);
cudaFreeHost(...);
// Instantiate & launch the graph
```

Graphs-Ready Code

```
cudaMallocHost(...);
cudaStreamBeginCapture(stream, ...);
cudaMallocAsync(..., stream);
cudaMemcpyAsync(..., stream);
cudaLaunchHostFunc(stream, hostLogic, ...);
libraryCall(stream);
cudaFreeAsync(..., stream);
cudaStreamEndCapture(stream, &graph);
// Instantiate & launch the graph
cudaFreeHost(...);
```

Tip #2: Put any important logic into a CPU callback

GRAPH ADOPTION TIPS

Avoid Common Pitfalls

Capturable Code

```
cudaMallocHost(...);
cudaStreamBeginCapture(stream, ...);
cudaMallocAsync(..., stream);
cudaMemcpyAsync(..., stream);
hostLogic(...);
libraryCall(stream);
cudaFreeAsync(..., stream);
cudaStreamEndCapture(stream, &graph);
cudaFreeHost(...);
// Instantiate & launch the graph
```

Graphs-Ready Code

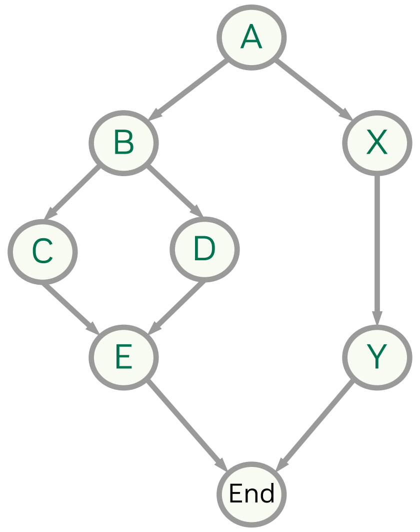
```
cudaMallocHost(...);
cudaStreamBeginCapture(stream, threadLocal);
cudaMallocAsync(..., stream);
cudaMemcpyAsync(..., stream);
cudaLaunchHostFunc(stream, hostLogic, ...);
libraryCall(stream);
cudaFreeAsync(..., stream);
cudaStreamEndCapture(stream, &graph);
// Instantiate & launch the graph
cudaFreeHost(...);
```

Tip #3: If your application is multithreaded and the threads run independently, consider using thread-local capture mode

THREE-STAGE EXECUTION MODEL

Minimizes Execution Overheads – Pre-Initialize As Much As Possible

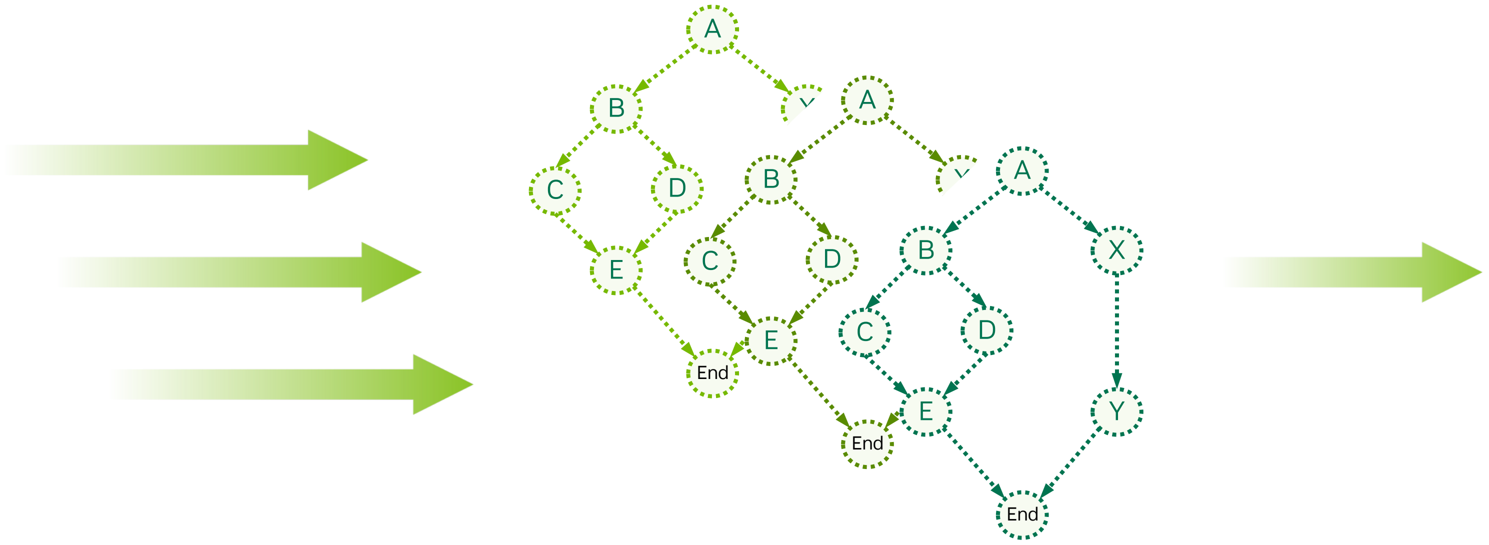
1. Define



Single Graph “Template”

Created in host code
or built up from libraries

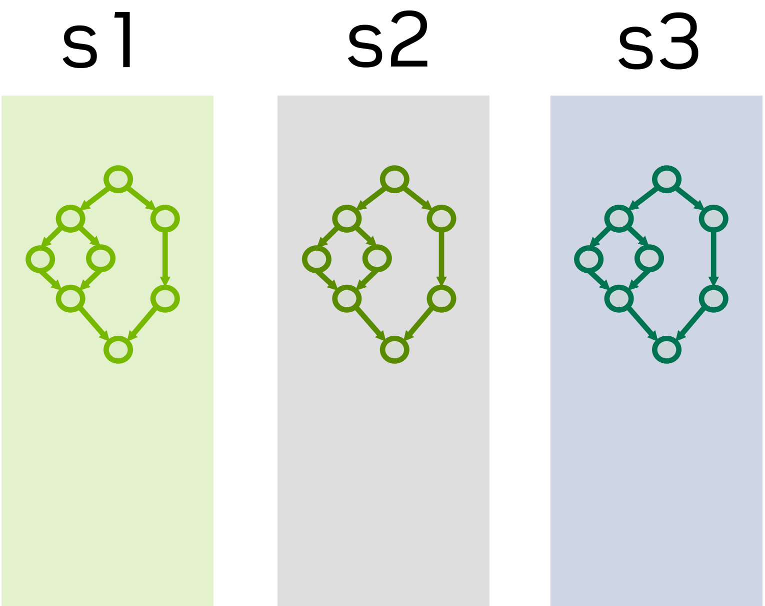
2. Instantiate



Multiple
“Executable Graphs”

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

3. Execute



Executable Graphs
Running in CUDA Streams

Concurrency in graph
is not limited by stream

GRAPH INSTANTIATION

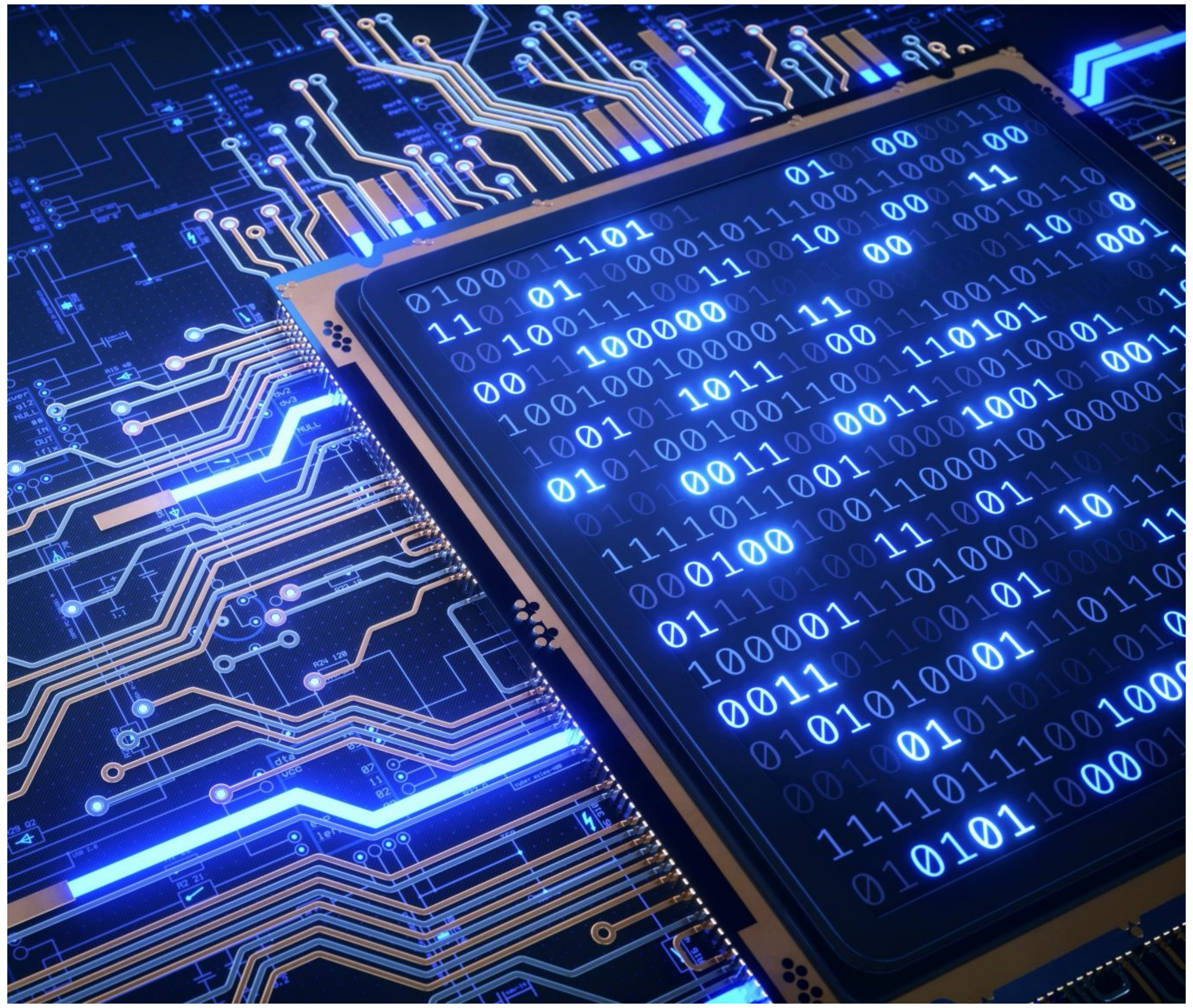
Preparing The Graph For Launch

Graph instantiation is equivalent to a compilation step for the graph

- Prepares and optimizes the graph for execution
- Executable graph structure is locked when you instantiate
  - Structural changes require re-instantiation

As with code compilation, instantiation is not a trivial step and takes some additional time

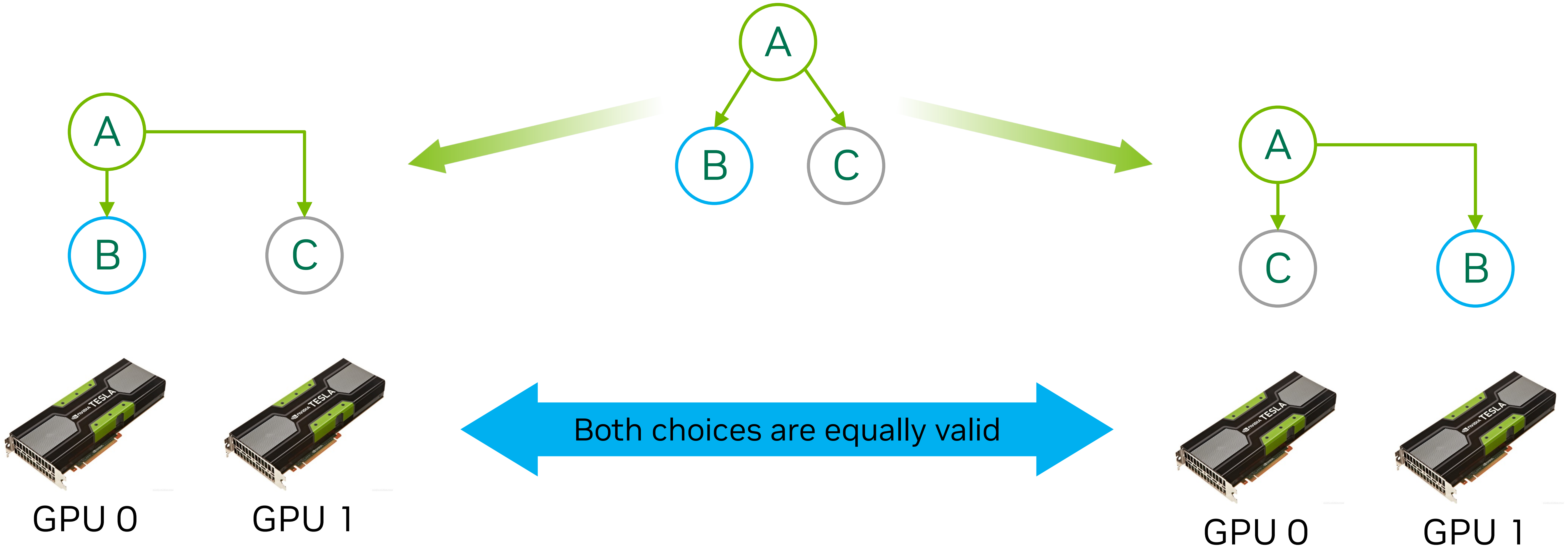
And, like any compilation step, instantiation will not do everything for you…



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

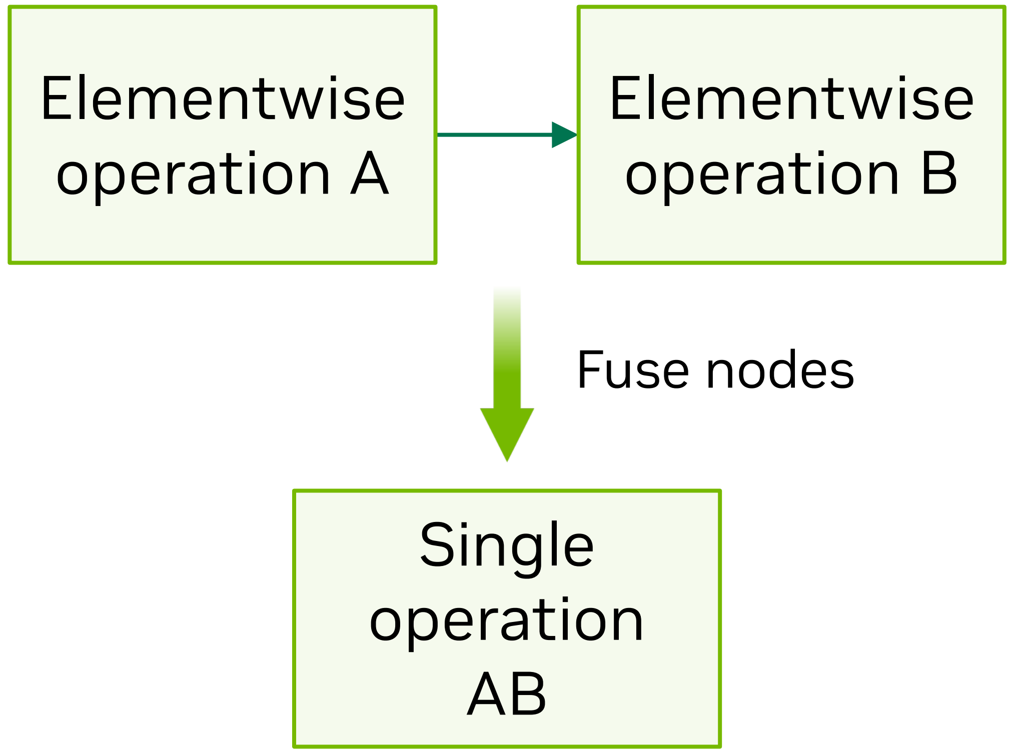
NO STRUCTURAL CHANGES

Execution Layer Does Not Have The Information Needed To Do This

No **splitting** of graph nodes

No **merging** of graph nodes

No reassigning execution **location** of a node



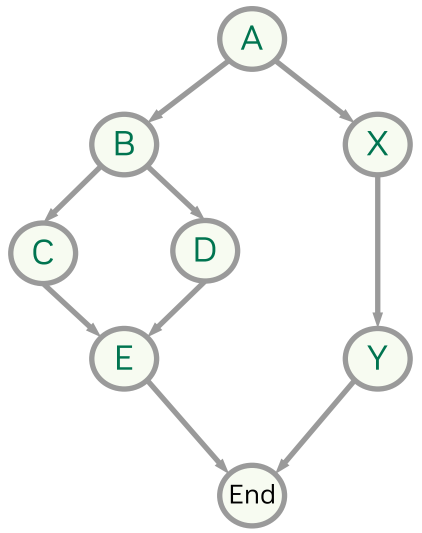
Elementwise operations can trivially be fused,
but only if operation semantics are known.

Execution layer sees only binary code,
so cannot perform this merge

THREE-STAGE EXECUTION MODEL

Minimizes Execution Overheads – Pre-Initialize As Much As Possible

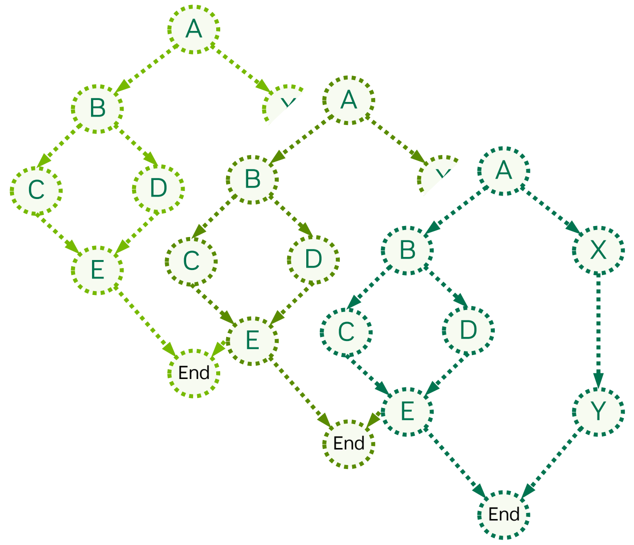
1. Define



Single Graph "Template"

Created in host code
or built up from libraries

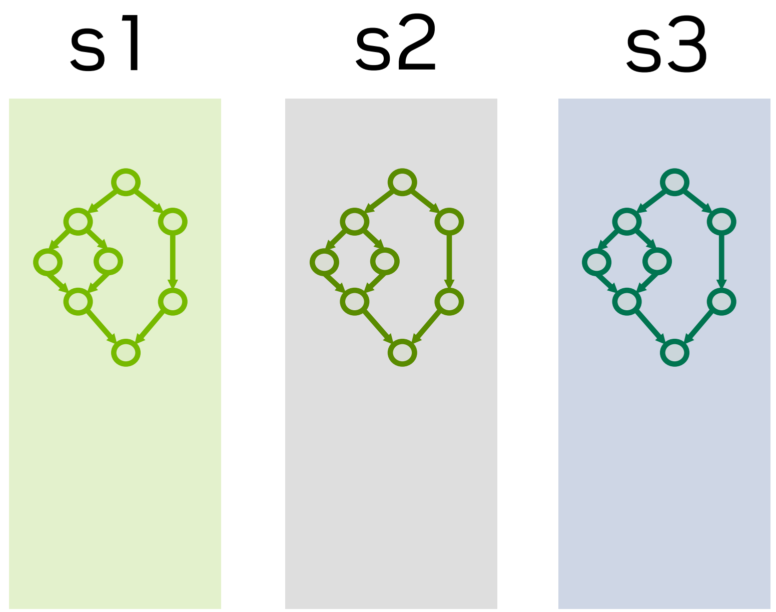
2. Instantiate



Multiple
"Executable Graphs"

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

3. Execute



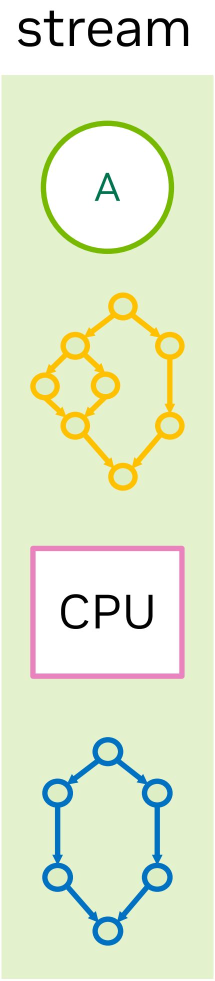
Executable Graphs
Running in CUDA Streams

Concurrency in graph
is not limited by stream

GRAPH EXECUTION SEMANTICS

Order Graph Work With Other Non-Graph CUDA Work

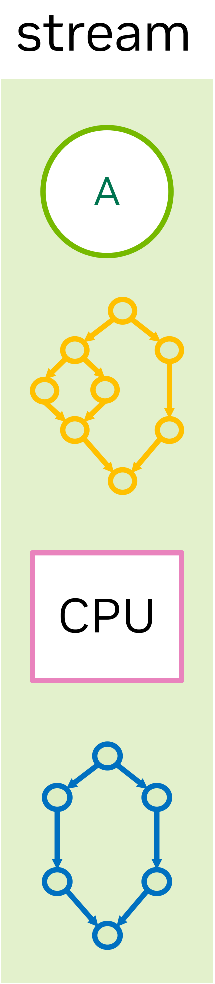
```
launchWork(cudaGraphExec_t i1, cudaGraphExec_t i2,
           CPU_Func cpu, cudaStream_t stream) {

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

    cudaStreamSynchronize(stream);
}
```



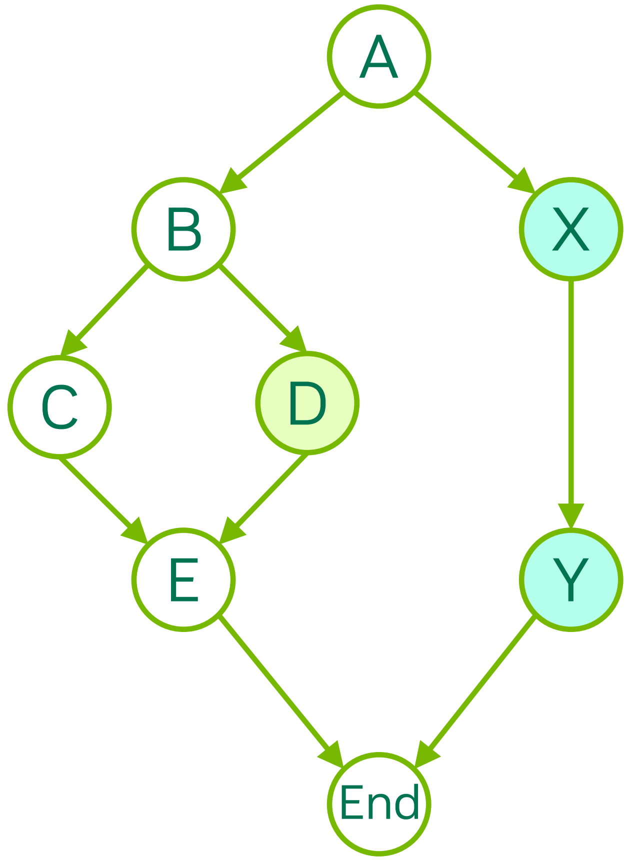
If you can put it in a CUDA stream, you can run it together with a graph

GRAPHS IGNORE STREAM SERIALIZATION RULES

Launch Stream Is Used Only For Ordering With Other Work



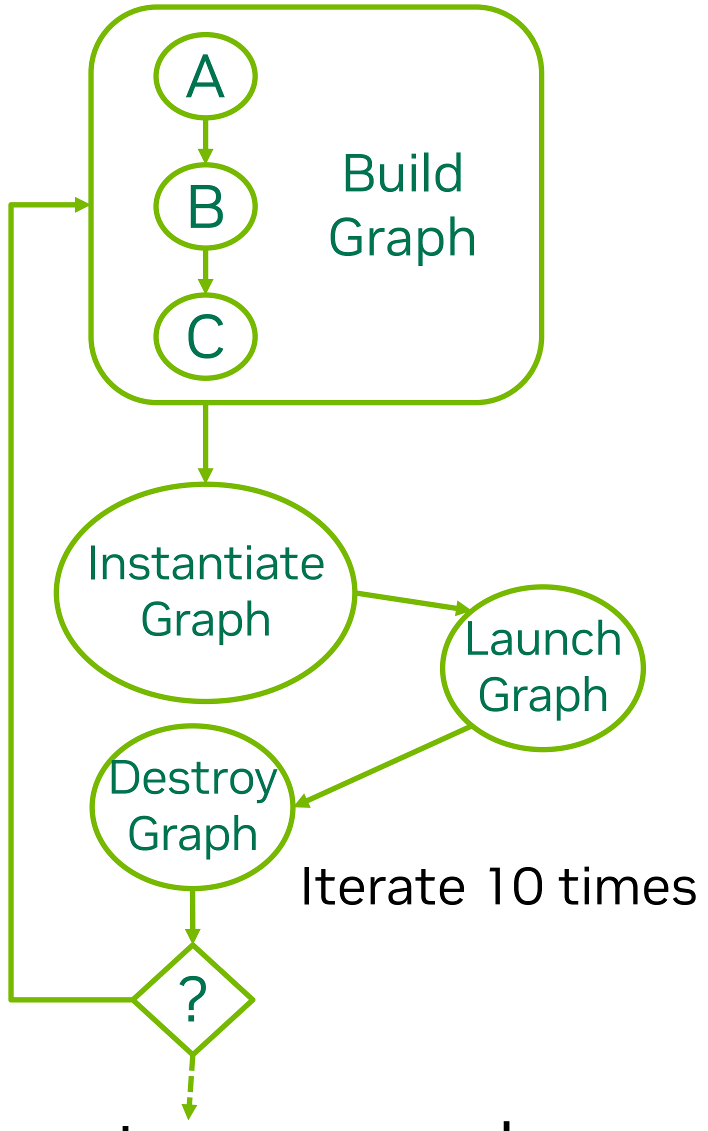
Branches in graph still execute concurrently even though graph is launched into a stream



3 WAYS TO LAUNCH WORK

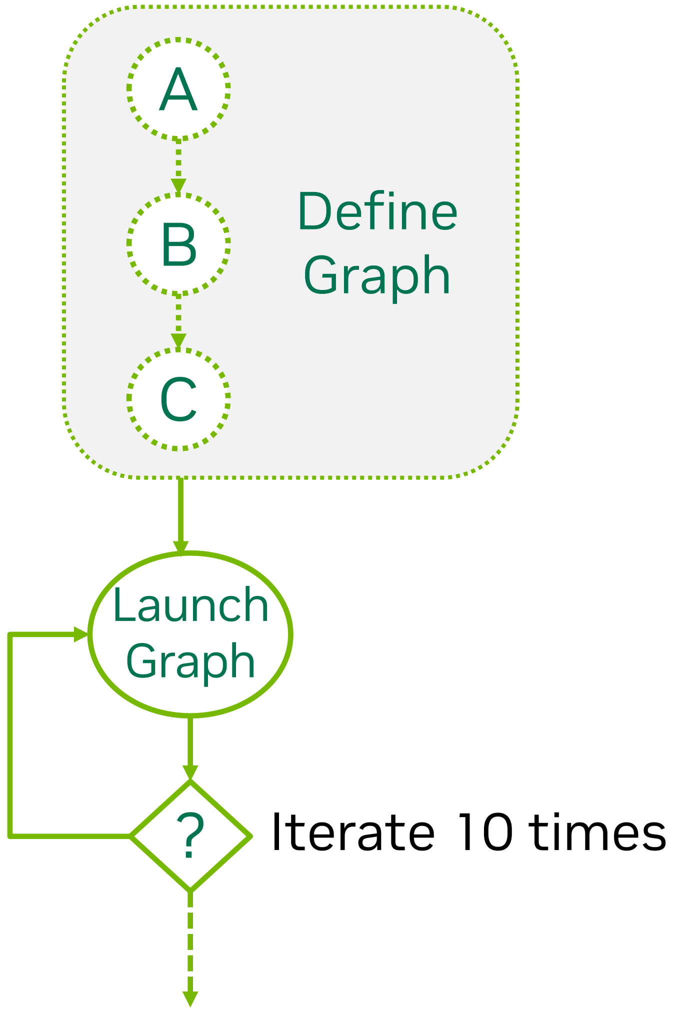
Increasing Performance **comes with** Increasing Restrictions

Graph Re-instantiation



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

Graph Re-Launch

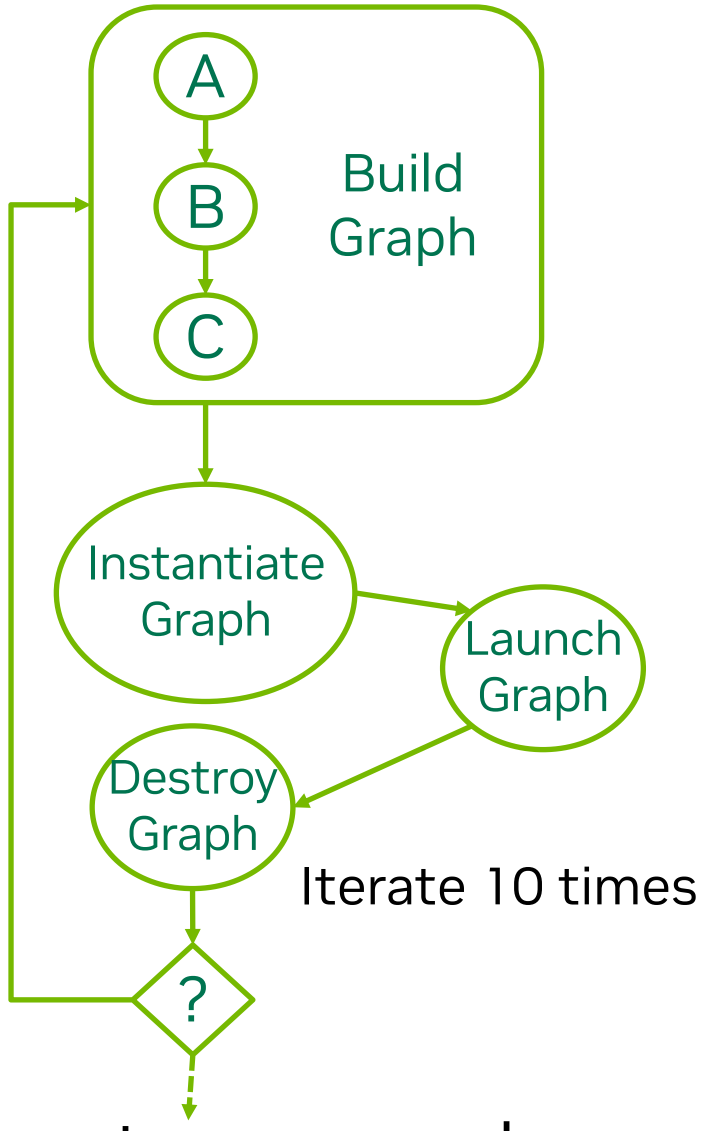


Parameters: **may not** change
Topology: **may not** change

3 WAYS TO LAUNCH WORK

Increasing Performance **comes with** Increasing Restrictions

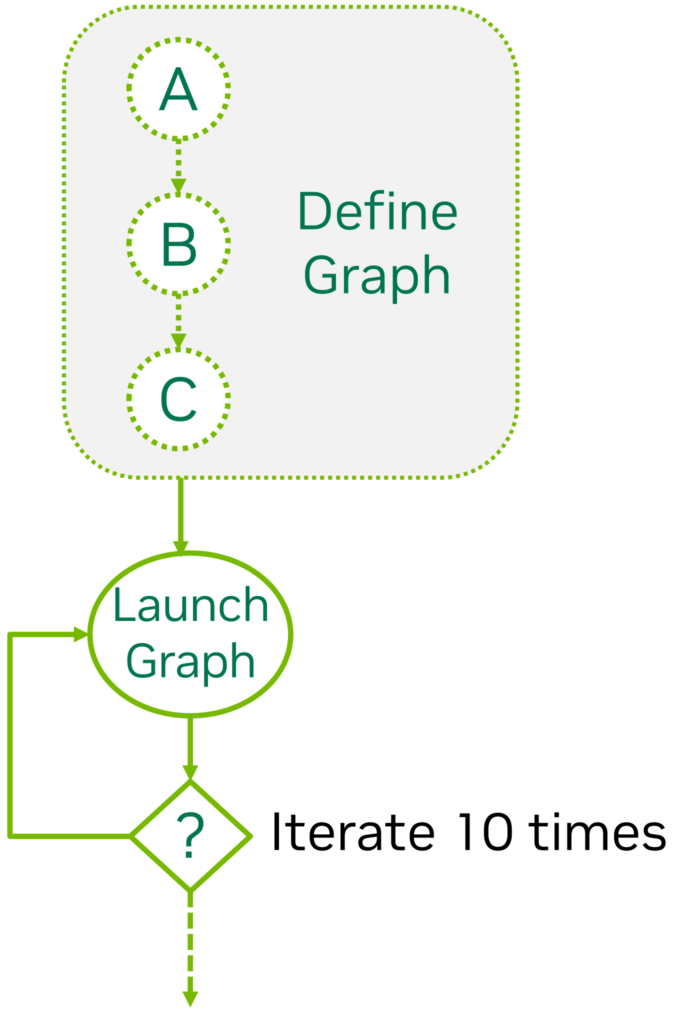
Graph Re-instantiation



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

What if only the parameters have changed? Is re-instantiation the only option?

Graph Re-Launch

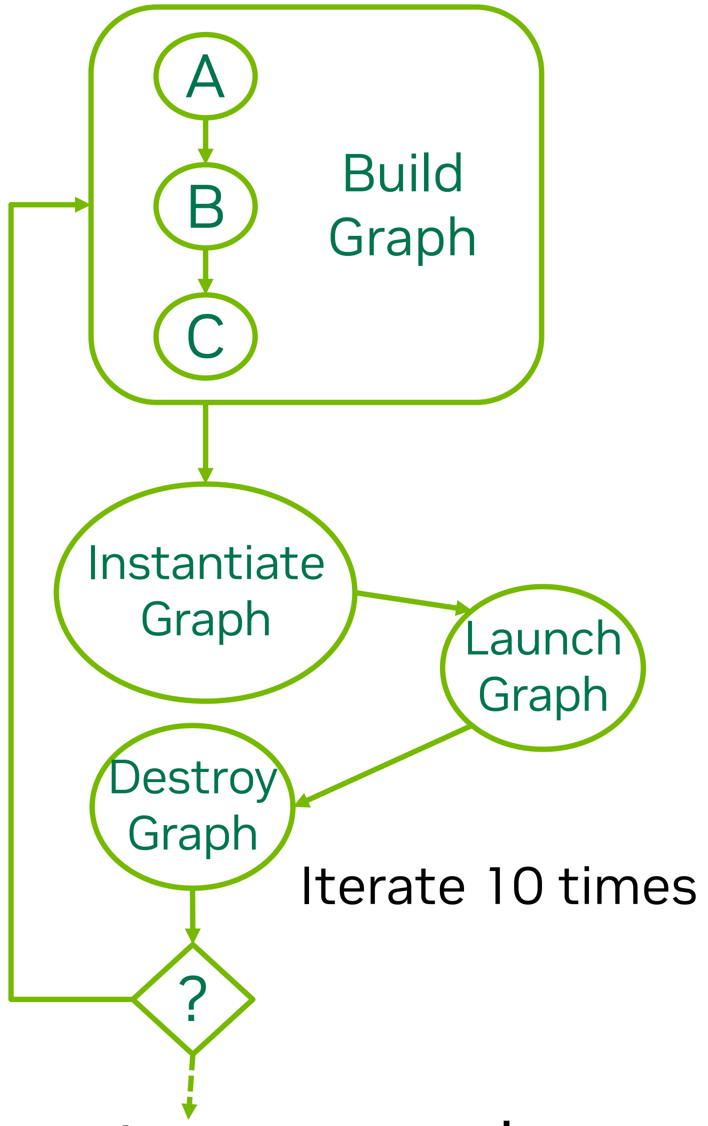


Parameters: **may not** change
Topology: **may not** change

3 WAYS TO LAUNCH WORK

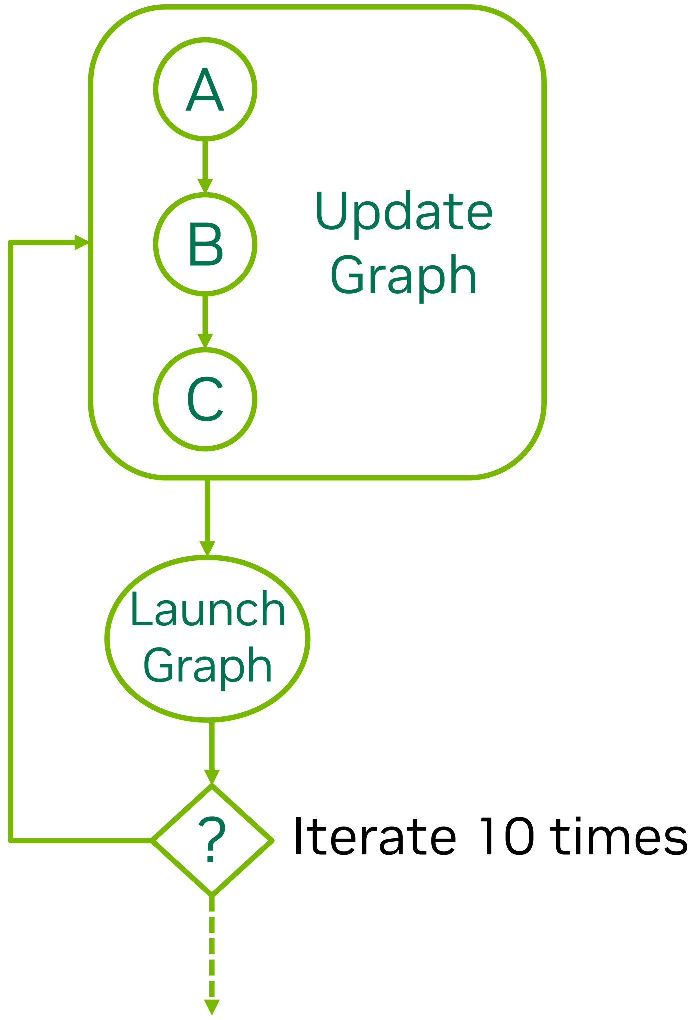
Increasing Performance **comes with** Increasing Restrictions

Graph Re-instantiation



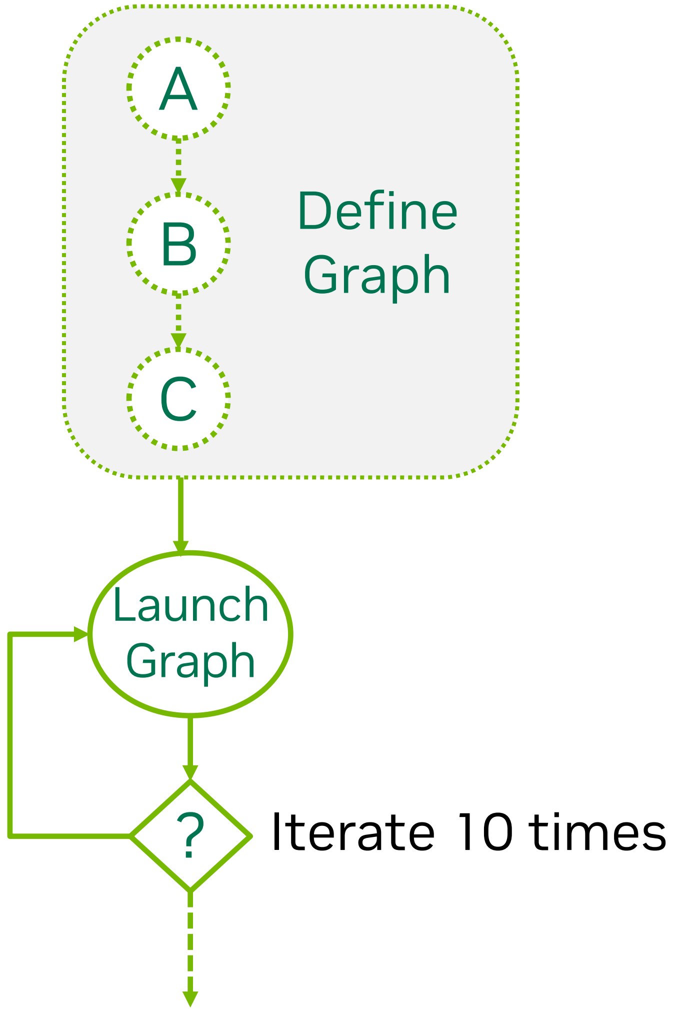
Parameters: **may** change
Topology: **may** change

Graph Update



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

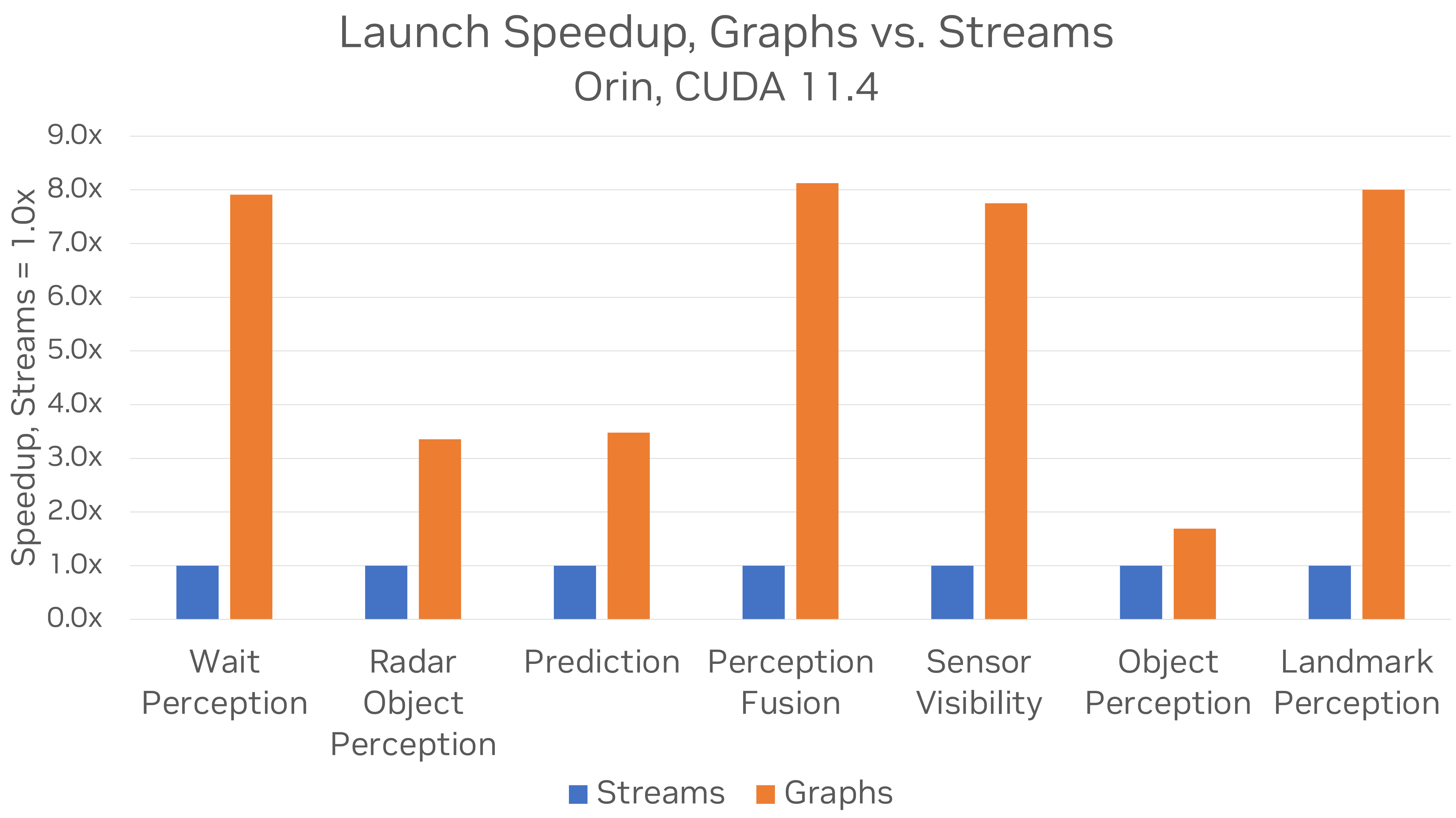
Graph Re-Launch



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

CASE STUDY: AUTONOMOUS VEHICLES

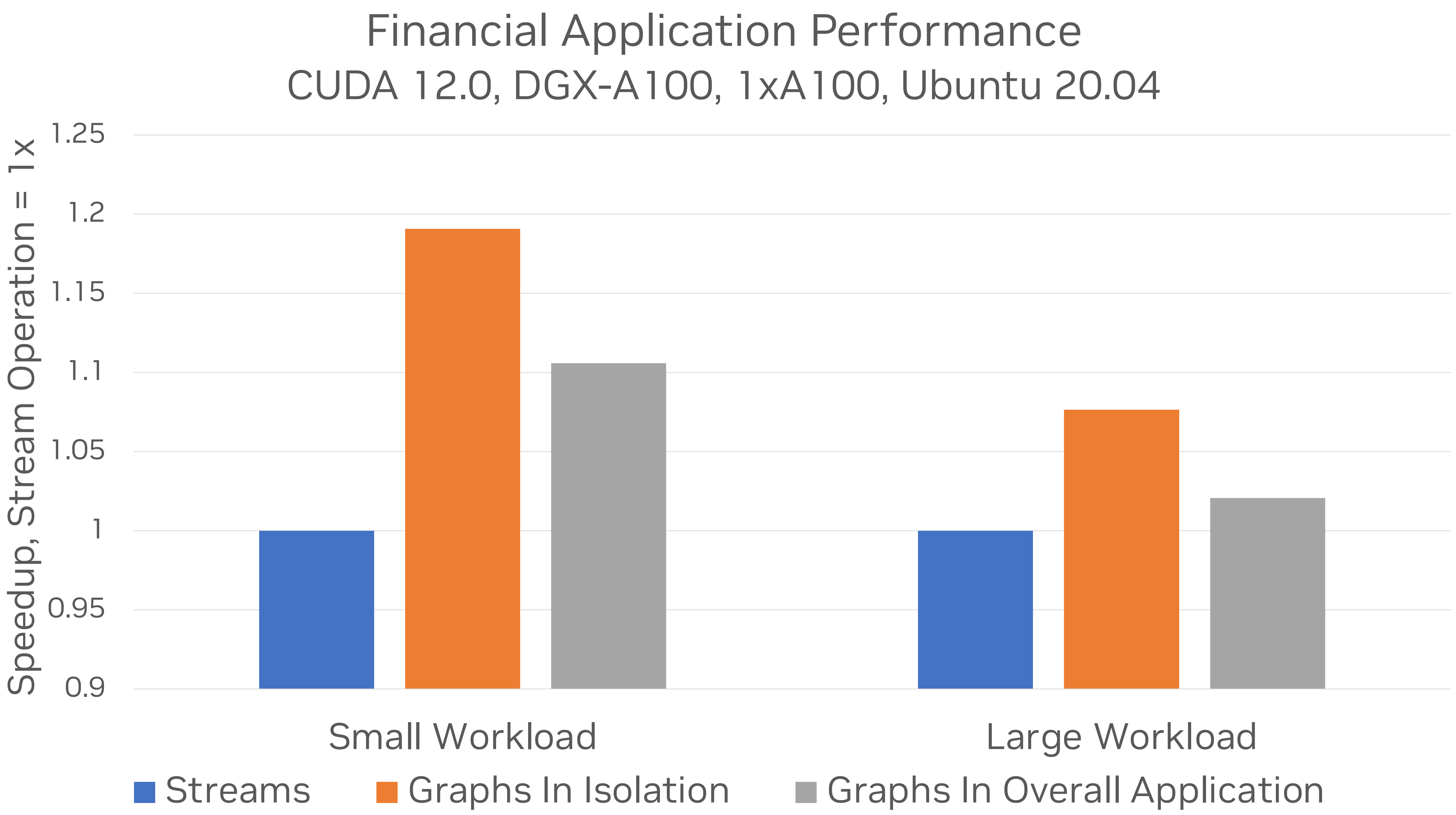
Relaunch Without Update



In real-time systems, graphs must remain static
for reliable launch times

CASE STUDY: FINANCIAL APPLICATION

Relaunch With Update



Graph speedup in isolation is not the full picture

Overall application is 20-30% graphable, so smaller speedups overall

ADAPTING EXISTING CODE TO GRAPHS

Choose Method Based On Program Structure

Graph Re-instantiation

```
for(i=0; i<N; i++) {
    cudaStreamBeginCapture(stream);
    A<<< ..., stream >>>(data);
    B<<< ..., stream >>>(data);
                .
                .
    Z<<< ..., stream >>>(data);
    cudaStreamEndCapture(stream, &g);
    cudaGraphInstantiate(g, &graph);

    cudaGraphLaunch(graph, stream);
    cudaStreamSynchronize(stream);
}
```

Rebuild work every iteration

Not faster than streams

Graph Update

```
for(i=0; i<N; i++) {
    cudaStreamBeginCapture(stream);
    A<<< ..., stream >>>(data[i]);
    B<<< ..., stream >>>(data[i]);
                .
                .
    Z<<< ..., stream >>>(data[i]);
    cudaStreamEndCapture(stream, g);
    cudaGraphExecUpdate(graph, g);

    cudaGraphLaunch(graph, stream);
    cudaStreamSynchronize(stream);
}
```

Update graph every iteration

Up to 1.2x faster than streams

Graph Re-Launch

```
cudaStreamBeginCapture(stream);
A<<< ..., stream >>>(data);
B<<< ..., stream >>>(data);
            .
            .
Z<<< ..., stream >>>(data);
cudaStreamEndCapture(stream, &g);
cudaGraphInstantiate(g, &graph);

for(i=0; i<N; i++) {
    cudaGraphLaunch(graph, stream);
    cudaStreamSynchronize(stream);
}
```

Launch same graph every time

Up to 2.5x faster than streams

SINGLE NODE UPDATE

A More Fine-Grained Method of Updating Parameters

```
// Define graph
cudaGraphCreate(&graph);
cudaGraphAddNode(graph, kernel_a, {}, ...);
...
// Instantiate graph
cudaGraphInstantiate(&graphExec, graph);

// Iterate 100 times
for(int i=0; i<100; i++) {
        generateNewParams(&newParams);
        // Update the parameters for A between launches
        cudaGraphExecKernelNodeSetParams(graphExec, kernel_a, newParams);
        cudaGraphLaunch(graphExec, stream);
}
```

If you know your workflow, you can update nodes individually

SINGLE NODE ENABLE/DISABLE

Avoid Re-Instantiation On Minor Topology Changes

```
// Define graph
cudaGraphCreate(&graph);
cudaGraphAddNode(graph, kernel_a, {}, ...);
...
// Instantiate graph
cudaGraphInstantiate(&graphExec, graph);

// Iterate 100 times
for(int i=0; i<100; i++) {
        checkIfShouldEnable(&enableNode);
        // Toggle A on/off between launches
        cudaGraphNodeSetEnabled(graphExec, kernel_a, enableNode);
        cudaGraphLaunch(graphExec, stream);
}
```

Nodes can also be enabled/disabled entirely

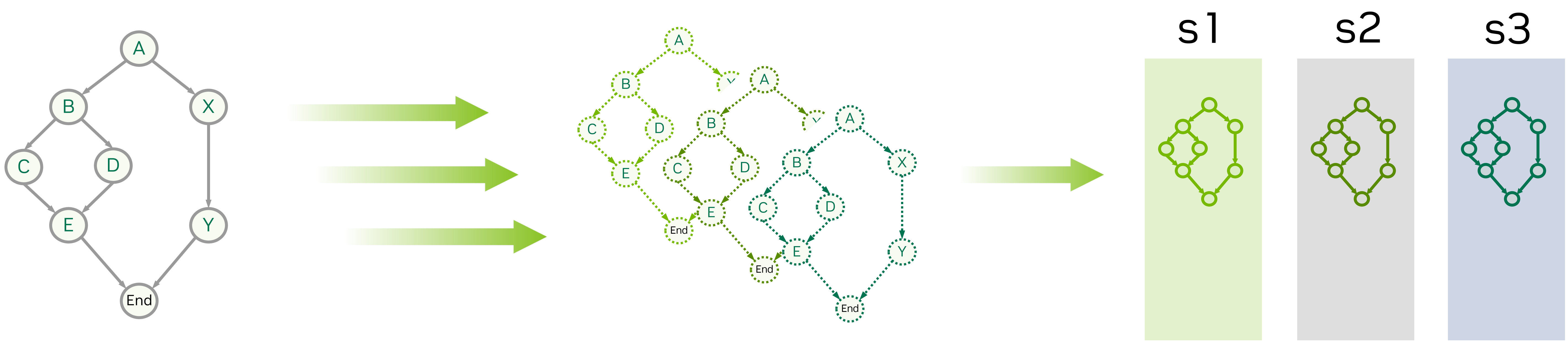
THREE-STAGE EXECUTION MODEL

Minimizes Execution Overheads – Pre-Initialize As Much As Possible

1. Define

2. Instantiate

3. Execute



Single Graph “Template”

Created in host code
or built up from libraries

Multiple
“Executable Graphs”

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

Executable Graphs
Running in CUDA Streams

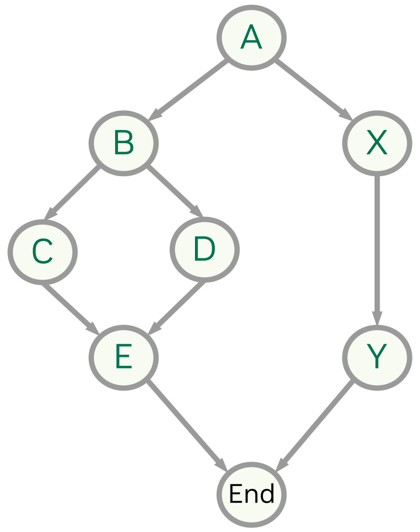
Concurrency in graph
is not limited by stream

FOUR-STAGE EXECUTION MODEL

Upload Step Can Be Separated From Launch For Better Pipelining

Launch

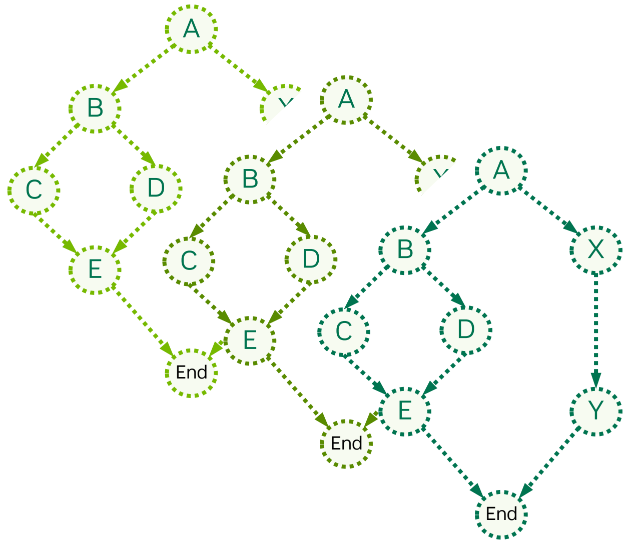
1. Define



Single Graph “Template”

Created in host code
or built up from libraries

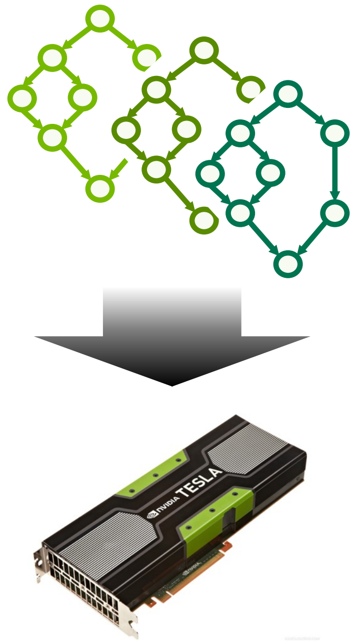
2. Instantiate



Multiple
“Executable Graphs”

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

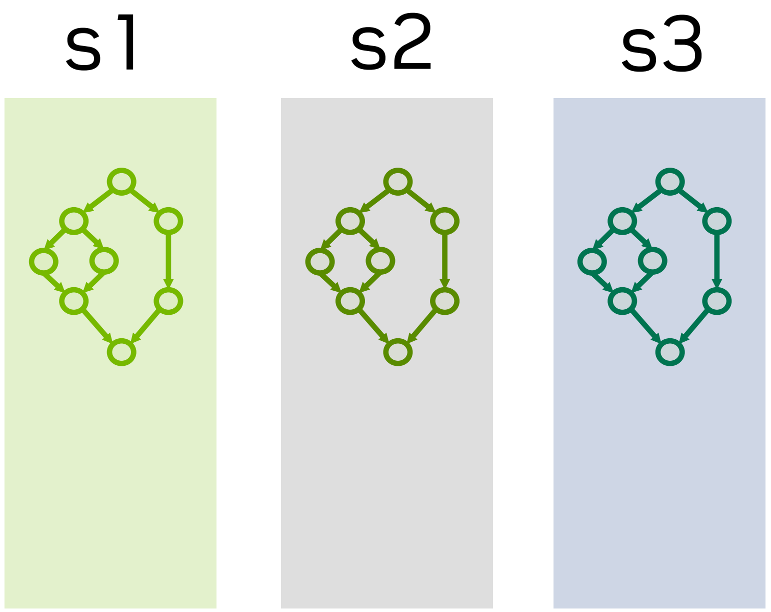
<u>3a. Upload</u>



Pre-Upload Launch
Resources

Usually implicitly performed
as part of first launch
Can be explicitly requested

3b. Execute

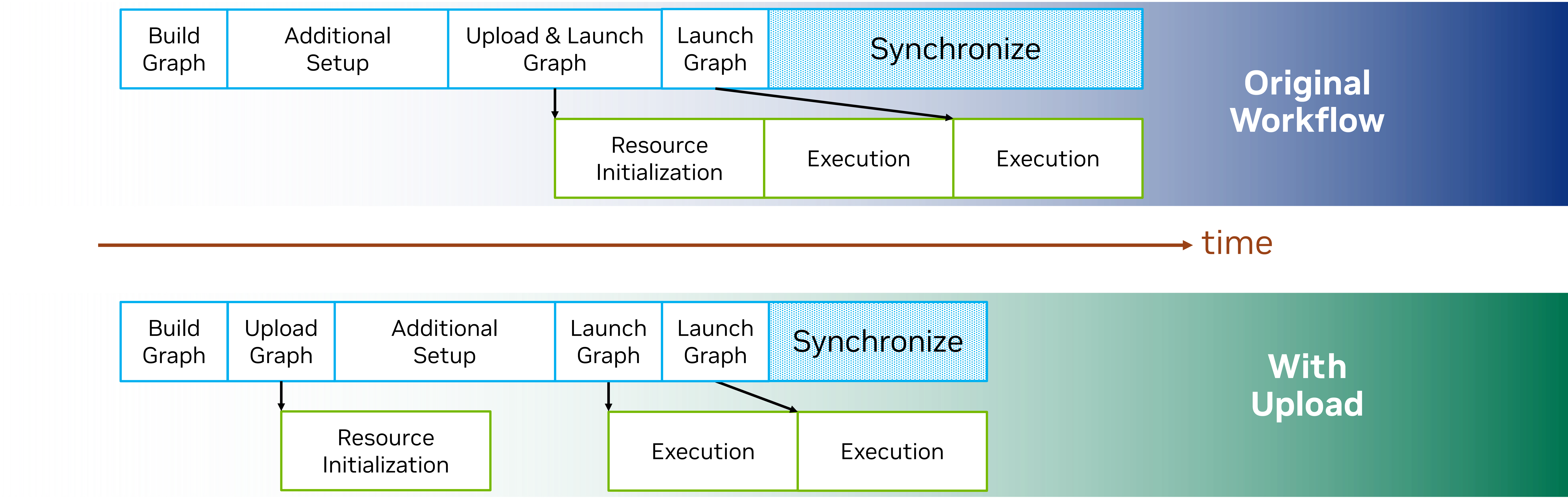


Executable Graphs
Running in CUDA Streams

Concurrency in graph
is not limited by stream

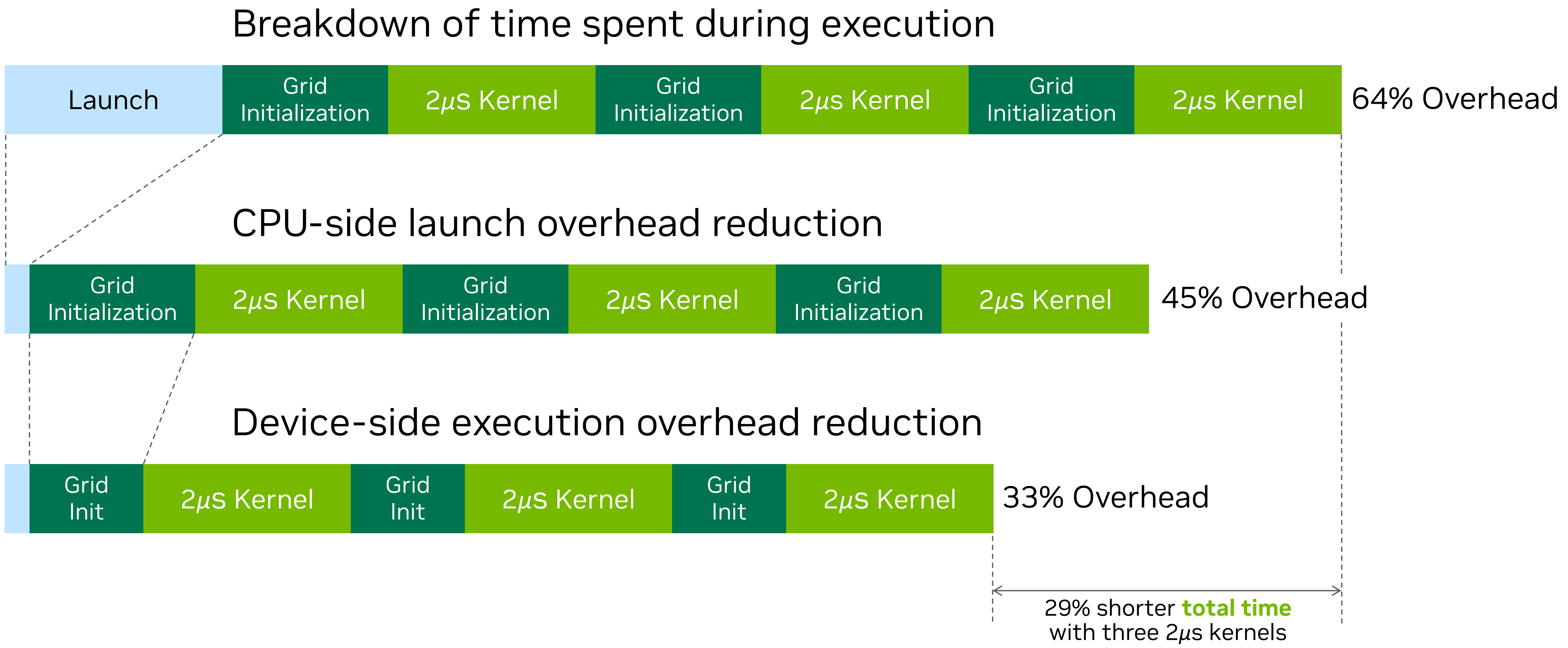
GRAPH UPLOAD

Reduce First Launch Costs



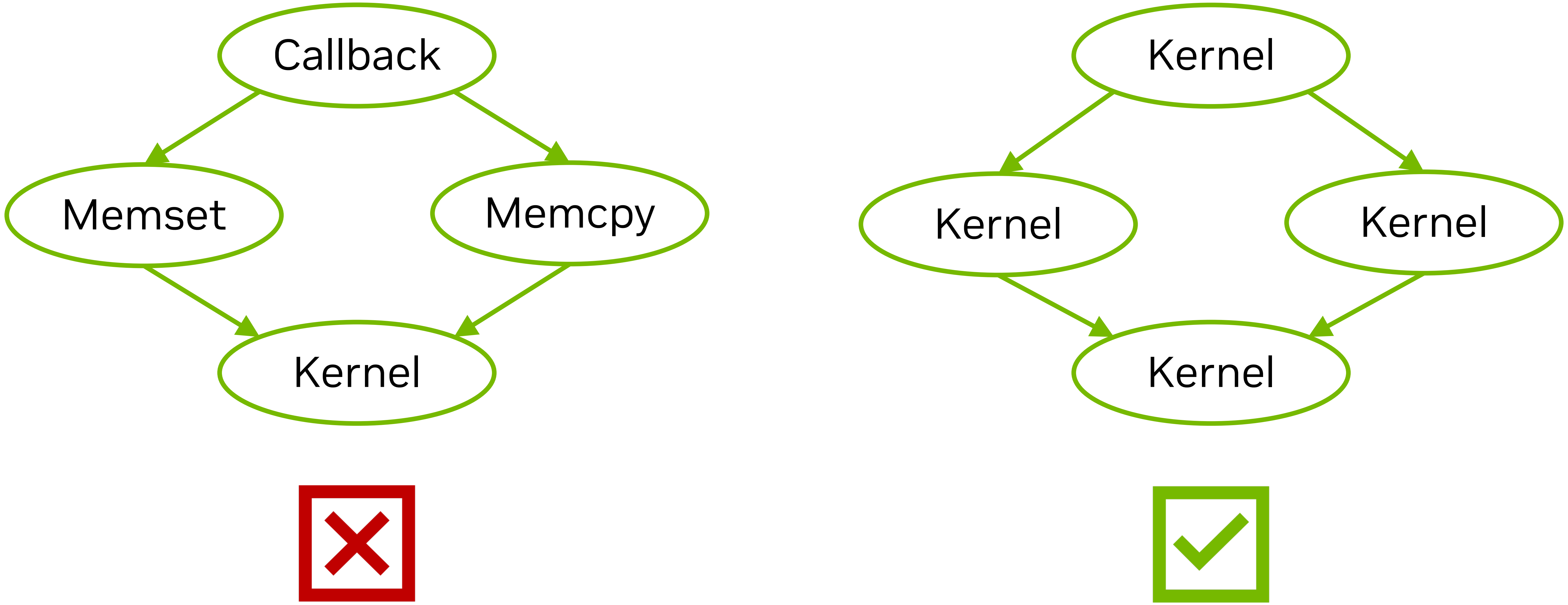
RECAP: WHERE IS PERFORMANCE COMING FROM?

Reducing System Overheads Around Short-Running Kernels



PERFORMANCE TIPS AND TRICKS

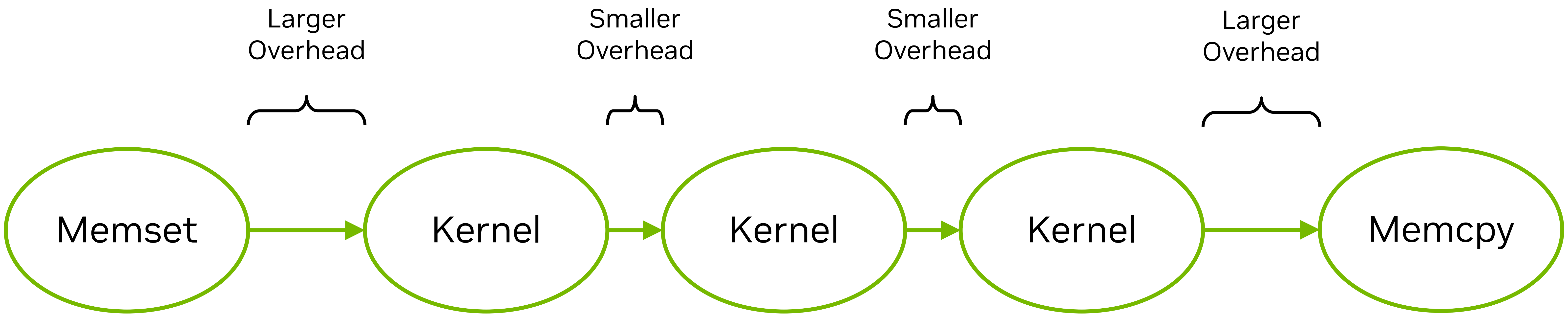
Optimizing GPU Runtime By Focusing On Kernels



Kernel-heavy workflows will see more GPU acceleration

PERFORMANCE TIPS AND TRICKS

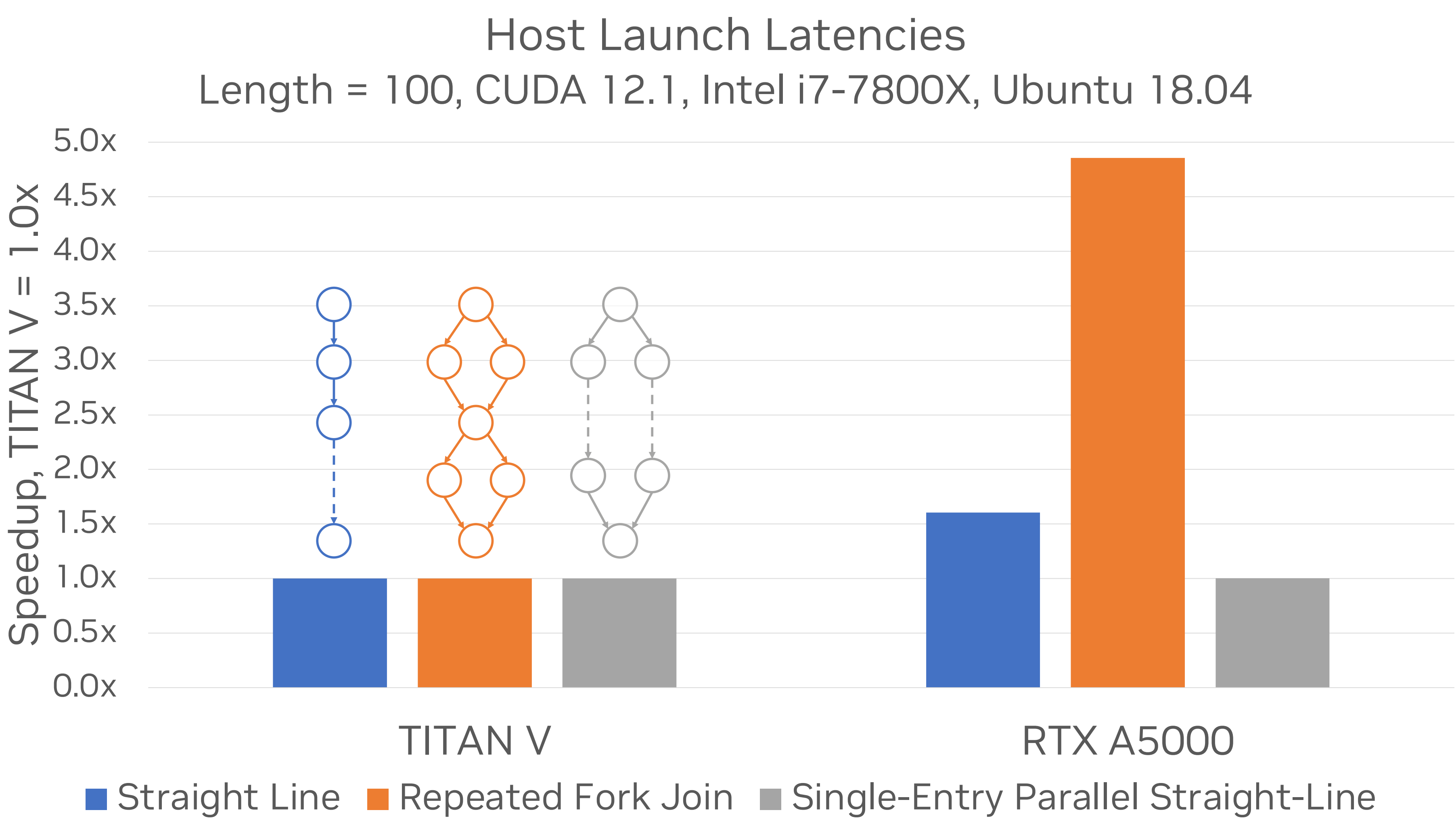
Improving Dependency Resolution

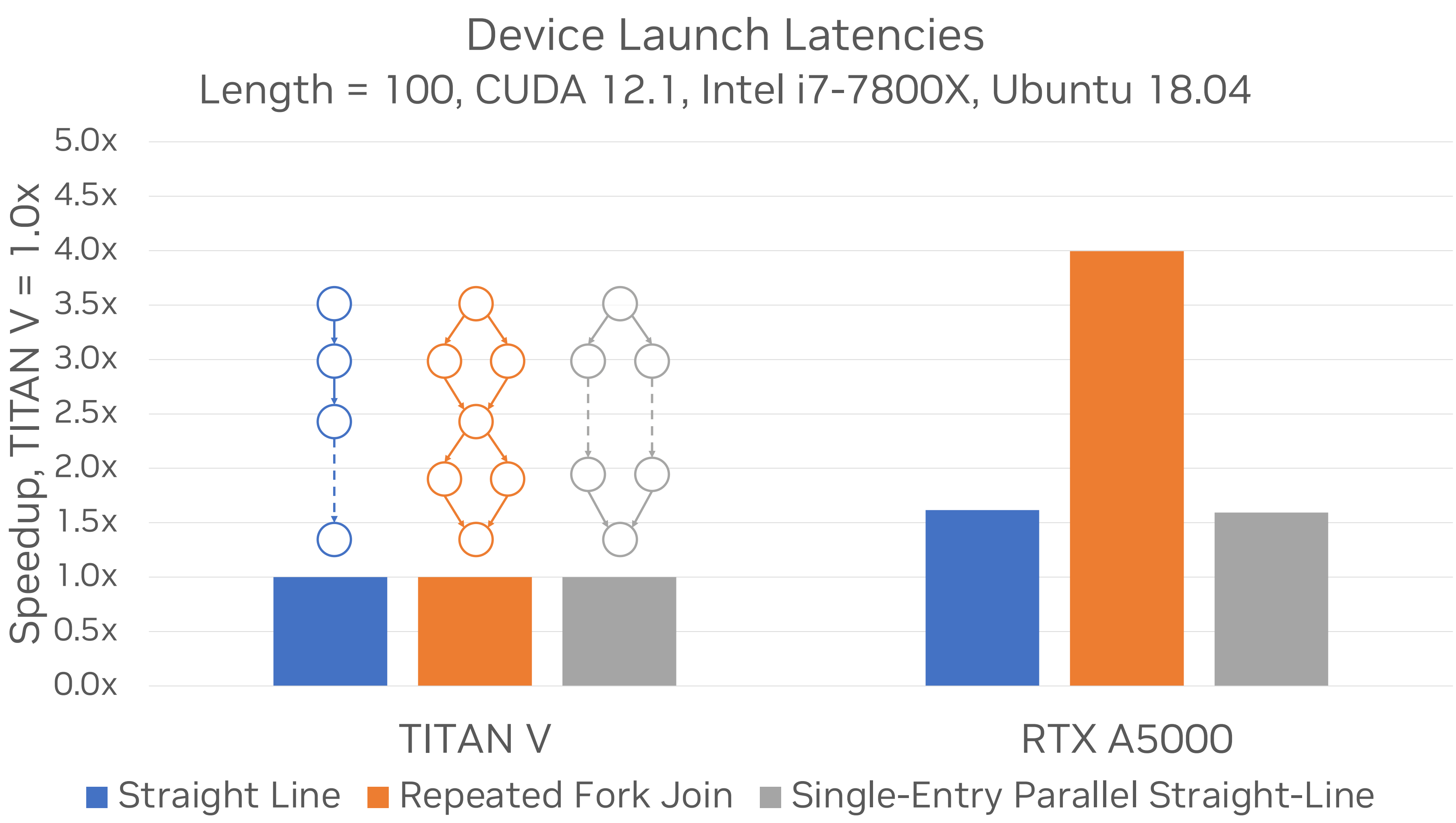


Kernel-to-kernel dependency resolution
is faster than kernel-to-other

PERFORMANCE TIPS AND TRICKS

Ampere Brings In New HW Capabilities For Graphs





Upgrade to Ampere or later to benefit from new HW features

PERFORMANCE TIPS AND TRICKS

Some Tips For Evaluating Performance

Before doing real runs, create and instantiate a maximum-size graph

- Warms up driver resources so subsequent graphs are instantiated faster

Benchmark first launch separately from second launch

- First launch contains an upload step that is absent from second launch

Use the right tool for the job

- Profiler is better for profiling individual nodes
- CUDA events are better for whole-graph timings

ICYMI: WHAT'S NEW SINCE CUDA 11

CUDA 11.0

CUDA 11.1

- Event Nodes
- External Dependency Nodes
- Empty Node Update
- Child Graph Update
- Graph Upload

CUDA 11.2

- Kernel Node Function Update

CUDA 11.3

- User Objects For Graphs
- Generation of a DOT File of a Graph For Debugging

CUDA 11.4

- Memory Allocation & Free Nodes

CUDA 11.6

- Single-Node Enable/Disable

CUDA 11.7

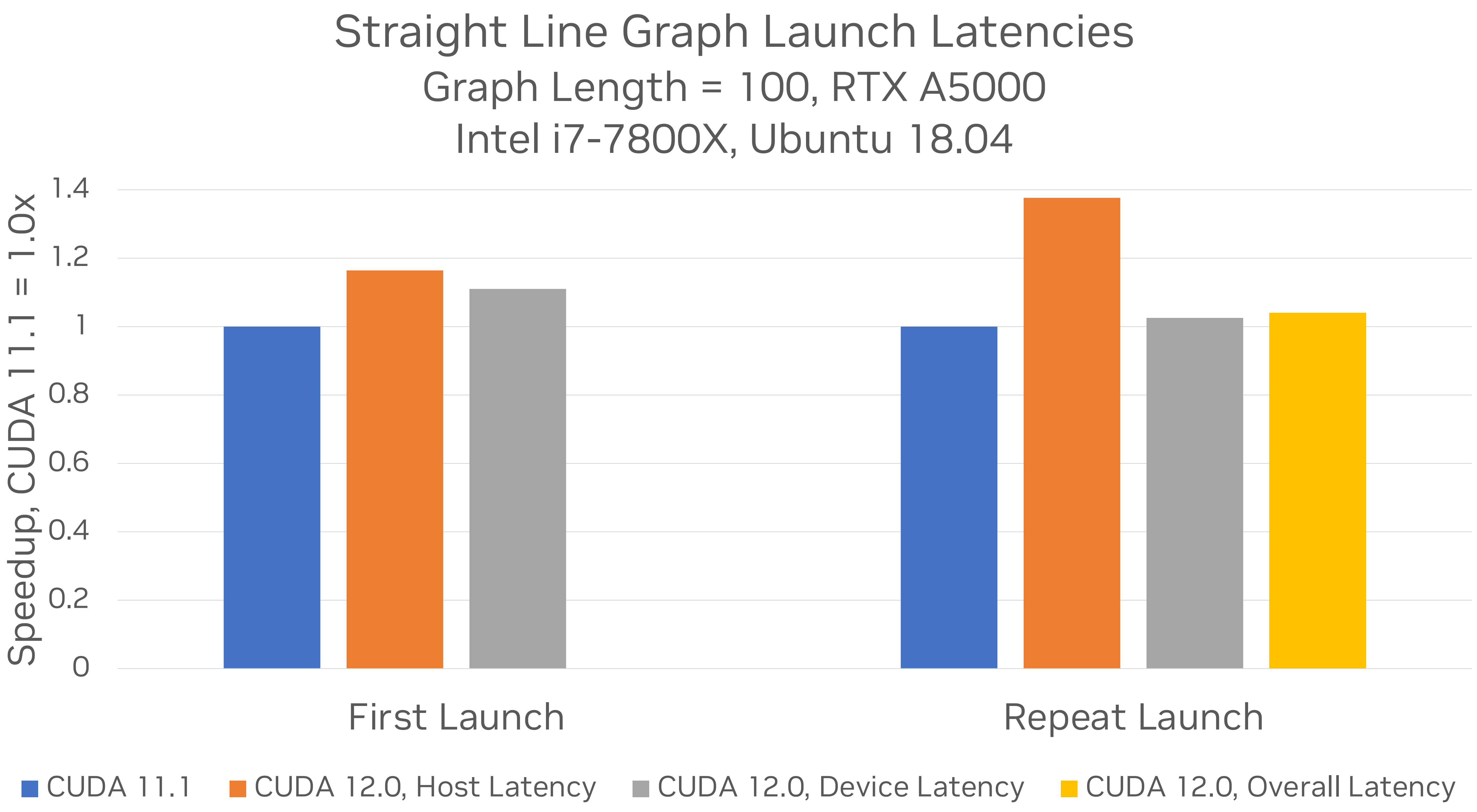
- Per-node Priorities

CUDA 12.0

- Device Graph Launch

ICYMI: WHAT'S NEW SINCE CUDA 11

Performance Improvements



Performance has also improved since CUDA 11!

ICYMI: WHAT'S NEW SINCE CUDA 11

CUDA 11.0

CUDA 11.1

- Event Nodes
- External Dependency Nodes
- Empty Node Update
- Child Graph Update
- Graph Upload

CUDA 11.2

- Kernel Node Function Update

CUDA 11.3

- User Objects For Graphs
- Generation of a DOT File of a Graph For Debugging

CUDA 11.4

- Memory Allocation & Free Nodes

CUDA 11.6

- Single-Node Enable/Disable

CUDA 11.7

- Per-node Priorities

CUDA 12.0

- **Device Graph Launch**

DEVICE GRAPH LAUNCH

Dynamic Control Flow For Graphs

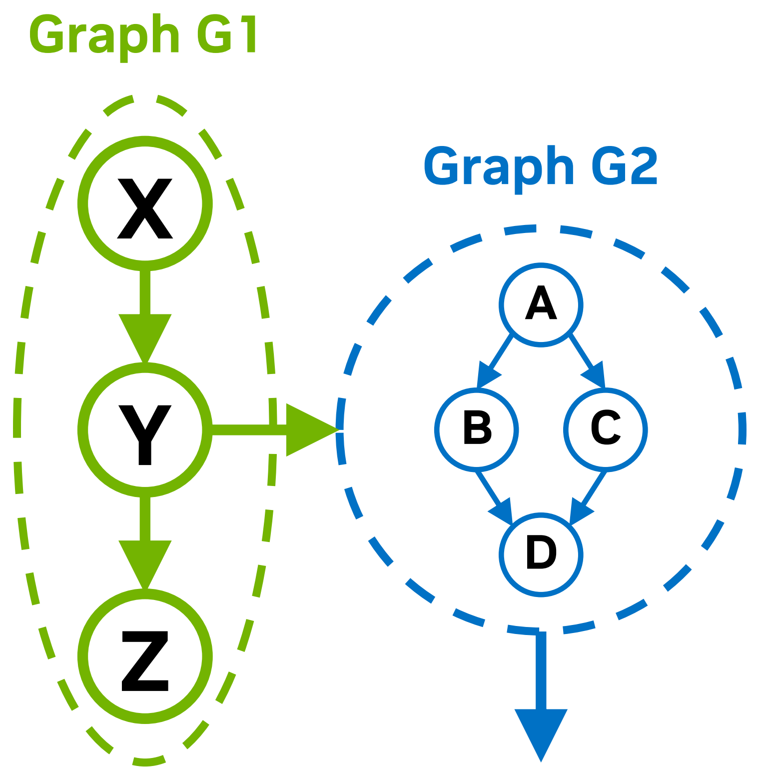
device_launch.cu

CPU portion

```
void main() {
    cudaGraphCreate(&G1);
    // Build graph G1 = XYZ
    cudaGraphInstantiate(G1);

    cudaGraphCreate(&G2);
    // Build graph G2 = ABCD
    cudaGraphInstantiate(G2, DeviceLaunch);
    cudaGraphUpload(G2, ...);

    cudaGraphLaunch(G1, ...);
}
```

GPU portion

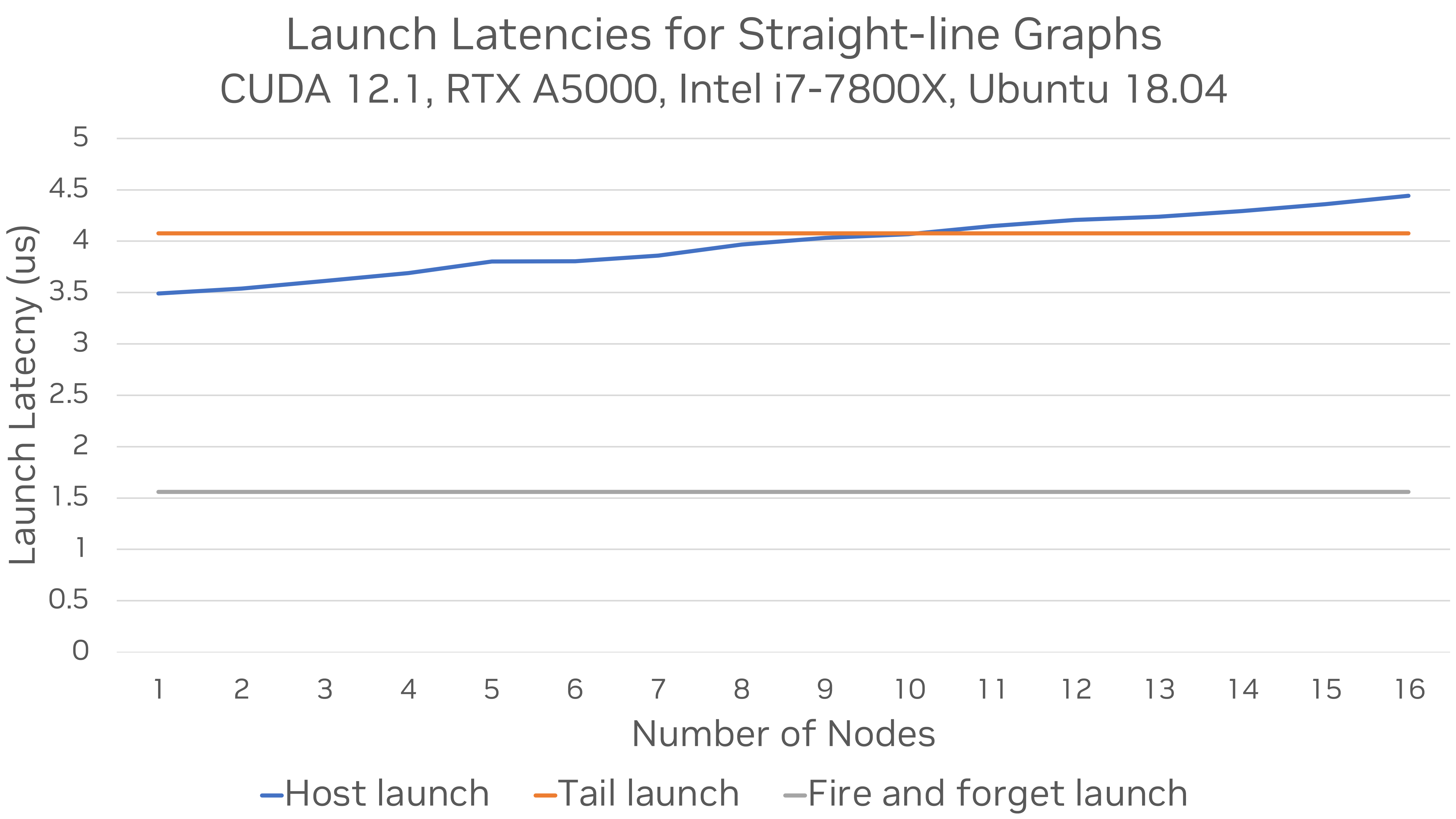
```
__global__ void Y(cudaGraphExec_t G2) {
    cudaGraphLaunch(G2, ...);
}
```



Device-side graph launch

DEVICE LAUNCH PERFORMANCE

How does it compare to host launch?

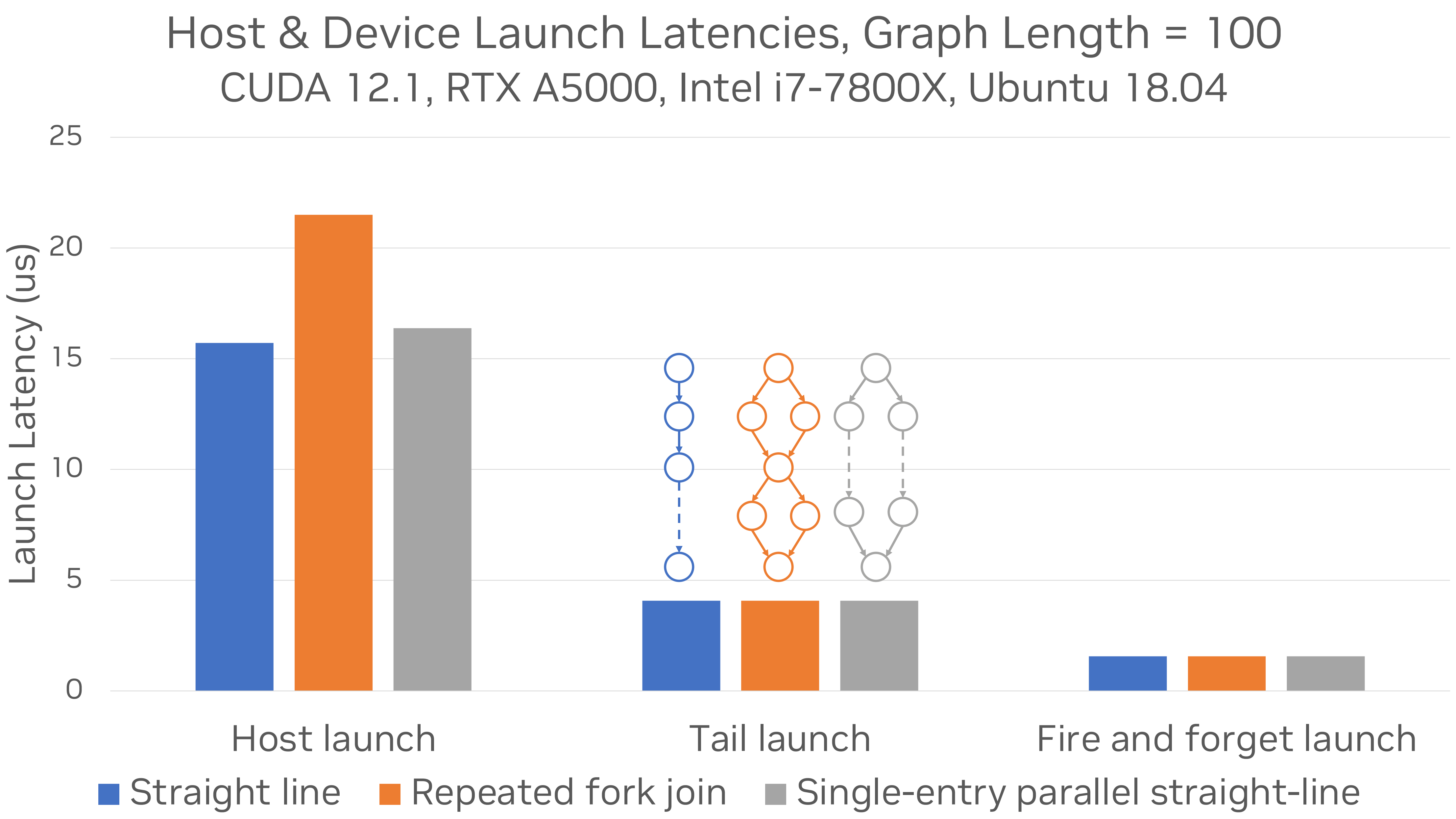


All you need is 11 nodes for device relaunch to be faster than host launch

And fire-and-forget launch is always at least 2x faster than host launch!

DEVICE LAUNCH PERFORMANCE

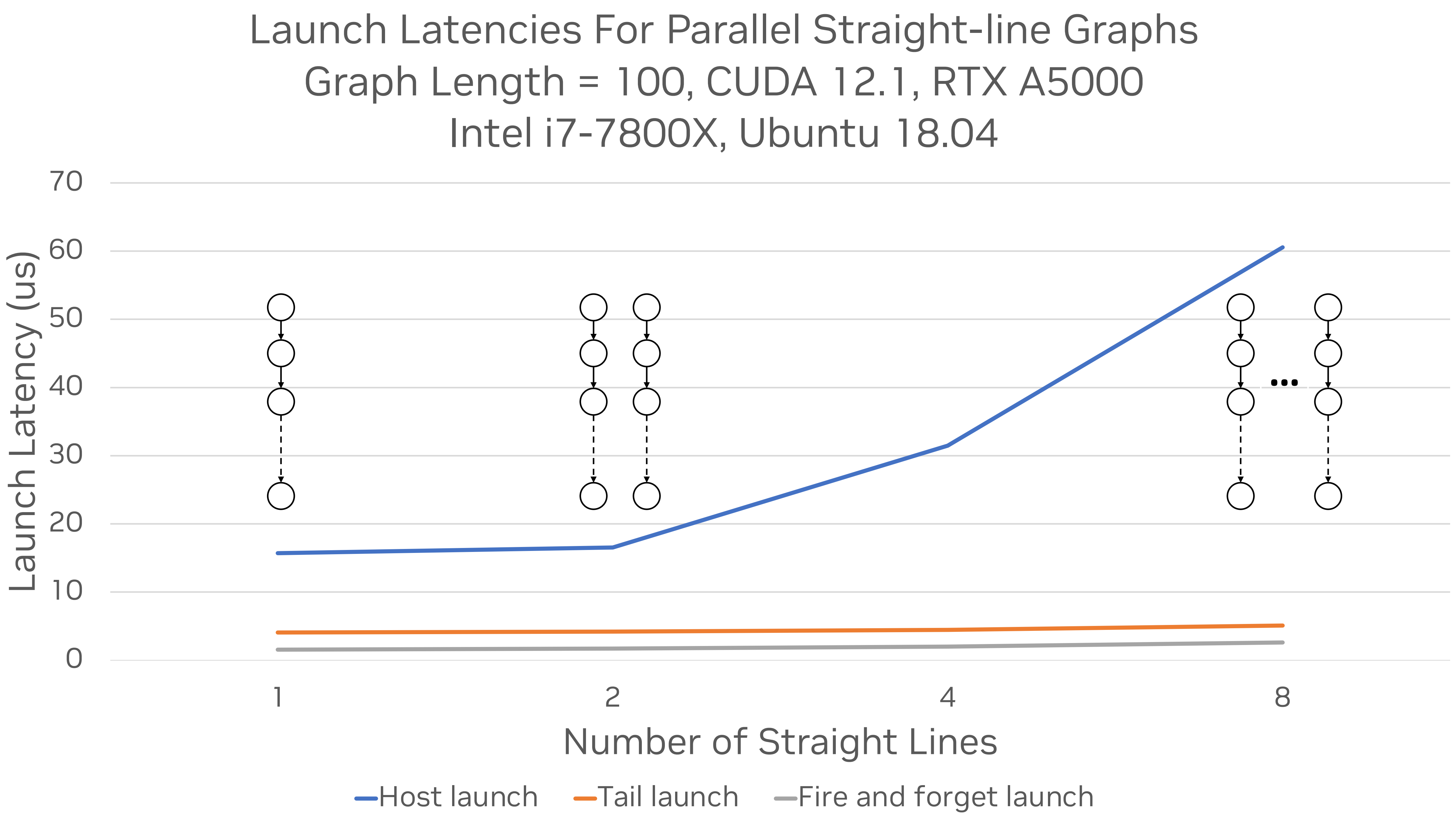
How does it compare to host launch?



Device launch is less impacted by graph structure

DEVICE LAUNCH PERFORMANCE

How does it compare to host launch?



Device launch also scales better to graph width

DEVICE GRAPH CREATION

Rules For Device Graphs

A graph cannot be launched from the device unless…

1. It contains only kernels, memcopies, and memsets

device_launch.cu

CPU

```
void main() {
    cudaGraphCreate(&G1);
    // Build graph G1
    cudaGraphInstantiate(G1);

    cudaGraphCreate(&G2);
    // Build graph G2
    cudaGraphInstantiate(G2, DeviceLaunch);
    cudaGraphUpload(G2, ...);

    cudaGraphLaunch(G1, ...);
}
```

GPU

```
__global__ void Y(cudaGraphExec_t G2) {
    cudaGraphLaunch(G2, ...);
}
```

DEVICE GRAPH CREATION

Rules For Device Graphs

A graph cannot be launched from the device unless…

1. It contains only kernels, memcopies, and memsets
2. All nodes reside on a single device

device_launch.cu

```
void main() {
    cudaGraphCreate(&G1);
    // Build graph G1
    cudaGraphInstantiate(G1);

    cudaGraphCreate(&G2);
    // Build graph G2
    cudaGraphInstantiate(G2, DeviceLaunch);
    cudaGraphUpload(G2, ...);

    cudaGraphLaunch(G1, ...);
}
```

CPU

```
__global__ void Y(cudaGraphExec_t G2) {
    cudaGraphLaunch(G2, ...);
}
```

GPU

DEVICE GRAPH CREATION

Rules For Device Graphs

A graph cannot be launched from the device unless...

1. It contains only kernels, memcopies, and memsets
2. All nodes reside on a single device
3. It is instantiated for device launch

device_launch.cu

CPU

```
void main() {
    cudaGraphCreate(&G1);
    // Build graph G1
    cudaGraphInstantiate(G1);

    cudaGraphCreate(&G2);
    // Build graph G2
    cudaGraphInstantiate(G2, DeviceLaunch);
    cudaGraphUpload(G2, ...);

    cudaGraphLaunch(G1, ...);
}
```

GPU

```
__global__ void Y(cudaGraphExec_t G2) {
    cudaGraphLaunch(G2, ...);
}
```

DEVICE GRAPH CREATION

Rules For Device Graphs

A graph cannot be launched from the device unless…

1. It contains only kernels, memcopies, and memsets
2. All nodes reside on a single device
3. It is instantiated for device launch
4. It has been explicitly uploaded to the device
   (if not launched from the host)

device_launch.cu

CPU

```
void main() {
    cudaGraphCreate(&G1);
    // Build graph G1
    cudaGraphInstantiate(G1);

    cudaGraphCreate(&G2);
    // Build graph G2
    cudaGraphInstantiate(G2, DeviceLaunch);
    cudaGraphUpload(G2, ...);

    cudaGraphLaunch(G1, ...);
}
```

GPU

```
__global__ void Y(cudaGraphExec_t G2) {
    cudaGraphLaunch(G2, ...);
}
```

DEVICE GRAPH CREATION

Rules For Device Graphs

A graph cannot be launched from the device unless...

1. It contains only kernels, memcopies, and memsets
2. All nodes reside on a single device
3. It is instantiated for device launch
4. It has been explicitly uploaded to the device (if not launched from the host)
5. It is launched from another graph

device_launch.cu

```
void main() {
    cudaGraphCreate(&G1);
    // Build graph G1
    cudaGraphInstantiate(G1);

    cudaGraphCreate(&G2);
    // Build graph G2
    cudaGraphInstantiate(G2, DeviceLaunch);
    cudaGraphUpload(G2, ...);

    cudaGraphLaunch(G1, ...);
}
```

CPU

```
__global__ void Y(cudaGraphExec_t G2) {
    cudaGraphLaunch(G2, ...);
}
```

GPU

DEVICE GRAPH CREATION

Rules For Device Graphs

A graph cannot be launched from the device unless…

1. It contains only kernels, memcopies, and memsets
2. All nodes reside on a single device
3. It is instantiated for device launch
4. It has been explicitly uploaded to the device (if not launched from the host)
5. It is launched from another graph

Graphs that are device-launchable are **device graphs**

- All other graphs are **host graphs**

device_launch.cu

CPU

```
void main() {
    cudaGraphCreate(&G1);
    // Build graph G1
    cudaGraphInstantiate(G1);

    cudaGraphCreate(&G2);
    // Build graph G2
    cudaGraphInstantiate(G2, DeviceLaunch);
    cudaGraphUpload(G2, ...);

    cudaGraphLaunch(G1, ...);
}
```

Host Graph

Device Graph

GPU

```
__global__ void Y(cudaGraphExec_t G2) {
    cudaGraphLaunch(G2, ...);
}
```

DEVICE GRAPH LAUNCH ENCAPSULATION

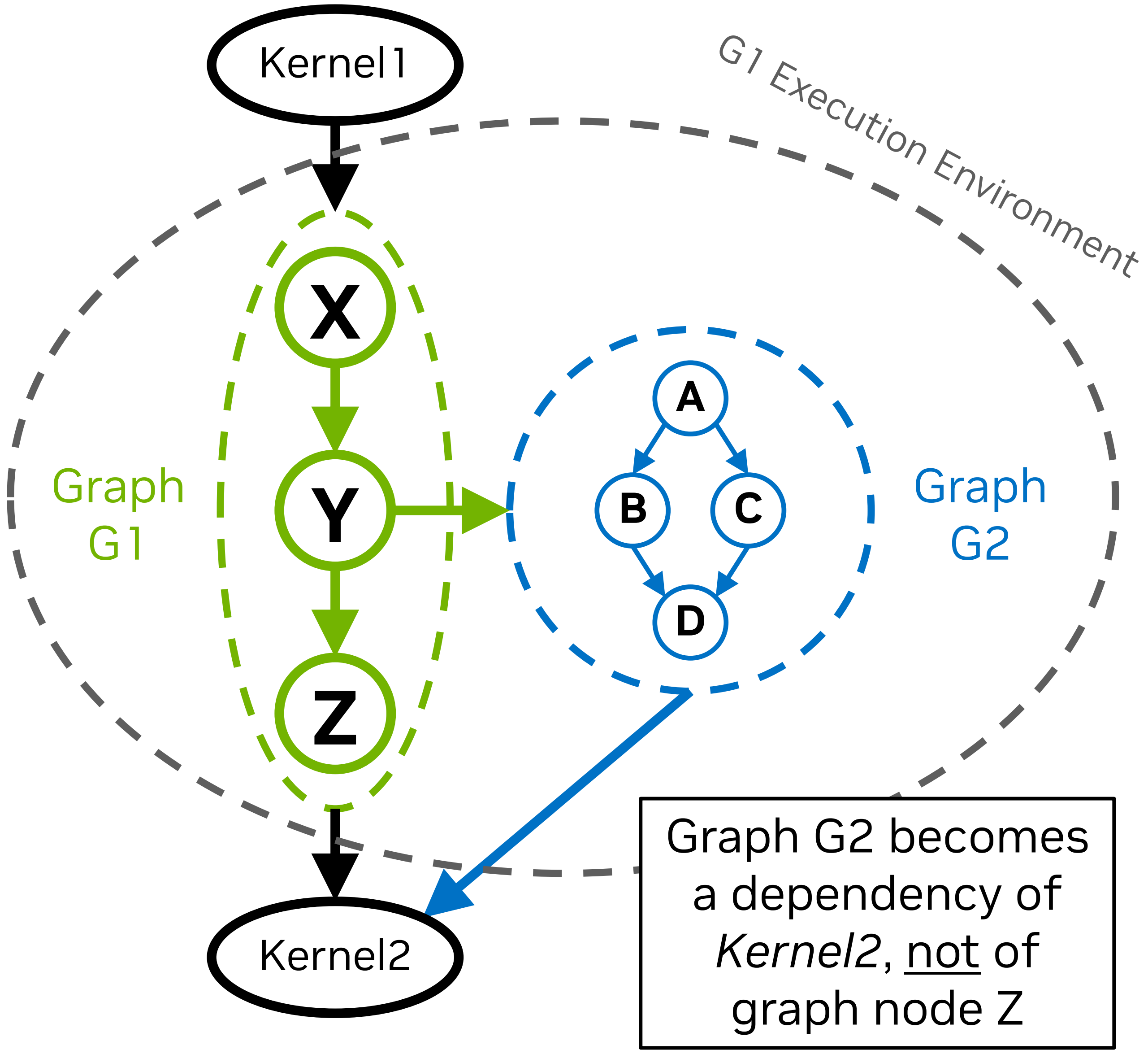
Dependency Resolution Occurs At Whole-Graph Granularity

Graph encapsulation boundary is the whole launching graph

- This boundary is called the **execution environment**

Graph launch cannot create a new dependency within the parent graph

- No fork/join parallelism inside a graph



DEVICE GRAPH LAUNCH MODES

Named Stream: Fire-and-Forget

Fire-and-forget mode launches the graph immediately

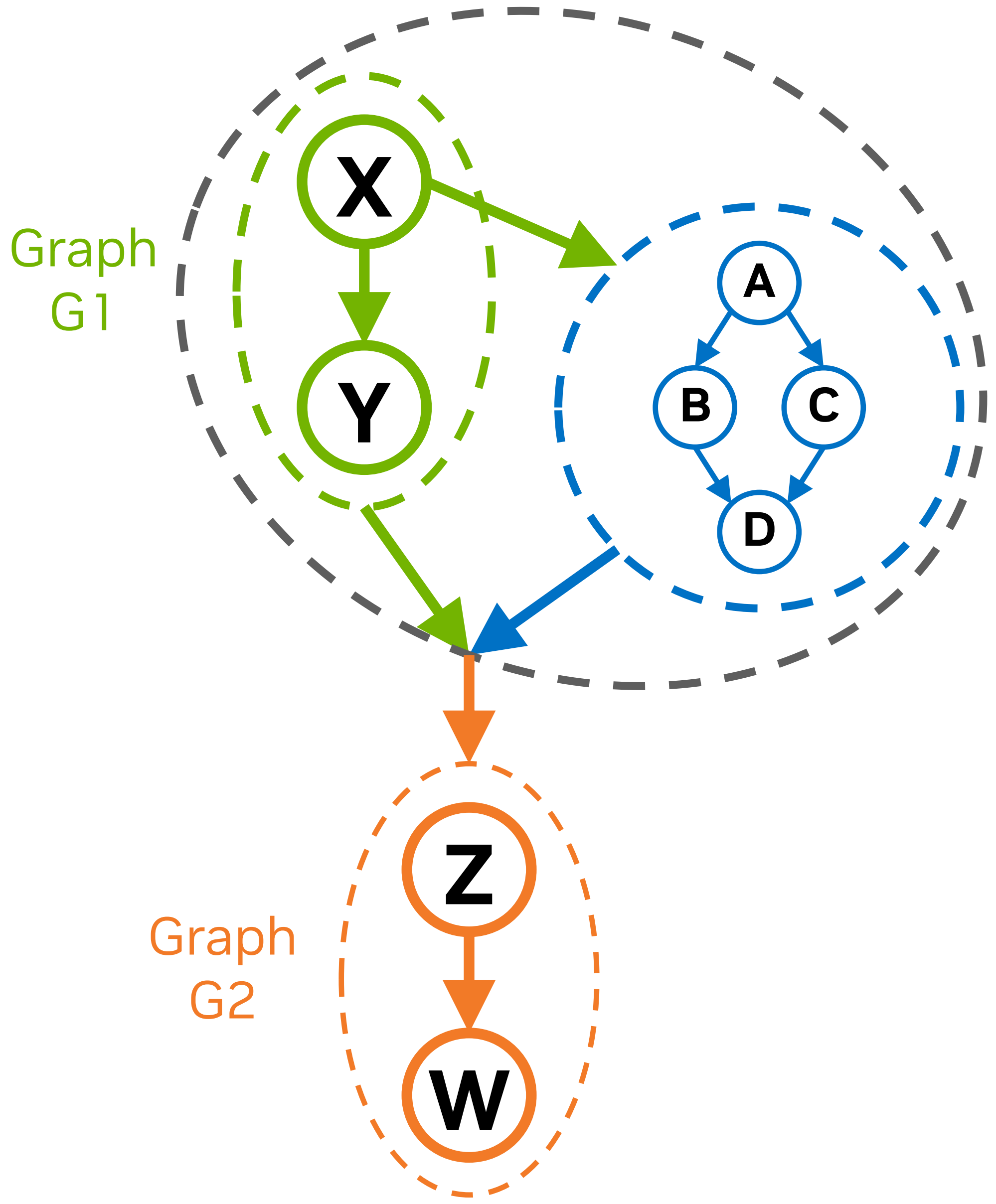
- Launched graph runs concurrently with parent

Subsequent work will implicitly join fire-and-forget launches

Fire-and-forget launches cannot be synced directly

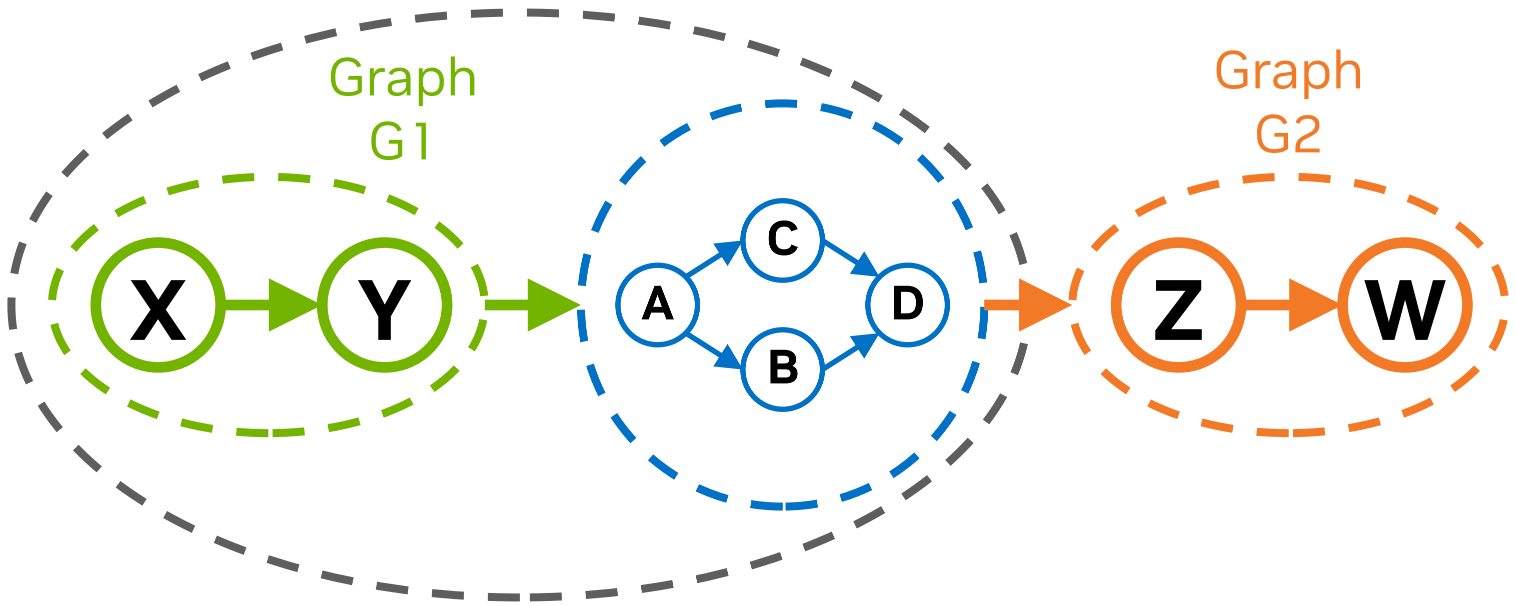
- Ie, via cudaDeviceSynchronize()

…So how do I insert work dependencies?



DEVICE GRAPH LAUNCH MODES

Named Stream: Tail Launch



Tail launches are launched sequentially after the calling graph completes

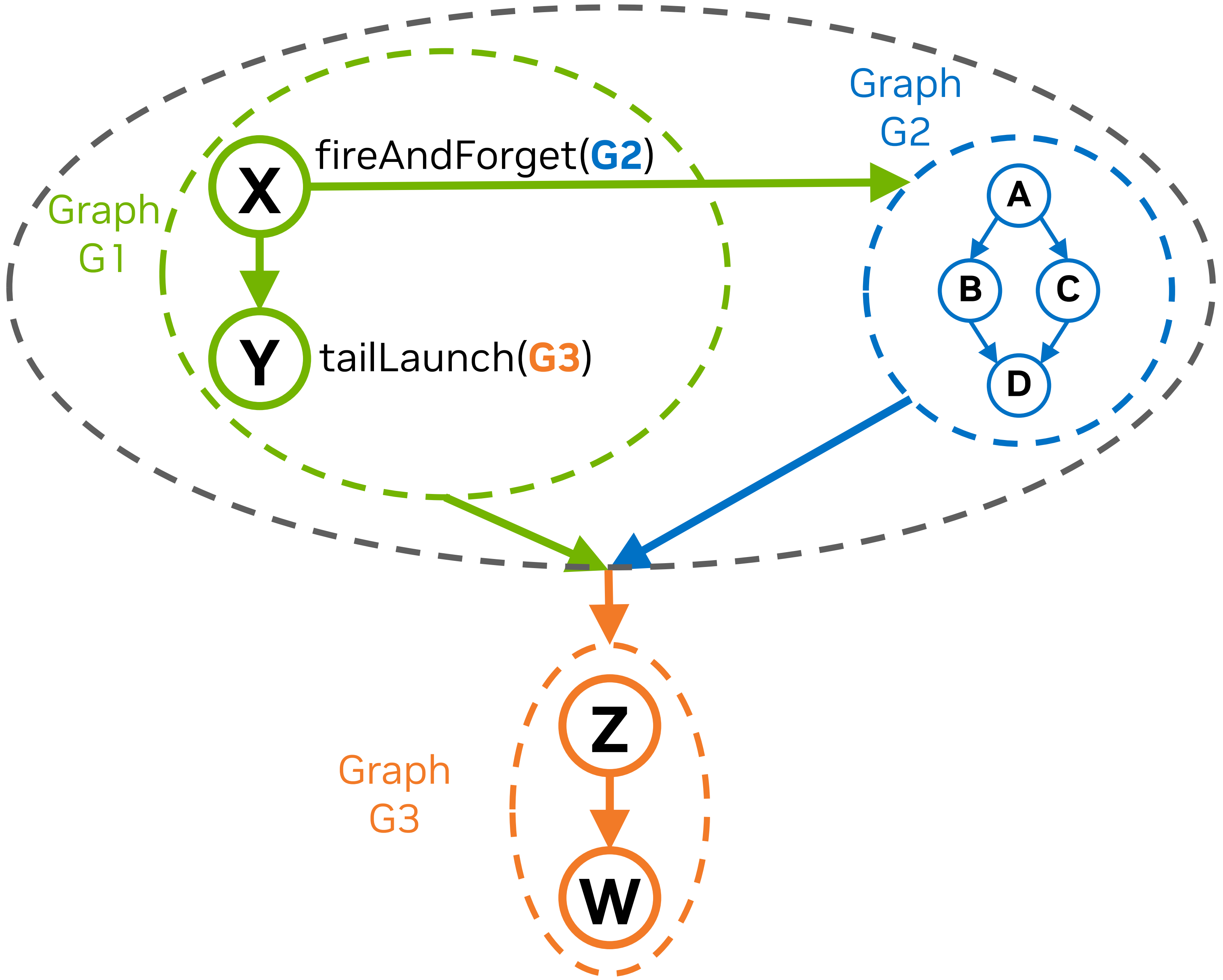
Provides a way to insert work dependencies

TAIL LAUNCH ENCAPSULATION

Joining Fire-and-Forget Launches

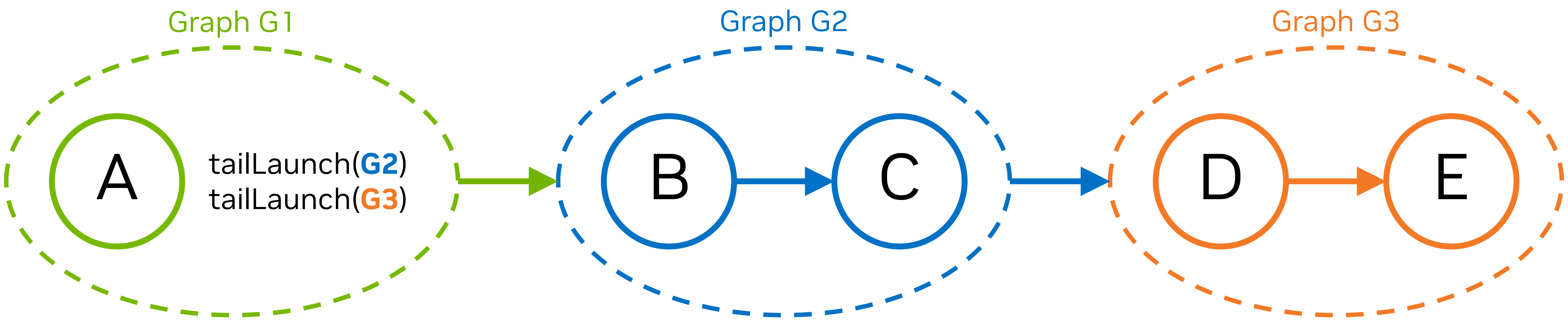
Fire-and-forget launches cannot be synced
- I.e, via cudaDeviceSynchronize()

Can only enforce ordering via tail launch
- Tail launch joins fire-and-forget work



TAIL LAUNCH ORDERING

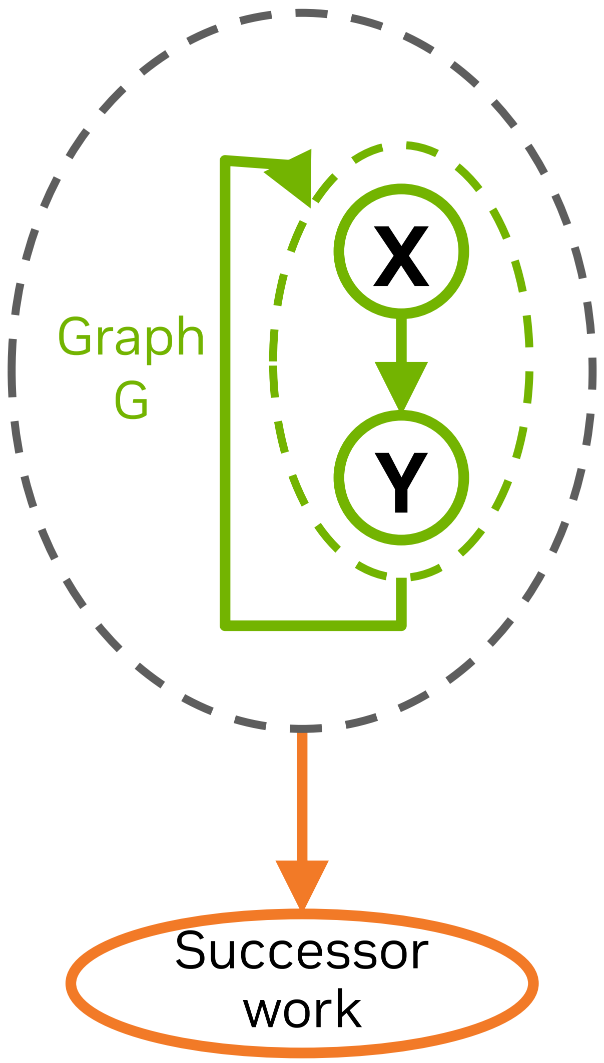
The Tail Launch Queue



Tail launches execute in the order in which they are enqueued

SELF RELAUNCH IS PERMITTED

Enable Host-Independent Loops In Your Applications



device_launch.cu

```
void main() {
    cudaGraphCreate(&G);
    // Build graph G
    cudaGraphInstantiate(G, DeviceLaunch);
    cudaGraphLaunch(G, ...);
}
```

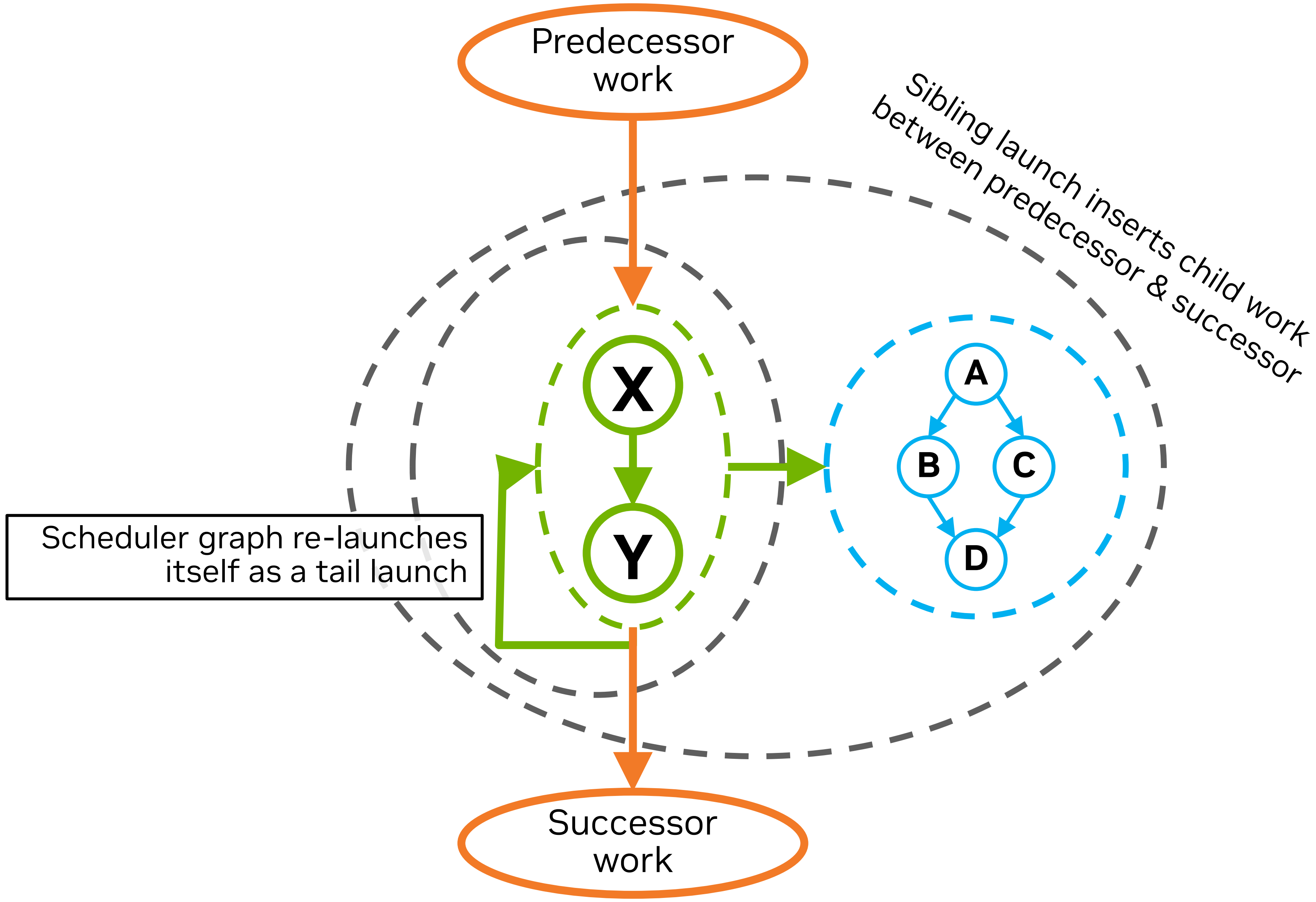
CPU

```
__global__ void Y() {
    if (condition) {
        G = cudaGetCurrentGraphExec();
        cudaGraphLaunch(G, tailLaunch);
    }
}
```

GPU

COMING SOON: SIBLING LAUNCH

Overcomes parent-graph encapsulation, transferring dependency to layer above



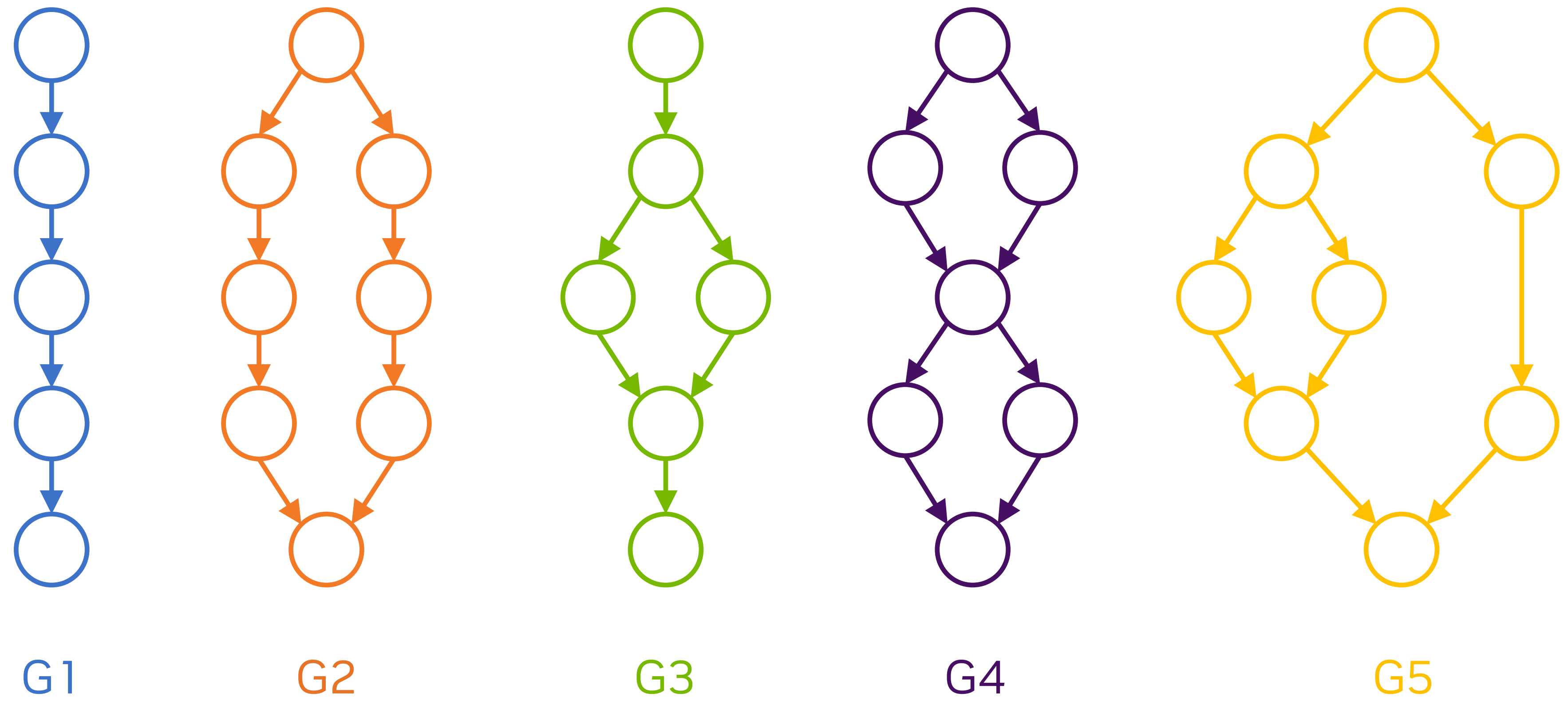
Sibling

Child work is launched concurrently with parent

Child work becomes a dependency of parent's parent but does not block re-launch of scheduler graph

SAMPLE USAGE

Run-Time Dynamic Work Scheduling

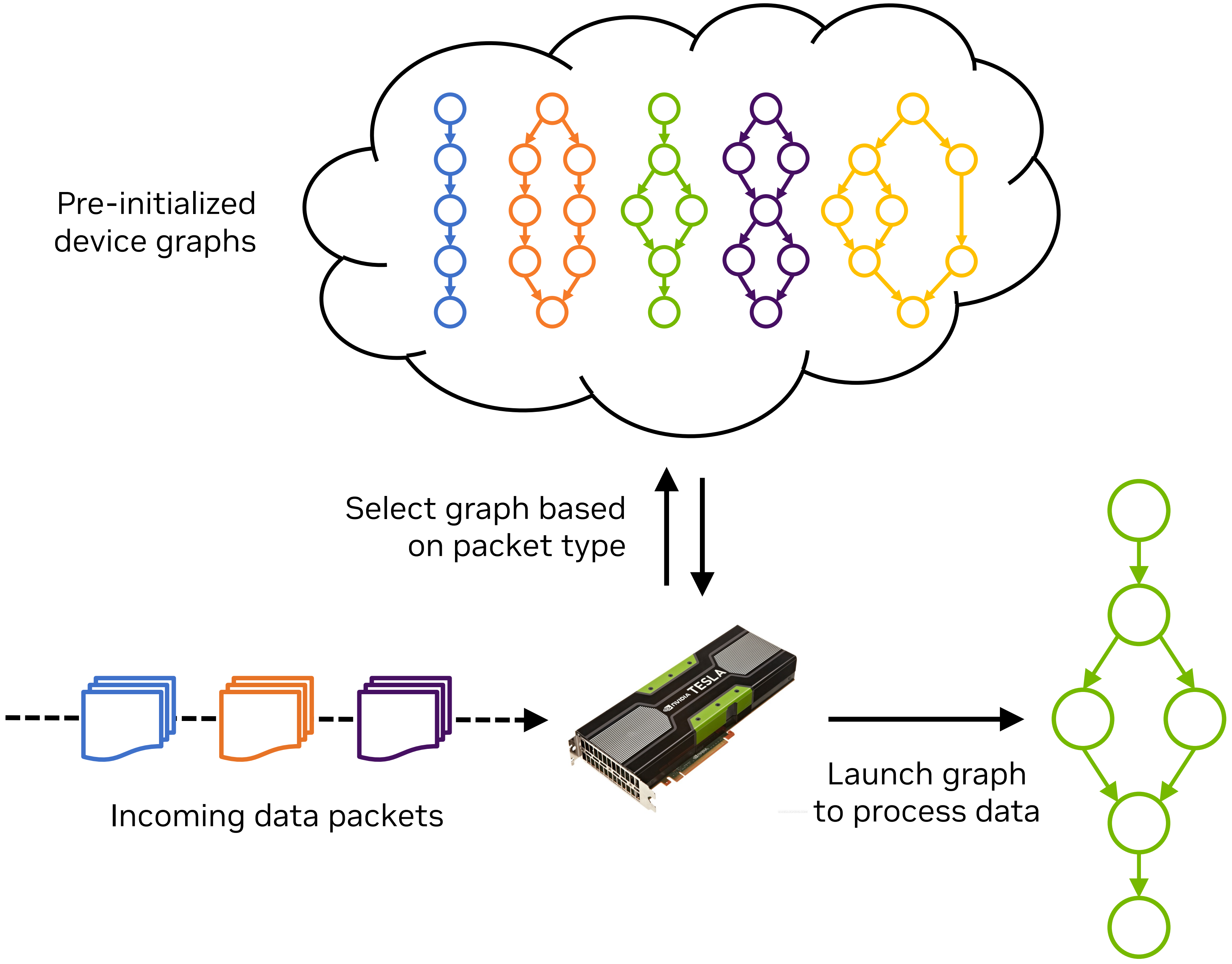



```
void init() {
    cudaGraphCreate(G1);
    ...                    // Set up graph G1

    cudaGraphCreate(G2);
    ...                    // Set up graph G2

    cudaGraphCreate(G3);
    ...                    // Set up graph G3

    cudaGraphCreate(G4);
    ...                    // Set up graph G4

    cudaGraphCreate(G5);
    ...                    // Set up graph G5

}
```

Create multiple graphs in host code
during program init

SAMPLE USAGE

Run-Time Dynamic Work Scheduling




```
__global__ void scheduler(...) {
    Packet data = receivePacket(...);

    switch(data.type) {
        case 1:
            cudaGraphLaunch(G1, fireAndForget);
            break;
        case 2:
            cudaGraphLaunch(G2, fireAndForget);
            break;
        case 3:
            cudaGraphLaunch(G3, fireAndForget);
            break;
        case 4:
            cudaGraphLaunch(G4, fireAndForget);
            break;
        case 5:
            cudaGraphLaunch(G5, fireAndForget);
            break;
    }

    // Re-launch the scheduler to run after processing
    currentGraphExec = cudaGetCurrentGraphExec();
    cudaGraphLaunch(currentGraphExec, tailLaunch);
}
```

Scheduler kernel executing on device

ADDITIONAL INFO

Get Started With Graphs

Read the CUDA graphs section of the programming guide

Check out the CUDA Samples:

- simpleCudaGraphs
- jacobiCudaGraphs
- graphMemoryNodes
- graphMemoryFootprint

Developer Blogs

- Getting Started With CUDA Graphs
- Employing CUDA Graphs in a Dynamic Environment
- Enabling Dynamic Control Flow With Device Graph Launch

GTC Talk

- Effortless CUDA Graphs

NVIDIA