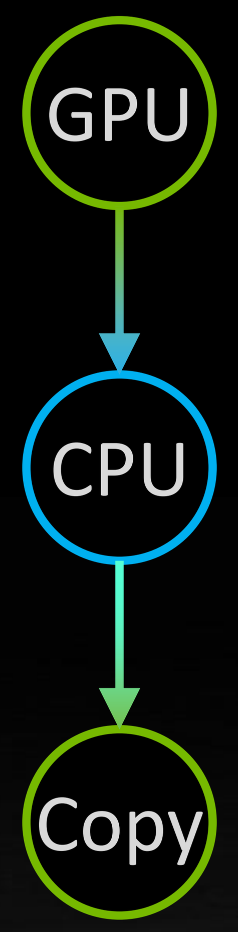
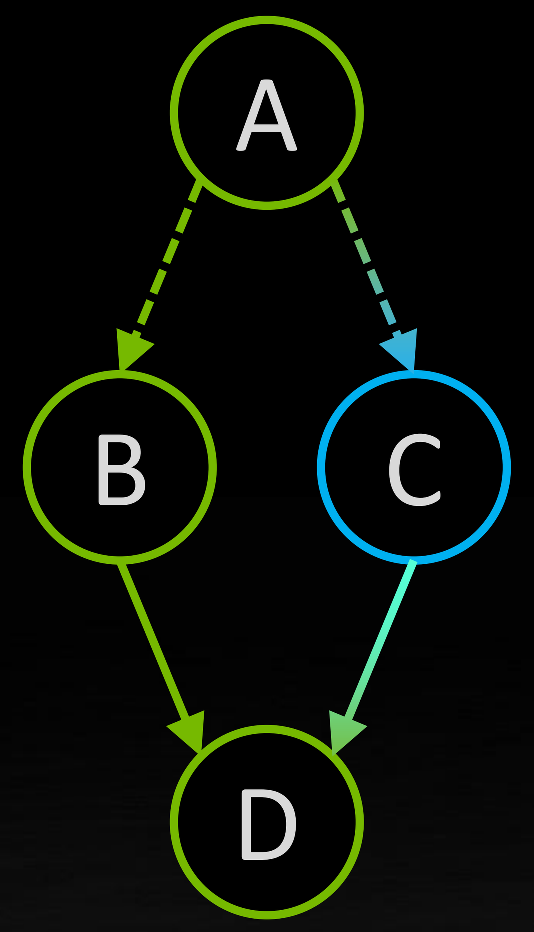
CUDA GRAPHS DYNAMIC CONTROL FLOW, SEPTEMBER 2023

STEPHEN JONES, NVIDIA

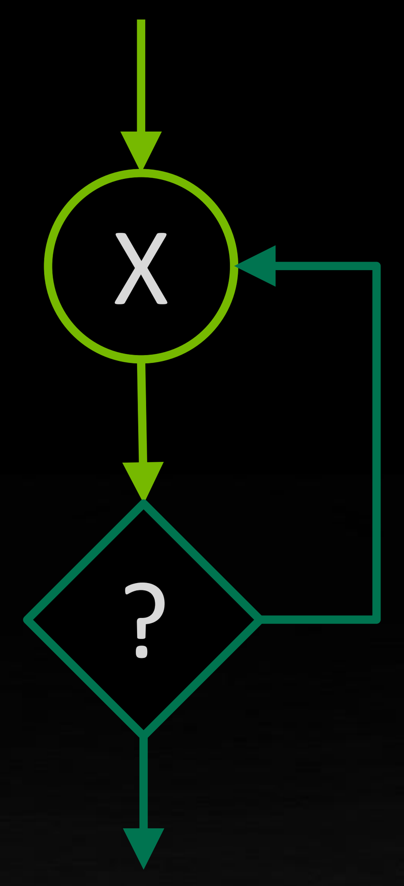
DYNAMIC CONTROL FLOW IN GRAPHS



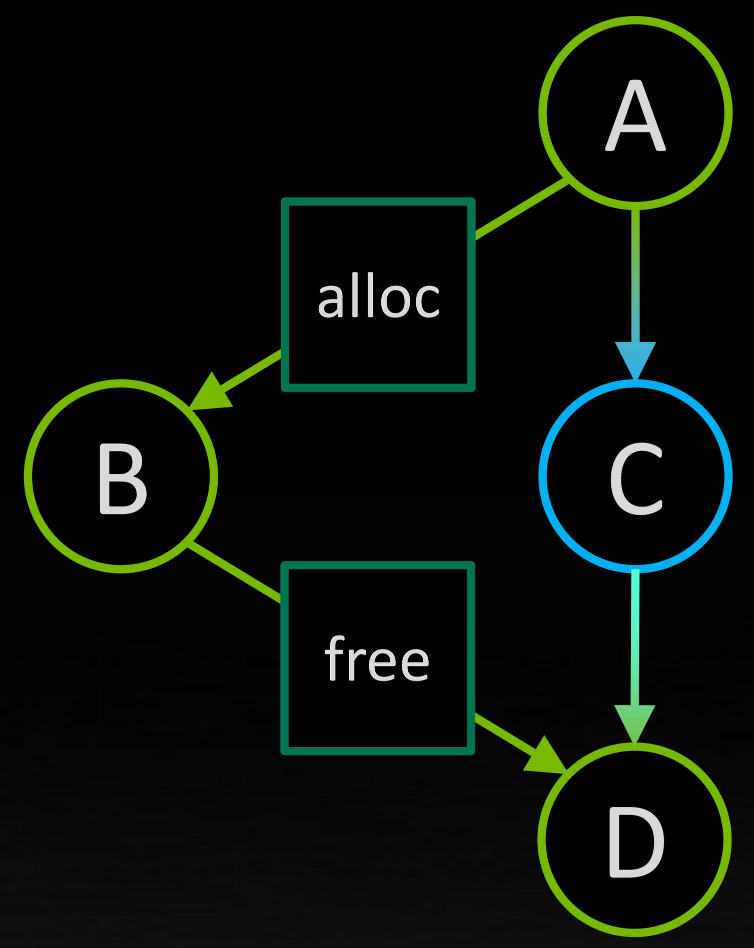
Heterogeneous Execution



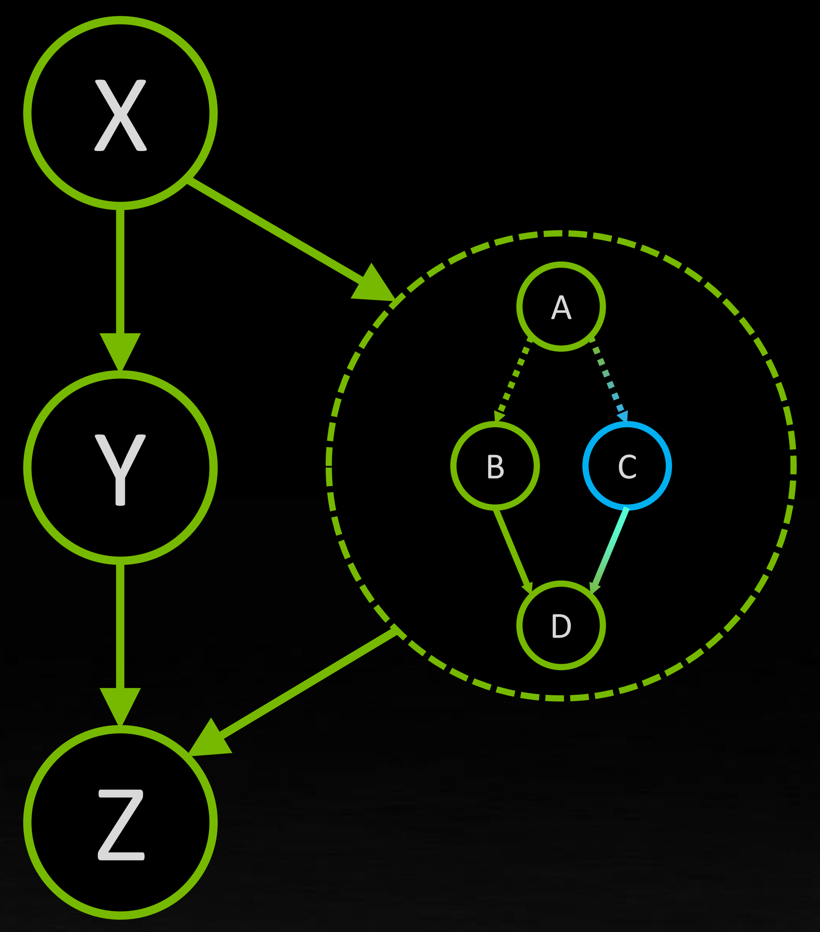
Dynamic Control Flow



Iterative Loops



Inline Memory Allocation



In-Kernel Graph Launch

GRAPH LAUNCH FROM A GPU KERNEL

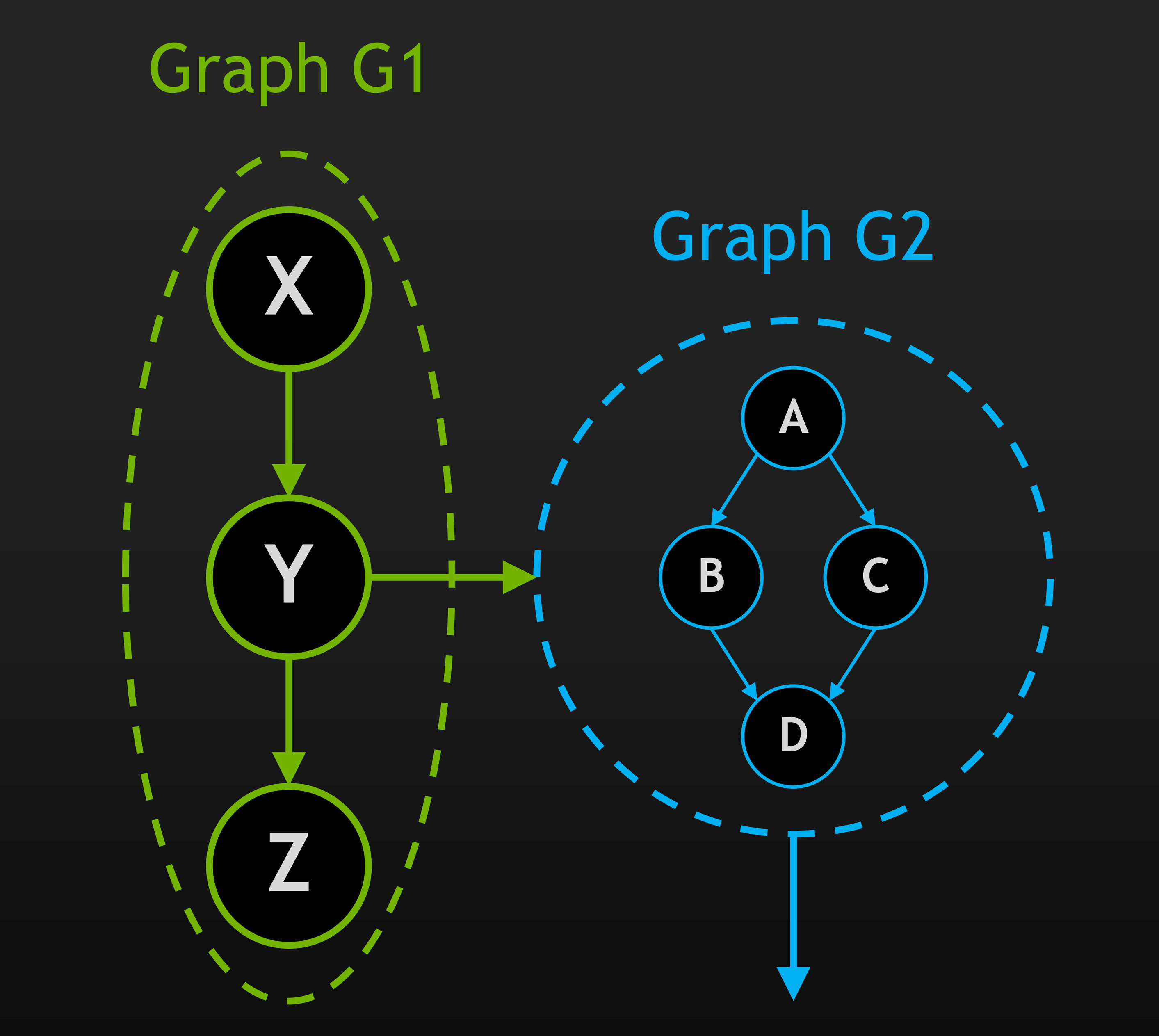
cdp_graphs.cu

CPU portion

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

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

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

GPU portion

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



Device-side graph launch

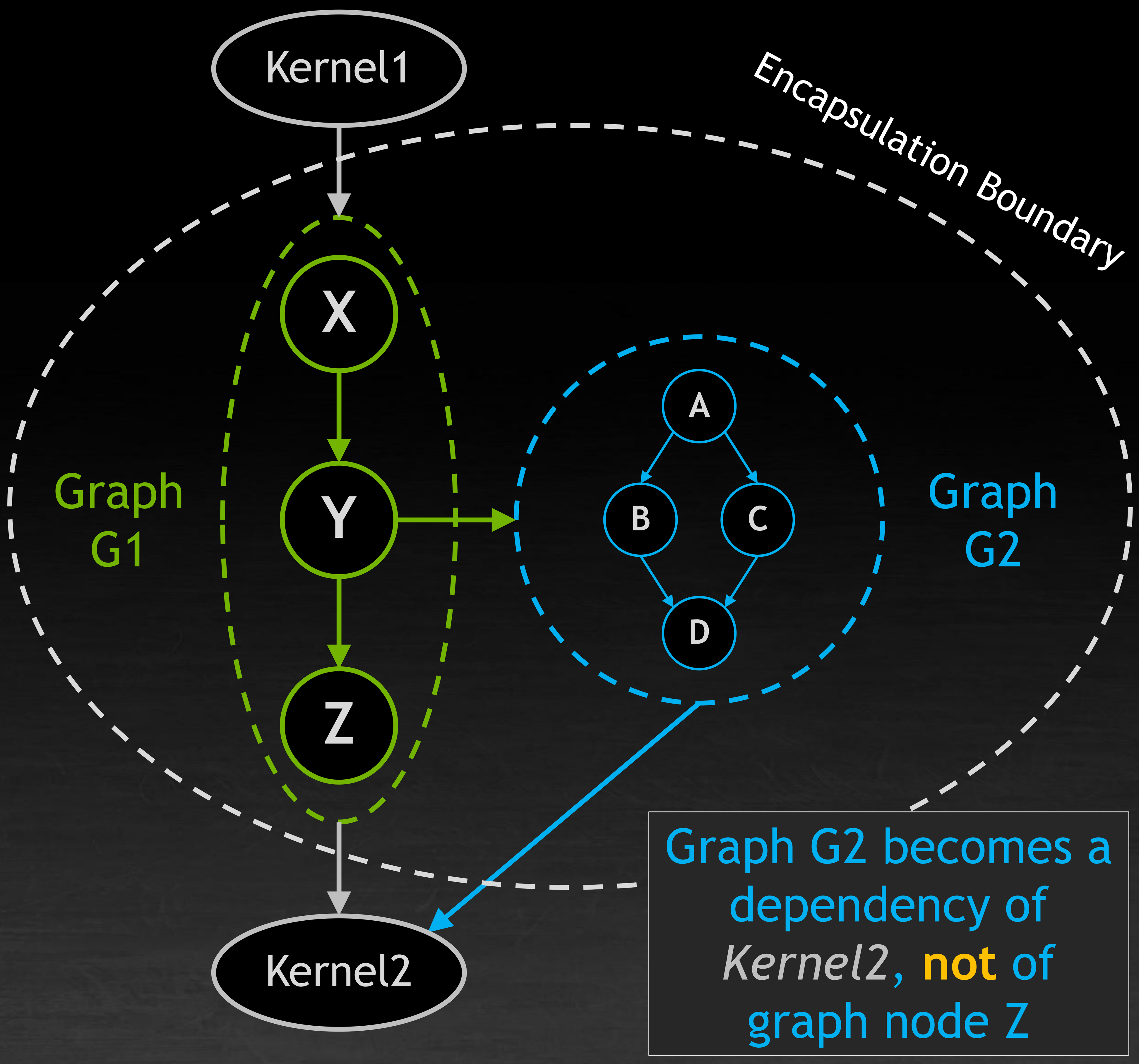
NVIDIA

ENCAPSULATION FOR DEVICE-SIDE GRAPH LAUNCH

Parent graphs are monolithic with respect to dependency resolution

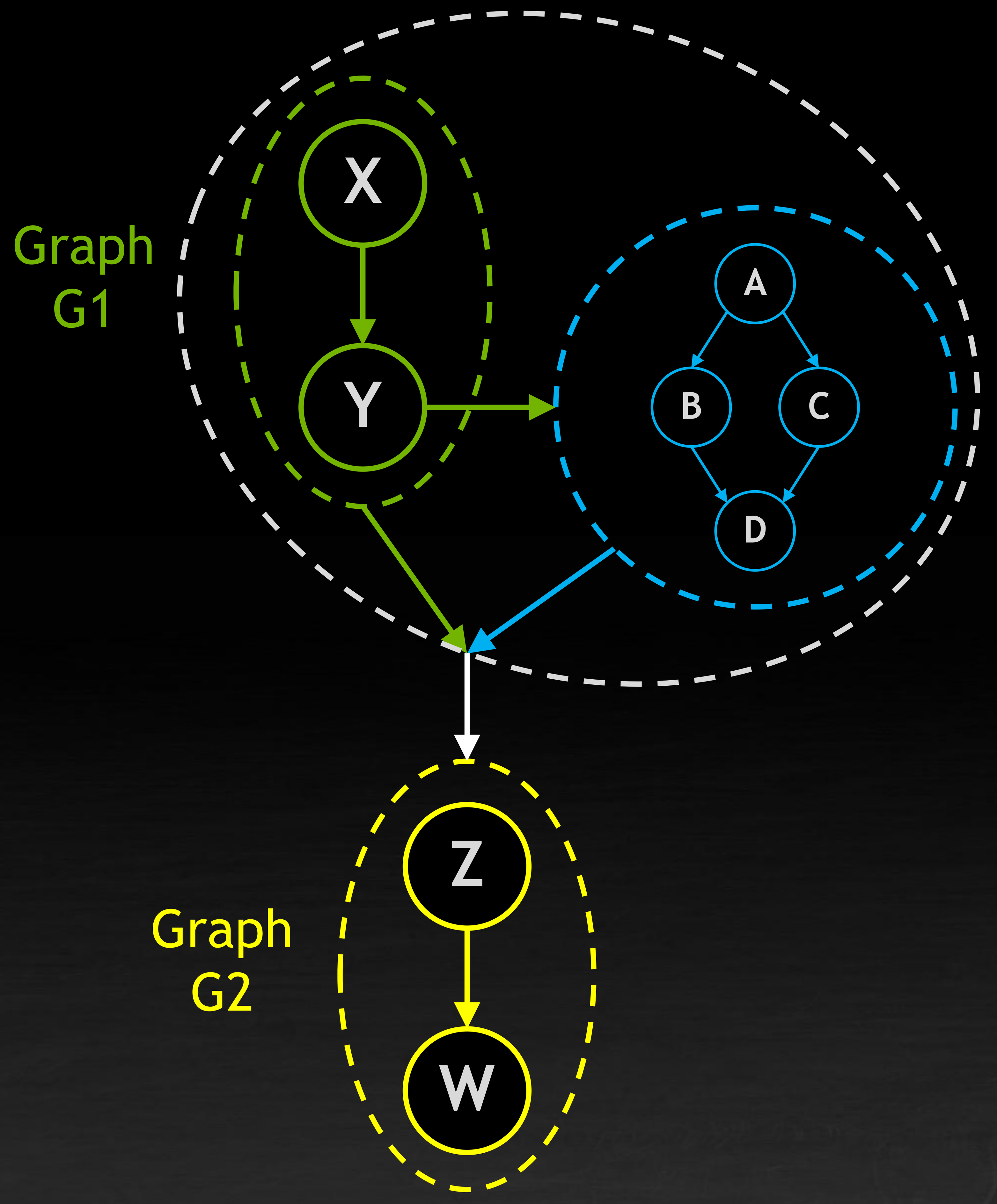
Graph encapsulation boundary is **the whole launching graph**

Graph launch cannot create a new dependency within the parent graph (i.e. no fork/join parallelism inside a graph)



DEVICE GRAPH LAUNCH NAMED STREAMS

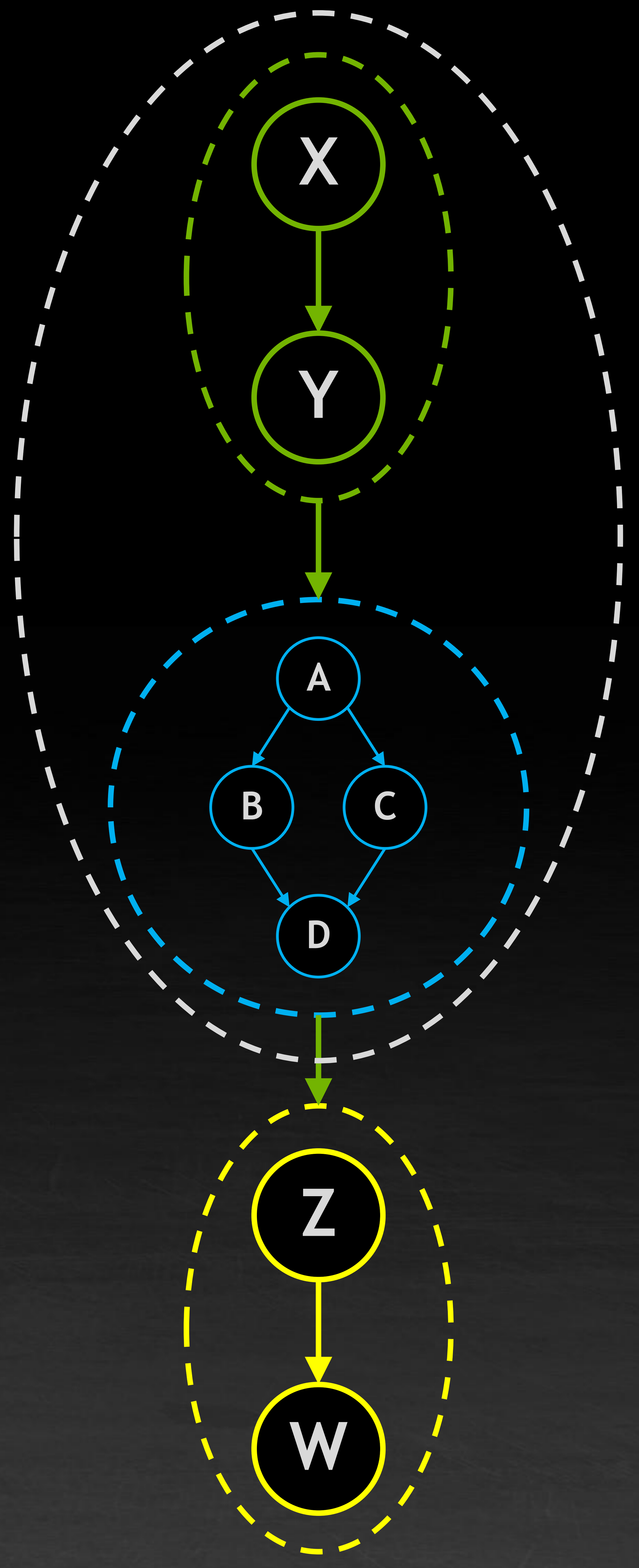
Identical semantics to dynamic parallelism single-kernel launch named streams, but at whole-graph granularity



Fire-and-Forget

Child work is launched concurrently with parent

Graph G2 now depends on G1 and child work



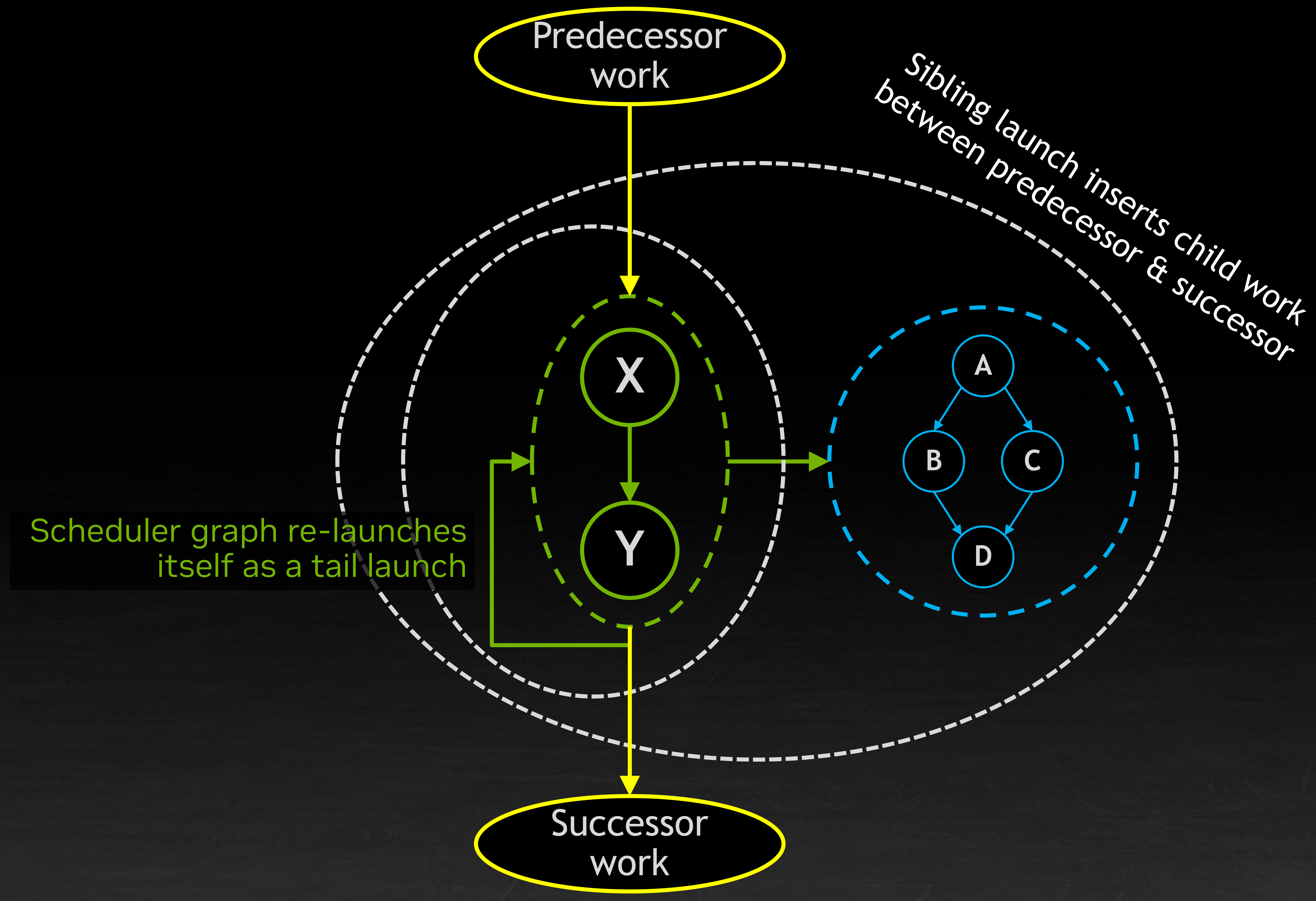
Tail Launch

Child work is launched sequentially after parent

Graph G2 now depends on child work (which in turn depends on parent)

UPCOMING NEW LAUNCH TYPE: “SIBLING” LAUNCH

Breaks parent-graph encapsulation boundary, creating dependency on layer above

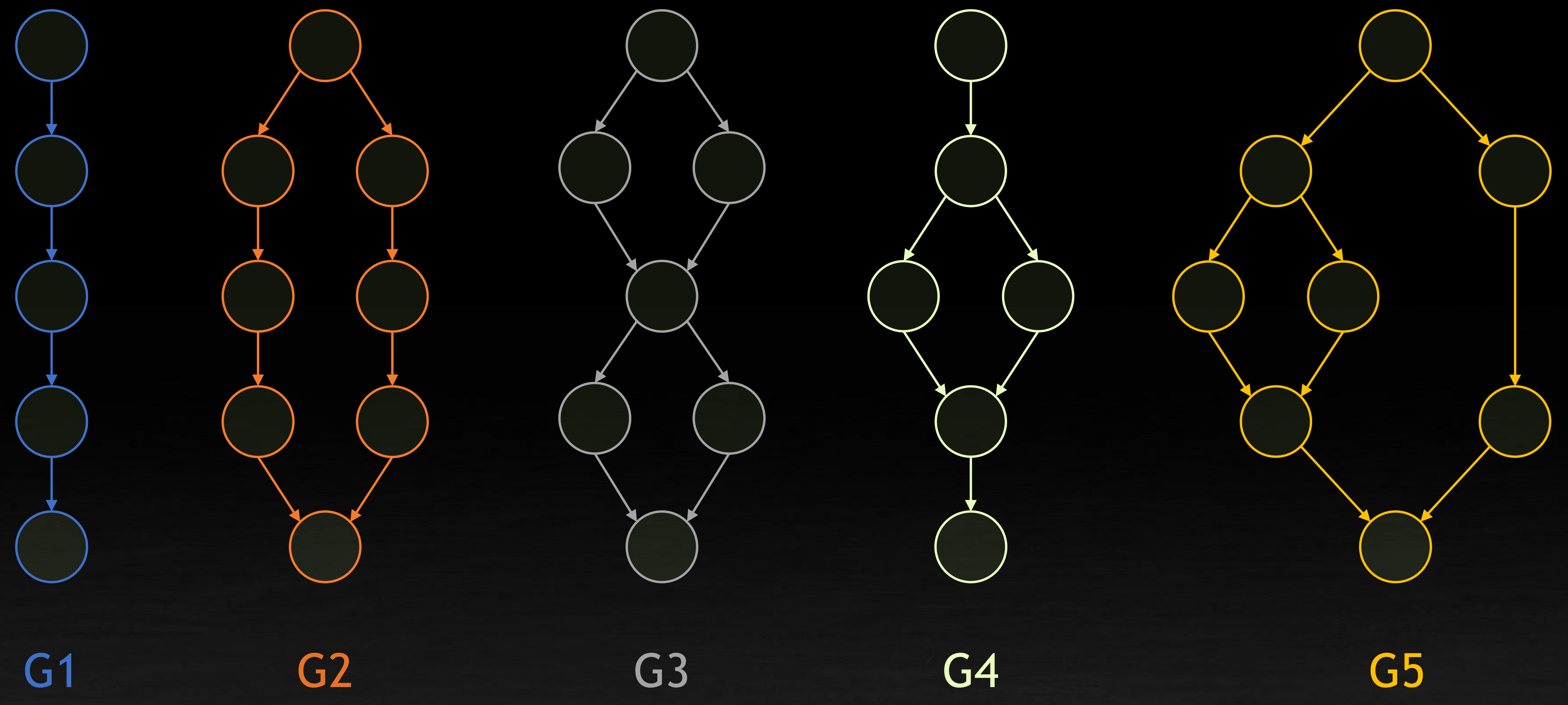


Sibling

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

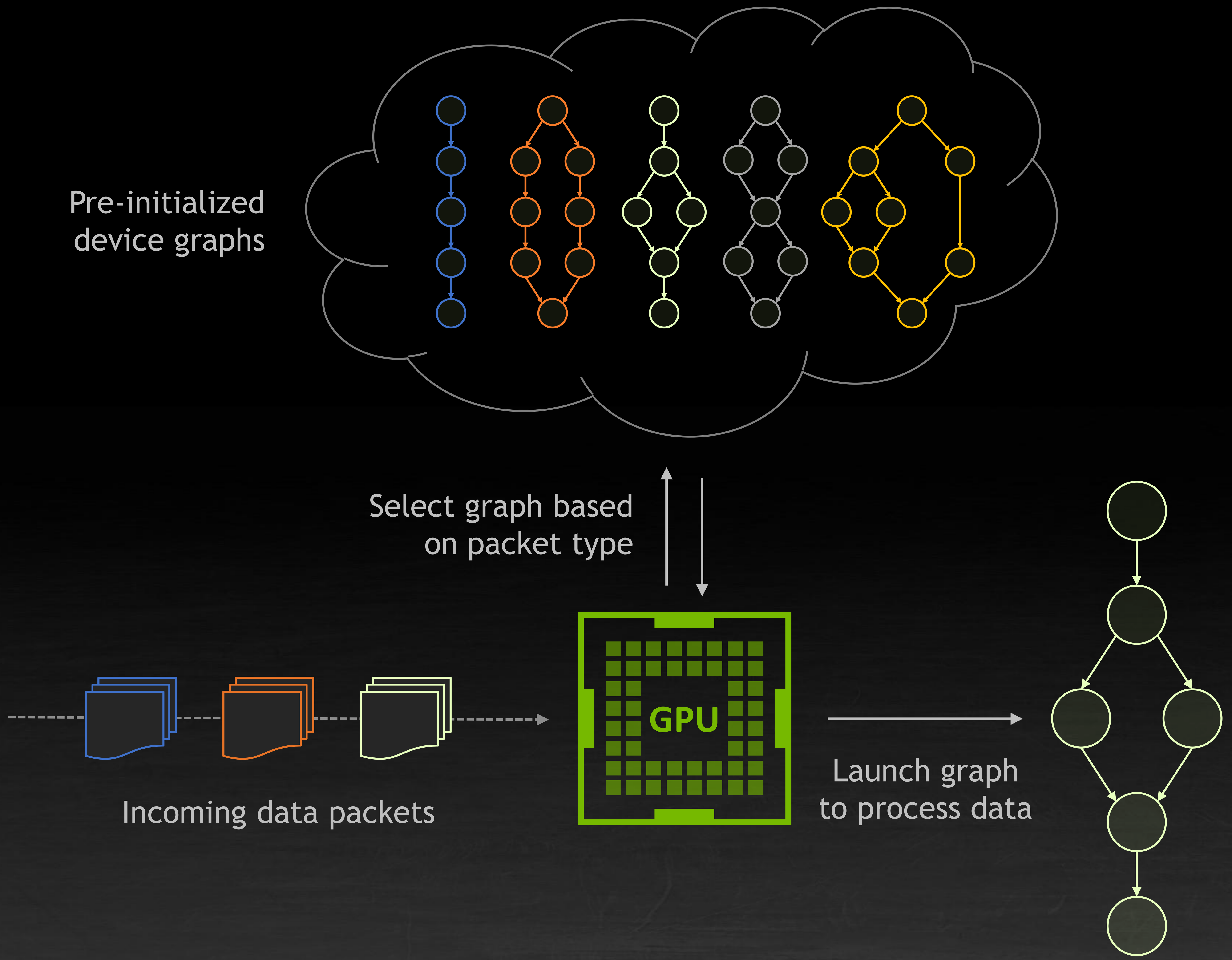
EXAMPLE: RUN-TIME DYNAMIC WORK SCHEDULING




```
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

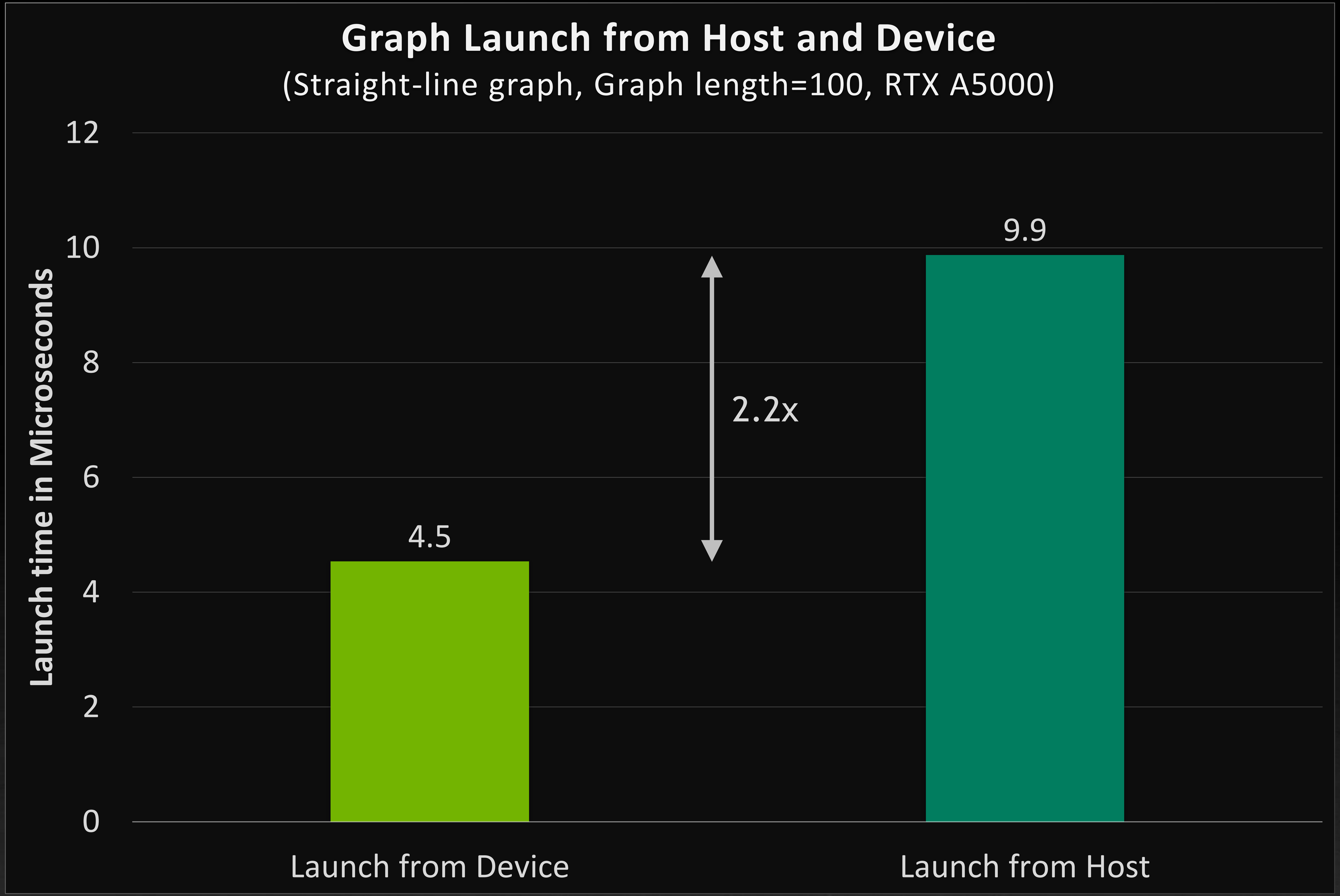
EXAMPLE: RUN-TIME DYNAMIC WORK SCHEDULING




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

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

    // Re-launch the scheduler to run after processing
    cudaGraphLaunch(scheduler, TailLaunch, ...);
}
```

Scheduler kernel executing on device

THE DEVICE-LAUNCH ADVANTAGE




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

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

    // Re-launch the scheduler to run after processing
    cudaGraphLaunch(scheduler, TailLaunch, ...);
}
```

Scheduler kernel executing on device

COMING SOON FOR DEVICE-GRAPH LAUNCH

1. Update of node parameters from a GPU kernel
   - This is known to be critical for graph re-use, and is assumed to be important for iterating from within a kernel as well

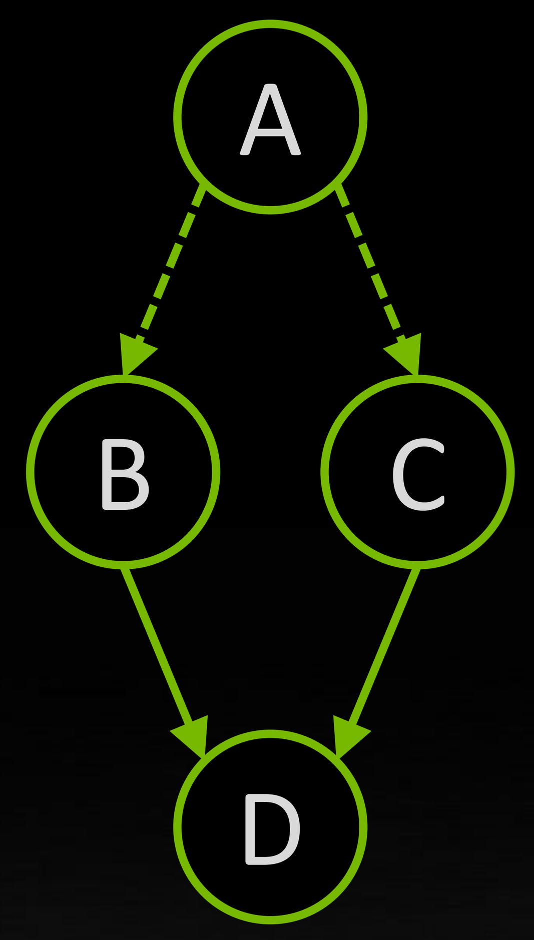
2. Various performance optimisations, especially related to CPU cost of launch

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

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

    // Re-launch the scheduler to run after processing
    cudaGraphLaunch(scheduler, TailLaunch, ...);
}
```

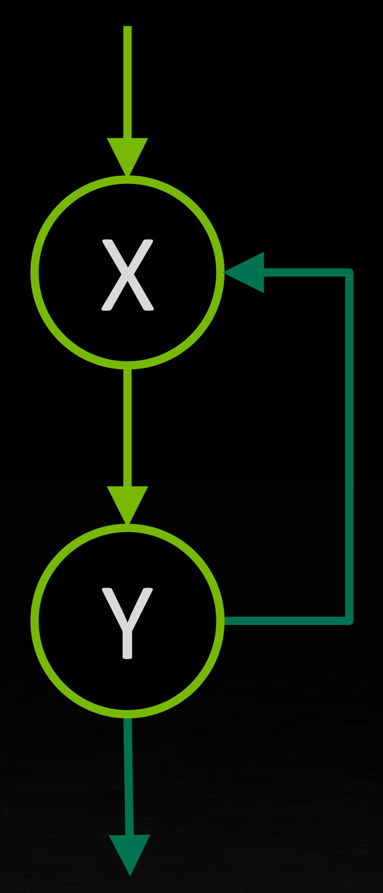
CONDITIONAL CONTROL FLOW WITHIN A GRAPH

Conditional nodes come in two flavours:

- **IF** condition for single-pass evaluation/activation
- **WHILE** loops which execute the subgraph repeatedly



Runtime Graph
Node Activation

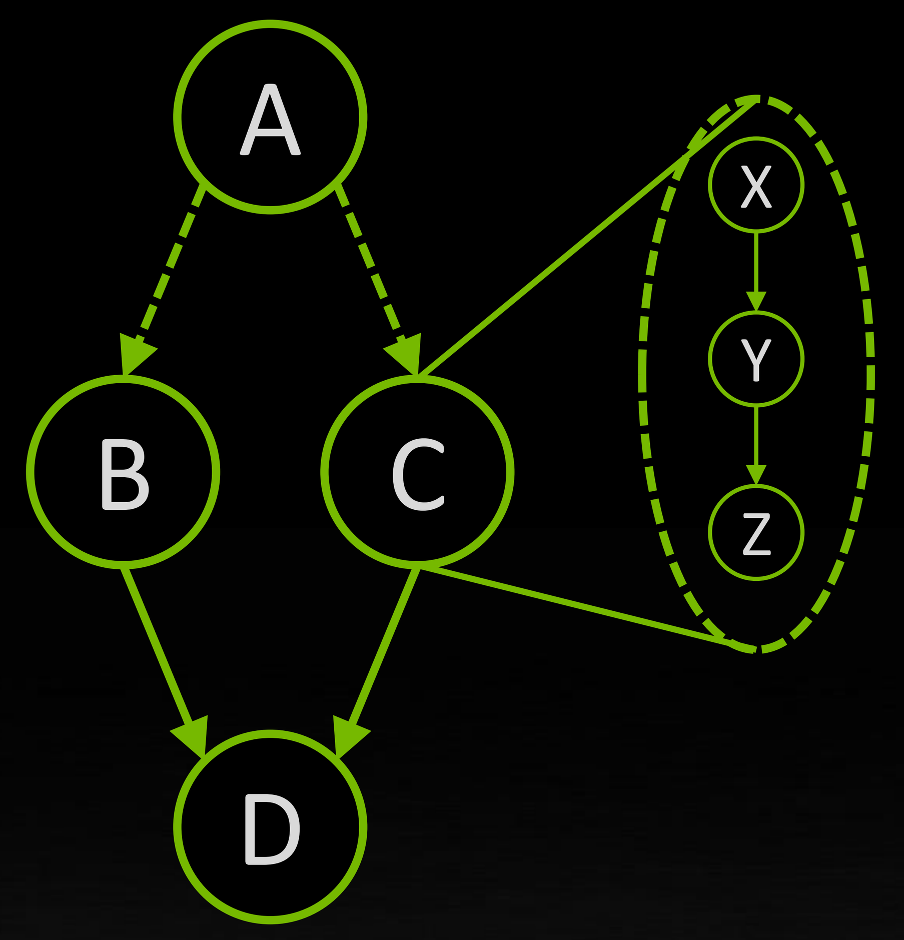


Iterative
“While” Loops

CONDITIONAL CONTROL FLOW WITHIN A GRAPH

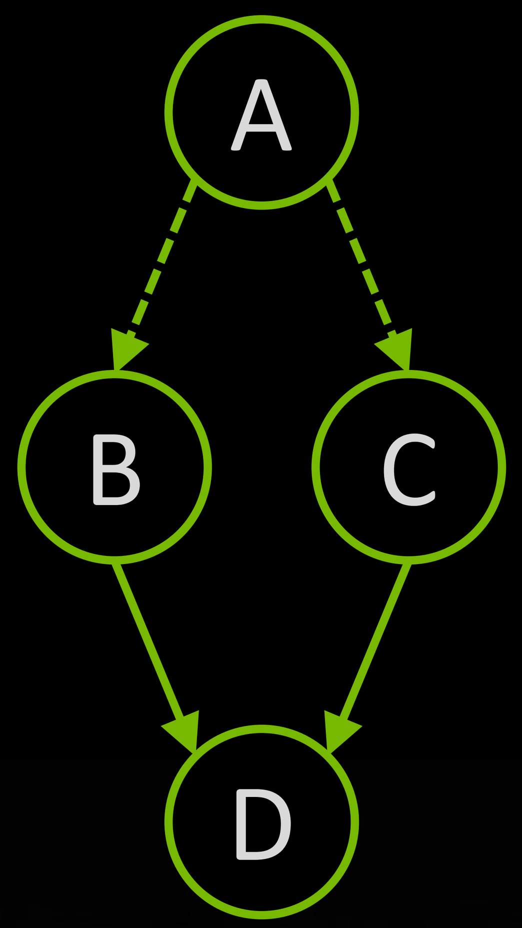
Graph conditionals operate by embedding sub-graphs within the main graph, as “conditional nodes”

1. Start by creating a conditional graph node – creation returns a handle to an empty graph
2. Populate this graph either explicitly or via cudaStreamCaptureToGraph()
3. Conditional nodes/subgraphs **MAY** be nested
4. The boolean activation value for a conditional node is associated with a conditional handle, which allows setting of this value by a GPU kernel
   - Pass the handle to an upstream kernel to allow that kernel to set the value and thus determine if the conditional node executes or not



Runtime Graph
Node Activation

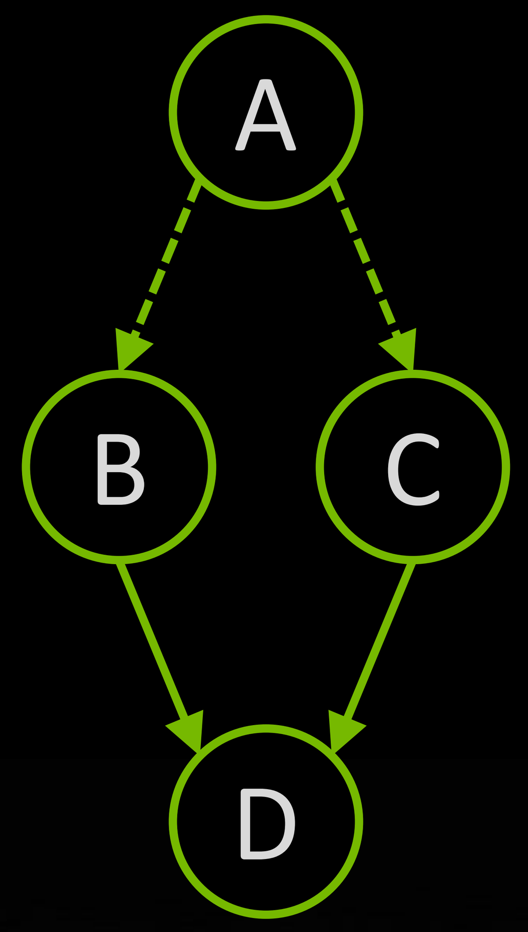
“IF” CONDITIONAL EXAMPLE




```
__global__ void A(cudaGraphConditionalHandle handleB,
                  cudaGraphConditionalHandle handleC) {
    ...
    // “value” here is true/false.
    // When true, the conditional node will run
    cudaGraphSetConditional(handleB, valueB);
    cudaGraphSetConditional(handleC, valueC);
    ...
}
```

```
void init() {
    cudaGraphConditionalHandle handle;
    cudaGraphConditionalHandleCreate(&handle, graph);

     // Use a kernel upstream of the conditional to set the handle value
     cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
     params.kernel.func = (void *)setHandle;
     params.kernel.gridDim.x = 1;
     params.kernel.blockDim.x = 1;
     params.kernel.kernelParams = kernelArgs;
     kernelArgs[0] = &handle;
     cudaGraphAddNode(&node, graph, NULL, 0, &params);

     cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
     cParams.conditional.handle = handle;
     cParams.conditional.type   = cudaGraphCondTypeIf;
     cParams.conditional.size   = 1;
     cudaGraphAddNode(&node, graph, &node, 1, &cParams);

     cudaGraph_t bodyGraph = cParams.conditional.phGraph_out[0];

     // Populate the body of the conditional node
     ...
     cudaGraphAddNode(&node, bodyGraph, NULL, 0, &params);

     cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
     cudaGraphLaunch(graphExec, 0);
}
```

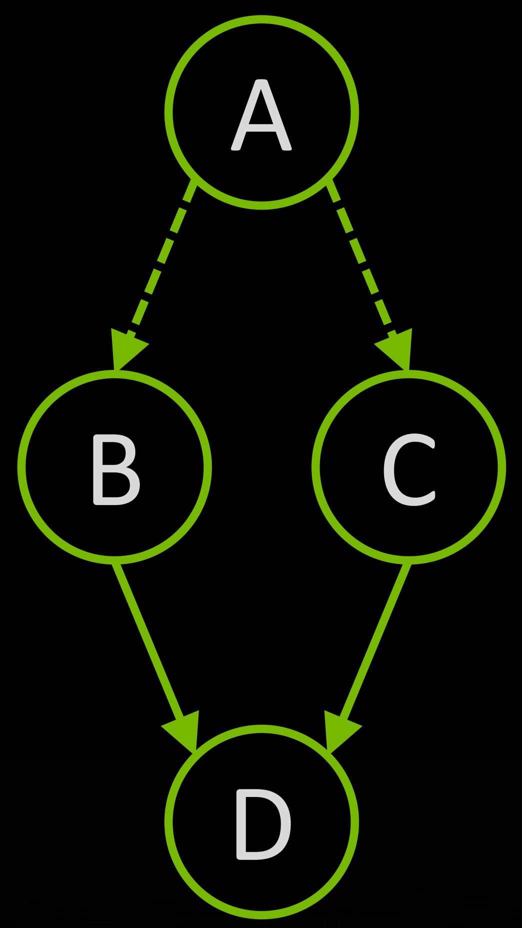
"IF" CONDITIONAL EXAMPLE




```
__global__ void A(cudaGraphConditionalHandle handleB,
                  cudaGraphConditionalHandle handleC) {
    ...
    // "value" here is true/false.
    // When true, the conditional node will run
    cudaGraphSetConditional(handleB, valueB);
    cudaGraphSetConditional(handleC, valueC);
    ...
}
```

```
void init() {
    cudaGraphConditionalHandle handle;
    cudaGraphConditionalHandleCreate(&handle, graph);

    // Use a kernel upstream of the conditional to set the handle value
    cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
    params.kernel.func = (void *)setHandle;
    params.kernel.gridDim.x = 1;
    params.kernel.blockDim.x = 1;
    params.kernel.kernelParams = kernelArgs;
    kernelArgs[0] = &handle;
    cudaGraphAddNode(&node, graph, NULL, 0, &params);

    cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
    cParams.conditional.handle = handle;
    cParams.conditional.type   = cudaGraphCondTypeIf;
    cParams.conditional.size   = 1;
    cudaGraphAddNode(&node, graph, &node, 1, &cParams);

    cudaGraph_t bodyGraph = cParams.conditional.phGraph_out[0];

    // Populate the body of the conditional node
    ...
    cudaGraphAddNode(&node, bodyGraph, NULL, 0, &params);

    cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    cudaGraphLaunch(graphExec, 0);
}
```

"IF" CONDITIONAL EXAMPLE




```
__global__ void A(cudaGraphConditionalHandle handleB,
                  cudaGraphConditionalHandle handleC) {
    ...
    // "value" here is true/false.
    // When true, the conditional node will run
    cudaGraphSetConditional(handleB, valueB);
    cudaGraphSetConditional(handleC, valueC);
    ...
}
```

```
void init() {
   cudaGraphConditionalHandle handle;
   cudaGraphConditionalHandleCreate(&handle, graph);

    // Use a kernel upstream of the conditional to set the handle value
    cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
    params.kernel.func = (void *)setHandle;
    params.kernel.gridDim.x = 1;
    params.kernel.blockDim.x = 1;
    params.kernel.kernelParams = kernelArgs;
    kernelArgs[0] = &handle;
    cudaGraphAddNode(&node, graph, NULL, 0, &params);

    cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
    cParams.conditional.handle = handle;
    cParams.conditional.type   = cudaGraphCondTypeIf;
    cParams.conditional.size   = 1;
    cudaGraphAddNode(&node, graph, &node, 1, &cParams);

    cudaGraph_t bodyGraph = cParams.conditional.phGraph_out[0];

    // Populate the body of the conditional node
    ...
    cudaGraphAddNode(&node, bodyGraph, NULL, 0, &params);

    cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    cudaGraphLaunch(graphExec, 0);
}
```

"WHILE" CONDITIONAL EXAMPLE

CUDA Graphs are no longer a DAG

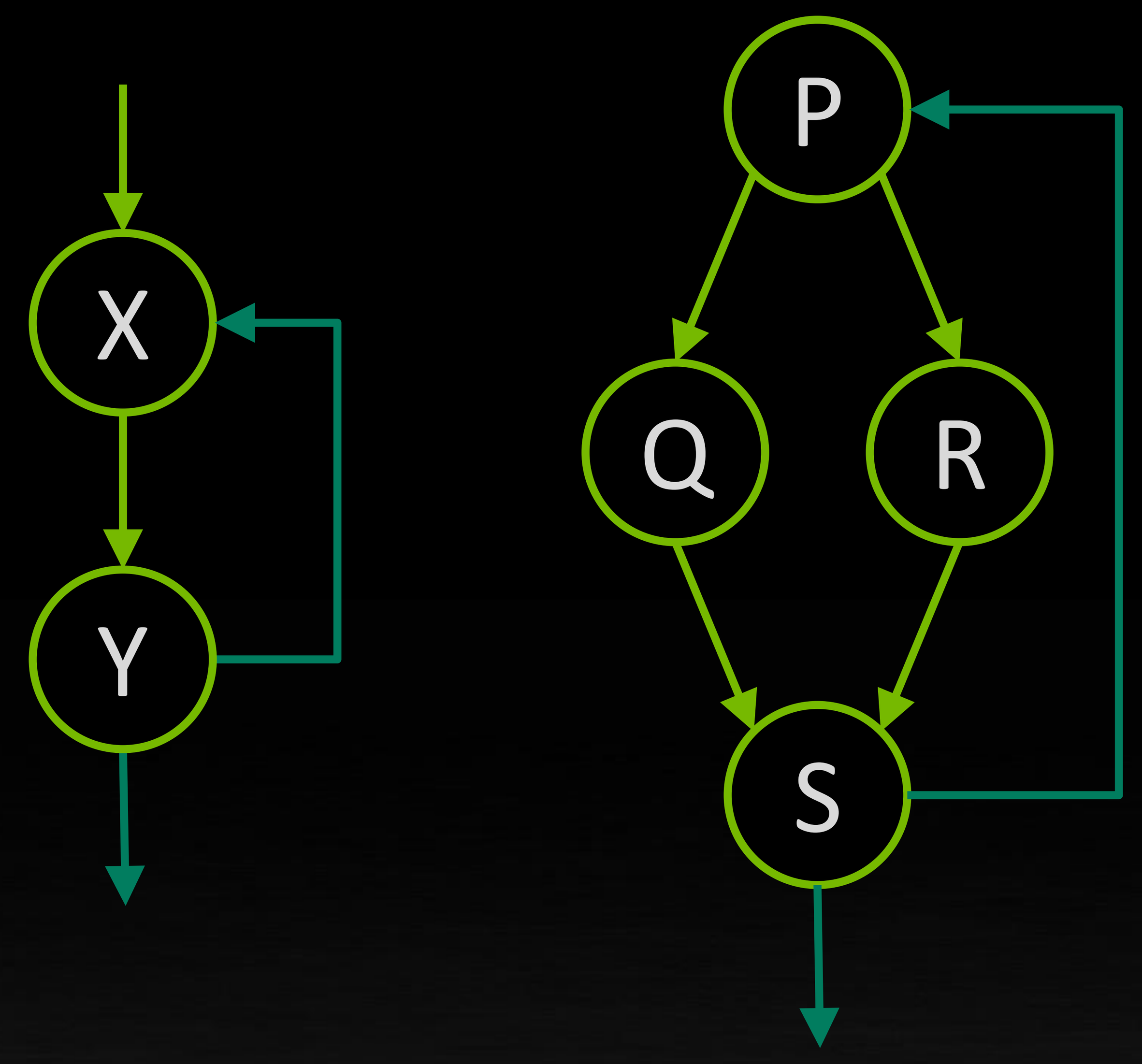
Conditional sub-graphs may not contain nodes which do not execute from within the GPU SM

Permitted

- All GPU kernels
- Memory copies between device memory, or to pinned host memory
- Memset on device memory or pinned host memory
- Child-graph nodes which satisfy these requirements
- Other conditional nodes

Not Permitted

- Memcpy operations to addresses not directly accessible by the GPU
- Memory allocation nodes
- Nodes which execute on a different GPU




```
__global__ void S(cudaGraphConditionalHandle handle, ...) {
    static int count = 10;
    cudaGraphSetConditional(handle, --count ? 1 : 0);
}
```


NVIDIA