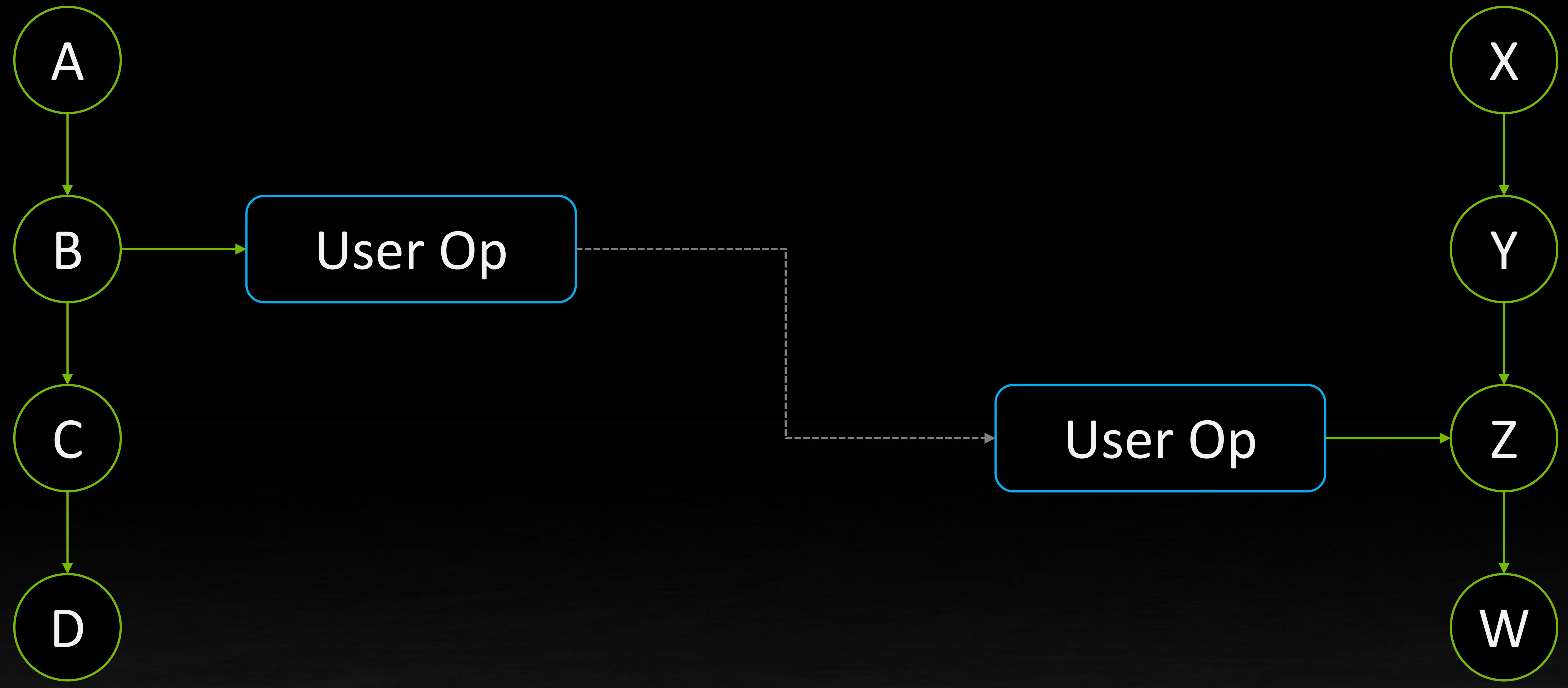
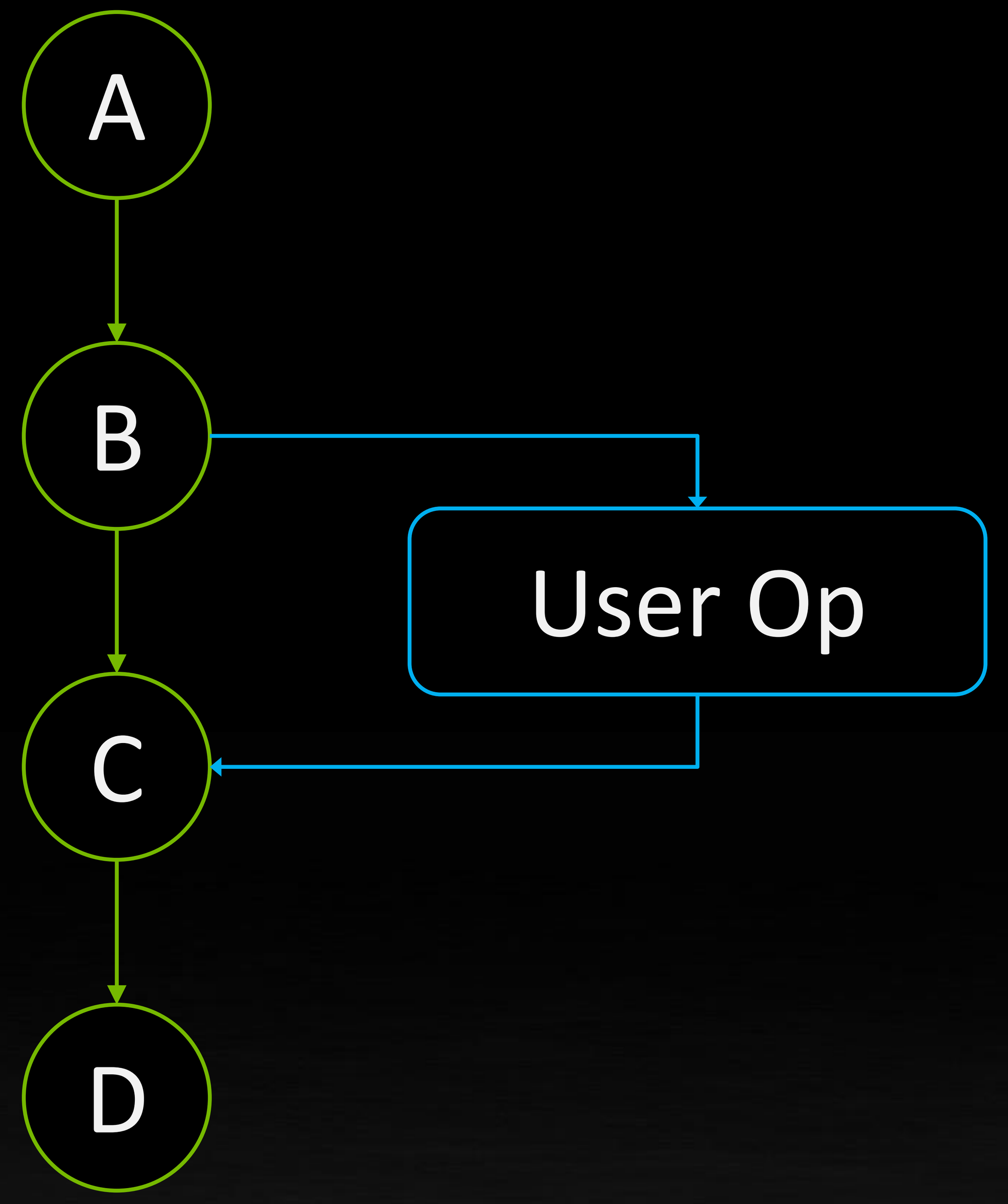
CUDA GRAPHS UPDATES, OCTOBER 2022

STEPHEN JONES, NVIDIA

EXTERNAL DEPENDENCIES VIA EVENTS & "MEMOPS" (CUDA 11.5 & 11.6)

Memops are: `cuStreamWaitValue()` & `cuStreamWriteValue()` - typically used for MPI & GPUDirect dependencies





Outgoing Dependency

Launch sequence
1. launch graph
2. launch user op

Incoming Dependency

Launch sequence
1. launch user op
2. launch graph

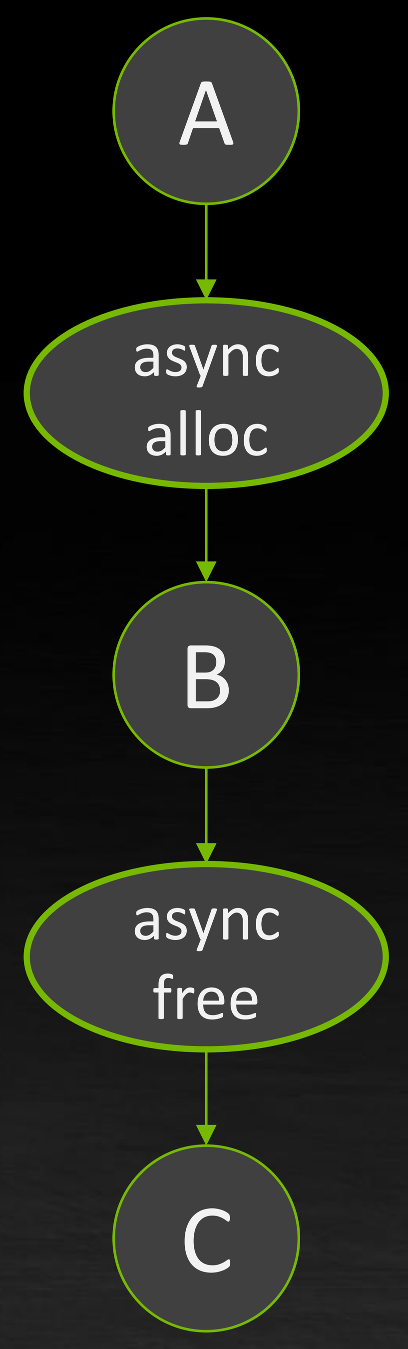
In/Out Dependencies

NOT PERMITTED
WITH CUDA GRAPHS

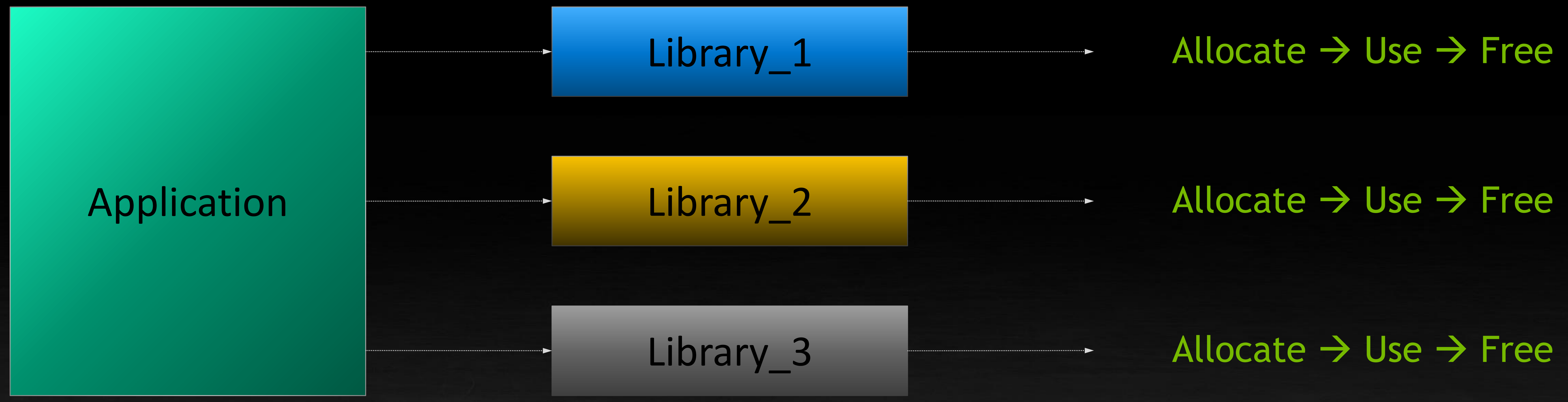
(must split graph at B-C)

STREAM ORDERED MEMORY ALLOCATION

```
void cuda_run() {
  a<<< ..., stream >>>();
  cudaMallocAsync(&b_ptr, N*3, stream);
  b<<< ..., stream >>>(b_ptr);
  cudaFreeAsync(b_ptr, stream);
  c<<< ..., stream >>>();
}
```



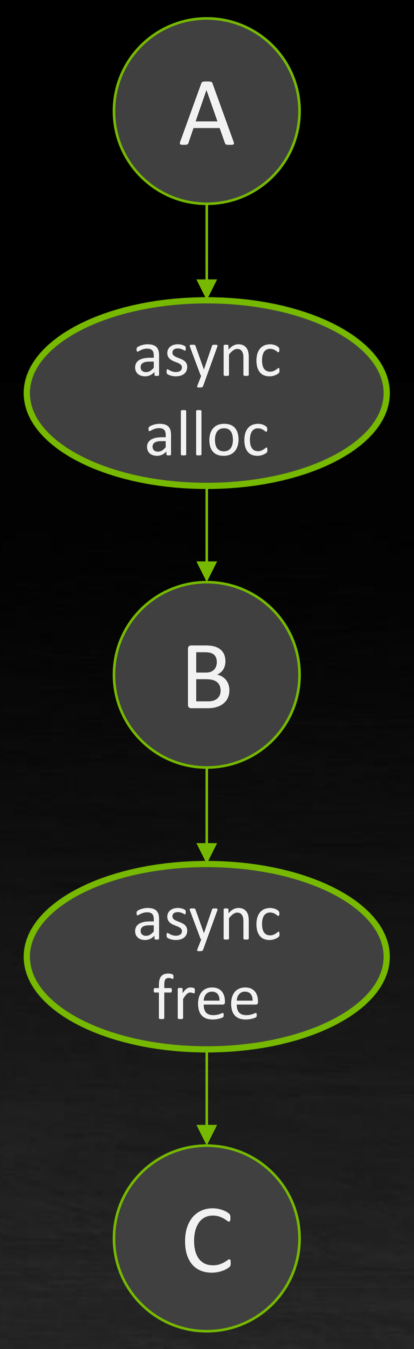
STREAM ORDERED MEMORY ALLOCATION



STREAM ORDERED MEMORY ALLOCATION

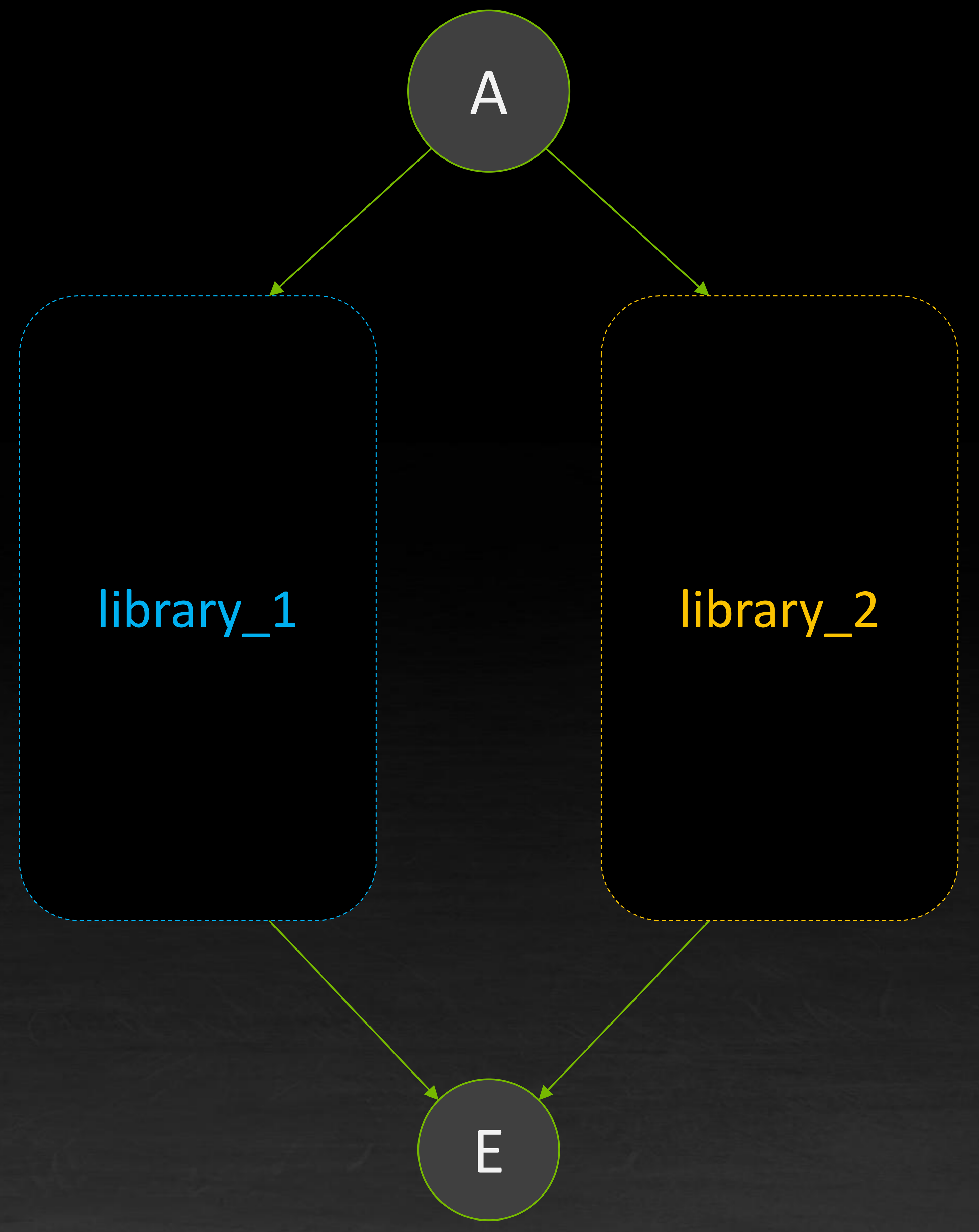
Why allocate memory asynchronously?

- Composable (allocate where you use it instead of globally)
- Faster allocation time
- Fine-grained sharing of resources
- Smaller memory footprint through re-use within a stream



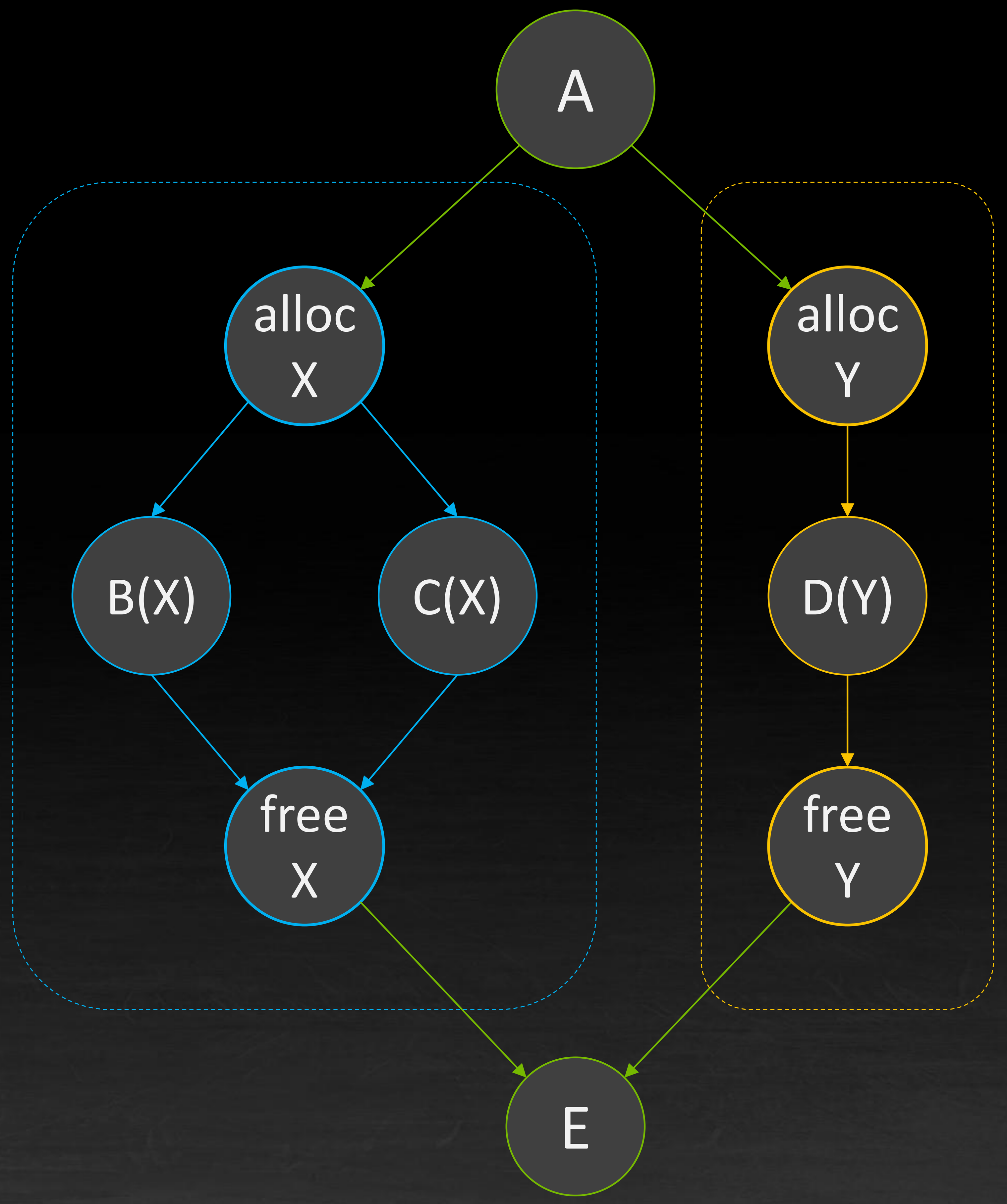
COMPOSABLE ALLOCATION

```
void application() {
  a<<< ..., s1 >>>();
  cudaEventRecord(e1, s1);
  cudaStreamWaitEvent(s2, e1);

  library_1(s1);
  library_2(s2);

  cudaEventRecord(e2, s2);
  cudaStreamWaitEvent(s1, e2);
  e<<< ..., stream >>>();
}
```



COMPOSABLE ALLOCATION

```
void application() {
  a<<< ..., s1 >>>();
  cudaEventRecord(e1, s1);
  cudaStreamWaitEvent(s2, e1);

  library_1(s1);
  library_2(s2);

  cudaEventRecord(e2, s2);
  cudaStreamWaitEvent(s1, e2);
  e<<< ..., stream >>>();
}
```




```
void library_1() {
  cudaMallocAsync(&X, ..., s2);
  cudaEventRecord(e2, s2);
  cudaStreamWaitEvent(s3, e2);
  b<<< ..., s2 >>>(X);
  c<<< ..., s3 >>>(X);
  cudaEventRecord(e3, s3);
  cudaStreamWaitEvent(s2, e3);
  cudaFreeAsync(X, s2);
}
```

```
void library_2() {
  cudaMallocAsync(&Y, s1);
  d<<< ..., stream >>>(Y);
  cudaFreeAsync(Y, s1);
}
```

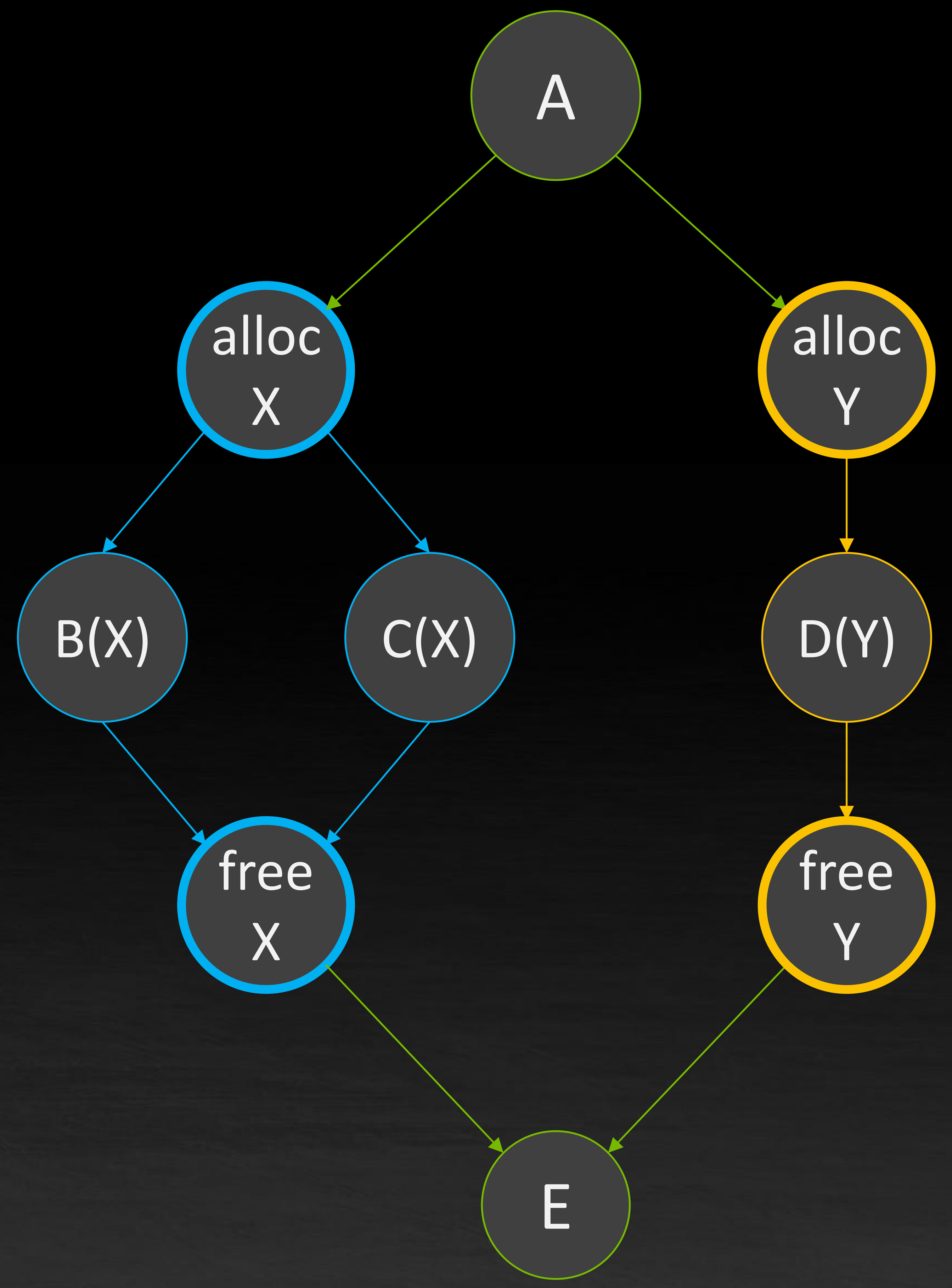
MEMORY ALLOCATION IN A TASK GRAPH (CUDA 11.4)

CUDA Graphs offers two new node types: **allocate** & **free**

Identical semantics to *cudaMallocAsync()* in a stream

- Pointer is returned **at node creation time**
- Returned pointer may be passed as arg to later nodes
- Using pointer is only valid downstream of allocation node & upstream of free node

Note: You may allocate in one graph and free in another – allocations persist between graphs until freed



NEW SEMANTICS UNIQUE TO GRAPHS

Allocation lifetime **MAY** extend outside the graph

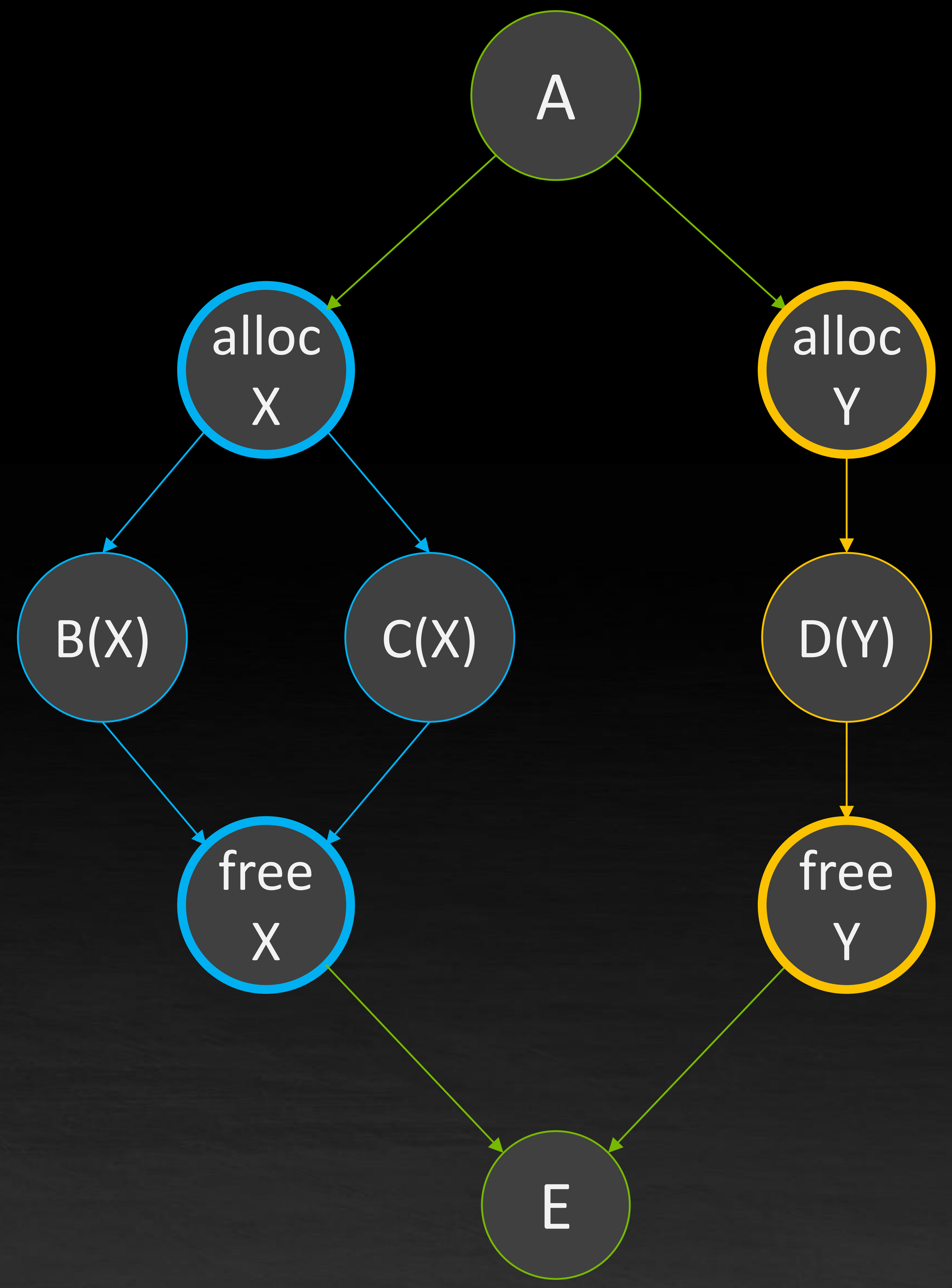
Each graph receives a unique VA range

Physical memory may be reused between graphs

Edges in graphs which have memory nodes may not be modified after creation

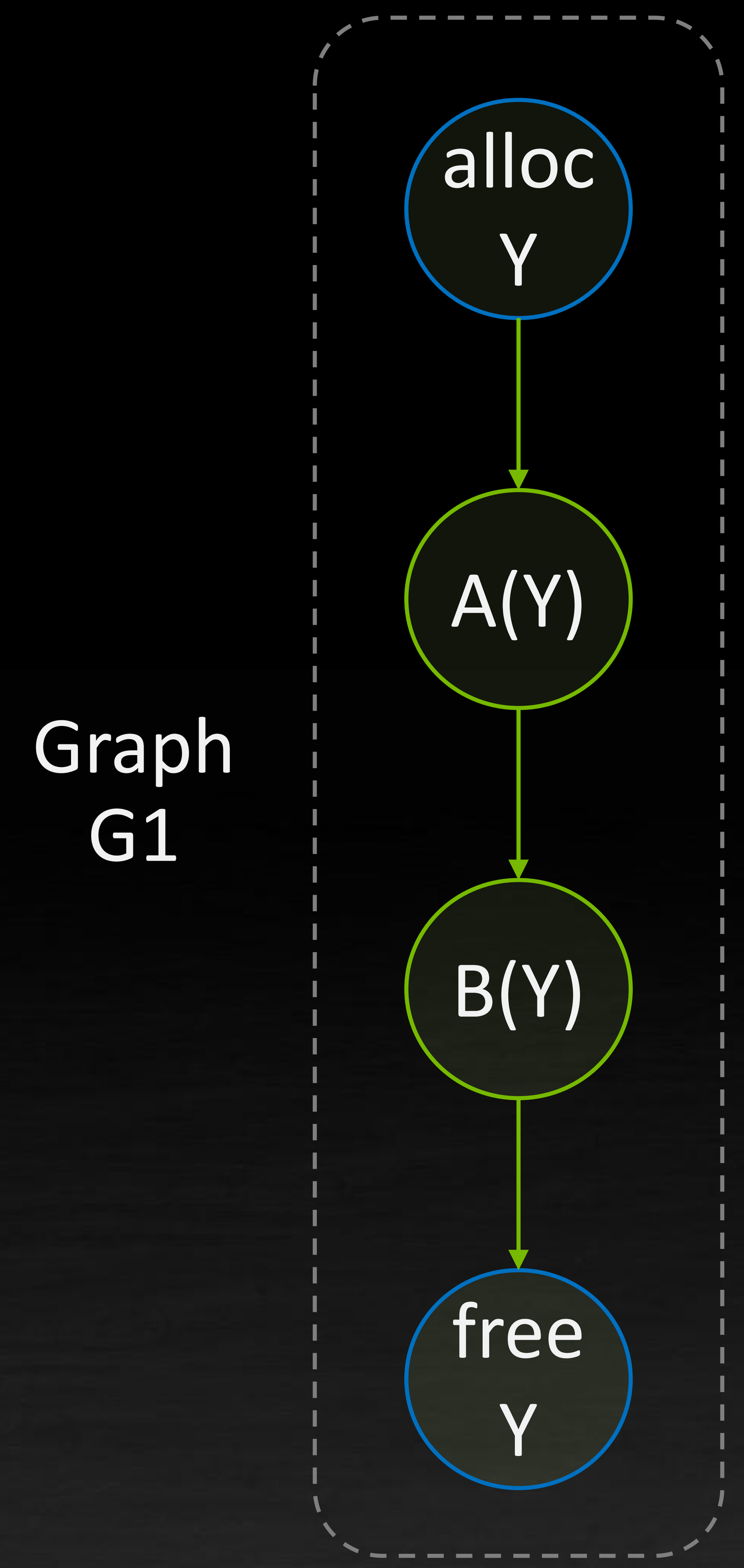
Allocation nodes may cause inter-graph serialization

IPC-shareability must be defined at allocation time

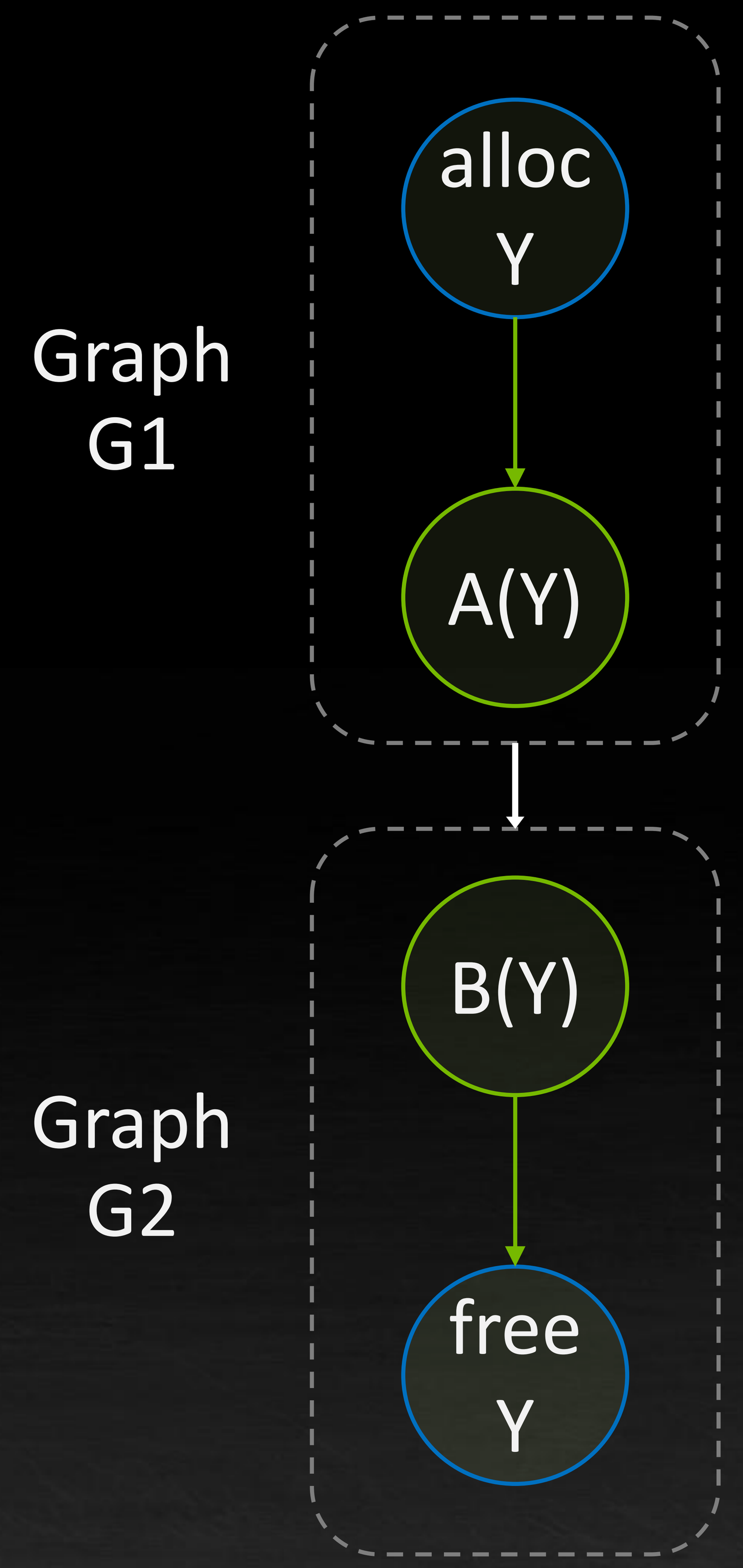


ALLOCATION LIFE MAY EXTEND OUTSIDE GRAPH

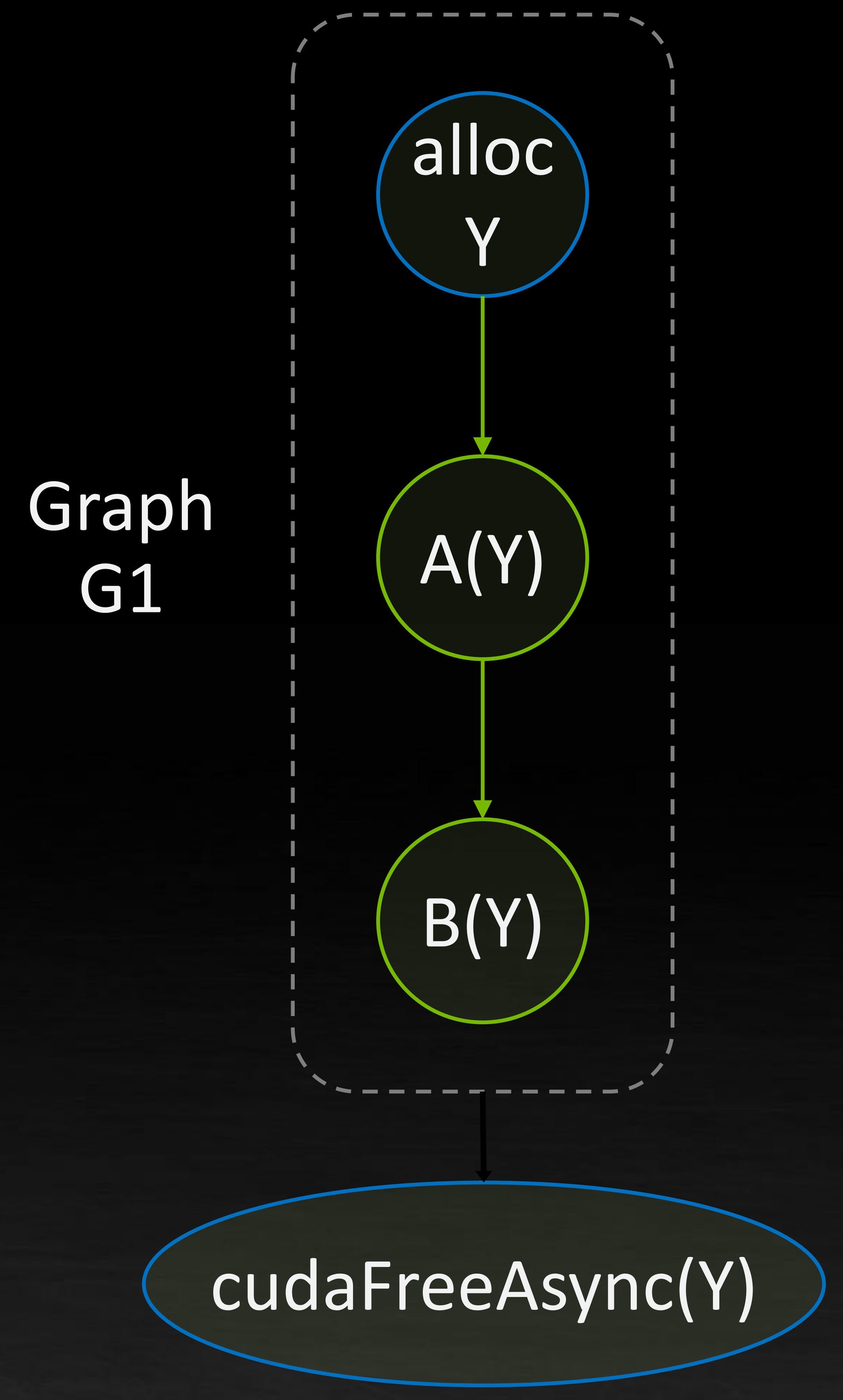
3 different patterns when freeing an allocation



Allocate & free
in same graph



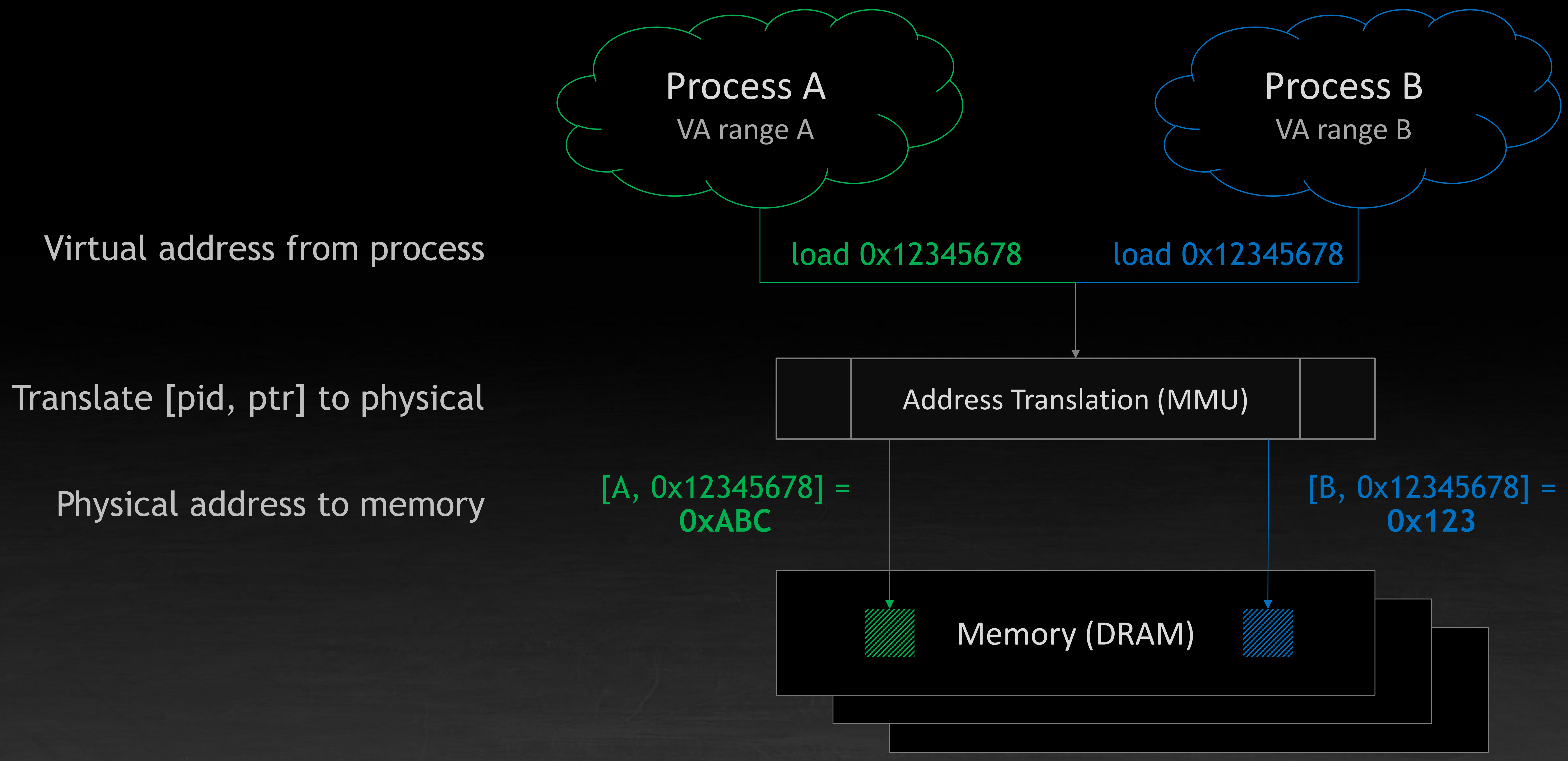
Allocate in one graph
free later in another



Allocate in one graph
free later via
cudaFreeAsync()

ADDRESS TRANSLATION

Virtual Addresses (VA) vs. Physical Addresses (PA)



MEMORY-SPACE MANAGEMENT IN GRAPHS

Virtual & Physical Space Lifetimes Are Different

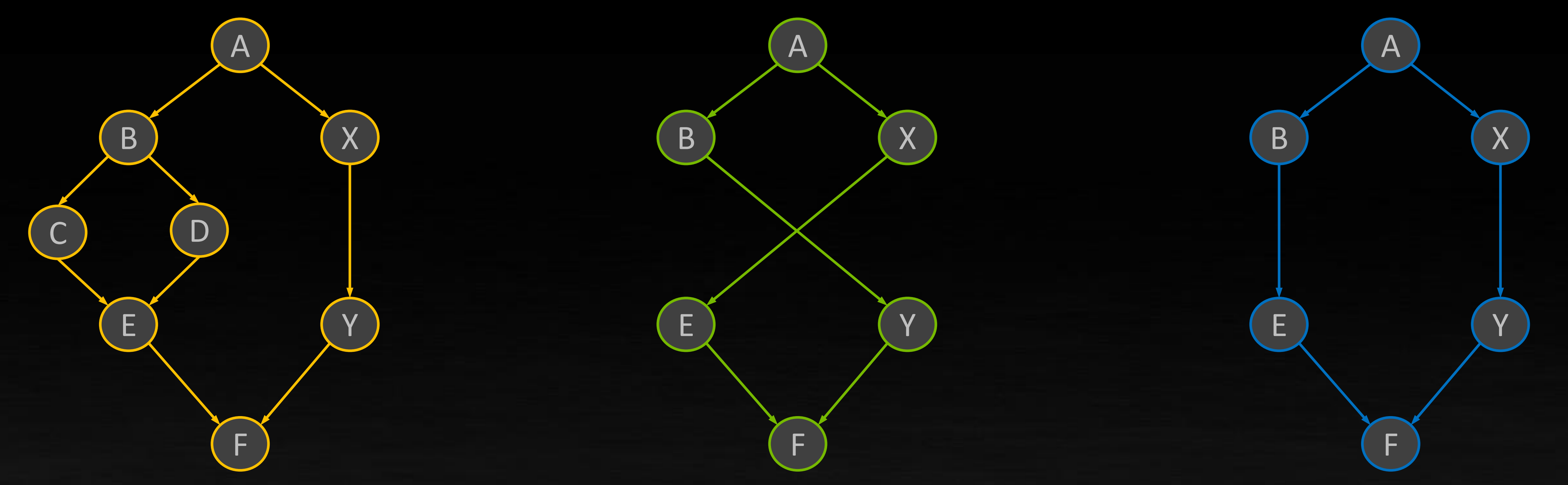
Graph-specific allocation behaviours

Each graph has a **unique** address space (VA), set up when it is created

Physical pages **are not mapped** at graph node creation – only a placeholder address is returned

Private address ranges remain valid for lifetime of a graph, until the graph is destroyed

Per-graph address ranges guarantee pointer lifetimes have graph lifetime



VA range 1 [start1 : end1]   VA range 2 [start2 : end2]   VA range 3 [start3 : end3]

SHARED PHYSICAL PAGE MAPPINGS

Goal: Reduce Physical Footprint Of Creating Lots Of Graphs

Virtual Address Range

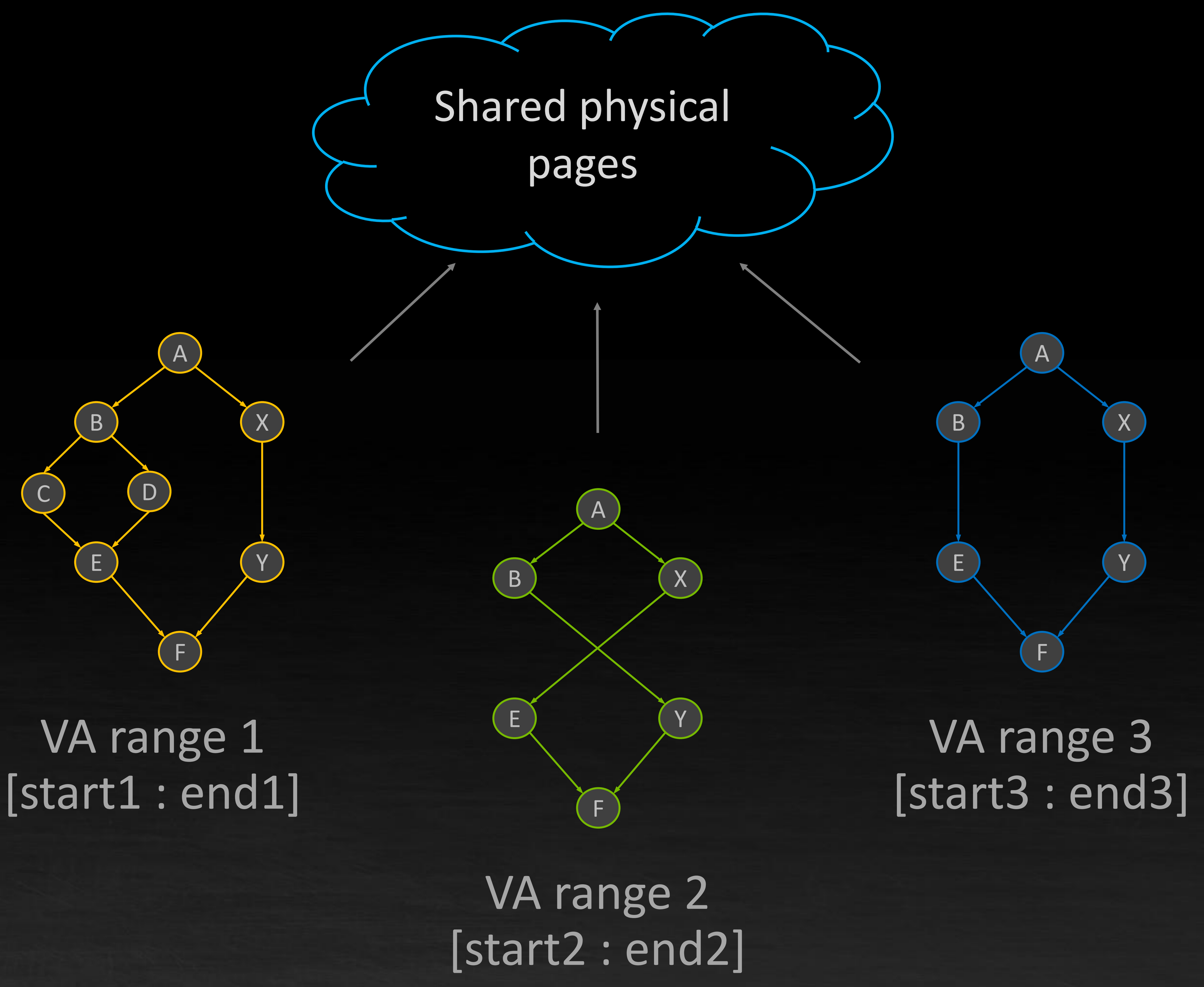
Each graph has a **private address range**, so pointer lifetimes have graph lifetime

Physical Pages

A **single set of pages** is reserved equal to the largest footprint of any graph

VA<->PA Mapping

All graphs **map to the same page set**, unless executing concurrently



CUDA DYNAMIC PARALLELISM HELLO WORLD

hello.cu

CPU portion

```
void main() {
    hello<<< 1, 1 >>>();
    cudaDeviceSynchronize();
}
```

GPU portion

```
__global__ void hello() {
    printf("Hello, CUDA\n");
}
```

```
$nvcc hello.cu -o hello_cuda
$ ./hello_cuda
Hello, CUDA

$
```

Console output

CUDA DYNAMIC PARALLELISM HELLO WORLD

hello.cu

CPU portion

```
void main() {
    hello<<< 1, 1 >>>();
    cudaDeviceSynchronize();
}
```

GPU portion

```
__global__ void hello() {
    child_hello<<< 1, 1 >>>();
}

__global__ void child_hello() {
    printf(“Hello, CUDA\n”);
}
```

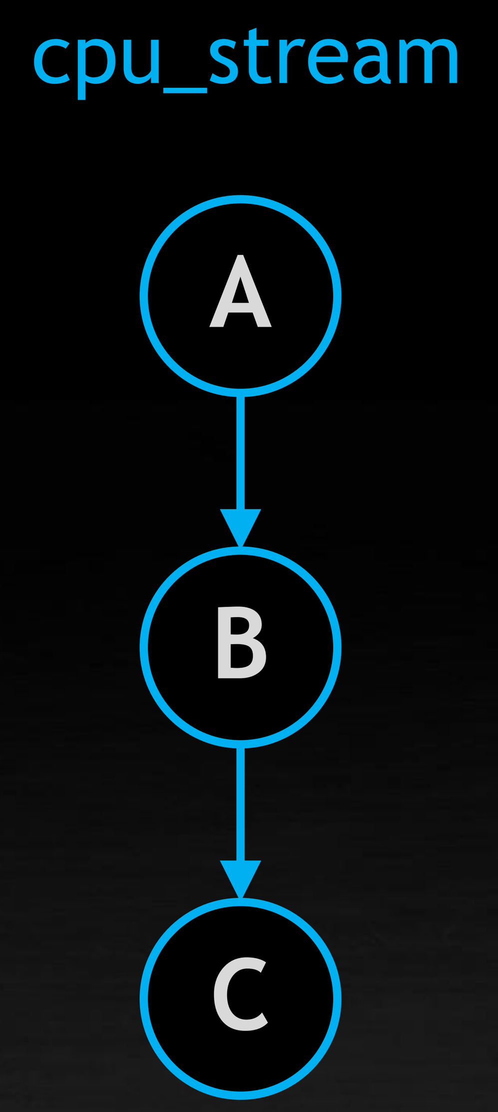
Launch a kernel from the GPU

```
$nvcc hello.cu -o hello_cuda
$ ./hello_cuda
Hello, CUDA

$
```

Console output

DYNAMIC PARALLELISM PROGRAMMING MODEL: ENCAPSULATION

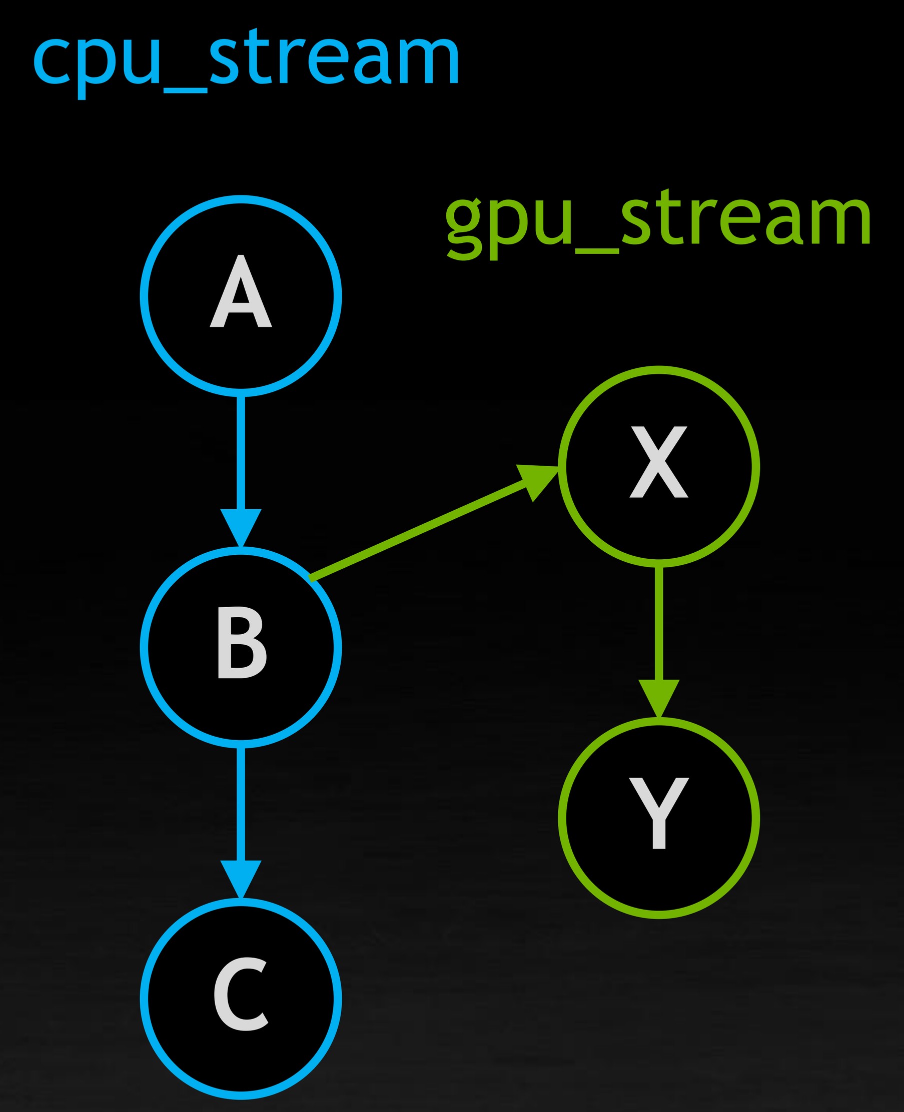
```
void main() {
    cudaStream_t cpu_stream;
    cudaStreamCreate(&cpu_stream);

    A <<< ..., cpu_stream >>>();
    B <<< ..., cpu_stream >>>();
    C <<< ..., cpu_stream >>>();

    cudaStreamSynchronize(cpu_stream);
}
```



DYNAMIC PARALLELISM PROGRAMMING MODEL: ENCAPSULATION

```
void main() {
    cudaStream_t cpu_stream;
    cudaStreamCreate(&cpu_stream);

    A <<< ..., cpu_stream >>>();
    B <<< ..., cpu_stream >>>();
    C <<< ..., cpu_stream >>>();

    cudaStreamSynchronize(cpu_stream);
}
```

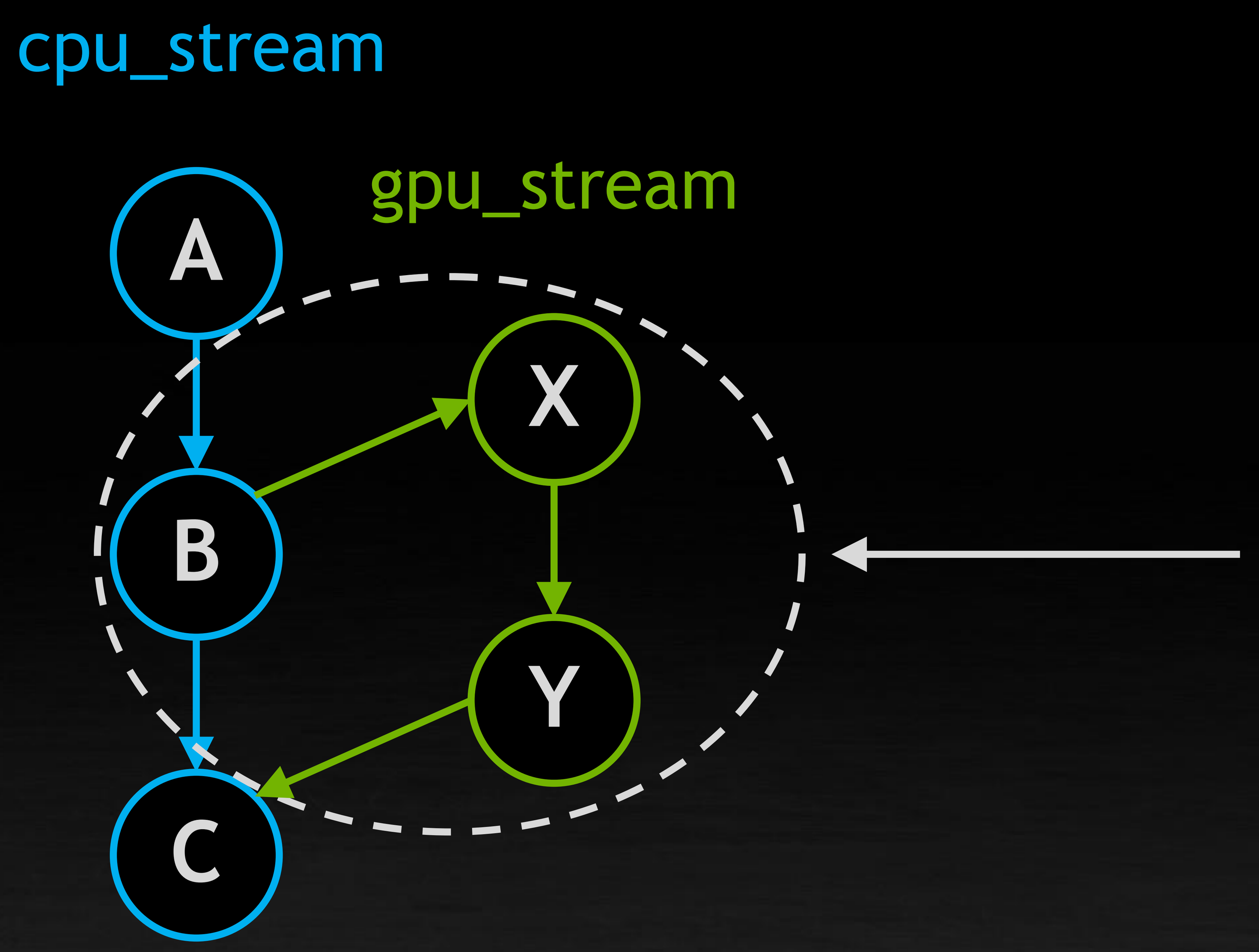
```
__global__ void B() {
    cudaStream_t gpu_stream;
    cudaStreamCreateWithFlags(&gpu_stream,
                              cudaStreamNonBlocking);

    X <<< ..., gpu_stream >>>();
    Y <<< ..., gpu_stream >>>();

    do_something();
}
```



DYNAMIC PARALLELISM PROGRAMMING MODEL: ENCAPSULATION

```
void main() {
    cudaStream_t cpu_stream;
    cudaStreamCreate(&cpu_stream);

    A <<< ..., cpu_stream >>>();
    B <<< ..., cpu_stream >>>();
    C <<< ..., cpu_stream >>>();

    cudaStreamSynchronize(cpu_stream);
}
```

```
__global__ void B() {
    cudaStream_t gpu_stream;
    cudaStreamCreateWithFlags(&gpu_stream,
                              cudaStreamNonBlocking);

    X <<< ..., gpu_stream >>>();
    Y <<< ..., gpu_stream >>>();

    do_something();
}
```



Encapsulation boundary

All launches from B just look like part of B from the outside

NAMED STREAMS

```
__global__ void B() {
    cudaStream_t gpu_stream;
    cudaStreamCreateWithFlags(&gpu_stream, cudaStreamPerThread);

    X <<< ..., gpu_stream >>>();
    Y <<< ..., gpu_stream >>>();
}
```

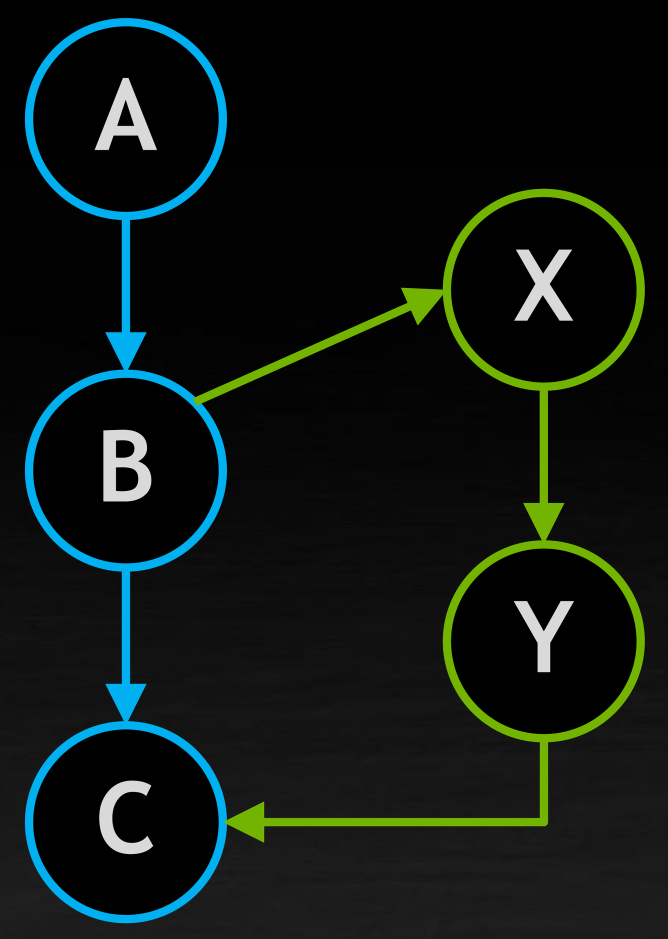
Previous example code using generic stream creation

```
__global__ void B() {
    X <<< ..., cudaStreamPerThread >>>();
    Y <<< ..., cudaStreamPerThread >>>();
}
```

Similar code using “named” stream

OPTIMIZING COMMON LAUNCH PATTERNS

A higher-performance, enhanced programming model using “named streams”



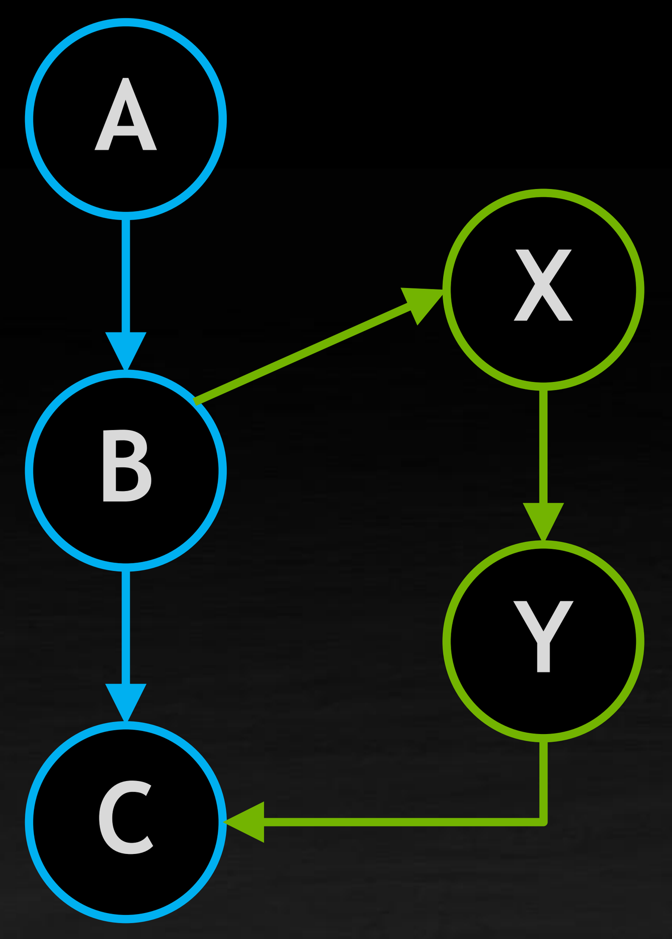

```
__global__ void B() {
    X <<< ..., cudaStreamPerThread >>>();
    Y <<< ..., cudaStreamPerThread >>>();
}
```

Per-Thread stream

X & Y execute **sequentially**, similar to existing stream launch

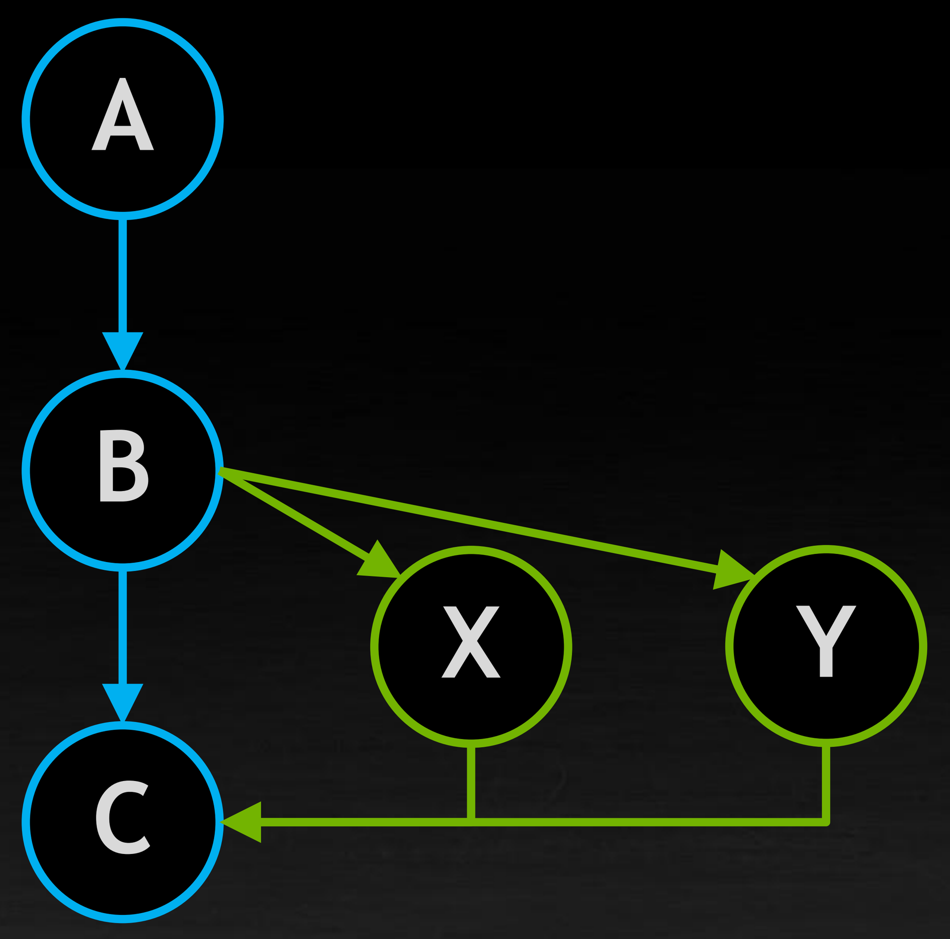
THREE NEW TYPES OF DEVICE-SIDE KERNEL LAUNCH

A higher-performance, enhanced programming model using “named streams”



Per-Thread stream

X & Y execute **sequentially**,
similar to existing stream launch



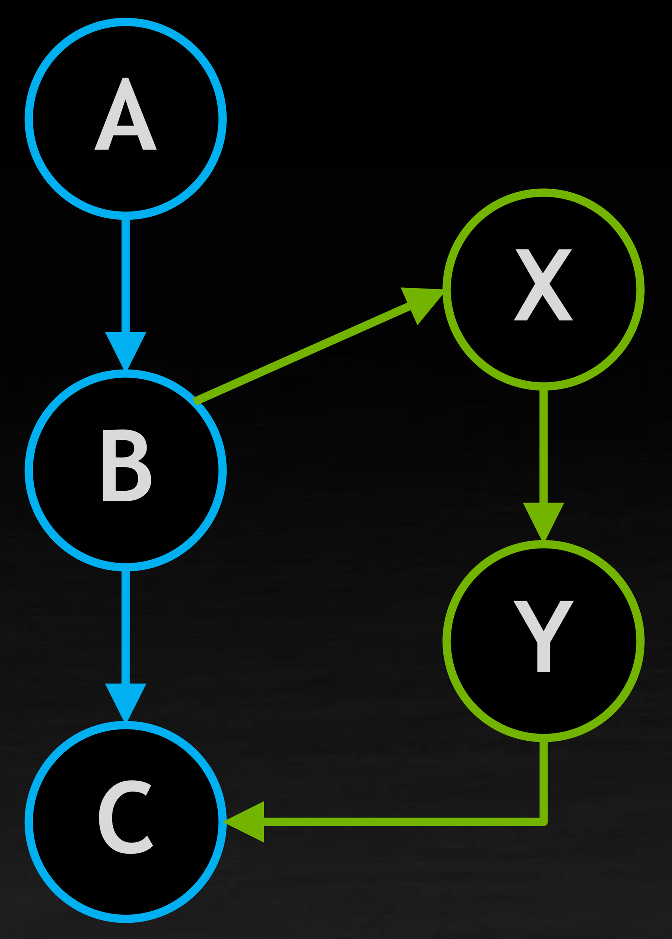
Fire-and-forget

X & Y execute **independently**
as if launched in separate streams

```
__global__ void B() {
    X <<< ..., cudaStreamFireAndForget >>>();
    Y <<< ..., cudaStreamFireAndForget >>>();
}
```

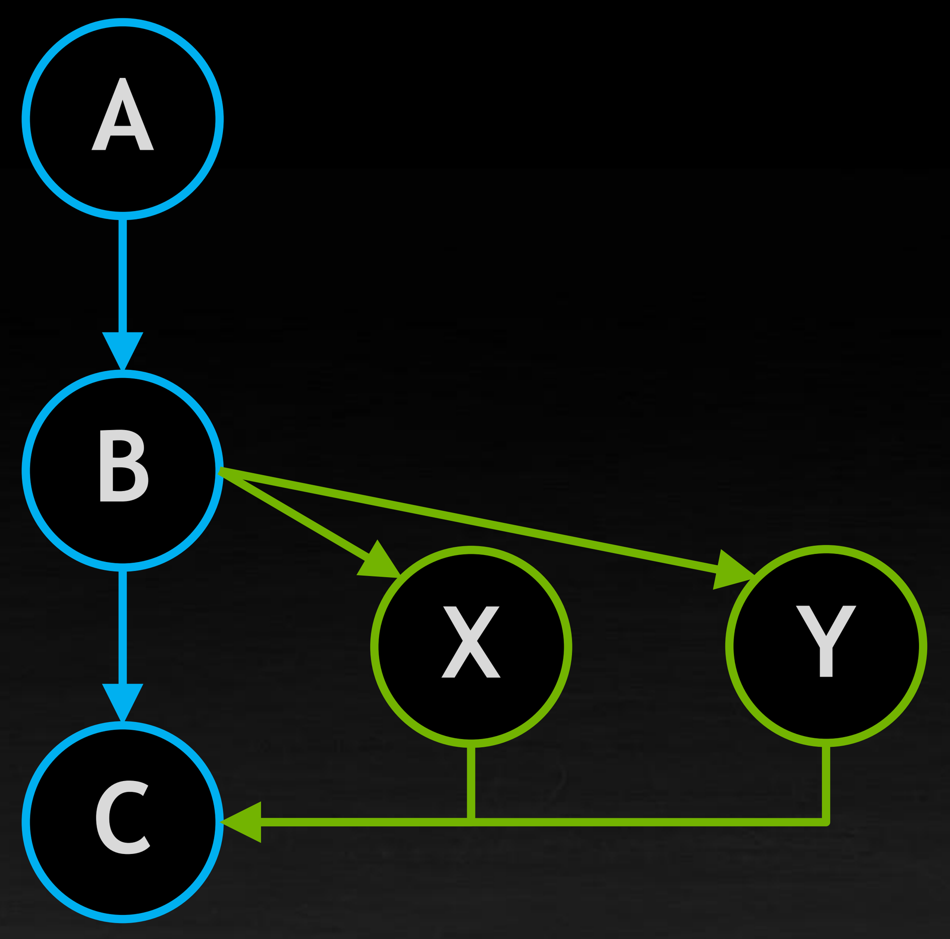
THREE NEW TYPES OF DEVICE-SIDE KERNEL LAUNCH

A higher-performance, enhanced programming model using “named streams”



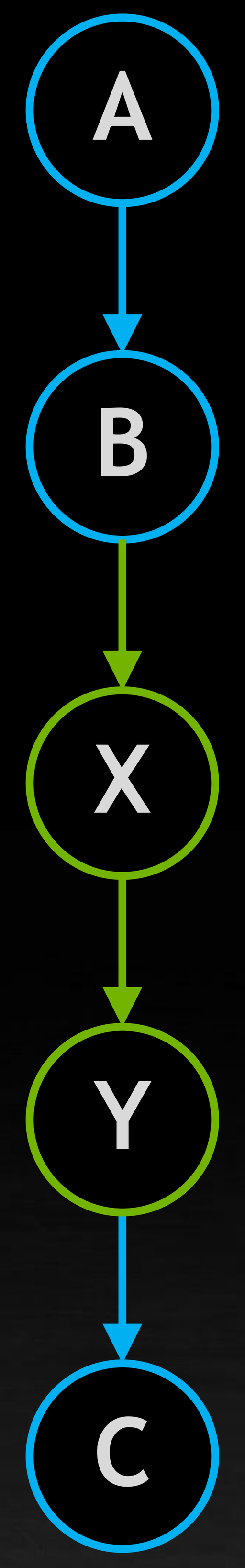
Per-Thread stream

X & Y execute **sequentially**,
similar to existing stream launch



Fire-and-forget

X & Y execute **independently**
as if launched in separate streams



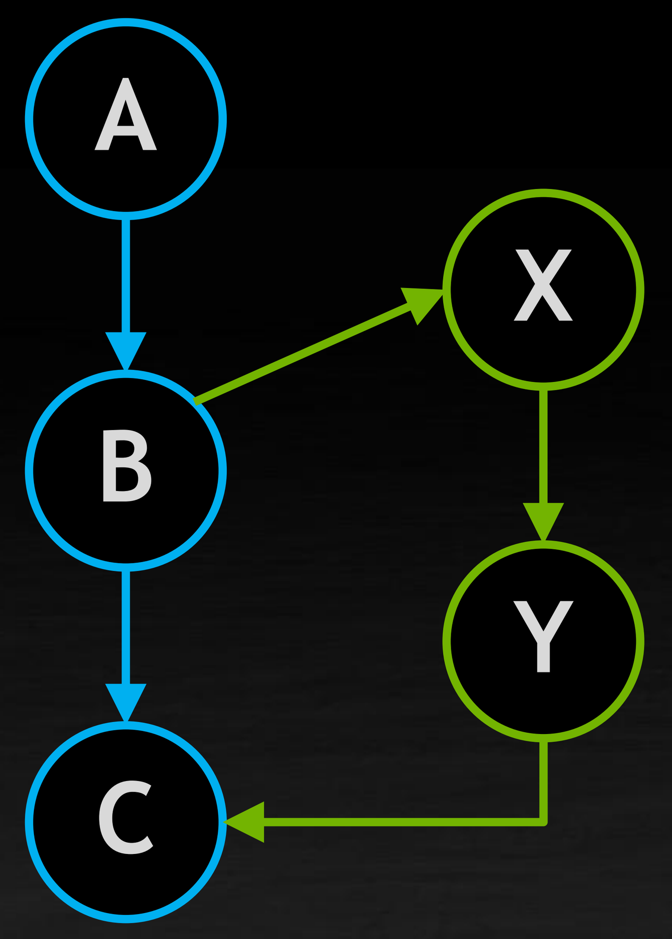

```
__global__ void B() {
    X <<< ..., cudaStreamTailLaunch >>>();
    Y <<< ..., cudaStreamTailLaunch >>>();
}
```

Tail launch

X & Y execute sequentially
after parent kernel completes

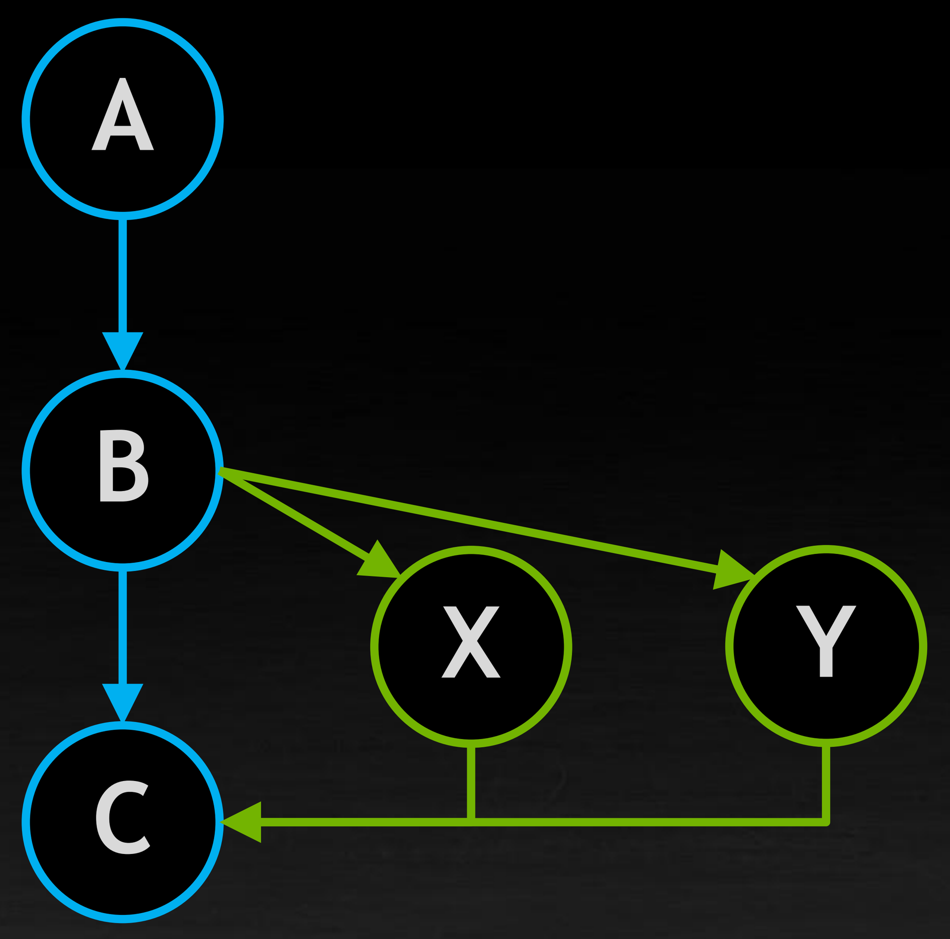
THREE NEW TYPES OF DEVICE-SIDE KERNEL LAUNCH

A higher-performance, enhanced programming model using “named streams”



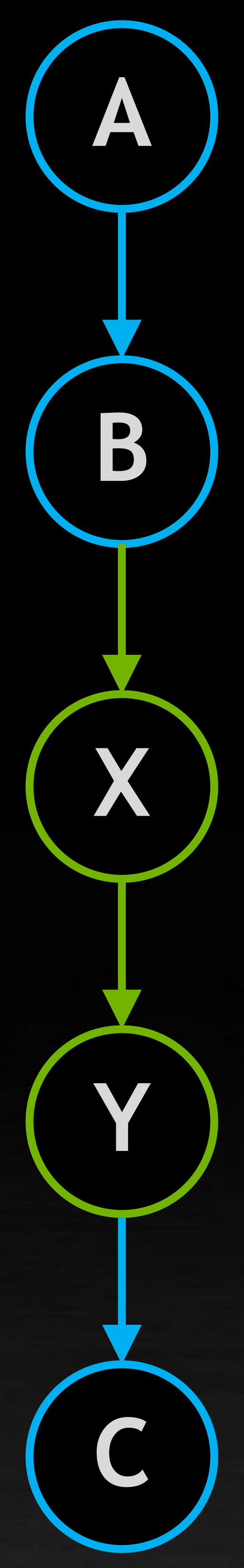
Per-Thread stream

X & Y execute **sequentially**, similar to existing stream launch



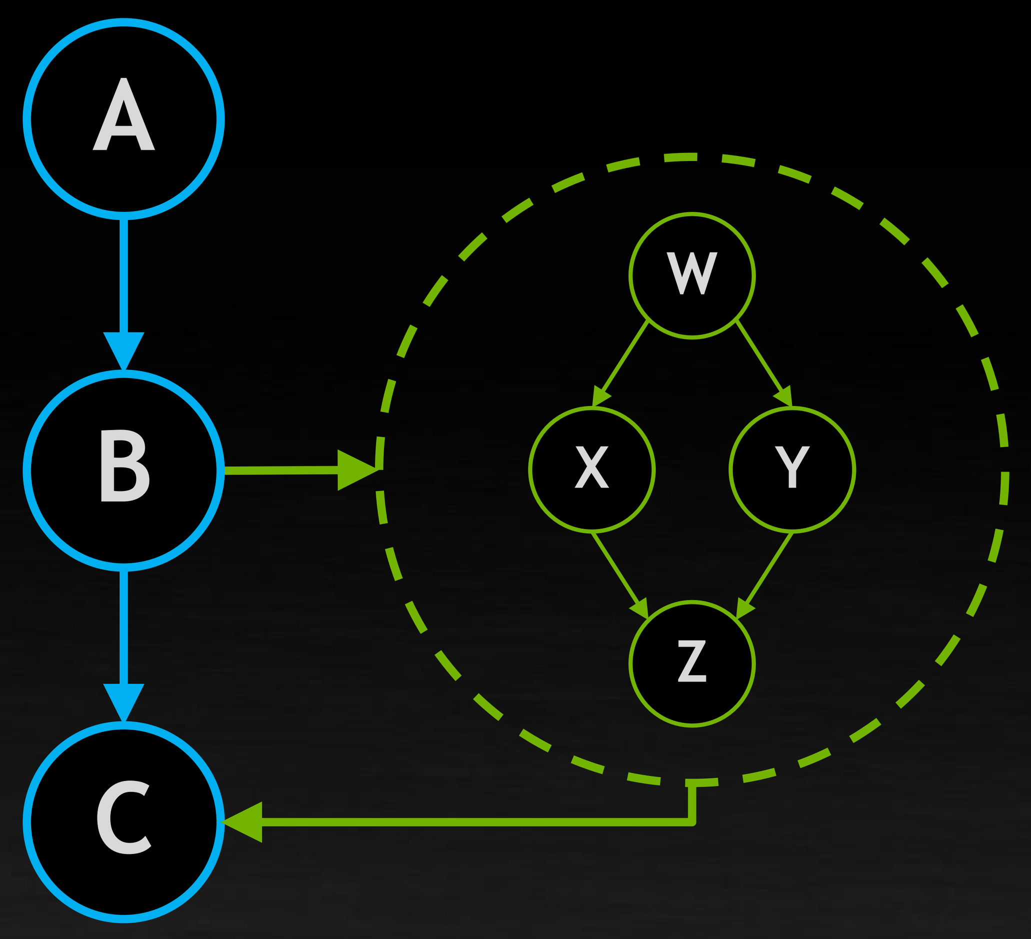
Fire-and-forget

X & Y execute **independently** as if launched in separate streams



Tail launch

X & Y execute sequentially **after** parent kernel completes



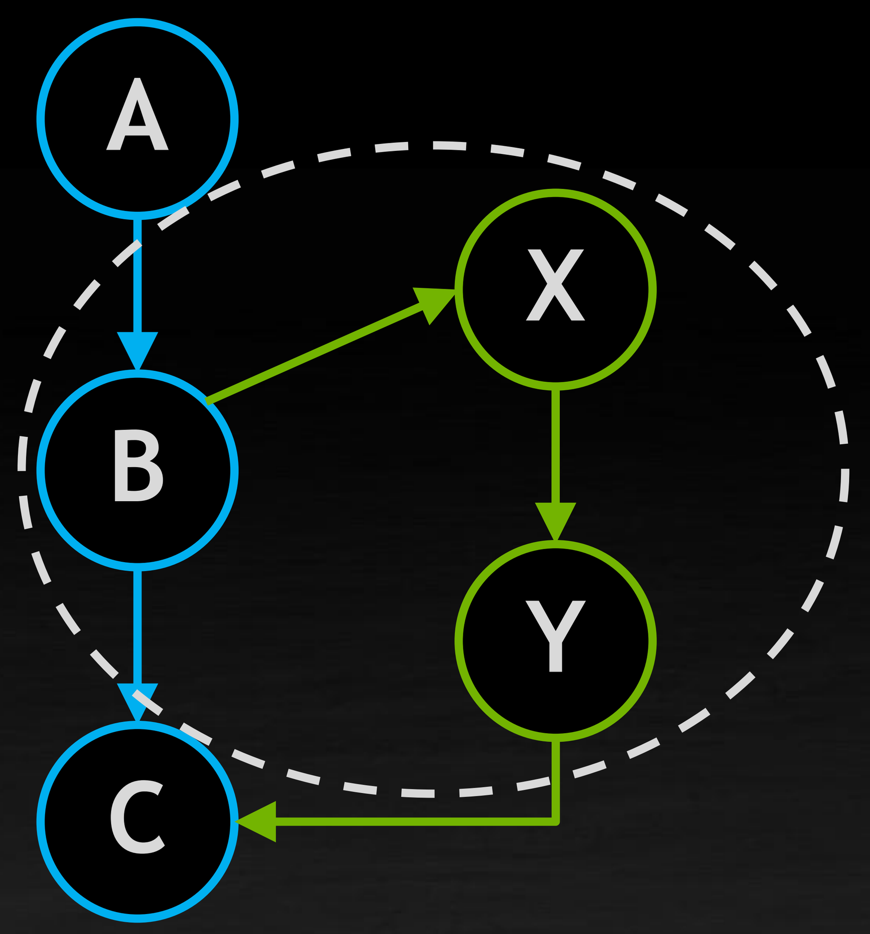
Graph launch

Can now launch whole graphs from a GPU kernel

NVIDIA

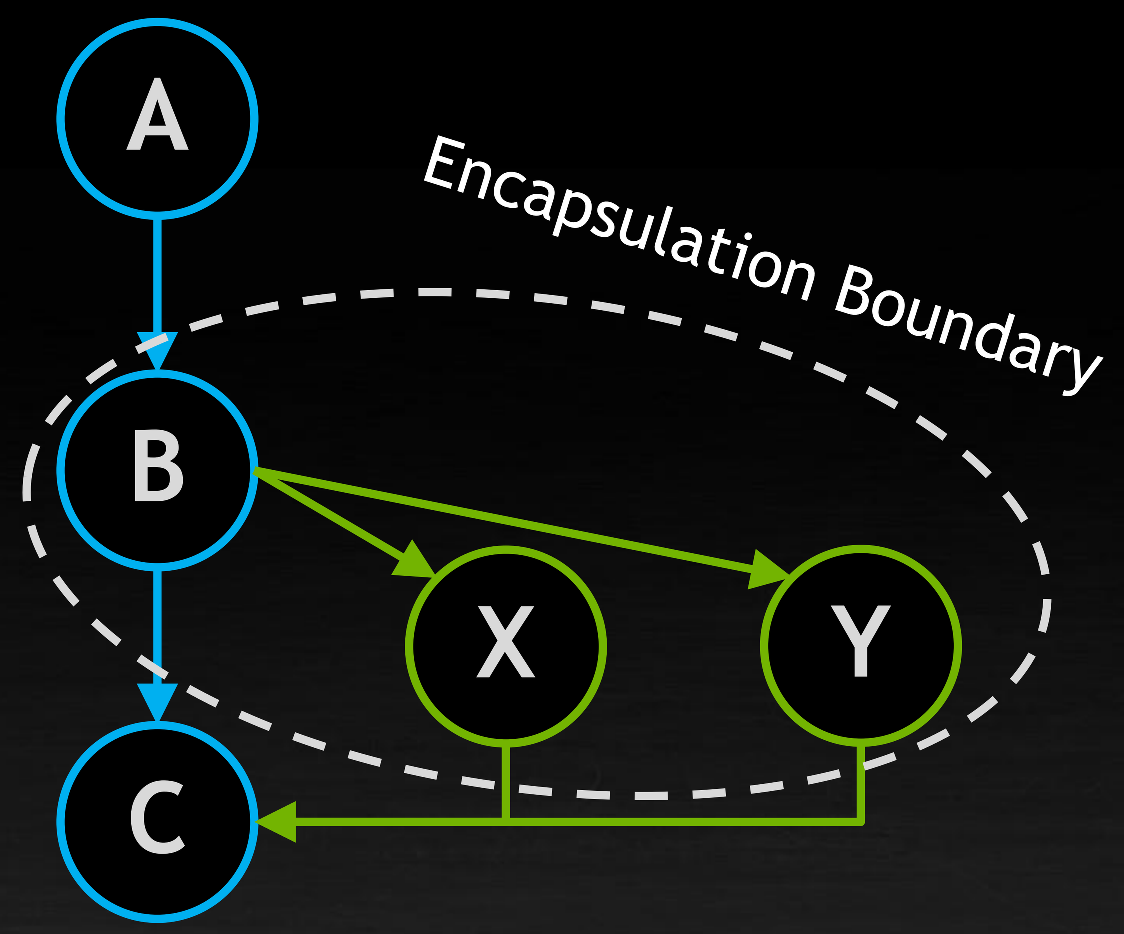
THREE NEW TYPES OF DEVICE-SIDE KERNEL LAUNCH

Encapsulation rule always applies



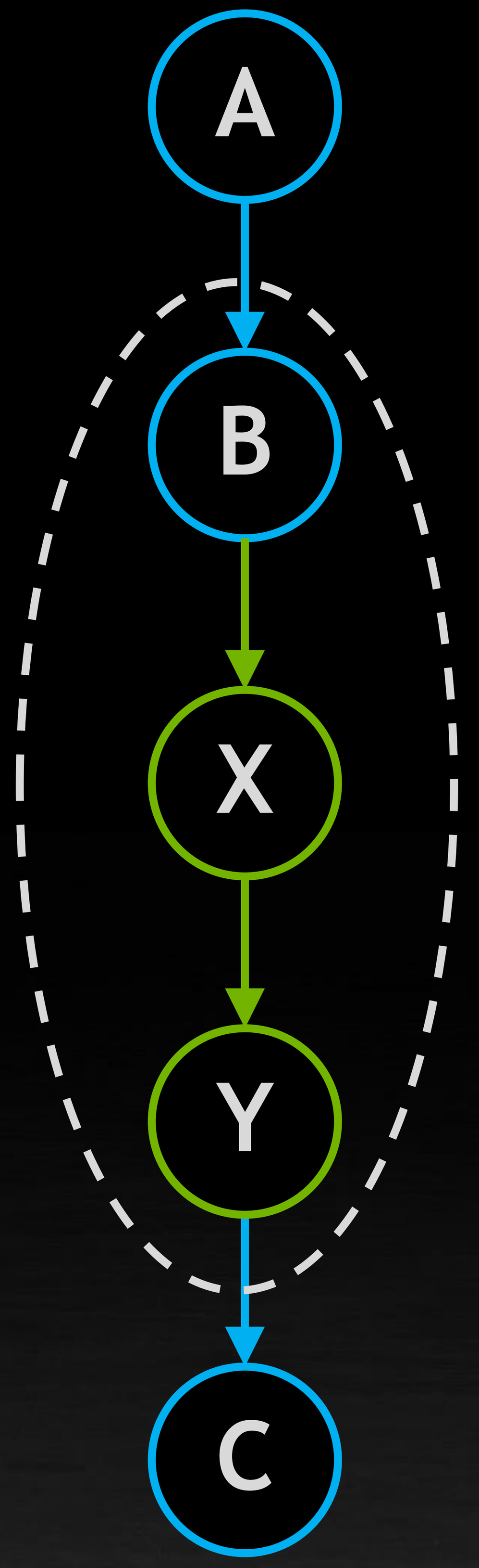
Per-Thread stream

X & Y execute **sequentially**, similar to existing stream launch



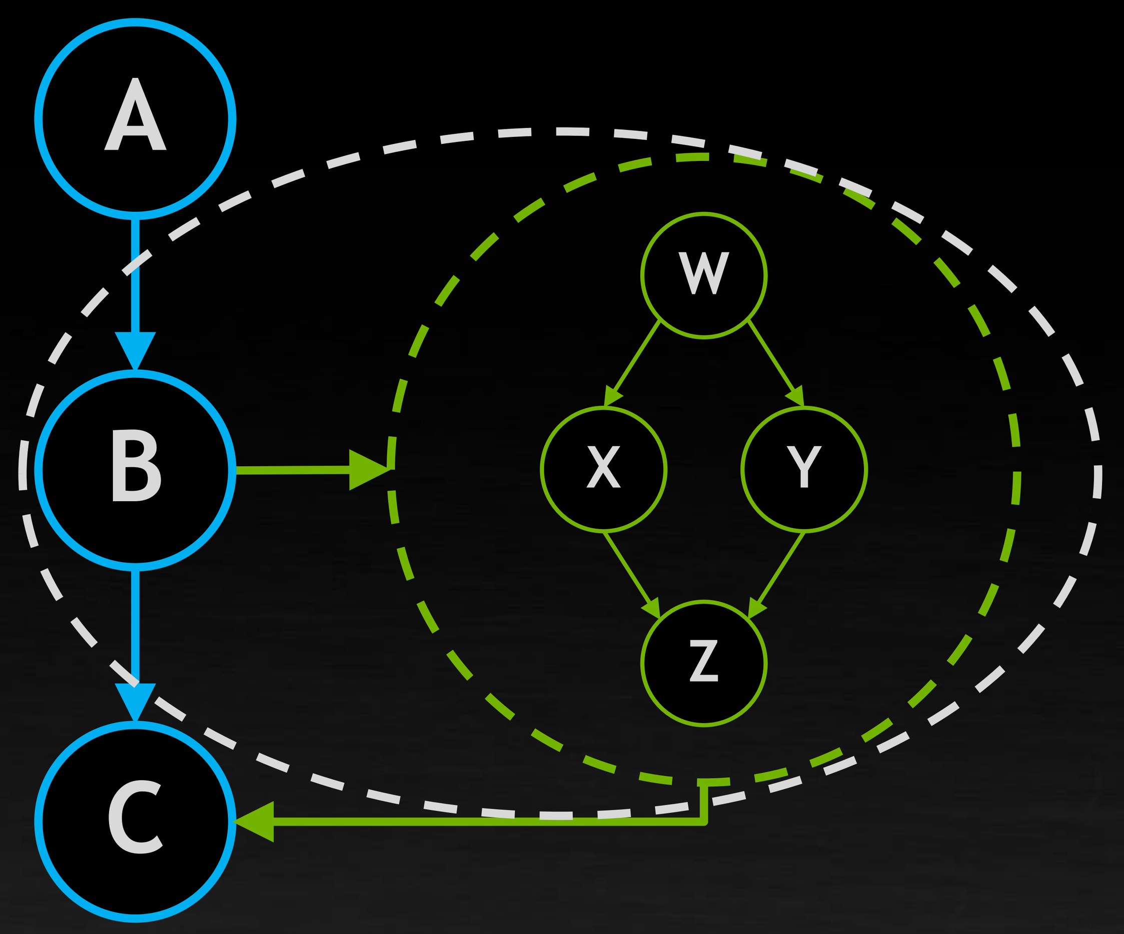
Fire-and-forget

X & Y execute **independently** as if launched in separate streams



Tail launch

X & Y execute sequentially **after** parent kernel completes

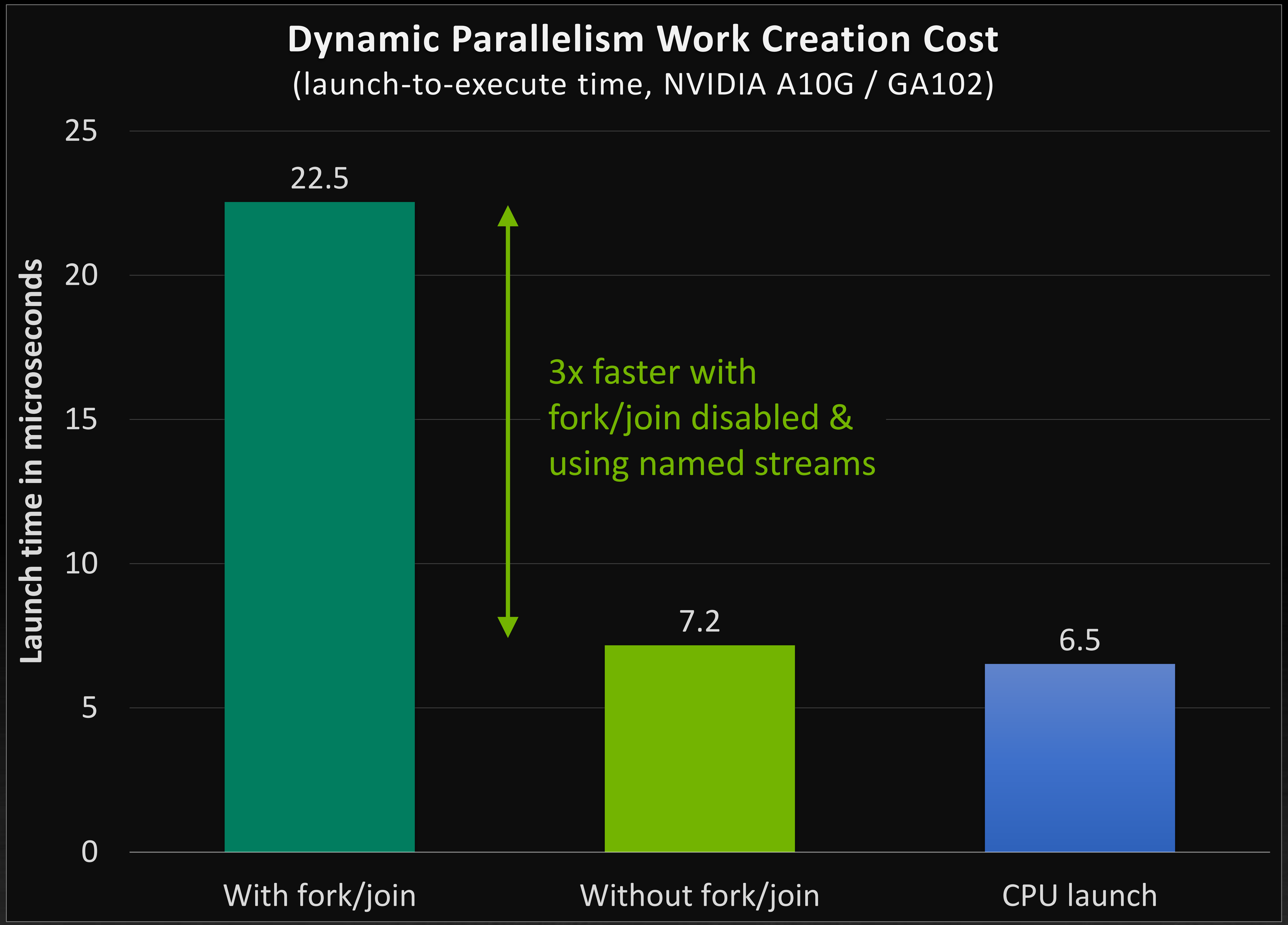


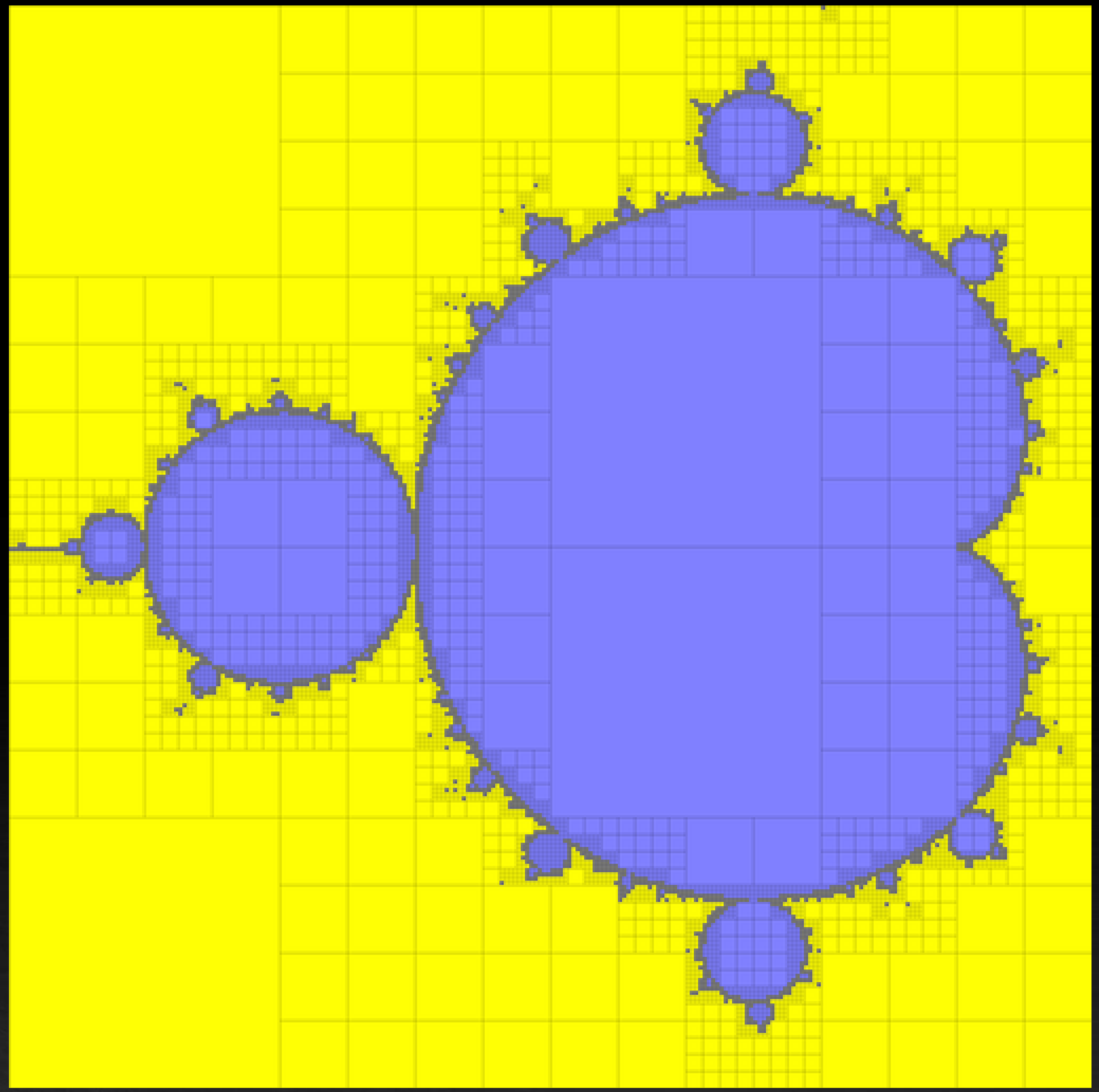
Graph launch

Can now launch whole graphs from a GPU kernel

NVIDIA

PUTTING IT ALL TOGETHER





Adaptive parallel Mandelbrot → 14% end-to-end speedup

Mariani-Silver Algorithm on 16384x16384 grid, NVIDIA A10G / GA102

Adaptive Parallel Computation with CUDA Dynamic Parallelism [Technical Blog]

DYNAMIC PARALLEL TASK GRAPHS

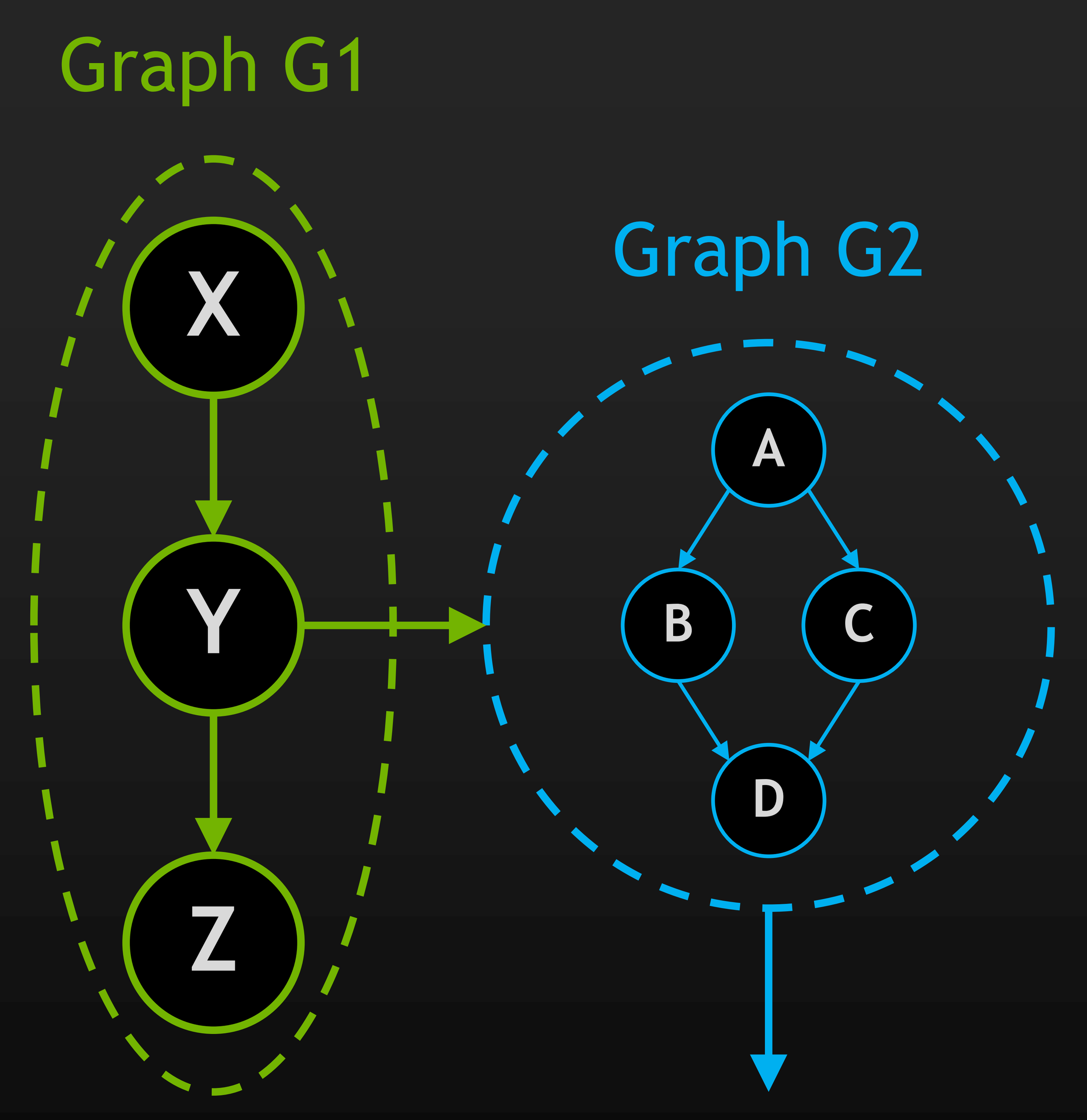
cdp_graphs.cu

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, ...);
}
```



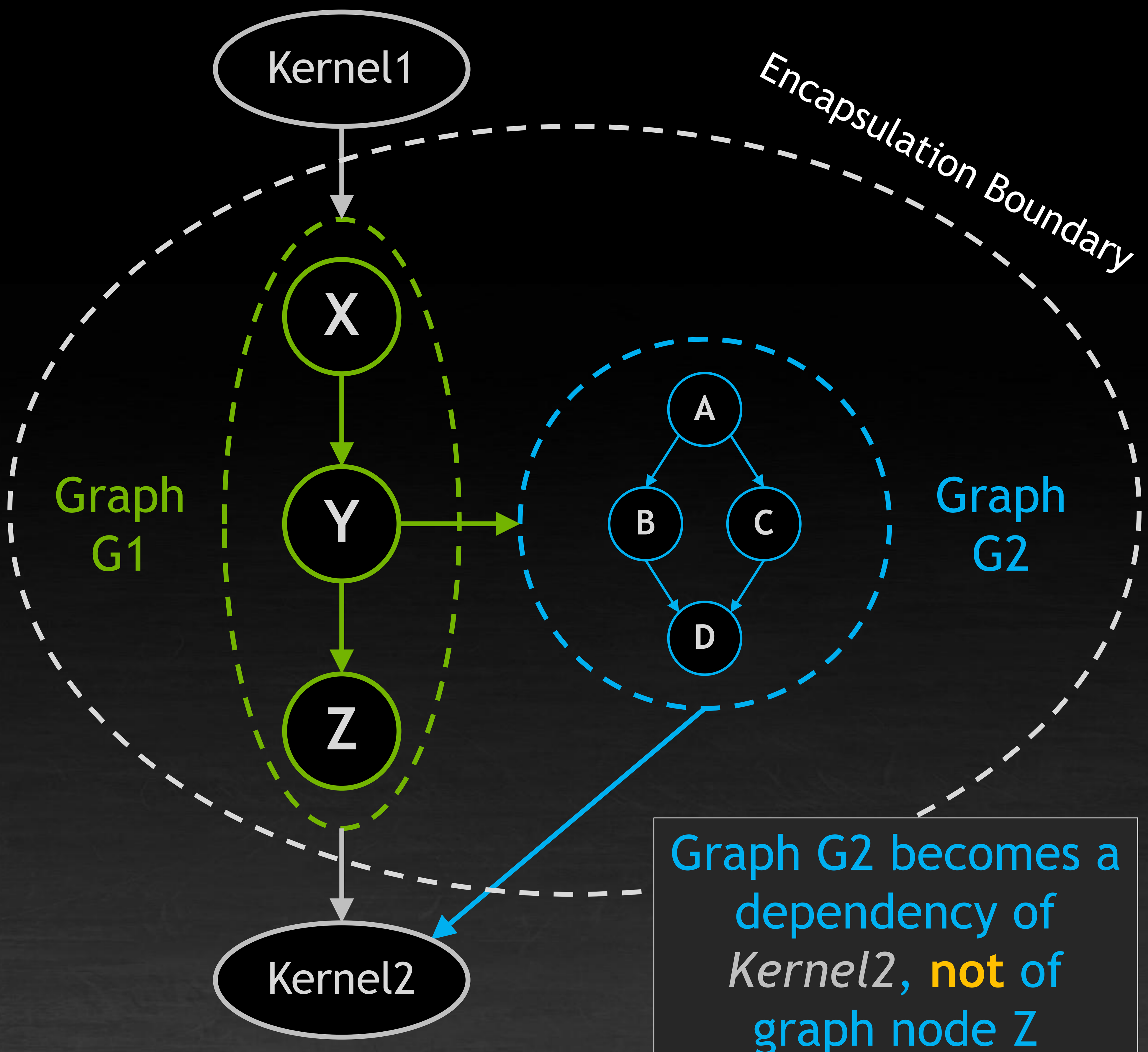
Device-side graph launch

ENCAPSULATION FOR DEVICE-SIDE GRAPH LAUNCH

Parent graphs are monolithic with respect to dependency resolution

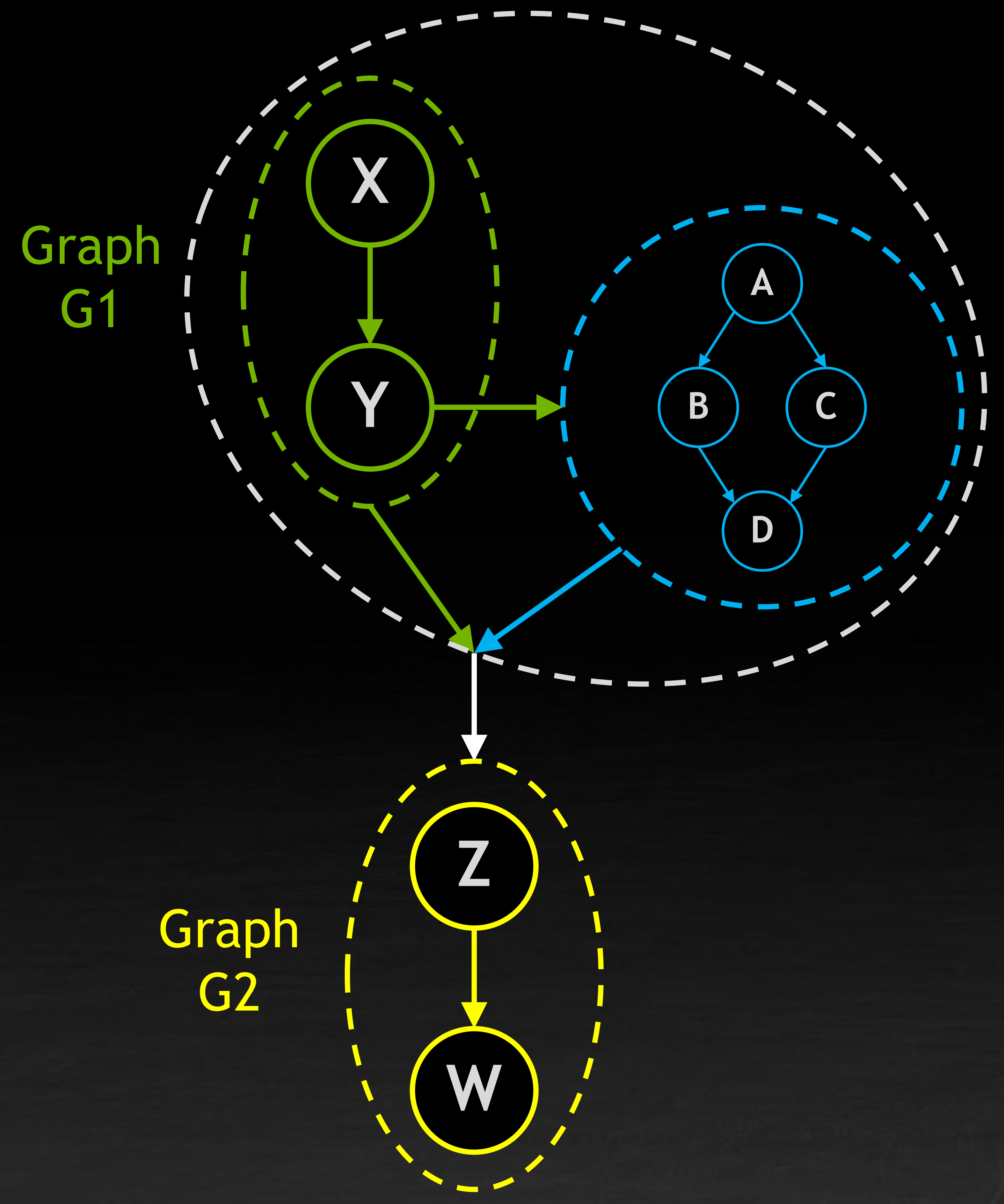
Graph encapsulation boundary is **the whole launching graph**

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

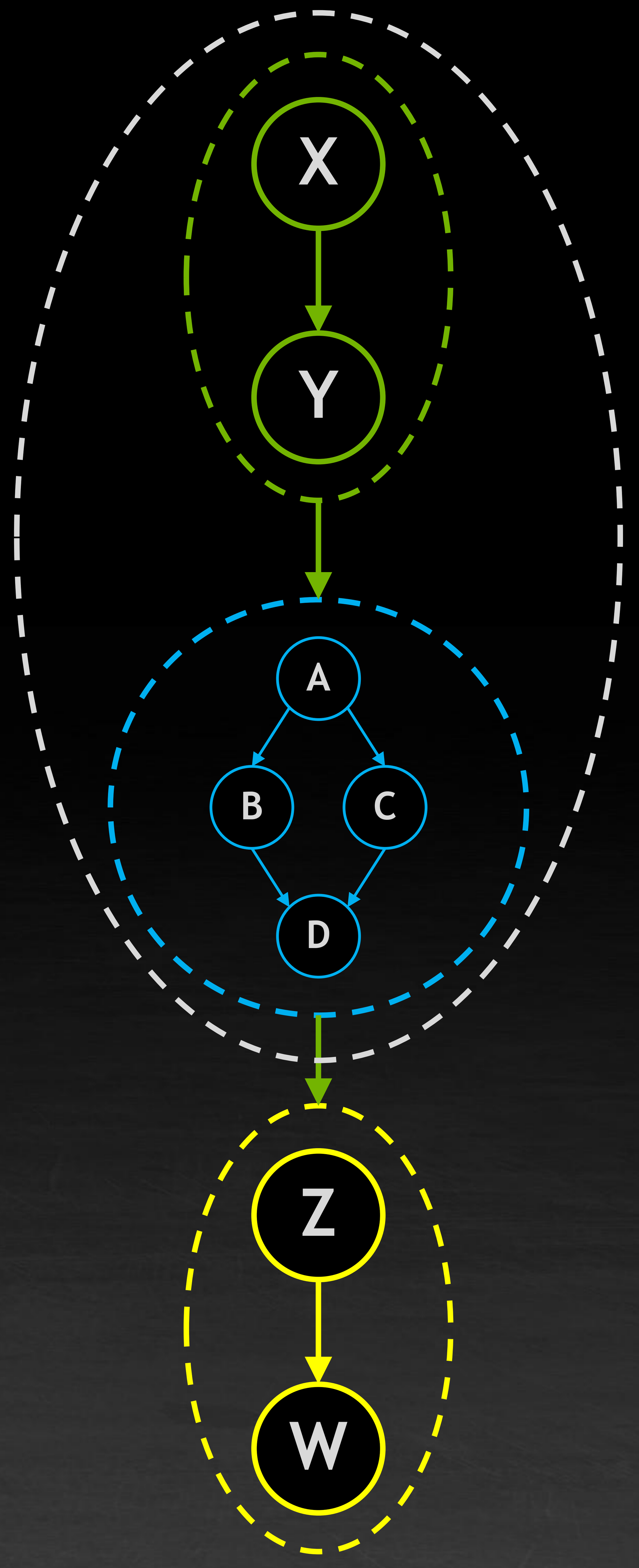
Identical semantics to dynamic parallelism single-kernel launch named streams, but at whole-graph granularity



Fire-and-Forget

Child work is launched concurrently with parent

Graph G2 now depends on G1 and child work



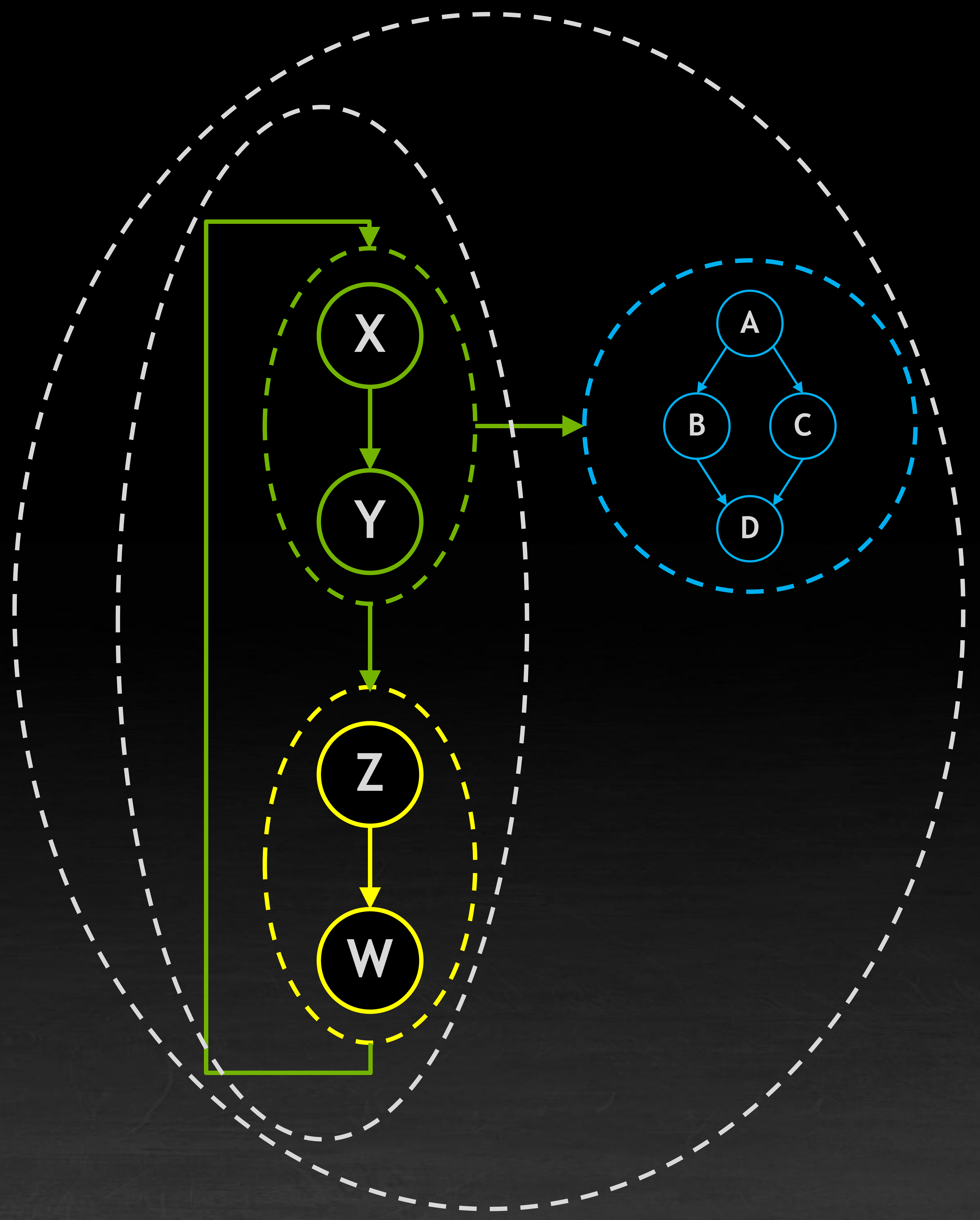
Tail Launch

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

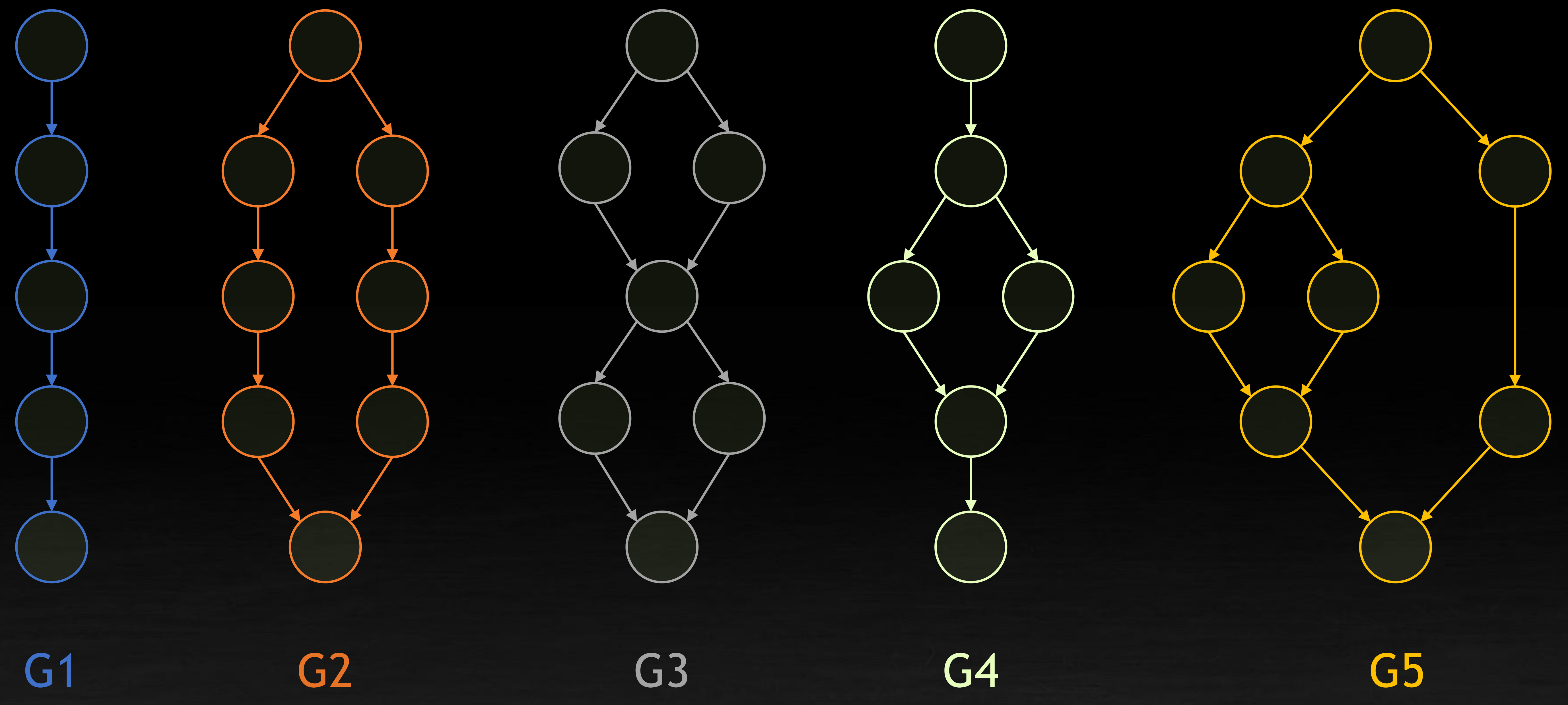


Sibling

Child work is launched concurrently with parent

Child work is now a dependency of parent’s parent

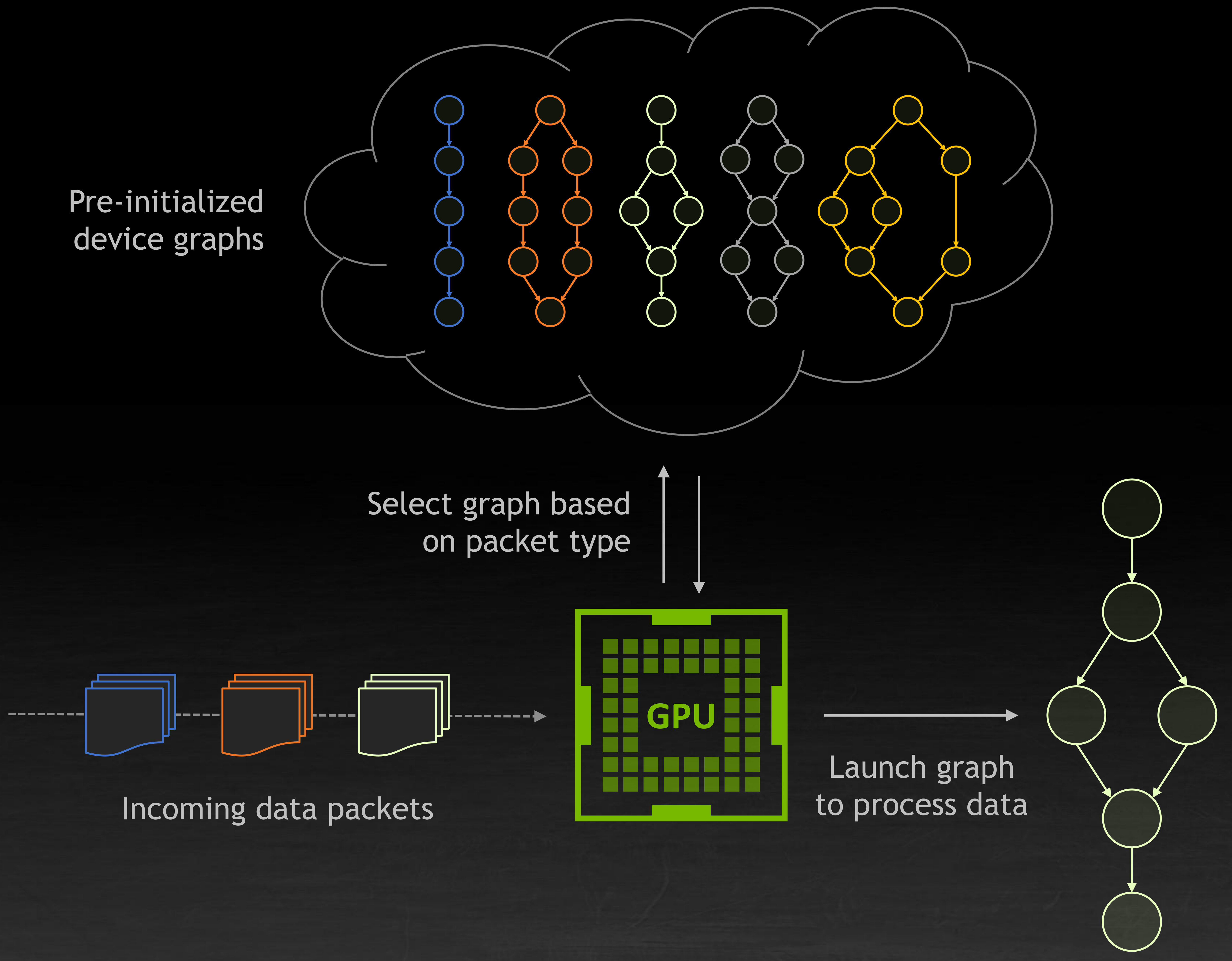
EXAMPLE: RUN-TIME DYNAMIC WORK SCHEDULING




```
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

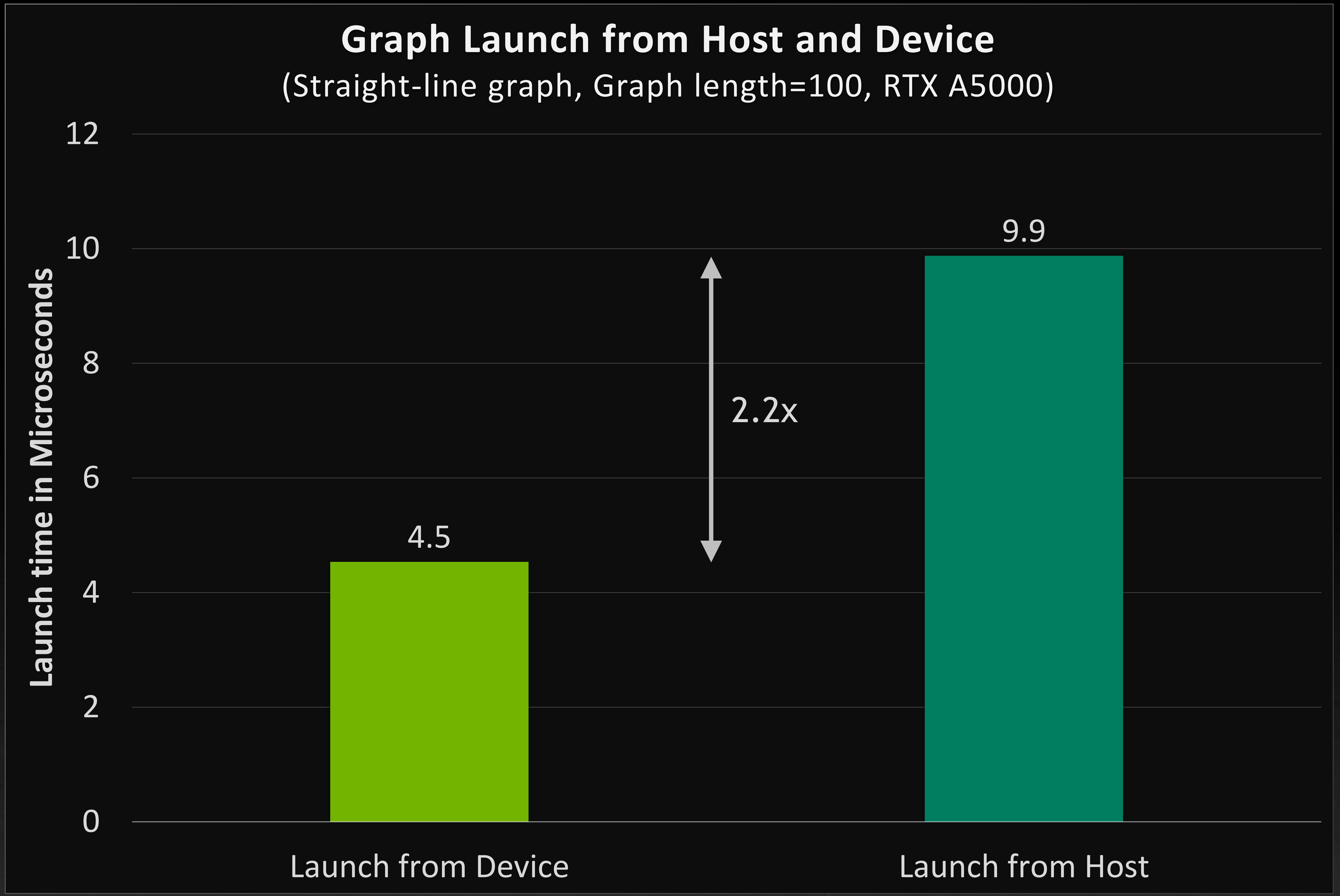
EXAMPLE: RUN-TIME DYNAMIC WORK SCHEDULING




```
__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

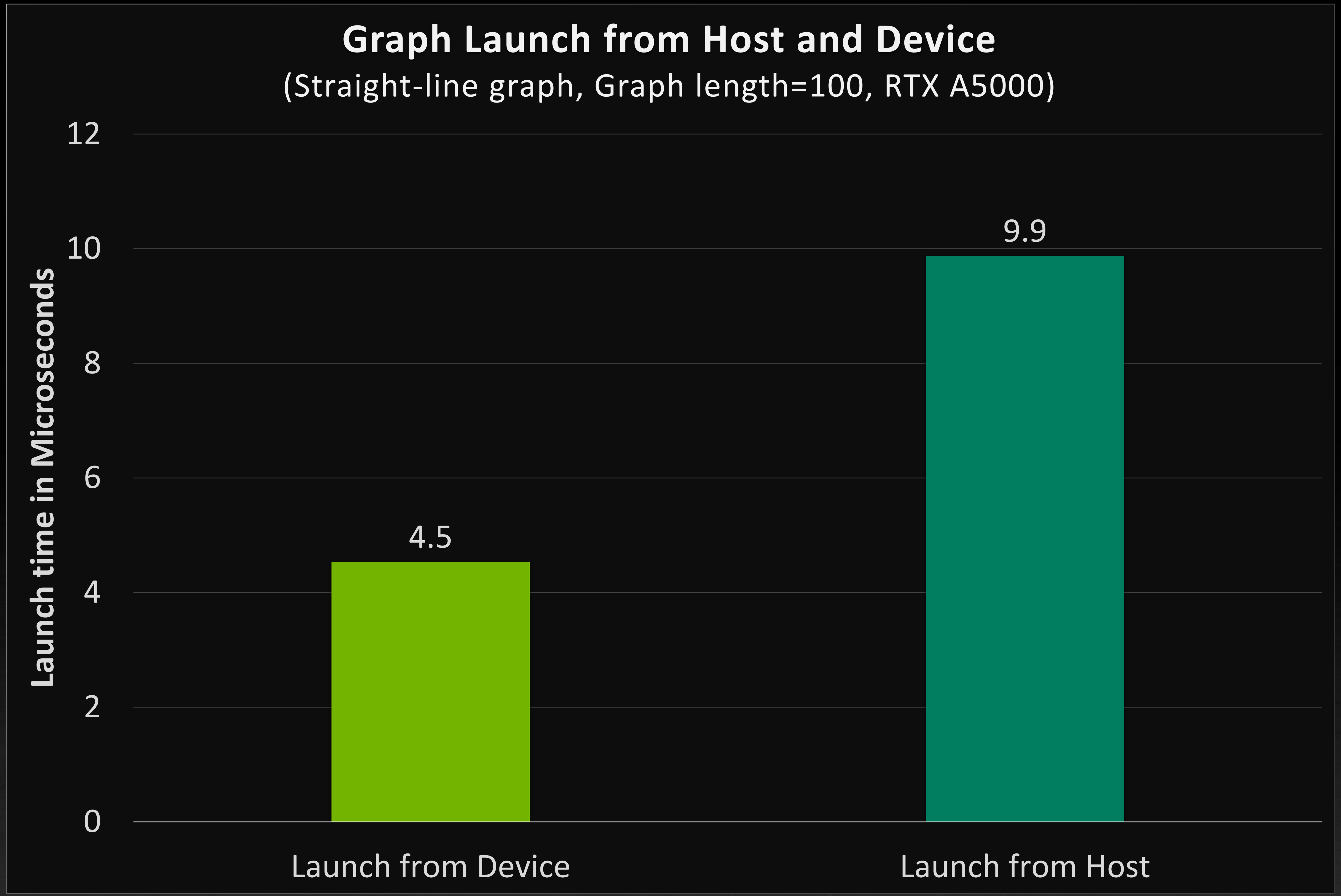
THE DEVICE-LAUNCH ADVANTAGE

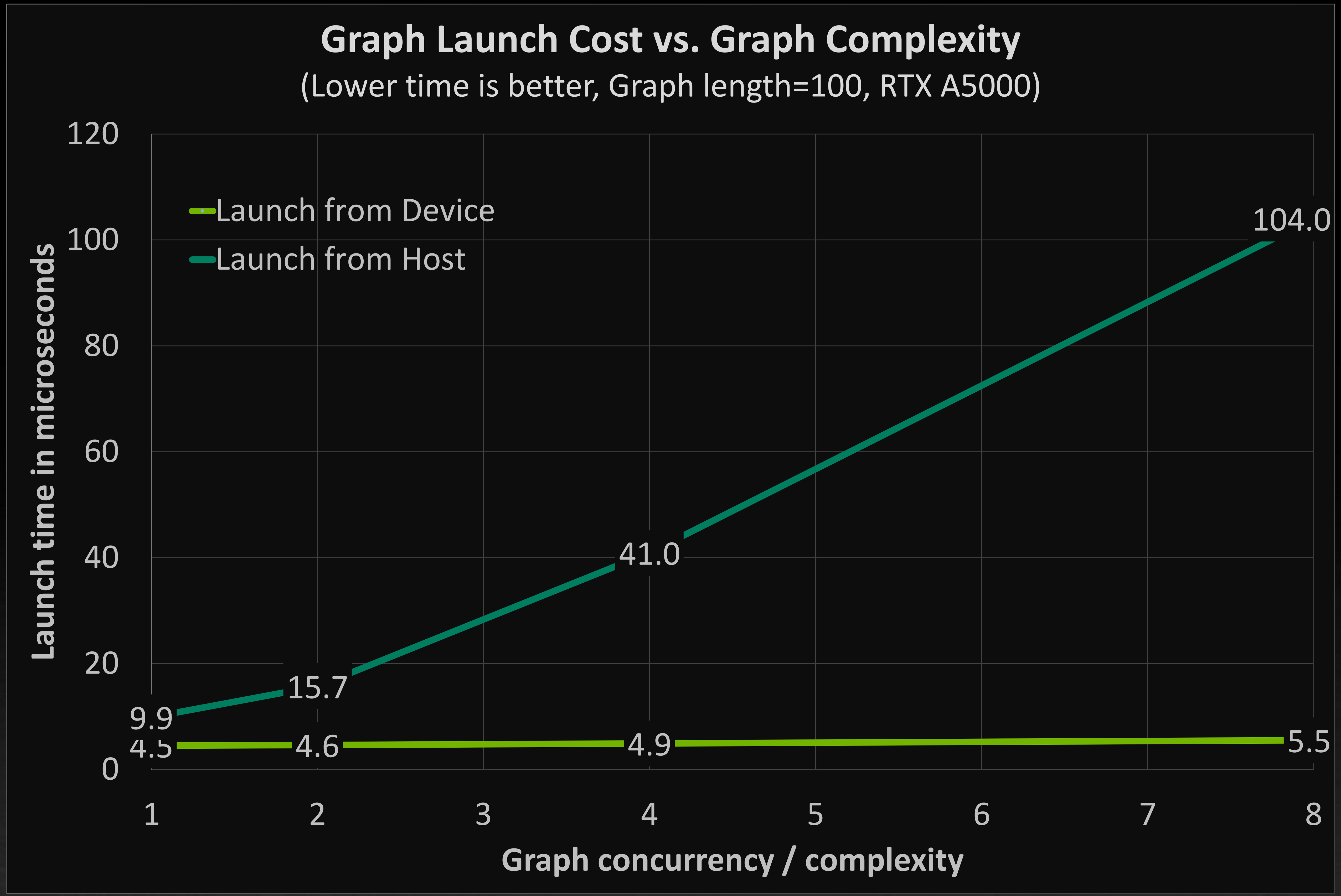



```
__global__ void scheduler(...) {
    Packet data = receivePacket(...);

    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

THE DEVICE-LAUNCH ADVANTAGE





TEMPORARY LIMITATIONS

To be removed in the future; not necessarily in the order listed here

1. You can only launch a graph from another graph; a kernel launched via <<<>>> cannot launch a graph in CUDA 12.0
2. You cannot launch the same graph twice without relaunching the parent; current design is focused on scheduler pattern
3. “Sibling” launch is not yet supported
4. Memory nodes are not yet supported in device graphs

NVIDIA