Performance Optimization Tutorial, Part 3 [S72686]:

CUDA Techniques to Maximize Concurrency and System Utilization

Myrto Papadopoulou (NVIDIA DevTech Compute)
Igor Terentyev (NVIDIA DevTech Compute)
Guillaume Thomas-Collignon (NVIDIA DevTech Compute)
***

GPU Technology Conference | March 18th, 2025

*** With help of: Akshay Subramaniam, Allard Hendriksen, Athena Elafrou, Ben Pinzone, David Clark

Performance Optimization Tutorials at GTC'25

List of presentations

- CUDA Techniques to Maximize Memory Bandwidth and Hide Latency [S72683]
- CUDA Techniques to Maximize Compute and Instruction Throughput [S72685]
- CUDA Techniques to Maximize Concurrency and System Utilization [S72686]
- CUDA Techniques to Maximize Application Performance on Grace – Hopper/Blackwell [S72687]

Agenda

- CUDA streams
- Programmatic Dependent Launch
- CUDA Graphs
- MIG, MPS, and Green Contexts
- Cluster Launch Control

Nomenclature

CTA (Cooperative Thread Array) == Thread Block
CGA (Cooperative Grid Array) == Thread Block Cluster

High-priority kernel == kernel associated with a high priority stream

Code snippets:

```
namespace cg = cooperative_groups;
using namespace cuda; // ~~cuda::~~ptx::*
```

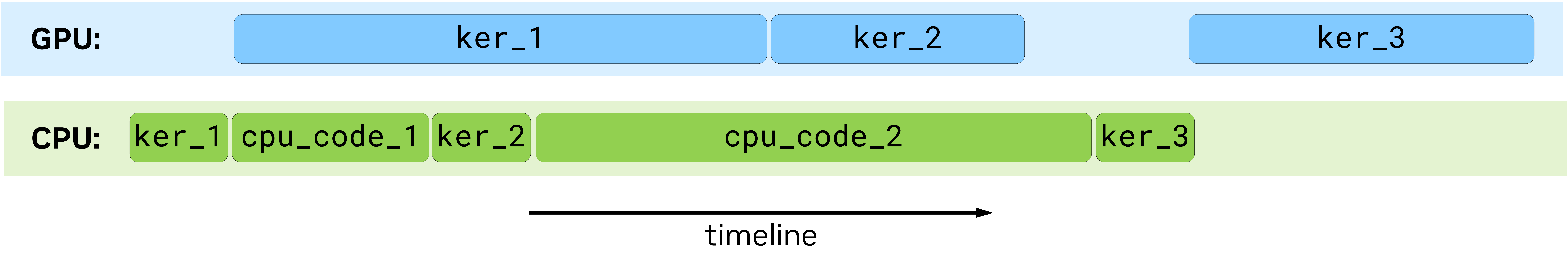
CUDA Streams

CUDA Streams

Asynchronous execution on GPU

CUDA/GPU tasks (such as kernels, async memory operations, host callbacks, etc.) execute asynchronously with CPU:

```
ker_1<<<grid_size, block_size>>>();
cpu_code_1();
ker_2<<<grid_size, block_size>>>();
cpu_code_2();
ker_3<<<grid_size, block_size>>>();
```



CUDA Streams

Asynchronous execution on GPU

CUDA/GPU tasks (such as kernels, memory operations, host call backs, etc.) execute asynchronously with CPU :

```
ker_1<<<grid_size, block_size, 0, stream1>>>();
cpu_code_1();
ker_A<<<grid_size, block_size, 0, stream2>>>();
ker_B<<<grid_size, block_size, 0, stream2>>>();
ker_2<<<grid_size, block_size, 0, stream1>>>();
cpu_code_2();
ker_3<<<grid_size, block_size, 0, stream1>>>();
```

And can execute in parallel with respect to each other:

GPU:
ker_A
ker_B
ker_1
ker_2
ker_3
CPU:
ker_1
cpu_code_1
ker_A
ker_B
ker_2
cpu_code_2
ker_3
timeline

CUDA Streams

Introduction

CUDA Stream – GPU analogue of a CPU thread:

- On creation, stream is associated with the GPU that is active (e.g., by `cudaSetDevice`).
- Stream's tasks execute in the order they are submitted by the CPU.
- Next task starts executing after the previous task has finished (exception – kernel overlap via PDL#).

Different streams may execute tasks out of order and concurrently with respect to each other.

Synchronization:

- CPU can synchronize a stream – wait for all preceding stream tasks to complete.
- Streams can synchronize with respect to each other – next stream task will not start until a certain task from another stream has completed.

# Programmatic Dependent Launch – covered later in this presentation

CUDA Streams

Default stream

Different streams may execute tasks out of order and concurrently with respect to each other.

Default-stream (`0`) is special:

```
kernel<<<grid_size, block_size>>> == kernel<<<grid_size, block_size, 0, 0>>>
```

- Implicitly created (for each context).
- Does not overlap operations with other streams (created with default flags).

  Example (kernels will not overlap):

```
kernel_A<<<grid_size, block_size, 0, stream_A>>>();
kernel_B<<<grid_size, block_size>>>();
kernel_C<<<grid_size, block_size, 0, stream_C>>>();
```

Removing implicit synchronization:

- Compiler option for async default-stream behavior: `nvcc --default-stream per-thread ...`
  `nvcc -DCUDA_API_PER_THREAD_DEFAULT_STREAM=1 ...`
- Do not sync with the default-stream: `cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);`

Recommendations:

- Avoid using the default stream
- Create explicit streams with `cudaStreamNonBlocking`

CUDA Streams

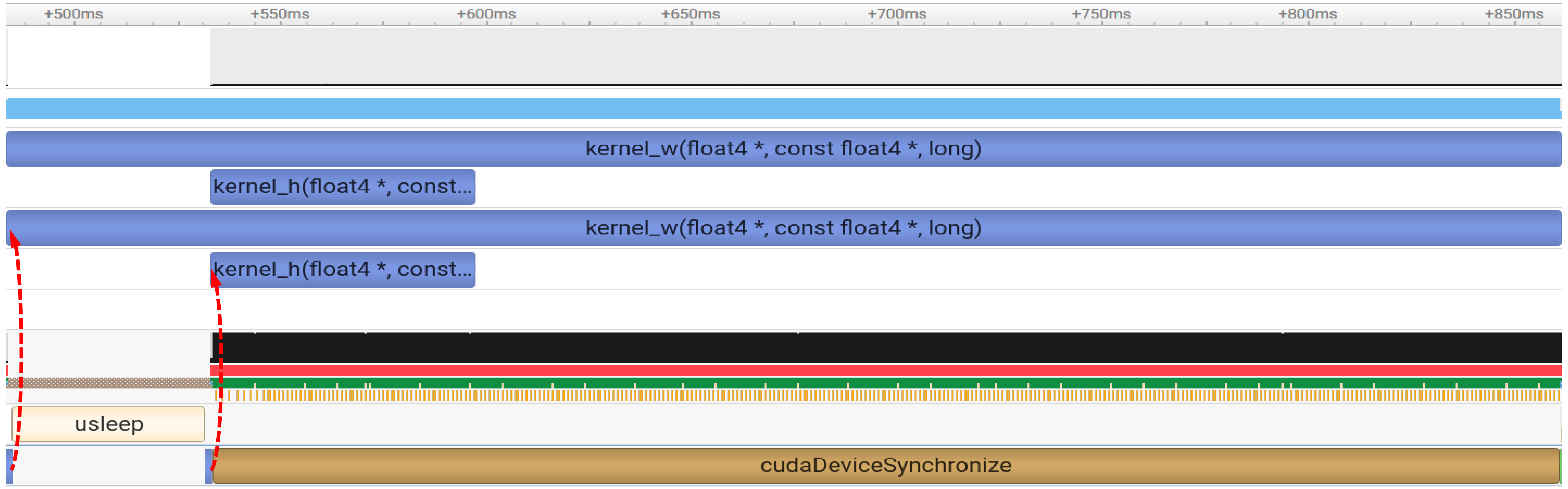
Stream priorities

Streams can have priorities:

- Supported range: `cudaDeviceGetStreamPriorityRange(...)`
- Stream with priority: `cudaStreamCreateWithPriority(...)`

Priority gives a **hint** for scheduling tasks in a stream!

E.g., higher-priority CTAs will run as already-running lower-priority CTAs finish,
remaining lower-priority CTAs will run after higher-priority CTAs finish.


+500ms
+550ms
+600ms
+650ms
+700ms
+750ms
+800ms
+850ms
kernel_w(float4 *, const float4 *, long)
kernel_h(float4 *, const...
kernel_w(float4 *, const float4 *, long)
kernel_h(float4 *, const...
usleep
cudaDeviceSynchronize

Launch `kernel_w` into lower-priority stream

Launch `kernel_h` into higher-priority stream

CUDA Streams

Stream priorities

Upcoming Nsight Systems – can now see stream priority (hover over stream).

Timeline View
Options...
1x
1 warning, 19 messages
0s
+414ms
+414.2ms
+414.4ms
+414.6ms
+414.8ms
+415ms
+415.2ms
+415.4ms
+415.6ms
+415.8ms
+416ms
+416.2ms
+416.4ms
+416.6ms
+416.8ms
+417ms
+417.2ms
+417.4m
CPU (48)
0 to 100%
CUDA HW (0000:45:00.0 - N
Kernel
Memory
[All Streams]
54.2% Stream 14
45.8% Stream 13
<0.1% De
m 7
Priority: 0
Threads (9)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
0 to 100%
☑ [47970] gtc-device-sy
OS runtime libraries
usleep
usleep
NVTX
CUDA API
cudaDeviceSynchronize
Profiler overhead
0 to 100%
☑ [47989] libc.so.6!__GI

CUDA Streams

GPU synchronization

Heavy synchronization:

```
    cudaDeviceSynchronize()
```

Will block until all GPU tasks (all streams) have finished!

504.85ms
+604.9ms
+604.95ms
+605ms
+605.05ms
+605.1ms
+605.15ms
+605.2ms
+605.25ms
+605.3ms
+605.35ms
+605.4ms
+605.45ms
+605.5ms
+605.55ms
+605.6ms
+605.65ms
+60
kernel(float4 *, const float4 *, lon...
kernel(float4 *, const float4 *, lo...
kernel(float4 *, const float4 *, lon...
kernel(float4 *, const float4 *, lon...
kernel(float4 *, const f
ke...
...
cudaDeviceSynchronize
...

Launch in advance =>
no launch latency gap

Cannot launch in advance =>
launch latency gap

CUDA Streams

GPU synchronization

Heavy synchronization:

`cudaDeviceSynchronize()`

Often-encountered “bad” practice: `cudaDeviceSynchronize()` after each kernel launch (only to check errors or timing)

604.85ms
+604.9ms
+604.95ms
+605ms
+605.05ms
+605.1ms
+605.15ms
+605.2ms
+605.25ms
+605.3ms
+605.35ms
+605.4ms
+605.45ms
+605.5ms
+605.55ms
+605.6ms
+605.65ms
+60
kernel(float4 *, const float4 *, lon...
kernel(float4 *, const float4 *, lo...
kernel(float4 *, const float4 *, lon...
kernel(float4 *, const float4 *, lon...
kernel(float4 *, const f
ke...
...
cudaDeviceSynchronize
...

Launch in advance =>
no launch latency gap

Cannot launch in advance =>
launch latency gap

CUDA Streams

GPU error checking and timing

Checking kernel errors :

- **`Kernel<<<...>>>` launch error:**
  - Due to invalid launch configuration.
  - Not set if kernel fails during execution.
  - `cudaGetLastError()` [resets error], `cudaPeekAtLastError()` [does not reset error].
  - Set by APIs that return execution error (`cudaDeviceSynchronize()`, etc.)

- **Execution error:**
  - Not reported by `cudaGetLastError()` and `cudaPeekAtLastError()` even if called after kernel failure.
  - Sticky.
  - Reported by `cudaDeviceSynchronize()`, `cudaStreamSynchronize(...)`, `cudaEventSynchronize(...)`.

CUDA Streams

GPU error checking and timing

Checking kernel errors :

- **`Kernel<<<...>>>` launch error:**
  - Due to invalid launch configuration.
  - Not set if kernel fails during execution.
  - `cudaGetLastError()` [resets error], `cudaPeekAtLastError()` [does not reset error].
  - Set by APIs that return execution error (`cudaDeviceSynchronize()`, etc.)

- **Execution error:**
  - Not reported by `cudaGetLastError()` and `cudaPeekAtLastError()` even if called after kernel failure.
  - Sticky.
  - Reported by `cudaDeviceSynchronize()`, `cudaStreamSynchronize(...)`, `cudaEventSynchronize(...)`.

```
kernel<<<...>>>();         // Launch ok, execution error
// CPU work here so execution error occurs before
// next line
cudaGetLastError();        // No error reported
cudaDeviceSynchronize();   // Error reported
cudaDeviceSynchronize();   // Error reported
```

```
kernel<<<...>>>();         // Launch ok, execution error


cudaDeviceSynchronize();   // Error reported
cudaGetLastError();        // Error reported
cudaGetLastError();        // No error reported
```

CUDA Streams

GPU error checking and timing

Checking kernel errors :

- **`Kernel<<<...>>>` launch error:**
  - Due to invalid launch configuration.
  - Not set if kernel fails during execution.
  - `cudaGetLastError()` [resets error], `cudaPeekAtLastError()` [does not reset error].
  - Set by APIs that return execution error (`cudaDeviceSynchronize()`, etc.)

- **Execution error:**
  - Not reported by `cudaGetLastError()` and `cudaPeekAtLastError()` even if called after kernel failure.
  - Sticky.
  - Reported by `cudaDeviceSynchronize()`, `cudaStreamSynchronize(...)`, `cudaEventSynchronize(...)`.

Good practice:

- Call `cudaGetLastError()` / `cudaPeekAtLastError()` after each `kernel<<<...>>>()`.
- Check the return value of every CUDA API call.

CUDA Streams

GPU error checking and timing

Checking kernel errors :

- **`Kernel<<<...>>>` launch error**:
  - Due to invalid launch configuration.
  - Not set if kernel fails during execution.
  - `cudaGetLastError()` [resets error], `cudaPeekAtLastError()` [does not reset error].
  - Set by APIs that return execution error (`cudaDeviceSynchronize()`, etc.)

- **Execution error:**
  - Not reported by `cudaGetLastError()` and `cudaPeekAtLastError()` even if called after kernel failure.
  - Sticky.
  - Reported by `cudaDeviceSynchronize()`, `cudaStreamSynchronize(...)`, `cudaEventSynchronize(...)`.

Timing:

- Nsight Systems (Nsight Compute).
- Via CUDA events.

CUDA Streams

GPU synchronization

Some other API calls are fully blocking & synchronous – as if surrounded by `cudaDeviceSynchronize()`:

```
cudaFree(...)
```

-771.4ms
+771.6ms
+771.8ms
+772ms
+772.2ms
+772.4ms
+772.6ms
+772.8ms
+773ms
+773.2ms
+773.4ms
+773.6ms
kernel(float4 *, const float4 *, long)
kernel(float4 *,
kernel(float4 *, const float4 *, long)
kernel(float4 *, c
ker...
cudaFree

CUDA Streams

GPU synchronization

Some other API calls can be fully blocking & synchronous – as if surrounded by `cudaDeviceSynchronize()`:

- `cudaMemcpy(..., {cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost})`#
- some other API functions like `cudaDeviceSetCacheConfig(...)`

+798.2ms
+798.4ms
+798.6ms
+798.8ms
+799ms
+799.2ms
+799.4ms
+799.6ms
+799.8ms
+800ms
+800.2ms
+800.4ms
+800.6ms
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
ke...
cudaMemcpy

# Not a reference: **behavior may vary** depending on HW, pageable/pinned memory, size.

CUDA Streams

GPU synchronization

Non-blocking & synchronous – default-stream kernel behavior:

```
cudaMemcpy(..., cudaMemcpyDeviceToDevice)
cudaMemset(...)
```

+754.5ms
+754.6ms
+754.7ms
+754.8ms
+754.9ms
+755ms
+755.1ms
+755.2ms
+755.3ms
+755.4ms
+755.5ms
+755.6ms
+755.7ms
+755.8ms
+755.9ms
+756ms
+756.1ms
+756.2ms
+756.3ms
+756.4ms
+756.5ms
+756.6ms
+756.7ms
+756.8ms
+756.9
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
ke...
...

CUDA Streams

GPU synchronization

Blocking & asynchronous:

```
cudaMalloc(...)
```

969.2ms
+969.3ms
+969.4ms
+969.5ms
+969.6ms
+969.7ms
+969.8ms
+969.9ms
+970ms
+970.1ms
+970.2ms
+970.3ms
+970.4ms
+970.5ms
+970.6ms
+970.7ms
+970.8ms
+970.9ms
+971ms
+971.1ms
+971.2ms
+971.3ms
+971.4ms
+971.5
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const float4 *, long)
kernel(float4 *, const floa
kernel
...

# Not a reference: **behavior may vary**.

CUDA Streams

GPU synchronization tips

Tips:

- Use Nsight Systems to verify blocking/nonblocking and synchronous/asynchronous behavior!
- Use stream synchronization and only when necessary (data dependencies).
- Avoid heavy synchronizations by using Async operations:
  `cudaMemcpyAsync`, `cudaMemsetAsync`, `cudaMallocAsync`, `cudaFreeAsync`
  - Take stream argument and follow stream semantics.
  - Can be synchronous. E.g, `cudaMemcpyAsync` with pageable host memory.
- CPU tasks with stream semantics: `cudaLaunchHostFunc(stream, host_fn, data_ptr)`

CUDA Streams

Batched asynchronous memcpy

Commonly used cudaMemcpy/cudaMemcpyAsync:

- Asynchronous versions can synchronize, and synchronous versions can behave asynchronously…
- Many small copies – launch latency overheads dominate…
- Not best performance for system-allocated memory on coherent (GH/GB) systems…

New API (CUDA 12.8) for batched copies:

```
cudaMemcpyBatchAsync(void** dsts, void** srcs, size_t* sizes, size_t count,
                     cudaMemcpyAttributes* attrs, size_t* attrsIdxs, size_t numAttrs,
                     size_t* failIdx, cudaStream_t stream)
```

GH/GB: runs concurrently with pageable memory

CUDA Streams

Stream synchronization

Stream synchronization: `cudaStreamSynchronize(stream)`
blocks CPU until all previously submitted to `stream` tasks complete

GPU:
ker_A
ker_B
ker_1
ker_2
ker_3
CPU:
ker_1
ker_A
ker_B
ker_2
cudaStreamSynchronize
ker_3
timeline

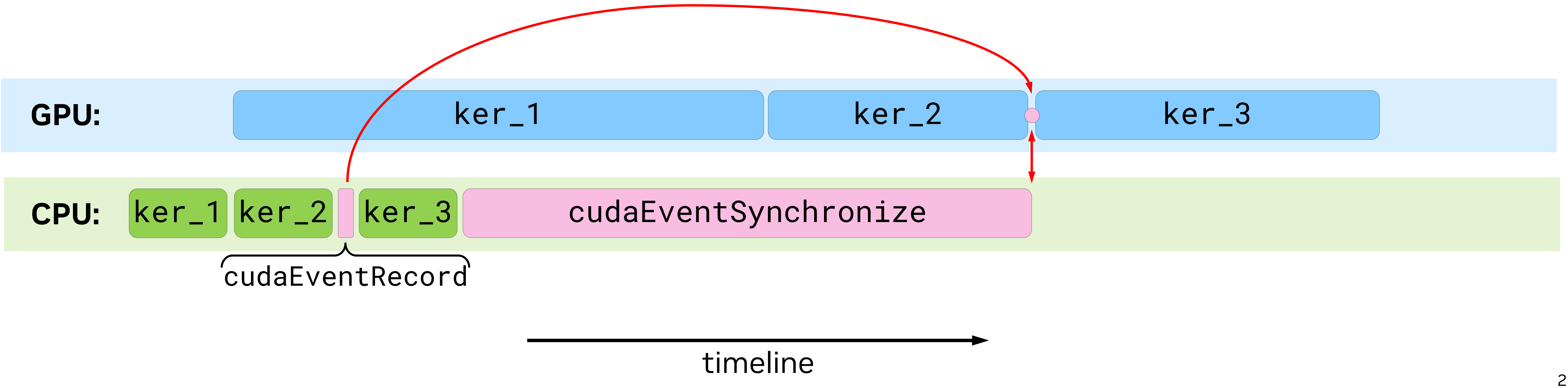
CUDA Streams

Events

Events – “checkpoints” within streams

- Complete when all preceding tasks in stream finish.
- Reuseable – new `cudaEventRecord(...)` resets completion status.
- Event synchronization:
  - `cudaEventSynchronize(...)` – block till completion.
  - `cudaEventQuery(...)` – non-blocking completion check.

- Can have completion timestamps:
  `cudaEventElapsedTime(start_event, stop_event)` – compute time between two completed events



CUDA Streams

Events

Synchronization is required due to data dependencies…

Most common case:

task in `stream_1` consumes data produced by task in `stream_2`

(task in `stream_1` produces data after task in `stream_2` finished consuming)

`cudaStreamWaitEvent(stream, event_from_other_stream)`:

- Synchronization between streams “bypassing CPU”.
- Non-blocking call.
- Can sync streams from different devices!

GPU:
ker_1
ker_2
ker_3
CPU:
Ker_1
Ker_2
ker_3
cudaEventRecord
cudaStreamWaitEvent
timeline

CUDA Streams

Example – wavelet transform

Output: 2D CPU array, each column computed independently
Input: 1D CPU array
fast
coord

CUDA Streams

Example – wavelet transform

Output: 2D CPU array, each column computed independently

Input: 1D CPU array

fast coord

Step 1: copy to GPU

CUDA Streams

Example – wavelet transform

Output: 2D CPU array, each column computed independently
Input: 1D CPU array
fast coord
Step 1: copy to GPU
Step 2: compute column 1

CUDA Streams

Example – wavelet transform

Output: 2D CPU array, each column computed independently
Input: 1D CPU array
fast coord
Step 1: copy to GPU
Step 2: compute column 1
Step 3: copy to CPU column 1

CUDA Streams

Example – wavelet transform

Output: 2D CPU array, each column computed independently
Input: 1D CPU array
fast coord
Step 1: copy to GPU
Step 2: compute column 2

CUDA Streams

Example – wavelet transform

Output: 2D CPU array, each column computed independently
Input: 1D CPU array
Loop over columns
fast coord
Step 1: copy to GPU
Step 2: compute column 2
Step 3: copy to CPU column 2

CUDA Streams

Example – wavelet transform

Goal: overlap compute and copy

Compute col. 0
Compute col. 1
...
Compute col. i+1
...
Compute col. N-1
Copy col. 0
Copy col. i
Copy col. N-2
Copy col. N-1

CUDA Streams

Example – wavelet transform

X86: copy through pinned buffer

GH/GB: no need in pinned buffer if `cudaMemcpyBatchAsync(...)`

Output: 2D CPU array, each column computed independently
Input: 1D CPU array
fast coord
Step 1: copy to GPU
Step 2: compute column 2
Step 3a: copy to pinned
Step 3b: copy H2H

CUDA Streams

Example – wavelet transform

Even column
Output: 2D CPU array, each column computed independently
Input: 1D CPU array
fast coord
Step 1: copy to GPU
Step 2: compute column 2
Step 3a: copy to pinned
Step 3b: copy H2H

CUDA Streams

Example – wavelet transform

Odd column
Output: 2D CPU array, each column computed independently
Input: 1D CPU array
fast coord
Step 1: copy to GPU
Step 2: compute column 2
Step 3a: copy to pinned
Step 3b: copy H2H

CUDA Streams

Example – wavelet transform

NVIDIA Nsight Systems
File View Tools Help
h200-dev-pinned-kernel2d.nsys-rep
Timeline View
Options...
2x
14 warnings, 31 messages
6s
+840ms +845ms +850ms +855ms +860ms +865ms +870ms +875ms +880ms +885ms +890ms +895ms +900ms +905ms +910ms +915ms +920ms +925ms +930ms +935ms +940ms
CPU (48)
100%
0
Processes (6)
[8281] python3
92.5% Stream 14
Memcpy DtoH
7.5% Stream 13
void <unna... void vecto... v...
[8281] python3
[8366] python3
OS runtime libraries
sem_wait
sem_w...
NVTX
memcpy_h2h [9.3...
memcpy_h2h [9.5...
memcpy_h2h [9.6...

CUDA Streams

Example – wavelet transform

NVIDIA Nsight Systems
File View Tools Help
h200-dev-pinned-kernel2d.nsys-rep
Timeline View
Options...
2x
14 warnings, 31 messages
6s
+840ms +845ms +850ms +855ms +860ms +865ms +870ms +875ms +880ms +885ms +890ms +895ms +900ms +905ms +910ms +915ms +920ms +925ms +930ms +935ms +940ms
CPU (48)
Processes (6)
[8281] python3
92.5% Stream 14
7.5% Stream 13
[8281] python3
[8366] python3
OS runtime libraries
NVTX
100%
0
Memcpy DtoH
void <unna...
void vecto...
v...
sem_wait
sem_w...
memcpy_h2h [9.3...
memcpy_h2h [9.5...
memcpy_h2h [9.6...

One output column

Asynchronous CPU tasks typically implemented via another CPU thread:

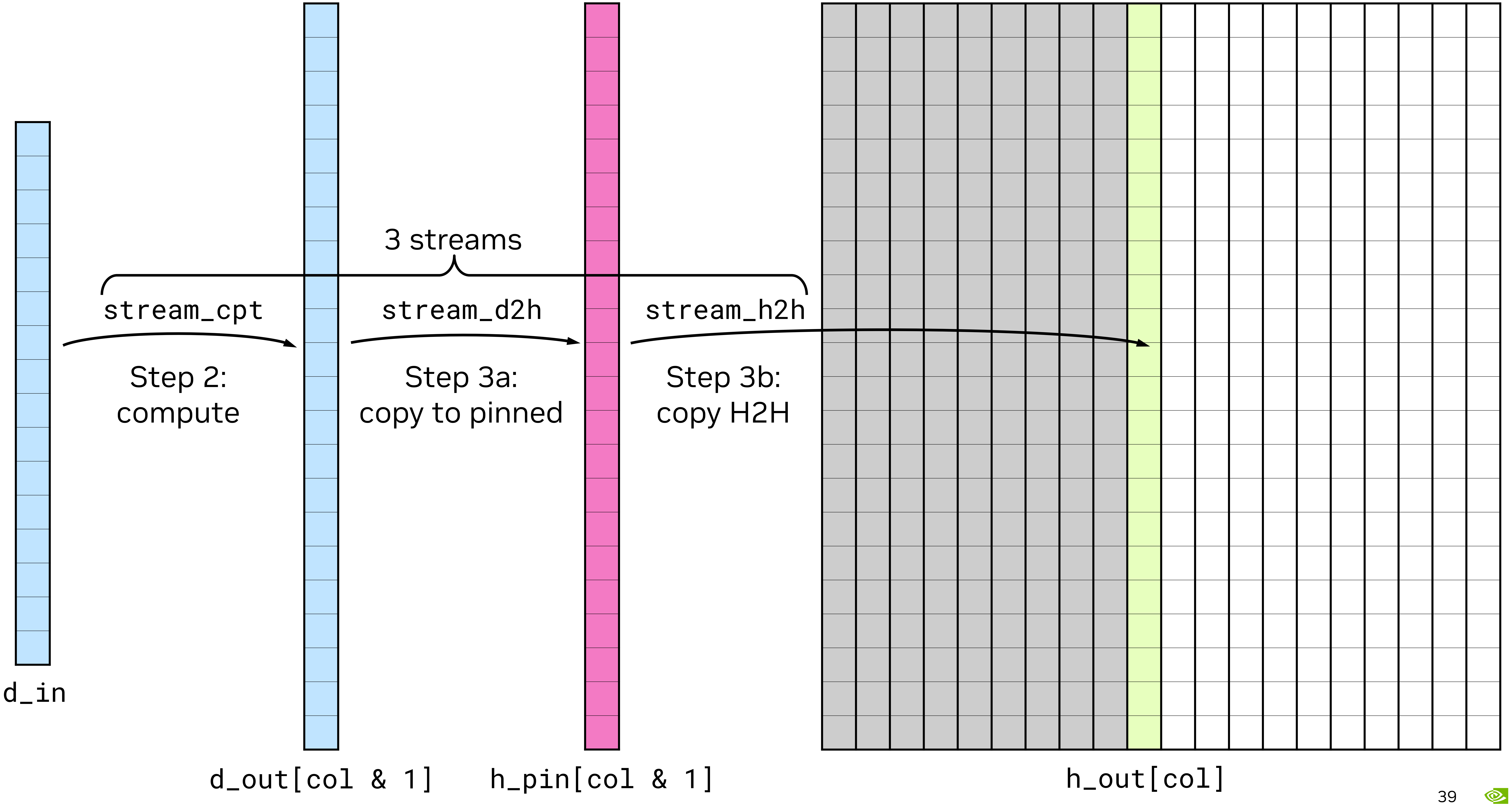
Explicit multithreaded code, synchronization (mutex), etc.

Another approach:

CUDA streams and `cudaLaunchHostFunc(...)`

CUDA Streams

Example – wavelet transform



CUDA Streams

Example – wavelet transform

```
for (int col = 0; col < ncol; ++col) {

    kernel<<<..., stream_cpt>>>(d_out[col & 1], d_in, ...);



    cudaMemcpyAsync(h_pin[col & 1], d_out[col & 1], ..., stream_d2h);


    cudaLaunchHostFunc(stream_h2h, fn_h2h, &pars_h2h);

}
```

CUDA Streams

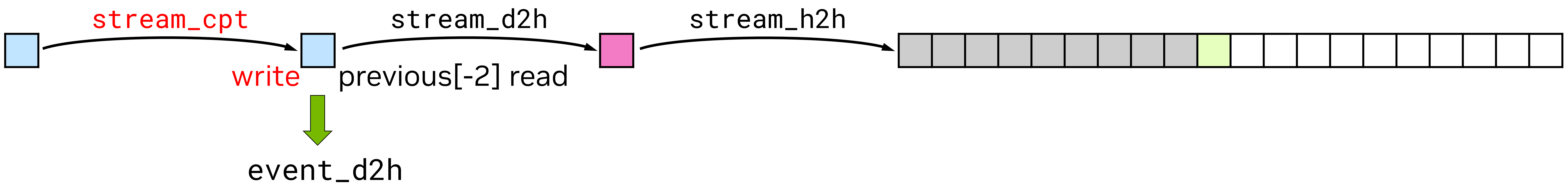
Example – wavelet transform

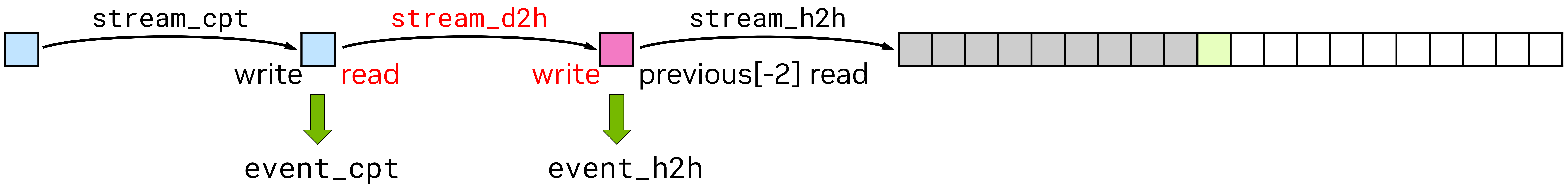
```
Pars_h2h pars_h2h = {h_pin, 0, ...};

for (int col = 0; col < ncol; ++col) {

    kernel<<<..., stream_cpt>>>(d_out[col & 1], d_in, ...);



    cudaMemcpyAsync(h_pin[col & 1], d_out[col & 1], ..., stream_d2h);



    cudaLaunchHostFunc(stream_h2h, fn_h2h, &pars_h2h);

}
```

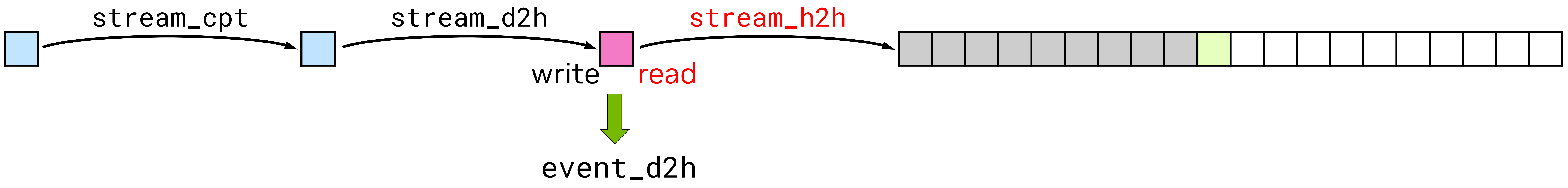
```
struct Pars_h2h
{
    char** h_pin;
    int    col;

    char*  h_out;
    size_t count;
};

void fn_h2h (void* pars)
{
    auto p = (Pars_h2h*)pars;

    char** h_pin = p->h_pin;
    int&   col   = p->col;
    char*  h_out = p->h_out;
    size_t count = p->count;

    std::memcpy(h_out + col * count,
                h_pin[col & 1], count);
    ++col;
}
```

CUDA Streams

Example – wavelet transform







CUDA Streams

Example – wavelet transform

```
Pars_h2h pars_h2h = {h_pin, 0, ...};

for (int col = 0; col < ncol; ++col) {

    kernel<<<..., stream_cpt>>>(d_out[col & 1], d_in, ...);
    cudaEventRecord(event_cpt, stream_cpt);


    cudaMemcpyAsync(h_pin[col & 1], d_out[col & 1], ..., stream_d2h);
    cudaEventRecord(event_d2h[col & 1], stream_d2h);


    cudaLaunchHostFunc(stream_h2h, fn_h2h, &pars_h2h);
    cudaEventRecord(event_h2h[col & 1], stream_h2h);
}
```

```
struct Pars_h2h
{
    char** h_pin;
    int    col;

    char*  h_out;
    size_t count;
};

void fn_h2h (void* pars)
{
    auto p = (Pars_h2h*)pars;

    char** h_pin = p->h_pin;
    int&   col   = p->col;
    char*  h_out = p->h_out;
    size_t count = p->count;

    std::memcpy(h_out + col * count,
                h_pin[col & 1], count);
    ++col;
}
```

CUDA Streams

Example – wavelet transform

```
Pars_h2h pars_h2h = {h_pin, 0, ...};

for (int col = 0; col < ncol; ++col) {
    cudaStreamWaitEvent(stream_cpt, event_d2h[col & 1]);

    kernel<<<..., stream_cpt>>>(d_out[col & 1], d_in, ...);
    cudaEventRecord(event_cpt, stream_cpt);

    cudaStreamWaitEvent(stream_d2h, event_cpt);
    cudaStreamWaitEvent(stream_d2h, event_h2h[col & 1]);

    cudaMemcpyAsync(h_pin[col & 1], d_out[col & 1], ..., stream_d2h);
    cudaEventRecord(event_d2h[col & 1], stream_d2h);

    cudaStreamWaitEvent(stream_h2h, event_d2h[col & 1]);

    cudaLaunchHostFunc(stream_h2h, fn_h2h, &pars_h2h);
    cudaEventRecord(event_h2h[col & 1], stream_h2h);
}

cudaStreamSynchronize(stream_h2h);
```

```
struct Pars_h2h
{
    char** h_pin;
    int    col;

    char*  h_out;
    size_t count;
};

void fn_h2h (void* pars)
{
    auto p = (Pars_h2h*)pars;

    char** h_pin = p->h_pin;
    int&   col   = p->col;
    char*  h_out = p->h_out;
    size_t count = p->count;

    std::memcpy(h_out + col * count,
                h_pin[col & 1], count);
    ++col;
}
```

CUDA Streams

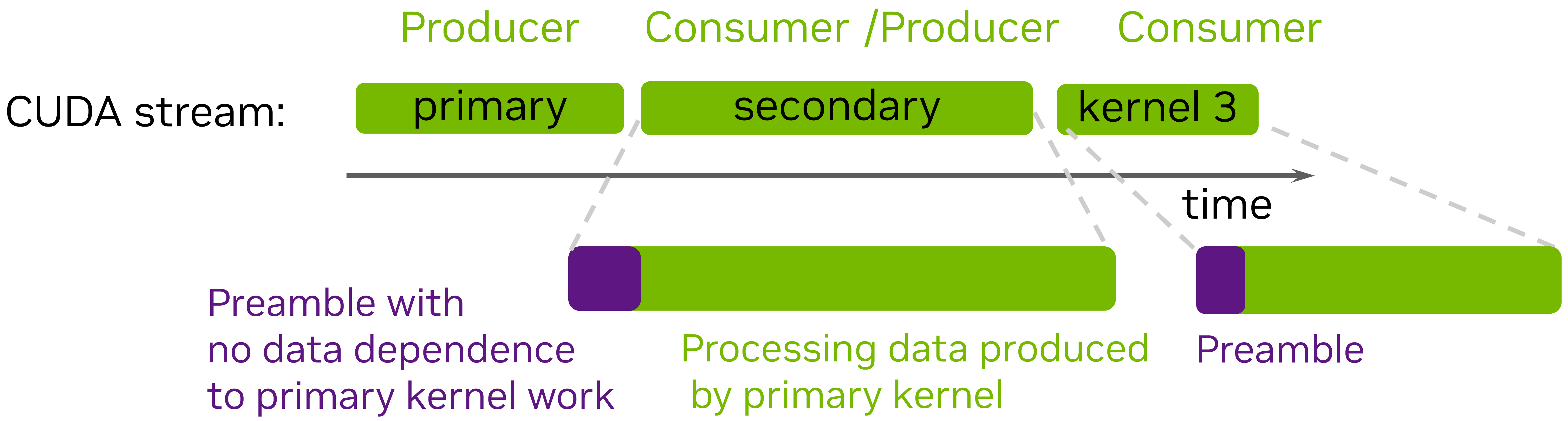
Tips summary

- Avoid heavy synchronization (e.g., `cudaDeviceSynchronize`).
- Create streams with `cudaStreamNonBlocking` flag and/or avoid default stream.
- Minimize synchronization (use for data dependencies).
- Use async APIs.
- Use `cudaMemcpyBatchAsync`.
- Create events with `cudaEventDisableTiming` flag for better performance.
- Environment variables:
  - `CUDA_DEVICE_MAX_CONNECTIONS` – compute and copy engine concurrent connections limit
  - `CUDA_DEVICE_MAX_COPY_CONNECTIONS` – copy engine concurrent connections limit
  - CUDA_SCALE_LAUNCH_QUEUES – launch queue size factor

Programmatic Dependent Launch (PDL)

Programmatic Dependent Launch (PDL)

Motivation



Producer

Consumer /Producer

Consumer

CUDA stream:

primary

secondary

kernel 3

time

Preamble with
no data dependence
to primary kernel work

Processing data produced
by primary kernel

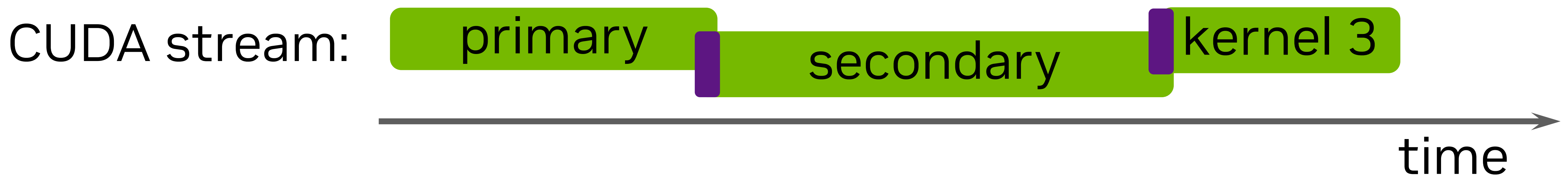
Preamble

Stream semantics guarantee
sequential execution

Data dependences across kernels
usually implied

e.g., shared memory initialization, pointer arithmetic, other setup work, reading read-only data from global memory

Programmatic Dependent Launch enables opportunistic overlap of consumer kernel’s preamble with producer kernel



CUDA stream:

primary

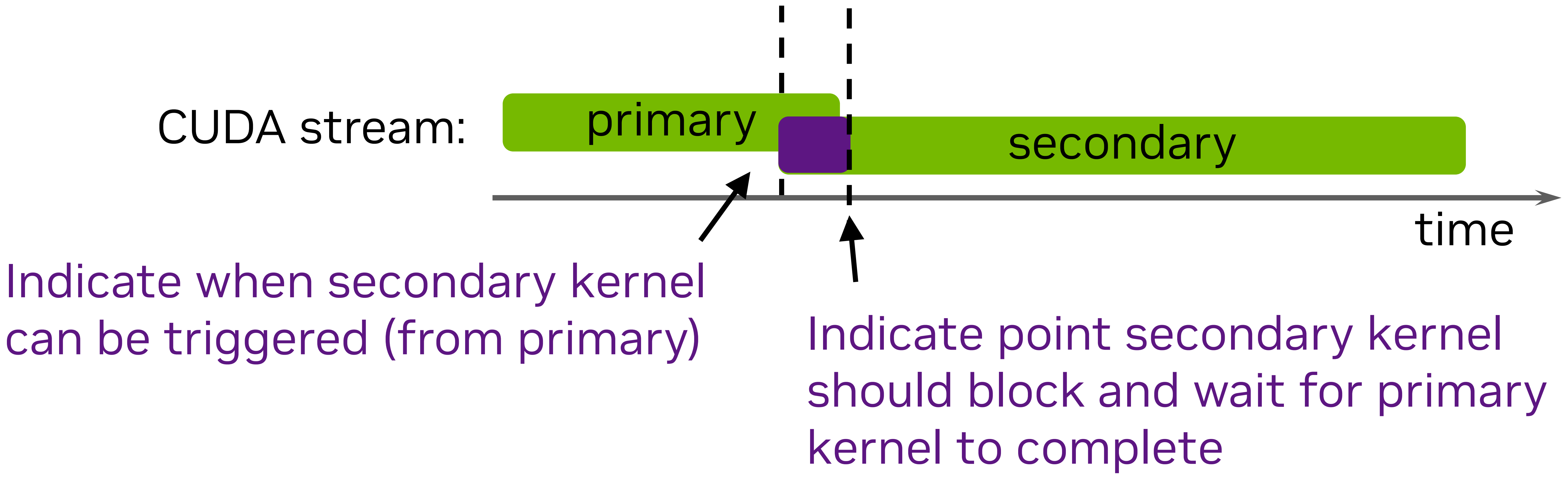
secondary

kernel 3

time

Programmatic Dependent Launch

Usage (CUDA streams)


CUDA stream:
primary
secondary
time
Indicate when secondary kernel can be triggered (from primary)
Indicate point secondary kernel should block and wait for primary kernel to complete

Device APIs for PDL (CC >= 9.0):

- cudaTriggerProgrammaticLaunchCompletion
  - Where: in the primary (producer) kernel

- cudaGridDependencySynchronize
  - Where: in the secondary (consumer) kernel

- Secondary (consumer) kernel has to be launched via cudaLaunchKernelEx
- No GPU work can exist on that same stream between primary and secondary, e.g., no CUDA event record

Programmatic Dependent Launch

Usage – Kernel Launch CPU side

Kernel signatures:

```
__global__ void primary_kernel(uint8_t* d_ptr);
__global__ void secondary_kernel(uint8_t* d_ptr);
```

Code to launch two kernels:

```
primary_kernel<<<grid_dim, block_dim, 0, strm>>>(d_ptr);
```

← Primary kernel's launch is unchanged

```
cudaLaunchConfig_t launch_cfg;
```

Secondary's kernel launch needs to use cudaLauncKernelEx*

```
launch_cfg.blockDim = dim3(threads);
launch_cfg.gridDim  = dim3(blocks);
launch_cfg.dynamicSmemBytes = 0;
launch_cfg.stream =  strm
```

Kernel's usual launch parameters

```
launch_cgf.numAttrs = 1;
cudaLaunchAttribute attrs[1];
attrs[0].id = cudaLaunchAttributeProgrammaticStreamSerialization;
attrs[0].val.programmaticStreamSerializationAllowed = 1;
```

Special attribute needed for PDL

```
cudaLaunchKernelEx(&launch_cfg, secondary_kernel, d_ptr);
```

kernel's arg.

Programmatic Dependent Launch

Usage – PDL APIs in Device Code

```
__global__ void primary_kernel(uint8_t* d_ptr) {
    work_A();
    cudaTriggerProgrammaticLaunchCompletion();
    work_B();
}
```

- Secondary kernel can be scheduled if every non-exited CTA from primary has called cudaTriggerProgrammaticLaunchCompletion() at least once (i.e., at least one thread per CTA)
- If no thread in a non-exited CTA calls that API, the secondary kernel can be scheduled when all warps from primary have completed.
- `cudaTriggerProgrammaticLaunchCompletion()` provides no memory visibility guarantee.

```
__global__ void secondary_kernel(uint8_t* d_ptr) {
    work_C();
    cudaGridDependencySynchronize();
    work_D();
}
```

- A thread will block at cudaGridDependencySynchronize and wait for primary kernel to complete (incl. work_B)
- Should be safe for work_C to execute in parallel with work_B
- work_C should have no data dependence to primary_kernel's work. For example:
  - should not access data modified by work_A or work_B
  - should not modify data accessed by work_A or work_B
  - Reading constant read-only data also read by work_A or B is OK
- typical work_C: local computations, shared memory initialization, reading read-only data from global memory, etc.
- work_D typically requires data produced by primary kernel

Programmatic Dependent Launch

Tips

```
__global__ void primary_kernel() {
    work_A();
    cudaTriggerProgrammaticLaunchCompletion();
    work_B();
}
```

```
__global__ void secondary_kernel(uint8_t* d_ptr) {
    work_C();
    cudaGridDependencySynchronize();
    work_D();
}
```

Any performance considerations with triggering (cudaTriggerProgrammaticLaunchCompletion) too early?

If duration(work_B) > duration(work_C), secondary kernel may wait a long time at cudaGridDependencySynchronize()

Relevant if kernels on other streams could have benefited from the SMs secondary_kernel may occupy.

What if cudaGridDependencySynchronize() is called after work_C incorrectly accesses data modified by work_B?

- Invalid use and possible race condition. Tools cannot capture such behavior, so be careful!
- Tip: call cudaGridDependencySynchronize before any global memory access in secondary_kernel for the common use case.

What if work_B and work_C are empty()?

- Almost no overlap (nanoseconds range benefit), but could still use

Guard PDL API calls with #if defined(__CUDA_ARCH__) && (__CUDA_ARCH__ >= 900)

Programmatic Dependent Launch

Usage – 3 kernels example

```
__global__ void primary_kernel() {
    …
    cudaTriggerProgrammaticLaunchCompletion();
    …
}
```

```
__global__ void secondary_kernel(uint8_t* d_ptr){
    …
    cudaGridDependencySynchronize();
    …
    cudaTriggerProgrammaticLaunchCompletion();
    …
}
```

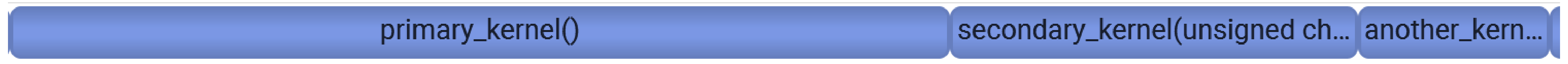
```
__global__ void another_kernel () {
    …
    cudaGridDependencySynchronize();
    ….
}
```

Secondary kernel acts both as a consumer and a producer

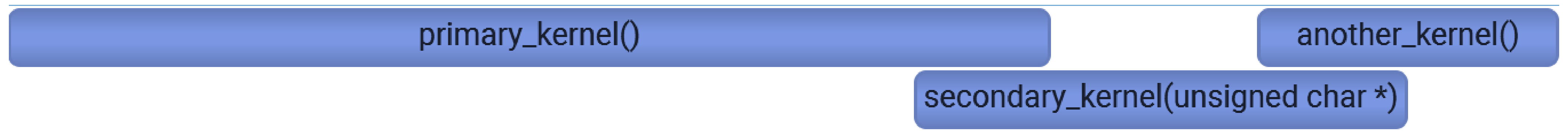
Programmatic Dependent Launch

GPU Timelines from Nsight Systems

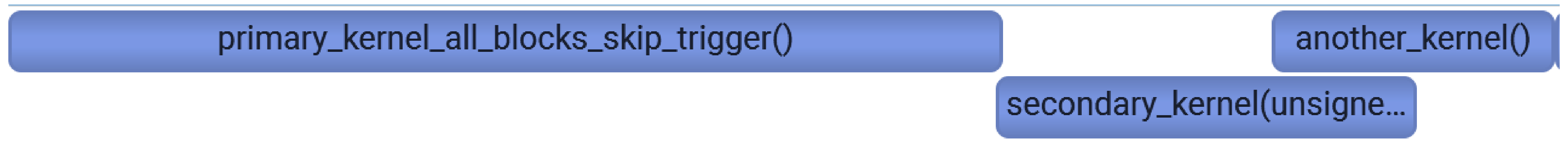
- No PDL



- PDL (all CTAs call trigger)
  - If primary is multi-wave, secondary can be scheduled only after the CTAs from the last wave called trigger



- PDL where secondary is implicitly triggered at the end
  - E.g., if no CTA from primary calls trigger


primary_kernel_all_blocks_skip_trigger()
another_kernel()
secondary_kernel(unsigne...

Code will be available here

Programmatic Dependent Launch

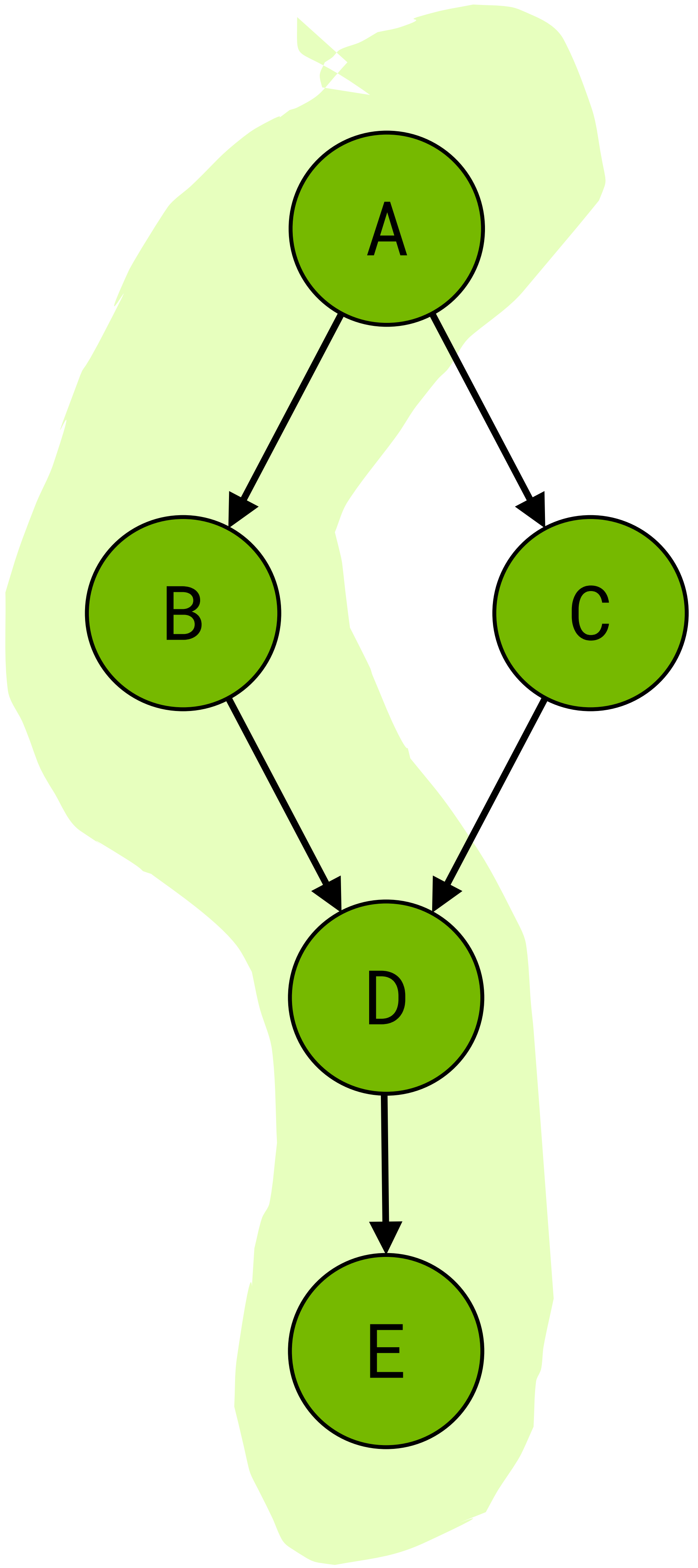
Further Reading

- Programmatic Dependent Launch in CUDA C++ Programming Guide
  - includes how to use PDL in CUDA graphs too
- CUDA runtime API reference for cudaTriggerProgrammaticLaunchCompletion and cudaGridDependencySynchronize
- Relevant ptx instructions, if using at that level: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#parallel-synchronization-and-communication-instructions-griddepcontrol
- cub’s merge sort can use PDL

CUDA Graphs

CUDA Graphs

A work submission model



Graph = operations & dependencies between operations

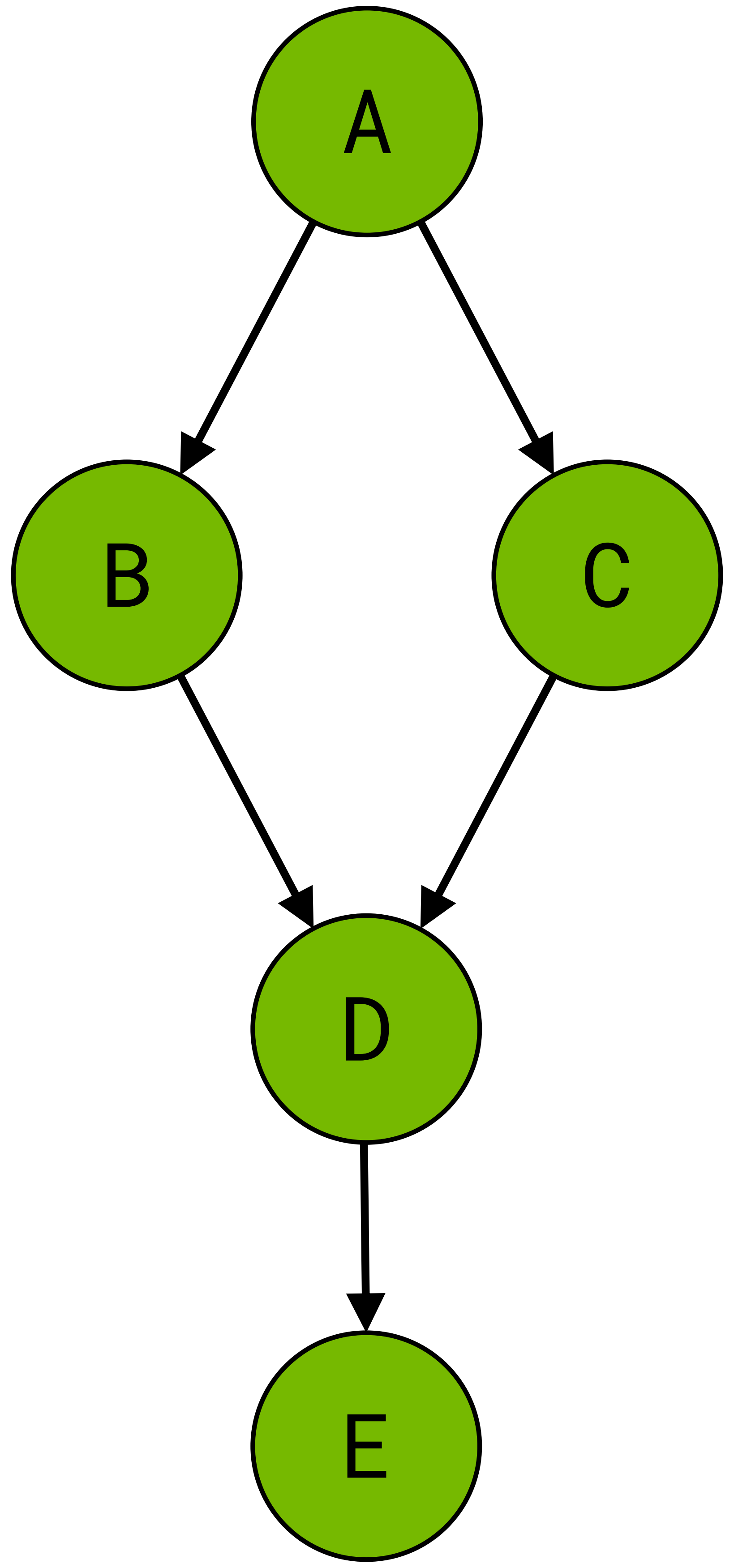
- Operation types:
  - kernel, memory copy, memset, conditional node, child graph, empty node, CPU function call, async. memory allocation node, etc.

- Dependencies highlight:
  - A, B, D, E should execute in this order. A, C, D, E too.
  - B and C may execute in parallel, or in any sequence, but should both execute after A and before D.

- Performance Benefits:
  - Reduce CPU launch submission overhead (esp. when relaunching same graph)
  - Enable potential optimizations on the GPU/driver side; work submitted as a whole
  - Enable other optimizations: conditional execution (condition changes at runtime), etc.

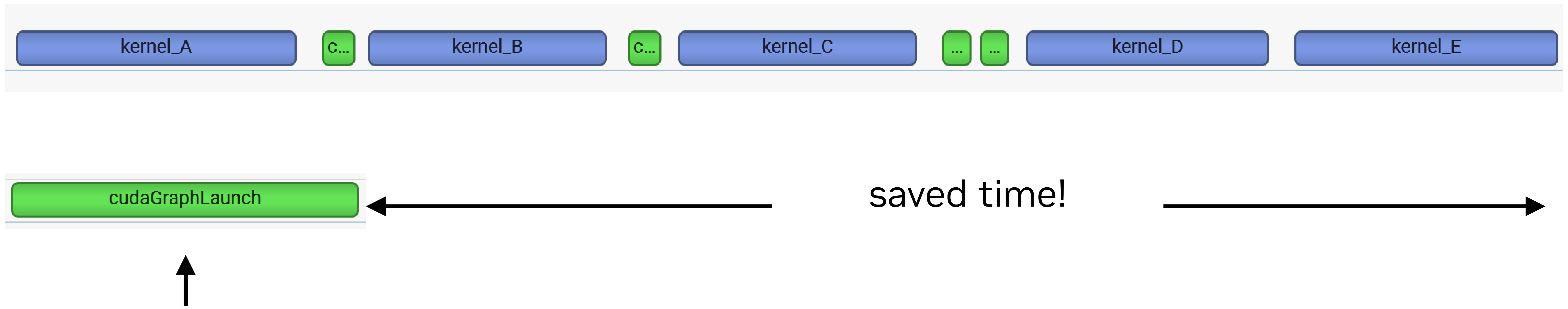
CUDA Graphs – CPU launch submission reduction

CPU Timeline



CPU timeline incl. CUDA APIs (2nd time operations launched)

- streams
- graphs



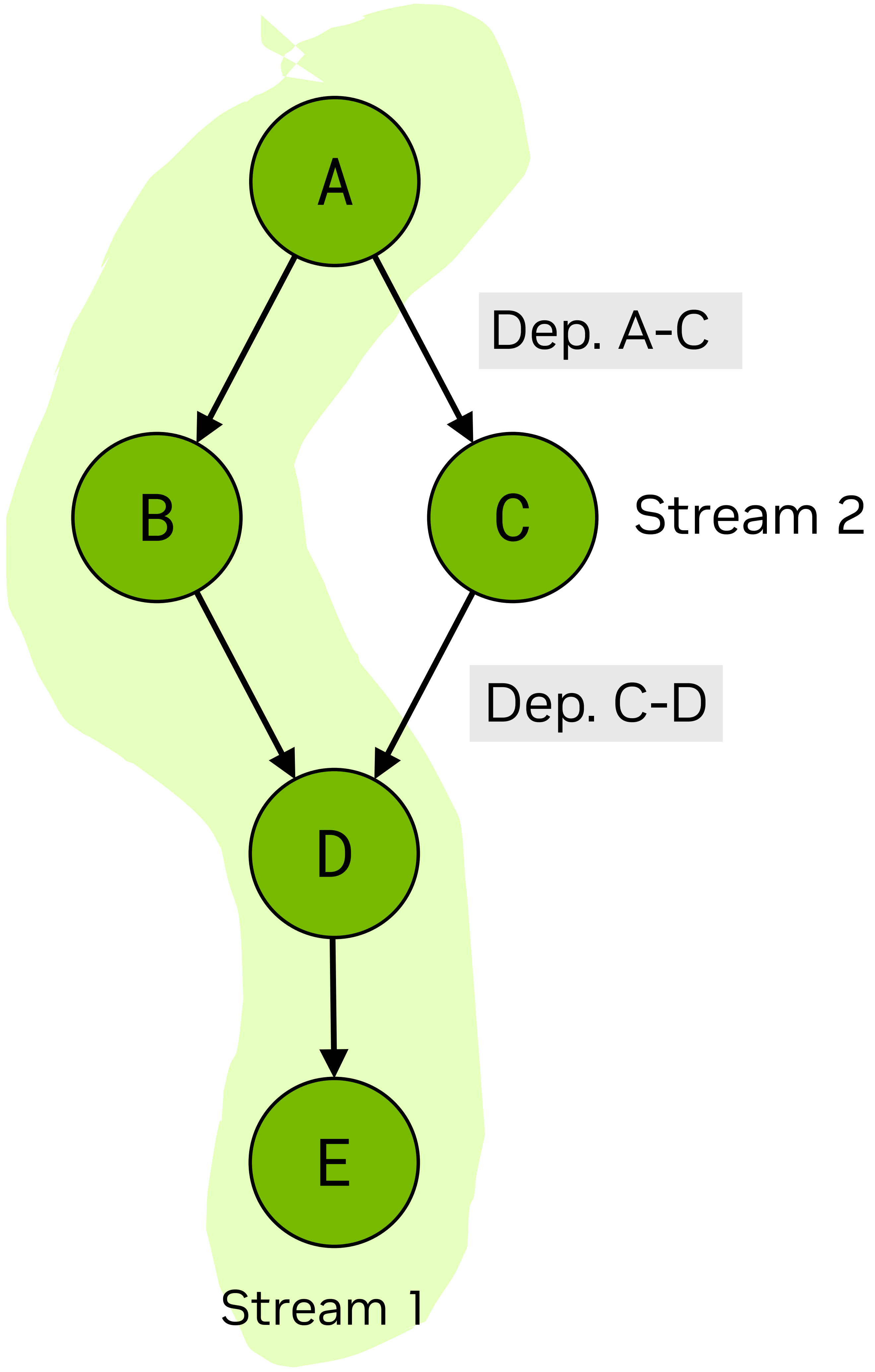
saved time!

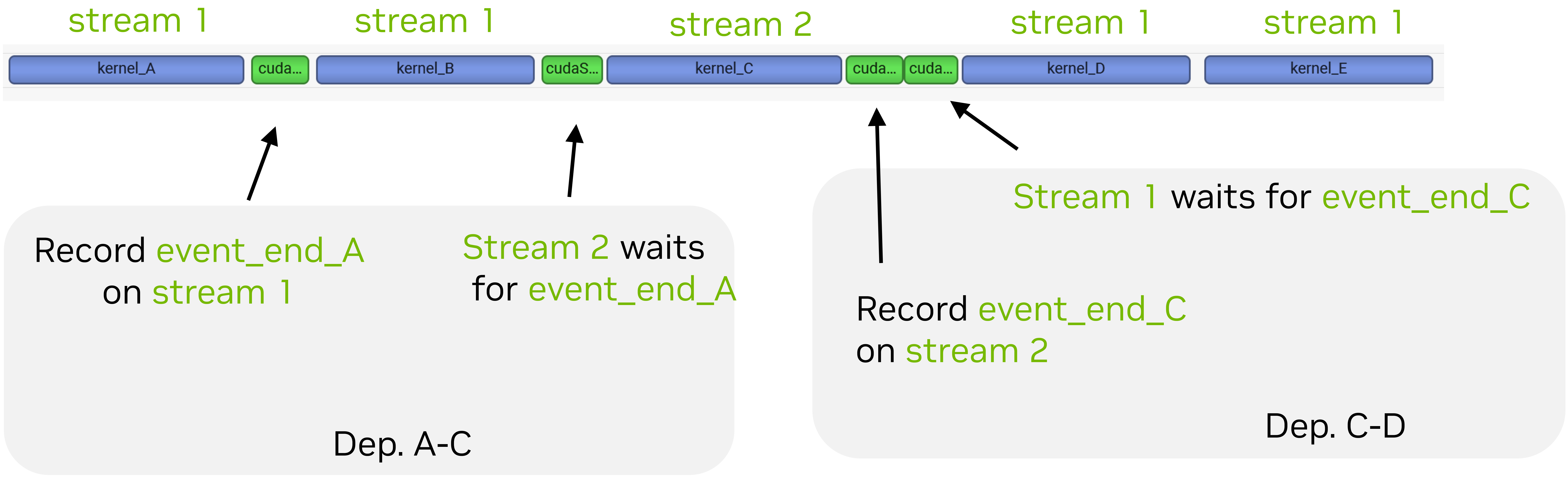
cudaGraphLaunch will launch the work encapsulated in the CUDA graph

Have CPU be ahead of your GPU, to avoid GPU timeline gaps because CPU is in the critical path.
CPU overhead effect more pronounced for short kernels.

CUDA Graphs – CPU launch submission reduction

Equivalent Streams Submission Timeline





Record event_end_A on stream 1

Stream 2 waits for event_end_A

Dep. A-C

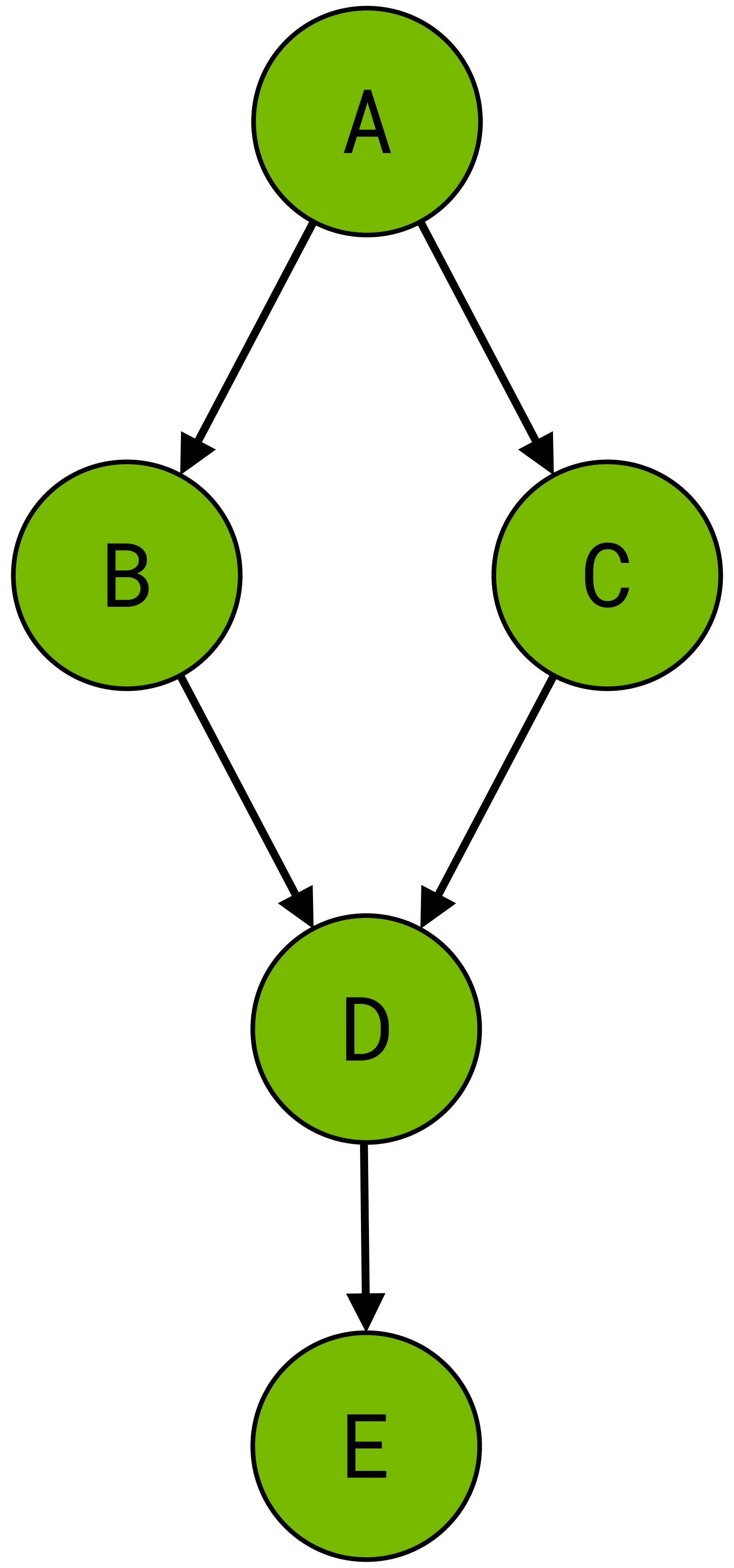
Stream 1 waits for event_end_C

Record event_end_C on stream 2

Dep. C-D

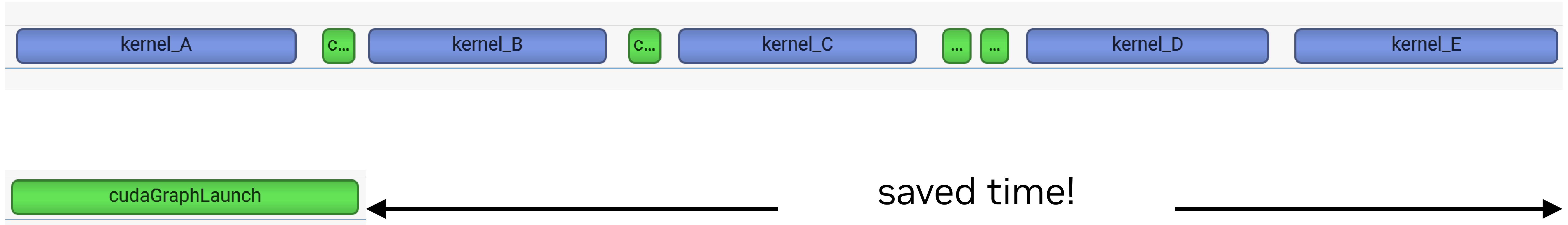
CUDA Graphs – CPU launch submission reduction

2nd time this graph or its equivalent streams' work is launched



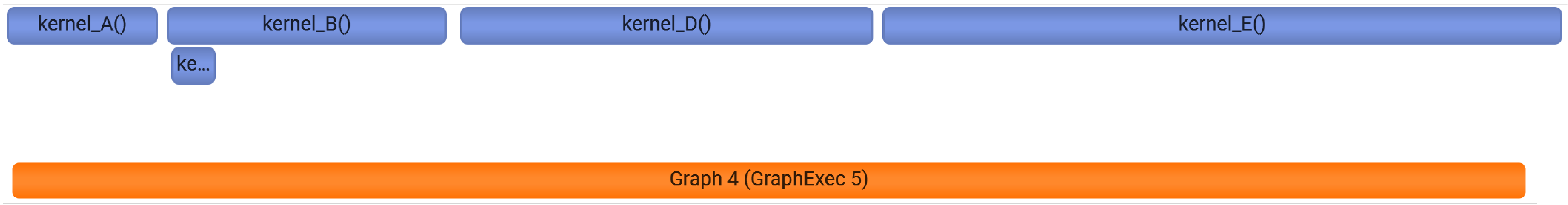
CPU timeline incl. CUDA APIs (2nd stream-based launch vs. graphs launch)

- streams
- graphs



GPU timeline (shown for completeness; timescale different than CPU)

- streams
- graphs



Timelines from Nsight Systems, using the default –cuda-graph-trace graph mode
Node level tracing possible via --cuda-graph-trace node, but can have significant overhead

CUDA Graph

Key steps: Definition, Instantiation, Launch

Define graph

Operations + dependencies encapsulated in a cudaGraph_t graph
2 ways: (a) stream capture or (b) using graph APIs

Instantiate graph

Instantiate the graph template generating an executable graph cudaGraphExec_t graph_exec
via cudaGraphInstantiate(&graph_exec, graph)

Launch exec. graph

The executable graph can be launched on a CUDA stream via cudaGraphLaunch(graph_exec, stream)
That stream is solely used for dependency tracking; gives no information about where graph nodes execute

Optional: can upload exec. graph to stream before launching via cudaGraphUpload(graph_exec, stream)

Next: Can relaunch the same executable graph as many times as needed.

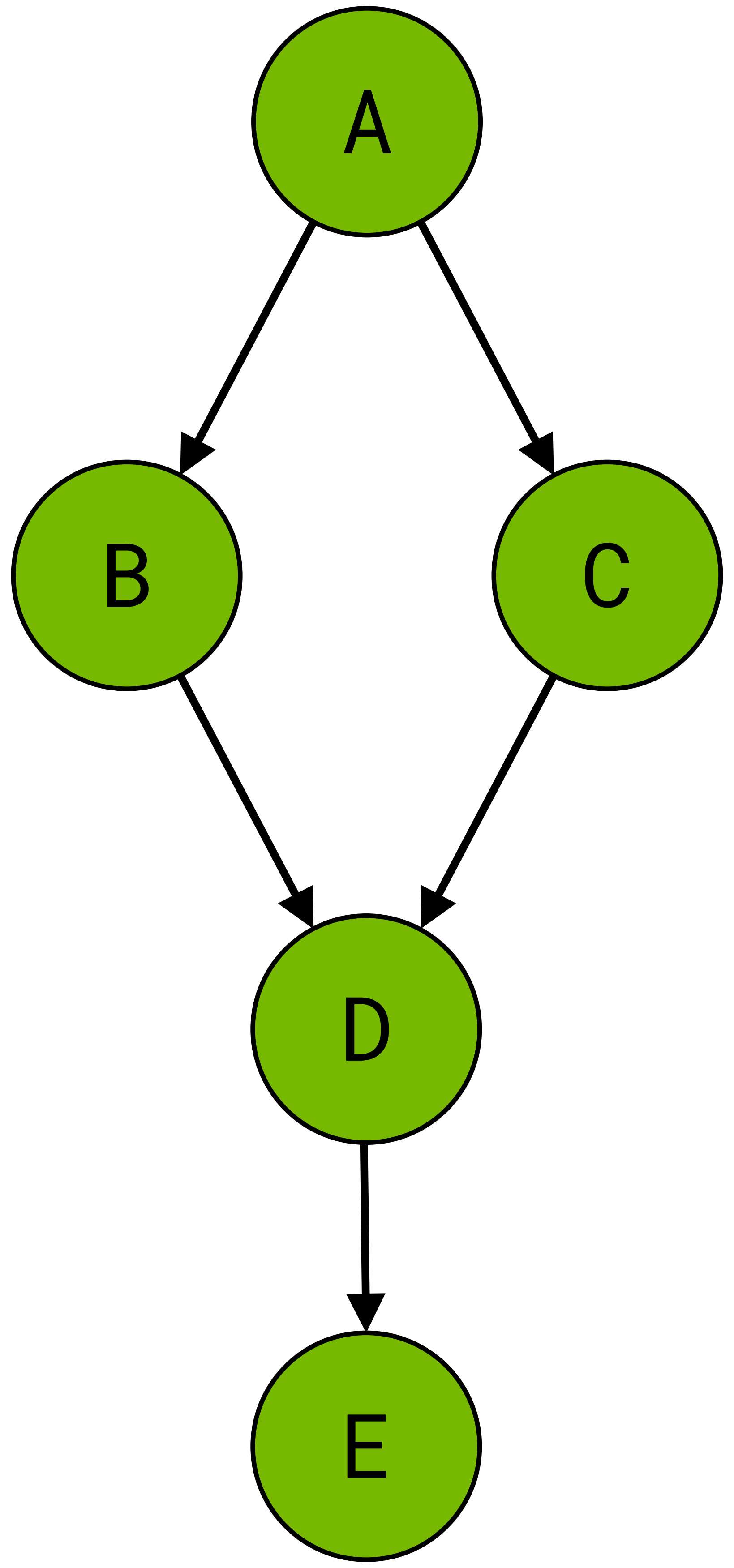
- Did kernel parameters change? Precede with node update as needed (cudaGraphExecKernelNodeSetParams) from CPU side
- Should some nodes not execute this time? Disable nodes (cudaGraphNodeSetEnabled) , if that's known from CPU before graph launch
- Much larger change? Re-instantiate.

Destroy graph

Destroy executable graph and graph

CUDA Graph Creation (2 ways)

Stream Capture



```
cudaGraph_t graph;
CUDA_CHECK(cudaStreamBeginCapture(strm1, cudaStreamCaptureModeGlobal));
kernel_A<<<1, 32, 0, strm1>>>();
CUDA_CHECK(cudaGetLastError());
CUDA_CHECK(cudaEventRecord(event_end_A, strm1));
…
kernel_C<<<1, 32, 0, strm2>>>();
…
kernel_E<<<1, 32, 0, strm1>>>();
CUDA_CHECK(cudaGetLastError());
CUDA_CHECK(cudaStreamEndCapture(strm1, &graph));
```

srtm1 capturing stream

srtm2 part of the capture; fork/join dep. w/ strm1

GPU work does not execute during stream-capture; it is just captured in graph.

Do not skip cudaGetLastError() after <<< >>> kernel launches, incl. when capturing! You could get silently missing kernels in your graph, if your kernel launch included an invalid argument (e.g., unsupported grid size, dyn. shared memory, etc.)
Disclaimer: We are skipping error checking code in the slides for brevity.

CUDA Graph Creation (cont'd)

Use graph APIs

```
cudaGraph_t graph;

cudaGraphCreate(&graph, 0);
cudaGraphNode_t node_A, node_B, node_C, node_D, node_E;

cudaKernelNodeParams params[5] = {};

// <… > // populate kernel node parameters

cudaGraphAddKernelNode(&node_A, graph, nullptr, 0, &params[0]);
```

graph node to add

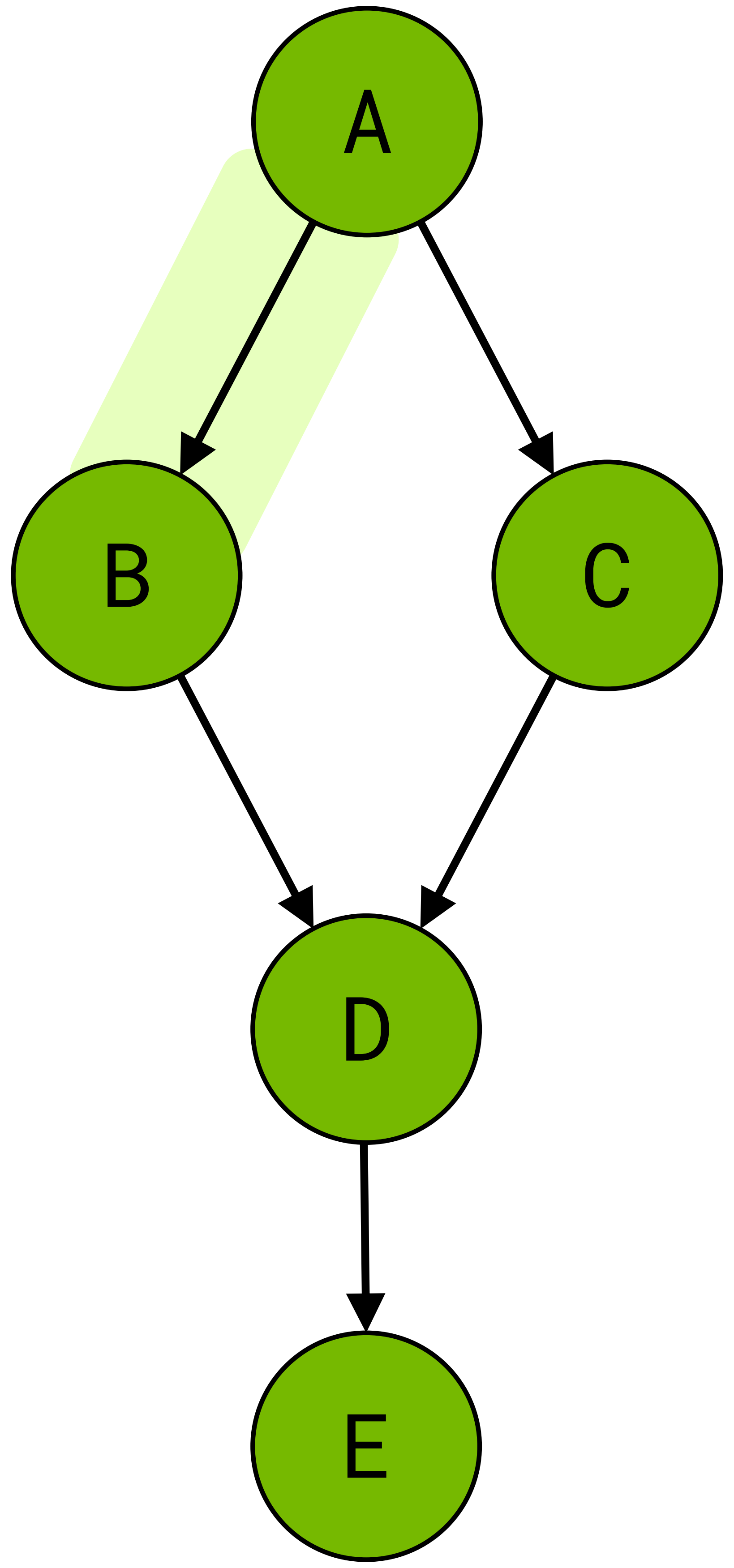
graph to
add node to

Dependencies of
the node

# dependencies
of node (0 if root)

parameters

```
cudaGraphAddKernelNode(&node_B, graph, &node_A, 1, &params[1]);
```

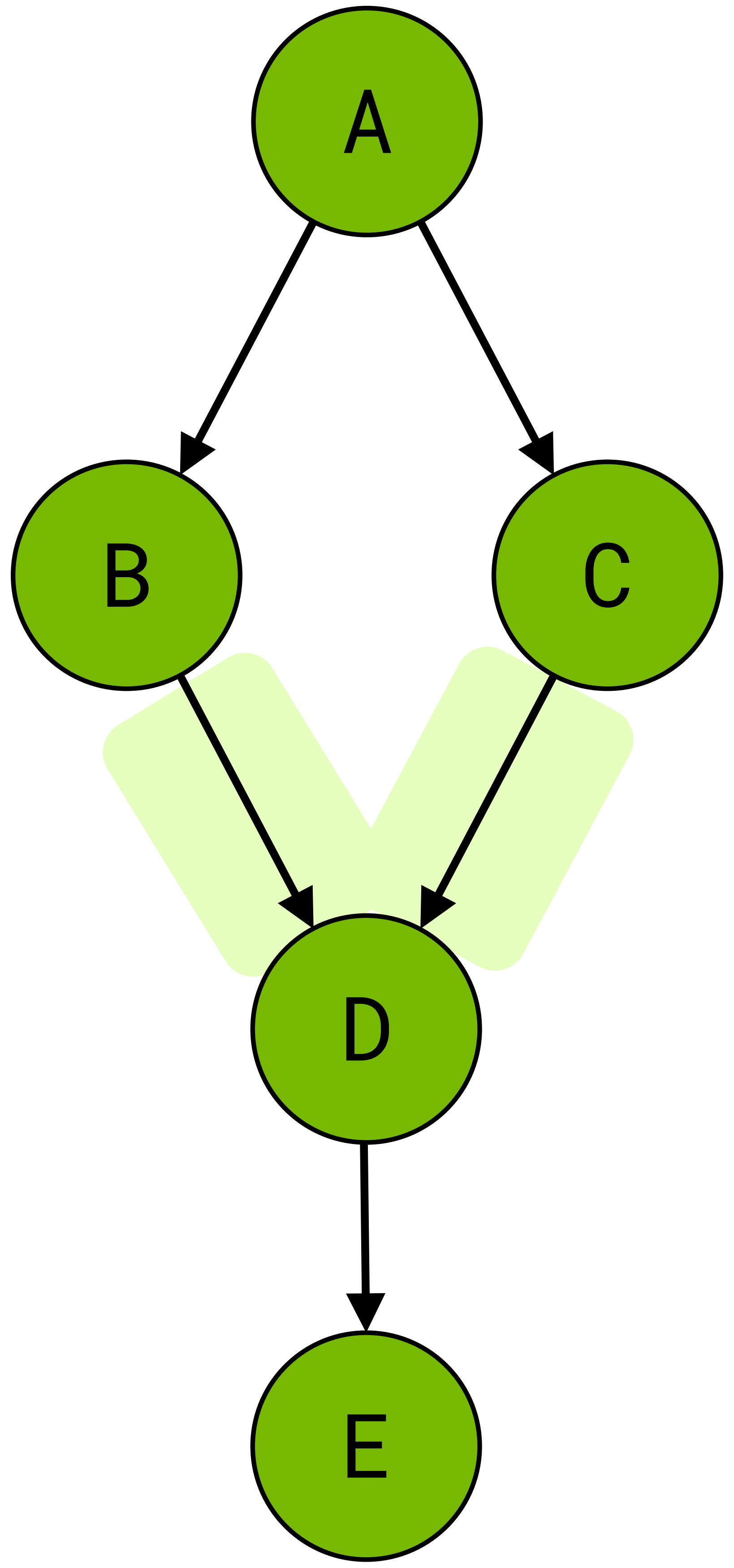


Reminder: Check the return value of every CUDA API call! Not shown here for brevity.

CUDA Graph Creation (cont'd)

Use graph APIs

```
cudaGraph_t graph;

cudaGraphCreate(&graph, 0));
cudaGraphNode_t node_A, node_B, node_C, node_D, node_E;
cudaKernelNodeParams params[5] = {};

// <… > // populate kernel node parameters

cudaGraphAddKernelNode(&node_A, graph, nullptr, 0, &params[0]);
cudaGraphAddKernelNode(&node_B, graph, &node_A, 1, &params[1]);

cudaGraphAddKernelNode(&node_C, graph, &node_A, 1, &params[2]);


std::vector<cudaGraphNode_t> node_deps = {node_B, node_C};
cudaGraphAddKernelNode(&node_D, graph, node_deps.data(), node_deps.size(), &params[3]);

cudaGraphAddKernelNode(&node_E, graph, &node_D, 1, &params[4]);
```



CUDA graphs creation 2 ways example: code will be available here

CUDA Graph Creation (cont'd)

Which way to choose?

- It depends! Some trade-offs to consider
- Stream Capture

  + may be the fastest way to leverage graphs if code already written

  + allows you to capture library calls too (e.g., into a subgraph)

  - But not all work is capturable (e.g., may need to convert synchronous calls to async., CPU logic to cudaLaunchHostFunc, etc.)

  - More work to update node parameters (need to get the node you need first)

- Graph creation with graph APIs (manual)

  + easier to express dependencies (no need for streams, events, etc.)

  + easier to explore different topologies/expand graph: just update dependencies of a node

  + easier to update node parameters or disable/enable nodes (nodes already known)

  - a separate code path would be needed to maintain both streams and graphs for existing code

CUDA Graph Tips/Reminders

Far from a complete list

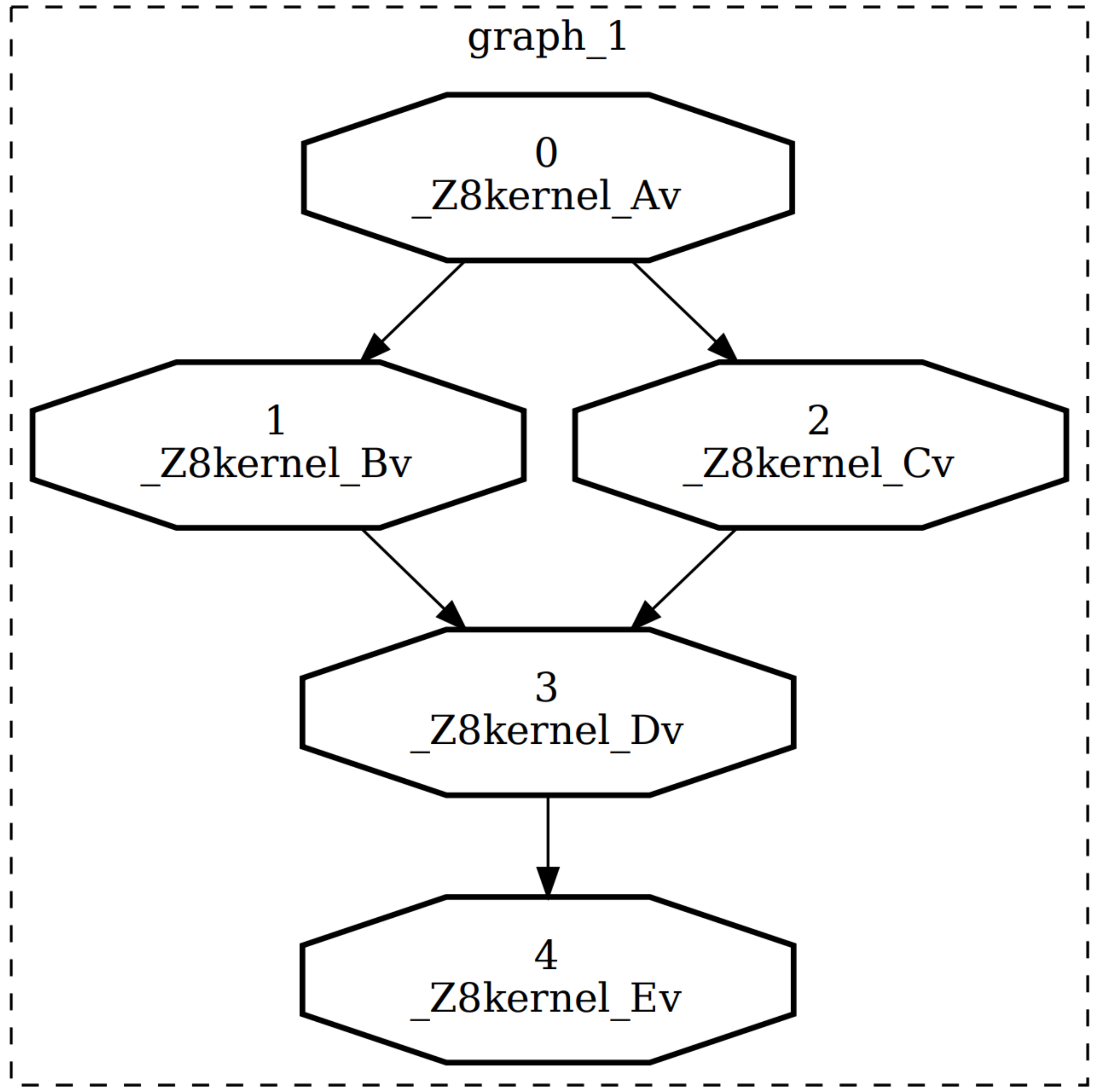
- cudaGraphDebugDotPrint to visualize a graph

  e.g., cudaGraphDebugDotPrint(graph, "graph", 0 /*or cudaGraphDebugDotFlagsVerbose */)

  Can convert dot file to pdf; can use c++filt to demangle names

- cudaGraphGetNodes to get number or list of nodes in a graph
  - Esp. useful for stream-captured graphs along with cudaGraphNodeGetType, cudaGraphKernelNodeGetParams, etc.

- cudaGraphExecKernelNodeSetParams to update kernel node parameters
- cudaGraphNodeSetEnabled to enable/disable a node in graph exec.

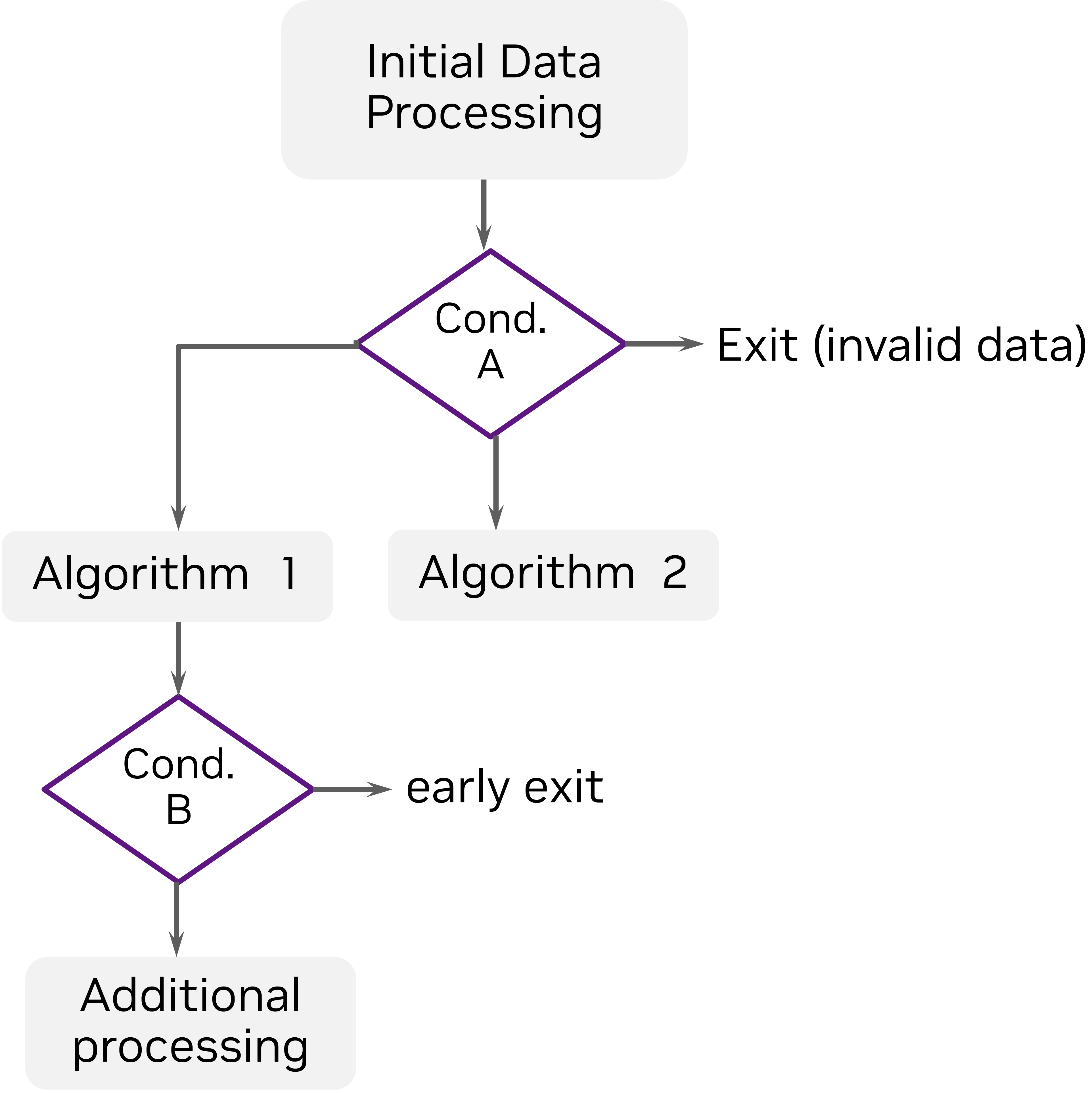


- If possible, create and instantiate your graph(s) outside your critical path. Do graph updates and launches there.

Enabling Conditional Execution

Motivation

Processing can depend on runtime condition(s) known after some processing (GPU work).



Examples:

- If your data has some characteristics, do additional processing
- If you've reached a good enough answer, skip subsequent processing
- If your processing is taking a long time, exit early
- … your use case here …

Enabling Conditional Execution

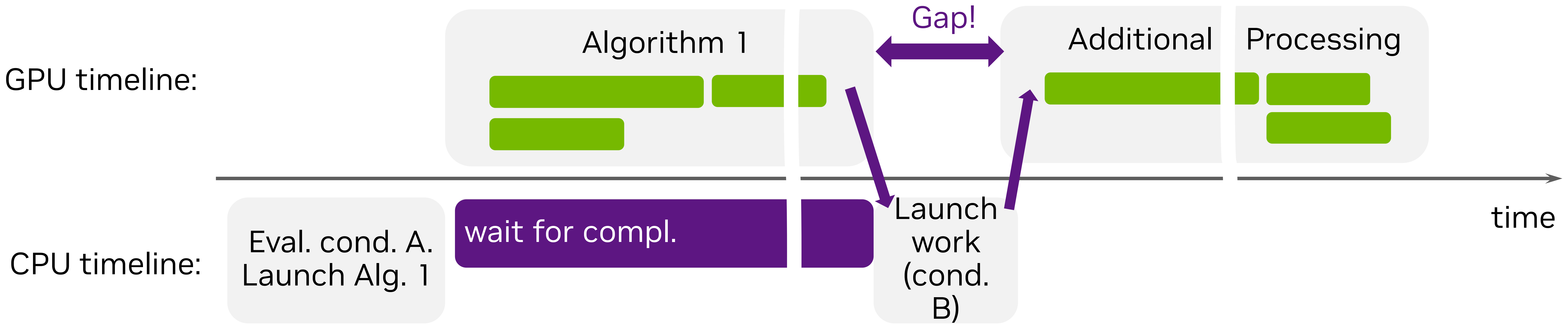
Motivation (cont'd)

Gap!
Gap!
GPU timeline:
Initial Data Processing
Algorithm 1
Additional Processing
time
CPU timeline:
Launch initial data processing
Wait for completion
Eval. cond. A. Launch Alg. 1
wait for compl.
Launch work (cond. B)
Critical path
Critical path

CPU evaluates condition and decides what to launch next:
Potential Issues: CPU not far ahead, GPU timeline gaps, launch overhead in critical path

Enabling Conditional Execution

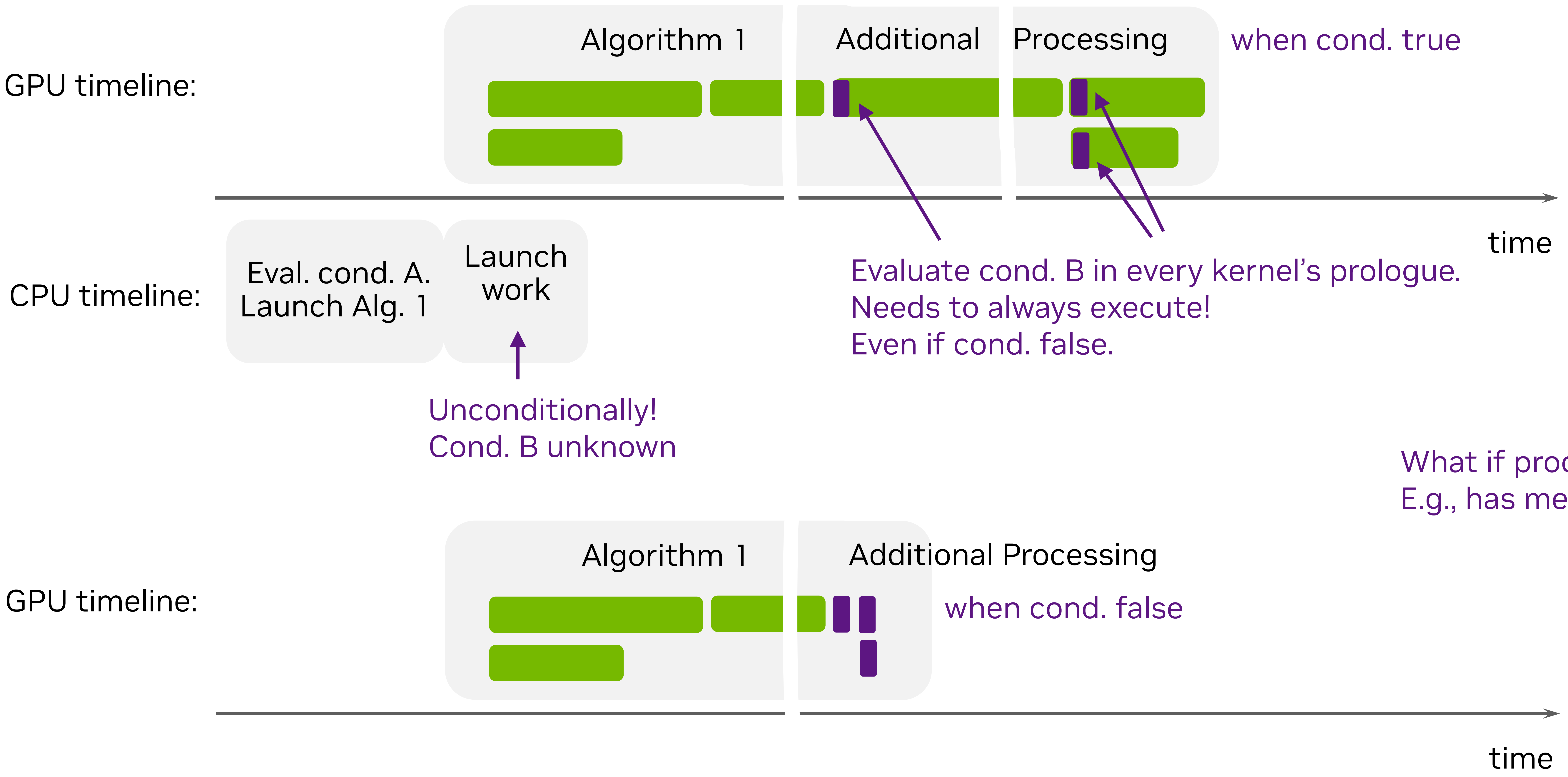
Motivation (cont’d)



What if we could evaluate the condition on the GPU?

Enabling Conditional Execution

Motivation (cont'd)



GPU timeline:

Algorithm 1

Additional Processing

when cond. true

time

CPU timeline:

Eval. cond. A.
Launch Alg. 1

Launch work

Unconditionally!
Cond. B unknown

Evaluate cond. B in every kernel's prologue.
Needs to always execute!
Even if cond. false.

GPU timeline:

Algorithm 1

Additional Processing

when cond. false

time

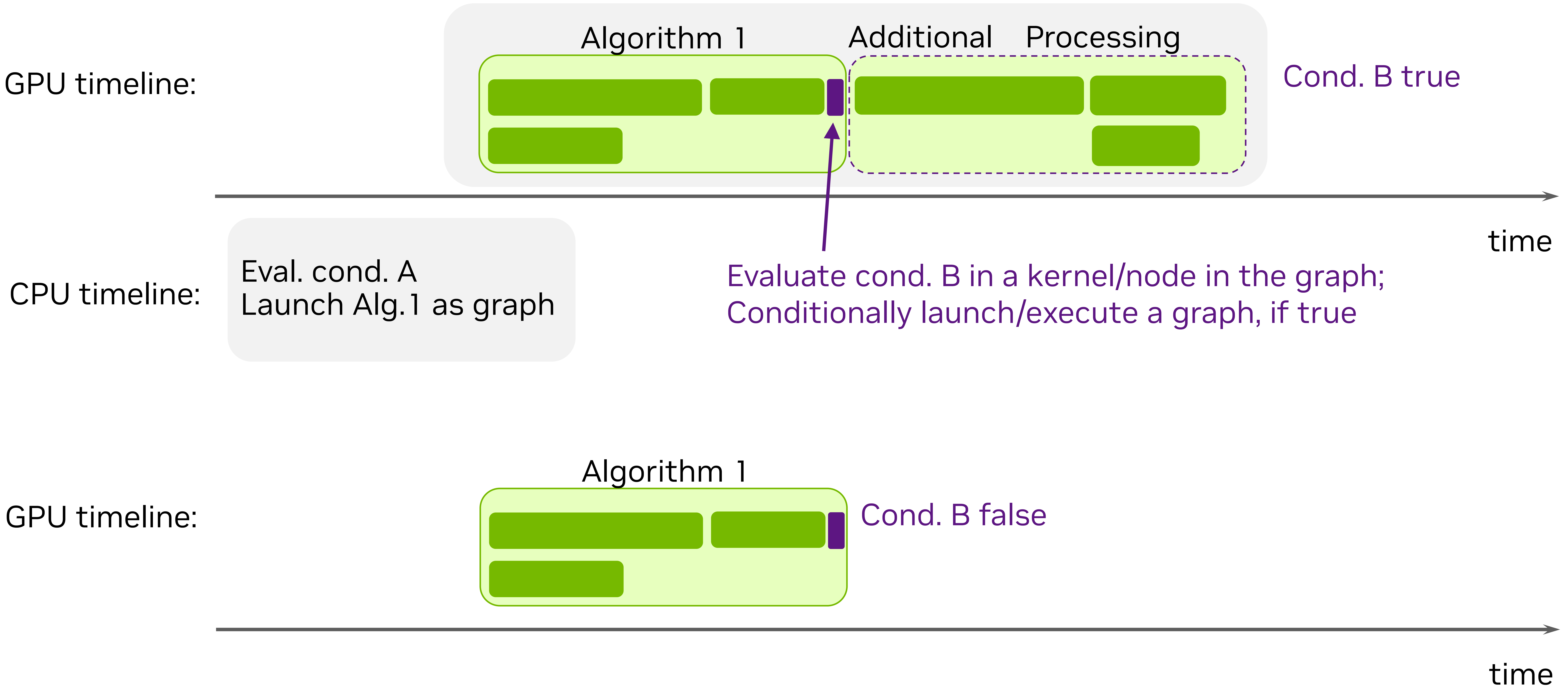
What if processing not just kernels?
E.g., has memcpy operation(s)?

Unconditionally launch GPU work & have GPU evaluate condition in every kernel's prologue

Potential Issues: not scalable (every kernel needs to be modified), n/a for non kernel work

Enabling Conditional Execution

Motivation (cont’d)



Encapsulate runtime-condition-dependent work into body-graph of a conditional node or a device-launched graph

CUDA Graphs

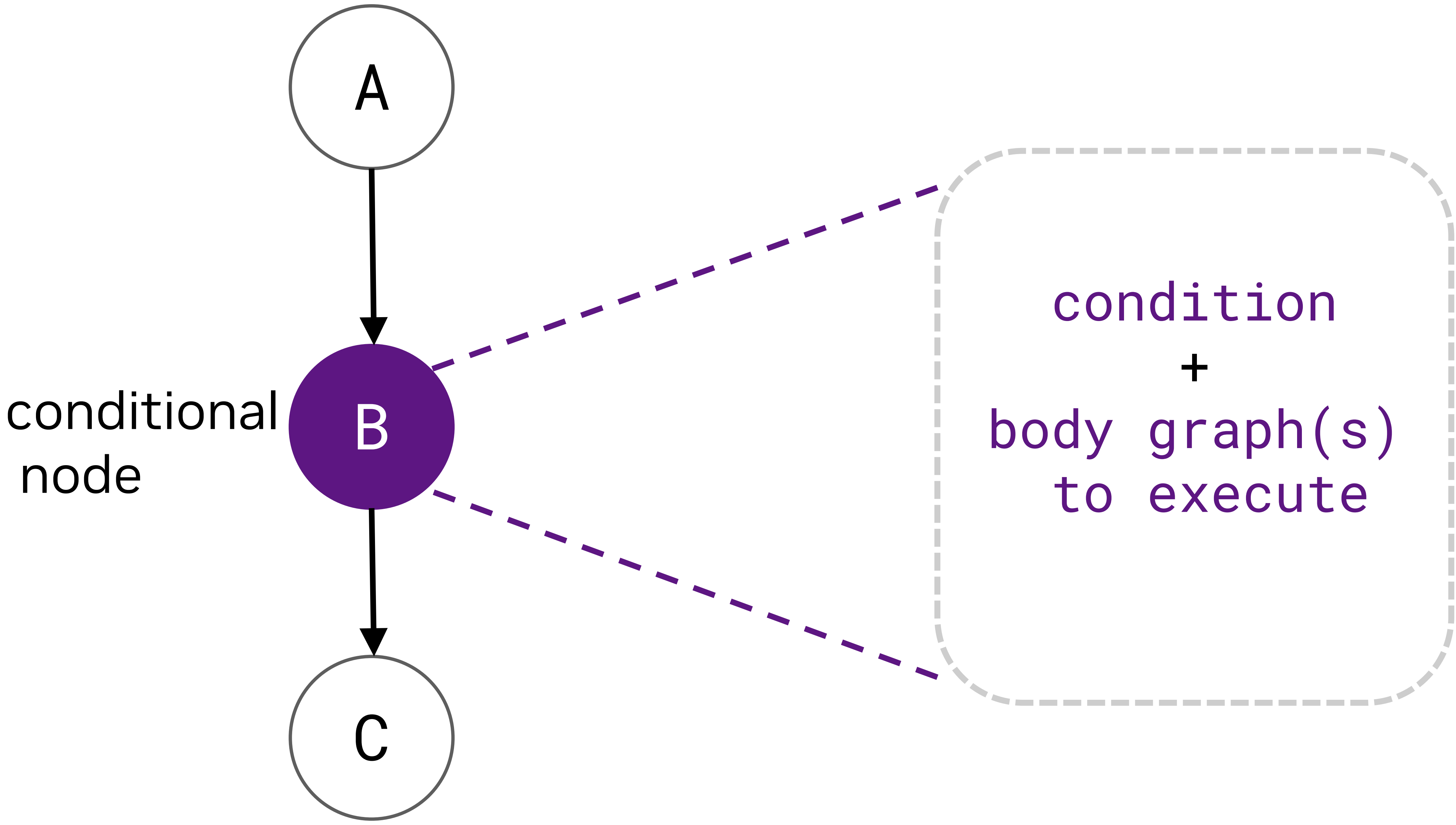
Enabling conditional execution

Processing can depend on runtime condition(s) known after some processing (GPU work)

Approaches	Pros	Cons
Go back to CPU, evaluate condition and launch appropriate work	No GPU kernel work modification	CPU waits for GPU completion; not far ahead GPU timeline gap(s) Launch overhead in critical path
Launch all GPU work unconditionally and evaluate condition on the GPU.	CPU not in the critical path. No GPU gaps	Modify prologue of every affected kernel to early exit if cond. false. Poor scaling. Kernels' prologue should always execute Not applicable if GPU work is not just kernel(s).
Encapsulate conditional work into body-graph of conditional node or a cond. launched device graph	CPU not in the critical path. No GPU gaps No GPU kernel modification; work need not be just kernels	May need to add extra join/fork graph nodes

Conditional Graph Nodes

Overview



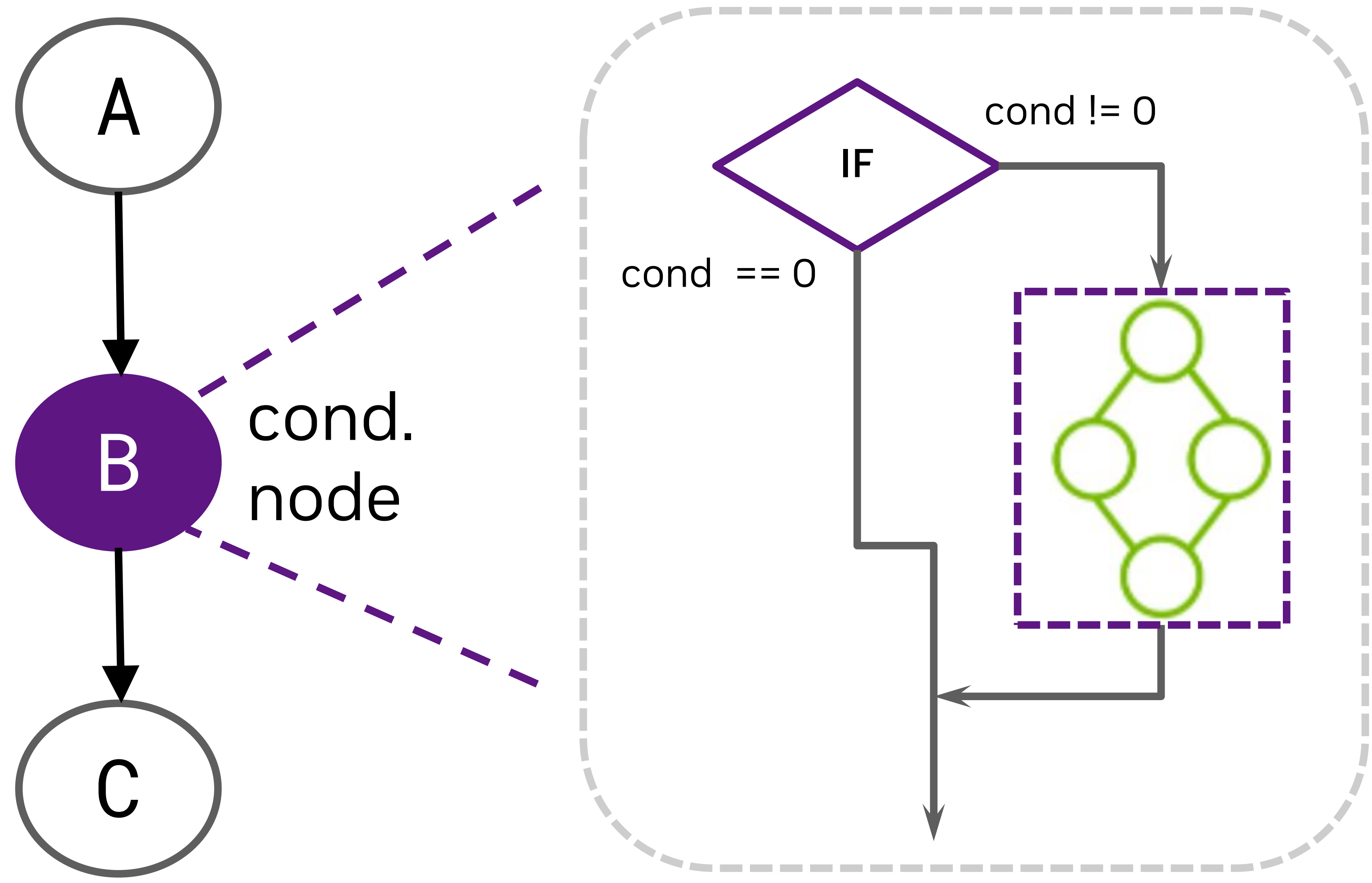
- A conditional node has:
  - a type
  - a condition (accessed via a conditional handle)
  - one or more body graphs associated with it

- Timeline:
  - kernel A executes
  - Conditional node B executes:
    - condition is checked & appropriate body graph, if any, executes depending on the node type and condition value
  - Kernel C executes after applicable body graph has completed

Conditional Graph Nodes

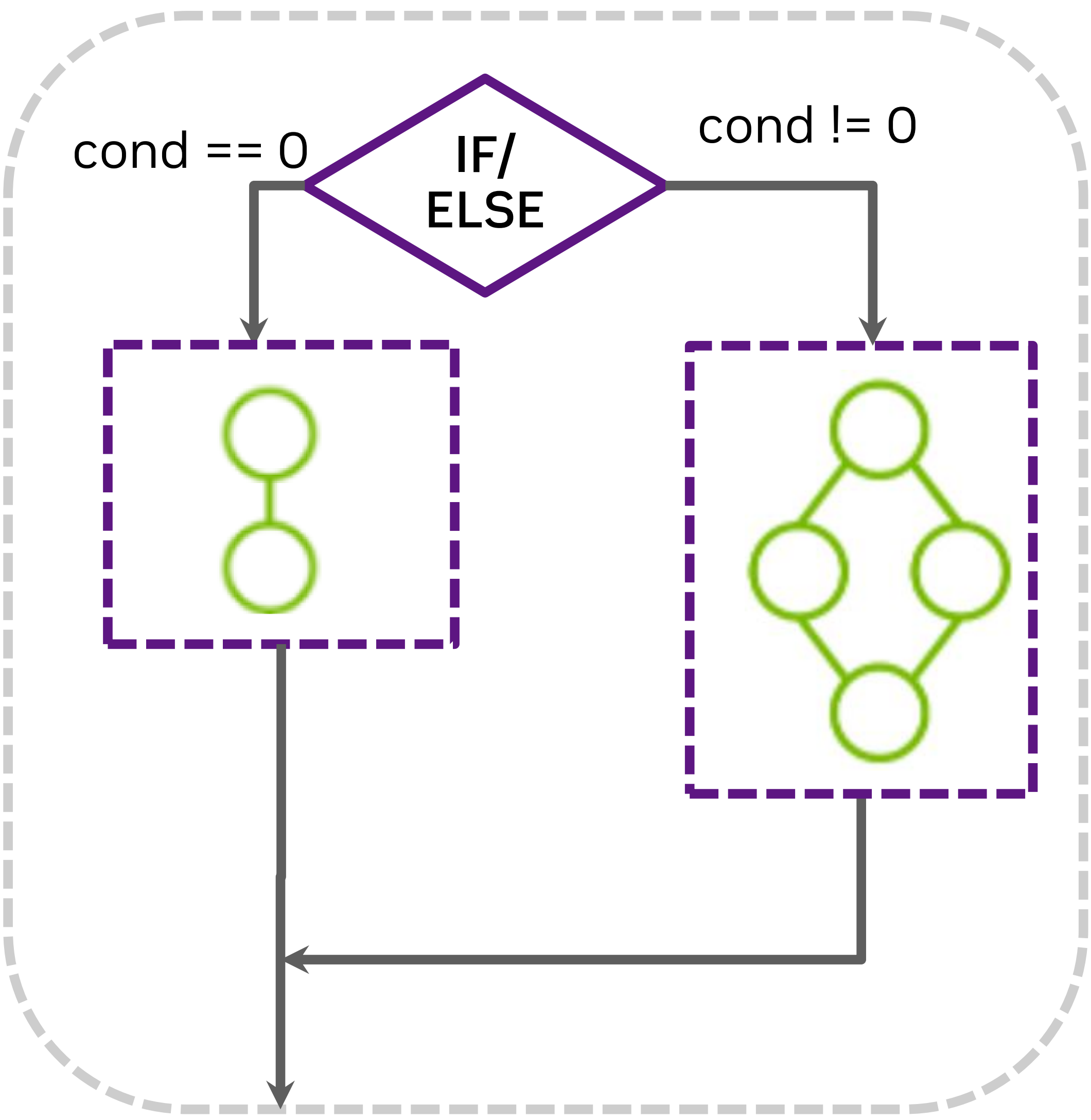
Node Types and their Body Graphs

IF



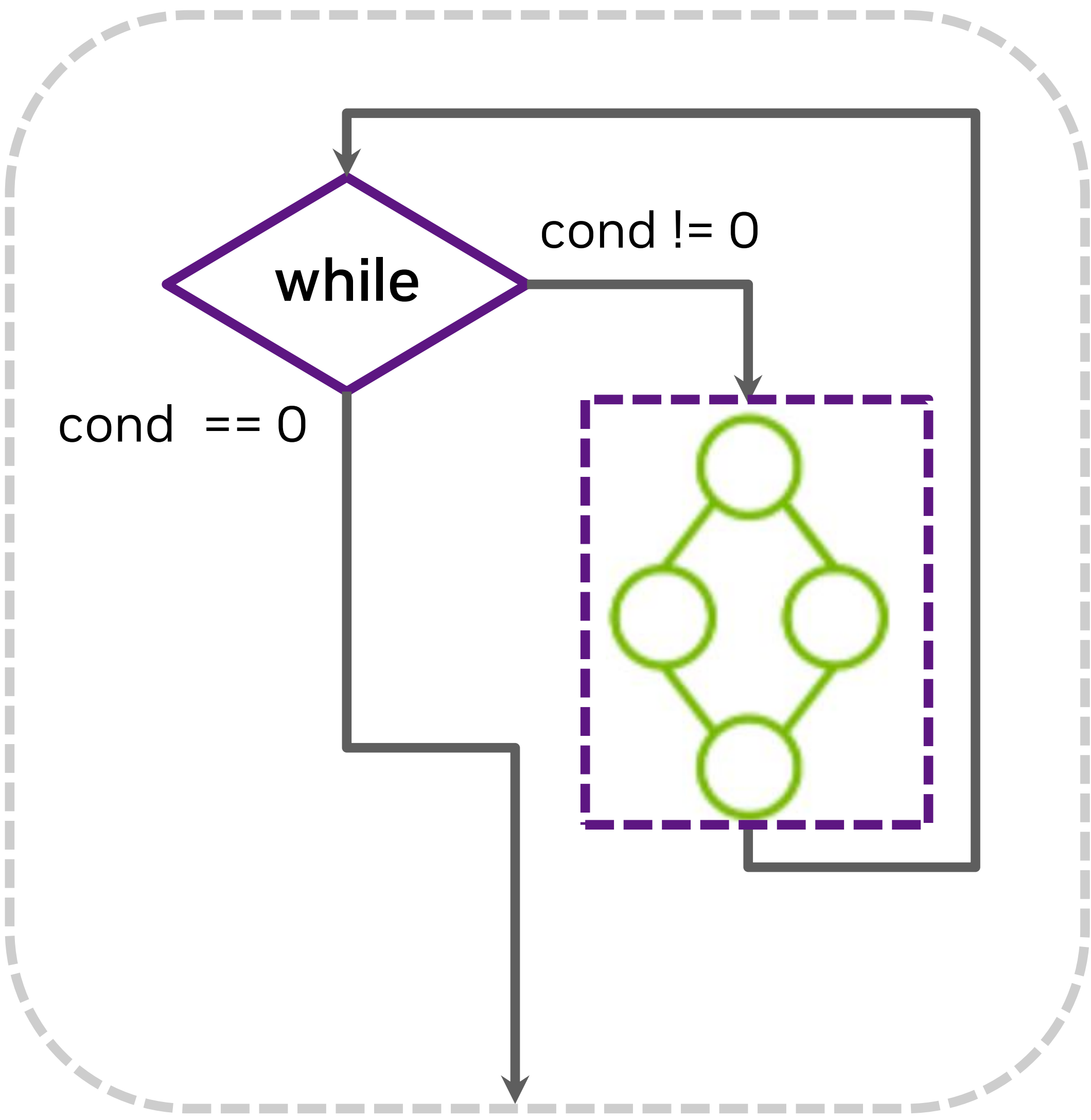
1 body graph

IF/ELSE



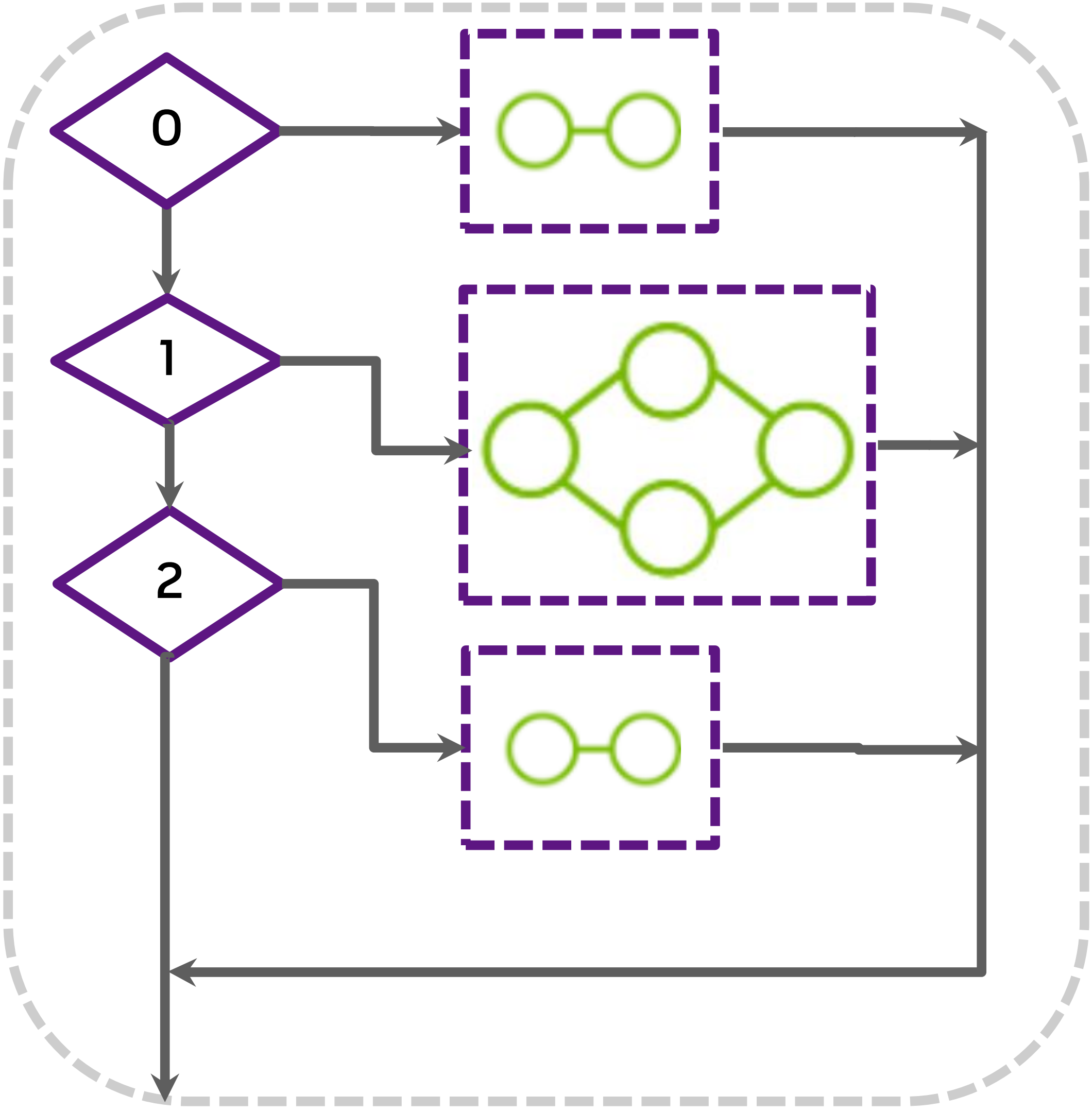
2 body graphs

WHILE



1 body graph

SWITCH



# case statements (N) body graphs

Conditional Graph Nodes

Controlling the condition

- Condition accessed via a cudaGraphConditionalHandle

- To create a handle:

```
cudaGraphConditionalHandle cond_handle;
cudaGraphConditionalHandleCreate(&cond_handle,   graph,   default_value,    flags);
```

Check returned value

cudaGraph_t created with cudaGraphCreate()

optional (see flags); applied on each graph launch

0 (no default value)
or
cudaGraphCondAssignDefault
(default_value)

- To set the condition after handle creation:

```
 __global__ void upstream_kernel(cudaGraphConditionalHandle handle, unsigned int new_cond_value, …) {
     if (threadId.x == 0) {
         cudaGraphSetConditional(handle, new_cond_value); // device only function; call from 1 thread
     }
}
```

Conditional Graph Nodes

Creating a conditional node

1. Create a conditional handle for this graph

```
cudaGraphConditionalHandle cond_handle;
cudaGraphConditionalHandleCreate(&cond_handle,   graph,   default_value,    flags);
```

2. Create and add a conditional node associated with this handle to the graph

```
cudaGraphNodeParams params = { cudaGraphNodeTypeConditional };
params.conditional.handle = cond_handle;  // previously created
params.conditional.type   = cudaGraphCondTypeIf; //or cudaGraphCondTypeWhile,cudaGraphCondTypeSwitch
params.conditional.size   = 1; // number of body graph(s); depends on node type

cudaGraphAddNode(&cond_node, graph, cond_node_deps.data(), cond_node_deps.size(), &params);
```

3. Populate conditional node's body graph(s), accessed via `params.conditional.phGraph_out[i]`

```
// Add node as root to the body graph of the cond. if node
cudaGraphAddNode(&node, params.conditional.phGraph_out[0], nullptr, 0, &pNodeParams);
```

4. Ensure condition is populated by one thread in an upstream kernel

Device Graphs

Overview

- A device graph can be launched from the device (as well as from the host)

- Device graph launch via cudaGraphLaunch from a kernel in a graph on a special stream
  - Only one thread from the grid should launch the graph!
  - Supported streams: cudaStreamGraphFireAndForget, cudaStreamGraphTailLaunch, cudaStreamGraphFireAndForgetAsSibling
  - You can control if the graph gets launched or not!

- Device graphs:
  - require a special flag cudaGraphInstantiateFlagDeviceLaunch during graph instantiation (cudaGraphInstantiate)
  - have additional restrictions compared to host graphs
    - E.g., a device graph can only contain kernel/memcpy/memset nodes and child graph nodes, etc. For example, graph instantiation will return an error if a device graph contains conditional nodes.
  - need to be uploaded to the device before launched from the device
    - Upload options: explicit cudaGraphUpload call, upload as part of instantiation via special flag, or extra launch from the host

References:

[1] https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html#group__CUDART__GRAPH_1g0b72834c2e8a3c93c443c6c67626d0d9

[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-graph-creation

CUDA Graphs

Further Reading

- CUDA graphs in the CUDA Programming Guide
- Graph Management API reference: runtime API and driver API

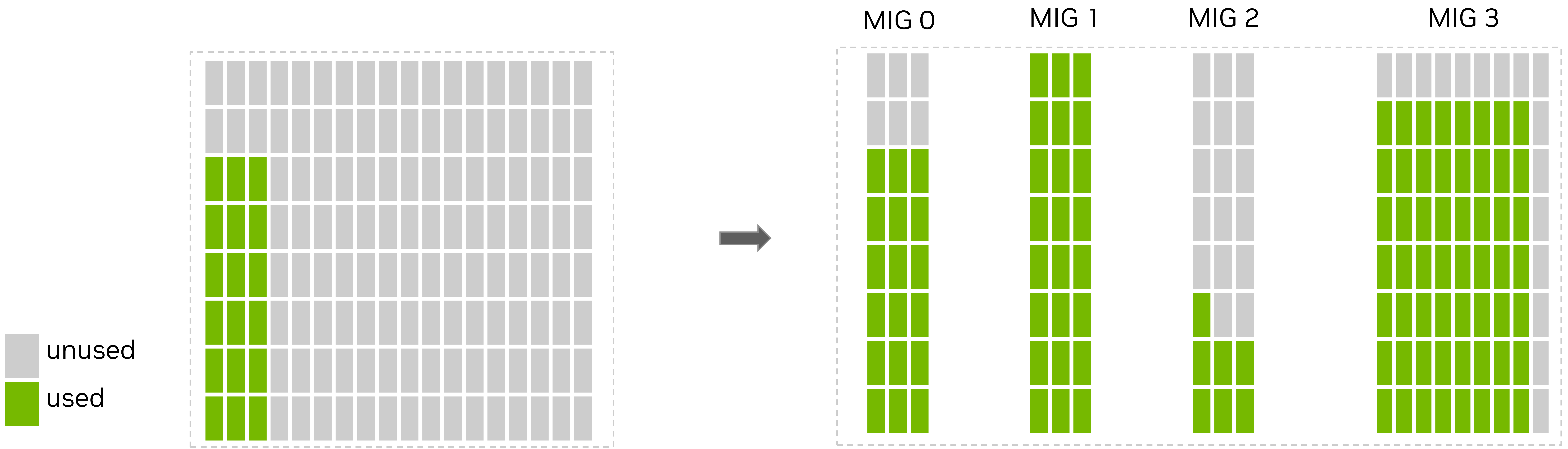
- NVIDIA Technical Blog Posts & GTC talks (not a complete list):
  - CUDA Graphs 101 | GTC Digital Spring 2023 | NVIDIA On-Demand
  - Getting Started with CUDA Graphs | NVIDIA Technical Blog
  - Constant Time Launch for Straight-Line CUDA Graphs and Other Performance Enhancements | NVIDIA Technical Blog
  - Dynamic Control Flow in CUDA Graphs with Conditional Nodes | NVIDIA Technical Blog
    - If, while (as of CUDA 12.4), if-else and switch (as of CUDA 12.8)
  - Constructing CUDA Graphs with Dynamic Parameters | NVIDIA Technical Blog
  - Optimizing llama.cpp AI Inference with CUDA Graphs | NVIDIA Technical Blog
  - Optimizing Drug Discovery with CUDA Graphs, Coroutines, and GPU Workflows | NVIDIA Technical Blog

- Relevant CUDA samples:
  - simpleCudaGraphs, jacobiCudaGraphs, graphMemoryNodes, graphMemoryFootprint, graphConditionalNodes, cudaGraphsPerfScaling

GPU Resource Partitioning Mechanisms

Motivation

A case for Multi-Instance GPU (MIG)



GPUs are getting bigger
Resources can be underutilized

Split a GPU into multiple "smaller GPUs" depending on your use case
Improve resource utilization by different applications at the same time

MIG (Multi-Instance GPU) can improve the utilization of your GPU

Motivation

A case for Multi-Process Service (MPS)

Process A
runs in isolation

Process B
runs in isolation

Processes A and B
run at the same time
(default compute mode, no MPS)

Processes A and B
run at the same time
with MPS

better
GPU utilization (%)
100
80
60
40
20
0
time
GPU utilization (%)
100
80
60
40
20
0
time
GPU utilization (%)
100
80
60
40
20
0
time
GPU utilization (%)
100
80
60
40
20
0
time

timeslice
+ ctx switch
overhead

🟩 process A

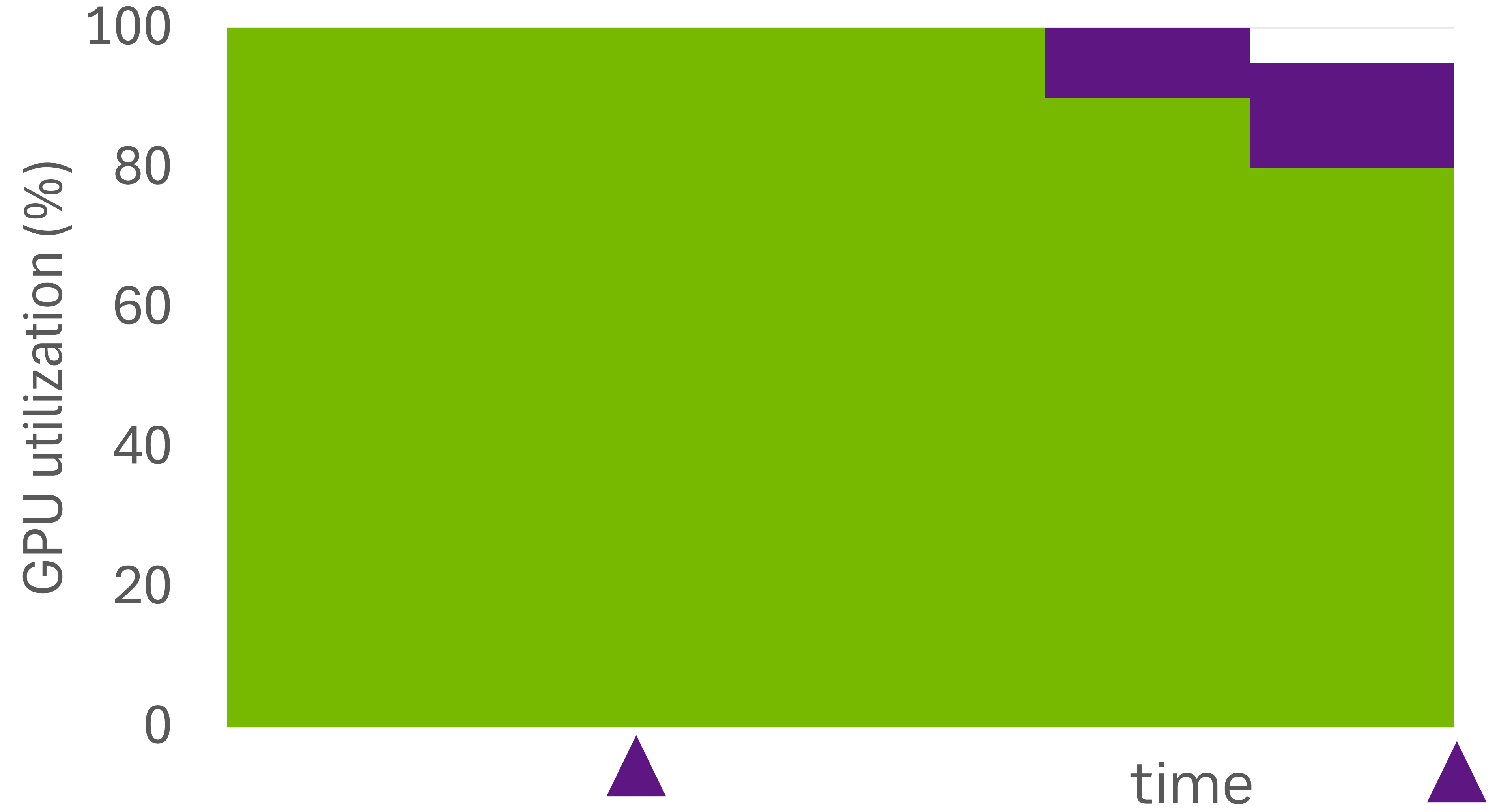
🟪 process B

MPS (Multi-Process Service) can improve the utilization of your GPU

Motivation

A case for Green Contexts (GCs)

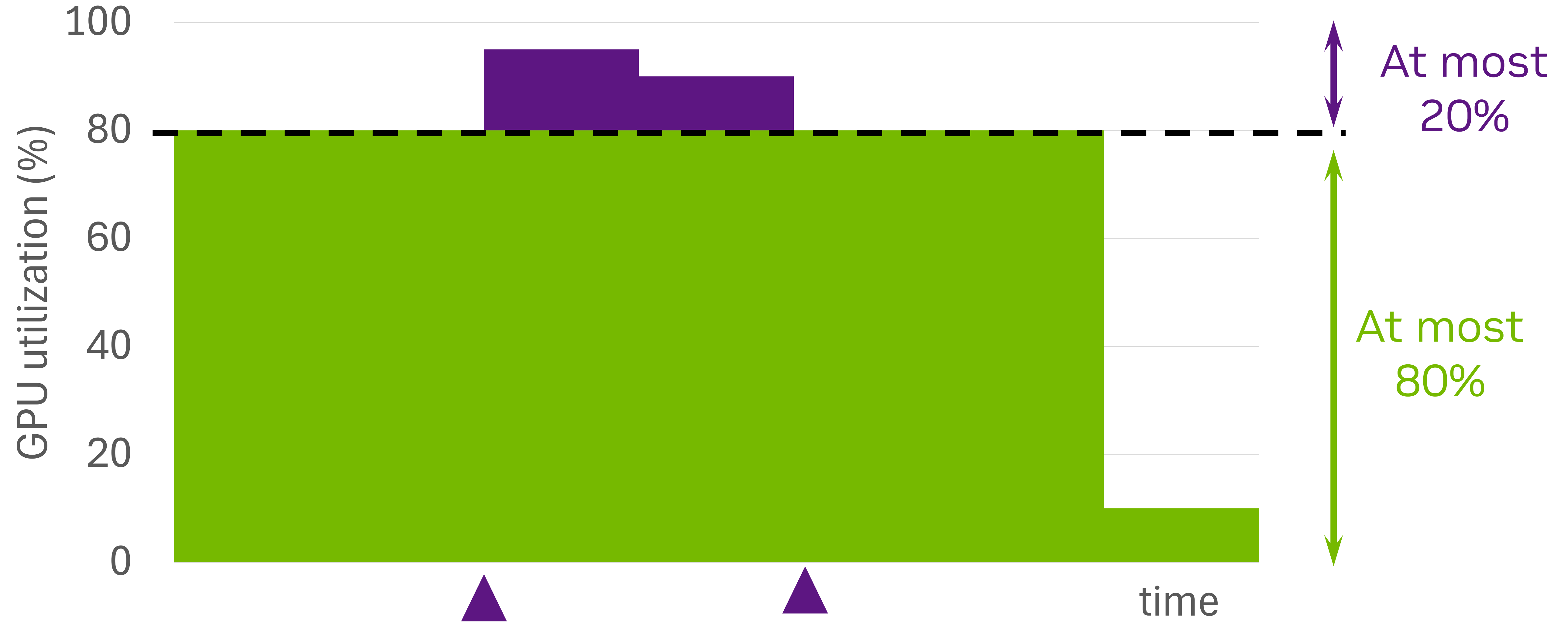
Critical work B gets delayed



independent + critical
work B launched
No resources available.

Work B
complete

Using 2 green contexts with 80% -20% SM split



independent + critical
work B launched

Work B
complete

🟩 kernel A

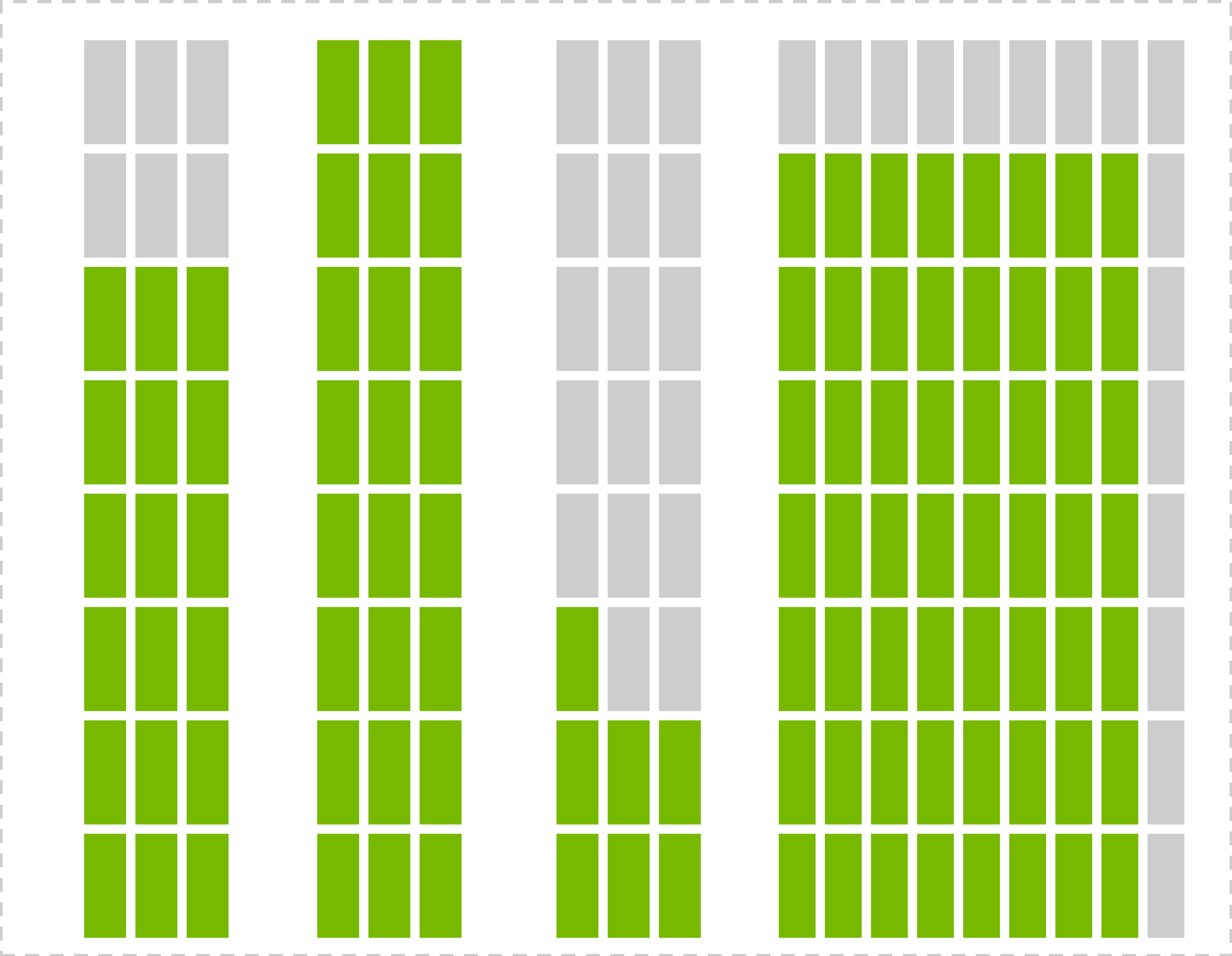
🟪 kernel B (time critical)

Green Contexts' static resource partitioning enables critical work B to complete sooner

Multi-Tenancy Options (Single-GPU)

Resource Partitioning Mechanisms (can be combined too)

MIG example

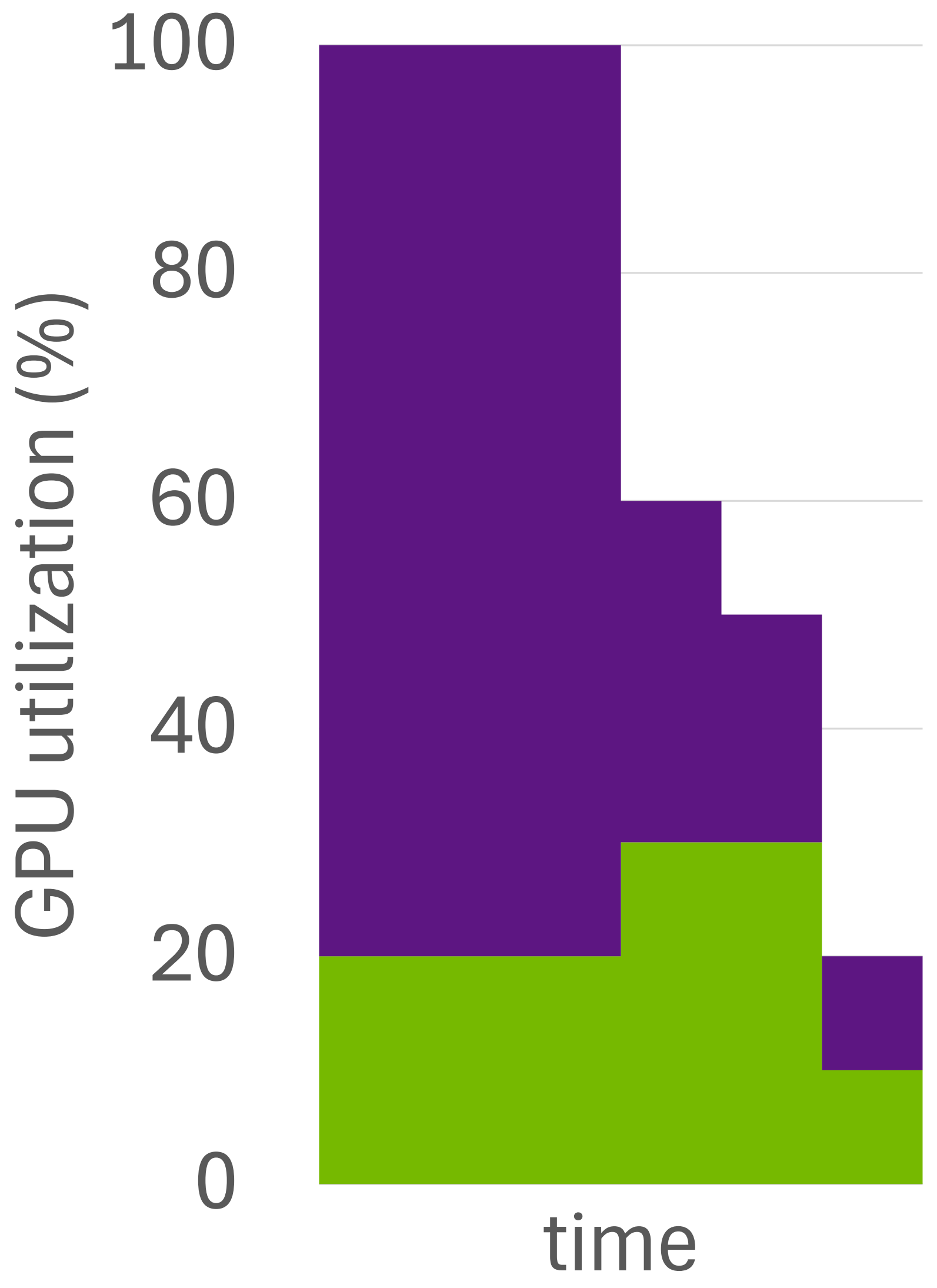


Statically partition GPU into multiple MIG instances ("smaller GPUs")

Different applications can use different MIG instances

Configured before application launch

MPS example

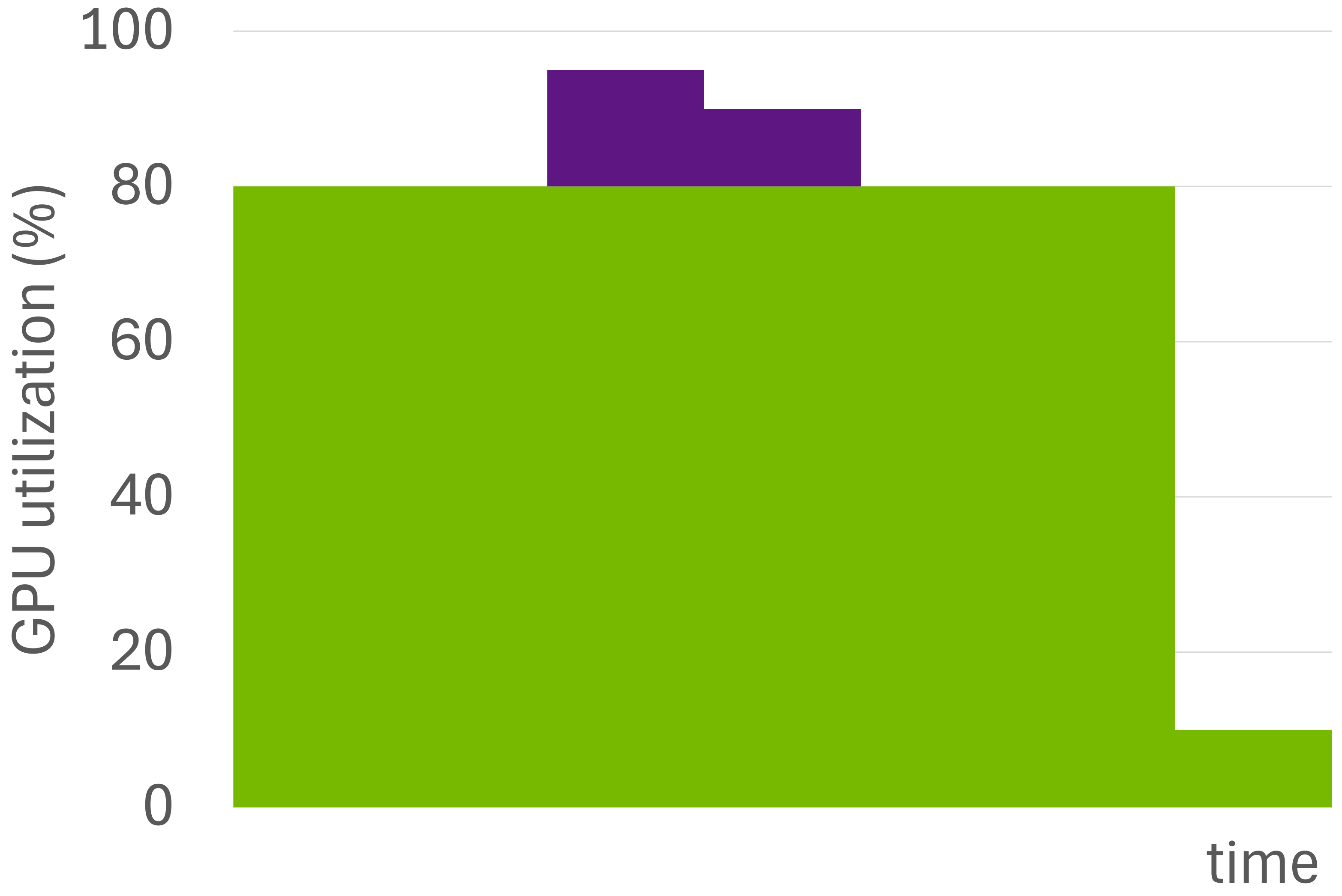


Dynamic partitioning

Mostly targets different processes

Requires MPS service to be running

Green Contexts example



Static partitioning of SMs

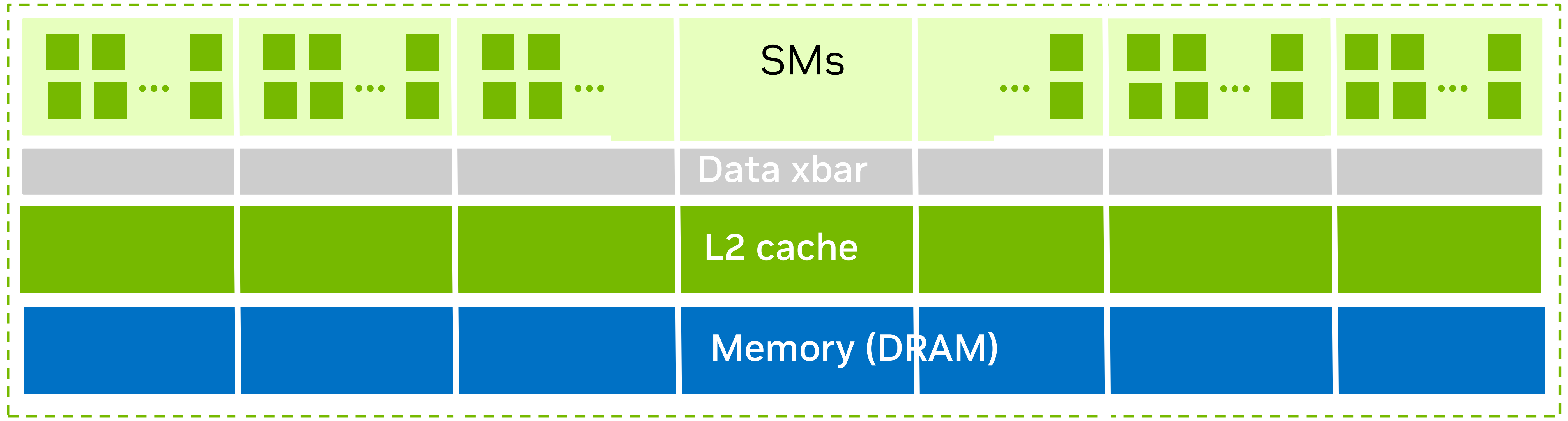
Partitions happen within an application

No extra service or configuration needed before application launch

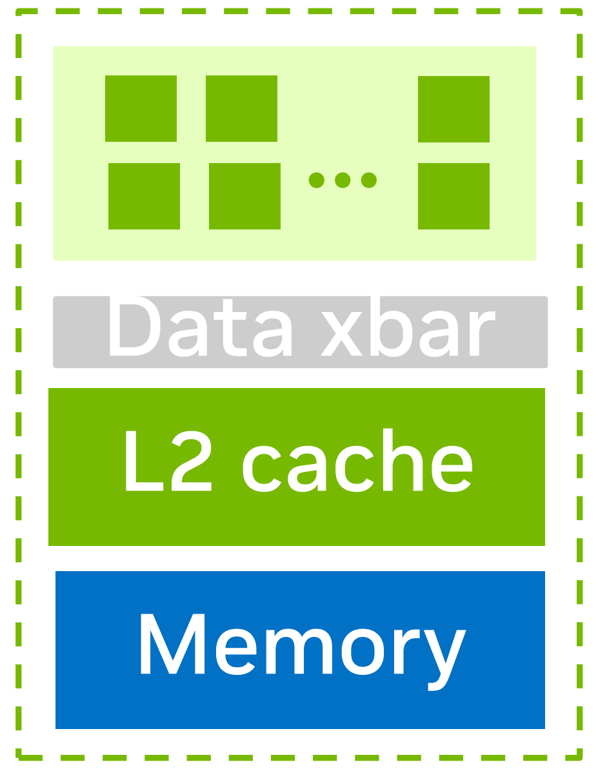
Multi-Instance GPU (MIG)

Overview

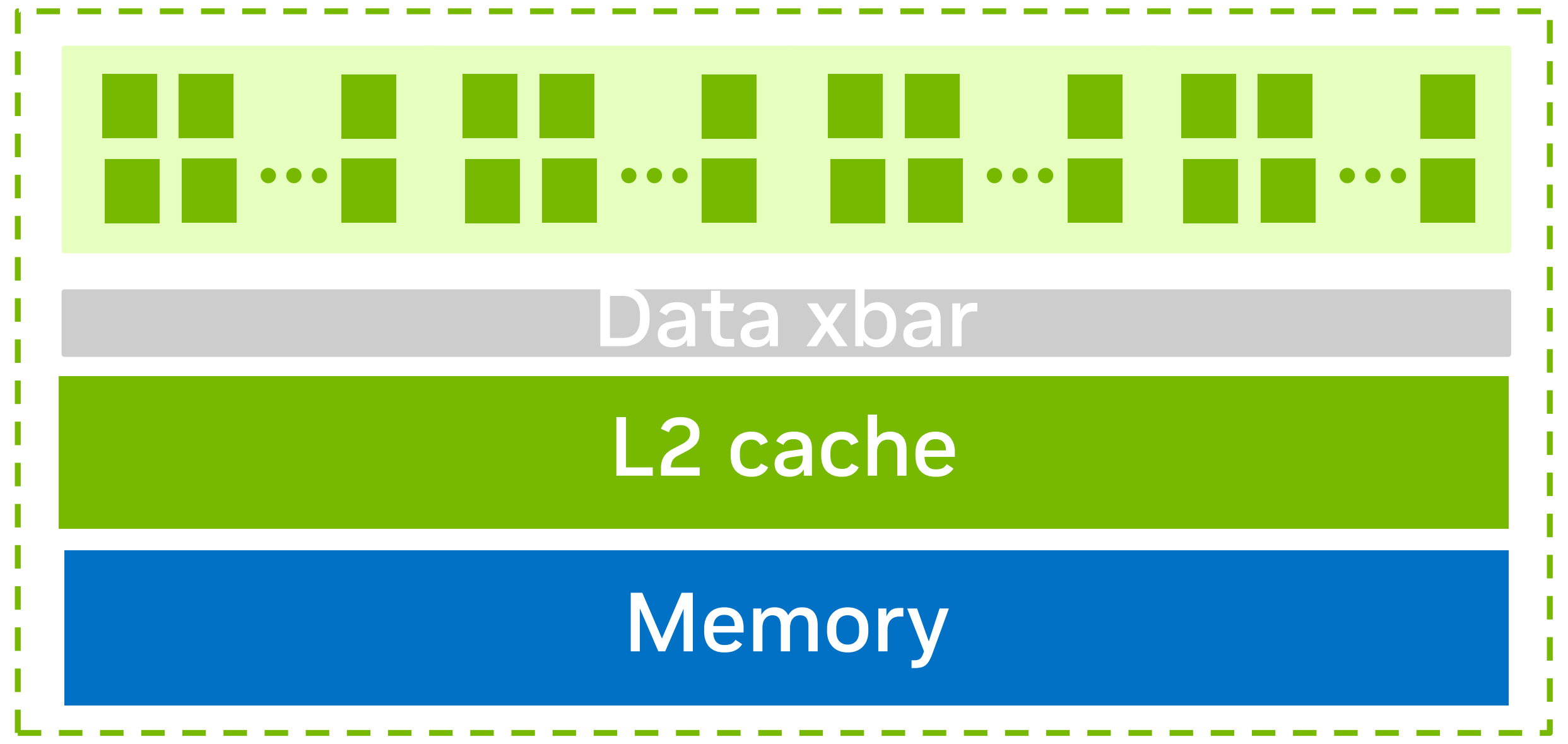
- Multiple MIG instances (physical partitions) possible on supported GPUs (CC >= 8.0)
  - Each instance has a predetermined fraction of GPU resources depending on the MIG profile used.



- A MIG profile (MIG Xg.Ygb) consists of X compute slices (SMs) + memory slices (L2, Mem.) with Y GB total memory



1g.12gb



4g.48gb

- MIG offers (across MIG instances): SM perf. isolation, error isolation, memory bandwidth QoS, memory protection

Multi-Instance GPU (MIG)

How to Use

- Enable MIG mode for your GPU, if supported via sudo nvidia-smi -i <GPU> -mig 1
  - sysadmin privileges needed; pass 0 to disable MIG mode in the end

- List supported GPU instance profiles via nvidia-smi mig –lgip
  Partial output (smallest + largest profile) from a GH200 below:

Name	ID	Instances Free/Total	Memory (GiB)	SM
MIG 1g.12gb	19	7/7	11.0	16
MIG 7g.96gb	0	1/1	93	132

- Create one or more instances via sudo nvidia-smi mig –cgi <profile ID1, profile ID2, ..,> -C
- Can list devices via nvidia-smi –L

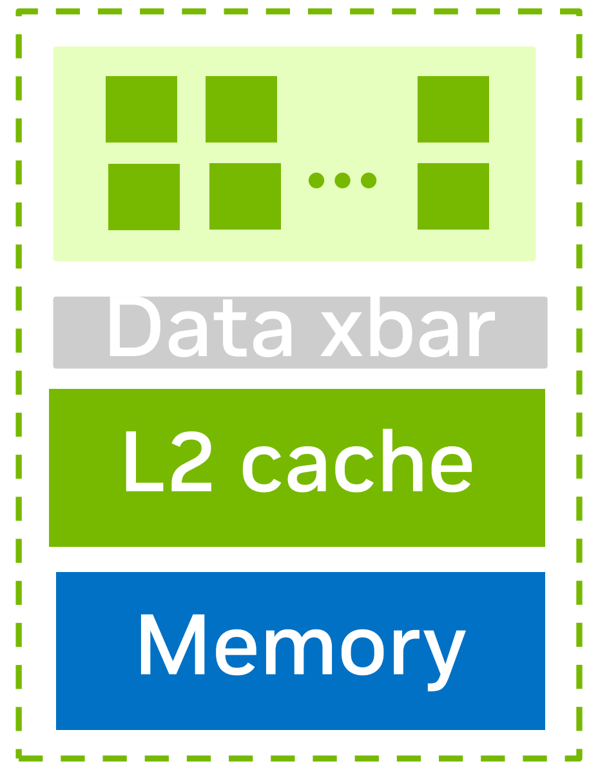
```
GPU 0: NVIDIA GH200 480GB (UUID: GPU-xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx)

  MIG 1g.12gb     Device  0: (UUID: MIG-xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx)
  MIG 1g.12gb     Device  1: (UUID: MIG-xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx)
```

- To run on a specific MIG instance, do:
  CUDA_VISIBLE_DEVICES=1 ./example_app or CUDA_VISIBLE_DEVICES=MIG-<UUID> ./example_app

- Destroy all MIG instances via sudo nvidia-smi mig -dci && sudo nvidia-smi mig –dgi and disable MIG mode

Multi-Instance GPU (MIG)

Summary



1g.12gb

Partition Type:	static (GPU resources only; no PCI-e)
When to enable/configure:	Before launching any application on the orig. GPU
Config. Options	MIG profiles to use; can affect app. performance
Application changes needed:	No
Use cases:	multiple-users or single user running different apps which underutilize the GPU, CSPs; QoS and isolation needed

References/Further reading:

- MIG: https://www.nvidia.com/en-us/technologies/multi-instance-gpu/
- MIG User Guide: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html
- GTC 2022 talk : Optimizing GPU Utilization: Understanding MIG and MPS
- NVIDIA Ampere Architecture whitepaper, “MIG (Multi-Instance GPU) Architecture” section: nvidia-ampere-architecture-whitepaper.pdf
- NVIDIA H100 Tensor Core GPU Architecture, “Second-Generation Secure MIG” section.

MPS

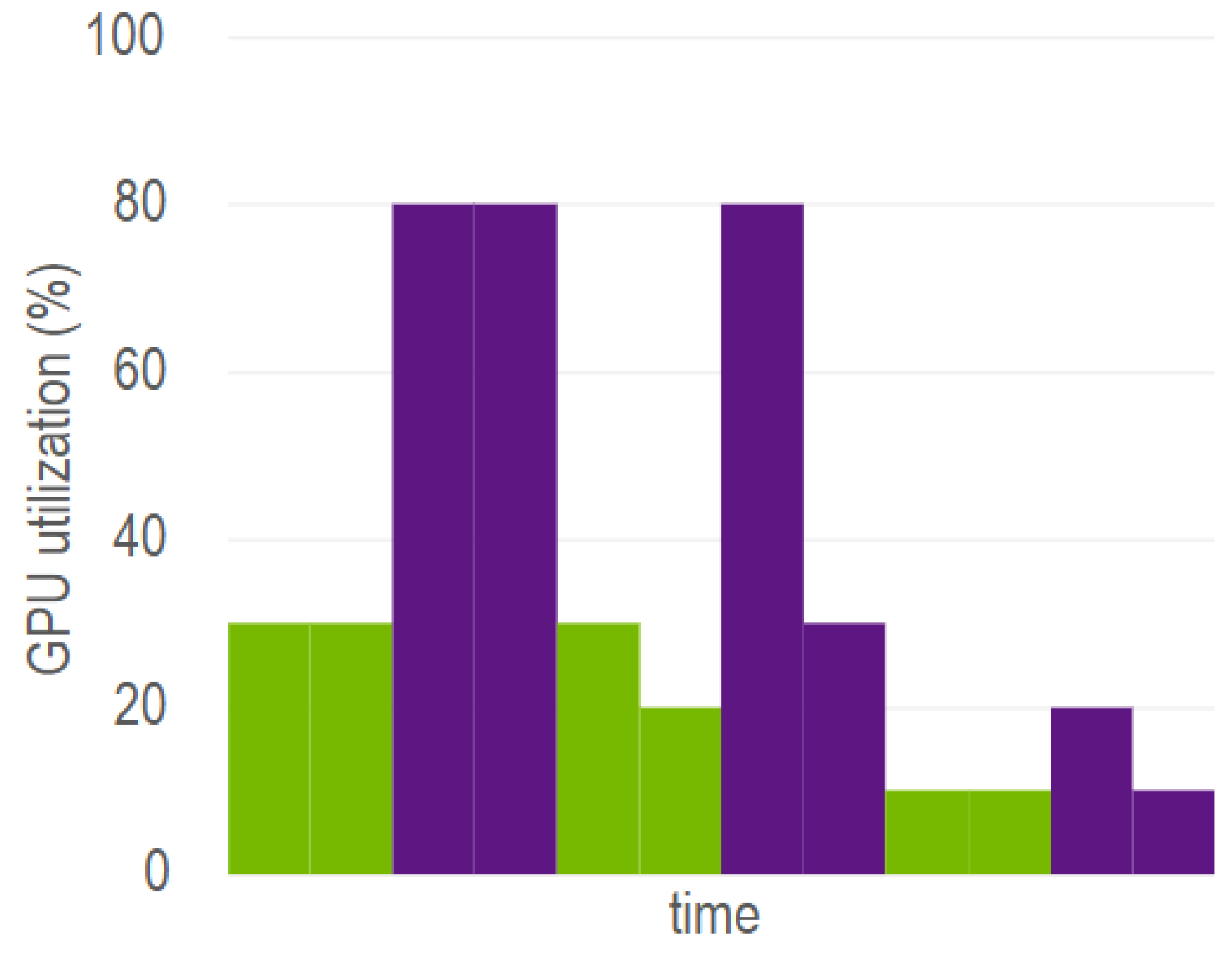
MPS (Multi-Process Service)

Overview

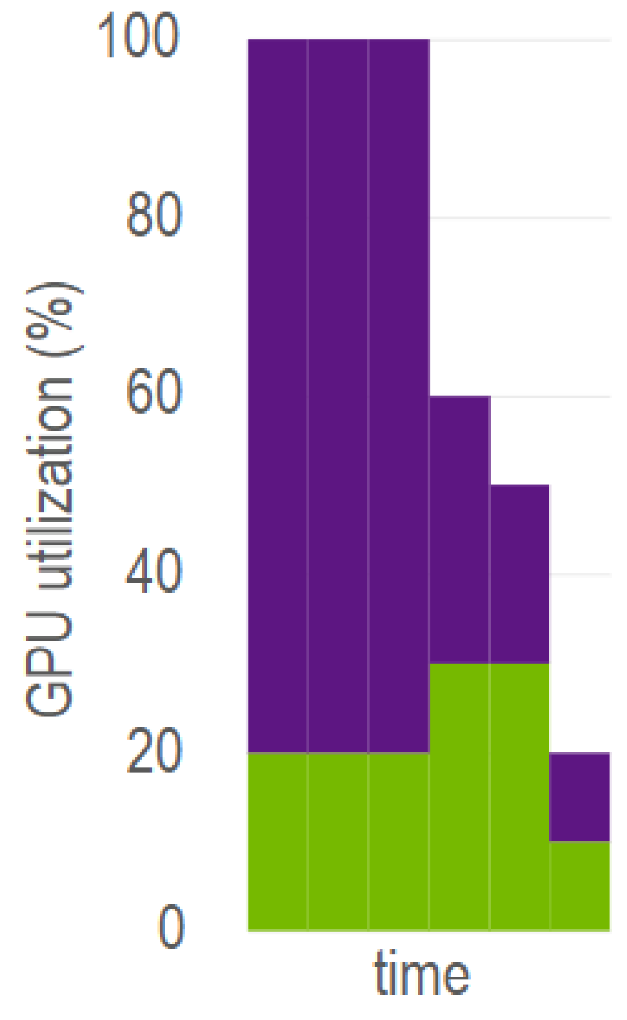
- MPS allows multiple processes to run on the GPU at the same time without time-slicing
  - as if work submitted by a single process in different streams
- By default, no static partitioning of resources or QoS
  - MPS clients contend for all SM resources
- No error isolation (unlike MIG)
  - e.g., an Illegal memory access error can affect all processes
- Up to 48 MPS clients per physical GPU (Volta+) depending on:
  - CUDA_DEVICE_MAX_CONNECTIONS env. var., each client's memory requirements, etc.

    Can get "CUDA-capable device(s) is/are busy or unavailable" error, if attempting more than supported
- To use, start MPS daemon before launching your MPS clients

Processes A and B
run at the same time
(default compute mode, no MPS)



Processes A and B
run at the same time
with MPS



MPS

How to use MPS

- Set appropriate environment variables:
  - export CUDA_VISIBLE_DEVICES=0 Select GPU to use; selected 0; could also specify GPU-UUID (incl. MIG instance)
  - export CUDA_MPS_PIPE_DIRECTORY=<accessible pipe path> Default dir. is /tmp/nvidia-mps
  - export CUDA_MPS_LOG_DIRECTORY=<accessible log path> Default dir. Is /var/log/nvidia-mps

- Recommended to set relevant GPU compute mode to exclusive:
  - sudo –E nvidia-smi -i 0 –c EXCLUSIVE_PROCESS
  - Possible –c or –compute-mode options are 0/DEFAULT, 2/PROHIBITED, 3/EXCLUSIVE_PROCESS
    - If your GPU is in EXCLUSIVE_PROCESS mode and no MPS service is running, then one process will successfully launch a GPU kernel, but for the other process a "cudaErrorDevicesUnavailable=46" error will be returned

- Start daemon and take advantage of MPS
  - sudo –E nvidia-cuda-mps-control –d
  - … can run process(es) under MPS
    - If an application had started, you'd see an nvidia-cuda-mps-server process, started by the CUDA MPS control daemon, under nvidia-smi
  - Shutdown daemon once done via sudo –E echo quit | sudo –E nvidia-cuda-mps-control

- From CUDA 12.4 and on, you can programmatically check if this process is an MPS client via

```
int mpsEnabled = 0;
CUresult res = cuDeviceGetAttribute(&mpsEnabled, CU_DEVICE_ATTRIBUTE_MPS_ENABLED, device));
```

Assuming res = CUDA_SUCCESS, mpsEnabled will be 1, if this process is an MPS client

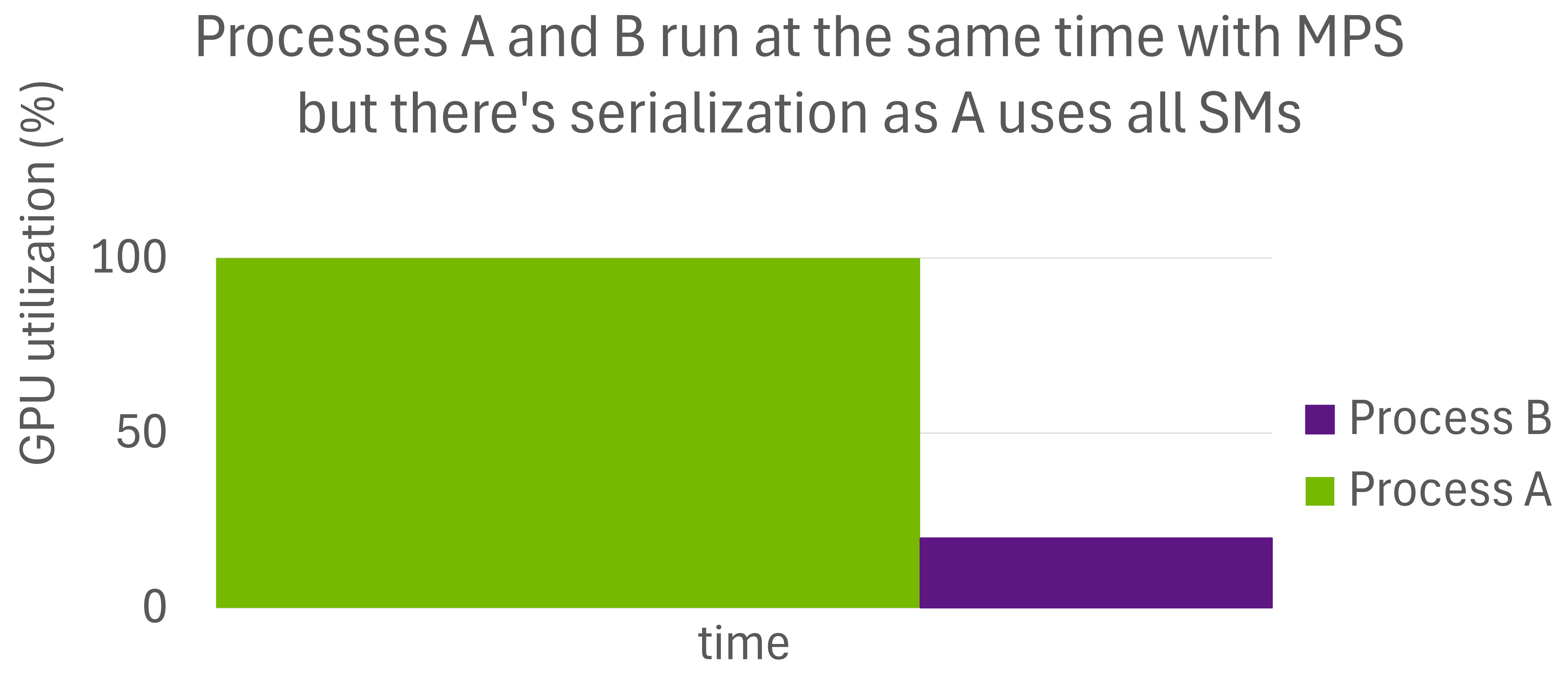
Resource Provisioning with MPS

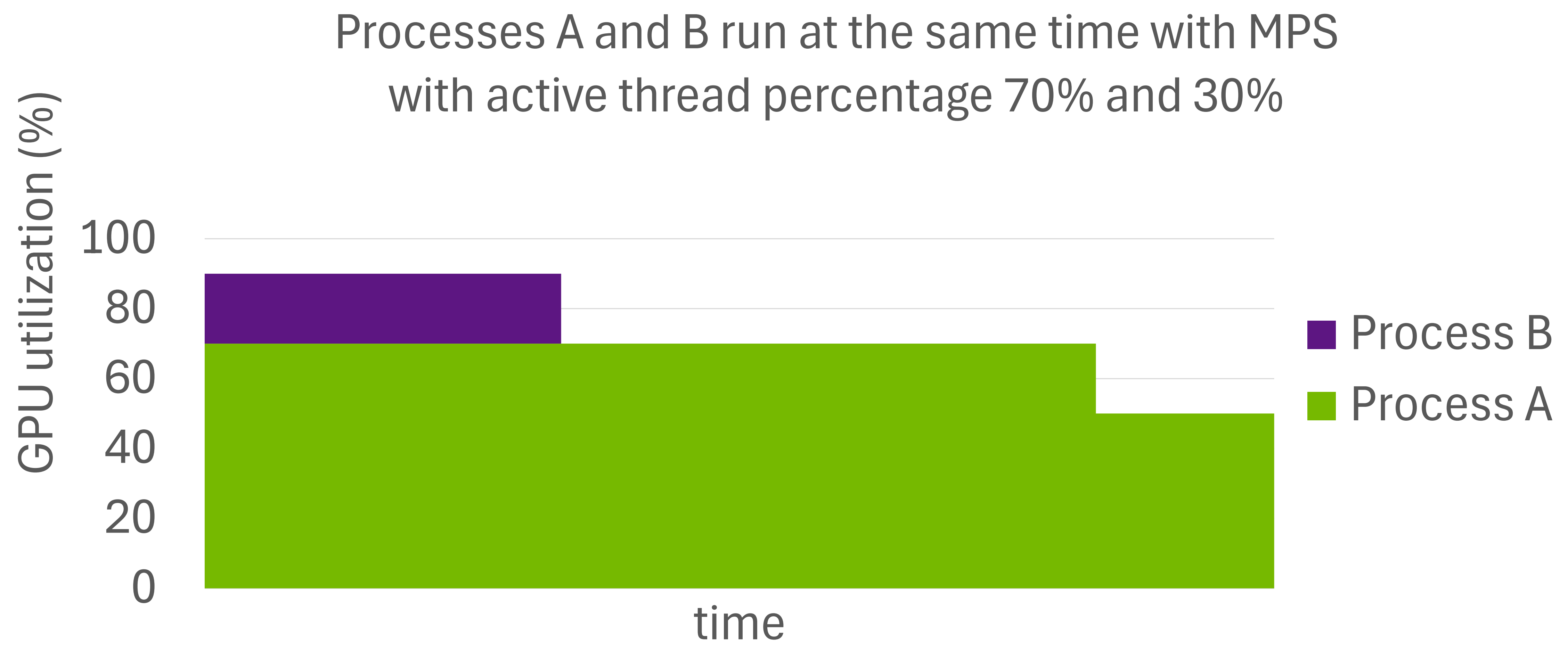
When to use

- Different MPS clients can contend, by default, for all GPU resources such as SMs, memory, etc.
  - If processes A and B each underutilized the GPU → MPS is perfect fit!
  - If A and B each fully utilize the GPU, but none is latency sensitive → individual duration longer with MPS, but can still overlap
  - If A and B, together, fully utilize the GPU and at least one is latency sensitive → latency sensitive app may suffer; no QoS
    - Solution: MPS resource provisioning

○ MPS resource provisioning (active thread percentage) places an upper limit on the % of SMs a client process can use

- Can provide some QoS for latency sensitive cases, preventing one MPS process (i.e., its GPU work) from using all SMs





Process B is latency sensitive

Active Thread Percentage with MPS

How to set

- Set via CUDA_MPS_ACTIVE_THREAD_PERCENTAGE env. variable (up to 100.0)

When set	Affects	Notes
Before MPS control daemon is launched	All future MPS clients	Query default via sudo –E echo get_default_active_thread_percentage | sudo –E nvidia-cuda-mps-control
When launching an MPS client CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=80 ./app	This client process	Limit can't be > than what MPS control daemon enforces

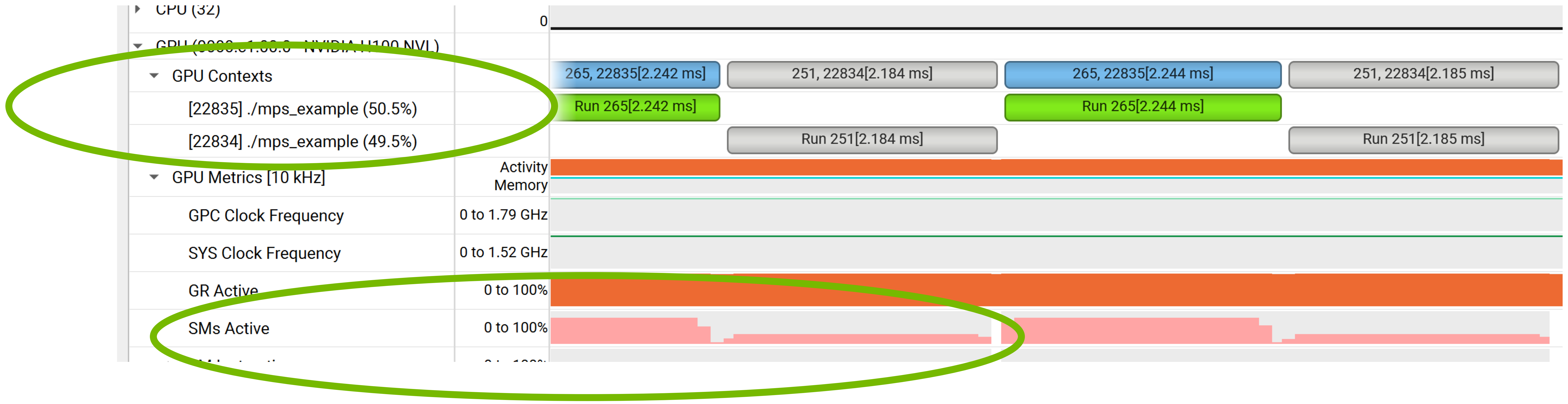
- cudaDevAttrMultiProcessorCount attribute will show active thread percentage * total GPU SMs limit for this process

- What value to use? Consider both the active thread % of each process in isolation and the Σ(active thread %) across all processes and experiment depending on your workload!
  - Reminder: $\Sigma_{\text{(active thread \%)}}$ > 100 & individual active thread % < 100 => oversubscribing while no single process can use all SMs
  - If homogeneous processing, using a uniform percentage may be a good approach.
  - If the applications underutilize the GPU even together → leave the default (100)
  - If no application is latency sensitive → can leave the default or set to < 100 to ensure no process can use all SMs
  - If one or more processes are latency sensitive → consider avoiding oversubscription to maintain some QoS

Nsight Systems Profiling Notes

Timeslicing, GPU Metrics

- Can see time-slicing behavior (no MPS and default compute mode) in Nsight Systems via –gpuctxsw=true



One process
runs at a time

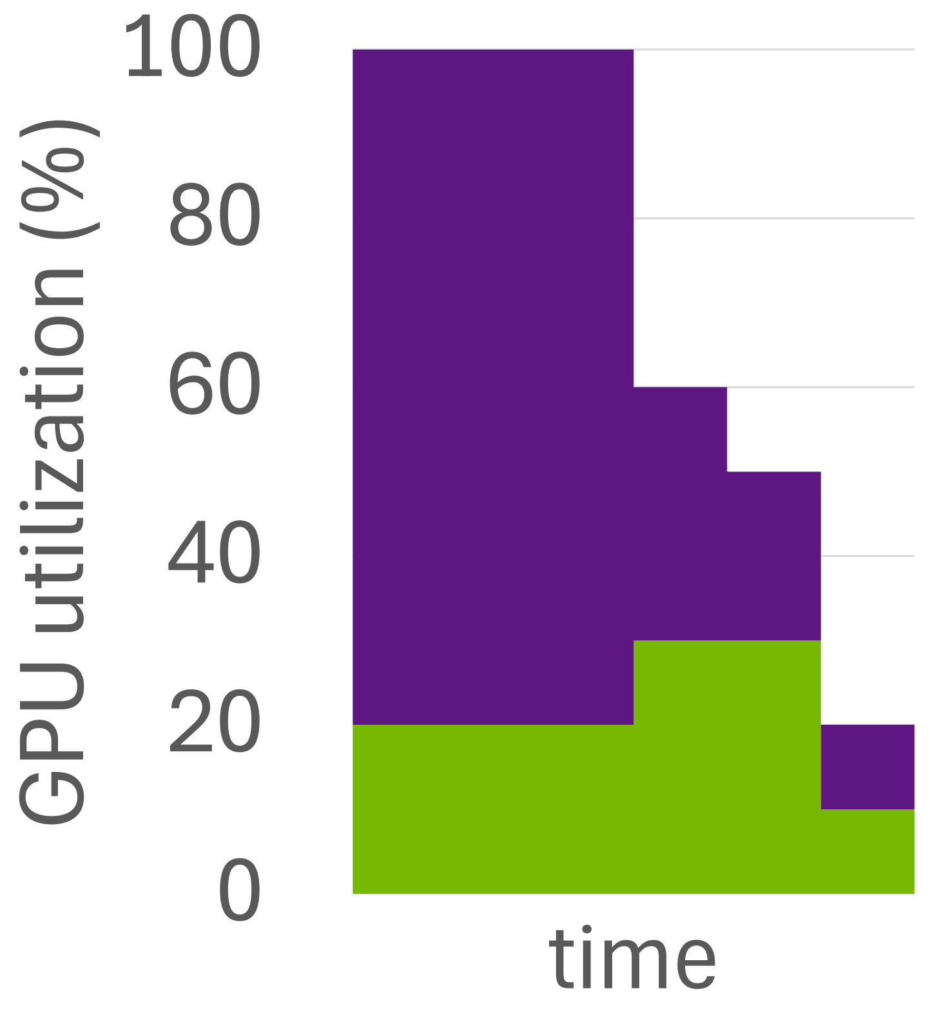
Profile collected via nsys profile --gpuctxsw=true --gpu-metrics-devices=0 ./launch_script

Example used default timeslice: sudo nvidia-smi compute-policy --set-timeslice 0 (possible values [0, 3])

- SMs Active (collected via GPU metrics from Nsight Systems) can provide useful insights into GPU utilization
  - Carefully review metric definition! SMs Active: "The ratio of cycles SMs had at least 1 warp in flight (allocated on SM) to the number of cycles as a percentage. [..] A value of 50% can indicate some gradient between all SMs active 50% of the sample period or 50% of SMs active 100% of the sample period."
  - Reminder: GPU utilization shown via nvidia-smi means "Percent of time over the past sample period during which one or more kernels was executing on the GPU." Could be 100% even if only one kernel was running on one SM
  - Metric collection can have some overhead and you may need to adjust sampling frequency

MPS

Summary



Partition Type:	Dynamic
When to enable/configure:	Start MPS service before launching an application Can further configure during application launch
Config. Options	Active thread percentage & other env. variables, as needed (e.g., CUDA_MPS_PINNED_DEVICE_MEM_LIMIT to limit how much device memory can be allocated)
Application changes needed:	No (unless you use driver API for context creation)
Use cases:	Different processes No error isolation needed

References/Further reading:

- Multi-Process Service
- GTC 2022 talk: Optimizing GPU Utilization: Understanding MIG and MPS
- Also see CUDA: New Features and Beyond [S72383]

Green Contexts

Green Contexts (GCs)

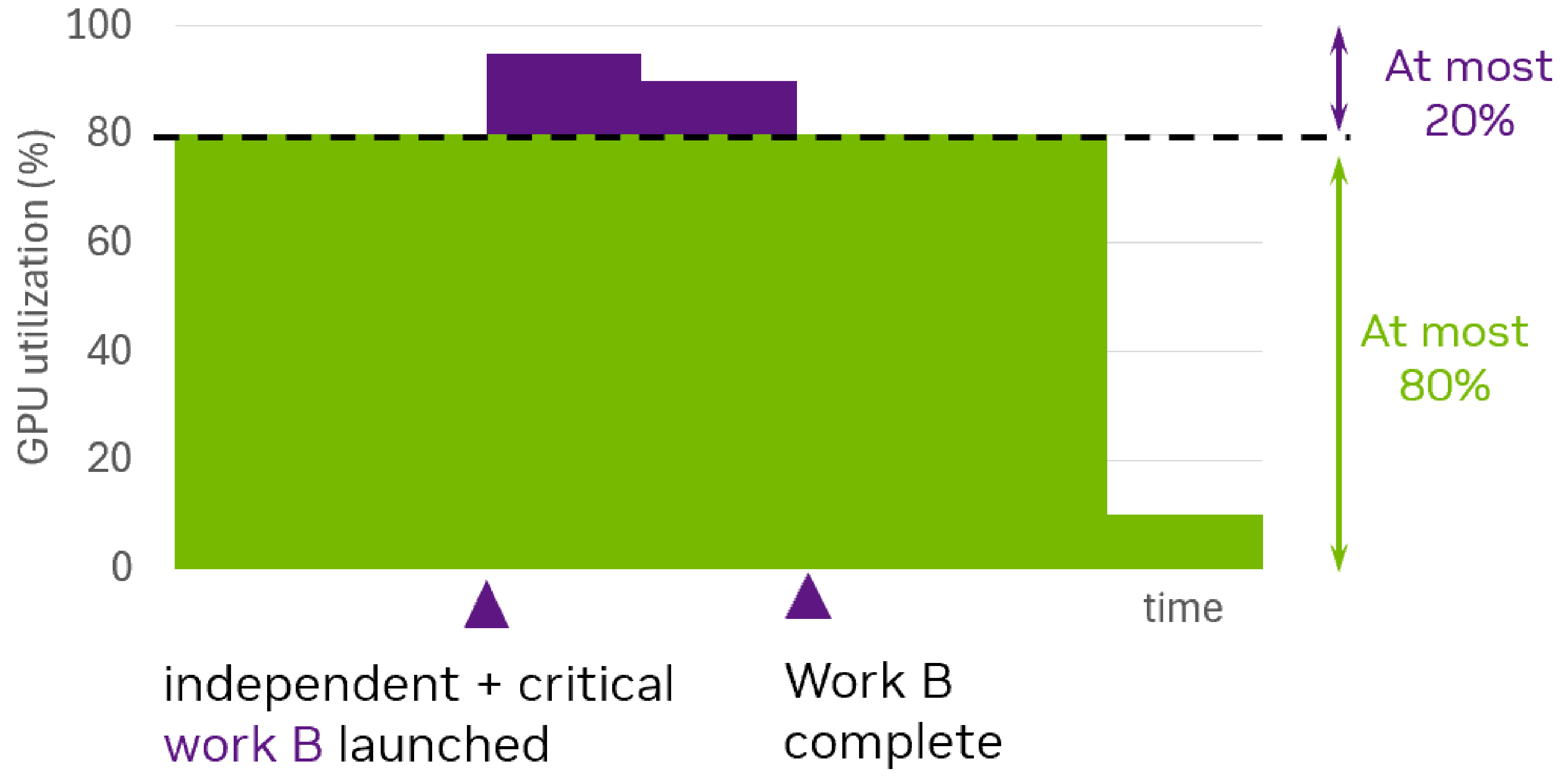
Overview

A green context (GC) is associated with a set of GPU resources, currently SMs

Green contexts allow us to spatially partition SMs, so GPU work (e.g., kernel, graphs) from GC can only use these SMs

Can have multiple green contexts within the same application

Using 2 green contexts with 80% -20% SM split



Green Contexts functionality is available via the CUDA Driver API (-lcuda)

- Requires small application changes (e.g., green context creation), but no GPU code changes (e.g., no kernel changes)

GC examples will assume CUDA 12.8

Green Contexts vs. MPS

A Comparison

- MPS primarily targets different processes, while Green contexts target an individual process
  - Don’t need to start MPS service to use GCs

- Assume MPS with 80% active thread percentage vs. Green Context with 80 SMs available resource (GPU with 100 SMs)
  - Both can use at most 80 SMs
  - MPS process can use any 80 SMs; SMs used can vary over time
  - GC can use the specific 80 SMs, spatially partitioned during GC creation

- Oversubscribed example with 3 processes or GCs (still GPU with 100 SMs, for convenience)
  - 3 MPS processes with active thread percentages 80, 20 and 40 vs. 3 green contexts with access to 80, 20 and 40 SMs.
  - The user could control, via how the GCs are created, how many SMs may be shared across the 3 GCs. Not possible with MPS.

Green Contexts

Device Resource & Resource Descriptor

- GCs spatially partition GPU SM resources, allowing GC work submitted via kernels/graphs etc. to target only them

- CUdevResource

```
struct {
    CUdevResourceType type; // enum with CU_DEV_RESOURCE_TYPE_INVALID=0, CU_DEV_RESOURCE_TYPE_SM=1
    union {
        CUdevSmResource sm; // struct with unsigned int smCount
    };
};
```

- CUdevResourceDesc: descriptor encapsulates resources

Green Context Creation Example

Overview

- Steps for green context creation:
  1. Get available GPU resources
  2. Split the resources (SMs) into one or more homogeneous partitions and a remaining partition
  3. Create a resource descriptor combining, if needed, different partitions.
  4. Create a green context from the descriptor

- After the green context has been created:
  - you can create CUDA streams belonging to that green context or set the green context as current context
  - Any work subsequently launched on such a stream will only have access to this context’s SM resources
    - incl. launching kernel via <<< >>> or using any of the CUDA driver or runtime APIs.

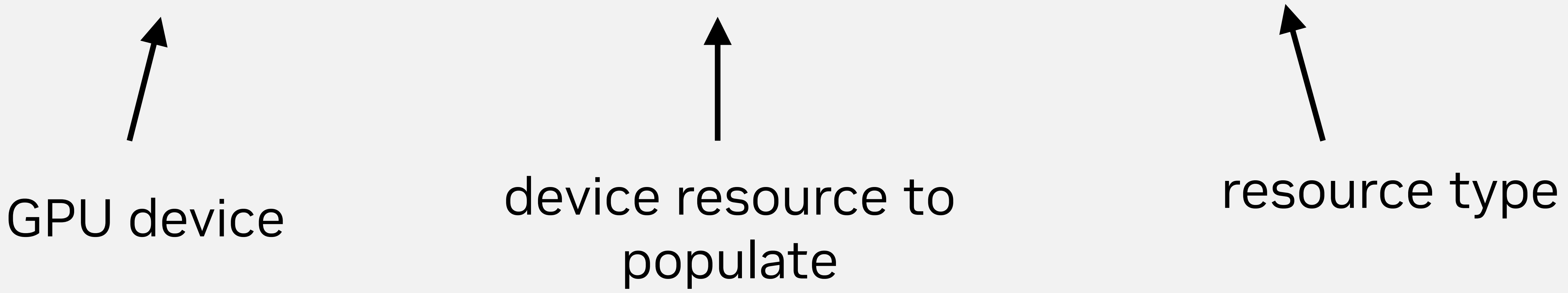
Green Context Creation: Get available SM resources

Step 1

- Get GPU SM resources we can partition and populate CUdevResource struct
  - From a device: CUresult **cuDeviceGetDevResource** (CUdevice device, CUdevResource* resource, CUdevResourceType type)
  - From a context: CUresult **cuCtxGetDevResource** (CUcontext hCtx, CUdevResource* resource, CUdevResourceType type)
  - From a green context: CUresult **cuGreenCtxGetDevResource** (CUgreenCtx hCtx, CUdevResource* resource, CUdevResourceType type)

Usually, your starting point will be a GPU device:

```
CUdevice current_device;
CU_CHECK(cuDeviceGet(&current_device, 0));

CUdevResource initial_resources = {};

CU_CHECK(cuDeviceGetDevResource(current_device, &initial_resources, CU_DEV_RESOURCE_TYPE_SM));
```



GPU device

device resource to populate

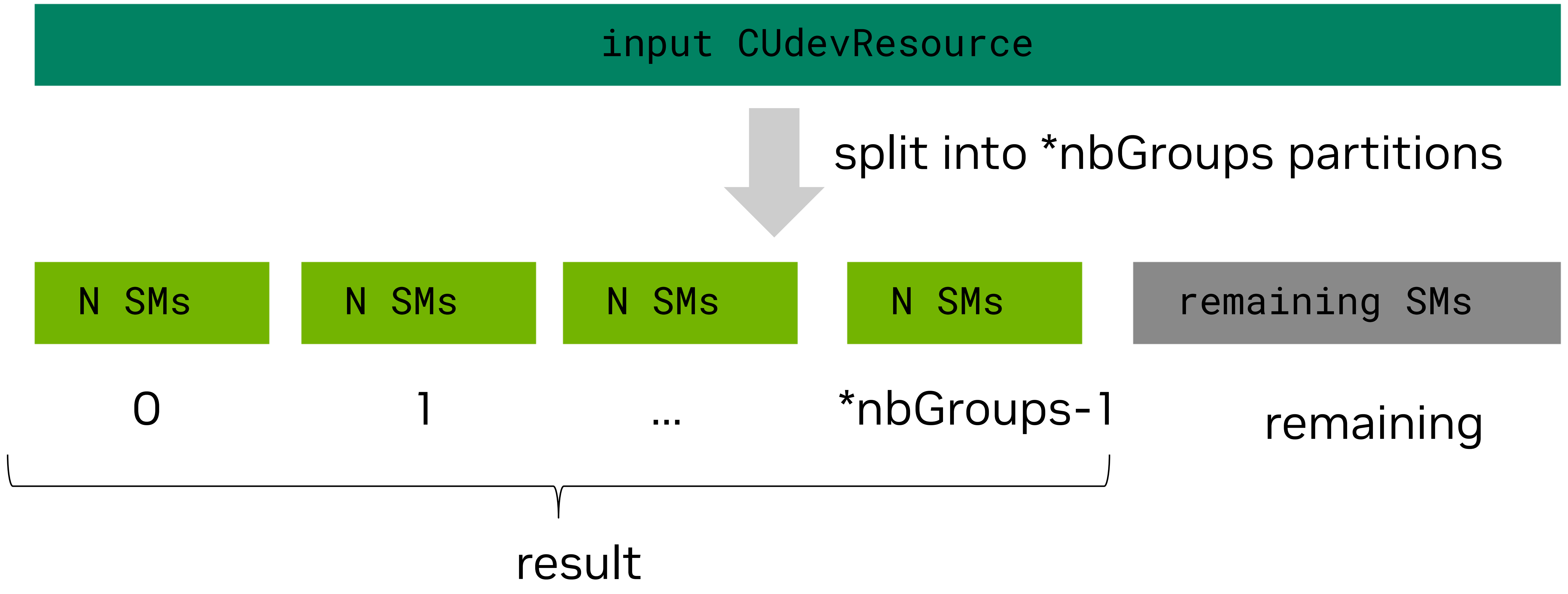
resource type

```
printf(“Initial resources: %d SMs\n”, initial_resources.sm.smCount);
```

Green Context Creation – Split SM resources

Step 2

Statically split available CUdevResource SM resources using **cuDevSmResourceSplitByCount()** API into one or more **homogeneous** partitions, with potentially some SMs left over in the remaining partition.



```
CUresult cuDevSmResourceSplitByCount(CUdevResource* result, unsigned int* nbGroups, const CUdevResource* input,
                                     CUdevResource* remaining, unsigned int  useFlags, unsigned int  minCount)
```

- Request to create *nbGroups homogeneous groups with minCount SMs each
- Outcome: updated *nbGroups (can be <= than requested) with N SMs each (N can be >= minCount)

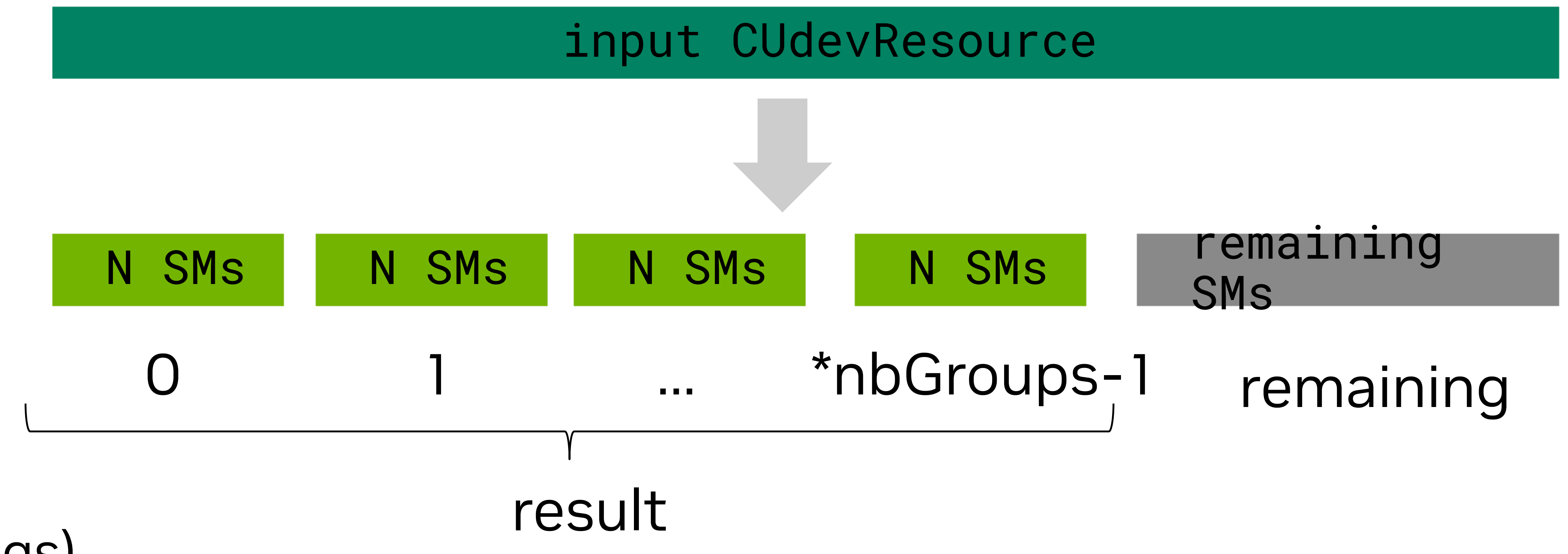
Green Context Creation – Split SM resources (cont'd)

Step 2

- CUresult cuDevSmResourceSplitByCount(CUdevResource* result, unsigned int* nbGroups, const CUdevResource* input, CUdevResource* remaining, unsigned int useFlags, unsigned int minCount)

***nbGroups**

- Starts as number of requested homogeneous groups (>= 1)
- Can be updated to a smaller number as part of the call



input CUdevResource

N SMs | N SMs | N SMs | N SMs | remaining SMs

0 | 1 | … | *nbGroups-1 | remaining

result

minCount (>= 0)

- Requested number of SMs per partition. Actual value (N in image) can be greater due to some granularity & min. value requirements
- E.g., Hopper: min of 8 SMs and multiple of 8 (can be changed with useFlags)

Requested			Actual (for GH200 w/ 132 SMs)		
***nbGroups**	**minCount**	**useFlags**	***nbGroups with N SMs**	**Remaining SMs**	**Reason**
2	72	0	1 group of 72 SMs	60	cannot exceed 132 SMs
6	11	0	6 groups with 16 SMs each	36	multiple of 8 requirement
6	11	CU_DEV_SM_RESOURCE_SPLIT_IGNORE_SM_COSCHEDULING	6 groups with 12 SMs each	60	Lowered to multiple of 2 req.
2	1	0	2 groups with 8 SMs each	116	min. 8 SMs requirement

Green Context Creation – Split SM Resources (cont’d)

Step 2 (Example Use)

- Example requesting to split resources in 5 groups of 8 SMs each

```
CUdevResource avail_resources = {};
// Code that has populated avail_resources not shown

unsigned int min_SM_count = 8;
unsigned int actual_split_groups = 5; // may be updated

CUdevResource actual_split_result[5] = {{}, {}, {}, {}, {}, {}};
CUdevResource remaining_partition = {};

CU_CHECK(cuDevSmResourceSplitByCount(&actual_split_result,
                                     &actual_split_groups,
                                     &avail_resources,
                                     &remaining_partition,
                                     0 /*use_flags */,
                                     min_SM_count));

printf(“Split %d SMs into %d groups with %d SMs each and a remaining group with %d SMs\n”,
        avail_resources.sm.smCount, actual_split_groups, actual_split_result[0].sm.smCount,
        remaining_partition.sm.smCount);
```

- Can use result=nullptr to query number of groups that would be created.
- Can use remaining = nullptr if you don’t care about these SMs

Green Context Creation – Generate Descriptor

Step 3

- After you have split your resources, you need to create a resource descriptor for each set of resources you plan to use.

  CUresult cuDevResourceGenerateDesc(CUdevResourceDesc* phDesc, CUdevResource* resources,
  unsigned int nbResources)

- Example to generate resource descriptor that encapsulates 3 groups of resources

```
CUdevResource actual_split_result[5] = {};
// code to populate actual_split_result not shown


// Generate resource desc. to encapsulate 3 resources: actual_split_result[2] to [4]
CUdevResourceDesc resource_desc;
CU_CHECK(cuDevResourceGenerateDesc(&resource_desc, &actual_split_result[2], 3));
```

CUDA API shown as of CUDA 12.8 (earlier versions only supported num_resources=1)

Green Context Creation – Create a context

Step 4

- Create a green context from a resource descriptor

```
Cudevice current_device;
CUdevResourceDesc resource_desc;
// Code to populate current_device and generate resource_desc not shown



// Create a green_ctx on current_device with access to resources from resource_des
CUgreenCtx green_ctx;
CU_CHECK(cuGreenCtxCreate(&green_ctx, resource_desc, current_device, CU_GREEN_CTX_DEFAULT_STREAM));
```

- To submit work on that green context, you can:
  - explicitly create a stream for it or
  - transform the green context into a CUcontext before you set it as current and submit work

Green Contexts

Launching work

- To submit work on a green context, you can create a stream for it

```
// Create green_ctx_stream on current_device for green_ctx with priority 0
Custream green_ctx_stream;
cuGreenCtxStreamCreate(&green_ctx_stream,
                       green_ctx, current_device, CU_STREAM_NON_BLOCKING, 0 /* priority */);


my_kernel<<<grid_dim, block_dim, 0, green_ctx_stream>>>();  // will use only the SMs of green_ctx
```

- Alternatively:

```
// Convert green context into a primary context with GC’s SM resources
CUcontext ctx_from_green_context;
cuCtxFromGreenCtx (&ctx_from_green_context, green_ctx);

// Set context as current
cuCtxSetCurrent(ctx_from_green_context);

// Create streams under that context:
cudaStream_t strm;
cudaStreamCreateWithFlags(&strm, cudaStreamNonBlocking);
my_kernel<<<grid_dim, block_dim, 0, strm>>>();  // will use only the SMs of green_ctx
```

More Green Contexts driver APIs

- CUresult **cuGreenCtxRecordEvent**(CUgreenCtx hCtx, CUevent hEvent)
  - Record an event capturing all work/activities of the specified green context at the time of this call
  - FAQ: How does that compare with cudaEventRecord?
    - Equivalent if you only had a single stream in a green context and recorded an event on it.
    - But what if you had multiple streams in that green context?
      - Without this new API, you'd need to record a separate event on every green context stream and then have dependent work wait separately for all these events!

- CUresult **cuGreenCtxWaitEvent**(CUgreenCtx hCtx, CUevent hEvent);
  - Have green context wait for an event
  - More convenient to use if you need all streams in this green context wait for an event to complete. Alternative: a separate cudaStreamWaitEvent for every stream in your green context.

- CUresult **cuStreamGetGreenCtx**(CUstream hstream, CUgreenCtx* phCtx);
  - Will update phCtx to the green context associated with hstream, if any, or set to NULL otherwise.

- Curesult **cuGreenCtxDestroy**(CUgreenCtx hCtx);
  - Destroy green context

Green Contexts Example

Static Resource Partitioning enables critical work to start and complete sooner

- Example timeline:
  - Launch long running kernel (delay_kernel_us) that takes multiple waves on the full GPU on stream strm1
  - Wait on the CPU for some time, and then launch shorter critical kernel (critical_kernel) on stream strm2
  - Measure GPU durations and time from CPU launch to completion for both kernels.

- Long running kernel proxy is a delay kernel where every CTA runs for delay_us and # CTAs > total SMs.

Example Run Scenarios		Expectations	
using GCs	Stream Priorities	delay kernel's duration	When will critical kernel start?
No	Same	as if running in isolation	Possibly after all CTAs of delay kernel have completed
No	Higher priority for critical kernel	Almost as if running in isolation, if critical kernel's duration is short in comparison	Can start executing as soon as some CTAs of delay kernel complete
Yes w/ non-overlapping splits	Do not matter	Duration increase due to use of fewer SMs	Almost immediately

- Code will be available here

Green Context example (cont'd)

Nsight Systems Timelines without Green Contexts

Timeline View
Options...
0s
+418ms
+420ms
+422ms
+424ms
+426ms
71.3% Stream 13
28.6% Stream 14
0.1% Stream 15
(high prio)
22.1% Green Context 2
100.0% Kernels
0.2% Green Context 3
100.0% Kernels
Threads (9)
0 to 100%
☑ [1405] stream_and_gc_e
OS runtime libraries
CUDA API
delay_kernel_us(unsigned int)
~10ms duration
critical kernel
~50us duration
lost time
(~0.9ms)
nanosleep
cudaStreamSynchronize

delay_kernel_us launch

critical_kernel launch

Green Context example (cont'd)

Nsight Systems Timelines with Green Contexts

- Partition GPU resources in green contexts: N SMs for critical_kernel and 7*N for long running kernel (with some leftover SMs), where N is max supported given some green context constraints.
- Example shown for H100 with 132 total SMs and with N=16.

Timeline View
Options...
0s
32ms
+434ms
+436ms
+438ms
+440ms
+442ms
+44
0.1% Stream 15
22.1% Green Context 2
100.0% Kernels
0.2% Green Context 3
100.0% Kernels
Threads (9)
0 to 100%
[1405] stream_and_gc_e
OS runtime libraries
CUDA API
delay_kernel_us(unsigned int)
nanosleep
cudaEventSynchronize

~12ms duration (+2ms from before)

~95us duration; almost no lost time!

delay_kernel launch

critical_kernel launch

Green Context example (cont'd)

Green Context Resources shown in Nsight Compute

Previous example has two green contexts with 16 SMs and 7*16 SMs respectively

- Nsight Compute view from Session page shows that green contexts in this example use different SMs, as expected

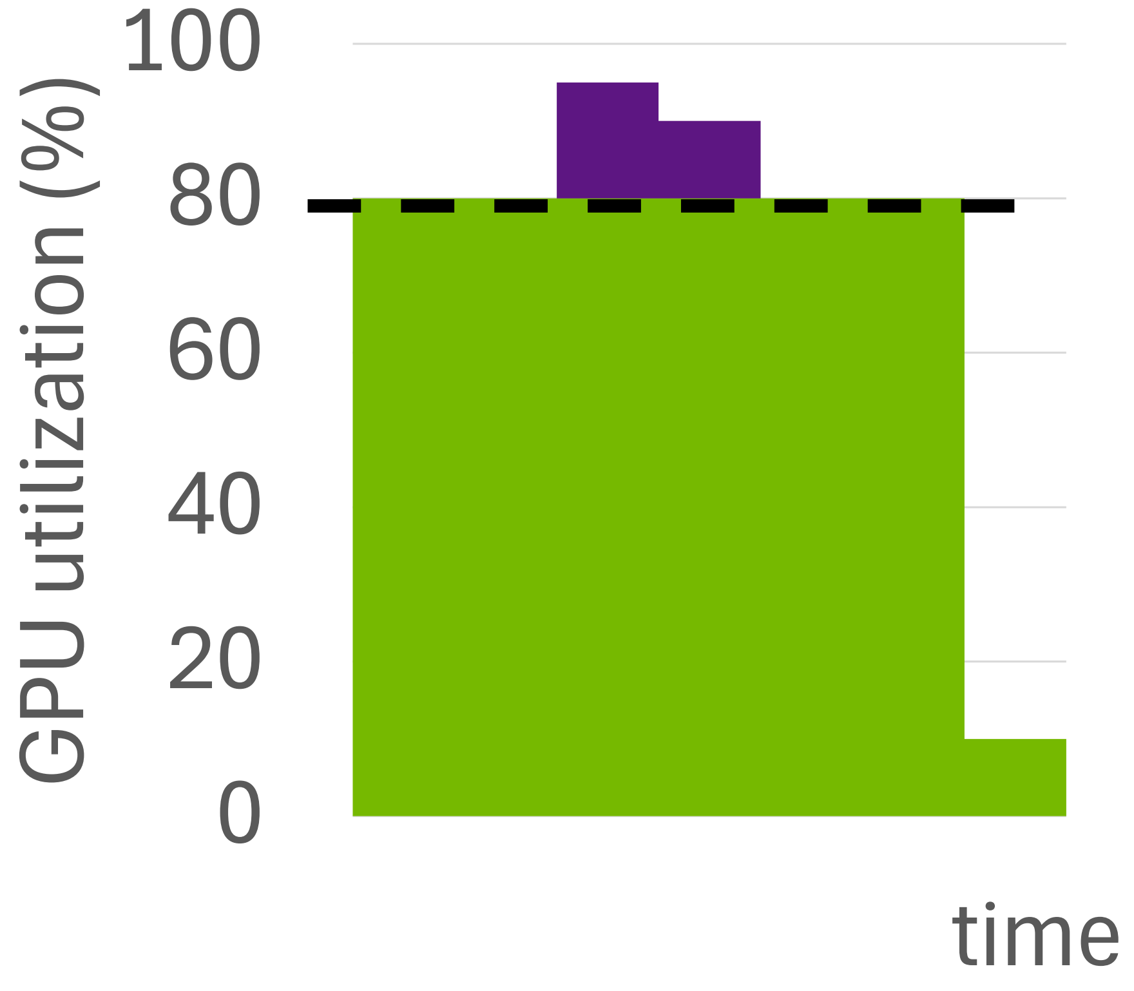
Green Context Resources

Overview of all Green Contexts with their respective resources included in the report.

Green Context ID	Result IDs	# TPCs	Enabled TPCs
2	8 ↗	56	🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩⬜🟩⬜🟩🟩⬜🟩🟩🟩⬜🟩🟩🟩🟩🟩⬜🟩🟩🟩🟩⬜🟩🟩🟩⬜🟩🟩🟩🟩⬜🟩🟩⬜🟩🟩🟩🟩🟩🟩⬜🟩
3	9 ↗	8	⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜🟩⬜🟩⬜⬜🟩⬜⬜⬜🟩⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜🟩⬜⬜⬜🟩⬜⬜⬜⬜🟩⬜⬜⬜⬜⬜⬜⬜⬜⬜🟩⬜

Green Contexts

Summary



Partition Type:	Static (SMs)
When to enable/configure:	From within your application, before launching work
Config. Options	SM count & SM overlaps determined by how you partition
Application changes needed:	Yes, but only outside kernel/GPU code.
Use cases:	Single process with different workload types; need to ensure SM resources are available for critical work

References:

- Green Contexts (CUDA Driver API)

Cluster Launch Control

Cluster Launch Control

Thread blocks

Thread block (CTA) – basic unit of problem's "work decomposition" for kernel execution:

- CTAs are completely independent of each other and do not interact (except in the case of a cooperative launch)
- Kernel launch parameters: CTA count and size (thread count):

```
        kernel<<<CTA_COUNT, THREAD_COUNT>>>(ARGUMENTS)
```

- CTAs are scheduled for execution in arbitrary order
- Driver/scheduler can interleave or overlap CTAs from different kernels, particularly, based on stream priorities
- Scheduling a CTA for execution incurs overhead
- Once a CTA starts, it runs to completion without interruption
  (time slicing between processes is possible)
  E.g., a CTA cannot be swapped out for CTAs from a higher-priority[#] kernel

[#] Kernel associated with higher-priority stream

Cluster Launch Control

How many thread blocks?

Main approaches to choosing CTA count:

- **Problem size – based** :

  Fixed/limited work per CTA

  Number of CTAs ~ problem size

```
__global__ void kernel (float* data, float alpha, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        data[i] *= alpha;
}
kernel<<<(n + 1023) / 1024, 1024>>>(data, alpha, n);
```

- **HW resources – based**
  (aka persistent kernels, e.g., grid-stride loop):

  Fixed/limited number of CTAs

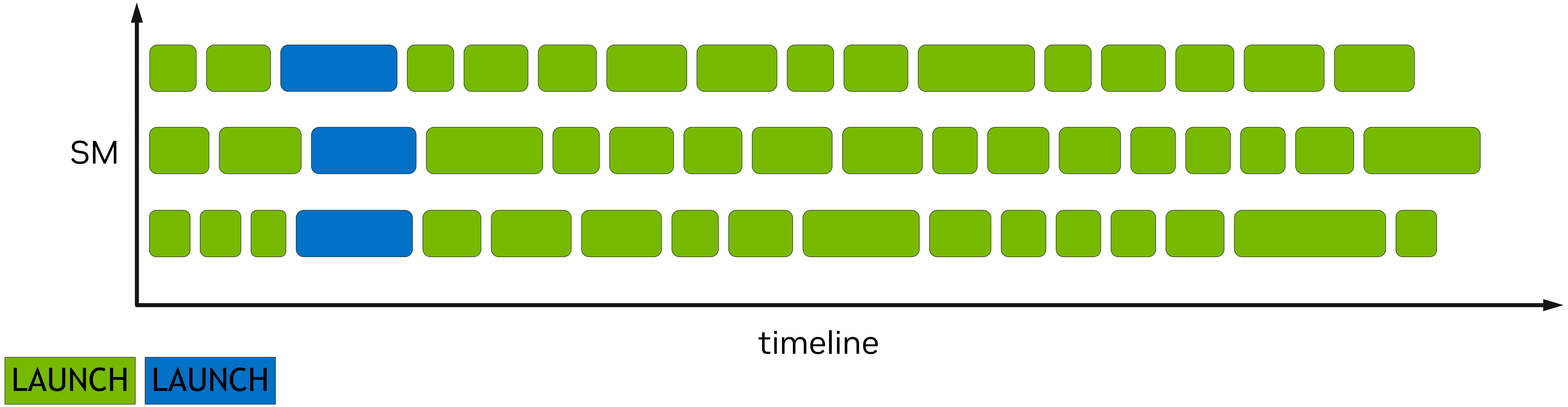
  Work per CTA ~ problem size

```
__global__ void kernel (float* data, float alpha, int n)
{
    for (i = blockIdx.x * blockDim.x + threadIdx.x;
             i < n; i += gridDim.x * blockDim.x)
        data[i] *= alpha;
}
kernel<<<sm_count * 2, 1024>>>(data, alpha, n);
```

Cluster Launch Control

Problem size – based thread block count

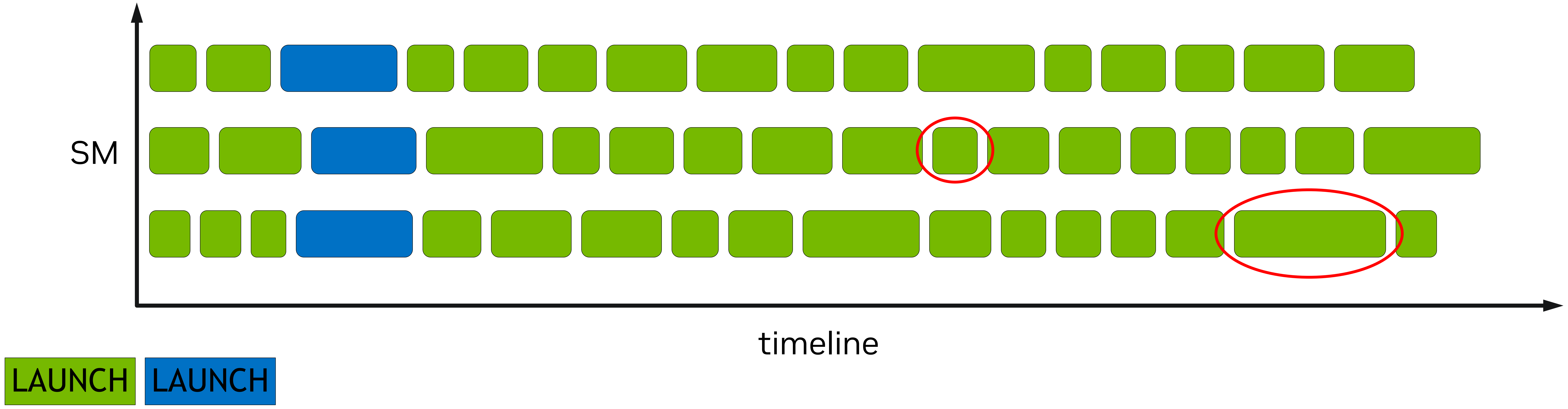
✓ **Preemption**: high-priority kernel execution “inserted” into low-priority kernel execution



Cluster Launch Control

Problem size – based thread block count

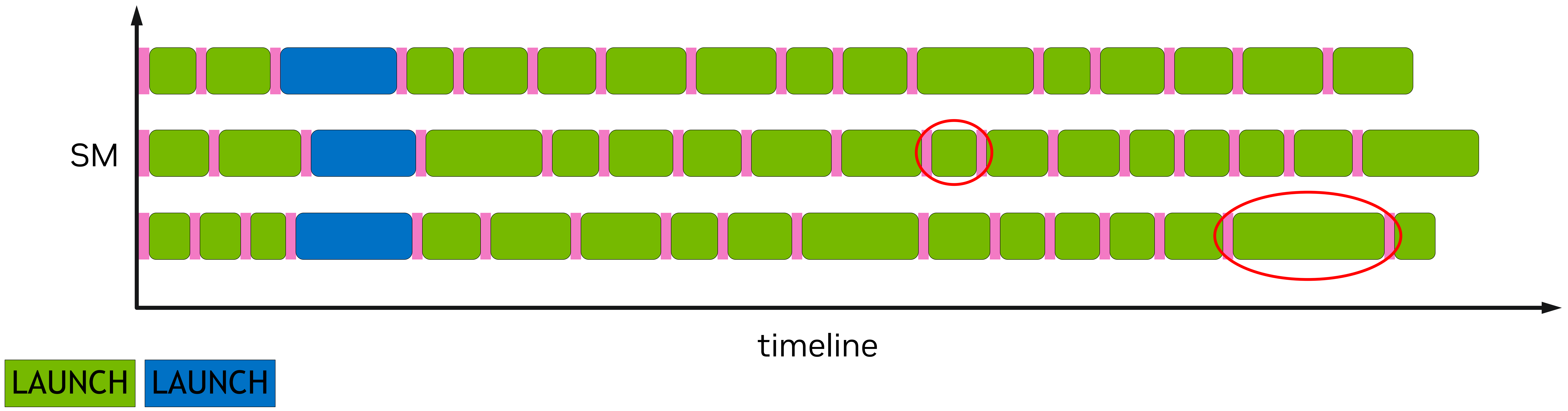
- ✓ **Preemption**: high-priority kernel execution “inserted” into low-priority kernel execution
- ✓ **Load balancing**: despite variability of individual run-times, SM run-times are similar ~ low-tail effect



Cluster Launch Control

Problem size – based thread block count

- ✓ **Preemption**: high-priority kernel execution “inserted” into low-priority kernel execution
- ✓ **Load balancing**: despite variability of individual run-times, SM run-times are similar ~ low-tail effect
- X **Scheduler overhead**



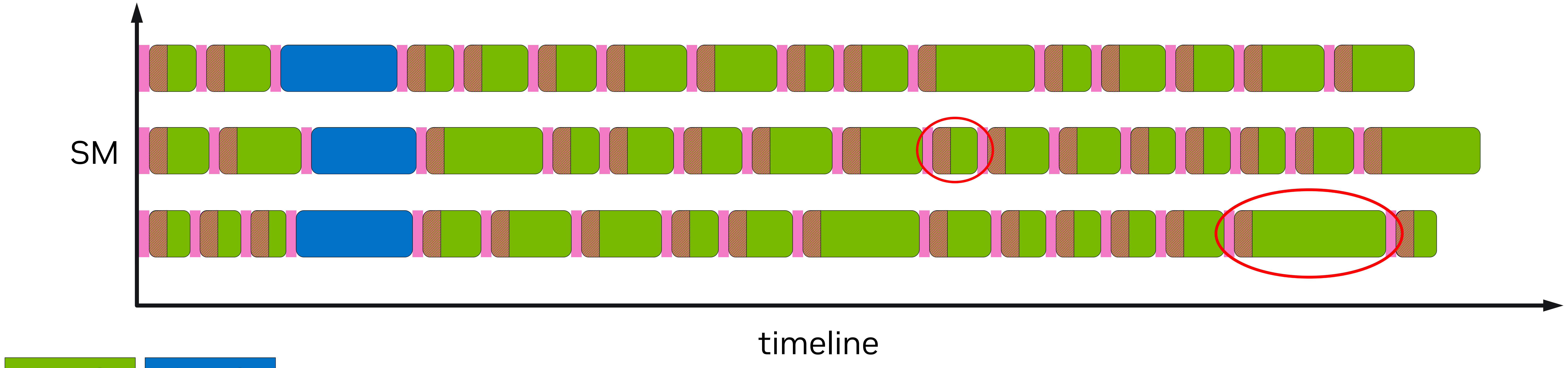
Cluster Launch Control

Problem size – based thread block count

✓ **Preemption**: high-priority kernel execution “inserted” into low-priority kernel execution

✓ **Load balancing**: despite variability of individual run-times, SM run-times are similar ~ low-tail effect

X **Scheduler overhead** and **common code overhead**

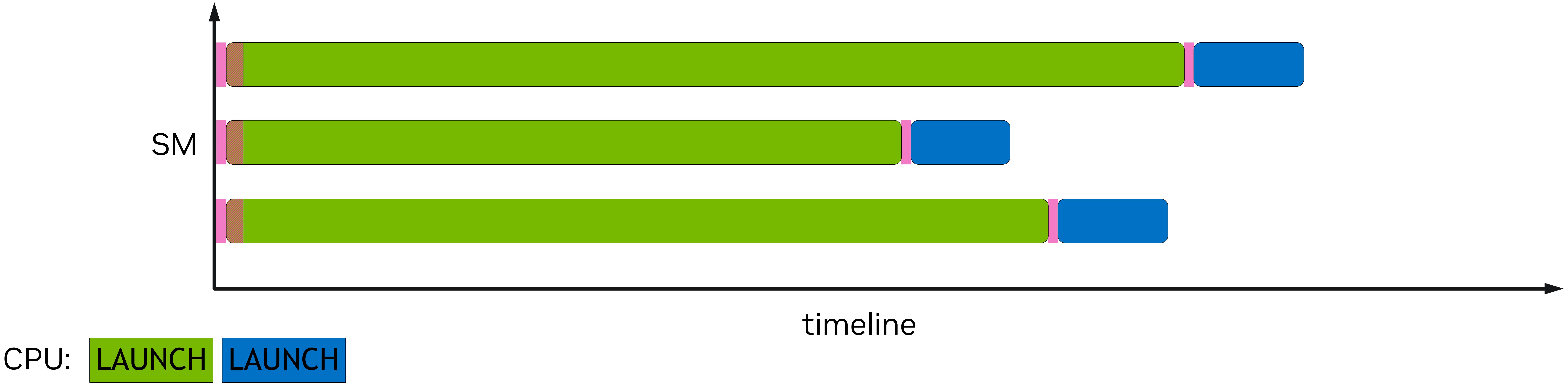


CPU: LAUNCH LAUNCH

Cluster Launch Control

Persistent kernels

- X **Preemption**: unable to execute high-priority kernel until started low-priority kernel's CTAs finish
- X **Load balancing**: variability of individual run-times => variability of SM run-times
- ✓ Scheduler overhead and **common code overhead**



Cluster Launch Control

Custom load balancing

Typical approach – atomic-based CTA counter.

```
if (threadIdx.x == 0)
    shared_counter = bid.fetch_add(1, cuda::memory_order_relaxed);
__syncthreads();
bx = shared_counter;

reset_counter<<<1, 1>>>();
kernel<<<sm_count, 1024>>>(n_blocks);
```

Disadvantages:

- Cumbersome solution, no dedicated HW support:
  - Need to reset counter before kernel call.
  - Need to launch with `sm_count` CTAs and pass actual number of CTAs as an argument
- No preemption
- Need multiple counters if kernel can be launched in parallel
- Shared memory synchronization
- For 2D/3D need to decode CTA index using divisions:

  ```
  bx = shared_counter % BX;
  by = (shared_counter / BX) % BY;
  bz = shared_counter / (BX * BY);
  ```

Cluster Launch Control

Pros/cons summary

	Problem size – based	HW resources – based	Custom (atomic counter)
Preemption	☑	☒	☒
Load balancing	☑	☒	☑
Overhead	☒	☑	☑
Ease of use	☑	☑	☒

Cluster Launch Control

Best of both worlds

	Problem size – based	HW resources – based	Custom (atomic counter)	Cluster Launch Control
Preemption	☑	☒	☒	☑
Load balancing	☑	☒	☑	☑
Overhead	☒	☑	☑	☑
Ease of use	☑	☑	☒	☑

Cluster Launch Control:

- Available from **Blackwell** (and **CUDA 12.8**)
- **Work-stealing** approach:

  Kernel requests index of a queued CTA:

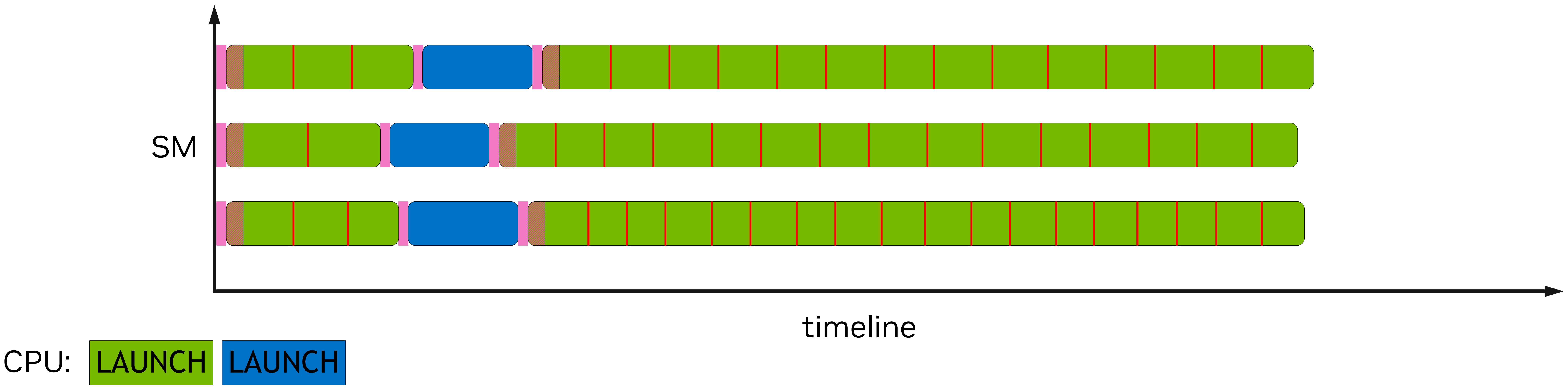
    On SUCCESS, index is removed from a pool of available CTAs,
    and the kernel executes work of a “stolen” (**cancelled**) CTA in a work-stealing loop

    On FAILURE, kernel typically exits

Cluster Launch Control

Best of both worlds

	Problem size – based	HW resources – based	Custom (atomic counter)	Cluster Launch Control
Preemption	☑	☒	☒	☑
Load balancing	☑	☒	☑	☑
Overhead	☒	☑	☑	☑
Ease of use	☑	☑	☒	☑



Cluster Launch Control

API introduction

To cancel a CTA:

1. Asynchronously request cancellation from **one** thread into `__shared__` memory result
2. Synchronize request with `__shared__` memory barrier based on transaction count
3. Check synchronization result for success
4. Extract CTA index from synchronization result

Note: cancellation requests from multiple threads is possible,
but **not recommended** and not required for typical workflows (cancellation is low latency)

Cluster Launch Control

API example

```
__shared__ uint4 result;  opaque cancellation result
__shared__ __mbarrier_t bar; } completion barrier and it’s phase
int phase = 0;
constexpr int tx_count = sizeof(uint4);

if (threadIdx.x == 0)
    ptx::mbarrier_init(&bar, 1);
__syncthreads();

// <<< PROLOGUE >>>

int bx = blockIdx.x;
while (true) {
    if (threadIdx.x == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        ptx::clusterlaunchcontrol_try_cancel(&result, &bar);
        ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta, ptx::space_shared, &bar, tx_count);
    }

    // <<< THREAD BLOCK bx COMPUTATION >>>

    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cta, &bar, phase))
    {}

    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
    if (!success)
        break;

    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    phase ^= 1;
    __syncthreads();
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_shared, ptx::scope_cluster);
}

// <<< EPILOGUE >>>
```

Cluster Launch Control

API example

```
__shared__ uint4 result;
__shared__ __mbarrier_t bar;
int phase = 0;
constexpr int tx_count = sizeof(uint4);
if (threadIdx.x == 0)
    ptx::mbarrier_init(&bar, 1);
__syncthreads();
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    if (threadIdx.x == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        ptx::clusterlaunchcontrol_try_cancel(&result, &bar);
        ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta, ptx::space_shared, &bar, tx_count);
    }
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cta, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    phase ^= 1;
    __syncthreads();
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_shared, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

single arrival

Cluster Launch Control

API example

```
__shared__ uint4 result;
__shared__ __mbarrier_t bar;
int phase = 0;
constexpr int tx_count = sizeof(uint4);
if (threadIdx.x == 0)
    ptx::mbarrier_init(&bar, 1);
__syncthreads();
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    if (threadIdx.x == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        ptx::clusterlaunchcontrol_try_cancel(&result, &bar);
        ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta, ptx::space_shared, &bar, tx_count);
    }
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cta, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    phase ^= 1;
    __syncthreads();
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_shared, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

transaction count – based completion

Cluster Launch Control

API example

```
__shared__ uint4 result;
__shared__ __mbarrier_t bar;
int phase = 0;
constexpr int tx_count = sizeof(uint4);
if (threadIdx.x == 0)
    ptx::mbarrier_init(&bar, 1);
__syncthreads();
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    if (threadIdx.x == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        ptx::clusterlaunchcontrol_try_cancel(&result, &bar);
        ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta, ptx::space_shared, &bar, tx_count);
    }
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cta, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    phase ^= 1;
    __syncthreads();
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_shared, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

asynchronous (“in flight”) request while running previous CTA’s computation

Cluster Launch Control

API example

```
__shared__ uint4 result;
__shared__ __mbarrier_t bar;
int phase = 0;
constexpr int tx_count = sizeof(uint4);
if (threadIdx.x == 0)
    ptx::mbarrier_init(&bar, 1);
__syncthreads();
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    if (threadIdx.x == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        ptx::clusterlaunchcontrol_try_cancel(&result, &bar);
        ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta, ptx::space_shared, &bar, tx_count);
    }
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cta, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    phase ^= 1;
    __syncthreads();
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_shared, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

cancellation completion

Cluster Launch Control

API example

```
__shared__ uint4 result;
__shared__ __mbarrier_t bar;
int phase = 0;
constexpr int tx_count = sizeof(uint4);
if (threadIdx.x == 0)
    ptx::mbarrier_init(&bar, 1);
__syncthreads();
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    if (threadIdx.x == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        ptx::clusterlaunchcontrol_try_cancel(&result, &bar);
        ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta, ptx::space_shared, &bar, tx_count);
    }
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cta, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    phase ^= 1;
    __syncthreads();
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_shared, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

__syncthreads does not fence async proxy

Cluster Launch Control

API example – optimizations

Use double-buffering
to get rid of overwrite protection
(__syncthreads)

```
__shared__ uint4 result[2];
__shared__ __mbarrier_t bar;
int phase = 0;
int const tx_count = (threadIdx.x == 0) ? sizeof(uint4) : 0;

if (threadIdx.x == 0)
    ptx::mbarrier_init(&bar, blockDim.x);
__syncthreads();

// <<< PROLOGUE >>>

int bx = blockIdx.x;
while (true) {
    if (threadIdx.x == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        cg::invoke_one(cg::coalesced_threads(), [&](){ptx::clusterlaunchcontrol_try_cancel(&result[phase], &bar);});
    }
    ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta, ptx::space_shared, &bar, tx_count);

    // <<< THREAD BLOCK bx COMPUTATION >>>

    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cta, &bar, phase))
    {}

    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result[phase]);
    if (!success)
        break;

    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result[phase]);
    phase ^= 1;
 // ~~__syncthreads();~~
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_shared, ptx::scope_cluster);
}

// <<< EPILOGUE >>>
```

Cluster Launch Control

API example – optimizations

Avoid peeling loop (for uniform instruction)
See explanation in GTC’24 optimization tutorial [S62192],
“Asynchronous Data Copies – TMA Details” section

```
__shared__ uint4 result[2];
__shared__ __mbarrier_t bar;
int phase = 0;
int const tx_count = (threadIdx.x == 0) ? sizeof(uint4) : 0;
if (threadIdx.x == 0)
    ptx::mbarrier_init(&bar, blockDim.x);
__syncthreads();
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    if (threadIdx.x == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        cg::invoke_one(cg::coalesced_threads(), [&](){ptx::clusterlaunchcontrol_try_cancel(&result[phase], &bar);});
    }
    ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta, ptx::space_shared, &bar, tx_count);
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cta, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result[phase]);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result[phase]);
    phase ^= 1;
 // __syncthreads();
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_shared, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

Cluster Launch Control

API example – optimizations

```
__shared__ uint4 result[2];
__shared__ __mbarrier_t bar;
int phase = 0;
int const tx_count = (threadIdx.x == 0) ? sizeof(uint4) : 0;
if (threadIdx.x == 0)
    ptx::mbarrier_init(&bar, blockDim.x);
__syncthreads();
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    if (threadIdx.x == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        cg::invoke_one(cg::coalesced_threads(), [&](){ptx::clusterlaunchcontrol_try_cancel(&result[phase], &bar);});
    }
    ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta, ptx::space_shared, &bar, tx_count);
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cta, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result[phase]);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result[phase]);
    phase ^= 1;
 // __syncthreads();
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_shared, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

Adjust accordingly for1D in y/z or 2D/3D CTAs

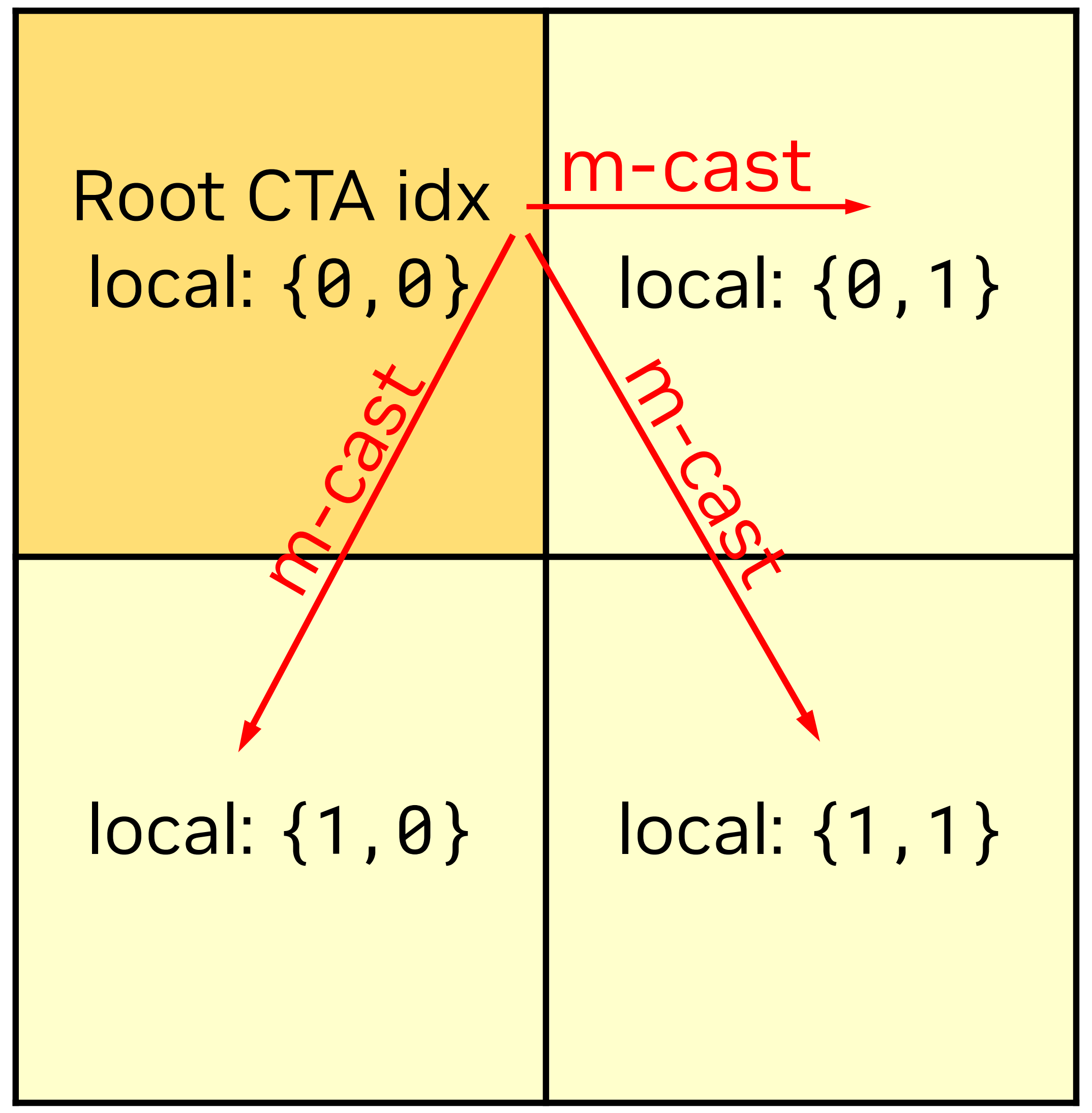
Cluster Launch Control

API introduction – cluster case

To cancel a CTA index:

1. Asynchronously request cancellation from **one** thread **of a cluster** (any CTA)
   Same cancellation result is multicasted into **local** `__shared__` memory of **each CTA of a cluster**
2. Synchronize request with **local** `__shared__` memory barrier of **each CTA of a cluster**
   based on transaction count
3. Check synchronization result for success
4. Extract **root** CTA index from synchronization result
5. Add local CTA offset to **root** CTA index
6. Todo: have all CTA started (via barrier) before first cancel

2x2 cluster



Note: cancelling requests from multiple threads is possible,
but **not recommended** and not required for typical workflows (cancellation is low latency)

Cluster Launch Control

API example – cluster case

```
__shared__ uint4 result;
__shared__ __mbarrier_t bar;
int phase = 0;
constexpr int tx_count = sizeof(uint4);
if (threadIdx.x == 0) {
    ptx::mbarrier_init(&bar, 1);
    ptx::fence_mbarrier_init(ptx::sem_release, ptx::scope_cluster);
}
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    cg::cluster_group::sync();
    if (cg::cluster_group::thread_rank() == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        cg::invoke_one(cg::coalesced_threads(), [&](){ptx::clusterlaunchcontrol_try_cancel_multicast(&result, &bar);});
    }
    if (threadIdx.x == 0)
        ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cluster, ptx::space_shared, &bar, tx_count);
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cluster, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    bx += cg::this_cluster().block_index().x;
    phase ^= 1;
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_cluster, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

Cluster Launch Control

API example – cluster case

```
__shared__ uint4 result;
__shared__ __mbarrier_t bar;
int phase = 0;
constexpr int tx_count = sizeof(uint4);
if (threadIdx.x == 0) {
    ptx::mbarrier_init(&bar, 1);
    ptx::fence_mbarrier_init(ptx::sem_release, ptx::scope_cluster);
}
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    cg::cluster_group::sync();
    if (cg::cluster_group::thread_rank() == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        cg::invoke_one(cg::coalesced_threads(), [&](){ptx::clusterlaunchcontrol_try_cancel_multicast(&result, &bar);});
    }
    if (threadIdx.x == 0)
        ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cluster, ptx::space_shared, &bar, tx_count);
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cluster, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    bx += cg::this_cluster().block_index().x;
    phase ^= 1;
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_cluster, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

Protect from overwrite by next iteration
Also ensure all CTAs are running at 1st iteration

Cluster Launch Control

API example – cluster case

```
__shared__ uint4 result;
__shared__ __mbarrier_t bar;
int phase = 0;
constexpr int tx_count = sizeof(uint4);
if (threadIdx.x == 0) {
    ptx::mbarrier_init(&bar, 1);
    ptx::fence_mbarrier_init(ptx::sem_release, ptx::scope_cluster);
}
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    cg::cluster_group::sync();
    if (cg::cluster_group::thread_rank() == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        cg::invoke_one(cg::coalesced_threads(), [&](){ptx::clusterlaunchcontrol_try_cancel_multicast(&result, &bar);});
    }
    if (threadIdx.x == 0)
        ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cluster, ptx::space_shared, &bar, tx_count);
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cluster, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    bx += cg::this_cluster().block_index().x;
    phase ^= 1;
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_cluster, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

request by a single **cluster** thread

complete request by **each** CTA

Cluster Launch Control

API example – cluster case

```
__shared__ uint4 result;
__shared__ __mbarrier_t bar;
int phase = 0;
constexpr int tx_count = sizeof(uint4);
if (threadIdx.x == 0) {
    ptx::mbarrier_init(&bar, 1);
    ptx::fence_mbarrier_init(ptx::sem_release, ptx::scope_cluster);
}
// <<< PROLOGUE >>>
int bx = blockIdx.x;
while (true) {
    cg::cluster_group::sync();
    if (cg::cluster_group::thread_rank() == 0) {
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire, ptx::space_cluster, ptx::scope_cluster);
        cg::invoke_one(cg::coalesced_threads(), [&](){ptx::clusterlaunchcontrol_try_cancel_multicast(&result, &bar);});
    }
    if (threadIdx.x == 0)
        ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cluster, ptx::space_shared, &bar, tx_count);
    // <<< THREAD BLOCK bx COMPUTATION >>>
    while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_cluster, &bar, phase))
    {}
    bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
    if (!success)
        break;
    bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x(result);
    bx += cg::this_cluster().block_index().x;
    phase ^= 1;
    ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release, ptx::space_cluster, ptx::scope_cluster);
}
// <<< EPILOGUE >>>
```

get this CTA's index from multicasted root

Cluster Launch Control

Load balancing example

Timings (NVIDIA B200, array size: 4GB):

Persistent kernel:	0.030 sec – number of blocks equals number of SMs
Cluster launch control:	0.031 sec

Another kernel running in parallel (occupying one SM):

Persistent kernel:	0.059 sec – doubles due to second wave
Cluster launch control:	0.031 sec

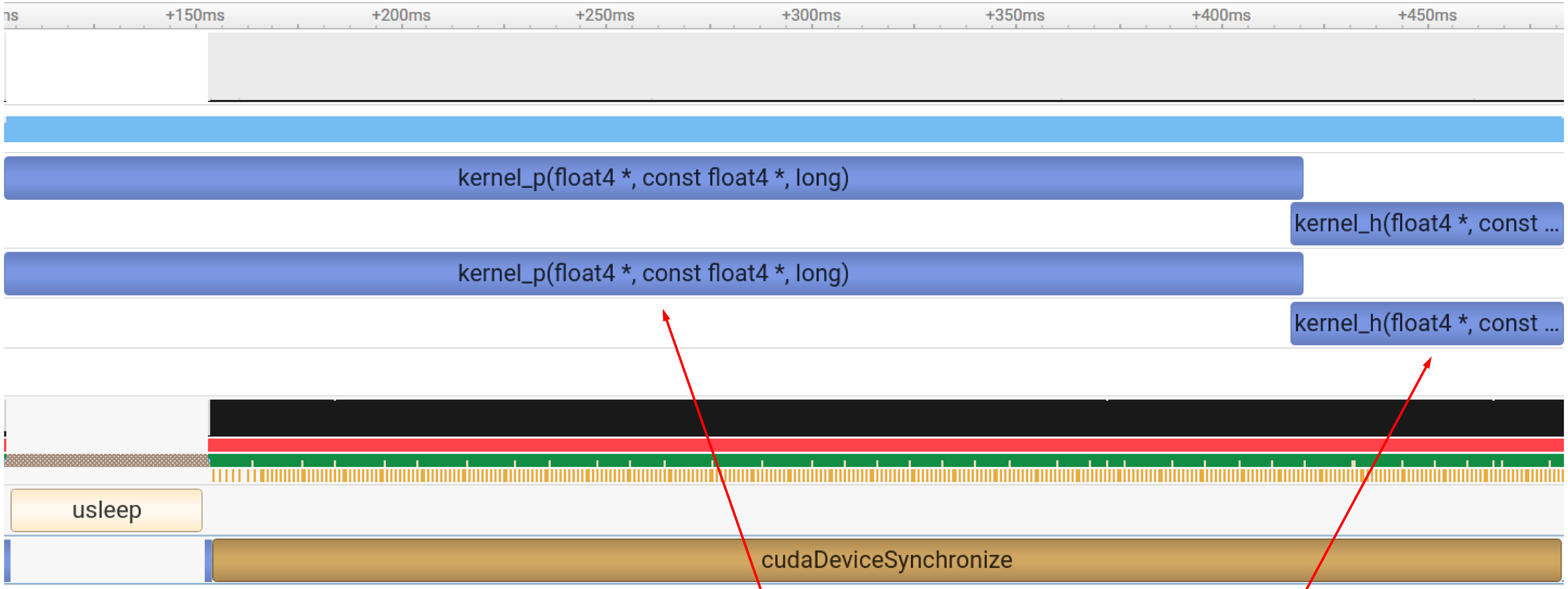
Thread blocks running 25% longer one SM:

Persistent kernel:	0.037 sec
Cluster launch control:	0.031 sec

Cluster Launch Control

Priority example

Timings (NVIDIA B200, array size: 4GB):



+150ms
+200ms
+250ms
+300ms
+350ms
+400ms
+450ms
kernel_p(float4 *, const float4 *, long)
kernel_h(float4 *, const ...
kernel_p(float4 *, const float4 *, long)
kernel_h(float4 *, const ...
usleep
cudaDeviceSynchronize

Launch persistent kernel `kernel_p`

Launch high-priority kernel `kernel_h`

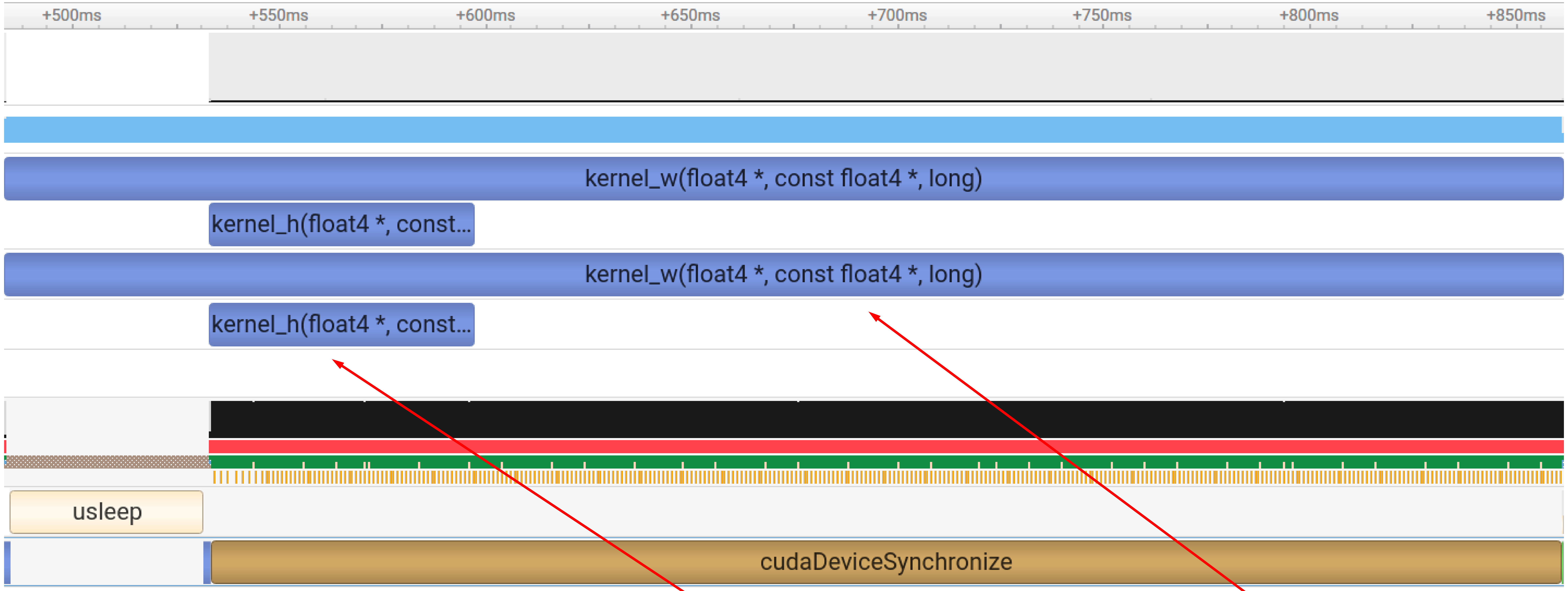
Persistent kernel occupies entire GPU

High-priority kernel cannot start earlier

Cluster Launch Control

Priority example

Timings (NVIDIA B200, array size: 4GB):



+500ms
+550ms
+600ms
+650ms
+700ms
+750ms
+800ms
+850ms
kernel_w(float4 *, const float4 *, long)
kernel_h(float4 *, const...
kernel_w(float4 *, const float4 *, long)
kernel_h(float4 *, const...
usleep
cudaDeviceSynchronize

Launch launch-control kernel `kernel_w`

Launch high-priority kernel `kernel_h`

High-priority kernel can start right away

Launch-control kernel allows to “yield”

CUDA Developer Sessions

General CUDA

- S72571 - What's CUDA All About Anyways?
- S72897 - How To Write A CUDA Program: The Parallel Programming Edition

CUDA Python

- S72450 - Accelerated Python: Tour of the Community and Ecosystem
- S72448 - The CUDA Python Developer's Toolbox
- S72449 - 1001 Ways to Write CUDA Kernels in Python
- S74639 - Enable Tensor Core Programming in Python With CUTLASS 4.0

CUDA C++

- S72574 - Building CUDA Software at the Speed-of-Light
- S72572 - The CUDA C++ Developer's Toolbox
- S72575 - How You Should Write a CUDA C++ Kernel

Developer Tools

- S72527 - It's Easier than You Think – Debugging and Optimizing CUDA with Intelligent Developer Tools

Connect with the Experts

- CWE72433 - CUDA Developer Best Practices
- CWE73310 - Using NVIDIA CUDA Compiler Tool Chain for Productive GPGPU Programming
- CWE72393 - What's in Your Developer Toolbox? CUDA and Graphics Profiling, Optimization, and Debugging Tools

Multi-GPU Programming

- S72576 - Getting Started with Multi-GPU Scaling: Distributed Libraries
- S72579 - Going Deeper with Multi-GPU Scaling: Task-based Runtimes
- S72578 - Advanced Multi-GPU Scaling: Communication Libraries

Performance Optimization

- S72683 - CUDA Techniques to Maximize Memory Bandwidth and Hide Latency
- S72685 - CUDA Techniques to Maximize Compute and Instruction Throughput
- S72686 - CUDA Techniques to Maximize Concurrency and System Utilization
- S72687 - Get the Most Performance from Grace Hopper

nvidia.com/gtc/sessions/cuda-developer

NVIDIA®