CUDA GRAPHS
ASSEMBLING A FLOW

5 June 2024 | Andreas Herten | Forschungszentrum Jülich

Overview, Outline

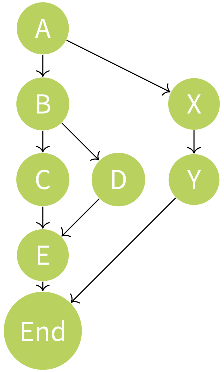
At a Glance

- CUDA Graph: Expose dependencies between kernels
- Capture once, launch repeatedly

Contents

Overview

- Graph: Series of operation (mostly kernel launches)
- Define graph ones, launch repeatedly
- Less CPU overhead: Most setup done in advance
- Enable CUDA optimization
- Phases of work submissions
  - Definition: Description of operations (graph nodes) and dependencies (graph edges)
  - Instantiation: Snapshot of graph template, validation, setup/init →*executable graph*
  - Execution: Launch graph (repeatedly)
- Every stream can be converted to graph



Details

- Available Operations

Kernel Launch	CUDA kernel running on GPU
CPU Function Call	Callback to function on CPU
Memcpy/Memset	GPU data management
Events	Waiting/recording event
External Dependency	External semaphores/events
Sub-Graph	Execute hierarchical sub-graph

Details

- Available Operations

Kernel Launch	CUDA kernel running on GPU
CPU Function Call	Callback to function on CPU
Memcpy/Memset	GPU data management
Events	Waiting/recording event
External Dependency	External semaphores/events
Sub-Graph	Execute hierarchical sub-graph

- Graph Creation
  1. Explicit graph API
  2. Stream capture

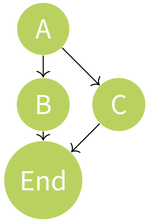
Generation: Explicit Graph API

```
// Create the graph - it starts out empty
cudaGraphCreate(&graph, 0);

// Create kernel launches as nodes of graph
cudaGraphAddKernelNode(&a, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&b, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&c, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&d, graph, NULL, 0, &nodeParams);

// Now set up dependencies on each node
cudaGraphAddDependencies(graph, &a, &b, 1);     // A->B
cudaGraphAddDependencies(graph, &a, &c, 1);     // A->C
cudaGraphAddDependencies(graph, &b, &d, 1);     // B->D
cudaGraphAddDependencies(graph, &c, &d, 1);     // C->D

cudaGraphInstantiate(...);
for (auto step = 0; step < N_step; ++step)
  cudaGraphLaunch(graph, stream);
```



Generation: Stream Capture

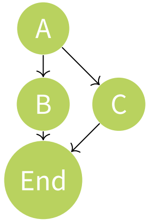
```
// stream1 is the origin stream
cudaStreamBeginCapture(stream1);
kernel_A<<< ..., stream1 >>>(...);

// Fork into stream2
cudaEventRecord(event1, stream1);
cudaStreamWaitEvent(stream2, event1);

kernel_B<<< ..., stream1 >>>(...);
kernel_C<<< ..., stream2 >>>(...);

// Join stream2 back to origin stream (stream1)
cudaEventRecord(event2, stream2);
cudaStreamWaitEvent(stream1, event2);

kernel_D<<< ..., stream1 >>>(...);
// End capture in the origin stream
cudaStreamEndCapture(stream1, &graph);
```



Conclusions

Conclusions: CUDA Graphs

- Expose dependencies
- Remove overhead for repeated kernel launches
- Capture or build

Conclusions: CUDA Graphs

- Expose dependencies
- Remove overhead for repeated kernel launches
- Capture or build

Thank you for your attention!

a.herten@fz-juelich.de