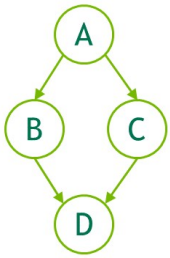
CUDA Graph

Creating a Graph Using Graph APIs

Graphs can be created via two mechanisms: explicit API and stream capture. The following is an example of creating and executing the below graph.

Creating a Graph Using Graph APIs Example




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

// For the purpose of this example, we'll create
// the nodes separately from the dependencies to
// demonstrate that it can be done in two stages.
// Note that dependencies can also be specified
// at node creation.
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
```

Creating a Graph Using Stream Capture

Stream capture provides a mechanism to create a graph from existing stream-based APIs. A section of code which launches work into streams, including existing code, can be bracketed with calls to `cudaStreamBeginCapture()` and `cudaStreamEndCapture()`. See below.

```
cudaGraph_t graph;

cudaStreamBeginCapture(stream);

kernel_A<<< ..., stream >>>(...);
kernel_B<<< ..., stream >>>(...);
libraryCall(stream);
kernel_C<<< ..., stream >>>(...);

cudaStreamEndCapture(stream, &graph);
```

A call to `cudaStreamBeginCapture()` places a stream in capture mode. When a stream is being captured, work launched into the stream is not enqueued for execution. It is instead appended to an internal graph that is progressively being built up. This graph is then returned by calling `cudaStreamEndCapture()`, which also ends capture mode for the stream. A graph which is actively being constructed by stream capture is referred to as a *capture graph.*

Stream capture can be used on any CUDA stream except cudaStreamLegacy (the “NULL stream”). Note that it *can* be used on cudaStreamPerThread. If a program is using the legacy stream, it may be possible to redefine stream 0 to be the per-thread stream with no functional change.

Whether a stream is being captured can be queried with cudaStreamIsCapturing().

The following questions (Q1-Q3) require a book.h file. Please download here.

Question 1:

The purpose of this lab is to get you familiar with using the CUDA streaming API by implementing vector addition of the average of *three values a and b*, then store in *c*.

Edit the code in the P10Q1 to perform the following:
• Use 1 CUDA stream in your program
`cudaStreamCreate(&stream)`

- Allocate device memory
- Allocate host locked memory, used to stream

```
cudaHostAlloc((void**)&host_a,FULL_DATA_SIZE * sizeof(int),
          cudaHostAllocDefault)
```

- Initialize thread block and kernel grid dimensions
- Invoke CUDA kernel
- Copy results from device to host asynchronously

```
cudaMemcpyAsync(host_c + i, dev_c,
                N * sizeof(int),
                cudaMemcpyDeviceToHost,
                stream)
```

- Free memory for host, device and stream.

```
cudaFreeHost(host_a)
cudaStreamDestroy(stream)
```

Instructions about where to place each part of the code is demarcated by the //@@ comment lines.

Question 2:

Edit the solution in Q1 so that it uses depth-first, i.e. add the copy of a, copy of b, kernel execution, and copy of c to stream 0 before starting to schedule on stream 1

Edit the code in the P10Q2 to perform the following:

- Use 2 CUDA stream in your program
- Allocate device memory
- Allocate host locked memory, used to stream

```
cudaHostAlloc((void**)&host_a,FULL_DATA_SIZE * sizeof(int),
          cudaHostAllocDefault)
```

- Interleave the host memory copy to device asynchronously
- Initialize thread block and kernel grid dimensions
- Invoke CUDA kernel
- Copy results from device to host asynchronously

```
cudaMemcpyAsync(host_c + i, dev_c,
                N * sizeof(int),
                cudaMemcpyDeviceToHost,
                stream)
```

Instructions about where to place each part of the code is demarcated by the //@@ comment lines.

Question 3:

Edit the solution in Question 2 so that it uses breadth-first, i.e. bounce back and forth between the streams assigning work. Add the copy of a to stream 0, and then add the copy of a to stream 1. Then, add the copy of b to stream 0, and then add the copy of b to stream 1. Enqueue the kernel invocation in stream 0, and then enqueue one in stream 1. Finally, enqueue the copy of c back to the host in stream 0 followed by the copy of c in stream 1.

Edit the code in the P10Q3 to perform the following:

- Use 2 CUDA stream in your program
- Allocate device memory
- Allocate host locked memory, used to stream

```
cudaHostAlloc((void**)&host_a,FULL_DATA_SIZE * sizeof(int),
          cudaHostAllocDefault)
```

- Interleave the host memory copy to device asynchronously
- Initialize thread block and kernel grid dimensions
- Invoke CUDA kernel
- Copy results from device to host asynchronously

```
cudaMemcpyAsync(host_c + i, dev_c,
                N * sizeof(int),
                cudaMemcpyDeviceToHost,
                stream)
```

Instructions about where to place each part of the code is demarcated by the //@@ comment lines.