Technical University of Munich

Lecture 8: Task-based parallelism and CUDA graphs

Informatik elective: GPU Computing

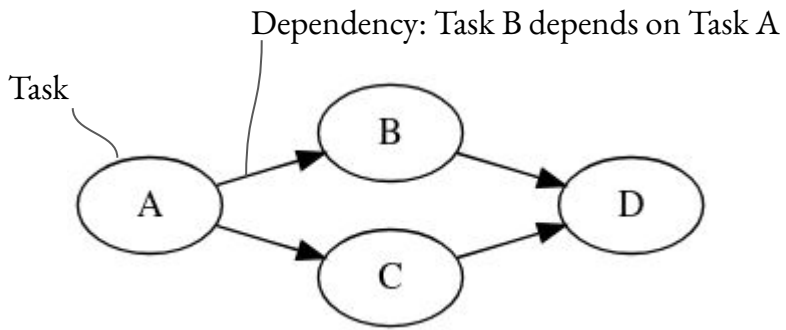
Pratik Nayak

In this session

- Task-based parallelism.
- Using CUDA graphs
- Using graph capture to generate and submit task graphs

Task-based parallelism

- A generic approach for parallel computing.
  - Abstracts away the operations to be performed as “tasks”, which are then scheduled on available resources.
  - Dependencies, such as data dependencies, between these operations or tasks are can be represented with a task graph.
  - A task graph is a directed-acyclic graph (DAG).



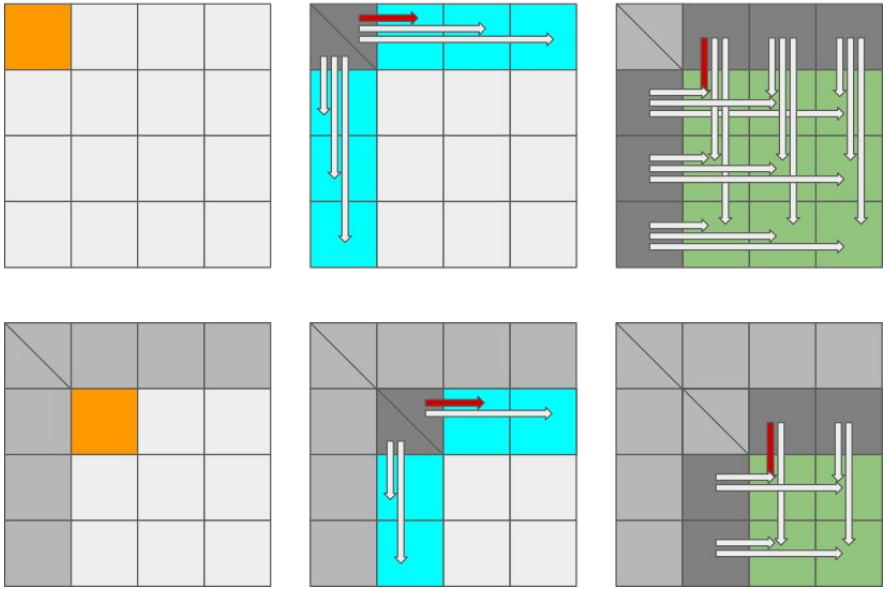
A task graph

Task-based parallelism

- *Implicit task graph*: User specifies task implementation, and the input and output data.
  - The runtime system constructs the task graph implicitly, and executes the tasks in a sequentially consistent order (as specified by the user).
- *Explicit task graph*: User specifies the task implementation, and the task dependencies.
  - The overhead of constructing the graph is minimized and the task graph can be directly submitted to the machine.

A more complex example

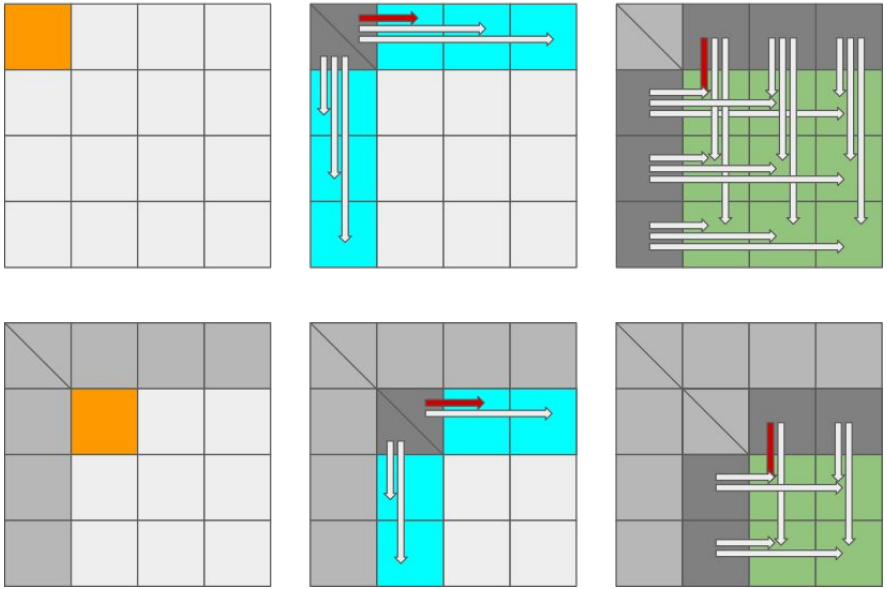
- Consider a matrix algorithm that proceeds as follows:
  - Compute a diagonal block (in orange)
  - Update corresponding block row and block columns (in blue)
  - Update the trailing matrix (in green)
  - Repeat recursively until all diagonal blocks have been processed.



[HPCN2, 2021]

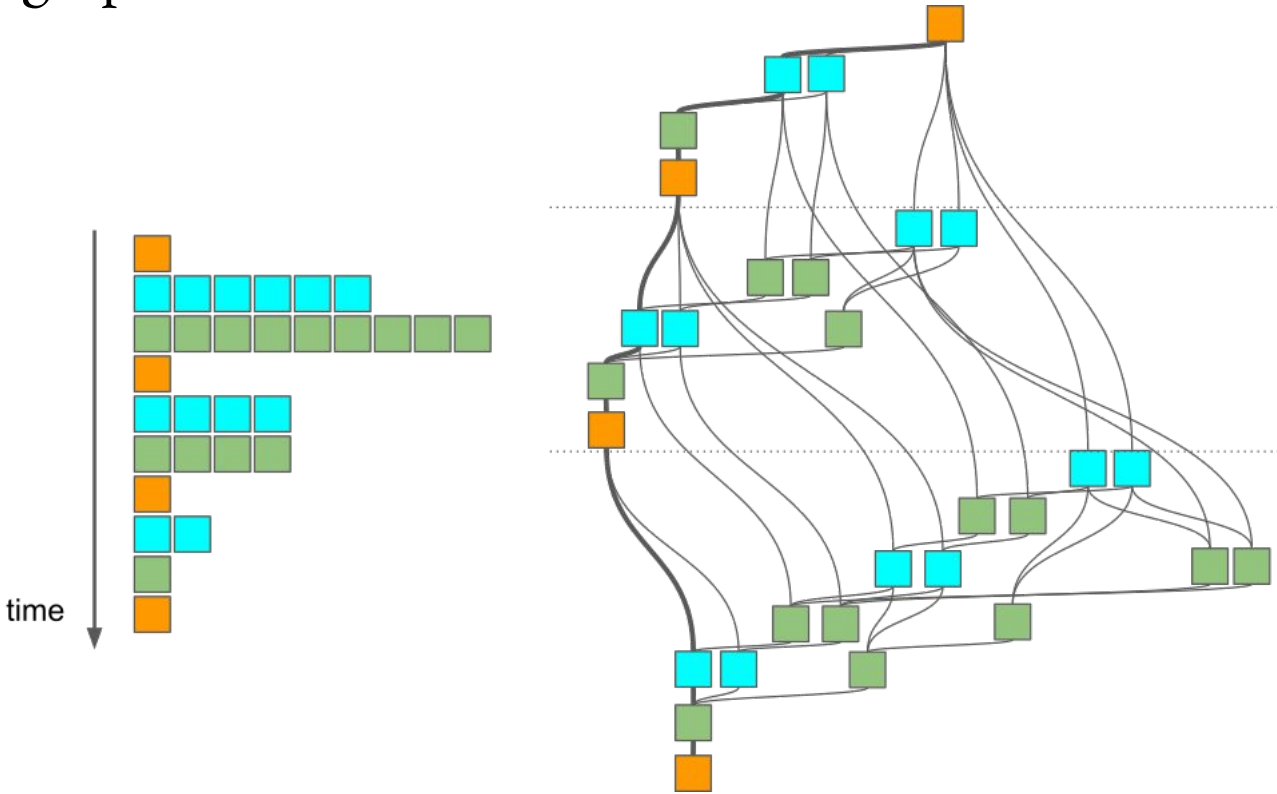
What are the dependencies ?

- Block row and block column updates depend on diagonal block.
  - Orange before Blue
- Trailing matrix depends on the block row and block column.
  - Blue before Green
- The subsequent diagonal block depends on the trailing matrix update.
  - Green before Orange



[HPCN2, 2021]

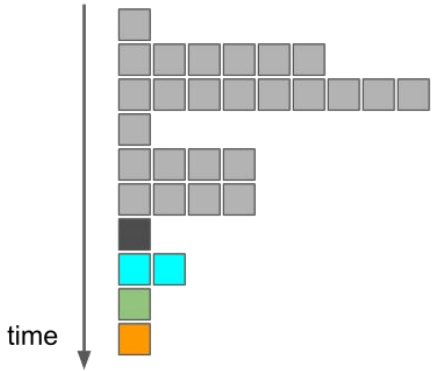
The task graph

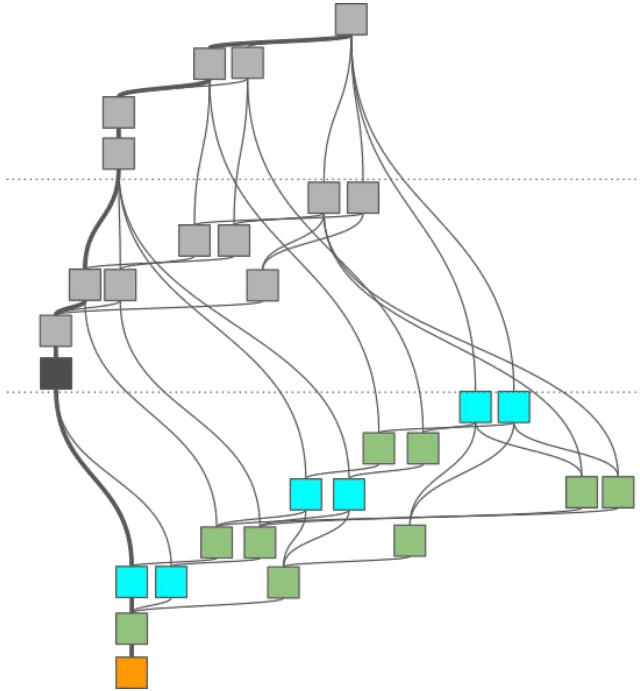


[HPCN2, 2021]

The task graph

- Better load balancing with task-based parallelism.
- Schedule tasks to minimize idle time of resources.
- Possibly enable another task graph with independent tasks to be executed concurrently.

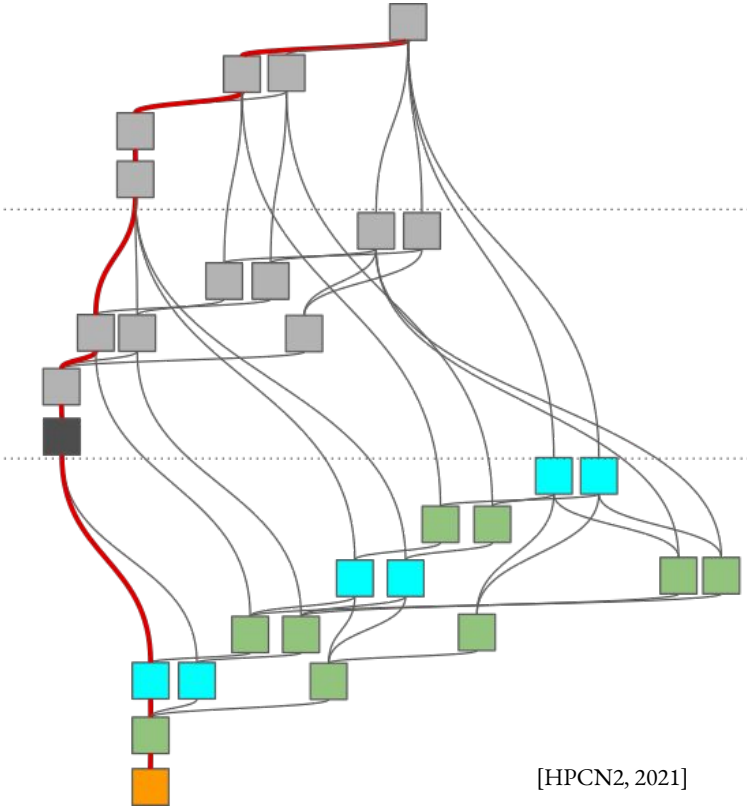




[HPCN2, 2021]

The task graph: critical path

- Critical path: Longest path through the task graph in terms of the execution time
- Provides us with a lower bound for the execution time of the algorithm.
- Can be used to balance load, mapping tasks to resources in an efficient manner.

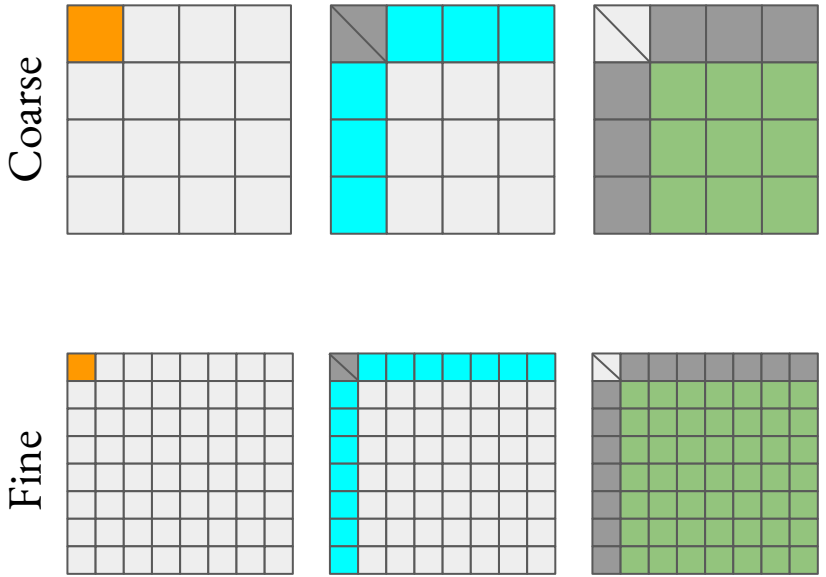


[HPCN2, 2021]

Task granularity:

- User has control over the granularity of tasks:
  - "Size" of task (Gr) , and the number of tasks (degree of concurrency, #C)

$$\#C \propto \frac{1}{Gr}$$



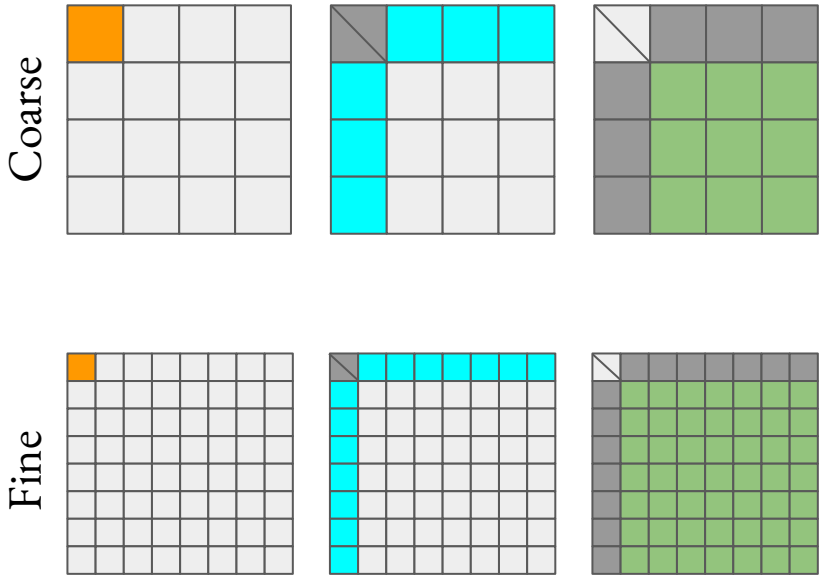
[HPCN2, 2021]

Task granularity:

- User has control over the granularity of tasks:
  - "Size" of task (Gr) , and the number of tasks (degree of concurrency, #C)

$$\#C \propto \frac{1}{Gr}$$

- Trade-offs:
  - Amount of parallelism
  - Load balance
  - Task graph overhead.



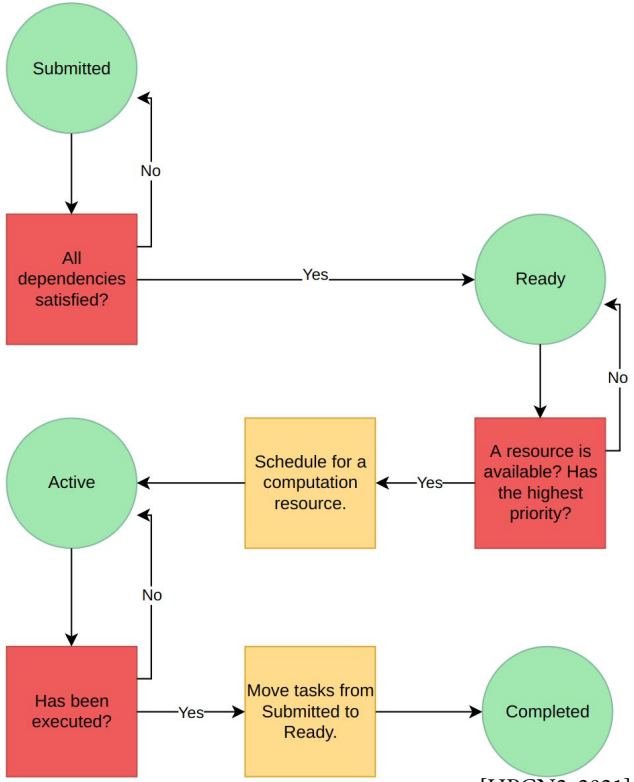
[HPCN2, 2021]

Scheduling

- Given a task graph, how do we map the tasks to the compute resources available ?
  - Given an explicit task graph, we try to map the critical path their dependencies with higher priority.
  - Given an implicit task graph, we have two options:
    - Build the entire task graph, analyze the critical path and then schedule these tasks as eary as possible.
      - Requires to know the entire task graph, and also the execution time of each task.
      - Generally impractical.
    - Use heuristics to find a optimal schedule.

Scheduling

- Schedulers maintain 4 task pools:
  - Submitted tasks: Tasks submitted, but not ready for scheduling.
  - Ready tasks: Tasks ready for scheduling
  - Active tasks: Tasks being executed
  - Completed tasks: Tasks that have finished execution.
- Users can provide additional info such as priorities, to give hints to the scheduler to prioritize which tasks to schedule first.



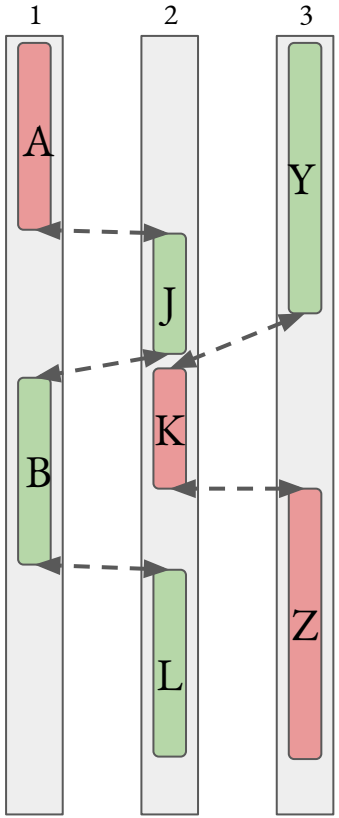
[HPCN2, 2021]

CUDA scheduling

- In CUDA, each kernel is mapped to a stream (execution pipeline)
  - By default, to the default blocking stream
  - User can create and map kernel to a specific stream
- *Coarse grained control*: Mapping the kernels in different streams to compute resources is handled by the CUDA runtime (and their proprietary scheduler).
- *Fine grained control*: CUDA runtime also has control over (In one kernel):
  - Mapping thread-blocks to SMs
  - Mapping warps to functional units.

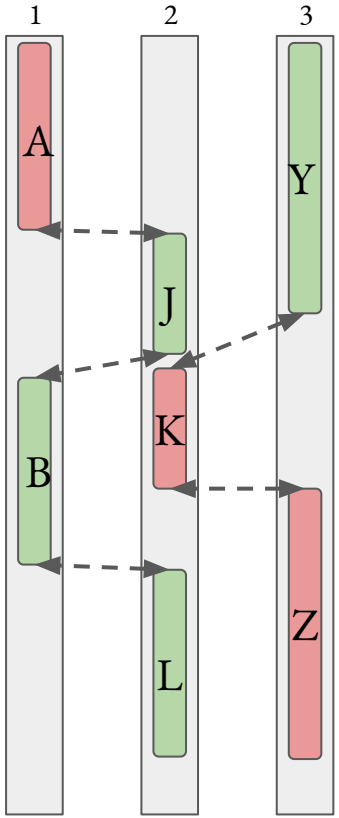
CUDA scheduling

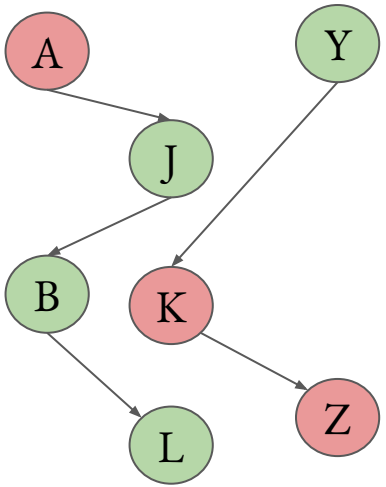
- All CUDA work can be viewed as a task graph.
- Within one stream, kernels are executed as submitted to the stream.
- Kernels in different streams can be executed concurrently; dependencies need to enforced explicitly.



CUDA scheduling

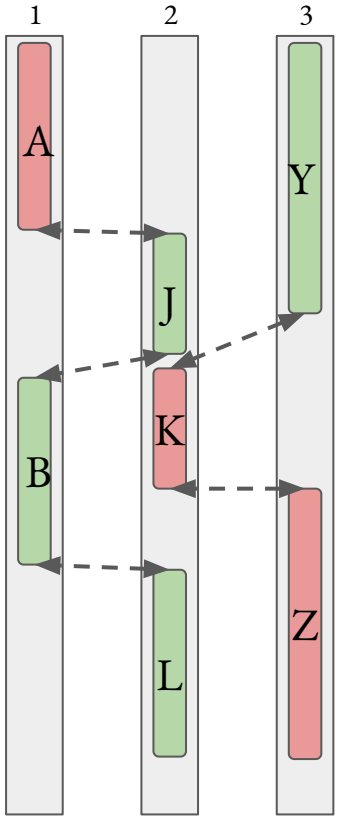
- All CUDA work can be viewed as a task graph.
- Within one stream, kernels are executed as submitted to the stream.
- Kernels in different streams can be executed concurrently; dependencies need to enforced explicitly.

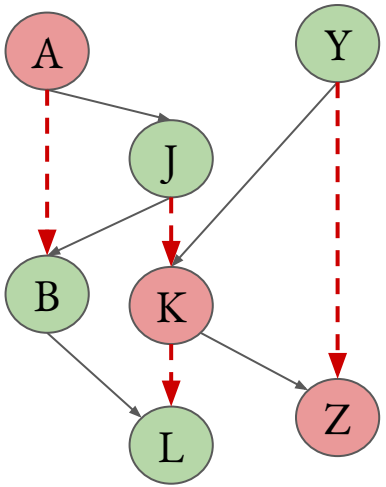




CUDA scheduling

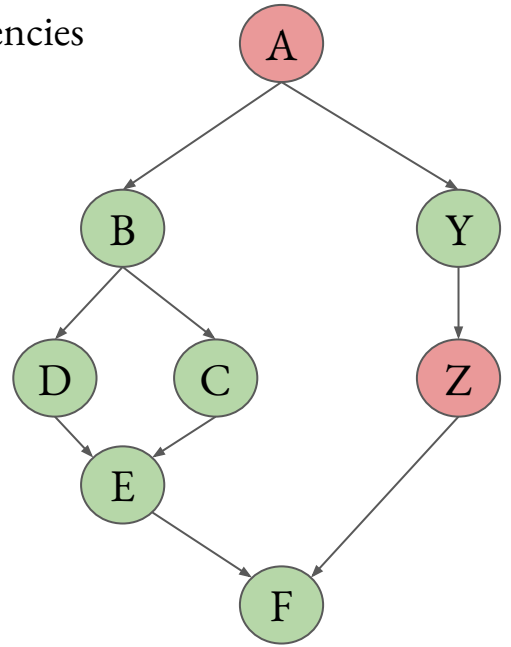
- All CUDA work can be viewed as a task graph.
- Within one stream, kernels are executed as submitted to the stream.
- Kernels in different streams can be executed concurrently; dependencies need to enforced explicitly.
- Note: Implicit dependencies within streams are also dependencies in the task graph





CUDA graphs

- A CUDA graph consists of a sequence of graph nodes connected by dependencies
  - Any asynchronous CUDA operation can be a graph node.
  - Not necessarily restricted to kernel launches.
- Graph node can consist of:
  - Kernel launch: Kernel to be executed on the GPU
  - CPU call: Callback to the CPU
  - Memory operations: copy, alloc, free, memset etc.
  - Another CUDA graph: CUDA graphs are hierarchical.
  - Event operations (wait, record)
  - Conditional node

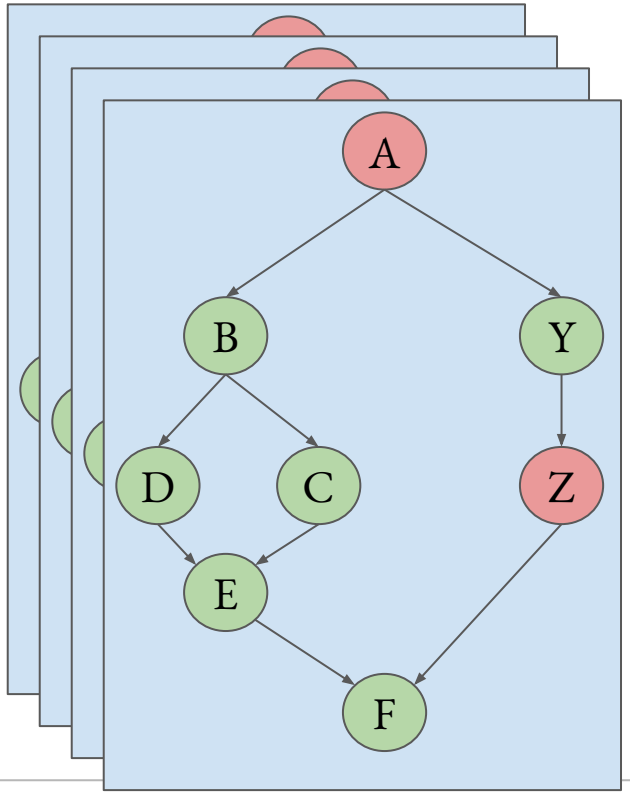


CUDA graph execution model

- Generate graph once, and the graph can then be repeatedly launched.

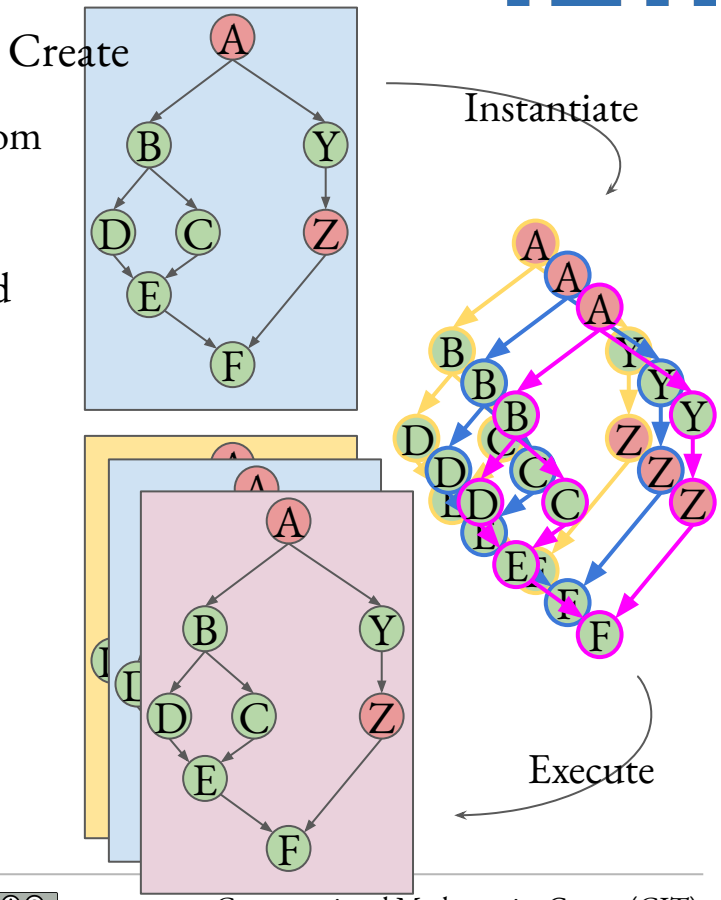
```
for(int i=0; i<1000; i++) {
    launch_graph( G );
}
```

- Three stages:
    - Creation,
    - Instantiation,
    - Execution.



Execution model

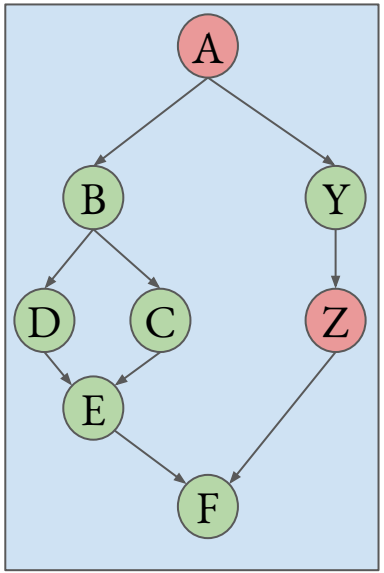
- Creation: Create the “template” graph in host code, or build from libraries.
- Instantiation: Initialize and setup GPU structures with required parameters, and data etc.
- Execution: Execute the graph on the GPU



Creating CUDA graphs

```
// Define graph of work + dependencies
cudaGraphCreate(&graph);
...
cudaGraphAddNode(kernel_a, graph, {}, ...);
cudaGraphAddNode(kernel_b, graph,{ kernel_a }, ...);
cudaGraphAddNode(kernel_c, graph, { kernel_b }, ...);
...
cudaGraphAddNode(kernel_f, graph, { kernel_e, kernel_z }, ...);
// Instantiate graph and apply optimizations
cudaGraphInstantiate(&instance, graph);

// Launch executable graph 100 times
for(int i=0; i<100; i++)
      cudaGraphLaunch(instance, stream);
```



CUDA graph API

```
cudaGraphCreate(graph, flags); // Creates an empty graph (not thread-safe!)
cudaGraphAddNode(node, graph, deps, ndeps, params); // Adds an operation as a graph
                                                    //  node to the CUDA graph.
cudaGraphInstantiate(exec_graph, gen_graph, flags);  // Instantiate the graph as an executable
                                    // graph. The runtime validates the graph for constraints
cudaGraphLaunch(graph, stream); // Launches the graph on a stream
cudaGraphDestroy(graph); // Destroys the created graph
```

- You create a graph on the host, specify the dependencies, the launch, memcpy etc parameters, and then instantiate the graph.
- Stream-ordered semantics are followed. Multiple launches of the same `exec_graph` are ordered.
- The CUDA graph API has a lot more functionalities. See the docs for more details: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html#group__CUDART__GRAPH

CUDA graph API

```
cudaGraphCreate(graph, flags); // Creates an empty graph (not thread-safe)
cudaGraphAddNode(node, graph, deps, ndeps, params); // Adds an operation as a graph
                                                    //  node to the CUDA graph.
cudaGraphInstantiate(exec_graph, ger                 e the graph as an executable
                                    /                the graph for constraints
cudaGraphLaunch(graph, stream); // Launches the graph on a stream
cudaGraphDestroy(graph); // Destroys the created graph
```

DEMO

- You create a graph on the host, specify the dependencies, the launch, memcpy etc parameters, and then instantiate the graph.
- Stream-ordered semantics are followed. Multiple launches of the same `exec_graph` are ordered.
- The CUDA graph API has a lot more functionalities. See the docs for more details: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html#group__CUDART__GRAPH

Automatic graph capture

- As you can imagine, specifying the dependencies and the operations by yourself is not simple.
- CUDA therefore provides an API to automatically capture operations in a stream to a graph.

```
cudaStreamBeginCapture(stream, mode); // Begins graph capture on a stream
cudaStreamEndCapture(stream, graph); // Ends capture on a stream and returns the captured graph.
```

- Any operations input to stream between `BeginCapture` and `EndCapture` do not get executed, but are only captured to the graph.
- The dependencies between the nodes are inferred from the stream or from the event API calls.
- Simple to use, but configurations (kernel launch config and parameters) need to be same between different generated and instantiated graphs. If they change, then the graph is not valid anymore and needs to be re-captured.

Automatic graph capture

- As you can imagine, specifying the dependencies and the operations by yourself is not simple.
- CUDA therefore provides an API to automatically capture operations in a stream to a graph.

```
cudaStreamBeginCapture(stream, mode)                    stream
cudaStreamEndCapture(stream, graph);                    d returns the captured graph.
```

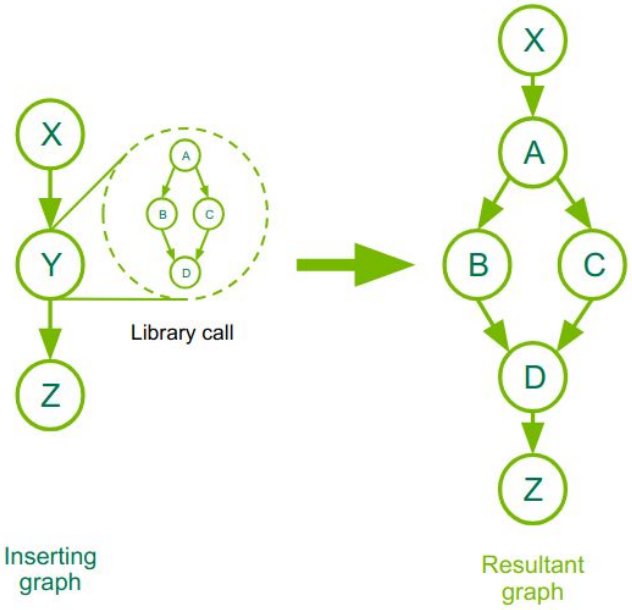
DEMO

- Any operations input to stream between `BeginCapture` and `EndCapture` do not get executed, but are only captured to the graph.
- The dependencies between the nodes are inferred from the stream or from the event API calls.
- Simple to use, but configurations (kernel launch config and parameters) need to be same between different generated and instantiated graphs. If they change, then the graph is not valid anymore and needs to be re-captured.

Capturing library calls

- You can also capture external library calls into a graph node

```
// Create root node of graph via explicit API
cudaGraphAddNode(main_graph, X, {}, ...);
// Capture the library call into a subgraph
cudaStreamBeginCapture(&stream);
libraryCall(stream); // Launches A, B, C, D
cudaStreamEndCapture(stream, &libraryGraph);
// Insert the subgraph into main_graph as node "Y"
cudaGraphAddChildGraphNode(Y, main_graph, { X } ... libraryGraph);
// Continue building main graph via explicit API
cudaGraphAddNode(main_graph, Z, { Y }, ...);
```

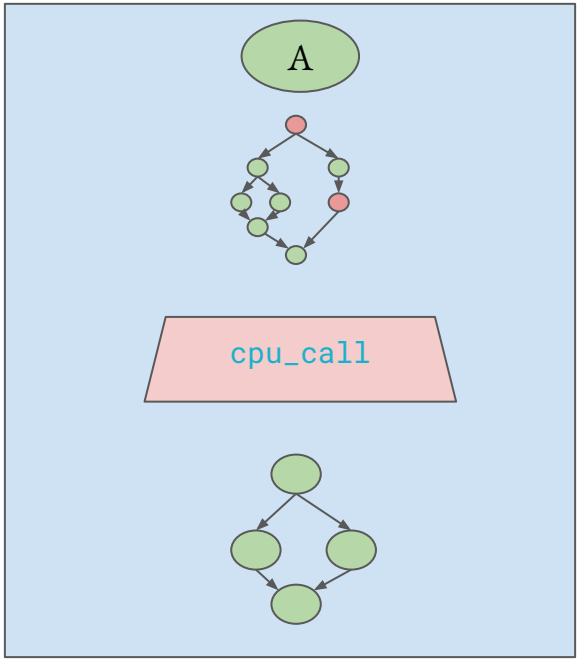


[NVIDIA@ORNL, 2021]

Ordering graph work

- It is always possible to order external work with graph work

```
launchWork(cudaGraphExec_t i1, cudaGraphExec_t i2,
CPU_Func cpu, cudaStream_t stream) {
      A <<< 256, 256, 0, stream >>>(); // Kernel launch
      cudaGraphLaunch(i1, stream); // Graph launch
      cudaStreamAddCallback(stream, cpu_call); // CPU callback
      cudaGraphLaunch(i2, stream); // Graph launch
      cudaStreamSynchronize(stream);
}
```



[NVIDIA@ORNL, 2021]

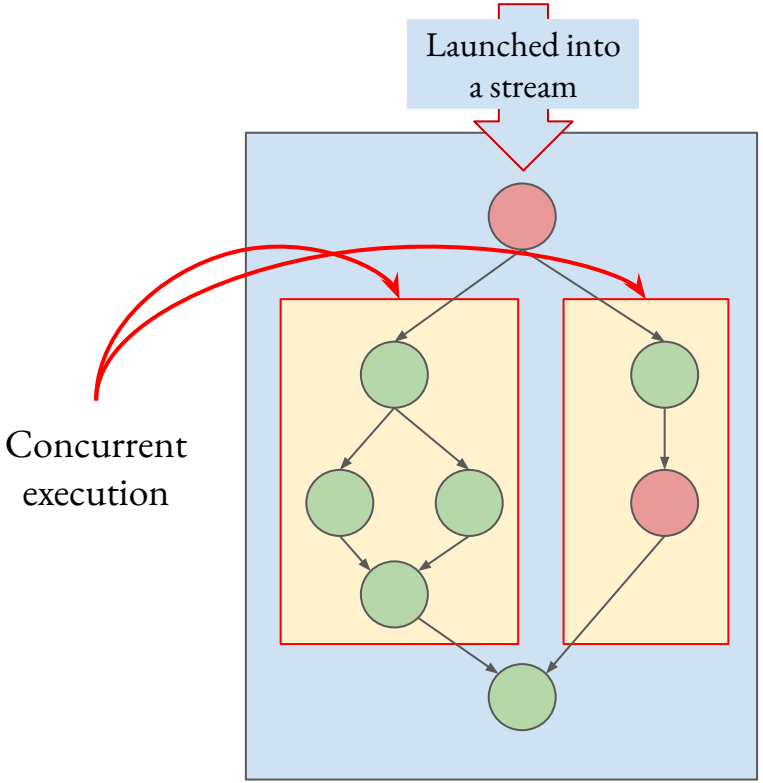
Re-instantiation of graphs

- A graph is a snapshot of the workflow.
    - The runtime stores the kernels, parameters, and dependencies for efficient and rapid replay.
- Re-instantiation is necessary if the graph topology changes as the runtime needs to perform topological optimizations again.
    - This can be reduce the benefits of using CUDA graphs.
- For cases where the topology has not changed, but only the parameters have changed, you can use the graph update API instead.
- The new input graph needs to be topologically identical to the instantiated exec_graph

```
cudaGraphExecUpdate(exec_graph, graph_new, err); // Updates exec_graph with the parameters of
                                                 graph_new
```

Graph execution semantics

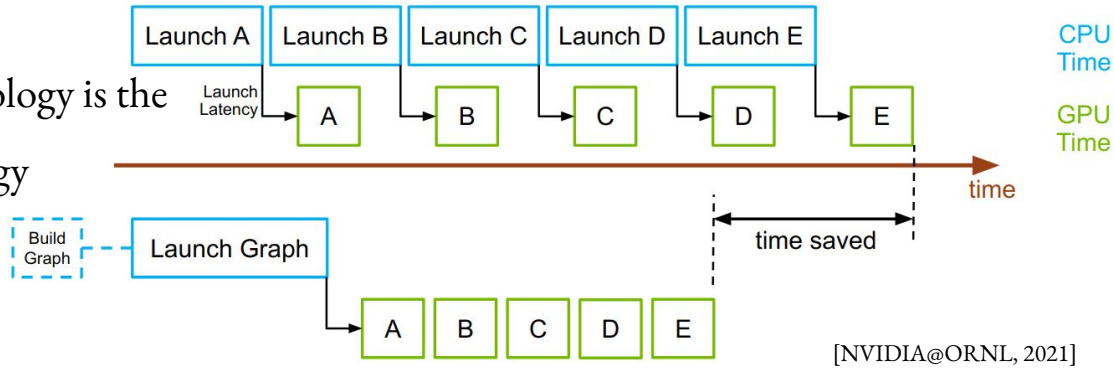
- Graphs ignore stream serialization rules
- Branches in graphs can execute concurrently even if launched into the same stream
- The launch stream is used only for ordering with other work in the stream.



[NVIDIA@ORNL, 2021]

Why CUDA graphs ?

- Rapid re-issue of work: Cost of graph-instantiation ~ cost of kernel launch.
- Reduce single kernel launch overhead, particularly for kernels with a small runtime.
- A nicer view of dependencies → more expressive programming.
- Possibility to update graphs when topology is the same and take advantage of the topology optimizations from CUDA runtime.



[NVIDIA@ORNL, 2021]

Summary

- Task-based programming can enable efficient use of resources and programming especially for complex workflows.
- CUDA provides a graph API, enabling asynchronous operations with more expressive programming.

Next lecture

- CUDA libraries: cuBLAS, cuSPARSE, thrust.