GPU Teaching Kit

Accelerated Computing

Module 23 – Dynamic Parallelism

Lecture 23 - In depth study of Dynamic Parallelism

Dynamic Parallelism

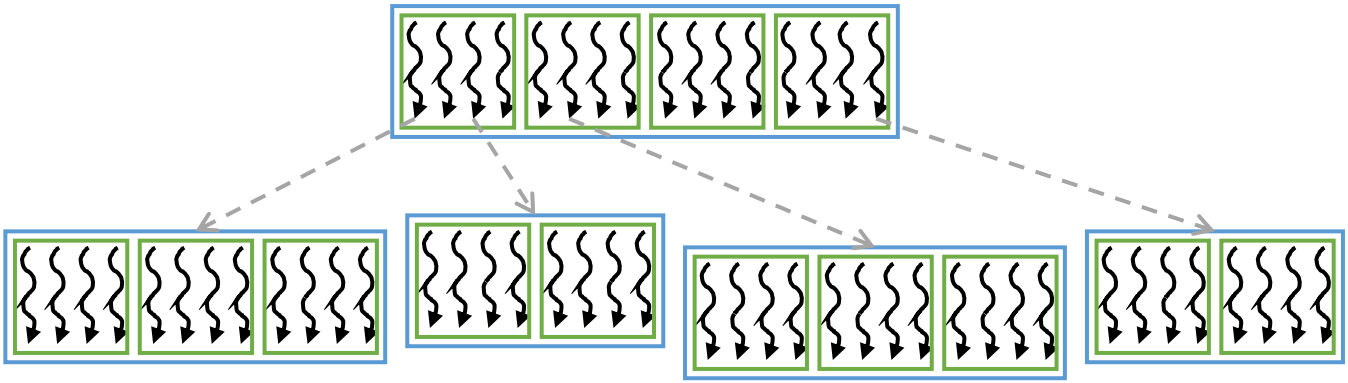
Izzat El Hajj
American University of Beirut
Wen-mei Hwu
University of Illinois at Urbana-Champaign

Objective

- To learn about CUDA Dynamic Parallelism
- Applications that benefit from Dynamic parallelism
- Dynamic parallelism in action with BFS algorithm
- Dynamic parallelism Optimization

Dynamic Parallelism

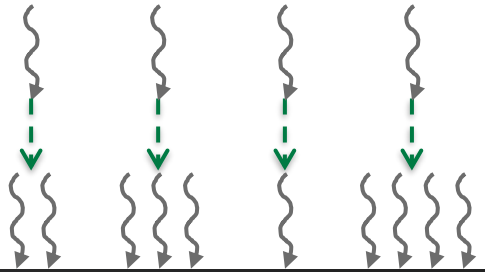
– CUDA **dynamic parallelism** refers to the ability of threads executing on the GPU to launch new grids



Nested Parallelism

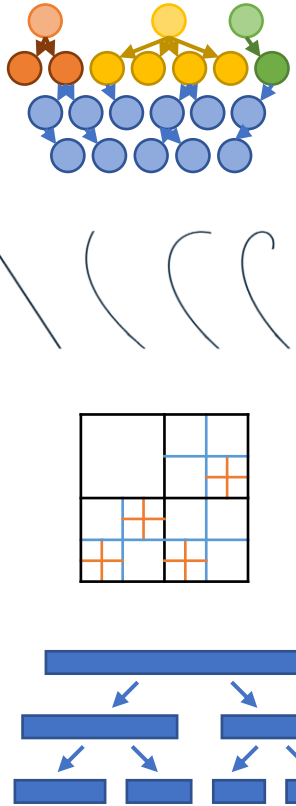
– Dynamic parallelism is useful for programming applications with **nested parallelism** where each thread discovers more work that can be parallelized

– Dynamic parallelism is particularly useful when the amount of nested work is dynamically determined at execution time, so enough threads cannot be launched up front

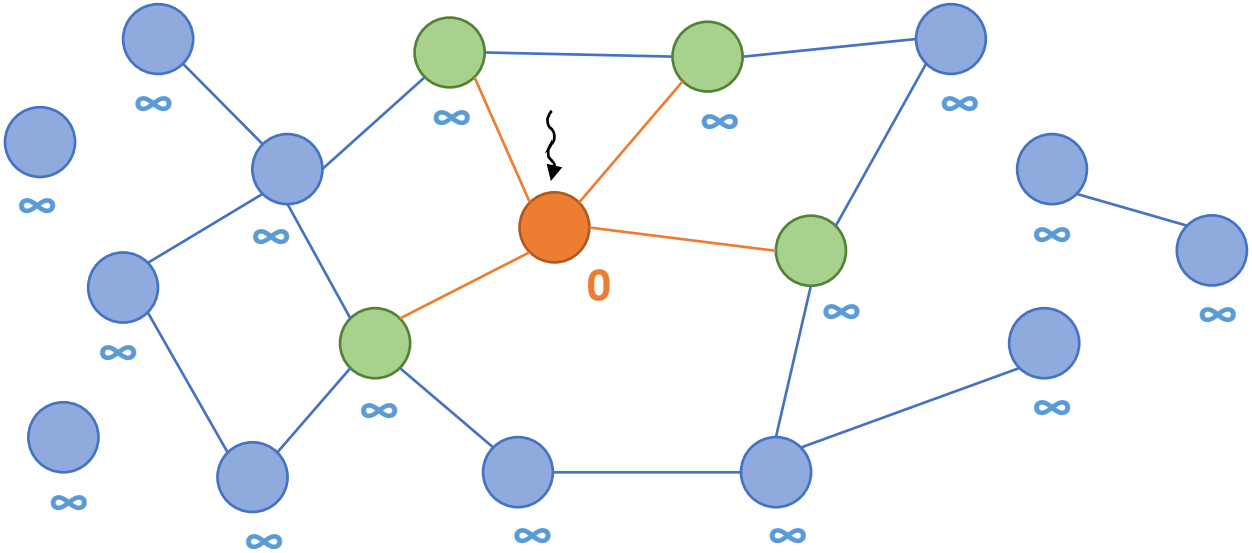


Applications of Dynamic Parallelism

- Applications whose amount of nested work may be unknown before execution time:
  - Nested parallel work is **irregular** (varies across threads)
    - e.g., graph algorithms (each vertex has a different #neighbors)
    - e.g., Bézier curves (each curve needs different #points to draw)
  - Nested parallel work is **recursive** with data-depdent depth
    - e.g., tree traversal algorithms (e.g., quadtrees and octrees)
    - e.g., divide and conquer algorithms (e.g., quicksort)



Example: BFS

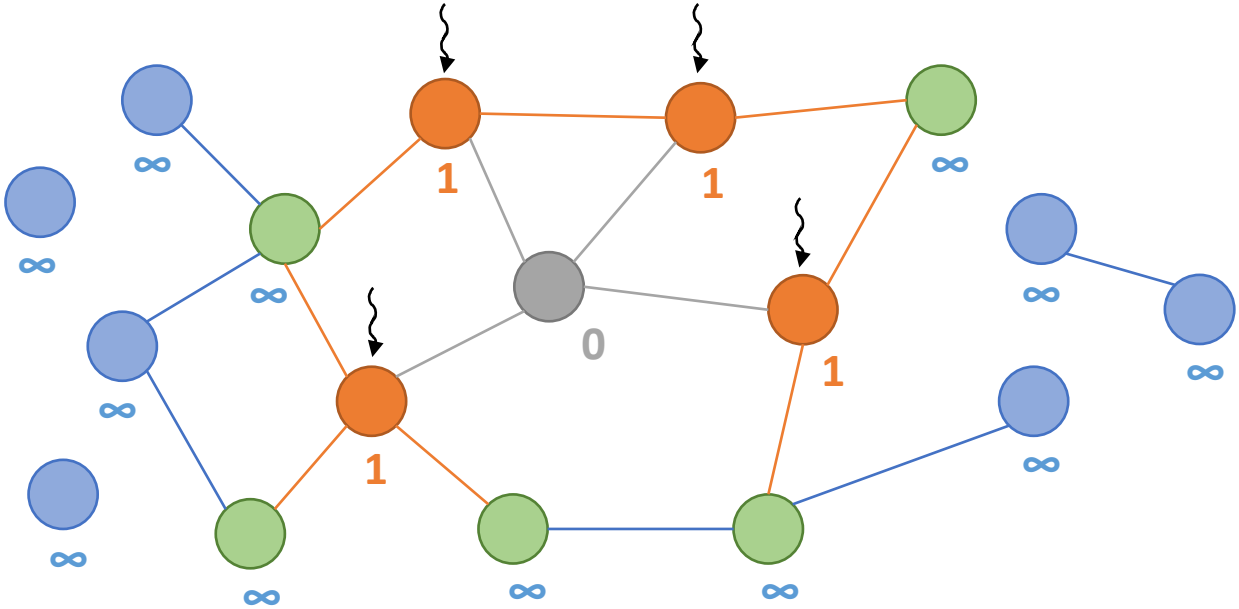


Level 0 => Level 1

Previous Frontier:

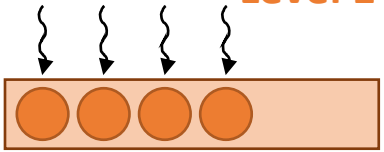
Next Frontier:

Example: BFS



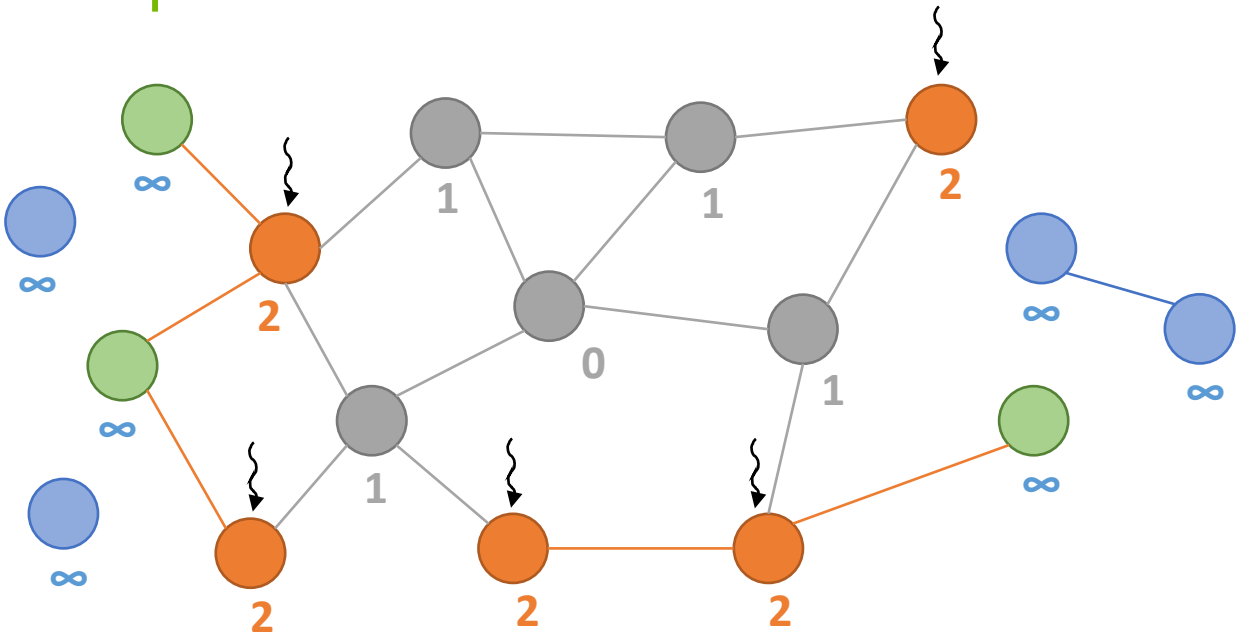
Level 1 => Level 2

Previous Frontier:

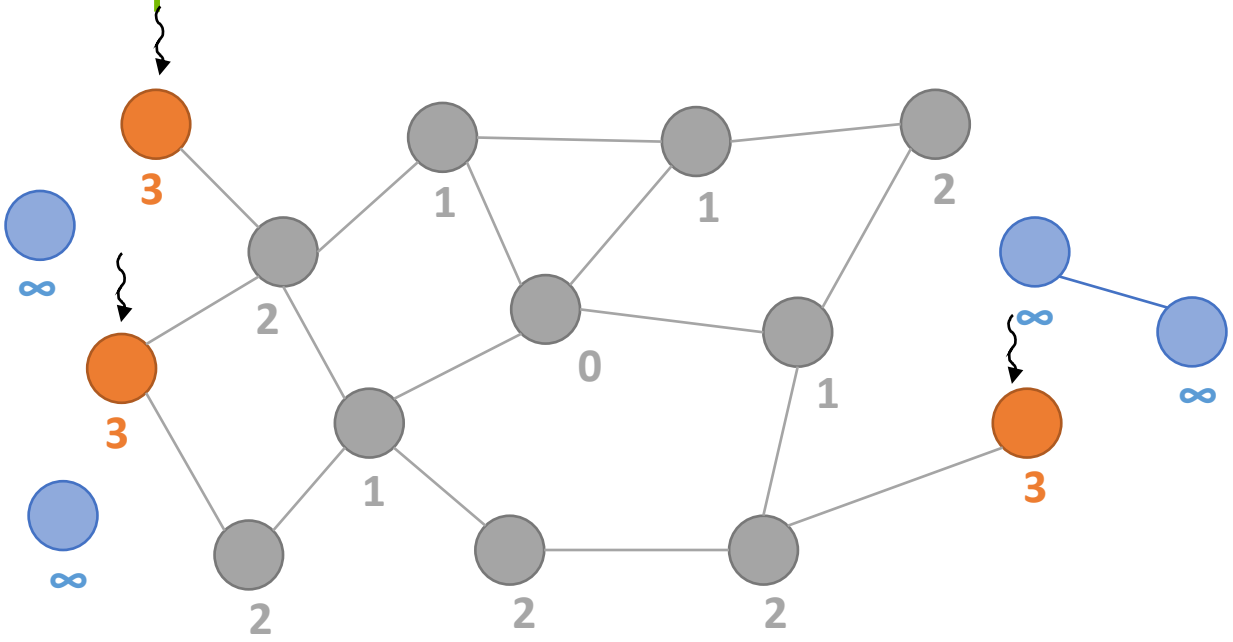


Next Frontier:

Example: BFS

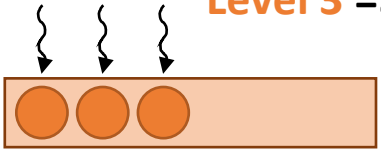

∞
∞
∞
∞
∞
∞
∞
1
1
0
1
1
2
2
2
2
2
2
Level 2 => Level 3
Previous Frontier:
Next Frontier:

Example: BFS



Level 3 => Level 4

Previous Frontier:



Next Frontier:

BFS Code

```
__global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* nodeLevel, unsigned int* prevFrontier,
                  unsigned int* currFrontier, unsigned int numPrevFrontier, unsigned int*
numCurrFrontier,
                  unsigned int level) {

    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < numPrevFrontier) {
        unsigned int node = prevFrontier[i];
        unsigned int start = csrGraph.nodePtrs[node];
        unsigned int numNeighbors = csrGraph.nodePtrs[node + 1] - start;
        for(unsigned int i = 0; i < numNeighbors; ++i) {
            unsigned int edge = start + i;
            unsigned int neighbor = csrGraph.neighbors[edge];
            if(atomicCAS(&nodeLevel[neighbor], UINT_MAX, level) == UINT_MAX) { // Not previously visited
                unsigned int currFrontierIdx = atomicAdd(numCurrFrontier, 1);
                currFrontier[currFrontierIdx] = neighbor;
            }
        }
    }

}
```

Loop over neighbors can be parallelized

Dynamic Parallelism API

- The device code for calling a kernel to launch a grid is the **same as the host code**

- Memory is needed for buffering grid launches that have not started executing
  - The limit on the number of dynamic launches is referred to as the **pending launch count**
  - By default, the runtime supports 2048 launches, and exceeding this limit will cause an error
  - The limit can be increased as follows:

```
cudaDeviceSetLimit(cudaLimitDevRuntimePendingLaunchCount, < new limit >);
```

BFS Code with Dynamic Parallelism

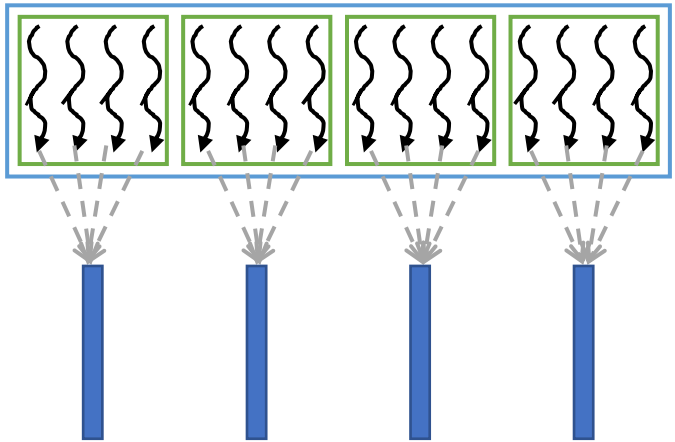
```
__global__ void bfs_neighbors_kernel(CSRGraph csrGraph, unsigned int* nodeLevel, unsigned int*
currFrontier,
        unsigned int* numCurrFrontier, unsigned int level, unsigned int start, unsigned int numNeighbors)
{

    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < numNeighbors) {
        unsigned int edge = start + i;
        unsigned int neighbor = csrGraph.neighbors[edge];
        if(atomicCAS(&nodeLevel[neighbor], UINT_MAX, level) == UINT_MAX) { // Not previously visited
            unsigned int currFrontierIdx = atomicAdd(numCurrFrontier, 1);
            currFrontier[currFrontierIdx] = neighbor;
        }
    }

}

__global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* nodeLevel, unsigned int* prevFrontier,
                  unsigned int* currFrontier, unsigned int numPrevFrontier, unsigned int* numCurrFrontier,
                  unsigned int level) {

    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < numPrevFrontier) {
        unsigned int node = prevFrontier[i];
        unsigned int start = csrGraph.nodePtrs[node];
        unsigned int numNeighbors = csrGraph.nodePtrs[node + 1] - start;
        unsigned int numThreadsPerBlock = 1024;
        unsigned int numBlocks = (numNeighbors + numThreadsPerBlock - 1)/numThreadsPerBlock;
        bfs_neighbors_kernel <<< numBlocks, numThreadsPerBlock >>>
                        (csrGraph, nodeLevel, currFrontier, numCurrFrontier, level, start, numNeighbors);
    }

}
```

Loop index becomes a thread index

Loop becomes a kernel call

Streams

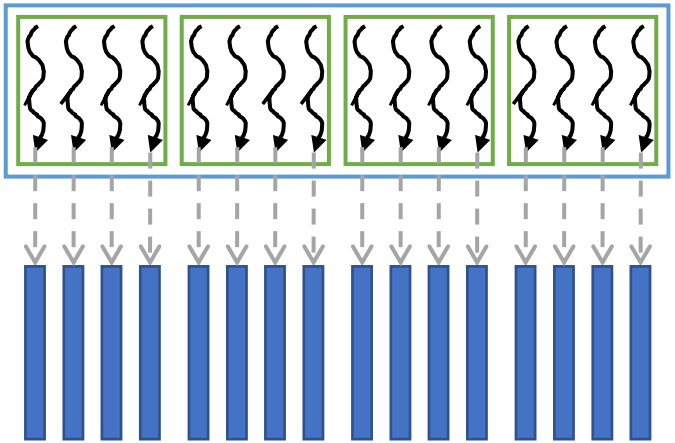
- Recall: without specifying a stream when calling a kernel, grids get launched into a **default stream**
- For device launches, threads in the same block share the same default stream
  - Launches by threads in the same block are serialized



per-block default stream

Per-Thread Streams

- Parallelism can be improved by creating a different stream per thread
  - Approach #1: Use stream API just like on host
  - Approach #2: Use compiler flag `--default-stream per-thread`



per-thread stream

Optimizations

- Pitfalls:
  - Launching very small grids may not be worth the overhead (more efficient to serialize)
  - Launching too many grids causes queueing delays
- Optimization: apply a **threshold** to the launch
  - Only launch the large grids that are worth the overhead and serialize the rest
  - Threshold value is data dependent and can be tuned
- Optimization: **aggregate** launches
  - Have one thread collect the work of multiple threads and launch a single grid on their behalf

BFS Code with Threshold

```
__global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* nodeLevel, unsigned int* prevFrontier,
                  unsigned int* currFrontier, unsigned int numPrevFrontier, unsigned int*
numCurrFrontier,
                  unsigned int level) {

    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < numPrevFrontier) {
        unsigned int node = prevFrontier[i];
        unsigned int start = csrGraph.nodePtrs[node];
        unsigned int numNeighbors = csrGraph.nodePtrs[node + 1] - start;
        if(numNeighbors > 1200) {
            unsigned int numThreadsPerBlock = 1024;
            unsigned int numBlocks = (numNeighbors + numThreadsPerBlock - 1)/numThreadsPerBlock;
            bfs_neighbors_kernel <<< numBlocks, numThreadsPerBlock >>>
                        (csrGraph, nodeLevel, currFrontier, numCurrFrontier, level, start,
numNeighbors);
        } else {
            for(unsigned int i = 0; i < numNeighbors; ++i) {
                unsigned int edge = start + i;
                unsigned int neighbor = csrGraph.neighbors[edge];
                if(atomicCAS(&nodeLevel[neighbor], UINT_MAX, level) == UINT_MAX) {
                    unsigned int currFrontierIdx = atomicAdd(numCurrFrontier, 1);
                    currFrontier[currFrontierIdx] = neighbor;
                }
            }
        }
    }

}
```

Check if the threshold is met

Offloading Host Driver (Control) Code

- In some applications, the host code that drives the computation launches **multiple consecutive grids** to synchronize across all threads between launches
  - e.g., BFS launches a new grid for each level

- Another application of dynamic parallelism is to **offload the driver code** to the device
  - Main advantage is to free up the host to do other things

BFS Driver Kernel with Dynamic Parallelism

```
__global__ void bfs_driver_kernel(CSRGraph csrGraph, unsigned int* nodeLevel, unsigned int* prevFrontier,
    unsigned int* currFrontier, unsigned int* numCurrFrontier) {

    unsigned int numPrevFrontier = 1;
    unsigned int numThreadsPerBlock = 256;
    for(unsigned int level = 1; numPrevFrontier > 0; ++level) {

        // Visit nodes in previous frontier
        *numCurrFrontier = 0;
        unsigned int numBlocks = (numPrevFrontier + numThreadsPerBlock - 1)/numThreadsPerBlock;
        bfs_kernel <<< numBlocks, numThreadsPerBlock >>>
               (csrGraph, nodeLevel, prevFrontier, currFrontier, numPrevFrontier, numCurrFrontier,
level);
        cudaDeviceSynchronize();

        // Swap buffers
        unsigned int* tmp = prevFrontier;
        prevFrontier = currFrontier;
        currFrontier = tmp;
        numPrevFrontier = *numCurrFrontier;

    }

}
```

Launch a single thread to drive the computation

Host code:

```
bfs_driver_kernel <<< 1, 1 >>> (csrGraph, nodeLevel, prevFrontier, currFrontier, numCurrFrontier);
```

Memory Visibility

Operations on **global memory** made by a parent thread before the launch are visible to the child

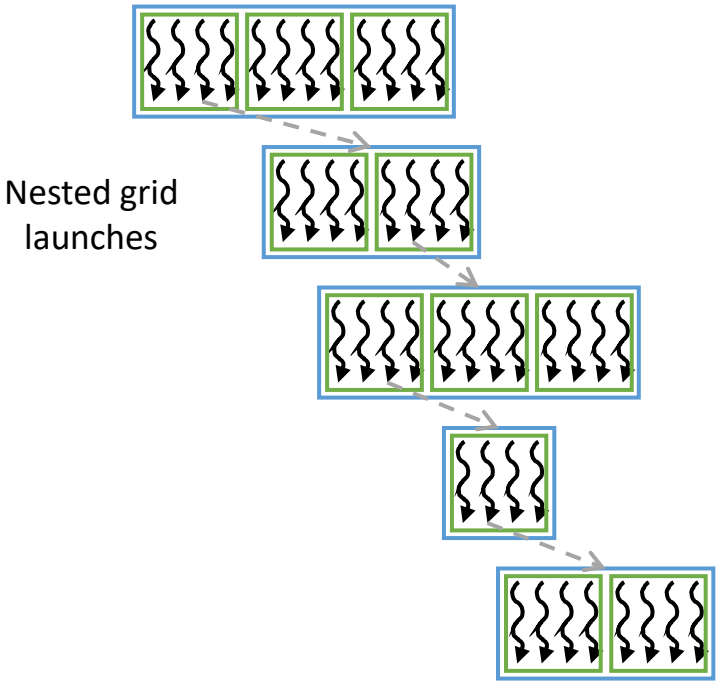
- Operations made by the child are visible to the parent after the child returns and the parent has synchronized

A thread's **local memory** and a block's **shared memory** cannot be accessed by child threads

- Child threads launched by a parent thread may run on a different SM

Nesting Depth

- The **nesting depth** refers to how deeply dynamically launched grids may launch other grids
  - Determined by the hardware (typical value is 24)



GPU Teaching Kit

Accelerated Computing