P&S Heterogeneous Systems

Dynamic Parallelism

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Spring 2023

9 June 2023
Dynamic Parallelism

- GPU programming frameworks provide an interface to express dynamic refinement algorithms in a more natural way
  - This dynamic parallelism interface allows GPU threads to launch GPU kernels when new work is dynamically discovered
  - Recall BFS
    - Each node in the frontier has a different number of neighbors
- CUDA Dynamic Parallelism
  - Important semantics when a kernel is launched from a kernel
  - Performance considerations
Dynamic Parallelism in CUDA

- Device-side kernel launches
  - Kepler GK110 architecture
  - Typical use cases
    - Dynamic load balancing
    - Data-dependent execution
    - Recursion
    - Library (with kernels) calls from kernels
  - Programmability and maintainability

Fermi: Only CPU can generate GPU work.

Kepler: GPU can generate work for itself.
Previously, kernels could only be launched from the host (painful to program!)
Dynamic Parallelism

Kernels threads can launch new kernels on the device without host communication
Easier to write programs with dynamically discovered parallelism
Parent-Child Synchronization

- **Synchronization**
  - **Parent to child:** memory consistency
  - **Child to parent:** after `cudaDeviceSynchronize()`
A Simple Example
A Simple Example (I)

Without dynamic parallelism

```
__global__ void kernel(unsigned int start, unsigned int end,
float* someData, float* moreData) {
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    doSomeWork(someData[i]);
    for(unsigned int j = start; j < end; ++j) {
        doMoreWork(moreData[j], i);
    }
}
```
A Simple Example (II)

- Without dynamic parallelism, **non-uniform workload**

```c
__global__ void kernel(unsigned int* start, unsigned int* end,
                        float* someData, float* moreData) {

    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    doSomeWork(someData[i]);

    for(unsigned int j = start[i]; j < end[i]; ++j) {
        doMoreWork(moreData[j]);
    }

}
```
A Simple Example (III)

- With dynamic parallelism, non-uniform workload

```c
__global__ void kernel_parent(unsigned int* start, unsigned int* end,
                              float* someData, float* moreData) {

    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    doSomeWork(someData[i]);

    kernel_child <<< ceil((end[i]-start[i])/256.0) , 256 >>>
                  (start[i], end[i], moreData);
}

__global__ void kernel_child(unsigned int start, unsigned int end,
                             float* moreData) {

    unsigned int j = start + blockIdx.x*blockDim.x + threadIdx.x;

    if(j < end) {
        doMoreWork(moreData[j]);
    }
}
```

Child threads

Kernel calls
A Recursive Example
Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold

A Recursive Example: Quadtree (I)

Depth = 0

Threshold = 2
A Recursive Example: Quadtree (II)

- Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold

```
Depth = 1

Threshold = 2
```
A Recursive Example: Quadtree (III)

- Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold

Depth = 2

Threshold = 2
Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold.

Depth = 3

Threshold = 2
A Recursive Example: Quadtree (V)

1 thread block is launched from host

Depth = 0

Outline of recursive kernel

Assign block to node

points > min_points && depth < max_depth

Y

Compute center of bounding box

Count points in children

Scan for offsets

Reorder points

Launch 4 children

N

Exit
Each block launches 1 child grid of 4 blocks
A Recursive Example: Quadtree (VII)

- Each block launches 1 child grid of 4 blocks

<table>
<thead>
<tr>
<th>Block 000</th>
<th>Block 001</th>
<th>Block 002</th>
<th>Block 003</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 030</td>
<td>Block 031</td>
<td>Block 032</td>
<td>Block 033</td>
</tr>
</tbody>
</table>

Buffer 0: b c e f m n g j o a d h k p q i l r s t u
Buffer 1: b c e f m n g j o a d h k p q i l r s t u

Depth = 2
A Recursive Example: Quadtree (VIII)

Each block launches 1 child grid of 4 blocks

Depth = 3

Buffer 1: bc e f m n g j o a d h k p q i l r t s u

Buffer 0: bc e f m n g j o a d h k p q i l r t s u
Summary: Quadtrees Construction

- The execution starts with host launching one thread block
  - At each recursion, if the number of atoms in the quadrant is less than or equal to the threshold, the thread block exits
- At each recursion, threads in each thread block that do not exit will collaboratively
  - Determine the number of atoms that belong in each quadrant
  - Perform a scan to determine the starting point of each quadrant
  - Reorder the atoms so that all atoms in the same quadrant are placed consecutively
  - One representative thread launches a kernel with 4 child blocks

- Oct Tree is for 3D space
  - A 3D space is divided into 8 Octants
  - Each block that does not exit launches 1 child grid of 8 blocks
Performance Limitations
Performance Limitations

- Dynamic Parallelism ensures better work balance, and offers advantages in terms of programmability.

- However, launching grids with a very small number of threads could lead to severe underutilization of the GPU resources.

- A general recommendation:
  - Child grids with a large number of thread blocks,
  - or at least thread blocks with hundreds of threads, if the number of blocks is small.

- Nested parallelism (tree processing):
  - Thick tree nodes (each node deploys many threads),
  - and/or branch degree is large (each parent node has many children)
  - As the nesting depth is limited in hardware, only relatively shallow trees can be implemented efficiently.
Optimization for Dynamic Parallelism
Alleviating Launch Overhead

- Dynamic Parallelism (CUDA, OpenCL 2.0)
  - Dynamic load balancing
  - Data-dependent execution
  - Recursion
  - Programmability and maintainability

- Many fine-grain child kernels incur high kernel launch overhead and underutilization of the GPU resources

- Launch overhead on the critical path and limited depth of call stack
Kernel Launch Aggregation

Performance of Kernel Launch Aggregation

Increasing aggregation granularity improves performance (geomean speedup of 6.58x for K-aggregation on Kepler)

HetSys Course: Lecture 14: Dynamic Parallelism (Fall 2022)

https://youtu.be/-l5_qqQHZQc
Recommended Readings (II)

  - Chapter 21 - CUDA dynamic parallelism
P&S Heterogeneous Systems
Dynamic Parallelism

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Spring 2023
9 June 2023