Dynamic Parallelism

- GPU programming frameworks provide an interface to express **dynamic refinement algorithms** in a more natural way
  - 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**
Example: Turbulence Simulation

- Fixed grid vs. dynamic grid for a turbulence simulation model

- **Initial Grid**
  - Statically assign conservative worst-case grid

- **Fixed Grid**

- **Dynamic Grid**
  - Dynamically assign performance where accuracy is required

4
CPU-GPU without and with dynamic parallelism

(a) Without Dynamic Parallelism

(b) With Dynamic Parallelism
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
Kernel Launch with Dynamic Parallelism

Easier to write programs with dynamically discovered parallelism
**Nested Dependencies**

- **B** is the **parent kernel** of **X**, **Y**, and **Z**
- **X**, **Y**, and **Z** are **child kernels** of **B**

```c
int main() {
    float *data;
    setup(data);
    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);
    cudaDeviceSynchronize();
    return 0;
}
```

```c
__global__ void B(float *data) {
    do_stuff(data);
    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaDeviceSynchronize();
    do_more_stuff(data);
}
```
Example: LU Decomposition

- Solving systems of linear equations

LU decomposition (Fermi)

```c
void dgetrf(int N, int N) {
    for (int j = 0; j < N; j += 64) {
        for (int i = 0; i < 64; i += 1) {
            idamax();
            dswap();
            dscal();
            dger();
        }
        memcpy();
        dlaswp();
        dtrsm();
        dgemm();
    }
}
```

LU decomposition (Kepler)

```c
void dgetrf(int N, int N) {
    dgetrf<<<>>>(N, N);
    synchronize();
}

void dgetrf(int N, int N) {
    for (int j = 0; j < N; j += 64) {
        for (int i = 0; i < 64; i += 1) {
            idamax();
            dswap();
            dscal();
            dger();
        }
        dlaswp();
        dtrsm();
        dgemm();
    }
    synchronize();
}
```
A parent kernel launches a child kernel with the same syntax as the host

```
kernel_name<<< Dg, Db, Ns, S>>>([kernel arguments]);
```

- **Dg** is of type `dim3` and specifies the dimensions and size of the grid
- **Db** is of type `dim3` and specifies the dimensions and size of each thread block
- **Ns** is of type `size_t` and specifies the number of bytes of shared memory that is dynamically allocated per thread block for this call
- **S** is of type `cudaStream_t` and specifies the stream associated with this call
Parent-Child Synchronization

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

- Without dynamic parallelism

```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; j < end; ++j) {
        doMoreWork(moreData[j], i);
    }
}
```

Threads

Iterations
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]);
    }
}
```
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 More Complex Example
Beziers Lines (I)

- Linear Bezier curves

\[ B(t) = P_0 + t(P_1 - P_0) = (1-t)P_0 + tP_1, \; t \in [0, 1] \]

- Quadratic Bezier curves

\[ B(t) = (1 - t)[(1 - t)P_0 + tP_1] + t[(1 - t)P_1 + tP_2], \; t \in [0, 1] \]

\[ B(t) = (1 - t)^2P_0 + 2(1 - t)tP_1 + t^2P_2, \; t \in [0, 1]. \]
Bezier Lines (II)

- Control points
  - Curvature calculation
  - Tessellation points

http://cubic-bezier.com
Without dynamic parallelism: One line per block

```c
__global__ void computeBezierLines(BezierLine *bLines, int nLines) {
    int bidx = blockIdx.x;
    if(bidx < nLines){
        //Compute the curvature of the line
        float curvature = computeCurvature(bLines);

        //From the curvature, compute the number of tessellation points
        int nTessPoints = min(max((int)(curvature*16.0f),4),32);
        bLines[bidx].nVertices = nTessPoints;

        //Loop through vertices to be tessellated, incrementing by blockDim.x
        for(int inc = 0; inc < nTessPoints; inc += blockDim.x){
            int idx = inc + threadIdx.x; //Compute a unique index for this point
            if(idx < nTessPoints){
                float u = (float)idx/(float)(nTessPoints-1); //Compute u from idx
                float omu = 1.0f - u; //pre-compute one minus u

                float B3u[3]; //Compute quadratic Bezier coefficients
                B3u[0] = omu*omo;
                B3u[1] = 2.0f*u*omo;
                B3u[2] = u*u;

                float2 position = {0,0}; //Set position to zero
                for(int i = 0; i < 3; i++){
                    //Add the contribution of the i'th control point to position
                    position = position + B3u[i] * bLines[bidx].CP[i];
                }

                //Assign value of vertex position to the correct array element
                bLines[bidx].vertexPos[idx] = position;
            }
        }
    }
}
```
Beziers Lines with Dynamic Parallelism (I)

- With dynamic parallelism
- Parent: One line per thread

```c
__global__ void computeBezierLines_parent(BezierLine *bLines, int nLines) {
    // Compute a unique index for each Bezier line
    int lidx = threadIdx.x + blockDim.x*blockIdx.x;
    if (lidx < nLines) {  
        // Compute the curvature of the line
        float curvature = computeCurvature(bLines);
        // From the curvature, compute the number of tessellation points
        bLines[lidx].nVertices = min(max((int)(curvature*16.0f),4),MAX_TESS_POINTS);
        cudaMalloc((void**)&bLines[lidx].vertexPos, bLines[lidx].nVertices*sizeof(float2));
        // Call the child kernel to compute the tessellated points for each line
        computeBezierLine_child<<<ceil((float)bLines[lidx].nVertices/32.0f), 32>>>  
            (lidx, bLines, bLines[lidx].nVertices);
    }
}
```
Beziers Lines with Dynamic Parallelism (II)

- With dynamic parallelism
- Child

```c
__global__ void computeBezierLine_child(int lidx, BezierLine* bLines,
int nTessPoints) {
  int idx = threadIdx.x + blockDim.x*blockIdx.x;  // Compute idx unique to this vertex
  if(idx < nTessPoints){
    float u = (float)idx/(float)(nTessPoints-1);   // Compute u from idx
    float omu = 1.0f - u;                          // Pre-compute one minus u

    float B3u[3];                                 // Compute quadratic Bezier coefficients
    B3u[0] = omu*omu;
    B3u[1] = 2.0f*u*omu;
    B3u[2] = u*u;

    float2 position = {0,0};                      // Set position to zero
    for(int i = 0; i < 3; i++) {
      // Add the contribution of the i'th control point to position
      position = position + B3u[i] * bLines[lidx].CP[i];
    }

    // Assign the value of the vertex position to the correct array element
    bLines[lidx].vertexPos[idx] = position;
  }
}
```
Launch Pool (I)

- Launch pool size
  - Fixed-size pool: default 2048
  - Variable-size pool

Before CUDA 6.0

Since CUDA 6.0
Launch Pool (II)

- Effect of increasing the fixed-size launch pool
  - Number of Bezier lines = 16384
CUDA Streams

- **CUDA streams** (command queues in OpenCL)
- Sequence of operations that are performed in order
  - 1. Data transfer CPU-GPU
  - 2. Kernel execution
    - D input data instances, B blocks
    - \#Streams: \((D / \#Streams)\) data instances, \((B / \#Streams)\) blocks
  - 3. Data transfer GPU-CPU

<Diagram of CUDA Streams with time intervals for data copy and kernel execution>
Streams for Child Kernel Launch (I)

- Default stream vs. several streams per parent block

![Diagram showing default stream vs. several streams per parent block]
Streams for Child Kernel Launch (II)

- Default stream vs. several streams per parent block

```c
//Call the child kernel to compute the tessellated points for each line
// Default stream
computeBezierLine_child<<ceil((float)bLines[lidx].nVertices/32.0f), 32>>>
(lidx, bLines, bLines[lidx].nVertices);

cudaStream_t stream;
// Create non-blocking stream
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);

//Call the child kernel to compute the tessellated points for each line
computeBezierLine_child<<ceil((float)bLines[lidx].nVertices/32.0f), 32, 0, stream>>>
(lidx, bLines, bLines[lidx].nVertices);

// Destroy stream
cudaStreamDestroy(stream);
```
Streams for Child Kernel Launch (III)

- Performance impact of per-thread streams

![Graph showing performance impact of per-thread streams on Tesla K40](image)

- Default stream
- Per-thread stream

Execution time (ms) vs Number of lines
A Recursive Example
Arranging Input for Gather Parallelization

- Input data distribution in space may be far from uniform
  - Quad/Oct Trees

- One may not be able to pre-determine the input elements needed for calculating output elements
  - Dynamic Search

- ...
A Recursive Example: Quadtree (I)

- 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 = 0

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.

![Diagram of Quadtree]

Depth = 1

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 = 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.

A Recursive Example: Quadtree (IV)

- 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

Exit N
A Recursive Example: Quadtree (VI)

- Each block launches 1 child grid of 4 blocks

Depth = 1
A Recursive Example: Quadtree (VII)

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

- Each block launches 1 child grid of 4 blocks

Depth = 3

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

Buffer 0: b c e f m n g j o a d h k p q i l r t s u
A Recursive Example: Quadtree (IX)

- Points in the same quadrant are grouped together

Each thread block may need to move its part to Buffer 0 before retiring, since there is where we expect the final output.
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
Other Use Cases
Simple library calls

```c
__global__ void libraryCall(float *a,
                          float *b,
                          float *c)
{
    // All threads generate data
    createData(a, b);
    __syncthreads();

    // The first thread calls library
    if (threadIdx.x == 0) {
        cublasDgemm(a, b, c);
        cudaDeviceSynchronize();
    }

    // All threads wait for results
    __syncthreads();

    consumeData(c);
}
```
Use Case: Avoid Launch Overhead (I)

- Lattice QCD

**Bottlenecks**

- Large number of calls to cuBLAS
- Dominated by CPU’s capability of launching cuBLAS kernels
- ARM CPU is not fast enough to quickly launch kernels: GPU is underutilized

Vishal Mehta, “Exploiting CUDA Dynamic Parallelism for low power ARM based prototypes,” GTC 2015
Use Case: Avoid Launch Overhead (II)

- Saving kernel launches in ARM CPUs

**CPU pipeline**

Start

**GPU pipeline**

CPU works as coordinator

GPU slave executes kernels

Apply matrix

cuBLAS dot kernel

cuBLAS AXPY kernel

End

**GPU wrapper coordinates the tasks**

CPU starts and ends wrapper

GPU wrapper coordinates the tasks

CPU pipeline

Start

Wrapper kernel, 1 control thread

GPU pipeline

Apply matrix

cuBLAS dot kernel

cuBLAS AXPY kernel

End
Use Case: Avoid Launch Overhead (III)

- Saving kernel launches in ARM CPUs

<table>
<thead>
<tr>
<th>No of kernel calls</th>
<th>cuBLAS calls by CPU (seconds)</th>
<th>cuBLAS calls GPU thread (seconds)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \times 10^3$</td>
<td>1.72</td>
<td>1.43</td>
<td>1.20 x</td>
</tr>
<tr>
<td>$3 \times 10^3$</td>
<td>2.23</td>
<td>1.62</td>
<td>1.37 x</td>
</tr>
<tr>
<td>$5 \times 10^3$</td>
<td>4.7</td>
<td>2.9</td>
<td>1.62 x</td>
</tr>
<tr>
<td>$10 \times 10^3$</td>
<td>7.52</td>
<td>3.5</td>
<td>2.14 x</td>
</tr>
<tr>
<td>$50 \times 10^3$</td>
<td>11.78</td>
<td>4.2</td>
<td>2.80 x</td>
</tr>
</tbody>
</table>
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
Problems:
Many kernels incur too much launch overhead

Fine-grain kernels underutilize the GPU resources
Motivation: Launch Overhead (II)

- Non-trivial overhead of device-side kernel launching
  - Parameter allocation, launch command, dispatch kernel
  - Pending kernels, suspended parents
- Ideal CDP vs. Actual CDP (speedup over non-CDP versions)

![Graph showing speedup of CDP implementations](image-url)
Idea: Kernel Launch Aggregation

Proposed Solution:

Kernel Launch Aggregation

Each thread can launch a separate kernel

Idea: Kernel Launch Aggregation

Each thread can launch a separate kernel

Warp-Granularity

Warp Granularity

Warp-Granularity Kernel Launch Aggregation

Block Granularity

Block-Granularity Kernel Launch Aggregation

Block Granularity

Block-Granularity Kernel Launch Aggregation

Kernel Launch Aggregation

Block Granularity Aggregation (I)

Original Kernel Call

allocate arrays for args, gD, and bD
store args in arg arrays
store gD in gD array, and bD in bD array
new gD = sum of gD array across warp/block
new bD = max of bD array across warp/block
if(threadIdx == launcher thread in warp/block) {
    kernel_agg <<< new gD, new bD >>>
    (arg arrays, gD array, bD array)
}

Transformed Kernel Call
(block-granularity aggregation example)

Block Granularity Aggregation (II)

__global__ void kernel(params) {
    kernel body
}

Original Kernel

__global__ void kernel_agg (param arrays, gD array, bD array) {
    calculate index of parent thread (original kernel index) #
    load params from param arrays
    load actual gridDim/blockDim from gD/bD arrays
    calculate actual blockIdx #
    if(threadIdx < actual blockDim) {
        kernel body (with kernel launches transformed and with using actual gridDim/blockDim/blockIdx)
    }
}

Transformed Kernel
(block-granularity aggregation example)

Block Granularity Aggregation (III)

calculate index of parent thread

gD = \{1, 2, 0, 1\}

gDs = \text{scan}(gD) = \{0, 1, 3, 4\} \quad \text{(performed by parent thread as optimization)}

blockIdx.x = 0 1 2 3

# = p | gDs[p] ≤ blockIdx.x < gDs[p+1] = 0 1 1 3 \quad \text{(n-arry search on gDs for p satisfying this condition)}

# = blockIdx.x - gDs[p] = 0 0 1 0

calculate actual blockIdx

Performance of Kernel Launch Aggregation

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

Profiling of Kernel Launch Aggregation

Performance improvement comes from reduced launch overhead and better resource utilization.

Summary: Dynamic Parallelism

- **CUDA Dynamic Parallelism**
  - Extends the CUDA programming model to allow kernels to launch kernels
  - Dynamic memory allocation
  - Dynamically discovered work
  - Recursive algorithms
  - Better work balance and more efficient memory usage

- **Performance limitations**

- **Launch overhead**
  - Kernel launch aggregation
Recommended Readings

  - Chapter 13 - CUDA dynamic parallelism
P&S Heterogeneous Systems
Dynamic Parallelism

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich
Fall 2021
22 December 2021