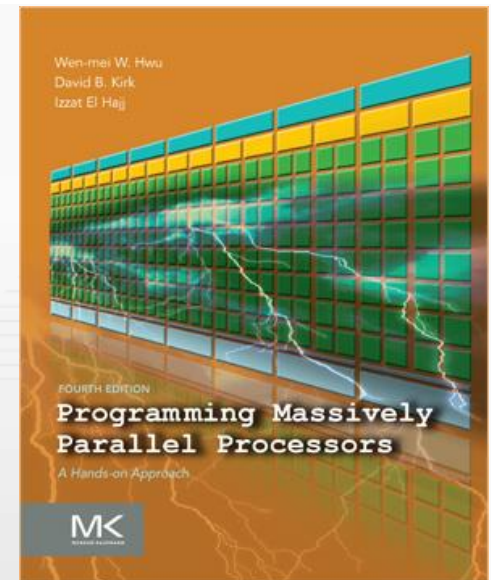


Programming Massively Parallel Processors

A Hands-on Approach

CHAPTER 21 ➤ CUDA Dynamic Parallelism



- Introduction to CUDA Dynamic Parallelism
- A simple example
- A more complex example: Bezier lines
- A recursive example: Quadtree
- Other cases:
 - Library calls
 - Saving kernel launches in ARM CPUs
- What is and what is not dynamic parallelism
- Summary

- 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

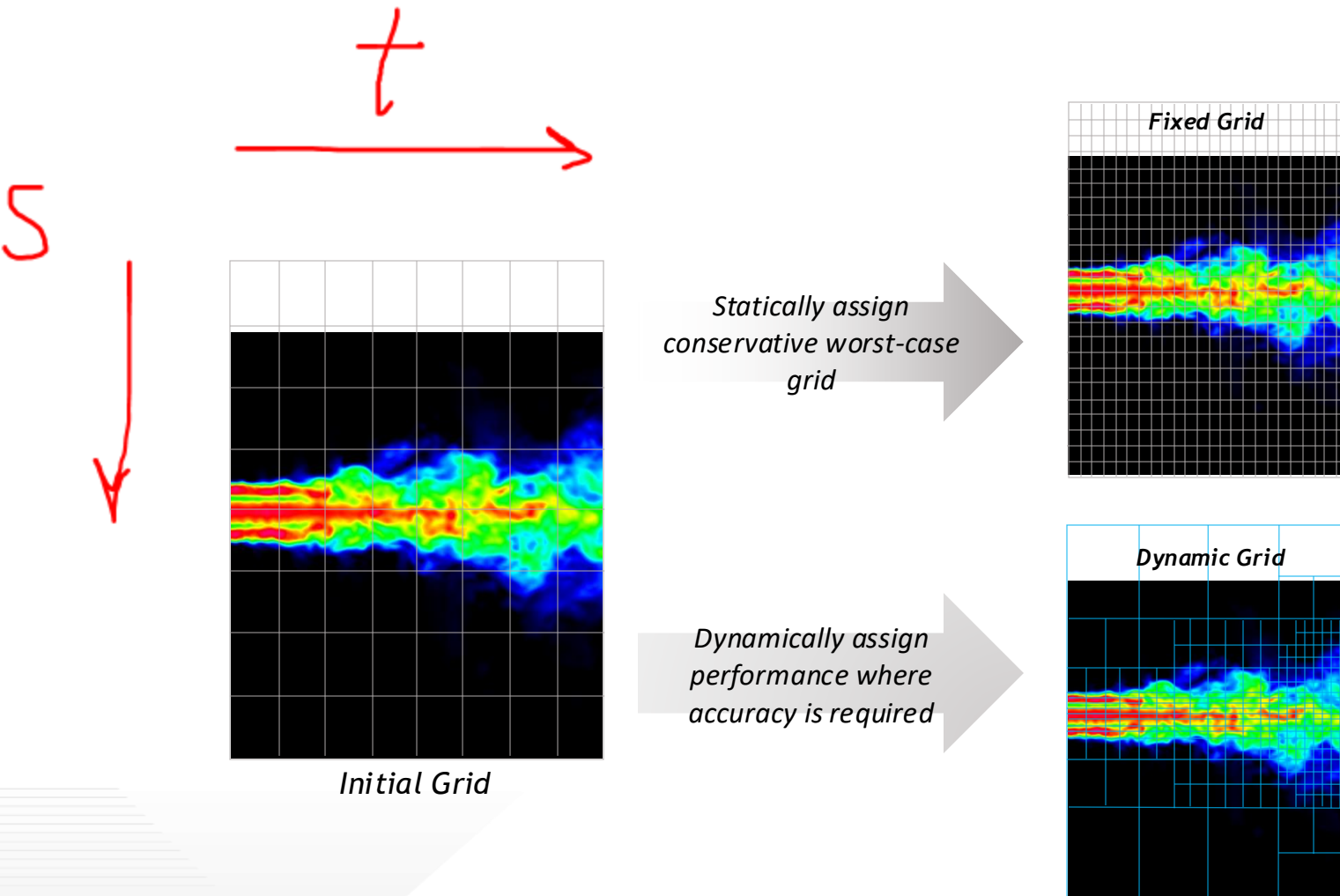


Fermi: Only CPU
can generate GPU work.



Kepler: GPU can
generate work for itself.

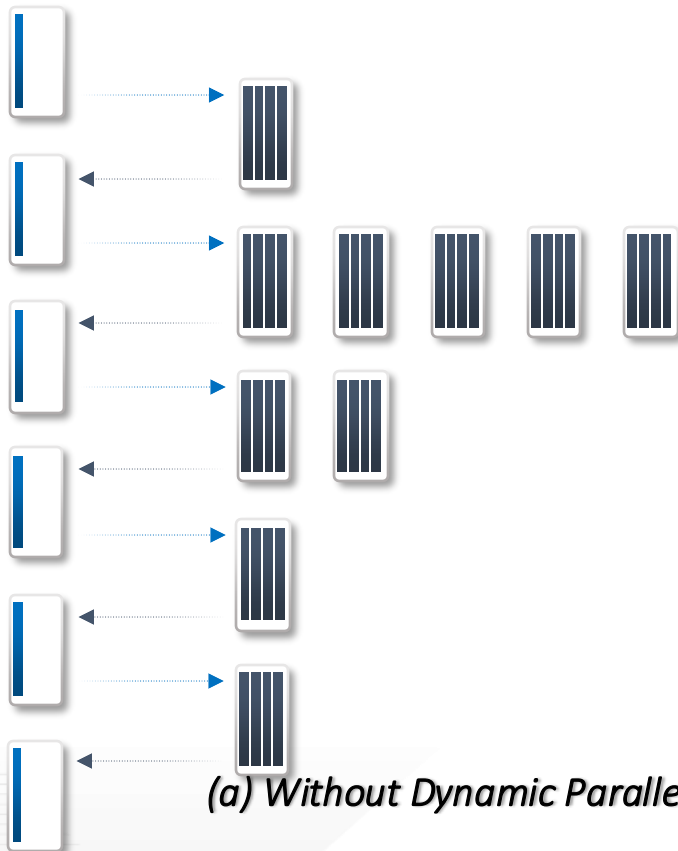
- Fixed grid vs. dynamic grid for a turbulence simulation model



- CPU-GPU without and with dynamic parallelism

CPU

GPU



(a) Without Dynamic Parallelism

CPU

GPU



(b) With Dynamic Parallelism

- Nested dependencies

```
int main() {
    float *data;
    setup(data);

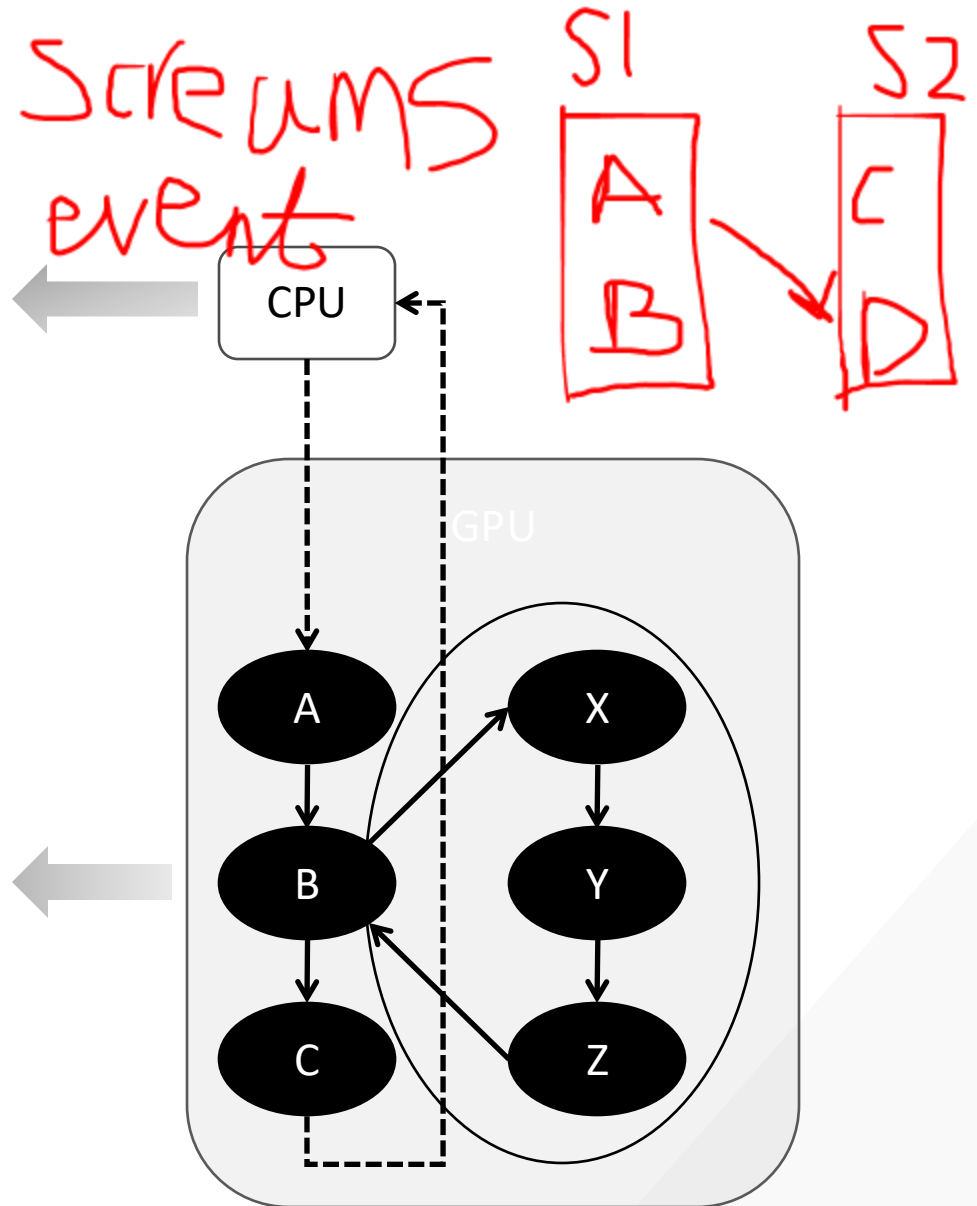
    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaDeviceSynchronize();
    return 0;
}
```

```
__global__ void B(float *data)
{
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaDeviceSynchronize();

    do_more_stuff(data);
}
```



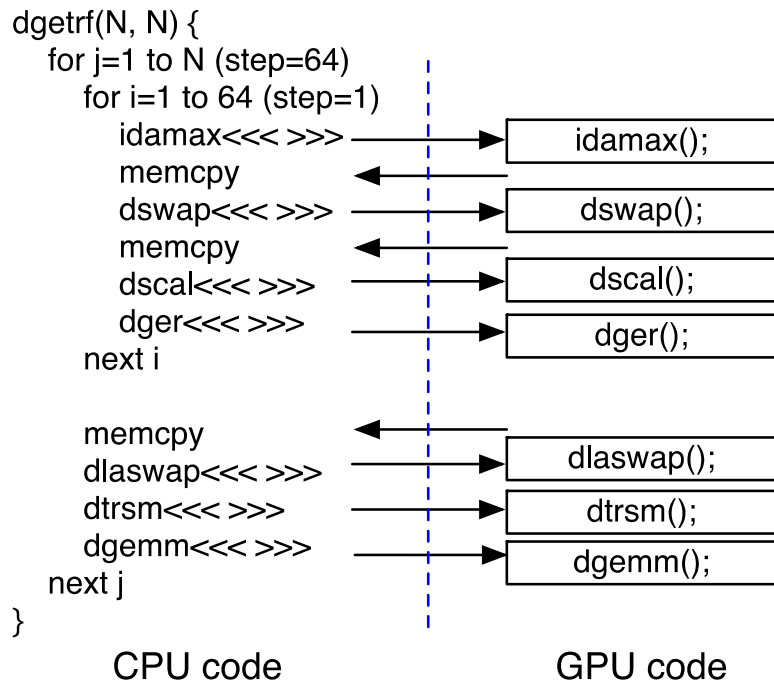
- Syntax

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

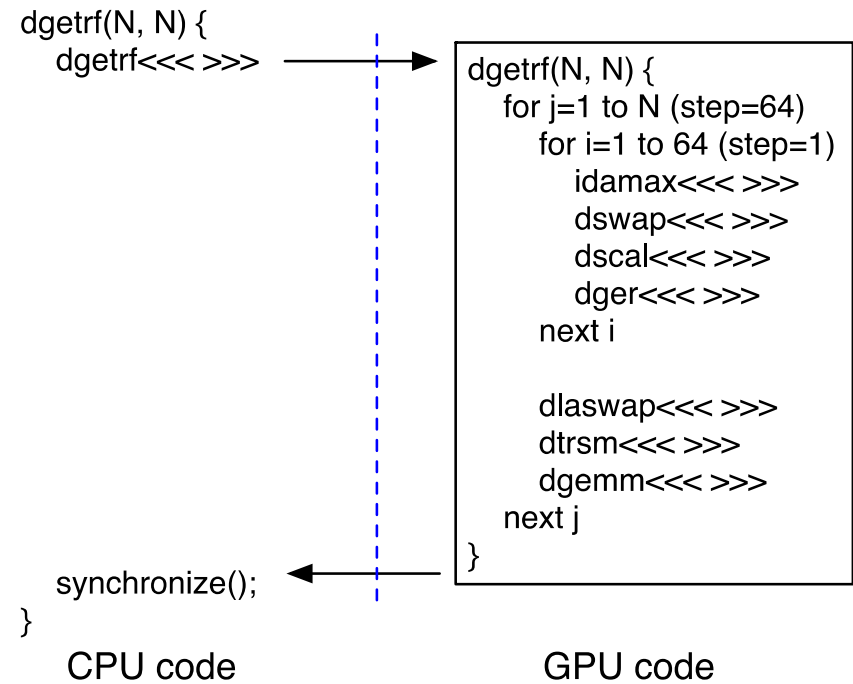
- Dg is of type dim3 and specifies the dimensions and size of the grid
- Db is of type dim3 and specifies the dimensions and size of each thread block
- Ns is of type size_t and specifies the number of bytes of shared memory that is dynamically allocated per thread block for this call.
- S is of type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

- LU decomposition

LU decomposition (Fermi)

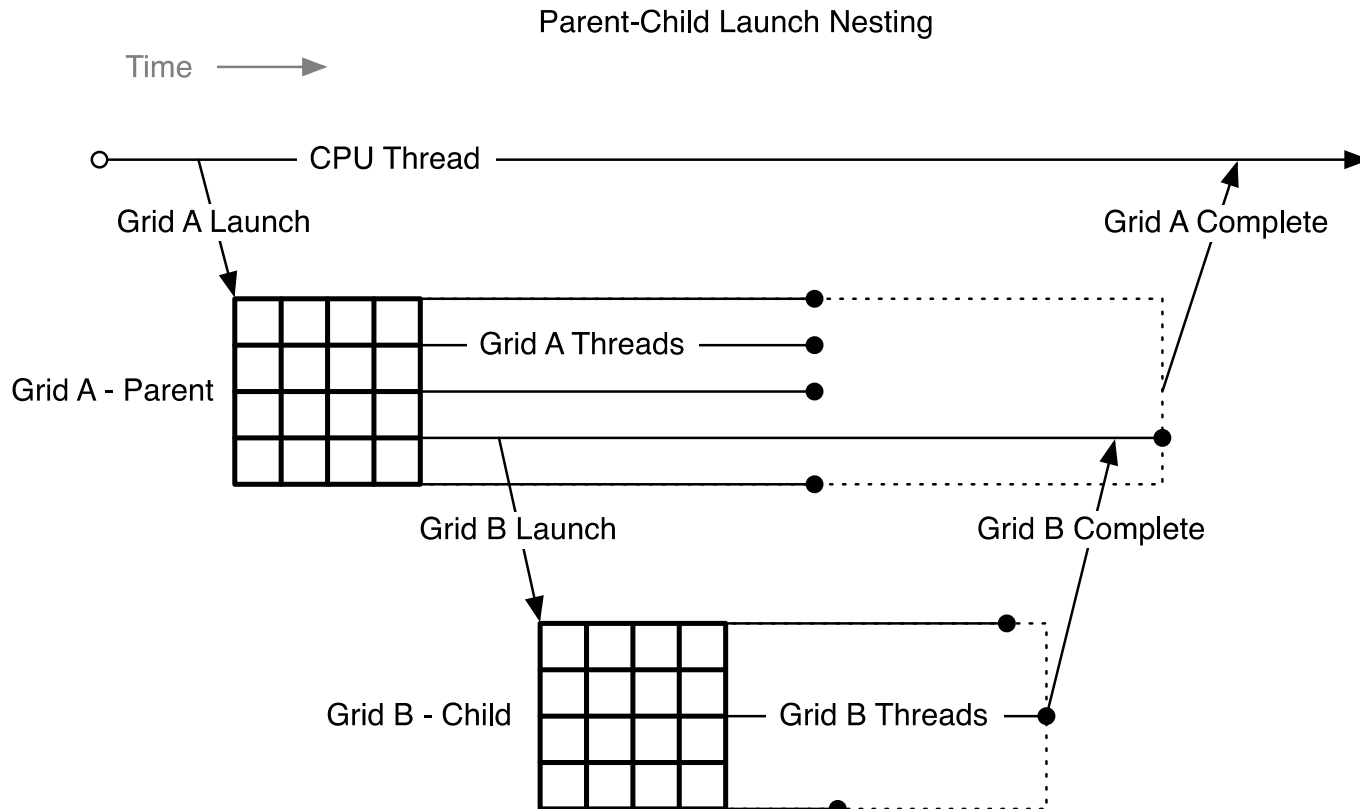


LU decomposition (Kepler)



- Synchronization

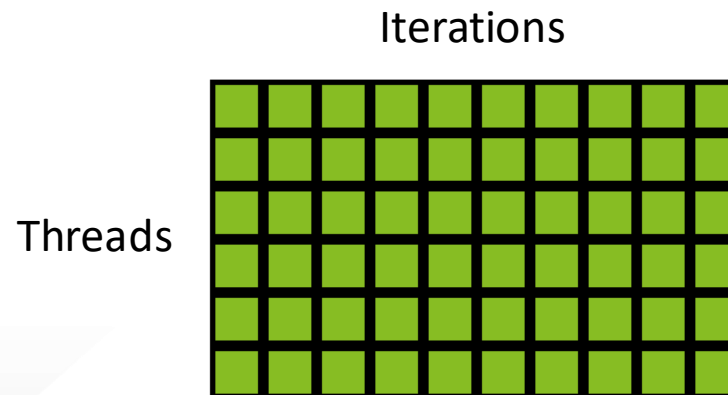
- Parent to child: memory consistency
- Child to parent: after `cudaDeviceSynchronize()`



- Introduction to CUDA Dynamic Parallelism
- **A simple example**
- A more complex example: Bezier lines
- A recursive example: Quadtree
- Other cases:
 - Library calls
 - Saving kernel launches in ARM CPUs
- What is and what is not dynamic parallelism
- Summary

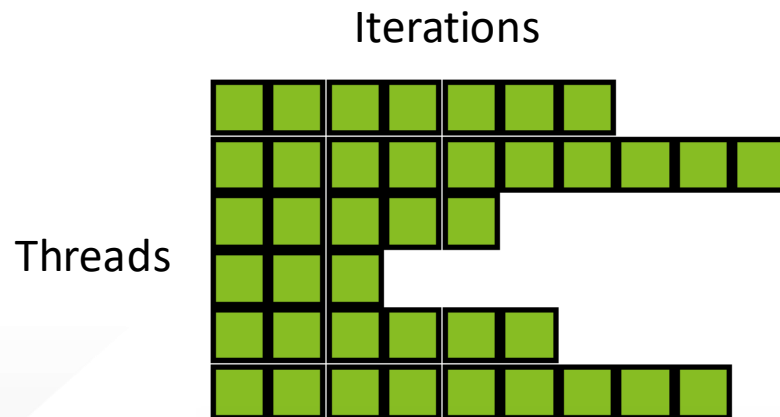
- Without dynamic parallelism, uniform workload

```
01  __global__ void kernel(unsigned int start, unsigned int end,  
02      float* someData, float* moreData) {  
03  
04      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;  
05      doSomeWork(someData[i]);  
06  
07      for(unsigned int j = start; j < end; ++j) {  
08          doMoreWork(moreData[j], i);  
09      }  
10  
11  }
```



- Without dynamic parallelism, non-uniform workload

```
01  __global__ void kernel(unsigned int* start, unsigned int* end,  
02      float* someData, float* moreData) {  
03  
04      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;  
05      doSomeWork(someData[i]);  
06  
07      for(unsigned int j = start[i]; j < end[i]; ++j) {  
08          doMoreWork(moreData[j]);  
09      }  
10  
11  }
```



- With dynamic parallelism, non-uniform workload

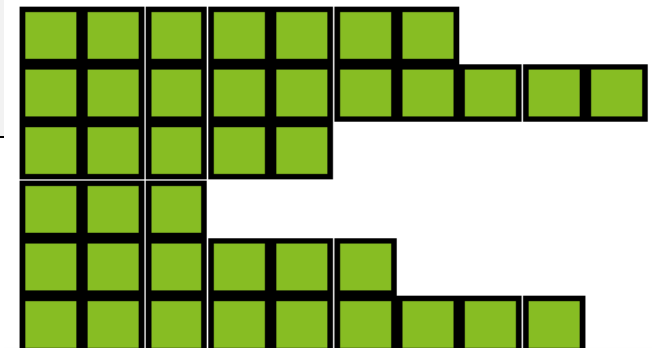
```

01  __global__ void kernel_parent(unsigned int* start, unsigned int* end,
02      float* someData, float* moreData) {
03
04      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
05      doSomeWork(someData[i]);
06
07      kernel_child <<< ceil((end[i]-start[i])/256.0) , 256 >>>
08          (start[i], end[i], moreData);
09
10  }
11
12  __global__ void kernel_child(unsigned int start, unsigned int end,
13      float* moreData) {
14
15      unsigned int j = start + blockIdx.x*blockDim.x + threadIdx.x;
16
17      if(j < end) {
18          doMoreWork(moreData[j]);
19      }
20
21  }

```

Kernel calls

Child threads



- Introduction to CUDA Dynamic Parallelism
- A simple example
- **A more complex example: Bezier lines**
- A recursive example: Quadtree
- Other cases:
 - Library calls
 - Saving kernel launches in ARM CPUs
- What is and what is not dynamic parallelism
- Summary

- Linear Bezier curves

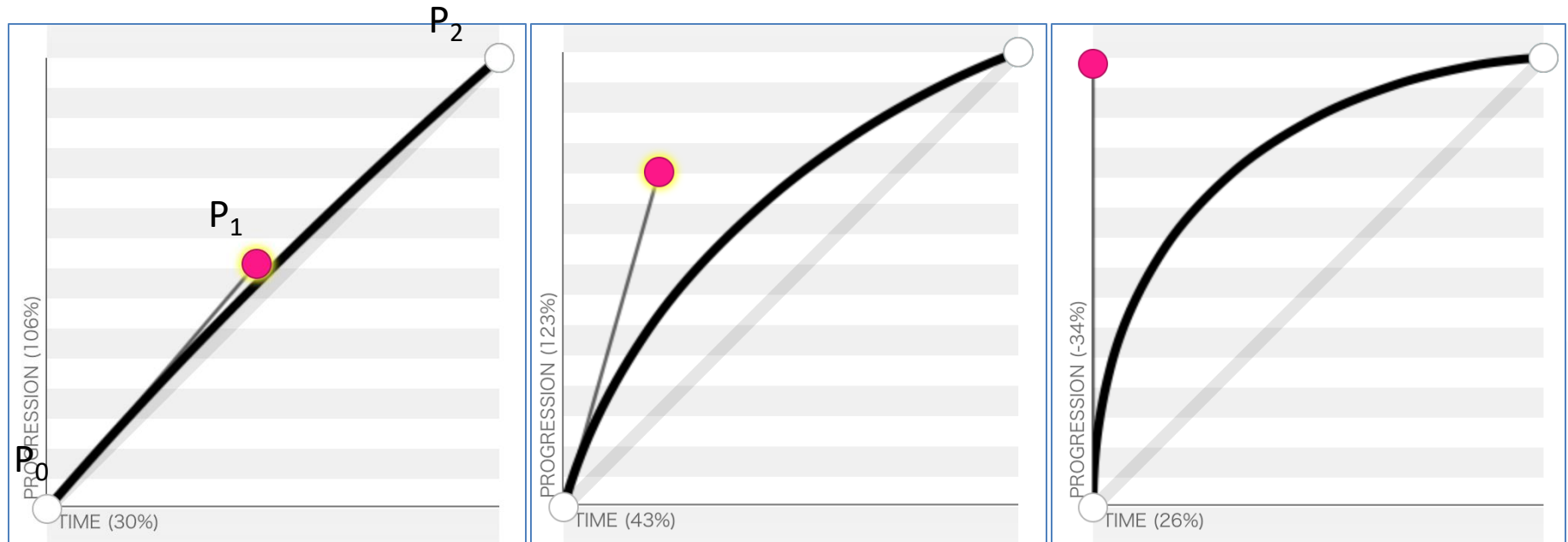
$$\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0) = (1 - t)\mathbf{P}_0 + t\mathbf{P}_1, t \in [0, 1]$$

- Quadratic Bezier curves

$$\mathbf{B}(t) = (1 - t)[(1 - t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1 - t)\mathbf{P}_1 + t\mathbf{P}_2], t \in [0, 1]$$

$$\mathbf{B}(t) = (1 - t)^2\mathbf{P}_0 + 2(1 - t)t\mathbf{P}_1 + t^2\mathbf{P}_2, t \in [0, 1].$$

- Control points
 - Curvature calculation
 - Tessellation points



- Without dynamic parallelism: One line per block

```

046  __global__ void computeBezierLines(BezierLine *bLines, int nLines) {
047      int bidx = blockIdx.x;
048      if(bidx < nLines){
049          //Compute the curvature of the line
050          float curvature = computeCurvature(bLines);
051
052          //From the curvature, compute the number of tessellation points
053          int nTessPoints = min(max((int)(curvature*16.0f),4),32);
054          bLines[bidx].nVertices = nTessPoints;
055
056          //Loop through vertices to be tessellated, incrementing by blockDim.x
057          for(int inc = 0; inc < nTessPoints; inc += blockDim.x){
058              int idx = inc + threadIdx.x; //Compute a unique index for this point
059              if(idx < nTessPoints){
060                  float u = (float)idx/(float)(nTessPoints-1); //Compute u from idx
061                  float omu = 1.0f - u; //pre-compute one minus u
062
063                  float B3u[3]; //Compute quadratic Bezier coefficients
064                  B3u[0] = omu*omu;
065                  B3u[1] = 2.0f*u*omu;
066                  B3u[2] = u*u;
067
068                  float2 position = {0,0}; //Set position to zero
069                  for(int i = 0; i < 3; i++){
070                      //Add the contribution of the i'th control point to position
071                      position = position + B3u[i] * bLines[bidx].CP[i];
072                  }
073                  //Assign value of vertex position to the correct array element
074                  bLines[bidx].vertexPos[idx] = position;
075              }
076          }
077      }

```

- With dynamic parallelism
- Parent: One line per thread

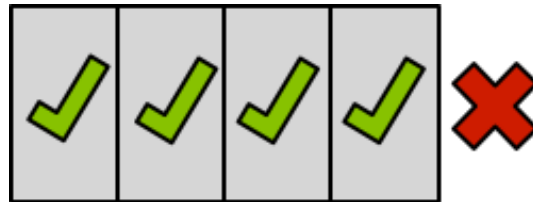
```
30  __global__ void computeBezierLines_parent(BezierLine *bLines, int nLines) {
31      //Compute a unique index for each Bezier line
32      int lid = threadIdx.x + blockDim.x*blockIdx.x;
33      if(lid < nLines){
34          //Compute the curvature of the line
35          float curvature = computeCurvature(bLines);
36          //From the curvature, compute the number of tessellation points
37          bLines[lid].nVertices = min(max((int)(curvature*16.0f), 4), MAX_TESS_POINTS);
38          cudaMalloc((void**) &bLines[lid].vertexPos, bLines[lid].nVertices*sizeof(float2));
39          //Call the child kernel to compute the tessellated points for each line
40          computeBezierLine_child<<<ceil((float)bLines[lid].nVertices/32.0f), 32>>>
41              (lid, bLines, bLines[lid].nVertices);
42      }
43  }
```

- With dynamic parallelism
- Child

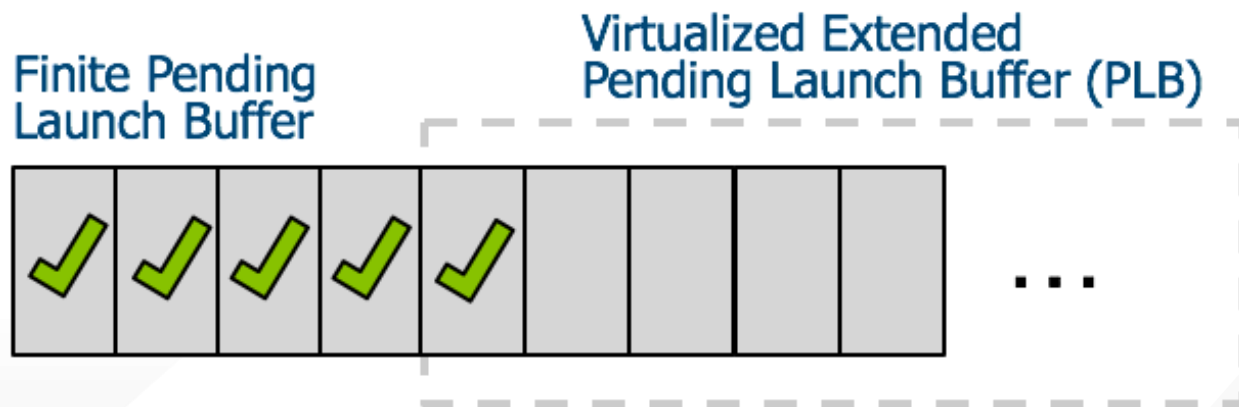
```
07  __global__ void computeBezierLine_child(int lidx, BezierLine* bLines,
08  int nTessPoints) {
09  int idx = threadIdx.x + blockDim.x*blockIdx.x;  //Compute idx unique to this vertex
10  if(idx < nTessPoints){
11      float u = (float)idx/(float)(nTessPoints-1);  //Compute u from idx
12      float omu = 1.0f - u;  //Pre-compute one minus u
13
14      float B3u[3];  //Compute quadratic Bezier coefficients
15      B3u[0] = omu*omu;
16      B3u[1] = 2.0f*u*omu;
17      B3u[2] = u*u;
18
19      float2 position = {0,0};  //Set position to zero
20      for(int i = 0; i < 3; i++) {
21          //Add the contribution of the i'th control point to position
22          position = position + B3u[i] * bLines[lidx].CP[i];
23      }
24
25      //Assign the value of the vertex position to the correct array element
26      bLines[lidx].vertexPos[idx] = position;
27  }
28  }
```

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

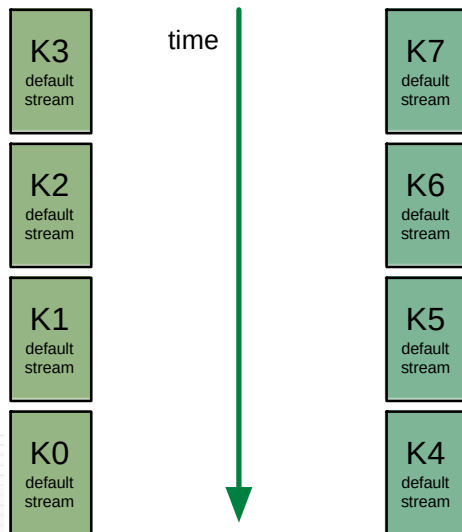
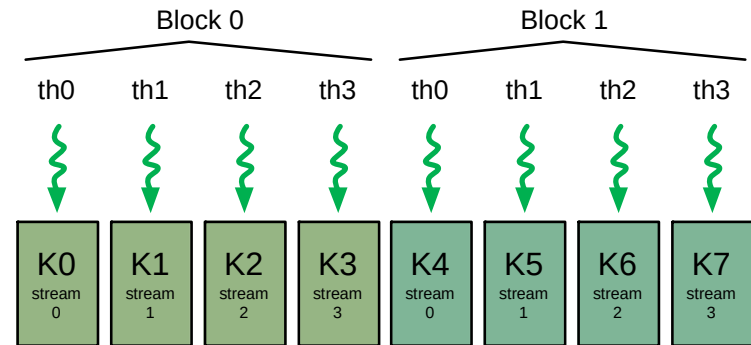
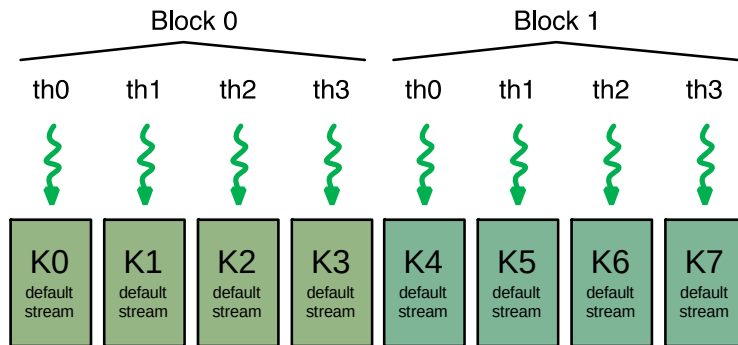
Before CUDA 6.0



Since CUDA 6.0



- Streams



- Streams

```
//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);
```

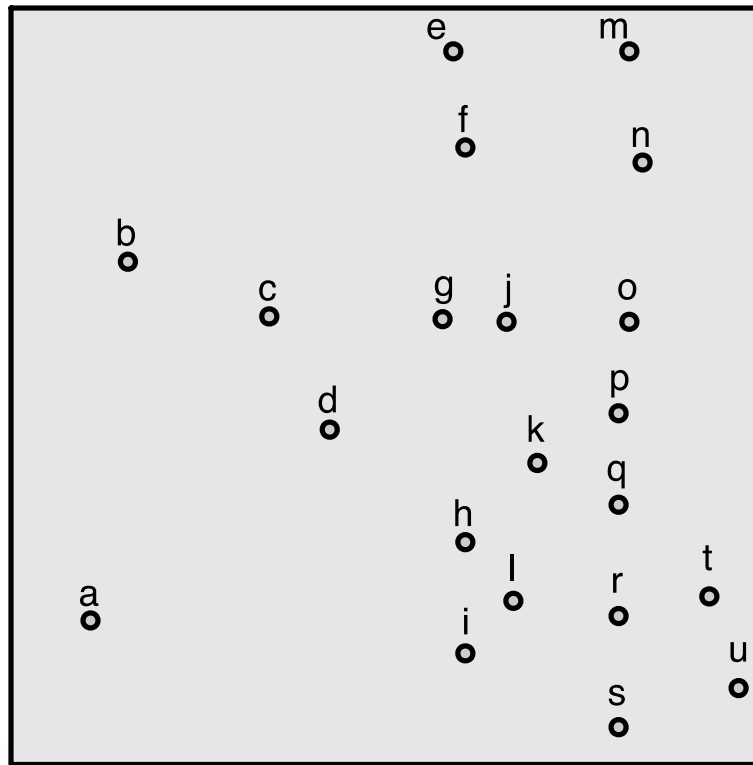
```
cudaStream_t stream;
// Create non-blocking stream
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);

//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);
```

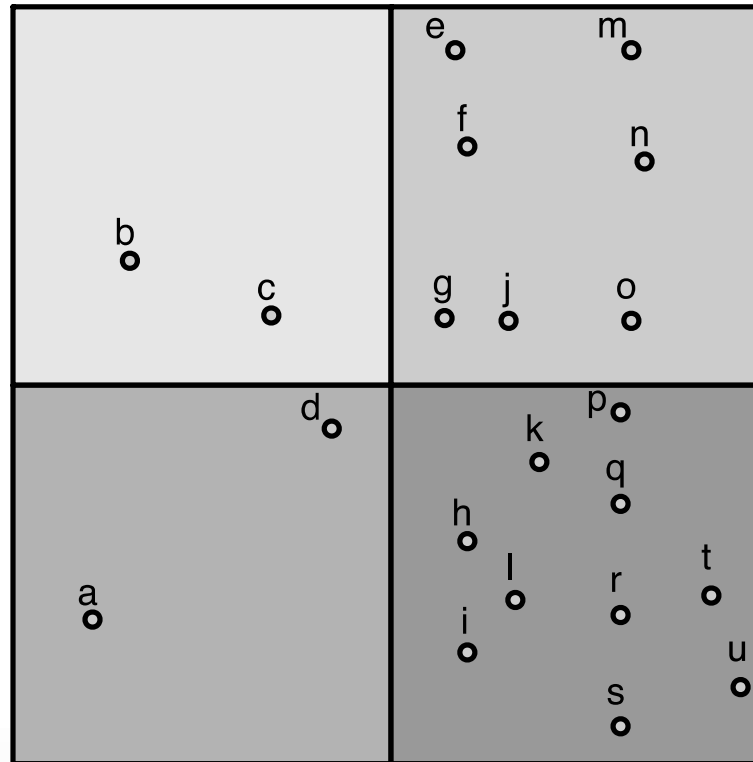
- Introduction to CUDA Dynamic Parallelism
- A simple example
- A more complex example: Bezier lines
- A recursive example: Quadtree
- Other cases:
 - Library calls
 - Saving kernel launches in ARM CPUs
- What is and what is not dynamic parallelism
- Summary

- Partitioning a 2D space by recursively dividing it into four quadrants



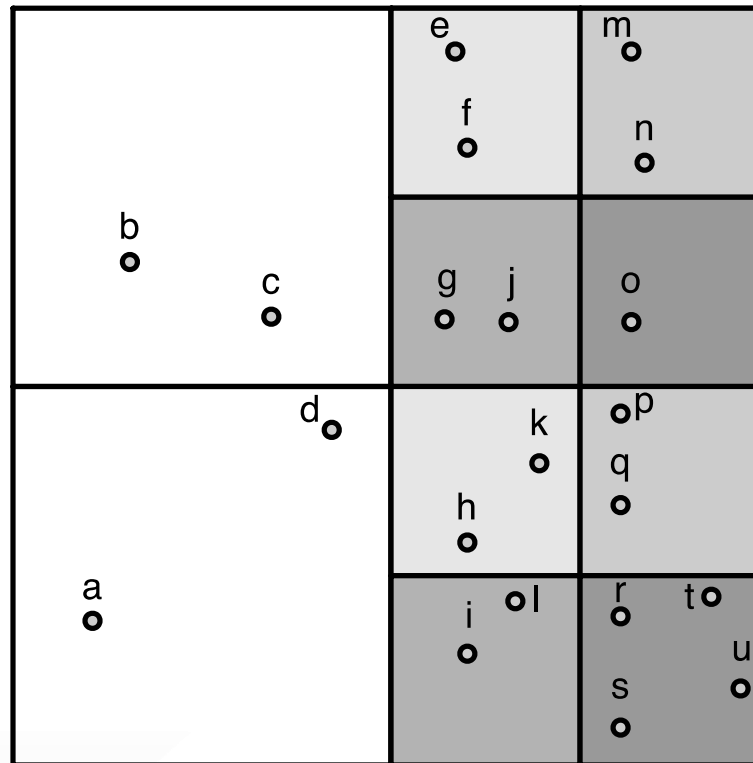
Depth = 0

- Partitioning a 2D space by recursively dividing it into four quadrants



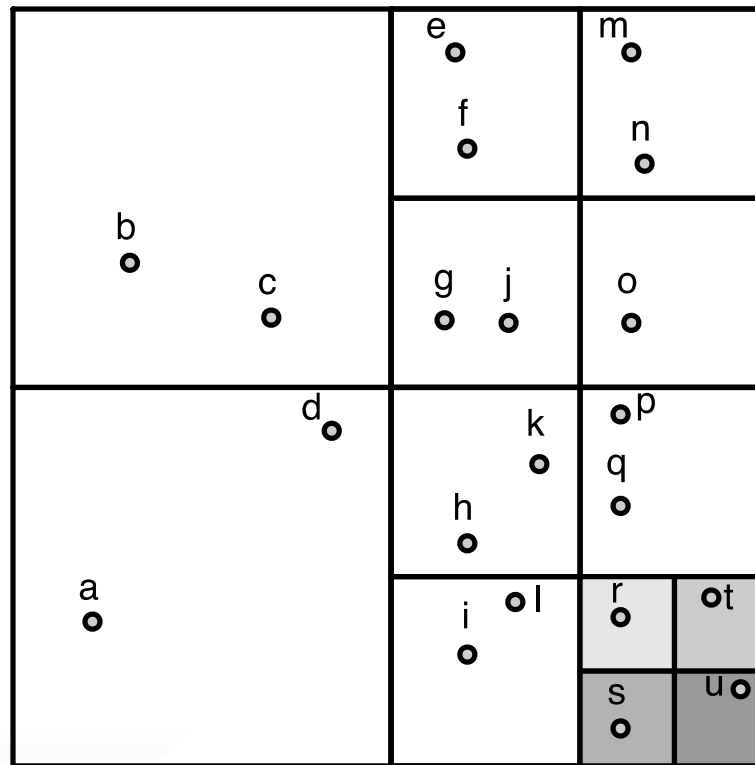
Depth = 1

- Partitioning a 2D space by recursively dividing it into four quadrants



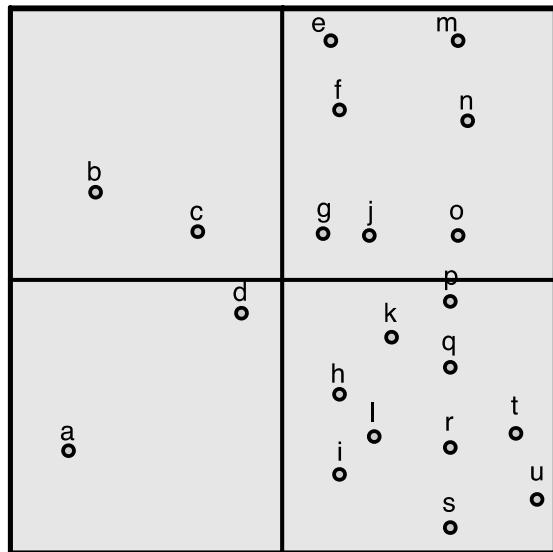
Depth = 2

- Partitioning a 2D space by recursively dividing it into four quadrants

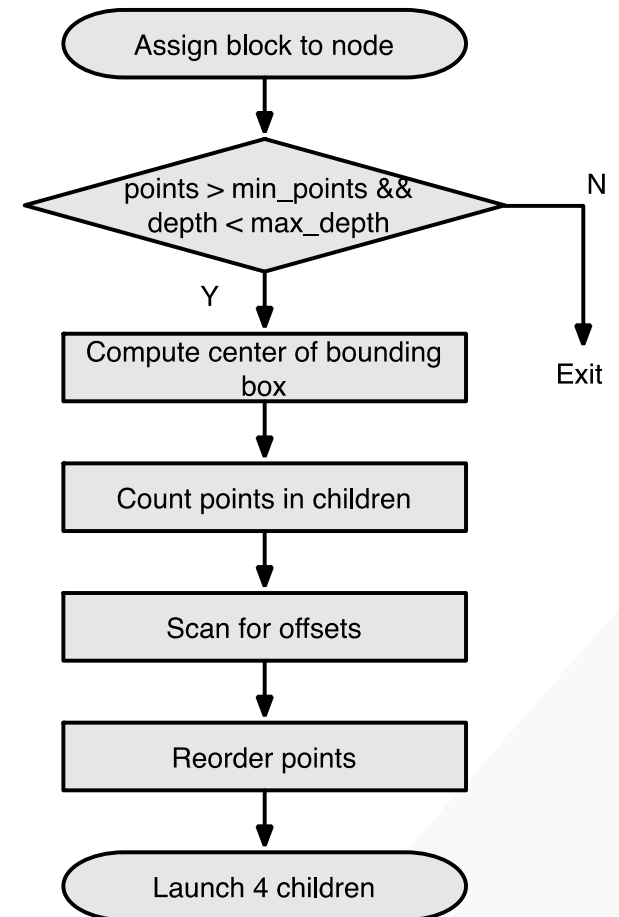
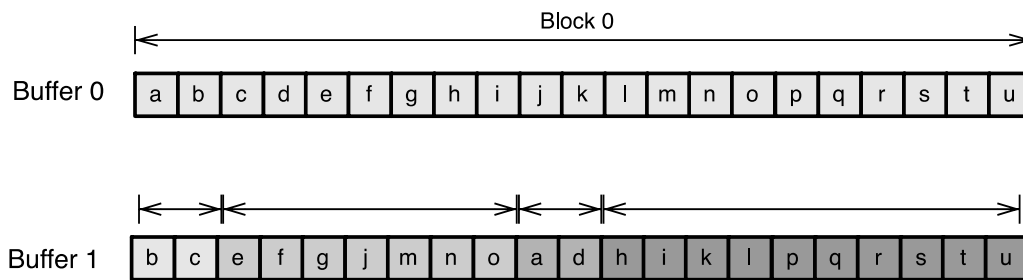
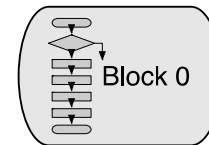


Depth = 3

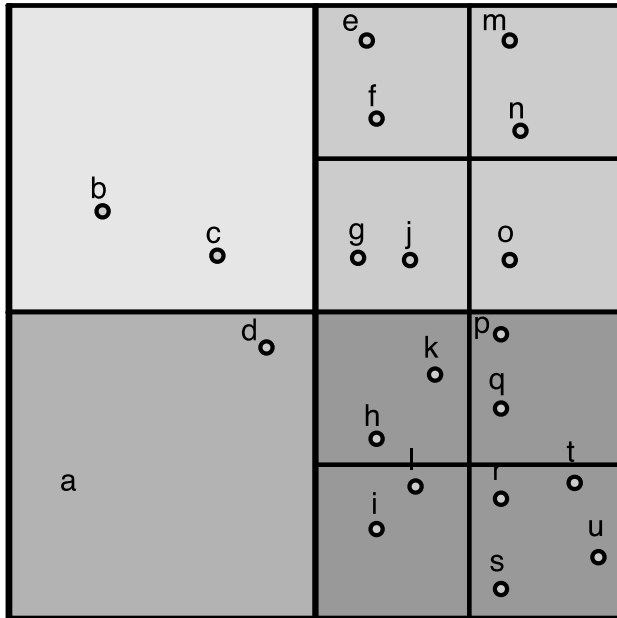
- 1 thread block is launched from host



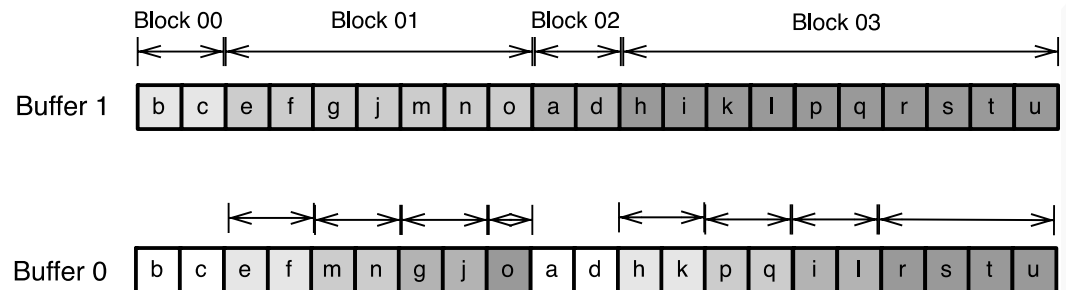
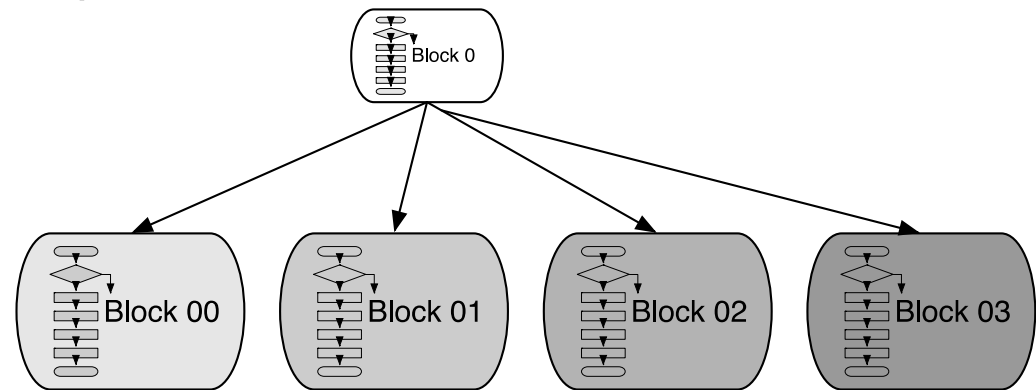
Depth = 0



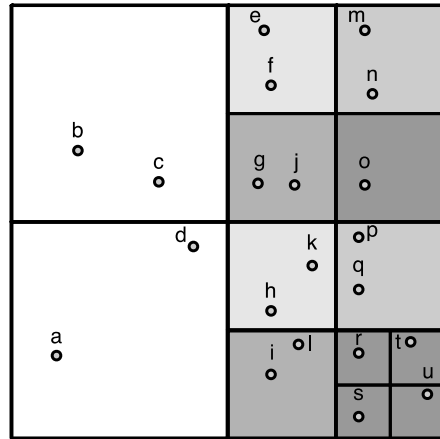
- Each block launches 1 child grid of 4 blocks



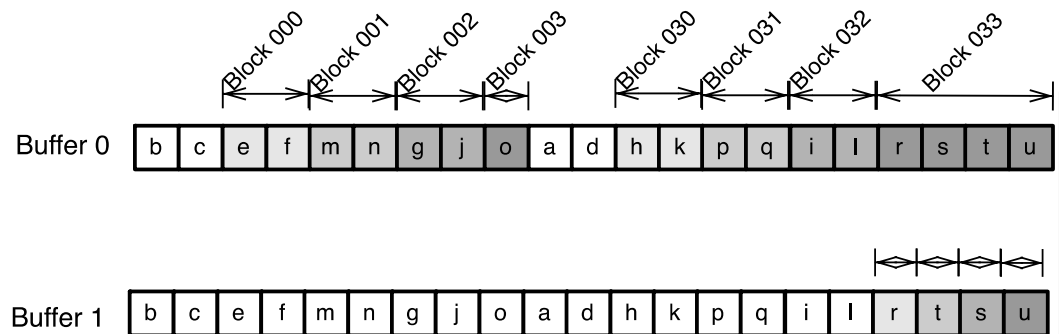
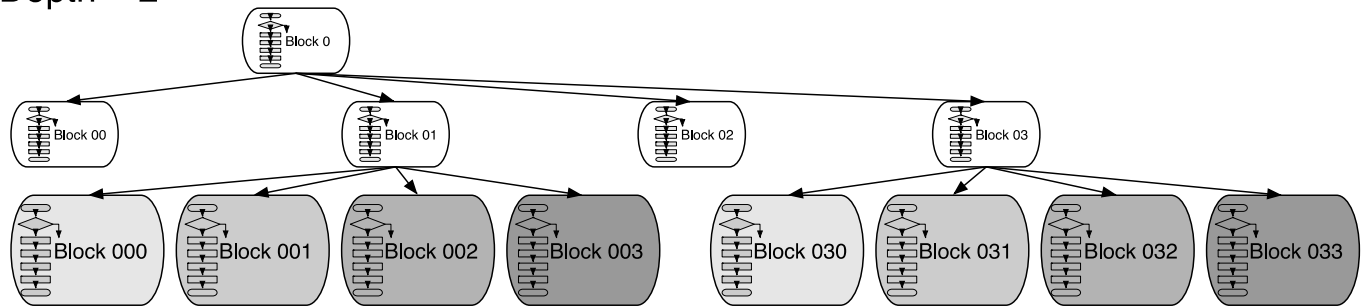
Depth = 1



- Each block launches 1 child grid of 4 blocks



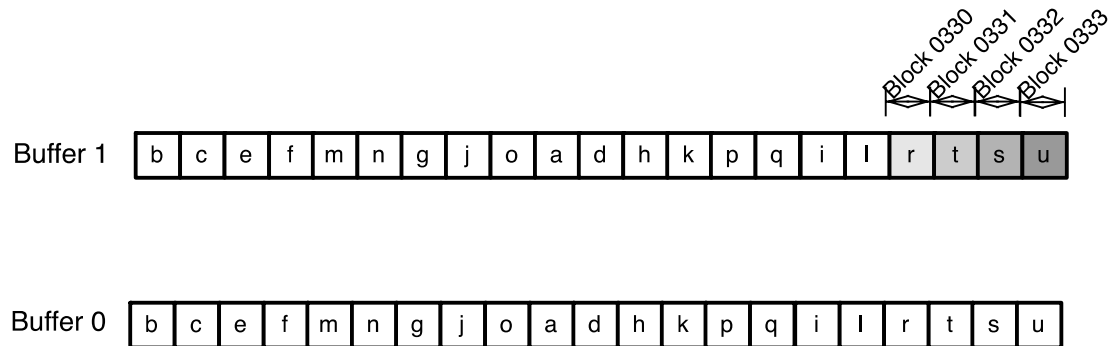
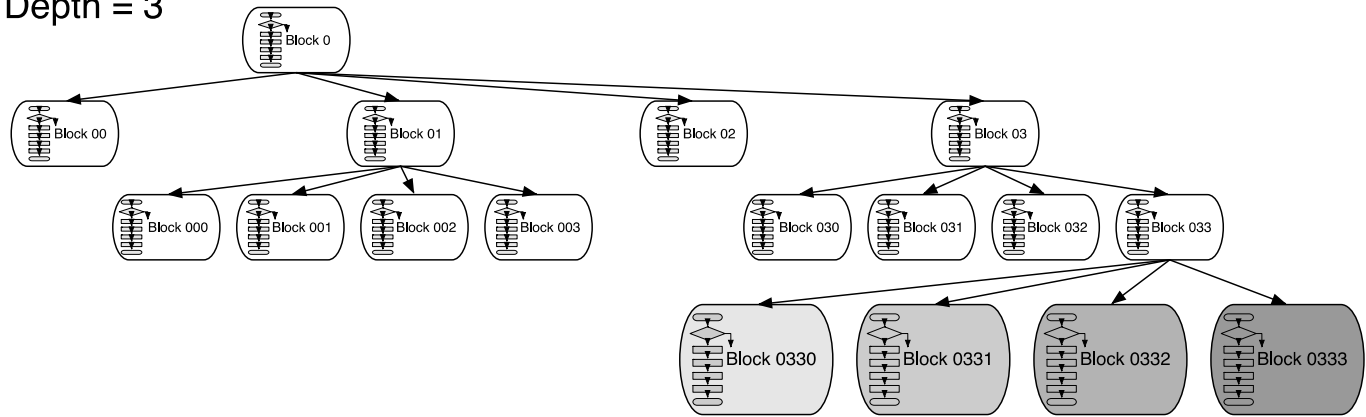
Depth = 2



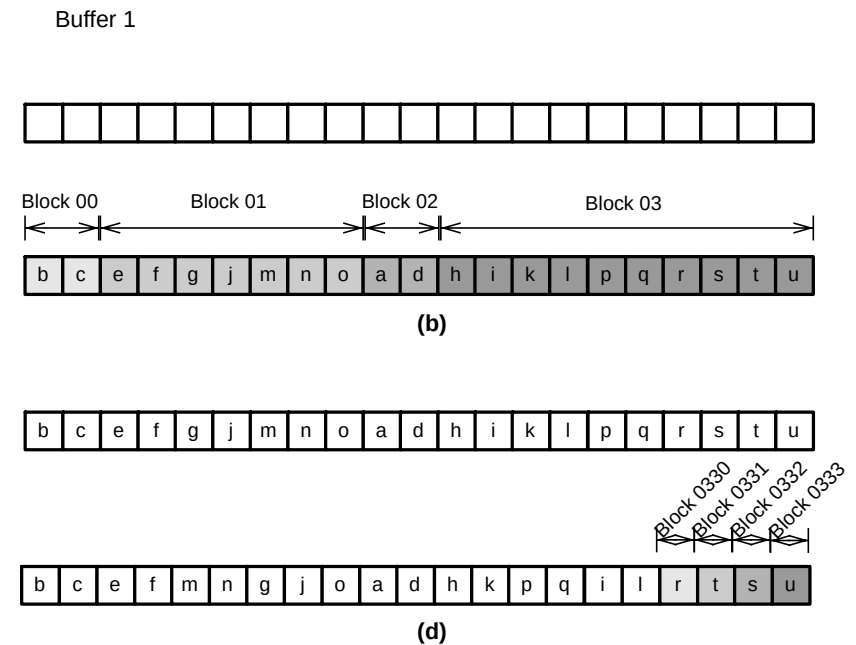
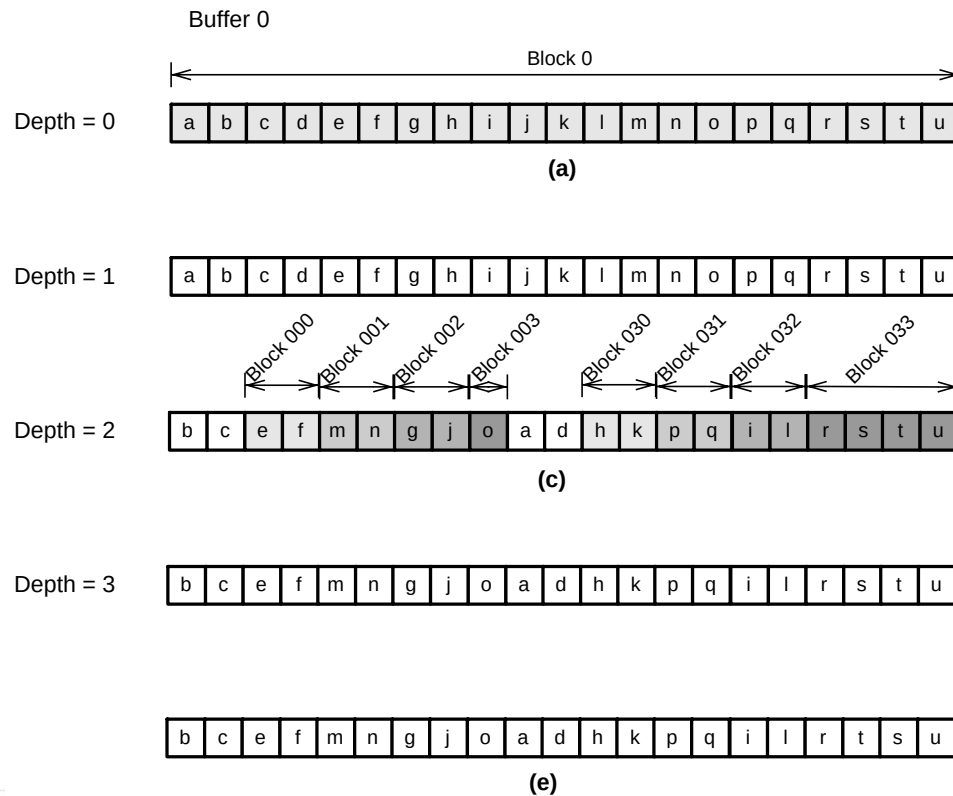
- Each block launches 1 child grid of 4 blocks

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

Depth = 3



- Points in the same quadrant are grouped together




```

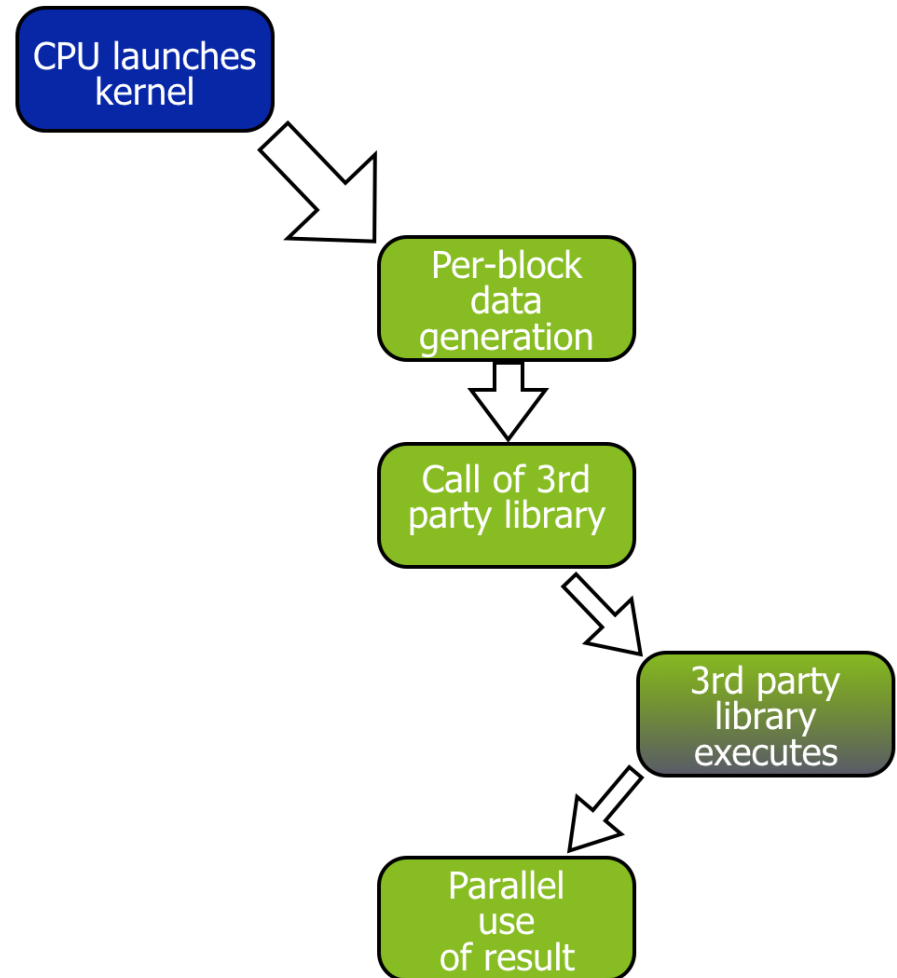
01 __global__ void build_quadtree_kernel
02     (Quadtree_node *nodes, Points *points, Parameters params) {
03     __shared__ int smem[8]; // Shared memory to store the number of points in each quadrant
04
05     // The current node
06     Quadtree_node &node = nodes[blockIdx.x];
07     node.set_id(node.id() + blockIdx.x);
08     int num_points = node.num_points(); // The number of points in the node
09
10     // Check the number of points and its depth
11     bool exit = check_num_points_and_depth(node, points, num_points, params);
12     if(exit) return;
13
14     // Compute the center of the bounding box of the points
15     const Bounding_box &bbox = node.bounding_box();
16     float2 center;
17     bbox.compute_center(center);
18
19     // Range of points
20     int range_begin = node.points_begin();
21     int range_end   = node.points_end();
22     const Points &in_points = points[params.point_selector]; // Input points
23     Points &out_points = points[(params.point_selector+1) % 2]; // Output points
24
25     // Count the number of points in each child
26     count_points_in_children(in_points, smem, range_begin, range_end, center);
27
28     // Scan the quadrants' results to know the reordering offset
29     scan_for_offsets(node.points_begin(), smem);
30
31     // Move points
32     reorder_points(out_points, in_points, smem, range_begin, range_end, center);
33
34     // Launch new blocks
35     if (threadIdx.x == blockDim.x-1) {
36         Quadtree_node *children = &nodes[params.num_nodes_at_this_level]; // The children
37
38         // Prepare children launch
39         prepare_children(children, node, bbox, smem);
40
41         // Launch 4 children.
42         build_quadtree_kernel<<<4, blockDim.x, 8*sizeof(int)>>>
43             (children, points, Parameters(params, true));
44     }
45 }

```

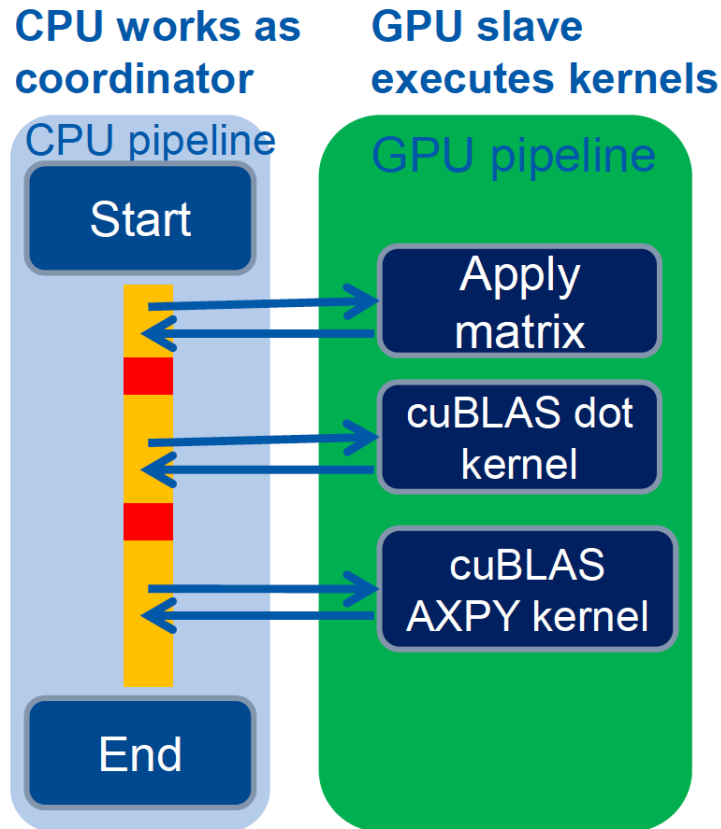
- Introduction to CUDA Dynamic Parallelism
- A simple example
- A more complex example: Bezier lines
- A recursive example: Quadtree
- **Other cases:**
 - Library calls
 - Saving kernel launches in ARM CPUs
- What is and what is not dynamic parallelism
- Summary

- Simple library calls

```
__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);  
}
```



- Lattice QCD



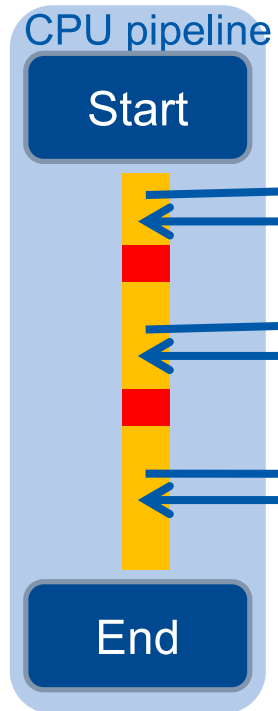
Bottlenecks

- 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

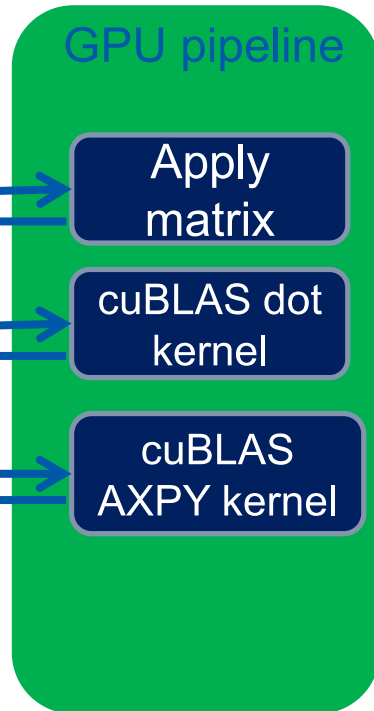
Vishal Mehta, Exploiting CUDA Dynamic Parallelism for low power ARM based prototypes, GTC'2015

- ARM CPUs

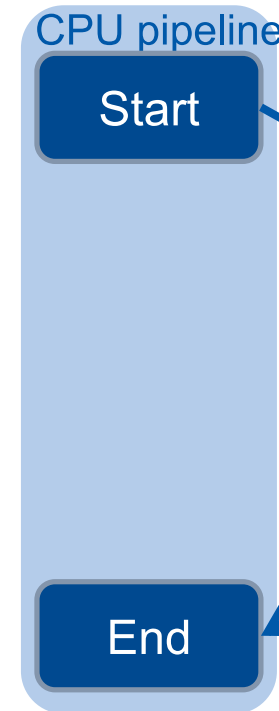
CPU works as coordinator



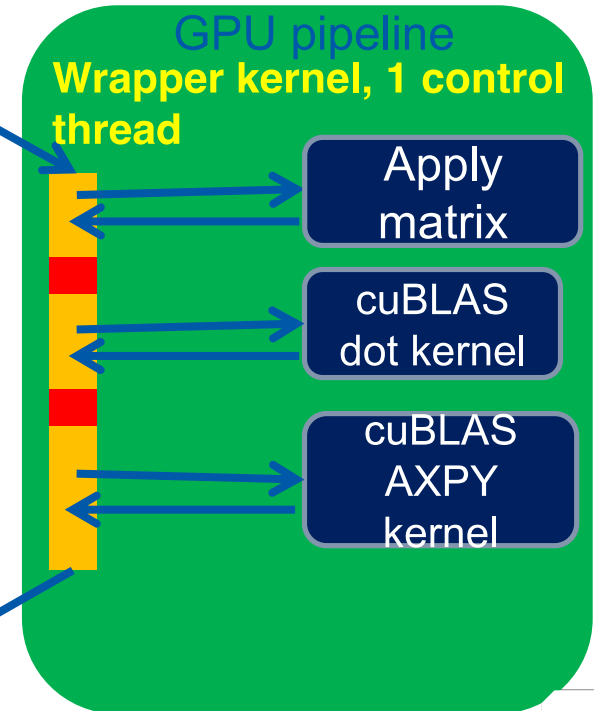
GPU slave executes kernels



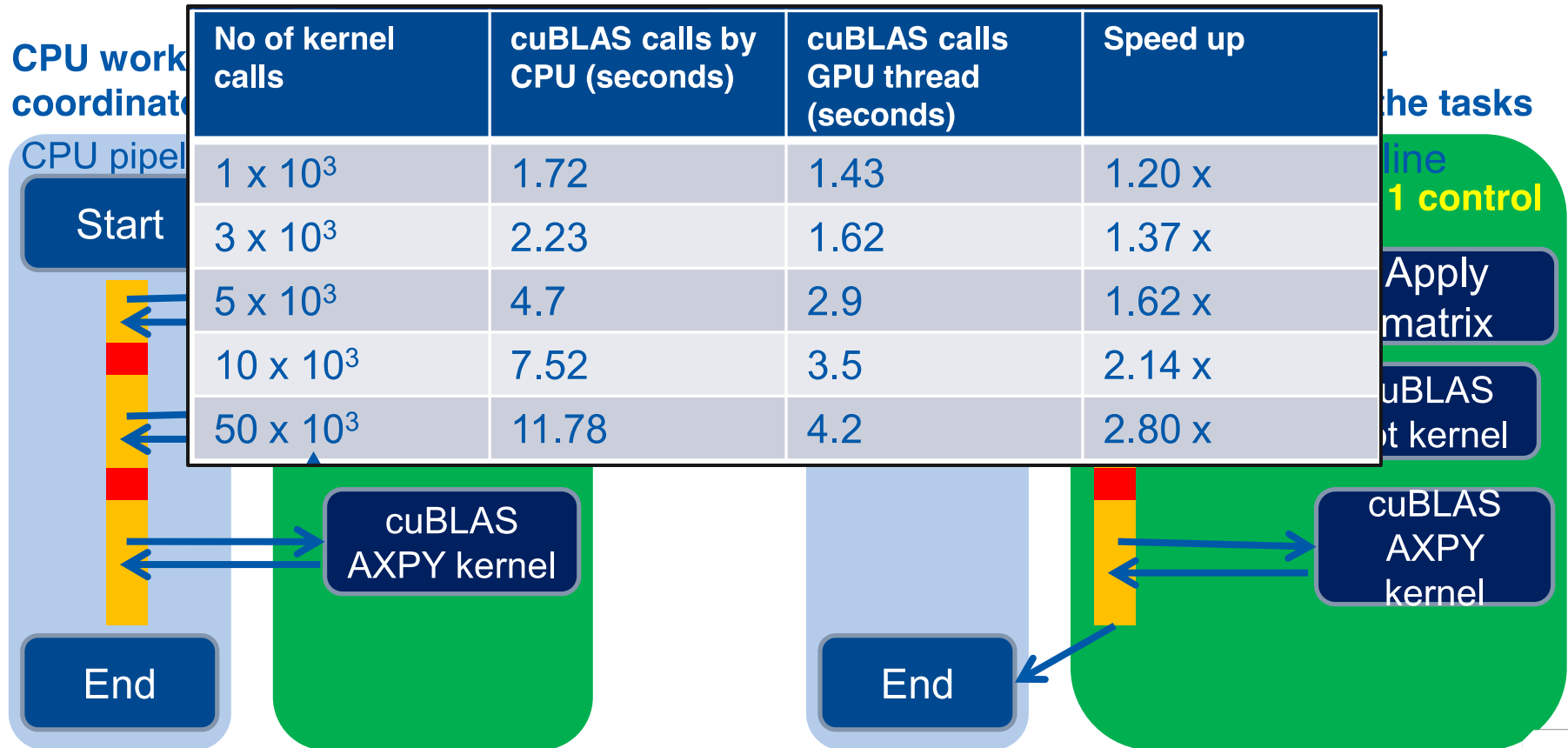
CPU starts and ends wrapper



GPU wrapper coordinates the tasks



- ARM CPUs



- Introduction to CUDA Dynamic Parallelism
- A simple example
- A more complex example: Bezier lines
- A recursive example: Quadtree
- Other cases:
 - Library calls
 - Saving kernel launches in ARM CPUs
- What is and what is not dynamic parallelism
- Summary

- CDP 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.

- 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
- What is and what is not CDP

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.