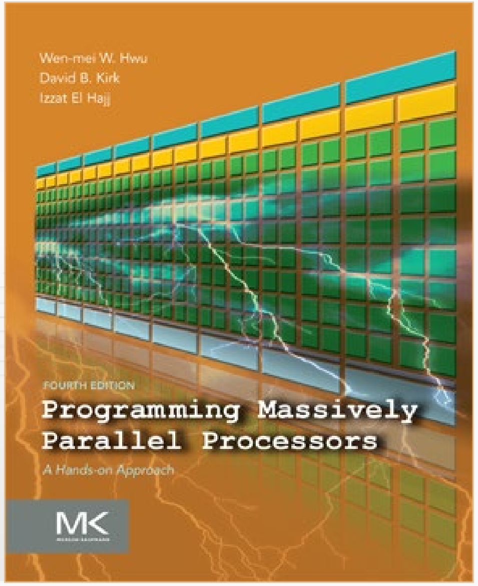
Programming Massively Parallel Processors

A Hands-on Approach

CHAPTER 21 › CUDA Dynamic Parallelism



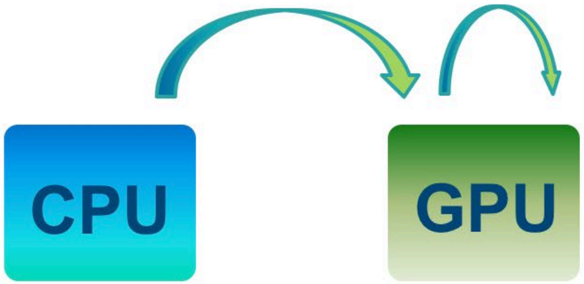
Outline

- 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

CUDA Dynamic Parallelism

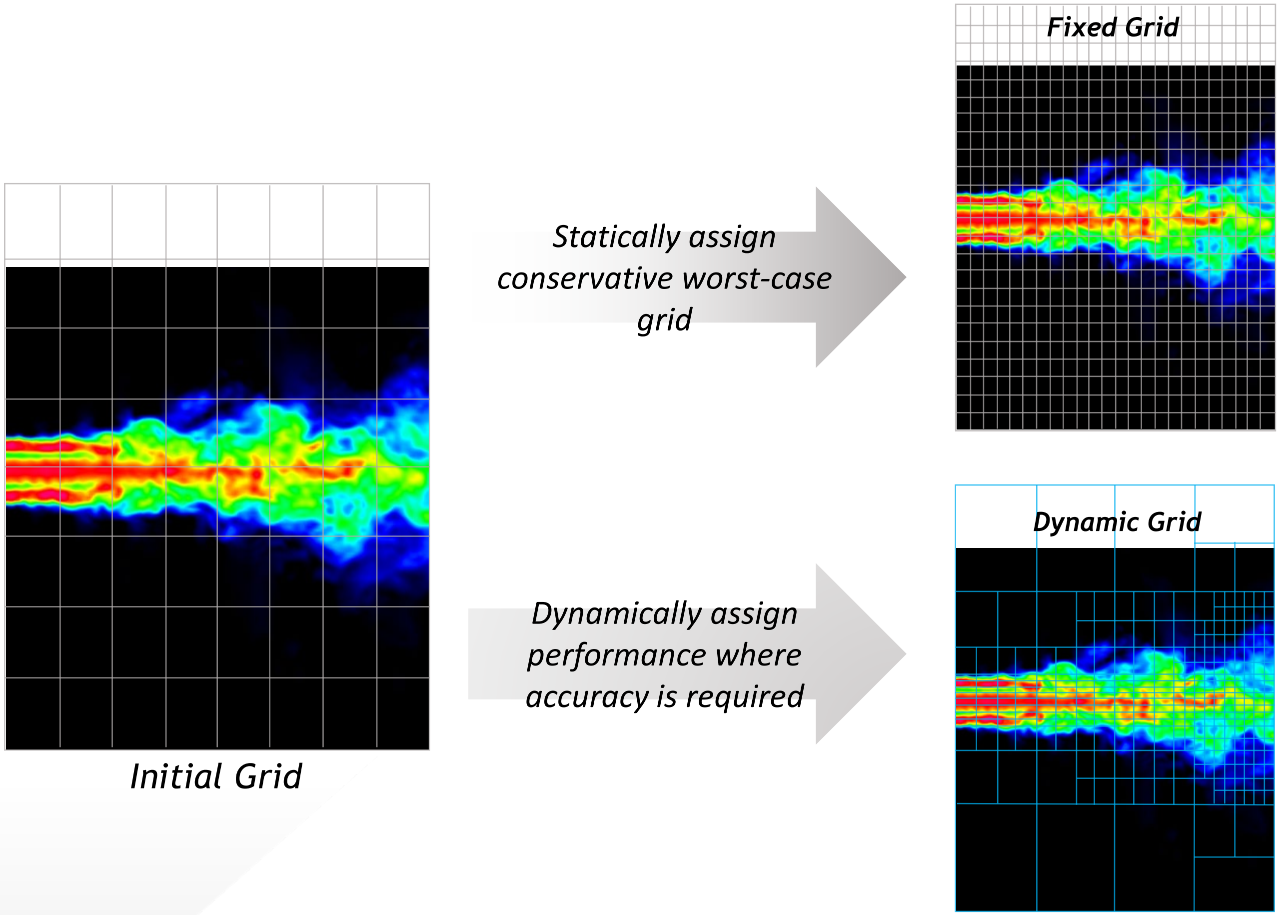
- Device-side kernel launches
  - Kepler GK110 architecture
  - Typical use cases
    - Dynamic load balancing
    - Data-dependent execution
    - Recursion
    - Library (with kernels) calls from kernels
  - Programmability and maintainability

Fermi: Only CPU can generate GPU work.

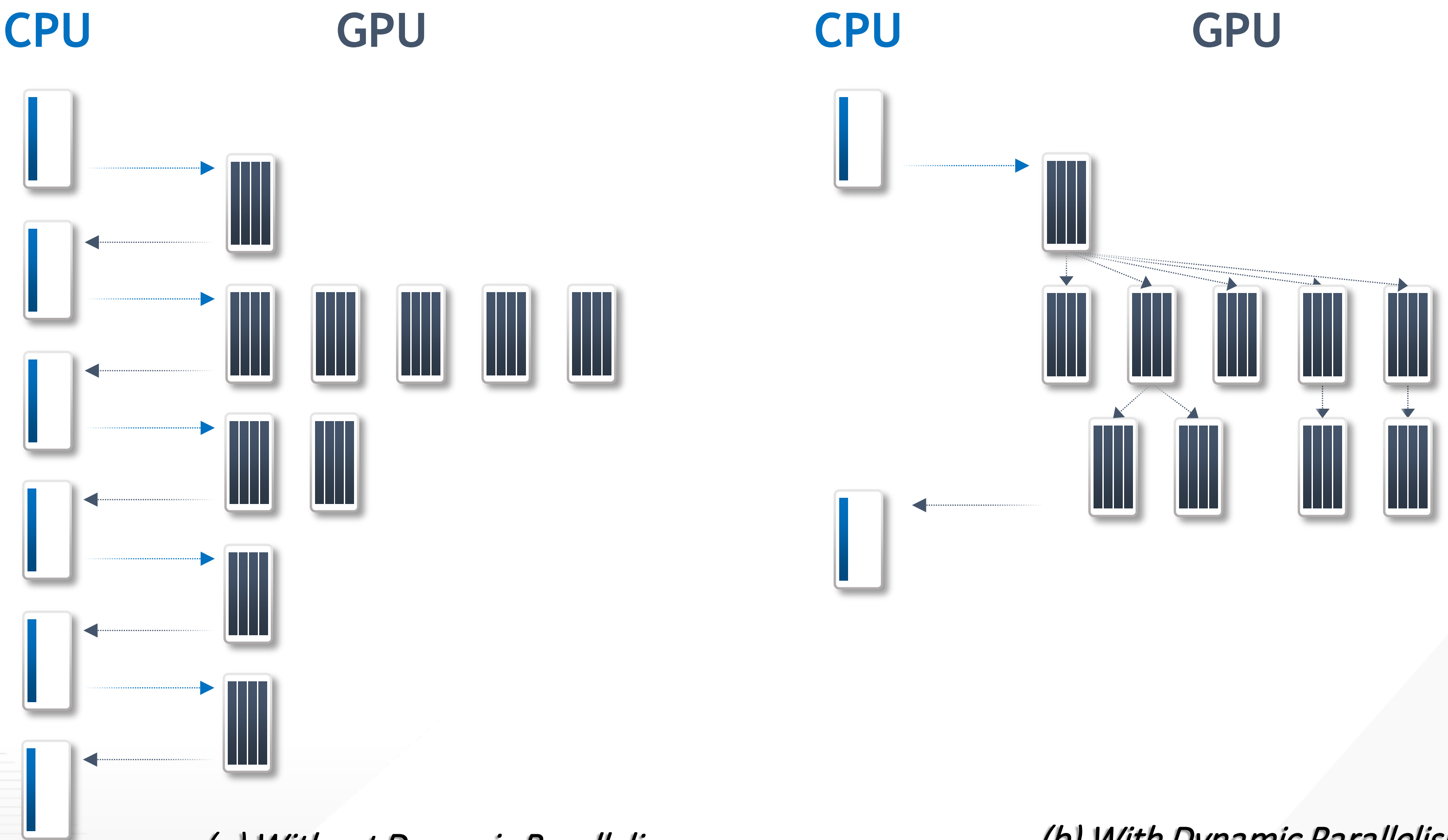


Kepler: GPU can generate work for itself.

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



- CPU-GPU without and with dynamic parallelism



(a) Without Dynamic Parallelism

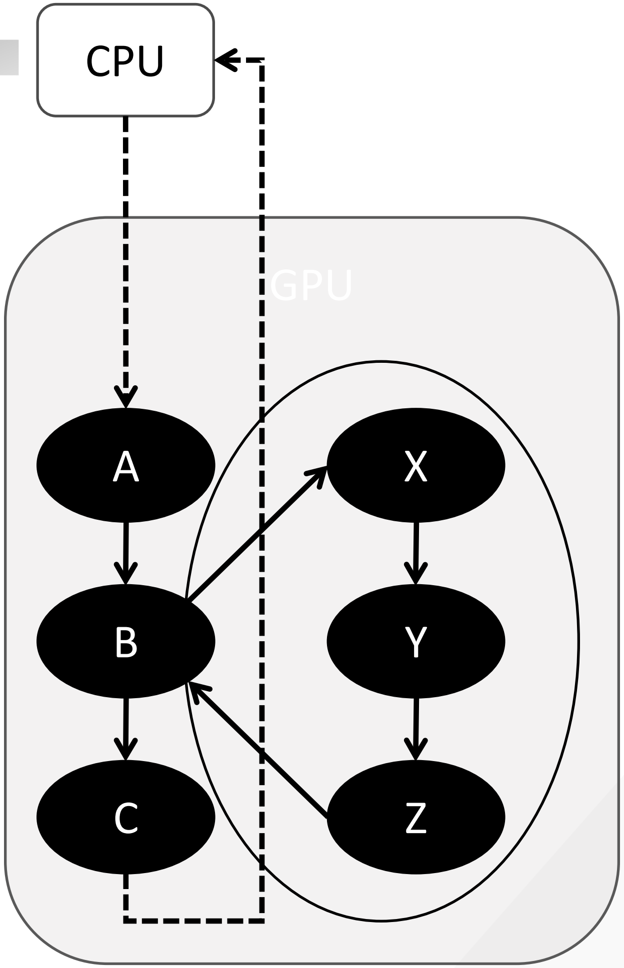
(b) With Dynamic Parallelism

CUDA 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);
}
```



CUDA Dynamic Parallelism

- Syntax

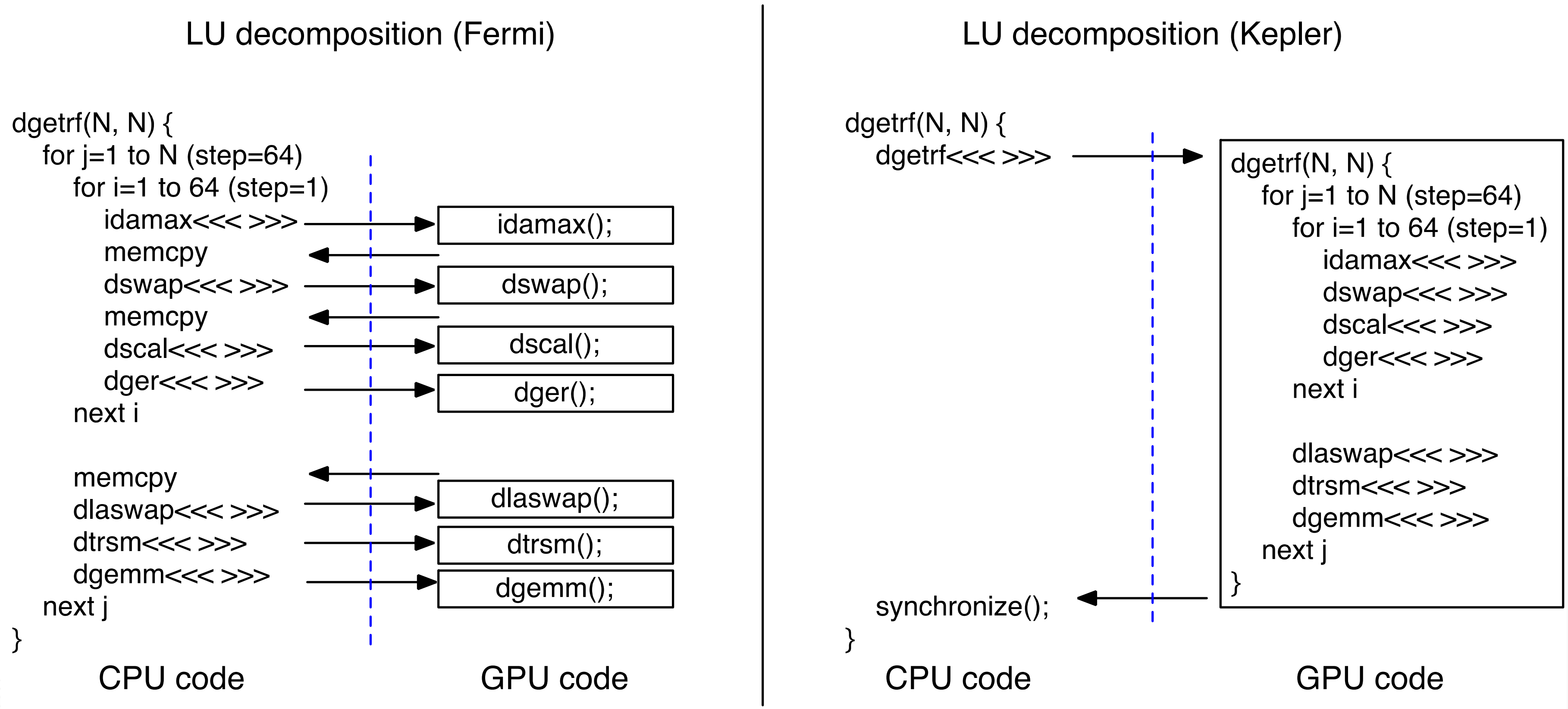
```
kernel_name<<< Dg, Db, Ns, S >>>([kernel arguments]);
```

  - 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.

CUDA Dynamic Parallelism

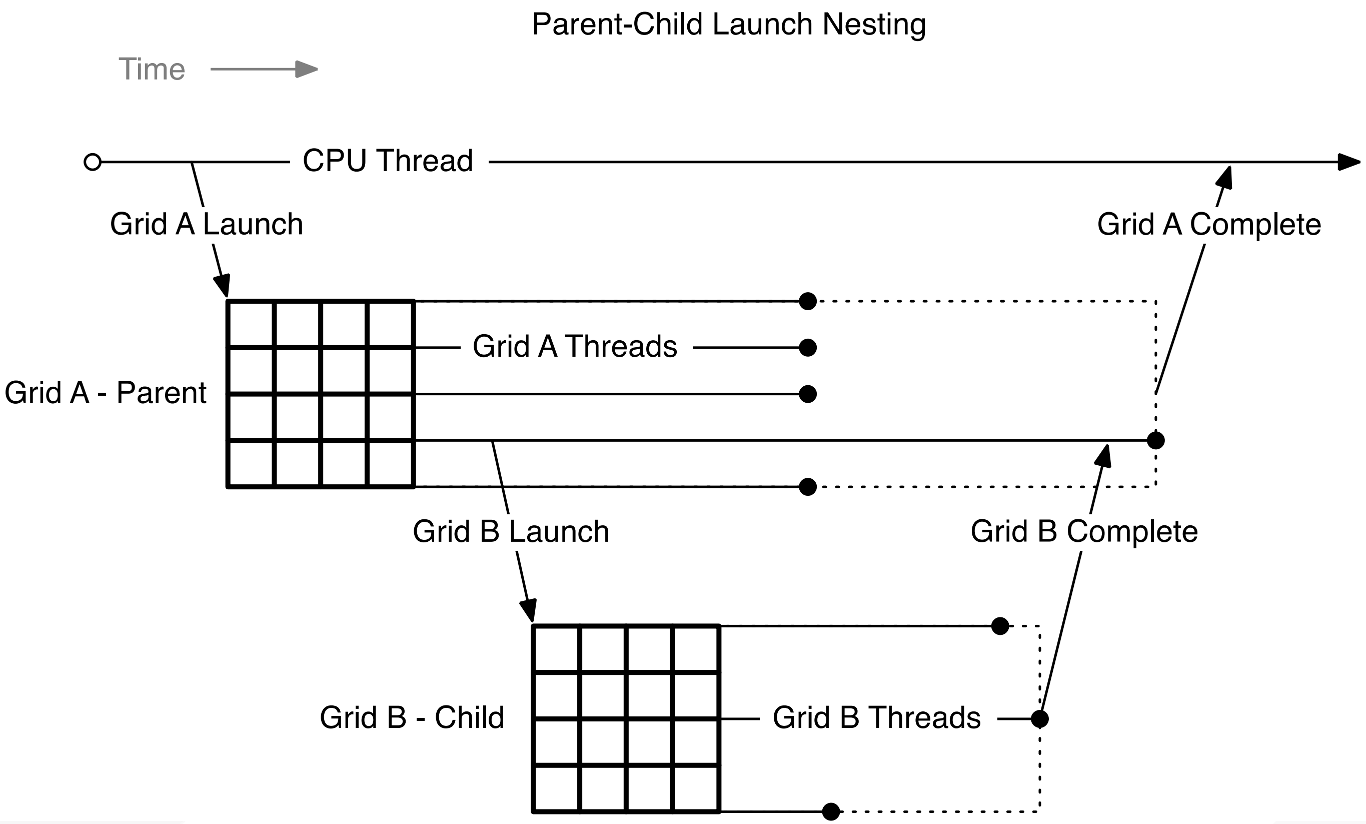
- LU decomposition

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$



Parent-child synchronization

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



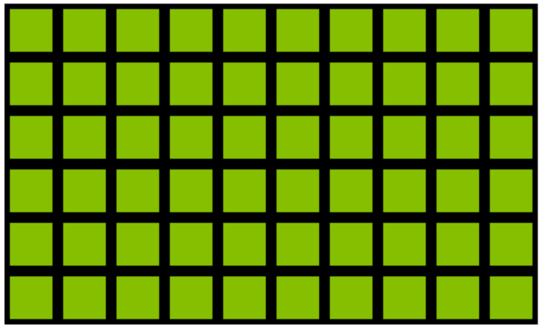
Outline

- 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

A simple example

- Without dynamic parallelism, uniform workload

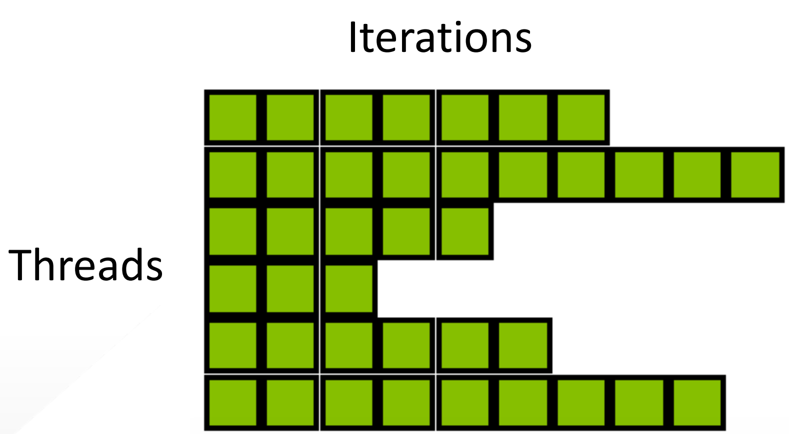
```
__global__ void kernel(unsigned int start, unsigned int end,
    float* someData, float* moreData) {

    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

- Without dynamic parallelism, non-uniform workload

```
__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]);
    }

}
```

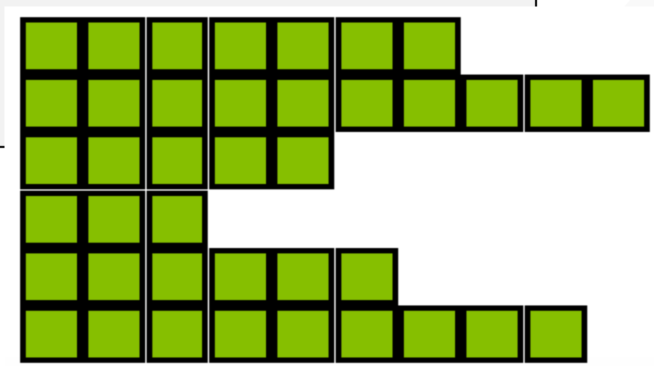


A simple example

- With dynamic parallelism, non-uniform workload

```
__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

Outline

- 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

A more complex example: Bezier lines

- Linear Bezier curves

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

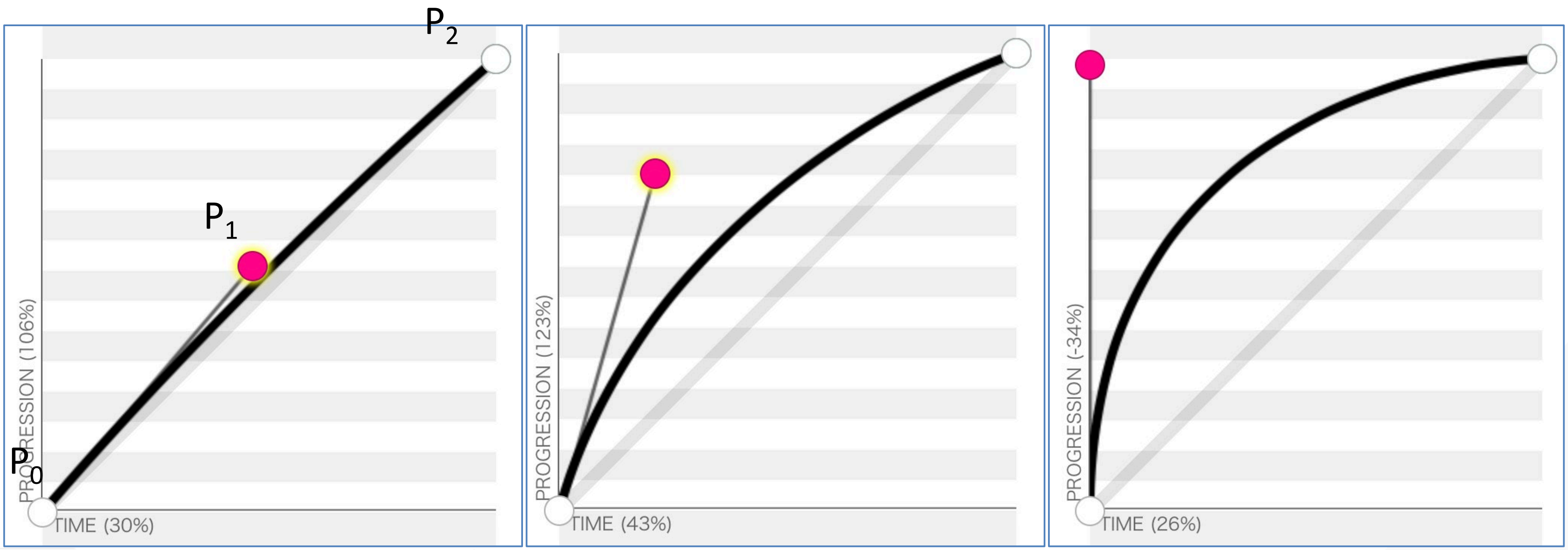
- Quadratic Bezier curves

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

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

A more complex example: Bezier lines

- Control points
  - Curvature calculation
  - Tessellation points



A more complex example: Bezier lines

- **Without dynamic parallelism**: One line per block

```
__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*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[bidx].CP[i];
         }
         //Assign value of vertex position to the correct array element
         bLines[bidx].vertexPos[idx] = position;
       }
     }
}
```

A more complex example: Bezier lines

- With dynamic parallelism
- Parent: One line per thread

```
__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);
  }
}
```

A more complex example: Bezier lines

- With dynamic parallelism
- Child

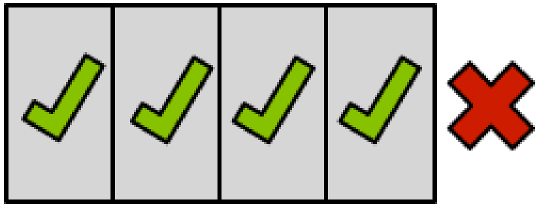
```
__global__ void computeBezierLine_child(int lidx, BezierLine* bLines,
  int nTessPoints) {
  int idx = threadIdx.x + blockDim.x*blockIdx.x;  //Compute idx unique to this vertex
  if(idx < nTessPoints){
    float u = (float)idx/(float)(nTessPoints-1);  //Compute u from idx
    float omu = 1.0f - u;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

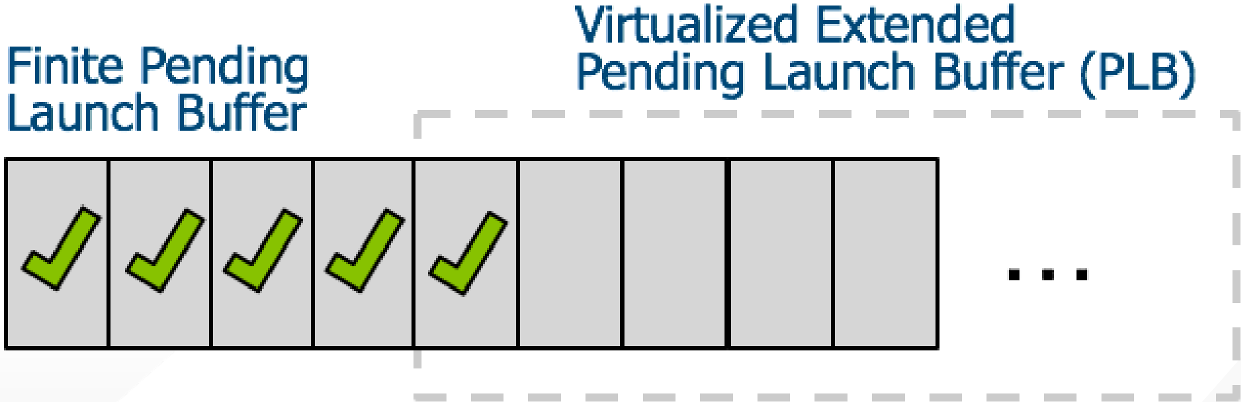
A more complex example: Bezier lines

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

Before CUDA 6.0

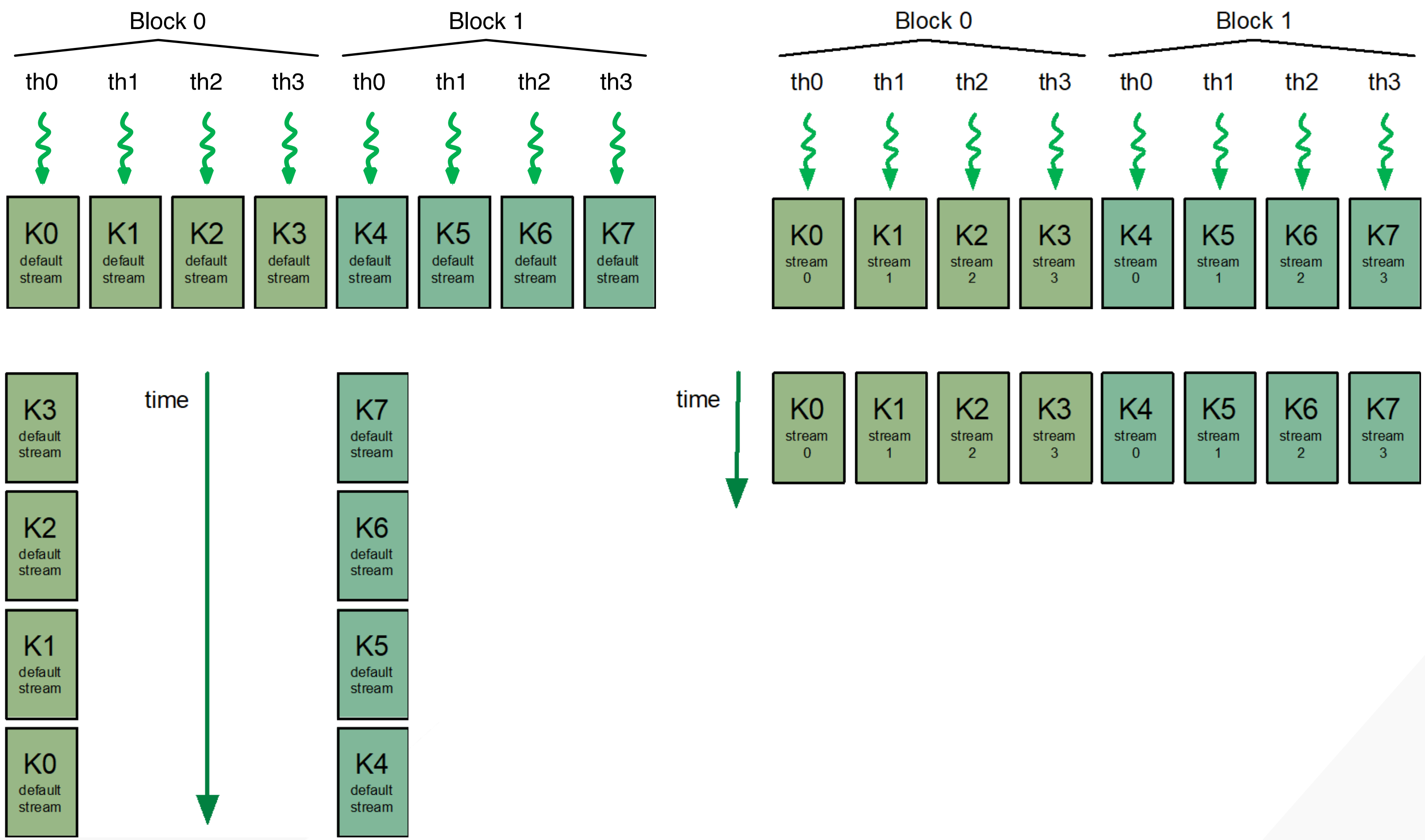
Since CUDA 6.0





A more complex example: Bezier lines

- Streams



A more complex example: Bezier lines

- 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);
```

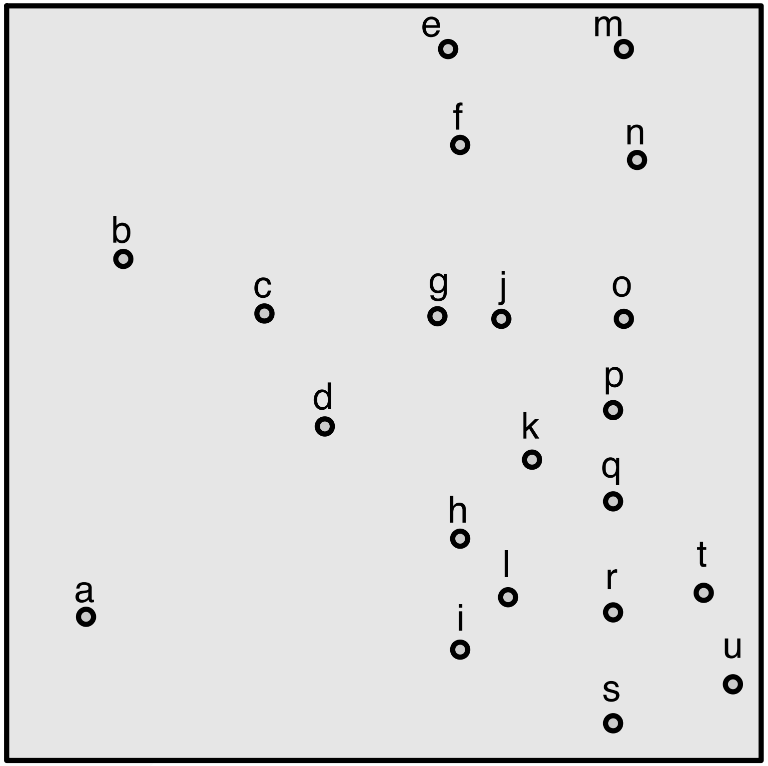
```
cudaStream_t stream;
// Create non-blocking stream
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);

//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);
```

Outline

- 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

A recursive example: Quadtree

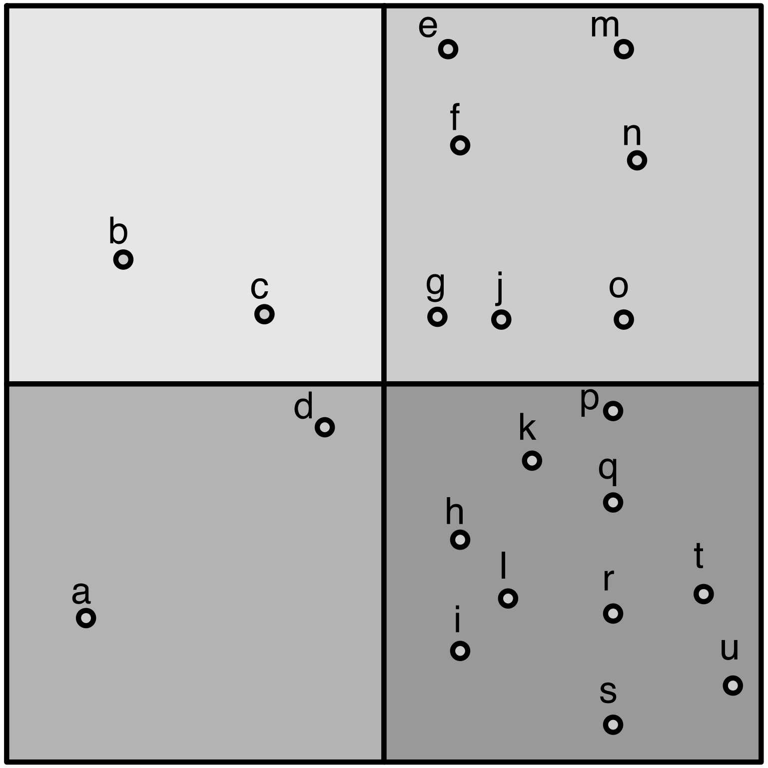
- Partitioning a 2D space by recursively dividing it into four quadrants



Depth = 0

A recursive example: Quadtree

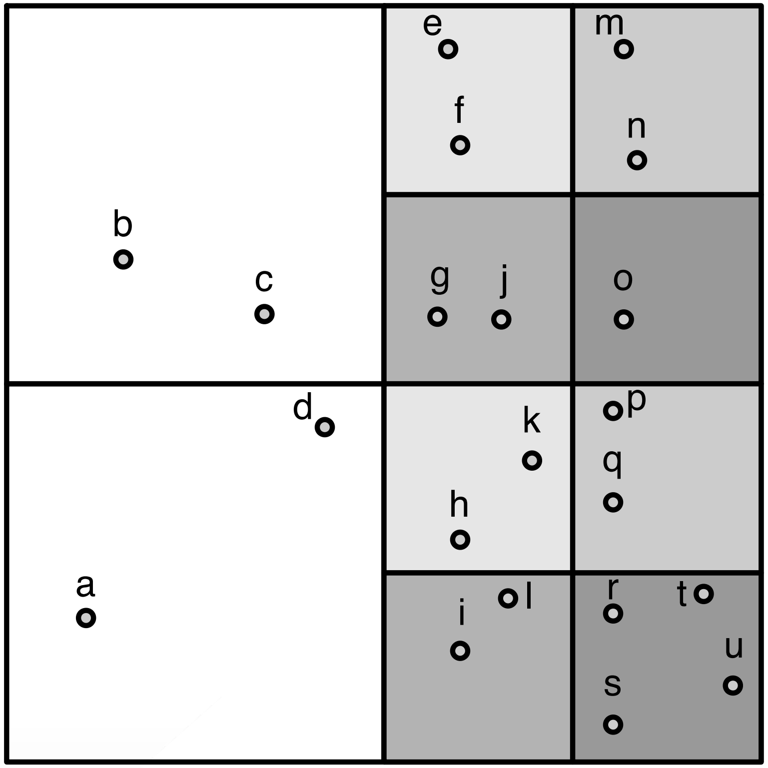
- Partitioning a 2D space by recursively dividing it into four quadrants



Depth = 1

A recursive example: Quadtree

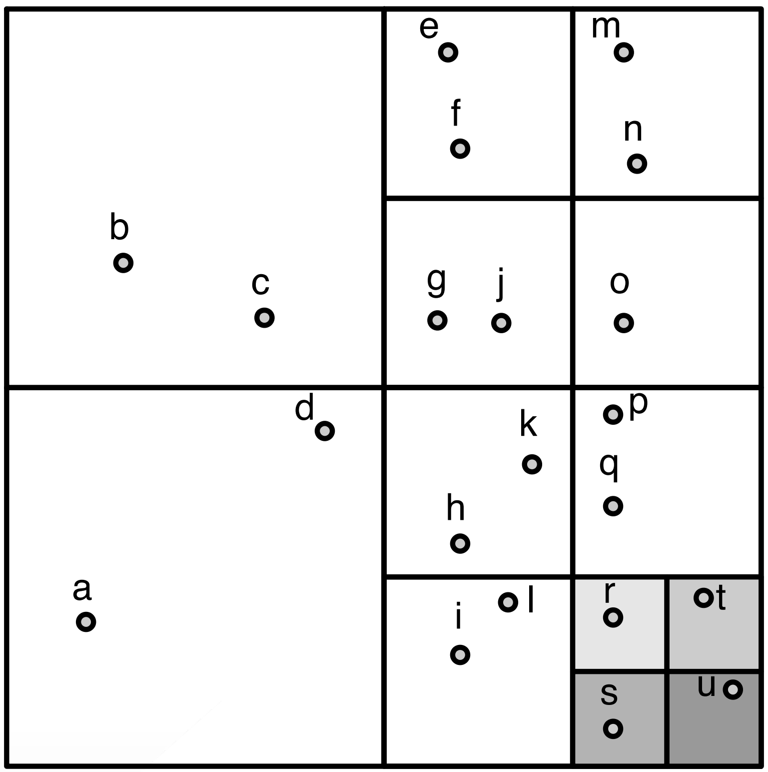
- Partitioning a 2D space by recursively dividing it into four quadrants



Depth = 2

A recursive example: Quadtree

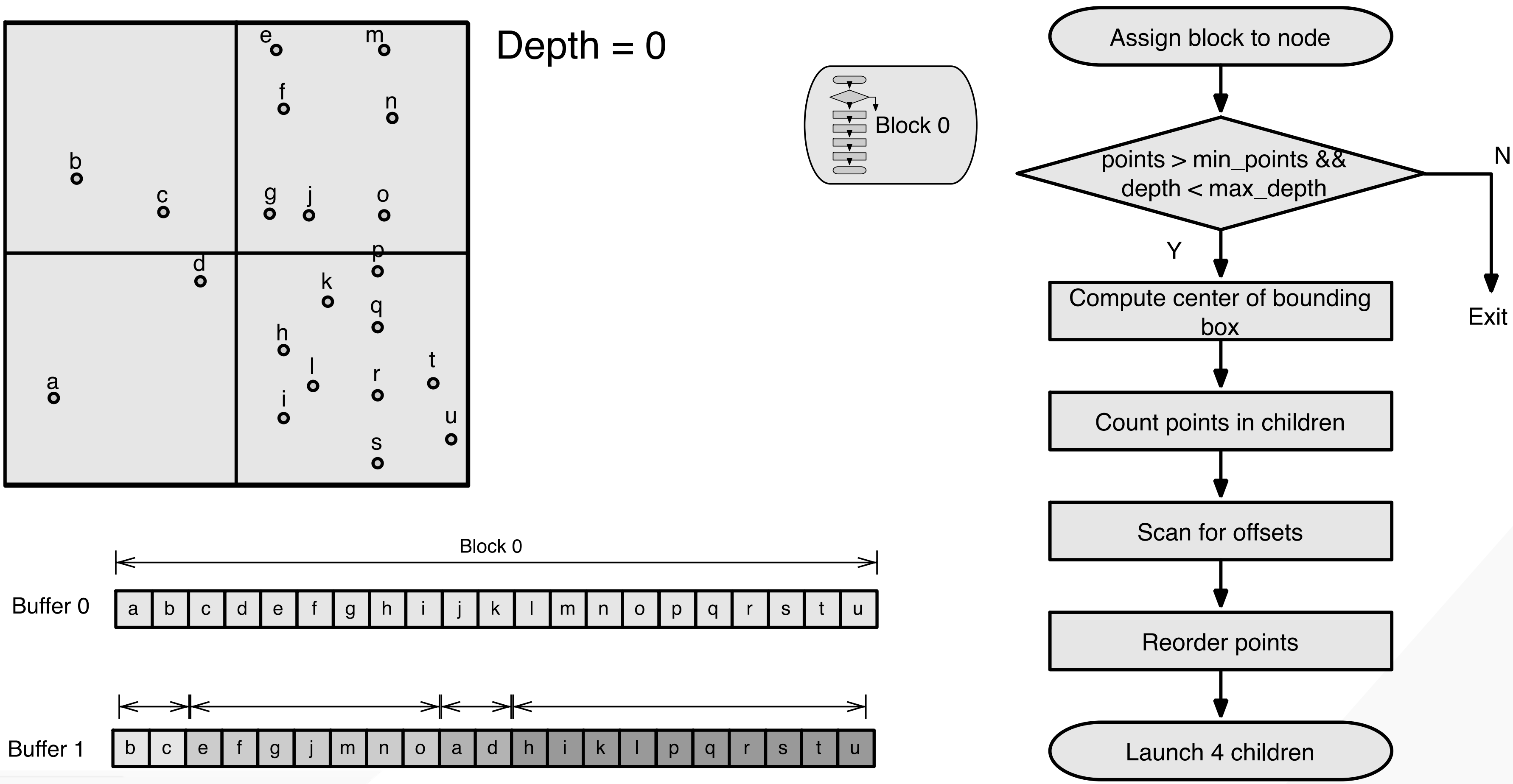
- Partitioning a 2D space by recursively dividing it into four quadrants



Depth = 3

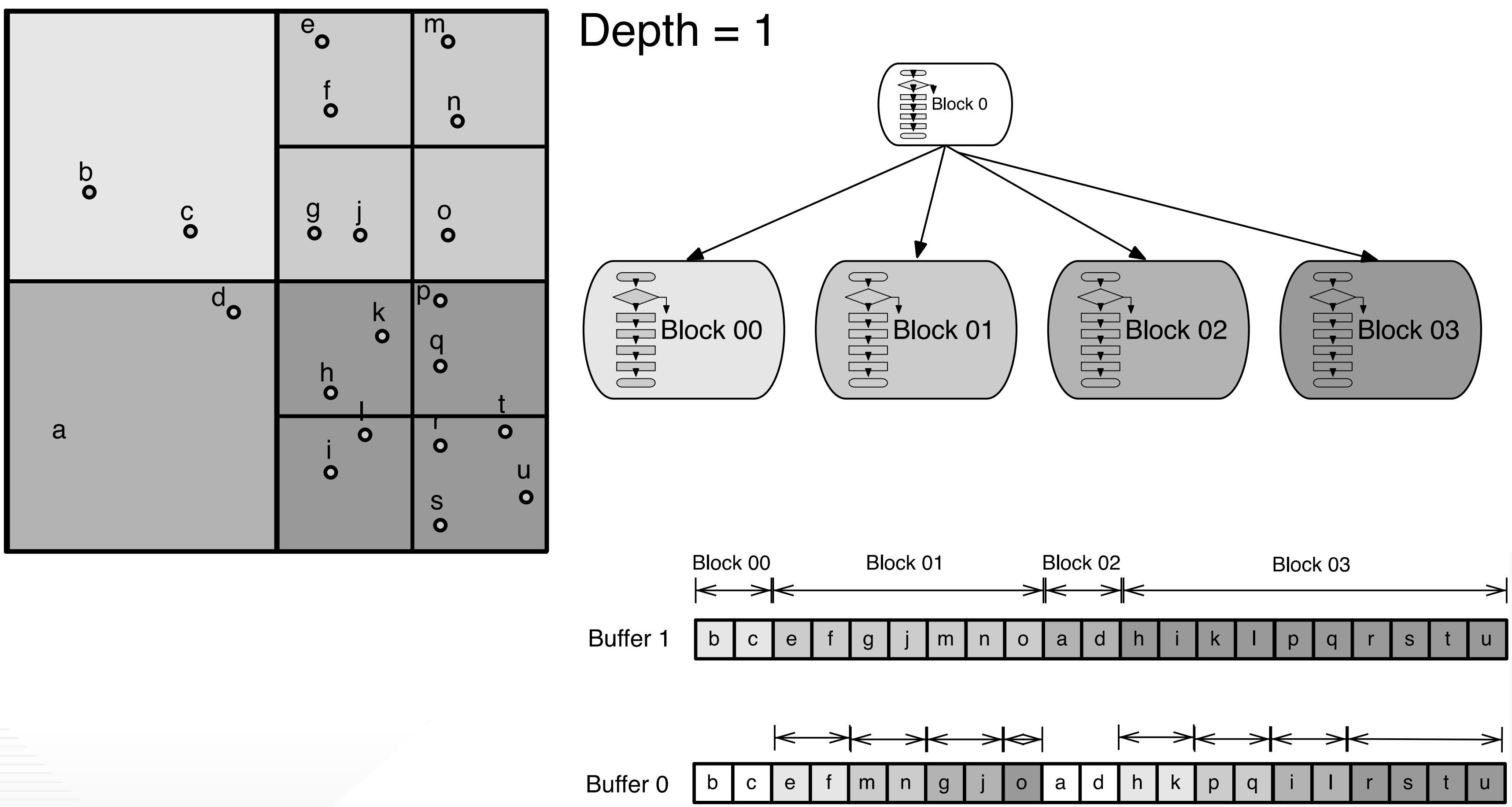
A recursive example: Quadtree

- 1 thread block is launched from host



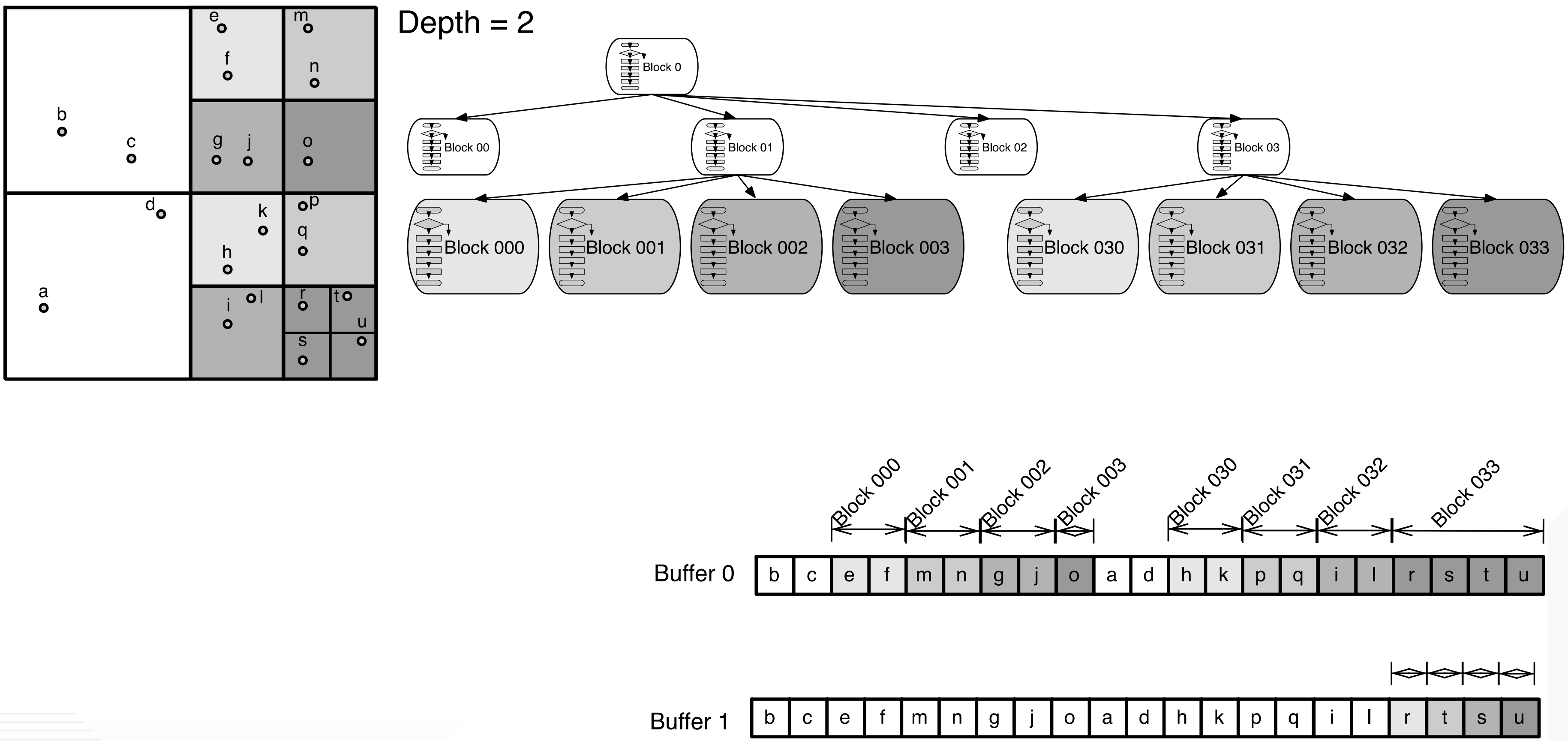
A recursive example: Quadtree

- Each block launches 1 child grid of 4 blocks



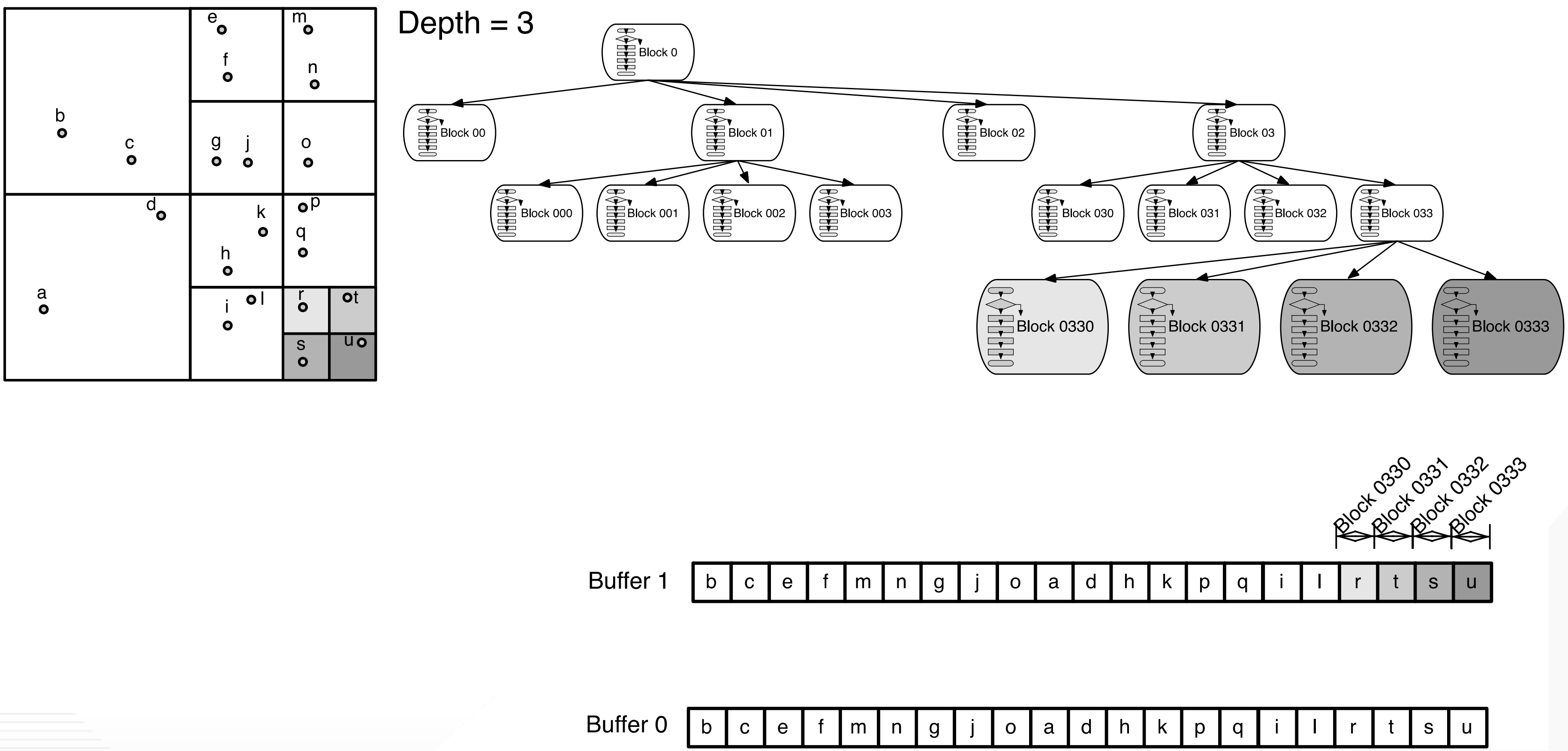
A recursive example: Quadtree

- Each block launches 1 child grid of 4 blocks



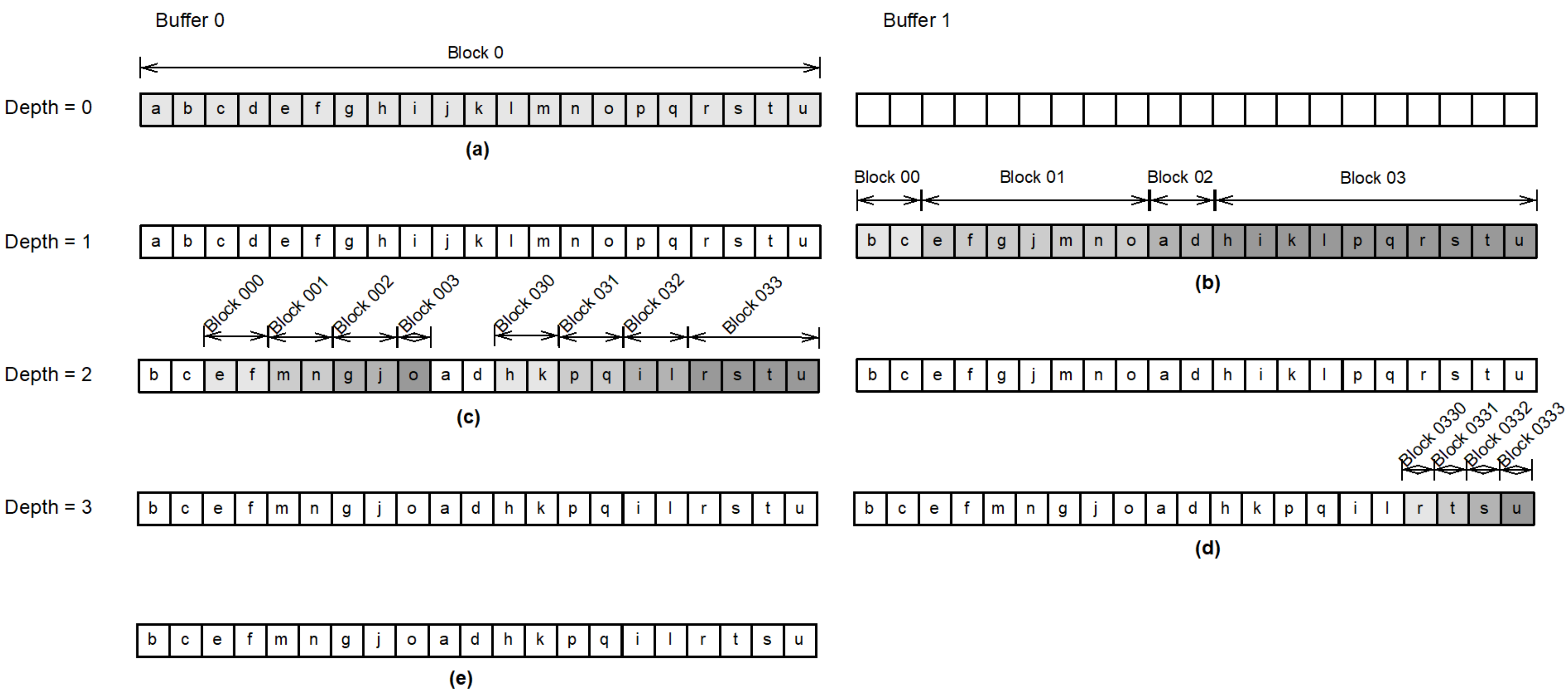
A recursive example: Quadtree

- Each block launches 1 child grid of 4 blocks



A recursive example: Quadtree

- Points in the same quadrant are grouped together



A recursive example: Quadtree

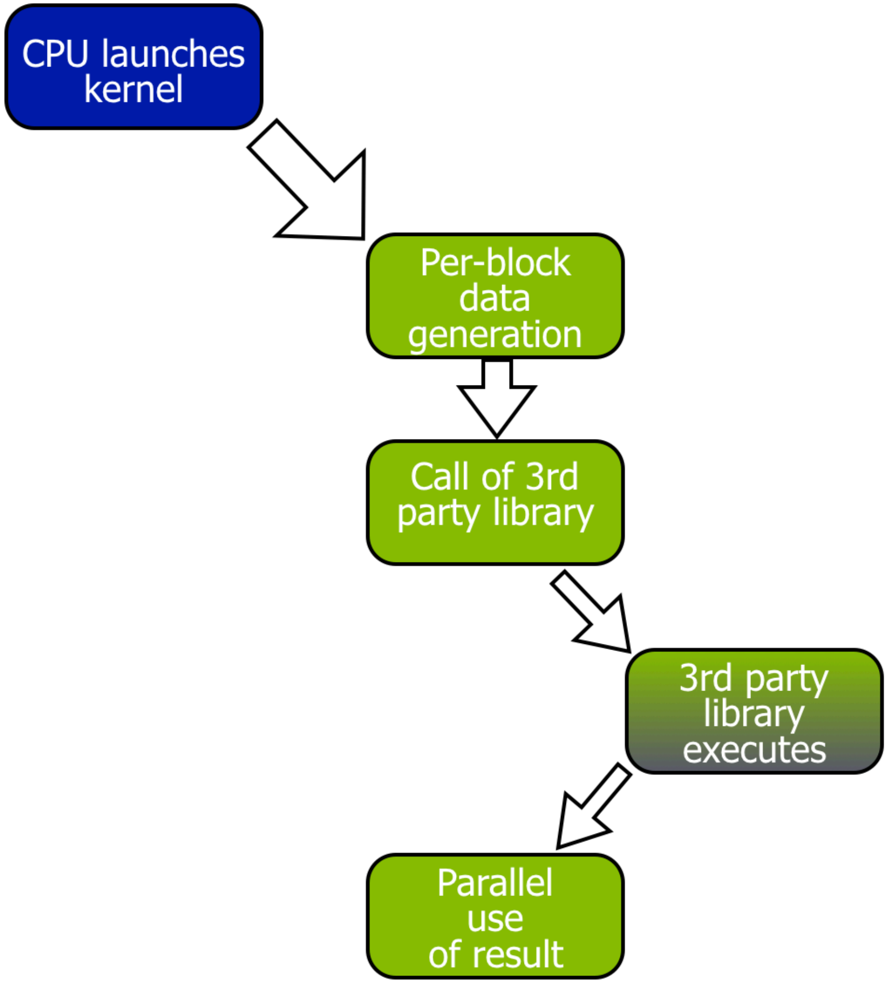
```
__global__ void build_quadtree_kernel
                (Quadtree_node *nodes, Points *points, Parameters params) {
    __shared__ int smem[8]; // Shared memory to store the number of points in each quadrant

    // The current node
    Quadtree_node &node = nodes[blockIdx.x];
    node.set_id(node.id() + blockIdx.x);
    int num_points = node.num_points(); // The number of points in the node

    // Check the number of points and its depth
    bool exit = check_num_points_and_depth(node, points, num_points, params);
    if(exit) return;

    // Compute the center of the bounding box of the points
    const Bounding_box &bbox = node.bounding_box();
    float2 center;
    bbox.compute_center(center);

    // Range of points
    int range_begin = node.points_begin();
    int range_end   = node.points_end();
    const Points &in_points = points[params.point_selector]; // Input points
    Points &out_points = points[(params.point_selector+1) % 2]; // Output points

    // Count the number of points in each child
    count_points_in_children(in_points, smem, range_begin, range_end, center);

    // Scan the quadrants' results to know the reordering offset
    scan_for_offsets(node.points_begin(), smem);

    // Move points
    reorder_points(out_points, in_points, smem, range_begin, range_end, center);

    // Launch new blocks
    if (threadIdx.x == blockDim.x-1) {
      Quadtree_node *children = &nodes[params.num_nodes_at_this_level]; // The children

      // Prepare children launch
      prepare_children(children, node, bbox, smem);

      // Launch 4 children.
      build_quadtree_kernel<<<4, blockDim.x, 8*sizeof(int)>>>
                             (children, points, Parameters(params, true));
    }
  }
```

Outline

- 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

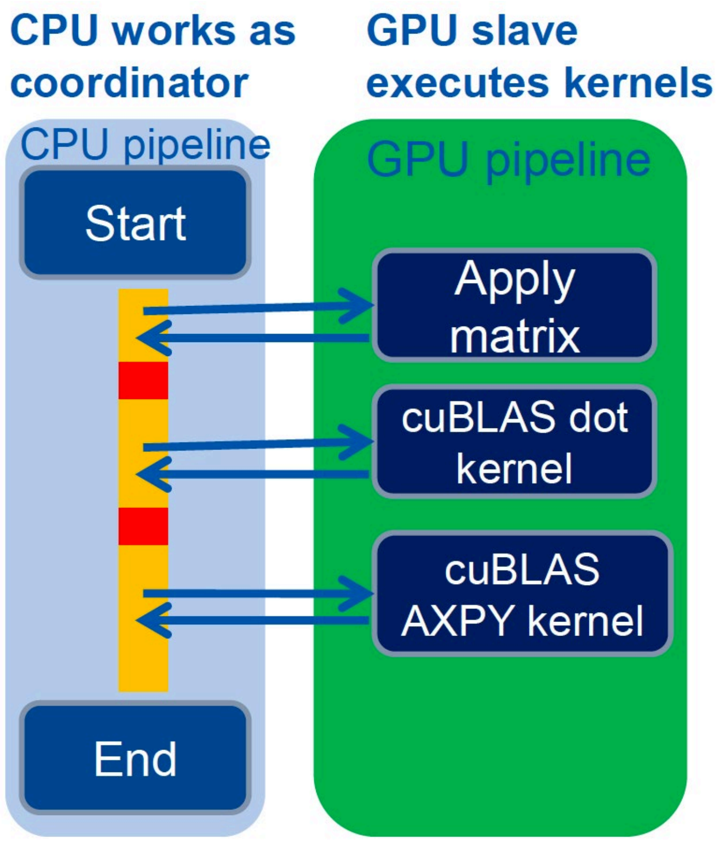
- Simple library calls

```
__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);
}
```



Avoid launch overhead

- Lattice QCD



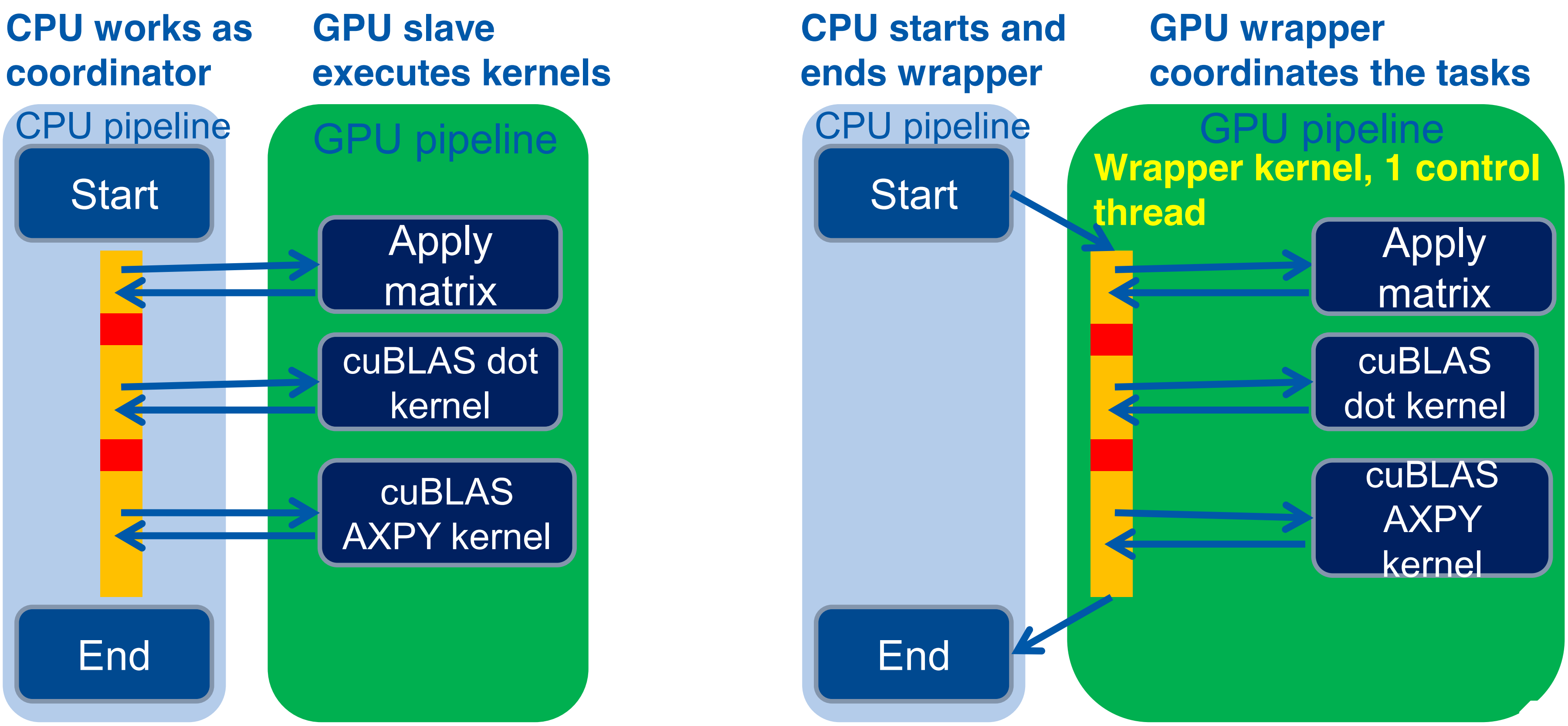
<u>Bottlenecks</u>

- 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

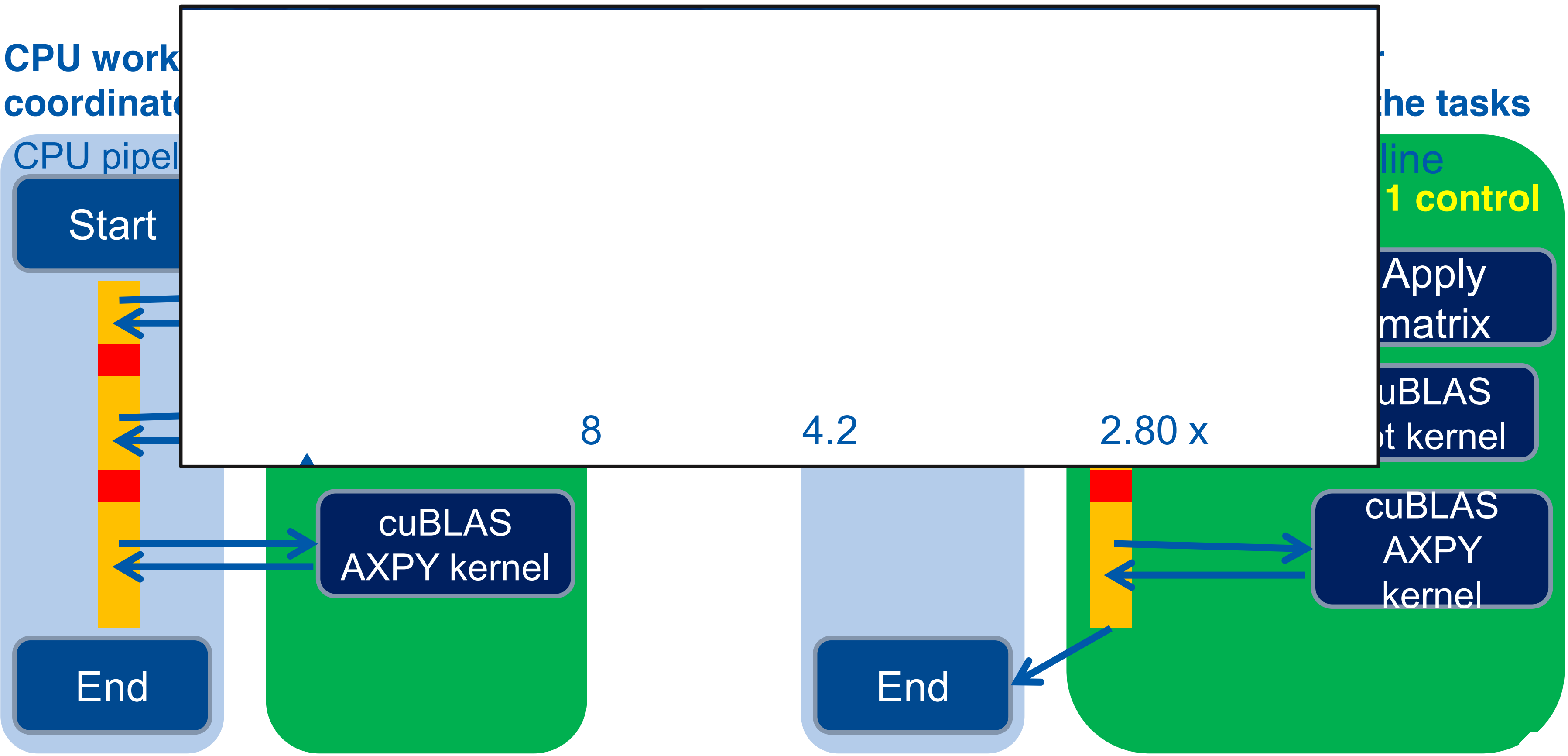
Vishal Mehta, Exploiting CUDA Dynamic Parallelism for low power ARM based prototypes, GTC'2015

Avoid launch overhead

- ARM CPUs



- ARM CPUs


CPU work
coordinate
the tasks
CPU pipel
line
1 control
Start
Apply
matrix
uBLAS
t kernel
8
4.2
2.80 x
cuBLAS
AXPY kernel
cuBLAS
AXPY
kernel
End
End

Outline

- 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

What is and what is not

- 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.

Summary

- 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

References

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