CSc 8530
Parallel Algorithms

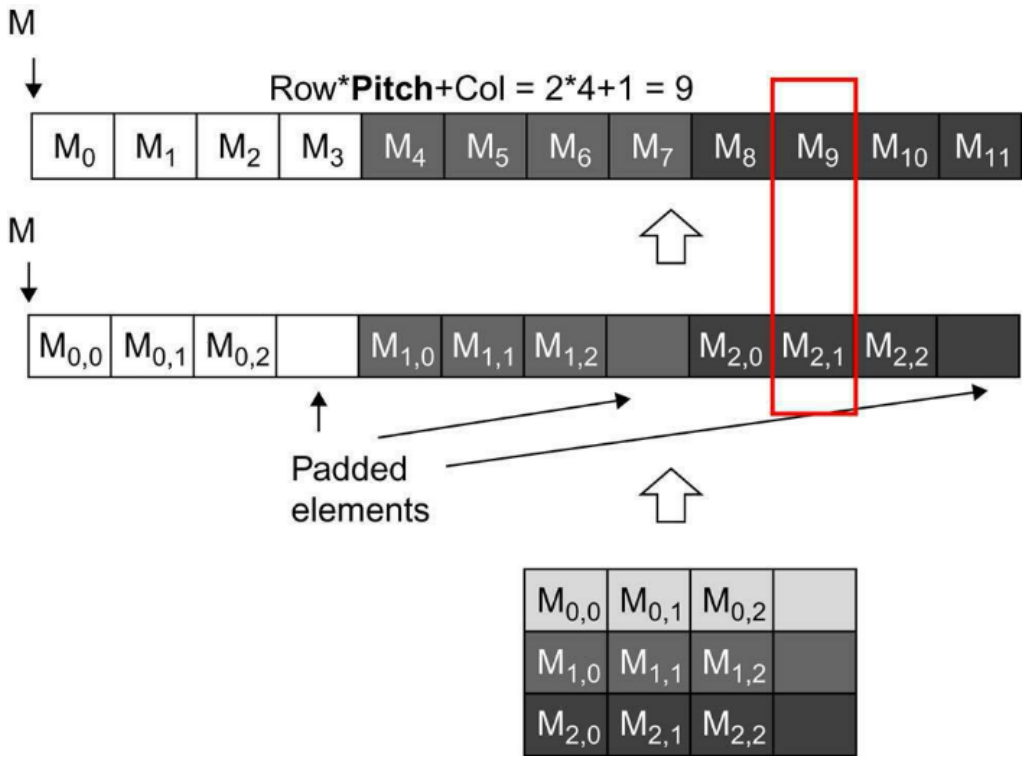
Spring 2019

April 25th, 2019

Simplified tiled kernel

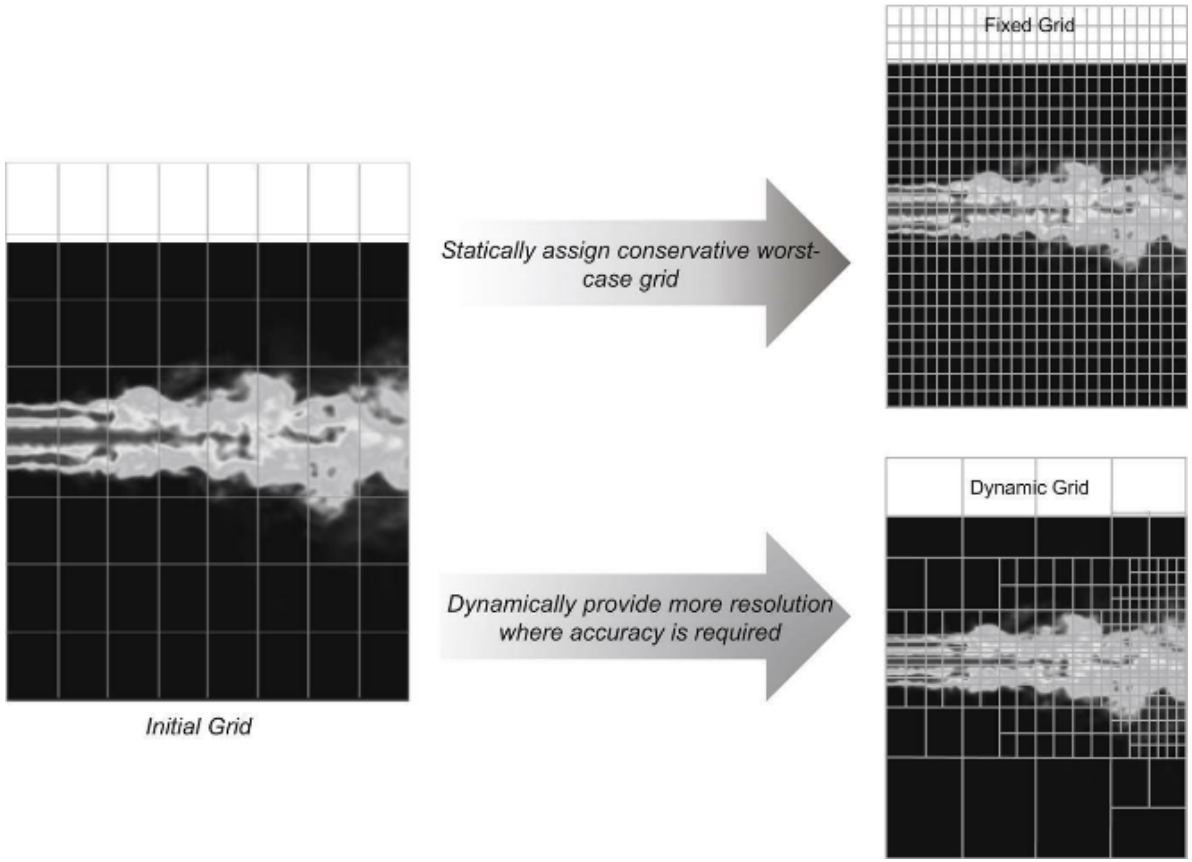
```
__global__ void convolution_1D_tiled_caching_kernel(float *N, float *P, int
  Mask_Width,int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE];

  N_ds[threadIdx.x] = N[i];

  __syncthreads();

  int This_tile_start_point = blockIdx.x * blockDim.x;
  int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
  int N_start_point = i - (Mask_Width/2);
  float Pvalue = 0;
  for (int j = 0; j < Mask_Width; j++) {
     int N_index = N_start_point + j;
     if (N_index >= 0  && N_index < Width) {
       if ((N_index >= This_tile_start_point)
         && (N_index < Next_tile_start_point)) {
         Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
       } else {
         Pvalue += N[N_index] * M[j];
       }
     }
  }
  P[i] = Pvalue;

}
```

Padded matrix access



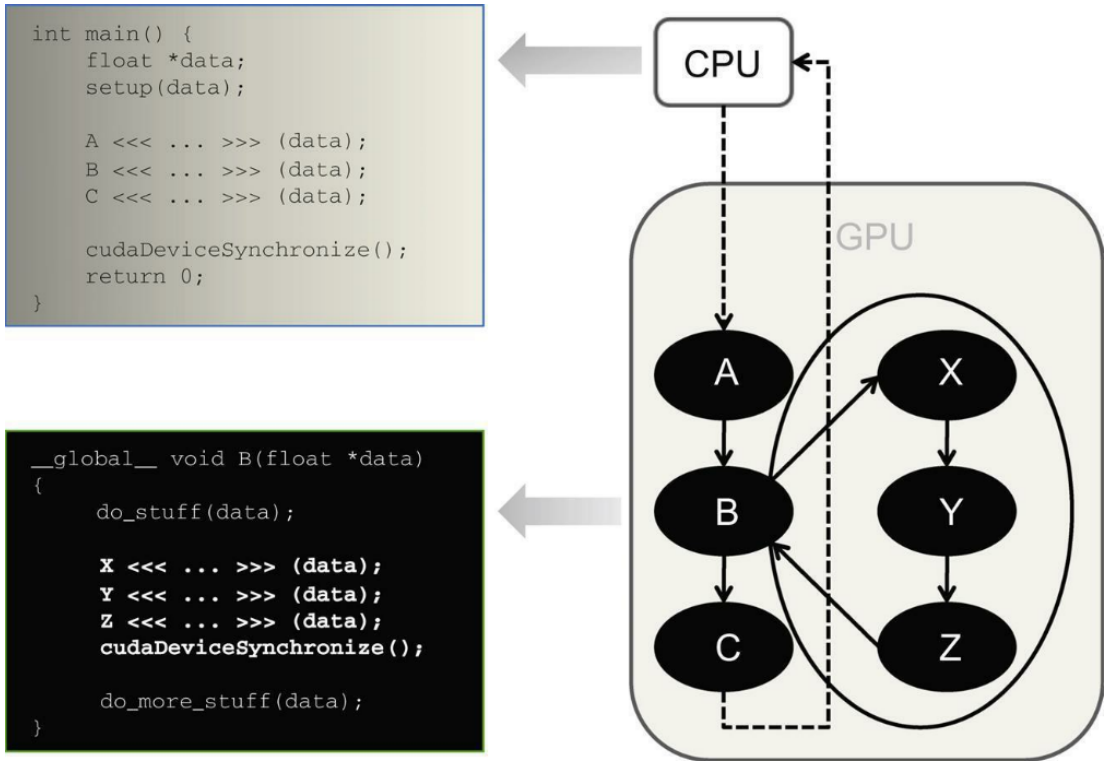
- We have to factor in the pitch size when moving along a linearized array
- The adjusted formula is Row*Pitch + Col

CUDA dynamic parallelism



- Data is often spatially, temporally, or structurally heterogenous
- The precision needed to process different subsets may differ

Dynamic parallelism overview



- Above, the CPU first launches three GPU kernels A, B, and C
- Kernel B then launches three additional kernels X, Y, and Z
  - This operation would have been invalid in pre-2012 versions of CUDA

Static vs. dynamic CUDA parallelism

```
01      __global__ void kernel(unsigned  int* start, unsigned int* end,float* someData,
02           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      }
```

- This code has two main limitations:
  1. We cannot parallelize the **for**-loop inside the kernel
  2. If the number of iterations per **for**-loop vary significantly, some threads will end up running for much longer than others
- Both problems are due to insufficient *granularity*
  - Static parallelism only allows one level of granularity or scale

Static vs. dynamic CUDA parallelism

```
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    }
```

- Above, we have a dynamically parallel version
- We specify the `kernel_child` function to handle the **for**-loop
- The **if**-statement inside the child kernel acts as the **for**-loop check in the original code

Dynamic parallelism remarks

- The previous example highlighted one of the key design patterns of parallel algorithms:
  - **Replace loops with parallel calls**
  - In principle, every **for**-loop is an opportunity to parallelize your code
  - With dynamic parallelism, we can go down an arbitrary number of nested levels
  - In practice, though, the additional overhead may not be worth the effort for small loops
- In addition, dynamic parallelism erases the software-level distinction between host and device
  - From the point of view of an algorithm, where it gets executed is immaterial

Memory data visibility

- We will now go over some memory management issues that arise in dynamic parallelism
- As always, a more nuanced used of memory resources will lead to much better performance
- **Memory data visibility** is the set of rules for who has access to what data
  - We already encountered it when comparing, e.g, shared memory (block-level) vs. registers (thread-level)
- The rules for dynamic parallelism are an extension of the rules we already know for static cases

Memory data visibility

- **Global memory:**
  - A parent and a child grid can make their global memory visible to each other at the beginning and end of the child's program execution:
    1. All global memory operations prior to spawning the child kernel are visible to the child grid
    2. After the child terminates, (and the parent issues a synchronization call), all memory operations are visible to the parent
- **Constant memory:**
  - Kernel functions cannot declare constants, no matter their nested level
  - Thus, all constant memory allocations must be done by the host prior to calling the first kernel function

Memory data visibility

- **Local memory:**
  - Local memory is private to each thread
  - A child kernel is executed by a different thread than its parent
  - Thus, it is illegal to pass a pointer to a local variable as an argument to a child kernel:

```
__device__ int value;
__device__ void x() {
    value = 5;
    child<<< 1, 1 >>>(&value);
}
```

(A) Valid–"value" is global storage

```
__device__ void y() {
    int value = 5;
    child<<< 1, 1 >>>(&value);
}
```

(B) Invalid–"value" is local storage

- **Shared memory:**
  - Shared memory is private to a block
  - Child kernels reside in a different *grid*, let alone a block, relative to the parent
  - Thus, it is illegal to pass shared memory variables as arguments to a child kernel

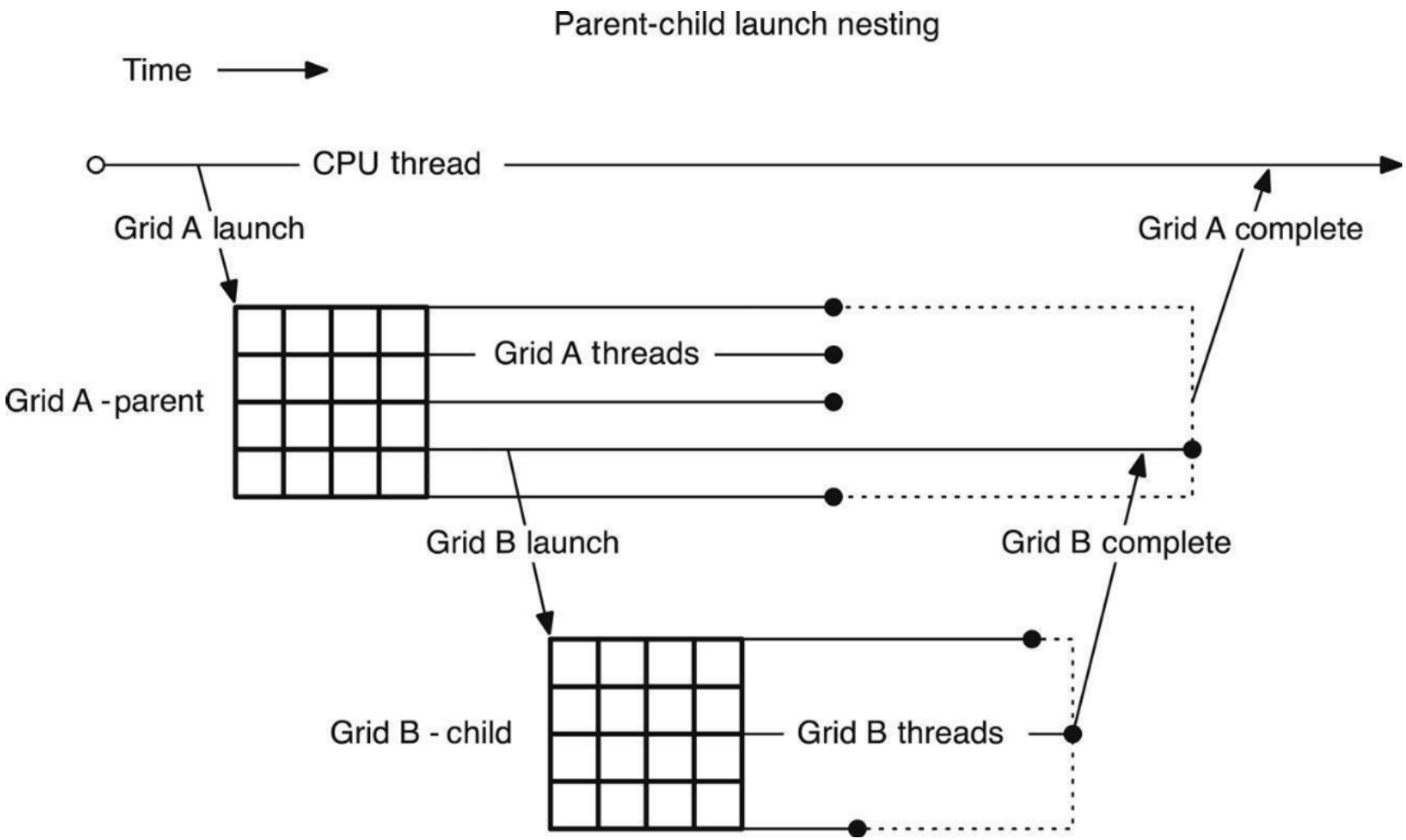
Other considerations

- **Dynamic memory:**
  - We can call `cudaMalloc()` and `cudaFree()` from inside a kernel function
  - The allocatable memory is limited to the device `malloc()` heap size
    - Will likely be much smaller than the host memory
  - In current-gen CUDA, you cannot free a pointer on the device that was allocated in the host (and vice versa)
- **Nesting levels:**
  - A kernel can call a kernel, which can then call a kernel, etc.
  - The number of such calls is the **nesting level**
  - The maximum nesting depth (i.e., the maximum number of times we can nest functions) is currently limited to 24
  - If a parent and child functions need to synchronize, the maximum-possible nesting depth may be less

Nested synchronization

- By default, nested kernel calls are non-blocking
  - The device will schedule them in the order that best matches the available hardware
- If a parent wants to synchronize with a child kernel, it must do so explicitly
- One way is by invoking `cudaDeviceSynchronize()`
  - A thread that invokes this function will wait until all kernels launched by any thread in the block have finished
  - Note that other threads in the block may continue processing their data
  - We must also call `__syncthreads()` to stop other threads
- If the programmer doesn't invoke any explicit synchronization, CUDA will synchronize a parent with its children *once the parent terminates*

Parent-child synchronization



- CUDA will ensure that a parent is synchronized with its children by the time it terminates

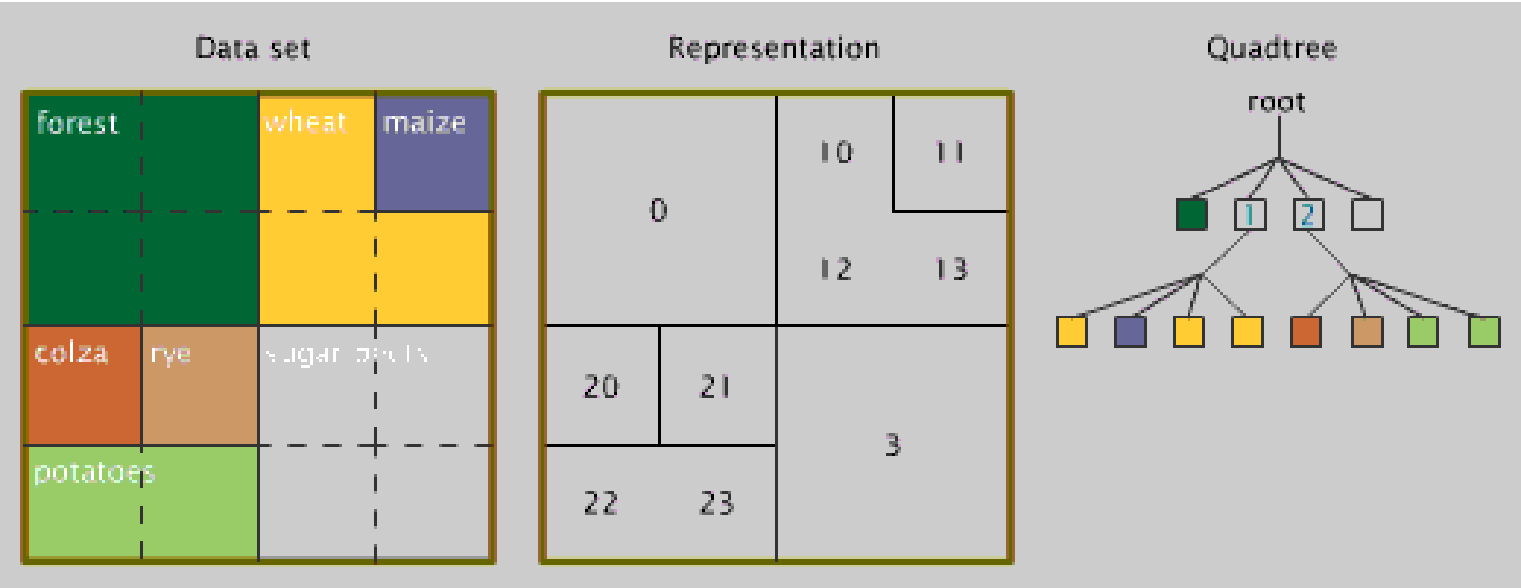
Synchronization depth

- If a parent must wait for a child kernel to finish, it must be put on hold
- Memory needs to be allocated to store the parent thread in the meantime
- Ancestors of the parent thread (if they're also blocking) will also have to be swapped out
- The deepest level for which we can swap threads in and out is the *synchronization depth*
- This depth is a function of the *backing store:* a chunk of memory reserved for potential swaps
  - Currently about 150MBs
  - Can be adjusted using the `cudaDeviceSetLimit()` function

Quadtrees

- Dynamic parallelism is ideally suited for implementing recursive algorithms/data structures
- Here, we will show this capability using a **quadtree** as an example
- Quadtrees are a tree-based data structure with the following properties:
  1. Each internal node has exactly four children
  2. Only the leaves contain data
  3. If a leaf node exceeds a "capacity" threshold, it is split into four children
- Quadtrees are most often used to hierarchically partition 2D data
- Their 3D counterparts are called (not surprisingly) *octrees*
- **Note:** not the same as *k-d trees*

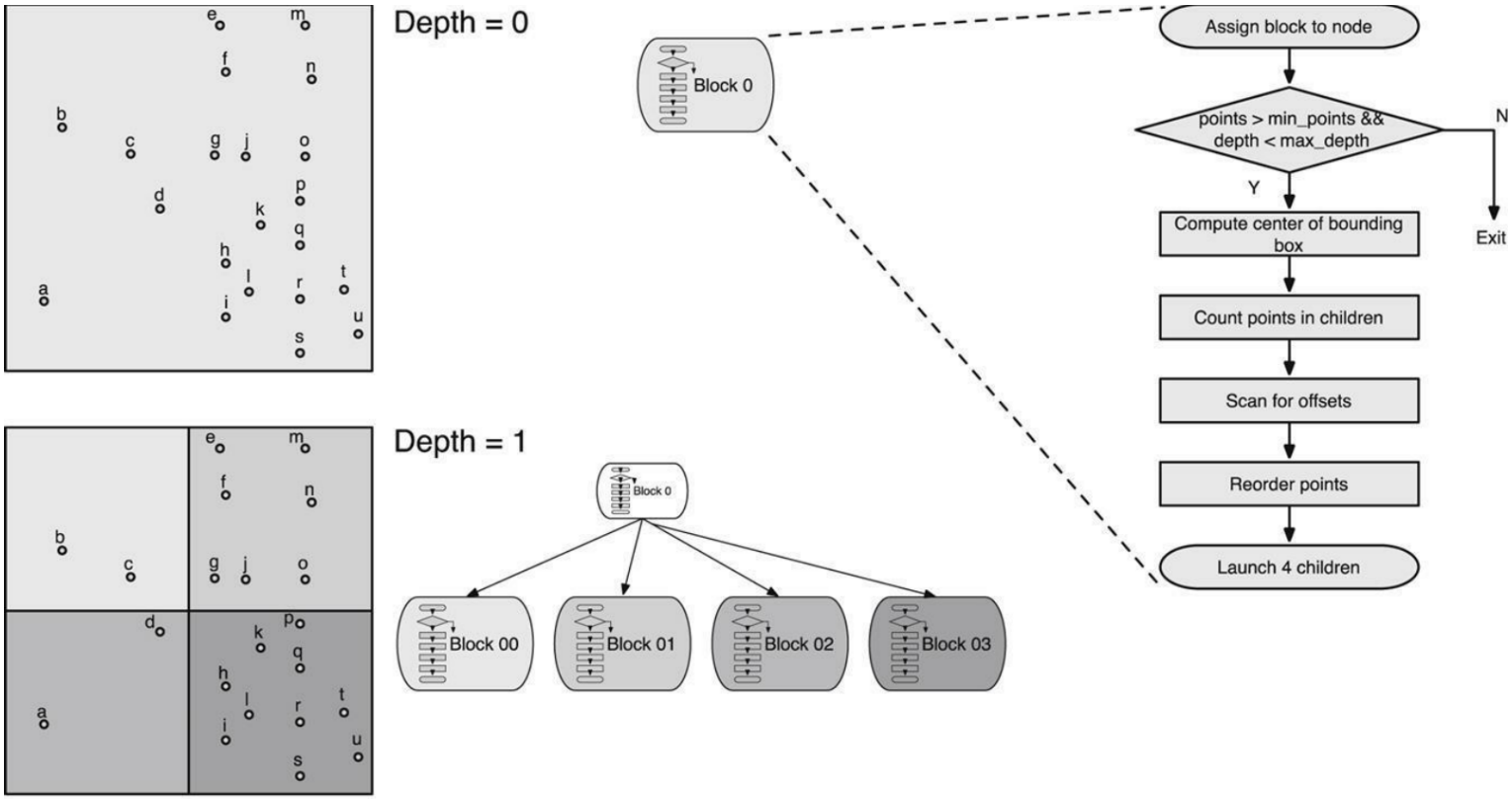
Quadtrees: simple example



Source: `http://www.gitta.info`

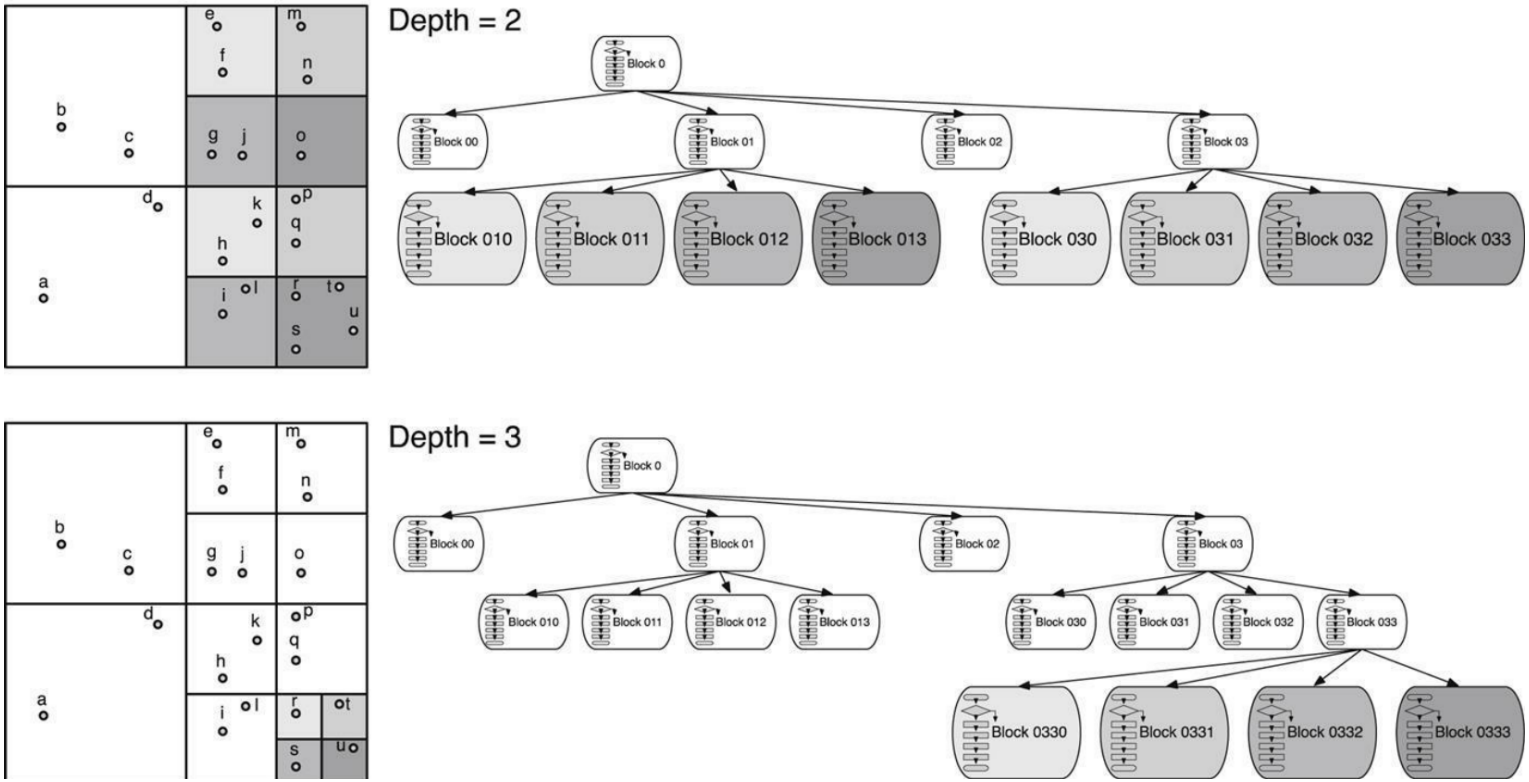
- If a leaf exceeds its carrying capacity, it is split in four
- Regions with more data are represented with more levels (i.e., more precision)

Quadtree: more complex example



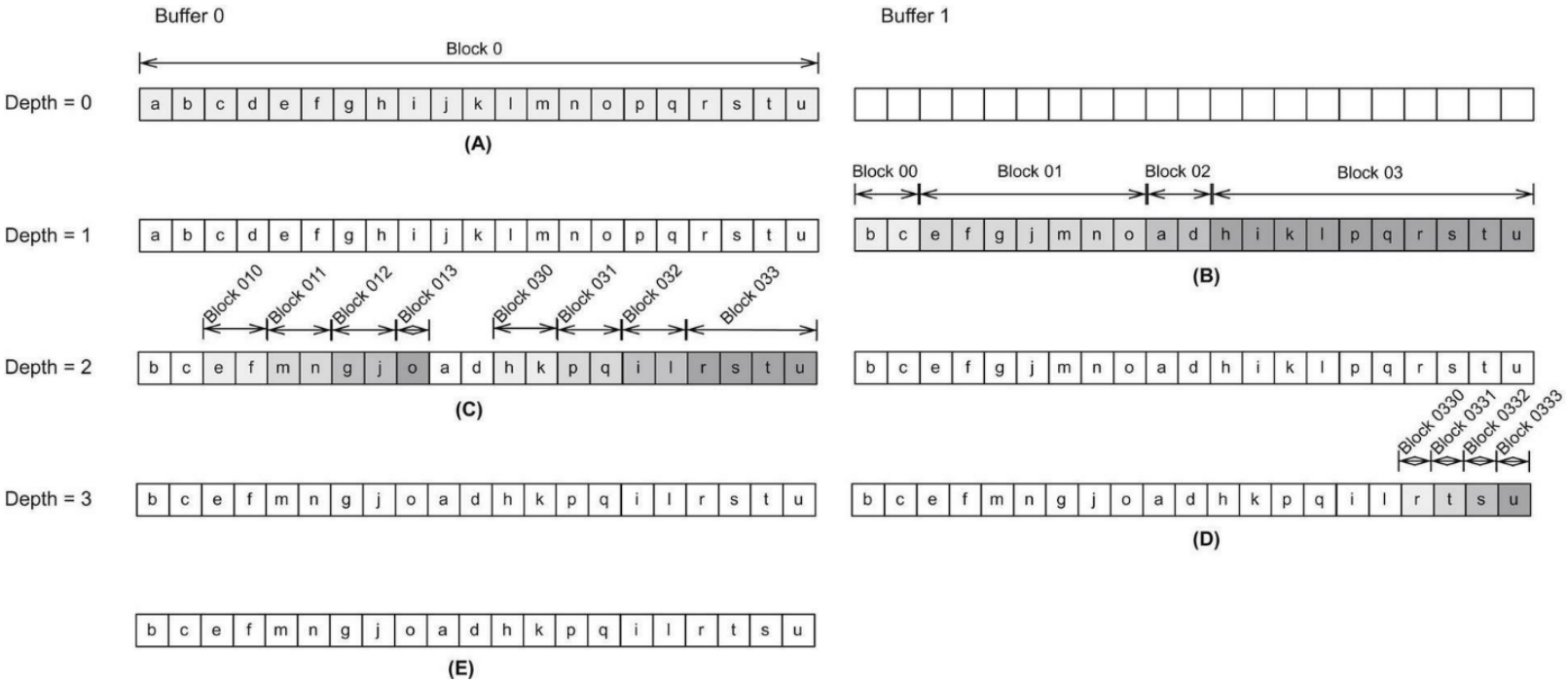
- As before, we hierarchically split a dataset
  - We assign new blocks of threads at each level
- The flowchart summarizes the steps of the kernel function

Quadtree: more complex example



- Regions that exceed the maximum capacity threshold (four, in this case), are recursively split

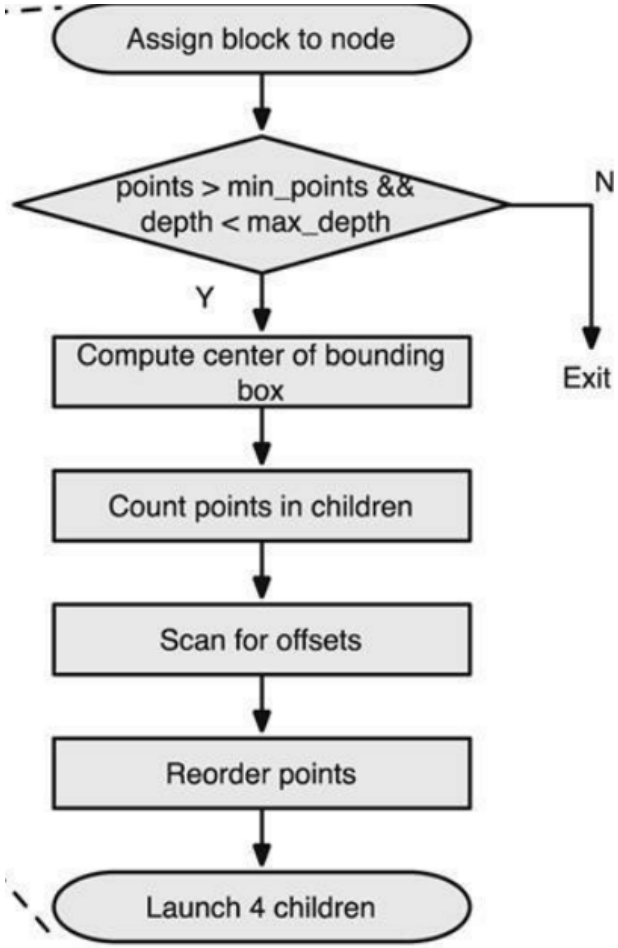
Quadtree representation



- At each level, we reorder the points to match the topology of the quadtree
- Same principle as binary heaps

Kernel function overview

- The flowchart summarizes the steps carried out by the kernel function
- Intuitively, we keep partitioning nodes until either they contain few enough points or we reach a maximum depth level
- Splitting consists of finding the center of the node's points and then reordering them to match the new quadrants



Code: Auxiliary classes

```
01    // A structure of 2D points
02    class Points {
03        float *m_x;
04        float *m_y;
05
06        public:
07        // Constructor
08        __host__ __device__ Points() : m_x(NULL), m_y(NULL) {}
09
10        // Constructor
11        __host__ __device__ Points(float *x, float *y) : m_x(x), m_y(y) {}
12
13        // Get a point
14        __host__ __device__ __forceinline__ float2 get_point(int idx) const {
15            return make_float2(m_x[idx], m_y[idx]);
16        }
17
18        // Set a point
19        __host__ __device__ __forceinline__ v  oid set_point(int idx, const float2 &p) {
20            m_x[idx] = p.x;
21            m_y[idx] = p.y;
22        }
23
24        // Set the pointers
25        __host__ __device__ __forceinline__ void set(float *x, float *y) {
26            m_x = x;
27            m_y = y;
28        }
29    };
30
31    // A 2D bounding box
32    class Bounding_box {
33        // Extreme points of the bounding box
34        float2 m_p_min;
35        float2 m_p_max;
```

Code: Auxiliary classes

```
37        public:
38        // Constructor. Create a unit box
39        __host__ __device__ Bounding_box(){
40            m_p_min = make_float2(0.0f, 0.0f);
41            m_p_max = make_float2(1.0f, 1.0f);
42        }
43
44        // Compute the center of the bounding-box
45        __host__ __device__ void compute_center(float2 &center) const {
46            center.x = 0.5f * (m_p_min.x + m_p_max.x);
47            center.y = 0.5f * (m_p_min.y + m_p_max.y);
48        }
49
50        // The points of the box
51        __host__ __devic e__ __forceinline__ const float2 &get_max() const {
52            return m_p_max;
53        }
54
55        __host__ __device__ __forceinline__ const float2 &get_min() const {
56            return m_p_min;
57        }
58
59        // Does a box contain a point
60        __host__ __device__ bool contains(const float2 &p) const {
61            return p.x>=m_p_min.x && p.x<m_p_max.x && p.y>=m_p_min.y && p.y< m_p_max.y;
62        }
63
64        // Define the bounding box
65        __host__  __device__ void set(float min_x, float min_y, float max_x, float max_y){
66            m_p_min.x = min_x;
67            m_p_min.y = min_y;
68            m_p_max.x = max_x;
69            m_p_max.y = max_y;
70        }
```

Code: Quadtree class

```
001  // A node of a quadree
002  class Quadtree_node {
003      // The identifier of the node
004      int m_id;
005      // The bounding box of the tree
006      Bounding_box m_bounding_box;
007      // The range of points
008      int m_begin, m_end;
009
010      public:
011      // Constructor
012      __host__ __device__ Quadtree_node() : m_id(0), m_begin(0), m_end(0) {}
013
014      // The ID of a node at its level
015      __host__ __device__ int id() const {
016          return m_id;
017      }
018
019      // The ID of a node at its level
020      __host__ __device__ void set_id(int new_id) {
021          m_id = new_id;
022      }
023
024      // The bounding box
025      __host__ __device__ __forceinline__ const Bounding_box &bounding_box() const {
026          return m_bounding_box;
027      }
028
029      // Set the bounding box
030      __host__ __device__ __forceinline__ void set_bounding_box(float min_x,
031          float min_y, float max_x, float max_y) {
032          m_bounding_box.set(min_x, min_y, max_x, max_y);
033      }
034
035      // The number of points in the tree
036      __host__ __device__ __forceinline__ int num_points() const {
037          return m_end - m_begin;
038      }
```

Code: Quadtree class cont. and Parameters class

```
039
040     // The range of points in the tree
041     __host__ __device__ __forceinline__ int points_begin() const {
042         return m_begin;
043     }
044
045     __host__ __device__ __forceinline__ int po  ints_end() const {
046         return m_end;
047     }
048
049     // Define the range for that node
050     __host__ __device__ __forceinline__ void set_range(int begin, int end) {
051         m_begin = begin;
052         m_end = end;
053     }
054 };
055
056 // Algorithm parameters
057 struct Parameters {
058     // Choose the right set of points to use as in/out
059     int point_selector;
060     // The number of nodes at a given level (2^k for level k)
061     int num_nodes_at_this_le vel;
062     // The recursion depth
063     int depth;
064     // The max value for depth
065     const int max_depth;
066     // The minimum number of points in a node to stop recursion
067     const int min_points_per_node;
068
069     // Constructor set to default values.
070     __host__ __device__ Parameters(int max_depth, int min_points_per_node) :
071         point_selector(0),
072         num_nodes_at_this_level(1),
073         depth(0),
074         max_depth(max_depth),
075         min_points_per_node(min_points_per_node) {}
```

- Parameters class is a C++ struct (properties default to public)

Code: Parameters cont. and main function

```
077     // Copy constructor. Changes the values for next iteration
078     __host__ __device__ Parameters(const Parameters &params, bool) :
079         point_selector((params.point_selector+1) % 2),
080         num_nodes_at_this_level(4*params.num_nodes_at_this_level),
081         depth(params.depth+1),
082         max_depth(params.max_depth),
083         min_points_per_node(params.min_points_per_node) {}
084 };
085
086 // Main function
087 void main(int argc, char **argv) {
088
089     // Constants to control the algorithm
090     const int num_points = atoi(argv[0]);
091     const int max_depth  = atoi(argv[1]);
092     const int min_points_per_node = atoi(argv[2]);
093
094     // Allocate memory for points
095     thrust::device_vector<float> x_d0(num_points);
096     thrust::device_vector<float> x_d1(num_points);
097     thrust::device_vector<float> y_d0(num_points);
098     thrust::device_vector<float> y_d1(num_points);
099
100     // Generate random points
101     Random_generator rnd;
102     thrust::generate(
103         thrust::make_zip_iterator(thrust::make_tuple(x_d0.begin(), y_d0.begin())),
104         thrust::make_zip_iterator(thrust::make_tuple(x_d0.end(), y_d0.end())),
105         rnd);
106
107     // Host structures to analyze the device ones
108     Points points_init[2];
109     points_init[0].set(thrust::raw_pointer_cast(&x_d0[0]),
110                        thrust::raw_pointer_cast(&y_d0[0]));
111     points_init[1].set(thrust::raw_pointer_cast(&x_d1[0]),
```

Code: Main function cont.

```
111     points_init[1].set(thrust::raw_pointer_cast(&x_d1[0]),
112                        thrust::raw_pointer_cast(&y_d1[0]));
113
114     // Allocate memory to store points
115     Points *points;
116     cudaMalloc((void **) &points, 2*sizeof(Points));
117     cudaMemcpy(points, points_init, 2*sizeof(Points), cudaMemcpyHostToDevice);
118
119     // We could use a close form...
120     int max_nodes = 0;
121
122     for (int i=0, num_nodes_at_level=1 ; i<max_depth ; ++i, num_nodes_at_level*=4)
123         max_nodes += num_nodes_at_level;
124
125     // Allocate memory to store the tree
126     Quadtree_node root;
127     root.set_range(0, num_points);
128     Quadtree_node *nodes;
129     cudaMalloc((void **) &nodes, max_nodes*sizeof(Quadtree_node));
130     cudaMemcpy(nodes, &root, sizeof(Quadtree_node), cudaMemcpyHostToDevice);
131
132     // We set the recursion limit for CDP to max_depth
133     cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth, max_depth);
134
135     // Build the quadtree
136     Parameters params(max_depth, min_points_per_node);
137     const int NUM_THREADS_PER_BLOCK = 128;
138     const size_t smem_size = 8*sizeof(int);
139     build_quadtree_kernel<<<1, NUM_THREADS_PER_BLOCK, smem_size>>>
140         (nodes, points, params);
141     cudaGetLastError();
142
143     // Free memory
144     cudaFree(nodes);
145     cudaFree(points);
```

Code: Main kernel function

```
01  __global__ void build_quadtree_kernel
02                (Quadtree_node *nodes, Points *points, Parameters params) {
03      __shared__ int smem[8];  // 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          // The children
37          Quadtree_node *children = &nodes[params.num_nodes_at_this_level];
38
39          // Prepare children launch
40          prepare_children(children, node, bbox, smem);
41
42          // Launch 4 children.
43          build_quadtree_kernel<<<4, blockDim.x, 8 *sizeof(int)>>>
44                          (children, points, Parameters(params, true));
45      }
```

Code: Device functions

```
001  // Check the number of points and its depth
002  __device__ bool check_num_points_and_depth(Quadtree_node &node, Points *points,
003                                             int num_points, Parameters params){
004      if(params.depth >= params.max_depth || num_points <= params.min_points_per_node) {
005      // Stop the recursion here. Make sure points[0] contains all the points
006        if(params.point_selector == 1) {
007          int it = node.points_begin(), end = node.points_end();
008          for (it += threadIdx.x ; it < end ; it += blockDim.x)
009            if(it < end)
010              points[0].set_point(it, points[1].get_point(it));
011        }
012        return true;
013      }
014      return false;
015  }
016
017  // Count the number of points in each quadrant
018  __device__ void count_points_in_children(const Points &in_points, int* smem,
019      int range_begin, int range_end, float2 center) {
020      // Initizalize shared memory
021      if(threadIdx.x < 4) smem[threadIdx.x] = 0;
022      __syncthreads();
023      // Compute the number of points
024      for(int iter=range_begin+threadIdx.x; iter<range_end; iter+=blockDim.x){
025          float2 p = in_points.get_point(iter); // Load the coordinates of the point
026          if(p.x < center.x && p.y >= center.y)
027              atomicAdd(&smem[0], 1); // Top-left point?
028          if(p.x >= center.x && p.y >= center.y)
029              atomicAdd(&smem[1], 1); // Top-right point?
030          if(p.x < center.x && p.y < center.y)
031              atomicAdd(&smem[2], 1); // Bottom-left point?
032          if(p.x >= center.x && p.y < center.y)
033              atomicAdd(&smem[3], 1); // Bottom-right point?
034      }
```

- **Note:** a `__syncthreads()` call at the bottom was cut off from the screenshot

Code: Device functions cont.

```
038 // Scan quadrants' results to obtain reordering offset
039 __device__ void scan_for_offsets(int node_points_begin, int* smem){
040     int* smem2 = &smem[4];
041     if(threadIdx.x == 0){
042     for(int i = 0; i < 4; i++)
043         smem2[i] = i==0 ? 0 : smem2[i-1] + smem[i-1]; // Sequential scan
044     for(int i = 0; i < 4; i++)
045         smem2[i] += node_points_begin;   // Global offset
046     }
047     __syncthreads();
048 }
049
050 // Reorder points in order to group the points in each quadrant
051 __device__ void reorder_points(
052                 Points& out_points, const Points &in_points, int* smem,
053                 int range_begin, int range_end, float2 center){
054     int* smem2 = &smem[4];
055     // Reorder points
056     for(int iter=range_begin+threadIdx.x; iter<range_end; iter+=blockDim.x){
057         int dest;
058         float2 p = in_points.get_point(iter); // Load the coordinates of the point
059         if(p.x<center.x && p.y>=center.y)
060             dest=atomicAdd(&smem2[0],1); // Top-left point?
061         if(p.x>=center.x && p.y>=center.y)
062             dest=atomicAdd(&smem2[1],1); // Top-right point?
063         if(p.x<center.x && p.y<center.y)
064             dest=atomicAdd(&smem2[2],1); // Bottom-left point?
065         if(p.x>=center.x && p.y<center.y)
066             dest=atomicAdd(&smem2[3],1); // Bottom-right point?
067         // Move point
068         out_points.set_point(dest, p);
069     }
070     __syncthreads();
```

Code: Device functions cont.

```
072
073  // Prepare children launch
074  __device__ void prepare_children(Quadtree_node *children, Quadtree_node &node,
075                                   const Bounding_box &bbox, int *smem){
076      int child_offset = 4*node.id(); // The offsets of the children at their level
077
078      // Set IDs
079      children[child_offset+0].set_id(4*node.id()+ 0);
080      children[child_offset+1].set_id(4*node.id()+ 4);
081      children[child_offset+2].set_id(4*node.id()+ 8);
082      children[child_offset+3].set_id(4*node.id()+12);
083
084      // Points of the bounding-box
085      const float2 &p_min = bbox.get_min();
086      const float2 &p_max = bbox.get_max();
087
088      // Set the bounding boxes of the children
089      children[child_offset+0].set_bounding_box(
090          p_min.x , center.y, center.x, p_max.y);     // Top-left
091      children[child_offset+1].set_bounding_box(
092          center.x, center.y, p_max.x , p_max.y);     // Top-right
093      children[child_offset+2].set_bounding_box(
094          p_min.x , p_min.y , center.x, center.y);    // Bottom-left
095      children[child_offset+3].set_bounding_box(
096          center.x, p_min.y , p_max.x , center.y);    // Bottom-right
097
098      // Set the ranges of the children.
099      children[child_offset+0].set_range(node.points_begin(),   smem[4 + 0]);
100     children[child_offset+1].set_range(smem[4 + 0], smem[4 + 1]);
101     children[child_offset+2].set_range(smem[4 + 1], smem[4 + 2]);
102     children[child_offset+3].set_range(smem[4 + 2], smem[4 + 3]);
103 }
```