Chapter-13
CUDA dynamic parallelism

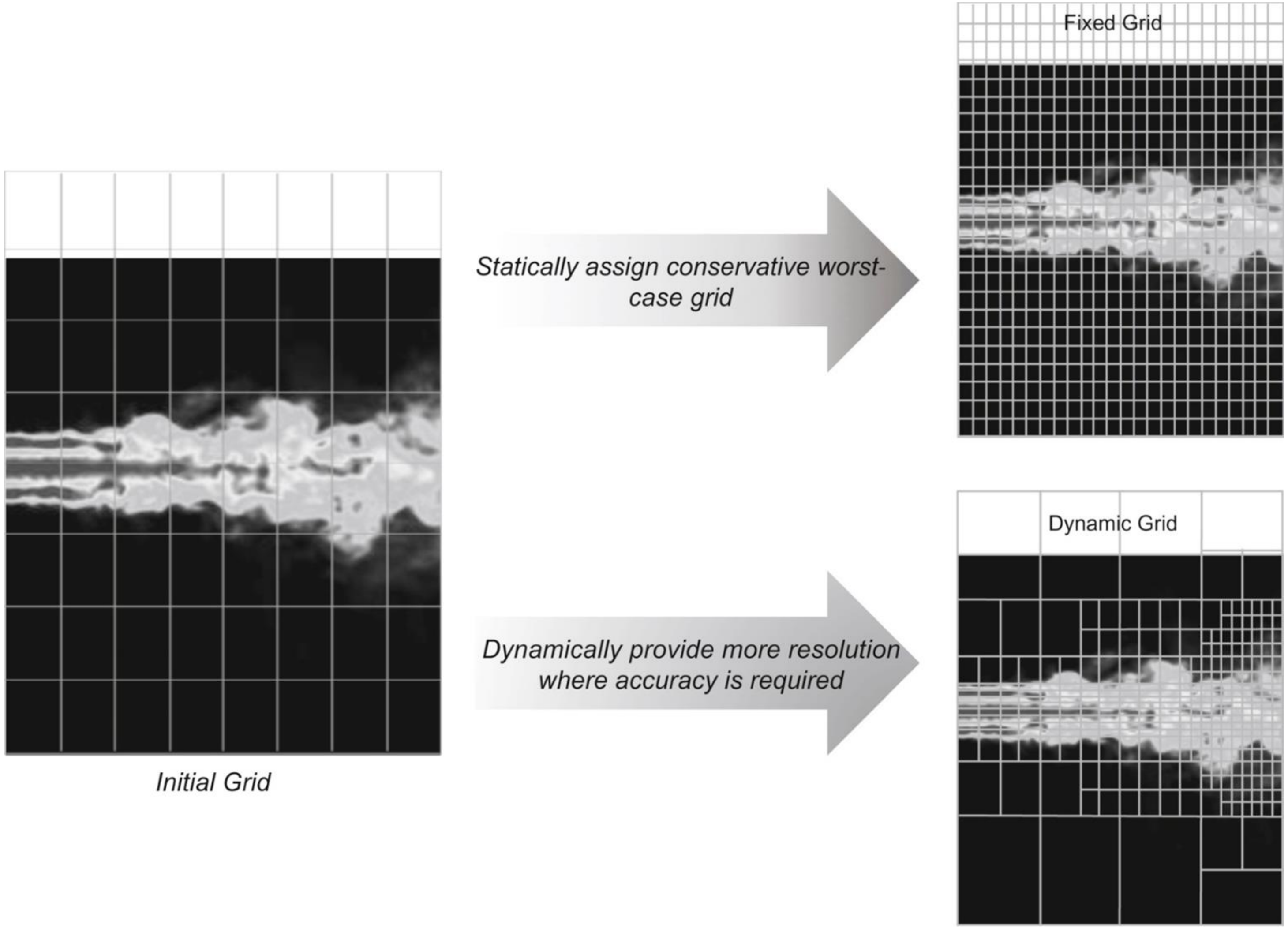

FIGURE 13.1: Fixed versus dynamic grids for a turbulence simulation model.

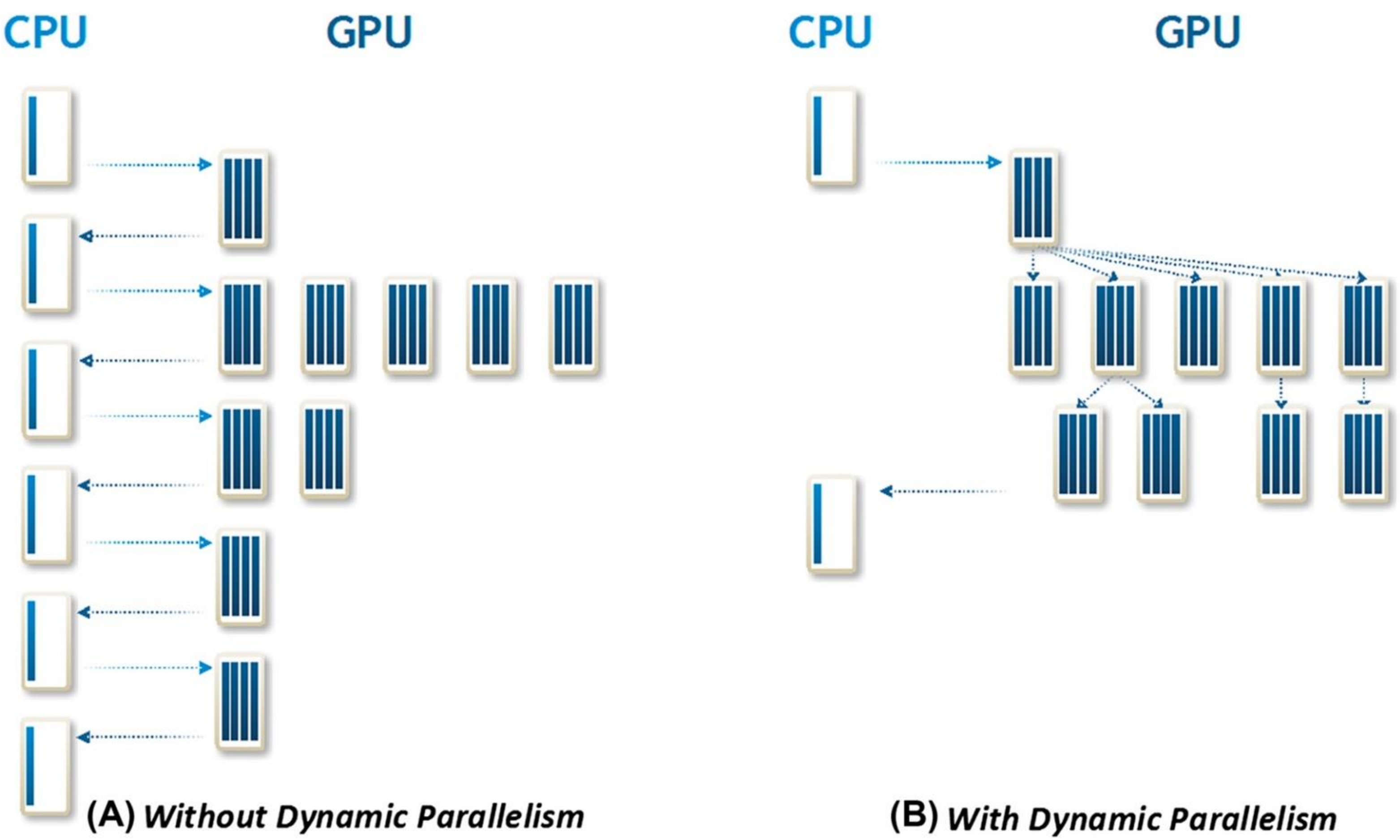

FIGURE 13.2: Kernel launch patterns for algorithms with dynamic work variation, with and without dynamic parallelism.

```
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);
}
```

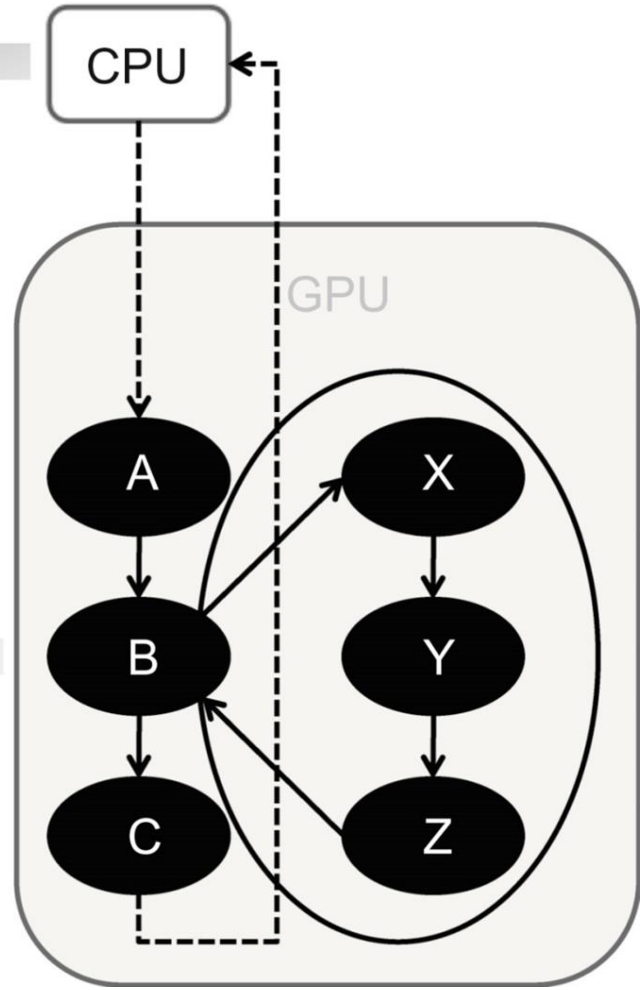

FIGURE 13.3: A simple example of a kernel (B) launching three kernels (X, Y, and Z).

```
__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]);
     }

}
```

FIGURE 13.4: A simple example of a hypothetical parallel algorithm coded in CUDA without dynamic parallelism.

```
__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]);
    }

}
```

FIGURE 13.5: A revised example using CUDA dynamic parallelism.

```
__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

FIGURE 13.6: Passing a pointer as an argument to a child kernel.

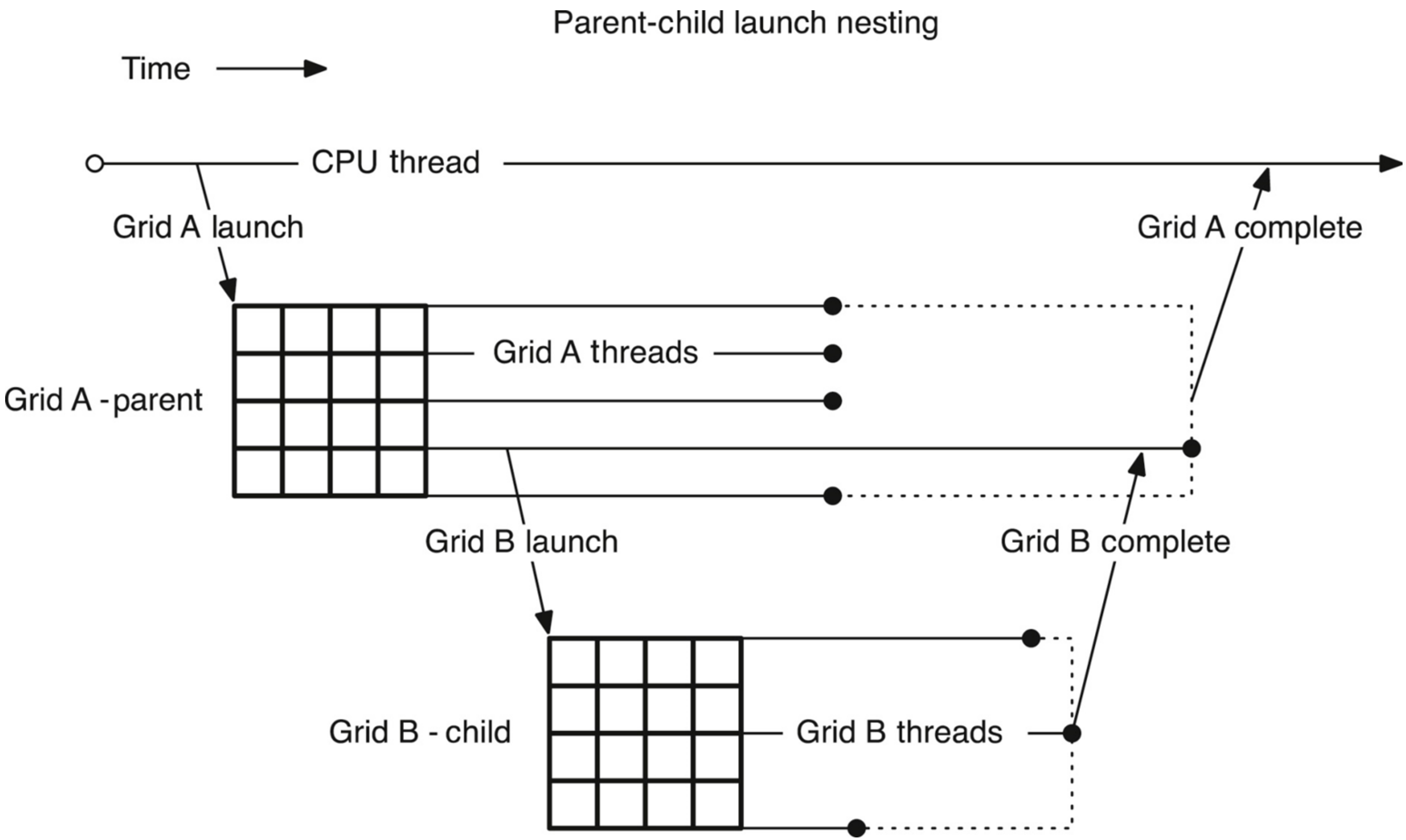

FIGURE 13.7: Completion sequence for parent and child grids.

```
#include <stdio.h>
#include <cuda.h>

#define MAX_TESS_POINTS 32

//A structure containing all parameters needed to tessellate a Bezier line
struct BezierLine {
    float2 CP[3];                       //Control points for the line
    float2 vertexPos[MAX_TESS_POINTS];  //Vertex position array to tessellate into
    int nVertices;                      //Number of tessellated vertices
};

__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;
            }
        }
    }
}

#define N_LINES 256
#define BLOCK_DIM 32

int main( int argc, char **argv ) {
    //Allocate and initialize array of lines in host memory
    BezierLine *bLines_h = new BezierLine[N_LINES];
    initializeBLines(bLines_h);

    //Allocate device memory for array of Bezier lines
    BezierLine *bLines_d;
    cudaMalloc((void**)&bLines_d, N_LINES*sizeof(BezierLine));
    cudaMemcpy(bLines_d,bLines_h, N_LINES*sizeof(BezierLine),cudaMemcpyHostToDevice);

    //Call the kernel to tessellate the lines
    computeBezierLines<<<N_LINES, BLOCK_DIM>>>(bLines_d, N_LINES );

    cudaFree(bLines_d); //Free the array of lines in device memory
    delete[] bLines_h;  //Free the array of lines in host memory
}
```

FIGURE 13.8: Bezier curve calculation without dynamic parallelism (support code in Fig. A13.8).

```
struct BezierLine {
    float2 CP[3];       //Control points for the line
    float2 *vertexPos; //Vertex position array to tessellate into
    int nVertices;      //Number of tessellated vertices
};
__global__ void computeBez ierLines_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);
    }
}
__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;
    }
}
__global__ void freeVertexMem(BezierLine *bLines, int nLines) {
    //Compute a unique index for each Bezier line
    int lidx = threadIdx.x + blockDim.x*blockIdx.x;
    if(lidx < nLines)
        cudaFree(bLines[lidx].vertexPos);   //Free the vertex memory for this line
}
int main( int argc, char **argv ) {
    //Allocate array of lines in host memory
    BezierLine *bLines_h = new BezierLine[N_LINES];
    initializeBLines(bLines_h);

    //Allocate device memory for array of Bezier lines
    BezierLine *bLines_d;
    cudaMalloc((void**)&bLines_d, N_LINES*sizeof(BezierLine));
    cudaMemcpy(bLines_d,bLines_h, N_LINES*sizeof(Bezi erLine),cudaMemcpyHostToDevice);

    computeBezierLines_parent<<<ceil((float)N_LINES/(float)BLOCK_DIM), BLOCK_DIM>>>
        (bLines_d, N_LINES);

    freeVertexMem <<<ceil((float)N_LINES/(float)BLOCK_DIM), BLOCK_DIM>>>
        (bLines_d, N_LINES);
    cudaFree(bLines_d);   //Free the array of lines in device memory
    delete[] bLines_h;    //Free the array of lines in host memory
}
```

FIGURE 13.9: Bezier calculation with dynamic parallelism (support code in Fig. A13.8).

```
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);
```

FIGURE 13.10: Child kernel launch with named streams.

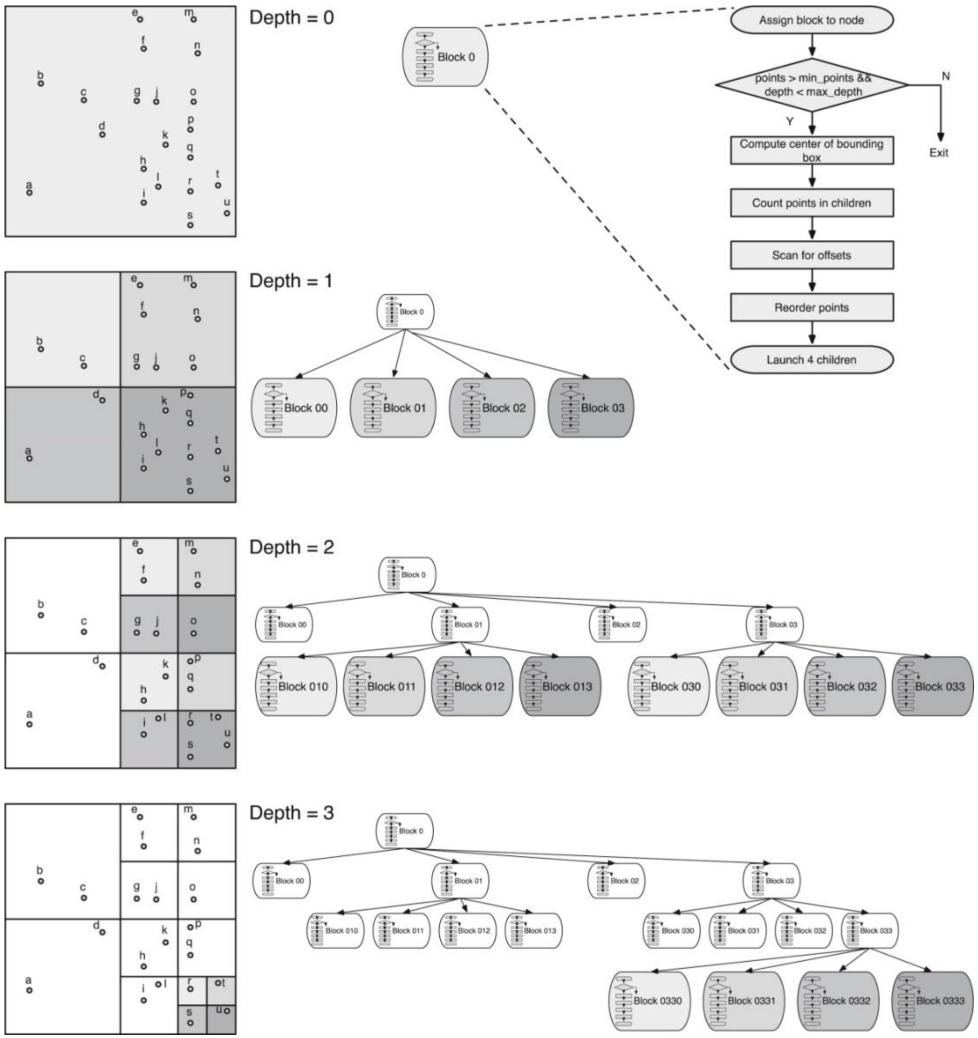

FIGURE 13.11: Quadtree example. Each thread-block is assigned to one quadrant. If the number of points in a quadrant is more than 2, the block launches 4 child blocks. Shadowed blocks are active blocks in each level of depth.

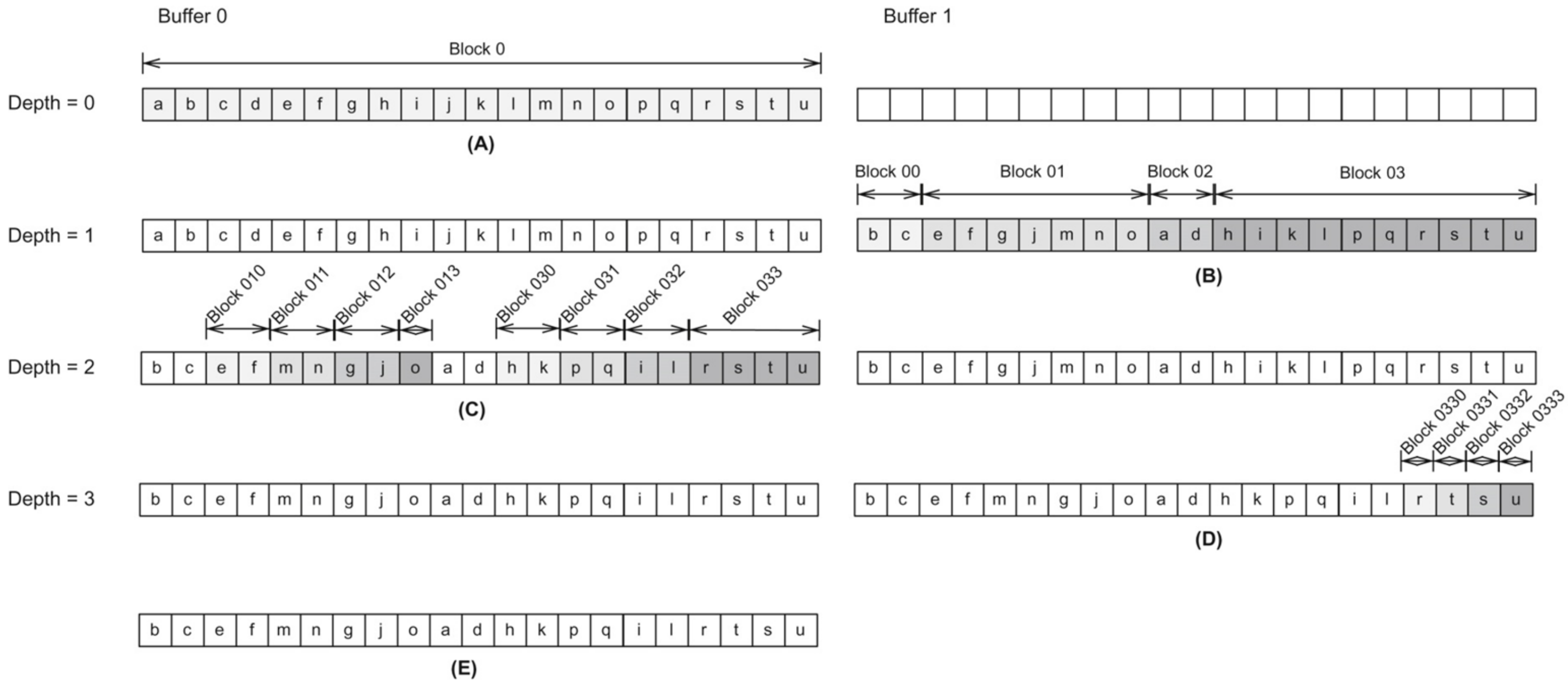

FIGURE 13.12: Quadtree example. At each level of depth, a block groups all points in the same quadrant together.

```
__global__ void build_quadtree_kernel
                (Quadtree_node *nodes, Points *points, Parameters params) {
    __shared__ int smem[8];  // 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) {
        // The children
        Quadtree_node *children = &nodes[params.num_nodes_at_this_level];

        // 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));
    }
}
```

FIGURE 13.13: Quadtree with dynamic parallelism: recursive kernel (support code in Fig. A13.13).

```
// Check the number of points and its depth
__device__ bool check_num_points_and_depth(Quadtree_node &node, Points *points,
                                            int num_points, Parameters params){
    if(params.depth >= params.max_depth || num_points <= params.min_points_per_node) {
    // Stop the recursion here. Make sure points[0] contains all the points
      if(params.point_selector == 1) {
        int it = node.points_begin(), end = node.points_end();
        for (it += threadIdx.x ; it < end ; it += blockDim.x)
          if(it < end)
            points[0].set_point(it, points[1].get_point(it));
      }
      return true;
    }
    return false;
}

// Count the number of points in each quadrant
__device__ void count_points_in_children(const Points &in_points, int* smem,
    int range_begin, int range_end, float2 center) {
    // Initizalize shared memory
    if(threadIdx.x < 4) smem[threadIdx.x] = 0;
    __syncthreads();
    // Compute the number of points
    for(int iter=range_begin+threadIdx.x; iter<range_end; iter+=blockDim.x){
        float2 p = in_points.get_point(iter); // Load the coordinates of the point
        if(p.x < center.x && p.y >= center.y)
            atomicAdd(&smem[0], 1); // Top-left point?
        if(p.x >= center.x && p.y >= center.y)
            atomicAdd(&smem[1], 1); // Top-right point?
        if(p.x < center.x && p.y < center.y)
            atomicAdd(&smem[2], 1); // Bottom-left point?
        if(p.x >= center.x && p.y < center.y)
            atomicAdd(&smem[3], 1); // Bottom-right point?
    }
    __syncthreads();
}

// Scan quadrants' results to obtain reordering offset
__device__ void scan_for_offsets(int node_points_begin, int* smem){
    int* smem2 = &smem[4];
    if(threadIdx.x == 0){
    for(int i = 0; i < 4; i++)
        smem2[i] = i==0 ? 0 : smem2[i-1] + smem[i-1]; // Sequential scan
    for(int i = 0; i < 4; i++)
        smem2[i] += node_points_begin;   // Global offset
    }
    __syncthreads();
}

// Reorder points in order to group the points in each quadrant
__device__ void reorder_points(
                Points& out_points, const Points &in_points, int* smem,
                int range_begin, int range_end, float2 center){
    int* smem2 = &smem[4];
    // Reorder points
    for(int iter=range_begin+threadIdx.x; iter<range_end; iter+=blockDim.x){
        int dest;
        float2 p = in_points.get_point(iter); // Load the coordinates of the point
        if(p.x<center.x && p.y>=center.y)
            dest=atomicAdd(&smem2[0],1); // Top-left point?
        if(p.x>=center.x && p.y>=center.y)
            dest=atomicAdd(&smem2[1],1); // Top-right point?
        if(p.x<center.x && p.y<center.y)
            dest=atomicAdd(&smem2[2],1); // Bottom-left point?
        if(p.x>=center.x && p.y<center.y)
            dest=atomicAdd(&smem2[3],1); // Bottom-right point?
        // Move point
        out_points.set_point(dest, p);
    }
    __syncthreads();
}
```

FIGURE 13.14: Quadtree with dynamic parallelism: device functions (support code in Fig. A13.14).

```
// Prepare children launch
__device__ void prepare_children(Quadtree_node *children, Quadtree_node &node,
                                 const Bounding_box &bbox, int *smem){
    int child_offset = 4*node.id(); // The offsets of the children at their level

    // Set IDs
    children[child_offset+0].set_id(4*node.id()+ 0);
    children[child_offset+1].set_id(4*node.id()+ 4);
    children[child_offset+2].set_id(4*node.id()+ 8);
    children[child_offset+3].set_id(4*node.id()+12);

    // Points of the bounding-box
    const float2 &p_min = bbox.get_min();
    const float2 &p_max = bbox.get_max();

    // Set the bounding boxes of the children
    children[child_offset+0].set_bounding_box(
        p_min.x , center.y, center.x, p_max.y);    // Top-left
    children[child_offset+1].set_bounding_box(
        center.x, center.y, p_max.x , p_max.y);    // Top-right
    children[child_offset+2].set_bounding_box(
        p_min.x , p_min.y , center.x, center.y);   // Bottom-left
    children[child_offset+3].set_bounding_box(
        center.x, p_min.y , p_max.x , center.y);   // Bottom-right

    // Set the ranges of the children.
    children[child_offset+0].set_range(node.points_begin(),   smem[4 + 0]);
    children[child_offset+1].set_range(smem[4 + 0], smem[4 + 1]);
    children[child_offset+2].set_range(smem[4 + 1], smem[4 + 2]);
    children[child_offset+3].set_range(smem[4 + 2], smem[4 + 3]);
}
```

FIGURE 13.14: (Continued)

```
//Some inline vector math functions
__forceinline__ __device__ float2 operator+(float2 a, float2 b) {
    float2 c;
    c.x = a.x + b.x;    c.y = a.y + b.y;
    return c;
}

__forceinline__ __device__ float2 operator -(float2 a, float2 b) {
    float2 c;
    c.x = a.x - b.x;    c.y = a.y - b.y;
    return c;
}

__forceinline__ __device__ float2 operator*(float a, float2 b) {
    float2 c;
    c.x = a * b.x;    c.y = a * b.y;
    return c;
}

__forceinline__ __device__ float length(float2 a) {
    return sqrtf(a.x*a.x + a.y*a.y);
}

//Device function that computes the curvature of a line
__device__ float computeCurvature(BezierLine *bLines){
    int bidx = blockIdx.x;
    float curvature = length(bLines[bidx].CP[1] - 0.5f*(bLines[bidx].CP[0]
        + bLines[bidx].CP[2]))/length(bLines[bidx].CP[2]
        - bLines[bidx].CP[0]);
    return curvature;
}

void initializeBLines(BezierLine *bLines_h) {
    //Set initial point to zero (last is last point in the previous segment)
    float2 last = {0,0};
    for(int i = 0; i < N_LINES; i++){
        //Set first point of this line to last point of previous line
        bLines_h[i].CP[0] = last;
        for(int j = 1; j < 3; j++) {
            //Assign random coordinate between 0 and 1
            bLines_h[i].CP[j].x = (float)rand()/(float)RAND_MAX;
            //Assign random coordinate between 0 and 1
            bLines_h[i].CP[j].y = (float)rand()/(float)RAND_MAX;
        }
        last = bLines_h[i].CP[2];   //keep the last point of this line
        //Set number of tessellated vertices to zero
        bLines_h[i].nVertices = 0;
    }
}
```

FIGURE A13.8 : Support code for Bezier Curve calculation without dynamic parallelism.

```
// A structure of 2D points
class Points {
    float *m_x;
    float *m_y;

    public:
    // Constructor
    __host__ __device__ Points() : m_x(NULL), m_y(NULL) {}

    // Constructor
    __host__ __device__ Points(float *x, float *y) : m_x(x), m_y(y) {}

    // Get a point
    __host__ __device__ __forceinline__ float2 get_point(int idx) const {
        return make_float2(m_x[idx], m_y[idx]);
    }

    // Set a point
    __host__ __device__ __forceinline__ v  oid set_point(int idx, const float2 &p) {
        m_x[idx] = p.x;
        m_y[idx] = p.y;
    }

    // Set the pointers
    __host__ __device__ __forceinline__ void set(float *x, float *y) {
        m_x = x;
        m_y = y;
    }
};

// A 2D bounding box
class Bounding_box {
    // Extreme points of the bounding box
    float2 m_p_min;
    float2 m_p_max;

    public:
    // Constructor. Create a unit box
    __host__ __device__ Bounding_box(){
        m_p_min = make_float2(0.0f, 0.0f);
        m_p_max = make_float2(1.0f, 1.0f);
    }

    // Compute the center of the bounding-box
    __host__ __device__ void compute_center(float2 &center) const {
        center.x = 0.5f * (m_p_min.x + m_p_max.x);
        center.y = 0.5f * (m_p_min.y + m_p_max.y);
    }

    // The points of the box
    __host__ __devic e__ __forceinline__ const float2 &get_max() const {
        return m_p_max;
    }

    __host__ __device__ __forceinline__ const float2 &get_min() const {
        return m_p_min;
    }

    // Does a box contain a point
    __host__ __device__ bool contains(const float2 &p) const {
        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;
    }

    // Define the bounding box
    __host__  __device__ void set(float min_x, float min_y, float max_x, float max_y){
        m_p_min.x = min_x;
        m_p_min.y = min_y;
        m_p_max.x = max_x;
        m_p_max.y = max_y;
    }
};
```

FIGURE A13.13 : Support code for quadtree with dynamic parallelism: definition of points and bounding box.

```
// A node of a quadree
class Quadtree_node {
    // The identifier of the node
    int m_id;
    // The bounding box of the tree
    Bounding_box m_bounding_box;
    // The range of points
    int m_begin, m_end;

    public:
    // Constructor
    __host__ __device__ Quadtree_node() : m_id(0), m_begin(0), m_end(0) {}

    // The ID of a node at its level
    __host__ __device__ int id() const {
        return m_id;
    }

    // The ID of a node at its level
    __host__ __device__ void set_id(int new_id) {
        m_id = new_id;
    }

    // The bounding box
    __host__ __device__ __forceinline__ const Bounding_box &bounding_box() const {
        return m_bounding_box;
    }

    // Set the bounding box
    __host__ __device__ __forceinline__ void set_bounding_box(float min_x,
        float min_y, float max_x, float max_y) {
        m_bounding_box.set(min_x, min_y, max_x, max_y);
    }

    // The number of points in the tree
    __host__ __device__ __forceinline__ int num_points() const {
        return m_end - m_begin;
    }

    // The range of points in the tree
    __host__ __device__ __forceinline__ int points_begin() const {
        return m_begin;
    }

    __host__ __device__ __forceinline__ int po  ints_end() const {
        return m_end;
    }

    // Define the range for that node
    __host__ __device__ __forceinline__ void set_range(int begin, int end) {
        m_begin = begin;
        m_end = end;
    }
};

// Algorithm parameters
struct Parameters {
    // Choose the right set of points to use as in/out
    int point_selector;
    // The number of nodes at a given level (2^k for level k)
    int num_nodes_at_this_le vel;
    // The recursion depth
    int depth;
    // The max value for depth
    const int max_depth;
    // The minimum number of points in a node to stop recursion
    const int min_points_per_node;

    // Constructor set to default values.
    __host__ __device__ Parameters(int max_depth, int min_points_per_node) :
        point_selector(0),
        num_nodes_at_this_level(1),
        depth(0),
        max_depth(max_depth),
        min_points_per_node(min_points_per_node) {}

```

FIGURE A13.14: Support code for quadtree with dynamic parallelism: definitions and main function.

```
    // Copy constructor. Changes the values for next iteration
    __host__ __device__ Parameters(const Parameters &params, bool) :
        point_selector((params.point_selector+1) % 2),
        num_nodes_at_this_level(4*params.num_nodes_at_this_level),
        depth(params.depth+1),
        max_depth(params.max_depth),
        min_points_per_node(params.min_points_per_node) {}
};

// Main function
void main(int argc, char **argv) {

    // Constants to control the algorithm
    const int num_points = atoi(argv[0]);
    const int max_depth  = atoi(argv[1]);
    const int min_points_per_node = atoi(argv[2]);

    // Allocate memory for points
    thrust::device_vector<float> x_d0(num_points);
    thrust::device_vector<float> x_d1(num_points);
    thrust::device_vector<float> y_d0(num_points);
    thrust::device_vector<float> y_d1(num_points);

    // Generate random points
    Random_generator rnd;
    thrust::generate(
        thrust::make_zip_iterator(thrust::make_tuple(x_d0.begin(), y_d0.begin())),
        thrust::make_zip_iterator(thrust::make_tuple(x_d0.end(), y_d0.end())),
        rnd);

    // Host structures to analyze the device ones
    Points points_init[2];
    points_init[0].set(thrust::raw_pointer_cast(&x_d0[0]),
                       thrust::raw_pointer_cast(&y_d0[0]));
    points_init[1].set(thrust::raw_pointer_cast(&x_d1[0]),
                       thrust::raw_pointer_cast(&y_d1[0]));

    // Allocate memory to store points
    Points *points;
    cudaMalloc((void **) &points, 2*sizeof(Points));
    cudaMemcpy(points, points_init, 2*sizeof(Points), cudaMemcpyHostToDevice);

    // We could use a close form...
    int max_nodes = 0;

    for (int i=0, num_nodes_at_level=1 ; i<max_depth ; ++i, num_nodes_at_level*=4)
        max_nodes += num_nodes_at_level;

    // Allocate memory to store the tree
    Quadtree_node root;
    root.set_range(0, num_points);
    Quadtree_node *nodes;
    cudaMalloc((void **) &nodes, max_nodes*sizeof(Quadtree_node));
    cudaMemcpy(nodes, &root, sizeof(Quadtree_node), cudaMemcpyHostToDevice);

    // We set the recursion limit for CDP to max_depth
    cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth, max depth);

    // Build the quadtree
    Parameters params(max_depth, min_points_per_node);
    const int NUM_THREADS_PER_BLOCK = 128;
    const size_t smem_size = 8*sizeof(int);
    build_quadtree_kernel<<<1, NUM_THREADS_PER_BLOCK, smem_size>>>
        (nodes, points, params);
    cudaGetLastError();

    // Free memory
    cudaFree(nodes);
    cudaFree(points);

}
```

FIGURE A13.14: (Continued)

```
//Some inline vector math functions
__forceinline__ __device__ float2 operator+(float2 a, float2 b) {
    float2 c;
    c.x = a.x + b.x;    c.y = a.y + b.y;
    return c;
}

__forceinline__ __device__ float2 operator -(float2 a, float2 b) {
    float2 c;
    c.x = a.x - b.x;    c.y = a.y - b.y;
    return c;
}

__forceinline__ __device__ float2 operator*(float a, float2 b) {
    float2 c;
    c.x = a * b.x;    c.y = a * b.y;
    return c;
}

__forceinline__ __device__ float length(float2 a) {
    return sqrtf(a.x*a.x + a.y*a.y);
}

//Device function that computes the curvature of a line
__device__ float computeCurvature(BezierLine *bLines){
    int bidx = blockIdx.x;
    float curvature = length(bLines[bidx].CP[1] - 0.5f*(bLines[bidx].CP[0]
        + bLines[bidx].CP[2]))/length(bLines[bidx].CP[2]
        - bLines[bidx].CP[0]);
    return curvature;
}

void initializeBLines(BezierLine *bLines_h) {
    //Set initial point to zero (last is last point in the previous segment)
    float2 last = {0,0};
    for(int i = 0; i < N_LINES; i++){
        //Set first point of this line to last point of previous line
        bLines_h[i].CP[0] = last;
        for(int j = 1; j < 3; j++) {
            //Assign random coordinate between 0 and 1
            bLines_h[i].CP[j].x = (float)rand()/(float)RAND_MAX;
            //Assign random coordinate between 0 and 1
            bLines_h[i].CP[j].y = (float)rand()/(float)RAND_MAX;
        }
        last = bLines_h[i].CP[2];    //keep the last point of this line
        //Set number of tessellated vertices to zero
        bLines_h[i].nVertices = 0;
    }
}
```

```
// A structure of 2D points
class Points {
    float *m_x;
    float *m_y;

    public:
    // Constructor
    __host__ __device__ Points() : m_x(NULL), m_y(NULL) {}

    // Constructor
    __host__ __device__ Points(float *x, float *y) : m_x(x), m_y(y) {}

    // Get a point
    __host__ __device__ __forceinline__ float2 get_point(int idx) const {
        return make_float2(m_x[idx], m_y[idx]);
    }

    // Set a point
    __host__ __device__ __forceinline__ voi  d set_point(int idx, const float2 &p) {
        m_x[idx] = p.x;
        m_y[idx] = p.y;
    }

```

```
        // Set the pointers
        __host__ __device__ __forceinline__ void set(float *x, float *y) {
            m_x = x;
            m_y = y;
        }
    };

    // A 2D bounding box
    class Bounding_box {
        // Extreme points of the bounding box
        float2 m_p_min;
        float2 m_p_max;

        public:
        // Constructor. Create a unit box
        __host__ __device__ Bounding_box(){
            m_p_min = make_float2(0.0f, 0.0f);
            m_p_max = make_float2(1.0f, 1.0f);
        }

        // Compute the center of the bounding-box
        __host__  __device__ void compute_center(float2 &center) const {
            center.x = 0.5f * (m_p_min.x + m_p_max.x);
            center.y = 0.5f * (m_p_min.y + m_p_max.y);
        }

        // The points of the box
        __host__ __device_ _ __forceinline__ const float2 &get_max() const {
            return m_p_max;
        }

        __host__ __device__ __forceinline__ const float2 &get_min() const {
            return m_p_min;
        }

        // Does a box contain a point
        __host__ __device__ bool contains(const float2 &p) const {
            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;
        }

        // Define the bounding box
        __host__  __device__ void set(float min_x, float min_y, float max_x, float max_y){
            m_p_min.x = min_x;
            m_p_min.y = min_y;
            m_p_max.x = max_x;
            m_p_max.y = max_y;
        }
    };
```