CSE 599 I
Accelerated Computing - Programming GPUS

CUDA Dynamic Parallelism

Objective

- Introduce dynamic parallelism, a relatively recent CUDA technique in which kernels launch kernels
- Learn about various rules and restrictions that apply to dynamic parallelism
- Study some prototypical applications of dynamic parallelism

What is Dynamic Parallelism

An extension to the CUDA programming model which allows a thread to launch another grid of threads executing another kernel

First introduced with the Kepler architecture (2012)

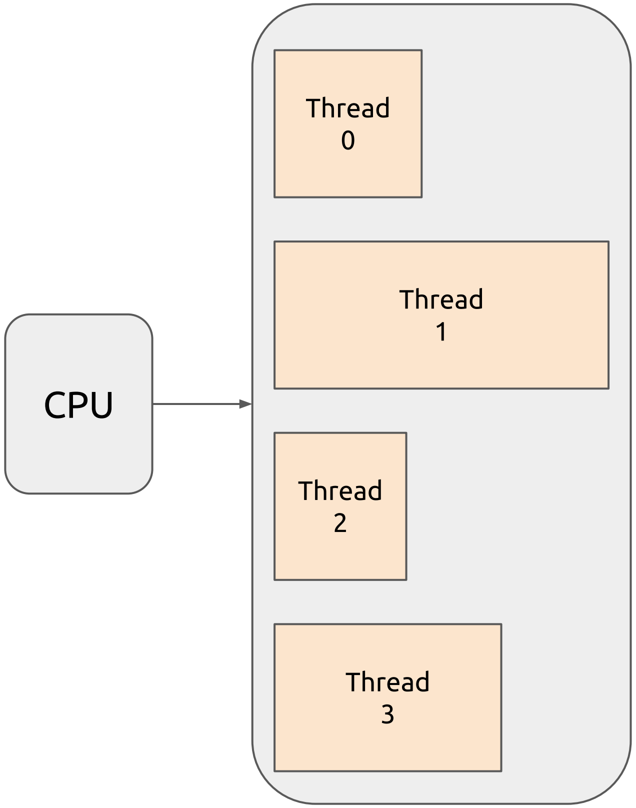
Uses for Dynamic Parallelism

- Recursive algorithms
- Processing at different levels of detail for different parts of the input (i.e. irregular grid structure)
- Algorithms in which new work is "uncovered" along the way

Work Discovery Without Dynamic Parallelism

```
__global__ void workDiscoveryKernel(const int * starts, const int * ends, float * data) {

    int i = threadIdx.x + blockDim.x * blockIdx.x;

    for (int j = starts[i]; j < ends[i]; ++j) {
        process(data[j]);
    }

}
```

Work Discovery

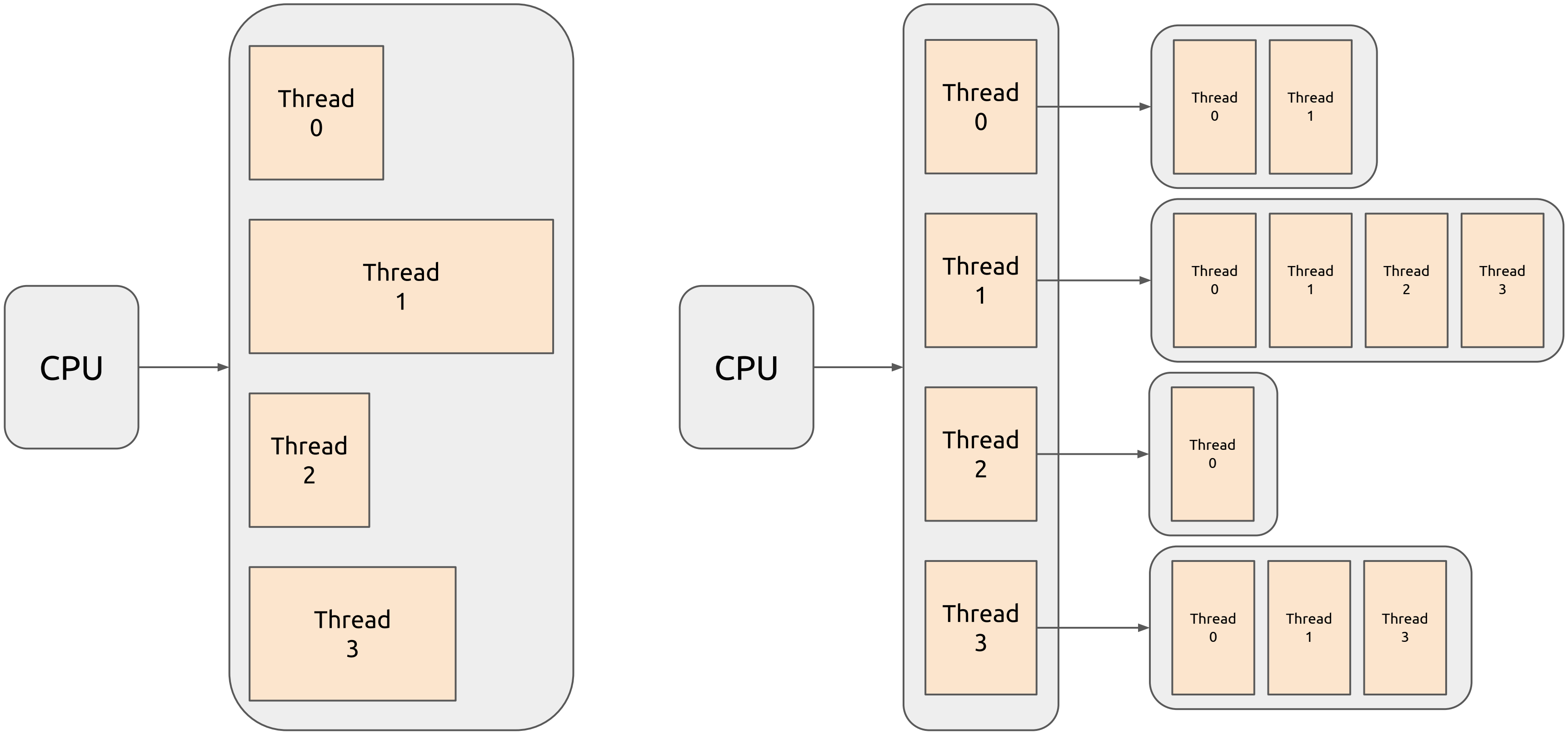


Without dynamic parallelism

Work Discovery With Dynamic Parallelism

```
__global__ void workDiscoveryKernel(const int * starts, const int * ends, float * data) {

     int i = threadIdx.x + blockDim.x * blockIdx.x;

     const int N = ends[i] - starts[i];

     workDiscoveryChildKernel<<<(N-1)/128+1,128>>>(data + starts[i], N);

}

__global__ void workDiscoveryChildKernel(float * data, const int N) {

     int j = threadIdx.x + blockDim.x * blockIdx.x;

     if (j < N) {
          process(data[j]);
     }

}
```

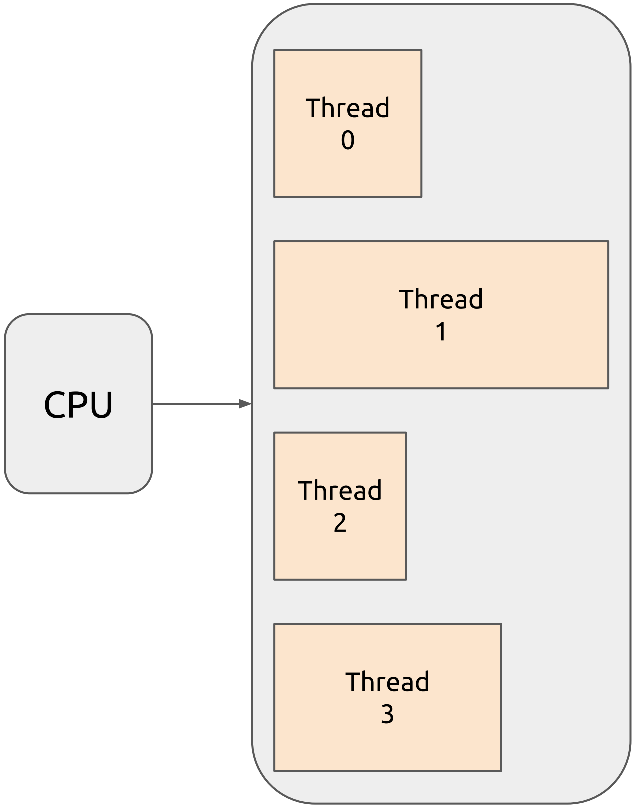
Work Discovery



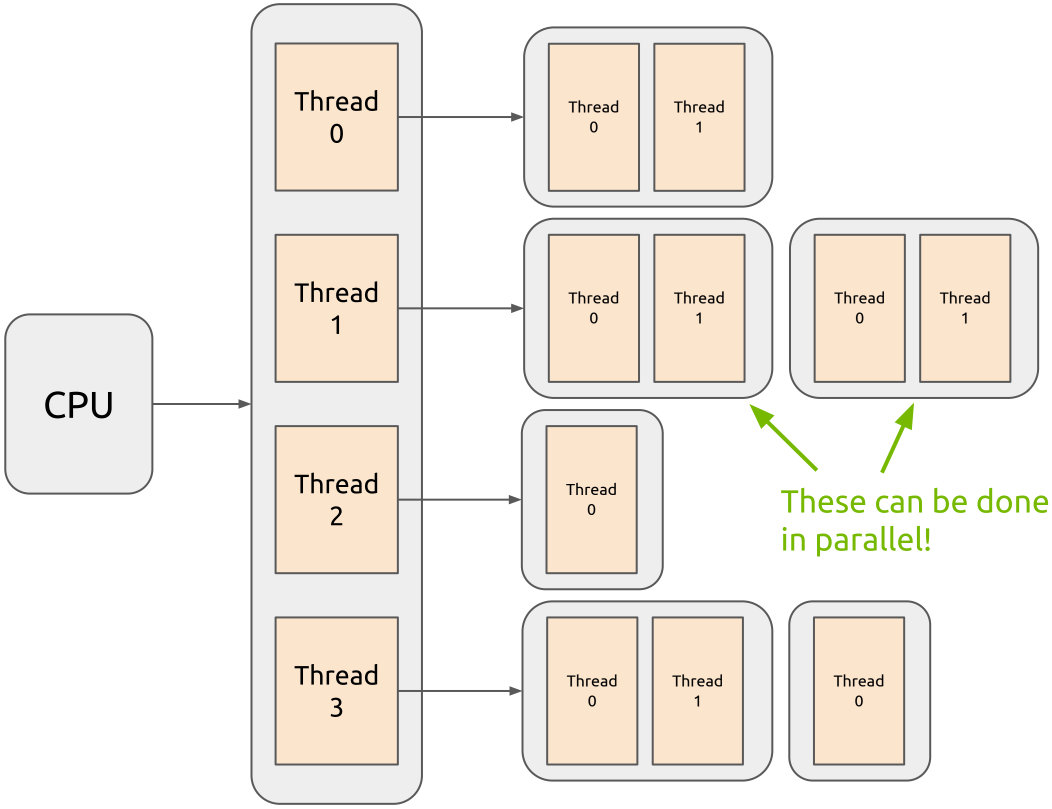
Without dynamic parallelism

With dynamic parallelism

Work Discovery



Without dynamic parallelism



With dynamic parallelism

Global Memory and Dynamic Parallelism

Parent and child grids have two points of guaranteed global memory consistency:

1. When the child grid is launched by the parent; all memory operations performed by the parent thread before launching the child are visible to the child grid when it starts
2. When the child grid finishes; all memory operations by any thread in the child grid are visible to the parent thread once the parent thread has synchronized with the completed child grid

Constant Memory and Dynamic Parallelism

Constant memory also cannot be changed from within a child grid or before launching a child grid

Thus, all constant memory must be set on the host before launching the parent kernel and remain constant for the duration of the entire kernel tree

Local Memory and Dynamic Parallelism

Local memory is private to a thread, and dynamic parallelism is not exception

Child grids have no privileged access to the parent thread's local data

Not OK

```
__global__ void badParentKernel() {

	float data[10];
	childKernel<<<...>>>(data);

}
```

```
__global__ void badParentKernel() {

	float value;
	childKernel<<<...>>>(&value);

}
```

OK

```
__global__ void goodParentKernel(float * data)
{

	childKernel<<<...>>>(data);

}
```

```
__device__ float value;
__global__ void goodParentKernel(float * data)
{

	childKernel<<<...>>>(&value);

}
```

Shared Memory and Dynamic Parallelism

Shared memory is private to a block of threads, and dynamic parallelism is no exception

Parent threads have no privileged access to a child block's shared memory

Memory Allocation from within a Kernel

In addition to kernel launches, dynamic parallelism allows memory allocation from within a kernel via `cudaMalloc()` and `cudaFree()`

A few differences about allocating memory from within a kernel:

- Cannot allocate zero-copy memory
- The allocation limit is the device malloc heap size, which may be smaller than the total device memory size
  - You can get or set this limit using `cudaDevice[Get/Set]Limit()` with the parameter `cudaLimitMallocHeapSize`
- Memory allocated with `cudaMalloc()` inside a kernel must be freed with `cudaFree()` from inside a kernel, and a kernel cannot call `cudaFree()` with a pointer that was allocated on the host

Kernels All the Way Down

A kernel launched from within a kernel can launch a kernel, which can also launch a kernel, etc.

The total “nesting depth” allowed with dynamic parallelism is limited to 24

There are other limits that tend to come up before the maximum nesting depth

Dynamic Parallelism with Multiple GPUs

Kernels launched from within a kernel cannot be executed on another GPU

Pending Launch Pool

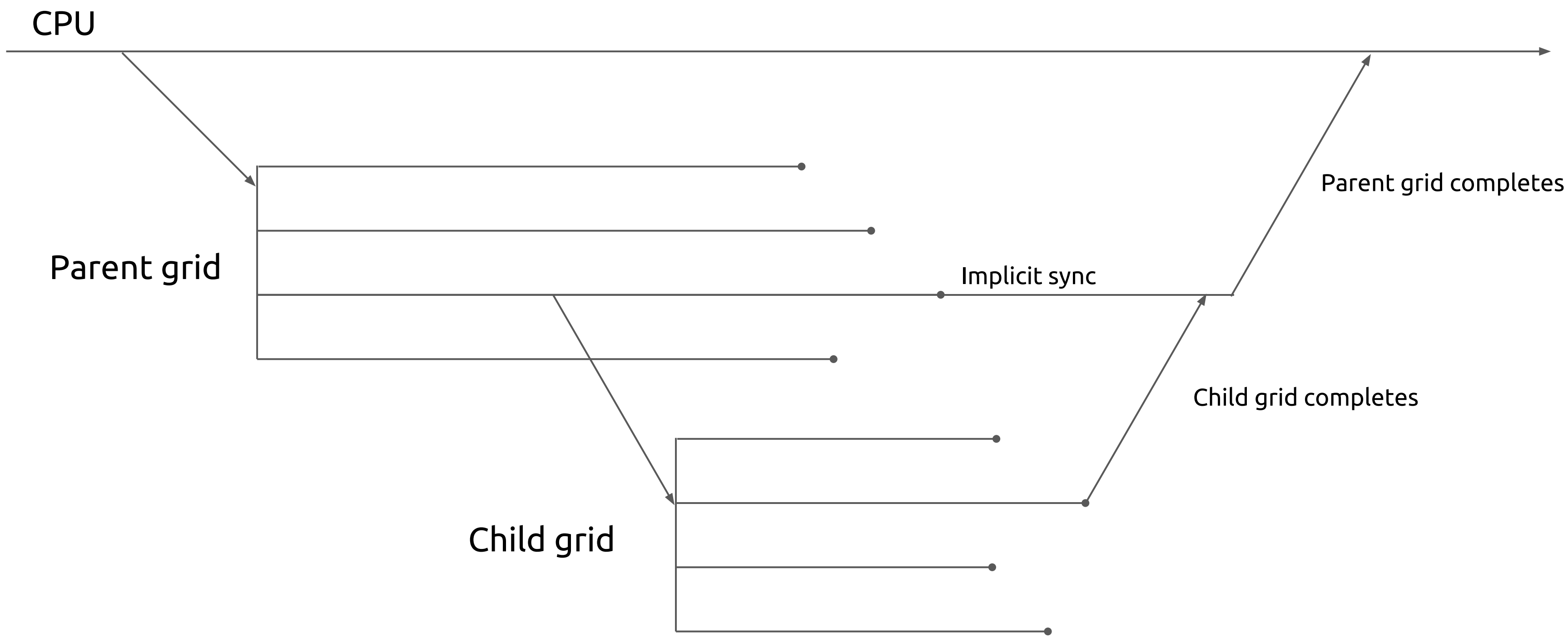
The pending launch pool is a buffer that keeps track of kernels that are currently being executed or waiting to be executed

By default, the pending launch pool has room for 2048 kernels before spilling into a virtualized pool, which is very slow

Like the device malloc heap size, this limit can be queried or set using `cudaDevice[Get/Set]Limit()`, this time with parameter `cudaLimitDevRuntimePendingLaunchCount`

Implicit Synchronization

A parent thread is implicitly synchronized with its children before terminating



Explicit Synchronization

A parent thread can also explicitly synchronize with child grids using `cudaDeviceSynchronize()`

This blocks the *calling thread* on all child grids created by all threads in the block

Blocking all threads can be done by calling `cudaDeviceSynchronize()` from all threads or following a call by one thread with `__syncthreads()`

Synchronization Depth

A parent kernel that performs explicit synchronization on a child grid may be swapped out while waiting for the child grid to finish

This requires storing the entire state of the kernel, i.e. registers, shared memory, program counters, etc.

The deepest nesting level at which synchronization is performed is referred to as the *synchronization depth*

Synchronization depth is limited by the size of the backing store, which can be checked or set using `cudaDevice[Get/Set]Limit()` and the parameter `cudaLimitDevRuntimeSyncDepth`

Streams and Dynamic Parallelism

- Kernels can launch new kernels in both the default and non-default streams to be executed concurrently

- Child kernels launched in explicit streams must use streams that were allocated from within the kernel that launched them

- The scope of a stream is a block; there can be no sharing of streams between host and device, between blocks, or between parent and child

Streams and Dynamic Parallelism

- If no stream is specified, the default stream is used, serializing all kernels launched in the same block (even by different threads)

- `cudaStreamSynchronize()` cannot be called by device code; `cudaDeviceSynchronize()` must be used to wait for all child grids luanched by the block

- All device streams must be non-blocking. To force awareness of this on the programmer, streams created by the device must use `cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking)`

Events and Dynamic Parallelism

Events also have some support in device code, but not the full functionality

Currently, only `cudaStreamWaitEvent()` is allowed to be called from a kernel (no timing or event synchronization)

Events are scoped to the block (like streams)

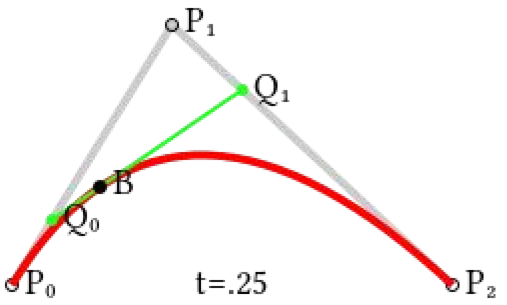
Events consume device memory, so there is no limit, but too many events risks reduced concurrency

Example: Drawing Bezier Curves

A Bezier curve is a smooth curved defined by a set of *n* control points, where *n* determines the degree of the curve

For *n* = 3, the curve is a quadratic Bezier curve defined by control points P_0, P_1, and P_2, and the following equation:

$$B(t) = (1-t)^2 P_0 + 2(1-t)t P_1 + t^2 P_2$$



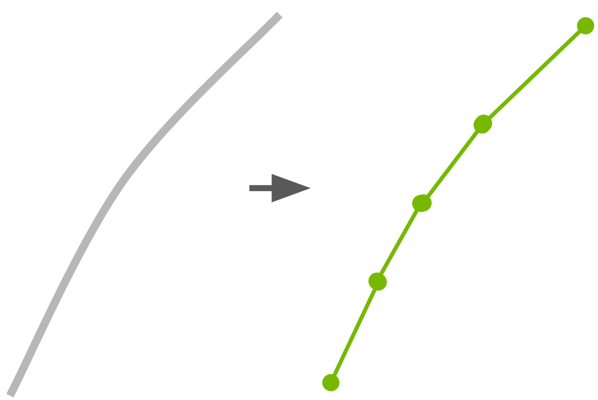
Example: Drawing Bezier Curves

A Bezier curve is defined over a continuous domain

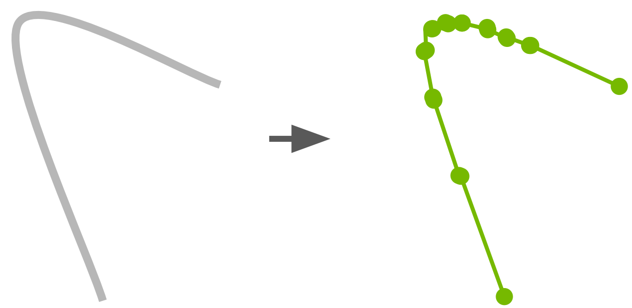
We'll be looking at a kernel to compute a set of discrete points along a user-defined Bezier curve

To make the curve look smooth, we'll want to compute more points in high-curvature regions

Low curvature:



High curvature:



Example: Drawing Bezier Curves

```
#define MAX_NUM_POINTS 128

struct BezierCurve {
     float2 controlPoints[3];
     float2 vertices[MAX_NUM_POINTS];
     int numVertices;
};

__device__ float computeCurvature(const BezierCurve * curve) {
     return length(curve->controlPoints[1] - 0.5*(curve->controlPoints[0] +
                   curve->controlPoints[2])) / length(curve->controlPoints[2] -
                   curve->controlPoints[0]);
}
```

We'll be given a curves with `controlPoints` set, and we want to compute `vertices`

Example: Drawing Bezier Curves

```
__global__ void computeBezierCurvesKernel(BezierCurve * curves, const int N) {

      if (blockIdx.x < N) {

            const float curvature = computeCurvature(&curves[blockIdx.x]);

            // compute number of points based on curvature, between 4 and MAX_NUM_POINTS
            const int nVertices = min(max((int)(curvature * 64.f),4,MAX_NUM_POINTS);
            curves[blockIdx.x].numVertices = nVertices;

            for (int p = threadIdx.x; p < nVertices; p += blockDim.x) {

                  const float t = p / (float)(nVertices - 1);

                  const float oneMinusT = 1.f - t;

                  float2 position = oneMinusT * oneMinusT * curves[blockIdx.x].controlPoints[0] +
                                    2.f * t * oneMinusT * curves[blockIdx.x].controlPoints[1] +
                                    t * t * curves[blockIdx.x].controlPoints[2];

                  curves[blockIdx.x].vertices[p] = position;

            }

      }

}
```

Example: Drawing Bezier Curves

```
#define MAX_NUM_POINTS 128

struct BezierCurve {
     float2 controlPoints[3];
     float2 * vertices;
     int numVertices;
};
```

With dynamic parallelism, we won't need to statically declare the size of the `vertices` buffer

Example: Drawing Bezier Curves

```
__global__ void computeBezierCurvesParentKernel(BezierCurve * curves, const int N) {

      const int i = threadIdx.x + blockDim.x * blockIdx.x;

      if (i < N) {

            const float curvature = computeCurvature(&curves[i]);

            // compute number of points based on curvature, between 4 and MAX_NUM_POINTS
            curves[i].numVertices = min(max((int)(curvature * 64.f),4,MAX_NUM_POINTS);
            cudaMalloc(&curves[i].vertices, curves[i].numVertices * sizeof(float2));

            cudaStream_t stream;
            cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);

            computeBezierCurvesChildKernel<<<(curves[i].numVertices-1)/32+1,32,0,stream>>>(&curves[i]);

            cudaStreamDestroy(stream);

      }

}
```

Example: Drawing Bezier Curves

```
__global__ void computeBezierCurveChildKernel(BezierCurve * curve) {

      const int p = threadIdx.x + blockDim.x * blockIdx.x;

      if (p < curve->numVertices) {

            const float t = p / (float)(nVertices - 1);

            const float oneMinusT = 1.f - t;

            float2 position = oneMinusT * oneMinusT * curves[blockIdx.x].controlPoints[0] +
                              2.f * t * oneMinusT * curves[blockIdx.x].controlPoints[1] +
                              t * t * curves[blockIdx.x].controlPoints[2];

            curve->vertices[p] = position;

      }

}
```

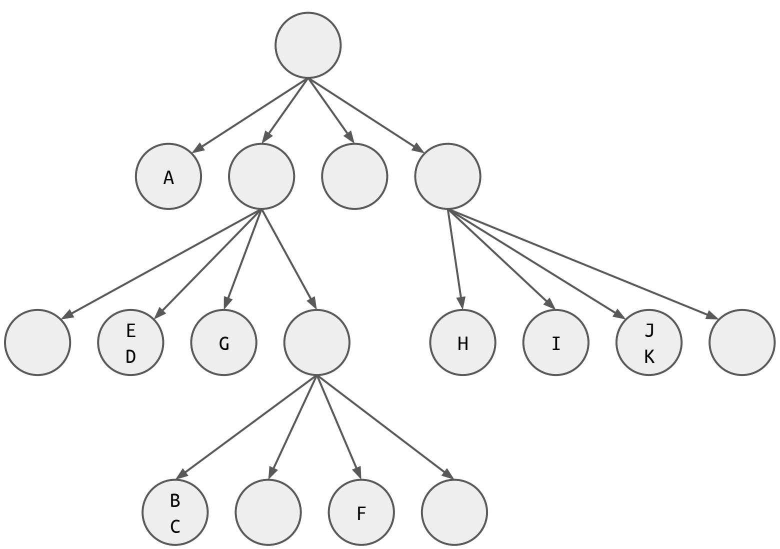
Example: Drawing Bezier Curves

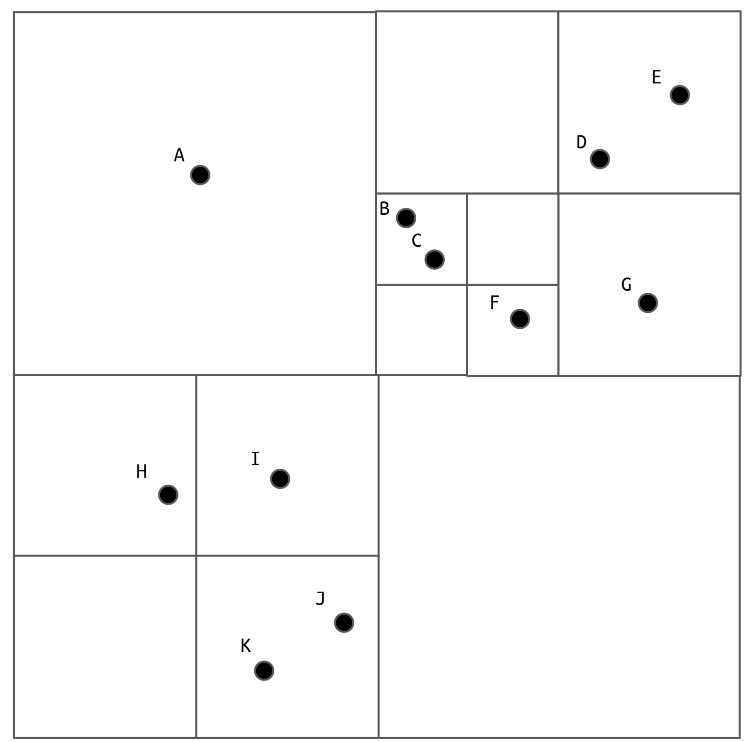
```
__global__ void cleanupKernel(BezierCurve * curves, const int N) {
      const int i = threadIdx.x + blockDim.x * blockIdx.x;
      if (i < N) {
            cudaFree(curves[i]->vertices);
      }
}
```

Recursive Example: Quadtrees

A quadtree is a tree specially designed for storing 2D points

Each node represents a square in the plane, and has exactly 4 children, each representing a quadrant of the square

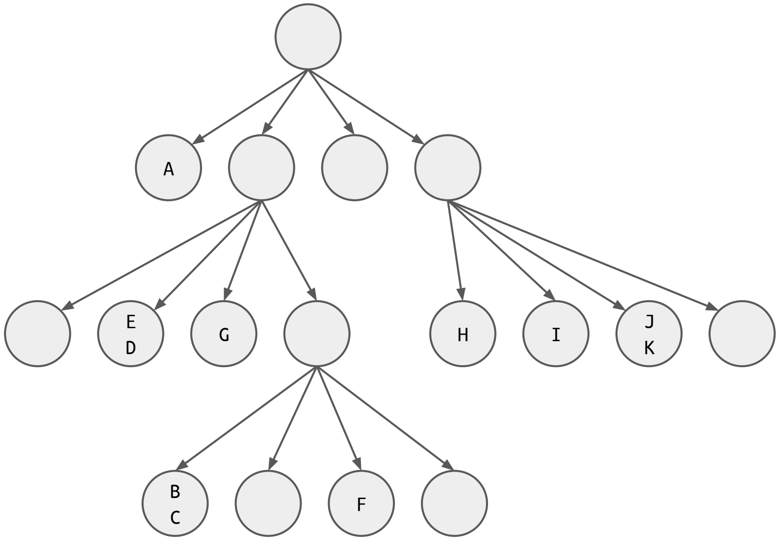


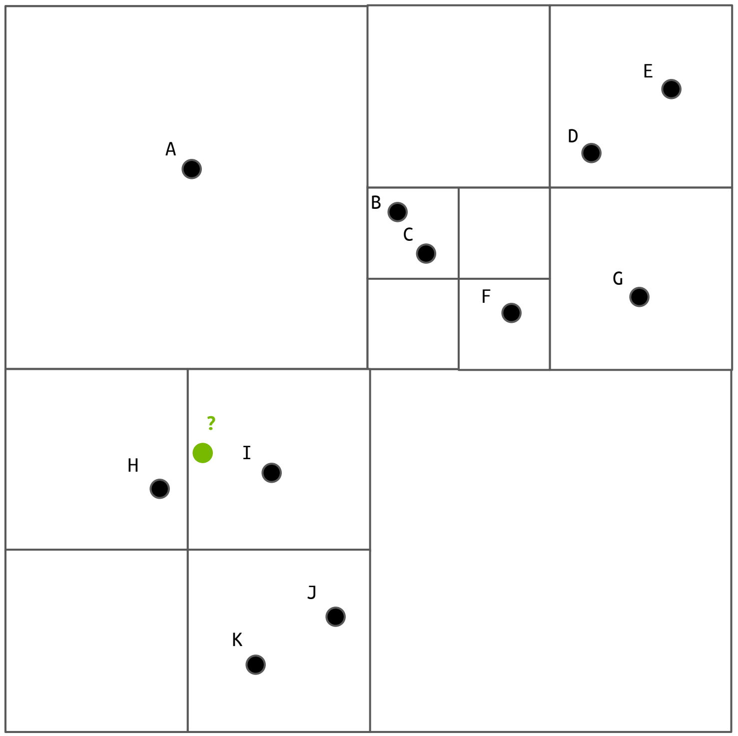


Recursive Example: Quadtrees

A quadtree is a tree specially designed for storing 2D points

Each node represents a square in the plane, and has exactly 4 children, each representing a quadrant of the square

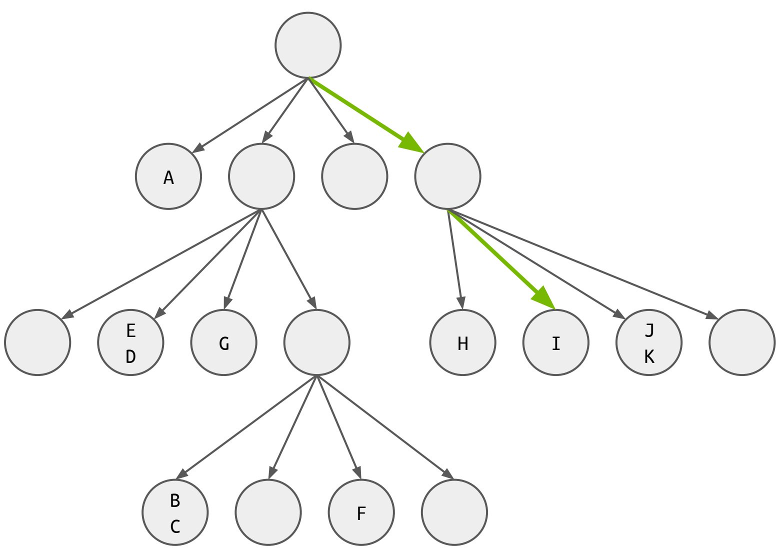


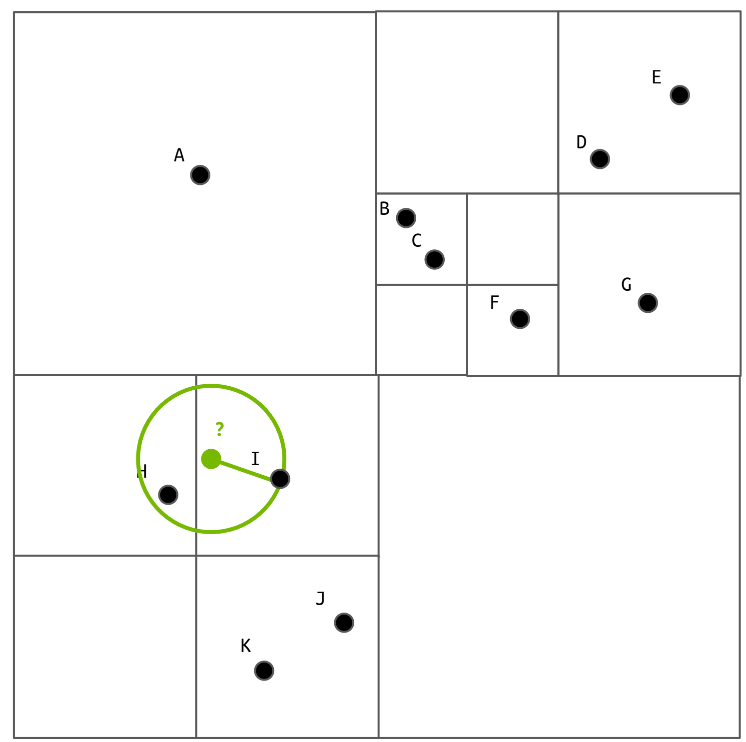


Recursive Example: Quadtrees

A quadtree is a tree specially designed for storing 2D points

Each node represents a square in the plane, and has exactly 4 children, each representing a quadrant of the square

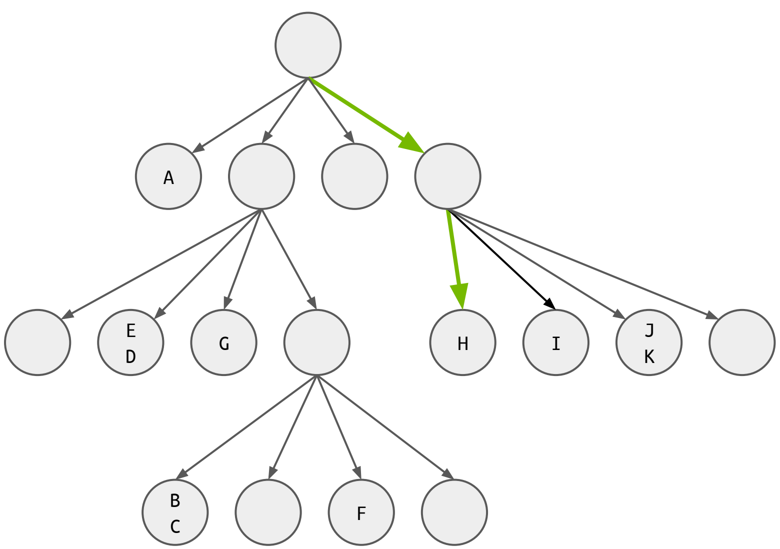


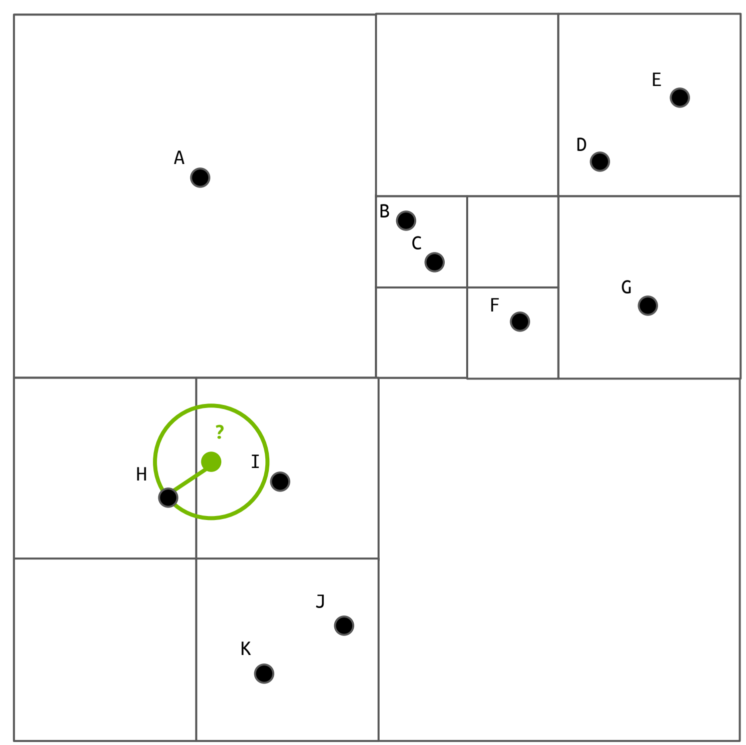


Recursive Example: Quadtrees

A quadtree is a tree specially designed for storing 2D points

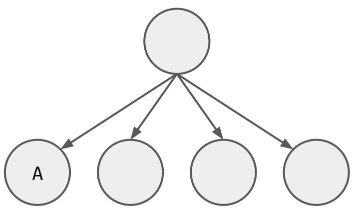
Each node represents a square in the plane, and has exactly 4 children, each representing a quadrant of the square

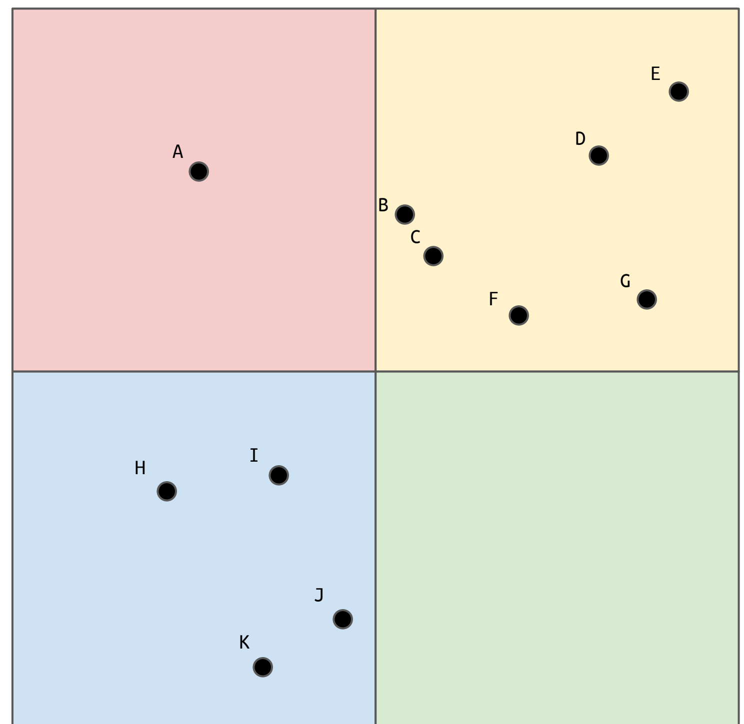




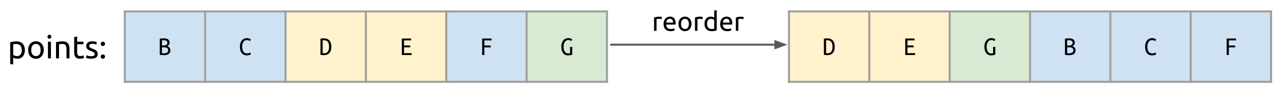
Recursive Example: Quadtrees

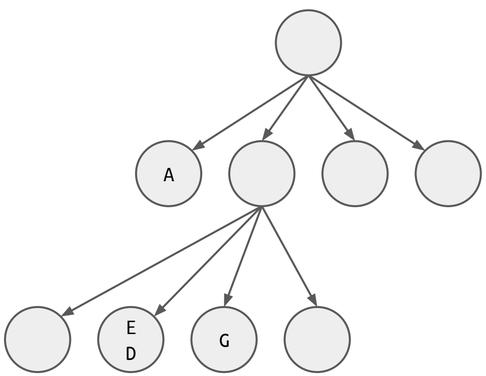
points:	A	B	C	D	E	F	G	H	I	J	K

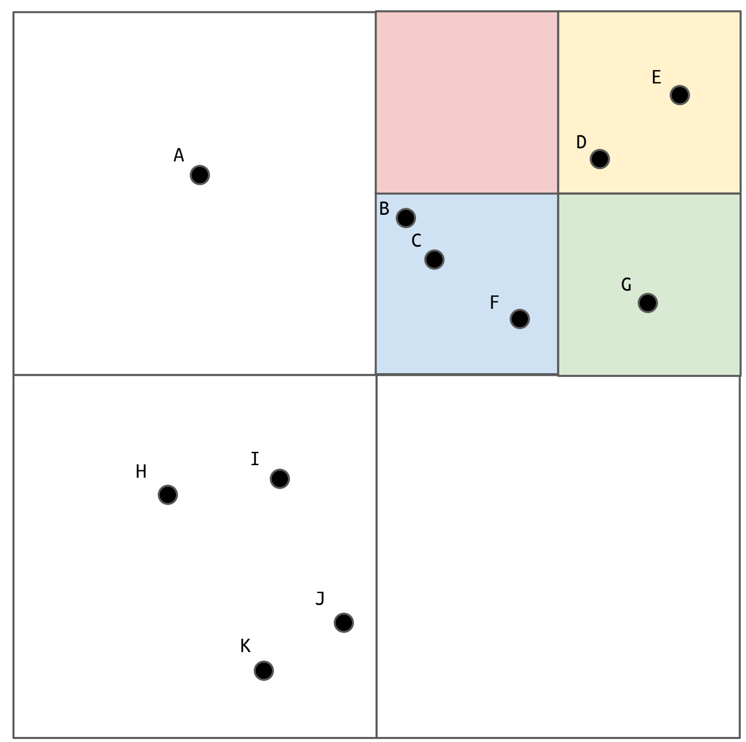




Recursive Example: Quadtrees

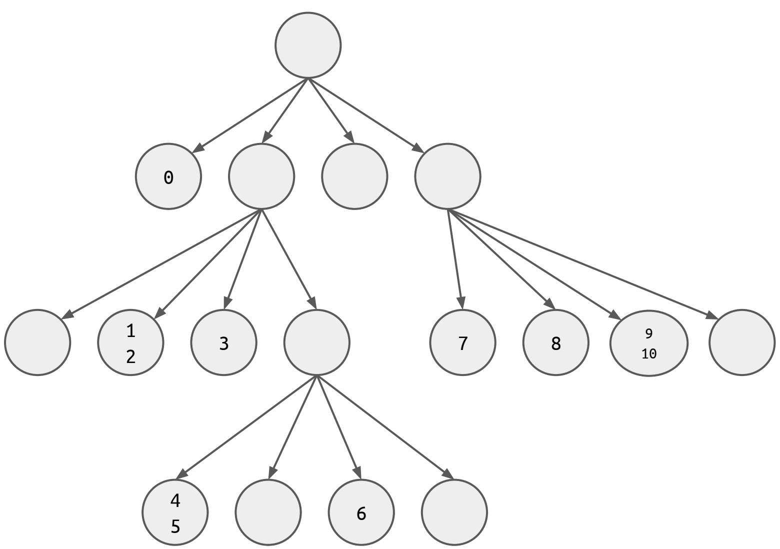


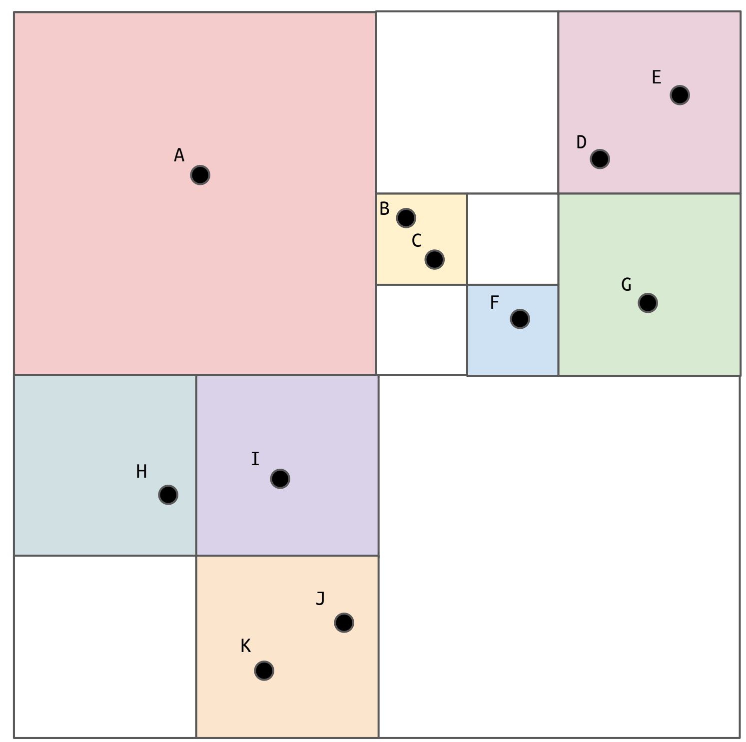




Recursive Example: Quadtrees

A	D	E	G	B	C	F	H	I	J	K





Recursive Example: Quadtrees

```
__global__ void buildQuadtreeKernel(QuadtreeNode * nodes, float2 * pointsA, float2 * pointsB, Parameters params) {

	__shared__ int smem[8];

	QuadtreeNode & node = nodes[blockIdx.x];
	const int numPoints = node.numPoints;

	// recursive base case
	if (numPoints < params.pointThreshold || node.depth() > params.maxDepth) return;

	const BoundingBox & bbox = node.boundingBox;
	const float2 center = bbox.center();

	const int pointsStart = node.pointsStart;
	const int pointsEnd = node.pointsEnd;

	// compute number of points for each child and store result in shared memory
	countPointsInChildNodes(pointsA + pointsStart, pointsEnd - pointsStart, center, smem);

	// do a scan on the number of points for each child to compute offsets
	scanForOffsets(smem);

	// move the points
	reorderPoints(pointsA + pointsStart, pointsB + pointsStart, pointsEnd - pointsStart, center, smem);

	if (threadIdx.x == blockDim.x - 1) {
		cudaMalloc(&node.children, 4 * sizeof(QuadtreeNode)); // allocate memory for the four children
		prepareChildren(node, smem); // set bounding boxes, etc. for the children
		buildQuadtreeKernel<<<4, blockDim.x>>>(node.children, pointsB, pointsA, params);
	}

}
```

Recursive Algorithms in CUDA Before 2012

Technically, recursion has always been possible

However, it required awkward loop unrolling

Essentially, one had to implement a call stack within the kernel

Conclusion / Takeaways

- Dynamic parallelism is a powerful new tool allowing kernels to perform recursive functions and dynamically redistribute work for better load balancing

Sources

https://www.wikipedia.org/

Kirk, David B., and W. Hwu Wen-Mei. Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, 2016.