CHAPTER

CUDA dynamic parallelism 13

Juan Gómez-Luna and Izzat El Hajj

CHAPTER OUTLINE

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00013-3

CUDA Dynamic Parallelism is an extension to the CUDA programming model enabling a CUDA kernel to create new thread grids by launching new kernels. Dynamic parallelism is introduced with the Kepler architecture, first appearing in the GK110 chip. In previous CUDA systems, kernels can only be launched from the host code. Algorithms that involved recursion, irregular loop structures, time-space variation, or other constructs that do not fit a flat, single level of parallelism needed to be implemented with multiple kernel launches, which increase burden on the host, amount of host-device communication, and total execution time. In some cases, programmers resort to loop serialization and other awkward techniques to support these algorithmic needs at the cost of software maintainability. The dynamic parallelism support allows algorithms that dynamically discover new work to prepare and launch kernels without burdening the host or impacting software maintainability. This chapter describes the extended capabilities of the CUDA architecture which enable dynamic parallelism, including the modifications and additions to the CUDA programming model, as well as guidelines and best practices for exploiting this added capacity.

13.1 BACKGROUND

Many real-world applications employ algorithms that either have variation of work across space or dynamically varying amount of work performed over time. As we saw in Chapter 12 Parallel Patterns: Graph Search, in a graph search, the amount of work done when processing each frontier vertex can vary dramatically in graphs like social networks. For another example, Fig. 13.1 shows a turbulence simulation example where the level of required modeling details varies across both space and time. As the combustion flow moves from left to right, the level of activities and intensity increases. The level of details required to model the right side of the model is much higher than that for the left side of the model. On one hand, using a fixed fine grid would incur too much work for no gain for the left side of the model. On the other hand, using a fixed coarse grid would sacrifice too much accuracy for the right side of the model. Ideally, one should use fine grids for the parts of the model that require more details and coarse grids for those that do not.

Previous CUDA systems require all kernels to be launched from host code. The amount of work done by a thread grid is pre-determined during kernel launch. With the SPMD programming style for the kernel code, it is tedious if not extremely difficult to have thread-blocks to use different grid spacing. This limitation favors the use of fixed grid system. In order to achieve the desired accuracy, such fixed grid

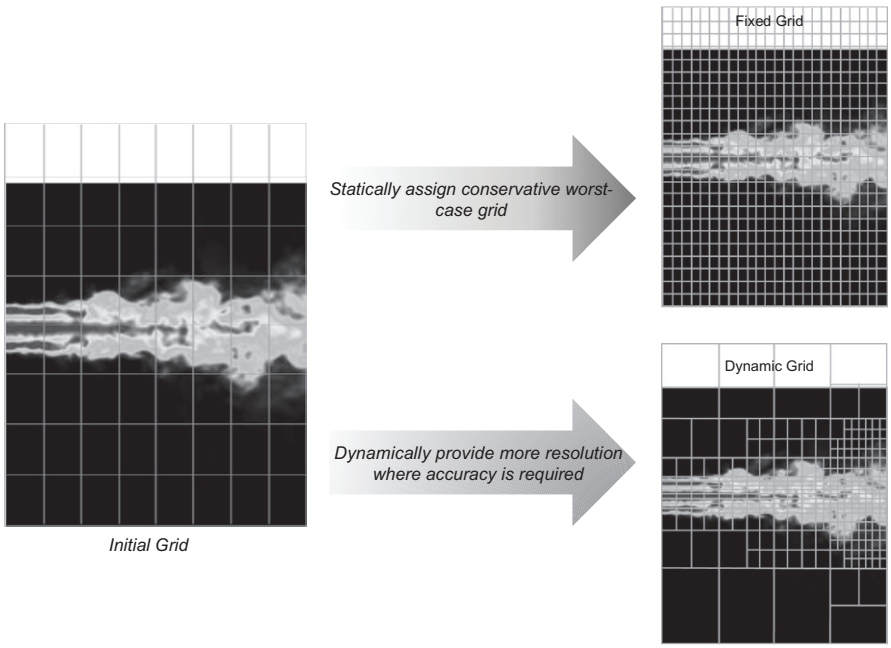

FIGURE 13.1

Fixed versus dynamic grids for a turbulence simulation model.

approach, as illustrated in the upper right portion of Fig. 13.1, typically needs to accommodate the most demanding parts of the model and perform unnecessary extra work in parts that do not require as much detail.

A more desirable approach is shown as the dynamic, variable grid in the lower right portion of Fig. 13.1. As the simulation algorithm detects fast changing simulation quantities in some areas of the model, it refines the grid in those areas to achieve desired level of accuracy. Such refinement does not need to be done for the areas that do not exhibit such intensive activity. This way, the algorithm can dynamically direct more computation work to the areas of the model that benefit from the addition work.

Fig. 13.2 shows a conceptual comparison of behavior between a system without dynamic parallelism and one with dynamic parallelism with respect to the simulation model in Fig. 13.1. Without dynamic parallelism, the host code must launch all kernels. If new work is discovered, such as refining the grid of an area of the model during the execution of a kernel, it needs to terminate itself, report back to the host code and have the host code to launch a new kernel. This is illustrated in Fig. 13.2(A), where the host launches a wave of kernels, receives information from these kernels after their termination, and launches the next level of kernels for any new work discovered by the completed kernels.

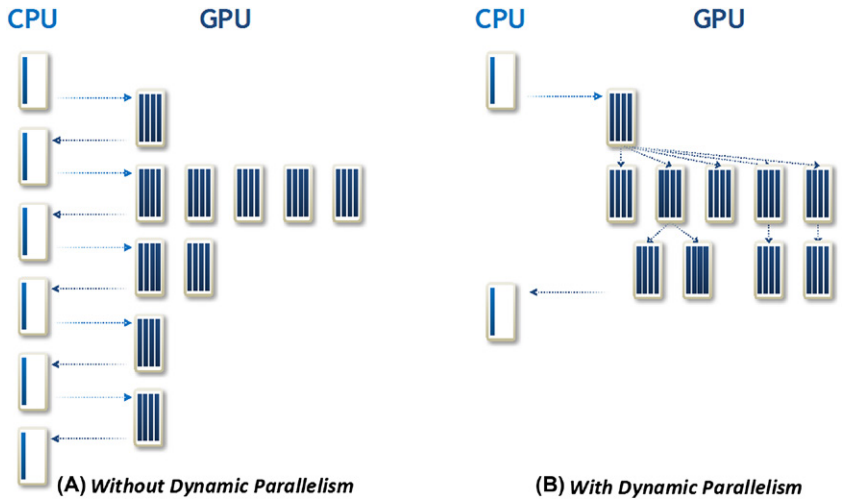

FIGURE 13.2

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

Fig. 13.2(B) shows that with dynamic parallelism, the threads that discover new work can just go ahead and launch kernels to do the work. In our example, when a thread discovers that an area of the model needs to be refined, it can launch a kernel to perform the computation step on the refined grid area without the overhead of terminating the kernel, reporting back to the host, and having the host to launch new kernels.

13.2 DYNAMIC PARALLELISM OVERVIEW

From the perspective of programmers, dynamic parallelism means that they can write a kernel launch statement in a kernel. In Fig. 13.3, the main function (host code) launches three kernels, A, B, and C. These are kernel launches in the original CUDA model. What is different is that one of the kernels, B, launches three kernels X, Y, and Z. This would have been illegal in previous CUDA systems.

The syntax for launching a kernel from a kernel is the same as that for launching a kernel from host code:

```
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, which is in addition to the statically allocated shared memory. Ns is an optional argument that defaults to 0.

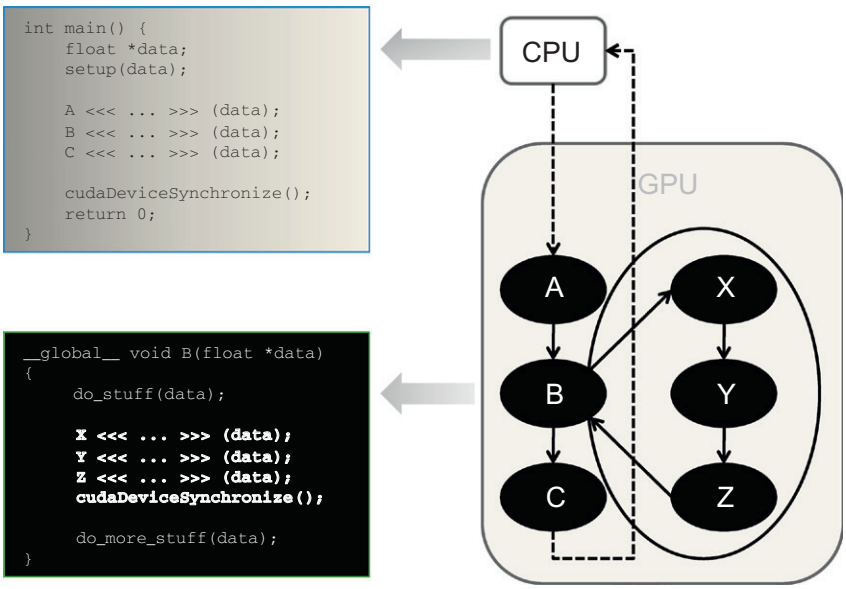

FIGURE 13.3

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

- S is of type cudaStream_t and specifies the stream associated with this call. The stream must have been **allocated in the same thread-block where the call is being made**. S is an optional argument that defaults to 0. Streams are discussed in more detail in Chapter 18.

13.3 A SIMPLE EXAMPLE

In this section, we provide a simple example of coding in each of two styles – first, in the original CUDA style, and second, in the dynamic parallelism style. The example is based on a hypothetical parallel algorithm that does not compute useful results, but provides a conceptually simple computational pattern that recurs in many applications. It serves to illustrate the difference between the two styles and how one can use the dynamic parallelism style to extract more parallelism while reducing control flow divergence when the amount of work done by each thread in an algorithm can vary dynamically.

Fig. 13.4 shows a simple example kernel coded without dynamic parallelism. In this example, each thread of the kernel performs some computation (line 05) then loops over a list of data elements it is responsible for (line 07), and performs another computation for each data element (line 08).

This computation pattern recurs frequently in many applications. For example, in graph search, each thread could visit a vertex then loop over a list of neighboring

```
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    }
```

FIGURE 13.4

A simple example of a hypothetical parallel algorithm coded in CUDA without dynamic 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    }
```

FIGURE 13.5

A revised example using CUDA dynamic parallelism.

vertices. The reader should find this kernel structure very similar to that of `BFS_Bqueue_kernel` in Fig. 12.8. In sparse matrix computations, each thread could first identify the starting location of a row of non-zero elements and loop over the non-zero values. In simulations such as the example in the beginning of the chapter, each thread could represent a coarse grid element and loop over finer grid elements.

There are two main problems with writing applications this way. First, if the work in the loop (lines 07-09) can be profitably performed in parallel, then we have missed out on an opportunity to extract more parallelism from the application. Second, if the number of iterations in the loop varies significantly between threads in the same warp, then the resulting control divergence can degrade the performance of the program.

Fig. 13.5 shows a version of the same program that uses dynamic parallelism. In this version, the original kernel is separated into two kernels, a parent kernel and

a child kernel. The parent kernel starts off the same as the original kernel, executed by a grid of threads referred to as the parent grid. Instead of executing the loop it launches a child kernel to continue the work (lines 07–08). The child kernel is then executed by another grid of threads called the child grid that performs the work that was originally performed inside the loop body (line 18).

Writing the program in this way addresses both problems that were mentioned about the original code. First, the loop iterations are now executed in parallel by the child kernel threads instead of serially by the original kernel thread. Thus, we have extracted more parallelism from the program. Second, each thread now executes a single loop iteration which results in better load balance and eliminates control divergence. Although these two goals could have been achieved by the programmer by rewriting the kernels differently, such manual transformations can be awkward, complicated and error prone. Dynamic parallelism provides an easy way to express such computational patterns.

13.4 MEMORY DATA VISIBILITY

In the next three sections, we will briefly explain some important details that govern the execution behavior of programs that use dynamic parallelism. It is important for a programmer to understand these details in order to use dynamic parallelism confidently. We will cover the rules for memory data visibility in this section. These rules specify how the data objects of a parent grid can be accessed by threads in a child grid. These rules are extensions to the data consistency rules between threads from the same grid vs. between threads from different grids in a non-dynamic-parallelism program. For example, the global memory data written by threads in a grid are not guaranteed to be visible to other threads until either an explicit memory fence or kernel termination. Such rules are extended in dynamic parallelism so that one can clearly understand how a parent and a child can make data values visible to each other.

GLOBAL MEMORY

A parent thread and its child grid can make their global memory data visible to each other, with weak consistency guarantees between child and parent. The memory views of the parent thread and the child grid are said to be consistent with each other if the effects of their memory operations are fully visible to each other. There are two points in the execution of a child grid when its view of memory is consistent with the parent thread:

1. When the child grid is created by a parent thread. That means that all global memory operations in the parent thread prior to invoking the child grid are visible to the child grid.
2. When the child grid completes as signaled by the completion of a synchronization API call in the parent thread. That means all memory operations of the child grid are visible to the parent after the parent has synchronized on the child grid's completion (see Section 13.6 for details about synchronization).

ZERO-COPY MEMORY

Zero-copy system memory has identical consistency guarantees as global memory, and follows the same semantics as detailed above. A kernel may not allocate or free zero-copy memory, however, but may use pointers passed in from the host code.

CONSTANT MEMORY

Constants may not be written to by a kernel, even between dynamic parallelism kernel launches. That is, the value of all __constant__ variables must be set from the host prior to launch of the first kernel. Constant memory variables are globally visible to all kernels, and so must remain constant for the lifetime of the entire dynamic parallelism launch tree invoked by the host code.

Taking the address of a constant memory object from within a thread has the same semantics as for non-dynamic-parallelism programs, and passing that pointer from parent to child or from a child to parent is fully supported.

LOCAL MEMORY

Local memory is private storage for a thread, and is not visible outside of that thread. It is illegal to pass a pointer to local memory as a launch argument when launching a child kernel. The result of dereferencing such a local memory pointer from a child is undefined. For example the following is illegal, with undefined behavior if `x_array` is accessed by any threads that execute the `child_launch` kernel:

```
int x_array[10];       // Creates x_array in parent's local memory
child_launch<<< 1, 1 >>>(x_array);
```

It is sometimes difficult for a programmer to know when a variable is placed into local memory by the compiler. As a general rule, all storage passed to a child kernel should be allocated explicitly from the global-memory heap, either with malloc() or new() or by declaring __device__ storage at global scope. For example, Fig. 13.5(A) shows a valid kernel launch where a pointer to a global memory variable is passed as an argument into the child kernel. Fig. 13.5(B) shows an invalid code where a pointer to a local memory (auto) variable is passed into the child kernel.

The NVIDIA CUDA C compiler will issue a warning if it detects that a pointer to local memory is being passed as an argument to a kernel launch. However, such detections are not guaranteed (Figure 13.6).

```
__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.

SHARED MEMORY

Shared memory is private storage for an executing thread-block, and data is not visible outside of that thread-block. Passing a pointer to a shared-memory variable to a child kernel either through memory or as an argument will result in undefined behavior.

TEXTURE MEMORY

Texture memory accesses (read-only) are performed on a memory region that may be aliased to the global memory region. Texture memory has identical consistency guarantees as global memory, and follows the same semantics. In particular, writes to memory prior to a child kernel launch are reflected in texture memory accesses of the child. Also, writes to memory by a child will be reflected in the texture memory accesses by a parent, after the parent synchronizes on the child's completion.

Concurrent texture memory access and writes to global memory objects which alias the texture memory objects between a parent and its children or between multiple children will result in undefined behavior.

13.5 CONFIGURATIONS AND MEMORY MANAGEMENT

Dynamic parallelism allows a CUDA thread to play the role of host code in launching kernels. There are two other types of major host code activities that support the kernel launch: configuring the device hardware and prepare the device memory for executing the kernel. A programmer also needs to understand how these activities are applied to the kernels launched by a CUDA thread.

LAUNCH ENVIRONMENT CONFIGURATION

A kernel launched with dynamic parallelism inherits all device configuration settings from its parent kernel. Such configuration settings include shared memory and L1 cache size as returned from cudaDeviceGetCacheConfig() and device execution parameter limits as returned from cudaDeviceGetLimit(). For example, if a parent kernel is configured with 16K bytes of shared memory and 48K bytes of L1 cache, then the child kernel it launches will have identical configurations. Likewise, a parent's device limits such as stack size will be passed as-is to its children.

MEMORY ALLOCATION AND LIFETIME

Dynamic parallelism makes it possible to invoke `cudaMalloc` and `cudaFree` from kernels. However they have slightly modified semantics. Within the device environment the total allocatable memory is limited to the device malloc() heap size, which may be smaller than the available unused device memory. Moreover, it is an error to invoke cudaFree from the host program on a pointer which was allocated

by `cudaMalloc` on the device, or to invoke cudaFree from the device program on a pointer which was allocated by cudaMalloc on the host. These limitations may be removed in future versions of CUDA.

	`cudaMalloc()` on Host	`cudaMalloc()` on Device
`cudaFree()` on Host	Supported	Not supported
`cudaFree()` on Device	Not supported	Supported
`Allocation limit`	Free device memory	cudaLimitMallocHeapSize

NESTING DEPTH

Kernels launched with dynamic parallelism may themselves launch other kernels, which may in turn launch other kernels, and so on. Each subordinate launch is considered a new "nesting level," and the total number of levels is the "nesting depth" of the program. The maximum nesting depth is limited in hardware to 24.

In the presence of parent-child synchronization, there are additional constraints on nesting depth due to the amount of memory required by the system to store parent kernel state. These constraints will be discussed in Section 13.6 when we discuss *synchronization depth*.

PENDING LAUNCH POOL CONFIGURATION

The pending launch pool is a buffer that tracks the kernels that are executing or waiting to be executed. This pool is allocated a fixed amount of space, thereby supporting a fixed number of pending kernel launches (2048 by default). If this number is exceeded, a virtualized pool is used, but leads to significant slowdown which can be an order of magnitude or more. To avoid this slowdown, the programmer can increase the size of the fixed pool by executing the cudaDeviceSetLimit() API call from the host function to set the cudaLimitDevRuntimePendingLaunchCount configuration.

ERRORS AND LAUNCH FAILURES

Like CUDA API function calls in host code, any CUDA API function called within a kernel may return an error code. Any failed kernel launch due to reasons such as insufficient execution resources also appears to return with an error code. The last error code returned is also recorded and may be retrieved via the `cudaGetLastError()` call. Errors are recorded on a per-thread basis, so that each thread can identify the most recent error that it has generated. The error code is of type cudaError_t, which is a 32-bit integer value.[1]

[1] No notification of ECC errors is available to code within a CUDA kernel. ECC errors are only reported at the host side. Any ECC errors which arise during execution of a dynamic parallelism kernel will either generate an exception or continue execution (depending upon error and configuration).

13.6 SYNCHRONIZATION, STREAMS, AND EVENTS

SYNCHRONIZATION

As with kernel launches from the host, kernel launches from the device are non-blocking. If a parent thread wants to wait for a child kernel to complete before proceeding, it must perform synchronization explicitly.

One way for a parent thread to perform synchronization with its child kernels on the device is by invoking `cudaDeviceSynchronize()`. A thread that invokes this call will wait until **all kernels launched by any thread in the thread-block have completed**. However, this does not mean that all threads in the block will wait, so if a block-wide synchronization is desired, then `cudaDeviceSynchronize()` invoked by one thread in the block must also be followed by `__syncthreads()` invoked by all threads in the block. Synchronization can also be performed on streams within the same thread-block (which will be discussed shortly).

If a parent kernel launches other child kernels and does not explicitly synchronize on the completion of those kernels, then the runtime will perform the synchronization implicitly **before the parent kernel terminates**. This ensures that the parent and child kernels are properly nested, and that no kernel completes before its children have completed. This implicit synchronization is illustrated in Fig. 13.7.

SYNCHRONIZATION DEPTH

If a parent kernel performs explicit synchronization on a child kernel, it may be swapped out of execution while waiting for the child kernel to complete. For this

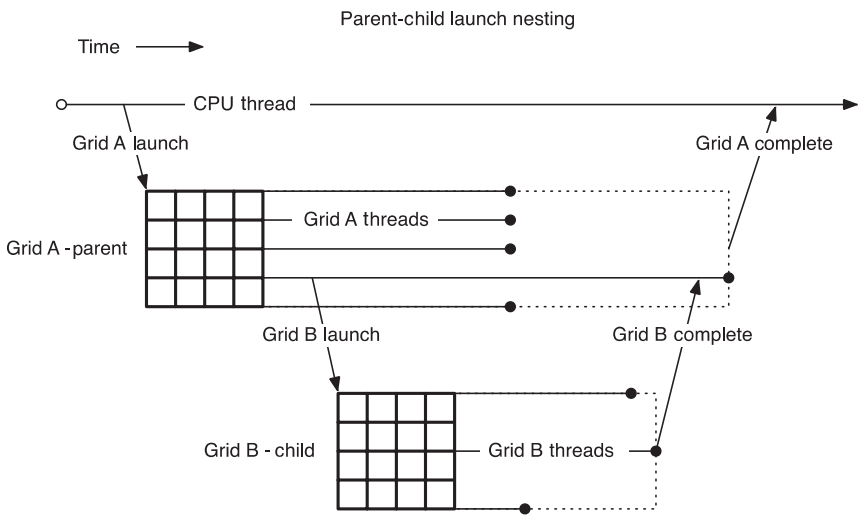

FIGURE 13.7

Completion sequence for parent and child grids.

reason, memory needs to be allocated as a backing-store for the parent kernel state. Ancestors of the synchronizing parent kernel may also be swapped out. Thus the backing store needs to be large enough to fit the state of all kernels up to the deepest nesting level at which synchronization is performed. This deepest nesting level defines the *synchronization depth*.

Conservatively, the amount of memory allocated for the backing store for each level of the synchronization depth must be large enough to support storing state for the maximum number of live threads possible on the device. On current generation devices, this amounts to ~150 MB per level, which will be unavailable for program use even if it is not all consumed. The maximum synchronization depth is thus limited by the amount of memory allocated by the software for the backing store, and is likely to be a more important constraint than the maximum nesting depth stipulated by the hardware.

The default amount of memory reserved for the backing store is sufficient for a synchronization depth of two. However, the programmer can increase this amount using the cudaDeviceSetLimit() API call from the host function to set a larger value for the cudaLimitDevRuntimeSyncDepth configuration parameter.

STREAMS

Just like host code can use streams to execute kernels concurrently, kernel threads can also use streams when launching kernels with dynamic parallelism. Both named and unnamed (NULL) streams can be used.

The scope of a stream is private to the block in which the stream was created. In other words, streams created by a thread may be used by any thread within the same thread-block, but stream handles should not be passed to other blocks or child/parent kernels. Using a stream handle within a block that did not allocate it will result in undefined behavior. Streams created on the host have undefined behavior when used within any kernel, just as streams created by a parent grid have undefined behavior if used within a child grid.

When a stream is not specified to the kernel launch, the default NULL stream in the block is used by all threads. This means that all kernels launched in the same block will be serialized even if they were launched by different threads. However, it is often the case that kernels launched by different threads in a block can be executed concurrently, so programmers must be careful to explicitly use different streams in each thread if they wish to avoid the performance penalty from serialization.

Similar to host-side launch, work launched into separate streams may run concurrently, but actual concurrency is not guaranteed. Programs that require concurrency between child kernels in order to run correctly are ill-formed and will have undefined behavior. An unlimited number of named streams are supported per block, but the maximum concurrency supported by the platform is limited. If more streams are created than can support concurrent execution, some of these may serialize or alias with each other.

The host-side NULL stream's global-synchronization semantic is not supported under dynamic parallelism. To make this difference between the stream behavior on the host-side and the device side with dynamic parallelism explicit, all streams created in a kernel must be created using the cudaStreamCreateWithFlags() API with the cudaStreamNonBlocking flag (an example is shown later in Fig. 13.10). Calls to cudaStreamCreate() from a kernel will fail with a compiler "unrecognized function call" error, so as to make clear the different stream semantic under dynamic parallelism.

The cudaStreamSynchronize() API is not available within a kernel, only cudaDeviceSynchronize() can be used to wait explicitly for launched work to complete. This is because the underlying system software implements only a block-wide synchronization call, and it is undesirable to offer an API with incomplete semantics (that is, the synchronization function guarantees that one stream synchronizes, but coincidentally provides a full barrier as a side-effect). Streams created within a thread-block are implicitly synchronized when all threads in the thread-block exit execution.

EVENTS

Only the inter-stream synchronization capabilities of CUDA events are supported in kernel functions. Events within individual streams are currently not supported in kernel functions. This means that cudaStreamWaitEvent() is supported, but cudaEventSynchronize(), timing with cudaEventElapsedTime(), and event query via cudaEventQuery() are not. *These may be supported in a future version.*[2]

Event objects may be shared between the threads within a block that created them but are local to that block and should not be passed to child/parent kernels. Using an event handle within a block that did not allocate it will result in undefined behavior.

An unlimited number of events are supported per block, but these consume device memory. Owing to resource limitations, if too many events are created (exact number is implementation-dependent), then device-launched grids may attain less concurrency than might be expected. Correct execution is guaranteed, however.

13.7 A MORE COMPLEX EXAMPLE

We now show an example that is a more interesting and useful case of adaptive subdivision of spline curves. This example illustrates a variable number of child kernel launches, according to the workload. The example is to calculate Bezier Curves [Wiki_Bezier], which are frequently used in computer graphics to draw smooth, intuitive curves that are defined by a set of *control points*, which are typically defined by a user.

[2] To ensure that this restriction is clearly seen by the user, dynamic parallelism `cudaEvents` must be created via `cudaEventCreateWithFlags()`, which currently only accepts the `cudaEventDisableTiming` flag value when called from a kernel.

Mathematically, a Bezier curve is defined by a set of control points $\mathbf{P}_0$ through $\mathbf{P}_n$, where n is called its order (n=1 for linear, 2 for quadratic, 3 for cubic, etc.). The first and last control points are always the end points of the curve; however, the intermediate control points (if any) generally do not lie on the curve.

LINEAR BEZIER CURVES

Given two control points $\mathbf{P}_0$ and $\mathbf{P}_1$, a linear Bezier curve is simply a straight line connecting between those two points. The coordinates of the points on the curve is given by the following linear interpolation formula:

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

QUADRATIC BEZIER CURVES

A quadratic Bezier curve is defined by three control points $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$. The points on a quadratic curve are defined as a linear interpolation of corresponding points on the linear Bezier curves from $\mathbf{P}_0$ to $\mathbf{P}_1$ and from $\mathbf{P}_1$ to $\mathbf{P}_2$, respectively. The calculation of the coordinates of points on the curve is expressed in the following formula:

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

which can be simplified into the following formula:

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

BEZIER CURVE CALCULATION (WITHOUT DYNAMIC PARALLELISM)

Fig. 13.8 shows a CUDA C program that calculates the coordinates of points on a Bezier curve. The main function (line 48) initializes a set of control points to random values (line 51[3]). In a real application, these control points are most likely inputs from a user or a file. The control points are part of the `bLines_h` array whose element type `BezierLine` is declared in line 07. The storage for the `bLines_h` array is allocated in line 50. The host code then allocates the corresponding device memory for the `bLines_d` array and copies the initialized data to `bLines_d` (lines 54–56). It then calls the `computeBezierLine()` kernel to calculate the coordinates of the Bezier curve.

The `computeBezierLine()` kernel starting at line 13 is designed to use a thread-block to calculate the curve points for a set of three control points (of the quadratic Bezier formula). Each thread-block first computes a measure of the curvature of the curve defined by the three control points. Intuitively, the larger the curvature, the

[3]Function `initializeBLines()` can be found in Fig. A13.8 in Code Appendix at the end of the chapter.

```
01  #include <stdio.h>
02  #include <cuda.h>
03
04  #define MAX_TESS_POINTS 32
05
06  //A structure containing all parameters needed to tessellate a Bezier line
07  struct BezierLine {
08      float2 CP[3];                       //Control points for the line
09      float2 vertexPos[MAX_TESS_POINTS]; //Vertex position array to tessellate into
10      int nVertices;                      //Number of tessellated vertices
11  };
12
13  __global__ void computeBezierLines(BezierLine *bLines, int nLines) {
14      int bidx = blockIdx.x;
15      if(bidx < nLines){
16          //Compute the curvature of the line
17          float curvature = computeCurvature(bLines);
18
19          //From the curvature, compute the number of tessellation points
20          int nTessPoints = min(max((int)(curvature*16.0f),4),32);
21          bLines[bidx].nVertices = nTessPoints;
22
23          //Loop through vertices to be tessellated, incrementing by blockDim.x
24          for(int inc = 0; inc < nTessPoints; inc += blockDim.x){
25              int idx = inc + threadIdx.x;  //Compute a unique index for this point
26              if(idx < nTessPoints){
27                  float u = (float)idx/(float)(nTessPoints-1);  //Compute u from idx
28                  float omu = 1.0f - u;   //pre-compute one minus u
29                  float B3u[3]; //Compute quadratic Bezier coefficients
30                  B3u[0] = omu*omu;
31                  B3u[1] = 2.0f*u*omu;
32                  B3u[2] = u*u;
33                  float2 position = {0,0};  //Set position to zero
34                  for(int i = 0; i < 3; i++){
35                      //Add the contribution of the i'th control point to position
36                      position = position + B3u[i] * bLines[bidx].CP[i];
37                  }
38                  //Assign value of vertex position to the correct array element
39                  bLines[bidx].vertexPos[idx] = position;
40              }
41          }
42      }
43  }
44
45  #define N_LINES 256
46  #define BLOCK_DIM 32
47
48  int main( int argc, char **argv ) {
49      //Allocate and initialize array of lines in host memory
50      BezierLine *bLines_h = new BezierLine[N_LINES];
51      initializeBLines(bLines_h);
52
53      //Allocate device memory for array of Bezier lines
54      BezierLine *bLines_d;
55      cudaMalloc((void**)&bLines_d, N_LINES*sizeof(BezierLine));
56      cudaMemcpy(bLines_d,bLines_h, N_LINES*sizeof(BezierLine),cudaMemcpyHostToDevice);
57
58      //Call the kernel to tessellate the lines
59      computeBezierLines<<<N_LINES, BLOCK_DIM>>>(bLines_d, N_LINES );
60
61      cudaFree(bLines_d); //Free the array of lines in device memory
62      delete[] bLines_h;  //Free the array of lines in host memory
63  }
```

FIGURE 13.8

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

more the points it takes to draw a smooth quadratic Bezier curve for the three control points. This defines the amount of work to be done by each thread-block. This is reflected in lines 20 and 21, where the total number of points to be calculated by the current thread-block is proportional to the curvature value.

In the `for`-loop in line 24, all threads calculate a consecutive set of Bezier curve points in each iteration. The detailed calculation in the loop body is based on the formula we presented earlier. The key point is that the number of iterations taken by threads in a block can be very different from that taken by threads in another block. Depending on the scheduling policy, such variation of the amount of work done by each thread-block can result in decreased utilization of SMs and thus reduced performance.

BEZIER CURVE CALCULATION (WITH DYNAMIC PARALLELISM)

Fig. 13.9 shows a Bezier curve calculation code using dynamic parallelism. It breaks the `computeBezierLine()` kernel in Fig. 13.8 into two kernels. The first part, `computeBezierLine_parent()`, discovers the amount of work to be done for each control point. The second part, `computeBezierLine_child()`, performs the calculation.

With the new organization, the amount of work done for each set of control points by the `computeBezierLines_parent()` kernel is much smaller than the original `computeBezierLines()` kernel. Therefore, we use one thread to do this work in `computeBezierLines_parent()`, as opposed to using one block in `computeBezierLines()`. In line 58, we only need to launch one thread per set of control points. This is reflected by dividing the `N_LINES` by `BLOCK_DIM` to form the number of blocks in the kernel launch configuration.

There are two key differences between the `computeBezierLines_parent()` kernel and the `computeBezierLines()` kernel. First, the index used to access the control points is formed on a thread basis (line 08 in Fig. 13.9) rather than block basis (line 14 in Fig. 13.8). This is because the work for each control point is done by a thread rather than a block, as we mentioned above. Second, the memory for storing the calculated Bezier curve points is dynamically determined and allocated in line 15 in Fig. 13.9. This allows the code to assign just enough memory to each set of control points in the `BezierLine` type. Note that in Fig. 13.8, each `BezierLine` element is declared with the maximal possible number of points. On the other hand, the declaration in Fig. 13.9 has only a pointer to a dynamically allocated storage. Allowing a kernel to call the `cudaMalloc()` function can lead to substantial reduction of memory usage for situations where the curvature of control points vary significantly.

Once a thread of the `computeBezierLines_parent()` kernel determines the amount of work needed by its set of control points, it launches the `computeBezierLines_child()` kernel to do the work (line 19 in Fig. 13.9). In our example, every thread from the parent grid creates a new grid for its assigned set of control points. This way, the work done by each thread-block is balanced. The amount of work done by each child grid varies.

After the `computeBezierLines_parent()` kernel terminates, the main function can copy the data back and draw the curve on an output device. It also calls a kernel to free all storage allocated to the vertices in the `bLines_d` data structure in parallel (line 61). This is necessary since the vertex storage was allocated on the device by the `computeBezierLines_parent()` kernel so it has to be freed by device code (Section 13.5).

```
01    struct BezierLine {
02        float2 CP[3];      //Control points for the line
03        float2 *vertexPos; //Vertex position array to tessellate into
04        int nVertices;     //Number of tessellated vertices
05    };
06    __global__ void computeBez ierLines_parent(BezierLine *bLines, int nLines) {
07        //Compute a unique index for each Bezier line
08        int lidx = threadIdx.x + blockDim.x*blockIdx.x;
09        if(lidx < nLines){
10            //Compute the curvature of the line
11            float curvature = computeCurvature(bLines);
12
13            //From the curvature, compute the number of tessellation points
14            bLines[lidx].nVertices = min(max((int)(curvature*16.0f),4),MAX_TESS_POINTS);
15            cudaMalloc((void**)&bLines[lidx].vertexPos,
16                bLines[lidx].nVertices*sizeof(float2));
17
18            //Call the child kernel to compute the tessellated points for each line
19            computeBezierLine_child<<<ceil((float)bLines[lidx].nVertices/32.0f), 32>>>
20                (lidx, bLines, bLines[lidx].nVertices);
21        }
22    }
23    __global__ void computeBezierLine_child(int lidx, BezierLine* bLines,
24        int nTessPoints) {
25        int idx = threadIdx.x + blockDim.x*blockIdx.x;//Compute idx unique to this vertex
26        if(idx < nTessPoints){
27            float u = (float)idx/(float)(nTessPoints-1);  //Compute u from idx
28            float omu = 1.0f - u;   //Pre-compute one minus u
29            float B3u[3];   //Compute quadratic Bezier coefficients
30            B3u[0] = omu*omu;
31            B3u[1] = 2.0f*u*omu;
32            B3u[2] = u*u;
33            float2 position = {0,0};  //Set position to zero
34            for(int i = 0; i < 3; i++) {
35                //Add the contribution of the i'th control point to position
36                position = position + B3u[i] * bLines[lidx].CP[i];
37            }
38            //Assign the value of the vertex position to the correct array element
39            bLines[lidx].vertexPos[idx] = position;
40        }
41    }
42    __global__ void freeVertexMem(BezierLine *bLines, int nLines) {
43        //Compute a unique index for each Bezier line
44        int lidx = threadIdx.x + blockDim.x*blockIdx.x;
45        if(lidx < nLines)
46            cudaFree(bLines[lidx].vertexPos);   //Free the vertex memory for this line
47    }
48    int main( int argc, char **argv ) {
49        //Allocate array of lines in host memory
50        BezierLine *bLines_h = new BezierLine[N_LINES];
51        initializeBLines(bLines_h);
52
53        //Allocate device memory for array of Bezier lines
54        BezierLine *bLines_d;
55        cudaMalloc((void**)&bLines_d, N_LINES*sizeof(BezierLine));
56        cudaMemcpy(bLines_d,bLines_h, N_LINES*sizeof(Bezi erLine),cudaMemcpyHostToDevice);
57
58        computeBezierLines_parent<<<ceil((float)N_LINES/(float)BLOCK_DIM), BLOCK_DIM>>>
59            (bLines_d, N_LINES);
60
61        freeVertexMem <<<ceil((float)N_LINES/(float)BLOCK_DIM), BLOCK_DIM>>>
62            (bLines_d, N_LINES);
63        cudaFree(bLines_d);    //Free the array of lines in device memory
64        delete[] bLines_h;     //Free the array of lines in host memory
65    }
```

FIGURE 13.9

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

LAUNCH POOL SIZE

As explained in Section 13.6, the launch pool storage may be virtualized when the fixed-pool size is full. That is, all launched grids will still be queued successfully. However, using the virtualized pool has a higher cost than using the fixed-size pool. The Bezier curve calculation with dynamic parallelism helps us to illustrate this.

Since the default size of the fixed-size pool is 2048 (it can be queried with `cudaDeviceGetLimit()`), launching more than 2048 grids will require the use of the virtualized pool, when the fixed-size pool is full. That is, if `N_LINES` (defined in Fig. 13.8, line 45) is set to 4096, half of the launches will use the virtualized pool. This will incur a significant performance penalty. However, if the fixed-size pool is set to 4096, the execution time will be reduced by an order of magnitude.

As a general recommendation, the size of the fixed-size pool should be set to the number of launched grids (if it exceeds the default size). In the case of the Bezier curves example, we would use `cudaDeviceSetLimit(cudaLimitDevRuntimePendingLaunchCount, N_LINES)` before launching the `computeBezierLines_parent()` kernel (line 58).

STREAMS

Named and unnamed (NULL) streams are offered by the device runtime, as mentioned in Section 13.6. One key consideration is that the default NULL stream is block-scope. This way, by default all launched grids within a thread-block will use the same stream, even if they are launched by different threads. As a consequence, these grids will execute sequentially.

The Bezier example launches as many grids as threads in the `computeBezierLines_parent()` kernel (line 19 in Fig. 13.9). Moreover, since `MAX_TESS_POINTS` is equal to 32 (see Fig. 13.8, line 04) and the thread-block size in `computeBezierLines_child()` is 32, the number of blocks per grid will be 1 for the `computeBezierLines_child()` kernel. If the default NULL stream is used, all these grids with one single block will be serialized. Thus, using the default NULL stream when launching the `computeBezierLines_child()` kernel can result in a drastic reduction in parallelism compared to the original, non-CDP kernel.

Given that `N_LINES` is 256 and `BLOCK_DIM` is 64, only four blocks are launched in `computeBezierLines_parent()`. Thus, only four default streams will be available for the `computeBezierLines_child()` kernel. Consequently, some streaming multiprocessors (SM) will remain unused on any GPU with more than four SMs. Since each grid in the same stream consists of only one thread-block and all grids in the same stream are serialized with respect to each other, each SM can also be underutilized.

If more concurrency is desired (with the aim of better utilizing all SM), named streams must be created and used in each thread. Fig. 13.10 shows the sequence of instructions that should replace line 19 in Fig. 13.9.

```
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.

Using the code in Fig. 13.10, kernels launched from the same thread-block will be in different streams and can run concurrently. This will better utilize all SMs in the situation described above, leading to a considerable reduction of the execution time.

13.8 A RECURSIVE EXAMPLE

Dynamic parallelism allows programmers to implement recursive algorithms. In this section, we illustrate the use of dynamic parallelism for implementing recursion with a quadtree [Quadtree 1974]. Quadtrees partition a two-dimensional space by recursively subdividing it into four quadrants. Each quadrant is considered a node of the quadtree, and contains a number of points. If the number of points in a quadrant is greater than a fixed minimum, the quadrant will be subdivided into four more quadrants, that is, four child nodes.

Fig. 13.11 depicts an overview of how the construction of a quadtree can be implemented with dynamic parallelism. In this implementation one node (quadrant) is assigned to one thread-block. Initially (depth = 0), one thread-block is assigned the entire two-dimensional space (root node), which contains all points. It divides the space into four quadrants, and launches one thread-block for each quadrant (depth = 1). These child blocks will again subdivide their quadrants if they contain more points than a fixed minimum. In this example the minimum is two; thus, blocks 00 and 02 do not launch children. Blocks 01 and 03 launch a kernel with four blocks each.

As the flow graph in the right-hand side of Fig. 13.11 shows, a block first checks if the number of points in its quadrant is greater than the minimum required for further division and the maximum depth has not been reached. If either of the conditions fails, the work for the quadrant is complete and the block returns. Otherwise, the block computes the center of the bounding box that surrounds its quadrant. The center is in the middle of four new quadrants. The number of points in each of them is counted. A four-element scan operation is used to compute the offsets to the locations where the points will be stored. Then, the points are reordered, so that those points in the same quadrant are grouped together and placed into their section of the point storage. Finally, the block launches a child kernel with four thread-blocks, one for each of the four new quadrants.

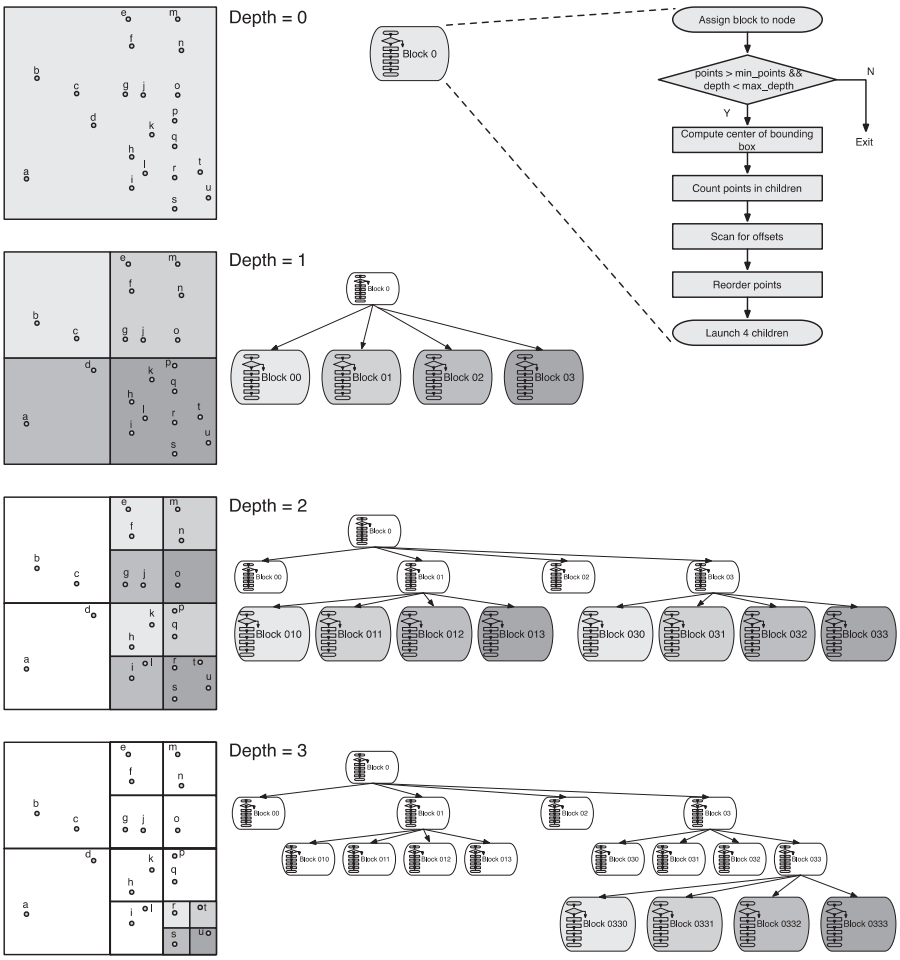

FIGURE 13.11

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.

Fig. 13.12 continues the small example in Fig. 13.11 and illustrates in detail how the points are reordered at each level of depth. In this example, we assume that each quadrant must have a minimum of two points in order to be further divided. The algorithm uses two buffers to store the points and reorder them. The points should be in buffer 0 at the end of the algorithm. Thus, it might be necessary to swap the buffer contents before leaving, in case the points are in buffer 1 when the terminating condition is met.

In the initial kernel launch from the host code (for depth = 0), thread-block 0 is assigned all the points that reside in buffer 0, shown in Fig. 13.12(A). Block 0 further

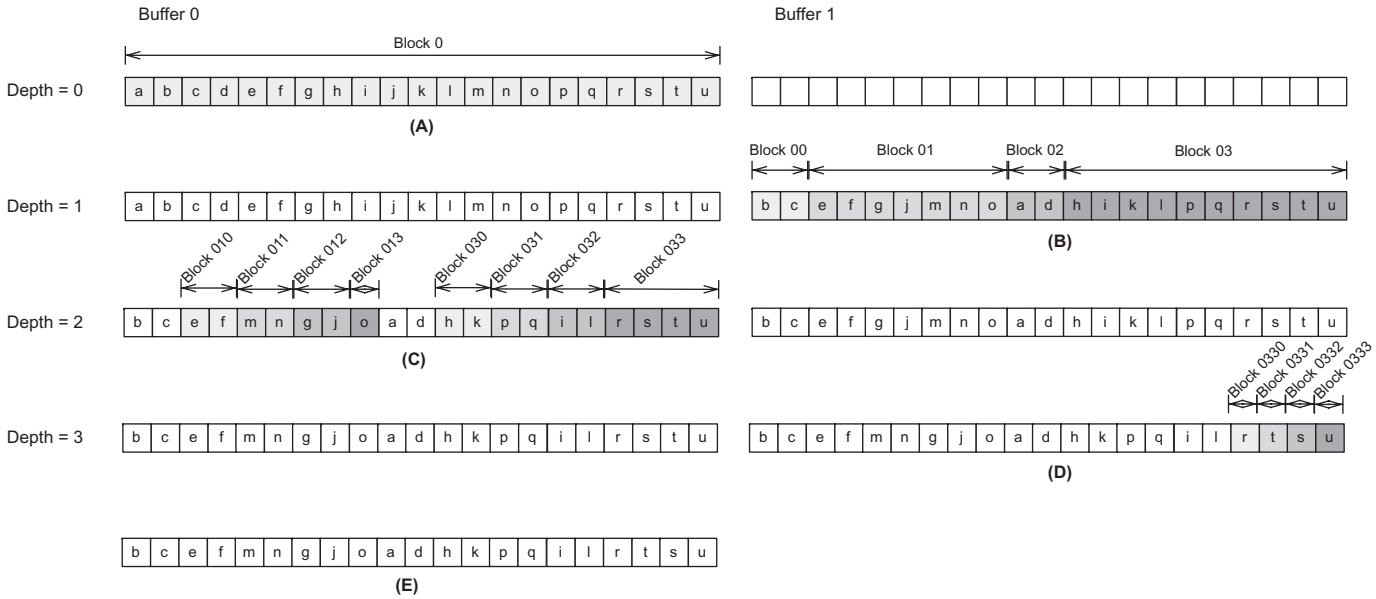

FIGURE 13.12

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

divides the quadrant into four child quadrants, groups together all points in the same child quadrant, and stores them in buffer 1, as shown in Fig. 13.12(B). Its four children, block 00 to block 03, are assigned each of the four new quadrants, shown as marked ranges in Fig. 13.12(B). Blocks 00 and 02 will not launch children, since the number of points in their respective assigned quadrant is only 2. They swap their points to buffer 0. Blocks 01 and 03 reorder their points to group those in the same quadrant, and launch four child blocks each, as shown in Fig. 13.12(C). Blocks 010, 011, 012, 013, 030, 031, and 032 do not launch children (they have 2 or fewer points) nor need to swap points (they are already in buffer 0). Only block 033 reorders its points, and launches four blocks, as shown in Fig. 13.12(D). Blocks 0330 to 0333 will exit after swapping their points to buffer 0, which can be seen in Fig. 13.12(E).

The kernel code in Fig. 13.13 implements the flow graph from Fig. 13.11 in CUDA. The quadtree is implemented with a node array, where each element contains all the pertinent information for one node of the quadtree (definition in Fig. A13.14

```
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      }
46  }
```

FIGURE 13.13

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

in Code Appendix). As the quadtree is constructed, new nodes will be created and placed into the array during the execution of the kernels. The kernel code assumes that the `node` parameter points to the next available location in the node array.

At each level of depth, every block starts by checking the number of points in its node (quadrant). Each point is a pair of floats representing `x` and `y` coordinates (definition in Fig. A13.13 in Code Appendix). If the number of points is less than or equal to the minimum or if the maximum depth is reached (line 11), the block will exit. Before exiting, the block carries out a buffer swap if necessary. This is done in the device function `check_num_points_and_depth()` shown in Fig. 13.14.

If the block doesn't exit, the center of the bounding box is computed (line 17). A bounding box is defined by its top-left and bottom-right corners. The coordinates of the center are computed as the coordinates of the middle point between these two corner points. The definition of a bounding box (including function `compute_center()`) is in Fig. A13.13 in the Code Appendix.

As the center defines the four quadrants, the number of points in each quadrant is counted (line 26). The device function `count_points_in_children()` can be found in Fig. 13.14[4]. The threads of the block collaboratively go through the range of points, and update atomically the counters in shared memory for each quadrant.

The device function `scan_for_offsets()` is called then (line 29). As can be seen in Fig. 13.14, it performs a sequential scan on the four counters in shared memory. Then, it adds the global offset of the parent quadrant to these values to derive the starting offset for each quadrant's group in the buffer.

Using the quadrants' offsets, the points are reordered with `reorder_points()` (line 32). For simplicity, this device function (Fig. 13.14) uses an atomic operation on one of the four quadrant counters to derive the location for placing each point.

Finally, the last thread of the block (line 35) determines the next available location in the node array (line 37), prepares the new node contents for the child quadrants (line 40), and launches one child kernel with four thread-blocks (line 43). The device function `prepare_children()` prepares the new node contents for the children by setting the limits of the children's bounding boxes and the range of points in each quadrant. The `prepare_children()` function can be found in Fig. 13.14.

The rest of the definitions and the main function can be found in Fig. A13.14 in the Code Appendix.

13.9 SUMMARY

CUDA dynamic parallelism extends the CUDA programming model to allow kernels to launch kernels. This allows each thread to dynamically discover work and launch new grids according to the amount of work discovered. It also supports dynamic allocation of device memory by threads. As we show in the Bezier Curve calculation example, these extensions can lead to better work balance across threads and blocks

[4] The device functions in Fig. 13.14 are simplified for clarity.

```
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      }
035      __syncthreads();
036  }
037
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();
071  }
```

FIGURE 13.14

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

```
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 }
```

FIGURE 13.14

(Continued)

as well as more efficient memory usage. CUDA Dynamic Parallelism also helps programmers to implement recursive algorithms, as the quadtree example shows.

Besides ensuring better work balance, dynamic parallelism offers many advantages in terms of programmability. However, it is important to keep in mind that launching grids with a very small number of threads could lead to severe underutilization of the GPU resources. A general recommendation is launching 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.

Similarly, nested parallelism, which can be seen as a form of tree processing, will provide a higher performance when tree nodes are thick (that is, each node deploys many threads), and/or when the branch degree is large (that is, each parent node has many children). As the nesting depth is limited in hardware, only relatively shallow trees can be implemented efficiently.

13.10 EXERCISES

1. **True or False:** Parent and child grids have coherent access to global memory, with weak consistency between child and parent.

2. **True or False:** Zero-copy system memory has no coherence and consistency guarantees between parent and children.

3. **True or False:** Parent kernels can define new __constant__ variables that will be inherited by child kernels.

4. **True or False:** Child kernels can inherit parent's shared and local memories, and coherence is guaranteed.

5. Six (6) blocks of 256 threads run the following parent kernel:

```
__global__ void parent_kernel(int *output, int *input,
int *size) {
    // Thread index
    int idx = threadIdx.x + blockDim.x*blockIdx.x;
    // Number of child blocks
    int numBlocks = size[idx] / blockDim.x;
    // Launch child
    child_kernel<<< numBlocks, blockDim.x >>>(output, input,
    size);
}
```

   How many child kernels could run concurrently?
   a. 1536
   b. 256
   c. 6
   d. 1

6. Choose the right statement for the Bezier example:
   a. If N_LINES = 1024, and BLOCK_DIM = 64, the number of child kernel launches will be 16.
   b. If N_LINES = 1024, the fixed-size pool should be set to 1024 (Note: Default size is 2048).
   c. If N_LINES = 1024, BLOCK_DIM = 64, and per-thread streams are used, a total of 16 streams will be deployed.
   d. If N_LINES = 1024, BLOCK_DIM = 64, and aggregation is used, the number of child kernel launches will be 16.

7. Consider a two-dimensional organization of 64 equidistant points that is classified with a quadtree. What will be the maximum depth of the quadtree (including the root node)?
   a. 21
   b. 4
   c. 64
   d. 16

8. For the same quadtree, what will be the total number of child kernel launches?
   a. 21
   b. 4
   c. 64
   d. 16

REFERENCES

Bezier Curves. <http://en.wikipedia.org/wiki/B%C3%A9zier_curve>.

Finkel, R. A., & Bentley, J. L. (1974). Quad trees: A data structure for retrieval on composite keys. *Acta informatica*, *4*(1), 1–9.

A13.1 CODE APPENDIX

```
01    //Some inline vector math functions
02    __forceinline__ __device__ float2 operator+(float2 a, float2 b) {
03        float2 c;
04        c.x = a.x + b.x;    c.y = a.y + b.y;
05        return c;
06    }
07
08    __forceinline__ __device__ float2 operator -(float2 a, float2 b) {
09        float2 c;
10        c.x = a.x - b.x;    c.y = a.y - b.y;
11        return c;
12    }
13
14    __forceinline__ __device__ float2 operator*(float a, float2 b) {
15        float2 c;
16        c.x = a * b.x;    c.y = a * b.y;
17        return c;
18    }
19
20    __forceinline__ __device__ float length(float2 a) {
21        return sqrtf(a.x*a.x + a.y*a.y);
22    }
23
24    //Device function that computes the curvature of a line
25    __device__ float computeCurvature(BezierLine *bLines){
26        int bidx = blockIdx.x;
27        float curvature = length(bLines[bidx].CP[1] - 0.5f*(bLines[bidx].CP[0]
28            + bLines[bidx].CP[2]))/length(bLines[bidx].CP[2]
29            - bLines[bidx].CP[0]);
30        return curvature;
31    }
32
33    void initializeBLines(BezierLine *bLines_h) {
34        //Set initial point to zero (last is last point in the previous segment)
35        float2 last = {0,0};
36        for(int i = 0; i < N_LINES; i++){
37            //Set first point of this line to last point of previous line
38            bLines_h[i].CP[0] = last;
39            for(int j = 1; j < 3; j++) {
40                //Assign random coordinate between 0 and 1
41                bLines_h[i].CP[j].x = (float)rand()/(float)RAND_MAX;
42                //Assign random coordinate between 0 and 1
43                bLines_h[i].CP[j].y = (float)rand()/(float)RAND_MAX;
44            }
45            last = bLines_h[i].CP[2];    //keep the last point of this line
46            //Set number of tessellated vertices to zero
47            bLines_h[i].nVertices = 0;
48        }
49    }
```

FIGURE A13.8

Support code for Bezier Curve calculation without dynamic parallelism.

```
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;
36
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        }
71    };
```

FIGURE A13.13

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

```
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      }
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) {}
076
```

FIGURE A13.14

Support code for quadtree with dynamic parallelism: definitions 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]),
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);
146
147 }
```

FIGURE A13.14

(Continued)