CHAPTER

CUDA Dynamic Parallelism 20

CHAPTER OUTLINE

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 increases burden on the host and amount of host-device communication. The dynamic parallelism support allows algorithms that dynamically discover new work to prepare and launch kernels without burdening the host. This chapter describes the extended capabilities of the CUDA architecture that enables dynamic parallelism, including the modifications and additions to the CUDA programming model necessary to take advantage of these, as well as guidelines and best practices for exploiting this added capacity.

20.1 BACKGROUND

Many real-world applications employ algorithms that dynamically vary the amount of work performed. For example, Figure 20.1 shows a turbulence simulation example where the level of required modeling details varies across 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 require as many details.

Previous CUDA systems require all kernels to be launched from the host code. The amount of work done by a thread grid is predetermined 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 systems

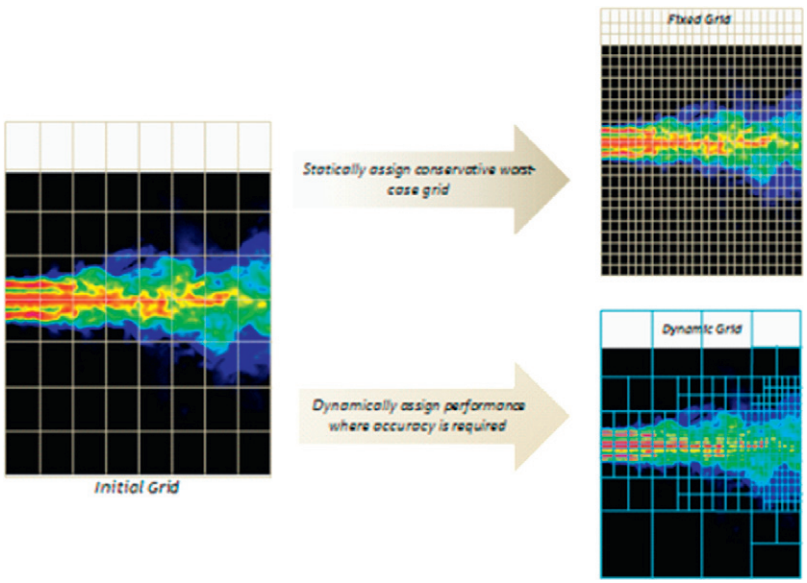

FIGURE 20.1

Fixed versus dynamic grids for a turbulence simulation model.

as we discussed in Chapter 12. To achieve the desired accuracy, such a fixed-grid approach, as illustrated in Figure 20.1, 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 grid in the lower right portion of Figure 20.1. As the simulation algorithm detects fast-changing simulation quantities in some areas of the model, it refines the grid in those areas to achieve the 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 additional work.

Figure 20.2 shows a conceptual comparison between the original CUDA and the dynamic parallelism version with respect to the simulation model in Figure 20.1. 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 report back to the host code and have the host code to launch a new kernel. This is illustrated in Figure 20.2(a), where the host launches a wave of kernels, receives information from these kernels, and launches the next level of kernels for any new work discovered by the completed kernels.

Figure 20.2(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

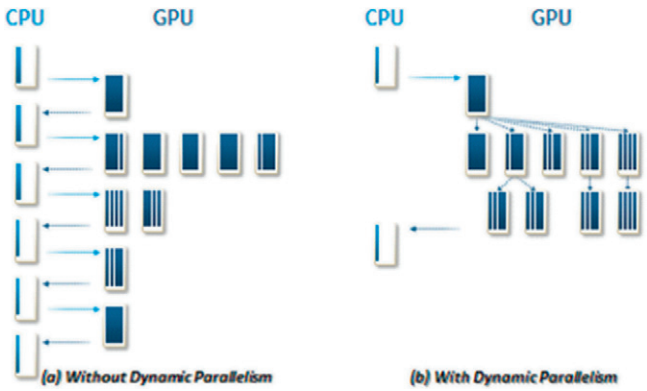

FIGURE 20.2

Kernel launch patterns for algorithms with dynamic work variation: (a) without dynamic parallelism and (b) with dynamic parallelism.

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.

20.2 DYNAMIC PARALLELISM OVERVIEW

From the programmer's perspective dynamic parallelism means that he or she can write a kernel launch statement in a kernel. In Figure 20.3, 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 are 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.

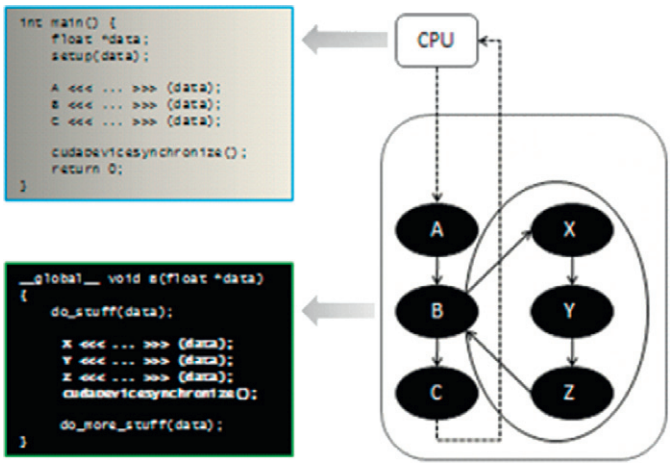

FIGURE 20.3

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.

20.3 IMPORTANT DETAILS

Although the syntax for launching a kernel from a kernel is similar to that for launching a kernel from the host code, there are several important differences that must be clearly understood by programmers.

Launch Environment Configuration

All device configuration settings (e.g., shared memory and L1 cache size as returned from `cudaDeviceGetCacheConfig()`, and device limits as returned from `cudaDeviceGetLimit()`) will be inherited from the parent. That is, if the parent is configured for 16 K bytes of shared memory and 48 K bytes of L1 cache, then the child's execution settings will be configured identically. Likewise, a parent's device limits such as stack size will be passed as-is to its children.

API 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. The last error code returned is 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.

Events

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

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.

Event objects may be shared between the threads within the CUDA thread-block that created them, but are local to that block and should not be passed to child/parent kernels. Event handles are not guaranteed unique between blocks, so using an event handle within a block that did not allocate it will result in undefined behavior.

Streams

Both named and unnamed (NULL) streams are available under dynamic parallelism. Named streams may be used by any thread within a thread block, but stream handles should not be passed to other blocks or child/parent kernels. In other words, a stream should be treated as private to the block in which it is created. Stream handles are not guaranteed to be unique between blocks, so using a stream handle within a block that did not allocate it will result in undefined behavior.

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 are ill-formed and will have undefined behavior.

The host-side NULL stream's global synchronization semantic is not supported under dynamic parallelism. To explicitly indicate this behavior change all streams must be created using the `cudaStreamCreateWithFlags()` API with the `cudaStreamNonBlocking` flag in a kernel. Calls to `cudaStreamCreate()` 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 (i.e., the synchronize guarantees one stream synchronizes, but coincidentally provides a full barrier as a side effect).

A thread that is part of an executing grid and configures and launches a new grid belongs to the parent grid, and the grid created by the launch is the child grid. As shown in Figure 20.4, the creation and completion of child grids is properly nested, meaning that the parent grid is not considered complete until all child grids created by its threads have completed.

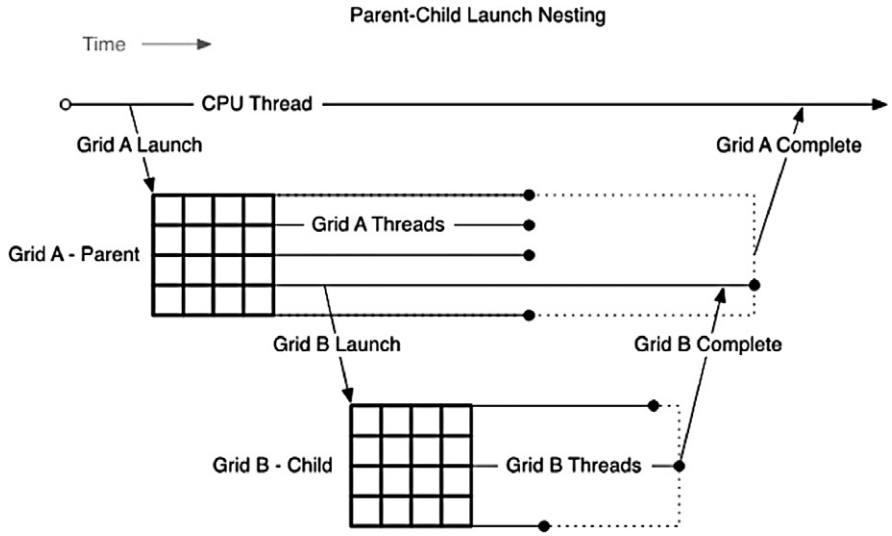

FIGURE 20.4

Completion sequence for parent and child grids.

Even if the parent threads do not explicitly synchronize on the child grids launched, the runtime guarantees an implicit synchronization between the parent and child by forcing the parent to wait for all its children to exit execution before it can exit execution.

Synchronization Scope

A thread in the parent grid may only perform synchronization on the grids launched by that thread (e.g., using `cudaDeviceSynchronize()`), other threads in the thread block (e.g., using `__synchthreads()`), or on streams created within the same thread block (e.g., using `cudaStreamWaitEvent()`). Streams created by a thread within a grid exist only within the thread's thread block scope and have undefined behavior when used outside of the thread block where they were created. Streams created within a thread block are implicitly synchronized when all threads in the thread block exit execution. The behavior of operations on a stream that has been modified outside of the thread block scope is undefined. 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.

20.4 MEMORY VISIBILITY

Global Memory

Parent and child grids have coherent access to global memory, with weak consistency guarantees between child and parent. There are two points in the execution of a child grid when its view of memory is fully consistent with the parent thread: (1) when the child grid is created by the parent, and (2) when the child grid completes as signaled by a synchronization API call in the parent thread.

All global memory operations in the parent thread prior to the child grid's invocation are visible to the child grid. All memory operations of the child grid are visible to the parent after the parent has synchronized on the child grid's completion.

Zero-Copy Memory

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

Constant Memory

Constants are immutable and 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 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 will be undefined. For example, the

following is illegal, with undefined behavior if `x_array` is accessed by `child_launch`:

```
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 be aware of 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 the global scope. For example, Figure 20.5(a) shows a valid kernel launch where a pointer to a global memory variable is passed as an argument into the child kernel. Figure 20.5(b) shows an invalid code where a pointer to a local memory (register) variable is passed into the child kernel.

The NVIDIA 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.

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 shared memory 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 that is writable. Coherence for texture memory is enforced at the invocation of a child grid and when a child

```
__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 20.5

Passing a pointer as an argument to a child kernel: (a) valid (`value` is global storage) and (b) invalid (`value` is local storage).

grid completes. This means that 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 that alias the texture memory objects between a parent and its children or between multiple children will result in undefined behavior.

20.5 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 problem is extracted from the divergent phase of a hypothetical parallel algorithm. It does not compute useful results but provides a conceptually simple calculation that can be easily verified. It serves to illustrate the difference between the two styles and how one can use the dynamic parallelism style to reduce control flow divergence when the amount of work done by each thread in an algorithm can vary dynamically.

Line 22 of Figure 20.6 shows the host code main function for the example coded without dynamic parallelism. It allocates the `foo` variable on the device (line 25) and initializes it to 0 (line 26). It then launches the `diverge_cta()` kernel to perform a calculation on `foo` (line 27). The kernel is launched with a grid of `K` (set to 2 in line 5) blocks of 32 × `M` (`M` set to 32 in line 4) threads each. Therefore, in this example, we are launching two blocks of 1,024 threads each.

In the `diverge_cta()` kernel, threads of which the `threadIdx.x` values are not a multiple of 32 will return immediately. In our example, only the threads with `threadIdx.x` values of 0, 32, 64, . . ., 960, 992 will continue to execute. In line 16, all remaining `M` threads of each block will call the `entry()` function, which will increment the `foo` variable `N` (set to 128 in line 3) times. This is done by the `for` loop in line 8. The atomic operation in line 9 is necessary because there are multiple blocks calling the `entry()` function at the same time. The atomic operation ensures that increments by one of the blocks are not trampled by those of other blocks. In our case, the atomic operation ensures that all increments by both thread blocks are properly reflected in the variable `foo`.

After all blocks have completed their increments, the value of `foo` should be `K` × `M` × `N`, since there are `K` blocks and each block has `M` active threads each incrementing the `foo` variable `N` times. In line 17, thread 0 of each block initializes a shared memory variable `x` (declared in line 13) to value 5, which

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

3.  #define N 128
4.  #define M 32
5.  #define K 2

6.  __device__ volatile int vint = 0;

7.  __device__ void entry( volatile int* foo )
    {
8.    for (int i = 0; i < N; ++i) {
9.         atomicAdd((int*)foo, 1);
       }
      }
10. extern "C"
11. __global__ void
12. diverge_cta( volatile int *foo )
{
13.     __shared__ int x;
14.     if ((threadIdx.x%32) != 0) {
15.     return;
          }
16.     entry(foo);

17.     if (threadIdx.x == 0) {
18.       x = 5;
19.       return;
       }
20.     __syncthreads();

21.    atomicAdd((int*)foo, x);
    }

22. int main( int argc, char **argv )
{
23.    int *foo;
24.    int h_foo;

25.    cudaMalloc((void**)&foo, sizeof(int));
26.    cudaMemset(foo, 0, sizeof(int));
27.    printf("foo addr: 0x%x\n", (unsigned)(size_t)foo);

28.    diverge_cta<<<K,M*32>>>( foo );
29.    cudaDeviceSynchronize();
30.    cudaMemcpy(&h_foo, foo, sizeof(int), cudaMemcpyDe viceToHost);
31.    if (h_foo == K*(M*N+5*(M-1))) {
32.        printf("simple_scan_test test PASSED\n");
33.    }
34.    else {
35.         printf("Result: %d\n", h_foo);
36.         printf("simple_scan_test test FAILED\n");
37.     }

38.     return 0;
39. }
```

FIGURE 20.6

A simple example of the divergent phase of a hypothetical parallel algorithm coded in CUDA without dynamic parallelism.

is visible to all threads in the same block. Thread 0 then terminates. After barrier synchronization (line 20), all remaining `M-1` threads in each block will perform an atomic operation on variable `foo` (line 21). The increment amount of the atomic operation is the value of `x` (5). Since there are only `M-1` threads executing (all of which the `threadIdx.x` values are multiples of 32), all threads in a block should jointly add `5*(M-1)` to the value of `foo`. With a total of `K` blocks in the grid, the total contribution due to line 21 among all blocks is `K*(5*(M-1))`.

After the kernel terminates (line 29), the host copies the value of `foo` into its variable `h_foo` (line 30). The host then performs a test and checks if the total value in `h_foo` is the expected value of `K*N*M + K*(5*(M-1))`, which is `K*(N*M + 5*(M-1))` (line 31).

Figure 20.7 shows a version of the source code based on dynamic parallelism. The main function is identical to that of Figure 20.6 and is not shown. Also, we only assign line numbers to the lines that are different from Figure 20.6. In this version, instead of having thread 0 of each block to call the device function `entry()`, we will have each of them to launch `entry()` as a kernel. In line 2, the device function `entry()` in Figure 20.6 is now declared as a kernel function.

In line 3, the `diverge_cta()` kernel launches the `entry()` kernel with only one block, which contains the `M` thread. `K` (set to 2) kernel launches are done. In our example, one is launched by thread 0 of block 0 and one by thread 0 of block 1. Instead of having each of the remaining `M` threads of a block to call `entry()` as a device function, we use thread 0 of each block to launch `entry()` as a kernel with `M` threads.

Note that the effect on the `foo` value remains the same. The `entry()` kernel is launched `K` times. For each launch, there are `M` threads executing the `entry()` kernel, and each thread increments the `foo` value by `N`. Therefore, the total changes due to all threads are `K*M*N`. However, amount of divergence changes. The original kernel still has divergence. However, the increments are now done by the `entry()` kernel where all neighboring threads are taking the same control flow path. The amount of time the code spends in control-divergent execution decreases.

20.6 RUNTIME LIMITATIONS

Memory Footprint

Memory is allocated as the backing-store for the parent kernel state to be used when synchronizing on a child launch. Conservatively, this memory

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

    #define N 100
    #define M 32
    #define K 2

    __device__ volatile int vint = 0;

1.  __global__ void
    entry( volatile int* foo )
    {
      for (int i = 0; i < N; ++i) {
        atomicAdd((int*)foo, 1);
      }
     }

    extern "C"
    __global__ void
    diverge_cta( volatile int *foo )
    {
      __shared__ int x;
      if ((threadIdx.x%32) != 0) {
          return;
      }
      if (threadIdx.x == 0) {
2.        entry<<<1,M>>>( foo );
3.        cudaDeviceSynchronize();
          x = 5;
          return;
     }
      __syncthreads();

      atomicAdd((int*)foo, x);
    }
```

FIGURE 20.7

The `diverge_cta()` kernel revised using dynamic parallelism.

must support storing of state for the maximum number of live threads possible on the GPU. This in turn means that each level of nesting requires ~150 MB of device memory in a current generation device, which will be unavailable for program use even if it is not all consumed. The dynamic parallelism runtime system detects if the parent exits without calling `cudaDeviceSynchronize()`. In this case, the runtime does not save the parent's state and the memory footprint required for the program will be much less than the conservative maximum.

In addition to the thread backing-store, more memory is used by the system software, for example, to store launch queues and events. The total memory footprint of dynamic parallelism is difficult to specify exactly, but may be queried at runtime.

Nesting Depth

Under dynamic parallelism, one kernel may launch another kernel, and that kernel may launch another, 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 64, but in software it may be limited to 63 or less. Practically speaking, the real limit will be the amount of memory required by the system for each new level (see the preceding "Memory Footprint" section). The number of levels to be supported must be configured before the top-level kernel is launched from the host, to guarantee successful execution of a nested program.

Memory Allocation and Lifetime

Currently, `cudaMalloc` and `cudaFree` have slightly modified semantics between the host and device environments (Table 20.1). 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. Also, it is an error to invoke `cudaFree` from the host program on a pointer that was allocated by `cudaMalloc` on the device, or to invoke `cudaFree` from the device program on a pointer that was allocated by `cudaMalloc` on the host. These limitations may be removed in a future version.

Table 20.1 Memory allocation and deallocation from host and device.

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

ECC Errors

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 that arise during execution of a dynamic parallelism kernel will either generate an exception or continue execution (depending on error and configuration).

Streams

Unlimited 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. In addition to block-scope named streams, each thread has an unnamed (NULL) stream, but named streams will not synchronize against it (indeed, all named streams must be created with a flag explicitly preventing this).

Events

Unlimited 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 GPU-launched grids may attain less concurrency than might be expected. Correct execution is guaranteed, however.

Launch Pool

When a kernel is launched, all associated data is added to a slot within the launch pool, which is tracked until the kernel completes. Launch pool storage may be virtualized by the system, between device and host memory; however, device-side launch pool storage has improved performance. The amount of device memory reserved for device-side launch pool storage is configurable prior to the initial kernel launch from the host.

20.7 A MORE COMPLEX EXAMPLE

We now show an example that is a more interesting and useful case of recursive, adaptive subdivision of spline curves. This illustrates a variable amount of child kernel launches, according to the workload. The example is to calculate Bezier curves [Wiki_Bezier 2012], 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.

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 are given by the following linear interpolation formula:

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

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:

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

which can be simplified into the following formula:

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

Bezier Curve Calculation (Predynamic Parallelism)

Figure 20.8 shows a CUDA C program that calculates the coordinates of points on a Bezier curve. The first part of the code defines several operators (`operator+`, `operator-`, `operator*`, `length`) for 2D coordinates that will be used in the kernel code. They should be quite self-explanatory so we will not elaborate on them.

The main function (line 20) initializes a set of control points to random values (lines 22, 23, and 24). In a real application, these control points are

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

//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);
}

#define MAX_TESS_POINTS 32

1.  struct BezierLine //A structure containing all the parameters we
      need to tessellate a Bezier line
    {
      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
2.    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]);
      //From the curvature, compute the number of tessellation points
3.    int nTessPoints = min(max((int)(curvature*16.0f),4),32);
4.    bLines[bidx].nVertices = nTessPoints;

      //Loop through the vertices to be tessellated, incrementing by
          blockDim.x
5.     for(int inc = 0; inc < nTessPoints; inc += blockDim.x){
```

FIGURE 20.8

Bezier curve calculation without dynamic parallelism.

```
6.      int idx = inc + threadIdx.x;  //Compute a unique index for
          this point
7.      if(idx < nTessPoints){
8.      float u = (float)idx/(float)(nTessPoints-1);  //Compute u
          from idx
9.      float omu = 1.0f - u;   //pre-compute one minus u

10.     float B3u[3]; //Compute quadratic Bezier coefficients
11.     B3u[0] = omu*omu;
12.     B3u[1] = 2.0f*u*omu;
13.     B3u[2] = u*u;

14.     float2 position = {0,0};  //Set position to zero
15.     for(int i = 0; i < 3; i++){
         //Add the contribution of the i'th control point to position
16.      position = position + B3u[i] * bLines[bidx].CP[i];
        }
        //Assign the value of the vertex position to the correct
        array element
17.     bLines[bidx].vertexPos[idx] = position;
      }
     }
    }
   }

18. #define N_LINES 256
19. #define BLOCK_DIM 32

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

    float2 last = {0,0};    //Set initial point to zero (last is the
          last point in the previous segment).
22.  for(int i = 0; i < N_LINES; i++){
23.   bLines_h[i].CP[0] = last; //Set first point of this line to last
          point of previous line
24.    for(int j = 1; j < 3; j++){
        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;  //Assign
          random coordinate between 0 and 1
       }
       last = bLines_h[i].CP[2];   //keep the last point of this line
       bLines_h[i].nVertices = 0;  //Set number of tessellated
          vertices to zero
      }

25.   BezierLine *bLines_d; //Pointer to array of Bezier lines in
      device memory
26.   cudaMalloc((void**)&bLines_d, N_LINES*sizeof(BezierLine));
          //Allocate device memory for array of Bezier lines
```

FIGURE 20.8

(continued)

```
27.   cudaMemcpy(bLines_d, bLines_h, N_LINES*sizeof(BezierLine),
          cudaMemcpyHostToDevice);

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

      //Do something to draw the lines here

      cudaFree(bLines_d); //Free the array of lines in device memory
```

FIGURE 20.8

(continued)

most likely inputs from a user. The control points are part of the `bLines_h` array of which the element type `BezierLine` is declared in line 1. The storage for the `bLines_h` array is allocated in line 21. The host code then allocates the corresponding device memory for the `bLines_d` array and copies the initialized data to `bLines_d` (lines 26–28). It then calls the `computeBezierLine()` kernel to calculate the coordinates of the Bezier curve.

The `computeBezierLine()` kernel 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 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 3 and 4, 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 5, 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 streaming multiprocessors and thus reduced performance.

Bezier Curve Calculation (with Dynamic Parallelism)

Figure 20.9 shows a Bezier curve calculation code using dynamic parallelism. It breaks the `computeBezierLine()` kernel in Figure 20.8 into two kernels. The first part, `computeBezierLineCDP()`, discovers the amount of work to be

```
1. struct BezierLine
   {
    float2 CP[3];      //Control points for the line
    float2 *vertexPos; //Vertex position array to tessellate into
    int nVertices;     //Number of tessellated vertices
   };

2. __global__ void computeBezierLinePositions(int lidx, BezierLine*
       bLines, int nTessPoints)
   {
3.  int idx = threadIdx.x + blockDim.x*blockIdx.x; //Compute an
       index unique to this vertex
4.  if(idx < nTessPoints){
5.   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];
     }

     bLines[lidx].vertexPos[idx] = position; //Assign the value of the
          vertex position to the correct array element
     }
    }
    __global__ void computeBezierLinesCDP(BezierLine *bLines, int
         nLines)
    {
6.   int lidx = threadIdx.x + blockDim.x*blockIdx.x;  //Compute a
        unique index for each Bezier line

7.   if(lidx < nLines){
       //Compute the curvature of the line
       float curvature = length(bLines[lidx].CP[1] - 0.5f*(bLines[lidx]
                .CP[0] + bLines[lidx].CP[2]))/length(bLines[lidx].CP[2]-
                bLines[lidx].CP[0]);
       //From the curvature, compute the number of tessellation points
       bLines[lidx].nVertices = min(max((int)(curvature*16.0f),4),
       MAX_TESS_POINTS);

8.     cudaMalloc((void**)&bLines[lidx].vertexPos, bLines[lidx]
       .nVertices*sizeof(float2));
       //Call the child kernel to compute the tessellated points for
       each line
9.     computeBezierLinePositions<<<ceil((float)bLines[lidx]
       .nVertices/32.0f), 32>>>(lidx, bLines,bLines[lidx].nVertices);
  }
}
```

FIGURE 20.9

Bezier calculation with dynamic parallelism.

```
    __global__ void freeVertexMem(BezierLine *bLines, int nLines)
    {
    int lidx = threadIdx.x + blockDim.x*blockIdx.x;   //Compute a unique
        index for each Bezier line
10. if(lidx < nLines)
11. cudaFree(bLines[lidx].vertexPos);   //Free the vertex memory for
        this line
    }

     #define N_LINES 256
    #define BLOCK_DIM 64

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

      float2 last = {0,0};    //Set last point to zero
       for(int i = 0; i < N_LINES; i++){
        bLines_h[i].CP[0] = last;   //Set first point of this line to
                last point of previous line
         for(int j = 1; j < 3; j++){
          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; //Assign
                random coordinate between 0 and 1
         }
      last = bLines_h[i].CP[2];   //keep the last point of this line
      bLines_h[i].vertexPos = NULL;   //Set the vertex position array
                to NULL
      bLines_h[i].nVertices = 0;  //Set number of tessellated vertices
                to zero
      }

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

13.  computeBezierLinesCDP<<<ceil((float)N_LINES/(float)BLOCK_DIM),
        BLOCK_DIM>>>(bLines_d, N_LINES);

      //Do something to draw the lines here

14.  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 20.9

(continued)

done for each control point. The second part, `computeBezierLinePositions()`, performs the calculation.

With the new organization, the amount of work done for each set of control points by the `computeBezierLinesCDP()` kernel is much smaller than the original `computeBezierLines()` kernel. Therefore, we use one thread to do this work in `computeBezierLinesCDP()`, as opposed to using one block in `computeBezierLinesPossitions()`. In line 13, 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 `computeBezierLinesCDP()` kernel and the `computeBezierLines()` kernel. First, the index used to access the control points is formed on a thread basis (line 6) rather than a block basis. This is because the work for each control point is done by a thread rather than a block as we mentioned before. Second, the memory for storing the calculated Bezier curve points is dynamically determined and allocated in line 8. This allows the code to assign just enough memory to each set of control points in the `BezierLine` type. Note that in Figure 20.8, each `BezierLine` element is declared with a maximal possible number of points. On the other hand, the declaration in Figure 20.9 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 varies significantly.

Once a thread of the `computeBezierLinesCDP()` kernel determines the amount of work needed by its set of control points, it launches the `computeBezierPositions()` kernel to do the work (line 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 `computeBezierLinesCDP()` kernel terminates, the main function can copy the data back and draw the curve on an output device. It can also call a kernel to free all storage allocated to the `bLines_d` storage in parallel (line 14). This can be faster than sequentially calling the `cudaFree()` function in a loop.

20.8 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. 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 as well as more efficient memory usage.

Reference

Bezier Curves, Available at: <*http://en.wikipedia.org/wiki/B%C3%A9zier_curve*>, 2012.