8.3 Programming GPUs

(Original section[1])

Programming multiprocessor GPUs is qualitatively different than programming other multiprocessors like multicore CPUs. GPUs provide two to three orders of magnitude more thread and data parallelism than CPUs, scaling to hundreds of processor cores and tens of thousands of concurrent threads. GPUs continue to increase their parallelism, doubling it about every 12 to 18 months, enabled by Moore's law [1965] of increasing integrated circuit density and by improving architectural efficiency. To span the wide price and performance range of different market segments, different GPU products implement widely varying numbers of processors and threads. Yet users expect games, graphics, imaging, and computing applications to work on any GPU, regardless of how many parallel threads it executes or how many parallel processor cores it has, and they expect more expensive GPUs (with more threads and cores) to run applications faster. As a result, GPU programming models and application programs are designed to scale transparently to a wide range of parallelism.

The driving force behind the large number of parallel threads and cores in a GPU is real-time graphics performance—the need to render complex 3D scenes with high resolution at interactive frame rates, at least 60 frames per second. Correspondingly, the scalable programming models of graphics shading languages such as Cg (C for graphics) and HLSL (*high-level shading language*) are designed to exploit large degrees of parallelism via many independent parallel threads and to scale to any number of processor cores. The CUDA scalable parallel programming model similarly enables general parallel computing applications to leverage large numbers of parallel threads and scale to any number of parallel processor cores, transparently to the application.

In these scalable programming models, the programmer writes code for a single thread, and the GPU runs myriad thread instances in parallel. Programs thus scale transparently over a wide range of hardware parallelism. This simple paradigm arose from graphics APIs and shading languages that describe how to shade one vertex or one pixel. It has remained an effective paradigm as GPUs have rapidly increased their parallelism and performance since the late 1990s.

This section briefly describes programming GPUs for real-time graphics applications using graphics APIs and programming languages. It then describes programming GPUs for visual computing and general parallel computing applications using the C language and the CUDA programming model.

Programming real-time graphics

APIs have played an important role in the rapid, successful development of GPUs and processors. There are two primary standard graphics APIs: *OpenGL* and *Direct3D*, one of the Microsoft DirectX multimedia programming interfaces. OpenGL, an open standard, was originally proposed and defined by Silicon Graphics Incorporated. The ongoing development and extension of the OpenGL standard [Segal and Akeley, 2006; Kessenich, 2006] is managed by Khronos, an industry consortium. Direct3D [Blythe, 2006], a de facto standard, is defined and evolved forward by Microsoft and partners. OpenGL and Direct3D are similarly structured, and continue to evolve rapidly with GPU hardware advances. They define a logical graphics processing pipeline that is mapped onto the GPU hardware and processors, along with programming models and languages for the programmable pipeline stages.

OpenGL: An open-standard graphics API.

Direct3D: A graphics API defined by Microsoft and partners.

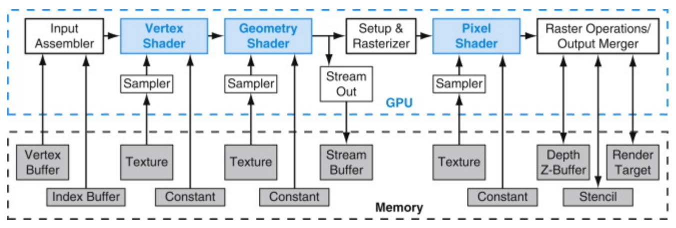
Logical graphics pipeline

The figure below illustrates the Direct3D 10 logical graphics pipeline. OpenGL has a similar graphics pipeline structure. The API and logical pipeline provide a streaming dataflow infrastructure and plumbing for the programmable shader stages, shown in blue. The 3D application sends the GPU a sequence of vertices grouped into geometric primitives—points, lines, triangles, and polygons. The input assembler collects vertices and primitives. The vertex shader program executes per-vertex processing, including transforming the vertex 3D position into a screen position and lighting the vertex to determine its color. The geometry shader program executes per-primitive processing and can add or drop primitives. The setup and rasterizer unit generates pixel fragments (fragments are potential contributions to pixels) that are covered by a geometric primitive. The pixel shader program performs per-fragment processing, including interpolating per-fragment parameters, texturing, and coloring. Pixel shaders make extensive use of sampled and filtered lookups into large 1D, 2D, or 3D arrays called *textures*, using interpolated floating-point coordinates. Shaders use texture accesses for maps, functions, decals, images, and data. The raster operations processing (or output merger) stage performs Z-buffer depth testing and stencil testing, which may discard a hidden pixel fragment or replace the pixel's depth with the fragment's depth, and performs a color blending operation that combines the fragment color with the pixel color and writes the pixel with the blended color.

Texture: A 1D, 2D, or 3D array that supports sampled and filtered lookups with interpolated coordinates.

Figure 8.3.1: Direct3D 10 graphics pipeline (COD Figure B.3.1).

Each logical pipeline stage maps to GPU hardware or to a GPU processor. Programmable shader stages are blue, fixed-function blocks are white, and memory objects are gray. Each stage processes a vertex, geometric primitive, or pixel in a streaming dataflow fashion.



The graphics API and graphics pipeline provide input, output, memory objects, and infrastructure for the shader programs that process each vertex, primitive, and pixel fragment.

Graphics shader programs

Real-time graphics applications use many different *shader* programs to model how light interacts with different materials and to render complex lighting and shadows. *Shading languages* are based on a dataflow or streaming programming model that corresponds with the logical graphics pipeline. Vertex shader programs map the position of triangle vertices onto the screen, altering their position, color, or orientation. Typically a vertex shader thread inputs a floating-point (x, y, z, w) vertex position and computes a floating-point (x, y, z) screen position. Geometry shader programs operate on geometric primitives (such as lines and triangles) defined by multiple vertices, changing them or generating additional primitives. Pixel fragment shaders each "shade" one pixel, computing a floating-point *red, green, blue, alpha* (RGBA) color contribution to the rendered image at its pixel sample (x, y) image position. Shaders (and GPUs) use floating-point arithmetic for all pixel color calculations to eliminate visible artifacts while computing the extreme range of pixel contribution values encountered while rendering scenes with complex lighting, shadows, and high dynamic range. For all three types of graphics shaders, many program instances can be run in parallel, as independent parallel threads, because each works on independent data, produces independent results, and has no side effects. Independent vertices, primitives, and pixels further enable the same graphics program to run on differently sized GPUs that process different numbers of vertices, primitives, and pixels in parallel. Graphics programs thus scale transparently to GPUs with different amounts of parallelism and performance.

Shader: A program that operates on graphics data such as a vertex or a pixel fragment.

Shading language: A graphics rendering language, usually having a dataflow or streaming programming model.

Users program all three logical graphics threads with a common targeted high-level language. HLSL (high-level shading language) and Cg (C for graphics) are commonly used. They have C-like syntax and a rich set of library functions for matrix operations, trigonometry, interpolation, and texture access and filtering, but are far from general computing languages: they currently lack general memory access, pointers, file I/O, and recursion. HLSL and Cg assume that programs live within a logical graphics pipeline, and thus I/O is implicit. For example, a pixel fragment shader may expect the geometric normal and multiple texture coordinates to have been interpolated from vertex values by upstream fixed-function stages and can simply assign a value to the COLOR output parameter to pass it downstream to be blended with a pixel at an implied (x, y) position.

The GPU hardware creates a new independent thread to execute a vertex, geometry, or pixel shader program for every vertex, every primitive, and every pixel fragment. In video games, the bulk of threads execute pixel shader programs, as there are typically 10 to 20 times more pixel fragments than vertices, and complex lighting and shadows require even larger ratios of pixel to vertex shader threads. The graphics shader programming model drove the GPU architecture to efficiently execute thousands of independent fine-grained threads on many parallel processor cores.

Pixel shader example

Consider the following Cg pixel shader program that implements the "environment mapping" rendering technique. For each pixel thread, this shader is passed five parameters, including 2D floating-point texture image coordinates needed to sample the surface color, and a 3D floating-point vector giving the reflection of the view direction off the surface. The other three "uniform" parameters do not vary from one pixel instance (thread) to the next. The shader looks up color in two texture images: a 2D texture access for the surface color, and a 3D texture access into a cube map (six images corresponding to the faces of a cube) to obtain the external world color corresponding to the reflection direction. Then the final four-component (red, green, blue, alpha) floating-point color is computed using a weighted average called a "lerp" or linear interpolation function.

```
void reflection(
   float2                texCoord        : TEXCOORD0,
   float3                reflection_dir   : TEXCOORD1,
   out float4            color           : COLOR,
   uniform float         shiny,
   uniform sampler2D     surfaceMap,
   uniform samplerCUBE   envMap)
{
// Fetch the surface color from a texture
   float4 surfaceColor = tex2D(surfaceMap, texCoord);

// Fetch reflected color by sampling a cube map
   float4 reflectedColor = texCUBE(environmentMap, refection_dir);

// Output is weighted average of the two colors
   color = lerp(surfaceColor, reflectedColor, shiny);
}
```

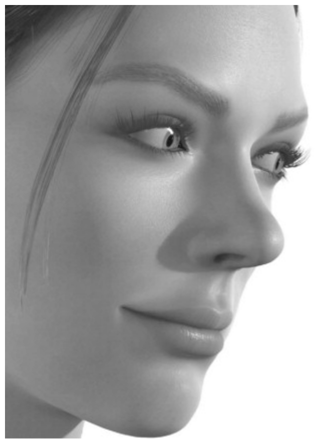
Although this shader program is only three lines long, it activates a lot of GPU hardware. For each texture fetch, the GPU texture subsystem makes multiple memory accesses to sample image colors in the vicinity of the sampling coordinates, and then interpolates the final result with floating-point filtering arithmetic. The multithreaded GPU executes thousands of these lightweight Cg pixel shader threads in parallel, deeply interleaving them to hide texture fetch and memory latency.

Cg focuses the programmer's view to a single vertex or primitive or pixel, which the GPU implements as a single thread; the shader program transparently scales to exploit thread parallelism on the available processors. Being application-specific, Cg provides a rich set of useful data types, library functions, and language constructs to express diverse rendering techniques.

The figure below shows skin rendered by a fragment pixel shader. Real skin appears quite different from flesh-color paint because light bounces around a lot before re-emerging. In this complex shader, three separate skin layers, each with unique subsurface scattering behavior, are modeled to give the skin a visual depth and translucency. Scattering can be modeled by a blurring convolution in a fattened "texture" space, with red being blurred more than green, and blue blurred less. The compiled Cg shader executes 1400 instructions to compute the color of one skin pixel.

Figure 8.3.2: GPU-rendered image (COD Figure B.3.2).

To give the skin visual depth and translancuency, the pixel shader program models three separate skin layers, each with unique subsurface scattering behavior. It executes 1400 instructions to render the red, green, blue, and alpha color components of each skin pixel fragment.



As GPUs have evolved superior floating-point performance and very high streaming memory bandwidth for real-time graphics, they have attracted highly parallel applications beyond traditional graphics. At first, access to this power was available only by couching an application as a graphics-rendering algorithm, but this GPGPU approach was often awkward and limiting. More recently, the CUDA programming model has provided a far easier way to exploit the scalable high-performance floating-point and memory bandwidth of GPUs with the C programming language.

Programming parallel computing applications

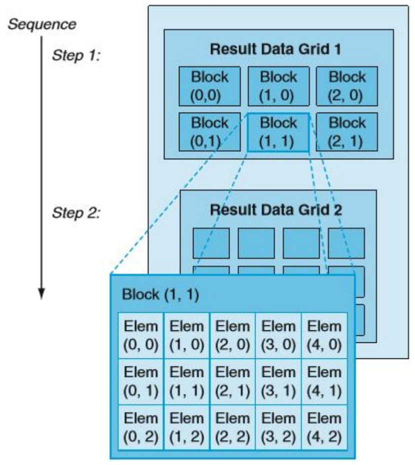
CUDA, Brook, and CAL are programming interfaces for GPUs that are focused on data parallel computation rather than on graphics. CAL (*Compute Abstraction Layer*) is a low-level assembler language interface for AMD GPUs. Brook is a streaming language adapted for GPUs by Buck et al. [2004]. CUDA, developed by NVIDIA [2007], is an extension to the C and C++ languages for scalable parallel programming of manycore GPUs and multicore CPUs. The CUDA programming model is described below, adapted from an article by Nickolls et al. [2008].

With the new model the GPU excels in data parallel and throughput computing, executing high-performance computing applications as well as graphics applications.

Data parallel problem decomposition

To map large computing problems effectively to a highly parallel processing architecture, the programmer or compiler decomposes the problem into many small problems that can be solved in parallel. For example, the programmer partitions a large result data array into blocks and further partitions each block into elements, such that the result blocks can be computed independently in parallel, and the elements within each block are computed in parallel. The figure below shows a decomposition of a result data array into a 3 × 2 grid of blocks, where each block is further decomposed into a 5 × 3 array of elements. The two-level parallel decomposition maps naturally to the GPU architecture: parallel multiprocessors compute result blocks, and parallel threads compute result elements.

Figure 8.3.3: Decomposing result data into a grid of blocks of elements to be computed in parallel (COD Figure B.3.3).



The programmer writes a program that computes a sequence of result data grids, partitioning each result grid into coarse-grained result blocks that can be computed independently in parallel. The program computes each result block with an array of fine-grained parallel threads, partitioning the work among threads so that each computes one or more result elements.

Scalable parallel programming with CUDA

The CUDA scalable parallel programming model extends the C and C++ languages to exploit large degrees of parallelism for general applications on highly parallel multiprocessors, particularly GPUs. Early experience with CUDA shows that *many* sophisticated programs can be readily expressed with a few easily understood abstractions. Since NVIDIA released CUDA in 2007, developers have rapidly developed scalable parallel programs for a wide range of applications, including seismic data processing, computational chemistry, linear algebra, sparse matrix solvers, sorting, searching, physics models, and visual computing. These applications scale transparently to hundreds of processor cores and thousands of concurrent threads. NVIDIA GPUs with the Tesla unified graphics and computing architecture (described in COD Section B.4 (Multithreaded multiprocessor architecture) and B.7 (Real Stuff: The NVIDIA GeForce 8800)) run

CUDA C programs, and are widely available in laptops, PCs, workstations, and servers. The CUDA model is also applicable to other shared memory parallel processing architectures, including multicore CPUs.

CUDA provides three key abstractions—a *hierarchy of thread groups*, *shared memories*, and *barrier synchronization*—that provide a clear parallel structure to conventional C code for one thread of the hierarchy. Multiple levels of threads, memory, and synchronization provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. The abstractions guide the programmer to partition the problem into coarse subproblems that can be solved independently in parallel, and then into finer pieces that can be solved in parallel. The programming model scales transparently to large numbers of processor cores: a compiled CUDA program executes on any number of processors, and only the runtime system needs to know the physical processor count.

The CUDA paradigm

CUDA is a minimal extension of the C and C++ programming languages. The programmer writes a serial program that calls parallel *kernels*, which may be simple functions or full programs. A kernel executes in parallel across a set of parallel threads. The programmer organizes these threads into a hierarchy of thread blocks and grids of thread blocks. A *thread block* is a set of concurrent threads that can cooperate among themselves through barrier synchronization and through shared access to a memory space private to the block. A *grid* is a set of thread blocks that may each be executed independently and thus may execute in parallel.

Kernel: A program or function for one thread, designed to be executed by many threads.

Thread block: A set of concurrent threads that execute the same thread program and may cooperate to compute a result.

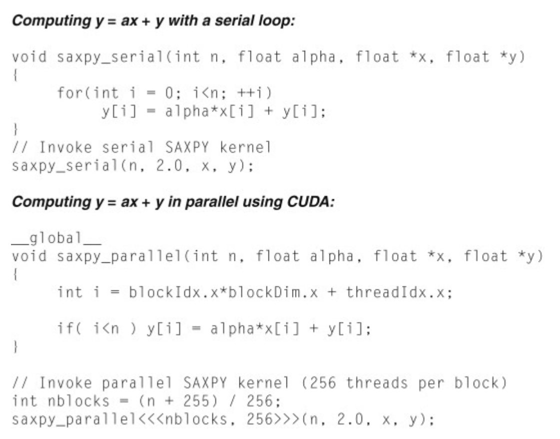
Grid: A set of thread blocks that execute the same kernel program.

When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks comprising the grid. Each thread is given a unique *thread ID* number `threadIdx` within its thread block, numbered `0`, `1`, `2`, `...`, `blockDim-1`, and each thread block is given a unique *block ID* number `blockIdx` within its grid. CUDA supports thread blocks containing up to 512 threads. For convenience, thread blocks and grids may have one, two, or three dimensions, accessed via `.x`, `.y`, and `.z` index fields.

As a very simple example of parallel programming, suppose that we are given two vectors *x* and *y* of *n* floating-point numbers each and that we wish to compute the result of *y* = *ax* + *y* for some scalar value *a*. This is the so-called `SAXPY` kernel defined by the BLAS linear algebra library. The figure below shows C code for performing this computation on both a serial processor and in parallel using CUDA.

Figure 8.3.4: Sequential code (top) in C versus parallel code (bottom) in CUDA for SAXPY (see COD Chapter 6 (Parallel Processor from Client to Cloud)) (COD Figure B.3.4).

CUDA parallel threads replace the C serial loop—each thread computes the same result as one loop iteration. The parallel code computes *n* results with *n* threads organized in blocks of 256 threads.



Computing y = ax + y with a serial loop:

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
     for(int i = 0; i<n; ++i)
          y[i] = alpha*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing y = ax + y in parallel using CUDA:

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
     int i = blockIdx.x*blockDim.x + threadIdx.x;

     if( i<n ) y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

The `__global__` declaration specifier indicates that the procedure is a kernel entry point. CUDA programs launch parallel kernels with the extended function call syntax:

```
kernel<<<dimGrid, dimBlock>>>(... parameter list ...);
```

where `dimGrid` and `dimBlock` are three-element vectors of type `dim3` that specify the dimensions of the grid in blocks and the dimensions of the blocks in threads, respectively. Unspecified dimensions default to one.

In the figure above, we launch a grid of *n* threads that assigns one thread to each element of the vectors and puts 256 threads in each block. Each individual thread computes an element index from its thread and block IDs and then performs the desired calculation on the corresponding vector elements. Comparing the serial and parallel versions of this code, we see that they are strikingly similar. This represents a fairly common pattern. The serial code consists of a loop where each iteration is independent of all the others. Such loops can be mechanically transformed into parallel kernels: each loop iteration becomes an independent thread. By assigning a single thread to each output element, we avoid the need for any synchronization among threads when writing results to memory.

The text of a CUDA kernel is simply a C function for one sequential thread. Thus, it is generally straightforward to write and is typically simpler than writing parallel code for vector operations. Parallelism is determined clearly and explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel.

Parallel execution and thread management is automatic. All thread creation, scheduling, and termination is handled for the programmer by the underlying system. Indeed, a Tesla architecture GPU performs all thread management directly in hardware. The threads of a block execute concurrently and may synchronize at a *synchronization barrier* by calling the `__syncthreads()` intrinsic. This guarantees that no thread in the block can proceed until all threads in the block have reached the barrier. After passing the barrier, these threads are also guaranteed to see all writes to memory performed by threads in the block before the barrier. Thus, threads in a block may communicate with each other by writing and reading per-block shared memory at a synchronization barrier.

Synchronization barrier: Threads wait at a synchronization barrier until all threads in the thread block arrive at the barrier.

Since threads in a block may share memory and synchronize via barriers, they will reside together on the same physical processor or multiprocessor. The number of thread blocks can, however, greatly exceed the number of processors. The CUDA thread programming model virtualizes the processors and gives the programmer the flexibility to parallelize at whatever granularity is most convenient. Virtualization into threads and thread blocks allows intuitive problem decompositions, as the number of blocks can be dictated by the size of the data being processed rather than by the number of processors in the system. It also allows the same CUDA program to scale to widely varying numbers of processor cores.

To manage this processing element virtualization and provide scalability, CUDA requires that thread blocks be able to execute independently. It must be possible to execute blocks in any order, in parallel or in series. Different blocks have no means of direct communication, although they may *coordinate* their activities using *atomic memory operations* on the global memory visible to all threads—by atomically incrementing queue pointers, for example. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, making the CUDA model scalable across an arbitrary number of cores as well as across a variety of parallel architectures. It also helps to avoid the possibility of deadlock. An application may execute multiple grids either independently or dependently. Independent grids may execute concurrently, given sufficient hardware resources. Dependent grids execute sequentially, with an implicit interkernel barrier between them, thus guaranteeing that all blocks of the first grid complete before any block of the second, dependent grid begins.

Atomic memory operation: A memory read, modify, write operation sequence that completes without any intervening access.

Threads may access data from multiple memory spaces during their execution. Each thread has a private *local memory*. CUDA uses local memory for thread-private variables that do not fit in the thread's registers, as well as for stack frames and register spilling. Each thread block has a *shared memory*, visible to all threads of the block, which has the same lifetime as the block. Finally, all threads have access to the same *global memory*. Programs declare variables in shared and global memory with the `__shared__` and `__device__` type qualifiers. On a Tesla architecture GPU, these memory spaces correspond to physically separate memories: per-block shared memory is a low-latency on-chip RAM, while global memory resides in the fast DRAM on the graphics board.

Local memory: Per-thread local memory private to the thread.

Shared memory: Per-block memory shared by all threads of the block.

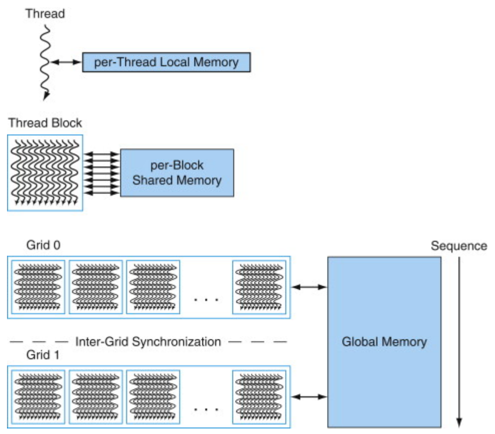
Global memory: Per-application memory shared by all threads.

Shared memory is expected to be a low-latency memory near each processor, much like an L1 cache. It can therefore provide high-performance communication and data sharing among the threads of a thread block. Since it has the same lifetime as its corresponding thread block, kernel code will typically initialize data in shared variables, compute using shared variables, and copy shared memory results to global memory. Thread blocks of sequentially dependent grids communicate via global memory, using it to read input and write results.

The figure below shows diagrams of the nested levels of threads, thread blocks, and grids of thread blocks. It further shows the corresponding levels of memory sharing: local, shared, and global memories for per-thread, per-thread-block, and per-application data sharing.

Figure 8.3.5: Nested granularity levels—thread, thread block, and grid—have corresponding memory sharing levels—local, shared, and global (COD Figure B.3.5).

Per-thread local memory is private to the thread. Per-block shared memory is shared by all threads of the block. Per-application global memory is shared by all threads.



A program manages the global memory space visible to kernels through calls to the CUDA runtime, such as `cudaMalloc()` and `cudaFree()`. Kernels may execute on a physically separate device, as is the case when running kernels on the GPU. Consequently, the application must use `cudaMemcpy()` to copy data between the allocated space and the host system memory.

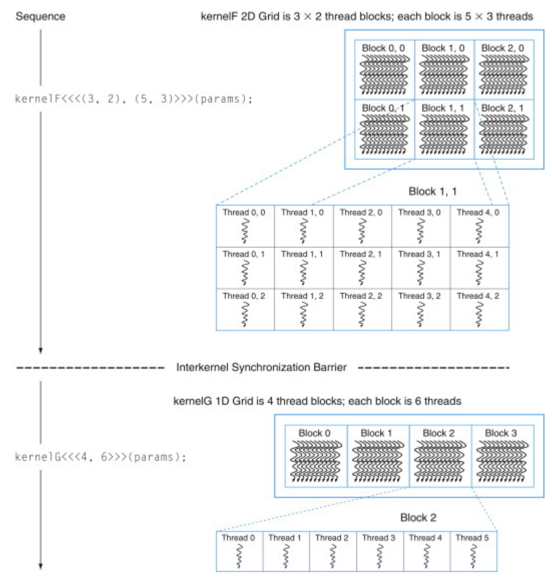
The CUDA programming model is similar in style to the familiar *single-program multiple data (SPMD)* model—it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. However, CUDA is more flexible than most realizations of SPMD, because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step. The programmer can use a convenient degree of parallelism for each kernel, rather than having to design all phases of the computation to use the same number of threads. The figure below shows an example of an SPMD-like CUDA code sequence. It first instantiates `kernelF` on a 2D grid of 3 × 2 blocks where each 2D thread block consists of 5 × 3 threads. It then instantiates `kernelG` on a 1D grid of four 1D thread blocks with six threads each. Because `kernelG` depends on the results of `kernelF`, they are separated by an interkernel synchronization barrier.

Single-program multiple data (***SPMD***): A style of parallel programming model in which all threads execute the same program. SPMD threads typically coordinate with barrier synchronization.

Figure 8.3.6: Sequence of kernel F (COD Figure B.3.6)

Figure [illegible]: Sequence of kernel F (COD Figure [illegible]).

Sequence of kernel ***F*** instantiated on a 2D grid of 2D thread blocks, an interkernel synchronization barrier, followed by kernel ***G*** on a 1D grid of 1D thread blocks.



The concurrent threads of a thread block express fine-grained data parallelism and thread parallelism. The independent thread blocks of a grid express coarse-grained data parallelism. Independent grids express coarse-grained task parallelism. A kernel is simply C code for one thread of the hierarchy.

Restrictions

For efficiency, and to simplify its implementation, the CUDA programming model has some restrictions. Threads and thread blocks may only be created by invoking a parallel kernel, not from within a parallel kernel. Together with the required independence of thread blocks, this makes it possible to execute CUDA programs with a simple scheduler that introduces minimal runtime overhead. In fact, the Tesla GPU architecture implements *hardware* management and scheduling of threads and thread blocks

Task parallelism can be expressed at the thread block level but is difficult to express within a thread block because thread synchronization barriers operate on all the threads of the block. To enable CUDA programs to run on any number of processors, dependencies among thread blocks within the same kernel grid are not allowed—blocks must execute independently. Since CUDA requires that thread blocks be independent and allows blocks to be executed in any order, combining results generated by multiple blocks must in general be done by launching a second kernel on a new grid of thread blocks (although thread blocks may *coordinate* their activities using atomic memory operations on the global memory visible to all threads—by atomically incrementing queue pointers, for example).

Recursive function calls are not currently allowed in CUDA kernels. Recursion is unattractive in a massively parallel kernel, because providing stack space for the tens of thousands of threads that may be active would require substantial amounts of memory. Serial algorithms that are normally expressed using recursion, such as quicksort, are typically best implemented using nested data parallelism rather than explicit recursion.

To support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data and results between host memory and device memory. The overhead of CPU-GPU interaction and data transfers is minimized by using DMA block transfer engines and fast interconnects. Compute-intensive problems large enough to need a GPU performance boost amortize the overhead better than small problems.

Implications for architecture

The parallel programming models for graphics and computing have driven GPU architecture to be different than CPU architecture. The key aspects of GPU programs driving GPU processor architecture are:

- *Extensive use of fine-grained data parallelism*: Shader programs describe how to process a single pixel or vertex, and CUDA programs describe how to compute an individual result.
- *Highly threaded programming model*: A shader thread program processes a single pixel or vertex, and a CUDA thread program may generate a single result. A GPU must create and execute millions of such thread programs per frame, at 60 frames per second.
- *Scalability*: A program must automatically increase its performance when provided with additional processors, without recompiling.
- *Intensive floating-point (or integer) computation*.
- *Support of high-throughput computations*.

(*1) This section is in original form.

Provide feedback on this section