GPU Series – VIII

CUDA THREAD ORGANIZATION

Synchronization and Transparent Scalability

In CUDA programming, threads within the same block can synchronize their activities using the barrier synchronization function ***__syncthreads()***. This function ensures that all threads in a block reach the same point in the code before any thread can proceed further. The syntax involves two underscore characters: __syncthreads().

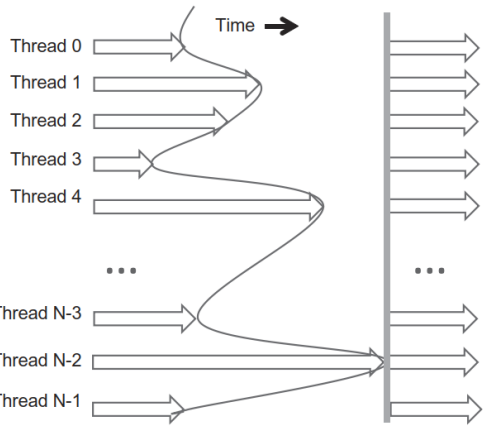

FIGURE 3.10

An example execution timing of barrier synchronization.

Img Src: Programming Massively Parallel Processors by David B.Kirk, Wen-mei W.Hwu

Syntax:

```
__syncthread();
```

How Does __syncthreads() Work?

__syncthreads() is a barrier synchronization function that halts the execution of each thread in a block at a specific point until all other threads in the same block reach the same point. Once every thread in the block has reached the barrier, they are all allowed to proceed to the next instruction simultaneously.

1. **Execution Context:**
    - Consider a block of threads, each executing the same kernel function. At some point in the kernel, you may need to ensure that all threads have completed a particular task before moving on to the next one.

    - For example, suppose threads are writing to a shared memory location that other threads will read from later in the program. The Programmer need to ensure that all threads have completed

their write operations before any thread starts reading.

2. **Barrier Synchronization:**
    - When a thread encounters the *__syncthreads()* function, it stops executing further instructions and waits.
    - Each thread in the block will stop at the *__syncthreads()* call and will remain in this state until all other threads in the block also reach this point.
3. **Synchronization Point:**
    - Once the last thread in the block reaches the *__syncthreads()* call, the barrier is lifted.
    - All threads are then simultaneously allowed to continue execution beyond the *__syncthreads()* point.
4. **Ensuring Data Integrity:**
    - This mechanism ensures that all threads have reached a common execution point, which is particularly important when threads are working with shared resources, like shared memory.
    - For example, in a situation where some threads are writing data to shared memory that other threads will later read, *__syncthreads()* ensures that the writing phase is completed by all threads before any thread begins reading.

The Importance of __syncthreads() in Conditional Statements

With Barrier Synchronization, "**No one is left behind**". *__syncthread()* function ensures that all threads within a block reach the same point in execution before any thread can proceed. When using __syncthreads() within conditional statements, its behavior becomes more complex and requires careful attention to avoid potential deadlocks and undefined behavior.

Conditional Execution and Synchronization Challenges

- **If-Block Synchronization**: When a *__syncthreads()* statement is placed within an if block, it's essential that all threads in the block either execute the if block or skip it entirely. If some threads execute *__syncthreads()* while others do not, the program can enter an undefined state or deadlock. This occurs because the threads that did not execute *__syncthreads()* will continue executing and may reach a point where they depend on data or state changes that were supposed to occur after the synchronization point but never did for the other threads.
- **If-Else Synchronization**: In an if-else structure where *__syncthreads()* appears in both branches, it's critical that either all threads in the block execute the if branch or the else branch, but never a mix of both. This ensures that no matter which path is taken, all threads will reach a *__syncthreads()* call, avoiding synchronization issues.

Temporal Proximity of Threads

Temporal proximity refers to the idea that threads within a block should execute their instructions at nearly the same time.

- **Close Execution Timing:** For synchronization to be effective and efficient, threads within a block must be running in close coordination. If one thread reaches the *__syncthreads()* call much earlier than others, it will have to wait for the slower threads, potentially leading to a situation where execution resources are underutilized while the faster threads are stalled.

- **Avoiding Long Wait Times:** If threads are not temporally aligned in their execution, some threads may reach the synchronization point and be forced to wait for an extended period until the rest of the threads catch up. This waiting time can significantly reduce the overall efficiency of the program, as the GPU's computational resources are being held up.

Resource Allocation and Synchronization

When a thread of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource. This condition ensures the temporal proximity of all threads in a block and prevents excessive or indefinite waiting time during barrier synchronization.

For threads to reach the synchronization point effectively, they must have consistent access to the necessary execution resources. CUDA ensures that all threads within a block are assigned to the same execution resources, which include Streaming Multiprocessors (SMs) and the cores within them.

- **Unified Resource Allocation:** When a thread block is scheduled to run on a Streaming Multiprocessor (SM), all threads in that block are assigned to the same SM. This unified allocation is crucial because it ensures that all threads have the same access to the SM's resources (e.g., shared memory, registers), enabling them to execute in a synchronized manner.

- **Ensuring Progression to Synchronization Points:** Since all threads in a block share the same execution resources, there is a reduced risk of some threads being unable to reach the synchronization point due to resource contention. This ensures that all threads can eventually progress to the synchronization point and the barrier can be effectively lifted.

CUDA's Design Tradeoff: Block-Level Synchronization Only

CUDA's design choice to restrict synchronization to threads within the same block, rather than allowing inter-block synchronization, introduces an important tradeoff that affects how CUDA programs are written and executed.

CUDA only allows threads within the same block to synchronize using *__syncthreads()*; threads in different blocks cannot synchronize with each other. This restriction has both benefits and limitations:

- **Benefit: Flexibility in Execution Order**: Since blocks do not depend on each other for synchronization, the CUDA runtime system has the flexibility to execute blocks in any order. This flexibility allows the GPU to maximize its utilization by scheduling blocks based on the availability of resources, without being constrained by dependencies between blocks.

- **Limitation: No Cross-Block Synchronization**: The inability to synchronize threads across different blocks means that developers must carefully design their algorithms to ensure that critical operations that require synchronization are confined to a single block. This can limit the complexity of some parallel algorithms, which might otherwise benefit from global synchronization.

The tradeoff of restricting synchronization to block-level has a direct impact on CUDA's scalability:

- **Scalable Execution Across Different Hardware Configurations:** Because blocks can be executed independently of one another, CUDA programs can scale across different GPU architectures with varying numbers of SMs and cores. The runtime system can dynamically allocate and execute blocks as resources become available, without needing to worry about the order in which blocks are executed.

- **Efficient Resource Utilization:** By avoiding global synchronization points, CUDA can more effectively utilize the available hardware resources. Blocks can be executed as soon as the necessary resources are available, leading to better overall throughput and reduced idle times.

Resource Limitations of Streaming Multiprocessors (SMs)

The **Streaming Multiprocessor (SM)** is the fundamental unit of computation on a GPU. Each SM is responsible for executing a large number of threads, managing their state, and performing computations in parallel. However, the capability of an SM is constrained by several factors, primarily the number of threads and blocks it can handle simultaneously.

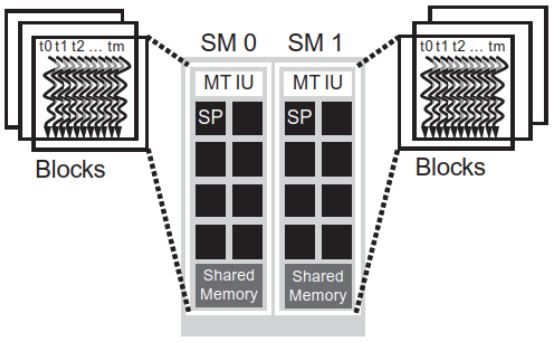

FIGURE 3.12

Thread block assignment to Streaming Multiprocessors (SMs).

Img Src: Programming Massively Parallel Processors by David B.Kirk, Wen-mei W.Hwu

Thread and Block Management by SMs

A key resource limitation of an SM is the number of threads and blocks it can simultaneously track and schedule. This limitation is influenced by the following factors:

- **Hardware Resources**: SMs have a fixed amount of hardware resources, including registers, shared memory, and execution units. These resources are used to track the state of each thread and block, such as their indices, execution status, and intermediate computation results.
- **Registers**: Registers are the fastest type of memory in an SM, used to store temporary variables and intermediate results during the execution of a thread. Each thread requires a certain number of registers, and the total number of registers available in an SM is limited. As the number of threads increases, the available registers must be divided among them, which can limit the total number of threads that can be active simultaneously.
- **Thread and Block Indexes**: To keep track of the execution status of each thread and block, the SM uses internal hardware to manage thread and block indexes. This tracking is necessary to ensure that threads execute correctly, data dependencies are resolved, and synchronization points are respected.

Maximum Number of Threads and Blocks per SM

Each generation of GPU architecture sets specific limits on the number of threads and blocks that can be assigned to an SM. These limits are defined based on the architectural design and available hardware resources of the SM.

Example from Fermi Architecture:

- **Maximum Threads per SM**: In the Fermi architecture, each SM can handle up to 1536 threads simultaneously. This means that no matter how many blocks are assigned to an SM, the total number of threads across all these blocks cannot exceed 1536.
- **Maximum Blocks per SM**: The Fermi SM can accommodate up to 8 blocks at a time. This limitation means that even if there are more blocks available to be executed, only 8 can be assigned to an SM at any given moment.

These limits directly influence how blocks and threads are scheduled on the GPU. For example, if each block contains 256 threads, the Fermi SM could handle up to 6 blocks (256 threads per block * 6 blocks = 1536 threads), but it could not handle more than 8 blocks regardless of the number of threads per block.

Block Configuration and Execution

When you launch a kernel in CUDA, you define the number of blocks and the number of threads per block. The GPU runtime system schedules these blocks and threads onto available SMs based on their resource limitations.

Example Scenarios:

- **Scenario 1: 6 Blocks of 256 Threads Each**:
    - In this scenario, 6 blocks with 256 threads each are assigned to an SM.
    - The total number of threads is 1536 (6 blocks * 256 threads/block), which is within the limit of 1536 threads per SM in the Fermi architecture.
    - All 6 blocks can be scheduled simultaneously on the SM.
- **Scenario 2: 12 Blocks of 128 Threads Each**:
    - Here, 12 blocks with 128 threads each are considered.
    - The total number of threads is 1536 (12 blocks * 128 threads/block), which is still within the thread limit.

- However, only 8 blocks can be assigned to the SM at any time due to the block limit, meaning 4 blocks will have to wait until the first 8 blocks complete their execution.

Example Calculation: Total Threads on a GPU

To further illustrate these concepts, let's calculate the total number of threads that can be simultaneously active on a hypothetical CUDA device.

Assumptions:

- The GPU has 30 SMs.
- Each SM can accommodate up to 1536 threads.
- The block size is configured such that each block contains 256 threads.

Calculation:

1. **Threads per SM**: Each SM can handle up to 1536 threads.
2. **Total Threads across all SMs**: With 30 SMs, the total number of threads that can be simultaneously active on the GPU is:
    - 1536 threads/SM * 30 SMs = 46,080 threads

This means that at any given moment, the GPU can have up to 46,080 threads running concurrently, distributed across its 30 SMs.

As we have explored the concept of thread synchronization and the limitations imposed by Streaming Multiprocessors (SMs) in CUDA, it becomes clear that optimizing a CUDA program requires a deep understanding of the underlying hardware capabilities.

To achieve the full potential of the GPU, developers must program their applications to match the specific characteristics of the device they are working with. This is where querying device properties comes into play. By accessing detailed information about the GPU's configuration—such as the number of SMs, available memory, and maximum thread counts—developers can make informed decisions on how to structure their kernels and allocate resources effectively.

In the next section, we will delve into the methods and tools provided by CUDA to query these device properties, enabling a more fine-tuned and efficient utilization of the GPU.

How to Query Device Properties

Each GPU can have a different number of Streaming Multiprocessors (SMs), varying amounts of memory, and different limits on the number of threads and blocks it can handle. These factors directly influence how a kernel should be configured and executed. For instance, knowing the number of SMs can help in determining the optimal number of blocks to launch, while understanding the maximum number of threads per block can guide decisions about kernel design and workload distribution.

CUDA provides a built-in mechanism for querying the properties of available devices through its runtime API. This capability allows the host code to determine the number of CUDA-capable devices in the system and retrieve detailed information about each device's configuration.

Getting the Number of Devices

The first step in querying device properties is to determine how many CUDA-capable devices are present in the system. This can be done using the *cudaGetDeviceCount()* function, which returns the number of devices available:

```
int dev_count;
cudaGetDeviceCount(&dev_count);
```

Here, *dev_count* will store the number of CUDA devices available, which can range from zero (if no CUDA-capable device is present) to several, depending on the system's configuration.

Retrieving Device Properties

Once you know the number of devices, the next step is to retrieve the properties of each device using the ***cudaGetDeviceProperties()*** function. This function populates a *cudaDeviceProp* structure with various details about the device, such as the number of SMs, the maximum number of threads per block, and memory specifications:

Syntax:

```
__host__cudaError_t cudaGetDeviceProperties ( cudaDeviceProp* prop, int  device )
```

Example usage:

```
cudaDeviceProp dev_prop;
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&dev_prop, i);
    // You can now access dev_prop fields to understand the device's capabilities
}
```

Few Fields in cudaDeviceProp:

The *cudaDeviceProp* structure provides a wide array of information about the GPU, but some fields are particularly important when it comes to designing and optimizing CUDA applications.

1. maxThreadsPerBlock: Maximum number of threads per block

One of the most critical fields in the *cudaDeviceProp* structure is *maxThreadsPerBlock*, which indicates the maximum number of threads that can be launched in a single block on the queried device. This limit is essential for configuring the thread block size in your CUDA kernels.

- While many devices allow up to 1024 threads per block, this is not a universal standard. Some older or lower-end GPUs may support fewer threads per block, whereas future GPUs might support more. Therefore, it is crucial to query this property for the specific device on which your application will run.

- Knowing the maxThreadsPerBlock helps in designing kernels that can maximize occupancy while staying within the hardware constraints. For example, if a device allows 1024 threads per block, you might design your kernel to use this maximum to fully utilize the SMs. However, if the device allows fewer threads, you would need to adjust your kernel to prevent launching more threads than the device can handle.

2. multiProcessorCount: Number of multiprocessors on device

The *multiProcessorCount* field reveals the number of Streaming Multiprocessors (SMs) on the device. This number is directly linked to the parallel processing power of the GPU.

- The number of SMs determines how many blocks can be processed simultaneously. A higher *multiProcessorCount* generally means that the GPU can handle more parallel workloads, leading to increased throughput and faster execution times for parallel tasks.
- By understanding the *multiProcessorCount*, you can better strategize how to launch your kernels. For instance, on a device with many SMs, you might launch more blocks to fully utilize the parallel capabilities, whereas on a device with fewer SMs, you might focus on optimizing the thread count within each block.

3. clockRate: (Depricated)

The *clockRate* field indicates the clock frequency of the GPU, measured in kilohertz (kHz). This rate is a key factor in determining the computational speed of the device.

The clock rate, combined with the number of SMs, provides a good indication of the GPU's execution capacity. A higher clock rate typically allows the GPU to perform more operations per second, which is particularly beneficial for compute-intensive tasks.

4. maxThreadsDim: Maximum Size of Each Dimension of a block

The *maxThreadsDim* array in the *cudaDeviceProp* structure defines the maximum number of threads that can be allocated along each dimension (x, y, z) of a thread block.

For applications that process multi-dimensional data, such as matrices or 3D grids, this property is crucial. It allows you to configure your thread blocks to cover the data space effectively while staying within the hardware limits.

5. maxGridSize: Maximum Size of each dimension of a grid

Similar to *maxThreadsDim*, the *maxGridSize* array specifies the maximum dimensions of a grid (x, y, z) that can be launched on the device.

The grid size determines the total number of blocks that can be launched across each dimension. This is particularly important for large-scale parallel applications that need to process extensive datasets.

6. totalGlobalMem: Global Memory available on device in bytes

The *totalGlobalMem* field indicates the total amount of global memory available on the device, which is crucial for storing data that needs to be processed by the kernels.

The amount of global memory affects how much data can be loaded and processed at one time. Applications with large datasets must be designed to manage this memory efficiently, possibly by breaking the data into smaller chunks or using memory pooling techniques.

7. warpSize: Warp Size in threads

The *warpSize* field specifies the number of threads that make up a warp, which is the smallest unit of execution in CUDA. Typically, the warp size is 32 threads.

In CUDA, warps are the fundamental units of thread scheduling. The warp size impacts how threads are executed in parallel and how memory accesses are merged.

The *CudaDeviceProp* fields will be discussed in more detail in upcoming articles.

~~ To be Continued ~~

Reference: Programming Massively Parallel Processors by David B.Kirk, Wen-mei W.Hwu