PMC Lecture 13

Gianmaria Romano

26 November 2025

Chapter 6

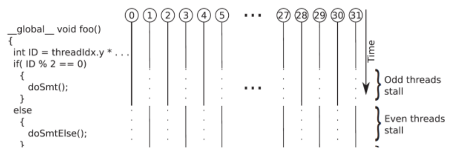
Compute Unified Device Architecture

6.1 Thread scheduling in CUDA programs

When dealing with CUDA programs, each thread runs on a specific streaming processor within a single streaming multiprocessor, which hosts all the threads contained a given block.
In particular, a streaming multiprocessor will (concurrently) run a block on a single instruction and, when done, it will start running a new block.

6.1.1 Scheduling using wraps

Threads within a block are normally grouped in contiguous warps, which typically contain 32 threads.
However, since threads within the same warp must always execute the same assembly task, a warp might experience a divergence when, typically upon a selective branch, some threads within the same warp try to perform different tasks.
In this case, the program must sequentially analyse each possible branch while stalling the threads that do not enter the current path, often leading to a lower usage of the resources.
Example: A warp divergence can happen if the code contains a branch based on whether the current thread's ID is an odd number or an even number.



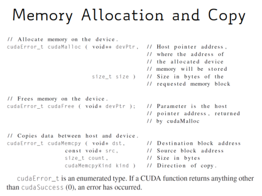
```
__global__ void foo()
{
  int ID = threadIdx.y * . . .
  if( ID % 2 == 0)
    {
      doSmt();
    }
  else
    {
      doSmtElse();
    }
```

6.1.2 Context switching for GPUs

A streaming multiprocessor can typically contain more warps that what it is able to execute concurrently, allowing it to efficiently switch between them.
In fact, each thread has a private execution context that is stored directly on the chip, meaning that context switching will not require memory transfers as on CPUs.
In particular, whenever a warp tries to execute an instruction that must wait for a long-latency operation, that warp will not be selected for execution, choosing another ready warp instead.
Therefore, the hardware is able to keep the streaming multiprocessor fully utilized as it can always find a warp that is ready to run.
Example: Suppose a CUDA device supports up to 8 blocks per streaming multiprocessor, with each block containing a maximum of 512 threads, and 1024 threads per streaming multiprocessor.
Using simple 8×8 blocks yields 64 threads per block and, since a streaming multiprocessor can support 8 blocks, it will allow a total of $64 \times 8 = 512$ resident threads, meaning that the device will not fully utilize its resources.
Using larger 16×16 blocks yields 256 threads per block and, since a streaming multiprocessor can support 8 blocks, it will allow a total of $256 \times 8 = 1024$ resident threads, meaning that the device will be able to fully utilize its resources.
Lastly, using 32×32 blocks yields 1024 threads per block, which, however, cannot be supported by the streaming multiprocessor.

6.2 Handling memory accesses in CUDA

Remember that, when performing memory allocation operations, data allocated on host memory cannot be seen from the GPU, and viceversa, therefore requiring explicit data transfers during communications.



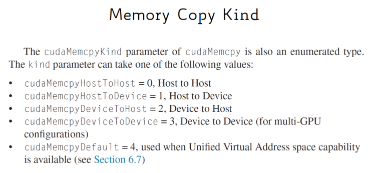
Memory Allocation and Copy

```
// Allocate memory on the device.
cudaError_t cudaMalloc ( void** devPtr,   // Host pointer address,
                                          // where the address of
                                          // the allocated device
                                          // memory will be stored
                         size_t size )    // Size in bytes of the
                                          // requested memory block

// Frees memory on the device.
cudaError_t cudaFree ( void* devPtr );    // Parameter is the host
                                          // pointer address, returned
                                          // by cudaMalloc

// Copies data between host and device.
cudaError_t cudaMemcpy ( void* dst,       // Destination block address
             const void* src,             // Source block address
             size_t count,                // Size in bytes
             cudaMemcpyKind kind )        // Direction of copy.
```

`cudaError_t` is an enumerated type. If a CUDA function returns anything other than `cudaSuccess` (0), an error has occurred.



Memory Copy Kind

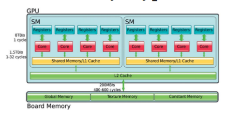
The `cudaMemcpyKind` parameter of `cudaMemcpy` is also an enumerated type. The `kind` parameter can take one of the following values:

- `cudaMemcpyHostToHost` = 0, Host to Host
- `cudaMemcpyHostToDevice` = 1, Host to Device
- `cudaMemcpyDeviceToHost` = 2, Device to Host
- `cudaMemcpyDeviceToDevice` = 3, Device to Device (for multi-GPU configurations)
- `cudaMemcpyDefault` = 4, used when Unified Virtual Address space capability is available (see Section 6.7)

6.3 The CUDA memory hierarchy

A CUDA GPU makes use of several memory components, some of which are implemented on the chip and others off the chip.

Memory Types



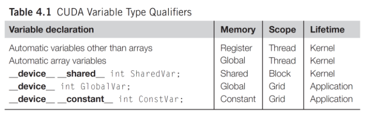


Table 4.1 CUDA Variable Type Qualifiers

Variable declaration	**Memory**	**Scope**	**Lifetime**
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Global	Thread	Kernel
__device__ __shared__ `int SharedVar;`	Shared	Block	Kernel
__device__ `int GlobalVar;`	Global	Grid	Application
__device__ __constant__ `int ConstVar;`	Constant	Grid	Application

6.3.1 Registers

Similarly to CPUs, GPU registers are used to store a thread's local variables.
Generally speaking, the available registers on a streaming multiprocessors are evenly distributed among all resident threads, but, whenever the register demand of a kernel exceeds the number of available registers per thread, some local variables will be spilled to the off-chip global memory, which, however, is much slower to access.
Since register usage affects how many threads can be active on a streaming multiprocessor, it will also influence the number of resident warps that can be supported by the streaming multiprocessor.
For this reason, it is possible to define occupancy as the ratio between the number of resident warps and the maximum number of warps that can be supported by the streaming multiprocessor.
Ideally, occupancy should be close to 1 so that the hardware has enough resources to effectively hide latencies.