# PMC Lecture 17

Gianmaria Romano

10 December 2025

# Chapter 6

# Compute Unified Device Architecture

## 6.1 The CUDA memory hierarchy

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)

From a more general point of view, a CUDA GPU makes use of several memory components, which can be implemented on the chip or off the chip.

Memory Types

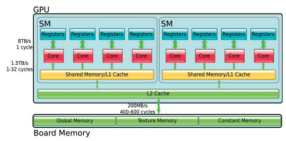


**Table 4.1** CUDA Variable Type Qualifiers

| 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 |

### 6.1.1 The global memory

The global memory is an off-chip memory component that can store large amounts of data but is physically distant from the processors.
For this reason, accessing data from the global memory requires higher latency compared to on-chip memory components.

### 6.1.2 Registers

Similarly to CPUs, GPU registers are used to store a thread's local variables.
Generally speaking, the available registers on a streaming multiprocessor 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.

### 6.1.3 On-chip shared memory

A shared memory is an on-chip memory component that is shared among threads within a block.
Despite having smaller capacity, the shared memory has a much lower latency compared to other (off-chip) components, making it useful for storing frequently accessed data that, otherwise, would take much longer to fetch.
**Example:** The following code provides an implementation of a stencil, which is a geometric tool that is often used for solving mathematical problems:

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

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS]; // RADIUS adds a padding on both sides.
    int gindex = threadIdx.x + blockIdx.x * blockDim.x; // Global index in the grid.
    int lindex = threadIdx.x + radius; // Local index in the shared memory.

    // Read input elements into the shared memory.
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    __syncthreads(); // Synchronize all threads within a block to avoid dependencies.

    // Apply the stencil to perform computations.
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++) {
        result += temp[lindex + offset];
    }

    // Store the result.
    out[gindex] = result;
}
```

### 6.1.4 The constant memory

A constant memory is a read-only memory component that is used to store constant values that do not change during execution.
In particular, the constant memory provides performance benefits because it is cached and supports broadcasting when threads within a warp need to access the same memory location.

```
__constant__ type variable_name; // static
cudaMemcpyToSymbol(variable_name, &host_src, sizeof(type), cudaMemcpyHostToDevice);
// warning: cannot be dynamically allocated
```

- data will reside in the constant memory address space
- has static storage duration (persists until the application ends)
- readable from all threads of a kernel

## 6.2 Performance estimation for GPUs

Generally speaking, GPU performance is measured in terms of the number of floating-point operations it can perform in one second.
In practice, however, the performance of a program is limited by the data transfer rate, causing the application to be "memory-bound".
For this reason, it can come in handy to define the compute-to-global-memory-access ratio, which measures the number of floating-point operations that are executed per global memory access within a certain region of the program.