PMC Lecture 18

Gianmaria Romano

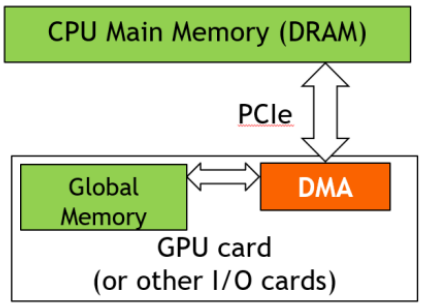
16 December 2025

Chapter 6

Compute Unified Device Architecture

6.1 Host-device data transfers in CUDA

Data transfers in CUDA rely on direct memory access hardware to achieve high efficiency as this approach allows to handle data transfers without involving the CPU, which is able to perform other tasks concurrently.

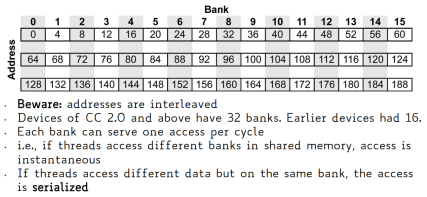


However, since direct memory access works on physical memory accesses, unmanaged memory access might cause conflicts if different processes try to access overlapping regions in the physical memory.
For this reason, many modern systems make use of virtual memory management, which maps the virtual memory space of a process onto physical memory pages, in order to provide memory isolation and safety during data transfers.
In this context, CUDA enables efficient data transfers through page-locked memory, which consists of virtual memory pages that are guaranteed to stay in the physical memory can cannot be swapped by the system, ensuring stable physical addresses that allow the hardware to directly access the host memory during

CPU-GPU data transfers.
Therefore, any source or destination located in the host virtual memory must be allocated as page-locked memory using dedicated allocation functions.
If this is not the case, CUDA can copy the data into a temporary page-locked buffer before performing the actual transfer, providing safety at the cost of some overhead.

6.2 Bank conflicts in the shared memory

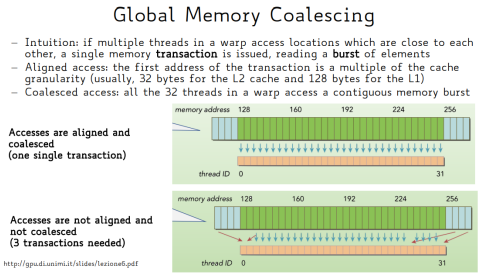
Generally speaking, the shared memory of a CUDA device is organized into a set of memory banks.
In particular, memory address are typically interleaved across different blocks so that different threads accessing different banks can perform their accesses simultaneously.
On the other hand, in the event of a bank conflict, which happens when different threads try to access different addresses in the same bank, the accesses must be serialized, leading to reduced performance.



N.B.: If all threads in a warp try to access the same memory address, it is more convenient for the address to perform a "broadcast read" and propagate the value to all threads, therefore avoiding any conflict

6.3 Global memory coalescing

When a thread accesses a global memory location, the hardware actually reads a contiguous block of memory addresses.
For this reason, when all threads in a warp try to execute a load instruction, the hardware should check whether these threads are accessing contiguous location and, if this is the case, it will coalesce the individual accesses into a single memory access, therefore improving efficiency.



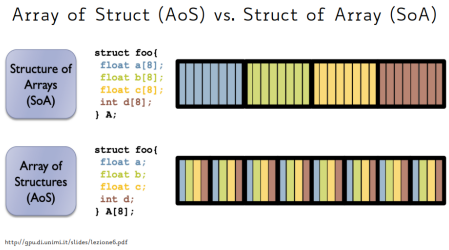
From a broader perspective, global memory accesses in CUDA support two main types of data loads:

- **Cached load:** used by default for devices with a L1 cache, this approach checks whether the desired item is located in the cache before accessing the global memory.
- **Non-cached load:** used for devices without a L2 cache, this approach checks whether the desired item is located in the L2 cache before accessing the global memory.

N.B.: In some cases, it is possible to disable the L1 cache and directly use non-cached loads in order to improve performance.

6.3.1 Data structures for coalesced accesses

When dealing with coalesced memory accesses, data in the memory can be organized either as an array of structs, in which each element is a complete structure containing all data fields, or as a struct of arrays, in which each data field is stored in a separate array.



Generally speaking, GPUs tend to benefit more from using a struct of arrays because this layout allows threads in a warp to access contiguous memory addresses for a specific field, therefore improving efficiency for coalesced accesses. On the other hand, CPUs tend to benefit more from using an array of structs because this layout improves locality when dealing with all fields of a single element.