GLOBAL MEMORY

GLOBAL MEMORY

- GPU global memory is global because it’s writable from both the GPU & CPU.
- It can be accessed from any device on the PCI-E bus.
- GPU cards can transfer data to and from one another, directly, without needing the CPU.
- The memory from the GPU is accessible to the CPU in one of three ways:
  - Explicitly with a blocking transfer.
  - Explicitly with a nonblocking transfer.
  - Implicitly using zero memory copy.

GLOBAL MEMORY

- The usual model of execution involves:

  the CPU transfers a block of data to the GPU,

  the GPU kernel processing it,

  and then the CPU initiating a transfer of the data back to the host memory.
- A slightly more advanced model of this is where we use streams to overlap transfers and kernels to ensure the GPU is always kept busy(fig)
- The same way that we can pipeline kernels, as is shown in the memory accesses are pipelined.

GLOBAL MEMORY

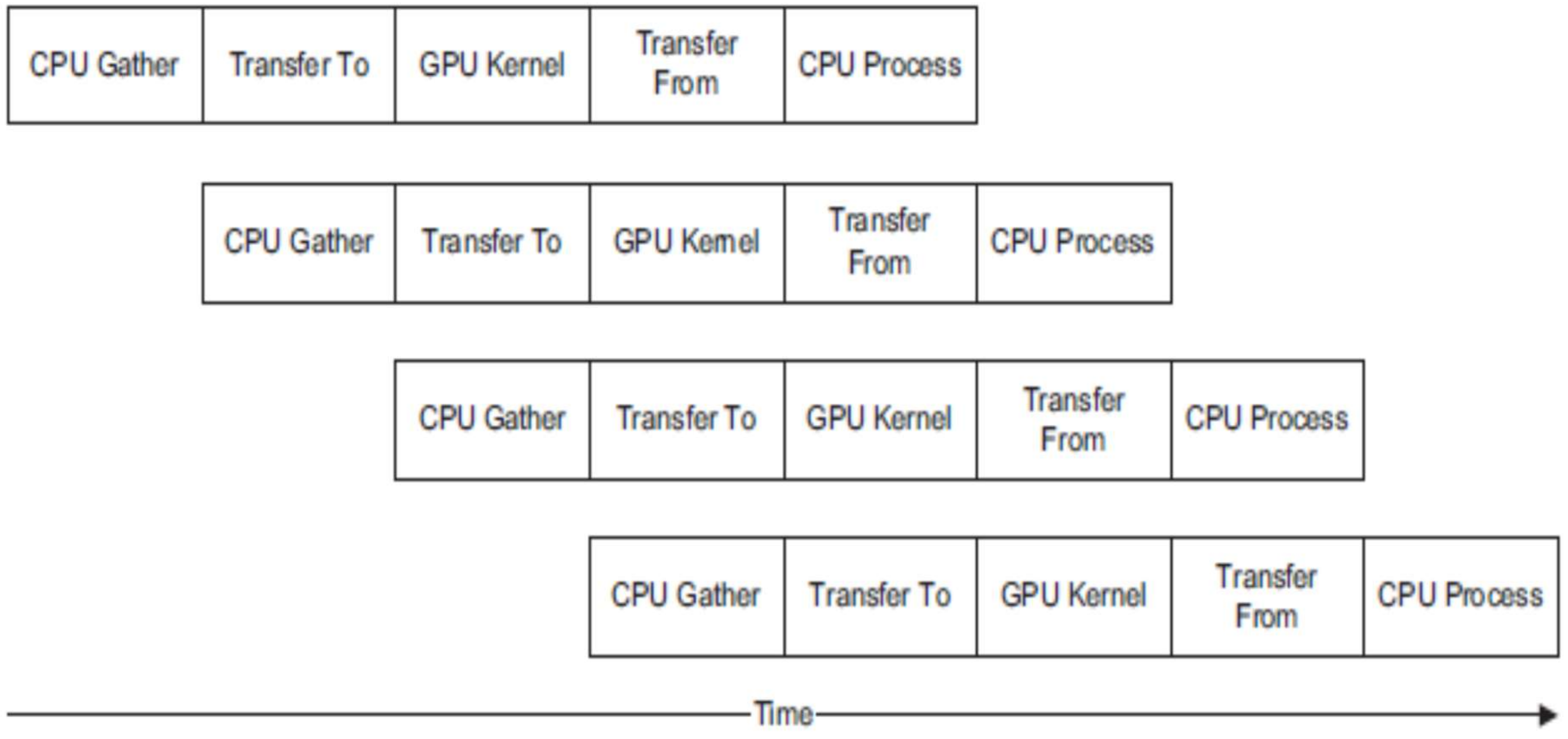


Fig : Overlapping kernel and memory transfers

GLOBAL MEMORY

- Coalescable pattern is where all the threads access a contiguous and aligned memory block.
- Here we have shown Addr as the logical address offset from the base location, assuming we are accessing byte-based data.
- TID represents the thread number
- Assuming we're accessing a single precision float or integer value, each thread will be accessing 4 bytes of memory.
- Memory is coalesced on a warp basis meaning we get 32 X 4 = 128 byte access to memory.
- Coalescing sizes supported are 32, 64, and 128 bytes, meaning warp accesses to byte, 16- and 32- bit data will always be coalesced if the access is a sequential pattern and aligned to a 32-byte boundary.

GLOBAL MEMORY

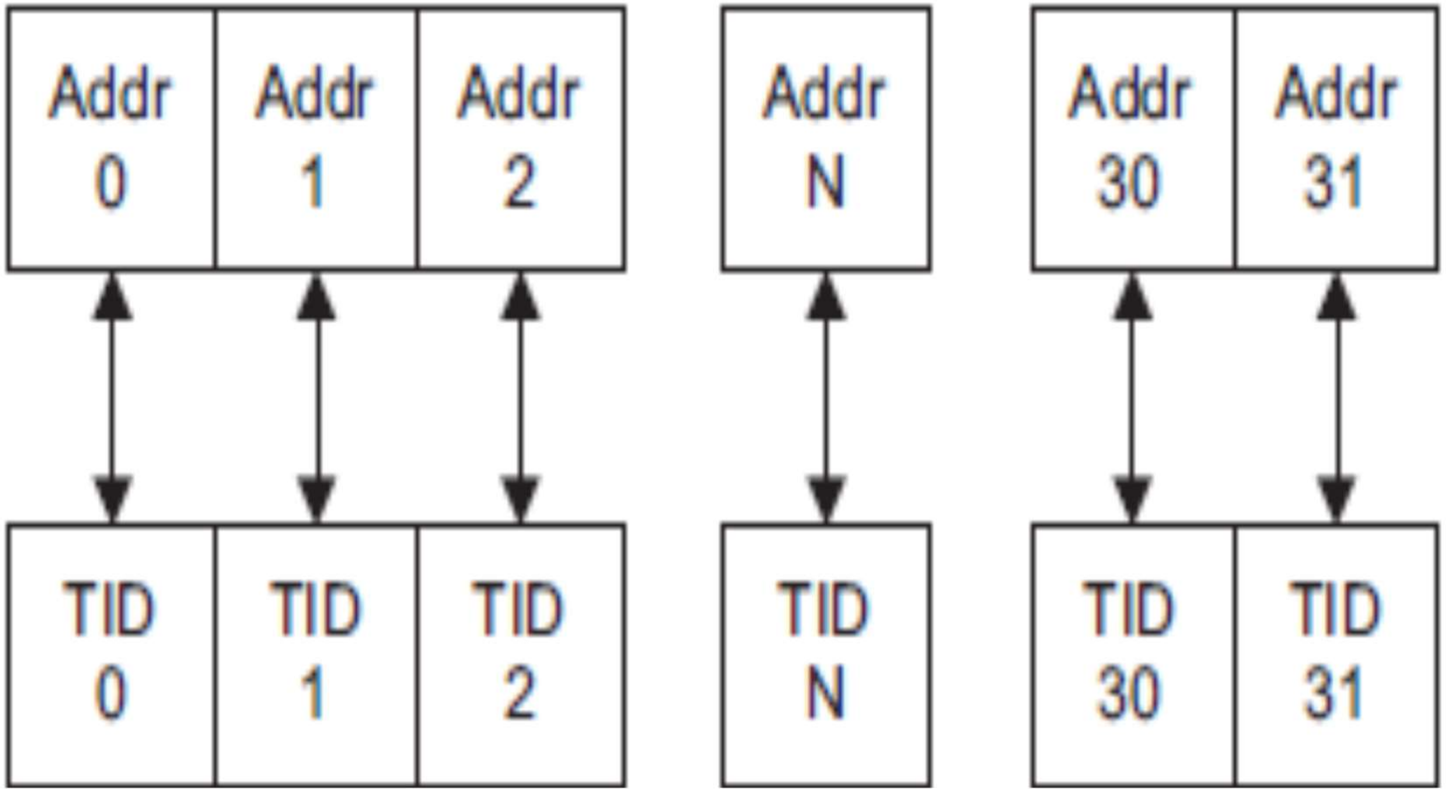


Fig: Addresses accessed by thread ID

GLOBAL MEMORY

- The alignment is achieved by using a special malloc instruction, replacing the standard ***cudaMalloc*** with ***cudaMallocPitch,*** which has the following syntax:

*extern __host__ cudaError_t CUDARTAPI cudaMallocPitch(void **devPtr, size_t *pitch, size_t width, size_t height);*

- This translates to cudaMallocPitch (pointer to device memory pointer, pointer to pitch, desired width of the row in bytes, height of the array in bytes).
- If you have an array of 100 rows of 60 float elements, using the conventional cudaMalloc, you would allocate 100 x 60x sizeof(float) bytes, or 100 x 60 x 4 = 24,000 bytes.

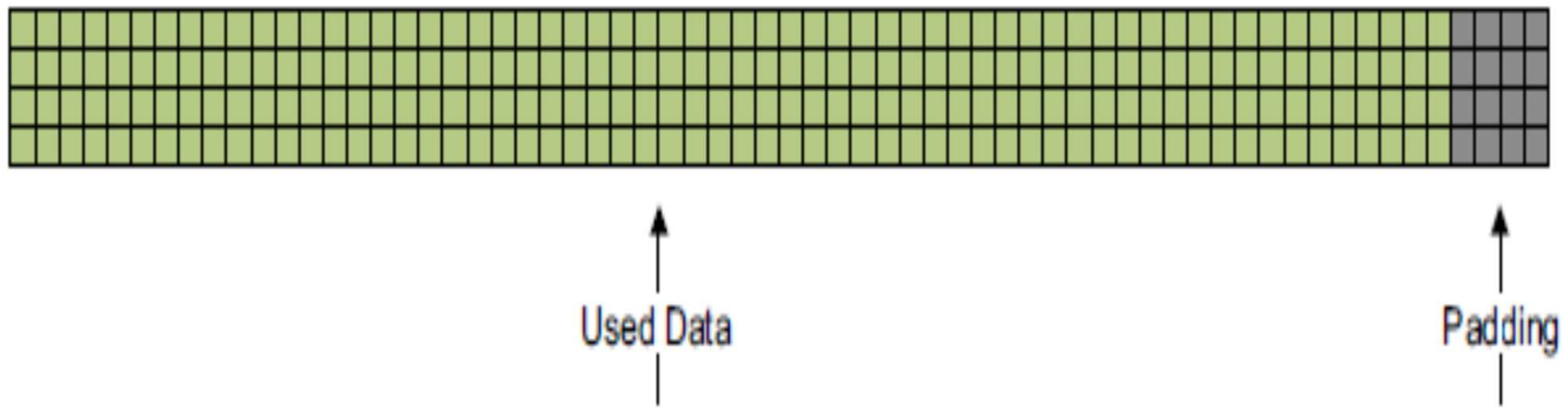
GLOBAL MEMORY

- Accessing arrayindex [1][0] (i.e., row one, element zero) would result in noncoalesced access. This is because the length of a single row of 60 elements would be 240 bytes, which is of course not a power of two.
- Using the ***cudaMallocPitch*** function the size of each row is padded by an

amount necessary for the alignment requirements of the given device.

- In our example, it would in most cases be extended to 64 elements per row, or 256 bytes.
- The pitch the device actually uses is returned in the pitch parameters passed to cudaMallocPitch.

GLOBAL MEMORY



- Fig: Padding Achieved using CudaMallocPitch

GLOBAL MEMORY

- **High Bandwidth**: GPU global memory typically has a much higher bandwidth compared to system RAM, allowing for faster data access by the GPU cores.
- **Parallel Access**: It is designed to be accessed in parallel by multiple GPU cores simultaneously. This is crucial for efficient parallel processing, which is the primary strength of GPUs.
- **Large Capacity**:Modern GPUs have substantial global memory capacities, often several gigabytes or even terabytes, depending on the model and type of GPU.
- **Data Transfer:** Data is transferred between the CPU's main memory and the GPU's global memory when processing tasks are offloaded to the GPU. This transfer can be a potential bottleneck, and minimizing data transfer is often a consideration in optimizing GPU applications.

GLOBAL MEMORY

- **Heterogeneous Memory Architecture (HMA):** Some GPUs may have heterogeneous memory architectures that incorporate various types of memory, such as High Bandwidth Memory (HBM), Graphics Double Data Rate (GDDR) memory, and traditional GDDR or DDR RAM. These different memory types can offer varying levels of performance and energy efficiency.

- **Explicit Memory Management:** In many GPU programming models like CUDA (Compute Unified Device Architecture) and OpenCL, developers need to explicitly manage data movement between CPU and GPU memory. This explicit control allows developers to optimize data transfers and memory usage for specific tasks.