LESSON 4

Global Memory

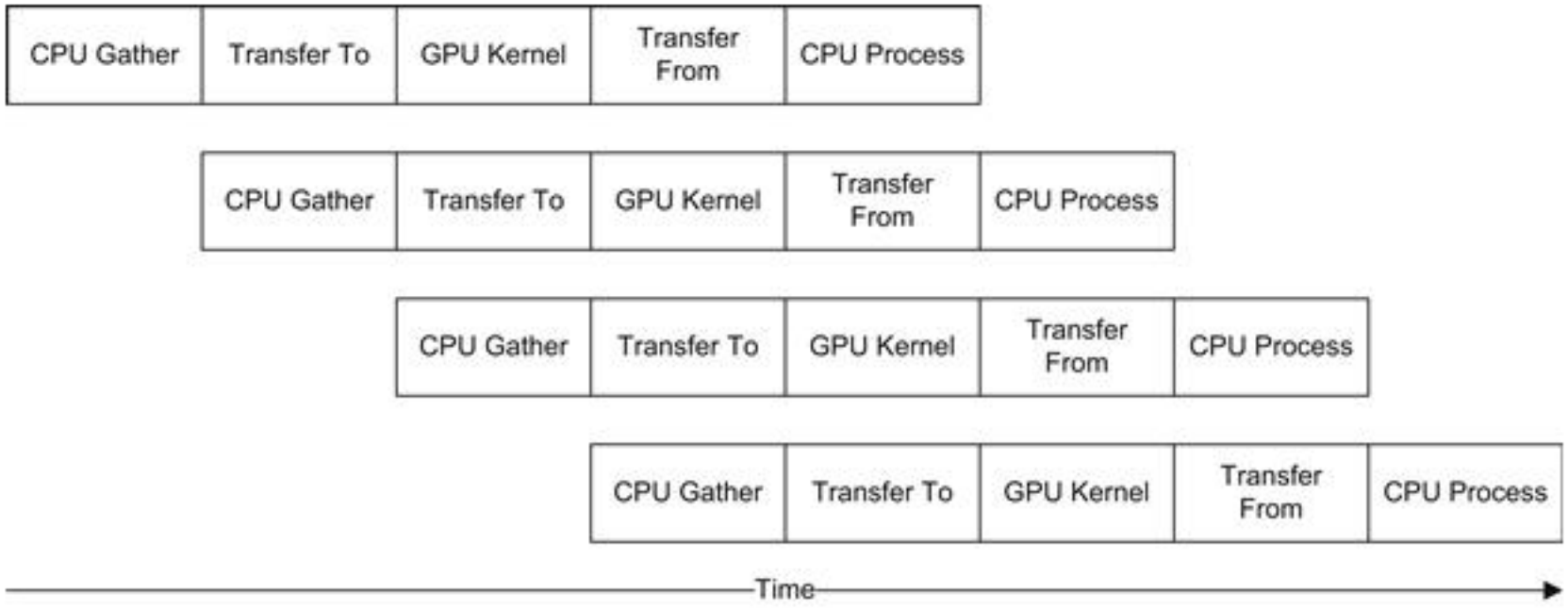
TOPICS

- SoA vs AoS

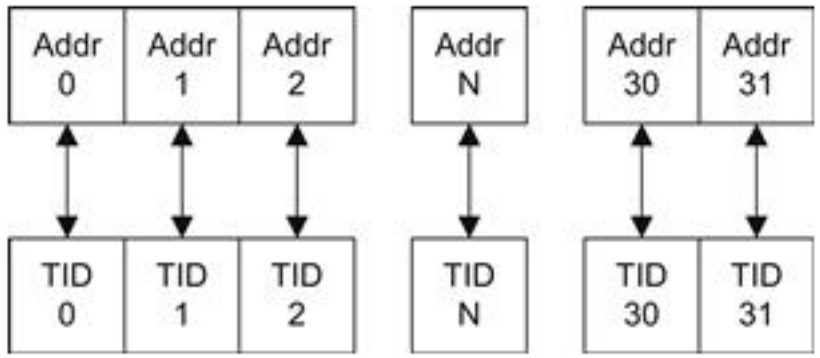
- Key Words: *Stream, deviceOverlap, cudaStream_t, cudaStreamCreate, cudaStreamDestroy, page-lock, pinned memory, pageable, cudaMalloc(),malloc(), cudaHostAlloc(), cudaMemcpyAsync(), cudaStreamSynchronize(), cudaFreeHost().*

Throughput

- GPU global memory is global because it is writable from both the device and the host.



- Memory is coalesed on a warp basis, meaning we get 32 x 4 = 128 bytes acess to memory. Coalescing sizes supported are 32, 64 and 128 bytes.

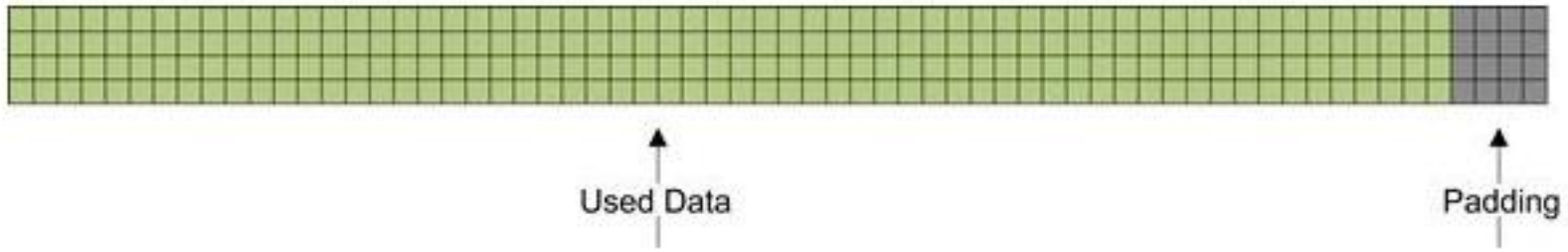


- Noaligned access result in multiple memory fetches being issued. While waiting for memory fetch, all the threads in a warp are stalled until all memory fetches are returned from the hardware.

- Memory accesses are much faster if the memory items are properly aligned. For example, while accessing 2D arrays, accessing the elements is faster if each row starts at 64-byte boundaries.

- The CUDA API provides features for "padding" the elements with extra bytes so that the final size of the items is always a multiple of 64 or 32.

- The alignment is achived by using a special malloc instruction, replacing the standard cudaMalloc() with cudaMallocPitch(). CudaMallocPitch() function determines the best pitch and returns it to the program.



- For global memory, as a general rule, the more scattered the addresses are, the more reduced the throughput is.

- When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads.

Example

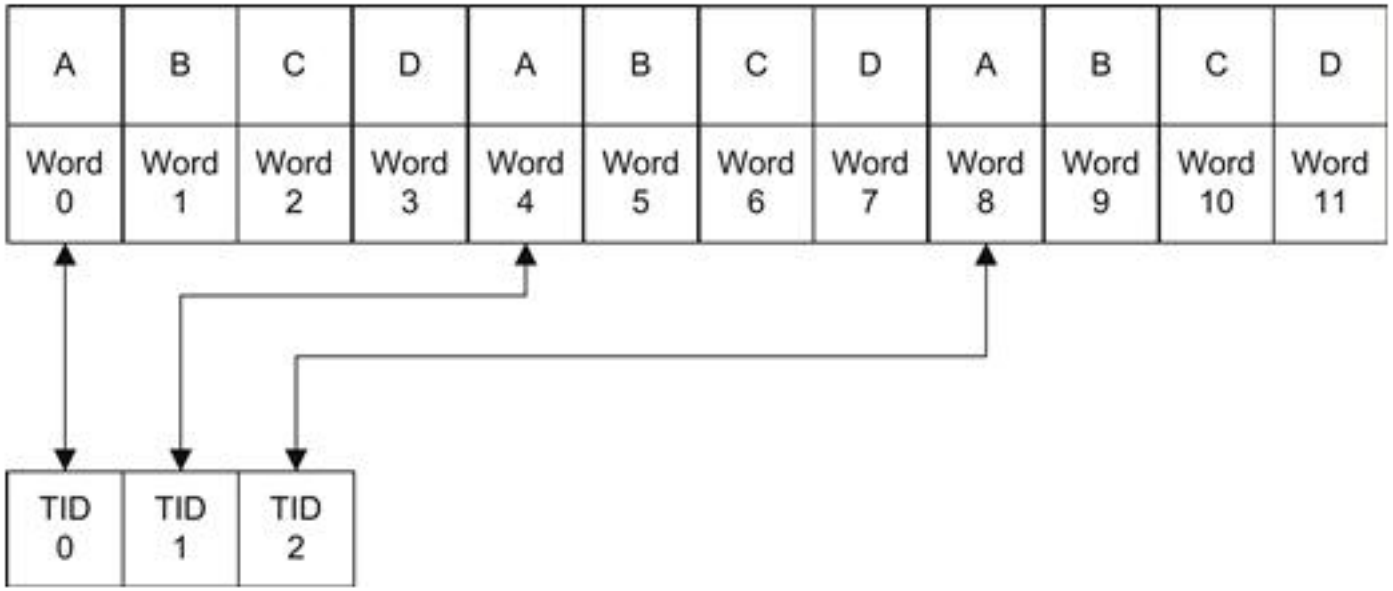
- 00matrix_pitch: This example show how to use the cudaMallocPitch() function.

AoS vs SoA

- Many applications store data as arrays of structures (AoS) that can lead to coalescing issues. From a GPU performance perpective, it is preferible to store data as a structure of arrays (SoA).
- Arranging data in this fashion leads to coalescing issues as the data are interleaved.
- Arranging data as a SoA makes full use of the memory bandwidth even when individual elements of the structure are utilized.
- There is no data interleaving; this data structure should provide coalesced memory access and high global memory performance.

```
typedef struct{

    int a;
    int b;
    int c;
    int d;
} aos;
```



Example

- 01aos_vs_soa: This example has a benchmark that compare the different memory access using AoS and SoA in the device and host.

Mapped Memory

- The way to make use of this memory is changing cudaHostAllocDefault to cudaHostAllocMapped. This flag tells the runtime that we intend to access this buffer from the GPU.

- This memory does not require copies to and from the GPU, we refer to it as zero-copy memory.

- It is necessary to know if the device is capable of use map host memory, we can test this option with the property canMapHostMemory.

- The flag cudaHostAllocWriteCombined indicates that the runtime should allocate the buffer as write-combined with respect to the CPU cache, which means that frees up the host's L1 and L2 cache resources, making more cache available to the rest of the application.

- Reading from write-combining memory from the host is prohibitively slow, so write-combining memory should in general be used for memory that the host only writes to.

```
// allocate host locked memory
HANDLER_ERROR_ERR(cudaHostAlloc((void**)&h_a.elements, FULL_ARRAY_BYTES,
                                cudaHostAllocWriteCombined | cudaHostAllocMapped ));
HANDLER_ERROR_ERR(cudaHostAlloc((void**)&h_b.elements, FULL_ARRAY_BYTES,
                                cudaHostAllocWriteCombined | cudaHostAllocMapped ));
```

- The GPU can access to this memory, but it has different virtual memory space , so it is important to get the CPU pointer for the memory, this is through the cudaHostGetDevicePointer().

```
HANDLER_ERROR_ERR(cudaHostGetDevicePointer(&d_a.elements, h_a.elements,0));
HANDLER_ERROR_ERR(cudaHostGetDevicePointer(&d_b.elements, h_b.elements,0));
```

- The contents of zero-copy memory are undefined during the execution of a kernel that potentially makes changes to its content, so it is important to stablish a synchronization using streams or events.
- .
- With streams:

```
HANDLER_ERROR_ERR( cudaStreamSynchronize( stream1 ) );
```

- Without streams:

```
HANDLER_ERROR_ERR( cudaDeviceSynchronize());
```

- It is necessary to free the memory.

```
// free host memory
HANDLER_ERROR_ERR(cudaFreeHost( h_a.elements ));
HANDLER_ERROR_ERR(cudaFreeHost( h_b.elements ));
```

- To finish, we need to call the cudaSetDeviceFlags(), by passing the flag cudaDeviceMapHost to indicate that we want the device to be allowed to map host memory flag, before any other CUDA call is performed.

- Otherwise, cudaHostGetDevicePointer() will return an error.

```
//This flag must be set in order to allocate pinned
//host memory that is accessible by the device
HANDLER_ERROR_ERR( cudaSetDeviceFlags(cudaDeviceMapHost) );
```

Mapped Memory

- cudaHostGetDevicePointer() also returns an error if the device does not support mapped page-locked host memory.

- Advantages

  - There is no need to allocate a block in device memory and copy data between this block and block in host memory, the data transfers are implicitily performed as it is needed by the kernel.

  - There is no need to use streams to overlap data transfers with the kernel in execution.

  - Mapped memory is able to exploit the full duplex of the PCIE bus by reading and writing at the same time.

- Disadvantages

- The page-locked memory is shared with the host and the device, any application must to avoid writing on both side simultaneously.

Example

- 02mapped_memory: This example show the use of mapped memory.

Practice

- 03matrix_transpose: Use the techniques already explained to inprove the performance. It is up to you to the way to implement it.