

Parallel Programming

CUDA Streams

Phạm Trọng Nghĩa
ptnghia@fit.hcmus.edu.vn

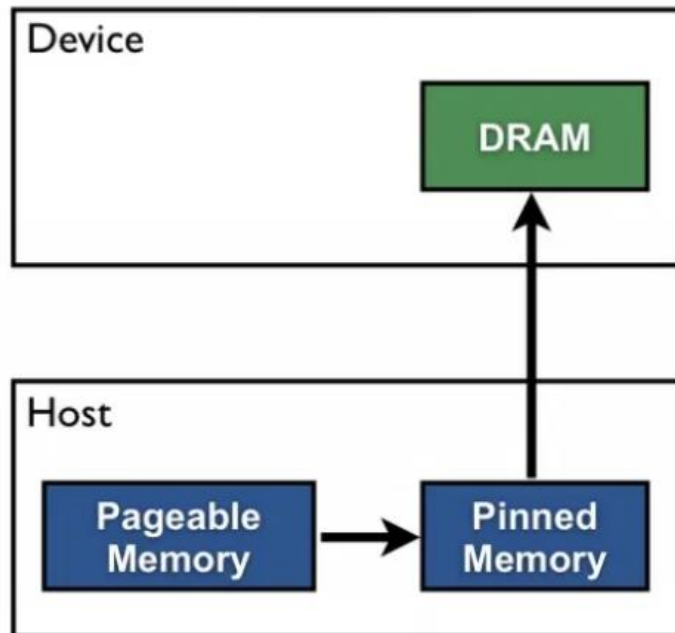
Host device data transfer

Memory Allocation Types

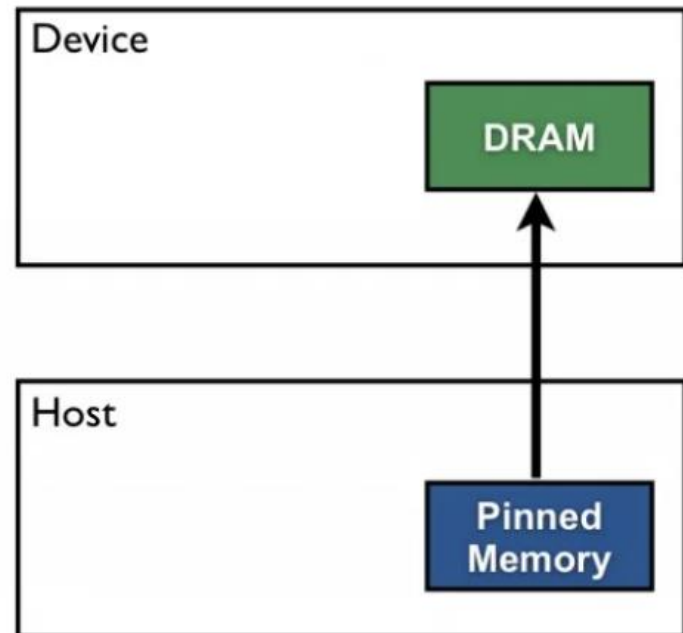
- Device memory (cannot be paged)
- (Host) Pageable memory
- (Host) Pinned memory
- (Both) Mapped memory
- (Both) Unified memory

What we learn

Pageable Data Transfer



Pinned Data Transfer

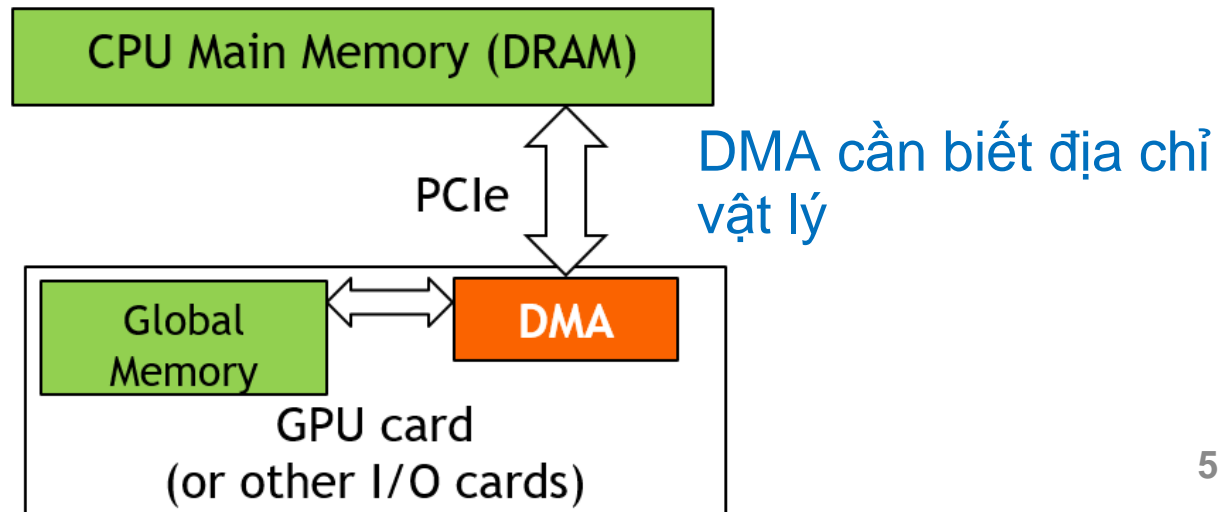


Virtual Memory Management

- Modern computers use **virtual memory management**
 - Many virtual memory spaces mapped into a single physical memory
 - **Virtual addresses** (pointer values) are translated into **physical addresses**
- Not all variables and data structures are always in the physical memory
 - Each virtual address space is divided into pages that are mapped into and out of the physical memory
 - Virtual memory pages can be mapped out of the physical memory (page-out) to make room
 - Whether or not a variable is in the physical memory is checked at address translation time

Memory Allocation Types

- **Pagable memory** is transferred using the host CPU (memory Map I/O)
- **Pinned memory** is transferred using the **DMA** engines
 - Frees the CPU for asynchronous execution
 - Achieves a higher percent of peak bandwidth
- ***cudaMemcpy()*** use **DMA (Direct Memory Access) hardware**
 - Hardware unit specialized to transfer a number of bytes requested by OS
 - Between **physical memory address** space regions
 - Uses system interconnect, typically PCIe in today's systems



Data Transfer and Virtual Memory

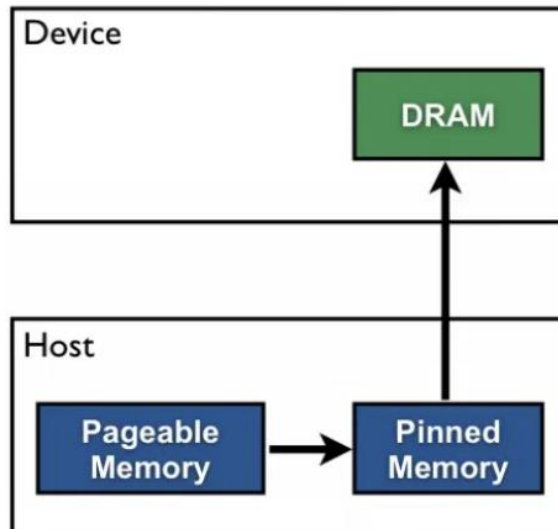
- DMA uses physical addresses
 - When cudaMemcpy() copies an array, it is implemented as one or more DMA transfers
 - Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
 - No address translation for the rest of the same DMA transfer so that high efficiency can be achieved
- The *OS could accidentally page-out the data that is being read or written by a DMA* and page-in another virtual page into the same physical location

Pinned Memory & DMA Data Transfer

- **Pinned memory** are **virtual** memory pages that are specially marked so that they ***cannot be paged out***
- Allocated with a special system API function call
- a.k.a. Page Locked Memory, Locked Pages, etc.
- **CPU memory** that serve as the source or destination of a DMA transfer ***must be allocated as pinned memory***

Pageable memory

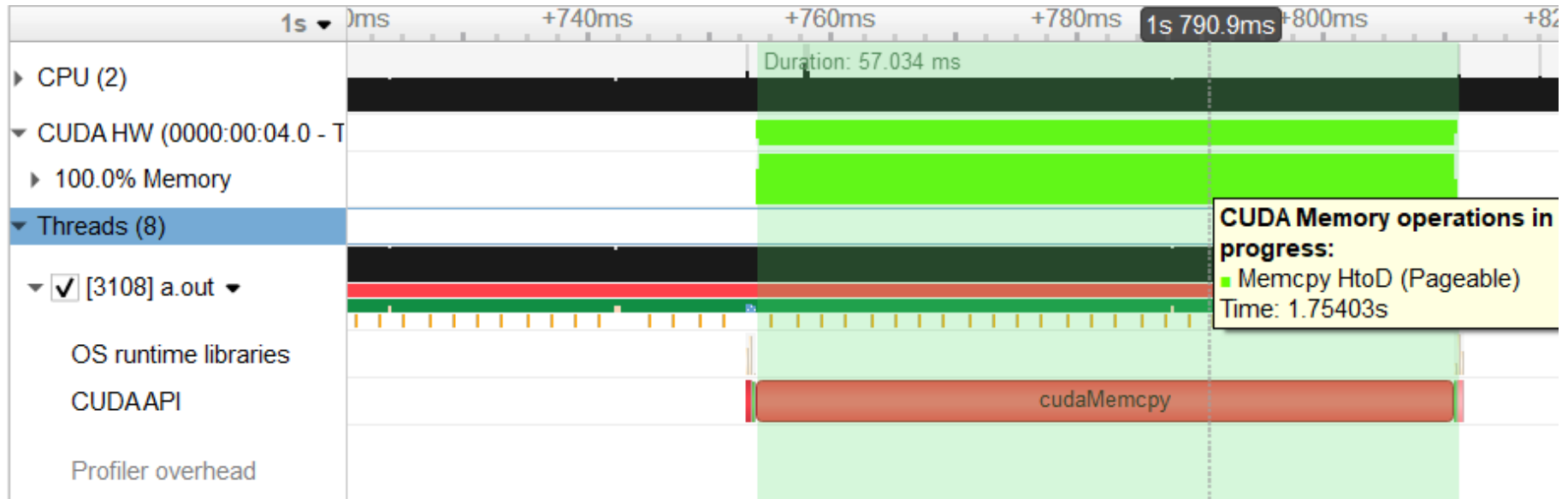
- The memory allocated in host is by **default *pageable memory*** (malloc)
- To transfer this data to the device, the CUDA run time ***copies this memory to a temporary pinned memory*** and then transfers to the device memory.
- There are **two memory transfers** → **Slow**



Pageable memory - Usage

```
int* h_in = (int*)malloc(nBytes);  
// Init data for h_in ...  
  
int* d_in;  
cudaMalloc(&d_in, nBytes);  
  
// Copy data to device memories  
cudaMemcpy(d_in, in, nBytes, cudaMemcpyHostToDevice));  
  
cudaFree(d_in);  
free(in);
```

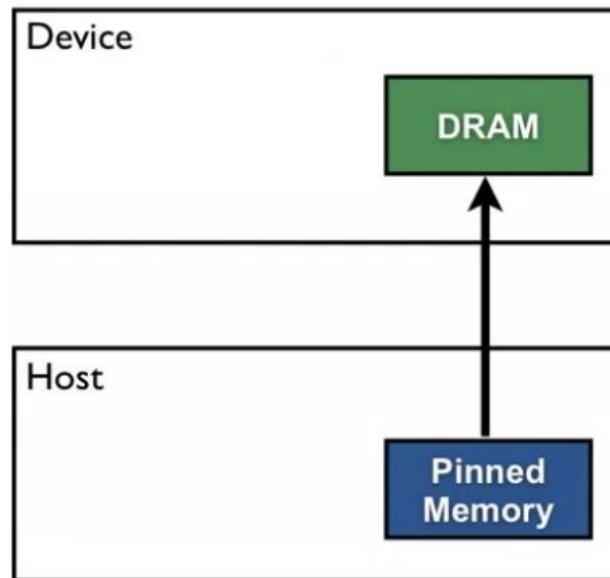
Pageable memory - Usage



Pageable

Pinned memory

- The data can be initialized directly in the **host pinned memory**. Không tốn thêm 1 lần copy, nhưng chỉ nên pin đúng lúc
- **Avoid two data transfers** as in pageable memory.
- Make the process faster but at the cost of host performance.
 - When data is initialized in the *pinned memory*, the memory availability for host processing is reduced



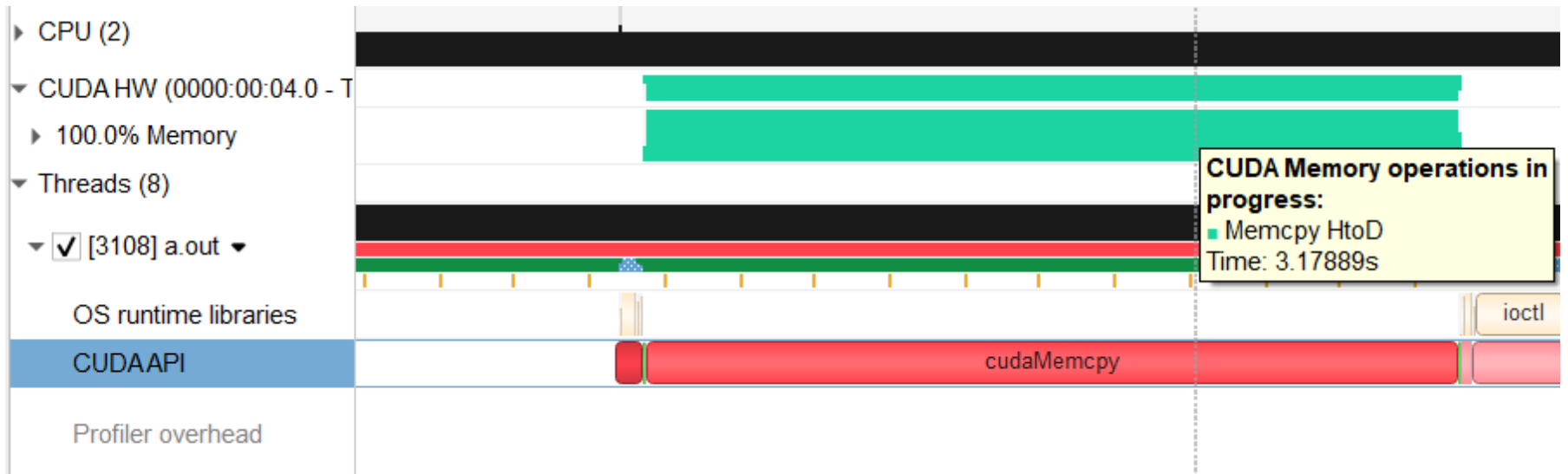
Pinned memory

- `cudaHostAlloc()`, three parameters:
 - Address of pointer to the allocated memory
 - Size of the allocated memory in bytes
 - Option (Ex: `cudaHostAllocDefault`)
- `cudaFreeHost()`, one parameter
 - Pointer to the memory to be freed

Pinned memory - Usage

```
int* h_in;  
int* d_in  
cudaMallocHost(&h_in, nBytes);  
// Init data for h_in ...  
  
cudaMalloc(&d_in, nBytes);  
  
// Copy data to device memories  
cudaMemcpy(d_in1, in1, nBytes, cudaMemcpyHostToDevice);  
//...  
cudaFree(d_in);  
cudaFreeHost(h_in);
```

Pinned memory - Usage



Mapped memory

- Copy trực tiếp CPU \Leftrightarrow GPU mà không cần vùng nhớ đệm
- **Mapped memory (zero-copy memory)**: pinned memory that is mapped into the device address space.
 - ***Both host and device have direct access*** to this memory.
 - **Pros:**
 - Can leverage host memory when there is insufficient device memory.
 - Can **avoid explicit data transfers** between host and device.
 - Improves PCIe transfer rates
 - **Cons:**
 - Transfer will happen during execution which will increase the processing time considerably

CUDA driver tự động
truyền dữ liệu cho mình

Mapped memory - Usage

- Host allocation:

```
int* h_a2, * h_b2, * h_c2;  
cudaHostAlloc((int**)&h_a2, nBytes, cudaHostAllocMapped);  
cudaHostAlloc((int**)&h_b2, nBytes, cudaHostAllocMapped);  
cudaHostAlloc((int**)&h_c2, nBytes, cudaHostAllocMapped);
```

- Device allocation:

```
int* d_a2, * d_b2, * d_c2;  
cudaHostGetDevicePointer(&d_a2, h_a2, 0);  
cudaHostGetDevicePointer(&d_b2, h_b2, 0);  
cudaHostGetDevicePointer(&d_c2, h_c2, 0);
```

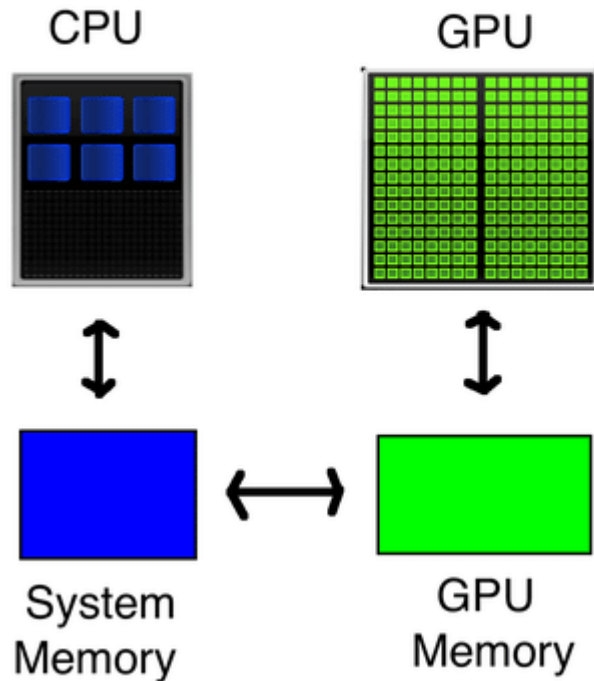
- ***cudaHostGetDevicePointer***. Passes back the device pointer corresponding to the mapped, pinned host buffer allocated by `cudaHostAlloc()`

Unified memory

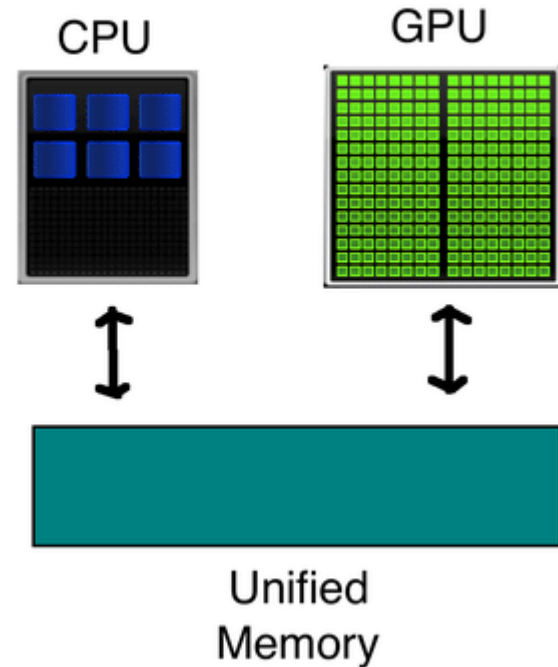
Vùng nhớ hợp nhất
=> Hiện giờ đang được recommend

- More general than Mapped memory

View without 'Unified Memory'



View with 'Unified Memory'



Unified memory

- Is a single memory address space ***accessible both from the host and from the device.***
- The hardware/software handles ***automatically*** the data **migration** between the host and the device maintaining **consistency** between them.
- **Pros:**
 - No explicit allocation and recovery of memory for device needed. This ***reduces programming complexity.***
 - Enabling ***larger arrays*** than the device memory size.
- **Cons:** Cần thêm các câu lệnh chạy ngầm định bên dưới
 - Adds additional instructions under the hood for memory management

Unified memory - Usage

- Only need 1 initialization Host và Device đều có thể dùng a,b,c

```
int *a, *b, *c;  
cudaMallocManaged((int **)&a, nBytes);  
cudaMallocManaged((int **)&b, nBytes);  
cudaMallocManaged((int **)&c, nBytes);
```

- `cudaMallocManaged()`
 - Allocates an object in the Unified Memory address space.
 - Address of a pointer to the allocated object
 - Size of the allocated object in terms of bytes
- `cudaFree()`
 - Frees object from unified memory.
- `cudaMemcpy()`
 - Copy data between different arrays, regardless of position
 - Direction: `cudaMemcpyDefault`

Comparision

- Runing Matrix addition with #rows = 4096, #cols = 8192
- Run on Tesla T4 (C.C 7.5)

Memory	Total run time
Pageable	152.19 ms
Pinned	38.22 ms
Mapped	22.00 ms
Unified	122.11 ms

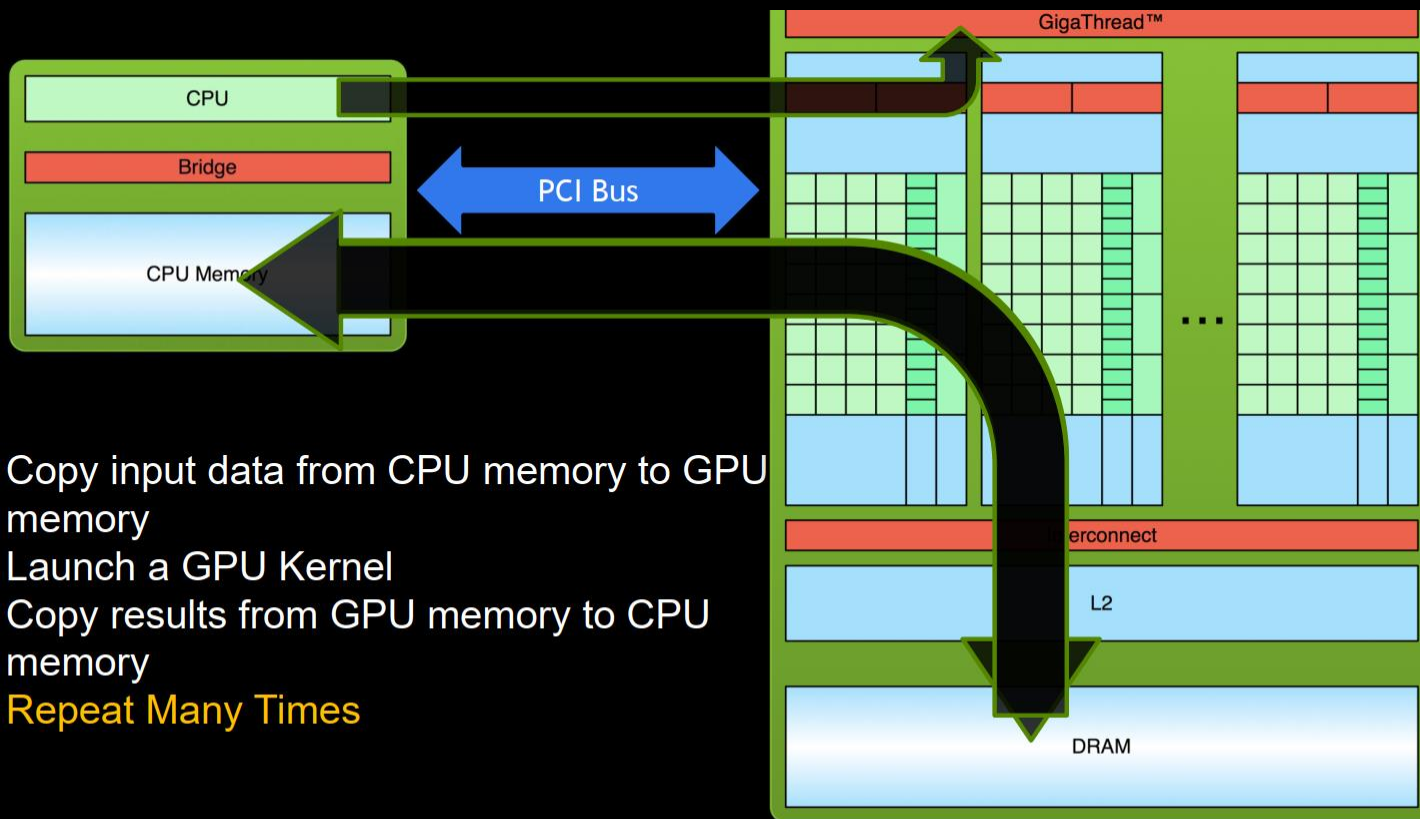
Note: *these numbers will change depend on hardware*

CUDA STREAM

Introduction

- **Optimization**: try to make full use of hardware resources, don't let any of them idle
- So far, we have discussed about optimization limited to **the scope of a kernel** **Phải có đủ block để che Latency**
 - Need enough blocks to utilize SMs
 - In each SM, need enough independent instructions (coming from one warp or from different warps) to utilize execution pipelines, hide latency
 - Minimize warp divergence
- Today, we will discuss about optimization in a bigger scope: **outside a kernel**

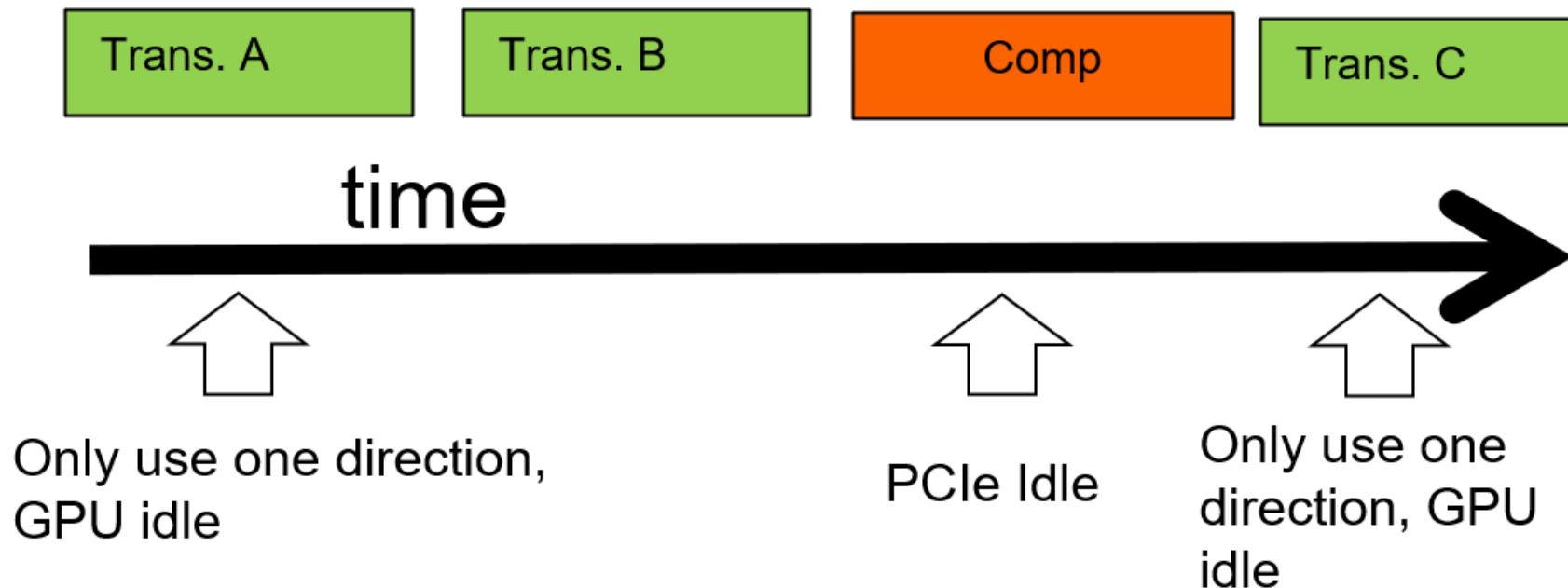
Simple processing Flow



1. Copy input data from CPU memory to GPU memory
2. Launch a GPU Kernel
3. Copy results from GPU memory to CPU memory
4. Repeat Many Times

Simple processing Flow

- So far, the way we use `cudaMemcpy` serializes data transfer and GPU computation for `MatAddKernel()`



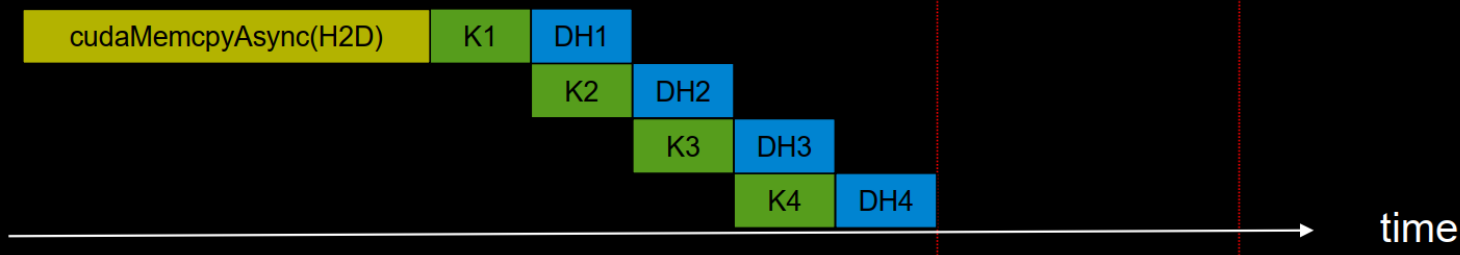
idle = rảnh, không làm việc

Concurrency Through Pipelining

Serial

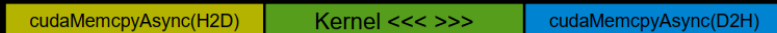


Concurrent- overlap kernel and D2H copy

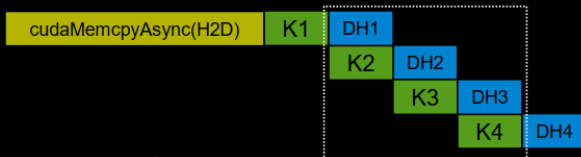


Concurrency Through Pipelining

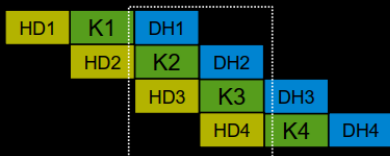
- Serial (1x)



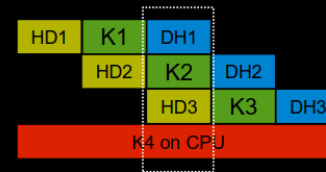
- 2-way concurrency (up to 2x)



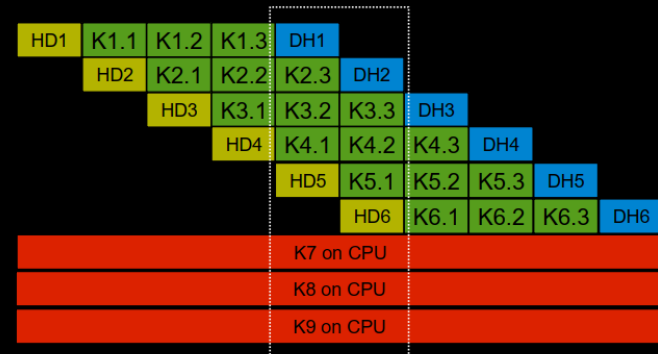
- 3-way concurrency (up to 3x)



- 4-way concurrency (3x+)



- 4+ way concurrency



Overlap tasks

- With most current devices, in the extreme, we can overlap:
 - many host computations (if utilize CPU cores)
 - and many device computations (kernels)
 - and one H2D
 - and one D2H
- Basic conditions to overlap tasks:
 - These tasks are independent of each other
 - There are enough hardware resources for these tasks

In CUDA, how to overlap tasks?

When host calls a CUDA command, there are 2 possible situations:

host gọi và đợi

- **Synchronous**: host sends this command to a device queue and waits until it finishes

E.g, cudaMemcpy

host gọi và có thể làm việc khác

- **Asynchronous**: host sends this command to a device queue and continues to do other works without waiting this command to finish

E.g., host calls a kernel

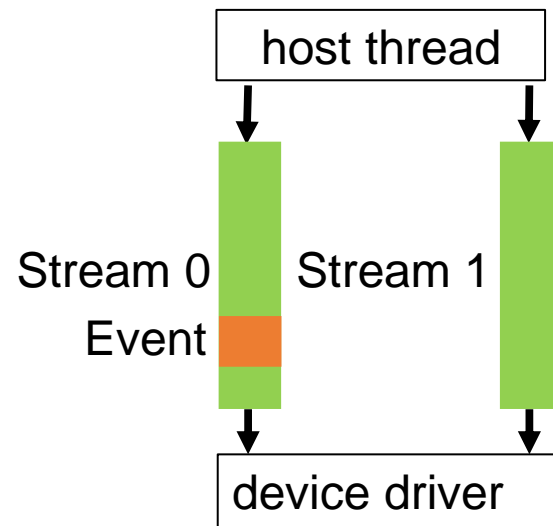
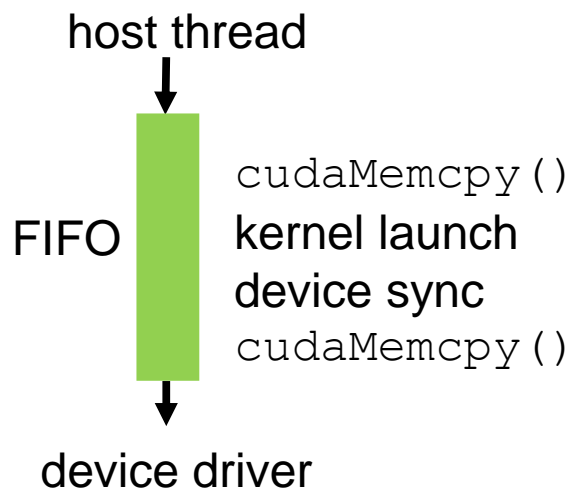


By default, we can overlap a host computation and a device computation

How to overlap other tasks (e.g., a kernel with another kernel)?

In CUDA, how to overlap tasks?

- A **CUDA stream** is a **task queue of device**
Host sends device tasks to this queue
- Tasks **in the same stream** will be executed by device **sequentially** in **FIFO** order
- Tasks **in different streams** will **have no order with each other** and **can overlap** with each other
- Driver ensures that commands in a queue are processed in sequence



Stream CUDA commands

- **Create stream**

```
cudaStream_t stream;
```

```
cudaStreamCreate(&stream);
```

 Tạo stream bình thường

- **Destroy stream**

```
cudaStreamDestroy(stream);
```

- **Send tasks to stream**

- kernel<<<gridSize, blockSize,
0, // Share memory

```
stream>>>(…);
```

 Kernel chạy ở stream nào

- cudaMemcpyAsync(dst, src, size, dir,
stream);

 Mem-copy ở stream nào

Default stream (stream 0 / NULL stream)

- By default, tasks will be sent to stream 0
- **Note: stream 0 synchronizes with other streams**

E.g., host sends tasks (T) to streams (S) in order T0-S0, T-S1, T-S2, T1-S0, then in device:

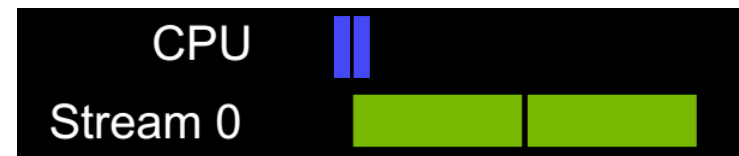
- First, execute T0-S0
 - After finishing T0-S0, execute T-S1, T-S2 (T-S1 and T-S2 can overlap)
 - After finishing T-S1 and T-S2, execute T1-S0
- To overlap
 - **Option 1**: replace stream 0 by stream non-0
 - Option 2: create stream non-0 as follows Tạo stream được phép // với stream 0
`cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking)`

Example: overlap kernels

- Assume a kernel foo only utilizes 50% of device resource

- **Use stream 0**

```
foo<<<blocks, threads>>>();  
foo<<<blocks, threads>>>();
```



- **Use stream 0 and stream non-0**

```
cudaStream_t stream1;  
cudaStreamCreate(&stream1);  
foo<<<blocks, threads>>>();  
foo<<<blocks, threads,  
        0, stream1>>>();  
cudaStreamDestroy(stream1);
```



Example: overlap kernels

- Assume a kernel foo only utilizes 50% of device resource

- **Use stream 0**

```
foo<<<blocks, threads>>>();  
foo<<<blocks, threads>>>();
```



- **Use stream 0 and stream non-0**

```
cudaStream_t stream1;  
cudaStreamCreateWithFlags(  
    &stream1, cudaStreamNonBlocking);  
foo<<<blocks, threads>>>();  
foo<<<blocks, threads,  
    0, stream1>>>();  
cudaStreamDestroy(stream1);
```

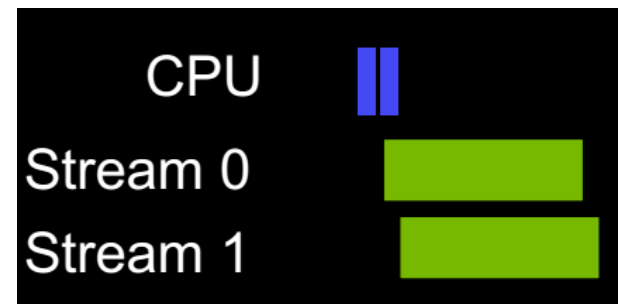
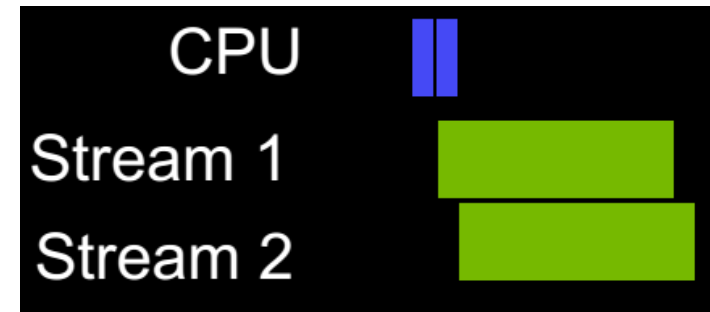


Image source: Justin Luitjens,
CUDA Streams, GTC2014

Example: overlap kernels

- Assume a kernel foo only utilizes 50% of device resource
- **Use stream non-0** Thường dùng cách này nhiều nhất
Dùng các stream khác 0

```
cudaStream_t stream1, stream2;  
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
foo<<<blocks, threads,  
    0, stream1>>>();  
foo<<<blocks, threads,  
    0, stream2>>>();  
cudaStreamDestroy(stream1);  
cudaStreamDestroy(stream2);
```

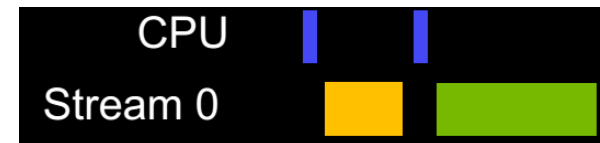


Example: overlap data transfer with other tasks

- **Example 1**

`cudaMemcpy(...);` Đây là hàm đồng bộ => CPU phải chờ

`foo<<<...>>>();`



- **Example 2** Trong lúc copy CPU chạy tiếp

`cudaMemcpyAsync(..., stream1);`

`foo<<<..., stream1>>>();`



Need to **pin** & host memory: replace `malloc` by `cudaMallocHost` (and free by `cudaFreeHost`)

Why? In order for host to continue to do other works and device hardware to transfer data, the physical memory storing data in host must be kept intact – must be pinned; otherwise, data stored in the physical memory of host can be changed by OS while device is transferring data (because of [virtual memory mechanism](#) in host) ☹️

Example: overlap data transfer with other tasks

- **Example 1**

```
cudaMemcpy(...);  
foo<<<...>>>();
```



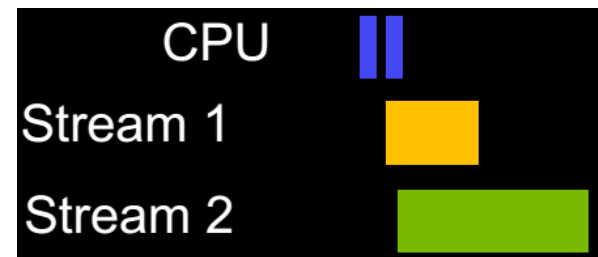
- **Example 2**

```
cudaMemcpyAsync(..., stream1);  
foo<<<..., stream1>>>();
```



- **Example 3**

```
cudaMemcpyAsync(..., stream1);  
foo<<<..., stream2>>>();
```



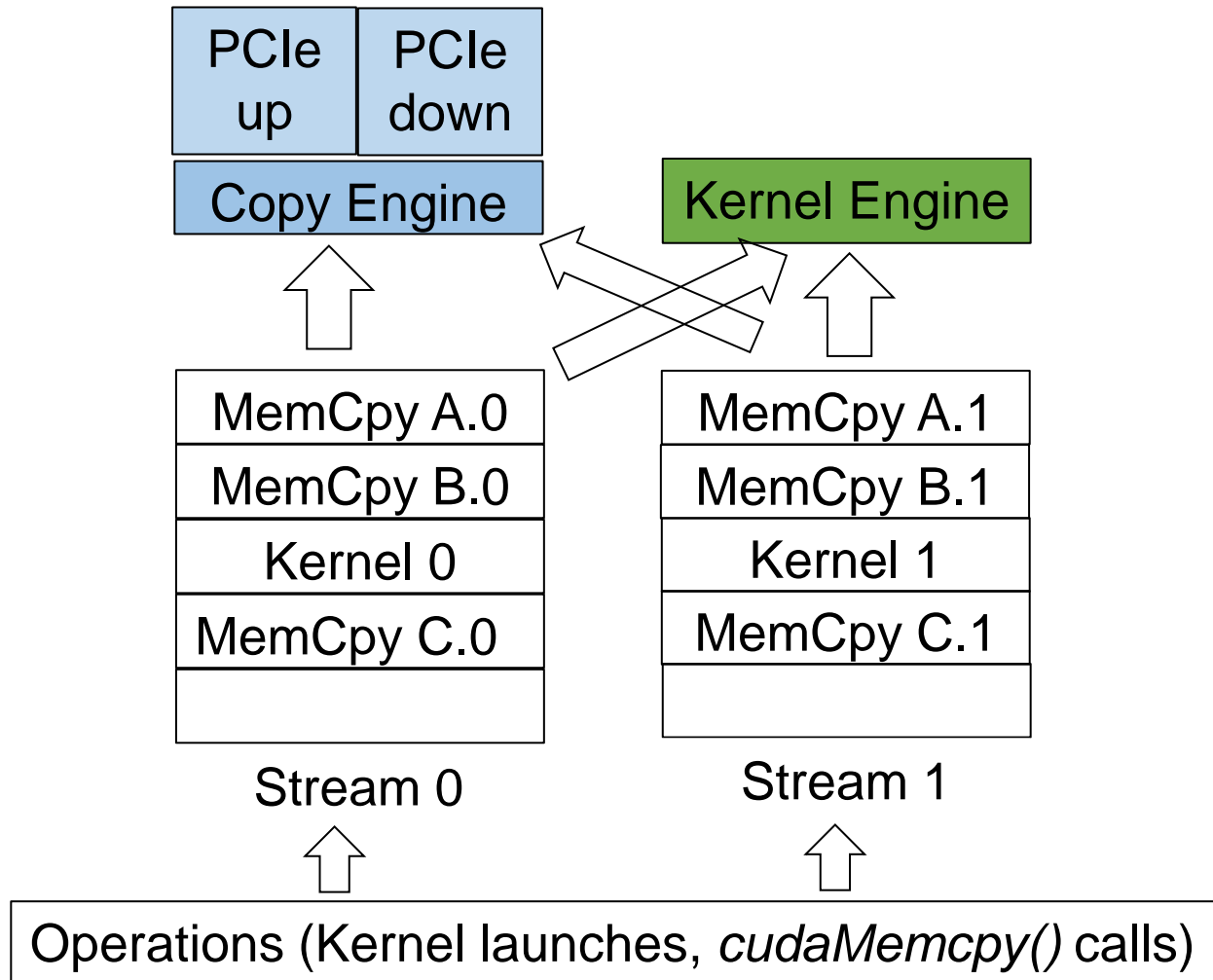
Device Overlap

- Some CUDA devices support device overlap
- Simultaneously execute a kernel while copying data between device and host memory

```
cudaDeviceProp devProv;  
cudaGetDeviceProperties(&devProv, 0);  
printf("Number of copy engines: %d\n",  
       devProv.asyncEngineCount);
```

- RTX 3050 Laptop GPU: **1**
 - Can only transfer 1 direction at one time.
- **Tesla T4 (Colab): 3**
 - H2D, D2H, NVLink (With 1 other GPU)
- Volta V100-32GB GPU: **6**
 - H2D, D2H, 4 NVLink (With 4 other GPU) 1 bộ gồm 5 card

Conceptual View of Streams



Host chủ động phân phối công việc cho các stream khác nhau

Concurrent memory copies

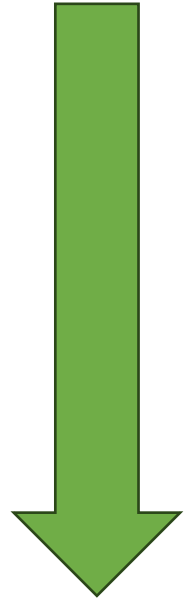
- `cudaMemcpy(...)`
 - Places transfer into default stream
 - Synchronous: Must complete prior to returning
- `cudaMemcpyAsync(..., &stream)`
 - Places transfer into stream and returns immediately
- To achieve concurrency 4 điều kiện để song song
 - Transfers must be in a ***non-default stream***
 - Must use ***async*** copies
 - ***1 transfer per direction*** at a time
 - Memory on the host must be ***pinned***

Synchronization

When we let tasks run asynchronously, we will need to synchronize at some point

- Synchronize everything
 - `cudaDeviceSynchronize()` *synchronize mạnh nhất*
 - Blocks host until all issued CUDA calls are complete
- Synchronize host w.r.t. a specific stream
 - `cudaStreamSynchronize(stream)`
 - Blocks host until all issued CUDA calls in stream are complete *host chờ 1 stream*
- Synchronize host or devices using *events*

More
Synchronization



Less
Synchronization

Synchronization

Synchronize host with device

```
cudaDeviceSynchronize();
```

Synchronize host with a stream

- `cudaStreamSynchronize(stream);`
- `cudaStreamQuery(stream);` Xem stream hoạt động xong chưa
 - Host doesn't have to wait
 - Return: `cudaSuccess` if all tasks in `stream` are finished;
`cudaErrorNotReady` otherwise

CUDA EVENTS

- Provide a mechanism to signal when operations have occurred in a stream
 - Useful for profiling and synchronization
- Events have a boolean state:
 - Occurred
 - Not Occurred
 - **Important: Default state = occurred**

Synchronization

Synchronize host with a point in a stream: use event

- Create event

```
cudaEvent_t event;  
cudaEventCreate(&event);
```

- Send event to stream

```
cudaEventRecord(event, stream);
```

- Set the event state to not occurred
- Event state is set to occurred when it reaches the front of the stream

- Synchronize host with event

- `cudaEventSynchronize(event);` Host dừng đến khi event xảy ra
- `cudaEventQuery(event);` // Similar to `cudaStreamQuery`

- Destroy event

```
cudaEventDestroy(event);
```

Synchronization

Synchronize streams with each other

`cudaStreamWaitEvent(stream, event)`

- `stream` waits `event` (of another stream) to happen, only then it continues to do tasks enqueued to `stream` after this command
- Host doesn't have to wait



THE END

Reference

- [1] Wen-Mei, W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022
- [2] Cheng John, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014
- [3] *Illinois-NVIDIA GPU Teaching Kit*
- [4] Cuda Streamsbest Practices And Common Pitfalls, Justin Luitjens -NVIDIA

