# GPU Computing and Accelerators: Part IV

# Compute Unified Device Architecture (CUDA)

## Streams

### Definition (Stream)

Streams are a mechanism that introduces another level of parallelism into the CUDA framework. While the basic setup, we have seen until here, is SIMD or more precisely SIMT, using streams one can have the GPU do different things at the same time. Streams are not as flexible and general purpose as tasks on the host CPU, though.

The basic power of streams is to have memory transfers and computational operations overlap in an asynchronous way. Note, however that not all CUDA enabled devices support overlapping these operations. On top of that not all CUDA enabled devices that do support the overlapping execution do so in the same way.

# Compute Unified Device Architecture (CUDA)

## Page-Locked Memory on the Host

Asynchronous data transfers in CUDA are not only performed without synchronization to the actual computation, they are also intended to interact with the computation as little as possible. Especially they should not interrupt the CPU from performing useful work in the program. They are therefore set up to use direct memory access (DMA) circumventing CPU interaction.

However, in order to do this, we need to use a special portion of host memory, that is guaranteed to stay in place during the operation. The default portion of host memory that we allocate using `malloc()` is paged memory. It can be anywhere in the virtual memory of the host and is allowed to move around (e.g., swapped to disk when more space is required).

# Compute Unified Device Architecture (CUDA)

## Page-Locked Memory on the Host

### Definition (page-locked memory)

Page-locked memory is a portion of memory that is guaranteed to keep its position in the virtual memory. It is not available for any kind of paging operations, such as swapping. Therefore, it is sometimes also called pinned memory.

**Advantages of pinned memory:**

- can be used for DMA safely
- transfer speeds can be up to 2× faster than to/from pageable memory

**Disadvantages:**

- memory fragmentation increases and thus the usability deteriorates.

# Compute Unified Device Architecture (CUDA)

## Streams and Compute Capabilities

Over the years Nvidia® has changed the way things are implemented. This is not only regarding the API in the CUDA toolkit, but also the underlying device hardware. The very first CUDA enabled devices could not overlap transfers and executions at all. Then some devices used separate engines for copy and kernel executions. Modern hardware usually has even two engines for performing transfers in direction to the host and to the device separately. Basically we can classify the devices as follows:

| Comp. Capab. | Properties |
|---|---|
| 1.0 | No overlap |
| 1.1- | 1 copy engine and 1 kernel execution engine |
| 2.x- | 1 kernel execution engine, 1 copy to host engine and 1 copy to device engine |
| 3.5- | eliminates the differences in asynchronous execution |

Table: classification of CUDA enabled devices with respect to the ability of overlapping memory transfers and computations.

# Compute Unified Device Architecture (CUDA)

## Streams and Compute Capabilities

### Hwo can i know what my device can do?

The `cudaDeviceProp` structure can be used to find out whether a device supports overlapped operation and how many execution engines are available. The important members are

- `int deviceOverlap` indicating the availability of overlapped operations
- `int asyncEngineCount` storing the number of asynchronous execution engines available.

The important information, which type of asynchronous execution model is implemented in the hardware can thus be fetched with the `cudaGetDeviceProperties()` function.

# Compute Unified Device Architecture (CUDA)

## An Introductory Asynchronous Transfer Example

We are following [5] What we want to do is

- copy data to the device
- perform some task (kernel) on it
- get the result back to the host

### Example

The the critical portion of the code would look like

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

according to what we have learned until now. This is regarding the default execution stream.

[5] https://developer.nvidia.com/content/how-overlap-data-transfers-cuda-cc

# Compute Unified Device Architecture (CUDA)

## An Introductory Asynchronous Transfer Example: Non-Default Streams

### Example (Creation and Destruction of Streams)

Consider we have the two variables

```
cudaStream_t stream1;
cudaError_t result;
```

Then we can create a new stream using

```
result = cudaStreamCreate(&stream1)
```

and later get rid of it via

```
result = cudaStreamDestroy(stream1)
```

# Compute Unified Device Architecture (CUDA)

## An Introductory Asynchronous Transfer Example: Non-Default Streams

### Example (Memory transfers)

Once we have acquired a new stream we have to tell the asynchronous copy routines to use it. The basic command `cudaMemcpyAsync()` takes the same arguments as `cudaMemcpy`. Only, it has an additional argument specifying the stream to use:

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1)
```

### Example (Kernel Execution)

We need to use the extended launch size specification here:

```
      <<< block distr., thread distr., dynamic memory per block,
           associated stream >>>
```

The third argument can be used to allocate additional  dynamic shared memory per block. We will use 0 here.

```
kernel<<<1,N,0,stream1>>>(d_a)
```

# Compute Unified Device Architecture (CUDA)

## An Introductory Asynchronous Transfer Example: Asynchronous Execution Engines

The influence of the number of engines (especially for copying data) is best displayed in a simple example.

Consider we have a group of streams cooperating on `kernel()`. Think of a situation where splitting the problem data into chunks is necessary to fit the data into the device memory. We basically have two ways to implement the cooperation,

1. loop over the entire copy-work-copy block
2. loop over the work and copies separately

Note that the asynchronous copy acts different on the control flow than the `cudaMemcpy()`. Where in the basic default stream using `cudaMemcpy()` we can rely on the fact that as soon as the command returns, all data has been transferred, in the case of `cudaMemcpyAsync()` does not even guarantee that the copy operation has started at all. It will only have scheduled the operation in a first in first out (FIFO) list of pending operation on the corresponding asynchronous execution engine.

# Compute Unified Device Architecture (CUDA)

## An Introductory Asynchronous Transfer Example: Asynchronous Execution Engines

### Example (Asynchronous Execution Version 1)

Looping over the entire block of copy-work-copy operations is described by the following code fragment

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, stream[i]);
  kernel<<>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, stream[i]);
}
```

# Compute Unified Device Architecture (CUDA)

## An Introductory Asynchronous Transfer Example: Asynchronous Execution Engines

### Example (Asynchronous Execution Version 2)

Looping over the single tasks in contrast looks like

```
for (int i = 0; i < nStreams; ++i) {
 int offset = i * streamSize;
 cudaMemcpyAsync(&d_a[offset], &a[offset],
                 streamBytes, cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  kernel<<>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&a[offset], &d_a[offset],
                  streamBytes, cudaMemcpyDeviceToHost, stream[i]);
}
```

# Compute Unified Device Architecture (CUDA)

## An Introductory Asynchronous Transfer Example: Asynchronous Execution Engines

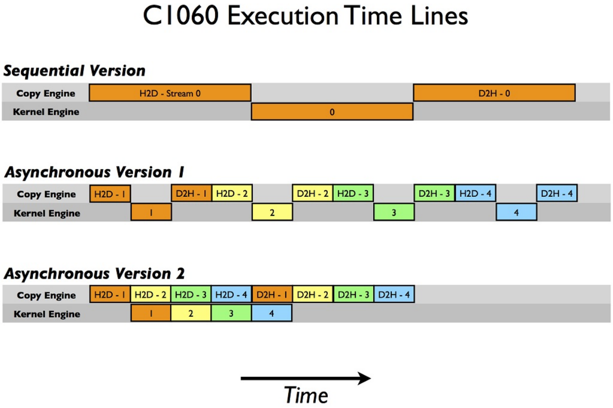

Figure: Execution time line on a device with a single copy engine.

# Compute Unified Device Architecture (CUDA)

## An Introductory Asynchronous Transfer Example: Asynchronous Execution Engines

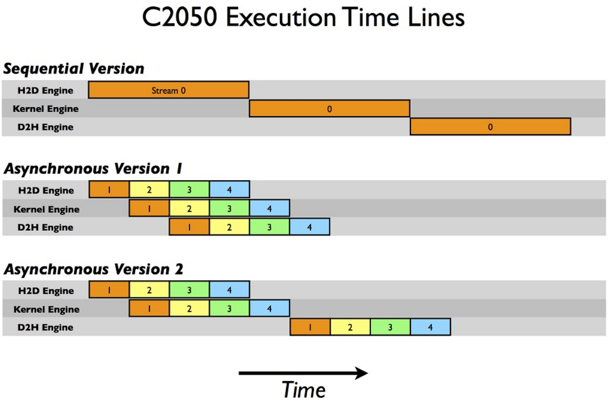

Figure: Execution time line on a device with separate copy engines for device to host (D2H) and host to device (H2D) operations.

# Compute Unified Device Architecture (CUDA)

## Interoperability with Graphics

Using the same GPU for computations and graphical display of results is possible. See, e.g. CUDA by Example (Chapter 8), or CUDA C Programming Guide.

# Compute Unified Device Architecture (CUDA)

## Interoperability with Graphics

Using the same GPU for computations and graphical display of results is possible. See, e.g. CUDA by Example (Chapter 8), or CUDA C Programming Guide.

## Usage of Multiple GPUs

Usage of multiple GPUs in a single program requires the concepts of zero-copy host memory, and portable pinned memory. An introduction can be found in CUDA by Example (Chapter 11).