High Energy Physics

Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

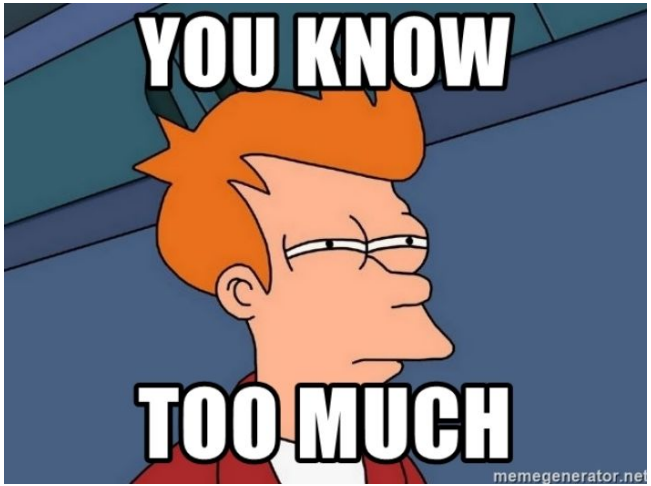
GPU programming module

Week 5 : CUDA advanced topics

Lecture 9 - October 10th 2024

What we learnt last week

- We discussed about data locality, data caching and were introduced to the coalesced memory data access pattern
- We learnt about the importance of shared memory and went over atomic operations
- We heard about the concept of the CUDA stream



Today

Today we will hear about :

- Paged & pinned memory
- Non-default CUDA streams
- CUDA events

Default CUDA stream

We remember from last week - A stream is ..

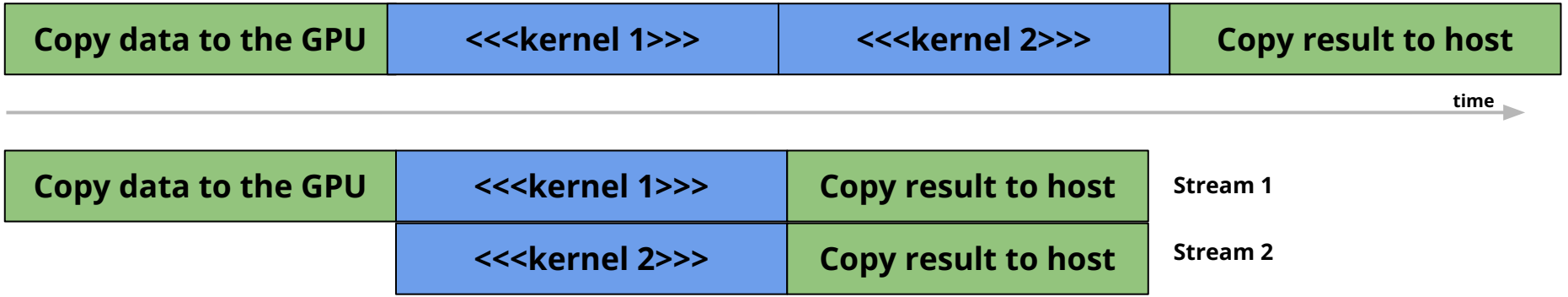
- **... a sequence of commands that execute in order**
  - Executed on the device in the order in which they are issued by the host code
  - Any instruction that runs in a stream must complete before the next can be issued
- Can execute various types of commands.
  - Kernel invocations
  - Memory transmissions
  - Memory (de)allocations
  - Memsets
  - Synchronizations

CUDA default stream

- CUDA has what we call a **default stream**
  - By default all CUDA kernels run in this default stream
- The default stream is blocking :
  - Other commands are not executed in parallel on the device

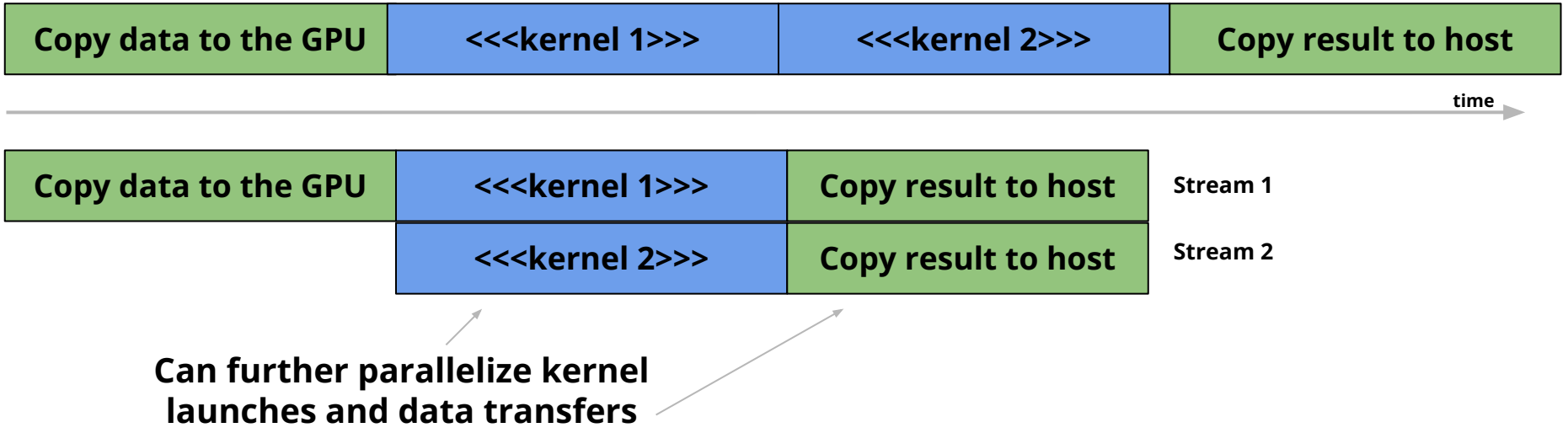
CUDA default stream

- In CUDA, we can also run multiple kernels on different streams concurrently
  - Non-default CUDA streams!



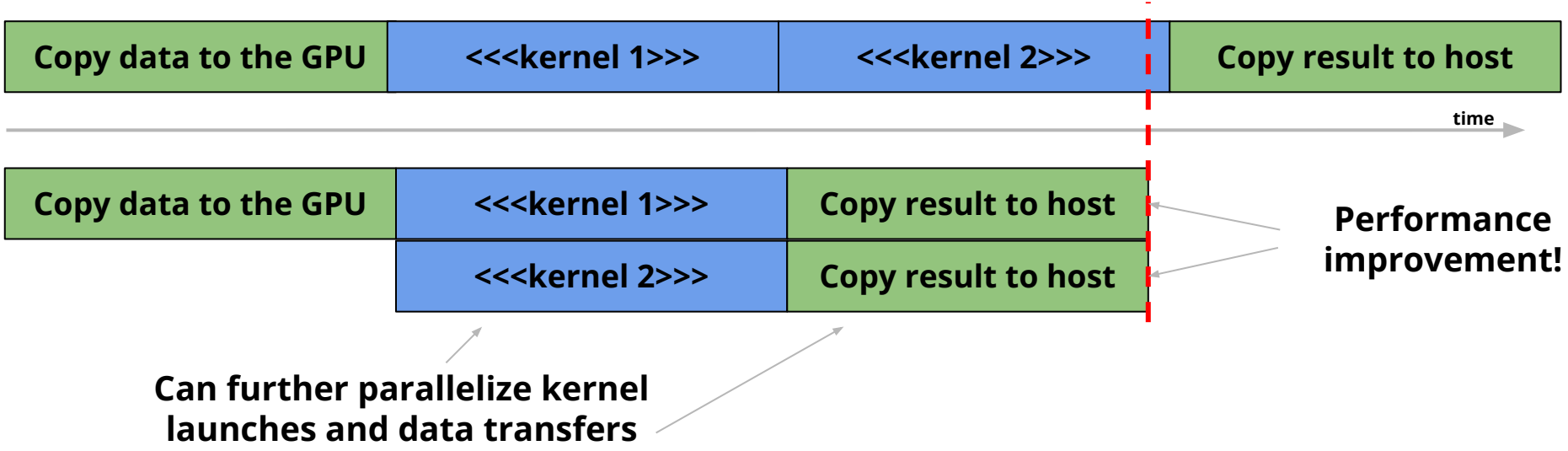
CUDA default stream

- In CUDA, we can also run multiple kernels on different streams concurrently
  - Non-default CUDA streams!



CUDA default stream

- In CUDA, we can also run multiple kernels on different streams concurrently
  - Non-default CUDA streams!



Pinned memory

Paged memory

- Memory paging technique allows the operating system to retrieve data from secondary storage in fixed-size blocks → pages
  - Allows system to use physical memory more efficiently
  - ~4KBs per page
  - Introduces mapping → OS maintains a page map of page numbers to physical RAM memory

Paged memory

- Memory paging technique allows the operating system to retrieve data from secondary storage in fixed-size blocks → pages
  - Allows system to use physical memory more efficiently
  - ~4KBs per page
  - Introduces mapping → OS maintains a page map of page numbers to physical RAM memory

- **Pageable memory :**
  - Regular memory that can be moved to and from disk (paged) by the OS e.g. retrieve data from secondary hard drive (e.g. SSD) and load it into the CPU RAM

Paged memory

- Memory paging technique allows the operating system to retrieve data from secondary storage in fixed-size blocks → pages
  - Allows system to use physical memory more efficiently
  - ~4KBs per page
  - Introduces mapping → OS maintains a page map of page numbers to physical RAM memory

- **Pros :** Programs can use “more” memory than physically available since only needed pages are loaded

Paged memory

- Memory paging technique allows the operating system to retrieve data from secondary storage in fixed-size blocks → pages
  - Allows system to use physical memory more efficiently
  - ~4KBs per page
  - Introduces mapping → OS maintains a page map of page numbers to physical RAM memory

- **Pros :** Programs can use “more” memory than physically available since only needed pages are loaded
- **Cons :** Can lead to large overhead due to page faults, memory swapping etc.

Pinned memory

- **Pinned memory** is allocated in a way that cannot be swapped out by the operating system.
- Remains in physical memory for the duration of its use

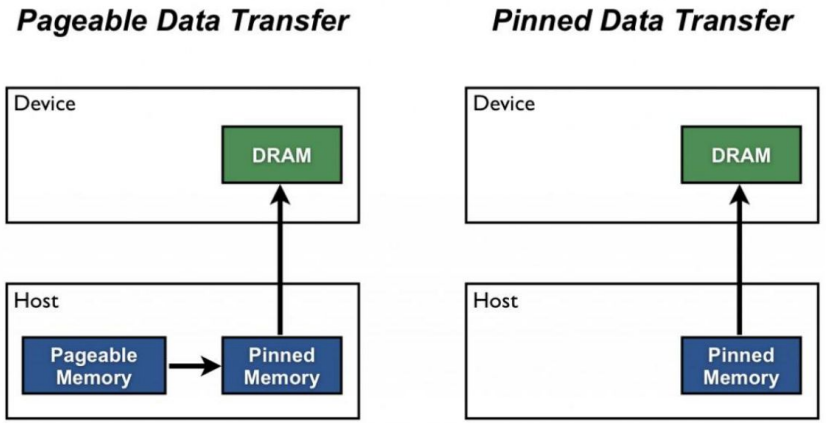


Image source [9]

Pinned memory

- **Pinned memory** is allocated in a way that cannot be swapped out by the operating system.
- Remains in physical memory for the duration of its use

- Allows faster host to/from device copies via **D**irect **M**emory **A**ccess (DMA)

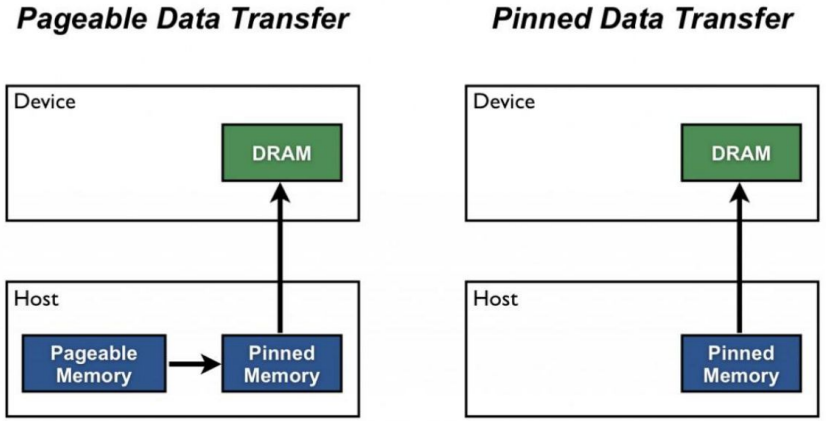
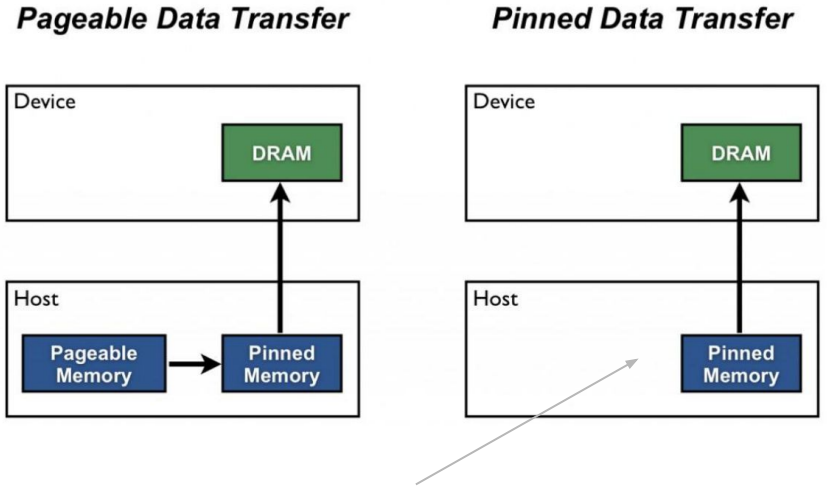


Image source [9]

Pinned memory

- **Pinned memory** is allocated in a way that cannot be swapped out by the operating system.
- Remains in physical memory for the duration of its use

- Allows faster host to/from device copies via **D**irect **M**emory **A**ccess (DMA)



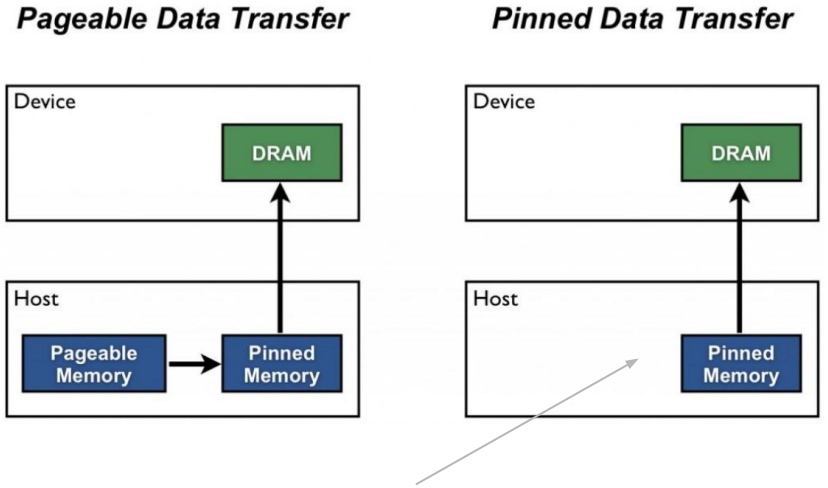
- **The CPU will instruct the DMA controller to start a transfer**

Image source [9]

Pinned memory

- **Pinned memory** is allocated in a way that cannot be swapped out by the operating system.
- Remains in physical memory for the duration of its use

- Allows faster host to/from device copies via **D**irect **M**emory **A**ccess (DMA)



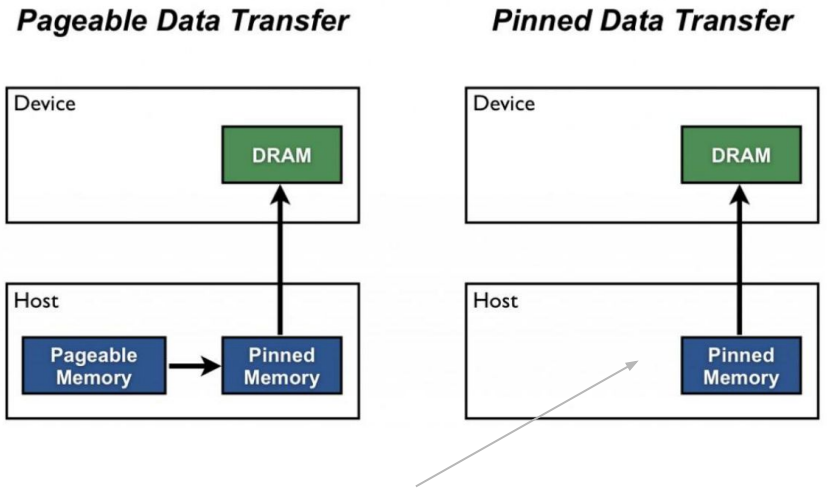
- **The CPU will instruct the DMA controller to start a transfer**
- **The DMS knows the memory addresses of the source and destination**

Image source [9]

Pinned memory

- **Pinned memory** is allocated in a way that cannot be swapped out by the operating system.
- Remains in physical memory for the duration of its use

- Allows faster host to/from device copies via **D**irect **M**emory **A**ccess (DMA)
- **Enables asynchronous memory copies to/from host and device**



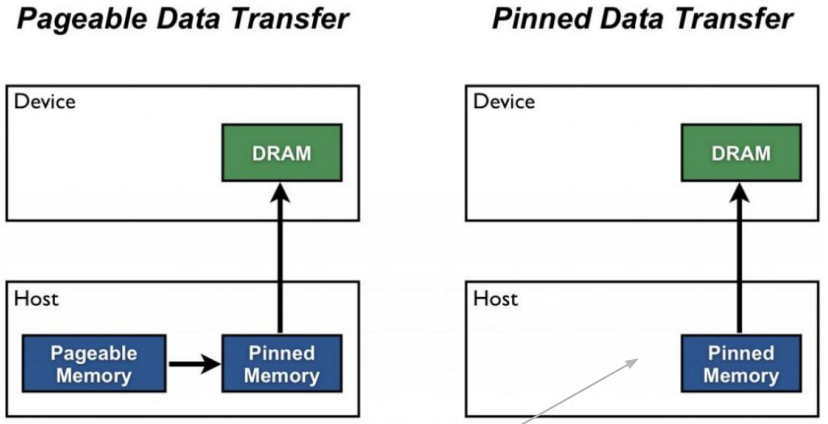
- **The CPU will instruct the DMA controller to start a transfer**
- **The DMS knows the memory addresses of the source and destination**

Image source [9]

Pinned memory

- **Pinned memory** is allocated in a way that cannot be swapped out by the operating system.
- Remains in physical memory for the duration of its use

- Allows faster host to/from device copies via **D**irect **M**emory **A**ccess (DMA)
- **Enables asynchronous memory copies to/from host and device**



- **The CPU will instruct the DMA controller to start a transfer**
- **The DMS knows the memory addresses of the source and destination**

- **Non blocking**
- CPU can perform other tasks while the data transfer occurs.

Image source [9]

Using pinned memory

- CUDA specific API functions for allocating mined memory on the host side

```
// Kernel to add two matrices
__global__ void myKernel(...) {
…
}
int main() {
    const size_t size = N * N * sizeof(float);
    // Pinned memory for matrices
    float *h_A, *h_B, *h_C;
    cudaMallocHost((void**)&h_A, size);
    …
    …
    // Device pointers
    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    …
    // Copy matrices from pinned host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Launch the kernel
    myKernel<<<numBlocks, threadsPerBlock>>>();
    // Copy result back to pinned host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
    …
    // Free pinned host memory
    cudaFreeHost(h_A);
    …
    return 0;
}
```

Image source [9]

Using pinned memory

- CUDA specific API functions for allocating mined memory on the host side
- Allocate memory for device memory buffers as usual

```
// Kernel to add two matrices
__global__ void myKernel(...) {
…
}
int main() {
    const size_t size = N * N * sizeof(float);
    // Pinned memory for matrices
    float *h_A, *h_B, *h_C;
    cudaMallocHost((void**)&h_A, size);
    …
    …
    // Device pointers
    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    …
    // Copy matrices from pinned host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Launch the kernel
    myKernel<<<numBlocks, threadsPerBlock>>>();
    // Copy result back to pinned host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
    …
    // Free pinned host memory
    cudaFreeHost(h_A);
    …
    return 0;
}
```

Image source [9]

Using pinned memory

- CUDA specific API functions for allocating mined memory on the host side
- Allocate memory for device memory buffers as usual
- Perform copy operations and kernel launches as usual

```
// Kernel to add two matrices
__global__ void myKernel(...) {
…
}
int main() {
    const size_t size = N * N * sizeof(float);
    // Pinned memory for matrices
    float *h_A, *h_B, *h_C;
    cudaMallocHost((void**)&h_A, size);
    …
    …
    // Device pointers
    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    …
    // Copy matrices from pinned host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Launch the kernel
    myKernel<<<numBlocks, threadsPerBlock>>>();
    // Copy result back to pinned host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
    …
    // Free pinned host memory
    cudaFreeHost(h_A);
    …
    return 0;
}
```

Image source [9]

Using pinned memory

- CUDA specific API functions for allocating mined memory on the host side
- Allocate memory for device memory buffers as usual
- Perform copy operations and kernel launches as usual
- Don't forget to deallocate the pinned memory once done

```
// Kernel to add two matrices
__global__ void myKernel(...) {
…
}
int main() {
    const size_t size = N * N * sizeof(float);
    // Pinned memory for matrices
    float *h_A, *h_B, *h_C;
    cudaMallocHost((void**)&h_A, size);
    …
    …
    // Device pointers
    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    …
    // Copy matrices from pinned host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Launch the kernel
    myKernel<<<numBlocks, threadsPerBlock>>>();
    // Copy result back to pinned host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
    …
    // Free pinned host memory
    cudaFreeHost(h_A);
    …
    return 0;
}
```

Image source [9]

Using pinned memory

- CUDA specific API functions for allocating mined memory on the host side
- Allocate memory for device memory buffers as usual
- Perform copy operations and kernel launches as usual
- Don't forget to deallocate the pinned memory once done

The `cudaMemcpyAsync` API function can be also used when using multiple streams **for asynchronous data transfers**

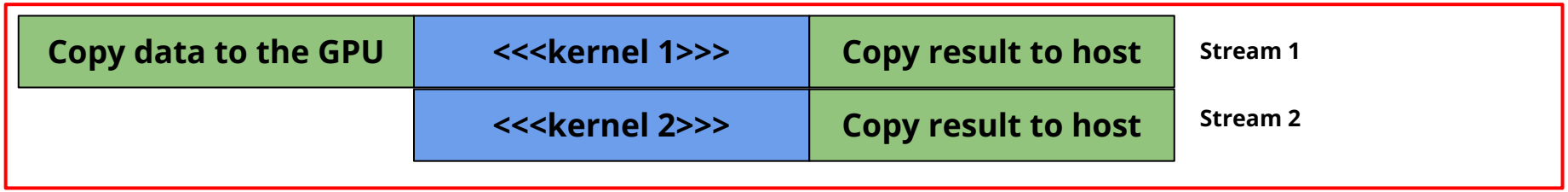
```
// Kernel to add two matrices
__global__ void myKernel(...) {
…
}
int main() {
    const size_t size = N * N * sizeof(float);
    // Pinned memory for matrices
    float *h_A, *h_B, *h_C;
    cudaMallocHost((void**)&h_A, size);
    …
    …
    // Device pointers
    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    …
    // Copy matrices from pinned host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Launch the kernel
    myKernel<<<numBlocks, threadsPerBlock>>>();
    // Copy result back to pinned host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
    …
    // Free pinned host memory
    cudaFreeHost(h_A);
    …
    return 0;
}
```

Image source [9]

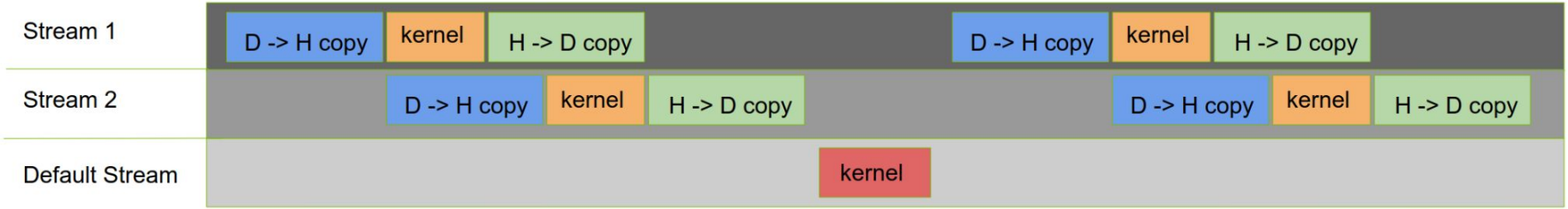
Non-default CUDA streams

Non default CUDA streams

- We can run multiple kernels/memory copies/synchronization operations on different streams concurrently



Simple example



If we wanted to code-up the above sequence of operations :

1. Create 2 streams

Image source [9]

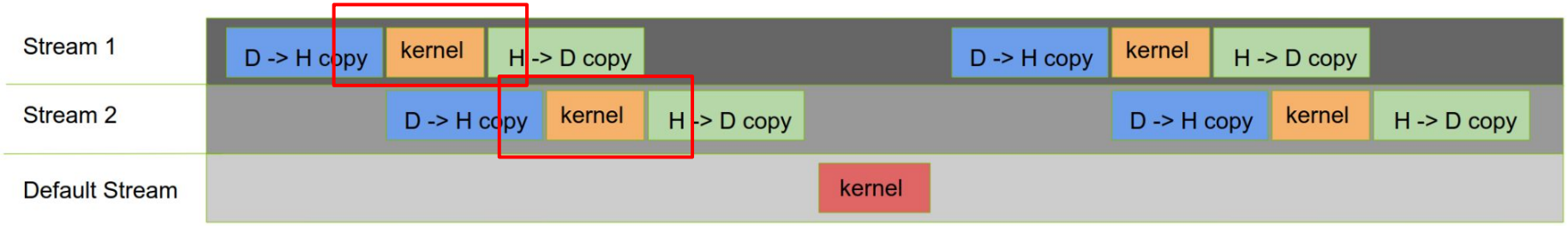
Simple example

If we wanted to code-up the above sequence of operations :

1. Create 2 streams
2. Copy data to device asynchronously for each stream

Image source [9]

Simple example

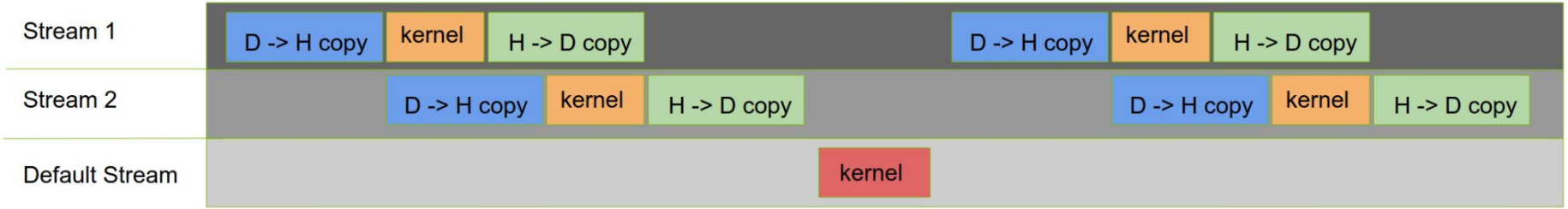


If we wanted to code-up the above sequence of operations :

1. Create 2 streams
2. Copy data to device asynchronously for each stream
3. Launch separate kernels in each stream

Image source [9]

Simple example

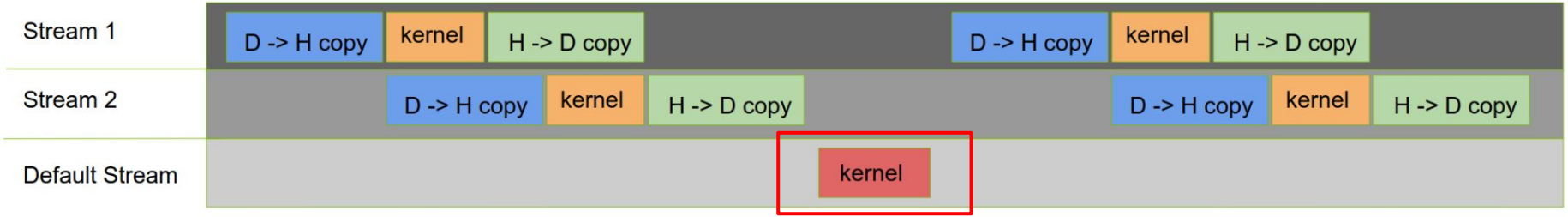


If we wanted to code-up the above sequence of operations :

1. Create 2 streams
2. Copy data to device asynchronously for each stream
3. Launch separate kernels in each stream
4. Synchronize streams to make sure results are available on device

Image source [9]

Simple example

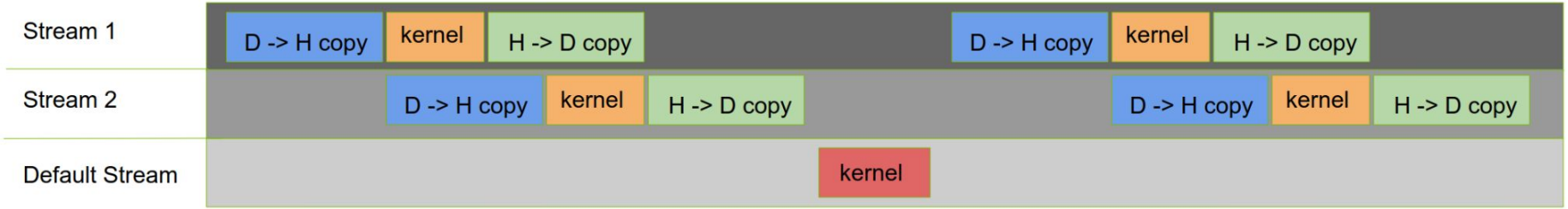


If we wanted to code-up the above sequence of operations :

1. Create 2 streams
2. Copy data to device asynchronously for each stream
3. Launch separate kernels in each stream
4. Synchronize streams to make sure results are available on device
5. Launch kernel on the default stream

Image source [9]

Simple example

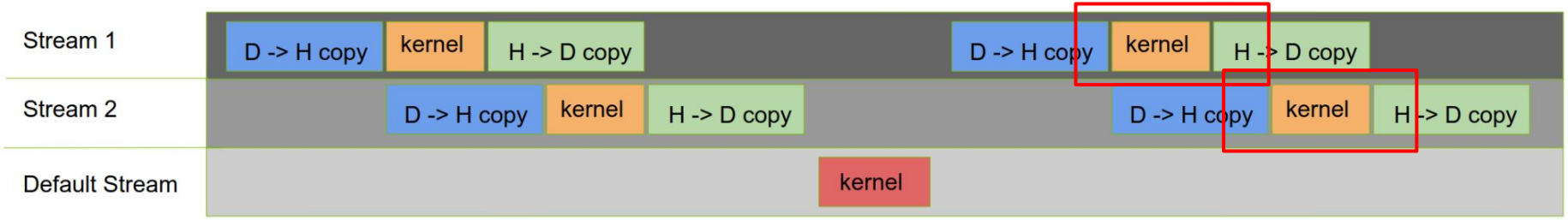


If we wanted to code-up the above sequence of operations :

1. Create 2 streams
2. Copy data to device asynchronously for each stream
3. Launch separate kernels in each stream
4. Synchronize streams to make sure results are available on device
5. Launch kernel on the default stream
6. Synchronize to make sure results are accessible on device

Image source [9]

Simple example

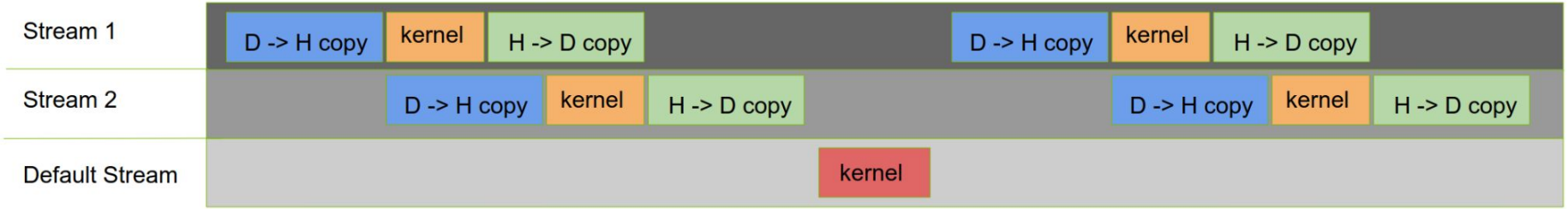


If we wanted to code-up the above sequence of operations :

1. Create 2 streams
2. Copy data to device asynchronously for each stream
3. Launch separate kernels in each stream
4. Synchronize streams to make sure results are available on device
5. Launch kernel on the default stream
6. Synchronize to make sure results are accessible on device
7. Repeat!

Image source [9]

Simple example



If we wanted to code-up the above sequence of operations :

1. Create 2 streams
2. Copy data to device asynchronously for each stream
3. Launch separate kernels in each stream
4. Synchronize streams to make sure results are available on device
5. Launch kernel on the default stream
6. Synchronize to make sure results are accessible on device
7. Repeat!
8. Finally copy results back to host

Image source [9]

Using non-default streams

- Create a non-default stream with **`cudaStreamCreate`**

```
__global__ void myKernel(int* d_data) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d_data[idx] += 1; // Increment each element by 1
}

int main() {
    const int N = 1000;
    int *h_data1, *h_data2, *d_data1, *d_data2;

    // Allocate host memory
    h_data1 = new int[N];
    h_data2 = new int[N];

    // Allocate device memory
    cudaMalloc(&d_data1, N * sizeof(int));
    cudaMalloc(&d_data2, N * sizeof(int));
    // Initialize host data
...
    // Create two non-default streams
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1)
    cudaStreamCreate(&stream2)

    // Copy data from Device to Host asynchronously on different streams
    cudaMemcpyAsync(h_data1, d_data1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(h_data2, d_data2, N * sizeof(int), cudaMemcpyDeviceToHost, stream2);

    // Launch a dummy kernel on different streams
    dummyKernel<<<...,stream1>>>(d_data1);
    dummyKernel<<<..., stream2>>>(d_data2);

    // Copy data from Host to Device asynchronously on different streams
    cudaMemcpyAsync(d_data1, h_data1, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_data2, h_data2, N * sizeof(int), cudaMemcpyHostToDevice, stream2);

    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2)

    // Free resources
    cudaFree(d_data1);
    cudaFree(d_data2);
    delete[] h_data1;
    delete[] h_data2;

    // Destroy streams
    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);

    return 0;
}
```

Using non-default streams

- Create a non-default stream with **`cudaStreamCreate`**
- Perform an asynchronous memory copy with **`cudaMemcpyAsync`**

```
__global__ void myKernel(int* d_data) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d_data[idx] += 1; // Increment each element by 1
}

int main() {
    const int N = 1000;
    int *h_data1, *h_data2, *d_data1, *d_data2;

    // Allocate host memory
    h_data1 = new int[N];
    h_data2 = new int[N];

    // Allocate device memory
    cudaMalloc(&d_data1, N * sizeof(int));
    cudaMalloc(&d_data2, N * sizeof(int));
    // Initialize host data
…
    // Create two non-default streams
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1)
    cudaStreamCreate(&stream2)

    // Copy data from Device to Host asynchronously on different streams
    cudaMemcpyAsync(h_data1, d_data1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(h_data2, d_data2, N * sizeof(int), cudaMemcpyDeviceToHost, stream2);

    // Launch a dummy kernel on different streams
    dummyKernel<<<...,stream1>>>(d_data1);
    dummyKernel<<<..., stream2>>>(d_data2);

    // Copy data from Host to Device asynchronously on different streams
    cudaMemcpyAsync(d_data1, h_data1, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_data2, h_data2, N * sizeof(int), cudaMemcpyHostToDevice, stream2);

    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2)

    // Free resources
    cudaFree(d_data1);
    cudaFree(d_data2);
    delete[] h_data1;
    delete[] h_data2;

    // Destroy streams
    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);

    return 0;
}
```

Using non-default streams

- Create a non-default stream with **`cudaStreamCreate`**
- Perform an asynchronous memory copy with **`cudaMemcpyAsync`**

```
__global__ void myKernel(int* d_data) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d_data[idx] += 1; // Increment each element by 1
}

int main() {
    const int N = 1000;
    int *h_data1, *h_data2, *d_data1, *d_data2;

    // Allocate host memory
    h_data1 = new int[N];
    h_data2 = new int[N];

    // Allocate device memory
    cudaMalloc(&d_data1, N * sizeof(int));
    cudaMalloc(&d_data2, N * sizeof(int));
    // Initialize host data
    ...
    // Create two non-default streams
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1)
    cudaStreamCreate(&stream2)

    // Copy data from Device to Host asynchronously on different streams
    cudaMemcpyAsync(h_data1, d_data1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(h_data2, d_data2, N * sizeof(int), cudaMemcpyDeviceToHost, stream2);

    // Launch a dummy kernel on different streams
    dummyKernel<<<...,stream1>>>(d_data1);
    dummyKernel<<<..., stream2>>>(d_data2);

    // Copy data from Host to Device asynchronously on different streams
    cudaMemcpyAsync(d_data1, h_data1, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_data2, h_data2, N * sizeof(int), cudaMemcpyHostToDevice, stream2);

    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2)

    // Free resources
    cudaFree(d_data1);
    cudaFree(d_data2);
    delete[] h_data1;
    delete[] h_data2;

    // Destroy streams
    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);

    return 0;
}
```

Note that even though you can technically allocate non-pinned memory to ensure that transfers are fully asynchronous, pinned memory should be used

Using non-default streams

- Create a non-default stream with **`cudaStreamCreate`**
- Perform an asynchronous memory copy with **`cudaMemcpyAsync`**
- Launch kernels separately on each stream

```
__global__ void myKernel(int* d_data) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d_data[idx] += 1; // Increment each element by 1
}

int main() {
    const int N = 1000;
    int *h_data1, *h_data2, *d_data1, *d_data2;

    // Allocate host memory
    h_data1 = new int[N];
    h_data2 = new int[N];

    // Allocate device memory
    cudaMalloc(&d_data1, N * sizeof(int));
    cudaMalloc(&d_data2, N * sizeof(int));
    // Initialize host data
…
    // Create two non-default streams
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1)
    cudaStreamCreate(&stream2)

    // Copy data from Device to Host asynchronously on different streams
    cudaMemcpyAsync(h_data1, d_data1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(h_data2, d_data2, N * sizeof(int), cudaMemcpyDeviceToHost, stream2);

    // Launch a dummy kernel on different streams
    dummyKernel<<<...,stream1>>>(d_data1);
    dummyKernel<<<..., stream2>>>(d_data2);

    // Copy data from host to device asynchronously on different streams
    cudaMemcpyAsync(d_data1, h_data1, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_data2, h_data2, N * sizeof(int), cudaMemcpyHostToDevice, stream2);

    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2)

    // Free resources
    cudaFree(d_data1);
    cudaFree(d_data2);
    delete[] h_data1;
    delete[] h_data2;

    // Destroy streams
    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);

    return 0;
}
```

Using non-default streams

- Create a non-default stream with **`cudaStreamCreate`**
- Perform an asynchronous memory copy with **`cudaMemcpyAsync`**
- Launch kernels separately on each stream
- Synchronize streams. This will ensure that all operations have been completed

```
__global__ void myKernel(int* d_data) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d_data[idx] += 1; // Increment each element by 1
}

int main() {
    const int N = 1000;
    int *h_data1, *h_data2, *d_data1, *d_data2;

    // Allocate host memory
    h_data1 = new int[N];
    h_data2 = new int[N];

    // Allocate device memory
    cudaMalloc(&d_data1, N * sizeof(int));
    cudaMalloc(&d_data2, N * sizeof(int));
    // Initialize host data
…
    // Create two non-default streams
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1)
    cudaStreamCreate(&stream2)

    // Copy data from Device to Host asynchronously on different streams
    cudaMemcpyAsync(h_data1, d_data1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(h_data2, d_data2, N * sizeof(int), cudaMemcpyDeviceToHost, stream2);

    // Launch a dummy kernel on different streams
    dummyKernel<<<...,stream1>>>(d_data1);
    dummyKernel<<<..., stream2>>>(d_data2);

    // Copy data from Host to Device asynchronously on different streams
    cudaMemcpyAsync(d_data1, h_data1, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_data2, h_data2, N * sizeof(int), cudaMemcpyHostToDevice, stream2);

    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2)

    // Free resources
    cudaFree(d_data1);
    cudaFree(d_data2);
    delete[] h_data1;
    delete[] h_data2;

    // Destroy streams
    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);

    return 0;
}
```

Using non-default streams

- Create a non-default stream with **`cudaStreamCreate`**
- Perform an asynchronous memory copy with **`cudaMemcpyAsync`**
- Launch kernels separately on each stream
- Synchronize streams. This will ensure that all operations have been completed
- Don't forget to destroy the streams

```
__global__ void myKernel(int* d_data) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d_data[idx] += 1; // Increment each element by 1
}

int main() {
    const int N = 1000;
    int *h_data1, *h_data2, *d_data1, *d_data2;

    // Allocate host memory
    h_data1 = new int[N];
    h_data2 = new int[N];

    // Allocate device memory
    cudaMalloc(&d_data1, N * sizeof(int));
    cudaMalloc(&d_data2, N * sizeof(int));
    // Initialize host data
…
    // Create two non-default streams
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1)
    cudaStreamCreate(&stream2)

    // Copy data from Device to Host asynchronously on different streams
    cudaMemcpyAsync(h_data1, d_data1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(h_data2, d_data2, N * sizeof(int), cudaMemcpyDeviceToHost, stream2);

    // Launch a dummy kernel on different streams
    dummyKernel<<<...,stream1>>>(d_data1);
    dummyKernel<<<..., stream2>>>(d_data2);

    // Copy data from Host to Device asynchronously on different streams
    cudaMemcpyAsync(d_data1, h_data1, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_data2, h_data2, N * sizeof(int), cudaMemcpyHostToDevice, stream2);

    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2)

    // Free resources
    cudaFree(d_data1);
    cudaFree(d_data2);
    delete[] h_data1;
    delete[] h_data2;

    // Destroy streams
    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);

    return 0;
}
```

Using non-default streams

- Create a non-default stream with **`cudaStreamCreate`**
- Perform an asynchronous memory copy with **`cudaMemcpyAsync`**
- Launch kernels separately on each stream
- Synchronize streams. This will ensure that all operations have been completed
- Don't forget to destroy the streams

Lets try an example out!
You can copy this code into a .cu file and try to run it.
Remember: To compile first set up your environment and then :
nvcc myscript.cu -o myscript
./myscript

```
__global__ void myKernel(int* d_data) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d_data[idx] += 1; // Increment each element by 1
}

int main() {
    const int N = 1000;
    int *h_data1, *h_data2, *d_data1, *d_data2;

    // Allocate host memory
    h_data1 = new int[N];
    h_data2 = new int[N];

    // Allocate device memory
    cudaMalloc(&d_data1, N * sizeof(int));
    cudaMalloc(&d_data2, N * sizeof(int));
    // Initialize host data
…
    // Create two non-default streams
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1)
    cudaStreamCreate(&stream2)

    // Copy data from Device to Host asynchronously on different streams
    cudaMemcpyAsync(h_data1, d_data1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(h_data2, d_data2, N * sizeof(int), cudaMemcpyDeviceToHost, stream2);

    // Launch a dummy kernel on different streams
    dummyKernel<<<...,stream1>>>(d_data1);
    dummyKernel<<<..., stream2>>>(d_data2);

    // Copy data from Host to Device asynchronously on different streams
    cudaMemcpyAsync(d_data1, h_data1, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_data2, h_data2, N * sizeof(int), cudaMemcpyHostToDevice, stream2);

    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2)

    // Free resources
    cudaFree(d_data1);
    cudaFree(d_data2);
    delete[] h_data1;
    delete[] h_data2;

    // Destroy streams
    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);

    return 0;
}
```

CUDA Events

CUDA Events

- **Markers used for synchronization and signaling**
  - Used for timing and complex synchronization between streams
  - Also used for timing and synchronization between streams and the host

CUDA Events

- Start by creating a CUDA event

```
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaEventRecord(start); // “record” issued into default stream
Kernel<<<b, t >>>( … );
cudaEventRecord(stop);
cudaEventSynchronize(stop); // wait for stream activity to reach stop event
cudaEventElapsedTime(&float_time, start, stop);
```

CUDA Events

- Start by creating a CUDA event
- CUDA Events are “recorded” when they are issued

```
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaEventRecord(start); // “record” issued into default stream
Kernel<<<b, t >>>( … );
cudaEventRecord(stop);
cudaEventSynchronize(stop); // wait for stream activity to reach stop event
cudaEventElapsedTime(&float_time, start, stop);
```

CUDA Events

- Start by creating a CUDA event
- CUDA Events are “recorded” when they are issued
- CUDA Events are “completed” when the stream has reached the point where it was recorded

```
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaEventRecord(start); // “record” issued into default stream
Kernel<<<b, t >>>( … );
cudaEventRecord(stop);
cudaEventSynchronize(stop); // wait for stream activity to reach stop event
cudaEventElapsedTime(&float_time, start, stop);
```

CUDA Events

- Start by creating a CUDA event
- CUDA Events are “recorded” when they are issued
- CUDA Events are “completed” when the stream has reached the point where it was recorded
- CUDA function that block the host until a specific event on the GPU has completed.

```
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaEventRecord(start); // “record” issued into default stream
Kernel<<<b, t >>>( … );
cudaEventRecord(stop);
cudaEventSynchronize(stop); // wait for stream activity to reach stop event
cudaEventElapsedTime(&float_time, start, stop);
```

CUDA Events

- Start by creating a CUDA event
- CUDA Events are “recorded” when they are issued
- CUDA Events are “completed” when the stream has reached the point where it was recorded
- CUDA function that block the host until a specific event on the GPU has completed.
- Is used to calculate the elapsed time in ms between two CUDA events

```
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaEventRecord(start); // “record” issued into default stream
Kernel<<<b, t >>>( … );
cudaEventRecord(stop);
cudaEventSynchronize(stop); // wait for stream activity to reach stop event
cudaEventElapsedTime(&float_time, start, stop);
```

CUDA Events

- Start by creating a CUDA event
- CUDA Events are “recorded” when they are issued
- CUDA Events are “completed” when the stream has reached the point where it was recorded
- CUDA function that block the host until a specific event on the GPU has completed.
- Is used to calculate the elapsed time in ms between two CUDA events

- For controlling the execution order of operations across multiple streams →**cudaStreamWaitEvent()**
  - Makes a stream wait for an event to happen

```
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaEventRecord(start); // “record” issued into default stream
Kernel<<<b, t >>>( … );
cudaEventRecord(stop);
cudaEventSynchronize(stop); // wait for stream activity to reach stop event
cudaEventElapsedTime(&float_time, start, stop);
```

Wrapping-up

Overview of today's lecture

- Heard about the differences of pinned and paged memory
- Learnt about the default and non-default CUDA streams
- Learned what CUDA events are
- **Optional** hands-on material :
  - You can try out the following hands-on exercises to get more familiar with CUDA streams:
    - https://github.com/matt-stack/TAC-HEP-Training-Feb2023.git
  - Note that these make use of nvidia profiling tools which we will learn about in the next weeks

Next week

- We will hear a lot about CUDA managed memory
- We will hear about C++ standards for parallelization

Back-up

Resources

1. NVIDIA Deep Learning Institute material link
2. 10th Thematic CERN School of Computing material link
3. Nvidia turing architecture white paper link
4. CUDA programming guide link
5. CUDA runtime API documentation link
6. CUDA profiler user's guide link
7. CUDA/C++ best practices guide link
8. NVidia DLI teaching kit link
9. https://tac-hep.org/assets/pdf/uw-gpu-fpga/CUDA_STREAMS_2023.pdf