# Heterogeneous computing with performance modelling

### Some advanced topics

Mirko Myllykoski mirkom@cs.umu.se

Department of Computing Science and HPC2N Umeå University

4-5. November 2020









#### Streams

As mentioned during the lecture 1.basics, the syntax

```
|| kernel_name <<< blocks , threads >>>( ... );
```

places a kernel kernel\_name into a **stream**.

- By default, the kernel is placed into the NULL stream (stream 0).
- The operations in a stream are executed in order.
  - Kernels are executed in the order they are issued.
  - Only one kernel can be active at any given time.
- Blocking functions, such as cudaMemcpy, wait until the NULL stream is empty.
  - Kernels and memory transfers do not overlap.









# Streams (create, destroy and syncronize)

- A CUDA program can contain several streams.
- A stream is created with

```
| __host__ cudaError_t cudaStreamCreate(cudaStream_t* pStream)
```

A stream is destroyed with

```
|| __host__ __device__ cudaError_t cudaStreamDestroy(cudaStream_t stream)
```

A stream is synchronized with

```
| __host__ cudaError_t cudaStreamSynchronize(cudaStream_t stream)
```

This causes the host thread to wait until the stream is empty.









## Streams (asynchronous functions)

Most familiar CUDA functions have asynchronous variants:

```
__host__ __device__ cudaError_t cudaMemcpyAsync (
   void* dst, const void* src, size_t count, cudaMemcpyKind kind,
   cudaStream_t stream = 0 )
  _host__ _device__ cudaError_t cudaMemcpy2DAsync (
    void* dst, size_t dpitch, const void* src, size_t spitch,
    size_t width, size_t height, cudaMemcpyKind kind,
    cudaStream_t stream = 0 )
__host__ __device__ cudaError_t cudaMemsetAsync (
    void* devPtr, int value, size_t count, cudaStream_t stream = 0 )
__host__ __device__ cudaError_t cudaMemset2DAsync (
    void* devPtr, size_t pitch, int value, size_t width, size_t height,
    cudaStream_t stream = 0 )
__host__ cudaError_t cudaMemPrefetchAsync (
    const void* devPtr, size_t count, int dstDevice,
    cudaStream_t stream = 0 )
```

- Note that all functions default to the NULL stream.
  - You can use asynchronous commands without creating a stream.



## Streams (kernels and a few comments)

A kernel is placed into a specific stream with the following notation:

```
|| kernel_name <<< blocks , threads, smem, stream >>>( ...);
```

- ▶ Note that the NULL stream is special<sup>1</sup>.
  - Other streams cannot run in parallel with the NULL stream.
  - Other streams synchronize implicitly with the NULL stream.
- In particular, the cudaDeviceSynchronize() function synchronizes all streams.
- Blocking data transfer functions, such as cudaMemcpy, also synchronize all streams.

<sup>1</sup>cudaStreamNonBlocking flag changes this.









# Streams (page-locked host memory)

- Host and device share the same memory address space.
  - A device cannot always access the host memory and vice versa.
- ▶ A device can access **page-locked** host memory:

```
__host__ cudaError_t cudaMallocHost ( void** ptr, size_t size )
__host__ cudaError_t cudaHostAlloc (
    void** pHost, size_t size, unsigned int flags )
__host__ cudaError_t cudaFreeHost ( void* ptr )
__host__ cudaError_t cudaHostRegister (
    void* ptr, size_t size, unsigned int flags )
__host__ cudaError_t cudaHostUnregister ( void* ptr )
```

- ► In most cases, the flag should be cudaHostAllocDefault or cudaHostRegisterDefault.
- ▶ Page-locked memory can be accessed with higher bandwidth than regular pageable memory (malloc()).
  - Page-locked memory is much slower that the global memory.
- Memory that is used in asynchronous data transfers should be page locked.



# Streams (page-locked host memory)

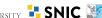
▶ In the earlier AX example (lecture 3.modelling), we reached only 5GB/s over the PCI-E bus. Let's change that:

```
cudaHostRegister(y, n*sizeof(double), cudaHostRegisterDefault);
struct timespec ts_start;
clock_gettime(CLOCK_MONOTONIC, &ts_start);
cudaMemcpy(d_y, y, n*sizeof(double), cudaMemcpyHostToDevice);
dim3 threads = 256;
dim3 blocks = max(1, min(256, n/threads.x));
ax_kernel <<<blocks, threads>>>(n, alpha, d_y);
cudaMemcpy(y, d_y, n*sizeof(double), cudaMemcpyDeviceToHost);
struct timespec ts_stop;
clock_gettime(CLOCK_MONOTONIC, &ts_stop);
```

Outcome:

```
Time = 0.641264 s
Floprate = 0.8 GFlops
Memory throughput = 12 GB/s
```



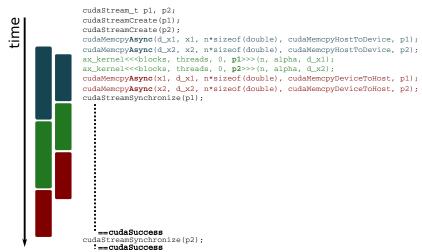






## Streams (example)

#### device host











#### **Events**

- Streams can be monitored and coordinated with events.
- An event must first be created:

- The cudaEventDestroy() function is safe, i.e., it frees all associated resources only after the event is no longer needed.
- After begin created, an event can be placed into a stream:

Events can be used for host thread synchronization:

```
| __host__ cudaError_t cudaEventSynchronize ( cudaEvent_t event )
```

▶ The host thread waits until the stream has reached the event.









# Events (continuation)

We can query the status of an event:

```
| __host__ cudaError_t cudaEventQuery ( cudaEvent_t event )
```

- If the stream has reached the event, cudaSuccess is returned. Otherwise, cudaErrorNotReady is returned.
- A stream can be made to wait until an other stream has reached an event:

Two events can be used for timing:

```
__host__ cudaError_t cudaEventElapsedTime (
    float* ms, cudaEvent_t start, cudaEvent_t end )
```

Returns the elapsed time between two events (in milliseconds).









# Managed memory

Modern GPUs can manage the memory automatically:

```
// allocate managea mom...
double *x;
cudaMallocManaged(&x, n*sizeof(double));

// initialize memory
for (int i = 0; i < n; i++)
    x[i] = 2.0 * rand()/RAND_MAX - 1.0;</pre>
 // issue the kernel directly
dim3 threads = 256;
dim3 blocks = (n+threads.x)/threads.x;
ax_kernel<<<br/>blocks, threads>>>(n, alpha, x);
cudaDeviceSynchronize();
cudaTree(x):
```

The host must call cudaDeviceSynchronize() before accessing the data.





# Managed memory (continuation)

- ▶ When a memory buffer that has been allocated with the cudaMallocManaged function is accessed, one of the following events occurs:
  - 1. If the corresponding memory page (4096 bytes) exists in the host/device memory, then the memory request is served by the memory controller or the caches.
  - 2. If the corresponding memory page does no exists in the host/device memory, then
    - 2.1 a page fault is triggered,
    - 2.2 memory transfer is initialized for the entire page, and
    - 2.3 the thread(s) pause until the memory transfer is ready.
- Fetching the entire page can be costly if the memory is accessed randomly. One must also pay attention to alignment.
- You can also prefetch the data to the global memory:

```
|| cudaMemPrefetchAsync(addr, size, cudaGetDevice(&device));
```









### Hands-ons

- Materials: https://git.cs.umu.se/mirkom/gpu\_course/
- ► Five hands-ons under hands-ons/4.advanced:
  - 1.async Learn how to use streams and asynchronous data transfers
  - 2.multi\_gemm Learn how to manage multiple streams.
    - 3.pipeline Learn how to pipeline computation and data transfers.
    - 4.managed Learn how to use managed memory.
      - 5.lu Learn what type of computations are suitable for GPUs.
- Solutions can be found under solutions/4.advanced.





