General-purpose programming on GPU

Asynchronous operations

Eugenio Rustico
rustico@dmi.unict.it

D.M.I. - Università di Catania

Updated: May 20, 2011

Overview

First step of 90% of GPU-based programs is uploading some data to the device.

First step of 90% of GPU-based programs is uploading some data to the device.

Last step for 100% of GPU-based programs is downloading some data back to the host.

First step of 90% of GPU-based programs is uploading some data to the device.

Last step for 100% of GPU-based programs is downloading some data back to the host.

Is it in general negligible?
(hint: think of a video streaming application...)

First step of 90% of GPU-based programs is uploading some data to the device.

Last step for 100% of GPU-based programs is downloading some data back to the host.

Is it in general negligible?
(hint: think of a video streaming application...)

Moreover: in multi-GPU applications with minimum problem interdependence, say n the number of GPUs, a typical need is $4 \cdot (n - 1)$ transfer requests per frame (why?)

There are two techniques that help us partically covering these latencies:

There are two techniques that help us partically covering these latencies:

- Asynchronous memory operations

There are two techniques that help us partically covering these latencies:

- Asynchronous memory operations
- Mapped memory

Some pairs of operations can be performed *simultaneously*:

Some pairs of operations can be performed *simultaneously*:

- Any CUDA device: kernel execution and host code (unless enviroment variable `CUDA_LAUNCH_BLOCKING` is set)

Some pairs of operations can be performed *simultaneously*:

- Any CUDA device: kernel execution and host code (unless enviroment variable `CUDA_LAUNCH_BLOCKING` is set)
- Any CUDA device: intra-device memcpys and host code

Some pairs of operations can be performed *simultaneously*:

- Any CUDA device: kernel execution and host code (unless enviroment variable `CUDA_LAUNCH_BLOCKING` is set)
- Any CUDA device: intra-device memcpys and host code
- Any CUDA device: host$\leftrightarrow$device memcpys of less than 64Kb

Some pairs of operations can be performed *simultaneously*:

- Any CUDA device: kernel execution and host code (unless enviroment variable `CUDA_LAUNCH_BLOCKING` is set)
- Any CUDA device: intra-device memcpys and host code
- Any CUDA device: host$\leftrightarrow$device memcpys of less than 64Kb
- Capability $\geq$ 1.1: kernel execution and host$\leftrightarrow$device memcpy

Some pairs of operations can be performed *simultaneously*:

- Any CUDA device: kernel execution and host code (unless enviroment variable `CUDA_LAUNCH_BLOCKING` is set)
- Any CUDA device: intra-device memcpys and host code
- Any CUDA device: host$\leftrightarrow$device memcpys of less than 64Kb
- Capability ≥ 1.1: kernel execution and host$\leftrightarrow$device memcpy
- Some devices with capability ≥ 2.0: concurrent execution of two different kernels

Some pairs of operations can be performed *simultaneously*:

- Any CUDA device: kernel execution and host code (unless enviroment variable `CUDA_LAUNCH_BLOCKING` is set)
- Any CUDA device: intra-device memcpys and host code
- Any CUDA device: host↔device memcpys of less than 64Kb
- Capability ≥ 1.1: kernel execution and host↔device memcpy
- Some devices with capability ≥ 2.0: concurrent execution of two different kernels
- Some devices with capability ≥ 2.0: concurrent execution of two memcpys in different directions (PCI is full duplex!)

Some pairs of operations can be performed *simultaneously*:

- Any CUDA device: kernel execution and host code (unless enviroment variable `CUDA_LAUNCH_BLOCKING` is set)
- Any CUDA device: intra-device memcpys and host code
- Any CUDA device: host$\leftrightarrow$device memcpys of less than 64Kb
- Capability ≥ 1.1: kernel execution and host$\leftrightarrow$device memcpy
- Some devices with capability ≥ 2.0: concurrent execution of two different kernels
- Some devices with capability ≥ 2.0: concurrent execution of two memcpys in different directions (PCI is full duplex!)

See deviceQuery output to check yours (at runtime, check specific booleans in the `deviceProperties`).

Some pairs of operations can be performed *simultaneously*:

- Any CUDA device: kernel execution and host code (unless enviroment variable `CUDA_LAUNCH_BLOCKING` is set)
- Any CUDA device: intra-device memcpys and host code
- Any CUDA device: host$\leftrightarrow$device memcpys of less than 64Kb
- Capability ≥ 1.1: kernel execution and host$\leftrightarrow$device memcpy
- Some devices with capability ≥ 2.0: concurrent execution of two different kernels
- Some devices with capability ≥ 2.0: concurrent execution of two memcpys in different directions (PCI is full duplex!)

See deviceQuery output to check yours (at runtime, check specific booleans in the `deviceProperties`).
Programming guide states that *when an application is run via a CUDA debugger or profiler all launches are synchronous*, but...

Concurrent kernel and host↔device memory transfer is particularly interesting:

Concurrent kernel and host$\leftrightarrow$device memory transfer is particularly interesting:

- Perfect to cover most transfer latencies

Concurrent kernel and host↔device memory transfer is particularly interesting:

- Perfect to cover most transfer latencies
- Available since capability 1.1

Concurrent kernel and host↔device memory transfer is particularly interesting:

- Perfect to cover most transfer latencies
- Available since capability 1.1
- Not too complicated APIs

Concurrent kernel and host↔device memory transfer is particularly interesting:

- Perfect to cover most transfer latencies
- Available since capability 1.1
- Not too complicated APIs

There are three requirements for async memcpys: **page-locked** host memory, use of **streams** and `-async` calls.

Virtual allocable memory on host is bigger than physical memory (RAM). This is possible through a *paging* mechanism that swaps pages from RAM to disk and vice-versa.

Virtual allocable memory on host is bigger than physical memory (RAM). This is possible through a *paging* mechanism that swaps pages from RAM to disk and vice-versa.

Asynchronous memcpys require host memory to be page-locked: even if calling thread is paused, the host memory area subject of transfer should not be paged.

Virtual allocable memory on host is bigger than physical memory (RAM). This is possible through a *paging* mechanism that swaps pages from RAM to disk and vice-versa.

Asynchronous memcpys require host memory to be page-locked: even if calling thread is paused, the host memory area subject of transfer should not be paged.

Allocating too much page-locked memory may decrease overall system performance; it is critical to allocate less space than physical memory.

CUDA offers two simple methods to easily allocate page-locked memory, and one to free it:

```
cudaError_t cudaMallocHost(void **ptr,
    size_t size [, unsigned int flags]);

cudaError_t cudaHostAlloc(void **pHost,
    size_t size, unsigned int flags);

cudaError_t cudaFreeHost(void *ptr);
```

CUDA offers two simple methods to easily allocate page-locked memory, and one to free it:

```
cudaError_t cudaMallocHost(void **ptr,
    size_t size [, unsigned int flags]);

cudaError_t cudaHostAlloc(void **pHost,
    size_t size, unsigned int flags);

cudaError_t cudaFreeHost(void *ptr);
```

`cudaMallocHost()` is a special case of `cudaHostAlloc()` with default parameters: in reference manual, there is no mention to `flags` for `cudaMallocHost()` (official reason: C/C++ interoperability?).

Flags:

`cudaHostAllocDefault` : default settings; causes `cudaHostAlloc()` to emulate `cudaMallocHost()`

`cudaHostAllocPortable` : allocated memory is available to all CUDA contexts, even if created before; the default is that only allocating thread may access it

`cudaHostAllocMapped` : maps the allocation into the CUDA address space (see later)

`cudaHostAllocWriteCombined` : disable caching of mapped memory

Example:

```
#define DIM (1024*1024)
float *harray, *harray_map;
err = cudaMallocHost(&harray, sizeof(float)*DIM);
err = cudaHostAlloc(&harray_map,
    sizeof(float)*DIM,
    cudaHostAllocPortable | cudaHostAllocMapped);
```

...

```
error = cudaFreeHost(harray);
error = cudaFreeHost(harray_map);
```

Overview

Streams are ideal structures used to communicate to the runtime the depencencies/parallelisms of memory operations and kernels.

Streams are ideal structures used to communicate to the runtime the depencencies/parallelisms of memory operations and kernels.

A stream is a sequence of operations to be executed in order; to achieve concurrent kernel/memcpy execution, one has to use at least two streams.

Streams are ideal structures used to communicate to the runtime the depencencies/parallelisms of memory operations and kernels.

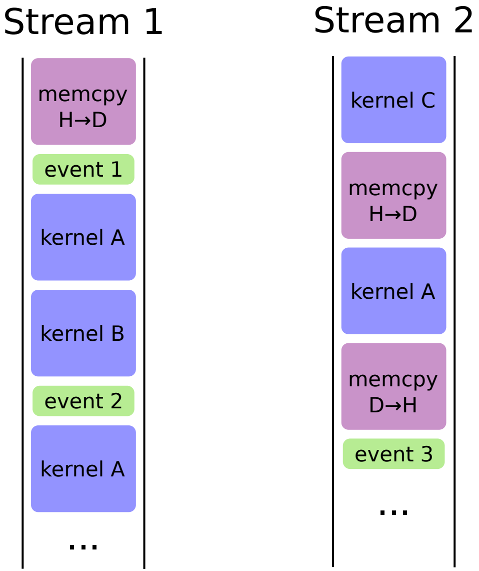
A stream is a sequence of operations to be executed in order; to achieve concurrent kernel/memcpy execution, one has to use at least two streams.

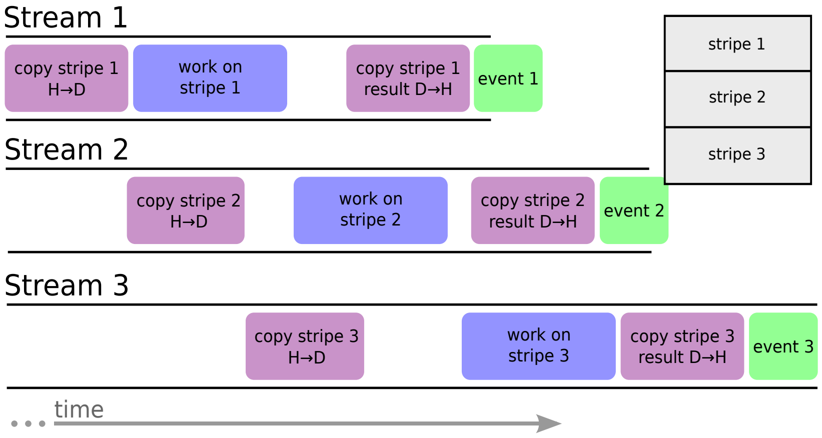
When stream is not specified to a kernel or memcpy operation, the default one (0) is used and operations are not concurrent.

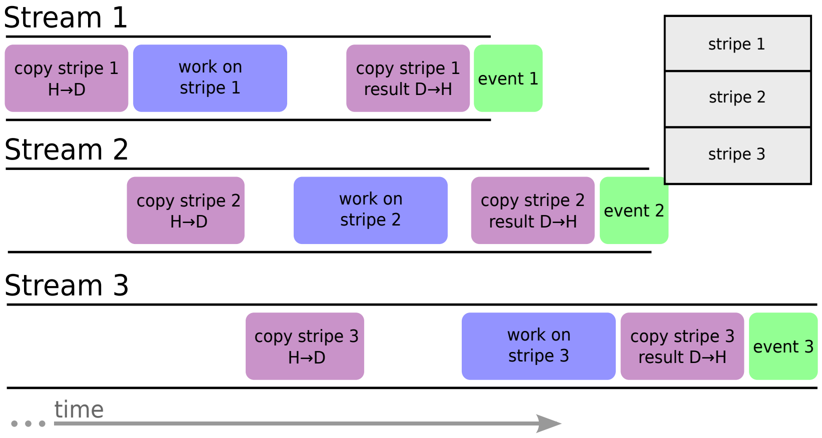
It is possible to enqueue in a stream, other than kernel launches and memory transfers, also CUDA events; they are used as separators for timing and inter-stream dependency purposes.

It is possible to enqueue in a stream, other than kernel launches and memory transfers, also CUDA events; they are used as separators for timing and inter-stream dependency purposes.

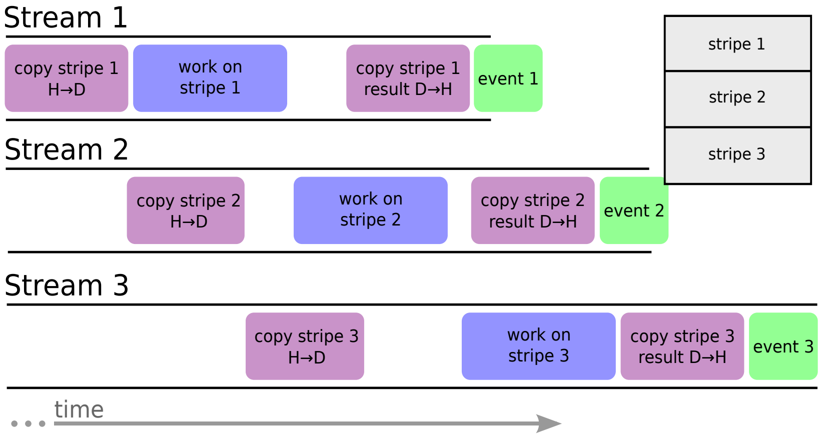
It not possible to create a real dependency graph, but with an appropriate usage of events it is possible to ensure quite complicated dependencies.


Stream 1
memcpy H→D
event 1
kernel A
kernel B
event 2
kernel A
...
Stream 2
kernel C
memcpy H→D
kernel A
memcpy D→H
event 3
...

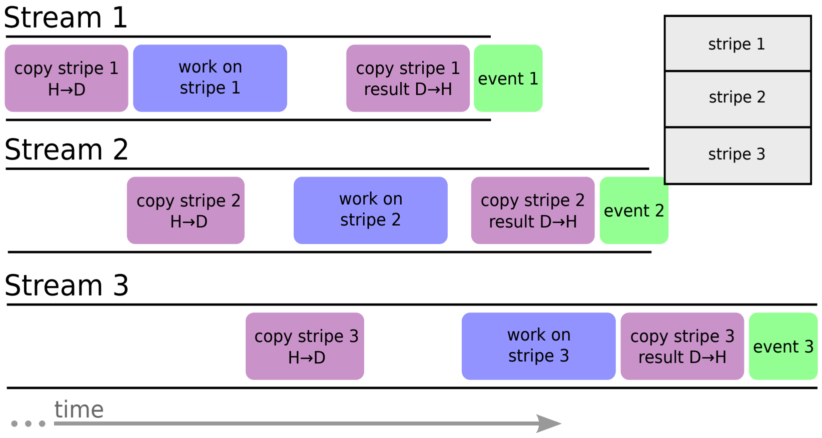

Stream 1
copy stripe 1 H→D
work on stripe 1
copy stripe 1 result D→H
event 1
Stream 2
copy stripe 2 H→D
work on stripe 2
copy stripe 2 result D→H
event 2
Stream 3
copy stripe 3 H→D
work on stripe 3
copy stripe 3 result D→H
event 3
stripe 1
stripe 2
stripe 3
. . . time



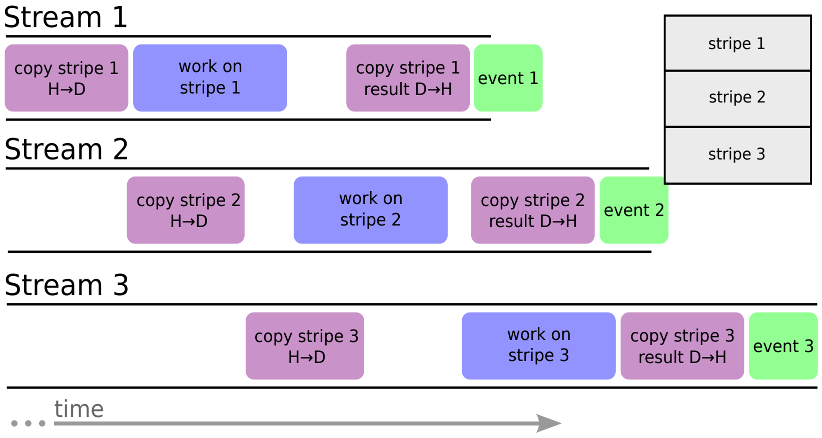
- One kernel at a time is executing



- One kernel at a time is executing
- One memcpy at a time is executing



- One kernel at a time is executing
- One memcpy at a time is executing
- Kenel and memcpys in different streams execute concurrently!



Kernel total time is the same as it was non concurrent; but download and upload times are partially covered, reducing total transfer time from $2 \cdot t$ to $2 \cdot \frac{t}{3}$.
See timeline profiling of SDK samepl simpleStreams

To enqueue a kernel launch in a given stream with high level API, pass the stream as the fourth parameter:

```
cudaStreamCreate(&mystream);
...
my_kernel<<< numBlocks, numThreads,
    0, mystream>>> ( [args] );
```

To enqueue a kernel launch in a given stream with high level API, pass the stream as the fourth parameter:

```
cudaStreamCreate(&mystream);
...
my_kernel<<< numBlocks, numThreads,
    0, mystream>>> ( [args] );
```

Memory transfer methods are identical but with a `stream` parameter and -async suffix:

```
cudaError_t cudaMemcpyAsync(
    void *dst,
    const void *src,
    size_t count,
    enum cudaMemcpyKind kind,
    cudaStream_t stream=0);
```

To enqueue a kernel launch in a given stream with high level API, pass the stream as the fourth parameter:

```
cudaStreamCreate(&mystream);
...
my_kernel<<< numBlocks, numThreads,
    0, mystream>>> ( [args] );
```

Memory transfer methods are identical but with a `stream` parameter and `-async` suffix:

```
cudaError_t cudaMemcpyAsync(
    void *dst,
    const void *src,
    size_t count,
    enum cudaMemcpyKind kind,
    cudaStream_t stream=0);
```

There is a `-async` version of every memcpy method. Note default null stream

Let's see a typical usage example.

Creation:

```
#define NSTREAMS 4
cudaStream_t stream[NSTREAMS];
for (int i = 0; i < NSTREAMS; ++i)
    cudaStreamCreate(&stream[i])
```

Let's see a typical usage example.

Creation:

```
#define NSTREAMS 4
cudaStream_t stream[NSTREAMS];
for (int i = 0; i < NSTREAMS; ++i)
    cudaStreamCreate(&stream[i])
```

Destruction:

```
for (int i = 0; i < NSTREAMS; ++i)
    cudaStreamDestroy(stream[i]);
```

Enqueueing:

```
// use multiple for cycles: prefer breadth first
for (int i = 0; i < NSTREAMS; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size,
        hostPtr + i * size, size,
        cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < NSTREAMS; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size,
        inputDevPtr + i * size, size);

for (int i = 0; i < NSTREAMS; ++i)
    cudaMemcpyAsync(hostPtr + i * size,
        outputDevPtr + i * size, size,
        cudaMemcpyDeviceToHost, stream[i]);
```

Synchronization:

```
// Wait for compute device to finish
cudaError_t cudaThreadSynchronize();

// Wait for a stream to complete everything
cudaError_t cudaStreamSynchronize(cudaStream_t stream);

// Waits for an event to complete
cudaError_t cudaEventSynchronize(cudaEvent_t event);

// Makes the given stream to wait for given
// event before any future operation is
// starte (inter-stream synchronization)
cudaError_t cudaStreamWaitEvent(cudaStream_t stream,
    cudaEvent_t event, unsigned int flags);
```

See the programming guide for more methods (e.g. queries) and implicit synchronization mechanisms

The GPU scheduler keeps two separate queues for kernels and memory operations. When an operation is in progress, it checks the other queue for possibly concurrent operations.

The GPU scheduler keeps two separate queues for kernels and memory operations. When an operation is in progress, it checks the other queue for possibly concurrent operations.

No operations are checked but the first in every queue. This means that **operations enqueued with a depth-first policy will be executed serially**.

(queue scheme)

Some GPUs, especially in notebooks, are integrated on the mainboard and their global memory is a part of the system RAM “shared” with the CPU.

Some GPUs, especially in notebooks, are integrated on the mainboard and their global memory is a part of the system RAM "shared" with the CPU.

In these special cases, do we really need explicit transfers?

If we allocate a host buffer with flag `cudaHostAllocMapped` the buffer is prepared to be *mapped* to a subrange of the address space of thedevice; then, we can obtain a device pointer, pointing to the same physical address, with `cudaHostGetDevicePointer()`:

If we allocate a host buffer with flag cudaHostAllocMapped the buffer is prepared to be *mapped* to a subrange of the address space of thedevice; then, we can obtain a device pointer, pointing to the same physical address, with cudaHostGetDevicePointer():

```
// flags must be 0
cudaError_t cudaHostGetDevicePointer(
    void **pDevice, void *pHost,
    unsigned int flags);
```

```
// alloc buffer
cudaHostAlloc((void**)&host_array,
    sizeof(float)*SIZE, cudaHostAllocMapped);

// init data
for(i=0; i < SIZE; i++)
    host_array[i]=(float)rand();

// get mapped pointer (flags must be 0)
cudaHostGetDevicePointer((void**)&device_pointer,
    (void*)host_array, 0);

// launch kernel: direct access to buffer!
vectorAdd<<<nblocks, nthreads>>>
    (device_pointer, SIZE);
```

The mapped buffer is cached. We can disable caching using the flag `cudaHostAllocWriteCombined` when allocating.

The mapped buffer is cached. We can disable caching using the flag `cudaHostAllocWriteCombined` when allocating.

Standing on the programming guide, reading from a "write combined" buffer may be much faster (up to 40%), but writing on it from host may be expensive. It is recommended only when device reads alot, host writes only once.

Hands on code: aynchronous operations & timeline profiling