CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Introduction to GPUs in HPC

Ben Cumming, CSCS
July 15, 2019

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Concurrency

Concurrency

Concurrency is the ability to perform multiple CUDA operations simultaneously, including:

- CUDA kernels;
- Copying from host to device;
- Copying from device to host;
- Operations on the host CPU.

Concurrency enables

- Both CPU and GPU to work at the same time.
- Multiple tasks to run on GPU simultaneously.
- Overlapping of communication and computation.

Host code

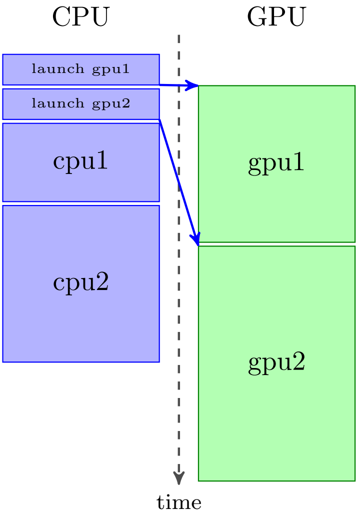
```
gpu1<<<...>>>(...);
gpu2<<<...>>>(...);
cpu1(...);
cpu2(...);
```

The host:

- launches the two CUDA kernels;
- then executes host calls sequentially.

The GPU:

- executes asynchronously to host;
- executes kernels sequentially.



The CUDA language and runtime libraries provide mechanisms for coordinating asynchronous GPU execution:

- **CUDA streams** can concurrently run independent kernels and memory transfers;
- **CUDA events** can be used to synchronize streams and query the status of kernels and transfers.

Streams

A CUDA stream is a sequence of operations that execute in **issue order** on the GPU.

- CUDA operations are kernels and copies between host and device memory spaces.

Streams and concurrency

- Operations in different streams **may** run concurrently
  - requires sufficient resources on the GPU (registers, shared memory, SMXs, etc).
- Operations in the same stream **are** executed sequentially.
- If no stream is specified, all kernels are launched in the default stream.

Managing streams

A stream is represented using a `cudaStream_t` type

- `cudaStreamCreate(cudaStream_t* s)` and `cudaStreamDestroy(cudaStream_t s)` can be used to create and free CUDA streams respectively.
- To launch a kernel on a stream specify the stream id as a fourth parameter to the launch syntax:

```
kernel<<<grid_dim, block_dim, shared_size, stream>>>(...)
```

- The default CUDA stream is the `NULL` stream, or stream 0 (`cudaStream_t` is an integer).

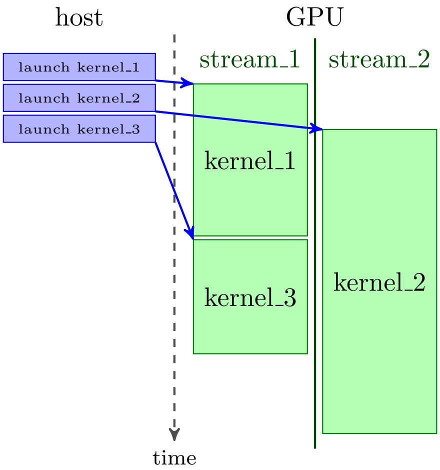
Basic cuda stream usage

```
// create stream
cudaStream_t stream;
cudaStreamCreate(&stream);
// launch kernel in stream
my_kernel<<<grid_dim, block_dim, shared_size, stream>>>(..)
// release stream when finished
cudaStreamDestroy(stream);
```

Host code

```
kernel_1<<<_,_,_,stream_1>>>();
kernel_2<<<_,_,_,stream_2>>>();
kernel_3<<<_,_,_,stream_1>>>();
```

- `kernel_1` and `kernel_3` are serialized in `stream_1`.
- `kernel_2` can run asynchronously in `stream_2`.
- **Note** `kernel_2` will only run concurrently if there are sufficient resources available on the GPU, i.e. if `kernel_1` is not using all of the SMXs.



Asynchronous copy

```
cudaMemcpyAsync(*dst, *src, size, kind, cudaStream_t stream = 0);
```

- Takes an additional parameter stream, which is 0 by default.
- Returns immediately after initiating copy:
  - Host can do work while copy is performed;
  - Only if **pinned memory** is used.
- Copies in the same direction (i.e. H2D or D2H) are serialized.
  - Copies from host→device and device→host are concurrent if in different streams.

Pinned memory

Pinned (or page-locked) memory will not be paged out to disk:

- The GPU can safely remotely read/write the memory directly without host involvement;
- Only use for transfers, because it easy to run out of memory.

Managing pinned memory

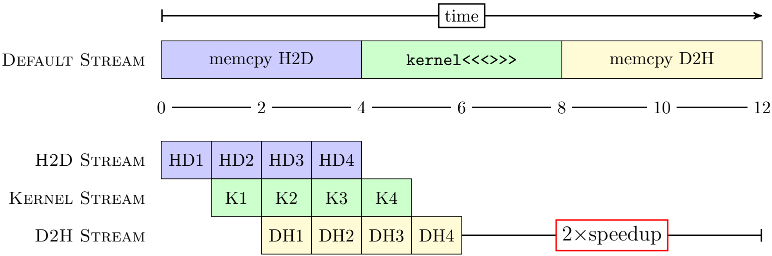
`cudaMallocHost(**ptr, size);` and `cudaFreeHost(*ptr);`

- Allocate and free pinned memory (`size` is in bytes).

Asynchronous copy example: streaming workloads

Computations that can be performed independently, e.g. our `axpy` example:

- Data in host memory has to be copied to the device, and the result copied back after the kernel is computed.
- Overlap copies with kernel calls by breaking the data into chunks.



CUDA events

To implement the streaming workload we have to coordinate operations on the GPU. CUDA events can be used for this purpose.

- Synchronize tasks in different streams, e.g.:
  - Don't start kernel in kernel stream until data copy stream has finished;
  - Wait until required data has finished copy from host before launching kernel.
- Query status of concurrent tasks:
  - Has kernel finished/started yet?
  - How long did a kernel take to compute?

Managing events

`cudaEventCreate(cudaEvent_t*);` and `cudaEventDestroy(cudaEvent_t);`

- Create and free `cudaEvent_t`.

`cudaEventRecord(cudaEvent_t, cudaStream_t);`

- Enqueue an event in a stream.

`cudaEventSynchronize(cudaEvent_t);`

- Make host execution wait for event to occur.

`cudaEventQuery(cudaEvent_t)`

- Test if the work before an event in a queue has been completed.

`cudaEventElapsedTime(float*, cudaEvent_t, cudaEvent_t);`

- Get time between two events.

Using events to time kernel execution

```
cudaEvent_t start, end;
cudaStream_t stream;
float time_taken;

// initialize the events and streams
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaStreamCreate(&stream);

cudaEventRecord(start, stream); // enqueue start in stream
my_kernel<<<grid_dim, block_dim, 0, stream>>>();
cudaEventRecord(end, stream);   // enqueue end in stream
cudaEventSynchronize(end);      // wait for end to be reached
cudaEventElapsedTime(&time_taken, start, end);

std::cout << "kernel took " << 1000*time_taken << " s\n";

// free resources for events and streams
cudaEventDestroy(start);
cudaEventDestroy(end);
cudaStreamDestroy(stream);
```

Copy→kernel synchronization

```
cudaEvent_t event;
cudaStream_t kernel_stream, h2d_stream;
size_t size = 100*sizeof(double);
double *dptr, *hptr;

// initialize
cudaEventCreate(&event);
cudaStreamCreate(&kernel_stream);
cudaStreamCreate(&h2d_stream);

cudaMalloc(&dptr, size);
cudaMallocHost(&hptr, size); // use pinned memory!

// start asynchronous copy in h2d_stream
cudaMemcpyAsync(dptr, hptr, size,
                cudaMemcpyHostToDevice, h2d_stream);
// enqueue event in stream
cudaEventRecord(event, h2d_stream);
// make kernel_stream wait for copy to finish
cudaStreamWaitEvent(kernel_stream, event, 0);
// enqueue my_kernel to start when event has finished
my_kernel<<<grid_dim, block_dim, 0, kernel_stream>>>();

// free resources for events and streams
cudaEventDestroy(event);
cudaStreamDestroy(h2d_stream);
cudaStreamDestroy(kernel_stream);
cudaFree(dptr);
cudaFreeHost(hptr);
```

Exercises

1. Open `include/util.hpp` and understand `copy_to_{host/device}_async()` and `malloc_pinned_host()`
2. Open `include/cuda_event.h` and `include/cuda_stream.h`
   - what is the purpose of these classes?
   - what does `cuda_stream::enqueue_event()` do?
3. Open `async/memcopy1.cu` and run
   - what does the benchmark test?
   - what is the effect of turning on `USE_PINNED`?
     Hint: try small and large values for `n` (8, 16, 20, 24)
4. Inspect `async/memcopy2.cu` and run
   - what effect does changing the number of chunks have?
5. Inspect `async/memcopy3.cu` and run
   - how does it differ from `memcopy2.cu`?
   - what effect does changing the number of chunks have?

Using events to time kernel execution: **with helpers**

```
CudaStream stream(true);

auto start = stream.enqueue_event();
my_kernel<<<grid_dim, block_dim, 0, stream.stream()>>>();
auto end = stream.enqueue_event();
end.wait();
auto time_taken = end.time_since(start);

std::cout << "kernel took " << 1000*time_taken << " s\n";
```

Copy→kernel synchronization: **with helpers**

```
CudaStream kernel_stream(true), h2d_stream(true);
auto size = 100;
auto dptr = device_malloc<double>(size);
auto hptr = pinned_malloc<double>(size);

copy_to_device_async<double>(hptr,dptr,size,h2d_stream.stream());
auto event = h2d_stream.enqueue_event();
kernel_stream.wait_on_event(event);
my_kernel<<<grid_dim, block_dim, 0, kernel_stream.stream()>>>();

cudaFree(dptr);
cudaFreeHost(hptr);
```

Rough guidelines for concurrency

Ideally for most workloads you don't want to rely on streams to fill the GPU with work.

- A sign that the working set per GPU is not large enough;
- Full concurrency is difficult in practice;
  - A low-level optimization strategy for the last few %.
- This isn't a hard and fast rule.

Streams come into their own for overlapping communication and computation.

- Possible to transfer data in both directions concurrently with kernel execution.