Lecture 7: CUDA Streams and dynamic parallelism

Informatik elective: GPU Computing

Pratik Nayak

In this session

- Asynchronous programming with CUDA streams
- Dynamic parallelism in CUDA

What is a kernel ?

- An C++ extension provided by CUDA to define functions that are executed N times in parallel by N different CUDA threads.
  - Defined with a `__global__` specifier.
  - The launch configuration (number of threads and organization) defined by `<<<...>>>` has to be specified.
  - The executing thread is given a unique thread ID, accessible within the kernel call.
  - Cannot be a class member and must have return type `void`
  - Call to a `__global__` function is asynchronous. The return back to the caller is immediate, before the device execution has completed.

Function specifiers in CUDA C++

- `__global__`: Declaration of a function as a kernel.
  - Executed on the device
  - Callable from the host
  - Also callable from the device for CC > 5.0 (Kernels from within kernels: Dynamic parallelism)
- `__device__`: Declaration of a device function.
  - Executed on the device, and compiled for the device.
  - Callable ONLY from the device.
- `__host__`: Declaration of a host function.
  - Executed on the host, and compiled for the host.
  - Callable ONLY from the host.

Function specifiers in CUDA C++

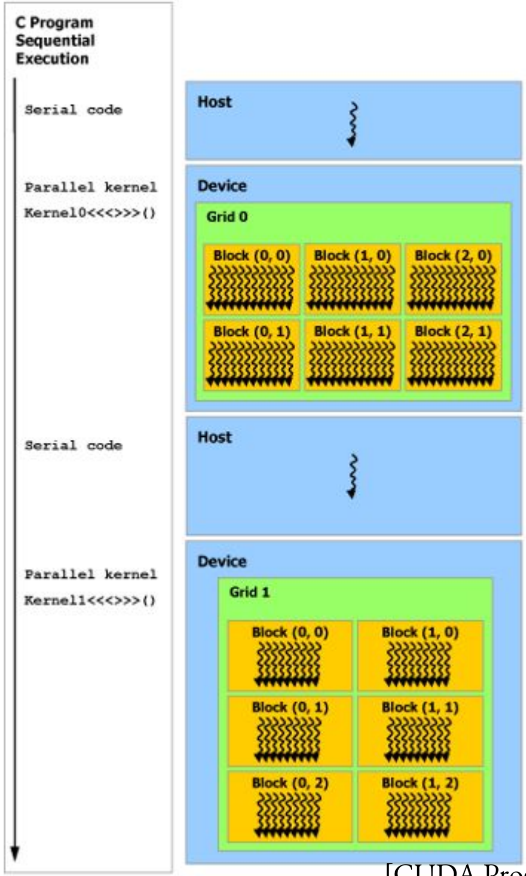
- `__global__`: Declaration of a function as a kernel.
  -
  -
  - m)
- `__d`
  -
  -
- `__h`
  - Executed on the host, and compiled for the host.
  - Callable ONLY from the host.

A function can be declared both `__device__` and `__host__` to specify that it should be compiled for both host and device and can be executed on both.

```
__host__ __device__ int update_value(...)
```

Kernel execution on device

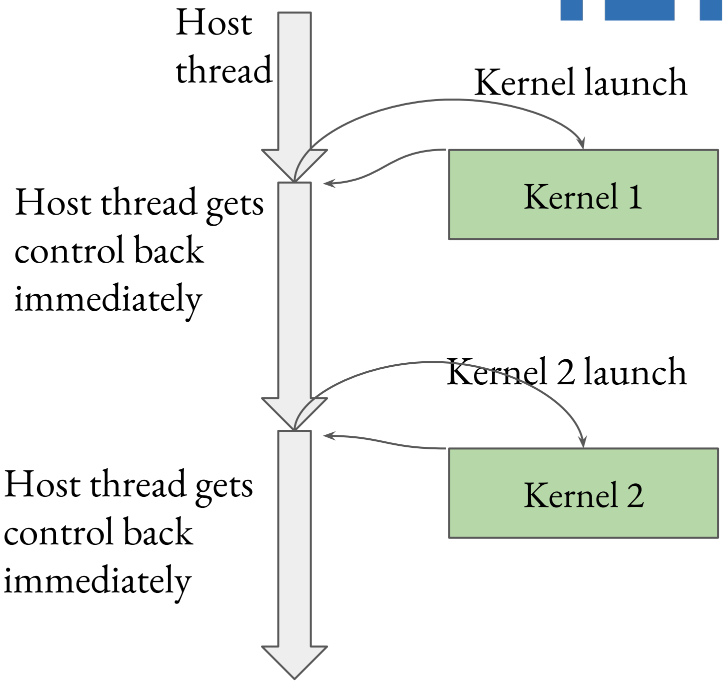
- When running a single-threaded CPU program, two different kernels can be interleaved with host code.
- As the device kernel launches are asynchronous with respect to host, host and device code can run concurrently.
- By default, device kernels launched one after the other will be serialized on the default CUDA "stream"



[CUDA Programming guide]

Asynchronous execution

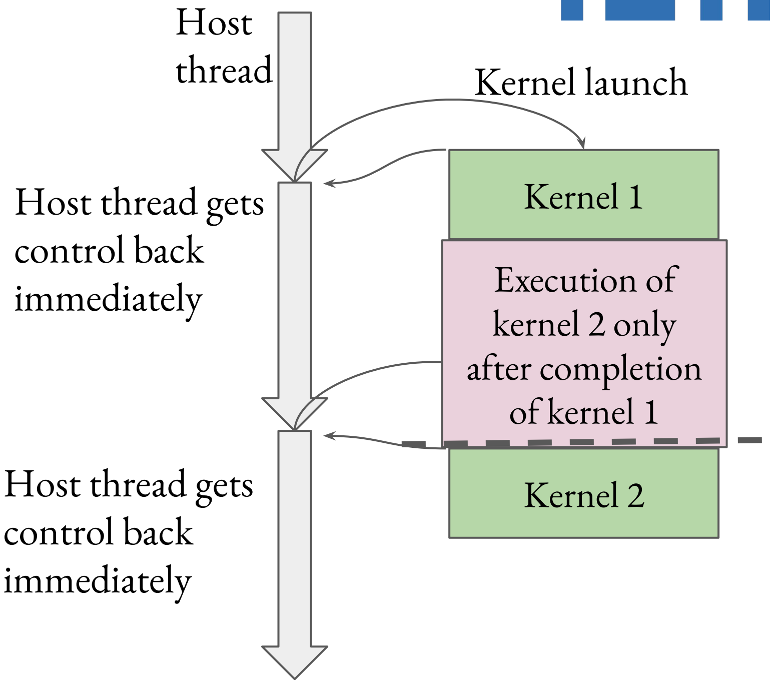
- The kernel launches are always asynchronous with respect to the host.



Asynchronous execution

- The kernel launches are always asynchronous with respect to the host.
- By default two subsequently launched kernels are serialized on the default stream

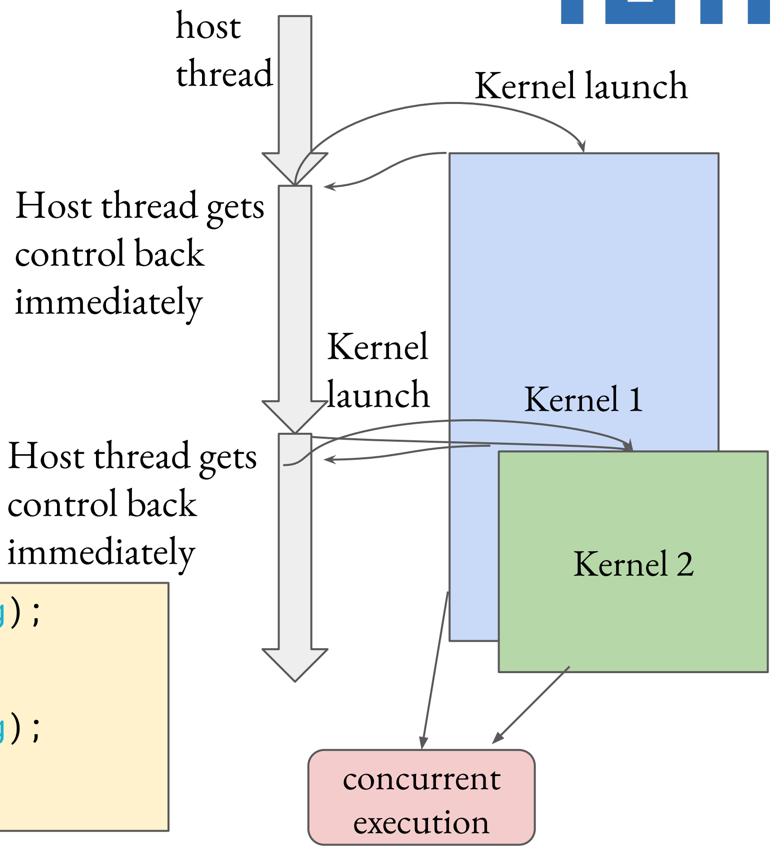
```
kernel<<<grid, block, shmem, stream>>>(...)
// By default, stream is NULL.
kernel<<<grid, block, shmem>>>(...)
```



Asynchronous execution

- This is inefficient if Kernel 1 and Kernel 2 have no data-dependencies.
- CUDA allows you to launch two independent kernels on two different streams, enabling concurrent GPU execution.

```
cudaStreamCreateWithFlags(&stream1, cudaStreamNonblocking);
kernel1<<<..., stream1>>>(...);
cudaStreamCreateWithFlags(&stream2, cudaStreamNonblocking);
kernel2<<<..., stream2>>>(...)
```



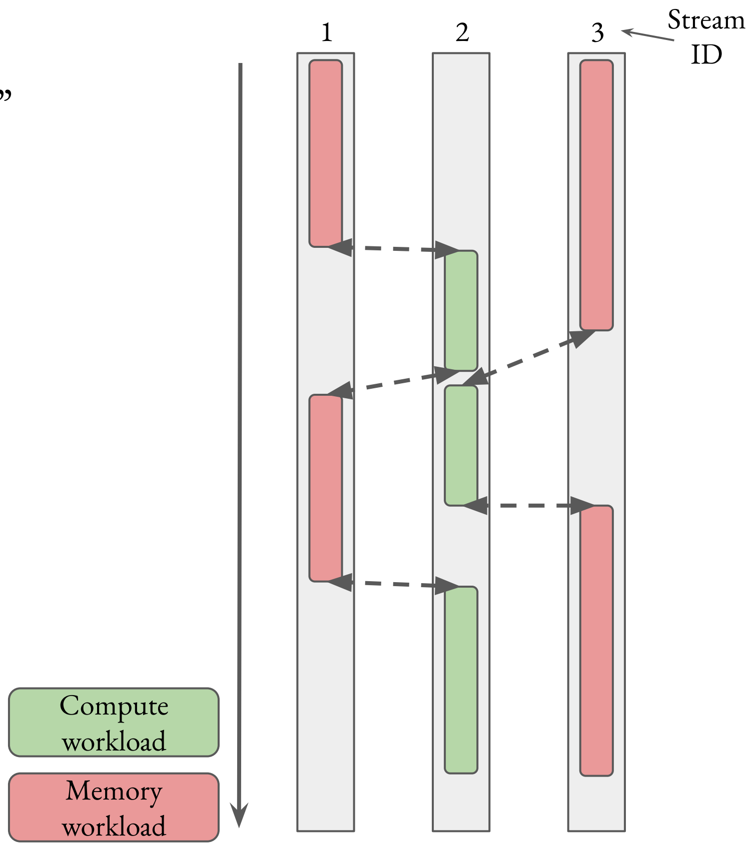
Stream API: basics

```
cudaStreamCreateWithFlags(&stream, flag); // __host__ and __device__
cudaStreamDestroy(stream);  // __host__ and __device__
cudaStreamSynchronize(stream);  // Only __host__, Caller synchronizes with stream
flag = {cudaStreamDefault,     // Default stream
        cudaStreamNonBlocking} // Non-blocking stream, no implicit synchronization with
                                  default stream
```

- Need to create and destroy required streams.
- On host, you can explicitly synchronize two streams.
- A `cudaDeviceSynchronize()` synchronizes all streams on device.

Stream behaviour

- You can think of streams as independent "execution pipelines" with some workload in them.
- They ***may*** be executed concurrently on the GPU, but are not guaranteed to.
- The runtime makes the scheduling decisions based on the available resources.
- Workloads mapped to one stream always run in sequence.
- Explicit synchronization necessary to exchange data between parallel workload in different streams.



Better control with Events

```
cudaEventCreateWithFlags(&event, flag); // __host__ and __device__
cudaEventDestroy(event);  // __host__ and __device__
cudaEventSynchronize(event);  // Only __host__, Waits for the event to complete
flag = {cudaEventDefault,     // Default stream
        cudaEventBlockingSync} // Enable host thread to block on event created with
                                 this flag
```

- As order of execution of workloads between different streams is unspecified, explicitly managing dependencies with synchronizations can be difficult.
- CUDA provides Event API to map specific operations to events called CUDA Events

Managing events on streams

```
cudaEventRecord(event, stream); // Record an event on a stream
cudaError_t cudaEventQuery(event); // Queries all work recorded by
                                      event and returns cudaSuccess if all work has completed.
cudaStreamWaitEvent(stream, event);  // Make a stream wait on an event. All future work
                                        submitted to this stream waits on event.
cudaEventElapsedTime(elapsed_time, event1, event2); // Computes elapsed time between events,
                                                       event1 and event2
```

- Just creating an event is not sufficient. You need to map an event to the stream. You can then synchronize and query at the event-level.
- Events are designed to be light-weight, so the overhead of creation, recording and querying should be low.
- Finer time measurement is possible with events → using GPU clock → independent of OS

Asynchronous memory operations

```
cudaMemcpyAsync(dest, src, byte_count, kind, stream) // __host__ and __device__
dest: destination memory address
src: source memory address
byte_count: Number of bytes to copy
kind: cudaMemcpyHostToDevice/ DeviceToHost etc, specifies direction of copy.
stream: The stream to associate the copy operation with
```

- Similar to kernel launches, you can assign a stream to memory copies.
- Enables overlap of computation and memory operations in different streams.
- Semantics same as before: In single streams, operations are executed in-order; in different streams order of operation is not guaranteed without explicit synchronizations.

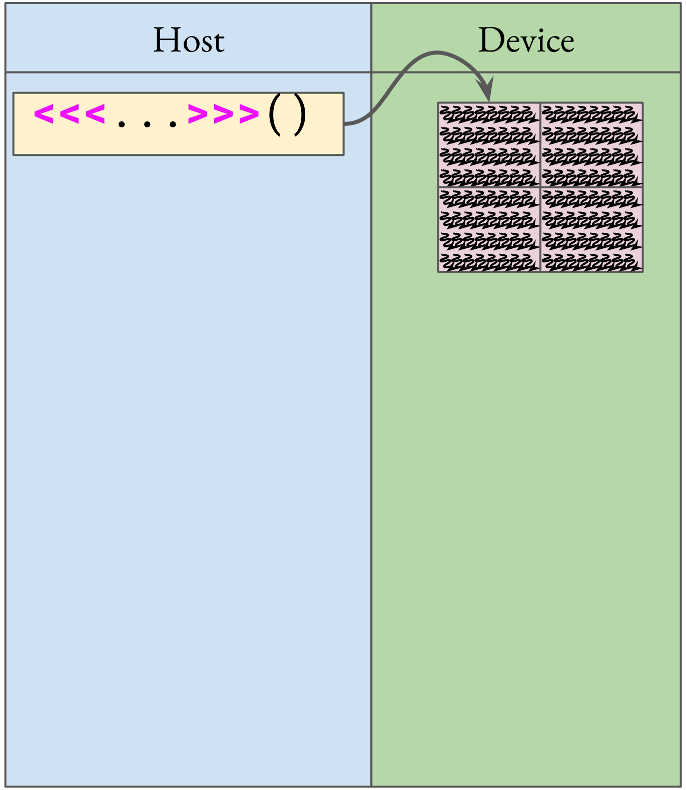
Stream-ordered memory allocator

```
cudaMallocAsync(ptr, stream); // Allocates memory with stream ordered semantics
cudaFreeAsync(ptr, stream); // Frees memory with stream ordered semantics
```

- `cudaMalloc` and `cudaFree` implicitly synchronize the device, which is necessary due to page management.
- CUDA provides stream-order allocator to allocate and free memory on streams, to remove the necessity for the implicit synchronization.
- The user is responsible for using the memory only in the promised stream order, otherwise the behaviour is undefined.

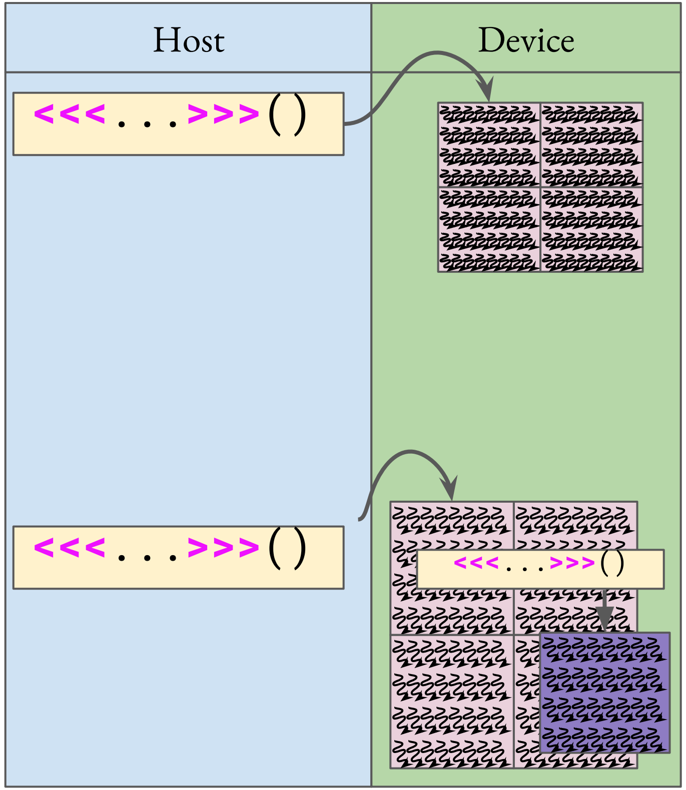
Dynamic parallelism in CUDA

- Till now we looked at how we can launch one kernel from the host with the <<<...>>>() syntax.
- Works well for algorithms with a flat, single-level of parallelism
- For implementations with recursion or irregular loop structure, these needed to be modified/control needed to be returned to host etc.



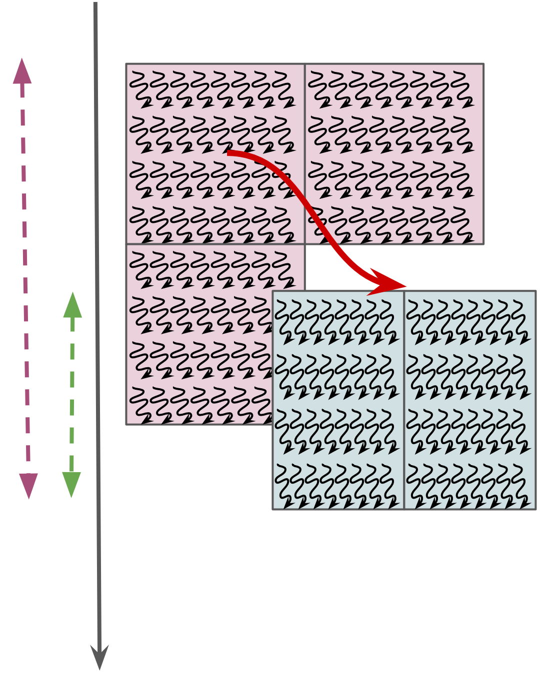
Dynamic parallelism in CUDA

- Till now we looked at how we can launch one kernel from the host with the <<<...>>>() syntax.
- Works well for algorithms with a flat, single-level of parallelism
- For implementations with recursion or irregular loop structure, these needed to be modified/control needed to be returned to host etc.
- With dynamic parallelism, you can launch kernels from inside kernels.

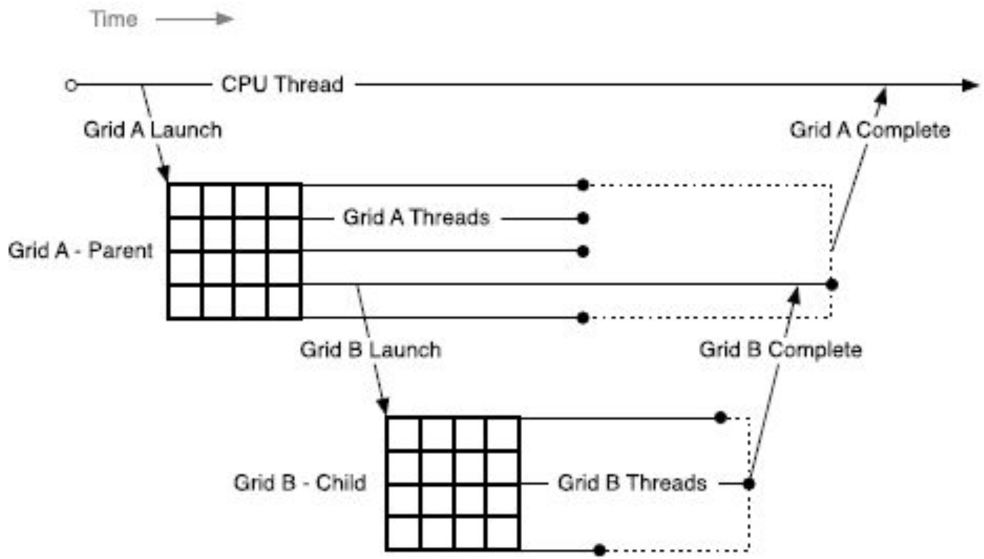


Some terminology

- <u>Grid</u>: A collection of threads which <u>execute a kernel</u>
- <u>Thread block:</u> A group of threads that execute on the same multiprocessor (SM).
- <u>Kernel:</u> An function that executes on the device with the CUDA execution and memory model for <u>every thread in the grid</u>
- <u>Parent:</u> A grid/thread-block/thread that has launched new grid(s) called child grid(s). It does not complete until all its children have also finished.
- <u>Child:</u> A grid/thread-block/thread which has been launched by a parent. It has to complete before its parent is considered to be completed.



Nested launches



Runtime guarantees implicit synchronization between parent and child grid

[CUDA Programming guide]

Launch syntax and behaviour

```
kernel_name<<< grid_size, block_size, sm_bytes, stream >>>([kernel arguments]);
```

- Device-side kernel launch syntax is the same standard host-kernel launch syntax.
- Similar to host-launches, device-kernel launches are also asynchronous wrt the launching thread. That is, `<<<>>>( )` call returns immediately, and the launching thread continues to execute.
- Any configuration setup (L1 cache, shared memory etc) will be inherited from the parent grid.

Stream behaviour

```
kernel_name<<< grid_size, block_size, sm_bytes, stream >>>([kernel arguments]);
```

- `stream` has to be created in the same grid that the kernel is being launched from.
- The `NULL` stream has different behaviour → It does not insert an implicit dependency.
- Named streams (not `NULL`) have to be created with `...CreateWithFlags()` with the `cudaStreamNonBlocking` flag.

Tail launch

- As `cudaDeviceSynchronize()` is too restrictive for device kernel launches, a specific named stream is available: `cudaStreamTailLaunch`, providing the same functionality.
  - Each grid has its own tail launch stream, enabling automatic ordering of subsequent kernel launches with proper implicit synchronization.

```
// In this example, C2 will only launch after C1 completes.
__global__ void P( ... ) {
   C1<<< ... , cudaStreamTailLaunch >>>( ... );
   C2<<< ... , cudaStreamTailLaunch >>>( ... );
}
```


Fire and forget

- In many cases, you don't care about the dependencies (example for some independent task).
- CUDA provides a fire-and-forget named stream for this purpose: `cudaStreamFireAndForget`
- No need to create a new stream per launch.
- No stream-tracking overhead.
- Will implicitly synchronize with a tail launch stream

```
// In this example, C2's launch will not wait for C1's completion
__global__ void P( ... ) {
   C1<<< ... , cudaStreamFireAndForget >>>( ... );
   C2<<< ... , cudaStreamFireAndForget >>>( ... );
}
```

Event support

- For device-kernel launches, events are supported, but restricted.
- `cudaStreamWaitEvent()` is supported, but event synchronization and timings are not supported.
- Events must be created with the `cudaEventDisableTiming` flag.

Summary

- CUDA streams and events: Enabling asynchronous computatations
- CUDA dynamic parallelism: Enabling better expression for more complex programs

Next lecture

- No lecture next week due to Dies Academicus

https://www.tum.de/en/studies/application/application-info-portal/dates-periods-and-deadlines/