High Performance Computing I
WS 2019/2020
Session #30
CUDA streams

Andreas F. Borchert and Michael C. Lehn
Ulm University

February 10, 2020

CUDA streams

- So far, nearly all CUDA activities were implicitly serialized. Just kernel function invocations were done asynchronously
- This limits parallelization potentials.
- CUDA streams are an abstraction that allows to specify serialized sequences which are independent from each other and which, in principle, can be parallelized.

Following activities can be executed in parallel, provided they are associated with different CUDA streams:

- CPU and GPU can operate independently from each other
- Transfer of data and the execution of kernel functions
- Multiple kernels can be executed in parallel on the same GPU (the Pascal microarchitecture on Livingstone supports up to 32 kernels in parallel)
- Multiple GPUs can run in parallel

cudaError_t cudaStreamCreate (cudaStream_t∗ stream)

Create a new CUDA stream. *cudaStream_t* is a pointer to a non-public data structure which may be copied.

cudaError_t cudaStreamSynchronize (cudaStream_t stream)

Wait until all activities associated with *stream* are finished.

cudaError_t cudaStreamDestroy (cudaStream_t stream)

Wait until all activities associated with *stream* are finished and release the associated ressources.

Asynchronous data transfer operations support a stream parameter:

cudaError_t cudaMemcpyAsync(**void**∗ *dst*, **const void**∗ *src*, *size_t count*, **enum** *cudaMemcpyKind kind*, *cudaStream_t stream*)

Works like *cudaMemcpy* but does not implicitly synchronize. Instead the copy operation is added to the operations associated with *stream* and will be executed after the previous operations of *stream* are finished.

The kernel configuration allows to specify a stream:

- $<<<$ *Dg*, *Db*, *Ns*, *S* $>>>$
- The last paraemter *S* is a stream.
- *Ns* can be specified as 0 (in this case we get default behaviour).

- A null pointer (i.e. **nullptr**) is permitted as stream.
- In this case, the operation will begin after all previously schedules CUDA operations are finished. (Much like an implicit call to *cudaDeviceSynchronize*.)
- Nonetheless, asynchronous operations will be non-blocking, i.e. the CPU can simply continue.
- All data transfers without a stream parameter imply the use of the NULL stream.

Pipelining can be helpful to boost performance if we have multiple kernel invocations with associated data transfers which are independent from each other:

- A stream is created for each kernel call invocation.
- Each stream is usually associated with at least three operations:
  - Data transfer to the device
  - Invocation of the kernel function
  - Data transfer from the device to the host

copy with support of streams

hpc/cuda/copy.hpp

```
template<
   template<typename> class MatrixA,
   template<typename> class MatrixB,
   typename T,
   Require<HostGe<MatrixA<T>>, DeviceGe<MatrixB<T>>> = true
>
void copy(const MatrixA<T>& A, MatrixB<T>& B, Stream& stream) {
   assert(A.numRows() == B.numRows() && A.numCols() == B.numCols() &&
      A.incRow() == B.incRow() && A.incCol() == B.incCol() &&
      consecutively_stored(A) && consecutively_stored(B));
   CHECK_CUDA(cudaMemcpyAsync, B.data(), A.data(),
      A.numRows() * A.numCols() * sizeof(T), cudaMemcpyHostToDevice,
      stream);
}
```

copy with support of streams

hpc/cuda/copy.hpp

```
template<
   template<typename> class MatrixA,
   template<typename> class MatrixB,
   typename T,
   Require<DeviceGe<MatrixA<T>>, HostGe<MatrixB<T>>> = true
>
void copy(const MatrixA<T>& A, MatrixB<T>& B, Stream& stream) {
   assert(A.numRows() == B.numRows() && A.numCols() == B.numCols() &&
      A.incRow() == B.incRow() && A.incCol() == B.incCol() &&
      consecutively_stored(A) && consecutively_stored(B));
   CHECK_CUDA(cudaMemcpyAsync, B.data(), A.data(),
      A.numRows() * A.numCols() * sizeof(T), cudaMemcpyDeviceToHost,
      stream);
}
```