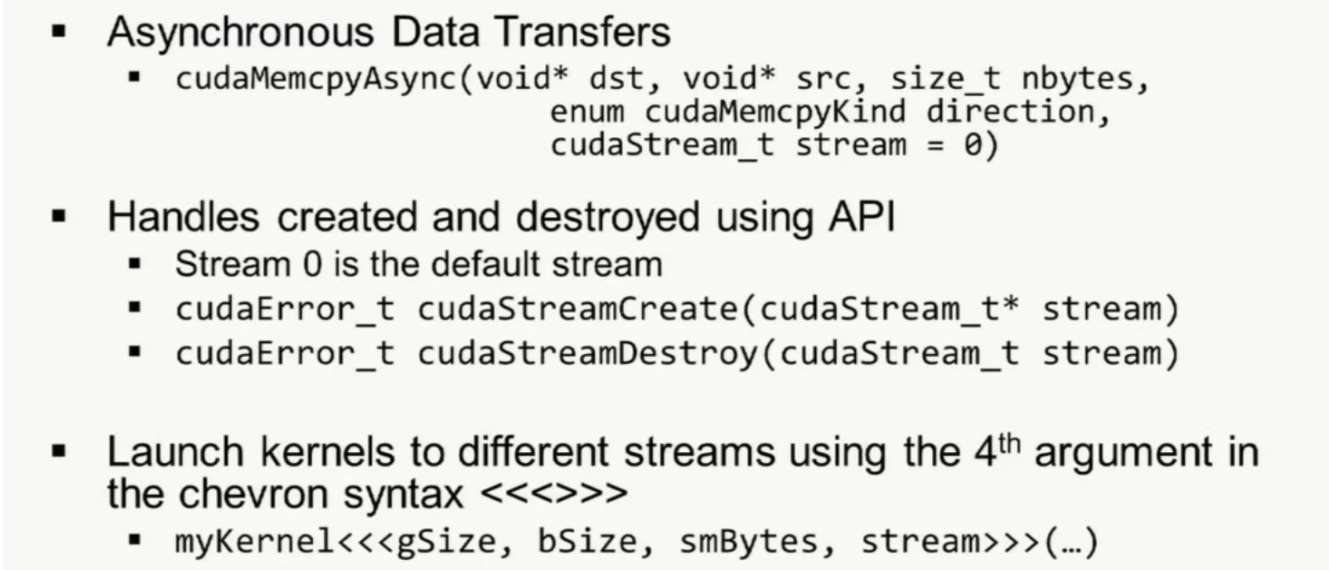
asynchronous operations & dynamic parallelism

- the high level goal here is to do useful things in the CPU while the GPU is executing a kernel. this could be
- remember, our kernel launches are already asynchronous. To wait for the GPU device to finish computation, we usually use cudaDeviceSynchronize() or cudaMemcpy() from device to host. mem copy blocks until GPU is done computing and copies over results after device is done with its computations
- if you have two kernel calls one after the other, each will return immediately and just get queued up. after the first one is done, the second one goes. meanwhile, after the two calls we can call a cpu function that does work on the CPU like reading the next batch of images for training from the disk or from a object store via the network.
- To transfer data while the GPU is doing something, there are asynchronous variations of cudaMemcpy() → cudaMemcpyasync()
- Streams: a stream is a sequence of operations that execute in order on the GPU. we can ask the gpu to run a set of streams. within the streams, the operations are sequential, but across streams scheduling order is undefined. so if you have data dependencies, make sure to keep them in the same stream one after the other. if there are no dependencies take advantage of parallelism and schedule as separate streams.
- stream syntax



- if no streams are defined, there is a default stream to which all operations are queued.
- the problem with transfers from host memory is that host memory is usually virtualized (virtual memory) in all modern OSes. transferring data that is not paged out to disk is much faster than if it has to be fetched from disk and then transferred. so for applications that run with some special privileges we can ask the OS to pin some memory so it does not get swapped out to disk. the details for how to do this are below

- Page-locked (“pinned”) host memory required for asynchronous transfers
  - GPU can transfer data to/from host if host buffer isn’t pageable
- `cudaMemcpyAsync()` does not require page-locked host memory
  - HOWEVER then transfers are synchronous
- `cudaMallocHost()` or `cudaHostAlloc()` and `cudaFreeHost()`
  - Allocate/free page-locked memory
- `cudaHostRegister()`/`cudaHostUnregister()`
  - Make existing memory page-locked

- cudaDeviceSynchronize() waits for all streams to finish.
- we can wait for specific streams to finish by calling cudaDeviceSynchronize(stream1)
- if you’d like to synchronize to check that a stream has reached a particular state, you can use an `event` for that. simply create an event, and fire it off at the right place in the stream of operations and call cudaEvenSynchronize(event) from the host to wait for that event to complete before continuing.
- cudaEventQuery() is a non blocking call, unlike synchronize which blocks. so if you want to decide how much work to do on the GPU
- dynamic parallelism allows a thread in a block in a grid to start a new kernel in a new grid that has a new block and each has threads.
- the goal of dynamic parallelism is if you have a bunch of computation to be done on the GPU, you shouldn’t have to keep moving data to host and schedule kernels from host.
- dynamic parallelism is well suited to algorithms that use recursive subdivision like mergesort and quicksort
- how do we launch kernels from within a kernel? just like we would from the host! streams work too. remember → you have to use global memory or constant memory to pass data. shared memory isn’t visible by the new kernel since its a new grid with new blocks and threads!