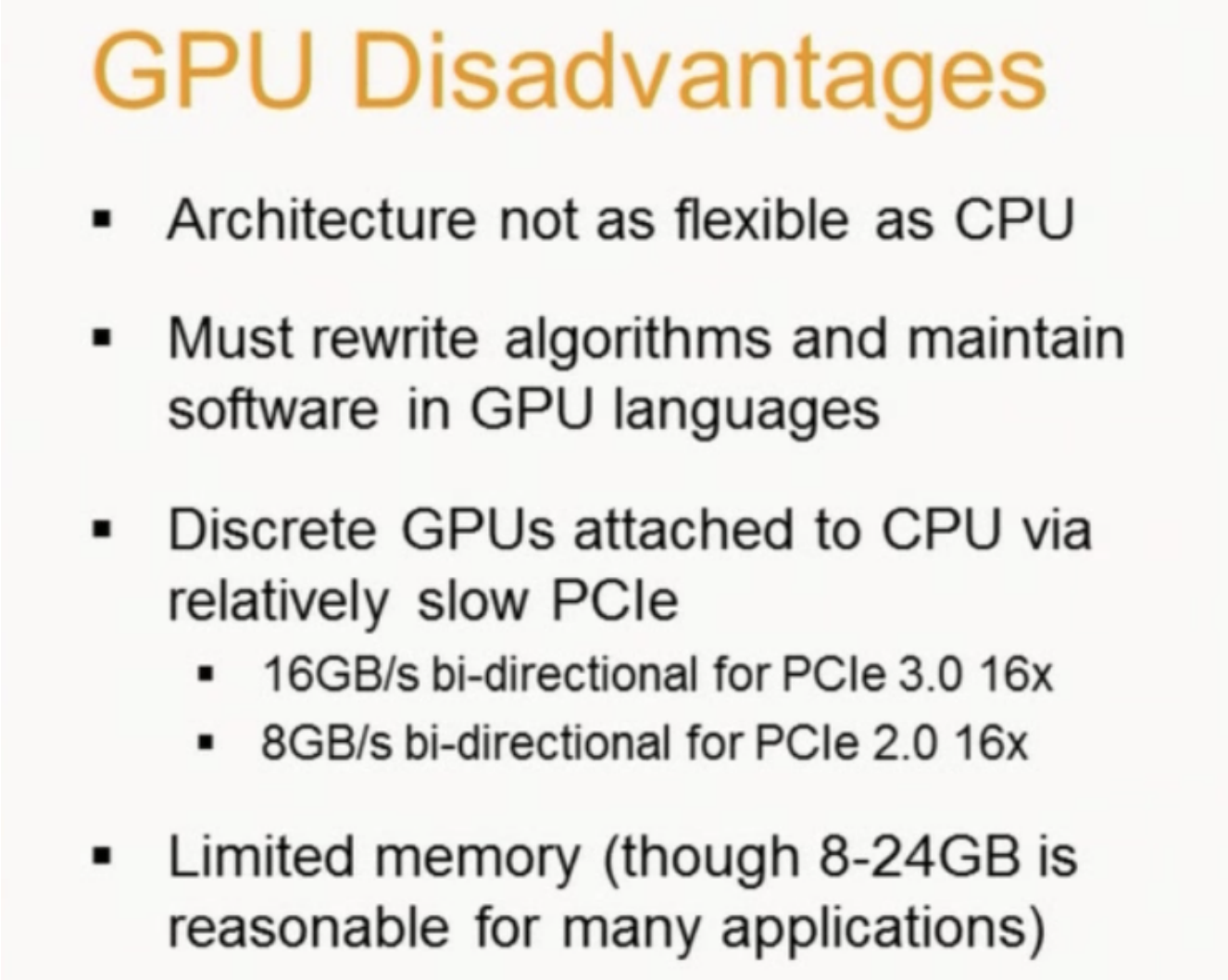
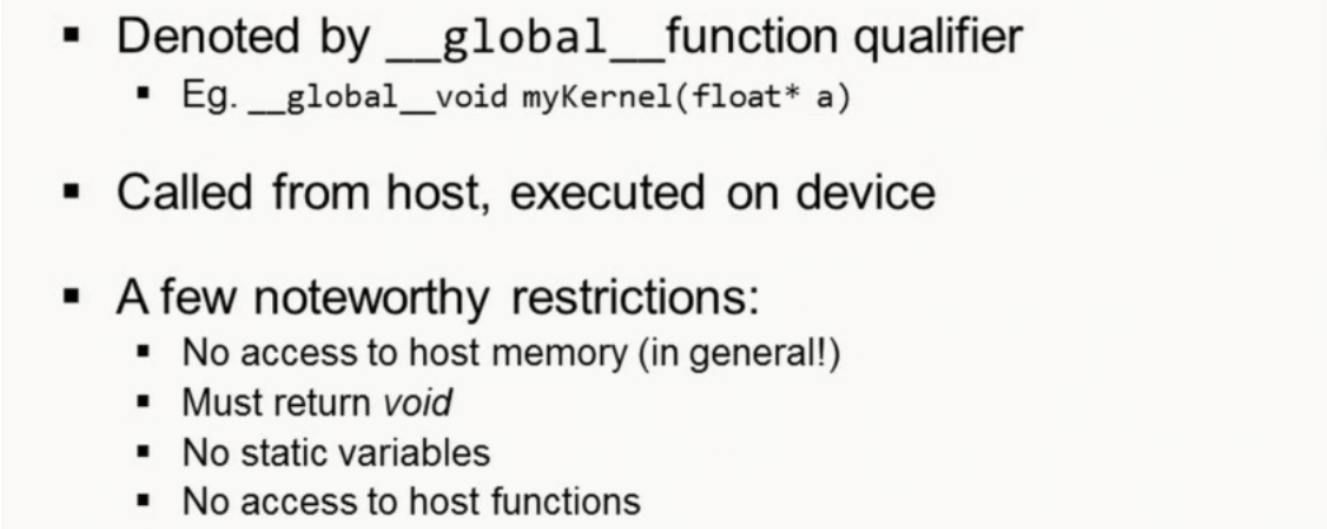
intro to cuda programming

- GPUs are suited for things where high thoroughput and high parallelism matters a lot. Latency may not matter a lot.
- Each GPU has a ton of cores (a few thousands). Each isn't as fast as a Intel processor in terms of clock frequency but parallelism is the key
- Also each core has an instruction set that isn't as full featured as an usual processor's instruction.
- GPUs have higher memory bandwidth than CPUs since they have to read in a lot of data from memory to the computing cores.
- Although GPUs consume a lot of power, the amount of computation measured in flops per watt is better on a GPU as well. A flop = $cores \times cycles/second \times floatingpointops/cycle$
- single precision is 32 bits and double precision float is 64 bits
- when this lecture was presented in 2015, the throughtput compared to CPU was 4-6x. (note how its not 1000s of times even though number of cores is more → clock cycle is slower but more importantly time spent moving data around etc)
- GPU disadvantages



- The interface between the memory and the GPU is PCIe which has bandwidth of 8-16 GB/s.

- host means the computer, device means the GPU.
- each GPU computation is divided into a set of tasks. each task works on some portion of the data and ideally the tasks are all executing the same code just on different parts of the data.
- GPU hardware has a lot of transistors for simplified compute logic as opposed to transistors for caches, branch prediction and conditionals etc.
- on a mobile processor, GPU and CPU talk over DRAM. there is no PCIe bus unlike on a desktop/server.
- data parallel functions executed on the GPU are called kernels. They are C/C++ functions with some restrictions and a few language extensions.
- Each kernel is executed by many threads (thousands) simultaneously.
- threads are grouped into thread blocks which are in turn grouped into grids.
- The CUDA C Runtime API is a lot more high level than the driver API or OpenCL. It takes care of a lot of setup and stuff like that.
- CUDA C and Driver API are now equivalent in terms of functionality.
- cuFunctionName() → driver API
  cudaFunctionName() → Runtime API



- `gridDim`, `blockDim`, `blockIdx` and `threadIdx` are read only variables available in the kernel (the kernel is the function running on the GPU)
- if, for and while loops are supported. common math operators and C math library is supported as well. a subset of C++ is supported as well.
- be careful while using things that interact with host CPU. on the GPU you basically don't have access to the host memory except whatever was transferred from the host memory before the kernel was started.
- functions declared with `__device__` can be used as helper functions that are called by `__global__` functions, but can't be called directly to run on the GPU by the host. if you'd like a function to also be compiled for the host in addition to GPU, add a `__host__`
- there are cuda version of memory management functions like cudaMalloc, cudaMemset and cudaFree
- make sure to differentiate between pointers that point to memory on host vs those that point to memory on device. kernels should only have access to on device memory and host functions shouldn't try to modify on device memory
- data transfer between host and GPU

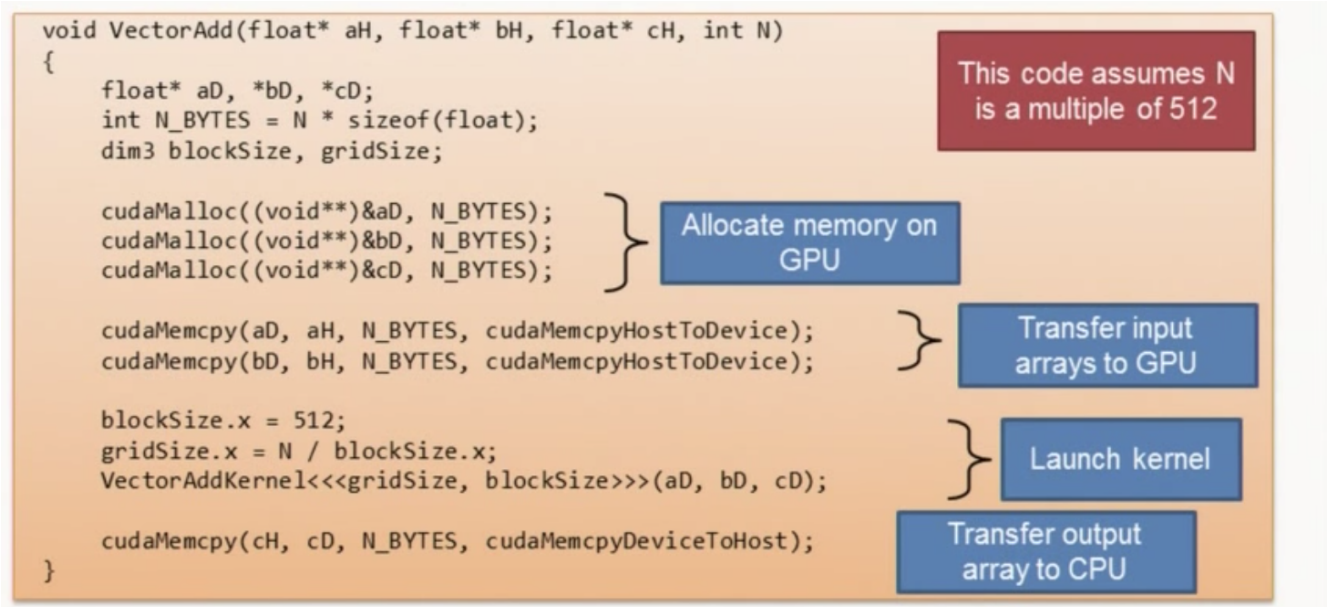
- Host code manages data transfers to and from the device:
  - `cudaMemcpy(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind direction);`
  - Direction is one of:
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost`
    - `cudaMemcpyDeviceToDevice`
    - `cudaMemcpyHostToHost`
    - `cudaMemcpyDefault`
  - Blocking call - returns once copy is complete
  - Waits for all outstanding CUDA calls to complete before starting transfer
  - With `cudaMemcpyDefault`, runtime determines which way to copy data

- how to know when a kernel function is done running → memcpy between device to host and host to device are synchronous but kernel launches are asynchronous

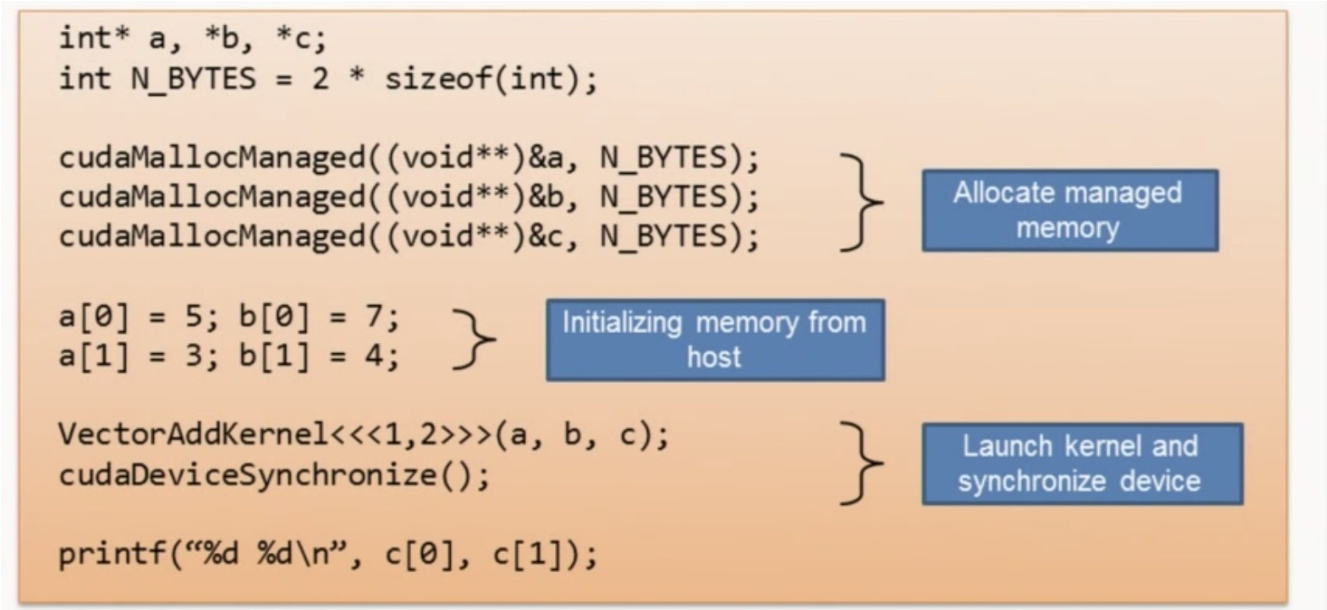
- Kernel launches are asynchronous
  - Control returns to CPU immediately
  - Kernel starts executing once all outstanding CUDA calls are complete
- cudaMemcpy() is synchronous
  - Blocks until copy is complete
  - Copy starts once all outstanding CUDA calls are complete
- cudaDeviceSynchronize()
  - Blocks until all outstanding CUDA calls are complete

```
cudaMemcpy(…,cudaMemcpyHostToDevice);

// Data is on the GPU at this point

MyKernel<<<…>>>(…);

// Kernel is launched but
// not necessarily complete

cudaMemcpy(…,cudaMemcpyDeviceToHost);
// CPU waits until kernel is complete
// and then transfers data

// Data is on the CPU at this point
```

- you can use cudaGetLastError() on host to get the errors from the GPU (or failure to launch on GPU device)
- sample vector addition



- `cudaMallocManaged` allocates memory on both host and device. when a kernel is called, we don't need to explicitly call cudaMemcpy to transfer from host to gpu before running kernel and from device to host after kernel completes. Example below. We just need to use cudaDeviceSynchronize() to make sure that kernel is done running before accessing results in the managed memory. Managed memory still needs to be freed!



- managed memory has worse performance than unmanaged → cuz it has to sync under the hood.