Introduction to CUDA C

Dmitri Nesteruk
@dnesteruk

pluralsight
hardcore developer training

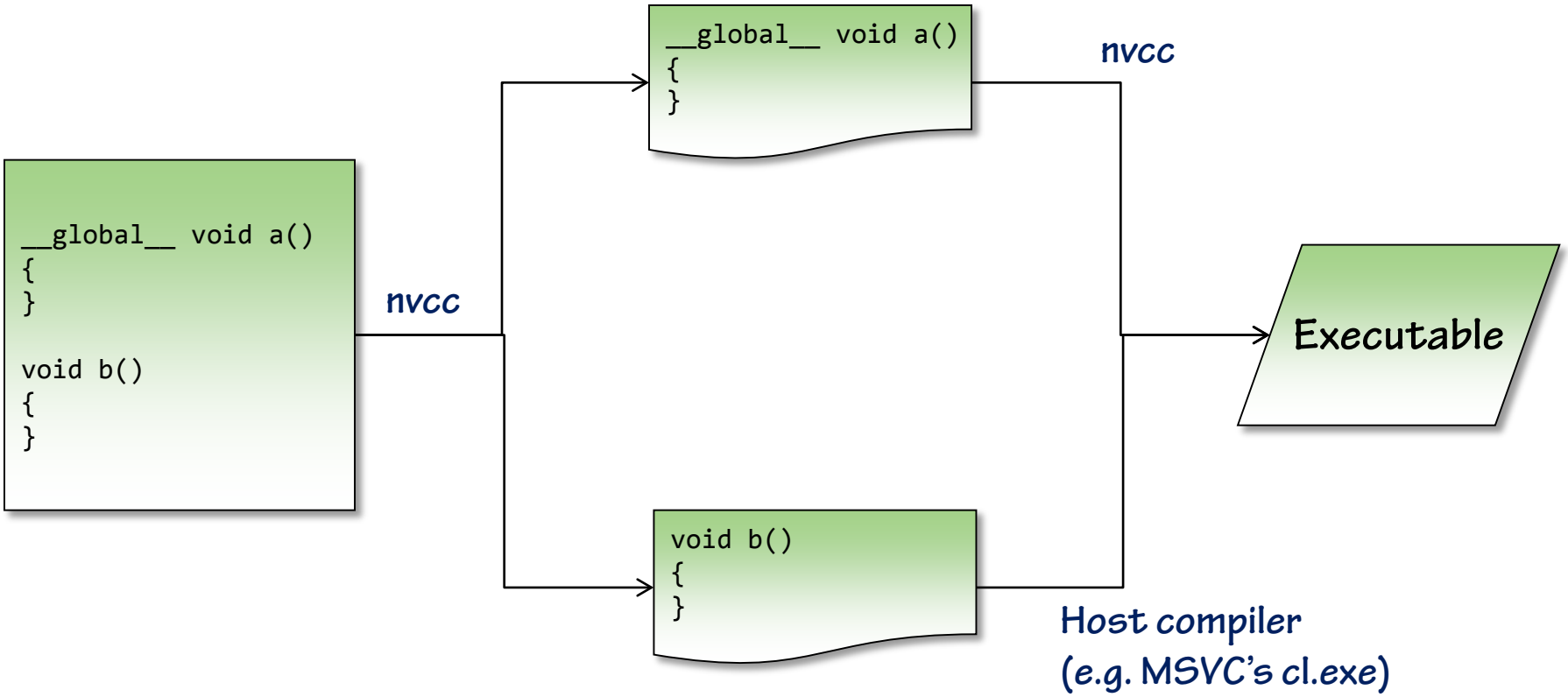
Overview

- Compilation Process
- Obligatory “Hello Cuda” demo
- Location Qualifiers
- Execution Model
- Grid and Block Dimensions
- Error Handling
- Device Introspection

NVidia Cuda Compiler (nvcc)

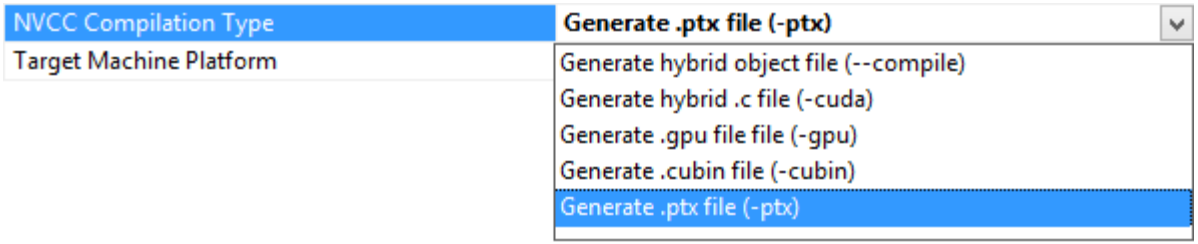
- **nvcc is used to compile CUDA-specific code**
  - Not a compiler!
  - Uses host C or C++ compiler (MSVC, GCC)
    - Some aspects are C++ specific
- **Splits code into GPU and non-GPU parts**
  - Host code passed to native compiler
- **Accepts project-defined GPU-specific settings**
  - E.g., compute capability
- **Translates code written in CUDA C into PTX**
  - Graphics driver turns PTX into binary code

NVCC Compilation Process



Parallel Thread Execution (PTX)

- **PTX is the 'assembly language' of CUDA**
  - Similar to .NET IL or Java bytecode
  - Low-level GPU instructions
- **Can be generated from a project**

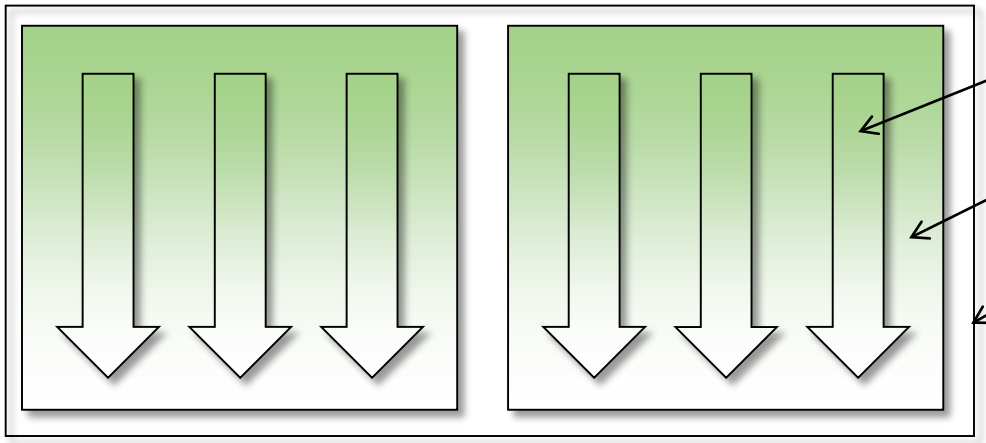


- **Typically useful to compiler writers**
  - E.g., GPU Ocelot https://code.google.com/p/gpuocelot/
- **Inline PTX (`asm`)**

Location Qualifiers

- **`__global__`**
  Defines a kernel.
  Runs on the GPU, called from the CPU.
  Executed with `<<<dim3>>>` arguments.
- **`__device__`**
  Runs on the GPU, called from the GPU.
  - Can be used for variables too
- **`__host__`**
  Runs on the CPU, called from the CPU.
- **Qualifiers can be mixed**
  - E.g. `__host__ __device__ foo()`
  - Code compiled for both CPU and GPU
  - Useful for testing

Execution Model

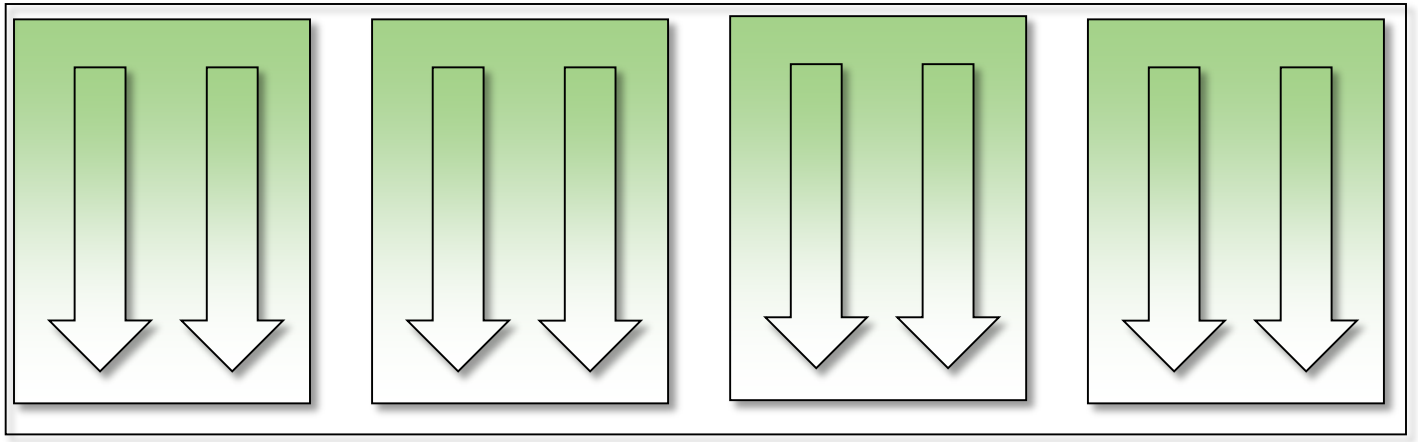


thread

thread block

grid

doSomething<<<2,3>>>()



doSomethingElse<<<4,2>>>()

Execution Model

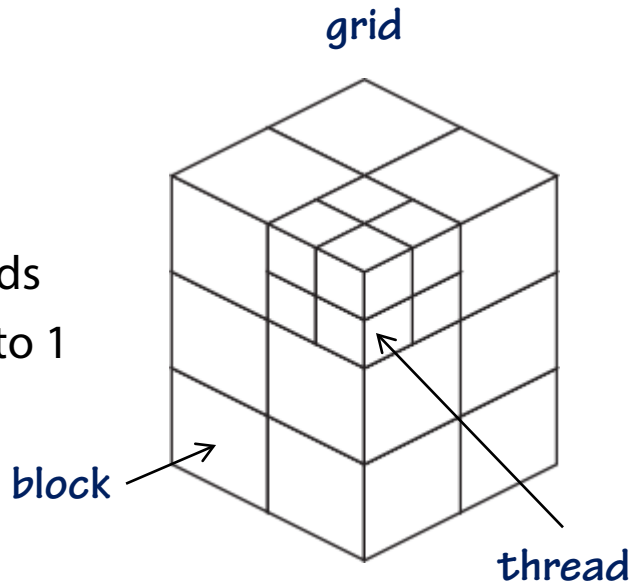
- **Thread blocks are scheduled to run on available SMs**
- **Each SM executes one block at a time**
- **Thread block is divided into warps**
  - Number of threads per warp depends on compute capability

WARP_SIZE	32

- **All warps are handled in parallel**
- **CUDA Warp Watch**

Dimensions

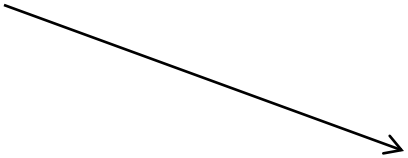
- **We defined execution as**
  - `<<<a,b>>>`
  - A grid of *a* blocks of *b* threads each
  - The grid and each block are 1D structures
- **In reality, these constructs are 3D**
  - A 3-dimensional grid of 3-dimensional blocks
  - You can define (a×b×c) blocks of (x×y×z) threads
  - Can have 2D or 1D by setting extra dimensions to 1
- **Defined in `dim3` structure**
  - Simple container with x, y and z values.
  - Some constructors defined for C++
  - Automatic conversion for
    `<<<a,b>>>` → `(a,1,1)` by `(b,1,1)`



Thread Variables

- **Execution parameters & current position**
- **blockIdx**
  - Where we are in the grid
- **gridDim**
  - The size of the grid
- **threadIdx**
  - Position of current thread in thread block
- **blockDim**
  - Size of thread block
- **Limitations**
  - Grid & block sizes
  - # of threads

MAX_BLOCK_DIM_X	512
MAX_BLOCK_DIM_Y	512
MAX_BLOCK_DIM_Z	64
MAX_GRID_DIM_X	65535
MAX_GRID_DIM_Y	65535
MAX_GRID_DIM_Z	1



MAX_THREADS_PER_BLOCK	512
MAX_THREADS_PER_MULTIPROCESSOR	1024

Error Handling

- **CUDA does not throw**
  - Silent failure
- **Core functions return `cudaError_t`**
  - Can check against `cudaSuccess`
  - Get description with `cudaGetErrorString()`
- **Libraries may have different error types**
  - E.g. cuRAND has `curandStatus_t`