CUDA Introduction

Martin Kruliš

History

1996 – 3Dfx Voodoo 1
- First graphical (3D) accelerator for desktop PCs

1999 – NVIDIA GeForce 256
- First Transform&Lightning unit

2000 – NVIDIA GeForce2, ATI Radeon

2001 – GPU has programmable parts
- DirectX – vertex and fragment shaders (v1.0)

2006 – OpenGL, DirectX 10, Windows Vista
- Unified shader architecture in HW
- Geometry shader added

History

2007 – NVIDIA CUDA
- First GPGPU solution, restricted to NVIDIA GPUs

2007 – AMD Stream SDK (previously CTM)

2009 – OpenCL, Direct Compute

2012 – NVIDIA Kepler Architecture

2014 – NVIDIA Maxwell Architecture

2016 – NVIDIA Pascal, Vulkan API

2018 – NVIDIA Volta

GPU in comparison with CPU

- CPU

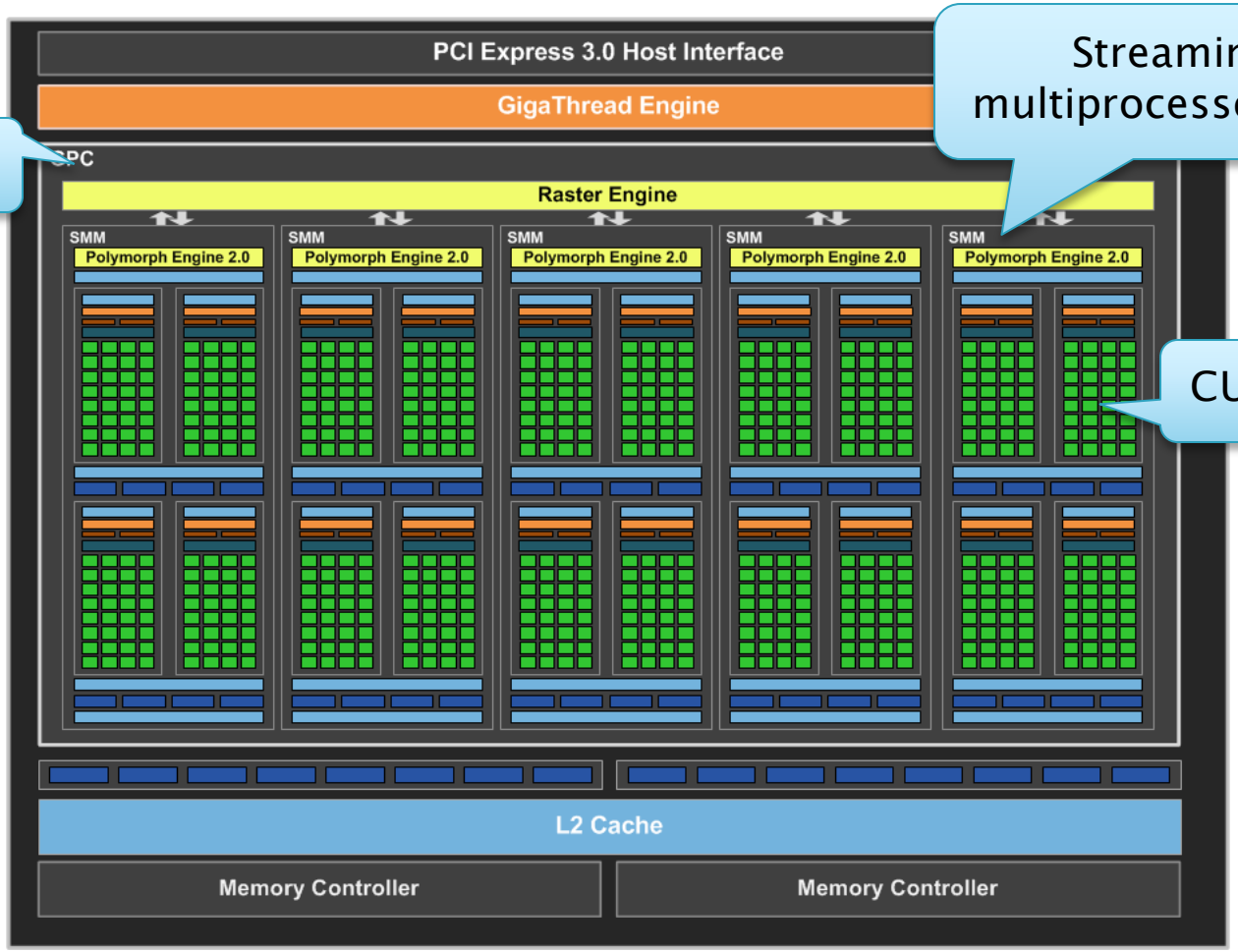
  - Few cores per chip
  - General purpose cores
  - Processing different threads
  - Huge caches to reduce memory latency
    - Locality of reference problem

- GPU

  - Many cores per chip
  - Cores specialized for numeric computations
  - SIMT thread processing
  - Huge amount of threads and fast context switch
    - Results in more complex memory transfers

Architecture Convergence

GPU Architecture



Maxwell GPU

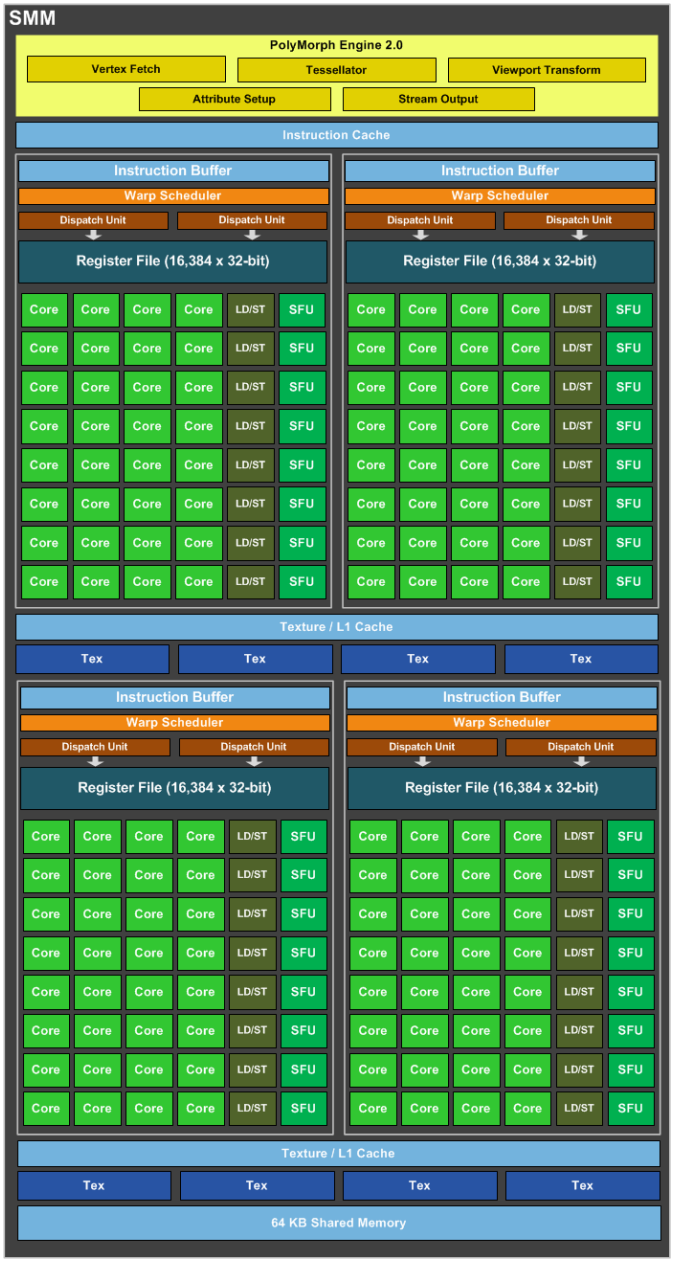
Streaming multiprocessor (SM)

CUDA core

GPU Architecture

- Maxwell Architecture
  - 4 identical parts
    - 32 cores
    - 64 kB shared memory
    - 2 instruction schedulers
  - CC 5.0
  - SMM

    (Streaming Multiprocessor – Maxwell)



GPU Arch.

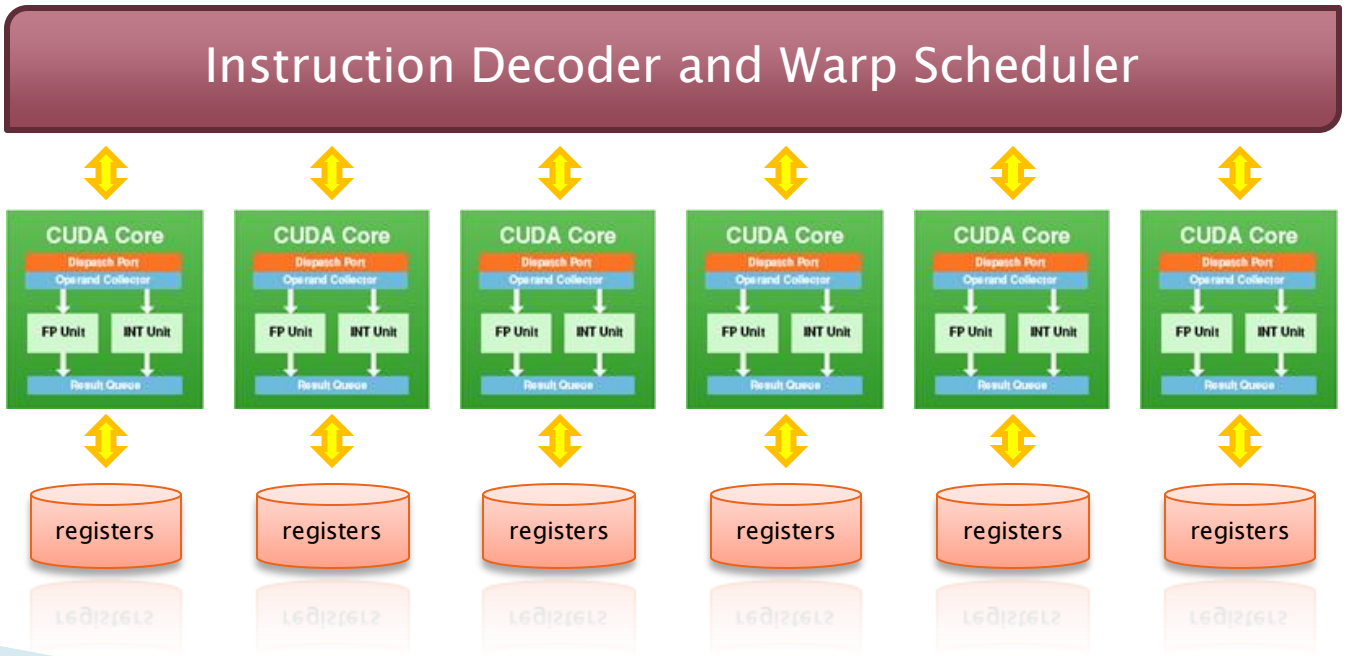
- Volta SM
  7.x CC
  8x tensor core
  (64 FMA/clock each)

GPU Execution Model

- Data Parallelism
  - Many data elements are processed concurrently by the same routine
  - GPUs are designed under this particular paradigm
    - Also have limited ways to express task parallelism
- Threading Execution Model
  - One function (kernel) is executed in many threads
    - Much more lightweight than the CPU threads
  - Threads are grouped into blocks (work groups) of the same size

SIMT Execution

- Single Instruction Multiple Threads
  - All cores are executing the same instruction
  - Each core has its own set of registers

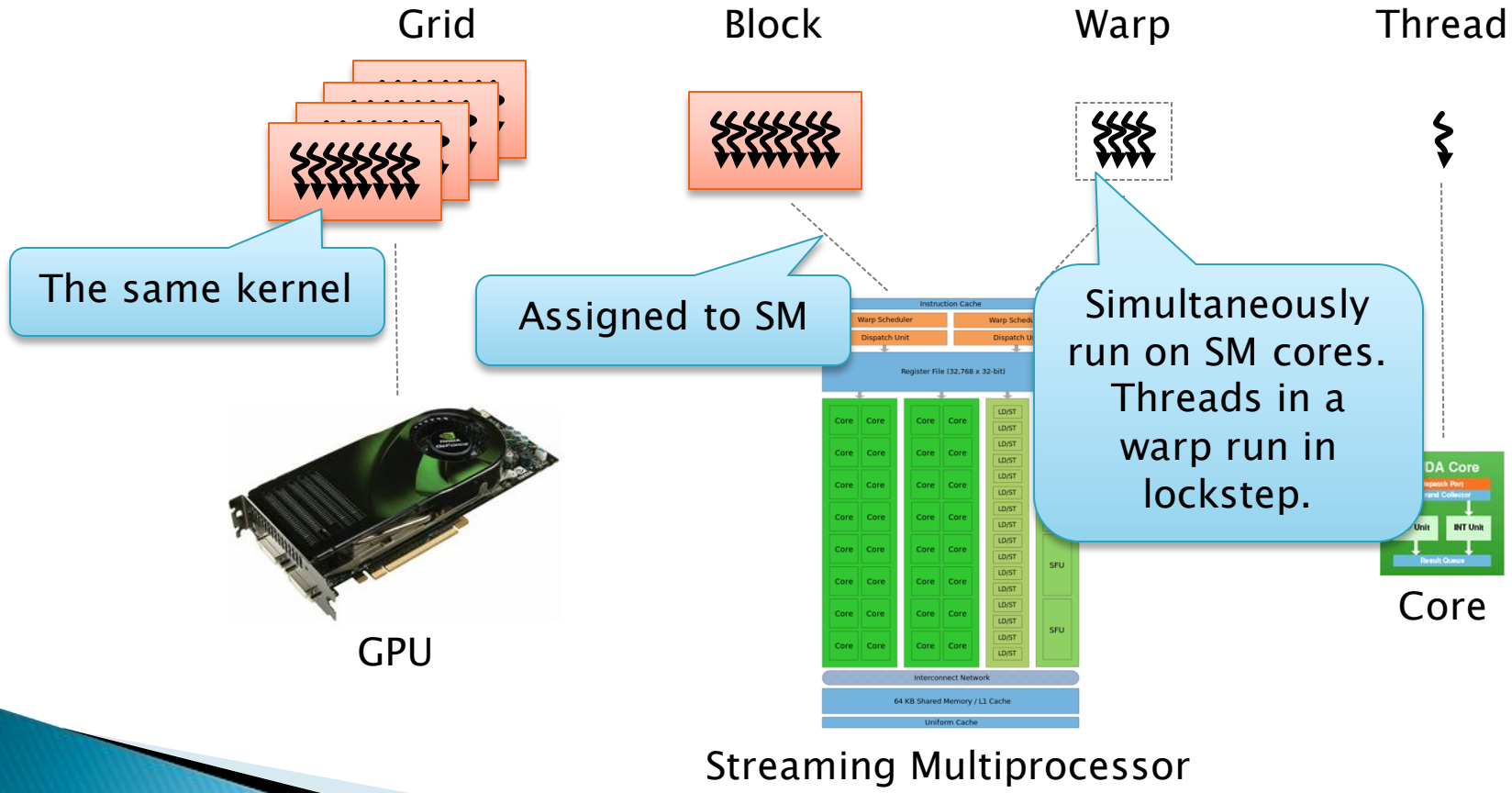


SIMT vs. SIMD

- Single Instruction Multiple Threads
  - Width-independent programming model
  - Serial-like code
  - Achieved by hardware with a little help from compiler
  - Allows code divergence
- Single Instruction Multiple Data
  - Explicitly expose the width of SIMD vector
  - Special instructions
  - Generated by compiler or directly written by programmer
  - Code divergence is usually not supported or tedious

Thread-Core Mapping

- How are threads assigned to SMPs



CUDA

- Compute Unified Device Architecture
  - NVIDIA parallel computing platform
  - Implemented solely for GPUs
  - First API released in 2007
  - Used in various libraries
    - cuFFT, cuBLAS, …
  - Many additional features
    - OpenGL/DirectX Interoperability, computing clusters support, integrated compiler, …
- Alternatives
  - Vulkan, OpenCL, AMD Stream SDK, C++ AMP

Device Detection

- Device Detection

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
...
cudaSetDevice(deviceIdx);
```

Device index is from range 0, `deviceCount-1`

- Querying Device Information

```
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp,
    deviceIdx);
```

Device Features

- Compute Capability
  - Prescribed set of technologies and constants that a GPU device must implement
    - Incrementally defined
  - Architecture dependent
  - CC for known architectures:
    - 1.0, 1.3 – Tesla, 2.0, 2.1 – Fermi
    - 3.x – Kepler (Tesla K20m – CC 3.5)
    - 5.x – Maxwell (GTX 980 – CC 5.2)
    - 6.x – Pascal
    - 7.x – Volta (most recent)

Kernel Execution

- Kernel
  - Special function declarations

```
__device__ void foo(…) { … }
__global__ void bar(…) { … }
```

  - Kernel Execution

```
bar<<<Dg, Db [, Ns [, S]]>>>(args);
```

    - **Dg** – dimensions and sizes of blocks spawned
    - **Db** – dimensions and sizes of threads per block
    - **Ns** – dynamically allocated shared memory per block
    - **S** – stream index

Kernel Execution

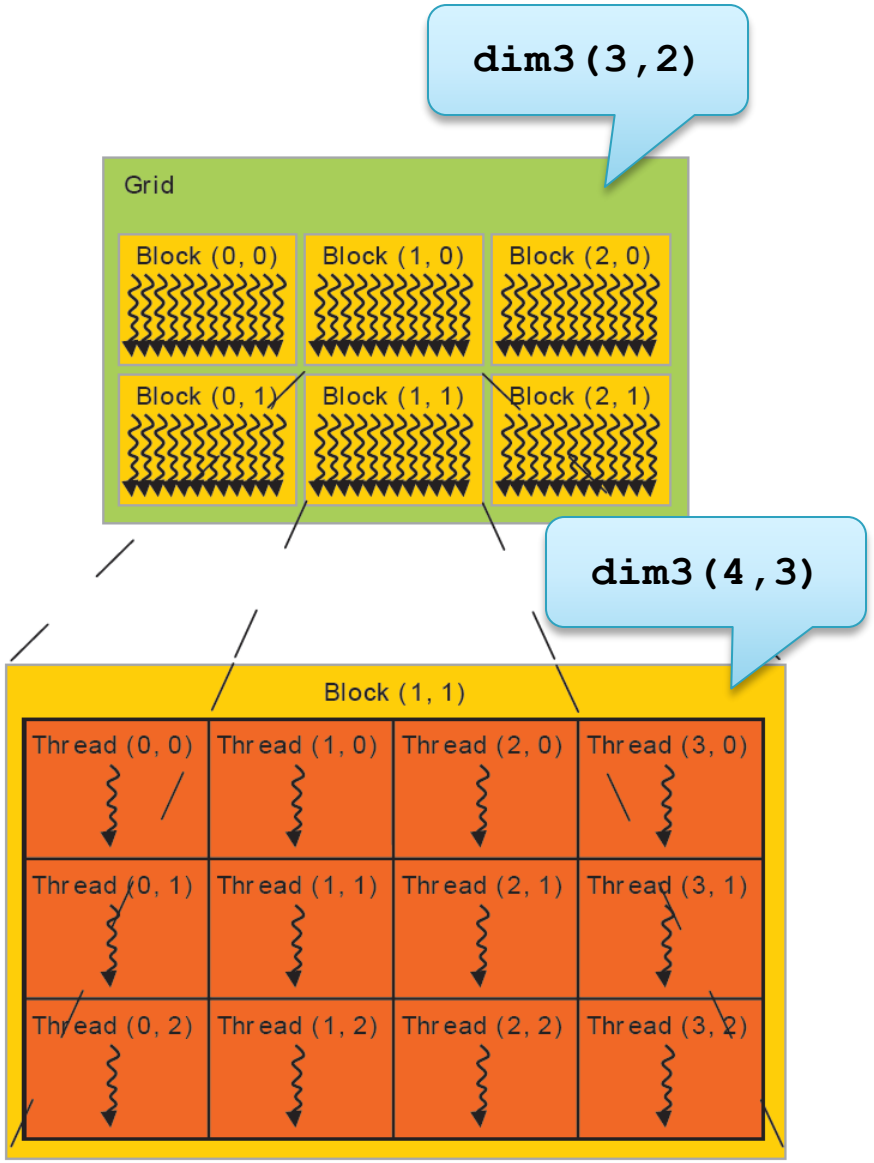
- Spawning Properties

```
__global__ void vecAdd(float *x) { … }
vecAdd<<<42, 64>>>(x);
```

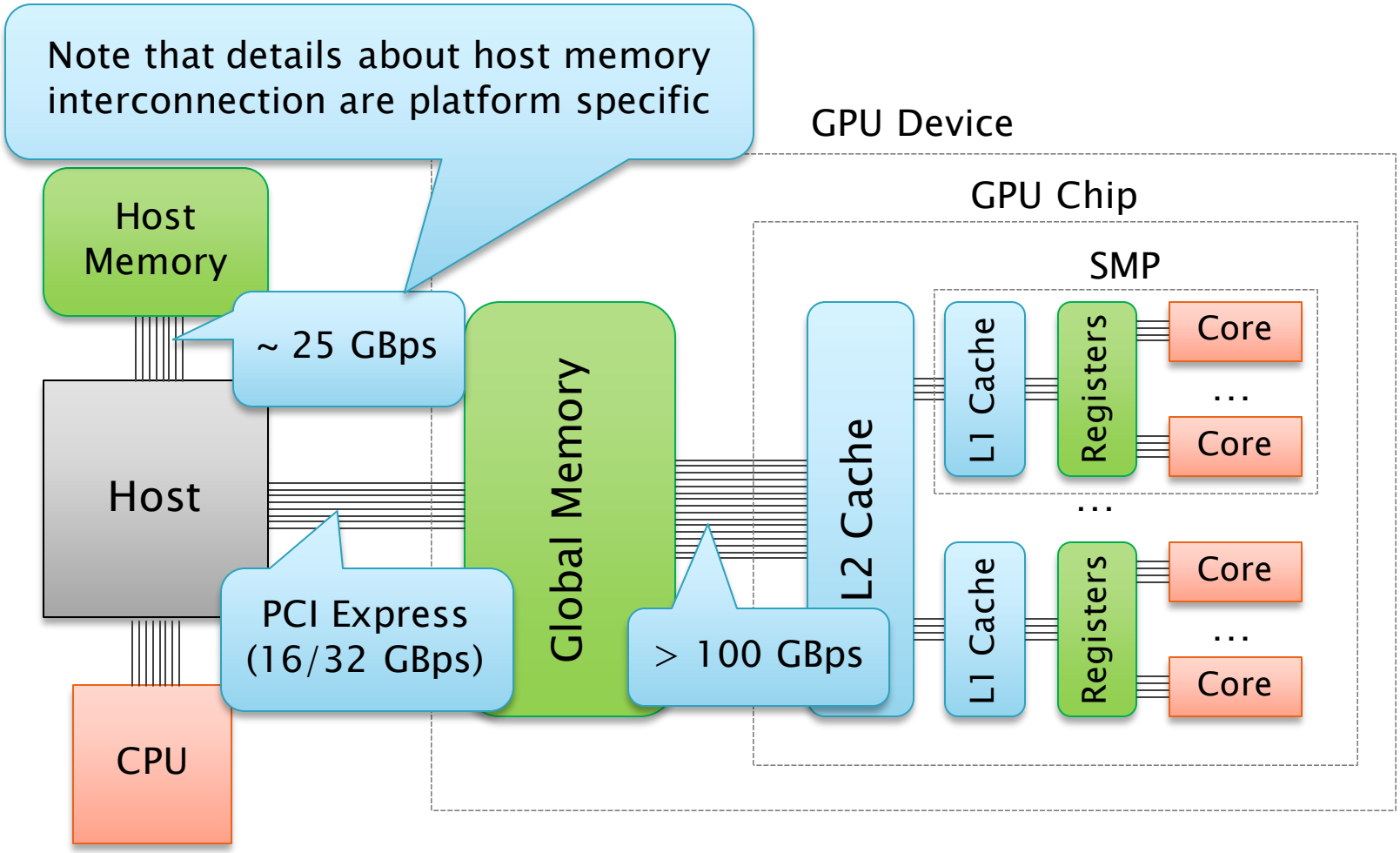
  - Spawns 42 blocks, 64 threads in each block
    - Not all of them has to run simultaneously
  - Number of blocks should be greater than # of SMPs
  - Number of threads should be multiple of warp size (32 on all current architectures), at least 64
  - Instead of numbers, **`dim3`** structures may be used
    - Specifying size of grid and blocks in 3 dimensions

Kernel Execution

- Grid
  - Consists of blocks
  - Up to 3 dimensions
- Each Block
  - Consist of threads
  - Same dimensionality
- Kernel Constants
  - **`gridDim`**, **`blockDim`**
  - **`blockIdx`**, **`threadIdx`**
  - **`.x`**, **`.y`**, **`.z`**



GPU Memory


Note that details about host memory interconnection are platform specific
GPU Device
GPU Chip
SMP
Host Memory
~ 25 GBps
Host
PCI Express (16/32 GBps)
CPU
Global Memory
> 100 GBps
L2 Cache
L1 Cache
Registers
Core
...
L1 Cache
Registers
Core
...
Core

Memory Allocation

- Device (Global) Memory Allocation
  - C-like allocation system
    - The programmer must distinguish host/GPU pointers!

```
float *vec;
cudaMalloc((void**)&vec, count*sizeof(float));
cudaFree(vec);
```

- Host-Device Data Transfers
  - Explicit blocking functions

```
cudaMemcpy(vec, localVec, count*sizeof(float),
        cudaMemcpyHostToDevice);
```

Code Example

```
__global__ void vec_mul(float *X, float *Y, float *res) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    res[idx] = X[idx] * Y[idx];
}
...
float *X, *Y, *res, *cuX, *cuY, *cuRes;
...
cudaSetDevice(0);
cudaMalloc((void**)&cuX, N * sizeof(float));
cudaMalloc((void**)&cuY, N * sizeof(float));
cudaMalloc((void**)&cuRes, N * sizeof(float));
cudaMemcpy(cuX, X, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(cuY, Y, N * sizeof(float), cudaMemcpyHostToDevice);
vec_mul<<<(N/64), 64>>>(cuX, cuY, cuRes);
cudaMemcpy(res, cuRes, N * sizeof(float), cudaMemcpyDeviceToHost);
```

Few More Things…

- Synchronization
  - Memory transfers are synchronous
    - Explicit **`cudaMemcpyAsync()`** exists
  - Kernel execution is asynchronous
    - But synced with other executions/memory transfers
  - **`cudaDeviceSynchronize()`**
- Error Checking
  - Most functions return error code
    - Should be equal to **`cudaSuccess`**
  - **`cudaGetLastError()`**
    - E.g., after kernel execution

Compilation

- The **nvcc** Compiler
  - Used for compiling both host and device code
  - Defers compilation of the host code to **gcc** (linux) or Microsoft VCC (Windows)

```
$> nvcc -cudart static code.cu -o myapp
```

  - Can be used for compilation only

```
$> nvcc -compile … kernel.cu -o kernel.obj
$> cc -lcudart kernel.obj main.obj -o myapp
```

  - Device code is generated for target architecture

```
$> nvcc -arch sm_13 …
$> nvcc -arch compute_35 …
```

Compile for real GPU with compute capability 1.3 and to PTX with capability 3.5

NVIDIA Tools

- System Management Interface
  - **`nvidia-smi`** (CLI application)
  - NVML library (C-based API)
    - Query GPU details

      `$> nvidia-smi -q`
    - Set various properties (ECC, compute mode …), …

      `$> nvidia-persistenced --persistence-mode`
      - Set drivers to persistent mode (recommended)
- NVIDIA Visual Profiler
  - **`$> nvvp &`**
  - Use X11 SSH forwarding from ubergrafik/knight

Discussion