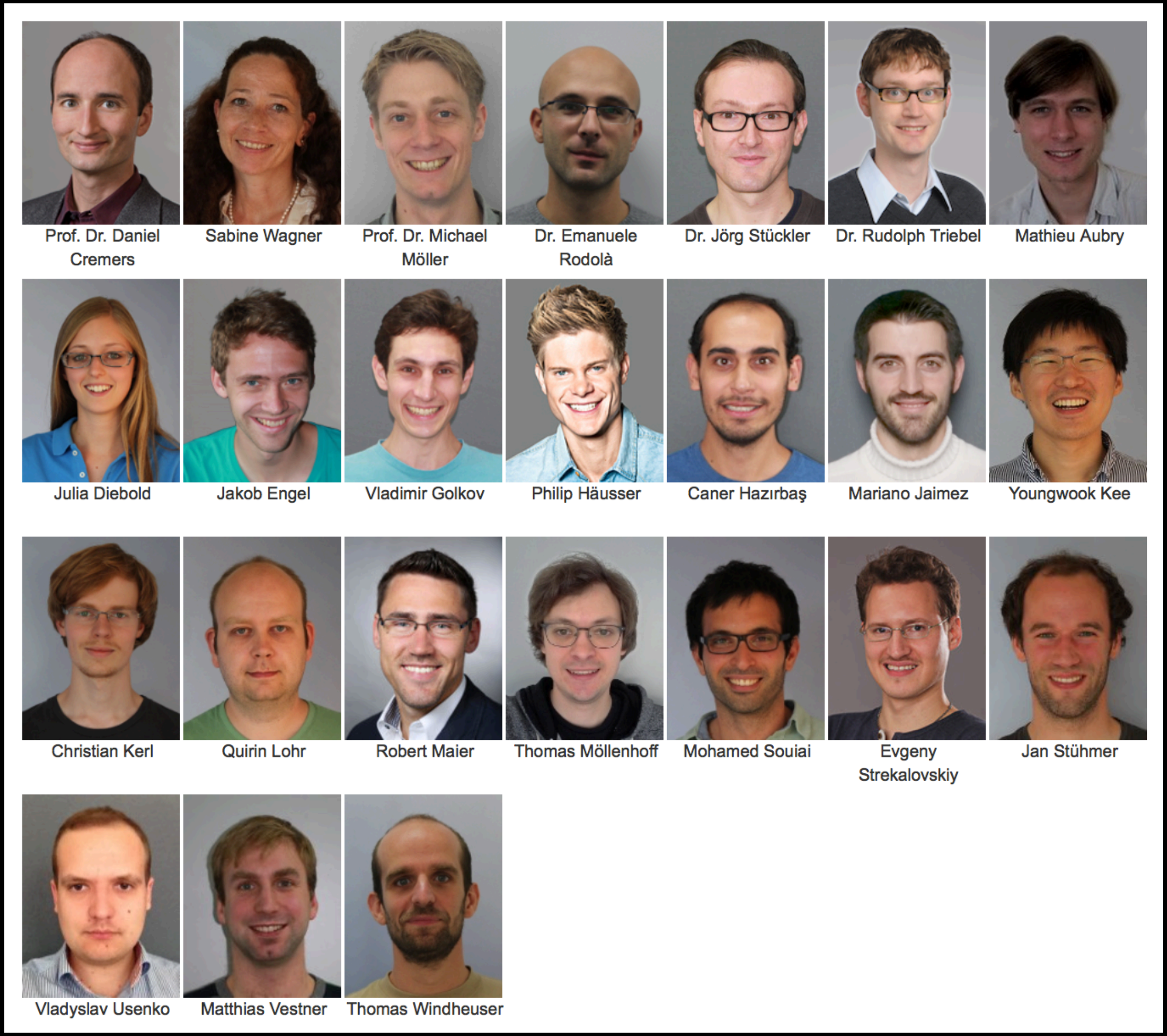
GPU Programming in Computer Vision

Thomas Möllenhoff, Robert Maier,
Mohamed Souiai, Caner Hazırbaş

Introduction to Parallel Computing

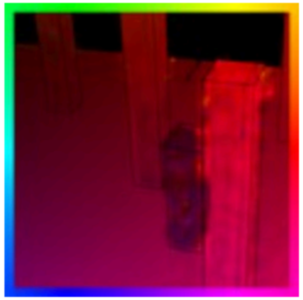
Technical University Munich, Computer Vision Group
Winter Semester 2014/2015, March 2 – April 3

Computer Vision Group



Our Research

Image-based 3D Reconstruction



Optical Flow Estimation

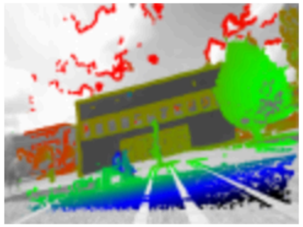
Shape Analysis

Quadrocopter

RGB-D Sensors (Kinect)

Image Segmentation

Convex Relaxation Methods



LSD-SLAM: Large-Scale Direct Monocular SLAM

This Course

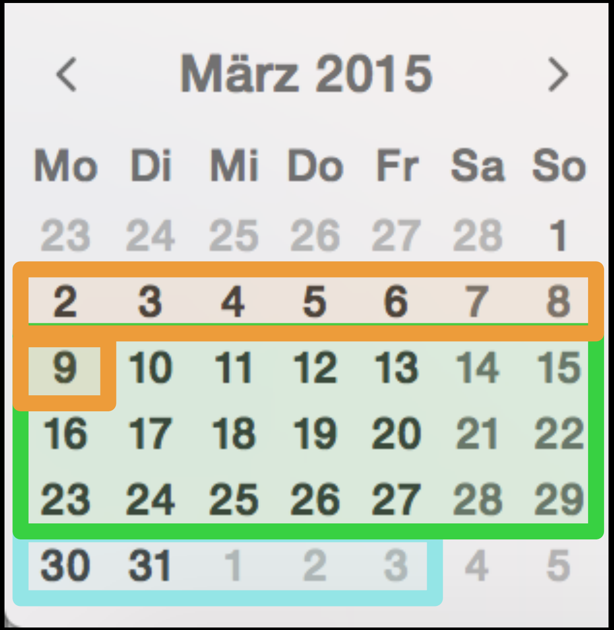
- Parallel Programming with CUDA
- Computer Vision Basics
  - image filtering (convolution, diffusion, denoising)
  - basic algorithms
  - regularization (dealing with noise)
- Optimization + Numerics

Course Goals

- Learn how to program massively parallel processors and achieve
  - high performance
  - functionality and maintainability
  - scalability across future generations
- Acquire technical knowledge required to achieve above goals
  - principles and patterns of parallel programming
  - processor architecture features and constraints
  - programming API, tools and techniques
- Apply this knowledge to implement computer vision algorithms efficiently

Course Timeline: 02.03 - 03.04

- March 2-9 (this week) : Lecture
  - 2-4h lectures (attendance mandatory)
  - programming exercises
  - groups of 2-3 students

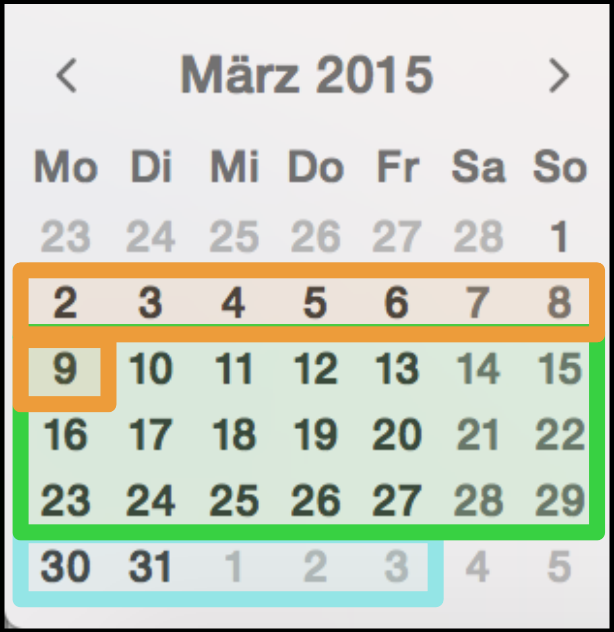


- March 10-29: Student project
  - advanced applications
  - unsupervised

- March 30-April 1: Presentations

Lecture Week

- Lecture
  - starts at 10:00 sharp each day
  - attendance mandatory to pass the course
- Exercises
  - until 18:00 each day
  - groups of 2-3 students
  - present solutions
    on next day after the lecture
    - exercises of March 2: March 4
    - exercises of March 6: March 9



“Work @ Home” ™

- You can access your computer remotely:

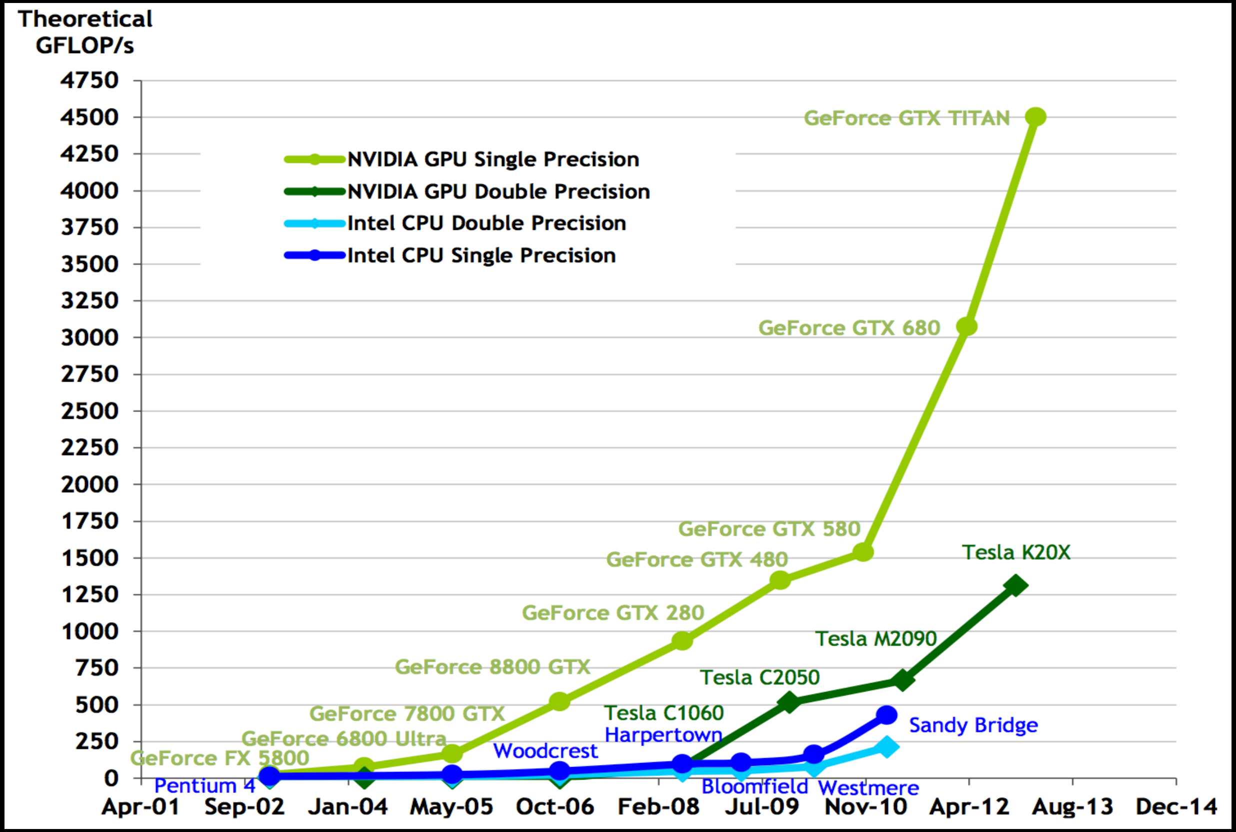
```
ssh -X p123@atradig789.informatik.tu-muenchen.de
```

- p123: replace with your login
- atradig789: replace with your computer name
  - type `hostname` to find out the name

- Works from within Linux or Mac
  - for Mac: install XQuartz first (X11 server)

Why Massively Parallel Processing?

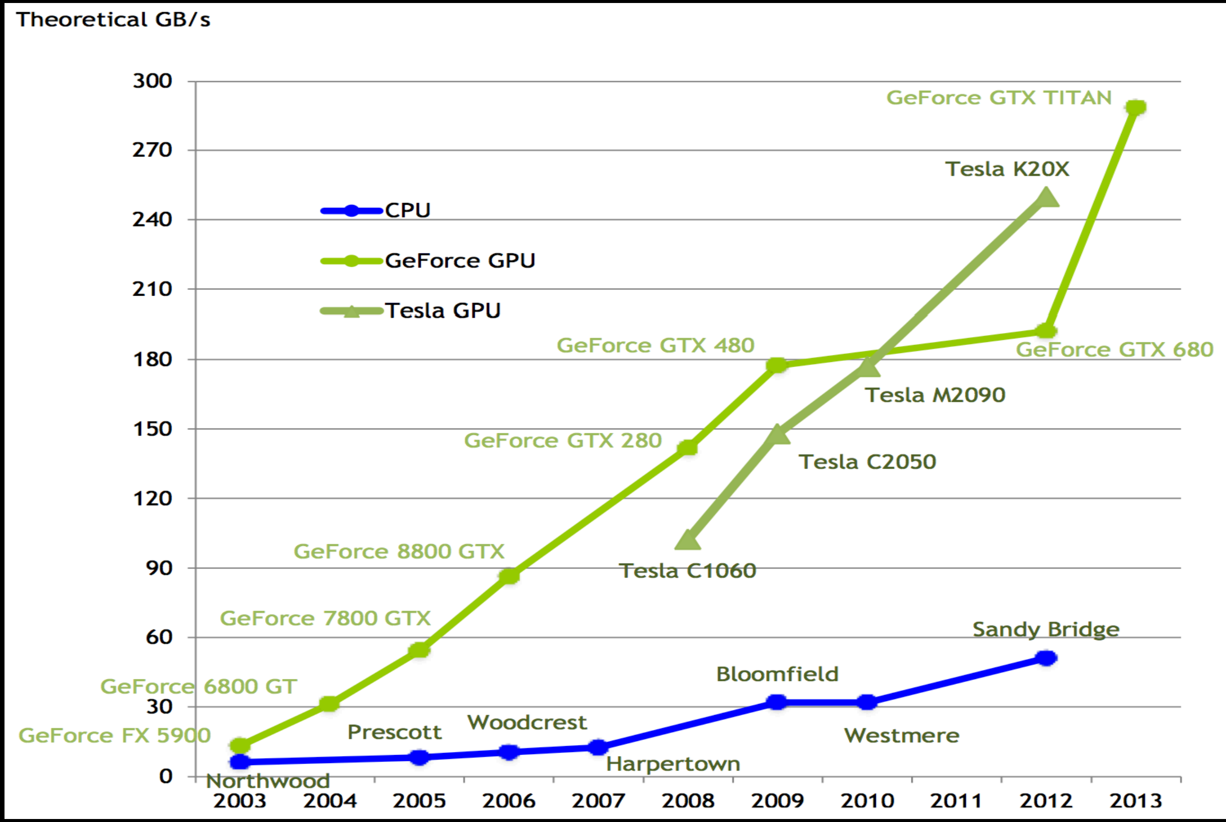
- A quiet revolution: Performance!
  - computations: TFLOPs vs. 100 GFLOPs



- GPU in every PC – massive volume & impact

Why Massively Parallel Processing?

- A quiet revolution: Performance!
  - bandwidth: ~5x



  - GPU in every PC – massive volume & impact

Serial Performance Scaling is Over

- Cannot continue to scale processor frequencies
  - no 10 GHz chips
- Cannot continue to increase power consumption
  - can’t melt chip
- Can continue to increase transistor density
  - as per Moore’s Law

How to Use Transistors?

- Larger caches … decreasing
- Instruction-level parallelism … decreasing
  - out-of-order execution, speculation, …
- Data-level parallelism … increasing
  - vector units, SIMD execution, …
  - Intel SSE, GPUs, …
- Thread-level parallelism … increasing
  - multithreading, multicore, manycore

Design Difference: CPU vs. GPU

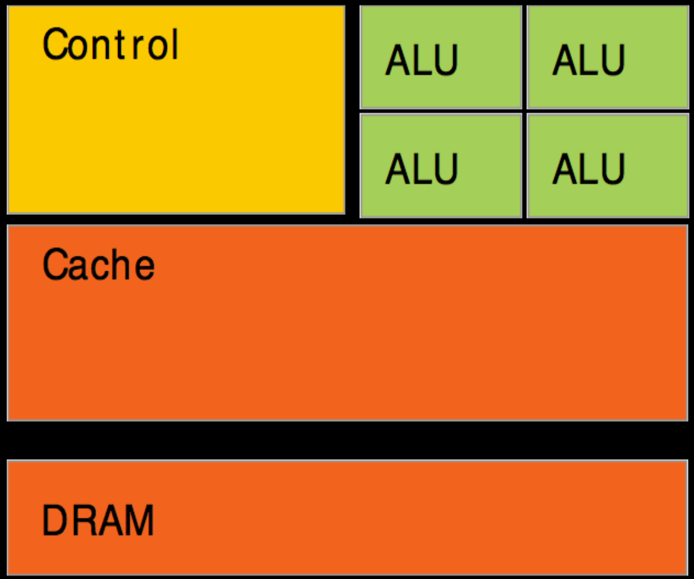
- Different goals produce different designs
  - CPU must be good at everything, parallel or not
  - GPU assumes work load is highly parallel

- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic

- GPU: maximize throughput of all threads
  - skip big caches, multithreading hides latency
  - share control logic across many threads, SIMD
  - create and run thousands of threads

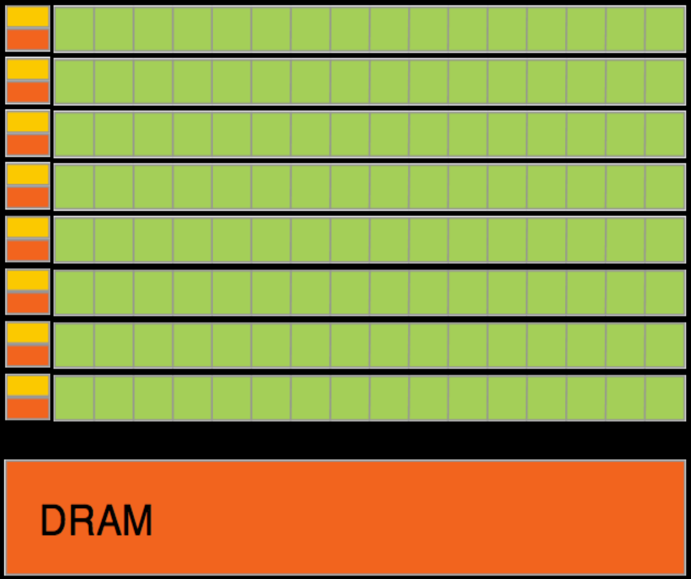
Design Difference: CPU vs. GPU

- **Different goals produce different designs**
  - **CPU must be good at everything, parallel or not**
  - **GPU assumes work load is highly parallel**



CPU

minimize latency



GPU

maximize throughput

Enter the GPU

- Massively parallel
- Affordable supercomputing

NVIDIA GPUs

- **Compute Capability**
  - **version number of the hardware architecture**
  - **core architecture and incremental improvements**

Arch	CC	GPUs	Features (e.g.)
Tesla (2007)	1.0	8800 GTX, Tesla C870	Basic functionality
	1.1	9800 GTX, Quadro FX 580	Atomics in global mem
	1.2	GT 240, Quadro FX 1800M	Atomics in shared mem
	1.3	GTX 285, Tesla C1060	Double precision
Fermi (2010)	2.0	GTX 480/580, Tesla C2070	Memory cache
	2.1	GTX 460, GTX 560 Ti	More cores (hardware)
Kepler (2012)	3.0	GTX 680/770, Tesla K10	Power efficiency, Many cores
	3.5	GTX 780/Titan, Tesla K20	Dynamic Parallelism, Hyper-Q
Maxwell (2014)	5.0	GTX 750, GTX 750 Ti	135% performance/core 200% performance/watt

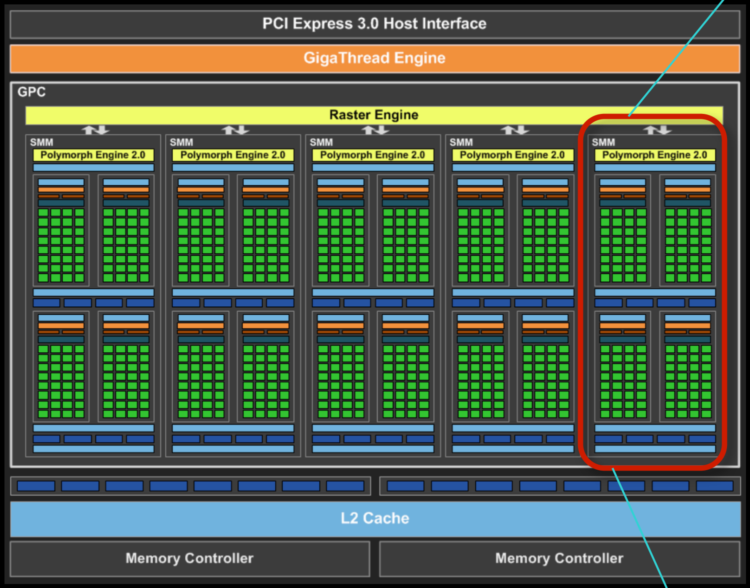
NVIDIA GPUs

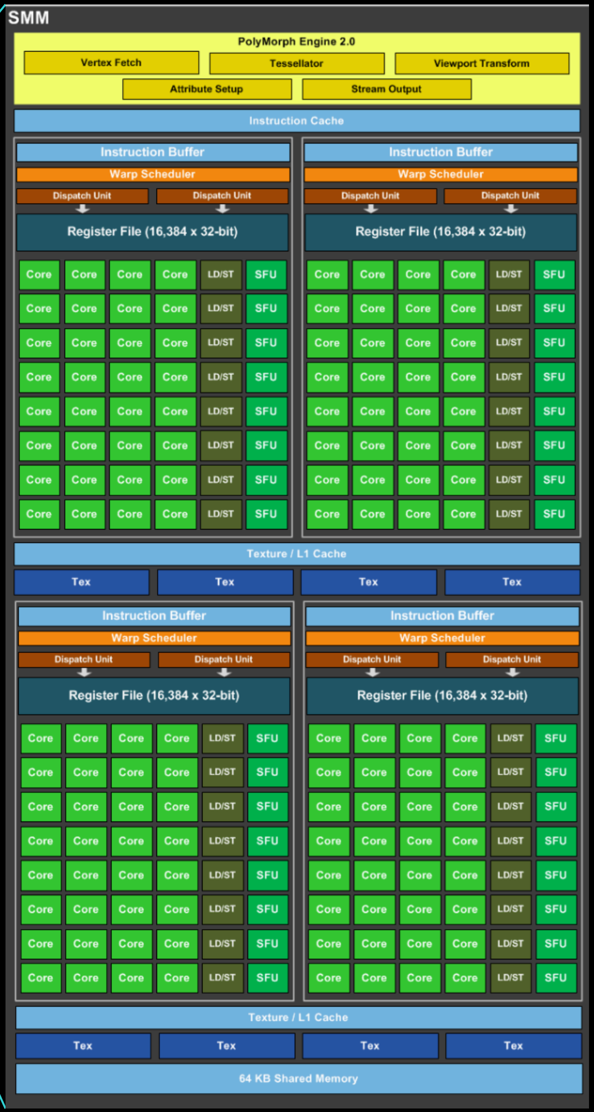
- Compute Capability
  - version number of the hardware architecture
  - core architecture and incremental improvements

- List of features for each Compute Capability:
  - see NVIDIA Programming Guide: Appendix G.1

NVIDIA GPUs: Current Architecture

1st gen Maxwell GPU





- **5 multiprocessors (up to)**
- **128 Cuda Cores per SM**
  - **640 Cores in total (up to)**

Enter CUDA
(“Compute Unified Device Architecture“)

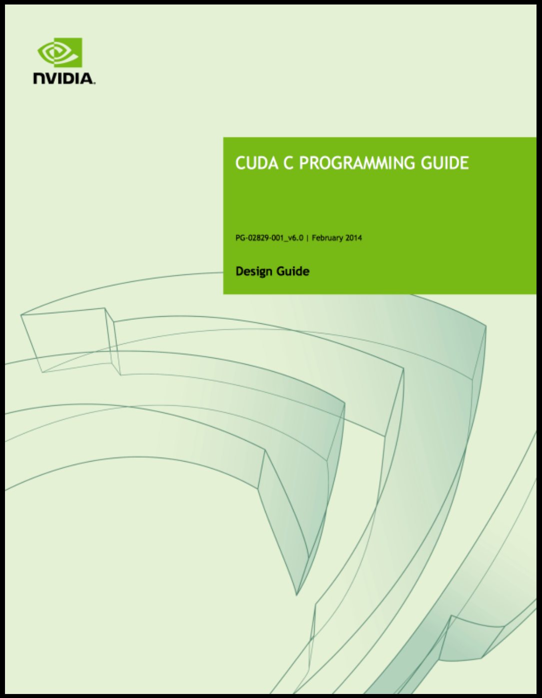
- Scalable parallel programming model
  - exposes the computational horsepower of GPUs
- Abstractions for parallel computing
  - let programmers focus on parallel algorithms
  - *not* mechanics of a parallel programming language
- Minimal extensions to familiar C/C++ environment to run code on the GPU
  - low learning curve

CUDA: Scalable Parallel Programming

- Provide straightforward mapping onto hardware
  - good fit to GPU architecture
  - maps well to multi-core CPUs too
- Execute code by many threads in parallel
- Scale to 100s of cores & 10,000s of threads
  - GPU threads are lightweight — create / switch is free
  - GPU needs 1000s of threads for full utilization

Reference: CUDA Programming Guide

- CUDA comes with excellent documentation
  - `doc/pdf` in the CUDA folder, have a look!
- CUDA Programming Guide
  - one of the best CUDA references
  - covers every CUDA feature
  - provides in-depth explanations
- Also: list of all CUDA functions:
  - `CUDA_Runtime_API.pdf`



Outline of CUDA Basics

- Kernels and Thread Hierarchy
- Execution on the GPU
- Memory Management
- Error Handling And Compiling

- See the Programming Guide for the full API

BASIC KERNELS AND THREAD HIERARCHY

CUDA Definitions

- **Device:** GPU
  - executes code in parallel
- **Host:** CPU
  - manages execution on the device
- **Kernel:** C/C++ function executed on the device
  - executed by many threads
  - each thread executes the same sequential program
  - each thread is free to execute a unique code path

Quick Example

- **CPU: Process subtasks serially one by one:**

```
for( int i=0; i<n; i++ )
{
    c[i] = a[i] + b[i];
}
```

- **GPU: Process each subtask in its own thread:**

```
__global__ void vecAdd (float *a, float *b, float *c)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    c[i] = a[i] + b[i];
}
```

Each thread knows its index

  - **launch enough threads to cover all data**

Thread Hierarchy

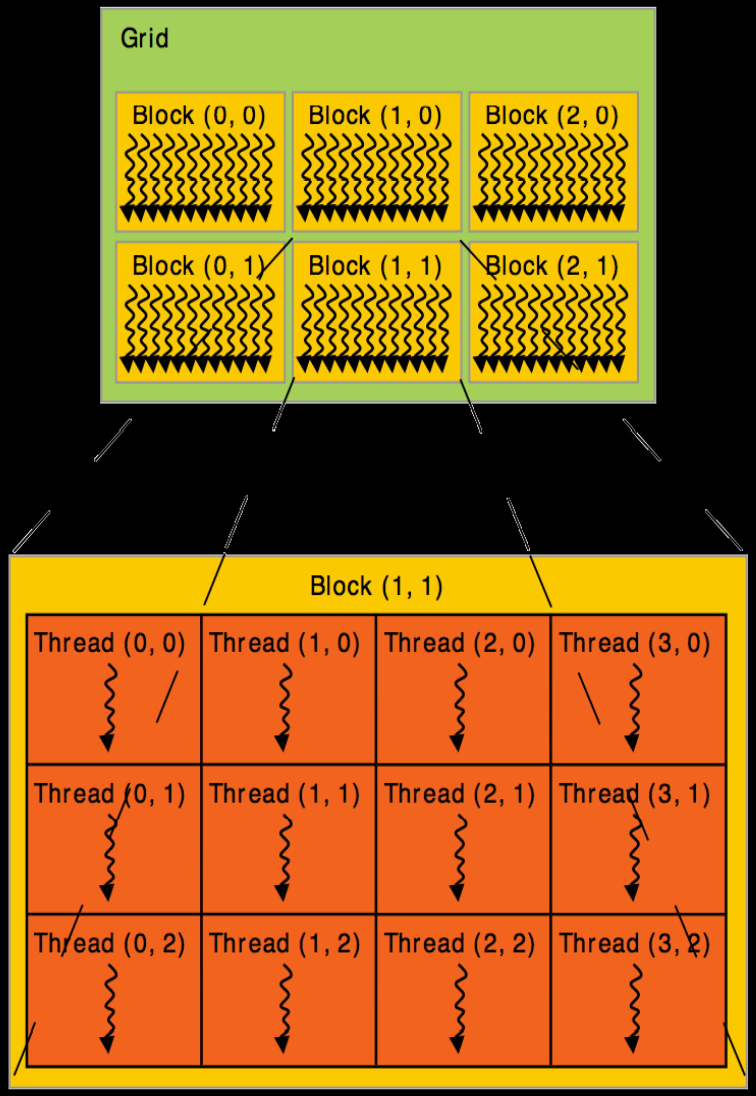
- Kernel threads are grouped into blocks
  - up to 512 or 1024 threads per block
- Idea: Threads from the same block can cooperate
  - synchronize their execution
  - communicate via shared memory
  - threads from different blocks cannot cooperate
- Allows transparent scaling to different GPUs
- All kernel blocks together form a grid

Thread Hierarchy

- # threads per block:
  up to 512 (CC 1.x),
  up to 1024 (CC>=2.0)

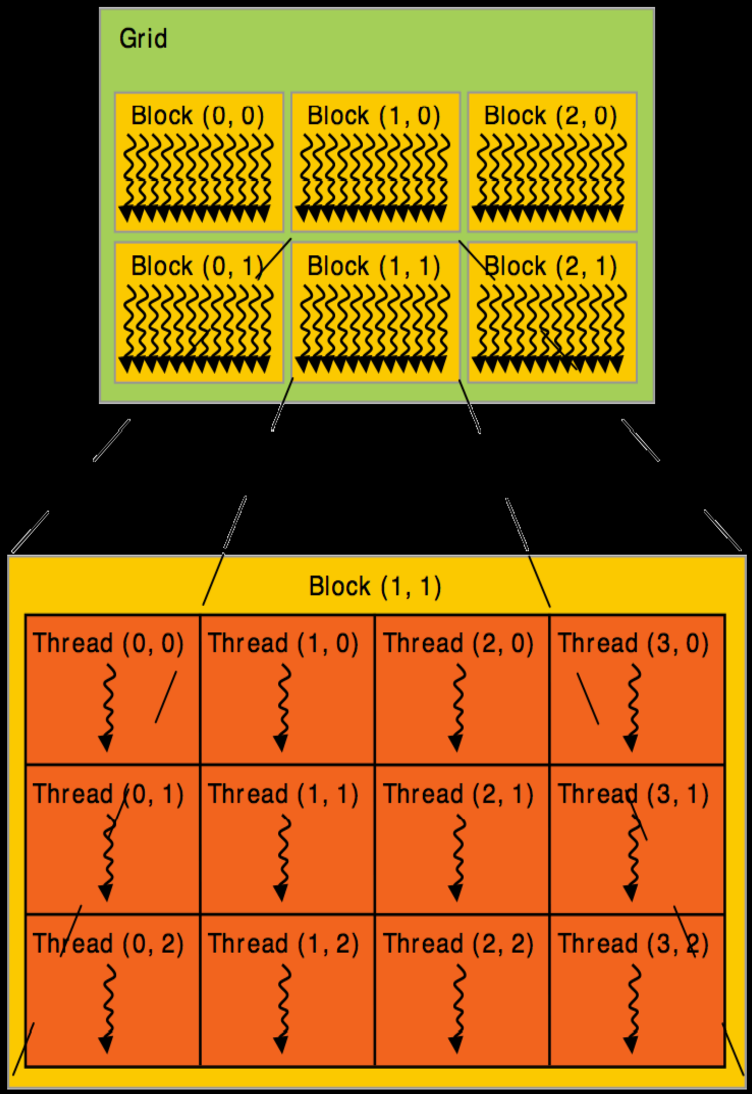
- Blocks can be 1D, 2D, or 3D
- Grids can be 1D, 2D, or 3D
  - CC 1.x: only 1D or 2D

- Dimensions set **at launch**
  - can be different for each grid



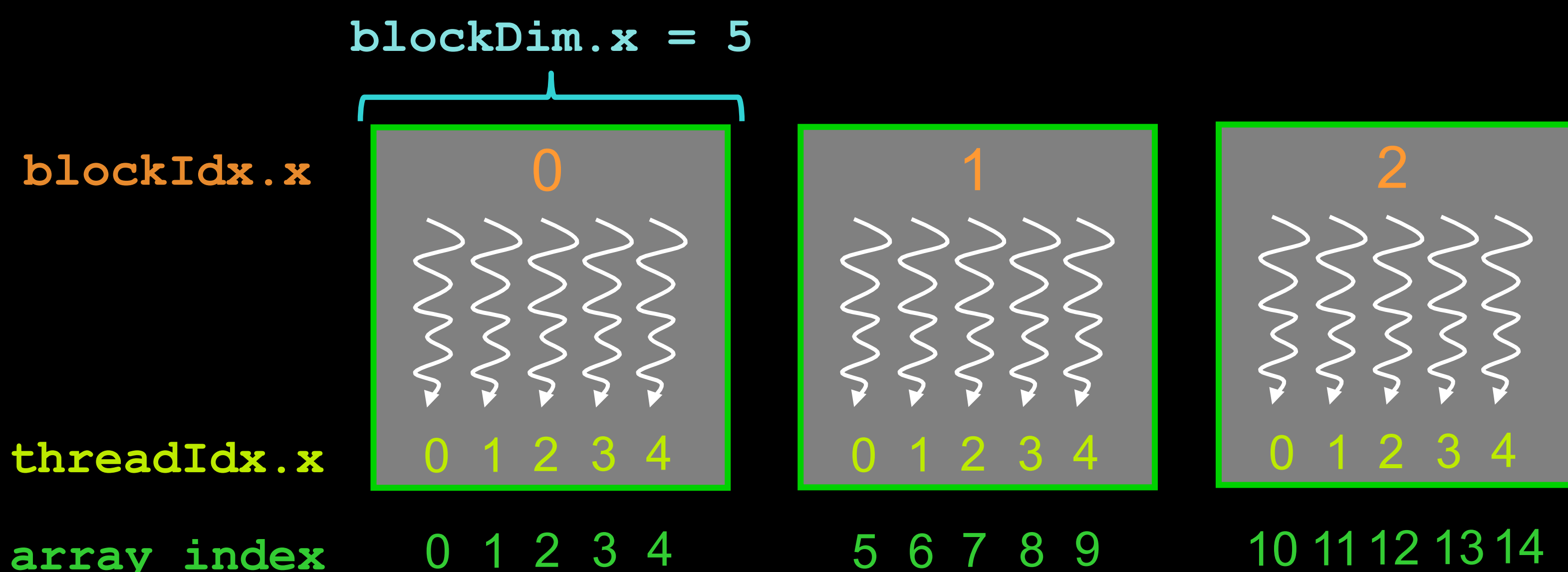
IDs and Dimensions

- **Threads:**
  - **3D IDs, unique within a block**
- **Blocks:**
  - **3D IDs, unique within a grid**
- **Built-in variables:**
  - **threadIdx, blockIdx**
  - **blockDim, gridDim**



Array Accesses: Index Calculation

- Obtain unique **array index** from block/thread IDs
  - threadIdx, blockIdx
  - blockDim, gridDim



`threadIdx.x + blockDim.x*blockIdx.x`

Kernel Launch

- Usual C/C++ function call, with an additional specification of grid and block sizes:

```
mykernel <<< grid, block >>> (...);
```

- `dim3 grid; dim3 block;`
  - three `ints`: `block.x, block.y, block.z`
- Kernel is launched by the CPU
  - CC >= 3.5: kernels can launch other kernels
- Executed on the GPU

Example: One-dimensional Kernel

```
__global__ void mykernel (int *a, int n)
{
    int ind = threadIdx.x + blockDim.x * blockIdx.x;
    if (ind<n) a[ind] = a[ind] + 1;
}

int main()
{
    dim3 block = dim3(128,1,1);  // 128 threads
    // ensure enough blocks to cover n elements (round up)
    dim3 grid = dim3( (n + block.x - 1) / block.x, 1, 1);
    mykernel <<<grid, block>>> (d_a, n);

    // Also possible:
    // launch 4 blocks, each with 128 threads
    mykernel <<<4,128>>> (d_a, n);
}
```

Example: Two-dimensional Kernel

```
__global__ void mykernel (int *a, int w, int h)
{
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    int y = threadIdx.y + blockDim.y * blockIdx.y;
    int ind = x + w*y;
    if (x<w && y<h) a[ind] = a[ind] + 1;
}

int main()
{
    dim3 block = dim3(32,8,1);   // 32*8 = 256 threads
    // ensure enough blocks to cover w * h elements (round up)
    dim3 grid = dim3( (w + block.x - 1) / block.x,
                      (h + block.y - 1) / block.y, 1 );
    mykernel <<<grid,block>>> (d_A, dimx, dimy);
}
```

Always Check Validity of Indices

- There may be more threads than array elements
  - Always test whether the indices are within bounds

```
__global__ void mykernel (int *a, int n)
{
    int ind = threadIdx.x + blockDim.x * blockIdx.x;
    if (ind<n) a[ind] = a[ind] + 1;
}

__global__ void mykernel (int *a, int w, int h)
{
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    int y = threadIdx.y + blockDim.y * blockIdx.y;
    int ind = x + w*y;
    if (x<w && y<h) a[ind] = a[ind] + 1;
}
```

Exercise: IDs of Threads and Blocks

```
kernel<<<4,4>>>(d_a);
```

```
__global__ void kernel (int *a)
{
    int idx = threadIdx.x  + blockDim.x * blockIdx.x;
    a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel (int *a)
{
    int idx = threadIdx.x  + blockDim.x * blockIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernel(int *a)
{
    int idx = threadIdx.x  + blockDim.x * blockIdx.x;
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Code Executed on GPU: Functions

- **Special qualifiers to declare GPU functions:**
  - __global__ : kernels
    - launched by CPU to run on the GPU
    - must return void
  - __device__ : auxiliary GPU functions
    - can only be called on the GPU
    - called from __global__ or __device__ functions
  - __host__ : “normal” CPU C/C++ functions
    - can only be called on the CPU
  - __host__ __device__ : qualifiers can be combined
    - callable from CPU and from GPU

Code Executed on GPU: Restrictions

- C/C++ with some restrictions
  - only access to GPU memory
    - not to CPU memory
    - can access „pinned“ CPU memory (special allocation needed)
    - from CUDA 6 and CC 3.0: GPU can access CPU memory
  - no access to host functions
  - no variable number of arguments
  - no static variables in functions or classes

Code Executed on GPU: Features

- Many C/C++ features available for GPU code
  - templates
  - recursion (CC >=2.0)
  - overloading
    - function overloading
    - operator overloading
  - classes
    - stack allocation
    - heap allocation (CC >= 2.0)
    - inheritance, virtual functions (CC >= 2.0)
  - function pointers (CC >= 2.0)
  - printf() formatted output (CC >= 2.0)
- Vector variants of basic types
  - `float2`, `float3`, `float4`, `double2`, `int4`, `char2`, etc.
  - `float2 a=make_float2(1,2); a.x=10; a.y=a.x;`

Blocks: Must Be Independent

- Any possible ordering of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order (order is unspecified)
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock
- Independence requirement gives scalability

Execution of Kernels: Asynchronous

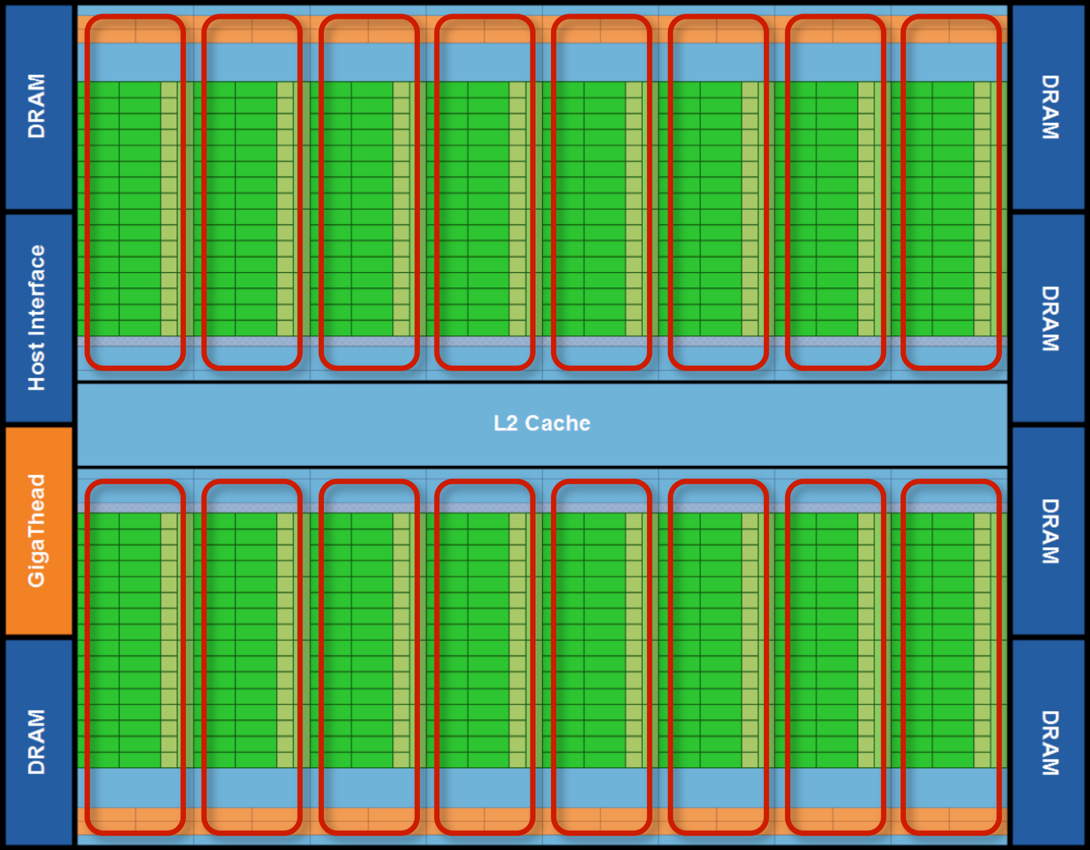
- Kernel launches are **asynchronous** w.r.t. CPU
  - after kernel launch, control **immediately** returns
  - CPU is free to do other work while the GPU is busy
- Kernel launches are **queued**
  - kernel doesn‘t start until previous kernels are finished
  - concurrent kernels possible for CC >= 2.0

    (given enough resources)
- **Explicit synchronization** if needed
  - `cudaDeviceSynchronize()`

EXECUTION ON GPU

NVIDIA GPU Architecture

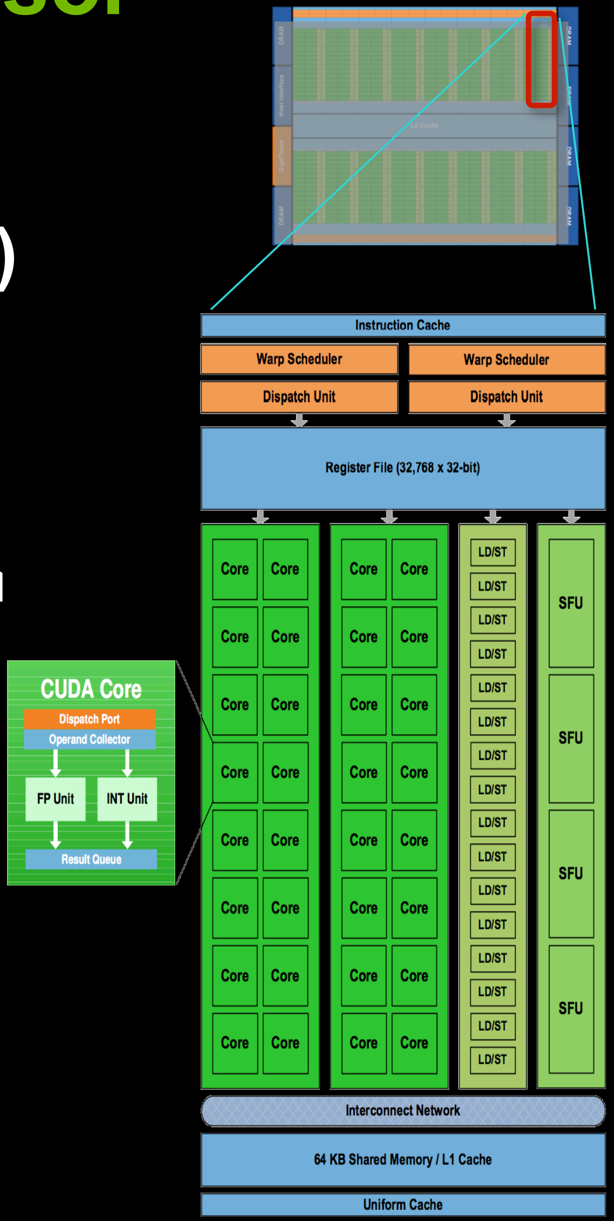
Fermi
GPU
(CC 2.x)



- **16 independent multiprocessors (SMs)**
- **No shared resources except global memory**
- **No synchronization, always work in parallel**

Single Fermi SM Multiprocessor

- 32 CUDA Cores per SM (512 total)
  - arithmetic/logic operations
- 16 memory load/store units
  - (slow) access to off-chip GPU mem
- 4 Special Function Units
  - 1/X, 1/SQRT(X), SIN, COS, EXP, …
- 64 KB on-chip shared memory
  - shared amongst CUDA cores
  - enables thread communication



NVIDIA GPU Architecture: Current

Kepler
GPU
(CC 3.x)

- 15 multiprocessors (up to)
- 192 Cuda Cores per SM
    - 2880 Cores in total (up to)

NVIDIA GPU Architecture: Current

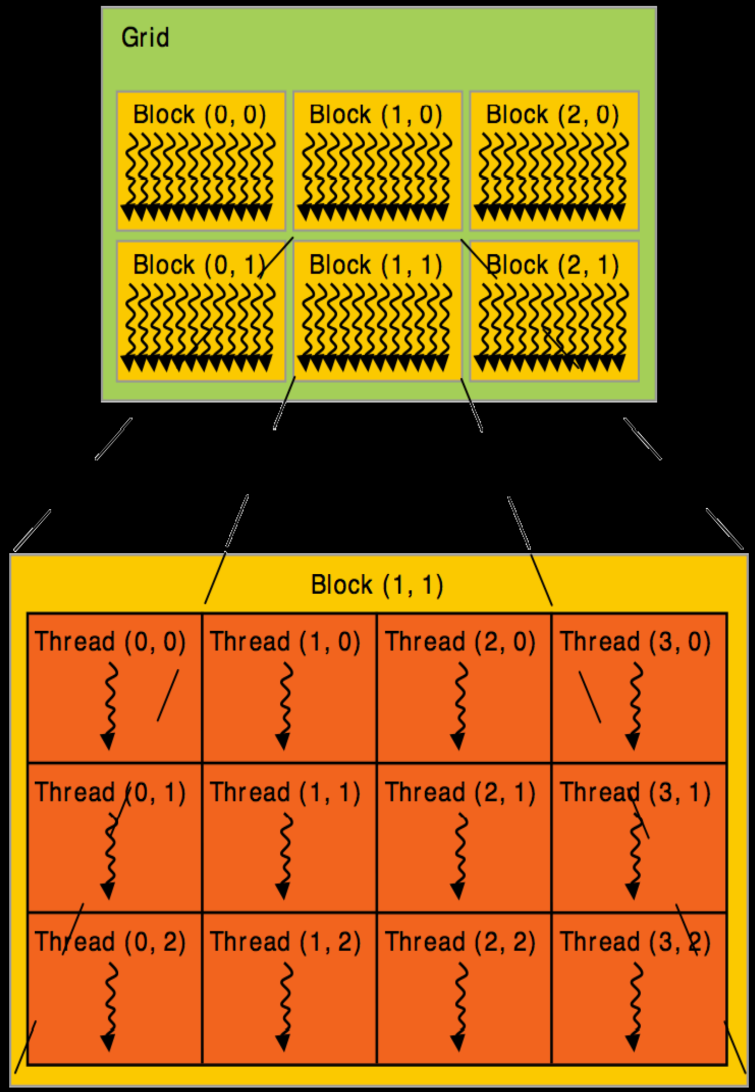
Maxwell GPU (CC 5.0)

- **5 multiprocessors (up to)**
- **128 Cuda Cores per SM**
  - **640 Cores in total (up to)**

Warps: Key Architectural Idea

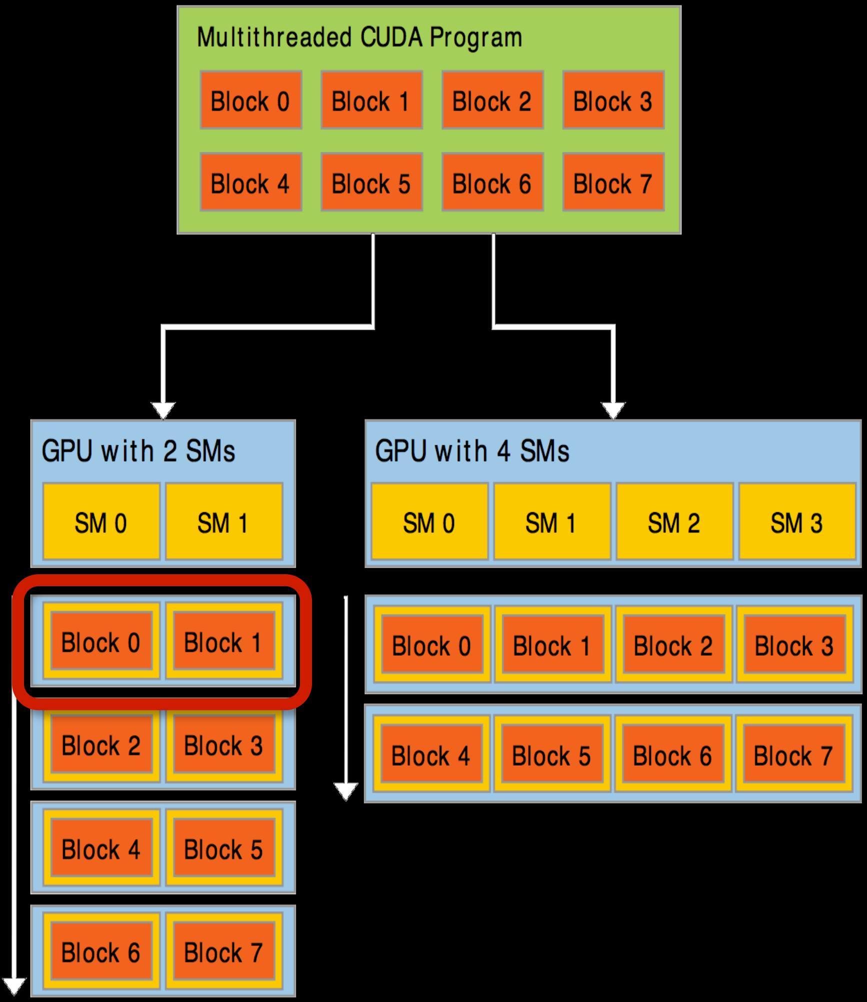
- SIMT (Single Instruction Multiple Thread) **execution**
  - **threads run in groups of 32 called warps**
- **All 32 threads in a warp execute the same instruction**
  - **always, no matter what (even if threads diverge)**
- **Threads are executed warp-wise by the GPU**
  - **for each warp, the 32 threads are executed in parallel**
  - **warps are executed one after another**
  - **but several warps can run simultaneously**
    - **up to 2 for CC 2.x, up to 6 for CC 3.x**

Thread Hierarchy



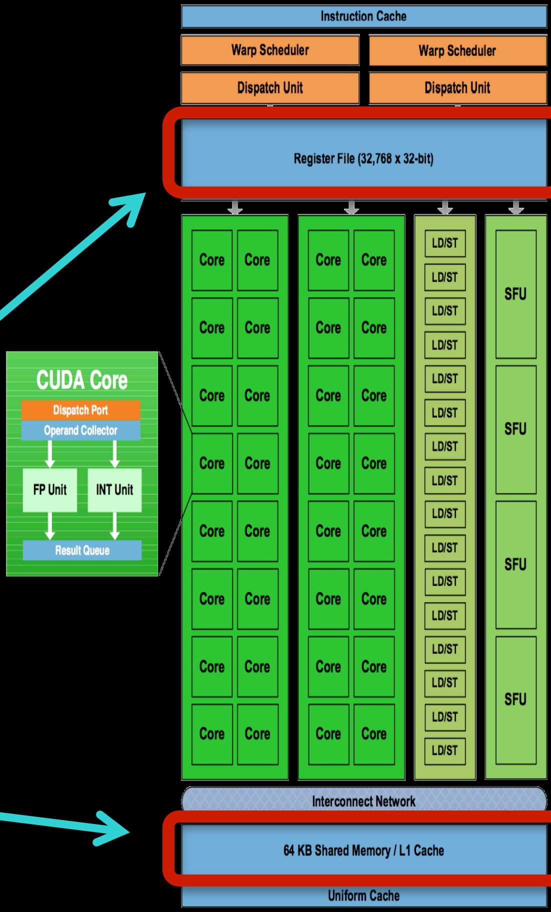
Execution of Kernels on the GPU

- Blocks are distributed across the Multiprocessors (SMs)
- Active blocks
  - are currently executed
  - reside on a multiprocessor
  - resources allocated
  - executed until finished
- Waiting blocks
  - wait to be executed
  - not yet assigned to a SM



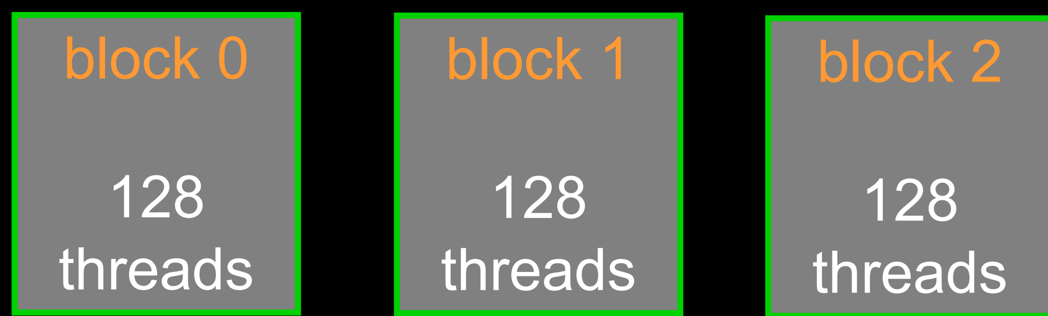
Blocks Execute on Multiprocessors

- Each block is executed on one Multiprocessor (SM)
  - cannot migrate
  - reason for block independence
- Several blocks per SM possible
  - if enough resources available
  - SM resources are divided among all blocks
- Block threads share SM resources
  - SM registers are divided up among the threads
  - SM shared memory can be read/written by all threads



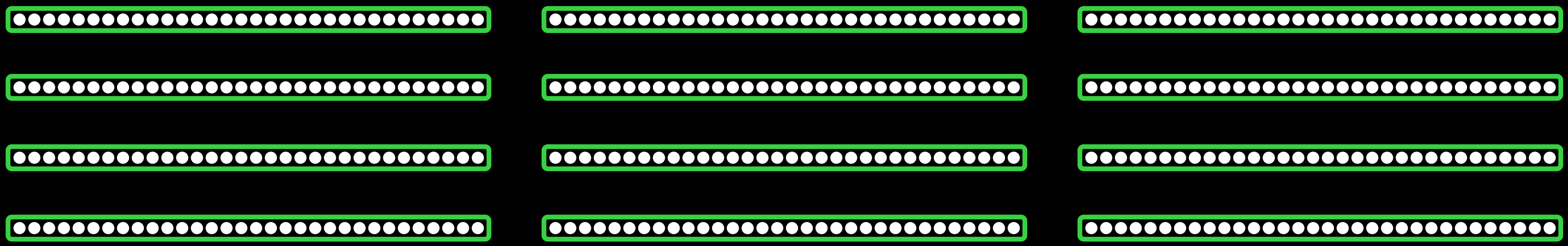
Execution on each Multiprocessor

- **Assume there are three blocks on one SM, with 128 threads per block:**



Execution on each Multiprocessor

- Threads from all blocks are divided into warps
- In our example:
  - 4 warps from every block (128 threads/32)
  - 12 warps overall on SM (3 blocks * 4 warps/block)
    - 12*32 = 384 threads

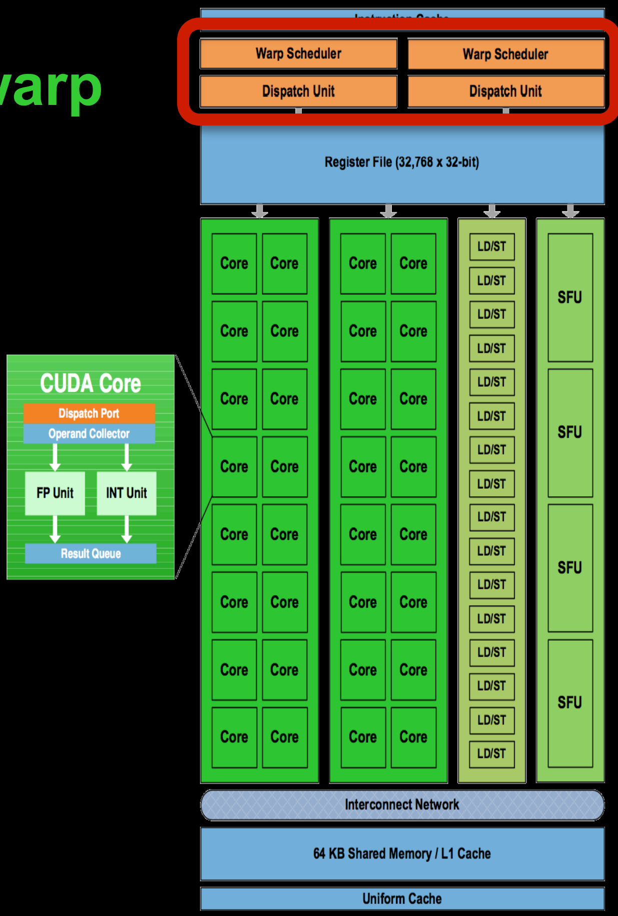


Execution on each Multiprocessor

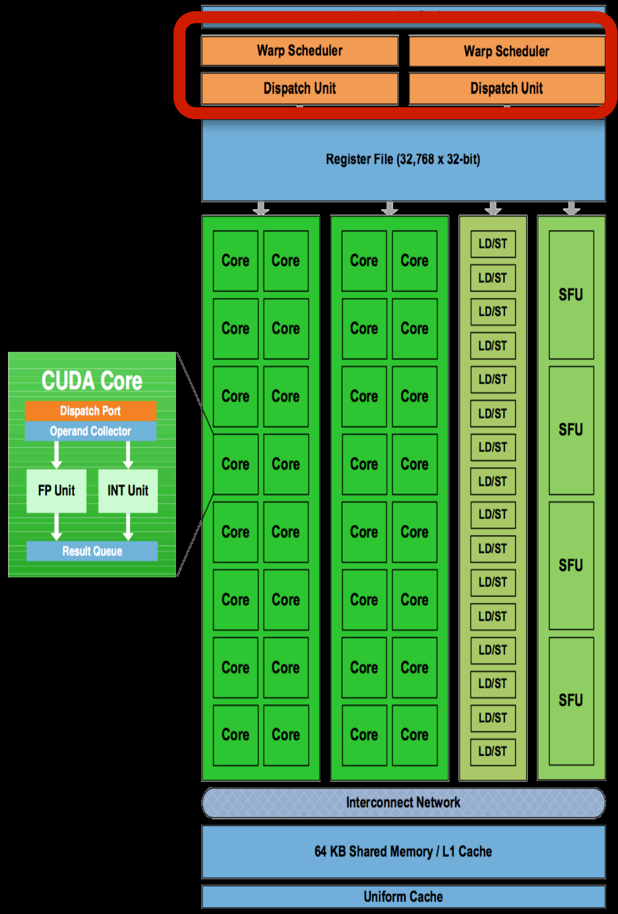
- Resources are allocated for all potential warps
  - the state of every potentially executable warp is always present on the Multiprocessor, until finished
  - overall many more potentially executable threads than CUDA Cores possible
- Therefore:
  - switching between warps is free
  - any non-waiting warp can run

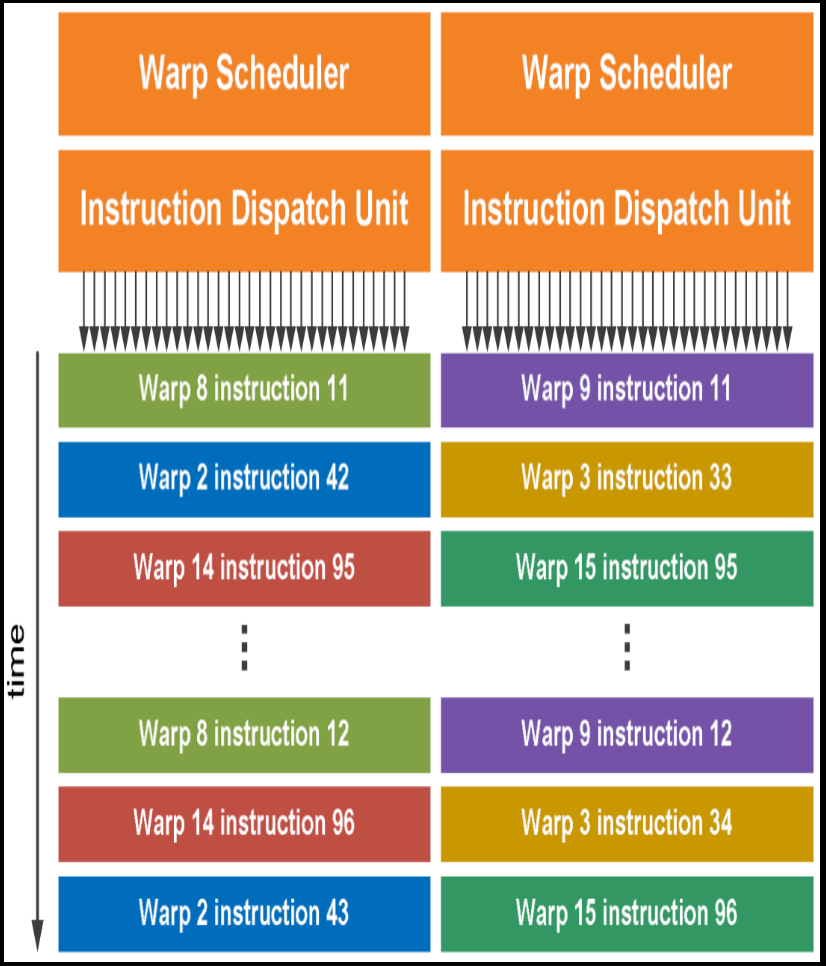
Execution on each Multiprocessor

- At each clock cycle
  - each warp scheduler chooses a warp which is ready to be executed
- For each chosen warp
  - the next instruction is executed for all 32 threads of the warp
  - issued for execution to
    - CUDA Cores
    - or load/store units
    - or special function units
    - or texture units



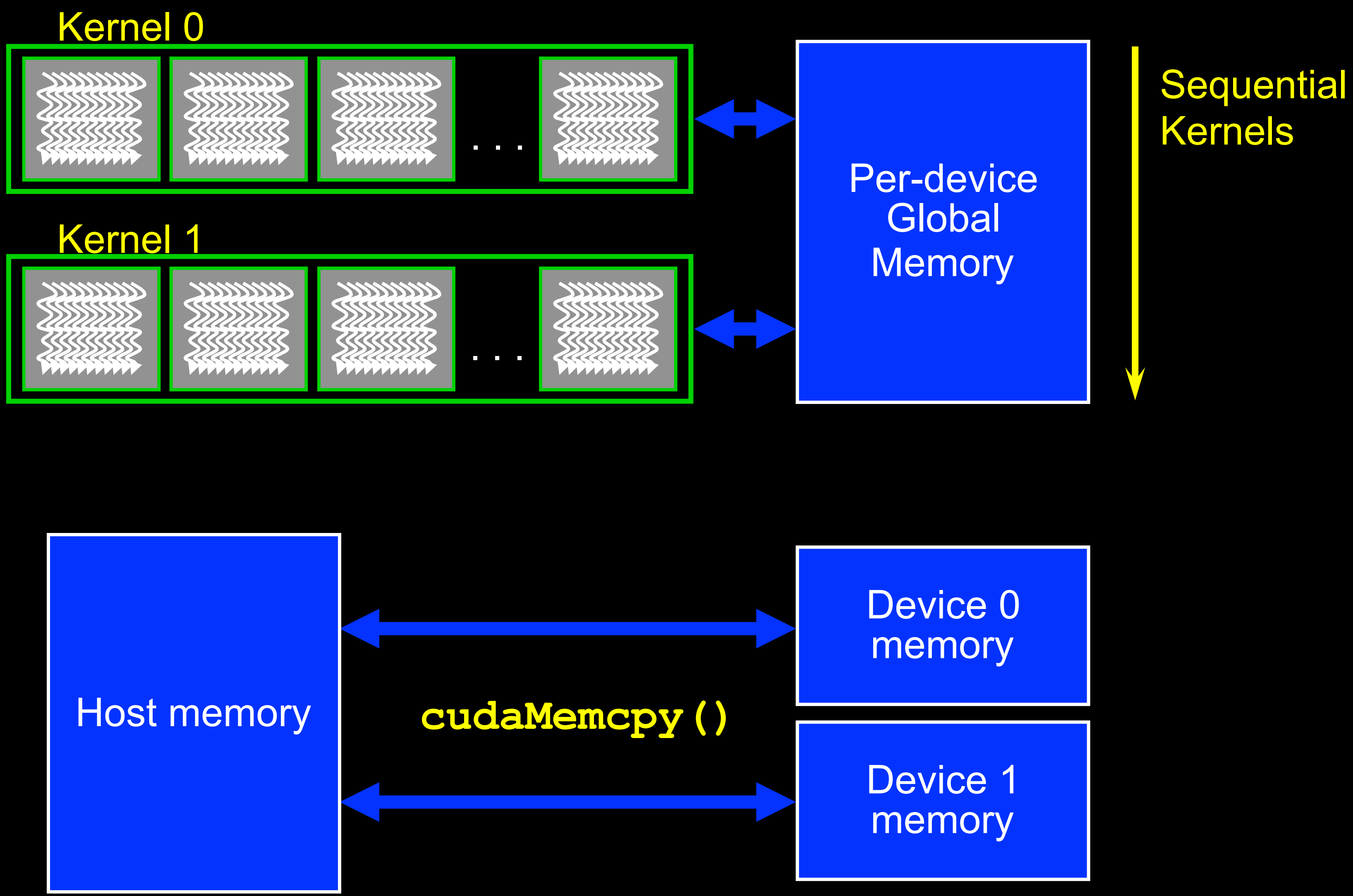
Execution on each Multiprocessor





MEMORY MANAGEMENT

GPU Memory


Kernel 0
Kernel 1
. . .
. . .
Per-device Global Memory
Sequential Kernels
Host memory
cudaMemcpy()
Device 0 memory
Device 1 memory

GPU Memory

- CPU and GPU have separate memory spaces
  - data is moved across PCIe bus
  - use functions to allocate/set/copy memory on GPU
    - very similar to corresponding C functions
- Pointers are just addresses
  - cannot tell from pointer if memory is on GPU or CPU
    - but possible for CC>=2.0: unified virtual addressing
  - must exercise care when dereferencing:
    - crash if GPU dereferences pointer to CPU memory
    - and vice versa

Allocation / Release

- Host (CPU) manages device (GPU) memory:
  - cudaMalloc (void **pointer, size_t nbytes)
  - cudaMemset (void *pointer, int value, size_t count)
  - cudaFree (void* pointer)

```
int n = 1024;
size_t nbytes = n*sizeof(int);
int *d_a = NULL;
cudaMalloc(&d_a, nbytes);
cudaMemset(d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies Between GPU and CPU

- cudaMemcpy (void *dst, void *src, size_t nbytes, cudaMemcpyKind direction);
  - blocks the CPU thread until all bytes have been copied
  - non-blocking variants are also available
  - doesn't start copying until all previous CUDA calls complete
- cudaMemcpyKind:
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

```
cudaMemcpy(dev_ptr, host_ptr, n*sizeof(float),
    cudaMemcpyHostToDevice);
```

Example Host Code

```
// allocate and initialize host (CPU) memory
float *h_a = ...,    *h_b = ...; *h_c = ... (empty)

// allocate device (GPU) memory
float *d_a, *d_b, *d_c;
cudaMalloc( &d_a, n * sizeof(float) );
cudaMalloc( &d_b, n * sizeof(float) );
cudaMalloc( &d_c, n * sizeof(float) );

// copy host memory to device
cudaMemcpy( d_a, h_a, n * sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( d_b, h_b, n * sizeof(float), cudaMemcpyHostToDevice );

// launch kernel
dim3 block = dim3(128,1,1);
dim3 grid = dim3((n + block.x - 1) / block.x, 1, 1);
vecAdd <<<grid,block>>> (d_a, d_b, d_c);

// copy result back to host (CPU) memory
cudaMemcpy( h_c, d_c, n * sizeof(float), cudaMemcpyDeviceToHost );

// do something with the result...

// free device (GPU) memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

Use `float` by Default

- GPUs can handle `double` since CC>=1.3
- But `float` operations are still much faster
  - by an order of magnitude
  - so use `double` only if `float` is not enough

- Avoid using `double` where not needed:
  - Add 'f' suffix to float literals:
    - `0.f`, `1.0f`, `3.1415f` are of type `float`
    - `0.0`, `1.0`, `3.1415` are of type `double`
  - Use `float` version of math functions:
    - `expf` / `logf` / `sinf` / `sqrtf` / etc. take and return `float`
    - `exp` / `log` / `sin` / `sqrt` / etc. take and return `double`

Blocks Size: How To Choose?

- Number of threads/block should be multiple of 32
  - because threads are always executed in groups of 32
- Rules of thumb:
  - not too small or too big: between 128 and 256 threads
  - start with `dim3(32,8,1)`, i.e. 256 threads
  - experiment with similar sized "power-of-2"-blocks:
    - (64,4,1), (128,2,1), (32,4,1), (64,2,1)
    - (32,16,1), (64,8,1), (128,4,1), (256,2,1)
  - measure the run time and choose the best block size!

ERROR HANDLING AND COMPILING

Error Handling

- Checking for errors is crucial for programming GPUs
- `cudaError_t cudaGetLastError()`
  - returns the code for the last error
  - resets the error flag back to `cudaSuccess`
  - `cudaPeetAtLastError()`: get error code without resetting it
  - if everything OK: `cudaSuccess`
- `char* cudaGetErrorString(cudaError_t code)`
  - returns a C-string describing the error

```
cudaMalloc(&d_a, n*sizeof(float));
cudaError_t e = cudaGetLastError();
if (e!=cudaSuccess)
{
    cerr << "ERROR: " << cudaGetErrorString(e) << endl;
    exit(1);
}
```

Error Handling

- **Kernel execution is asynchronous**
  - **first wait for the kernel to finish by `cudaDeviceSynchronize()`**
  - **only then call `cudaGetLastError()`**
    - otherwise it will be called too soon, the error may not have yet occured
  - **kernel launch itself may produce errors due to invalid configurations**
    - too many threads/block, too many blocks, too much shared memory requested
- **Kernels may produce subtle memory corruption errors**
  - **may get unnoticed even after `cudaDeviceSynchronize()`**
  - **subsequent CUDA calls may or may not fail because of such an error**
  - **if they do fail, they were not the origin of the error**
- **It helps to keep track of the previous x CUDA calls**
  - **x=1, or x=2, or x=10**

Compiling

- CUDA files have ending `.cu`: squareArray.cu
- NVidia CUDA Compiler: `nvcc`
  - handles the CUDA part
  - hands over pure C/C++ part to host compiler

```
nvcc -o squareArray squareArray.cu
```

- Additional info about the kernels by option

`--ptxas-options=-v`

```
nvcc -o squareArray squareArray.cu --ptxas-options=-v
```

```
ptxas info    : Compiling entry function '_Z18cuda_square_kernelPfi' for 'sm_10'
ptxas info    : Used 2 registers, 28 bytes smem
```

CUDA Short Summary

Thread Hierarchy

- thread - smallest executable unity
- block - group of threads, shared memory for collaboration
- grid - consists of several blocks
- warp - group of 32 threads

Keyword extensions for C/C++

- `__global__` - kernel - function called by CPU, executed on GPU
- `__device__` - function called by GPU and executed on GPU
- `__host__` - [optional] - function called and executed by CPU
- `<<<...>>>` - kernel launch, chevrons specify grid and block sizes

Compilation:

```
nvcc -o <executable> <filename>.cu --ptxas-options=-v
```