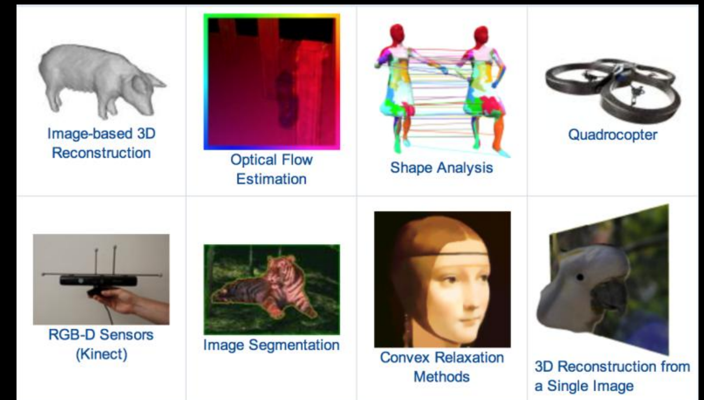
GPU Programming in Computer Vision

Introduction to Parallel Computing

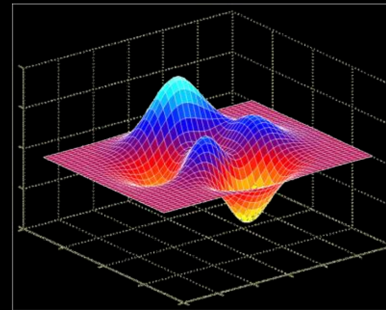
Computer Vision Group

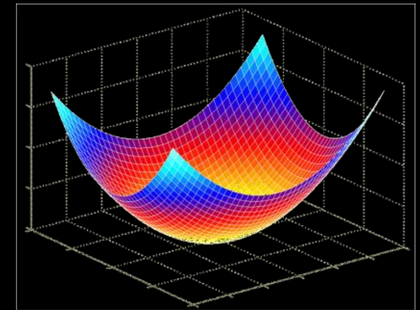
Prof. Dr. Daniel Cremers
Sabine Wagner
Dr. Juergen Sturm
Dr. Rudolph Triebel
Mathieu Aubry
Julia Bergbauer
Jakob Engel
Youngwook Kee
Christian Kerl
Maria Klodt
Quirin Lohr
Martin Oswald
Mohamed Souiai
Frank Steinbrücker
Evgeny Strekalovskiy
Jan Stühmer
Matthias Vestner
Thomas Windheuser

Research



Our Research is about


non-convex


convex

- Optimization
- Math in general
  - everything needs to broken down into functions, basic operations and numbers
- Numerics
  - continuous math on discrete hardware
- Programming (serial/parallel)
  - C/C++, CUDA, Matlab, ...
- Engineering

This Course covers

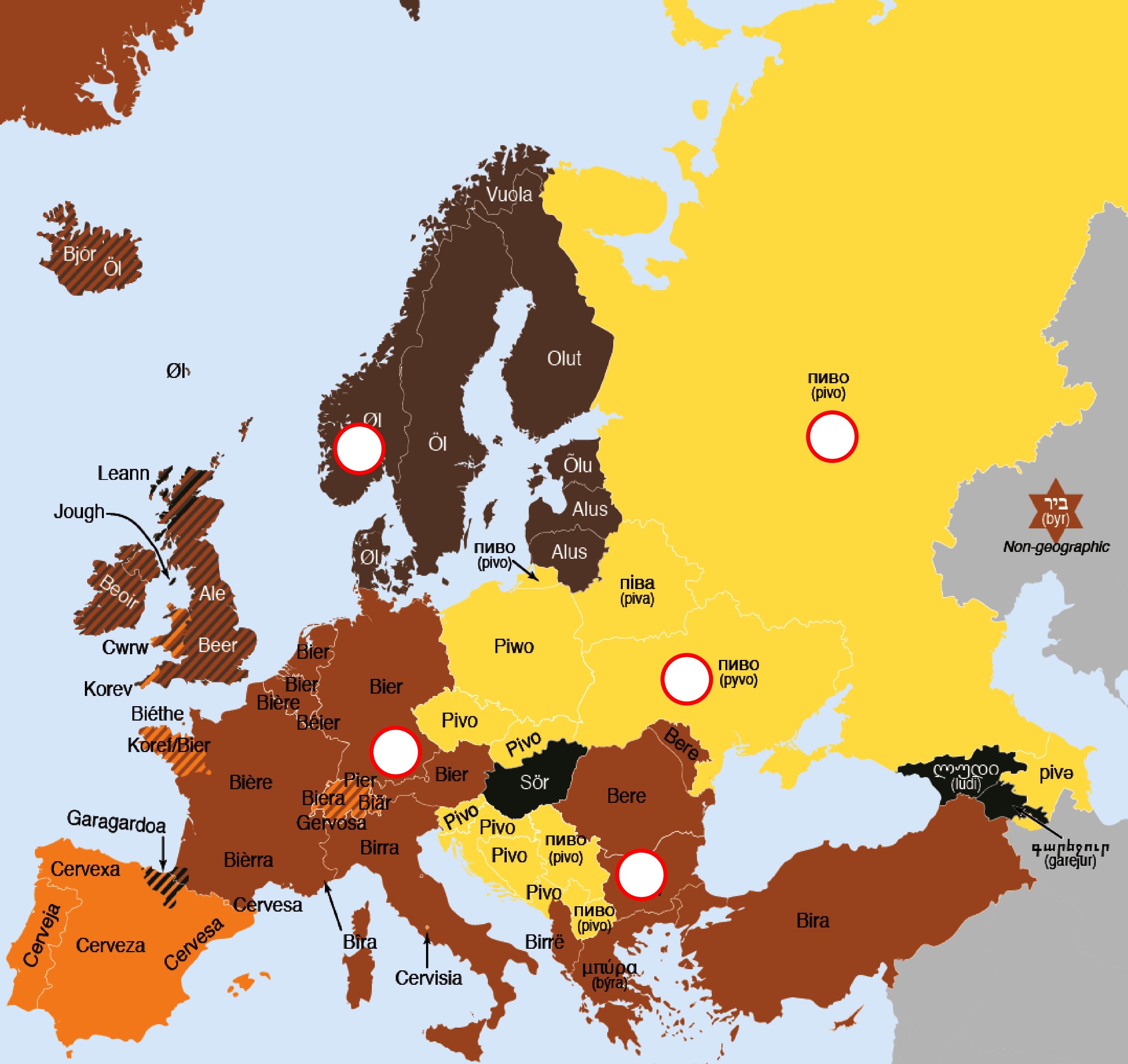
- Parallel Programming (with CUDA)
- Computer Vision Basics
  - Image Filtering (Convolution, Diffusion)
  - Regularization (dealing with noise, unique solutions)
- Optimization + Numerics
- Example Problems
  - Optical Flow Estimation
  - Superresolution

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

The Essential Map of Europe and Environs

	Beer/Bier
	Ale/Øl
	Pivo
	Cerveza
	Others



Based on the Wikimedia map of Europe by Júlio Reis and Marian "maix" Sigler (http://commons.wikimedia.org/wiki/File:Blank_map_of_Europe.svg)
Released under CreativeCommons Attribution ShareAlike (http://creativecommons.org/licenses/by-sa/2.5/

ARCTIC OCEAN
ARCTIC OCEAN
ARCTIC OCEAN
UNITED STATES
CANADA
UNITED STATES
RUSSIA
MONGOLIA
CHINA
NORTH ATLANTIC OCEAN
NORTH PACIFIC OCEAN
NORTH PACIFIC OCEAN
BRAZIL
SOUTH PACIFIC OCEAN
INDIAN OCEAN
AUSTRALIA
SOUTH ATLANTIC OCEAN
SOUTH PACIFIC OCEAN
SOUTHERN OCEAN
SOUTHERN OCEAN
SOUTHERN OCEAN
ANTARCTICA

Course Timeline

- Aug. 26-30 (this week) : Lecture
  - 4h lectures (attendance mandatory)
  - programming exercises
- Sep. 2-20: Student project
  - optical flow/superresolution
  - groups of 2-3 students
  - unsupervised
- Sep. 23-25: Presentations
  - 20 minutes presentation
  - 25 minutes questions

September

Mo.	Di.	Mi.	Do.	Fr.	Sa.	So.
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	1	2	3	4	5	6

Lecture Week

- Lecture
  - 10-14 (1h lunch pause) each day
  - attendance mandatory to pass the course
- Exercises
  - 14-18 each day
  - no need to be finished the same day
- Deadline for exercises:
  - 02.09.2013, 23:59
  - Submit all solutions by email in a zip achive

September

Mo.	Di.	Mi.	Do.	Fr.	Sa.	So.
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	1	2	3	4	5	6

Remote Login

- You can access your computer remotely:

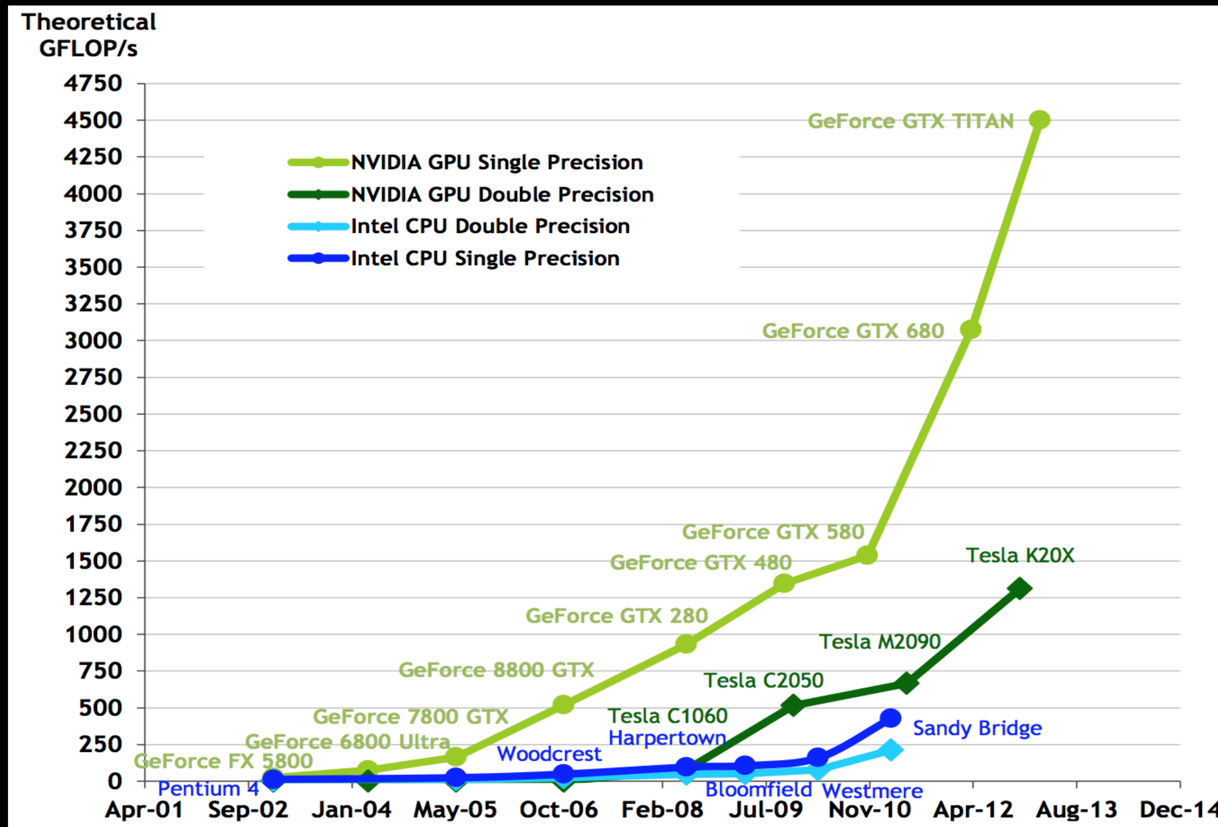
```
ssh -X p123@atradig789.informatik.tu-muenchen.de
```

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

- Works from within Linux or Mac
  - for Macs: install XQuartz first (X11 window system)

Why Massively Parallel Processing?

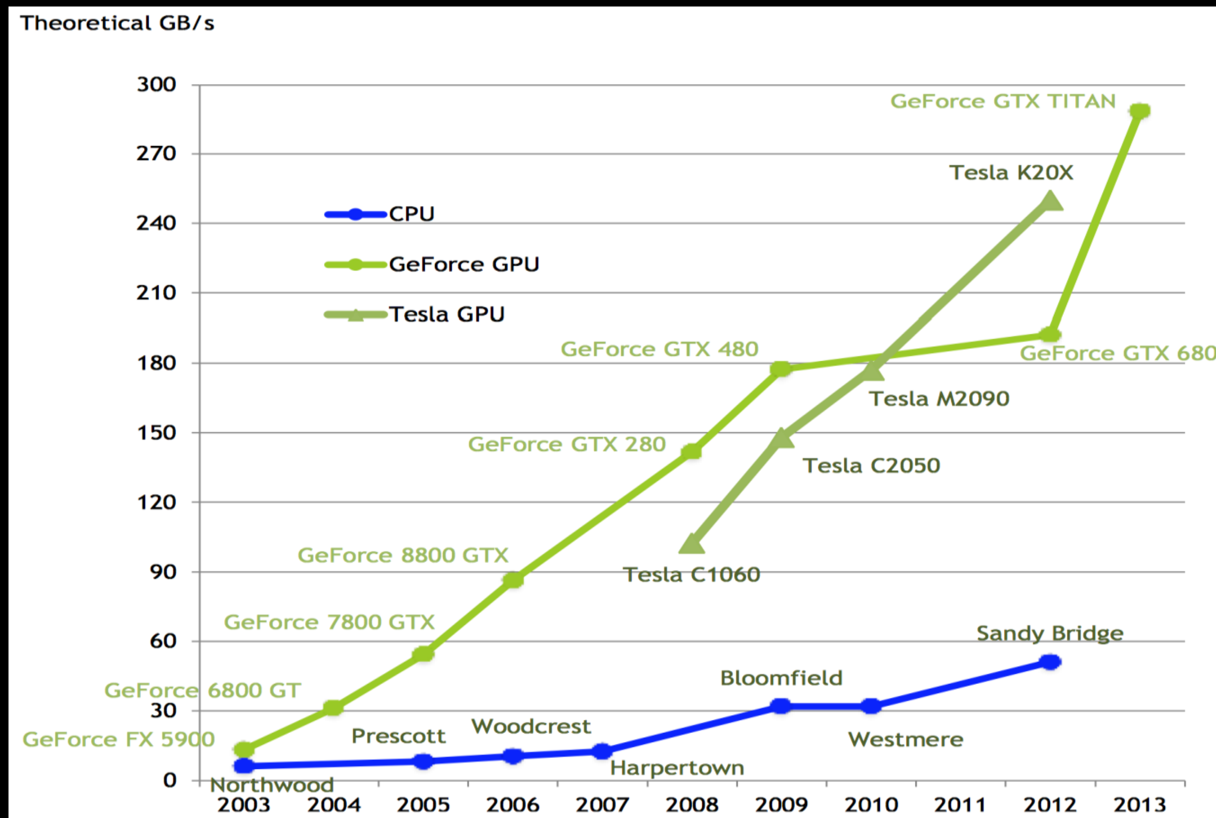
- A quiet revolution: Performance!
  - Computations: TFLOPs vs. 100 GFLOPs



  - 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?

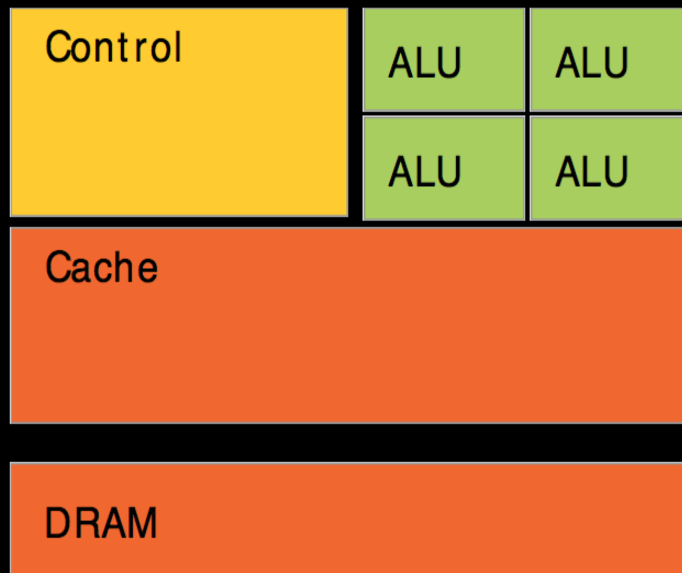
- Larger caches … decreasing
- Instruction-level parallelism … decreasing
  - out-of-order execution, speculation, …
- Data-level parallelism … increasing
  - vector units, SIMD execution, …
  - Intel SSE, GPUs, …
- Thread-level parallelism … increasing
  - multithreading, multicore, manycore

Design difference: CPU vs. GPU

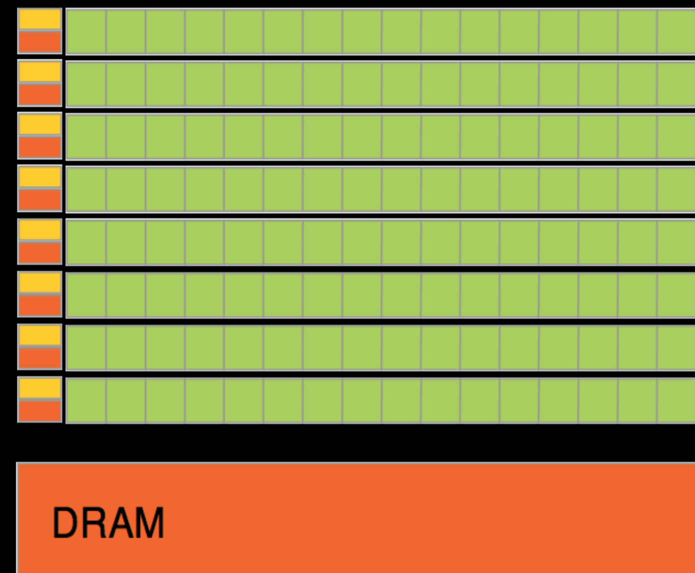
- Different goals produce different designs
  - CPU must be good at everything, parallel or not
  - GPU assumes work load is highly parallel
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - skip big caches, 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

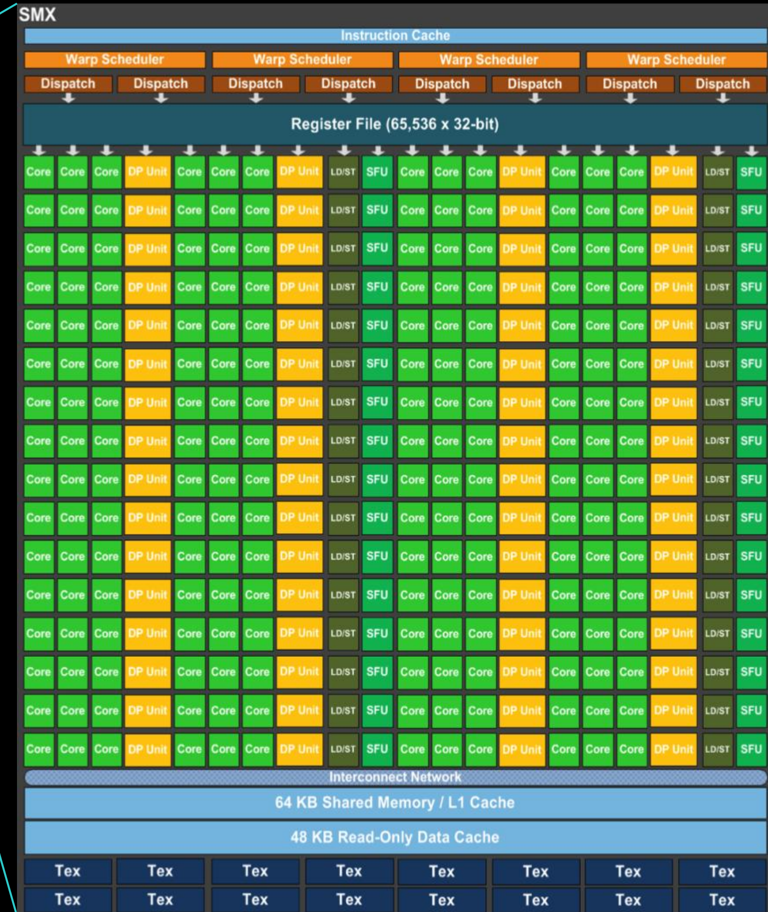
NVIDIA GPUs

- **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

NVIDIA GPUs: Current

Kepler GPU



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

Outline of CUDA Basics

- Kernels and Thread Hierarchy
- Execution on the GPU
- Memory Management

- 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

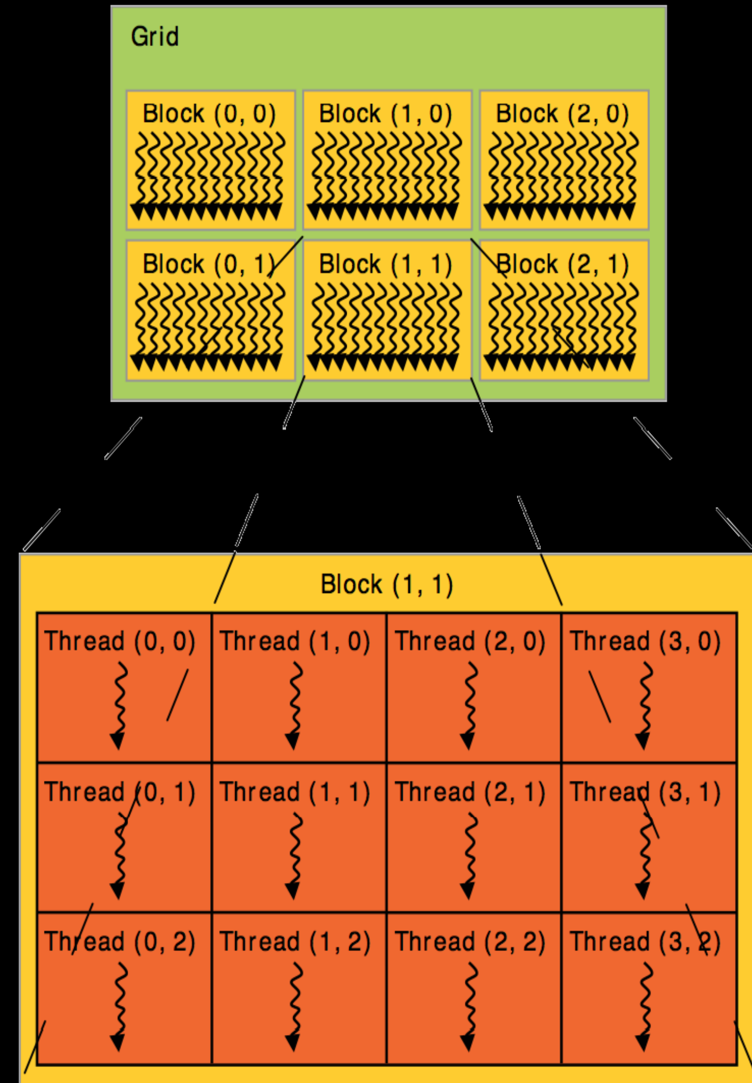
  - **Launch enough threads to cover all data**

Thread Hierarchy

- Kernel threads are grouped into blocks
  - up to 512 (CC 1.x), 1024 (CC 2.x), or 2048 (CC 3.x) 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

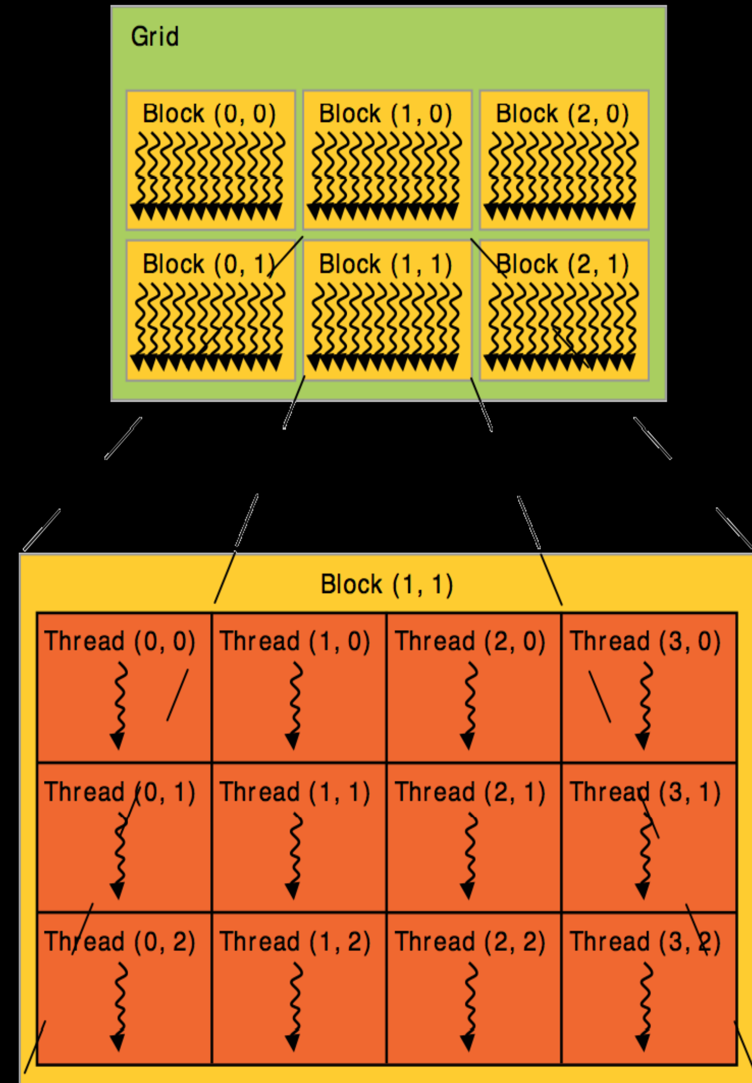
Thread Hierarchy

- # threads per block:
  - up to 512 (CC 1.x),
  - up to 1024 (CC 2.x),
  - up to 2048 (CC 3.x)
- 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



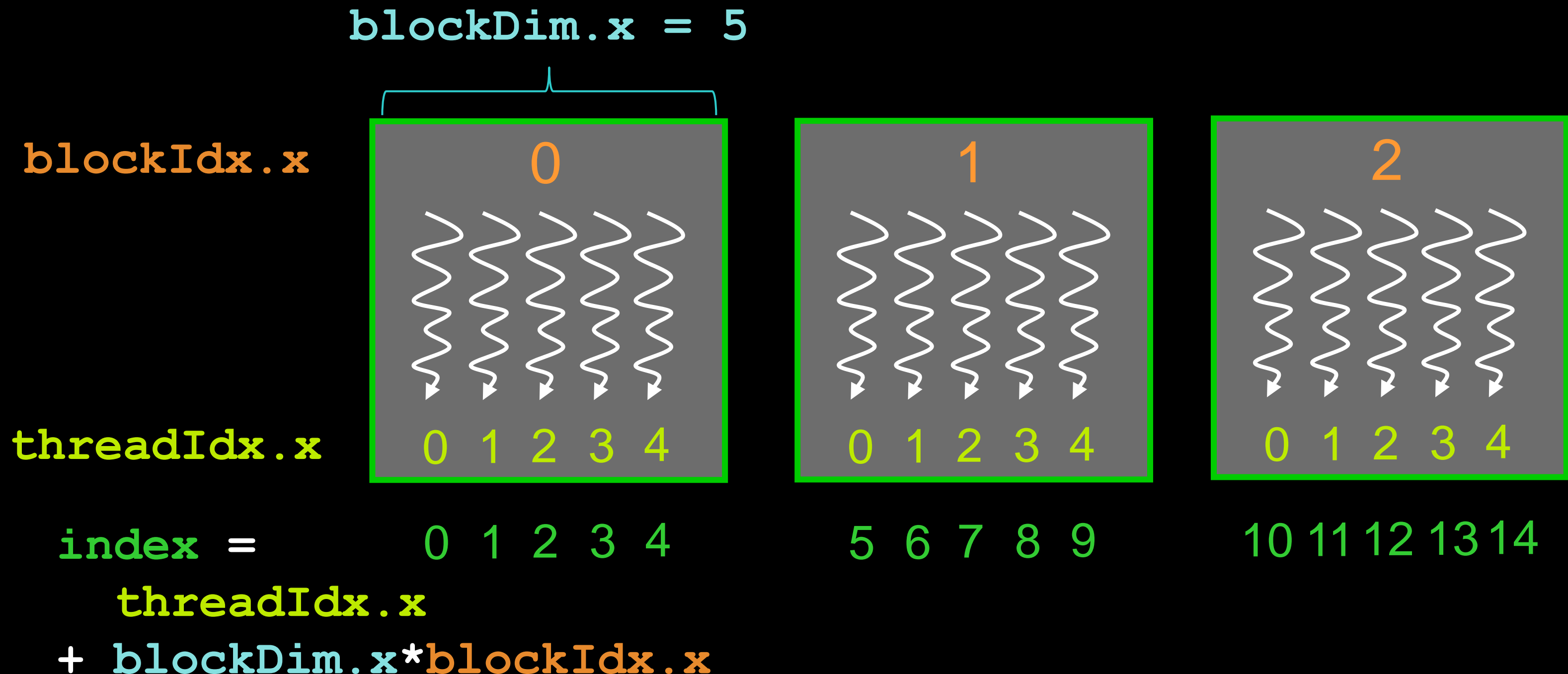
IDs and Dimensions

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



Array Accesses: Indexing

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



Kernel launch

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

```
mykernel <<< gridSize, blockSize >>> (...);
```

- `dim3 gridSize, dim3 blockSize`
  - three int's: `blockSize.x, blockSize.y, blockSize.z`
- Launched on the host side
  - CC 3.x: kernels can launch other kernels

Code executed on GPU: Restrictions

- C/C++ with some restrictions
  - Only access to GPU memory, cannot access CPU memory
    - (but access to „pinned“ host memory, requires special allocation)
  - No access to host functions
  - No variable number of arguments
  - No static variables

Code executed on GPU: Features

- Many C/C++ features available on the GPU
  - Templates
  - Operator overloading
  - Classes, inheritance
  - Recursion (CC >=2.0)
  - Function pointers (CC >= 2.0)
  - new / delete (CC >= 2.0)
  - Dynamic polymorphism, virtual functions (CC >= 2.0)
  - Even printf() ! (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;

Code executed on GPU: Specifiers

- 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__ and __device__ qualifiers can be combined

Example: Vector Addition Kernel

```
// Compute vector sum c = a + b
// Each thread performs one pair-wise addition
__global__ void vecAdd (float* a, float* b, float* c)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    c[i] = a[i] + b[i];
}

int main()
{
    ...
    // Run grid of N/256 blocks of 256 threads each
    vecAdd <<< N/256, 256 >>> (d_A, d_B, d_C);
}
```

Example: 2D Indexing

```
__global__ void kernel (int *a, int dimx, int dimy)
{
    int x   = threadIdx.x + blockDim.x * blockIdx.x;
    int y   = threadIdx.y + blockDim.y * blockIdx.y;
    int ind = x + dimx*y;
    a[ind]  = a[ind]+1;
}

int main()
{
    ...
    dim3 block = dim3( 32, 8, 1 );
    dim3 grid = dim3( dimx/block.x, dimy/block.y, 1);
    kernel <<<grid,block>>> (d_A, dimx, dimy);
}
```

Kernel Variations and Output

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

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

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

Blocks must be independent

- Any possible interleaving 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

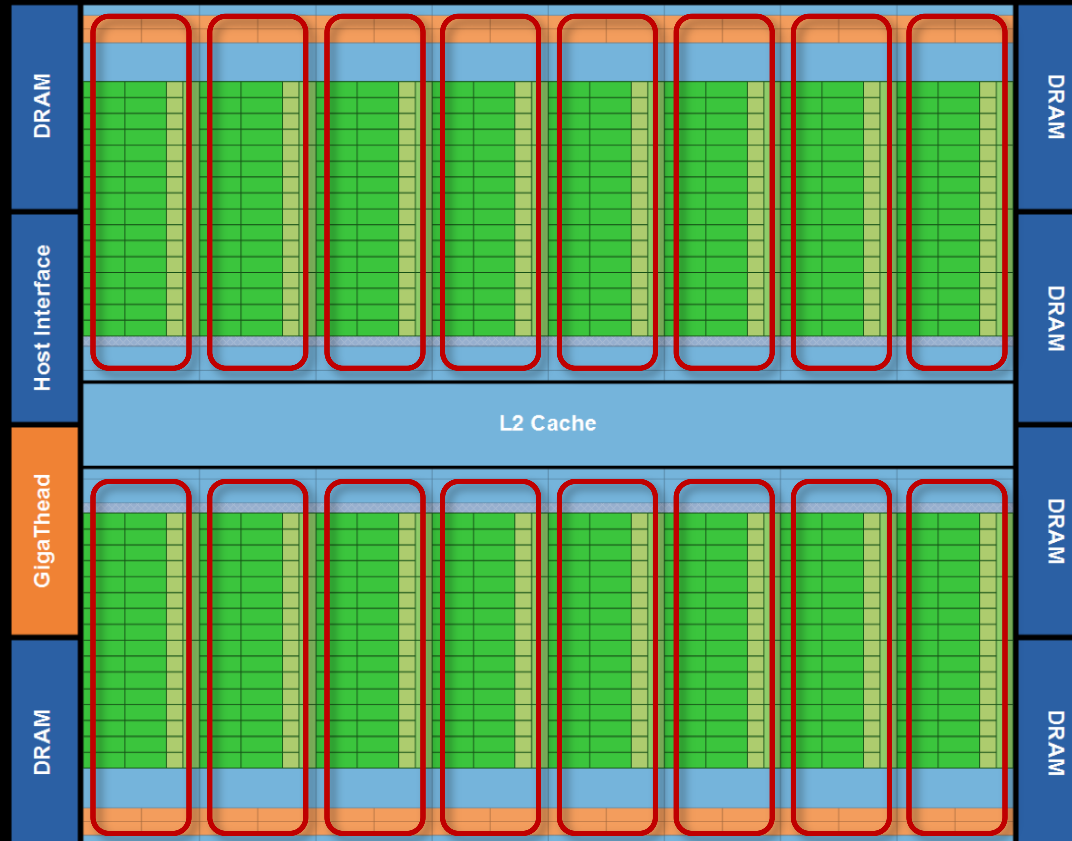
Execution of Kernels

- Kernel launches are asynchronous w.r.t. CPU
  - After kernel launch, control immediately returns
  - CPU is free to do other stuff 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

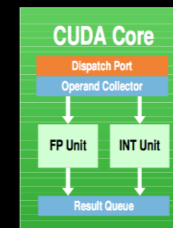
Fermi GPU

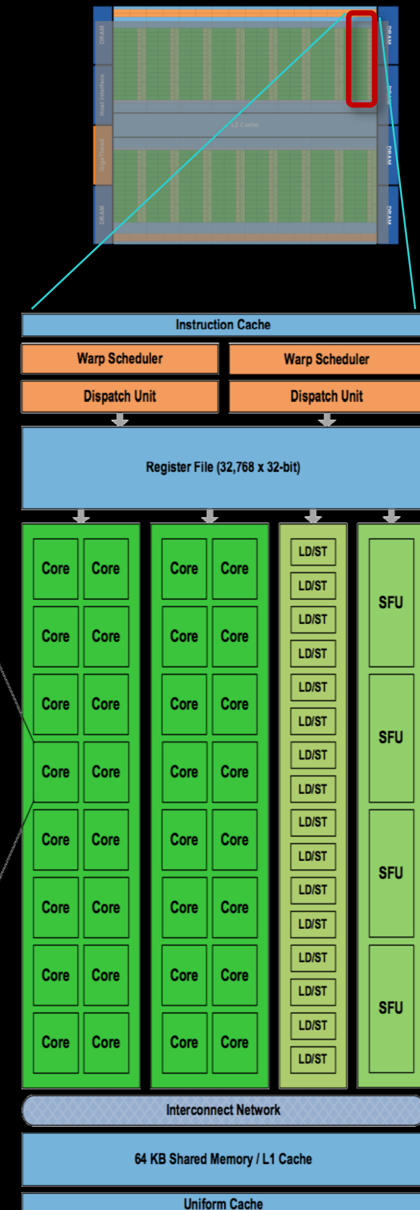


- 16 independent multiprocessors (SMs)
- No shared resources except global memory
- No synchronization, always work in parallel

Single Fermi SM Multiprocessor

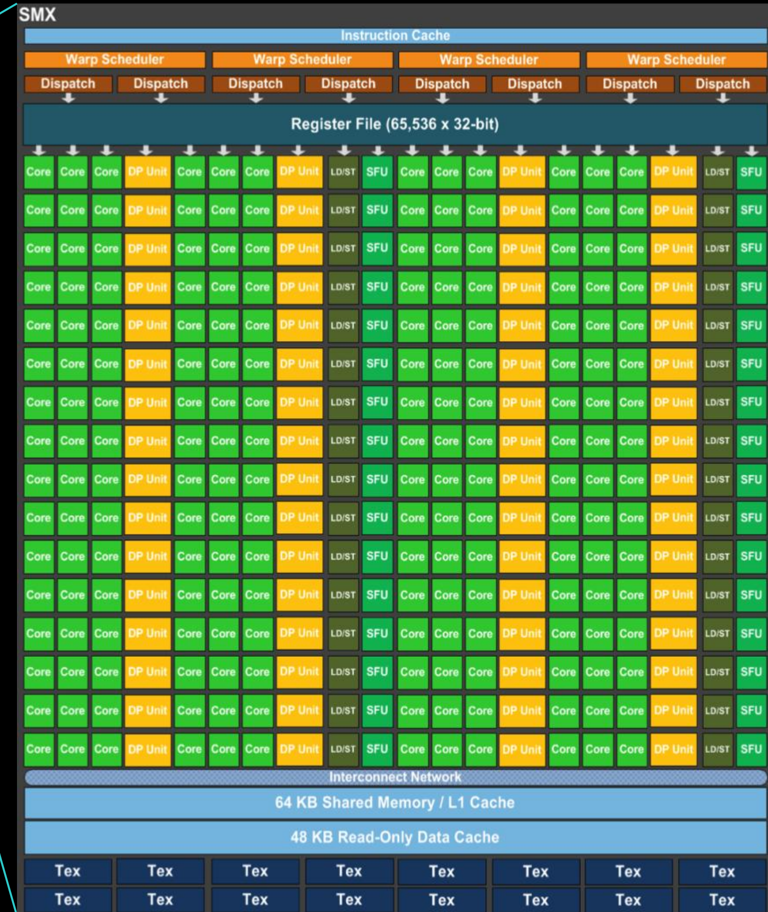
- 32 CUDA Cores per SM (512 total)
  - arithmetic/logic operations
- 16 memory load/store units
  - (slow) access to off-chip GPU mem
- 4 Special Function Units
  - 1/X, 1/SQRT(X), SIN, COS, EXP, …
- 64 KB on-chip shared memory
  - shared amongst CUDA cores
  - enables thread communication





NVIDIA GPU Architecture: Current

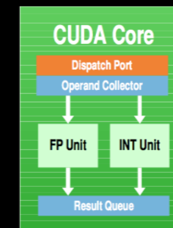
Kepler
GPU

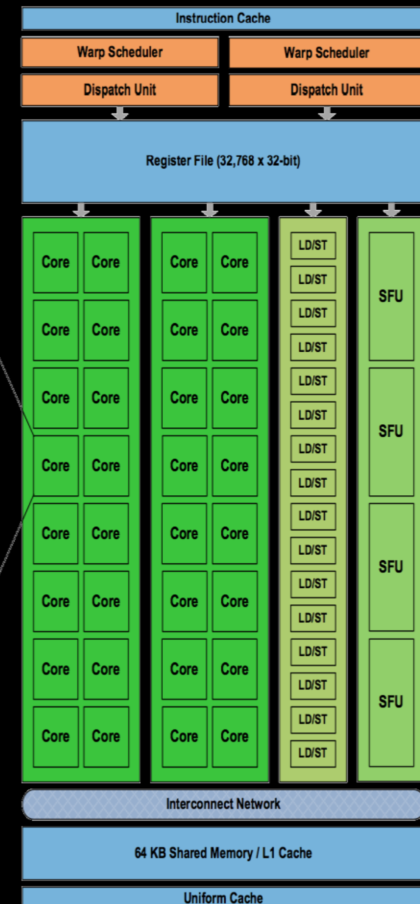


- 15 multiprocessors (up to)
- 192 Cuda Cores per SM
  - 2880 Cores in total (up to)

Key Architectural Ideas

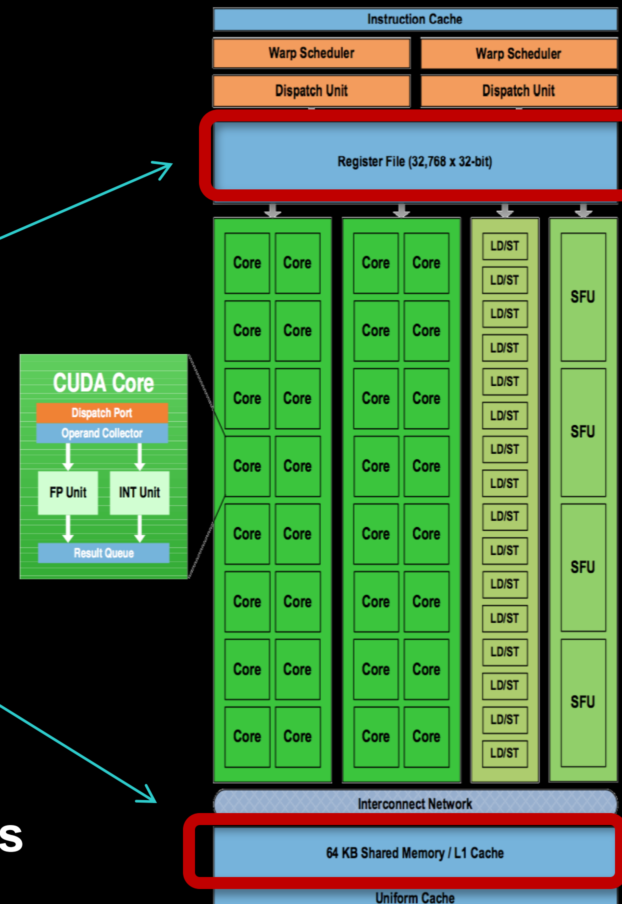
- **SIMT** (Single Instruction Multiple Thread) **execution**
  - **threads run in groups of 32 called warps**
  - **warp threads execute same instructions**
  - **HW automatically handles divergence**
- **Hardware multithreading**
  - **Allocate resources for many more threads than CUDA Cores**
  - **HW schedules which warp(s) to run next**
- **Any non-waiting warp can run**
  - **switching between warps is free**





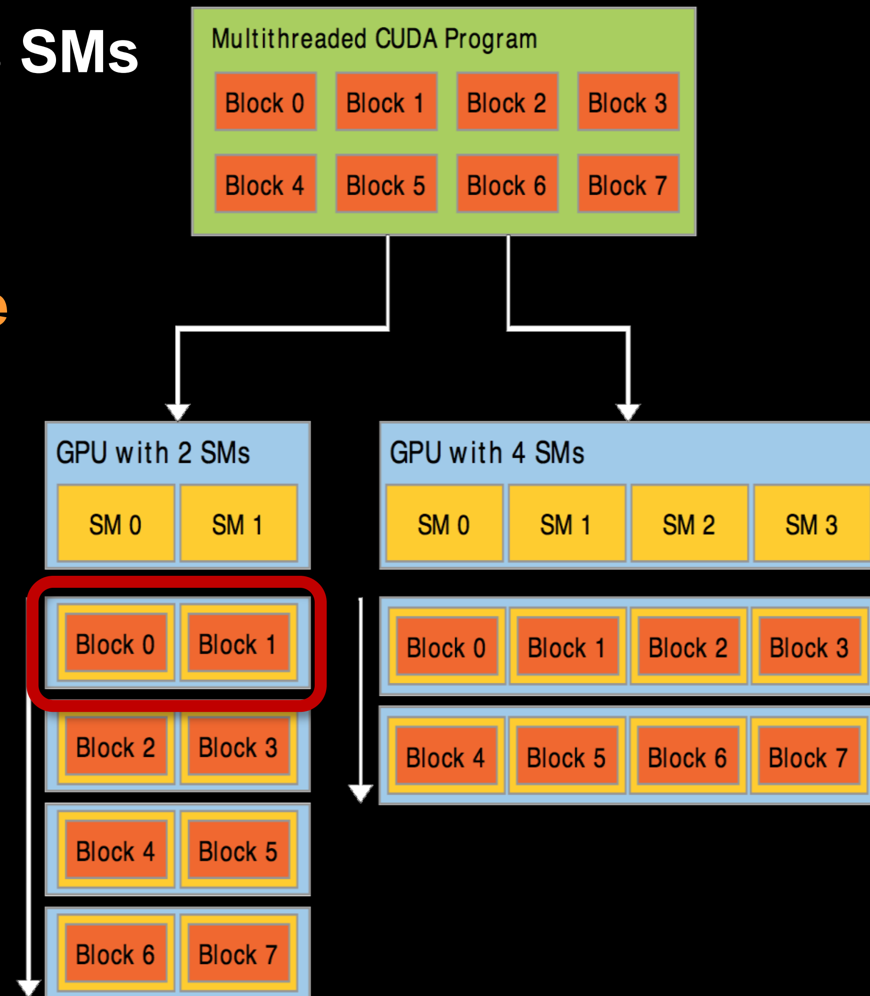
Execution of Kernels on the GPU

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



Execution of Kernels on the GPU

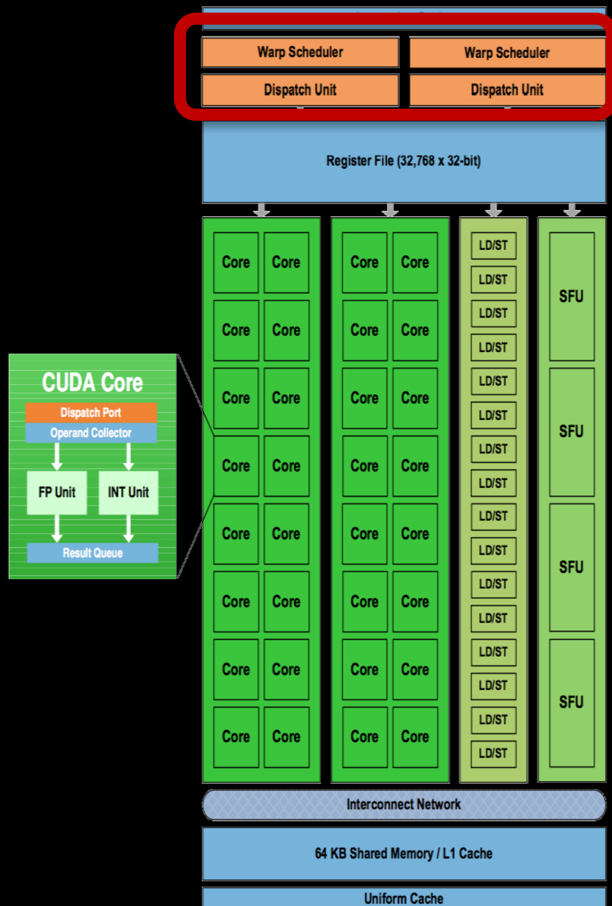
- **Blocks are distributed across SMs**
- **At each moment,**
  one or more blocks are active
  - **reside on a multiprocessor**
  - **resources allocated**
  - **executed until finished**
- **Others wait to be executed**
  - **not yet assigned to a SM**

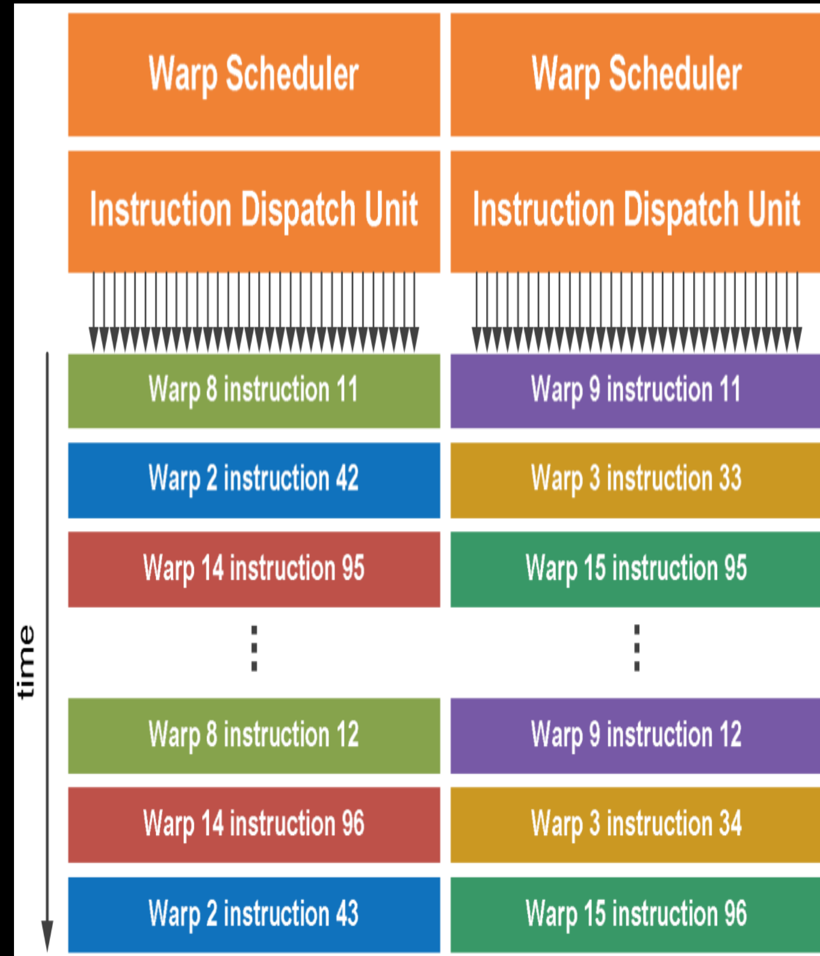


Execution on each Multiprocessor

- On each SM, all blocks which reside on it are divided into warps (groups of 32 threads)
- At each clock cycle:
  - Each warp scheduler chooses a warp which is ready to be executed
  - The next instruction of these warps are issued to the CUDA Cores
    - or to load/store units
    - or to special function units
    - or to texture units

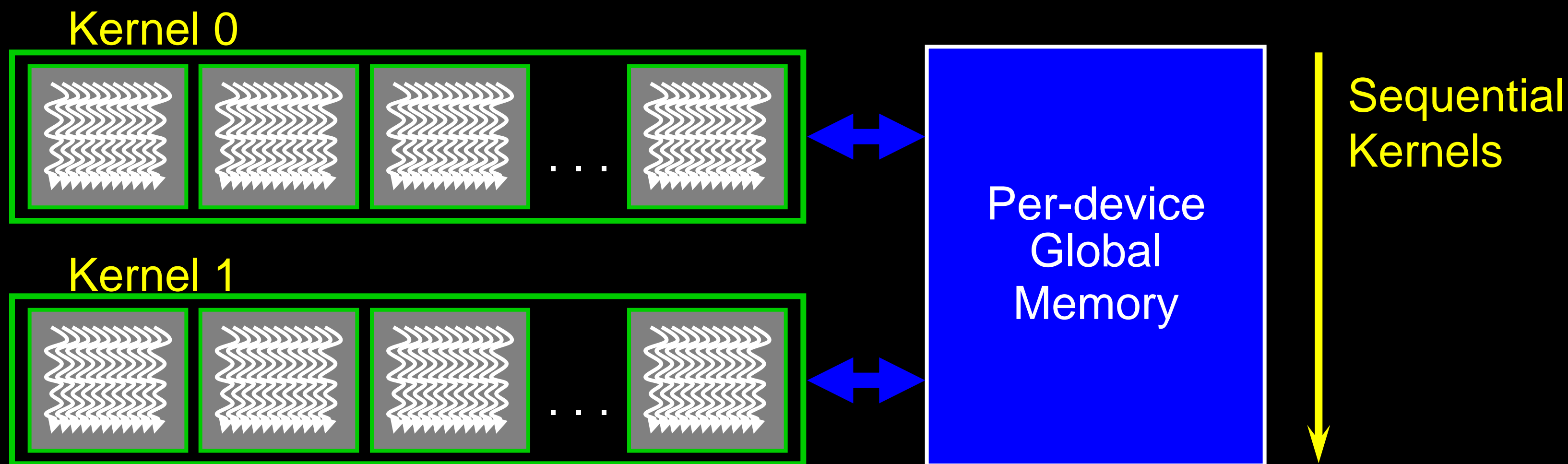
Execution on each Multiprocessor

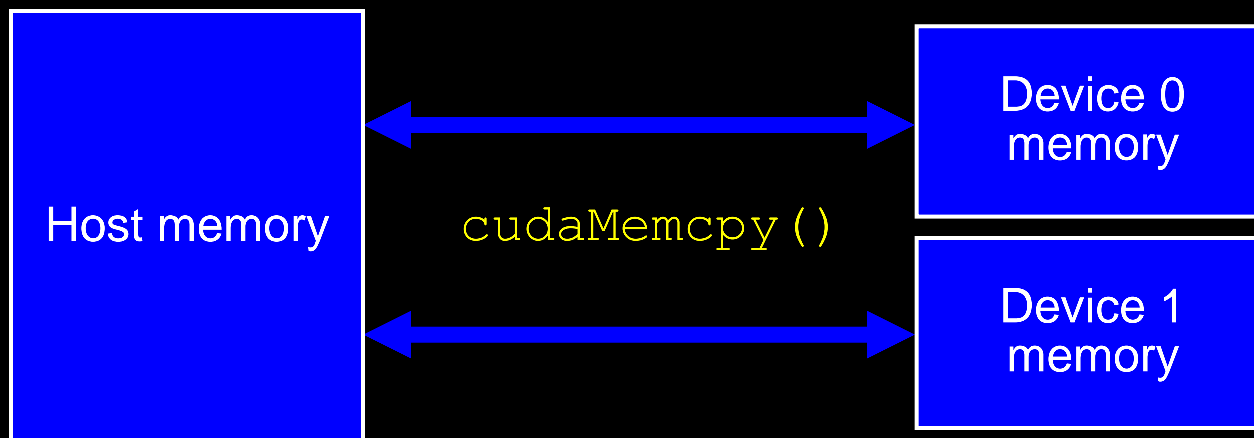




MEMORY MANAGEMENT

Memory Model





Memory Spaces

- 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
  - Can't tell from the pointer value whether the address is on CPU or GPU
    - possible if CC >= 2.0 using unified addressing
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa

GPU Memory 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;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

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

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

Example: Host code for `vecAdd`

```
// 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( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &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 );

// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Example: Host code for `vecAdd` (2)

```
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// copy result back to host 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);
```

CUDA Error Handling

- All CUDA calls return an error code
- `cudaError_t cudaGetLastError(void)`
  - returns the code for the last error
  - if no error: `cudaSuccess`
- `char* cudaGetErrorString(cudaError_t code)`
  - returns a C-string describing the error
- Kernel error detecting: Launches are asynchronous
  - error will not be reported to CPU right after kernel launch
  - first call cudaDeviceSynchronize(), then call cudaGetLastError()
  - kernel launch itself may produce errors for invalid configurations
    - e.g. too many blocks, too many grids, too much shared memory requested

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 <filename>.cu -o <executable>