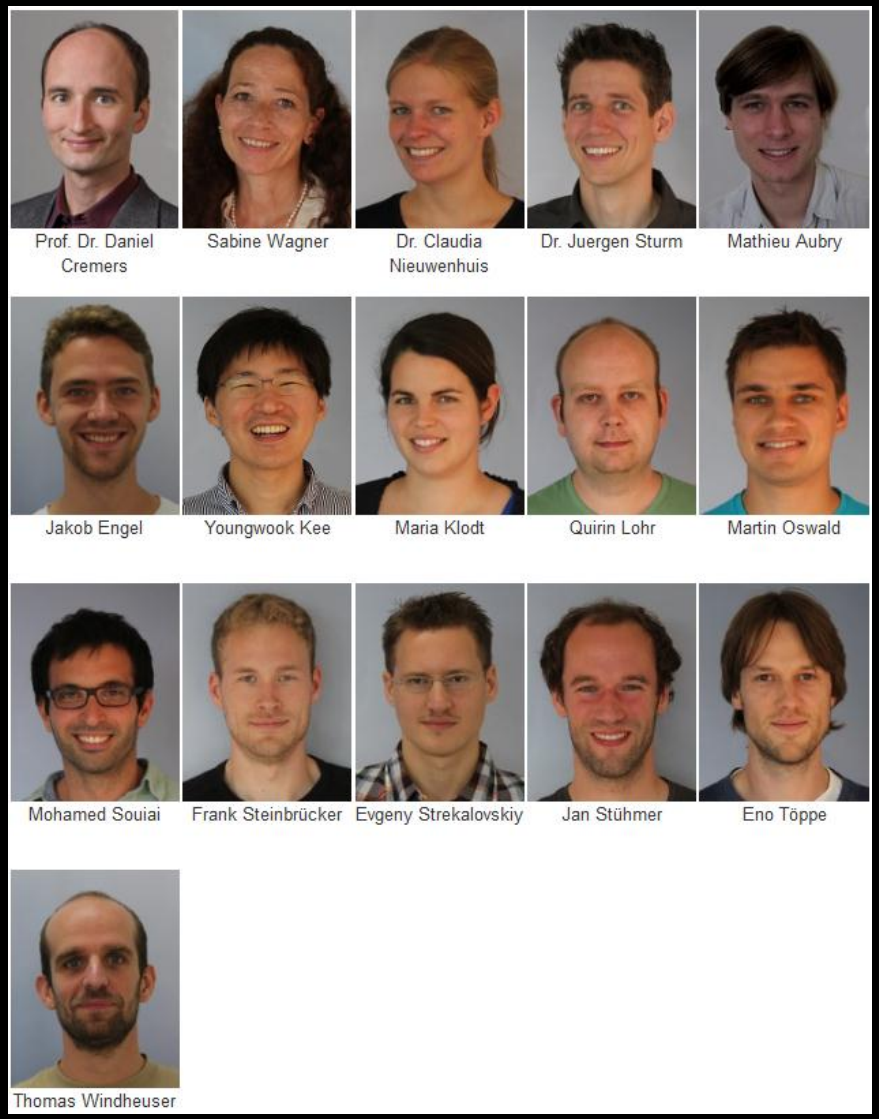
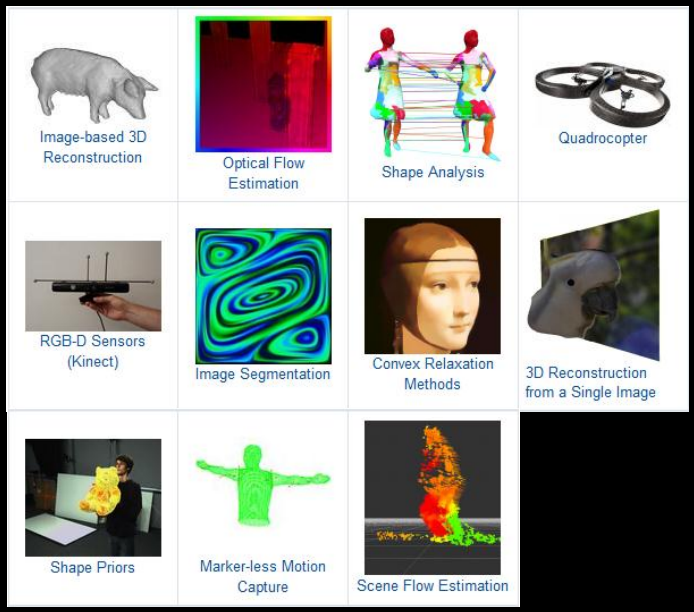
GPU Programming in Computer Vision

Introduction to Parallel Computing

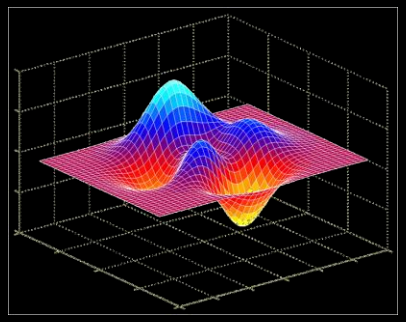
Computer Vision Group



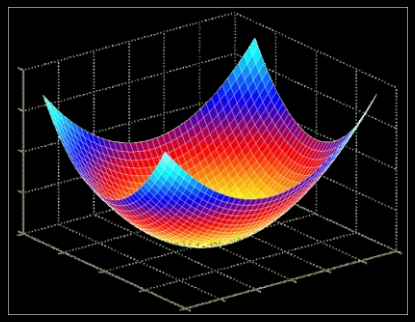
Research



Our Research is about



non-convex



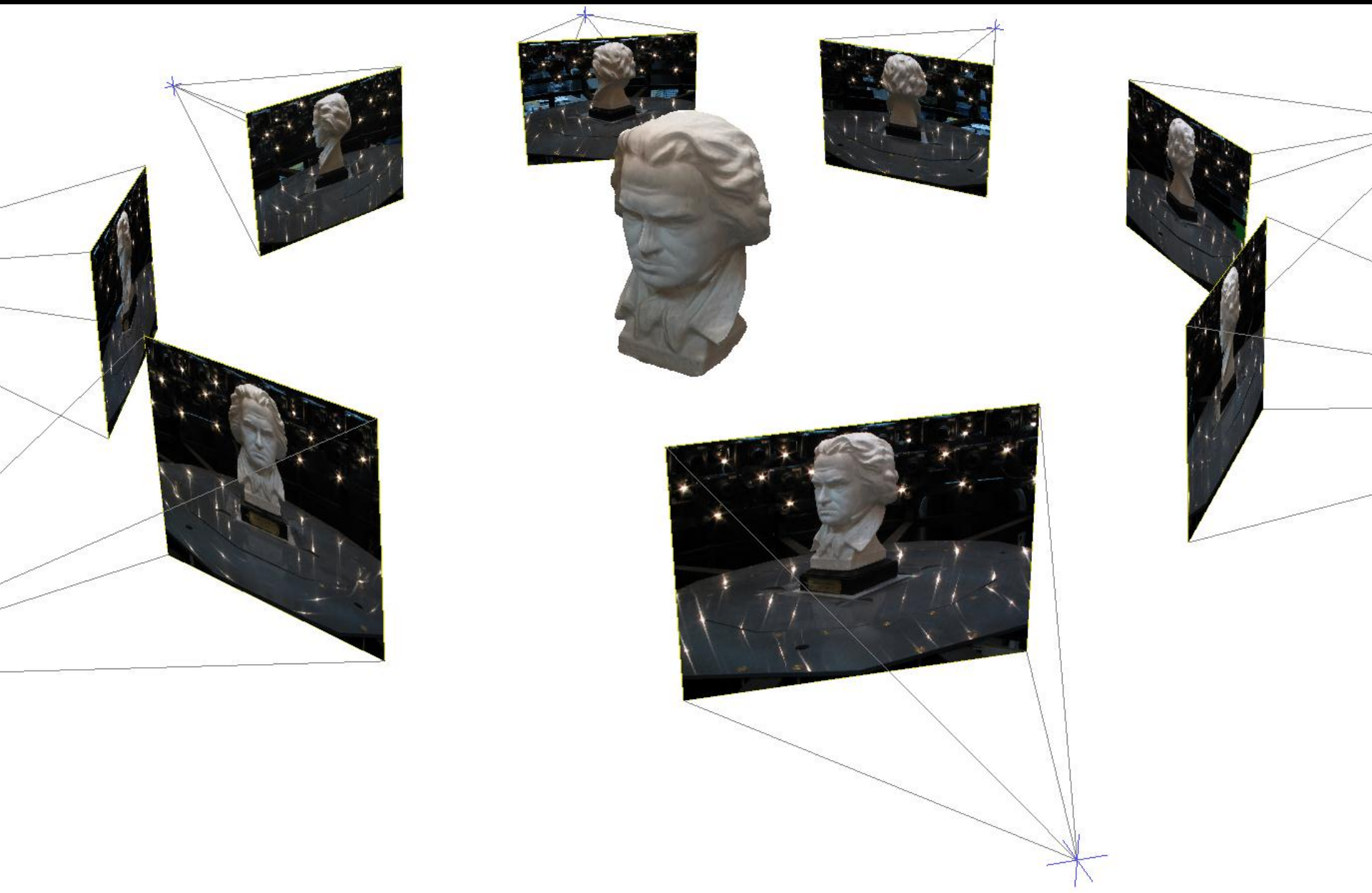
convex

- 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

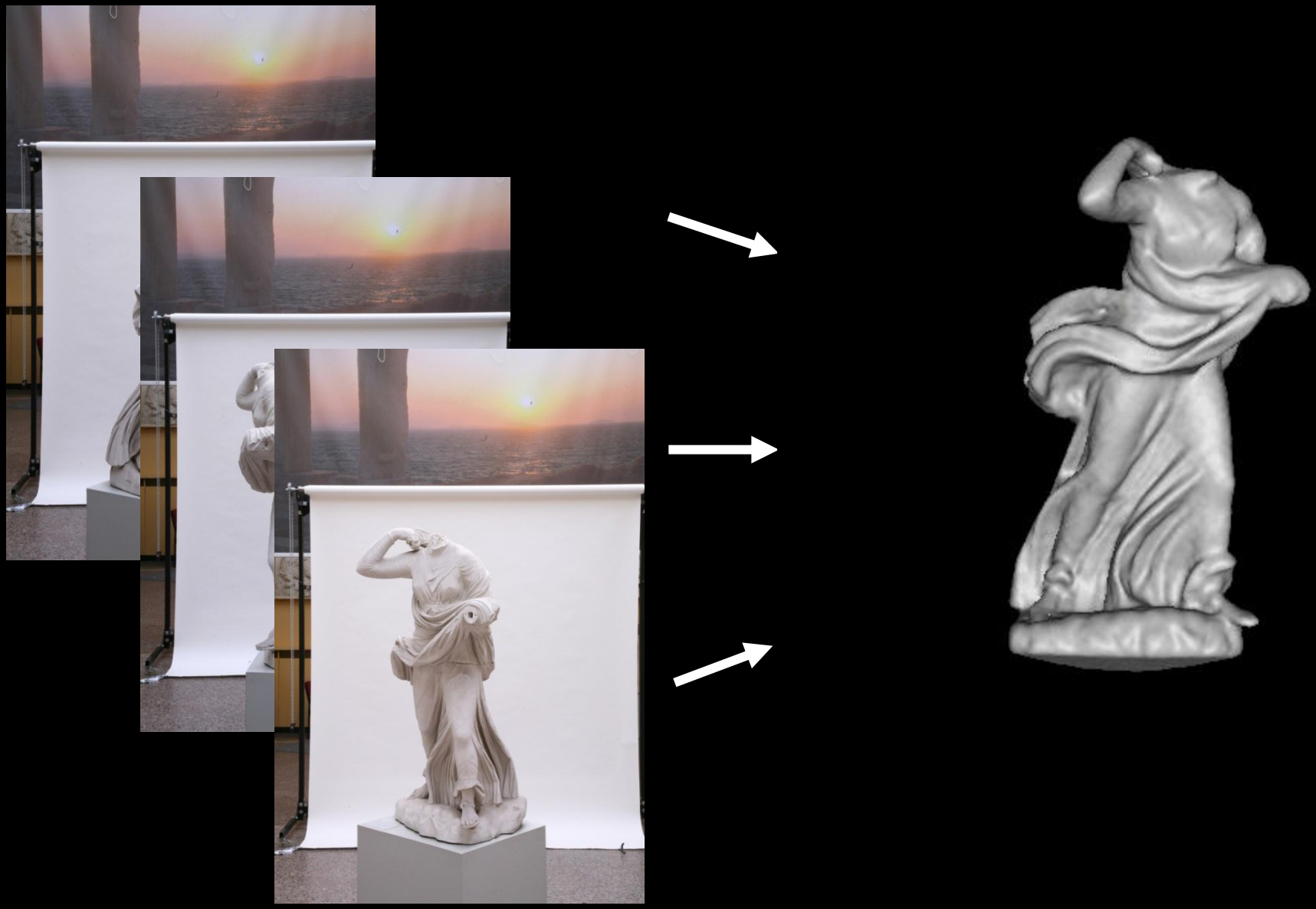
This Course covers

- 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

Example: 3D Reconstruction



Example: 3D Reconstruction



Kolev, Cremers, ECCV ’08, PAMI 2010

Course Goals

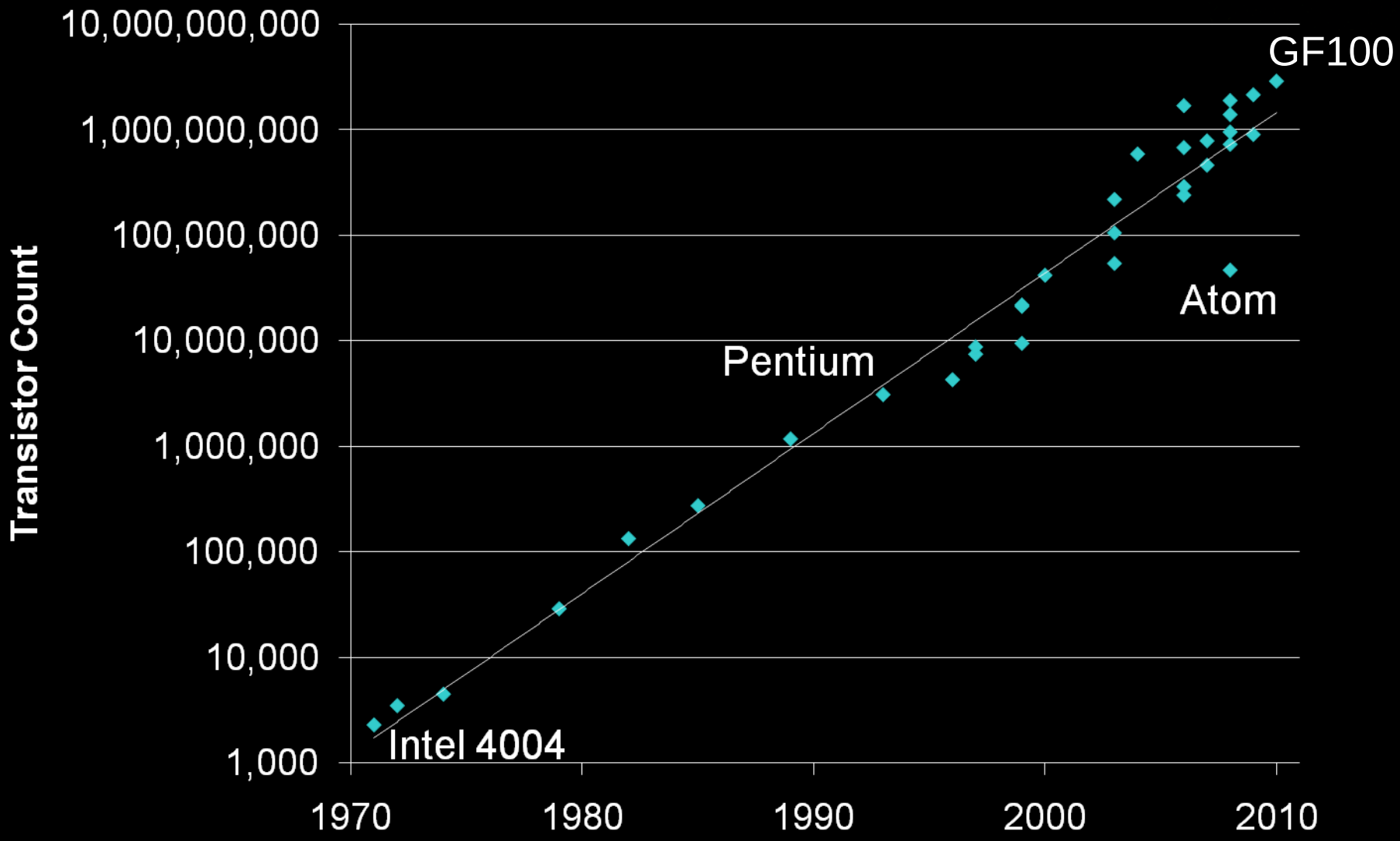
- Learn how to program massively parallel processors and achieve
  - High performance
  - Functionality and maintainability
  - Scalability across future generations
- Acquire technical knowledge required to achieve above goals
  - Principles and patterns of parallel programming
  - Processor architecture features and constraints
  - Programming API, tools and techniques
- Apply this knowledge to implement computer vision algorithms efficiently

Moore's Law (paraphrased)

"The number of transistors on an integrated circuit doubles every two years."

– Gordon E. Moore

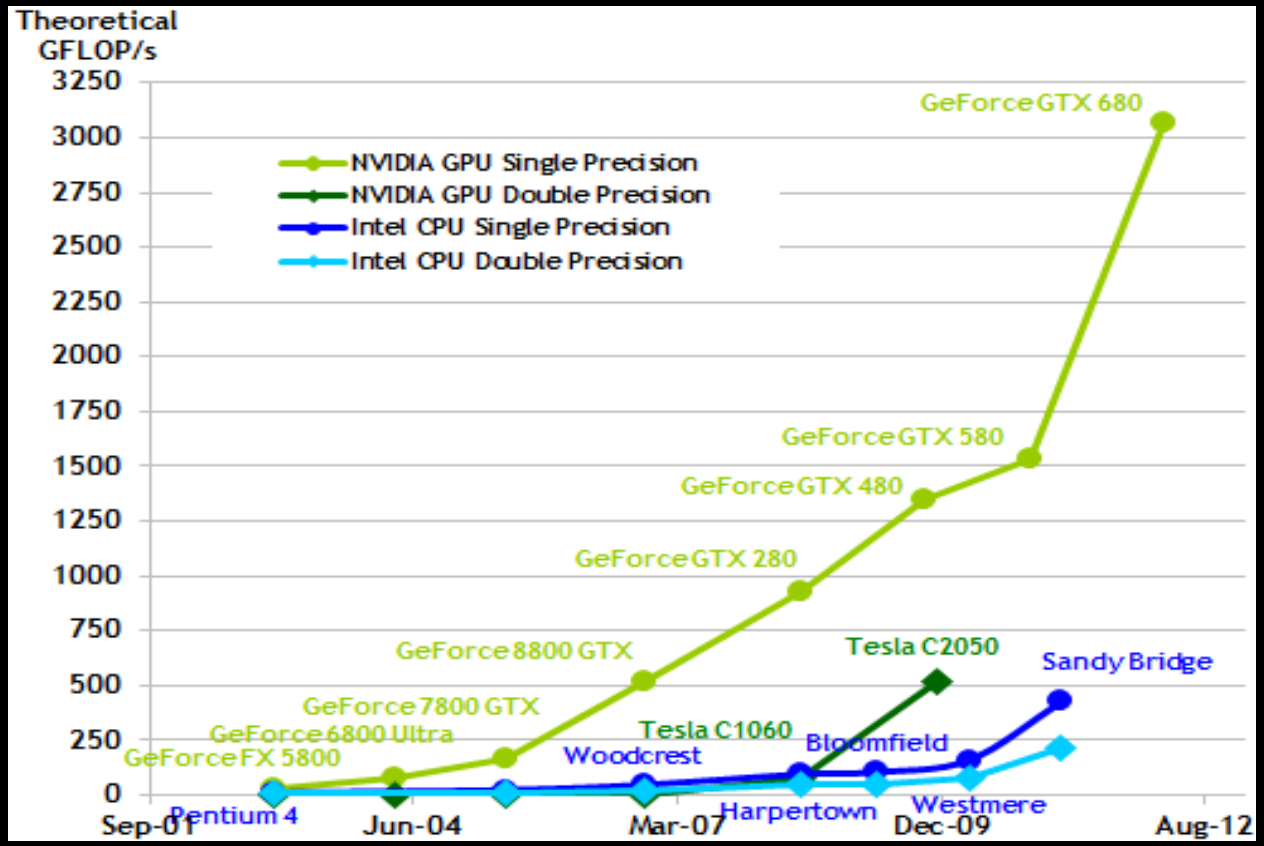
Moore's Law (Visualized)



Data credit: Wikipedia

Why Massively Parallel Processing?

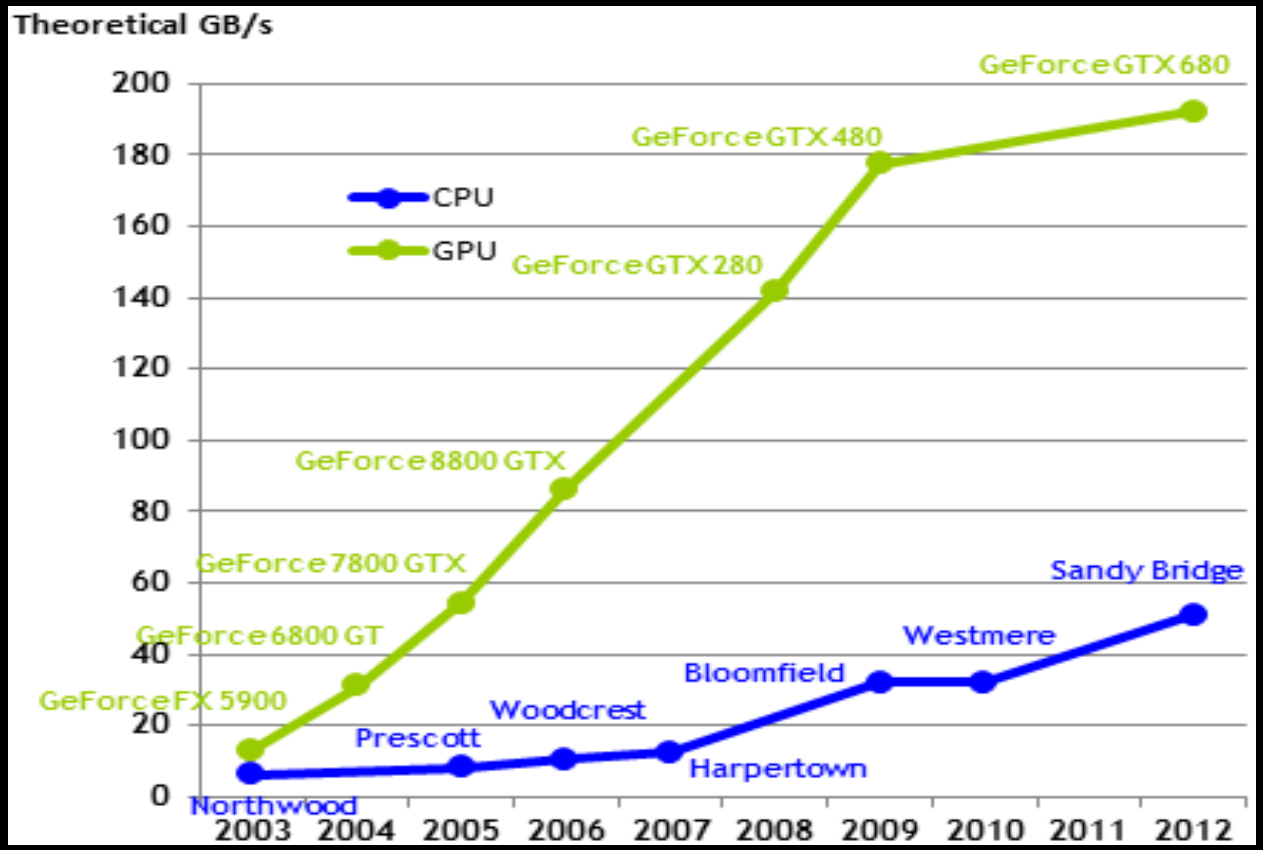
- A quiet revolution and potential build-up
  - Computation: TFLOPs vs. 100 GFLOPs



  - GPU in every PC – massive volume & potential impact

Why Massively Parallel Processing?

- A quiet revolution and potential build-up
  - Bandwidth: ~5x



  - GPU in every PC – massive volume & potential 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

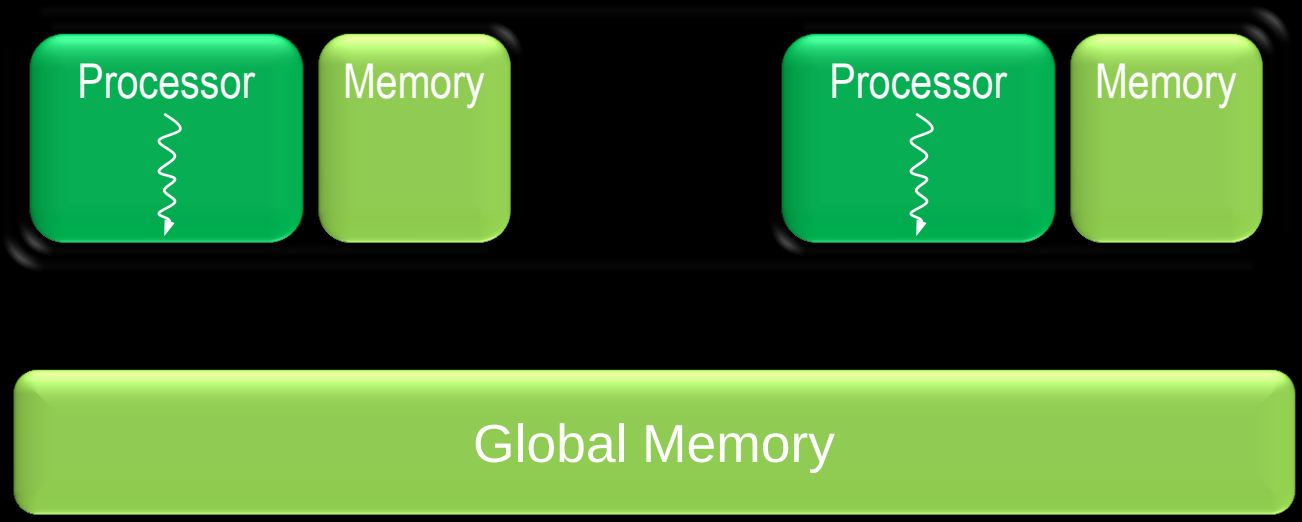
How to Use Transistors?

- Instruction-level parallelism
  - out-of-order execution, speculation, …
  - vanishing opportunities in power-constrained world
- Data-level parallelism
  - vector units, SIMD execution, …
  - increasing … SSE, AVX, Cell SPE, Clearspeed, GPU
- Thread-level parallelism
  - increasing … multithreading, multicore, manycore
  - Intel Core2, AMD Phenom, Sun Niagara, STI Cell, NVIDIA Fermi, …

The “New” Moore’s Law

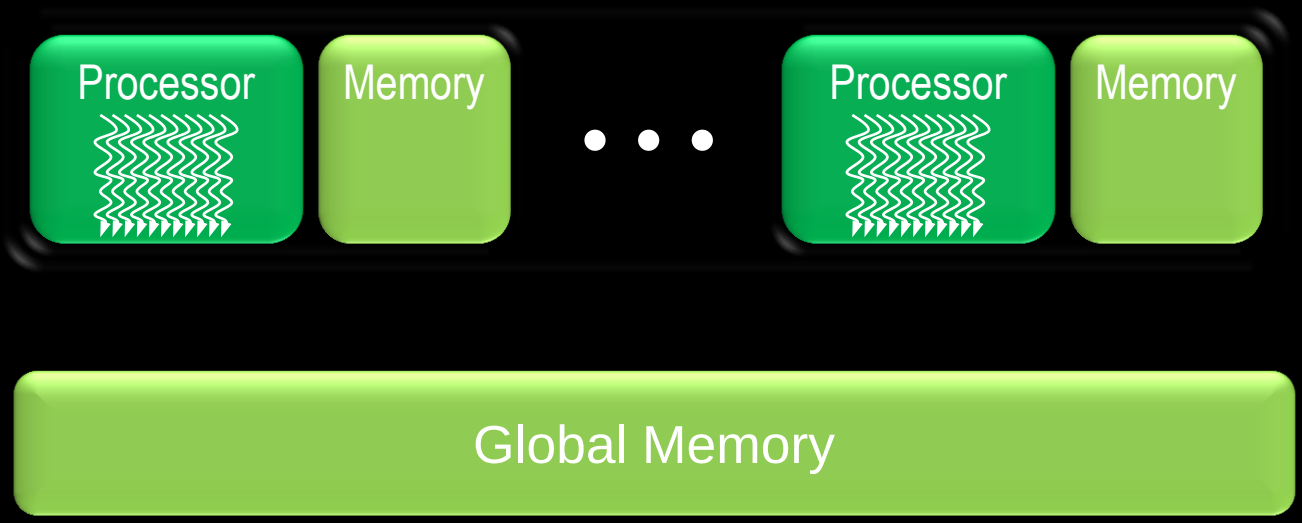
- Computers no longer get faster, just wider
- You *must* re-think your algorithms to be parallel !
- Data-parallel computing is most scalable solution
  - Otherwise: refactor code for ~~2 cores~~ ~~4 cores~~ ~~8 cores~~ 16 cores…
  - You will always have more data than cores – build the computation around the data

Generic Multicore Chip



- Handful of processors each supporting ~1 hardware thread
- On-chip memory near processors (cache, RAM, or both)
- Shared global memory space (external DRAM)

Generic Manycore Chip



- Many processors each supporting many hardware threads
- On-chip memory near processors (cache, RAM, or both)
- Shared global memory space (external DRAM)

Enter the GPU

- Massive economies of scale
- Massively parallel

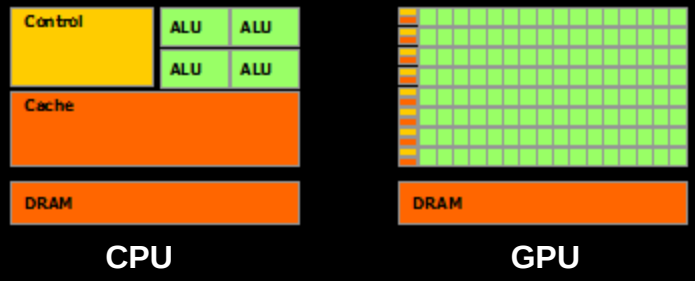
Lessons from Graphics Pipeline

- **Throughput** is paramount
  - must paint every pixel within frame time
  - scalability
- Create, run, & retire **lots of threads** very rapidly
  - measured 14.8 Gthread/s on `increment()` kernel
- Use **multithreading** to hide latency
  - 1 stalled thread is OK if 100 are ready to run

Why is this different from a CPU?

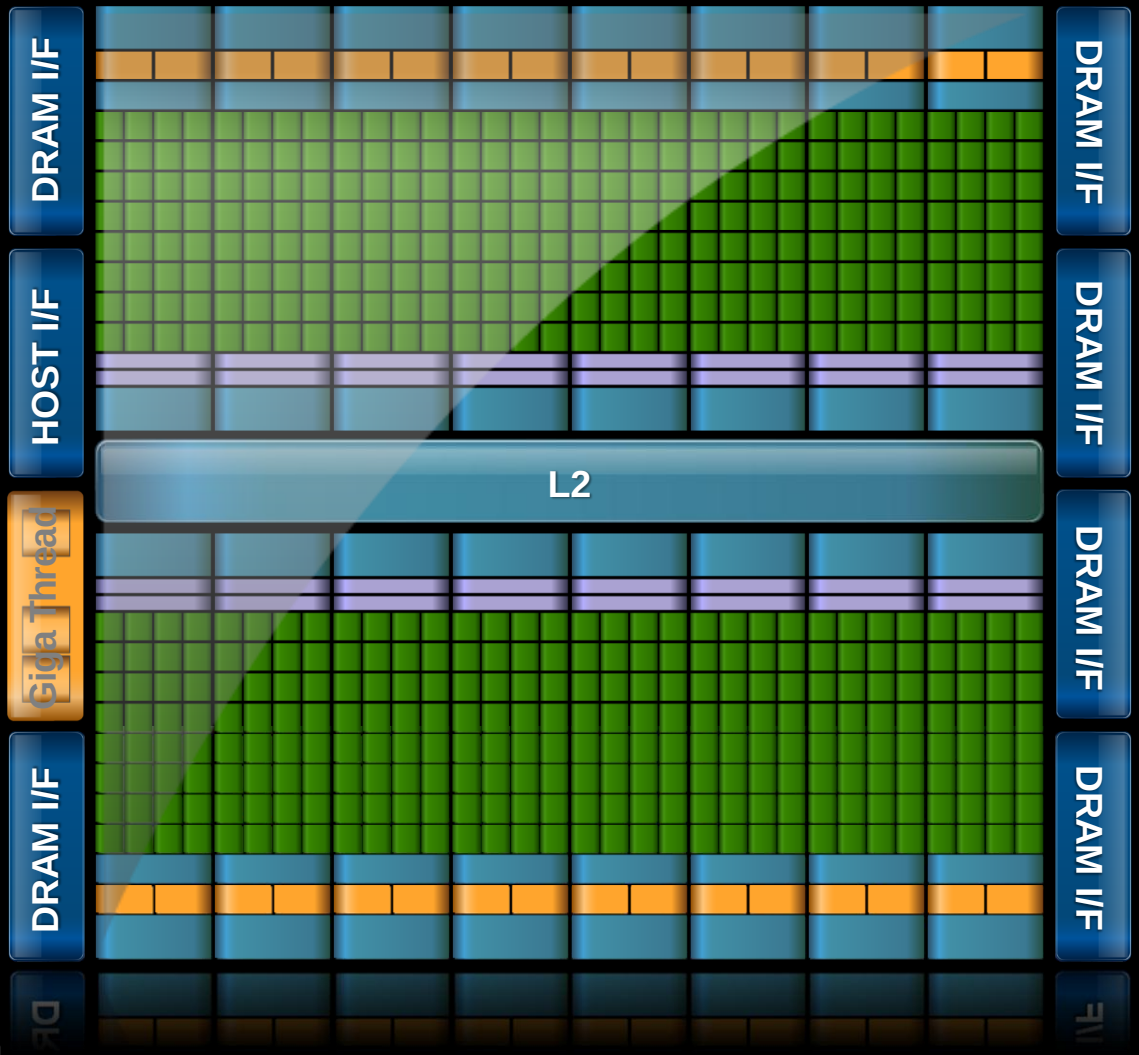
- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency => skip the big caches
  - share control logic across many threads

GPU devotes more transistors to data processing



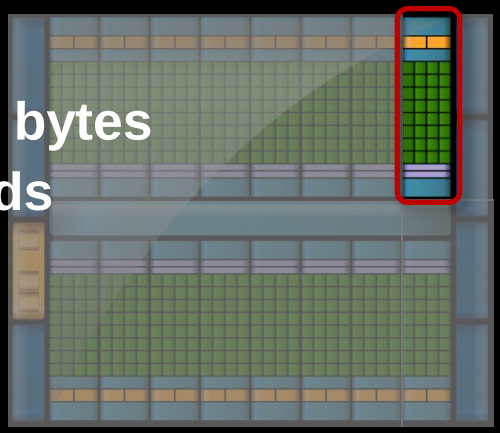
NVIDIA GPU Architecture

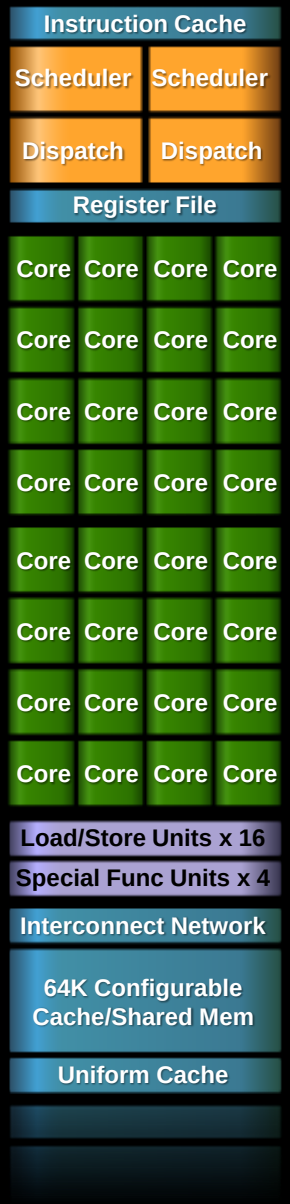
Fermi GF100



SM Multiprocessor

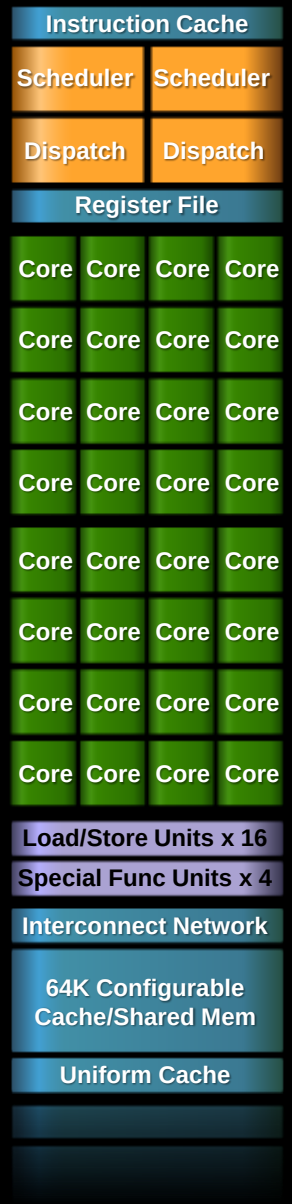
- 32 CUDA Cores per SM (512 total)
- 8x peak FP64 performance
  - 50% of peak FP32 performance
- Direct load/store to memory
  - Usual linear sequence of bytes
  - High bandwidth (Hundreds GB/sec)
- 64KB of fast, on-chip RAM
  - Software or hardware-managed
  - Shared amongst CUDA cores
  - Enables thread communication





Key Architectural Ideas

- **SIMT (**Single Instruction Multiple Thread**) execution**
  - **threads run in groups of 32 called warps**
  - **threads in a warp share instruction unit (IU)**
  - **HW automatically handles divergence**
- **Hardware multithreading**
  - **HW resource allocation & thread scheduling**
  - **HW relies on threads to hide latency**
- **Threads have all resources needed to run**
  - **any warp not waiting for something can run**
  - **context switching is (basically) free**



Enter CUDA
("Compute Unified Device Architecture")

- Scalable parallel programming model
- Minimal extensions to familiar C/C++ environment
- Heterogeneous serial-parallel computing

CUDA: Scalable parallel programming

- Augment C/C++ with minimalist abstractions
  - let programmers focus on parallel algorithms
  - *not* mechanics of a parallel programming language
- Provide straightforward mapping onto hardware
  - good fit to GPU architecture
  - maps well to multi-core CPUs too
- Scale to 100s of cores & 10,000s of parallel threads
  - GPU threads are lightweight — create / switch is free
  - GPU needs 1000s of threads for full utilization

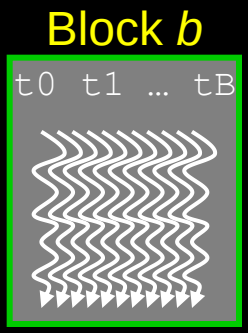
Key Parallel Abstractions in CUDA

- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads

Hierarchy of concurrent threads

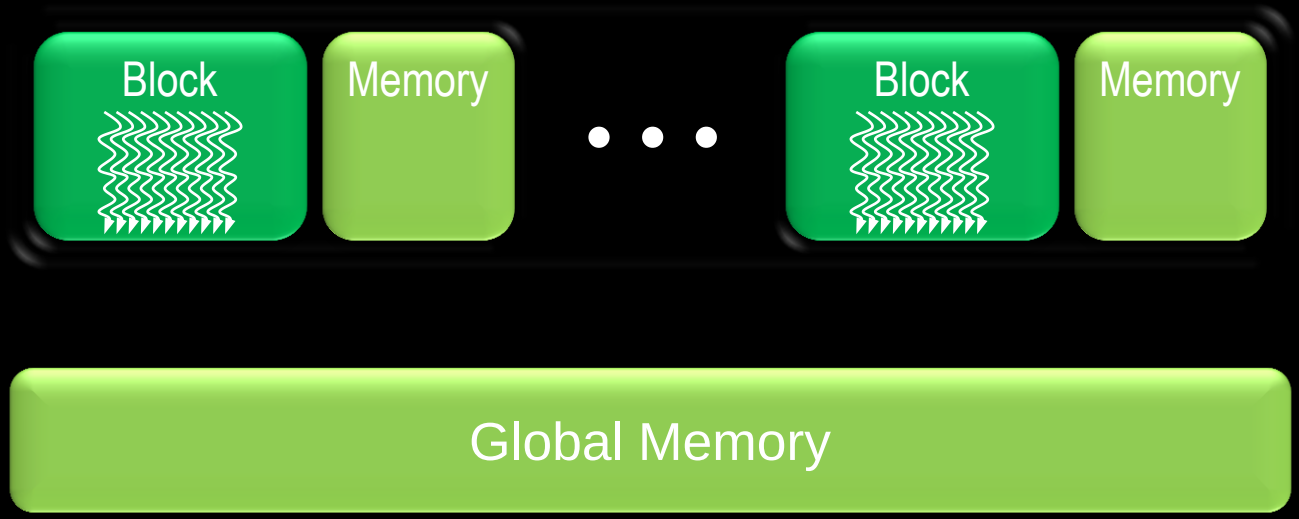
- Parallel kernels composed of many threads
  - all threads execute the same sequential program

- Threads are grouped into thread blocks
  - threads in the same block can cooperate



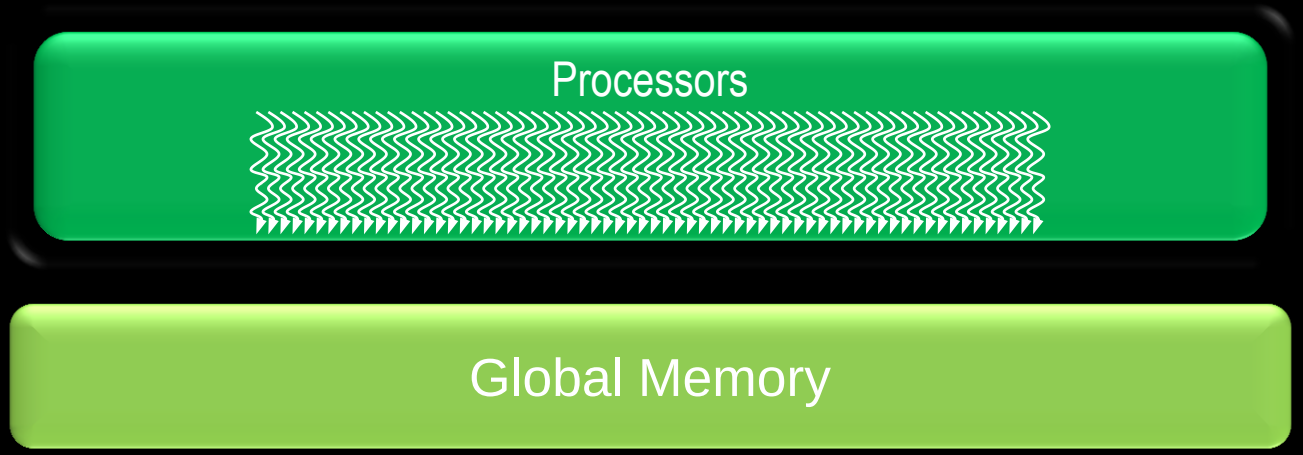
- Threads/blocks have unique IDs

CUDA Model of Parallelism



- CUDA virtualizes the physical hardware
  - thread is a virtualized scalar processor (registers, PC, state)
  - block is a virtualized multiprocessor (threads, shared mem.)
- Scheduled onto physical hardware without pre-emption
  - threads/blocks launch & run to completion
  - blocks should be independent

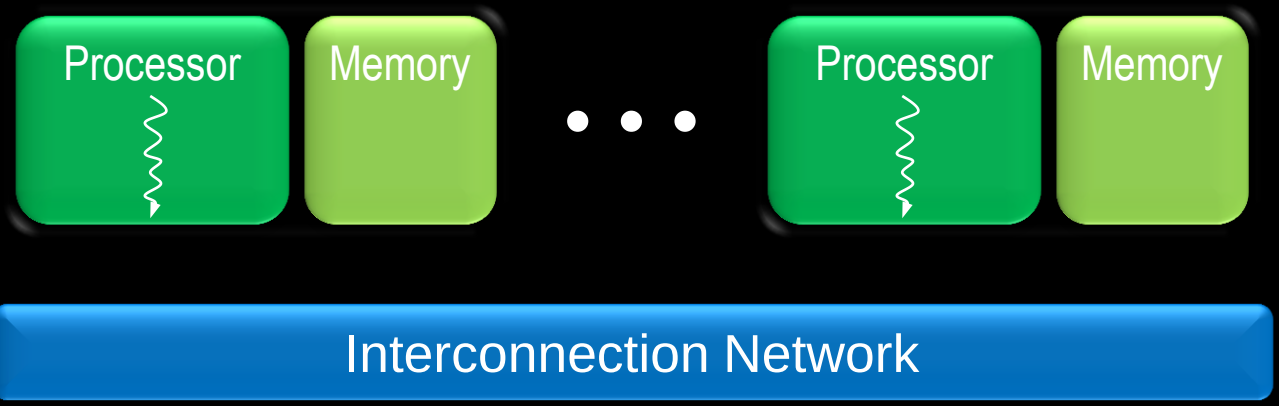
NOT: Flat Multiprocessor



- Global synchronization isn't cheap
- Global memory access times are expensive

- cf. PRAM (Parallel Random Access Machine) model

NOT: Distributed Processors



- Distributed computing is a different setting

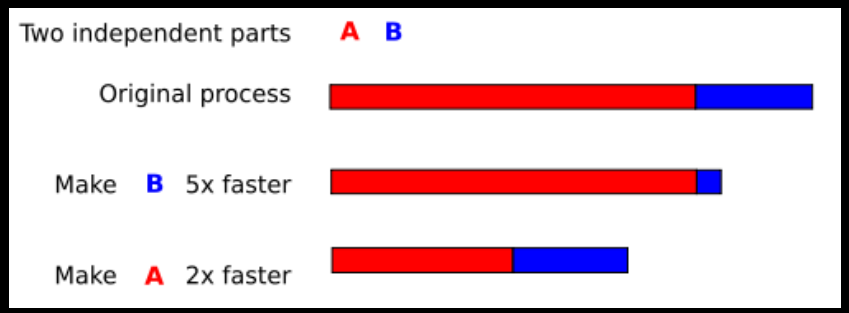
- cf. BSP (Bulk Synchronous Parallel) model, MPI

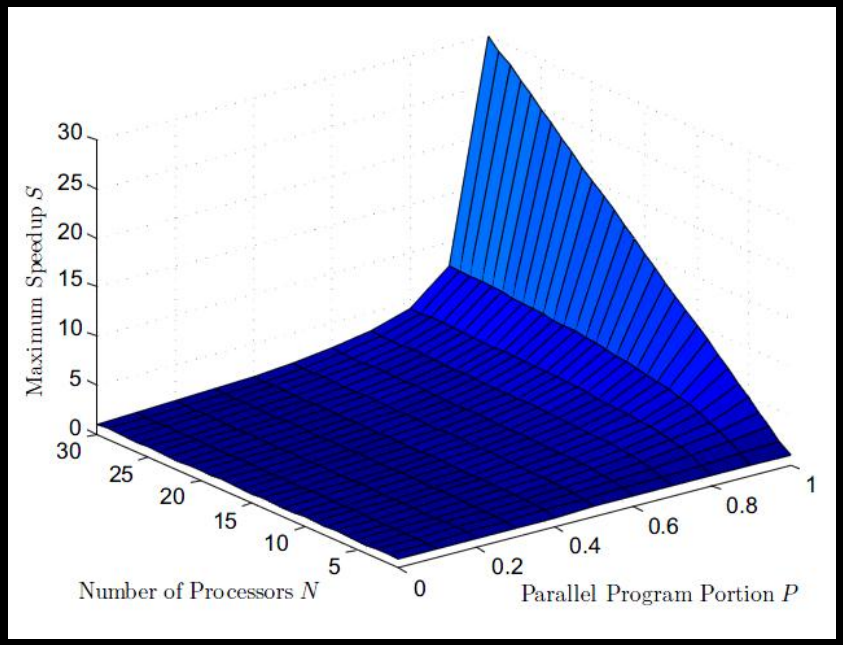
Sequential vs. Parallel

- **Speedup** $S = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}$

- **Amdahl's law**

$$S_{max} = \frac{1}{(1-P) + P/N}$$





Example: parallel portion P=90% → maximum Speedup S_{max}=10

Outline of CUDA Basics

- Basic Kernels and Execution on GPU
- Basic Memory Management
- Coordinating CPU and GPU Execution

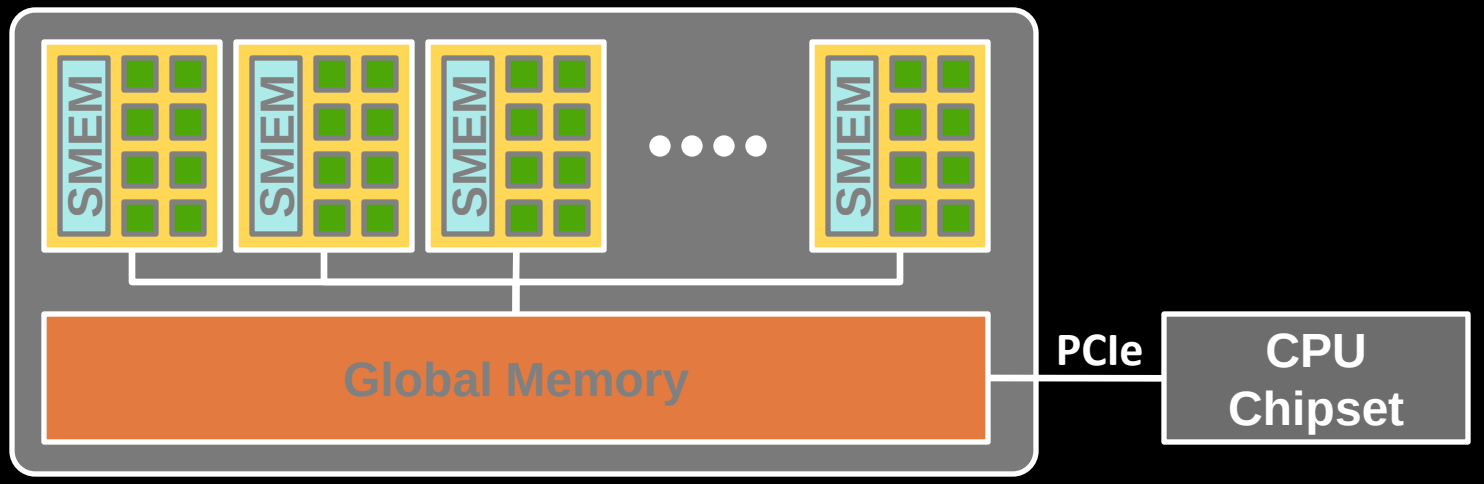
- See the Programming Guide for the full API

BASIC KERNELS AND EXECUTION ON GPU

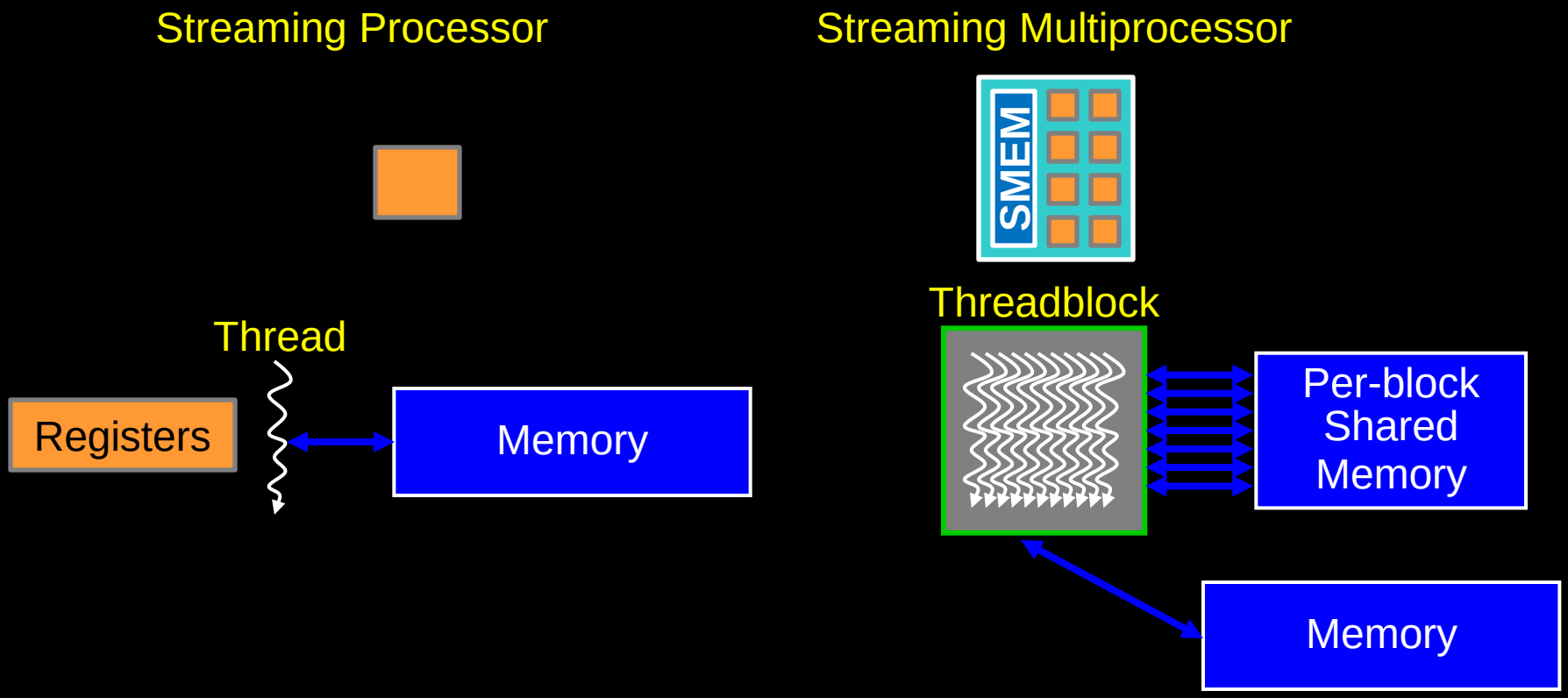
CUDA Programming Model

- Parallel code (kernel) is launched and executed on a device by many threads
- Launches are hierarchical
  - Threads are grouped into blocks
  - Blocks are grouped into grids
- Familiar serial code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables

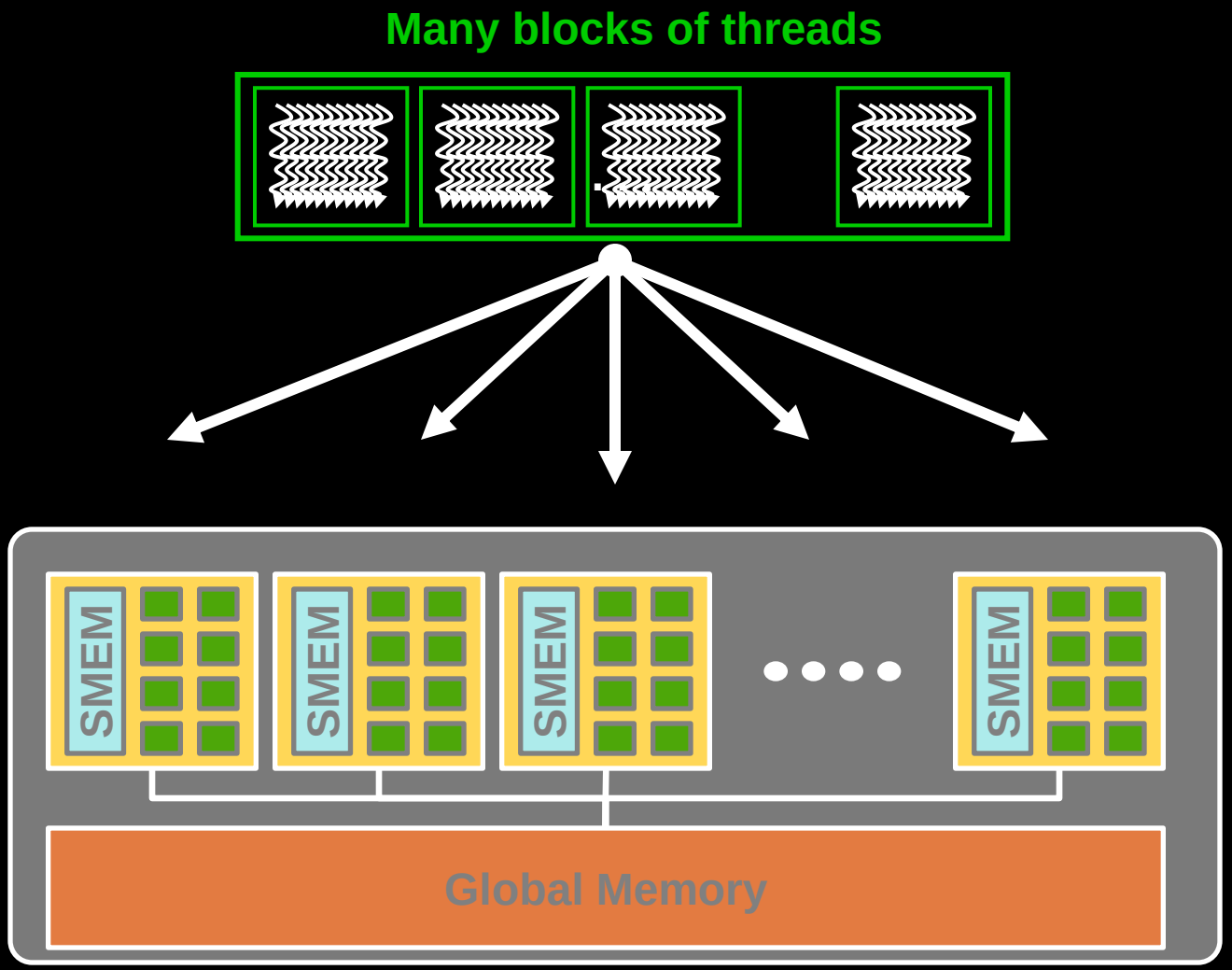
High Level View



Blocks of threads run on an SM



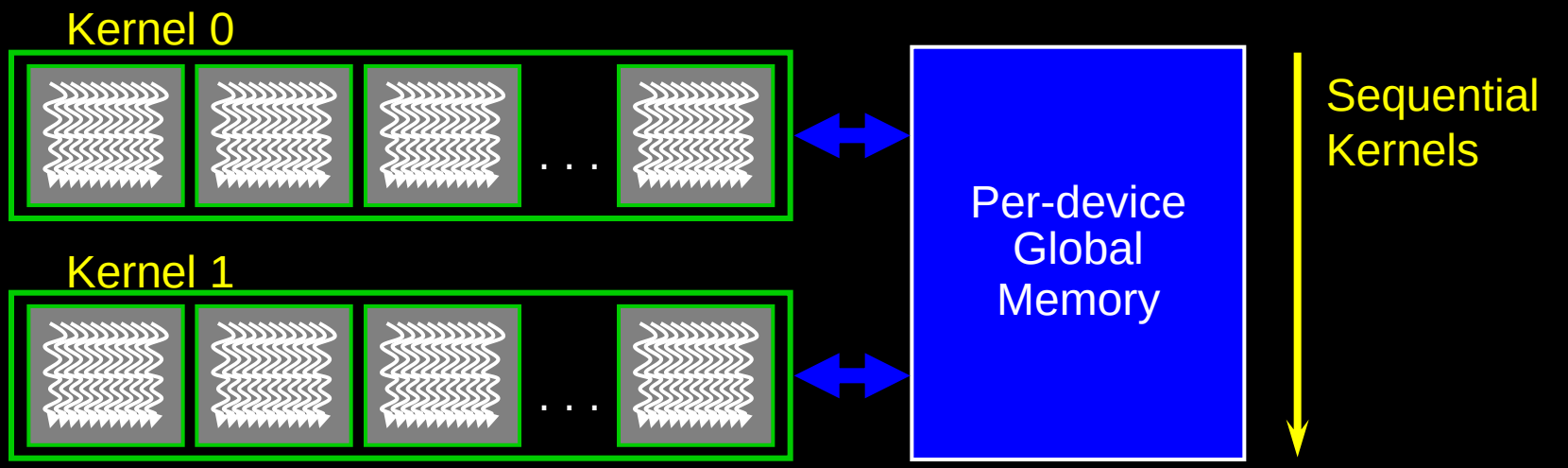
Whole grid runs on GPU



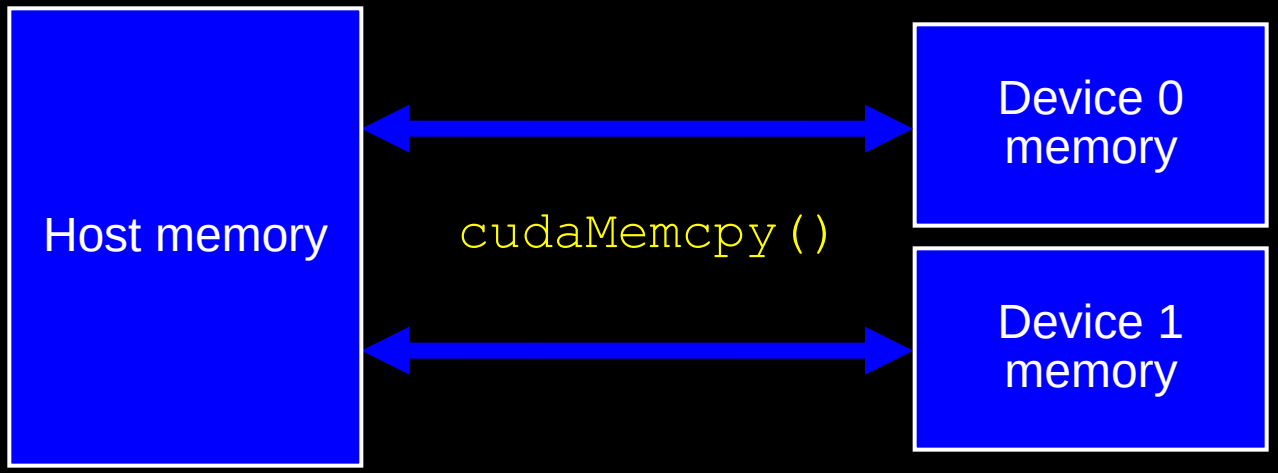
Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory

Memory Model



Memory Model



Example: Vector Addition Kernel

Device Code

```
// 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: 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];
}
```

Host Code

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

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);
```

Kernel Variations and Output

```
__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

Code executed on GPU

- C/C++ with some restrictions:
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
  - No recursion
  - No dynamic polymorphism
- Must be declared with a qualifier:
  - __global__ : launched by CPU,
    cannot be called from GPU must return void
  - __device__ : called from other GPU functions,
    cannot be called by the CPU
  - __host__ : can be called by CPU
  - __host__ and __device__ qualifiers can be combined
    - sample use: overloading operators

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
  - 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
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- Non-blocking copies are also available

Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
```

Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
```

Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes,
cudaMemcpyDeviceToHost );
```

Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Example: Shuffling Data

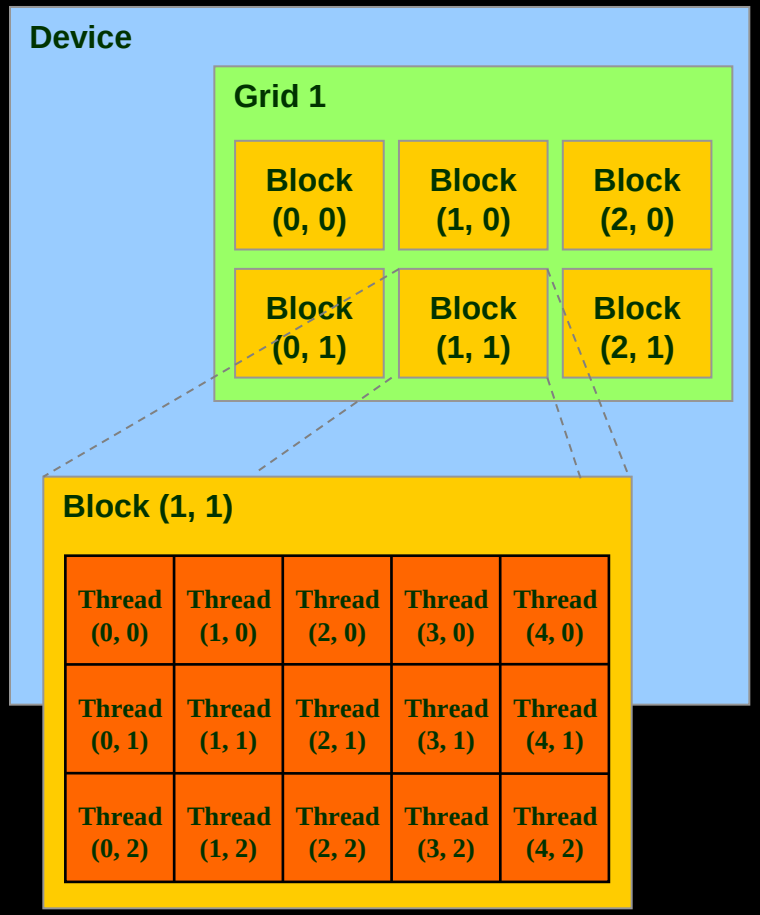
```
// Reorder values based on keys
// Each thread moves one element
__global__ void shuffle(int* prev_array, int*
    new_array, int* indices)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    new_array[i] = prev_array[indices[i]];
}
```

Host Code

```
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    shuffle<<< N/256, 256>>>(d_old, d_new, d_ind);
}
```

IDs and Dimensions

- **Threads:**
  - 3D IDs, unique within a block
- **Blocks:**
  - 2D IDs, unique within a grid
- **Dimensions set at launch**
  - Can be unique for each grid
- **Built-in variables:**
  - threadIdx, blockIdx
  - blockDim, gridDim



Kernel with 2D Indexing

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
```

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
```

```
int main()
{
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x  = dimx / block.x;
    grid.y  = dimy / block.y;

    kernel<<<grid, block>>>( d_a, dimx, dimy );

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int row=0; row<dimy; row++)
    {
        for(int col=0; col<dimx; col++)
            printf("%d ", h_a[row*dimx+col] );
        printf("\n");
    }

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

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
  - 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

CUDA Short Summary

Thread Hierarchy

thread - smallest executable unity
warp - group of 32 threads
block - 2-16 warps or 64 - 512 threads (with shared memory)
grid - consists of several blocks

Keyword extensions for 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