CS 193G

Lecture 2: GPU History & CUDA Programming Basics

Outline of CUDA Basics

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

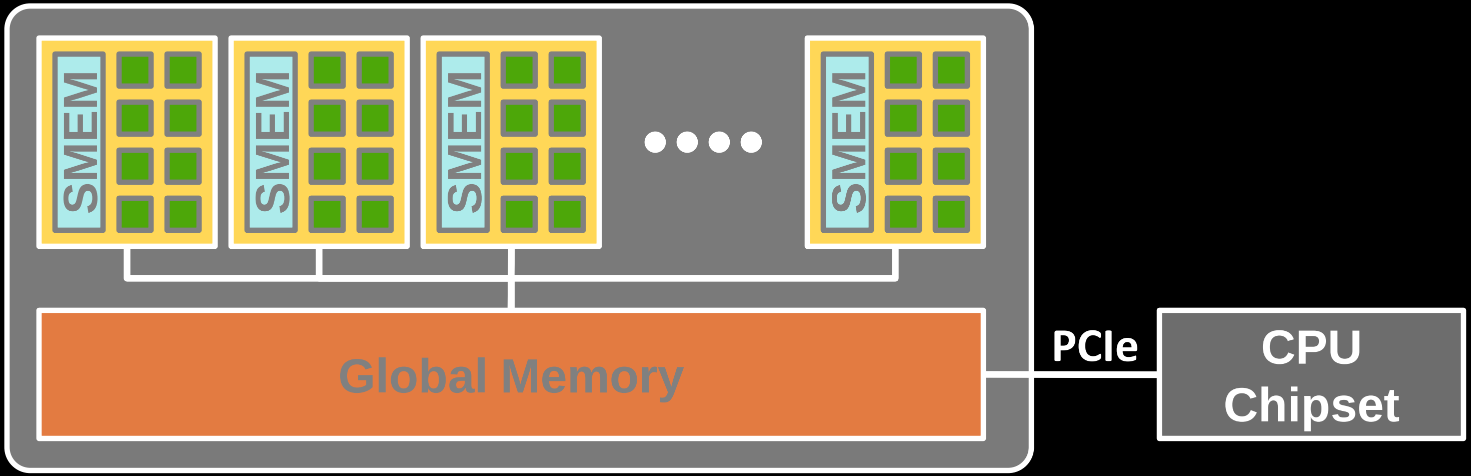
- See the Programming Guide for the full API

BASIC KERNELS AND EXECUTION ON GPU

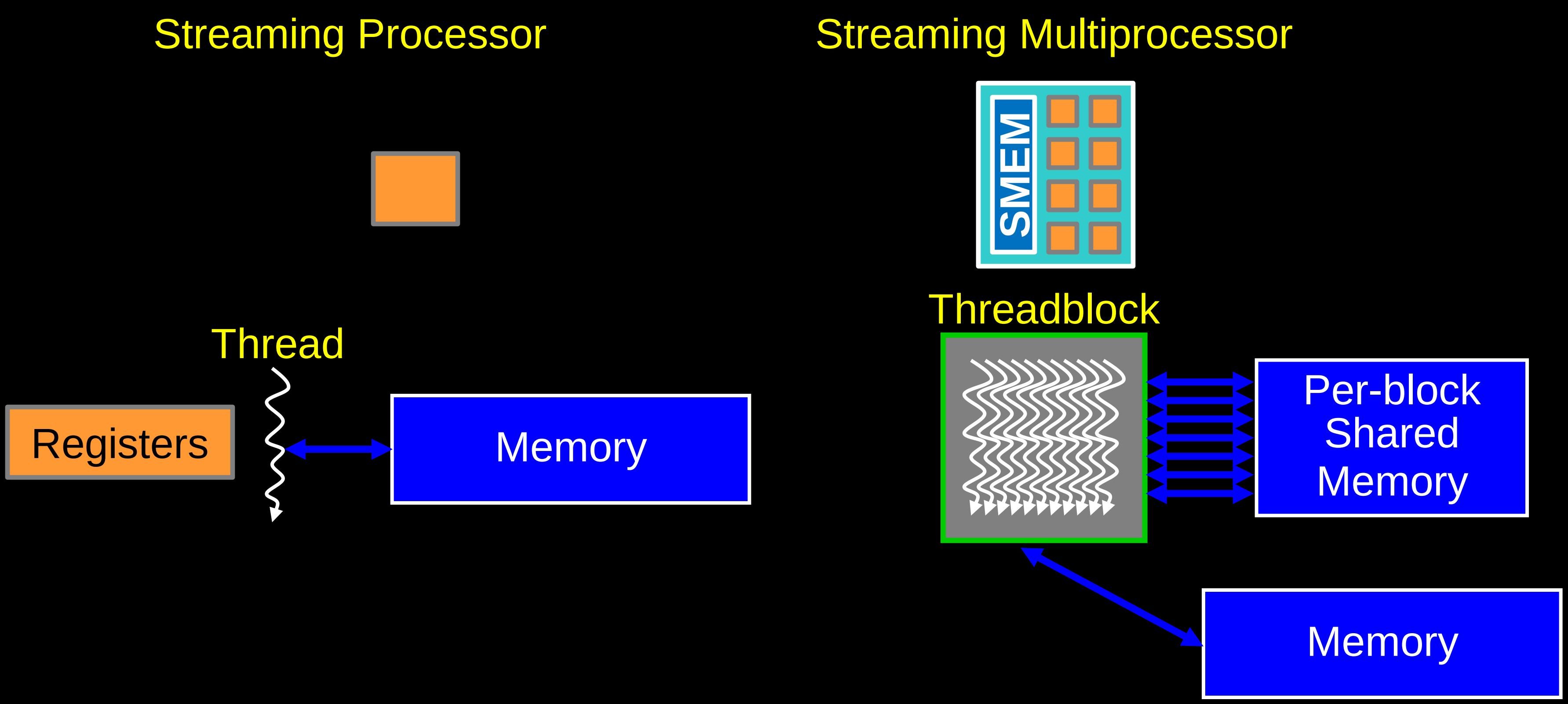
CUDA Programming Model

- 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

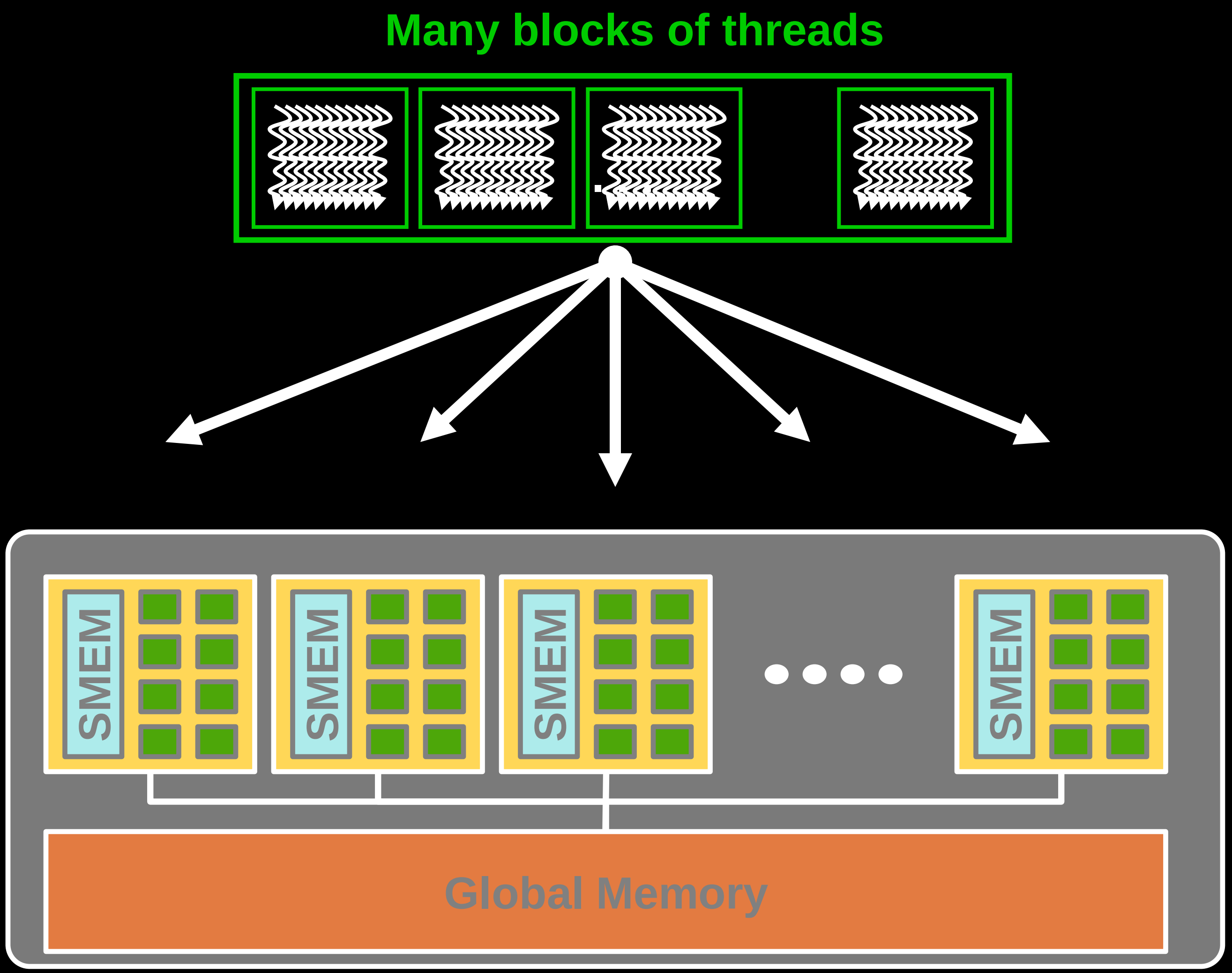
High Level View



Blocks of threads run on an SM



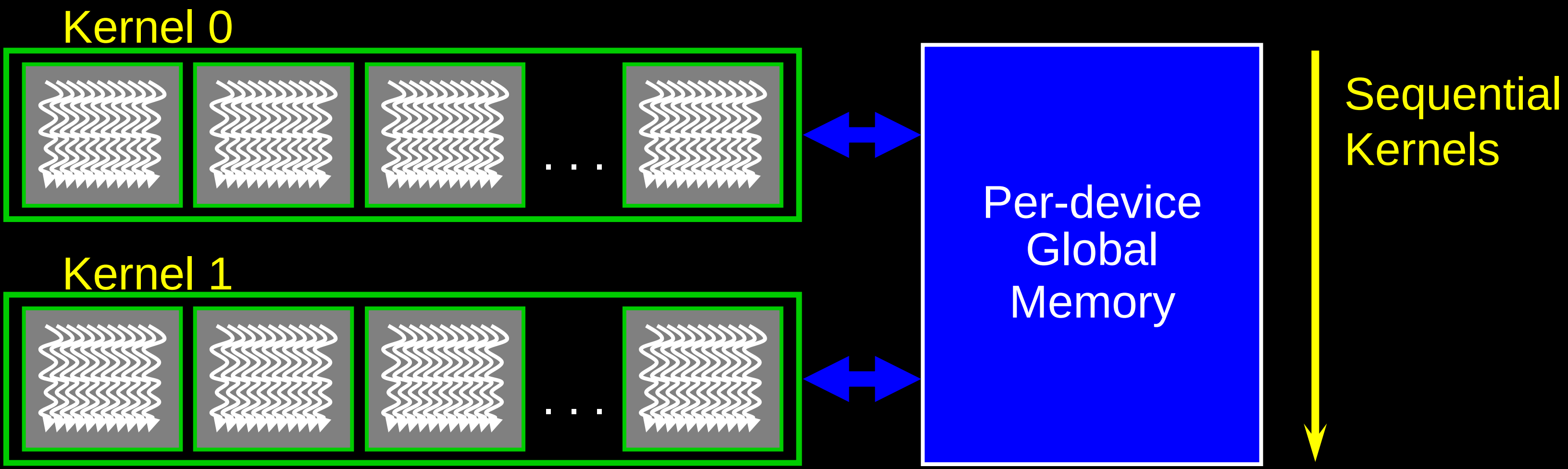
Whole grid runs on GPU



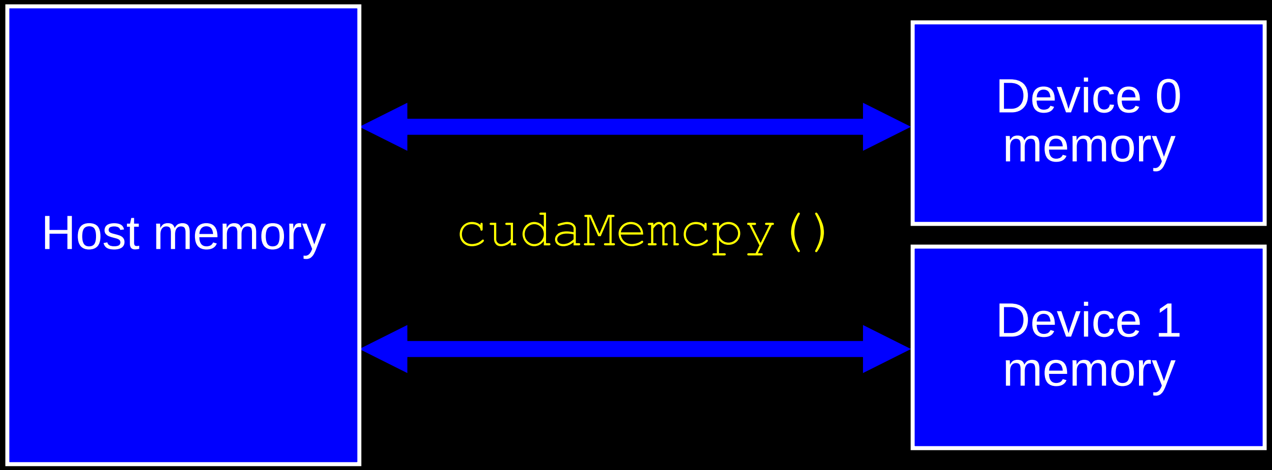
Thread Hierarchy

- 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

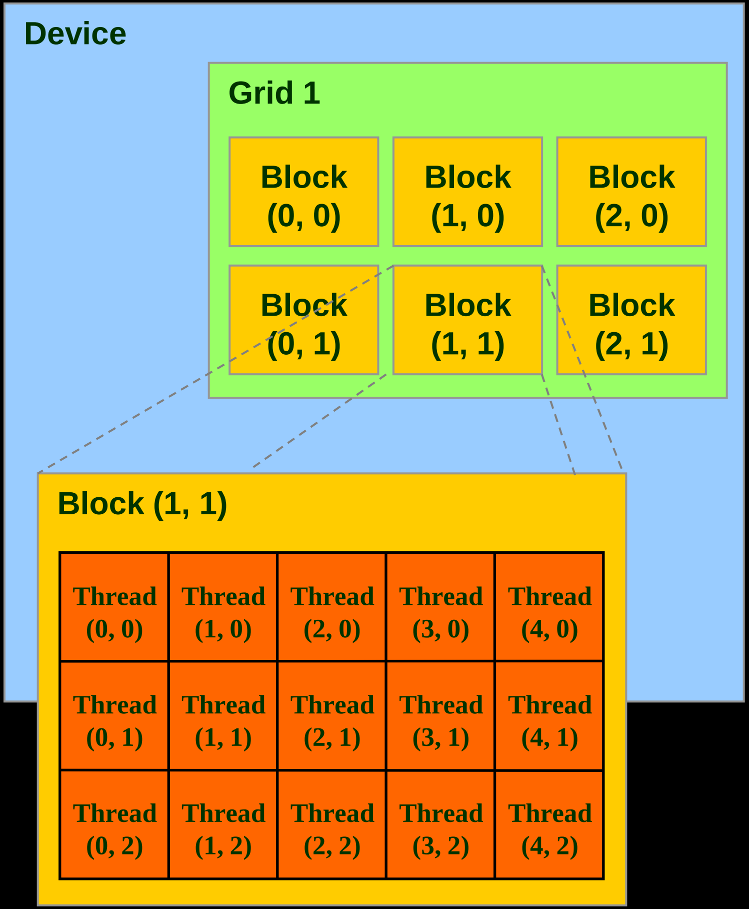
```
// Reorder values based on keys
// Each thread moves one element
__global__ void shuffle(int* prev_array, int*
   new_array, int* indices)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    new_array[i] = prev_array[indices[i]];
}
```

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

Questions?

CS 193G

History of GPUs

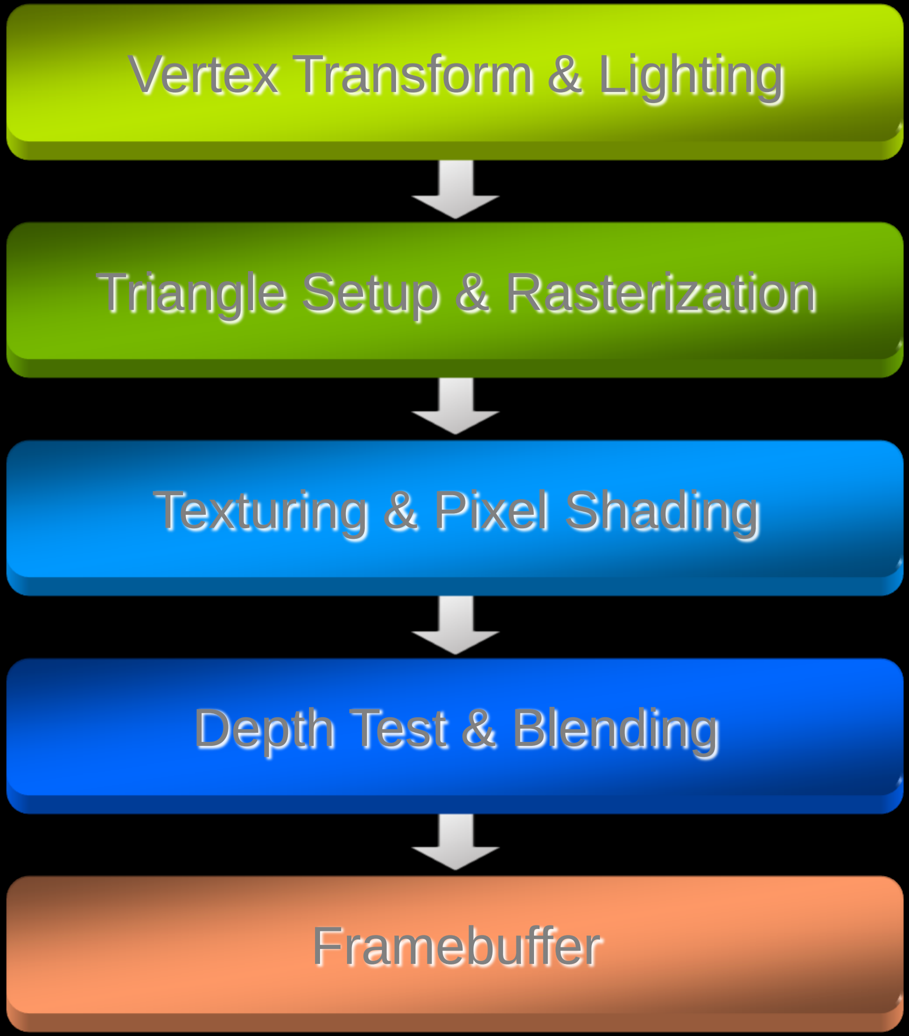
Graphics in a Nutshell

- Make great images
  - intricate shapes
  - complex optical effects
  - seamless motion

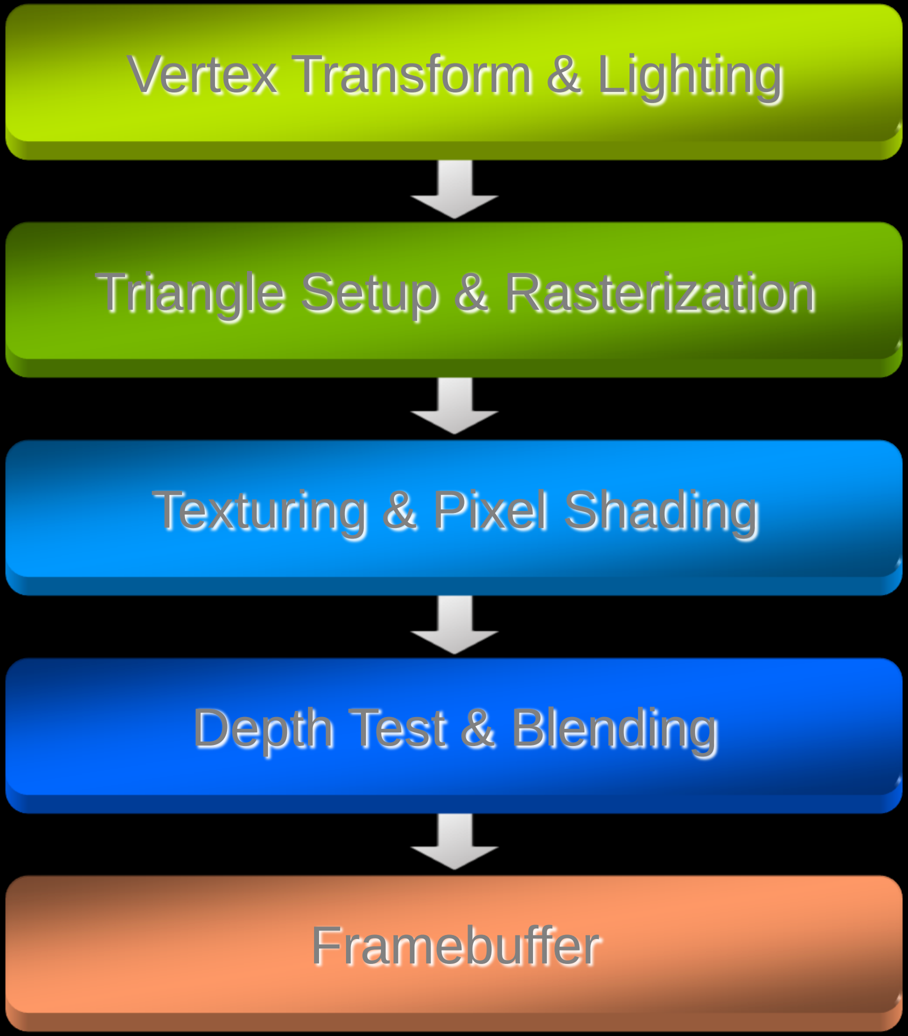
- Make them fast
  - invent clever techniques
  - use every trick imaginable
  - build monster hardware

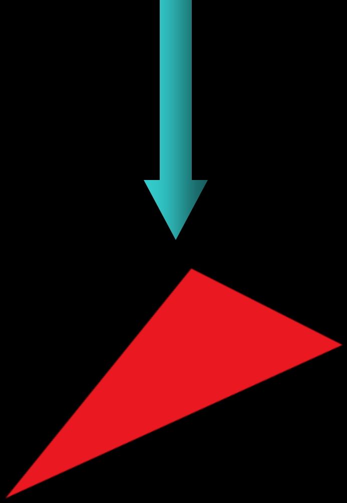
Eugene d'Eon, David Luebke, Eric Enderton
In *Proc. EGSR 2007* and *GPU Gems 3*

The Graphics Pipeline

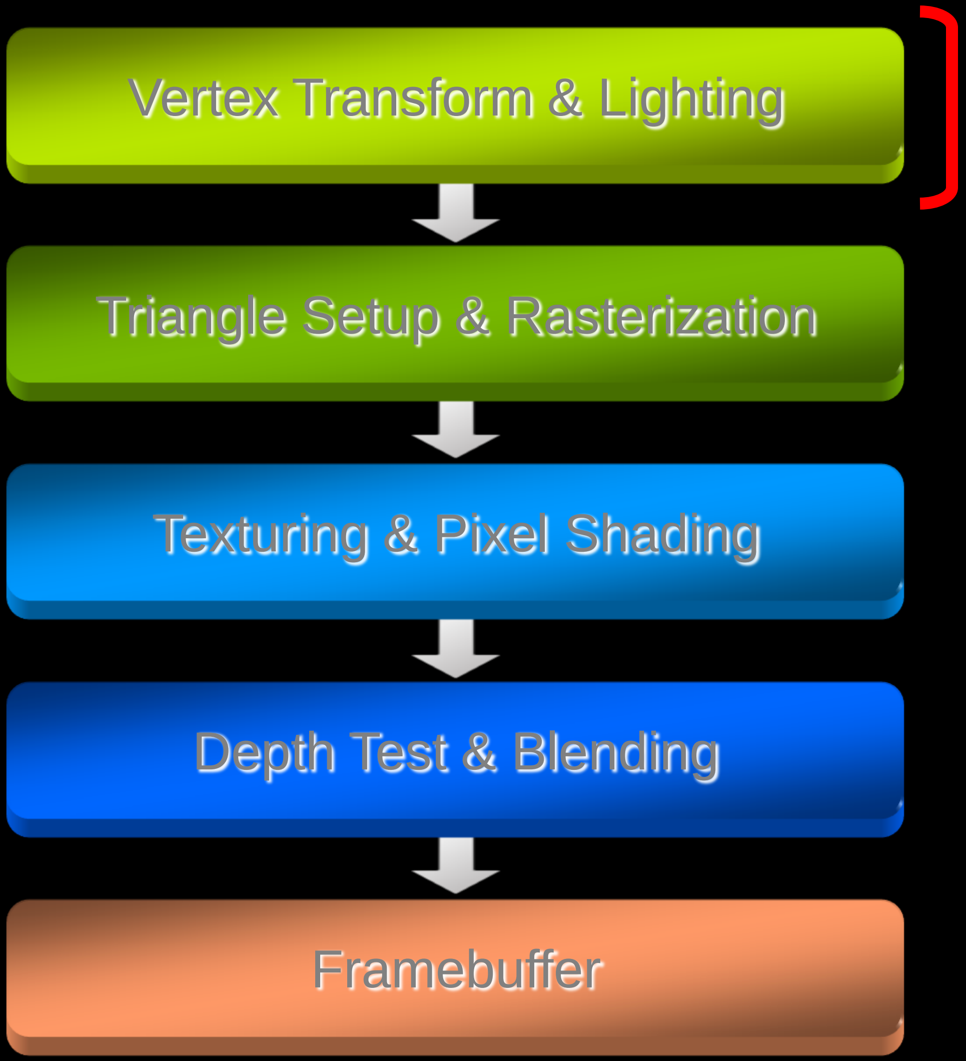

Vertex Transform & Lighting
Triangle Setup & Rasterization
Texturing & Pixel Shading
Depth Test & Blending
Framebuffer

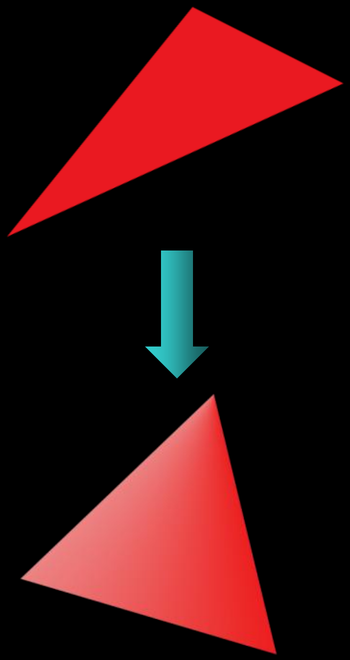
The Graphics Pipeline


Vertex Transform & Lighting
Triangle Setup & Rasterization
Texturing & Pixel Shading
Depth Test & Blending
Framebuffer

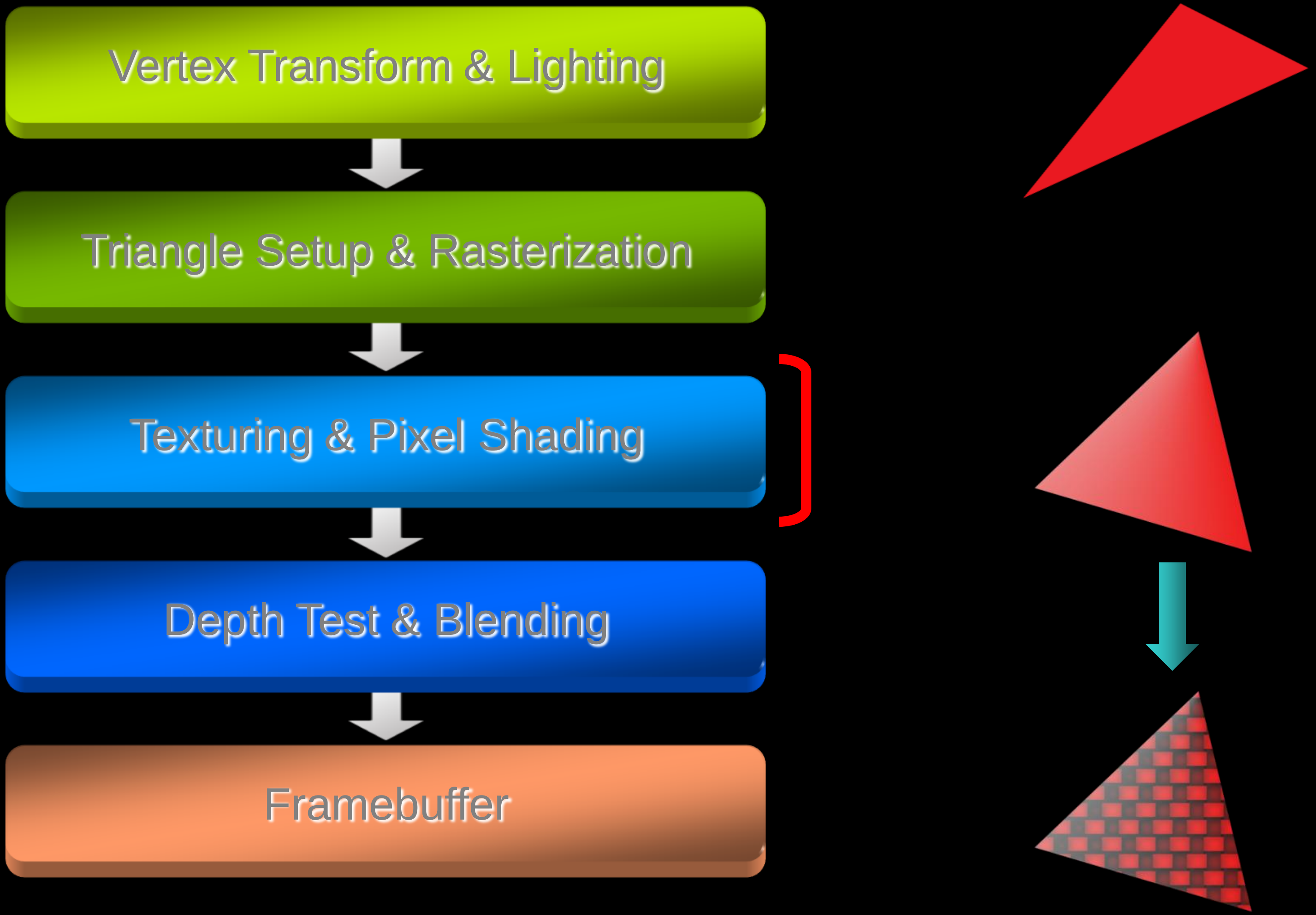


The Graphics Pipeline

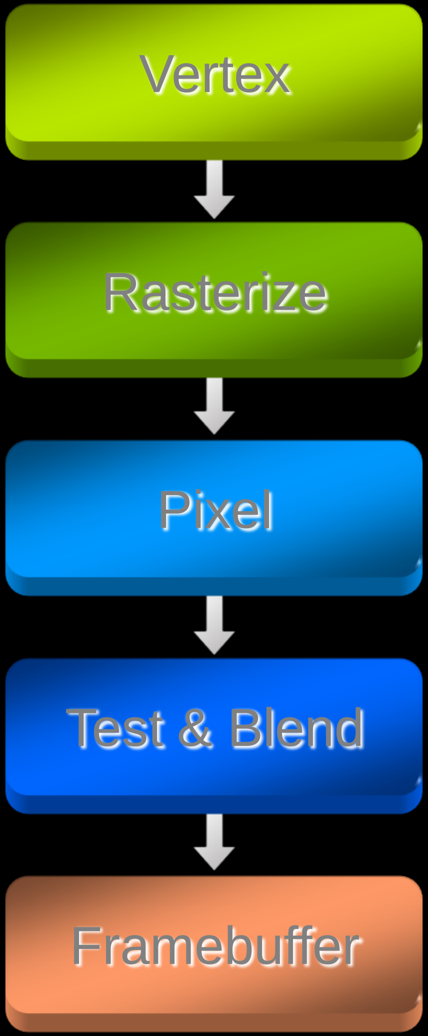

Vertex Transform & Lighting
Triangle Setup & Rasterization
Texturing & Pixel Shading
Depth Test & Blending
Framebuffer



The Graphics Pipeline

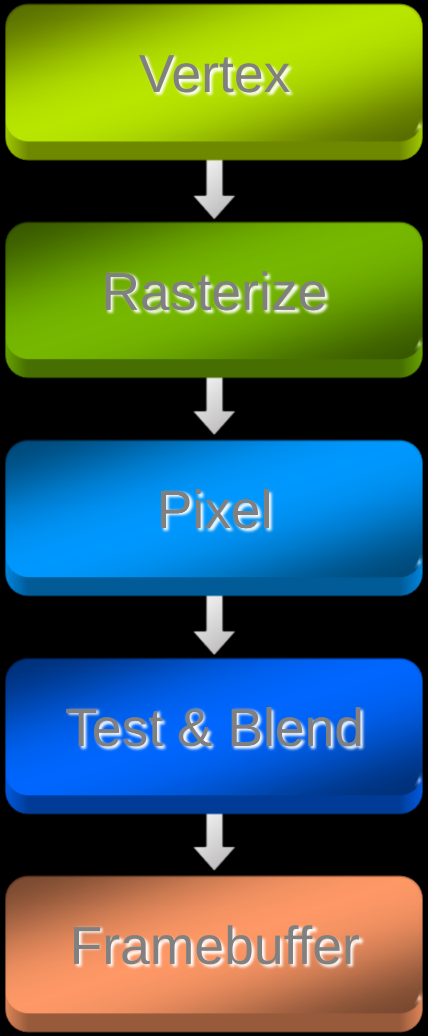

Vertex Transform & Lighting
Triangle Setup & Rasterization
Texturing & Pixel Shading
Depth Test & Blending
Framebuffer

The Graphics Pipeline



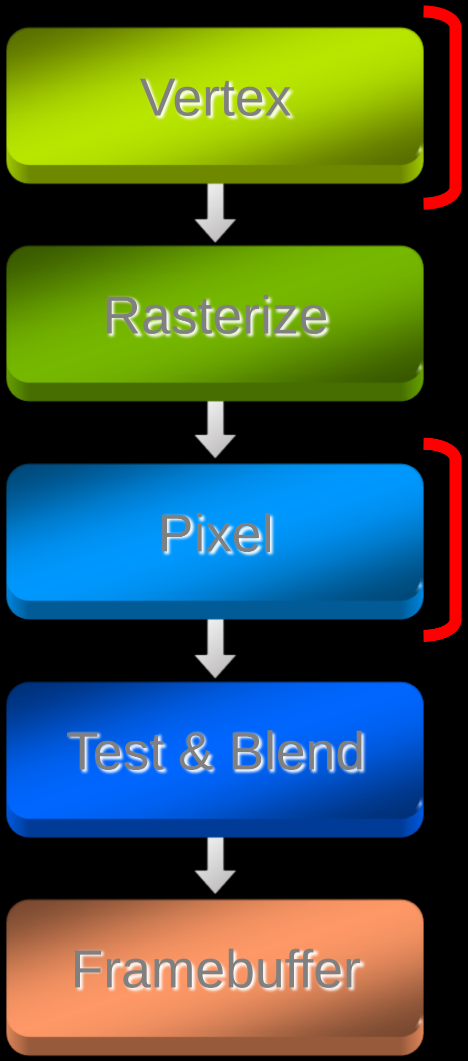
- Key abstraction of real-time graphics
- Hardware used to look like this
- One chip/board per stage
- Fixed data flow through pipeline

The Graphics Pipeline



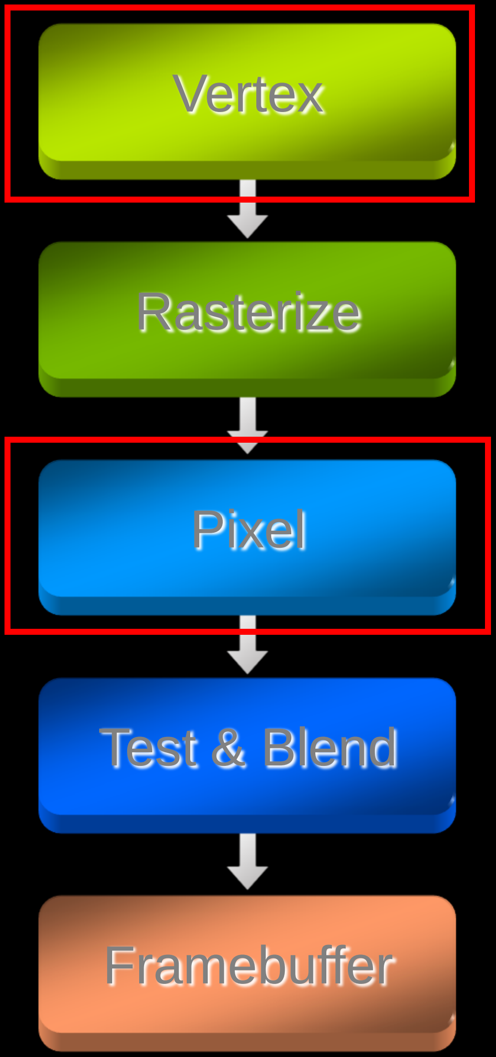
- Everything fixed function, with a certain number of modes
- Number of modes for each stage grew over time
- Hard to optimize HW
- Developers always wanted more flexibility

The Graphics Pipeline



- Remains a key abstraction
- Hardware used to look like this
- Vertex & pixel processing became programmable, new stages added
- GPU architecture increasingly centers around shader execution

The Graphics Pipeline

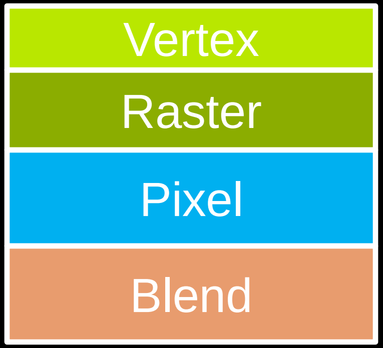


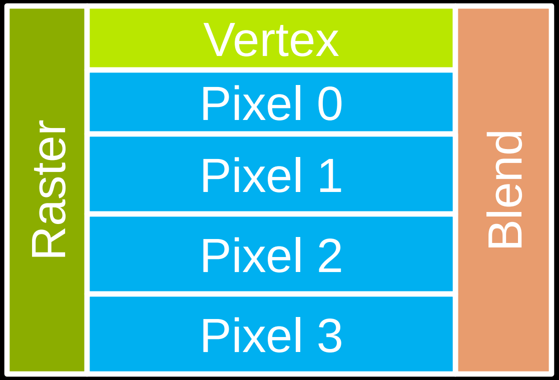
- Exposing a (at first limited) instruction set for some stages
- Limited instructions & instruction types and no control flow at first
- Expanded to full ISA

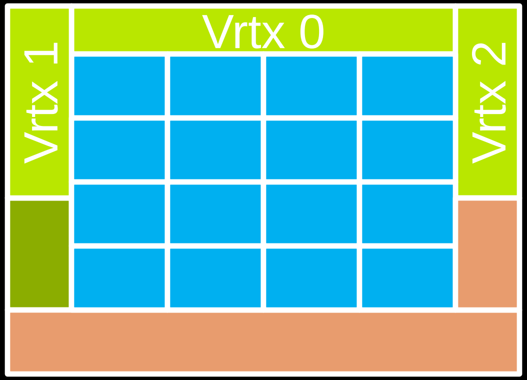
Why GPUs scale so nicely

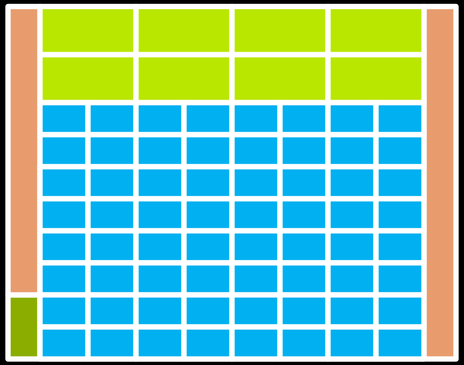
- Workload and Programming Model provide lots of parallelism
- Applications provide large groups of vertices at once
  - Vertices can be processed in parallel
  - Apply same transform to all vertices
- Triangles contain many pixels
  - Pixels from a triangle can be processed in parallel
  - Apply same shader to all pixels
- Very efficient hardware to hide serialization bottlenecks

With Moore's Law…



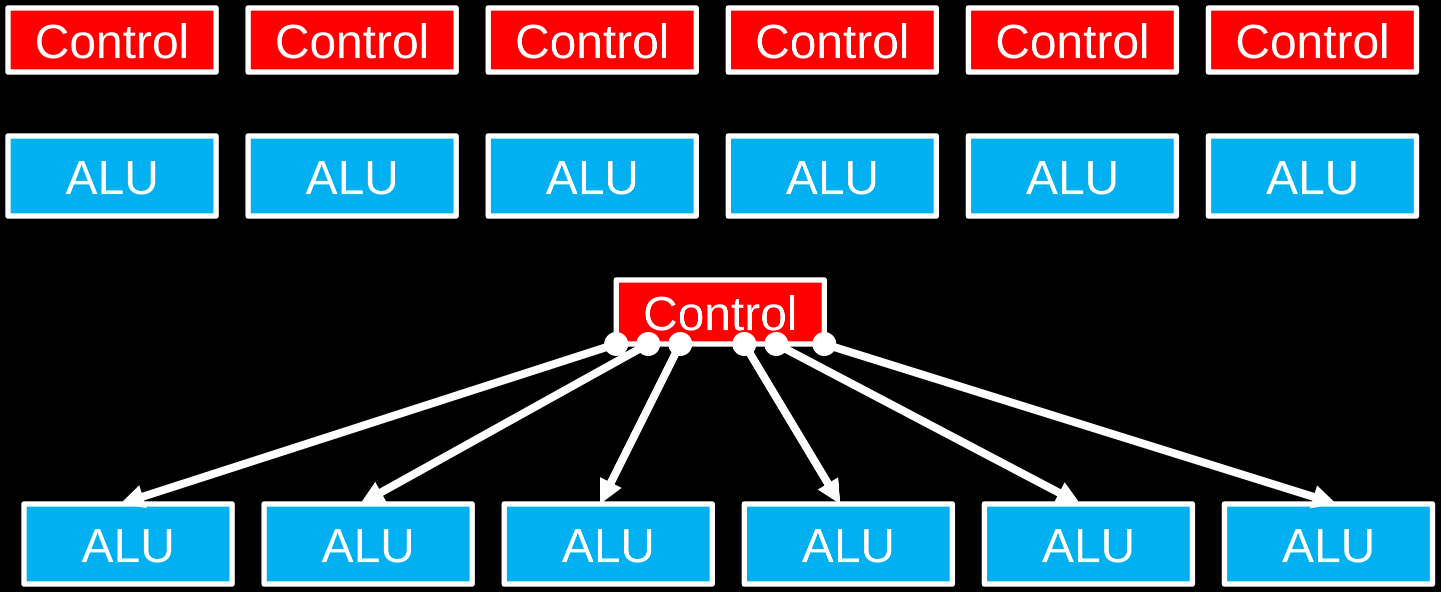






More Efficiency

- Note that we do the same thing for lots of pixels/vertices



- A warp = 32 threads launched together
  - Usually, execute together as well

Early GPGPU

- All this performance attracted developers
- To use GPUs, re-expressed their algorithms as graphics computations
- Very tedious, limited usability
- Still had some very nice results

- This was the lead up to CUDA

Questions?