CMSC 411
Computer Systems Architecture
Lecture 25
Graphics Processing Units (GPUs)

Alan Sussman
als@cs.umd.edu

Administrivia

- Homework #6 due Tuesday
- Practice final exam posted
- Course evaluations open, at https://CourseEvalUM.umd.edu
- Final exam Tuesday, May 20, 1:30-3:30PM

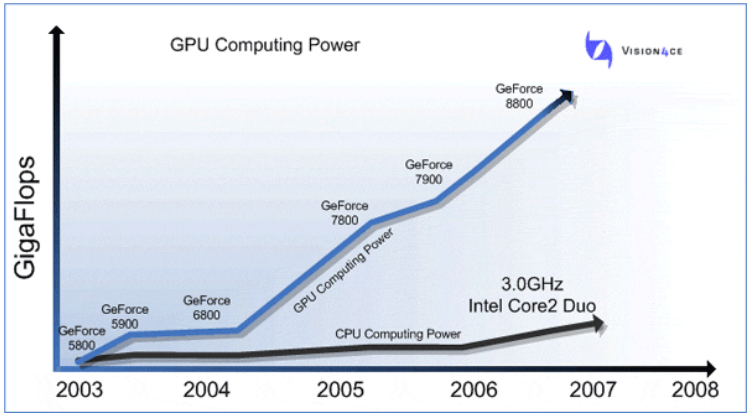
Graphics Processing Units (GPUs)

- CPUs
- Lots of instructions little data
  - » Out of order exec
  - » Branch prediction
- Reuse and locality
- Task parallel
- Needs OS
- Complex sync
- Latency machines

- GPUs
- Few instructions lots of data
  - » SIMD
  - » Hardware threading
- Little reuse
- Data parallel
- No OS
- Simple sync
- Throughput machines

GPU Performance

- Graphics Processing Units (GPUs) have been evolving at a rapid rate in recent years



CPU Performance

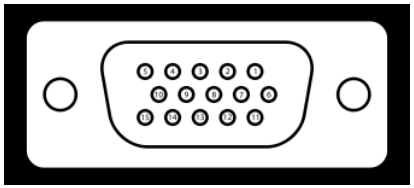
- CPUs have also been increasing functional unit counts
- But with a lot more complexity
  - Reorder buffers/reservations stations
  - Complex branch prediction
- This means that CPUs add raw compute power at a much slower rate

GPU vs. CPU

- Disparity is largely due to the specific nature of problems historically solved by the GPU
  - Same operations on many primitives (SIMD)
  - Focus on throughput over latency
  - Lots of special purpose hardware
- CPUs
  - Focus on reducing Latency
  - Designed to handle a wider range of problems

History of the GPU

- GPUs have mostly developed in the last 15 years
- Before that, graphics handled by Video Graphics Array (VGA) Controller
  - Memory controller, DRAM, display generator
  - Takes image data, and arranges it for output device

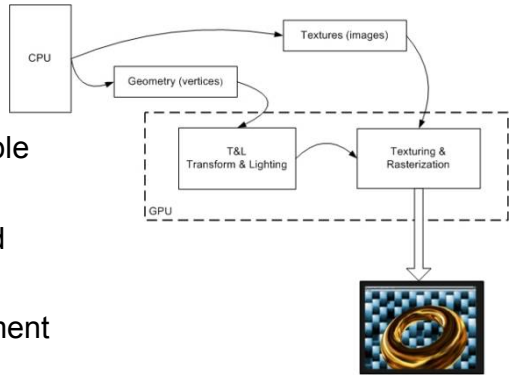


History of the GPU

- Graphics Acceleration hardware components were gradually added to VGA controllers
  - Triangle rasterization
  - Texture mapping
  - Simple shading
- Examples of early "graphics accelerators"
  - 3dfx Voodoo
  - ATI Rage
  - NVIDIA RIVA TNT2

History of the GPU

- NVIDIA GeForce 256 “first” GPU (1999)
  - Non-programmable (fixed-function)
  - Transforming and Lighting
  - Texture/Environment Mapping



History of the GPU

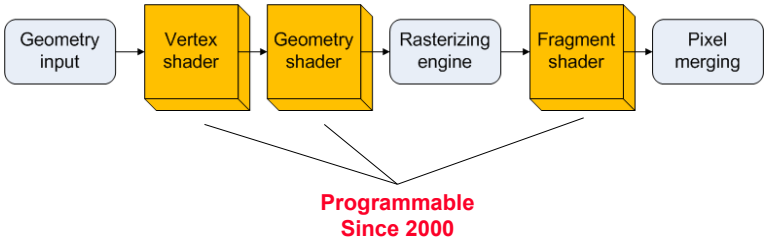
- Fairly early on in the GPU market, there was a severe narrowing of competition
- Early companies:
  - Silicon Graphics International
  - 3dfx
  - NVIDIA
  - ATI
  - Matrox
- Now only AMD and NVIDIA

History of the GPU

- Since their inception, GPUs have gradually become more powerful, programmable, and general purpose
  - Programmable geometry, vertex and pixel processors
  - Unified Shader Model
  - Expanding instruction set
  - CUDA, OpenCL

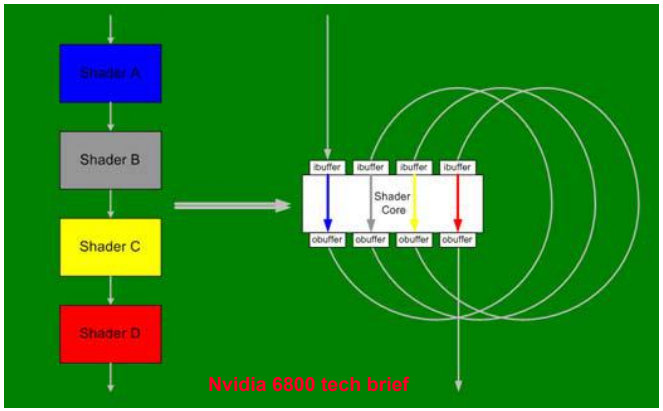
The (traditional) Graphics Pipeline

- Programmable elements of the graphics pipeline were historically fixed-function units, until about 2000



The Unified Shader

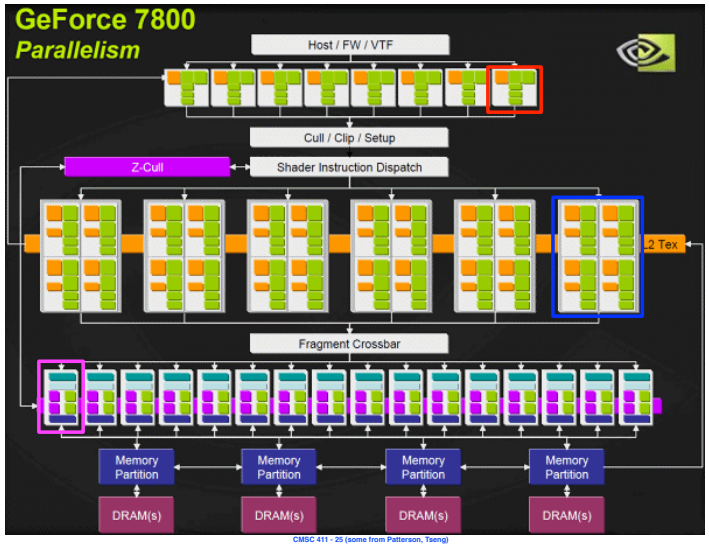
- With the introduction of the unified shader model, the GPU becomes essentially a many-core, streaming multiprocessor



GPU Chip Layouts

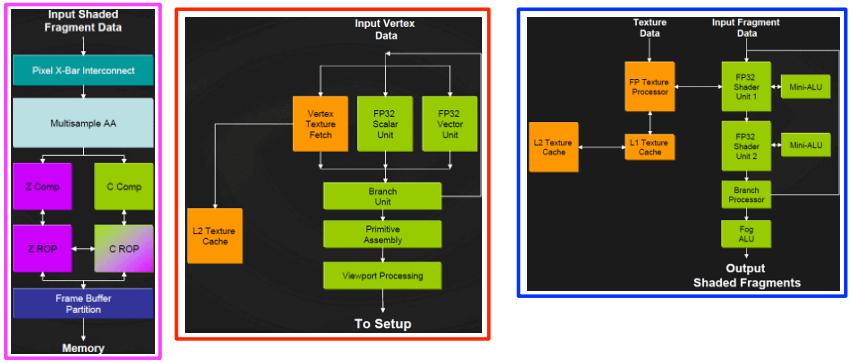
- GPU Chip layouts have been moving in the direction of general purpose computing for several years
- Some High-level trends
  - Unification of hardware components
  - Large increases in functional unit counts

GPU Chip Layouts



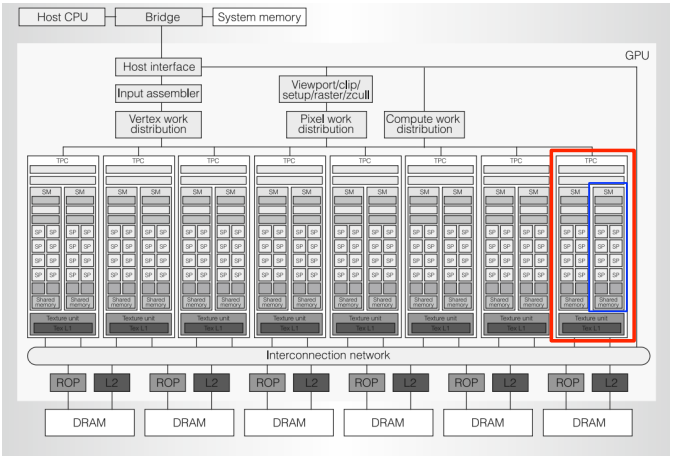
GPU Chip Layouts

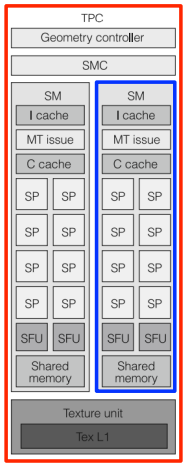
NVIDIA GeForce 7800



GPU Chip Layouts

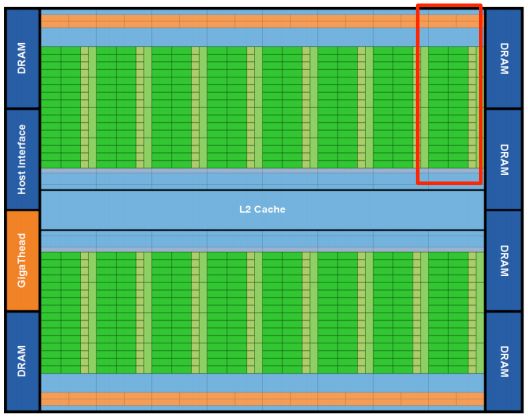
NVIDIA GeForce 8800

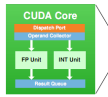


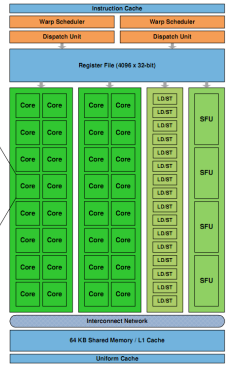


GPU Chip Layouts

NVIDIA GeForce 400 (Fermi architecture)



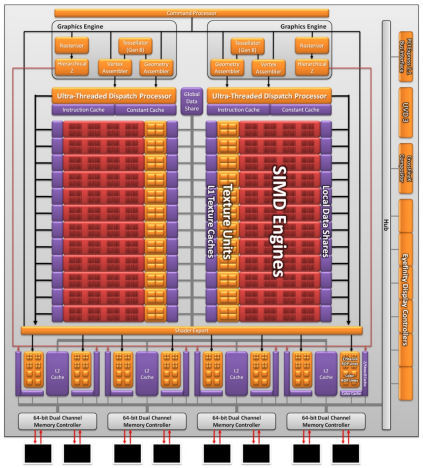




3 billion transisors

GPU Chip Layouts

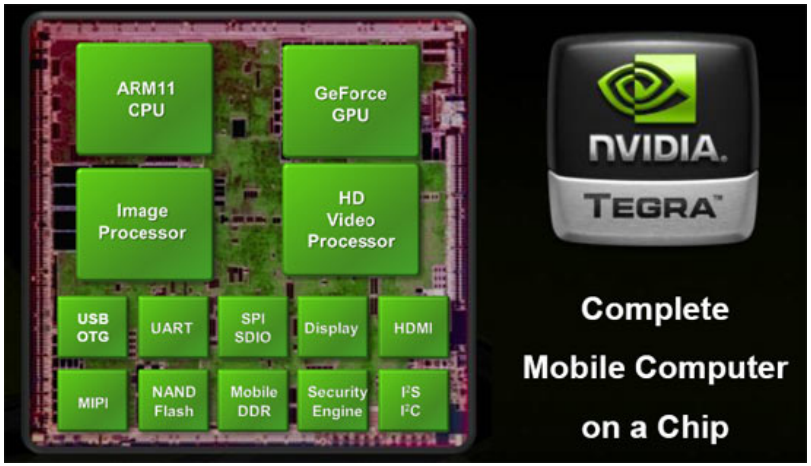
AMD Radeon 6800 (Cayman architecture)



2.64 billion transisors

"Hybrid" Chip Layouts

NVIDIA Tegra

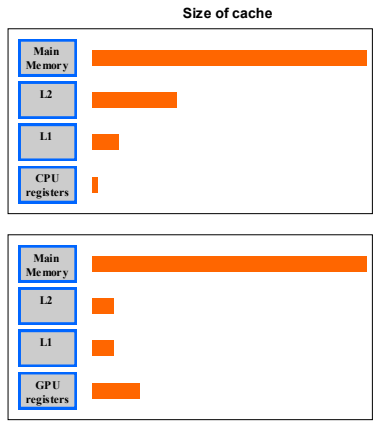


Emphasis on Throughput

- If your frame rate is 50 Hz, your latency can be approximately 2 ms ☺
- However, you need to do 100 million operations for that one frame ☹
- Result: very deep pipelines and high FLOPS
  - GeForce 7 had >200 stages for the pixel shader
  - Fermi: 1.5 TFLOPS, AMD 5870: 2.7 TFLOPS
  - Unified shader has cut down on the number of stages by allowing breaks from linear execution

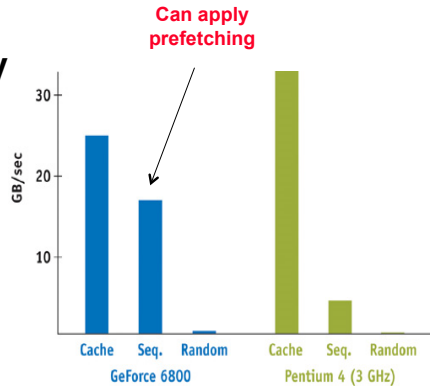
Memory Hierarchy

- **Cache size hierarchy is backwards from that of CPUs**
- **Caches serve to conserve precious memory bandwidth by intelligently prefetching**



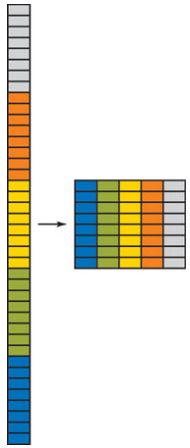
Memory Prefetching

- **Graphics pipelines are inherently high-latency**
- **Cache misses simply push another thread into the core**
- **Hit rates of ~90%, as opposed to ~100%**



Memory Access

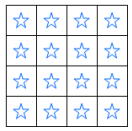
- **GPUs are all about 2D spatial locality, not linear locality**
- **GPU caches read-only (uses registers)**
- **Growing body of research optimizing algorithms for 2D cache model**



Instruction Set Differences

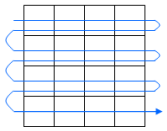
- Until very recently, scattered address space
  - 2009 saw the introduction of modern CPU-style 64-bit addressing
- Block operations versus sequential

```
block = 1:4 by 1:4
if y[i][j] = within block
       y[i][j] = y[i][j] + 1
```



Bam!

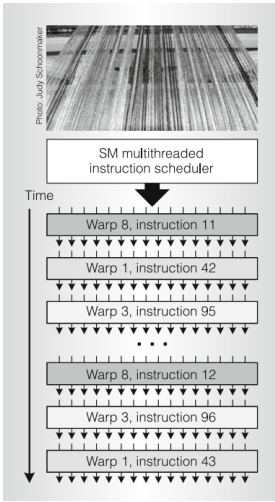
```
for i = 1 to 4
    for j = 1 to 4
        y[i][j] = y[i][j] + 1
```



- **SIMD: single instruction, multiple data**

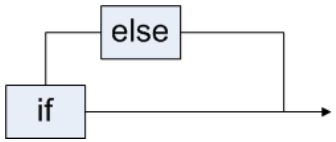
Single Instruction, Multiple Thread (SIMT)

- Newer GPUs are using a new kind of scheduling model called SIMT
- ~32 threads are bundled together in a "warp" and executed together
- Warps are then executed 1 instruction at a time, round robin



Instruction Set Differences

- Branch granularity
  - If one thread within a processor cluster branches without the rest, you have a branch divergence
  - Threads become serial until branches converge
  - Warp scheduling improves, not eliminates, hazards from branch divergence
- if/else may stall threads



Instruction Set Differences

- Unified shader
  - All shaders (since 2006) have the same basic instruction set layered on a (still) specialized core
  - Cores are very simple: hardware support for things like recursion may not be available
- Until very recently, dealing with speed hacks
  - Floating-point accuracy truncated to save cycles
  - IEEE FP specs are appearing on some GPUs
- Primitives limited to GPU data structures
  - GPUs operate on textures, etc
  - Computational variables must be mapped

GPU Limitations

- Relatively small amount of memory, < 4GB in current GPUs
- I/O directly to GPU memory has complications
  - Must transfer to host memory, and then back
  - If 10% of instructions are LD/ST and other instructions are...
    - » 10 times faster 1/(.1 + .9/10) ≈ speedup of 5
    - » 100 times faster 1/(.1 + .9/100) ≈ speedup of 9

Programming GPUs

- GPGPU
  - General purpose computing on GPUs
    - » Using special libraries (e.g. CUDA) to copy / process data
- Approach
  - GPUs can compute vector / stream operations in parallel
    - » Requires programs for both CPU & GPU
  - Compiler can simplify process of generating GPU code
    - » PGI compiler relies on user-inserted annotations to specify parallel region, vector operations

Programming GPUs

- Advantages
  - Supercomputer-like FP performance on commodity processors
- Disadvantages
  - Performance tuning difficult
  - Large speed gap between compiler-generated and hand-tuned code

Matrix Multiplication Example

- Original Fortran

```
do i = 1,n
  do j = 1,m
    do k = 1,p
      a(i,j) = a(i,j) + b(i,k)*c(k,j)
    enddo
  enddo
enddo
```

Matrix Multiplication Example

- Hand-written GPU code using CUDA

```
__global__ void matmulKernel( float* C, float* A, float* B, int N2, int N3 ){
  int bx = blockIdx.x,  by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;
  int aFirst = 16 * by * N2;
  int bFirst = 16 * bx;
  float Csub = 0;
  for( int j = 0; j < N2; j += 16 ) {
    __shared__ float Atile[16][16], Btile[16][16];
    Atile[ty][tx] = A[aFirst + j + N2 * ty + tx];
    Btile[ty][tx] = B[bFirst + j*N3 + b + N3 * ty + tx];
    __syncthreads();
     for( int k = 0; k < 16; ++k )
       Csub += Atile[ty][k] * Btile[k][tx];
    __syncthreads();
  }
  int c = N3 * 16 * by + 16 * bx;
  C[c + N3 * ty + tx] = Csub;
}
```


Matrix Multiplication Example

- Hand-written CPU code using CUDA

```
void matmul( float* A, float* B, float* C,
             size_t N1, size_t N2, size_t N3 ){
  void *devA, *devB, *devC;
  cudaSetDevice(0);
  cudaMalloc( &devA, N1*N2*sizeof(float) );
  cudaMalloc( &devB, N2*N3*sizeof(float) );
  cudaMalloc( &devC, N1*N3*sizeof(float) );
  cudaMemcpy( devA, A, N1*N2*sizeof(float), cudaMemcpyHostToDevice );
  cudaMemcpy( devB, B, N2*N3*sizeof(float), cudaMemcpyHostToDevice );
  dim3 threads( 16, 16 );
  dim3 grid( N1 / threads.x, N3 / threads.y);
  matmulKernel<<< grid, threads >>>( devC, devA, devB, N2, N3 );
  cudaMemcpy( C, devC, N1*N3*sizeof(float), cudaMemcpyDeviceToHost );
  cudaFree( devA );
  cudaFree( devB );
  cudaFree( devC );
}
```


Matrix Multiplication Example

- Annotated Fortran for PGI compiler (compiled to CUDA)

```
!$acc region
  !$acc do parallel
  do j=1,m
    do k=1,p
      !$acc do parallel, vector(2)
   do i=1,n
     a(i,j) = a(i,j) + b(i,k)*c(k,j)
   enddo
    enddo
  enddo
!$acc end region
```