• Originally designed just for graphics, strictly implementing the pipeline

• Programmable hardware showed up later, and became more and more flexible over time

• Modern GPUs are a sort of hybridization of CPUs and graphics hardware
CPU DESIGN

• CPU cores are designed for efficient execution of a single stream of instructions

• Branch prediction, instruction reordering, big caches, and so on on increase performance

• Usually clocked very fast (several GHz)

• Deep instruction pipelines

• Often have multiple (2-8) cores on a chip
CPU DESIGN

- Fetch/Decode
- ALU (Execute)
- Execution Context
- Data cache (a big one)
- Out-of-order control logic
- Fancy branch predictor
- Memory pre-fetcher
**Why?**

- Most programs are single threaded, and you want them to run as fast as possible

- Extra hardware for branch prediction / out-of-order execution / caches pays off big time
  
  - But costs chip area and power consumption...

- Deep instruction pipelines are fine as long as the features above are in place
Graphics pipeline

- Vertices: 1 in / 1 out
- Primitives: 3 in / 1 out (for tris)
- Fragments: 1 in / N out
- Pixels: 1 in / 0 or 1 out

Operations:
- Vertex Generation
- Vertex Processing
- Primitive Generation
- Primitive Processing
- Rasterization (Fragment Generation)
- Fragment Processing
- Frame-Buffer Ops

Memory:
- Uniform data
- Texture buffers

Frame Buffer
“Diamond” structure of graphics workload
Graphics workload

• Amount of work gets bigger as you go down the pipeline, always more work than hardware

• Every stage is highly parallel
  • Very few interdependencies
  • The same operations are happening to large chunks of data at each stage

• Do lots of the same thing in parallel
Example shader

```cpp
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

dependent vs frag shader
float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

1 unshaded fragment input record

1 shaded fragment output record
Simplify and Multiply

- Get rid of everything in the core specially designed to help a single instruction stream go fast
- Reduces power consumption, makes core smaller and simpler
  - Able to put many more cores in the same chip area
Simple core

- Fetch/Decode
- ALU (Execute)
- Execution Context

<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
c1mp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
2 cores
Instruction Streams

- All of those fragments are running the exact same instructions at the same time
- They should be able to share the fetch/decode unit
- This is SIMD: single instruction multiple data
  - The same instruction is run on several different pieces of data at the same time
SIMD shader

<diffuseShader>:
  sample r0, v4, t0, s0
  mul r3, v0, cb0[0]
  madd r3, v1, cb0[1], r3
  madd r3, v2, cb0[2], r3
  clmp r3, r3, l(0.0), l(1.0)
  mul o0, r0, r3
  mul o1, r1, r3
  mul o2, r2, r3
  mov o3, l(1.0)

<VEC8_diffuseShader>:
  VEC8_sample vec_r0, vec_v4, t0, vec_s0
  VEC8_mul vec_r3, vec_v0, cb0[0]
  VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
  VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
  VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
  VEC8_mul vec_o0, vec_r0, vec_r3
  VEC8_mul vec_o1, vec_r1, vec_r3
  VEC8_mul vec_o2, vec_r2, vec_r3
  VEC8_mov o3, l(1.0)
• 16 cores * 8 SIMD width = 128 ALUs
Heterogeneous Work

- Vertices
- Primitives
- Fragments
Branching in SIMD

- Must handle each case sequentially! (up to 8x slower)
**Stalls**

- There’s still an instruction pipeline on a GPU, so stalls can still happen
- CPU cores have fancy out-of-order execution units to help hide stalls
  - But we got rid of it... oops
- We still have way more work than ALUs
- If one instruction stream stalls, just run another one!
  - This is called latency hiding
Managing contexts

• Figuring out which context to switch in is not easy
• Contexts can take up a lot of storage space too
• GPUs have special thread scheduling hardware to handle all this for you, and dedicated memory for thread contexts
How fast is all this?

- Example: 16 cores, 8x SIMD width, 1 madd instruction per clock (2 FLOPS), 1 GHz clock
  - = 256 GFLOPS maximum throughput
  - With good latency hiding and parallelism, you can get pretty close to hitting that number
- Best GPUs are over 2.5 TFLOPS
- Best CPUs are around 140 GFLOPS (18x slower)
Hold up there...

- What about memory?
- GPUs are fast enough to consume about 20 TB/s
- The fastest memory available (GDDR5) has a bandwidth of about 200 GB/s
- That’s a ~100x difference in speed
  - Still 8x faster than CPU memory...
It gets worse

- CPUs have hierarchical caching schemes to minimize memory latency issues
- We got rid of most of that cache in the GPU, in order to make room for more ALUs... oops
CPU memory hierarchy

Processing Core (several per chip)

- OOO exec logic
- Branch predictor
- Fetch/Decode
- ALU
- Execution contexts

L1 cache (32 KB)

L2 cache (256 KB)

L3 cache (8 MB) shared across cores

25 GB/sec to memory
GPU memory hierarchy

- **Fetch/Decode**
  - ALU 1
  - ALU 2
  - ALU 3
  - ALU 4
  - ALU 5
  - ALU 6
  - ALU 7
  - ALU 8

- **Execution contexts** (128 KB)

- **Texture cache** (12 KB, read-only)

- **Scratchpad or L1 cache** (64 KB)

- **L2 cache** (768 KB) shared across cores

- **~150-200 GB/sec to memory**
How to deal

• Be smart about using bandwidth:
  • Workloads will often discard previous results (think fragments overwriting each other)
  • GPUs have complex buffers that will reorder and combine memory requests, catching overwrites before they’re sent out to memory
How to deal

• Use less memory:
  • Texture data will be shared among fragments that are close together, so they can share cache
  • Texture compression schemes are a very good idea
How to deal

• Do more math, instead of looking up in memory:
  • Memory is \(~100\)x more expensive than math
  • It’s more efficient to recompute a ton of math than it is to look up a previous result that’s not available in cache
The big picture
The punchline

- GPUs are very powerful, but only with:
  - Large amounts of completely independent work items
  - That can be executed in lockstep
  - And don’t branch very much
  - And use very little, very cohesive memory
• Kayvon Fatahalian’s CMU 15-869 slides
  • Most of this talk was basically lifted wholesale from Kayvon’s talks. They’re excellent references!