GPU Architecture Overview

Patrick Cozzi
University of Pennsylvania
CIS 565 - Fall 2012

Course Overview Follow-Up

- Read before or after lecture?
- Project vs. final project
- Closed vs. open source
- Feedback vs. grades
- Guest lectures

Announcements

- Project 0 due Monday 09/17
- Karl's office hours
  - Tuesday, 4:30-6pm
  - Friday, 2-5pm

Acknowledgements

- CPU slides – Varun Sampath, NVIDIA
- GPU slides
  - Kayvon Fatahalian, CMU
  - Mike Houston, AMD

CPU and GPU Trends

- *FLOPS* – *FL*oating-point *OP*erations per *S*econd
- *GFLOPS* - One billion (10^9) FLOPS
- *TFLOPS* – 1,000 GFLOPS

CPU and GPU Trends

- Compute
  - Intel Core i7 – 4 cores – 100 GFLOP
  - NVIDIA GTX280 – 240 cores – 1 TFLOP
- Memory Bandwidth
  - System Memory – 60 GB/s
  - NVIDIA GT200 – 150 GB/s
- Install Base
  - Over 200 million NVIDIA G80s shipped

CPU and GPU Trends

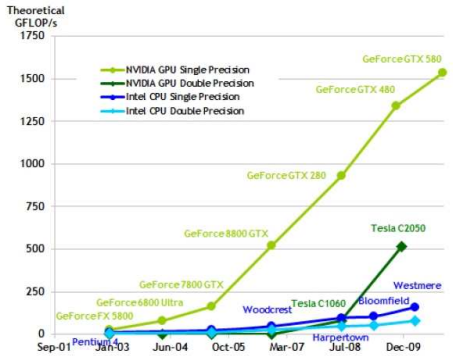


Chart from: http://proteneer.com/blog/?p=263

CPU Review

- What are the major components in a CPU die?

CPU Review

- Desktop Applications
  - Lightly threaded
  - Lots of branches
  - Lots of memory accesses

	`vim`	`ls`
Conditional branches	13.6%	12.5%
Memory accesses	45.7%	45.7%
Vector instructions	1.1%	0.2%

Profiled with `psrun` on ENIAC

Slide from Varun Sampath

Pipelining

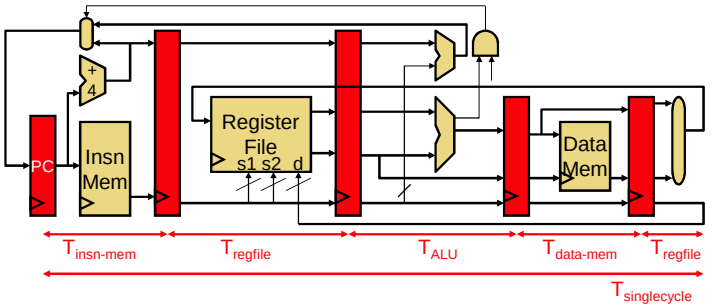
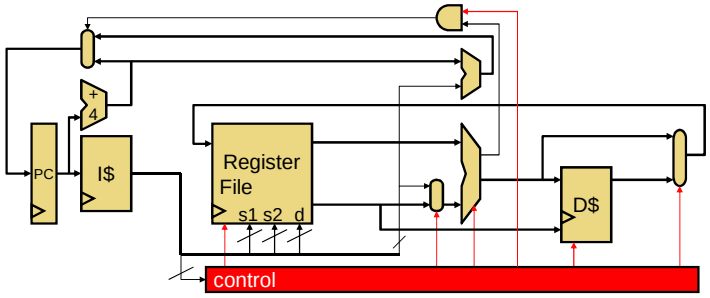


Image: Penn CIS501

A Simple CPU Core



Fetch → Decode → Execute → Memory → Writeback

Image: Penn CIS501

Branches

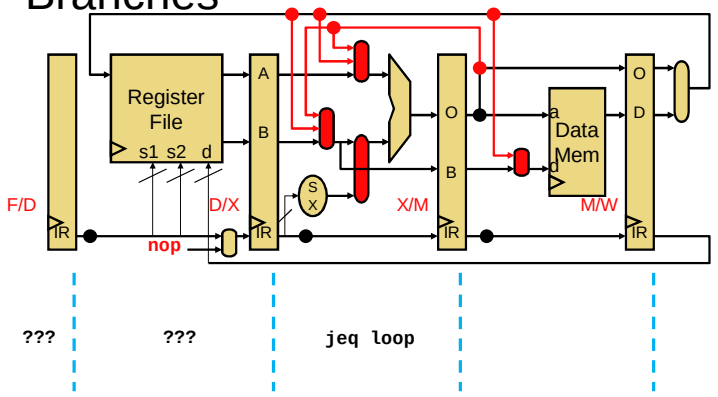


Image: Penn CIS501

Branch Prediction

+ Modern predictors > 90% accuracy
  - Raise performance *and* energy efficiency (why?)
– Area increase
– Potential fetch stage latency increase

Caching

- Keep data you need close
- Exploit:
  - Temporal locality
    - Chunk just used likely to be used again soon
  - Spatial locality
    - Next chunk to use is likely close to previous

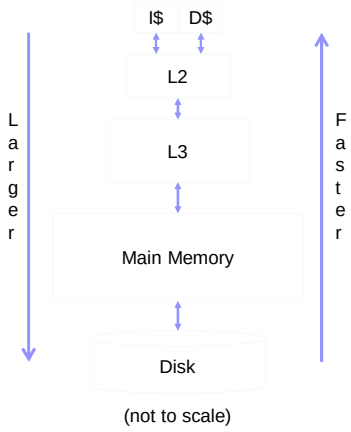
Memory Hierarchy

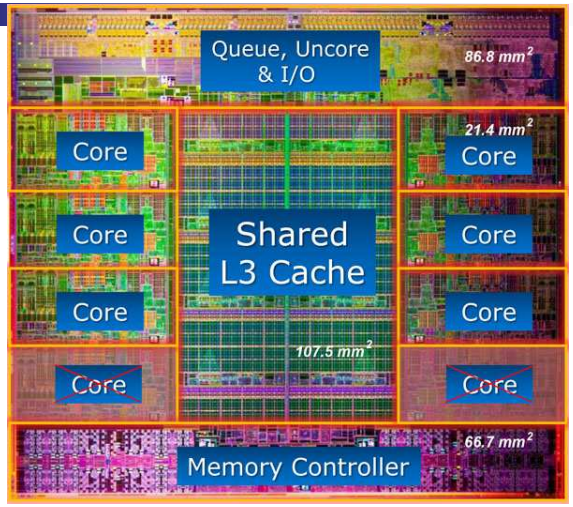
- Memory: the larger it gets, the slower it gets
- Rough numbers:

	Latency	Bandwidth	Size
SRAM (L1, L2, L3)	1-2ns	200GBps	1-20MB
DRAM (memory)	70ns	20GBps	1-20GB
Flash (disk)	70-90µs	200MBps	100-1000GB
HDD (disk)	10ms	1-150MBps	500-3000GB

Cache Hierarchy

- Hardware-managed
  - L1 Instruction/Data caches
  - L2 unified cache
  - L3 unified cache
- Software-managed
  - Main memory
  - Disk





Intel Core i7 3960X – 15MB L3 (25% of die). 4-channel Memory Controller, 51.2GB/s total

Source: www.lostcircuits.com

Scheduling

- Consider instructions:

```
xor r1,r2 -> r3
add r3,r4 -> r4
sub r5,r2 -> r3
addi r3,1 -> r1
```

- `xor` and `add` are dependent (Read-After-Write, RAW)
- `sub` and `addi` are dependent (RAW)
- `xor` and `sub` are *not* (Write-After-Write, WAW)

Improving IPC

- IPC (instructions/cycle) bottlenecked at 1 instruction / clock
- Superscalar – increase pipeline width

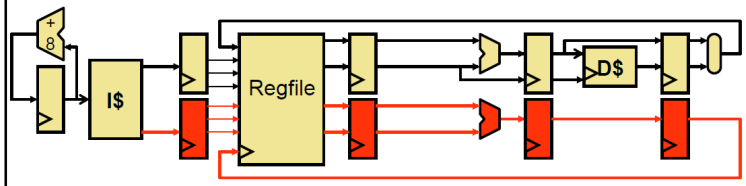


Image: Penn CIS371

Register Renaming

- How about this instead:

```
xor p1,p2 -> p6
add p6,p4 -> p7
sub p5,p2 -> p8
addi p8,1 -> p9
```

- `xor` and `sub` can now execute in parallel

Out-of-Order Execution

- Reordering instructions to maximize throughput
- Fetch → Decode → Rename → Dispatch → Issue → Register-Read → Execute → Memory → Writeback → Commit
- Reorder Buffer (ROB)
  - Keeps track of status for in-flight instructions
- Physical Register File (PRF)
- Issue Queue/Scheduler
  - Chooses next instruction(s) to execute

Vectors Motivation

```
for (int i = 0; i < N; i++)
    A[i] = B[i] + C[i];
```

Parallelism in the CPU

- Covered Instruction-Level (ILP) extraction
  - Superscalar
  - Out-of-order
- Data-Level Parallelism (DLP)
  - Vectors
- Thread-Level Parallelism (TLP)
  - Simultaneous Multithreading (SMT)
  - Multicore

CPU Data-level Parallelism

- Single Instruction Multiple Data (SIMD)
  - Let's make the execution unit (ALU) really wide
  - Let's make the registers really wide too

```
for (int i = 0; i < N; i+= 4) {
    // in parallel
    A[i] = B[i] + C[i];
    A[i+1] = B[i+1] + C[i+1];
    A[i+2] = B[i+2] + C[i+2];
    A[i+3] = B[i+3] + C[i+3];
}
```

Vector Operations in x86

- SSE2
  - 4-wide packed float and packed integer instructions
  - Intel Pentium 4 onwards
  - AMD Athlon 64 onwards
- AVX
  - 8-wide packed float and packed integer instructions
  - Intel Sandy Bridge
  - AMD Bulldozer

Simultaneous Multithreading

- Instructions can be issued from multiple threads
- Requires partitioning of ROB, other buffers

+ Minimal hardware duplication

+ More scheduling freedom for OoO

– Cache and execution resource contention can reduce single-threaded performance

Thread-Level Parallelism

- Thread Composition
  - Instruction streams
  - Private PC, registers, stack
  - Shared globals, heap
- Created and destroyed by programmer
- Scheduled by programmer or by OS

Multicore

- Replicate full pipeline
- Sandy Bridge-E: 6 cores

+ Full cores, no resource sharing other than last-level cache

+ Easier way to take advantage of Moore's Law

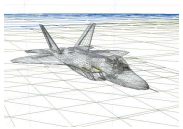
– Utilization

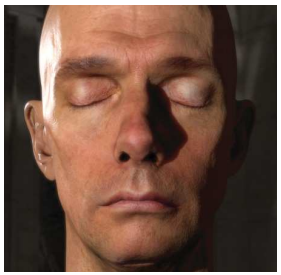
CPU Conclusions

- CPU optimized for sequential programming
  - Pipelines, branch prediction, superscalar, OoO
  - Reduce execution time with high clock speeds and high utilization
- Slow memory is a constant problem
- Parallelism
  - Sandy Bridge-E great for 6-12 active threads
  - How about 12,000?

Graphics Workloads

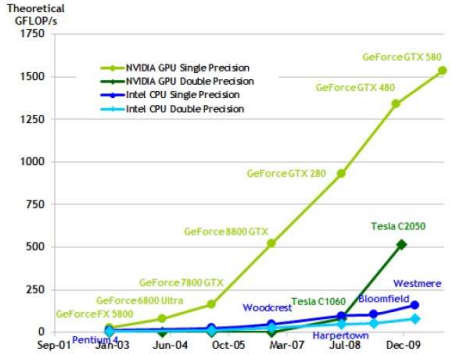
- Triangles/vertices and pixels/fragments





Right image from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch14.html

How did this happen?



http://proteneer.com/blog/?p=263

Early 90s – Pre GPU

Wolfenstein 3D, 1992

Doom I, 1993

- Interactive software rendering (no GPUs yet)
- NOTE: SGI was building interactive rendering supercomputers, but this was beginning of interactive 3D graphics on PC

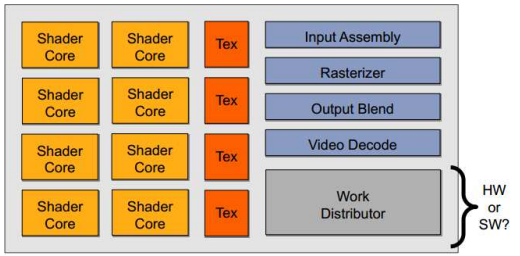
Slide from http://s09.idav.ucdavis.edu/talks/01-BPS-SIGGRAPH09-mhouston.pdf

Why GPUs?

- Graphics workloads are embarrassingly parallel
  - Data-parallel
  - Pipeline-parallel
- CPU and GPU execute in parallel
- Hardware: texture filtering, rasterization, etc.

What's in a GPU?

A GPU is a heterogeneous chip multi-processor (highly tuned for graphics)



Data Parallel

- *Beyond Graphics*
  - Cloth simulation
  - Particle system
  - Matrix multiply

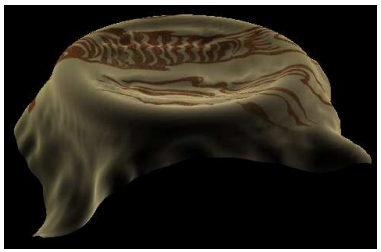


Image from: https://plus.google.com/u/0/photos/100838748547881402137/albums/5407605084626995217/5581900335460078306

A diffuse reflectance shader

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

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);
}
```

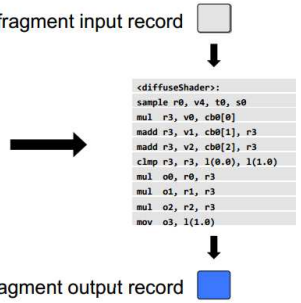
Shader programming model:

Fragments are processed independently,
but there is no explicit parallel programming

Compile shader

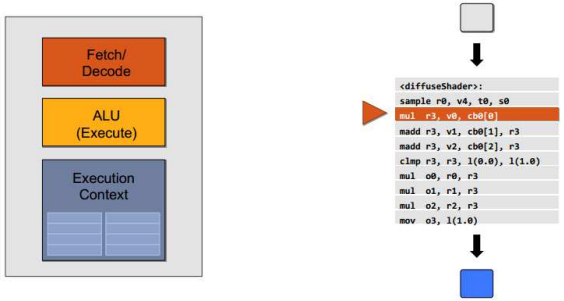
1 unshaded fragment input record

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

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);
}
```

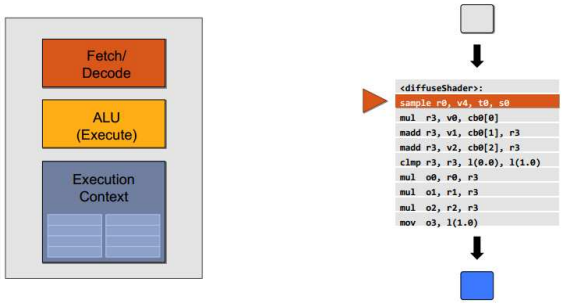


1 shaded fragment output record

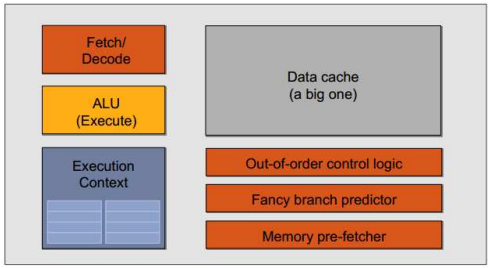
Execute shader



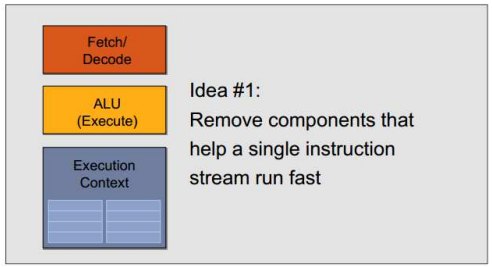
Execute shader



"CPU-style" cores

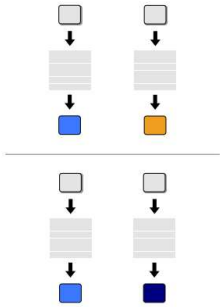


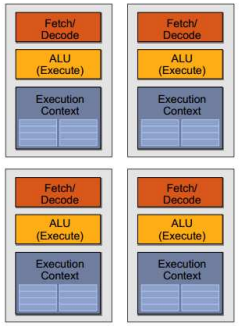
Slimming down



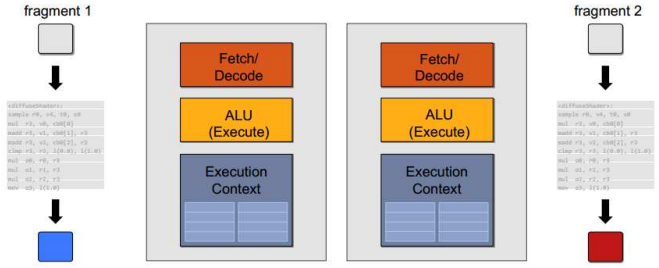
Idea #1:
Remove components that help a single instruction stream run fast

Four cores (four fragments in parallel)

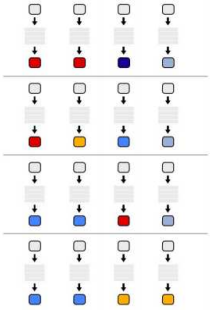


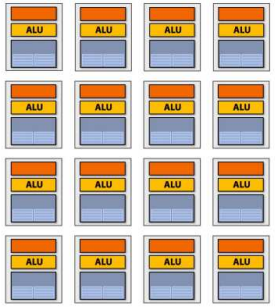


Two cores (two fragments in parallel)



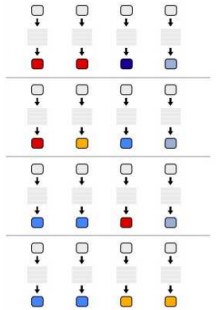
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

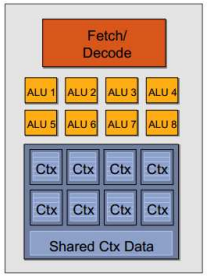
Instruction stream sharing



But ... many fragments should be able to share an instruction stream!

```
<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)
```

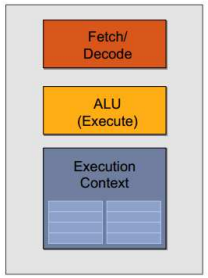

Add ALUs



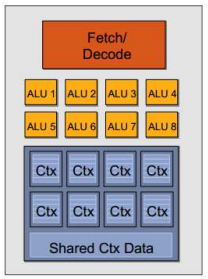
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

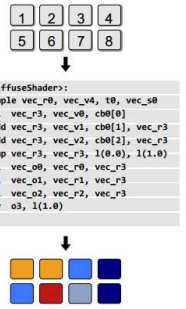
SIMD processing

Recall: simple processing core

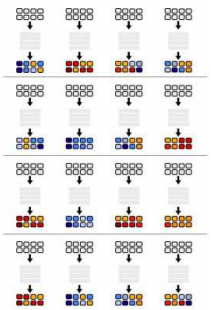


Modifying the shader

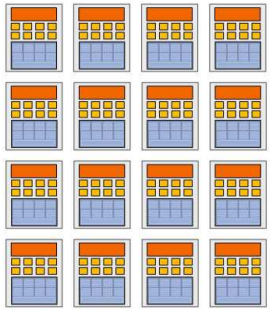




128 fragments in parallel

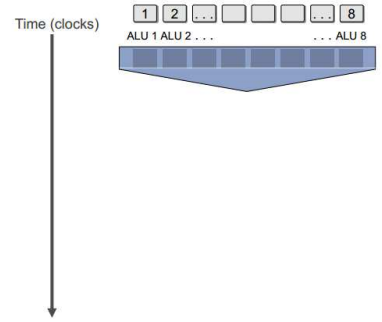


16 cores = 128 ALUs



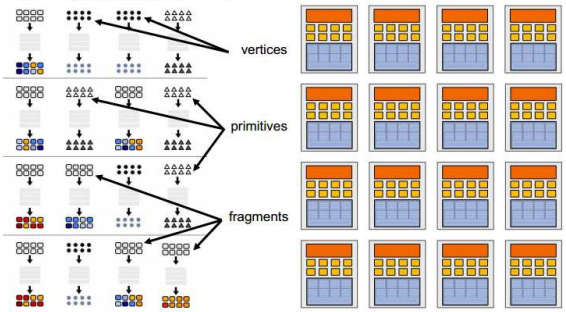
, 16 simultaneous instruction streams

But what about branches?

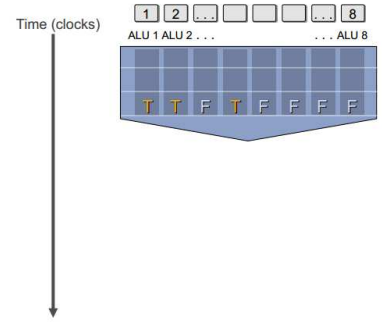



```
<unconditional
 shader code>

if (x > 0) {
   y = pow(x, exp);
   y *= Ks;
   refl = y + Ka;
} else {
   x = 0;
   refl = Ka;
}

<resume unconditional
 shader code>
```


128 [vertices/fragments primitives OpenCL work items] in parallel



But what about branches?




```
<unconditional
 shader code>

if (x > 0) {
   y = pow(x, exp);
   y *= Ks;
   refl = y + Ka;
} else {
   x = 0;
   refl = Ka;
}

<resume unconditional
 shader code>
```

But what about branches?

Not all ALUs do useful work!
Worst case: 1/8 peak performance

Stalls!

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

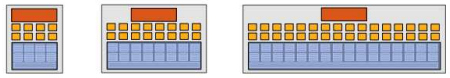
Texture access latency = 100's to 1000's of cycles

We've removed the fancy caches and logic that helps avoid stalls.

Clarification

SIMD processing does not imply SIMD instructions

- Option 1: explicit vector instructions
  - x86 SSE, AVX, Intel Larrabee
- Option 2: scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce ("SIMT" warps), ATI Radeon architectures ("wavefronts")



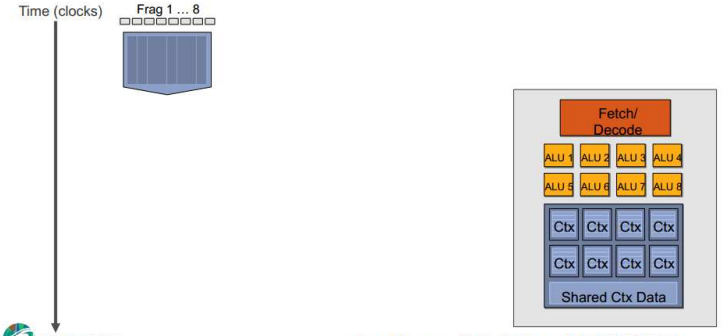
In practice: 16 to 64 fragments share an instruction stream.

But we have LOTS of independent fragments.

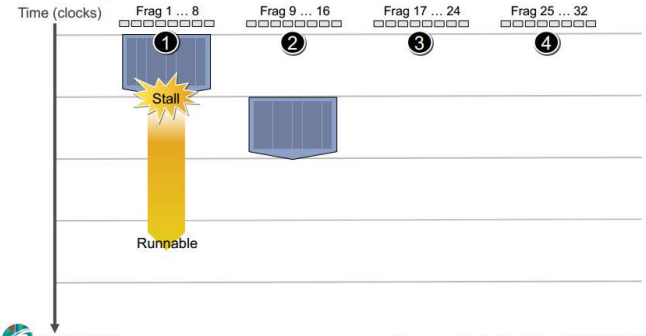
Idea #3:

Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

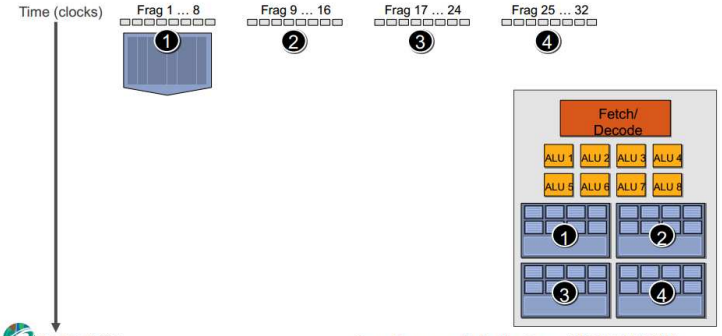
Hiding shader stalls



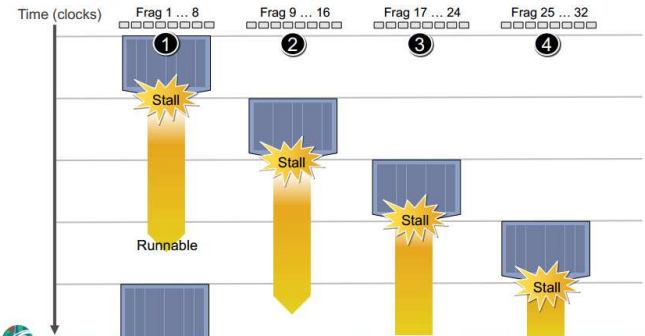
Hiding shader stalls



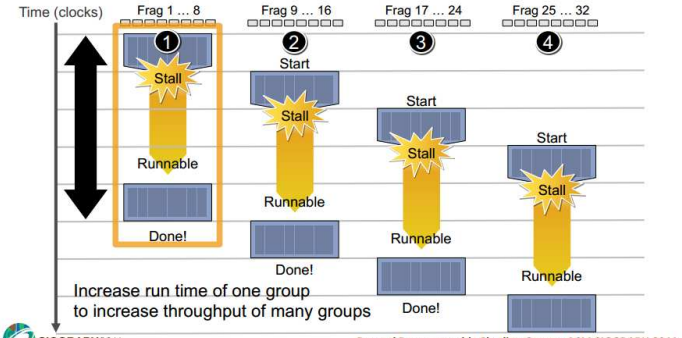
Hiding shader stalls



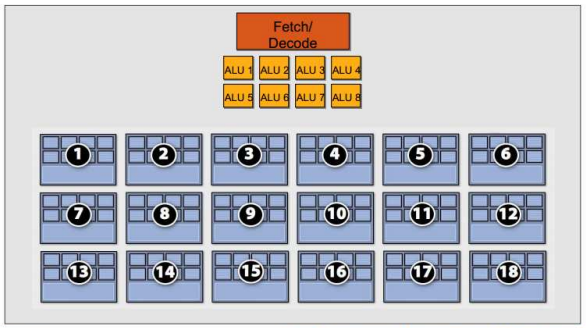
Hiding shader stalls



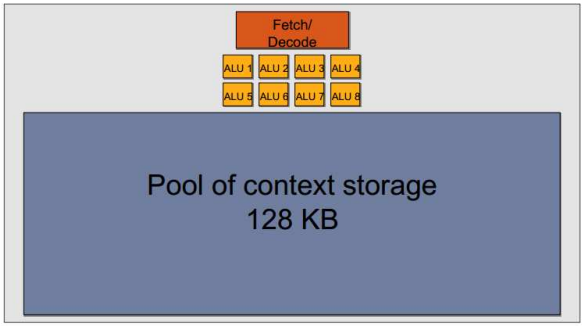
Throughput!



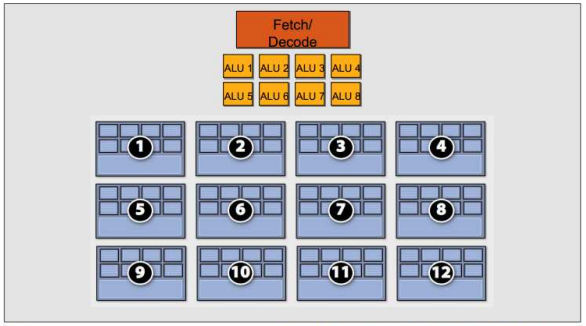
Eighteen small contexts (maximal latency hiding)



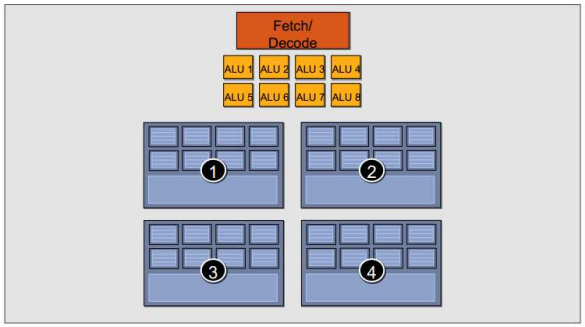
Storing contexts



Twelve medium contexts



Four large contexts (low latency hiding ability)



Example chip

16 cores

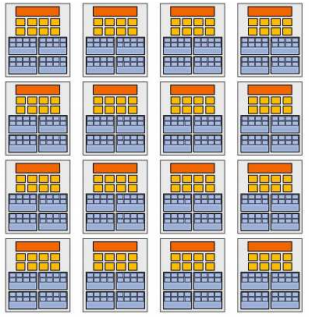
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



Clarification

Interleaving between contexts can be managed by hardware or software (or both!)

- NVIDIA / ATI Radeon GPUs
  - HW schedules / manages all contexts (lots of them)
  - Special on-chip storage holds fragment state
- Intel Larrabee
  - HW manages four x86 (big) contexts at fine granularity
  - SW scheduling interleaves many groups of fragments on each HW context
  - L1-L2 cache holds fragment state (as determined by SW)

Summary: three key ideas

1. Use many “slimmed down cores” to run in parallel

2. Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group

Reminders

- Piazza
  - Signup: https://piazza.com/upenn/fall2012/cis565/
- GitHub
  - Create an account: https://github.com/signup/free
  - Change it to an edu account: https://github.com/edu
  - Join our organization: https://github.com/CIS565-Fall-2012
- No class Wednesday, 09/12