GPU Architecture Overview

Bin ZHOU 2015 USTC

Acknowledgement

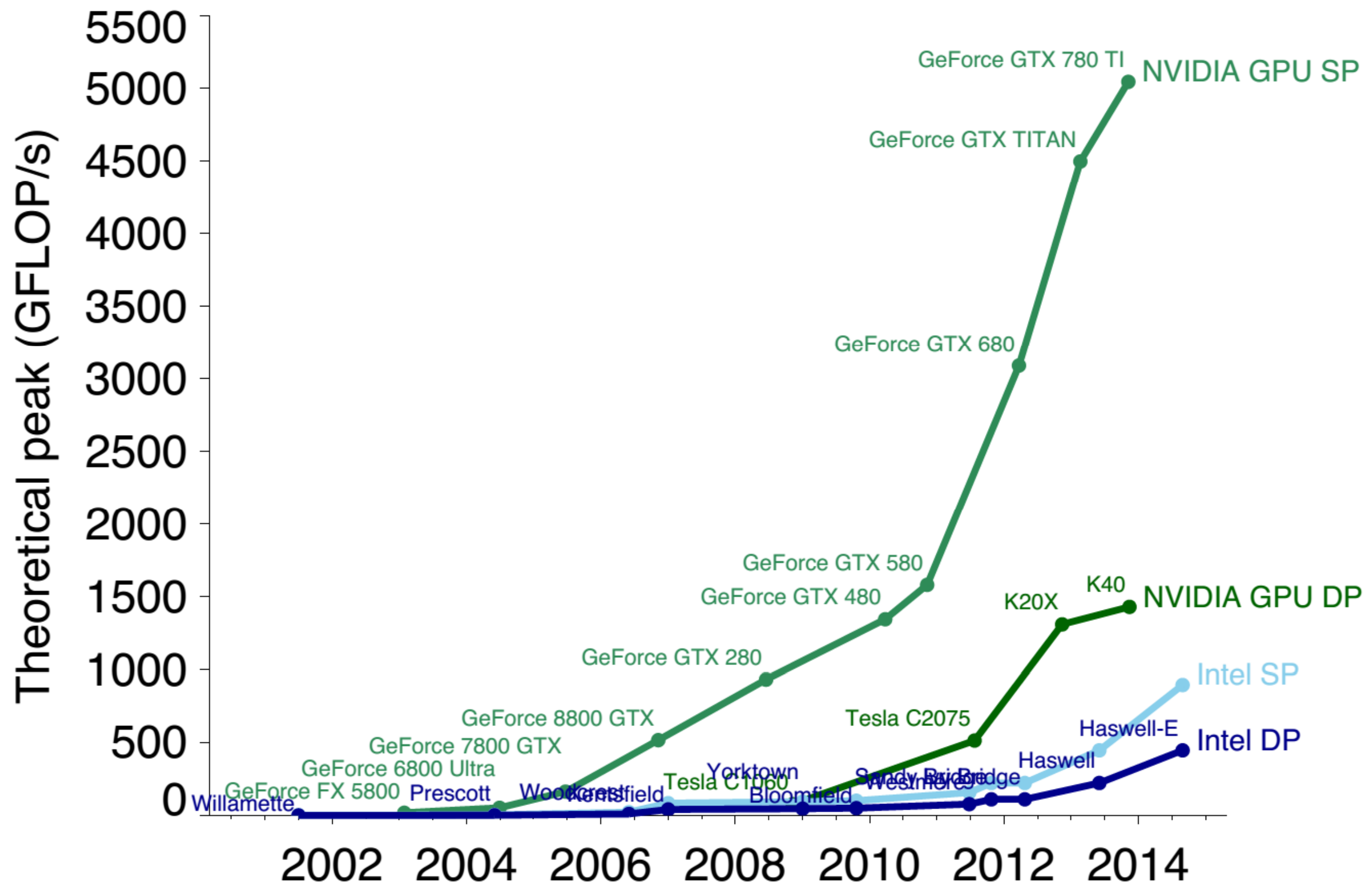
- Patrick Cozzi, University of Pennsylvania, CIS 565 - Fall 2013
- Some CPU slides – Varun Sampath, NVIDIA
- Some GPU slides Kayvon Fatahalian, CMU

Contents

Terms

- FLOPS – FLoating-point OPerations per Second
- GFLOPS – One billion (10^9) FLOPS
- TFLOPS – 1,000 GFLOPS

CPU and GPU Performance Trends



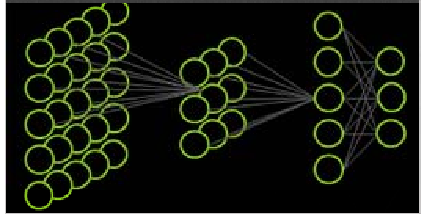
Why do we need GPU?

- More and More Complex Problems
- （Application Driven）

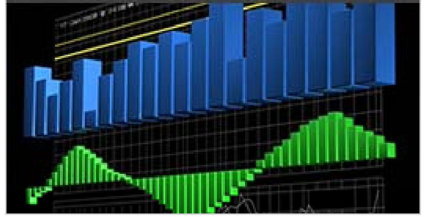
Applications Drives

BIOINFORMATICS

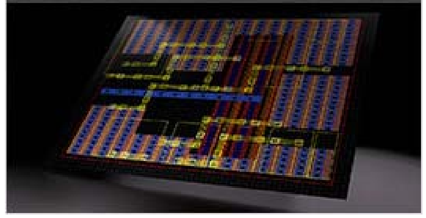
MACHINE LEARNING



COMPUTATIONAL FINANCE

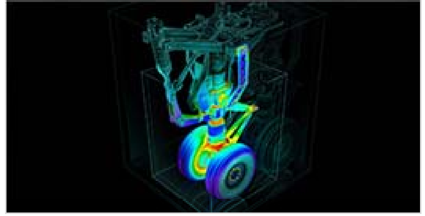


ELECTRONIC DESIGN AUTOMATION

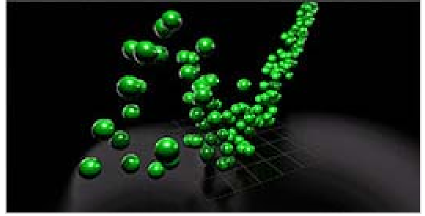


COMPUTATIONAL FLUID DYNAMICS

COMPUTATIONAL STRUCTURAL MECHANICS

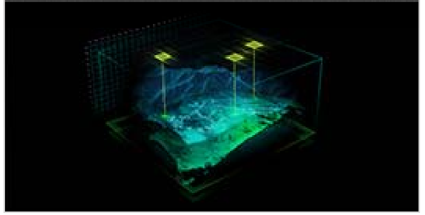


COMPUTATIONAL CHEMISTRY



DATA SCIENCE

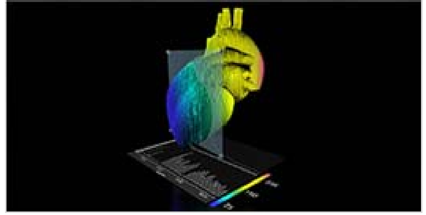
DEFENSE



WEATHER AND CLIMATE

IMAGING & COMPUTER VISION

MEDICAL IMAGING



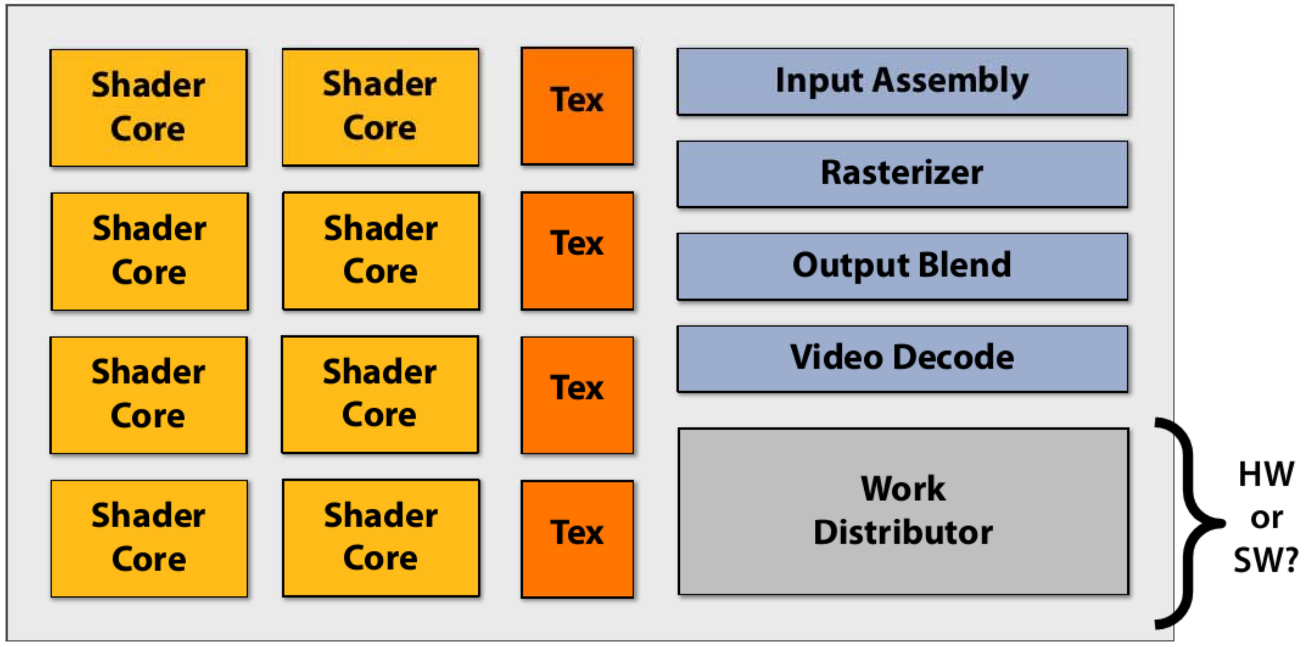
Any More Applications Hunger for Computing Power?

Or your research related problems?

GPU (Graphic Processing Unit) Architecture Diagram

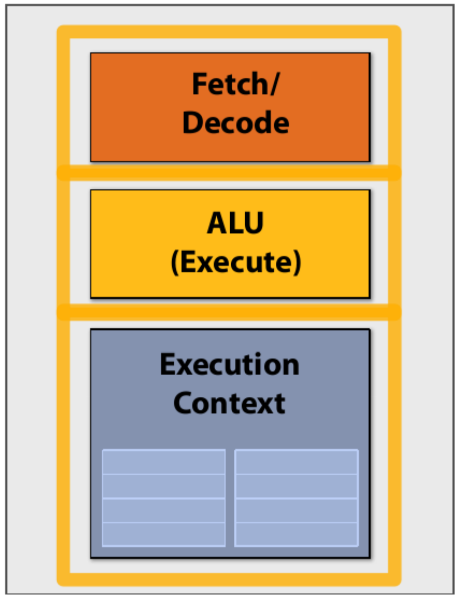
GPU is designed for embarrassingly parallel workloads

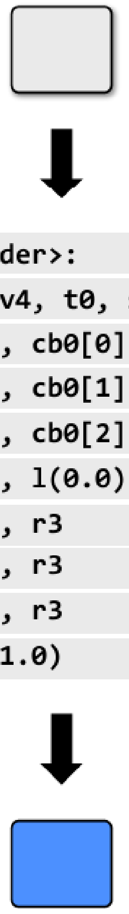
A GPU is a heterogeneous chip multi-processor (highly tuned for graphics)



Basic Exe Elements

Execute shader

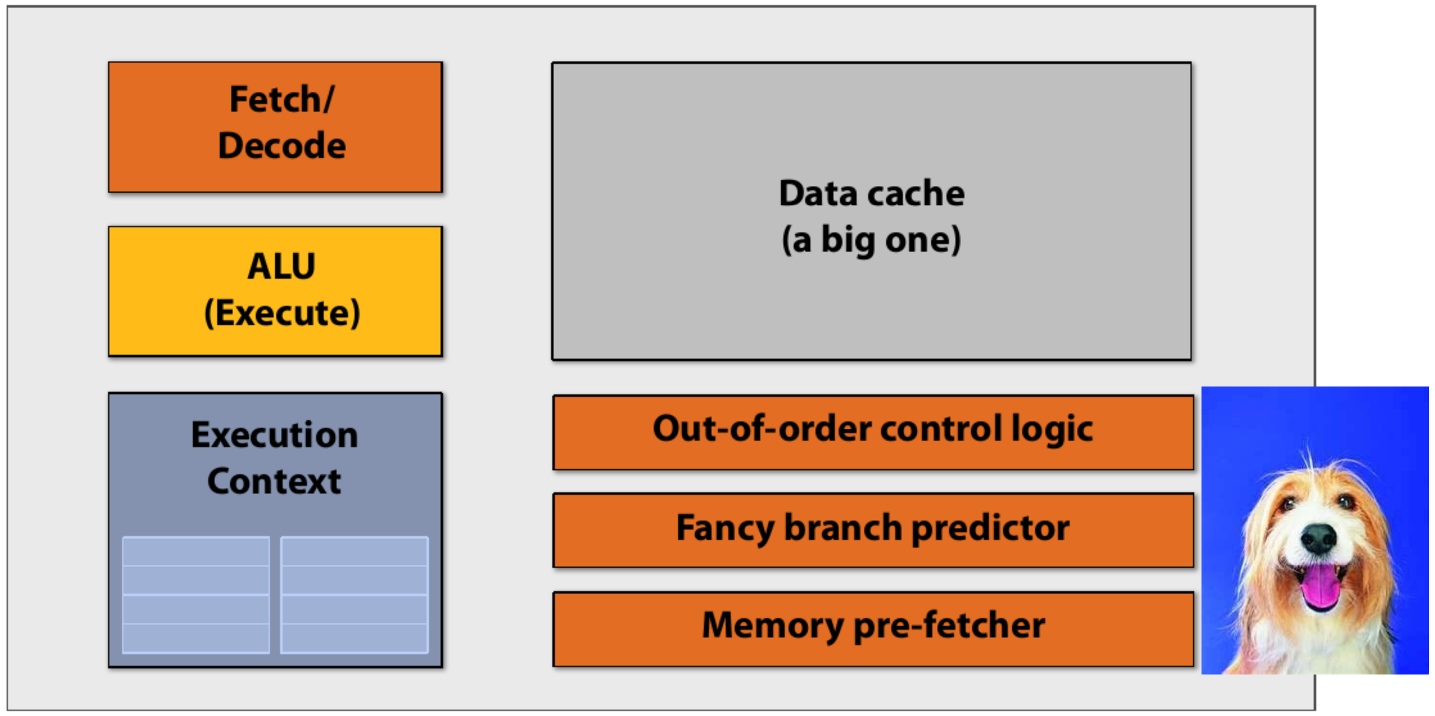




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

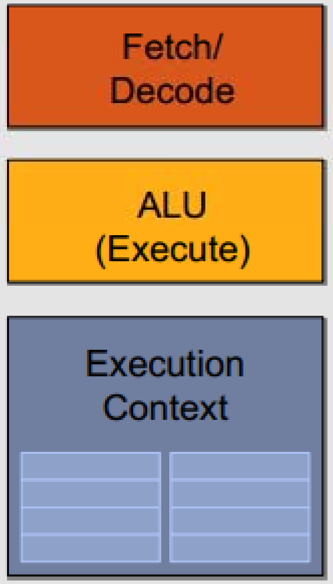
Recall CPU Architecture

“CPU-style” cores



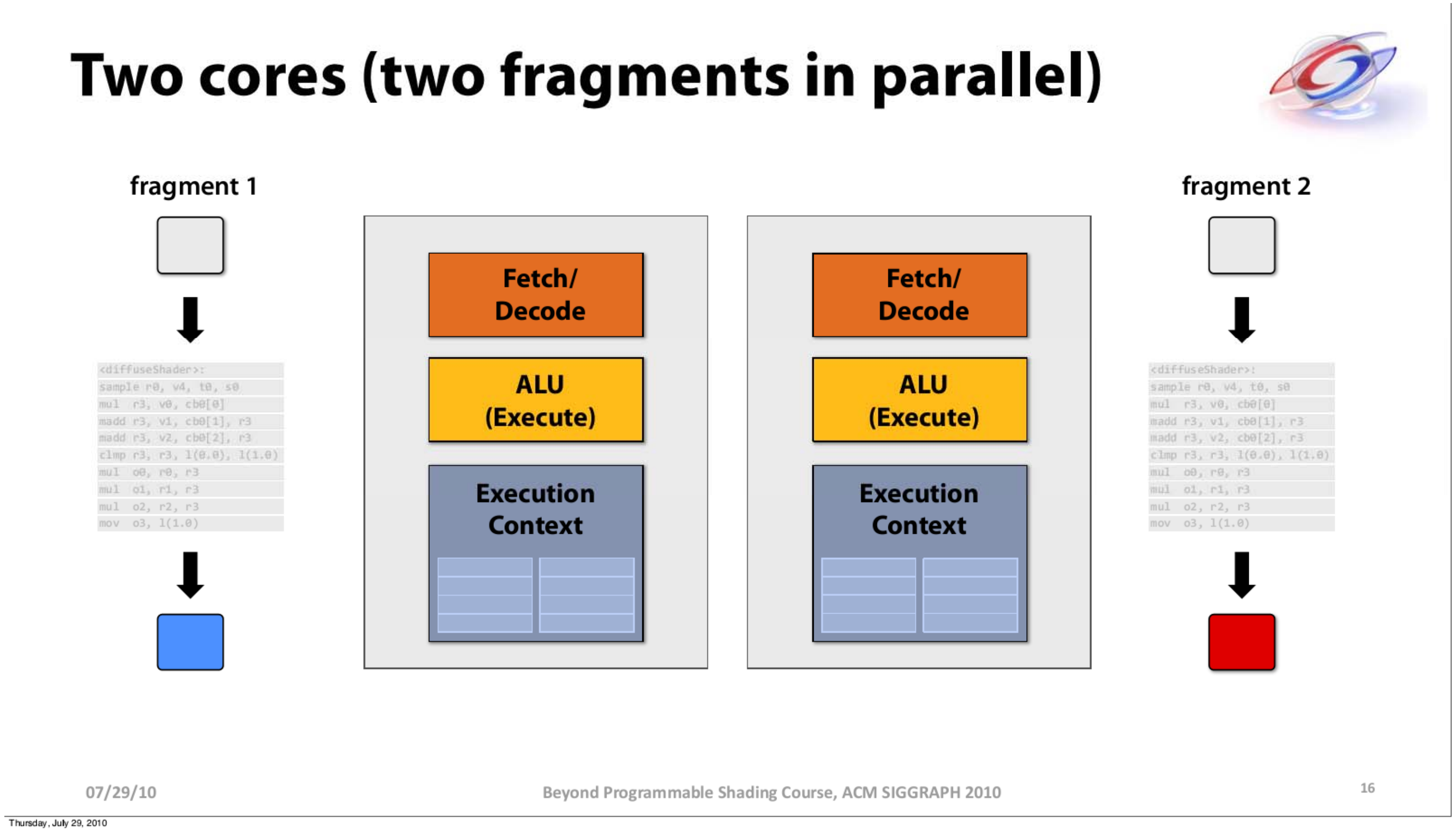
Idea 1:

Slimming down



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

Copy & Paste



Two cores (two fragments in parallel)

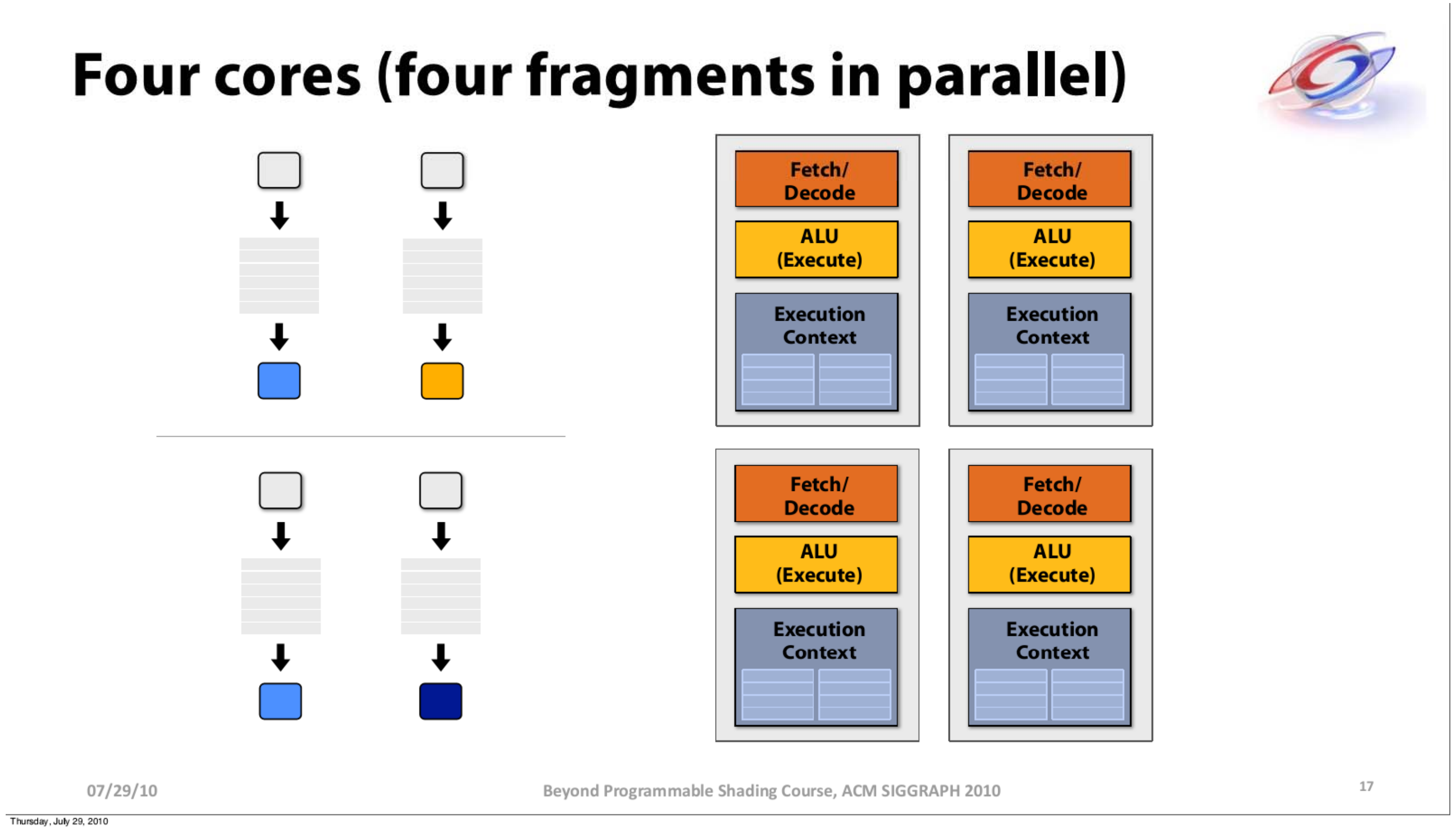
fragment 1

```
<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)
```

fragment 2

```
<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)
```

Copy and Paste Again!

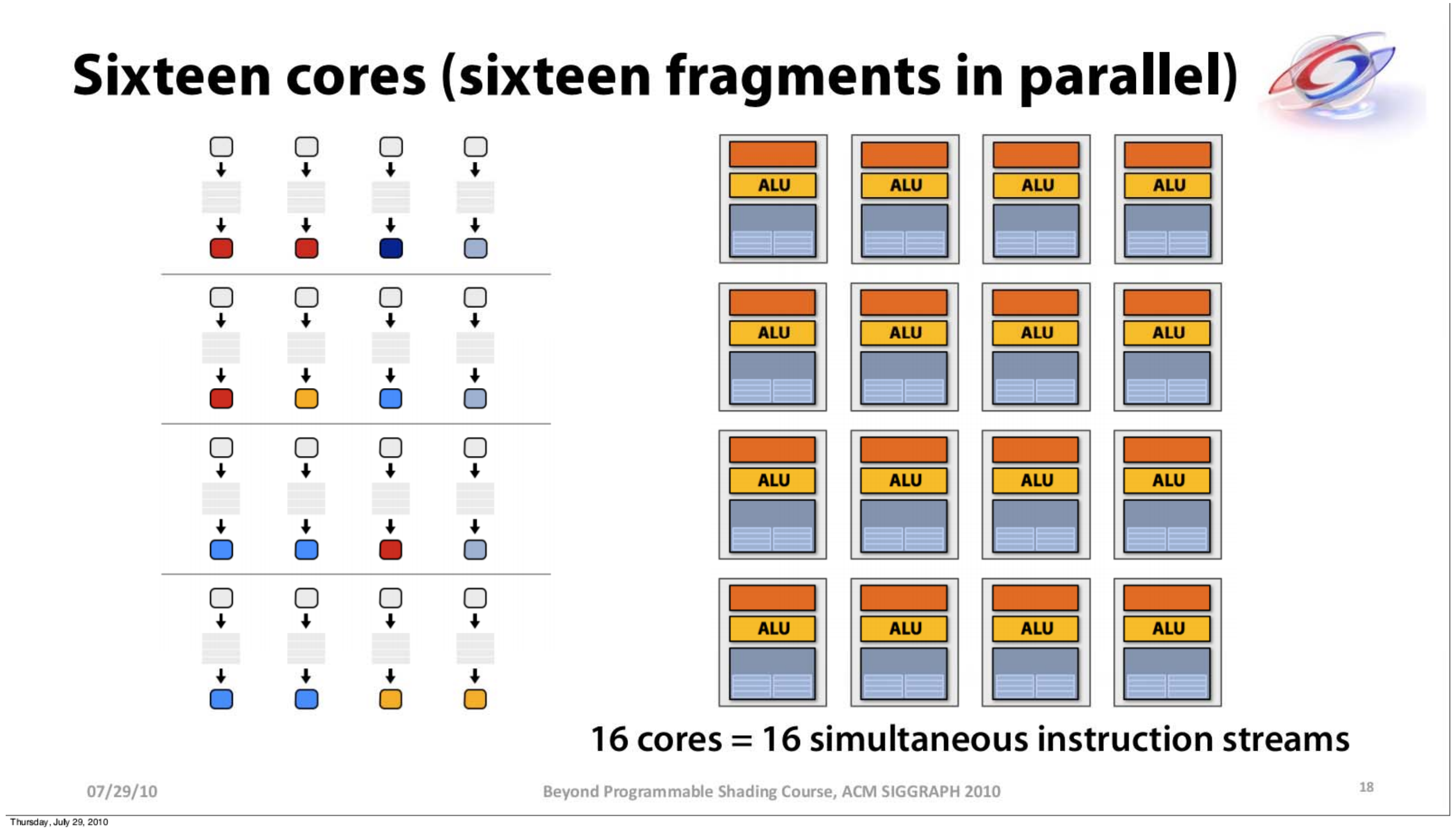


Four cores (four fragments in parallel)

07/29/10 Beyond Programmable Shading Course, ACM SIGGRAPH 2010 17

Thursday, July 29, 2010

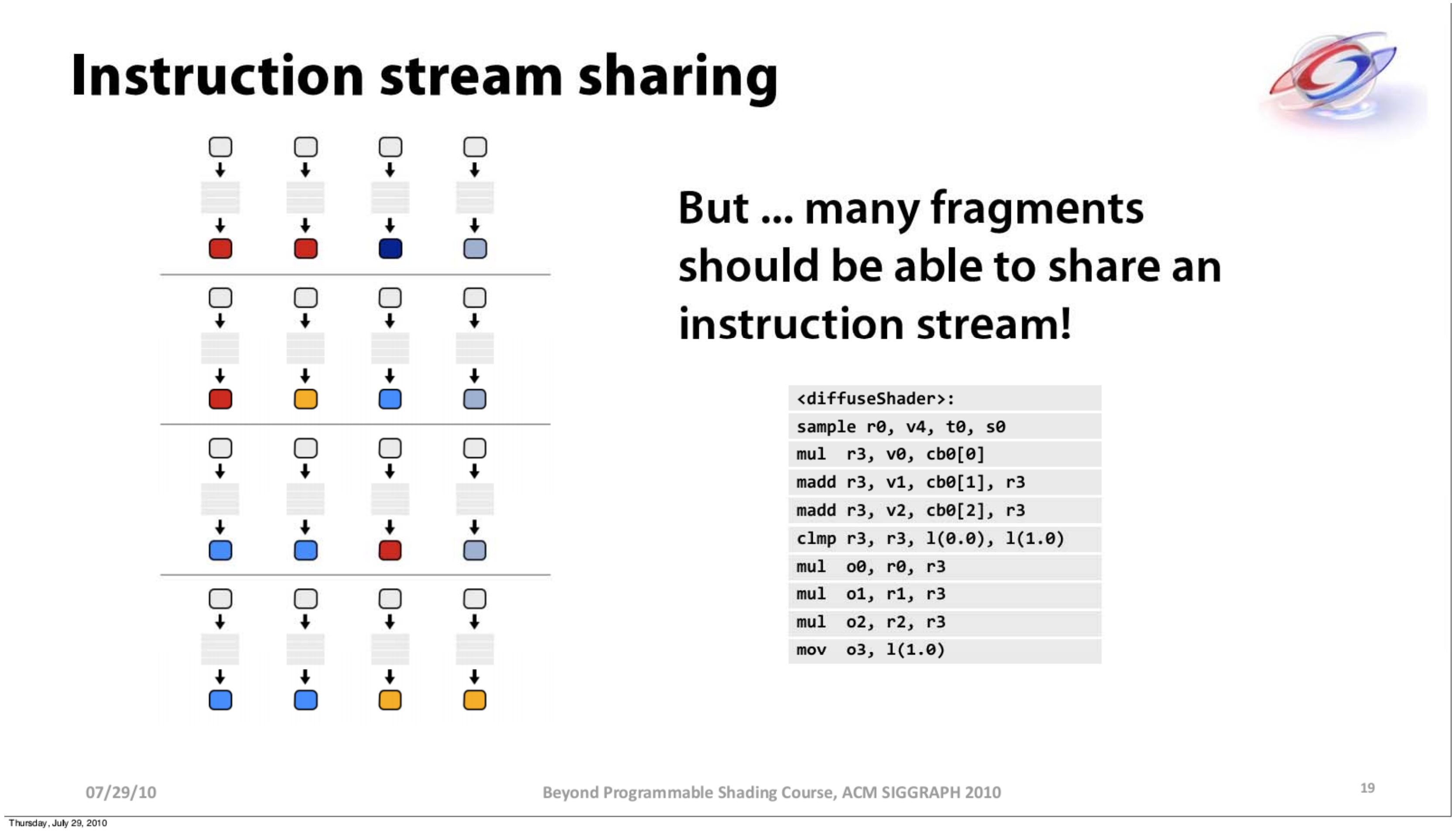
Copy and Paste 4 Times:



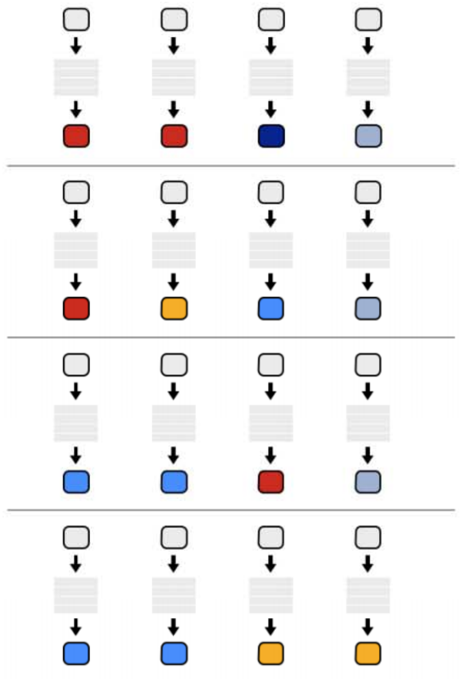
Sixteen cores (sixteen fragments in parallel)

16 cores = 16 simultaneous instruction streams

Sharing for Saving



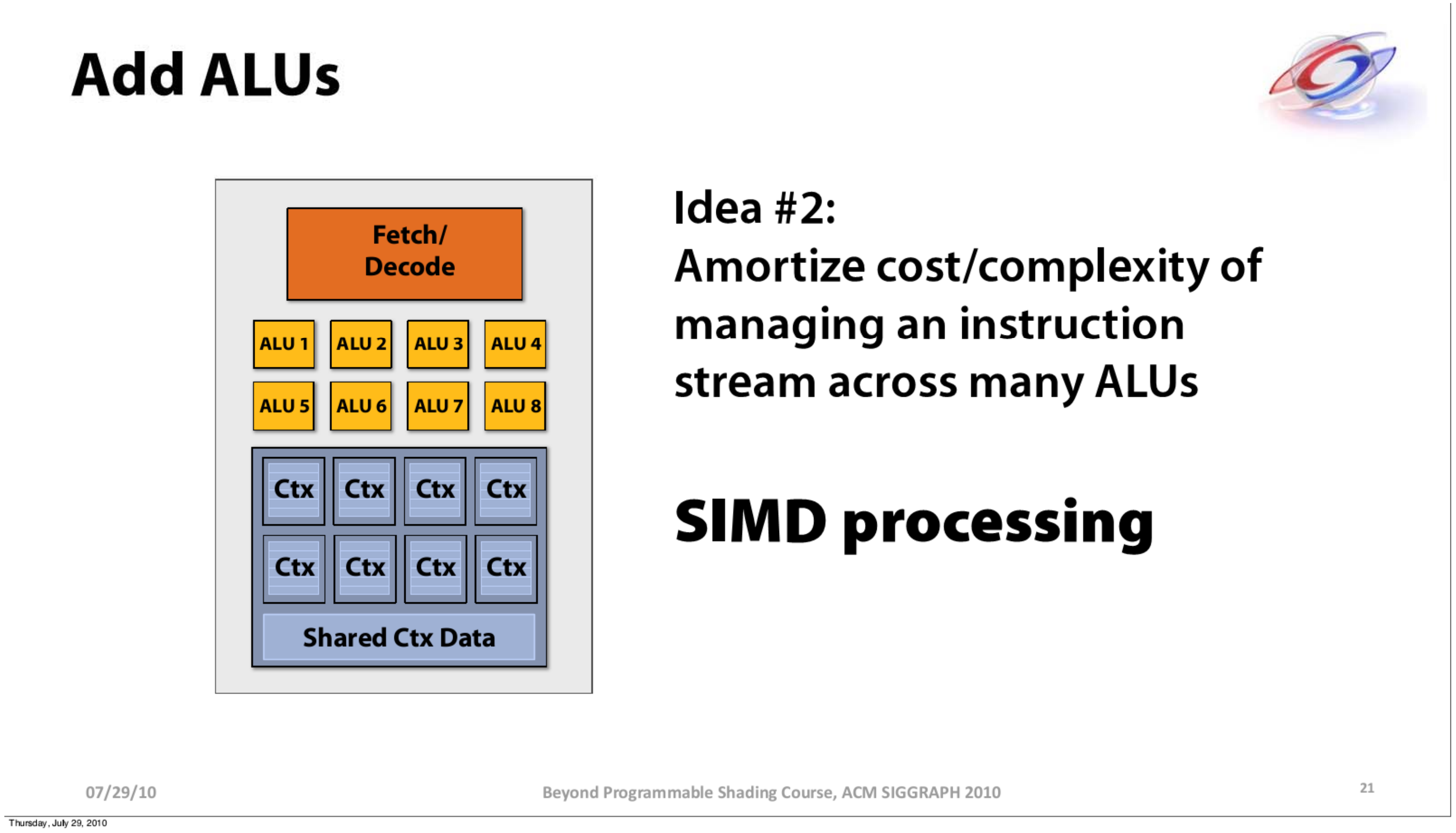
Instruction stream sharing



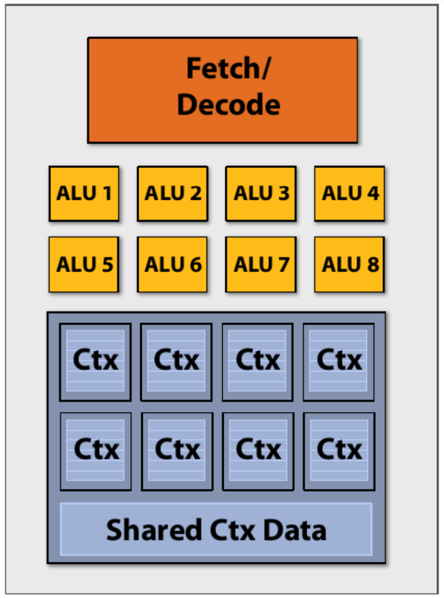
But ... many fragments should be able to share an instruction stream!

```
<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)
```

Idea 2:



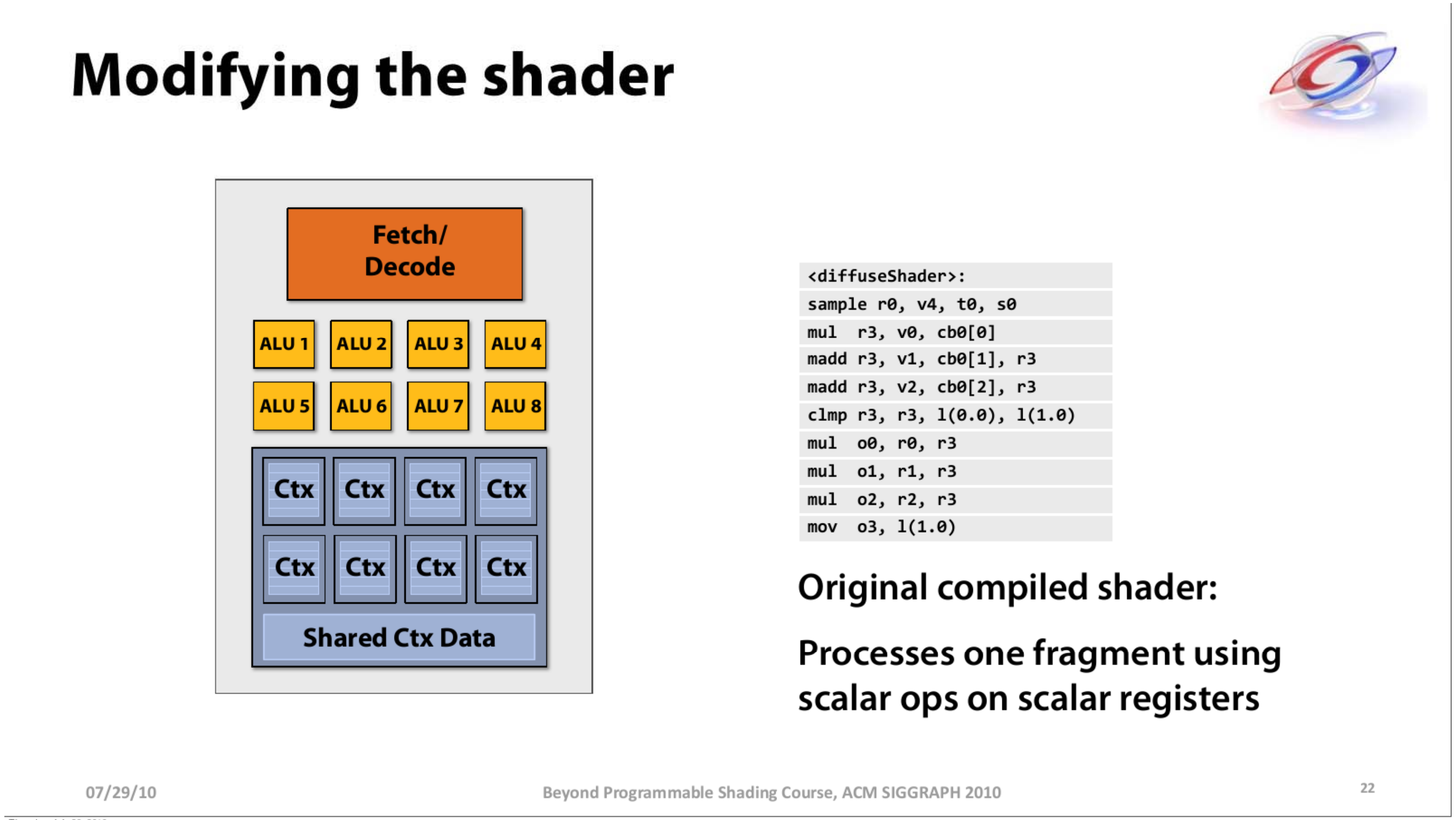
Add ALUs



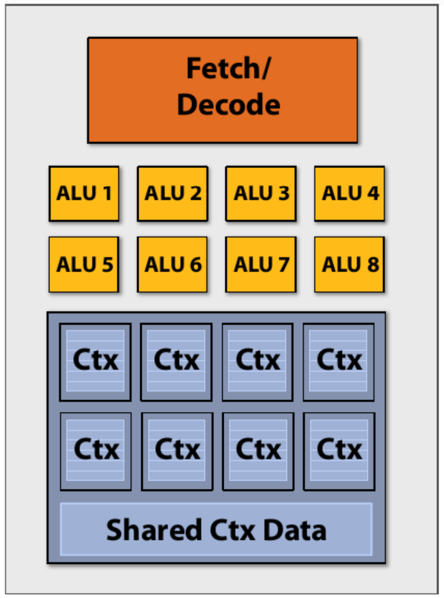
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Improve the Design



Modifying the 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)
```

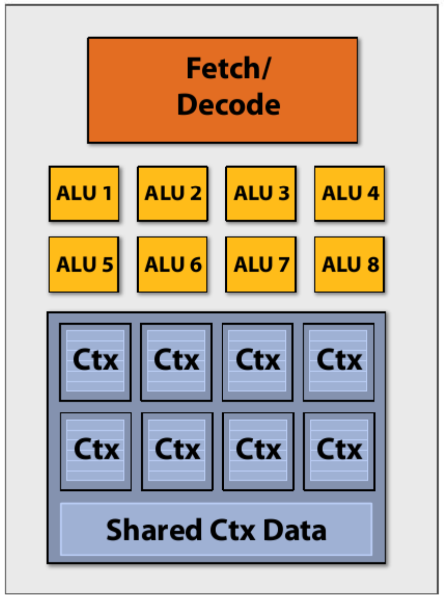
Original compiled shader:

Processes one fragment using scalar ops on scalar registers

Thursday, July 29, 2010

Vector Instructions:

Modifying the shader



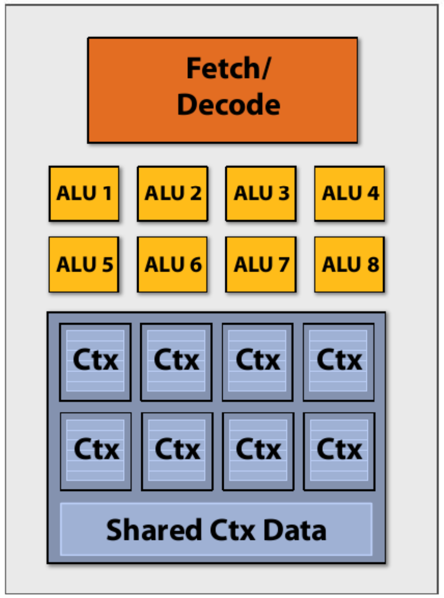
```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  o3, l(1.0)
```

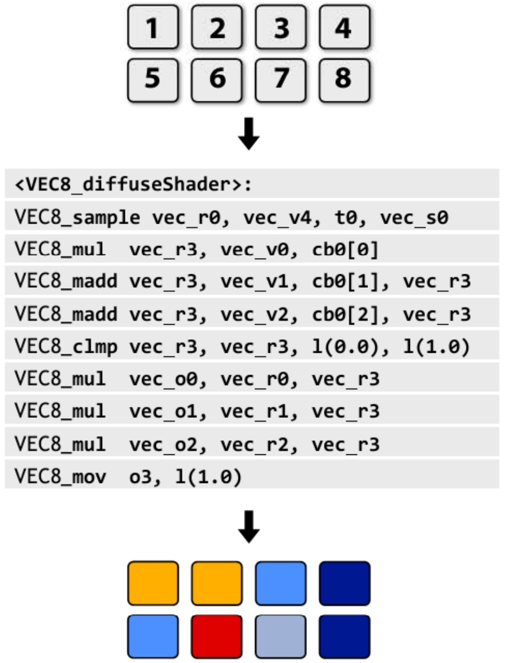
New compiled shader:

Processes eight fragments using vector ops on vector registers

Multiple Operands

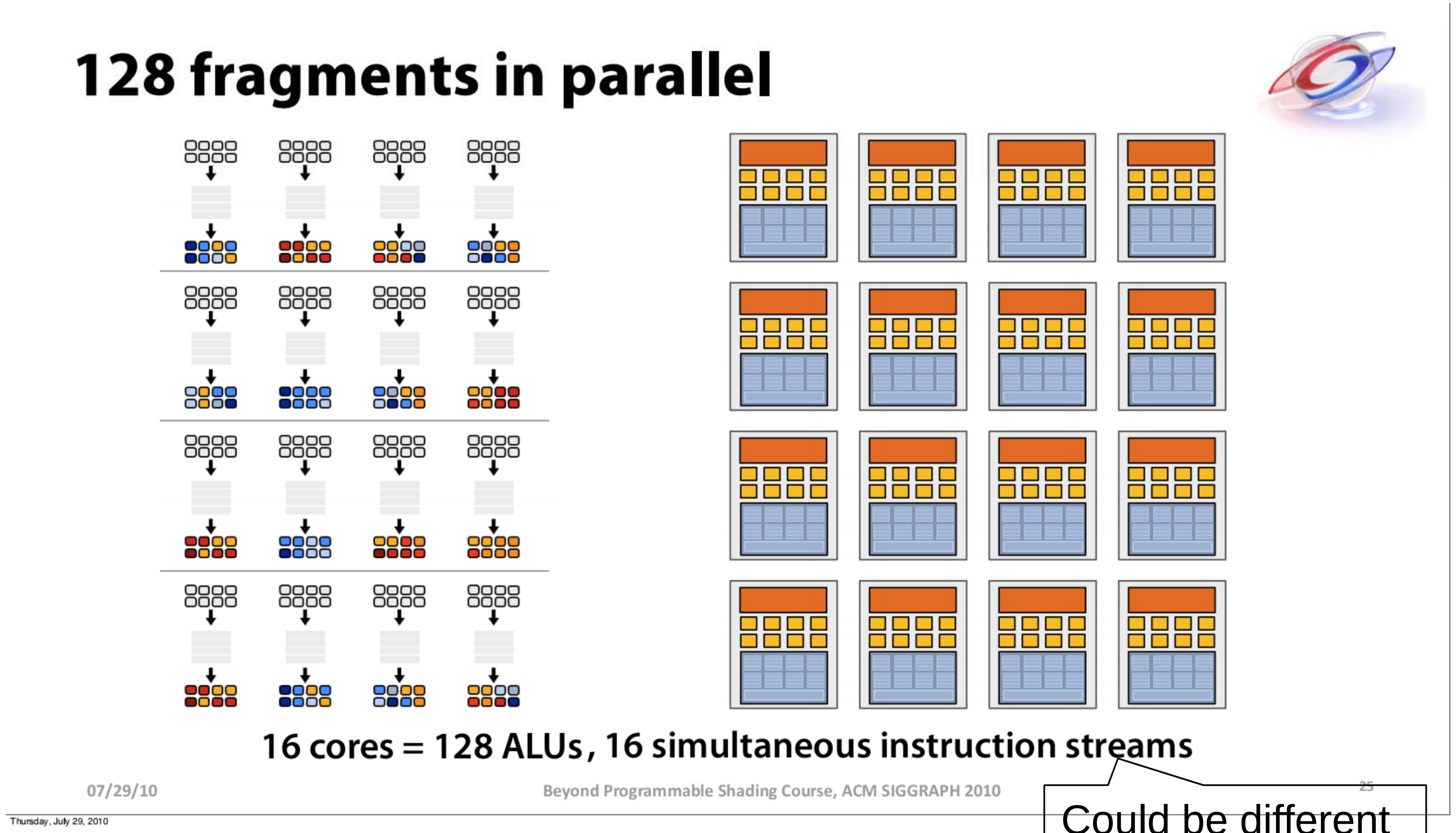
Modifying the shader





```
<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)
```

More fragments, More Simultaneous Instructions



128 fragments in parallel

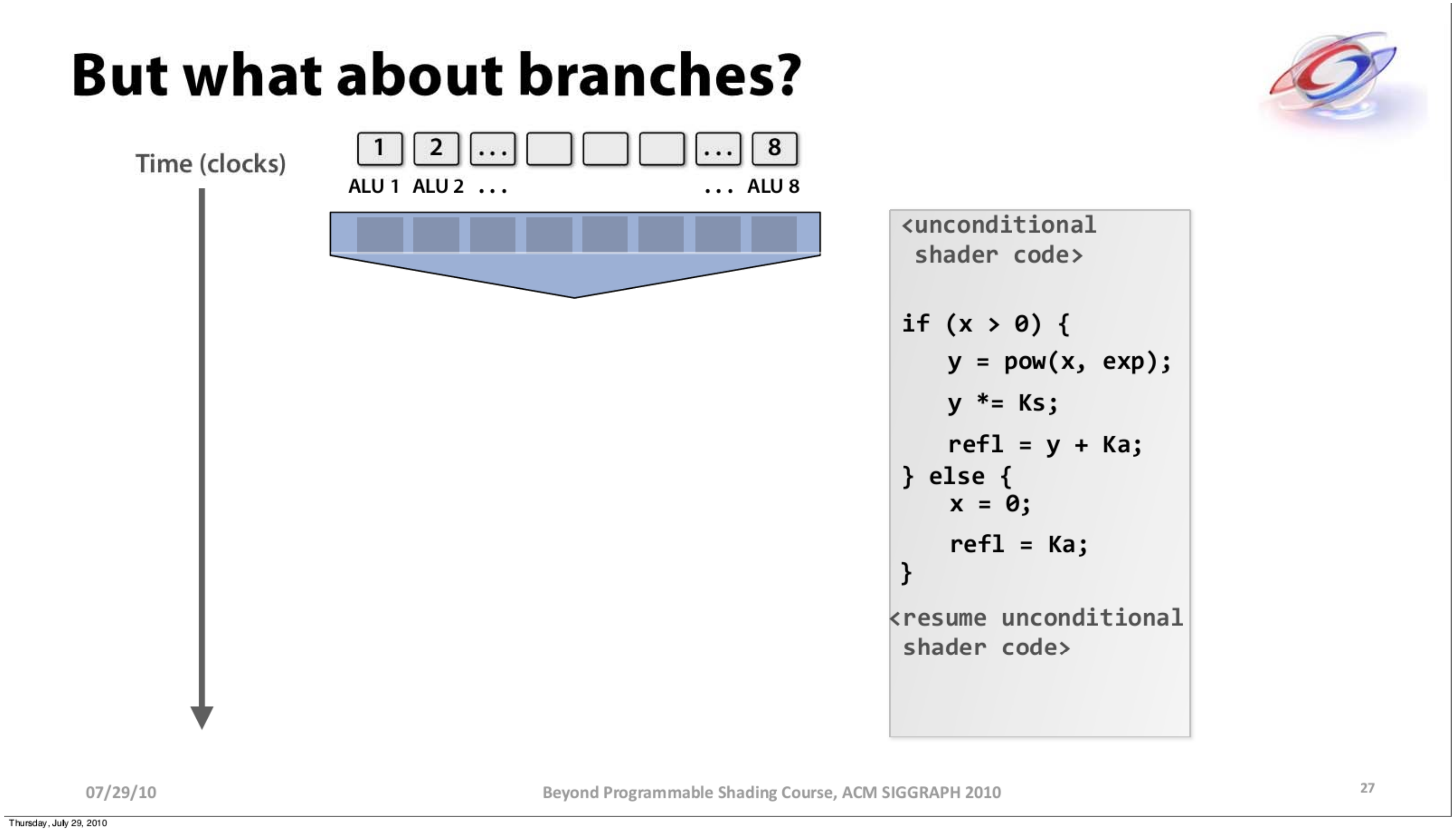
16 cores = 128 ALUs, 16 simultaneous instruction streams

07/29/10 Beyond Programmable Shading Course, ACM SIGGRAPH 2010 25

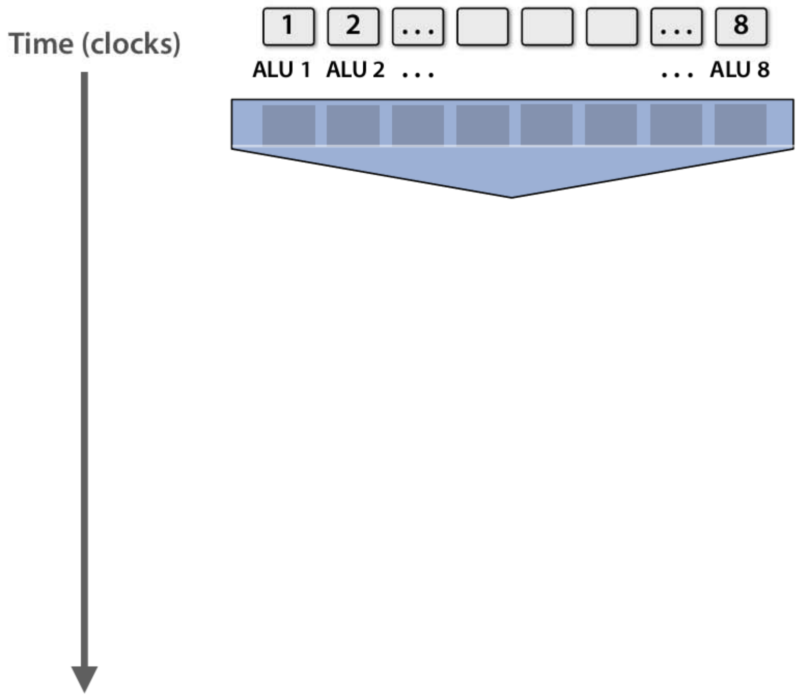
Thursday, July 29, 2010

Could be different

Sharing Dispatcher

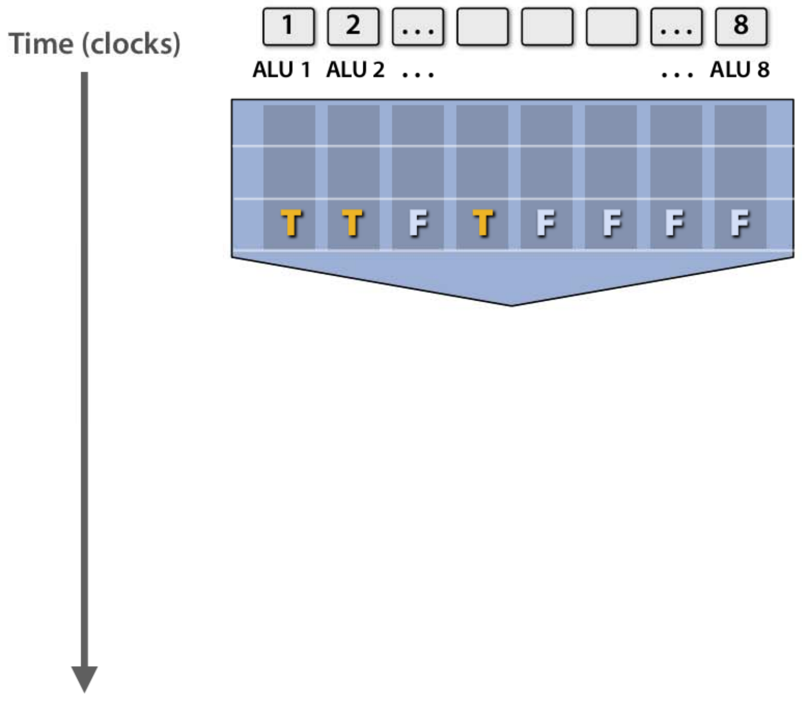


But what about branches?



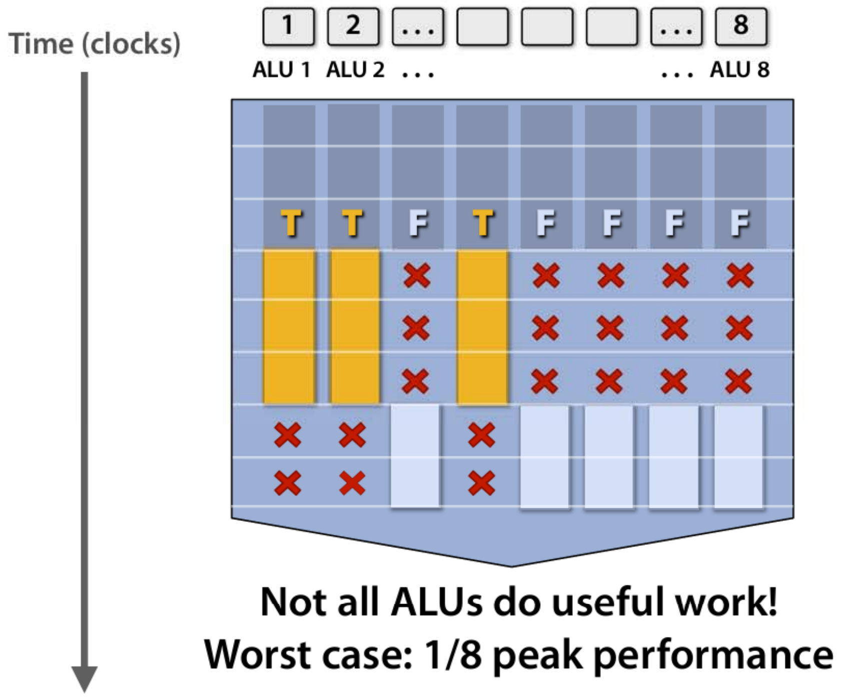
```
<unconditional
 shader code>

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?



```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

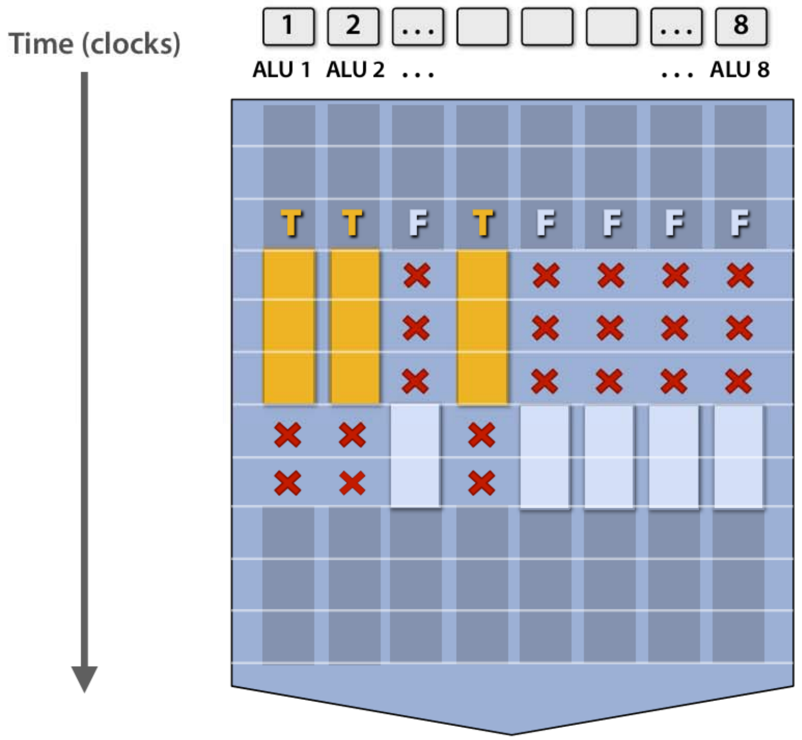
But what about branches?



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

```
<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?

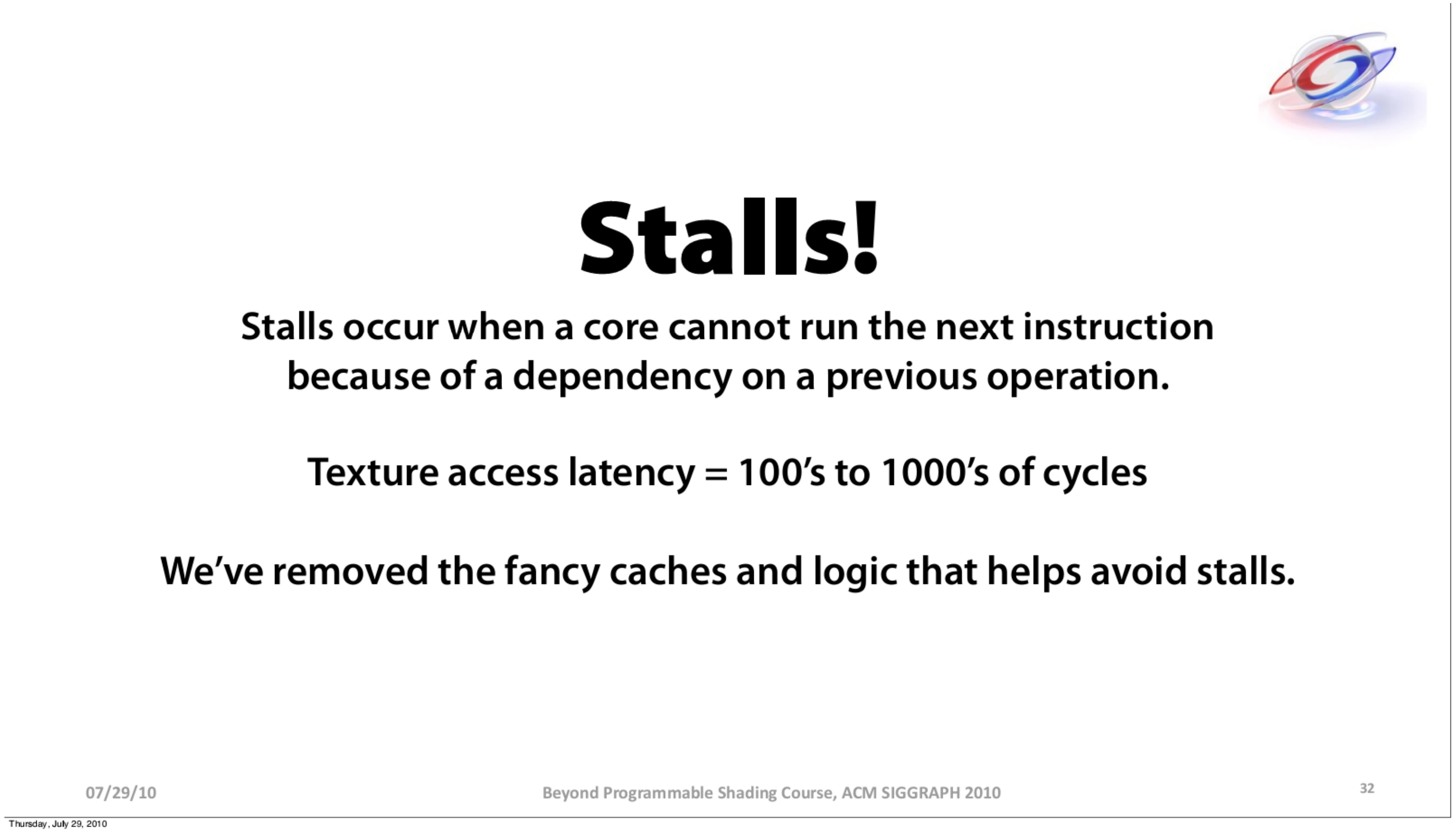


```
<unconditional
 shader code>

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

<resume unconditional
 shader code>
```

Clarification

- SIMD Processing doesn’t imply SIMD instructions
- Option 1: Explicit Vector Instructions:
  - SSE, AVX and etc.
- Option 2: Scalar instructions, implicit HW vectorization
  - Hardware decides instruction stream sharing across scalar ALUs
  - NVIDIA, AMD GPUs

Problems and Challenges



Stalls!

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

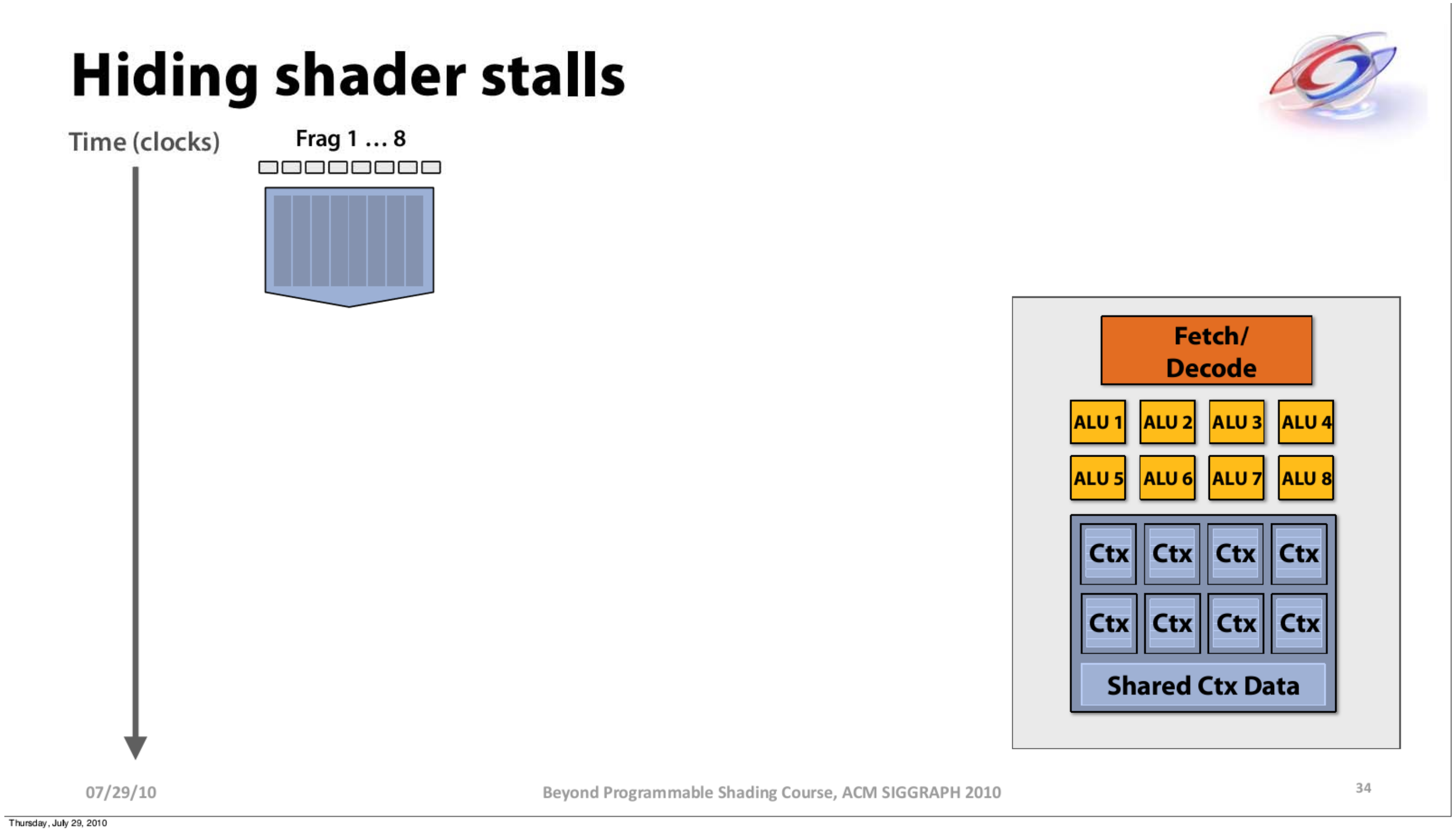
Texture access latency = 100’s to 1000’s of cycles

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

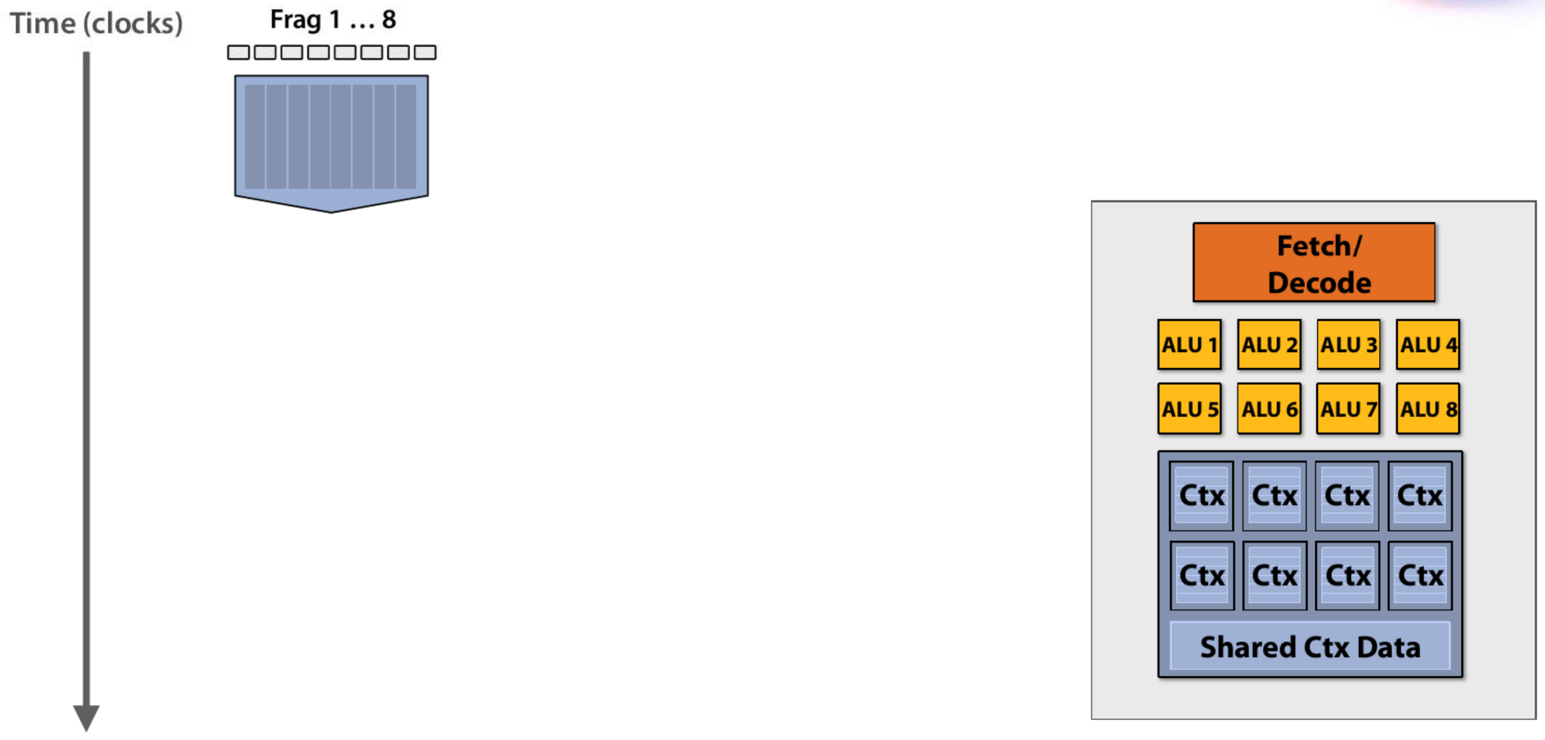
Solution: Idea 3

- Lots of independent fragments switching
- Interleave processing of many program slices on ALUs to avoid the stalls caused by high latency operations—like “Fill in the blanks”

Latency Hiding

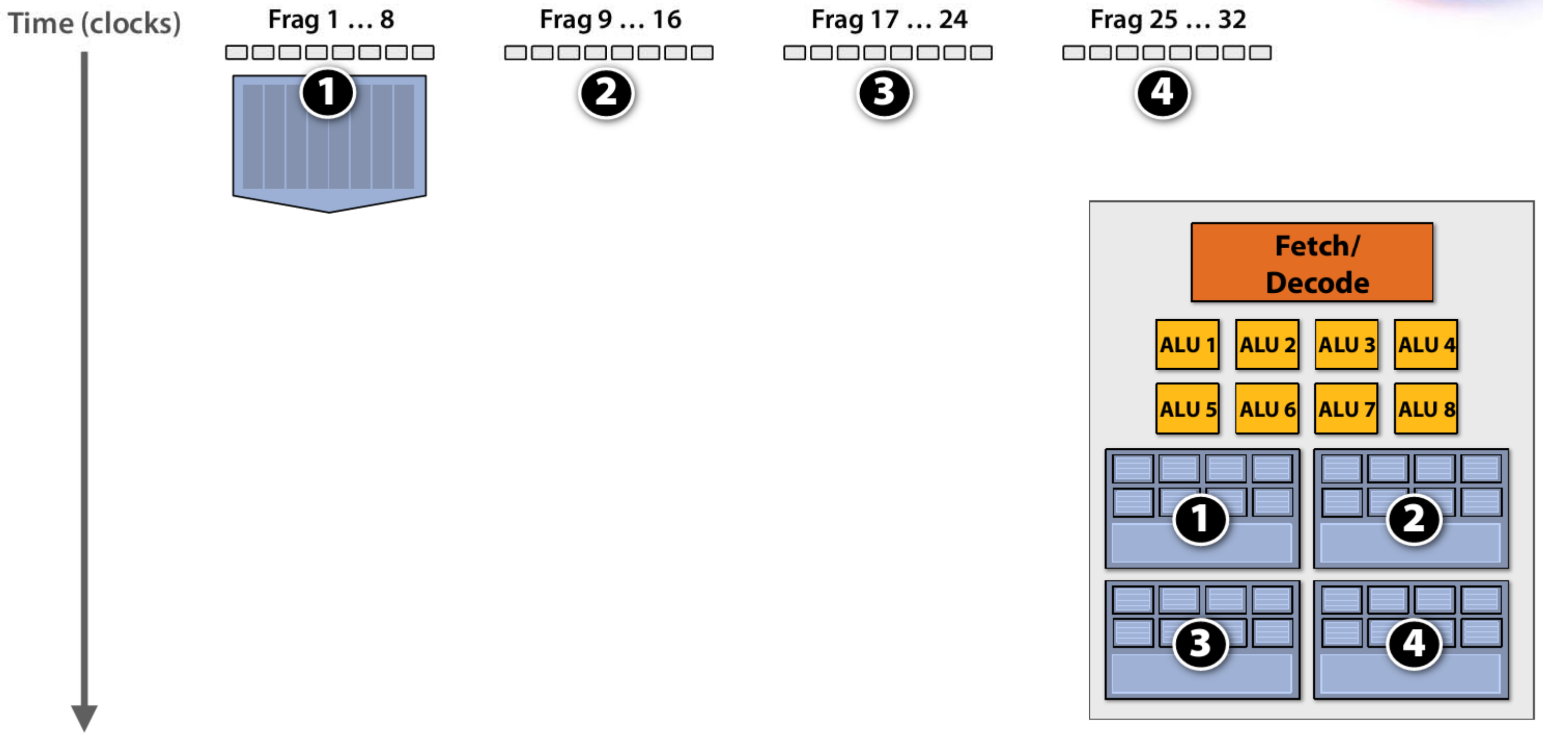


Hiding shader stalls

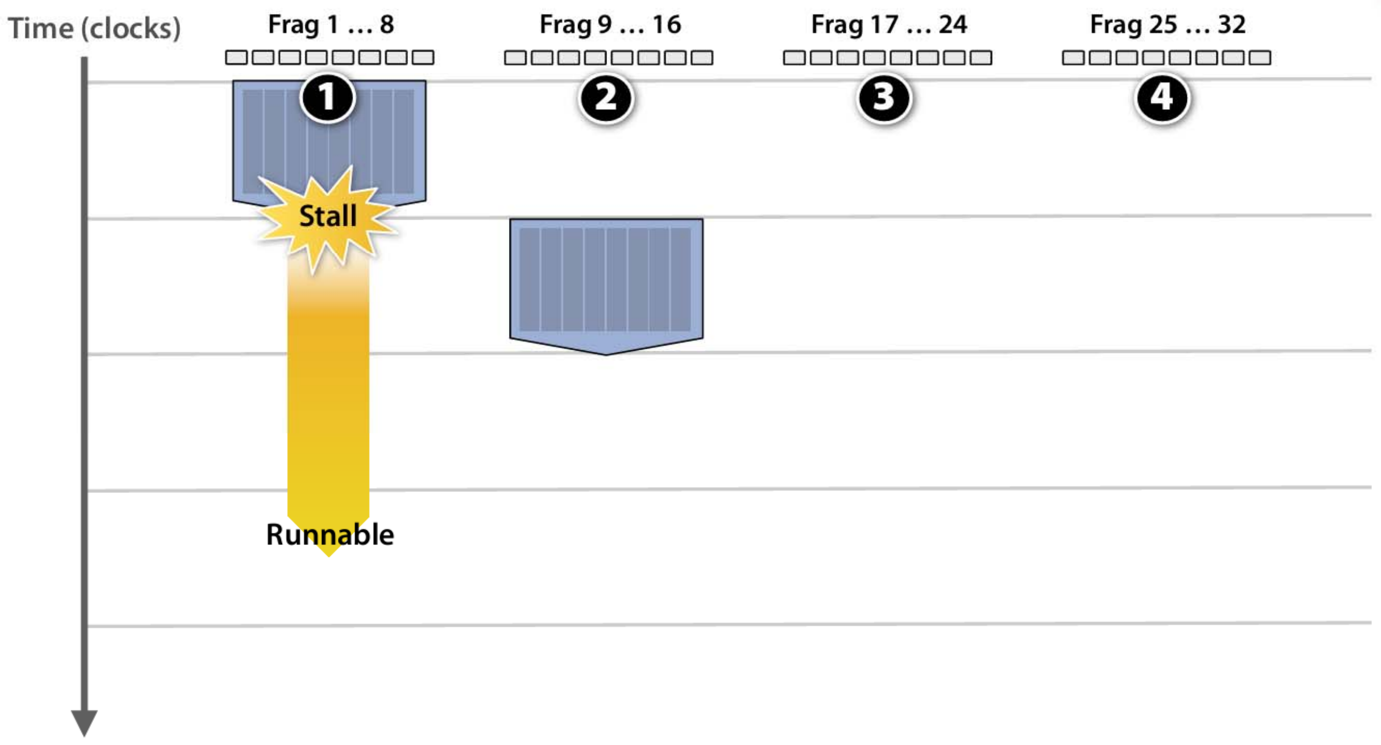


Thursday, July 29, 2010

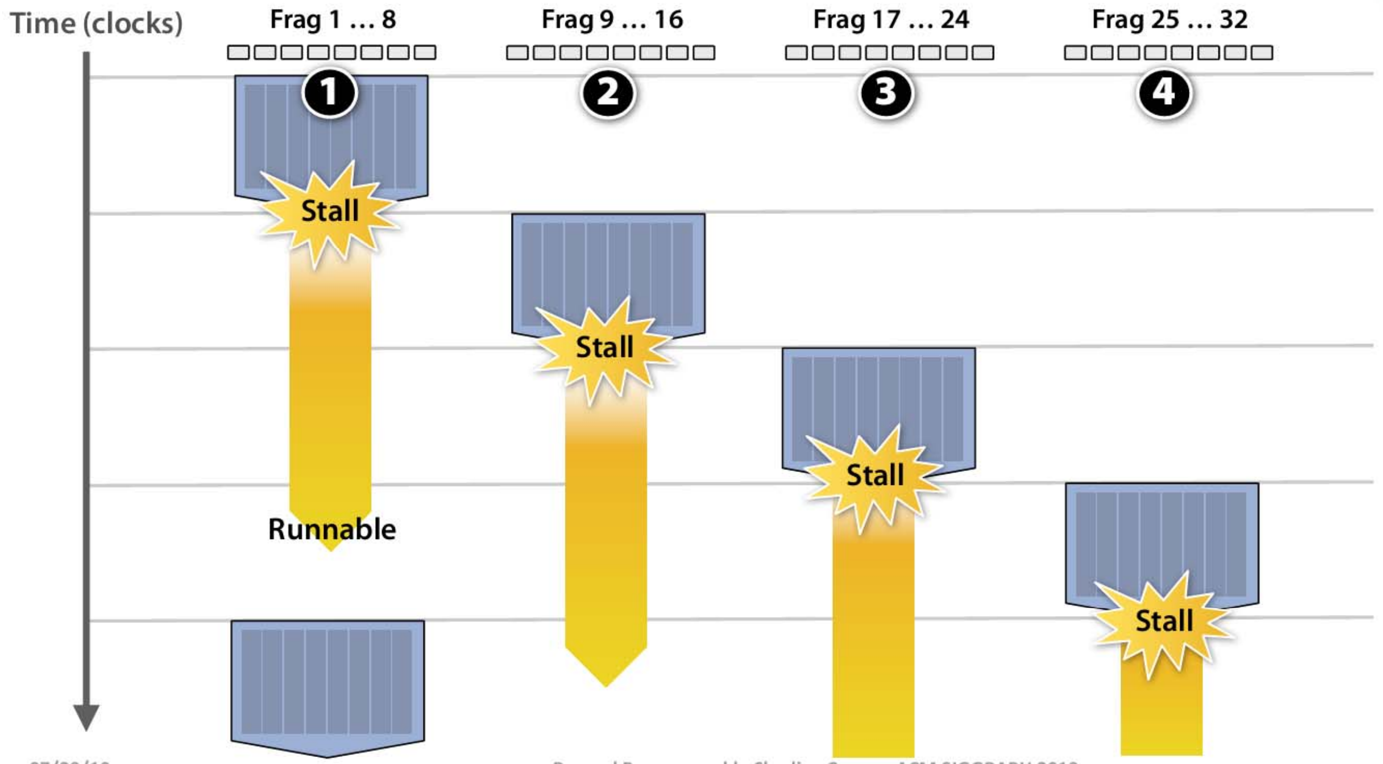
Hiding shader stalls



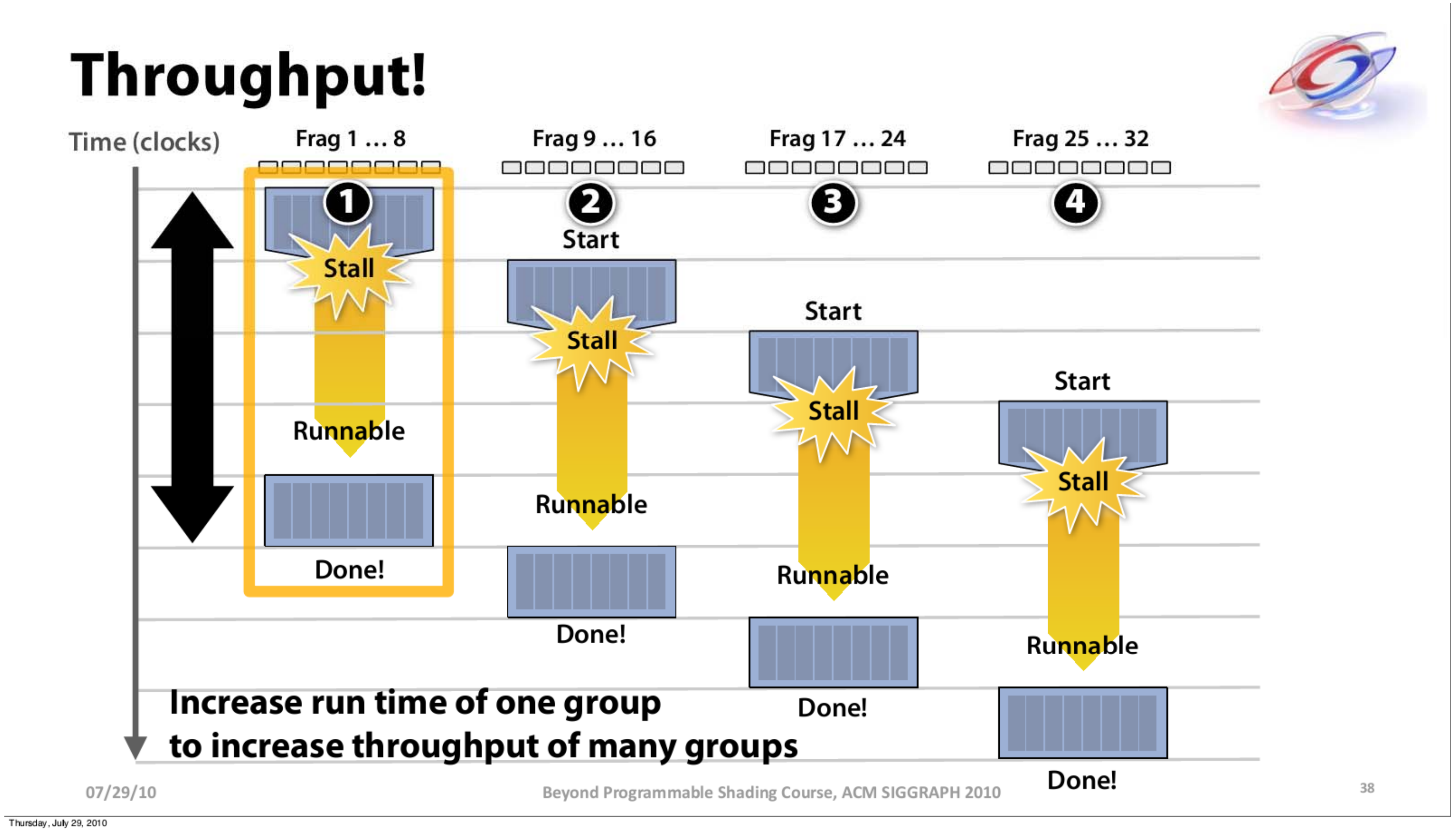
Hiding shader stalls



Hiding shader stalls



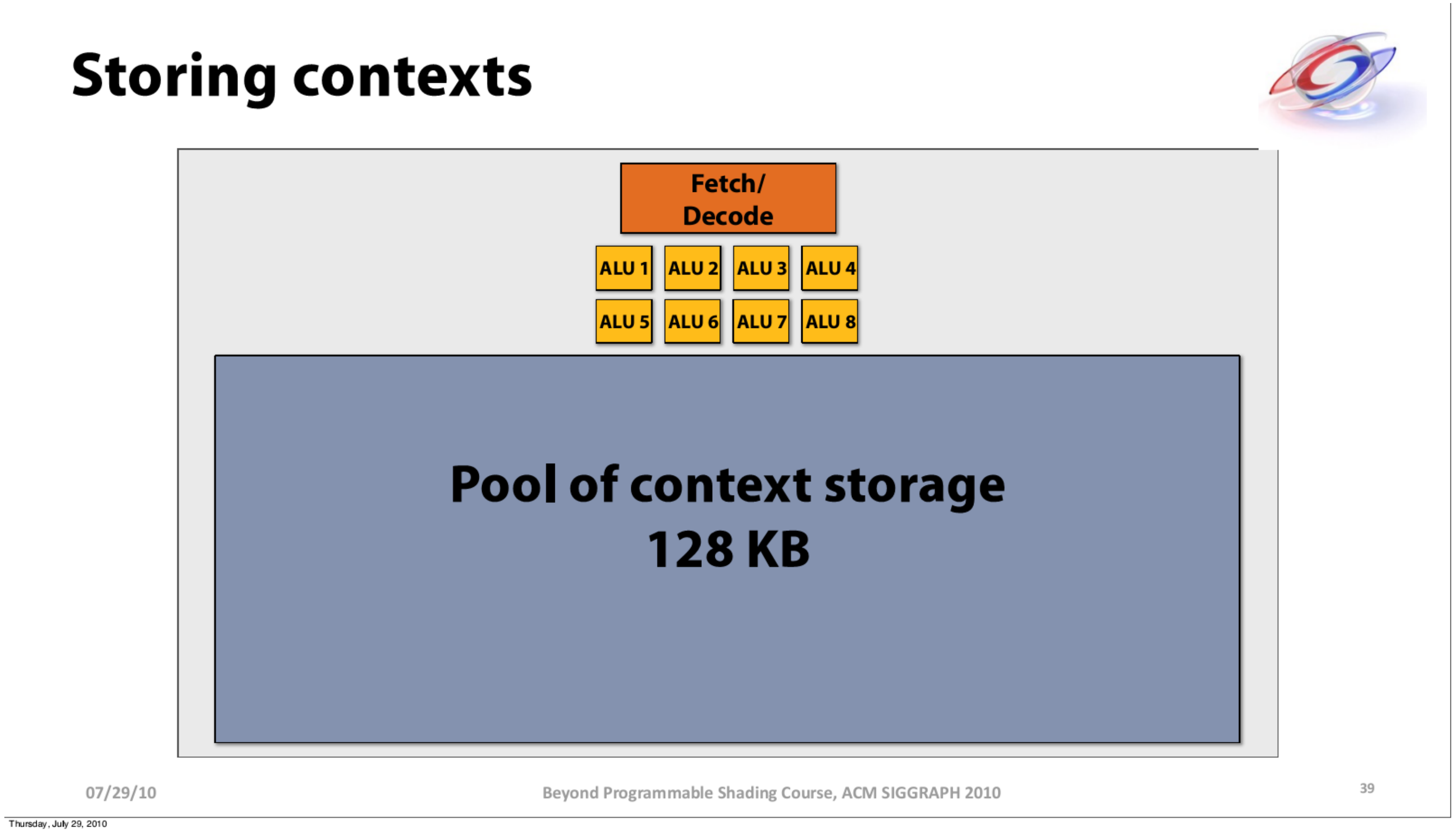
Gain a high throughput



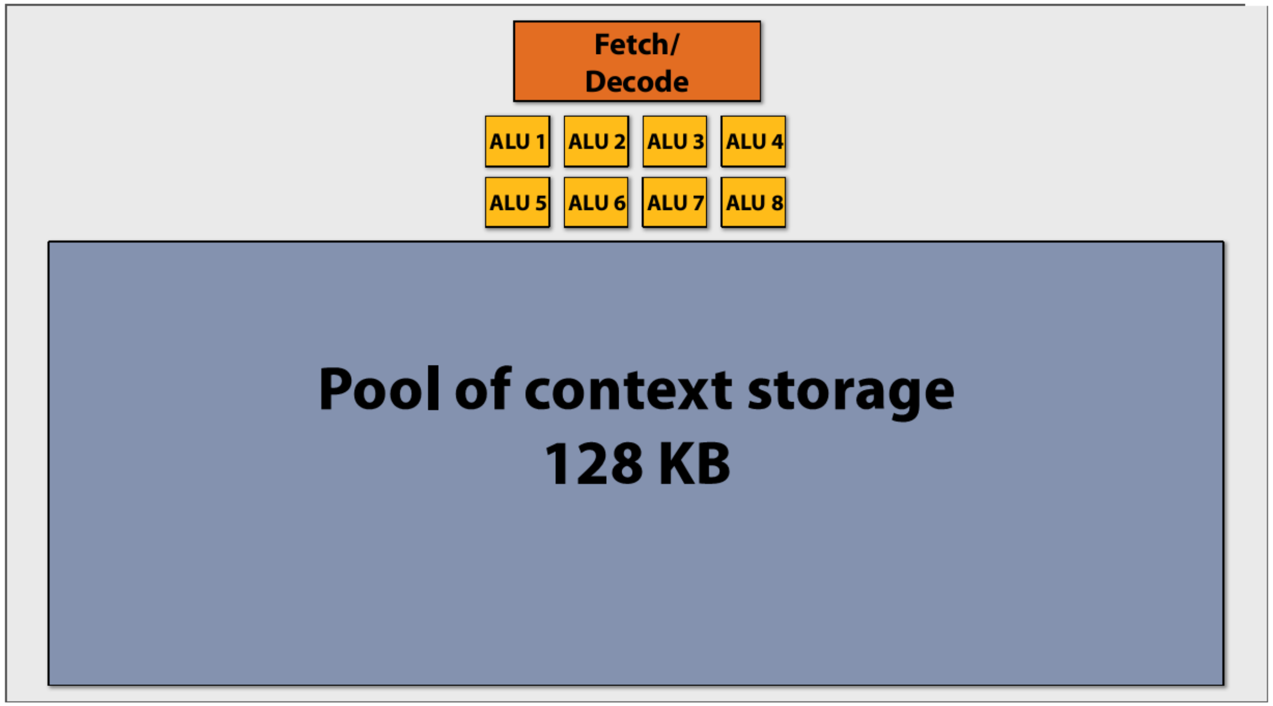
Throughput!

Increase run time of one group to increase throughput of many groups

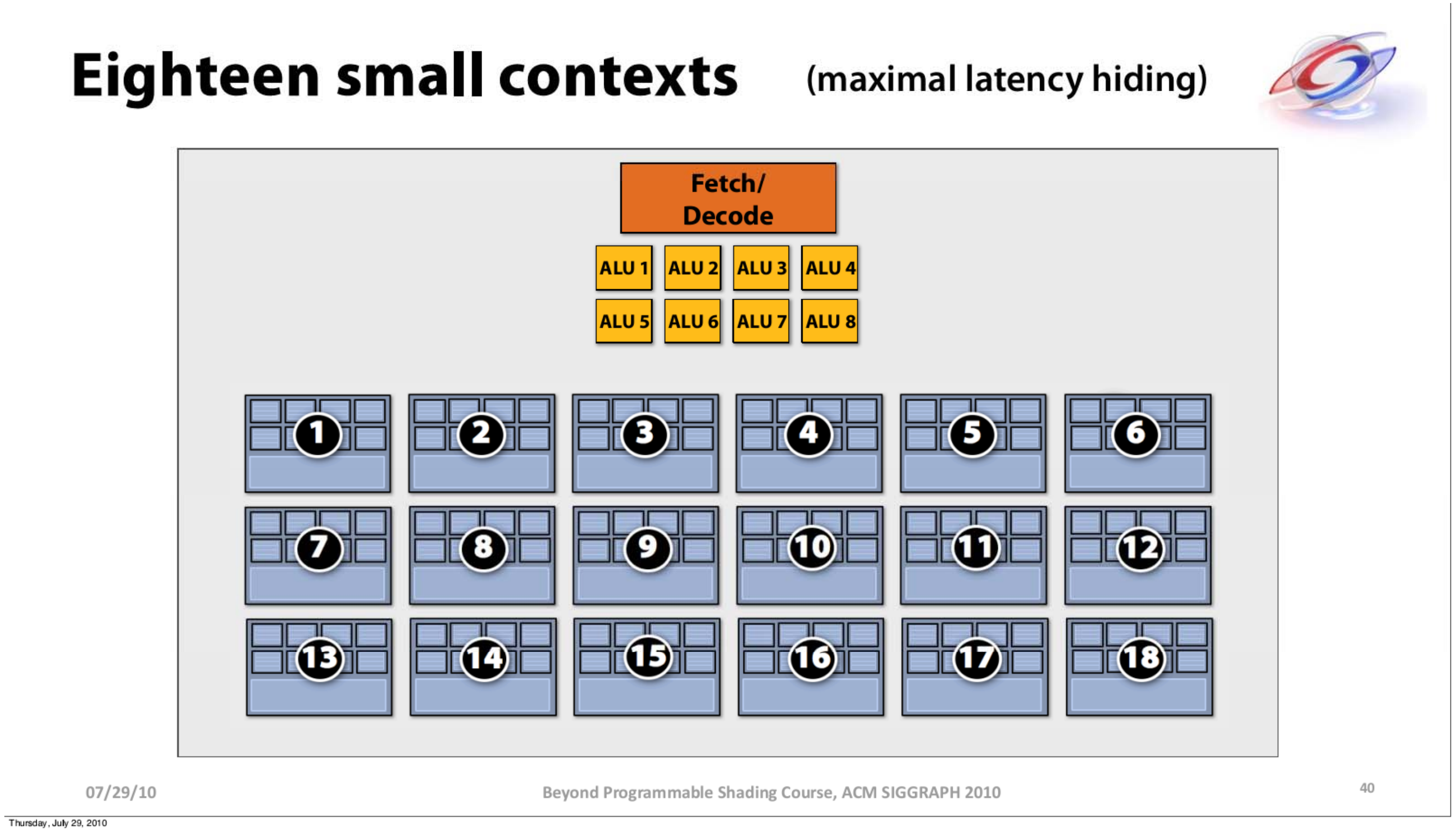
Dealing with the Context



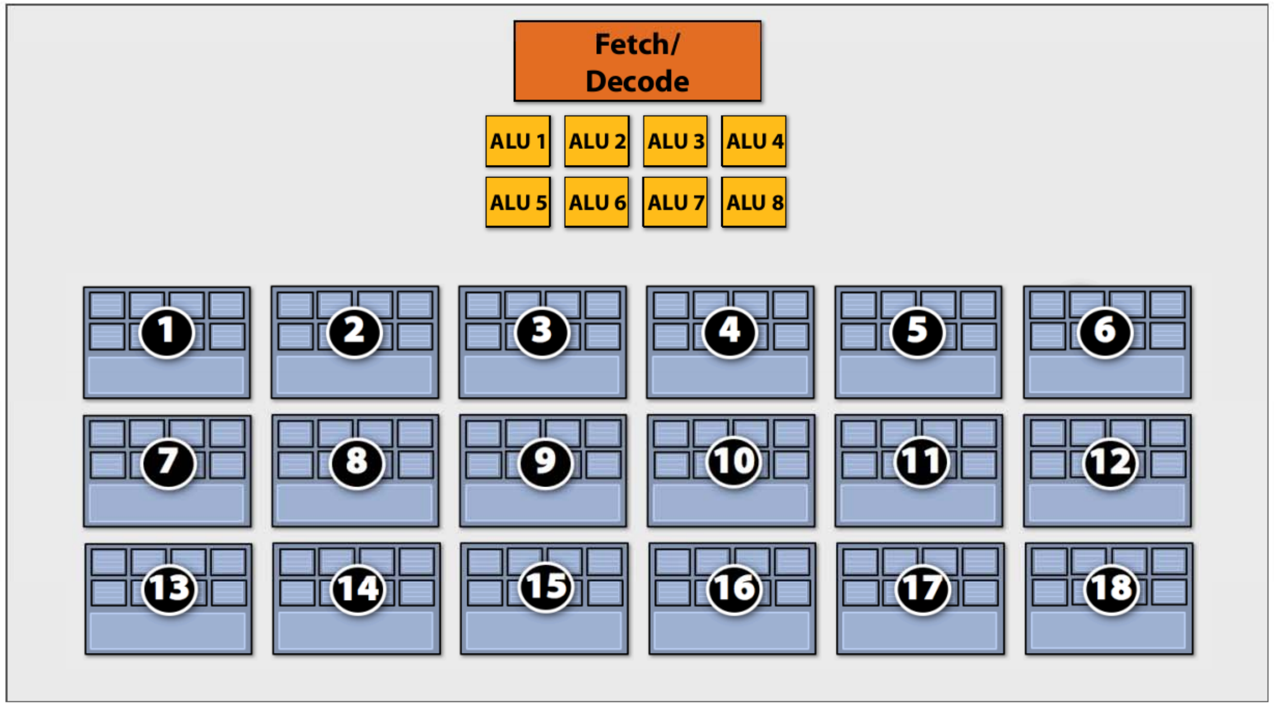
Storing contexts



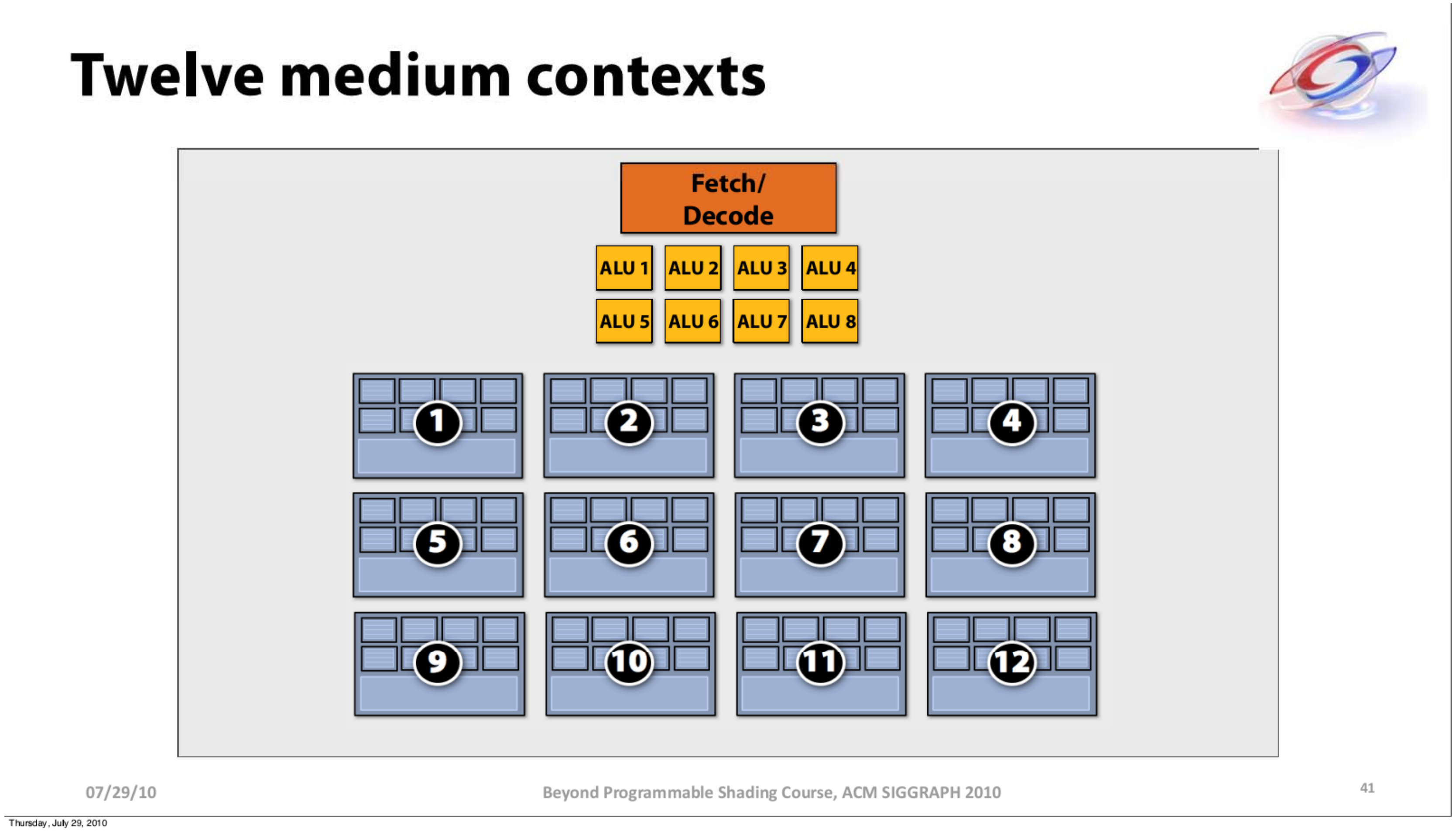
Smaller contexts: Pros: Better latency hiding; Cons: ?



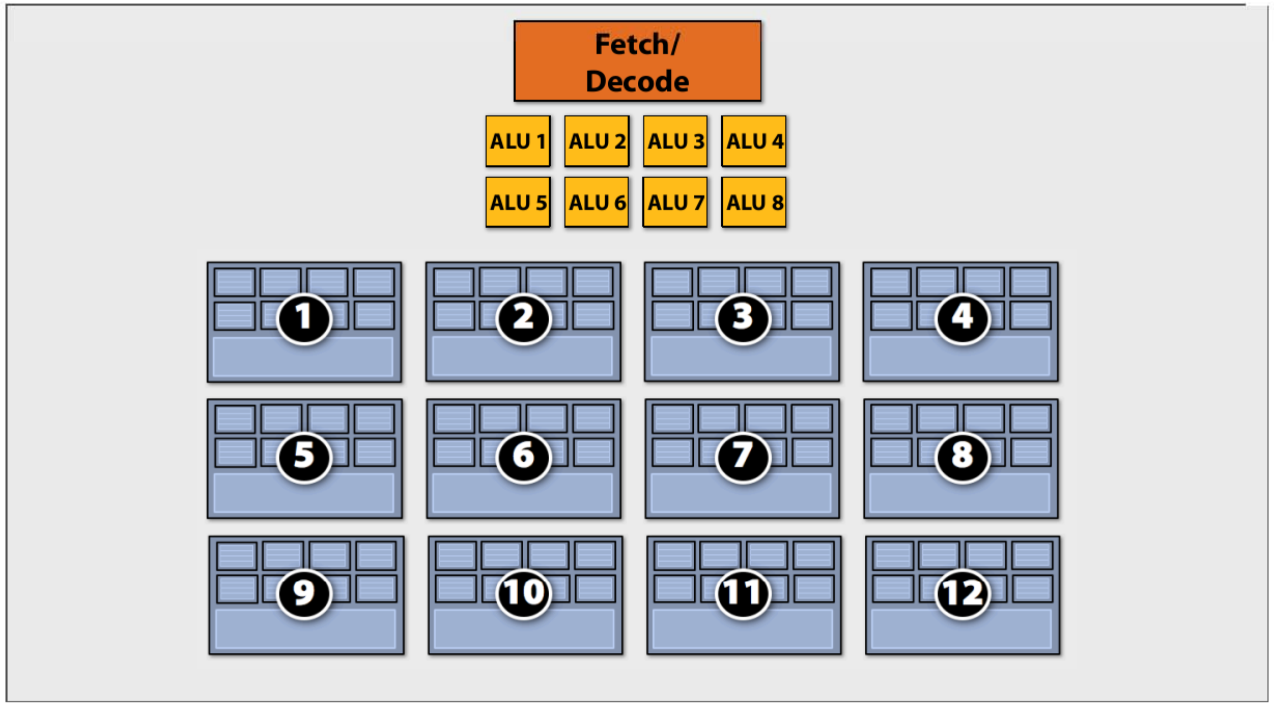
Eighteen small contexts (maximal latency hiding)



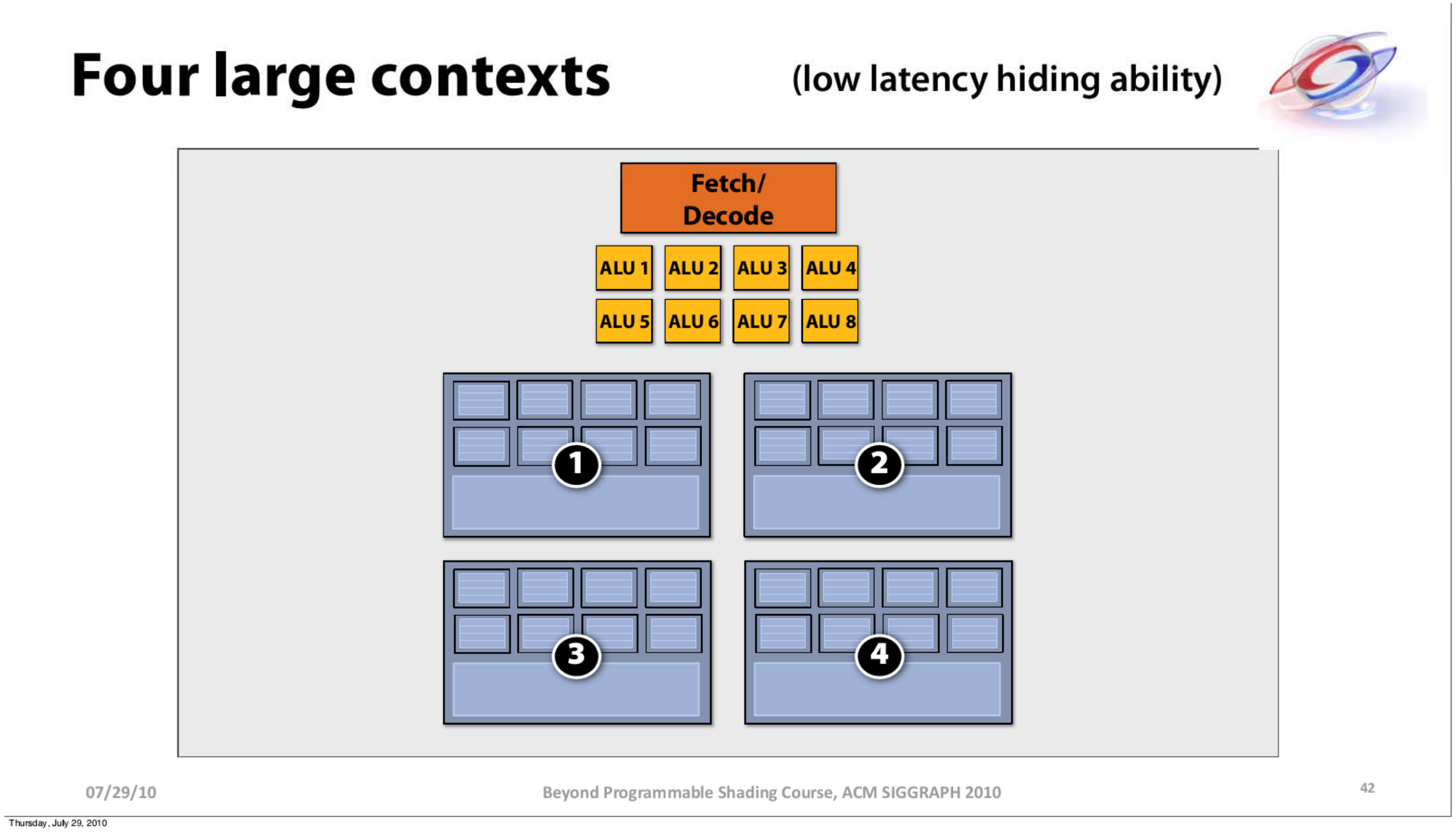
Medium Contexts: Pros:? Cons:?



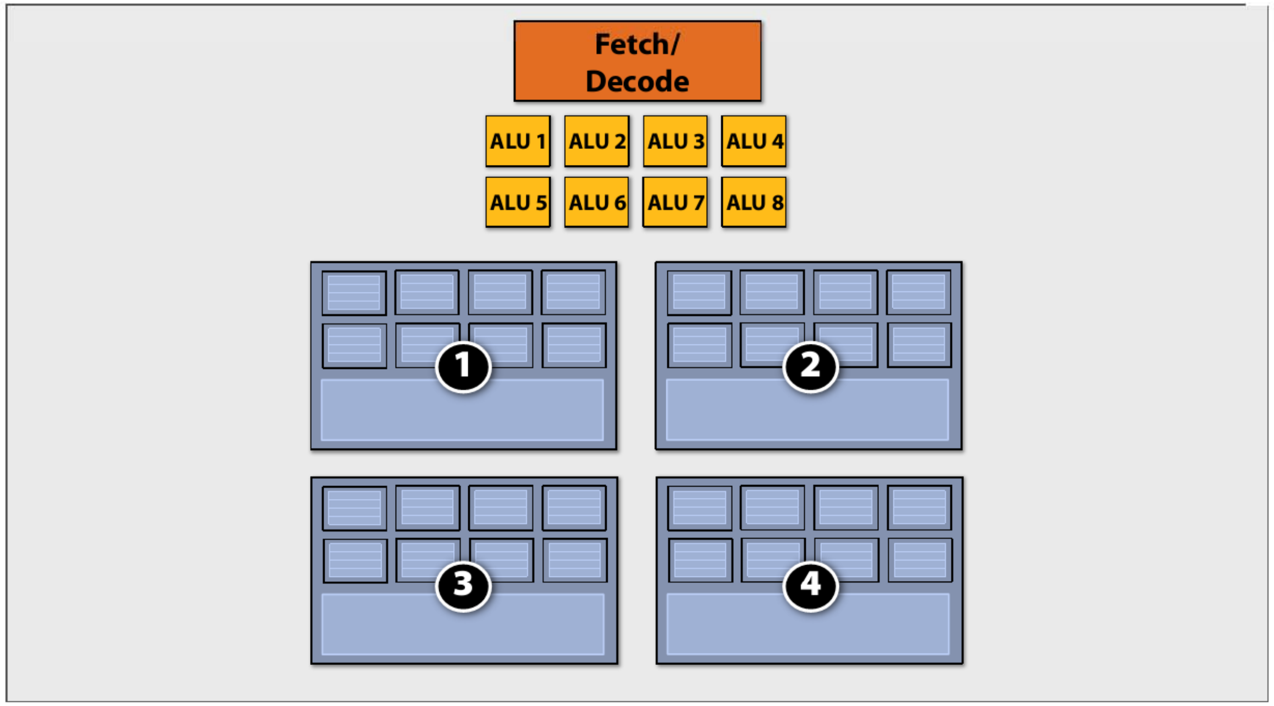
Twelve medium contexts



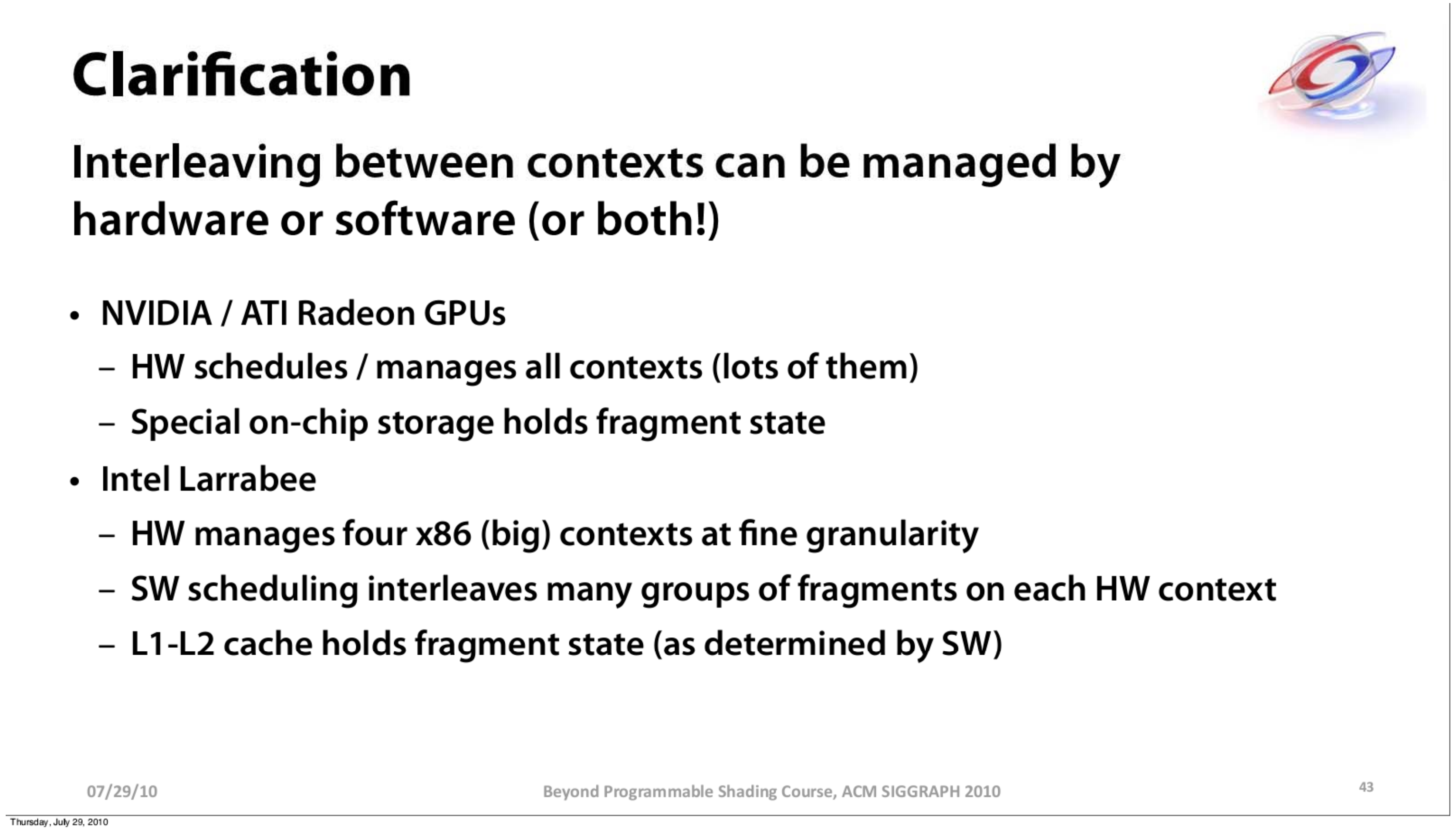
Large Contexts: Pros? Cons?



Four large contexts (low latency hiding ability)



The Tradeoff in Architecture Design



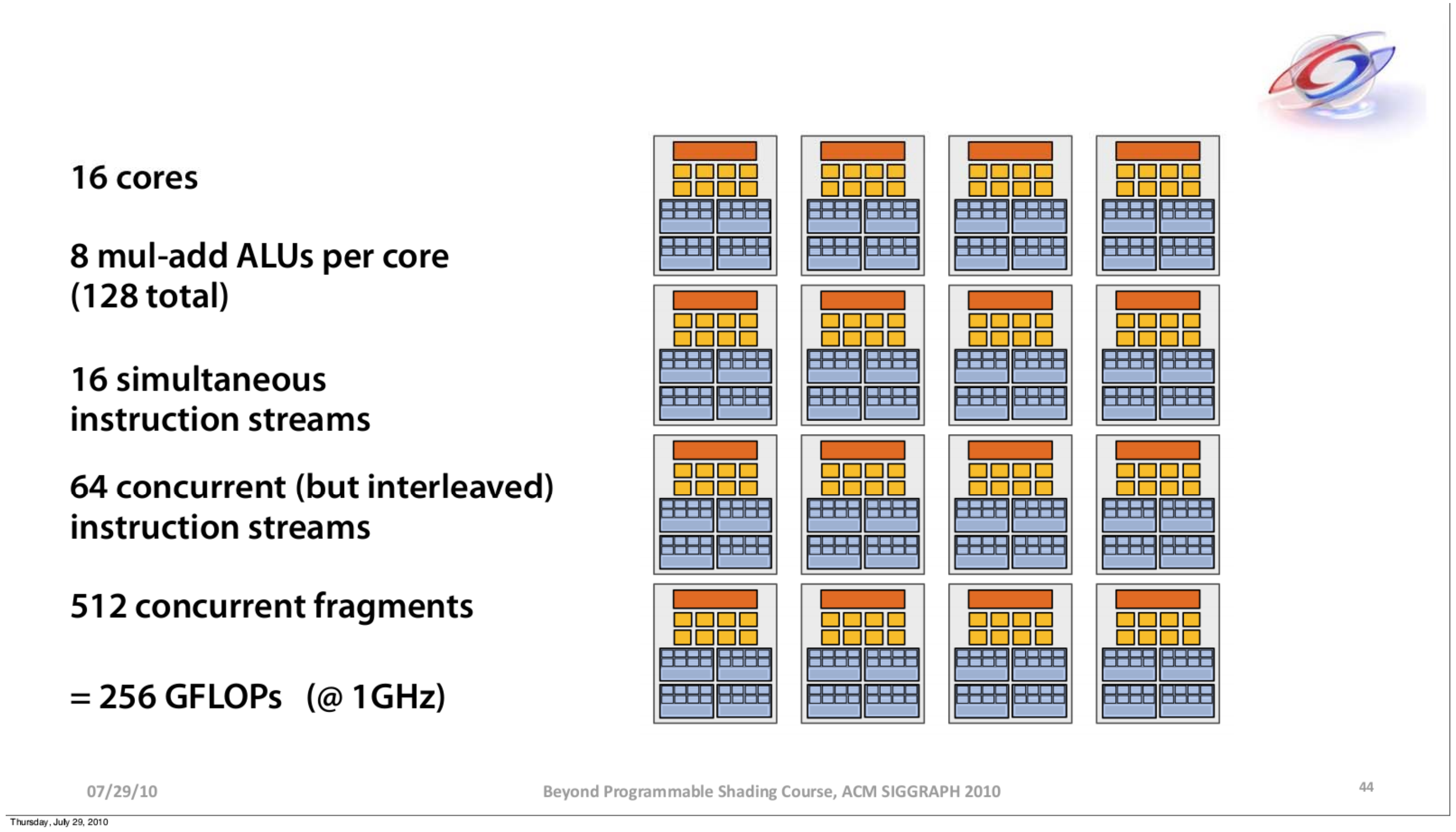
Clarification

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)

Thursday, July 29, 2010

Example Chip



16 cores

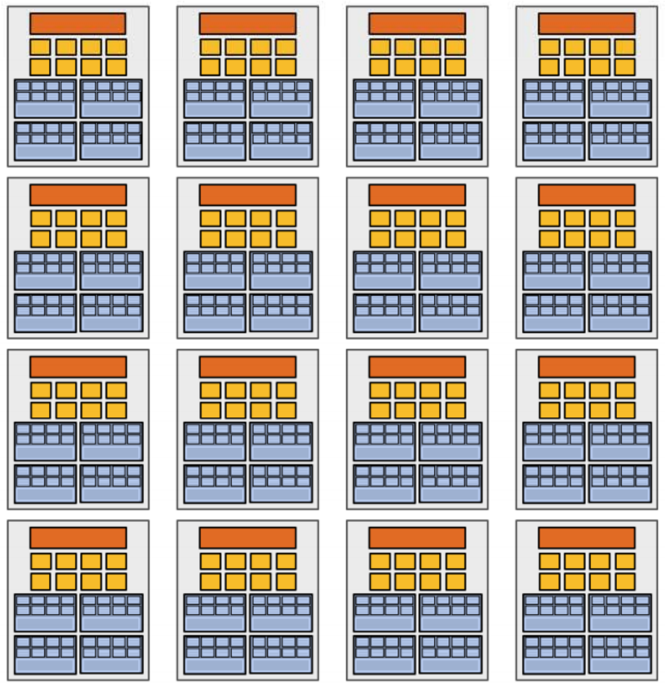
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

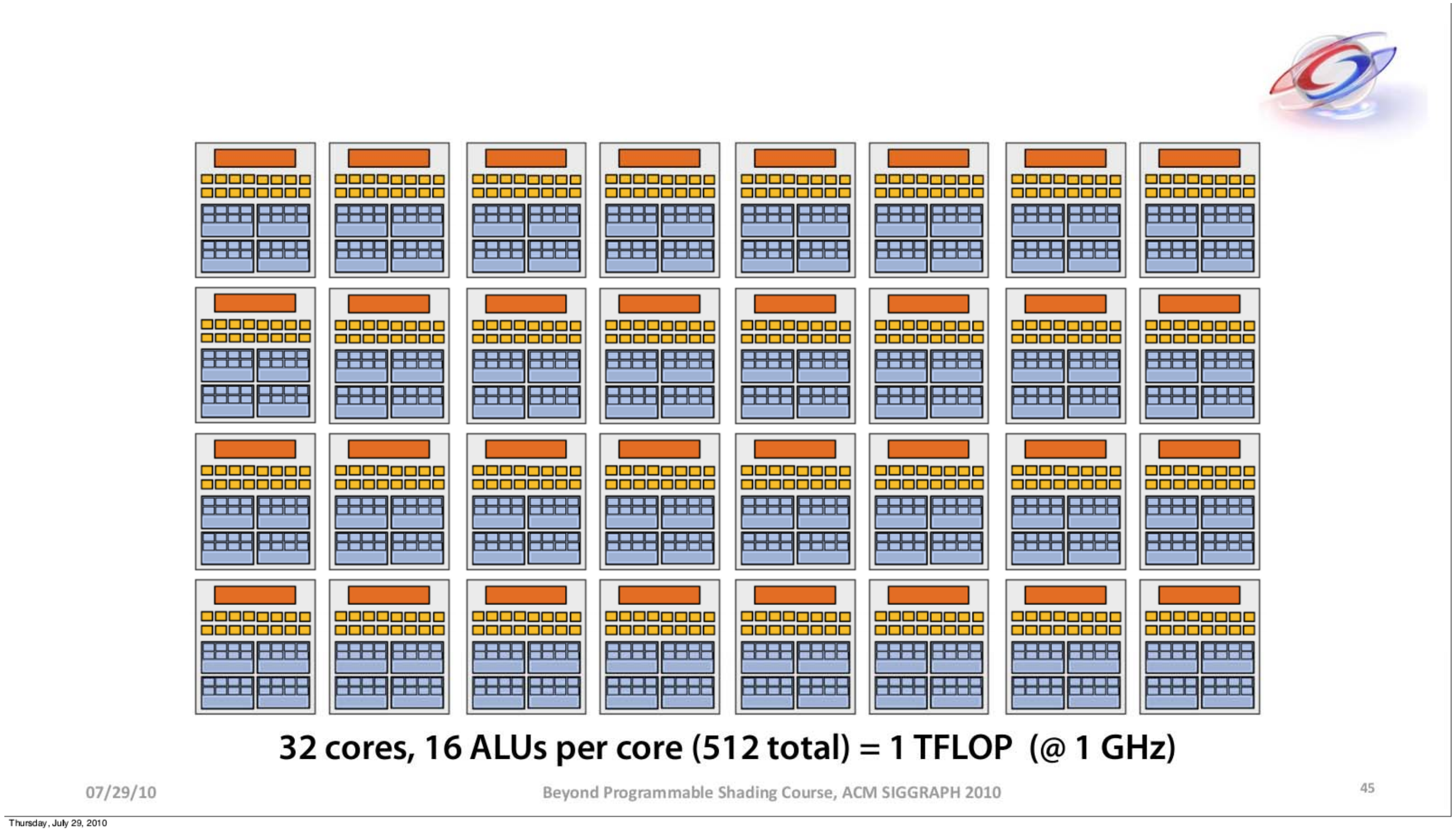
64 concurrent (but interleaved)
instruction streams

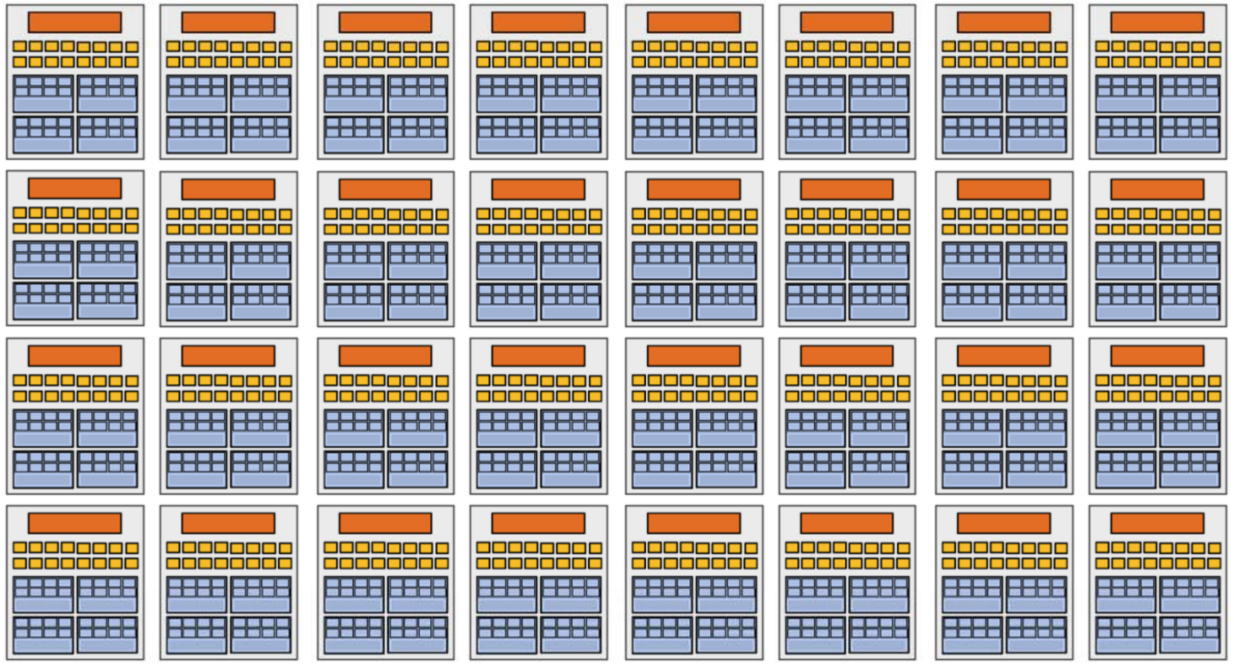
512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



Bigger Chip with higher performance





32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)

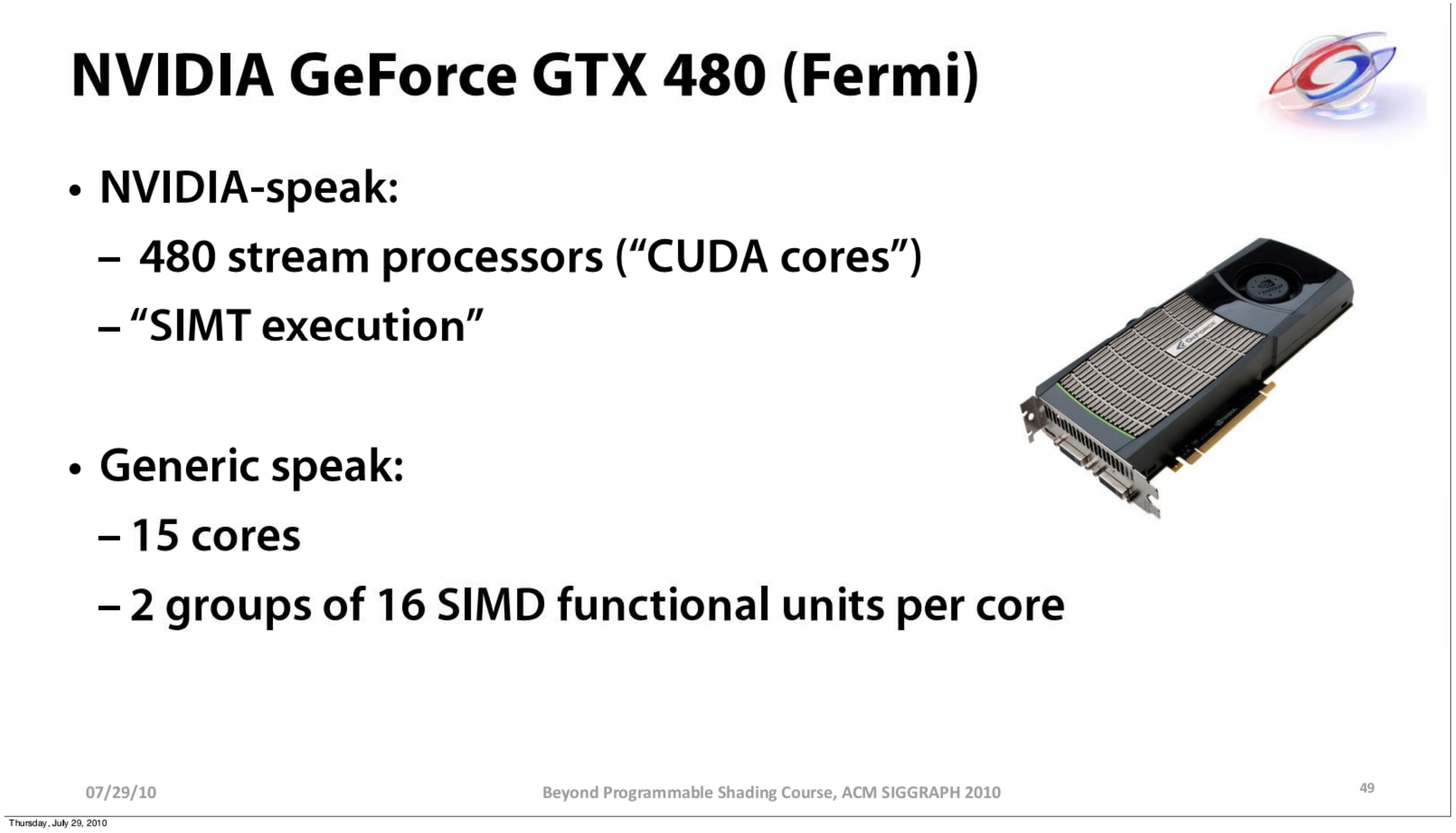
Summary: three key ideas

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

Fermi Architecture

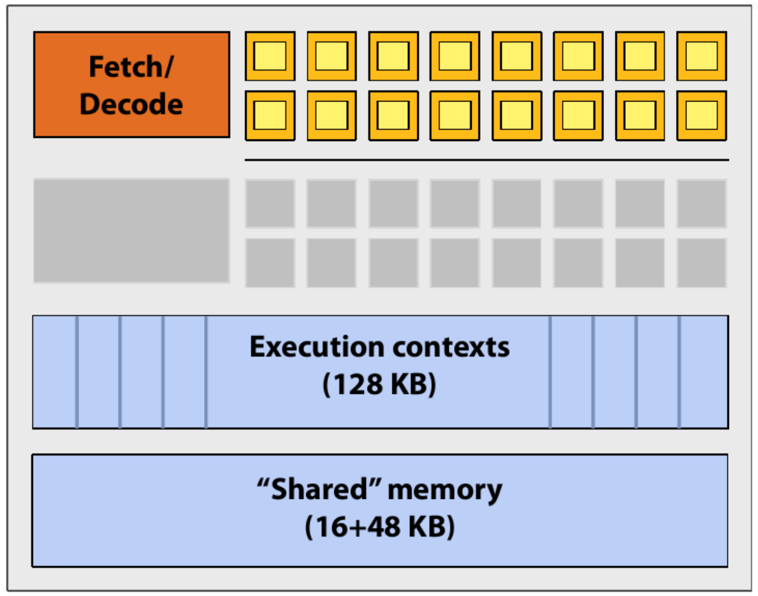


NVIDIA GeForce GTX 480 (Fermi)

- NVIDIA-speak:
  - 480 stream processors (“CUDA cores”)
  - “SIMT execution”
- Generic speak:
  - 15 cores
  - 2 groups of 16 SIMD functional units per core

Thursday, July 29, 2010

NVIDIA GeForce GTX 480 “core”

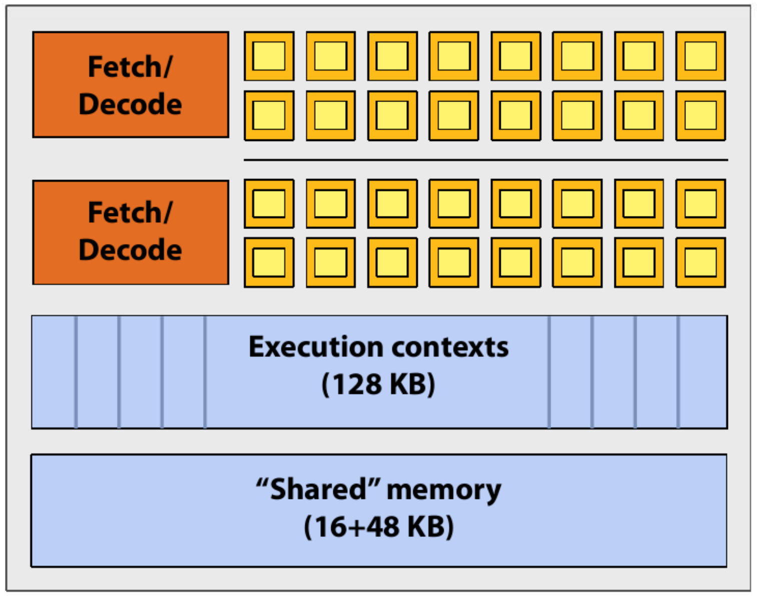


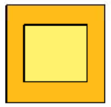
▨ = SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- Groups of 32 [fragments/vertices/CUDA threads] share an instruction stream
- Up to 48 groups are simultaneously interleaved
- Up to 1536 individual contexts can be stored

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GeForce GTX 480 "core"

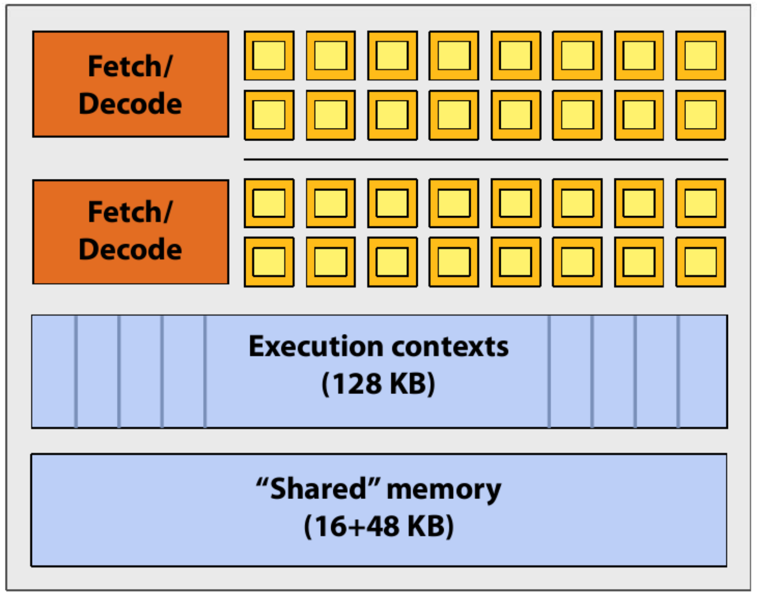


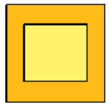

= SIMD function unit, control shared across 16 units (1 MUL-ADD per clock)

- The core contains 32 functional units
- Two groups are selected each clock (decode, fetch, and execute two instruction streams in parallel)

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GeForce GTX 480 **"SM"**

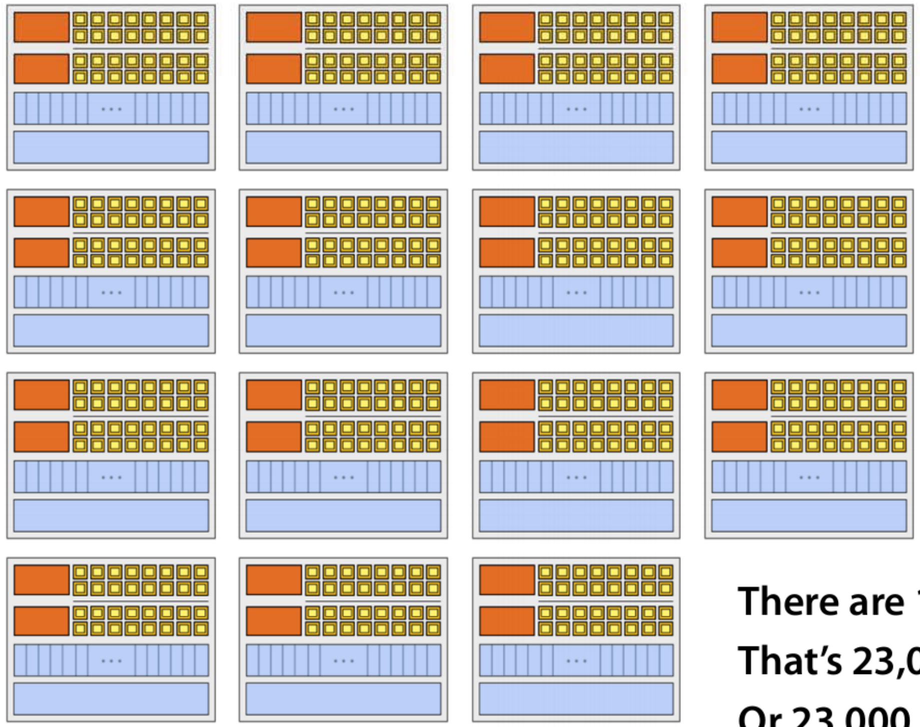



= **CUDA core**
(1 MUL-ADD per clock)

- The **SM** contains 32 **CUDA cores**
- Two **warps** are selected each clock (decode, fetch, and execute two **warps** in parallel)
- Up to 48 warps are interleaved, totaling 1536 **CUDA threads**

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GeForce GTX 480

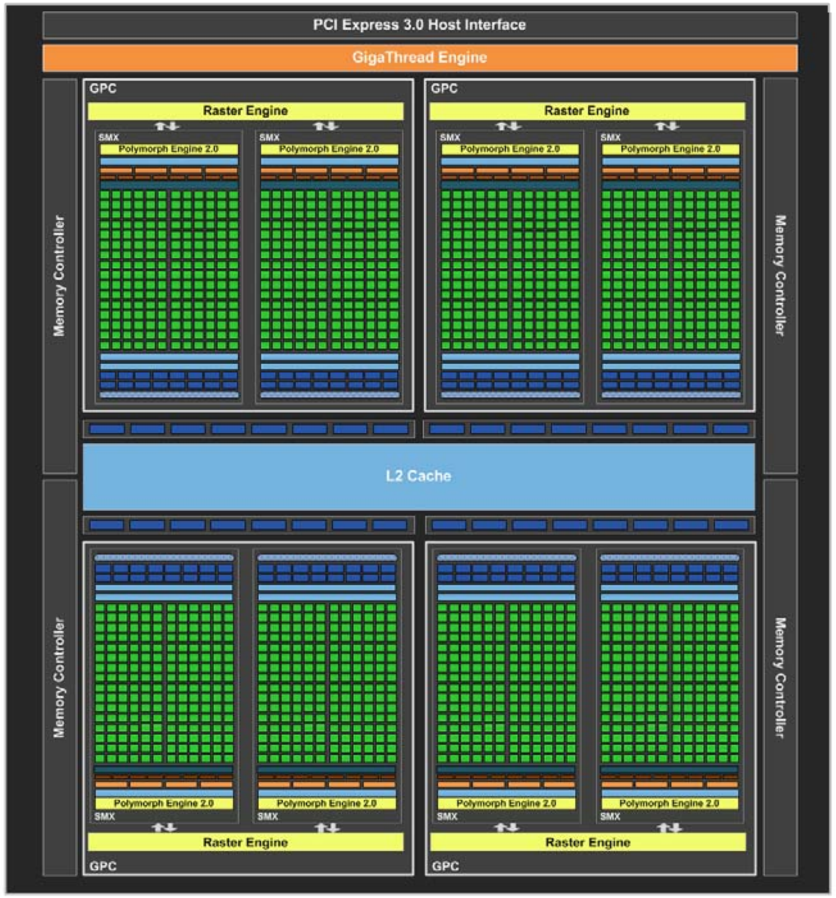


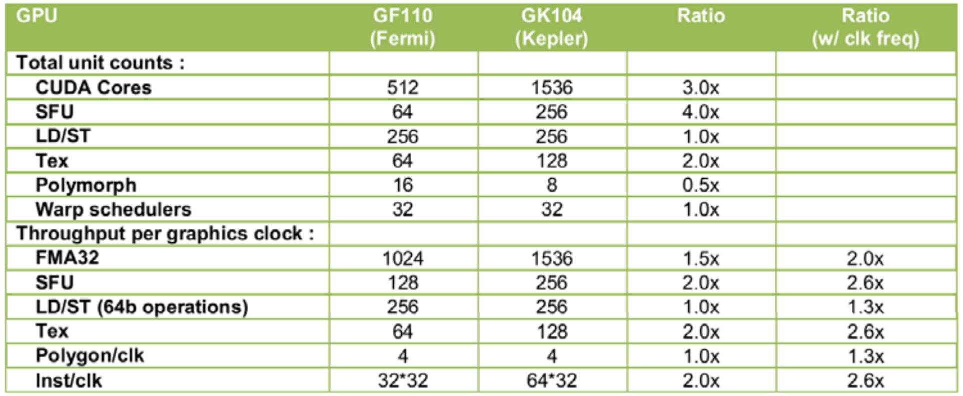
There are 15 of these things on the GTX 480:

That's 23,000 fragments!

Or 23,000 CUDA threads!

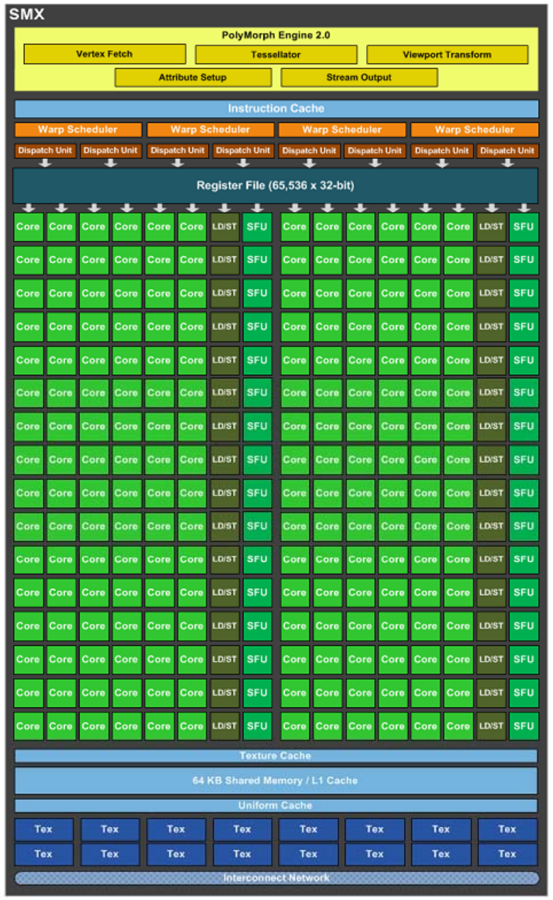
Kepler Architecture: GTX 680





GPU	GF110 (Fermi)	GK104 (Kepler)	Ratio	Ratio (w/ clk freq)
Total unit counts :				
CUDA Cores	512	1536	3.0x	
SFU	64	256	4.0x	
LD/ST	256	256	1.0x	
Tex	64	128	2.0x	
Polymorph	16	8	0.5x	
Warp schedulers	32	32	1.0x	
Throughput per graphics clock :				
FMA32	1024	1536	1.5x	2.0x
SFU	128	256	2.0x	2.6x
LD/ST (64b operations)	256	256	1.0x	1.3x
Tex	64	128	2.0x	2.6x
Polygon/clk	4	4	1.0x	1.3x
Inst/clk	32*32	64*32	2.0x	2.6x

SMX

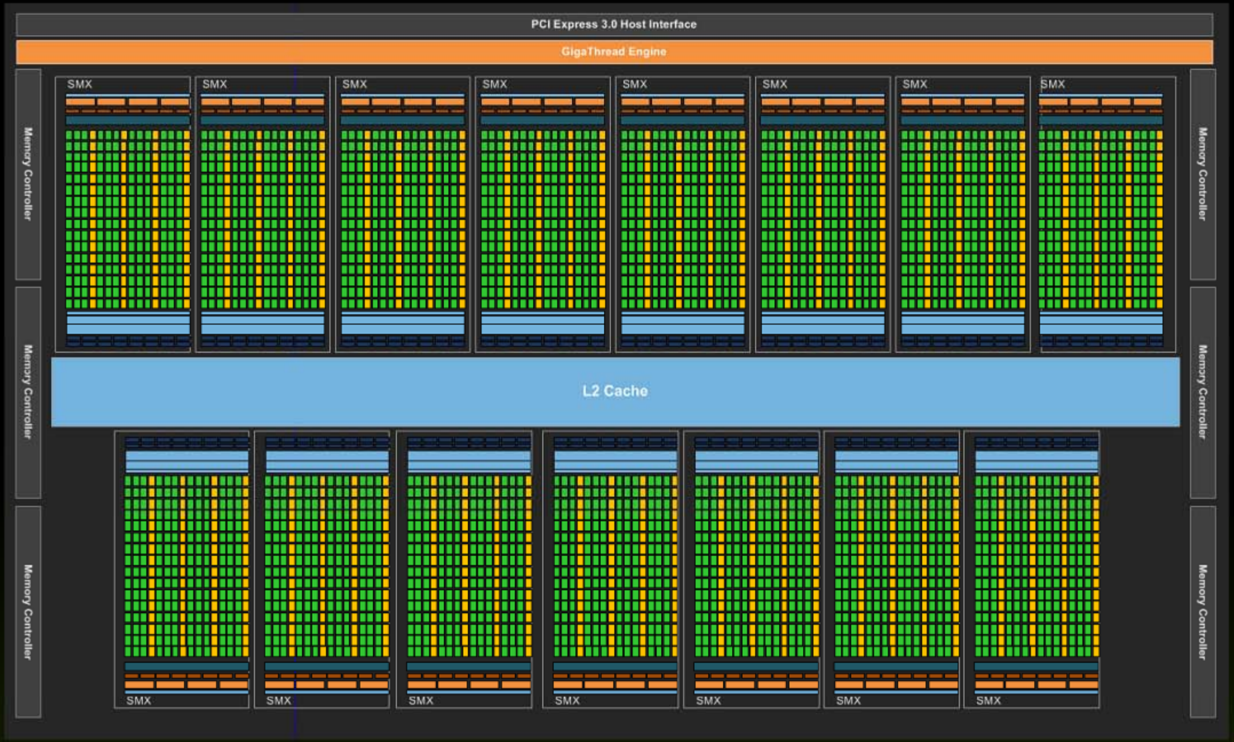


GPU	GF110 (Fermi)	GK104 (Kepler)	Ratio	Ratio (w/ clk freq)
Per SM unit counts :				
CUDA Cores	32	192	6.0x	
SFU	4	32	8.0x	
LD/ST	16	32	2.0x	
Tex	4	16	4.0x	
Polymorph	1	1	1.0x	
Warp schedulers	2	4	2.0x	
Throughput per graphics clock :				
FMA32	64	192	3.0x	3.9x
SFU	8	32	4.0x	5.2x
LD/ST (64b operations)	16	32	2.0x	2.6x
Tex	4	16	4.0x	5.2x
Polygon/clk	0.25	0.5	2.0x	2.6x
Inst/clk	32*2	32*8	4.0x	5.2x

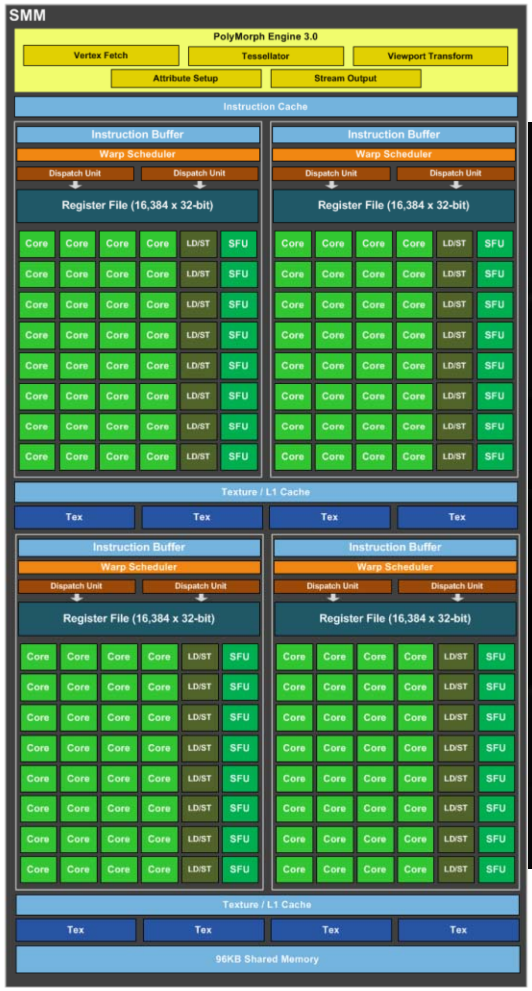
NVIDIA　GK110

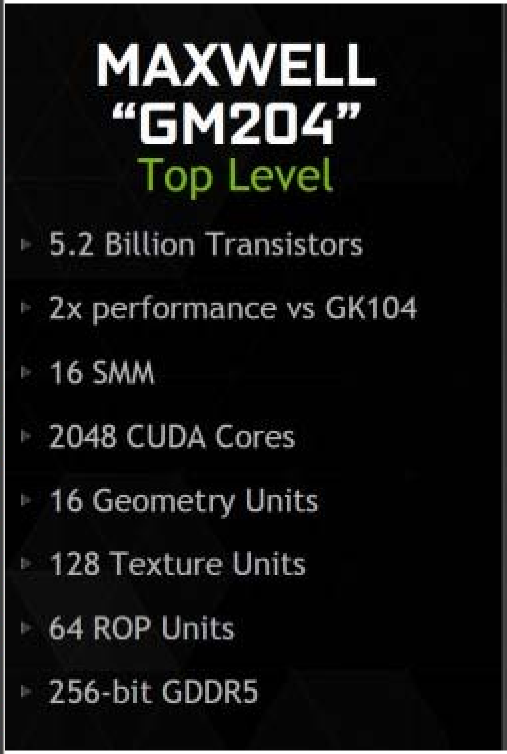
Architecture

- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3

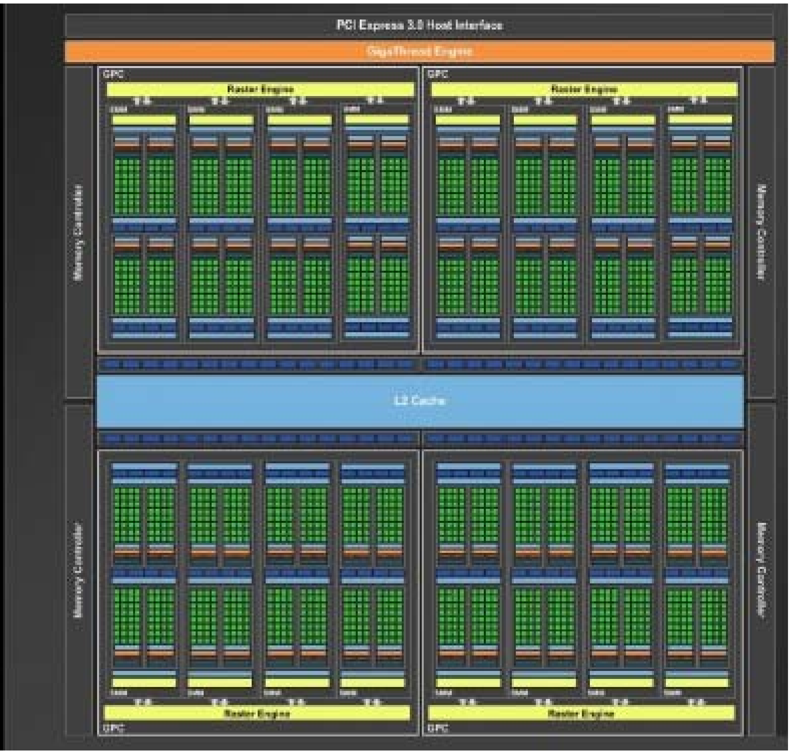


Maxwell Architecture SMM:

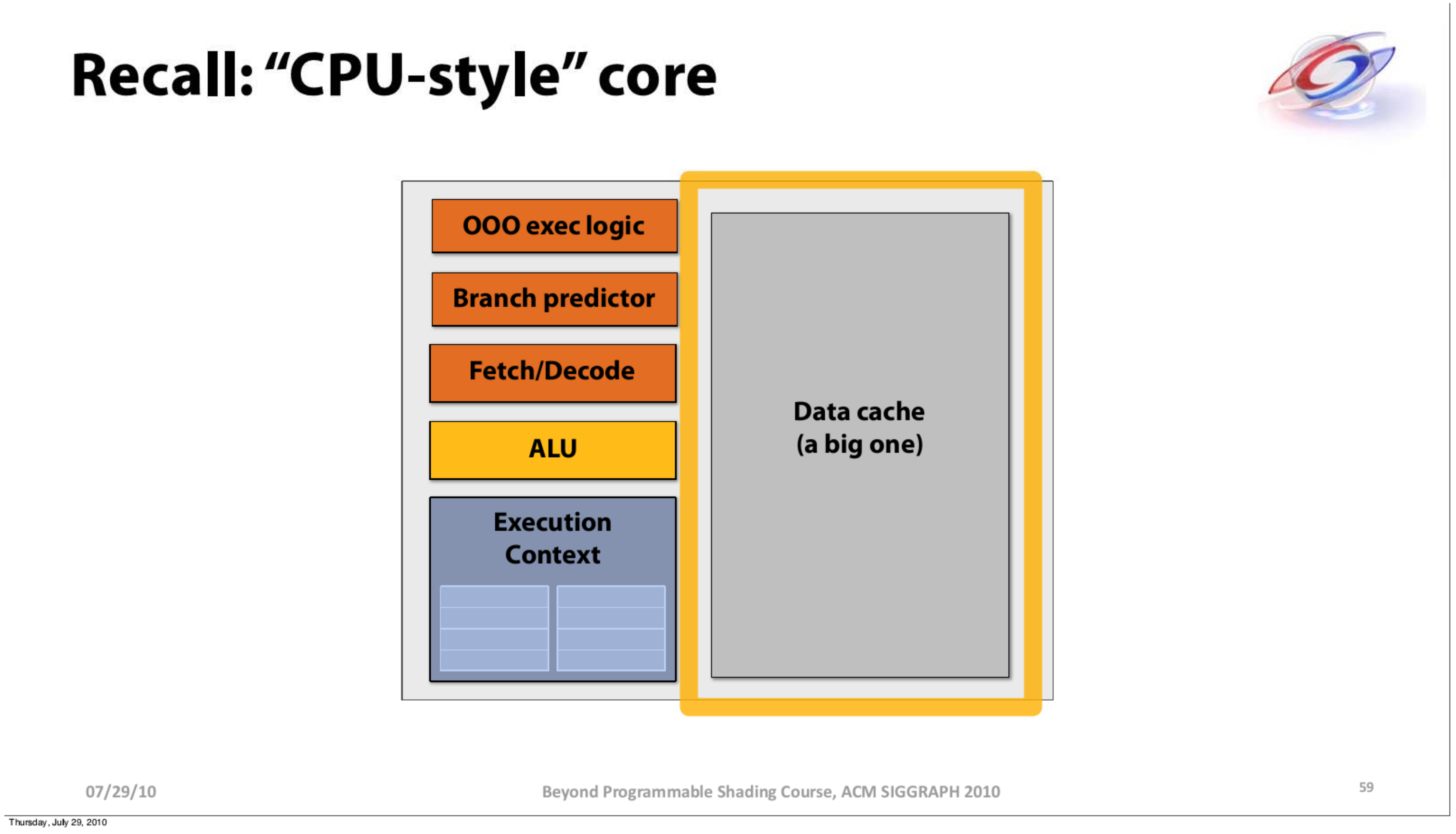



MAXWELL
"GM204"
Top Level

- 5.2 Billion Transistors
- 2x performance vs GK104
- 16 SMM
- 2048 CUDA Cores
- 16 Geometry Units
- 128 Texture Units
- 64 ROP Units
- 256-bit GDDR5

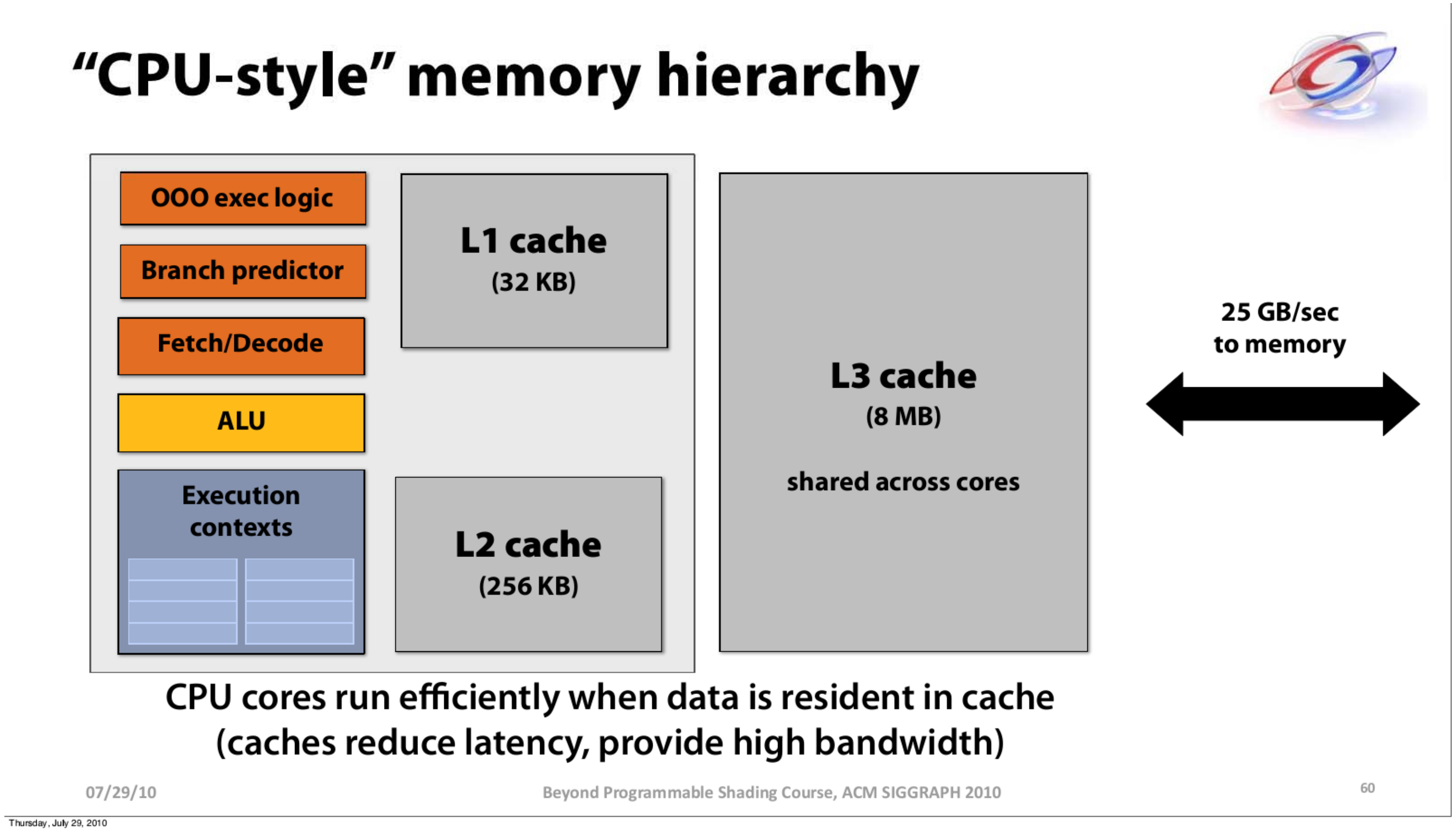


Memory and Data Access



Recall: "CPU-style" core

CPU Style Memory

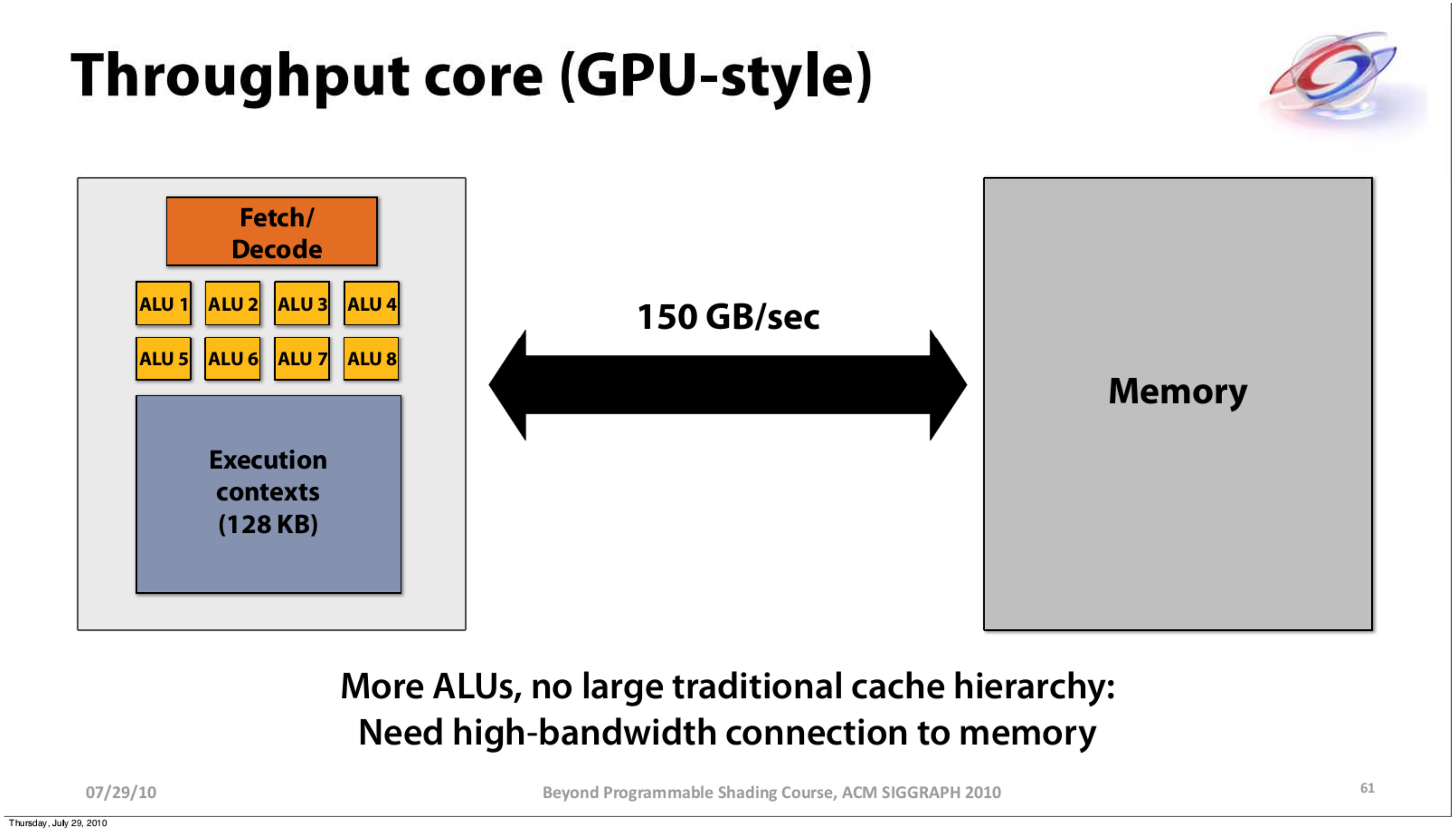


"CPU-style" memory hierarchy

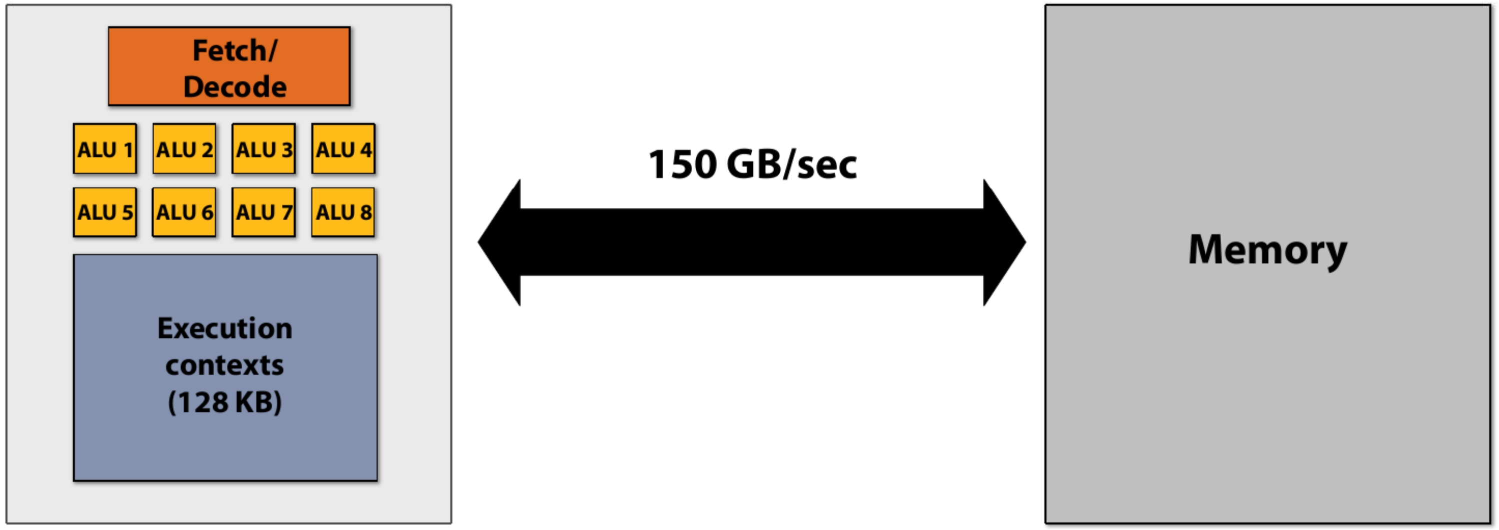
OOO exec logic
Branch predictor
Fetch/Decode
ALU
Execution contexts
L1 cache
(32 KB)
L2 cache
(256 KB)
L3 cache
(8 MB)
shared across cores
25 GB/sec
to memory

CPU cores run efficiently when data is resident in cache
(caches reduce latency, provide high bandwidth)

GPU Style : Throughput!

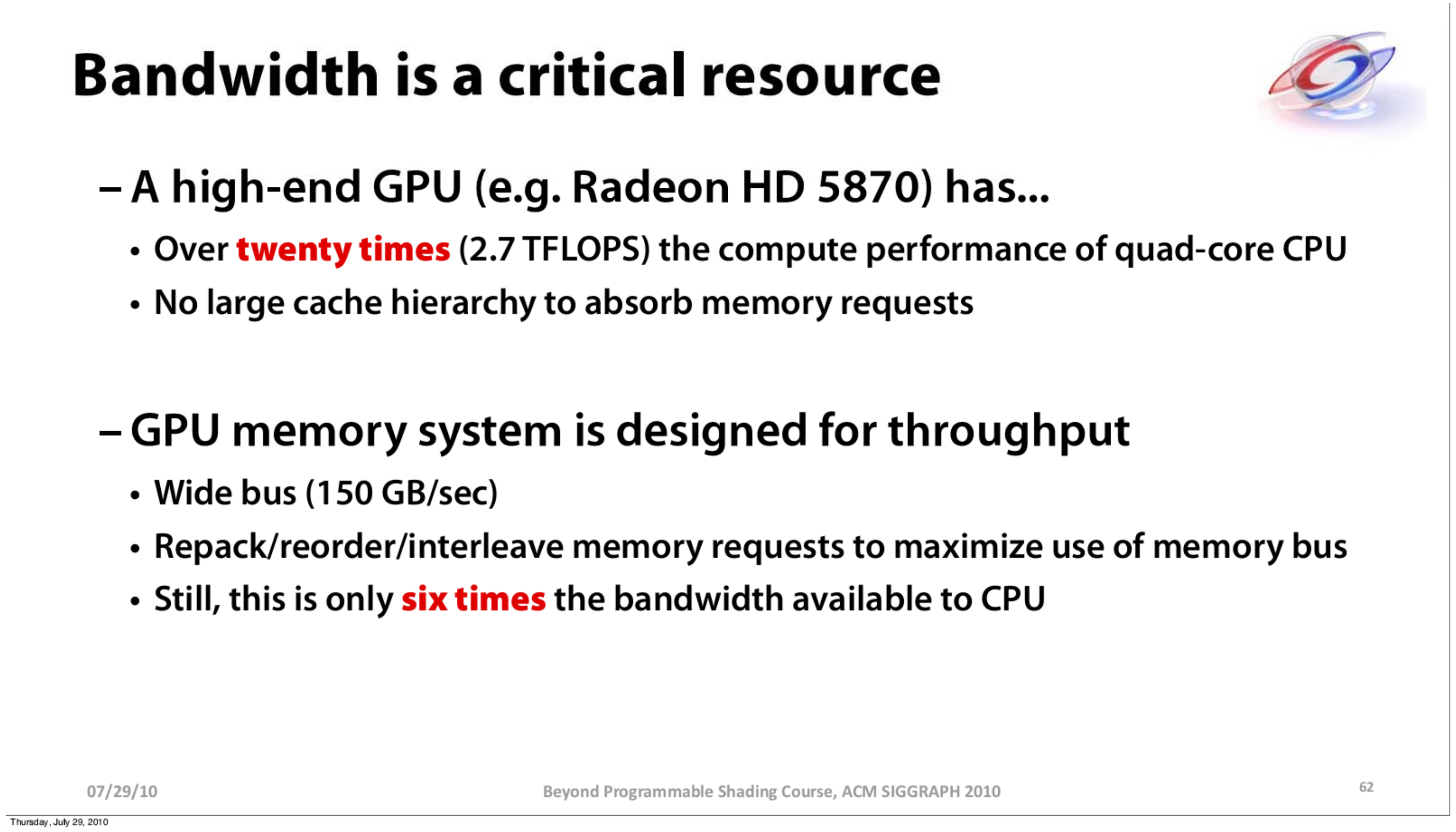


Throughput core (GPU-style)



More ALUs, no large traditional cache hierarchy:
Need high-bandwidth connection to memory

Memory Bound !



Bandwidth is a critical resource

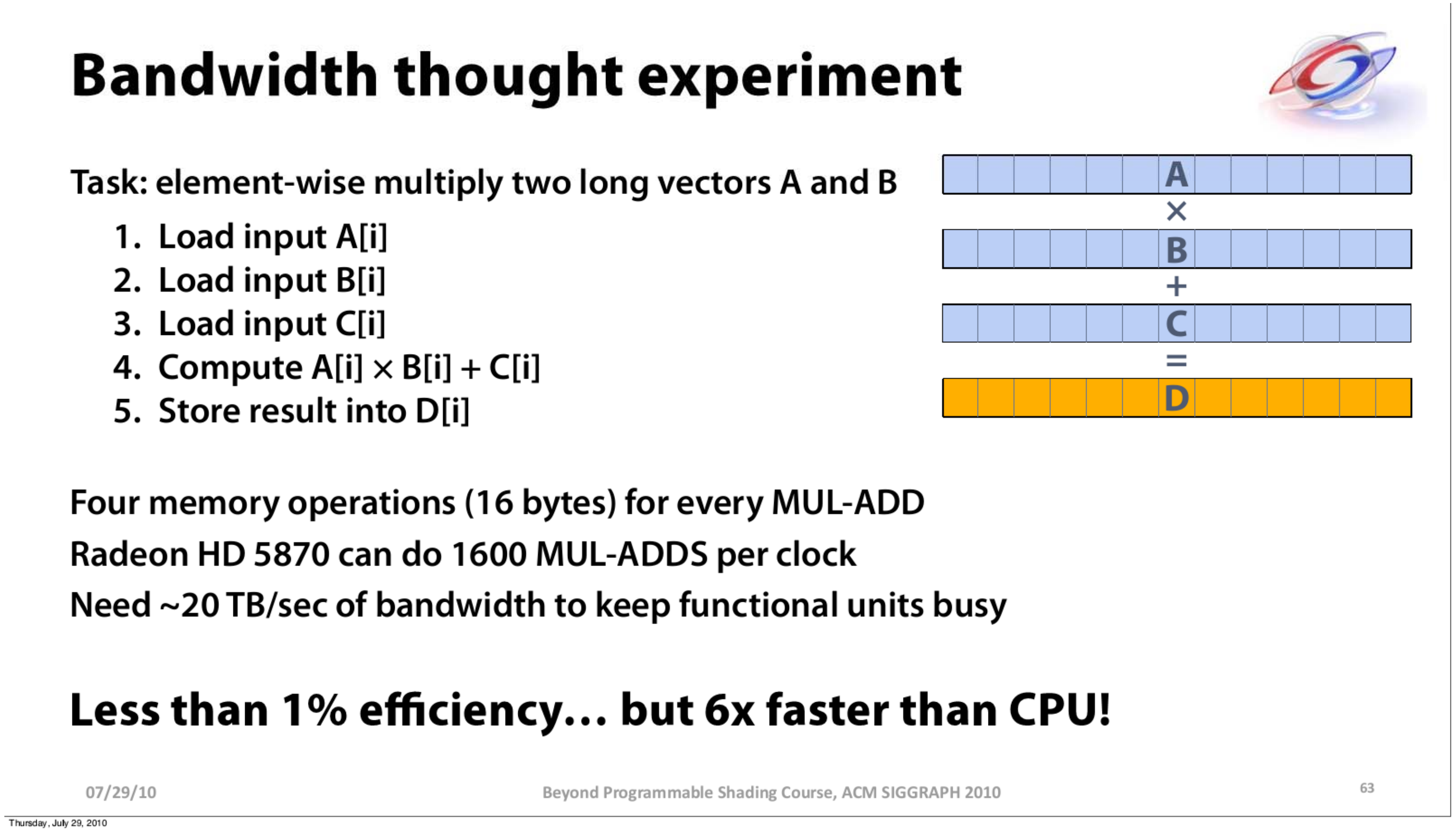
– A high-end GPU (e.g. Radeon HD 5870) has...

- Over **twenty times** (2.7 TFLOPS) the compute performance of quad-core CPU
- No large cache hierarchy to absorb memory requests

– GPU memory system is designed for throughput

- Wide bus (150 GB/sec)
- Repack/reorder/interleave memory requests to maximize use of memory bus
- Still, this is only **six times** the bandwidth available to CPU

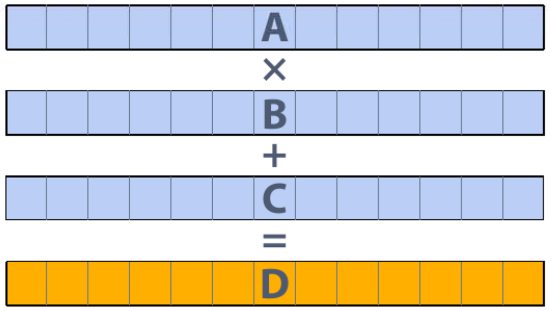
A Simple Example



Bandwidth thought experiment

Task: element-wise multiply two long vectors A and B

1. Load input A[i]
2. Load input B[i]
3. Load input C[i]
4. Compute A[i] × B[i] + C[i]
5. Store result into D[i]



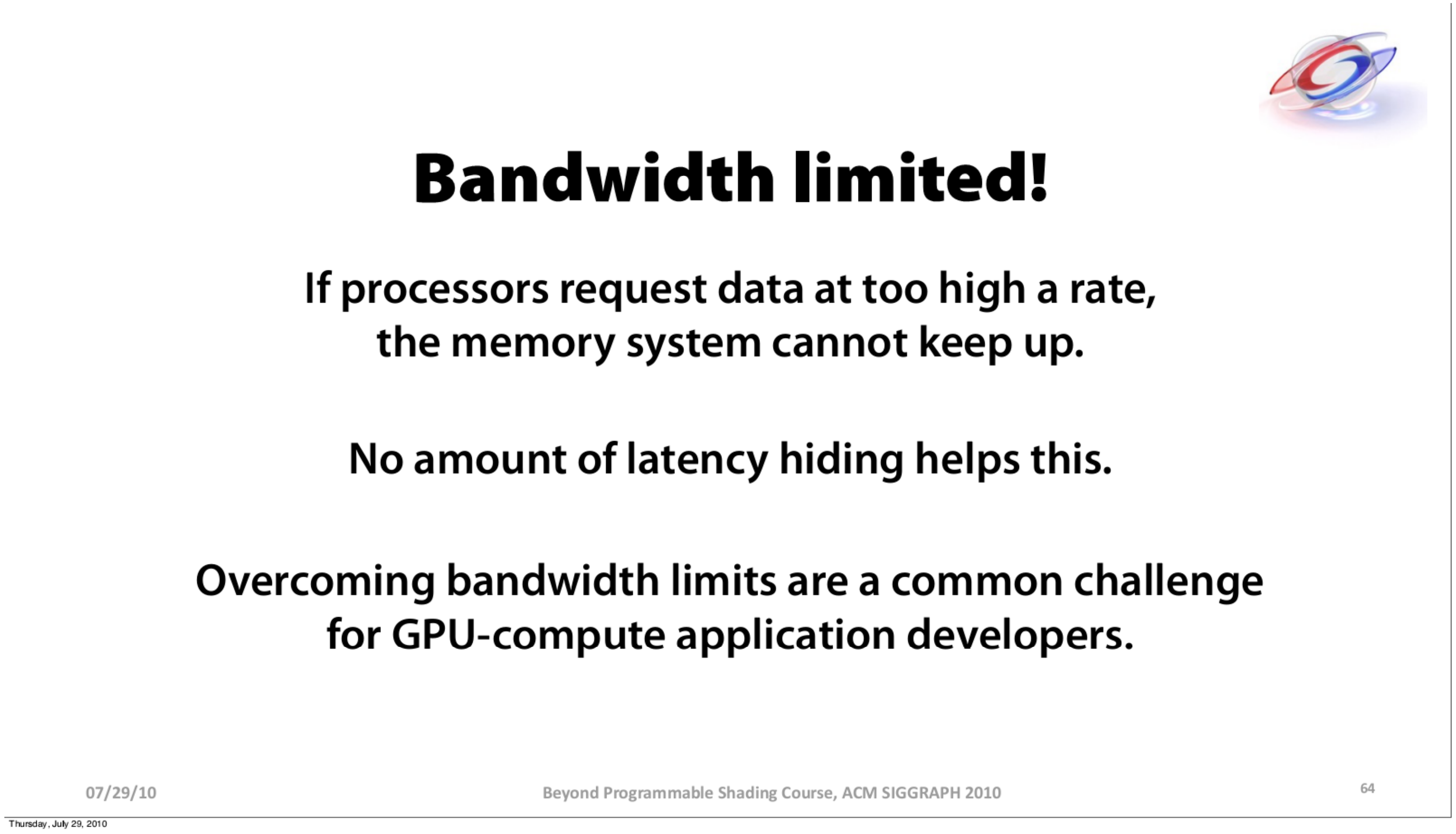
Four memory operations (16 bytes) for every MUL-ADD

Radeon HD 5870 can do 1600 MUL-ADDS per clock

Need ~20 TB/sec of bandwidth to keep functional units busy

Less than 1% efficiency… but 6x faster than CPU!

Memory Bound



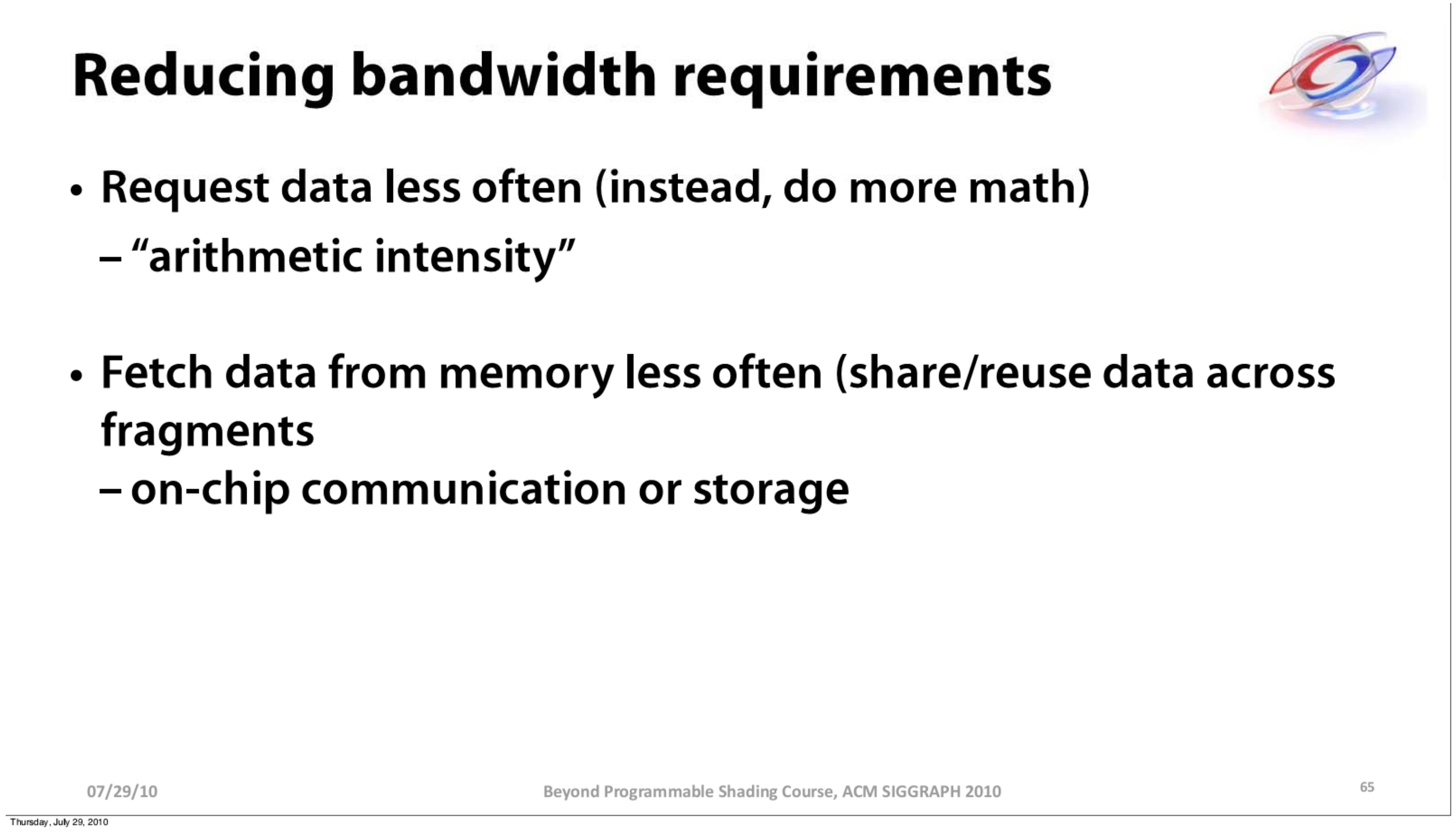
Bandwidth limited!

If processors request data at too high a rate,
the memory system cannot keep up.

No amount of latency hiding helps this.

Overcoming bandwidth limits are a common challenge
for GPU-compute application developers.

Try from the beginning



Reducing bandwidth requirements

- Request data less often (instead, do more math)
  - "arithmetic intensity"
- Fetch data from memory less often (share/reuse data across fragments
  - on-chip communication or storage

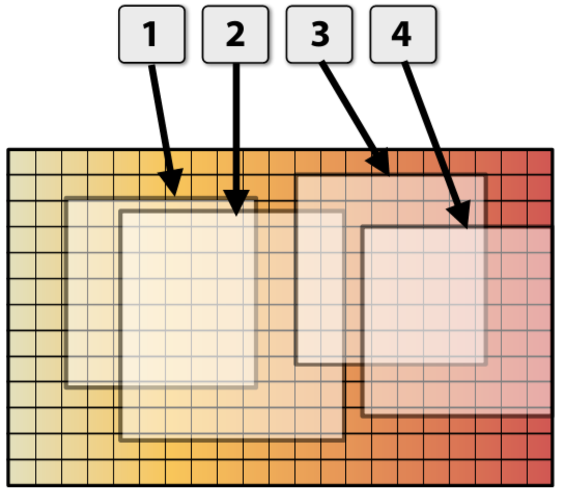
07/29/10 Beyond Programmable Shading Course, ACM SIGGRAPH 2010 65

Thursday, July 29, 2010

Reducing bandwidth requirements

- Two examples of on-chip storage
  - Texture caches
  - OpenCL "local memory" (CUDA shared memory)

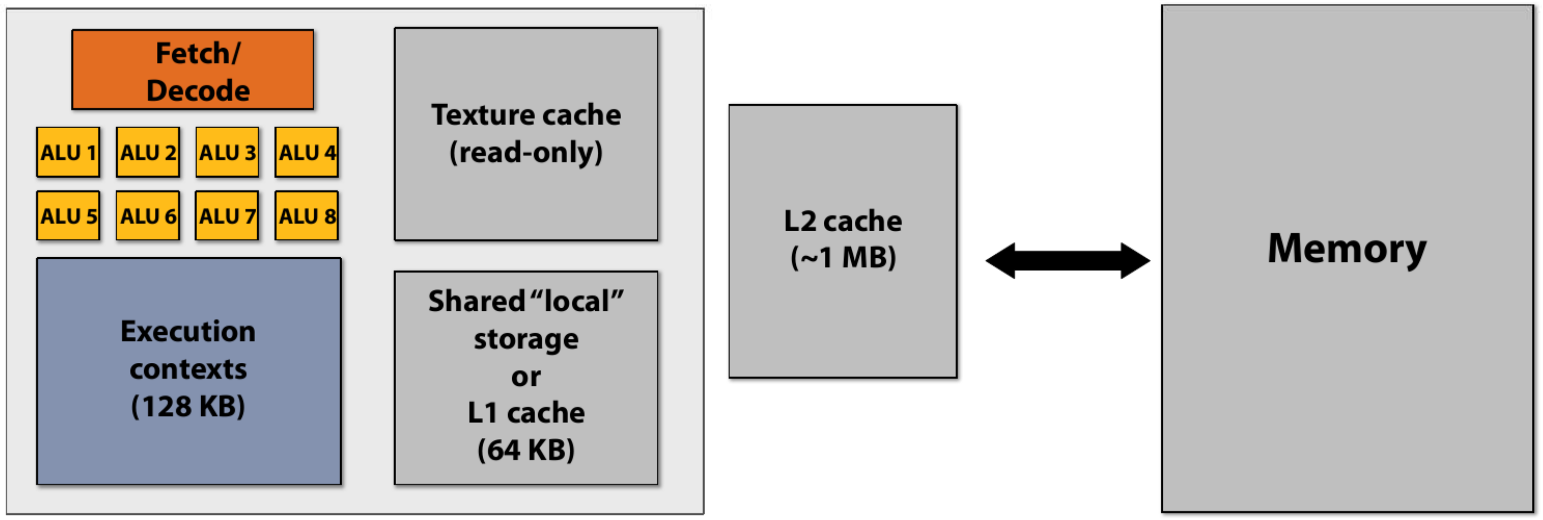
Texture data



Texture caches:

Capture reuse across fragments, not temporal reuse within a single shader program

Modern 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 (read-only)
Shared "local" storage or L1 cache (64 KB)
L2 cache (~1 MB)
Memory

On-chip storage takes load off memory system.
Many developers calling for more cache-like storage (particularly GPU-compute applications)

GPU is a heterogeneous many core computing system, highly-tuned for high throughput applications

Effective GPU Tasks Require

- Thousands of independent processing fragments
  - Utilize many ALUs
  - Hide the latency
- Better Sharing instruction streams
  - SIMD
- Computing Intensive
  - Nice computing and communication ratio
  - Not limited by bandwidth

举例：石油勘探

- 目前的CPU只能满足石油勘探的普通处理技术，如解编、预处理、叠后偏移等
- 目前的CPU不能完全满足
- 需要大量运算的处理技术，
- 如叠前时间偏移、叠前深
- 度偏移、波动方程偏移等

- 以叠前偏移为例，一般实现一道
- 偏移需要 1000000×6000×2次
- 数学运算，计算量和需要处理的数据量极其巨大

举例：气象预报

- 目前，气象预测对计算资源的需求日益增长
- 对于24小时的短期预报，要求
  - 一般在0.5~1小时内得到结果
  - 对于中期预报（10天，15公里），大概5～6小时
- 精细化预报
  - 网格<3km，甚至<1km
  - <每半小时完成一次

全国公路气象预报
2009年3月12日20时-3月13日20时
中国气象局与交通运输部联合发布
影响路段：
未受影响
大到暴雨
雾
雷暴
小到中雪
冻雨
小到中雨
大到暴雪
沙尘
乌鲁木齐
拉萨
呼和浩特
北京
天津
银川
大同
石家庄
济南
西宁
兰州
郑州
西安
南京
上海
成都
武汉
重庆
南昌
长沙
福州
台北
贵阳
昆明
南宁
广州
海口
哈尔滨
长春
沈阳
南海诸岛
0 325 650 1300 1950 2600 KM
70°E 80°E 90°E 100°E 110°E 120°E 130°E 140°E
20°N 30°N 40°N 50°N