GPU体系架构概述

周　斌 @ NVIDIA & USTC 2014年3月

内容

- 为什么需要GPU?
- 三种方法提升GPU的处理速度
- 实际GPU设计举例
  - --NVIDIA GTX 480: Fermi
  - --NVIDIA GTX 680: Kepler
- GPU的存储器设计

名词解释

- FLOPS – FLoating-point OPerations per Second
- GFLOPS - One billion (109) FLOPS
- TFLOPS – 1,000 GFLOPS

CPU和GPU的趋势曲线

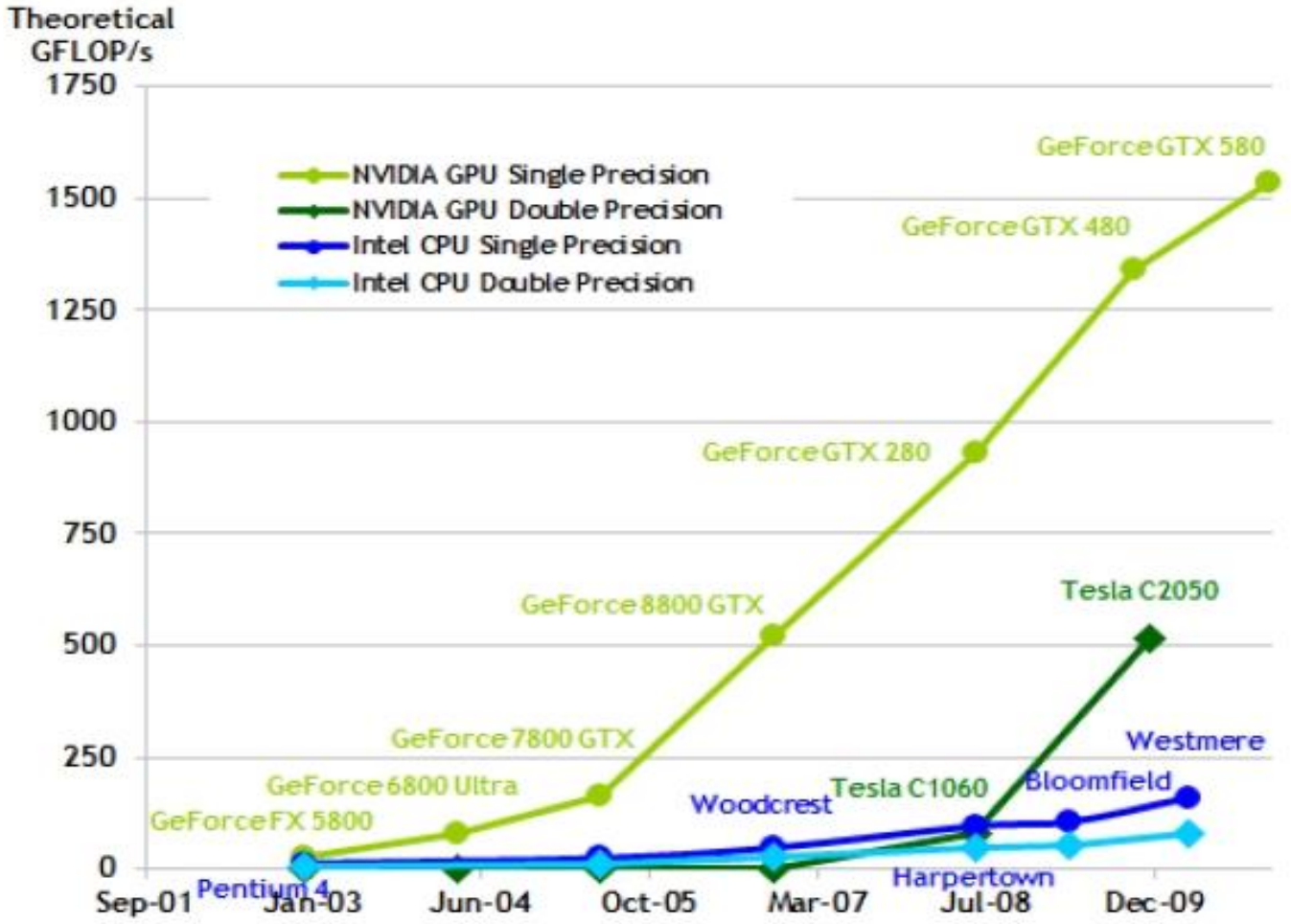


Chart from: http://proteneer.com/blog/?p=263

为什么需要GPU?

- 应用的需求越来越高
- 计算机技术由应用驱动
- （Application Driven)

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

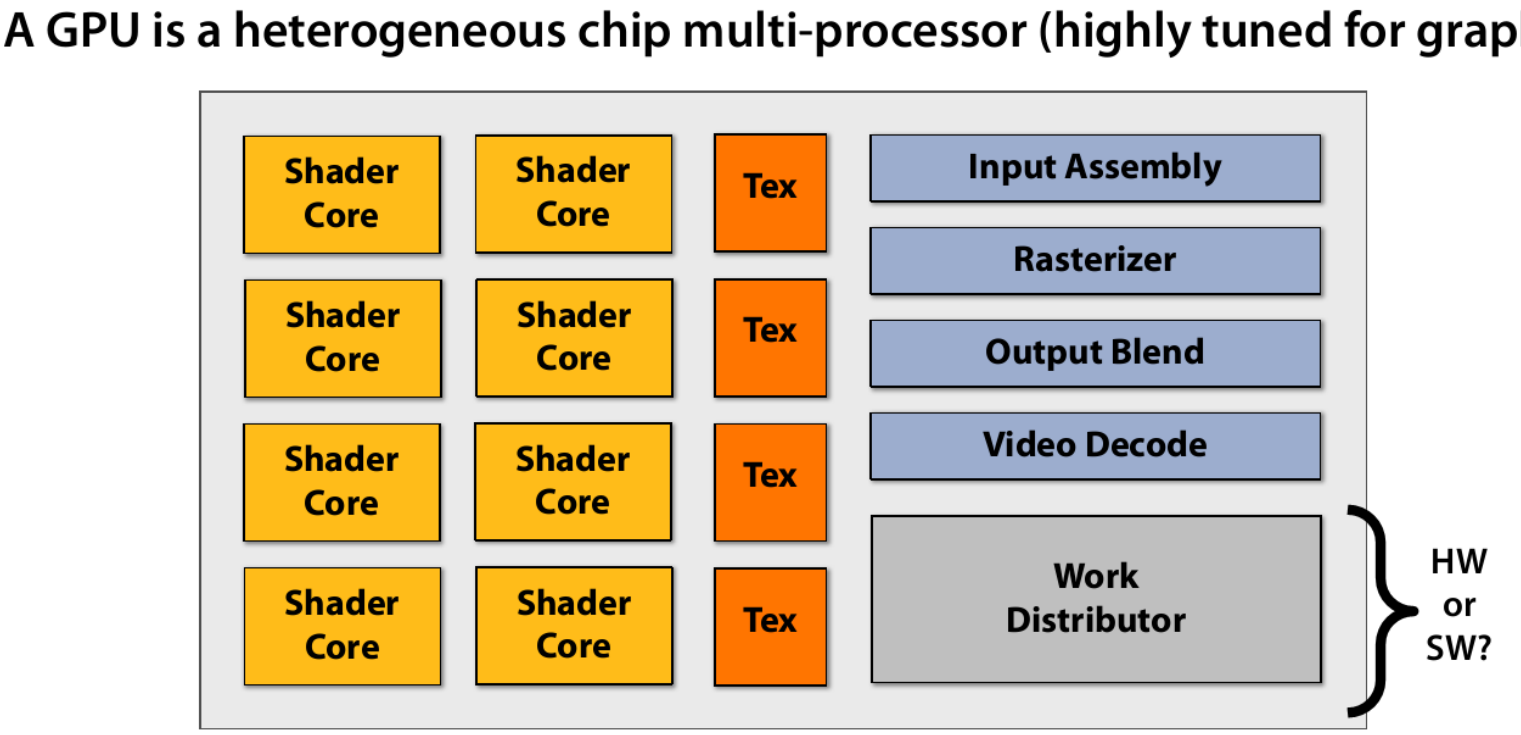
- 以叠前偏移为例，一般实现一道
- 偏移需要　1000000×6000×2次
- 数学运算，计算量和需要处理的数据量极其巨大

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

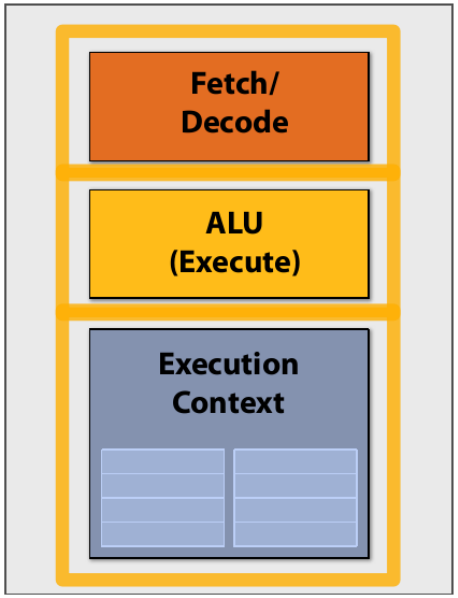
GPU（Graphic Processing Unit）结构略图

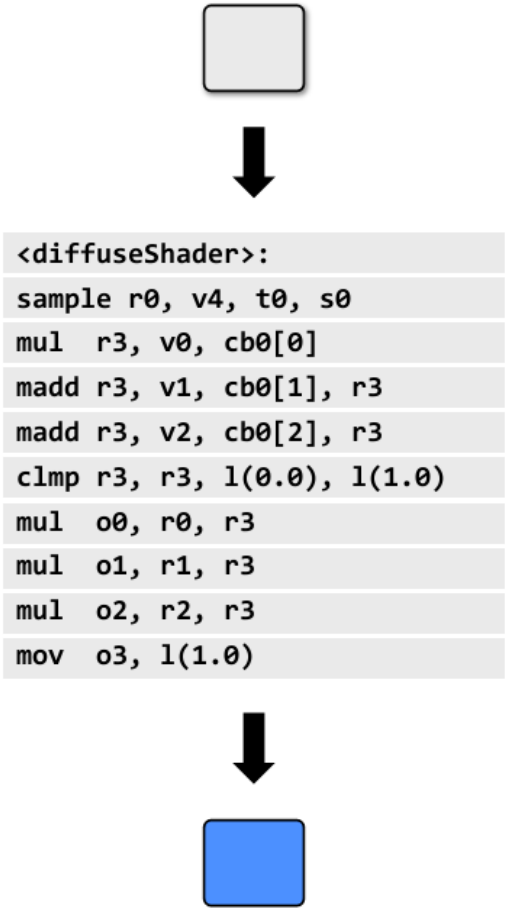
GPU是一个异构的多处理器芯片，为图形图像处理优化



执行单元

Execute shader

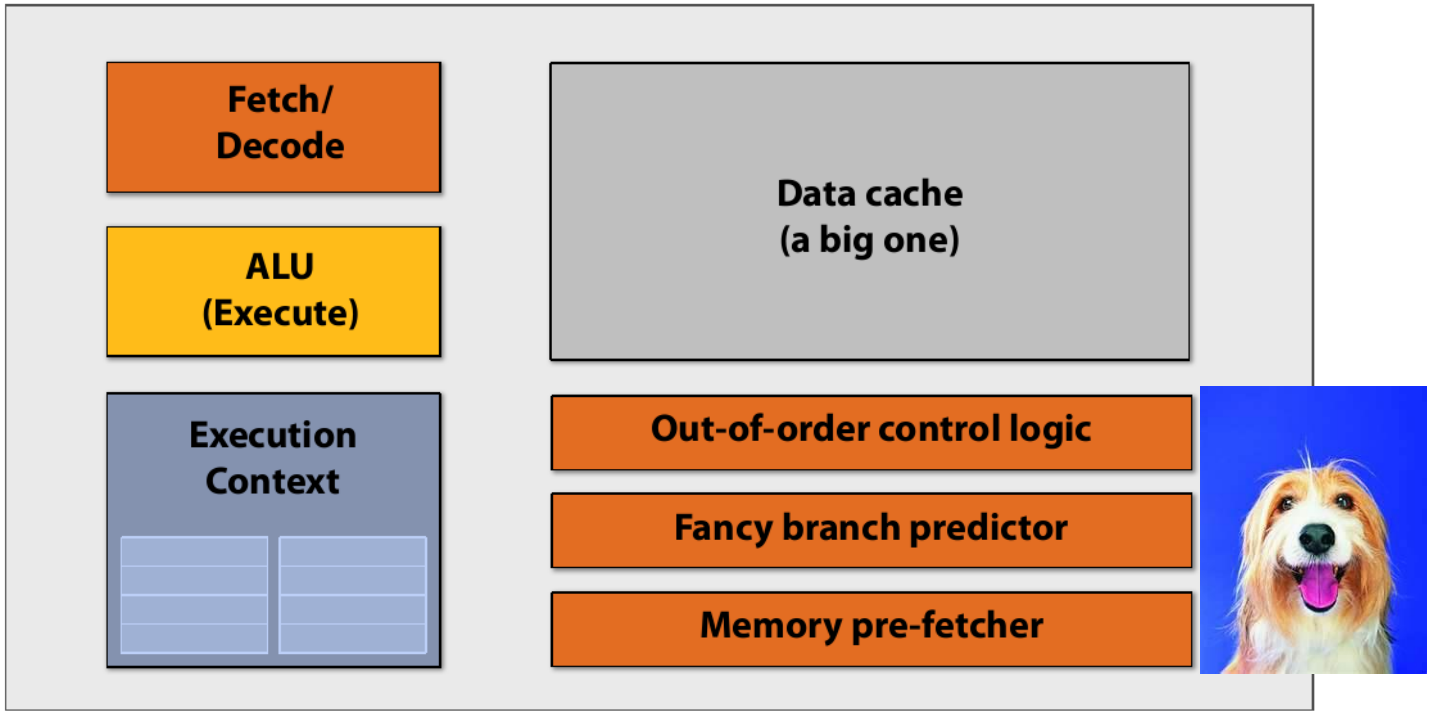




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

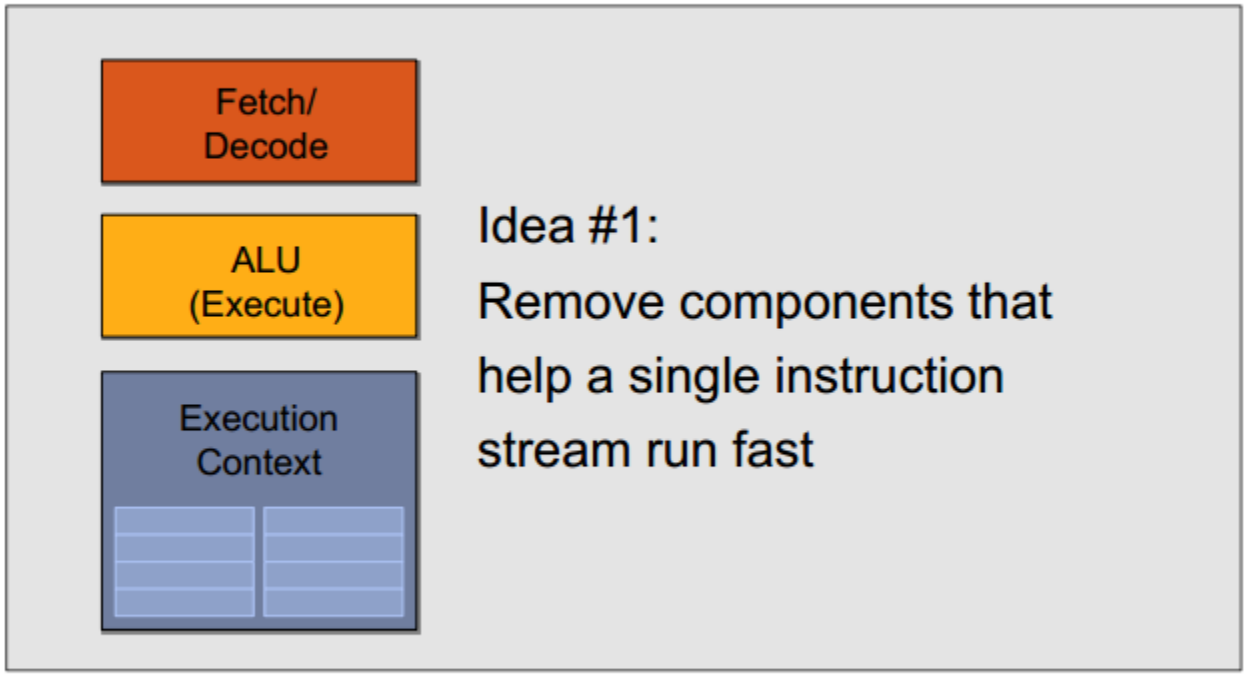
CPU类型的内核

"CPU-style" cores

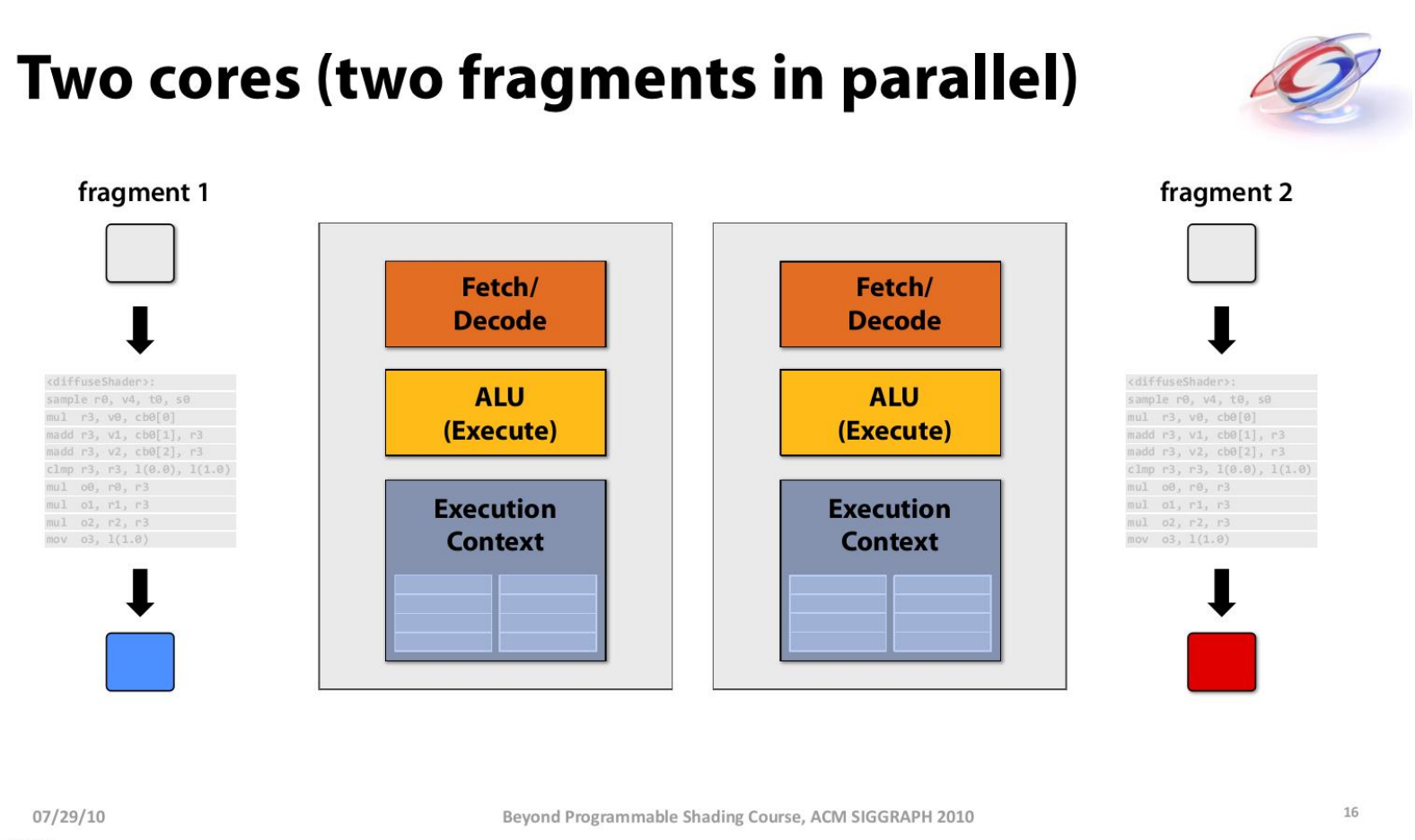


思路1：精简、减肥

Slimming down

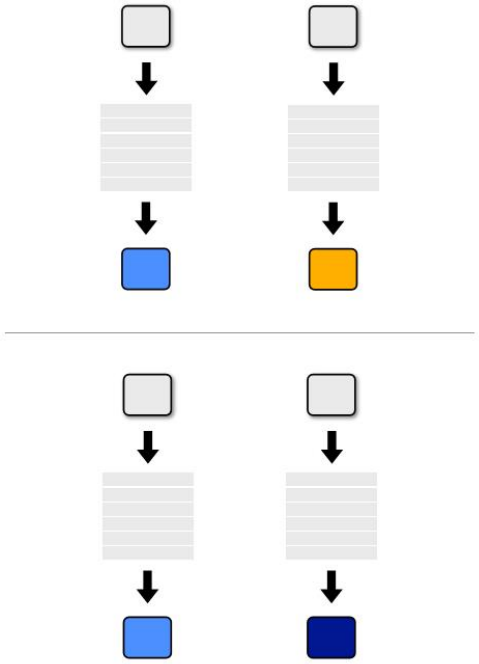


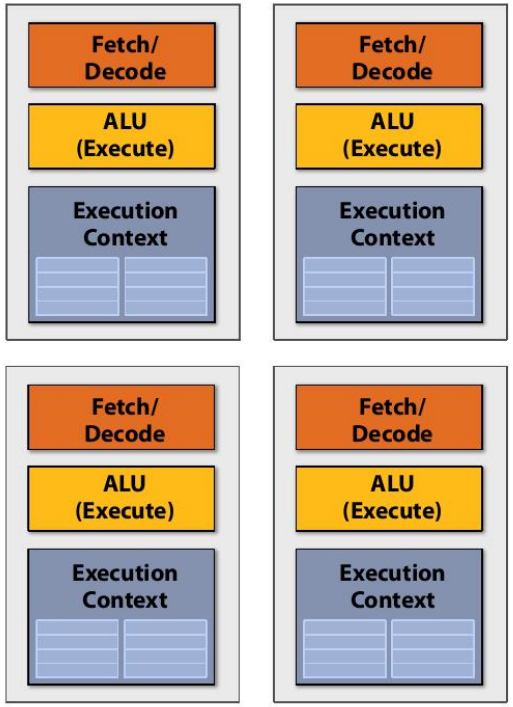
2个核、同时执行2个程序片元



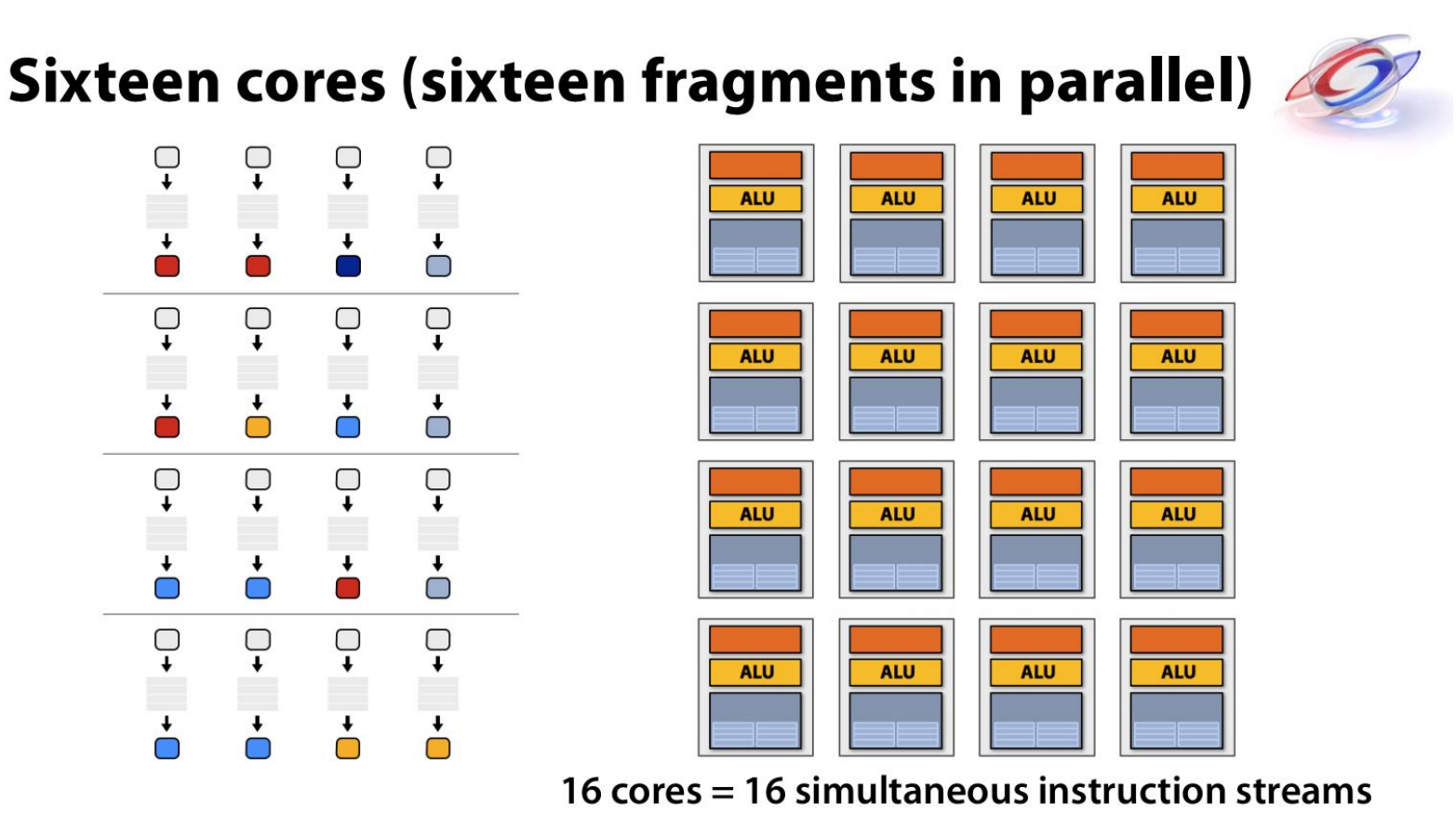
4个核、同时执行4个程序片元

Four cores (four fragments in parallel)



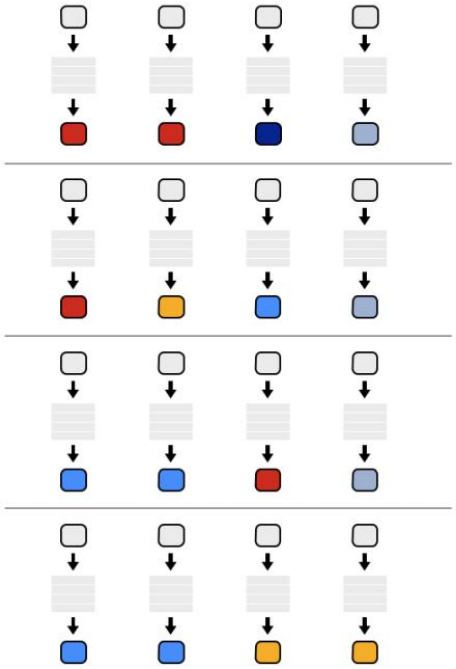


16个核、同时执行16个程序片元



指令流共享，多个程序片元共享指令流

Instruction stream sharing

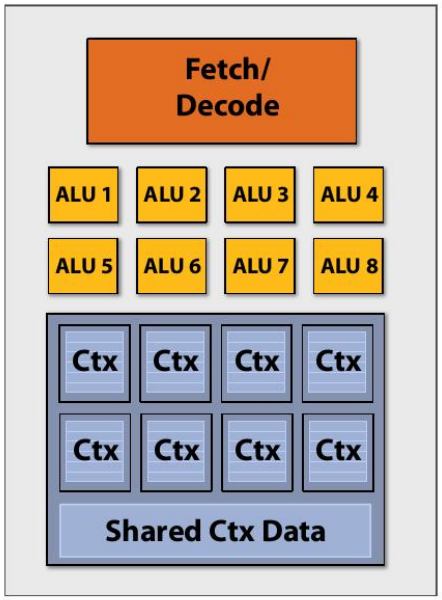


But ... many fragments should be able to share an instruction stream!

```
<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)
```

思路2：增加ALU，SIMD

Add ALUs

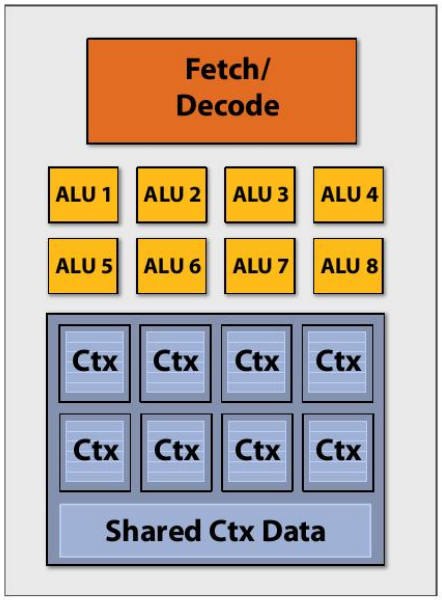


Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

改进的处理单元

Modifying the shader



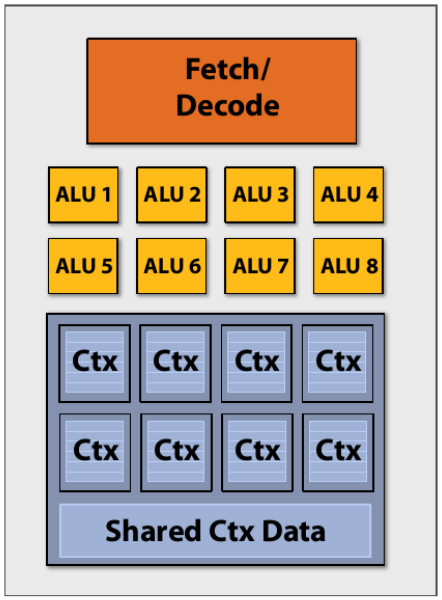

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using scalar ops on scalar registers

指令变化

Modifying the shader



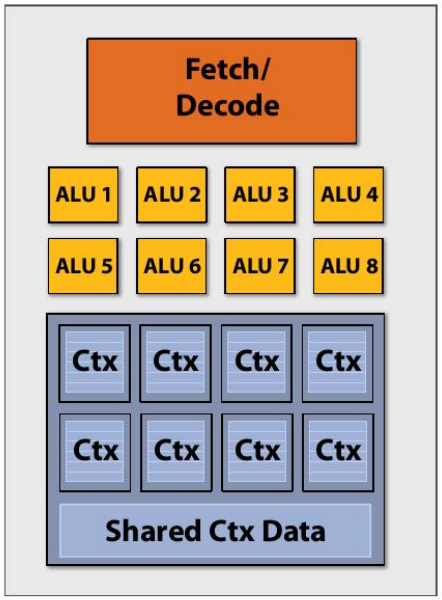

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul    vec_r3, vec_v0, cb0[0]
VEC8_madd   vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd   vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp   vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul    vec_o0, vec_r0, vec_r3
VEC8_mul    vec_o1, vec_r1, vec_r3
VEC8_mul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

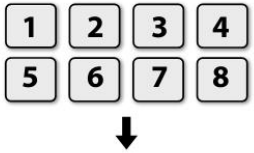
New compiled shader:

Processes eight fragments using vector ops on vector registers

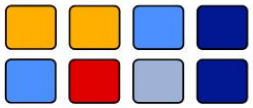
多数据执行

Modifying the shader



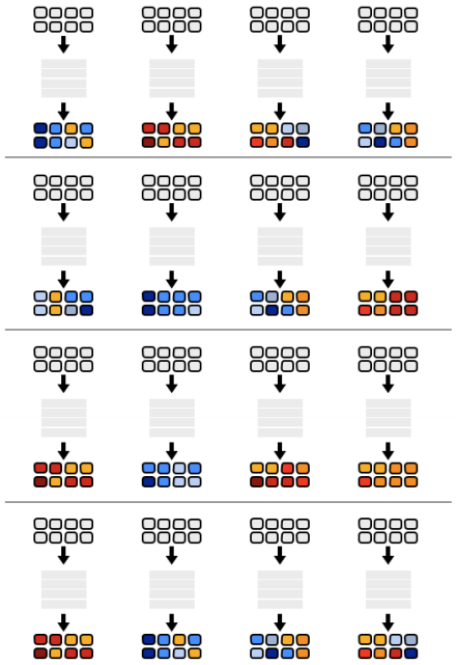


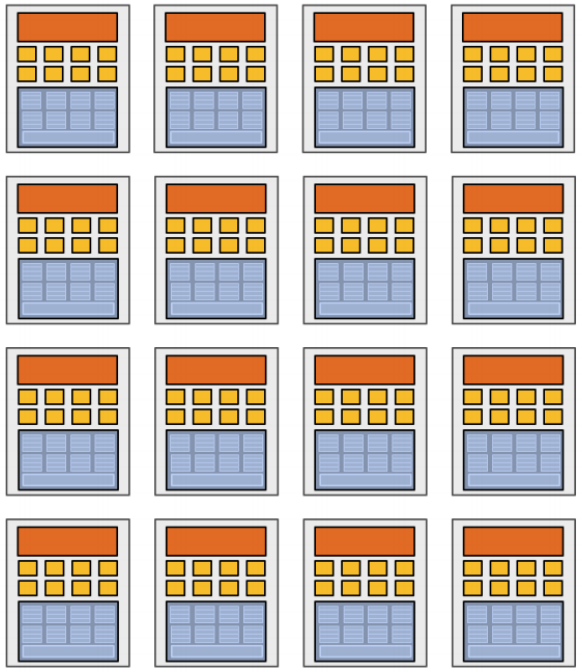

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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



128个程序片元同时执行，并发16路指令流

128 fragments in parallel

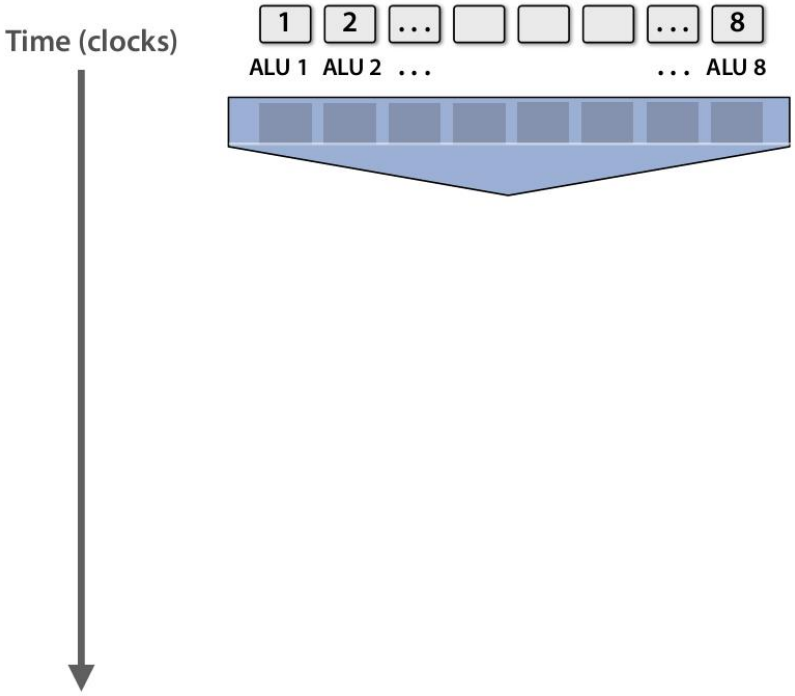




16 cores = 128 ALUs, 16 simultaneous instruction streams

分支处理如何办?

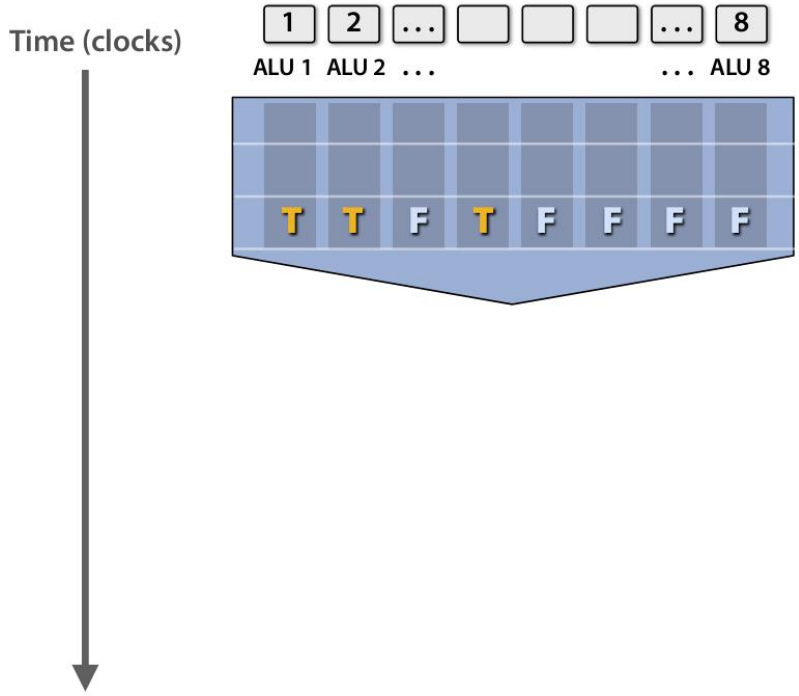
But what about branches?




```
<unconditional
 shader code>

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

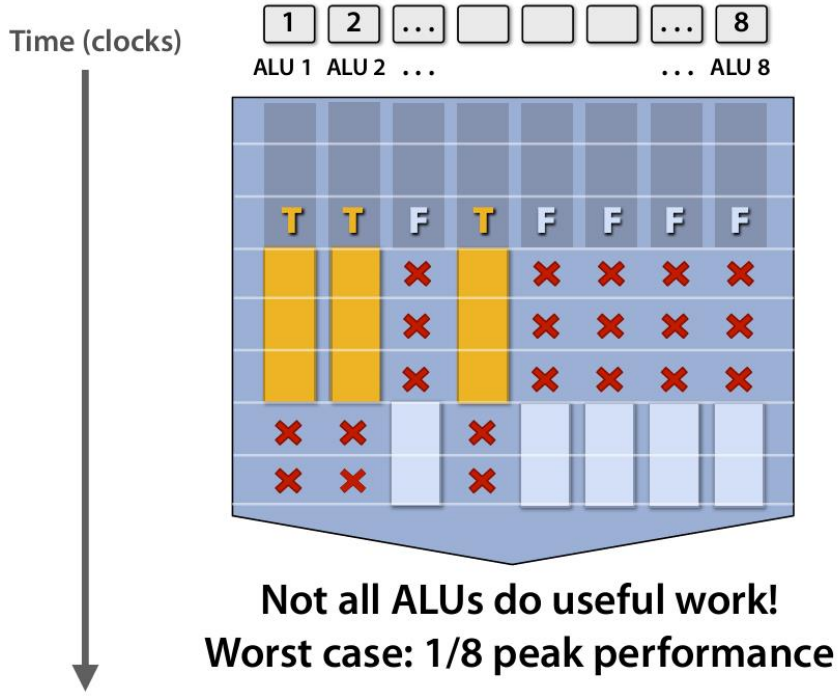
But what about branches?




```
<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?

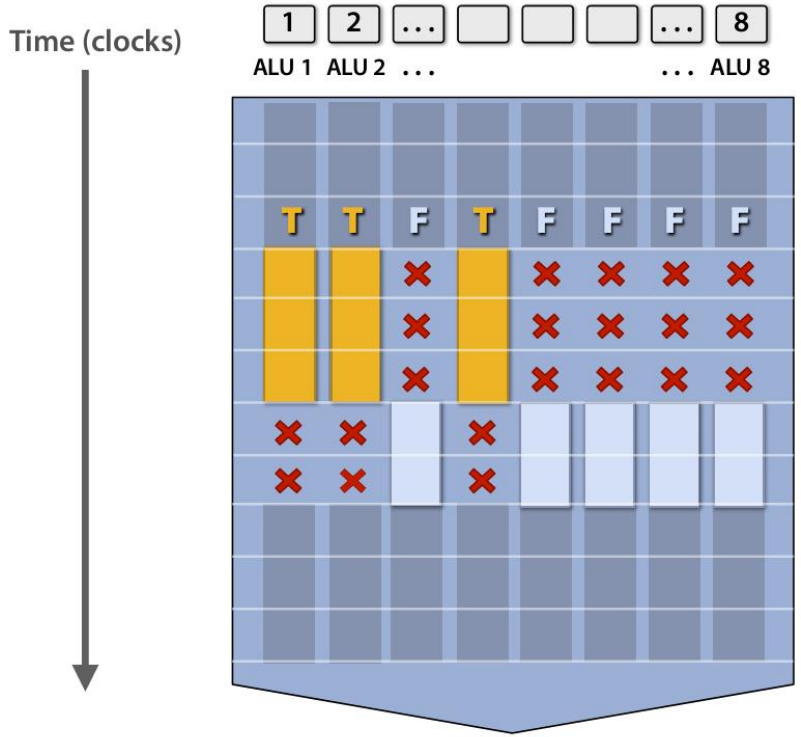



```
<unconditional
 shader code>

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

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

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>
```

澄清

- SIMD处理并不总是需要显式的SIMD指令
- 选项1：显示的向量运算指令
  - SSE等
- 选项2：标量指令，但是硬件进行矢量化
  - 硬件进行指令流共享
  - NVIDIA等架构
  - 多个片元共享指令流

停滞！

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

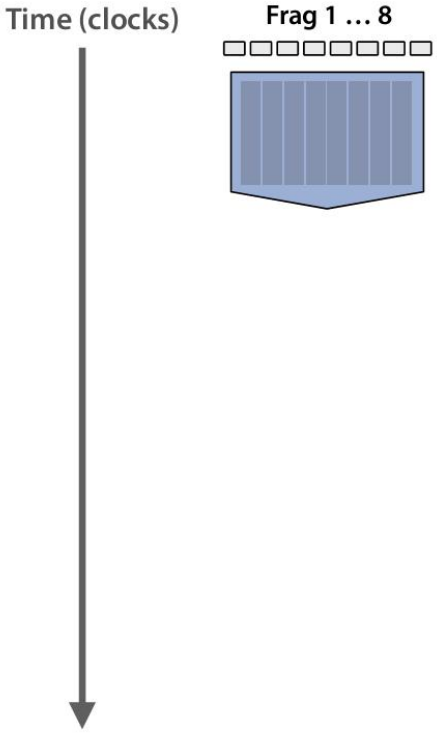
We've removed the fancy caches and logic that helps avoid stalls.

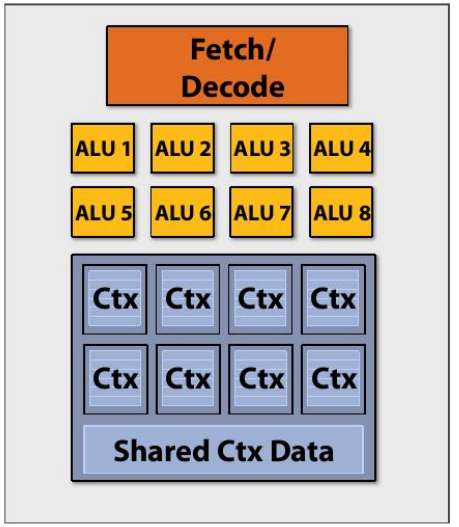
应对方法：第三个思路

- 大量的独立片元相互切换
- 通过片元切换来掩藏延迟

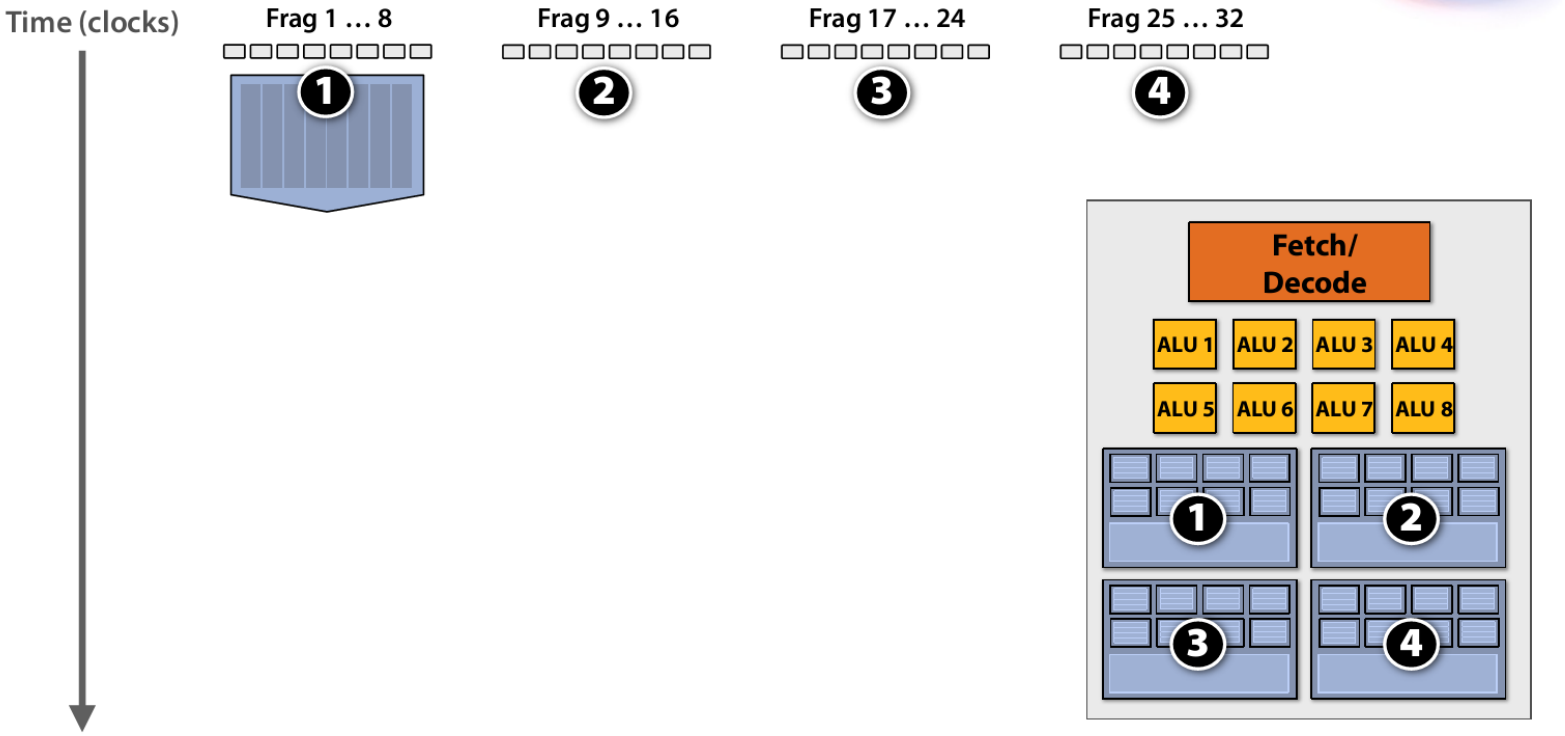
掩藏延迟、延迟掩藏

Hiding shader stalls

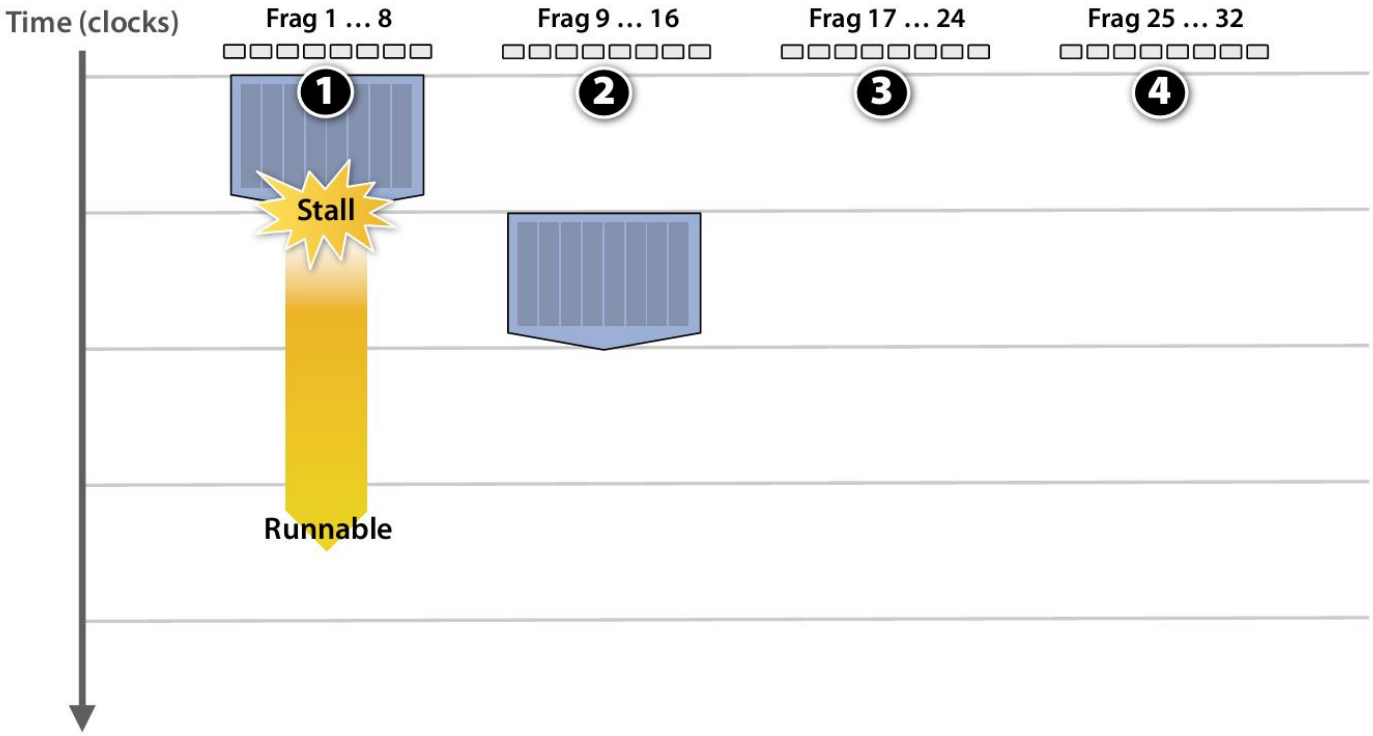




Hiding shader stalls

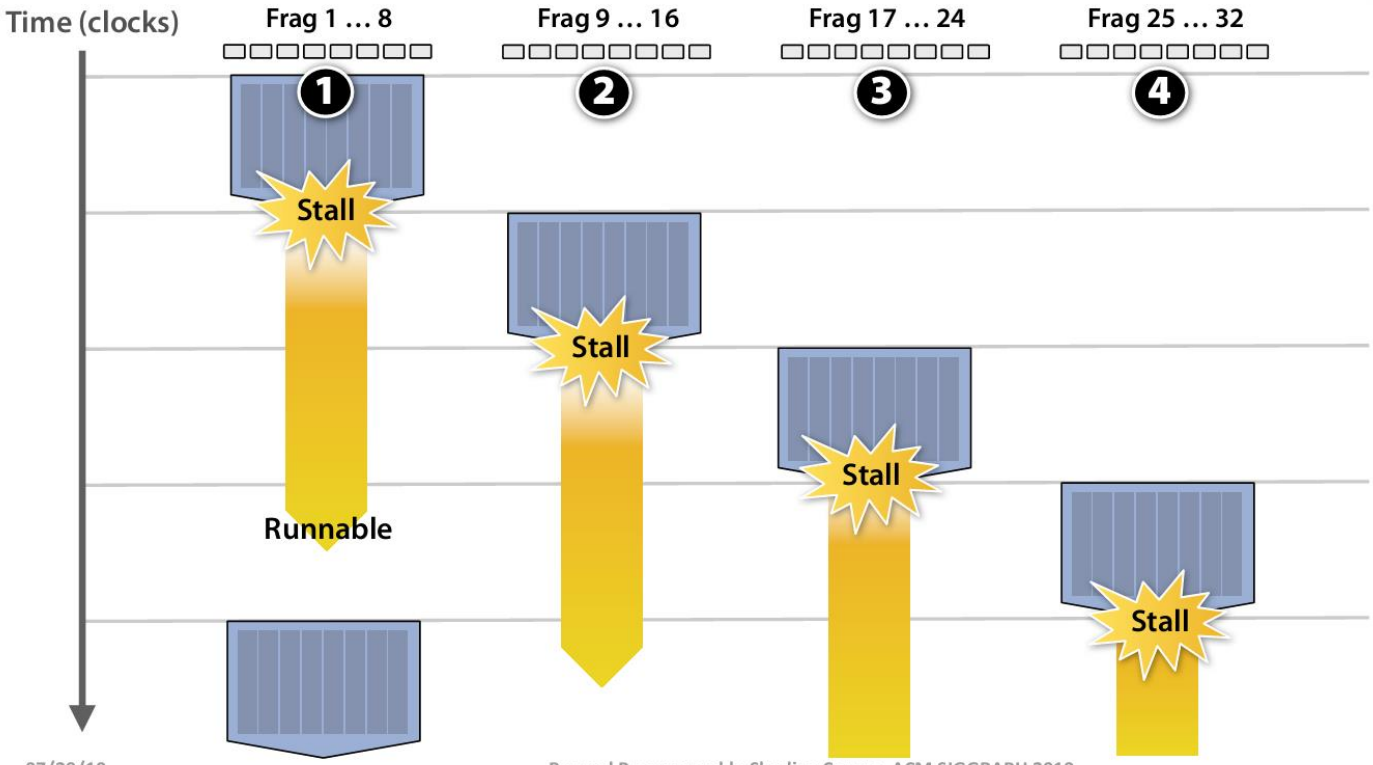


Hiding shader stalls

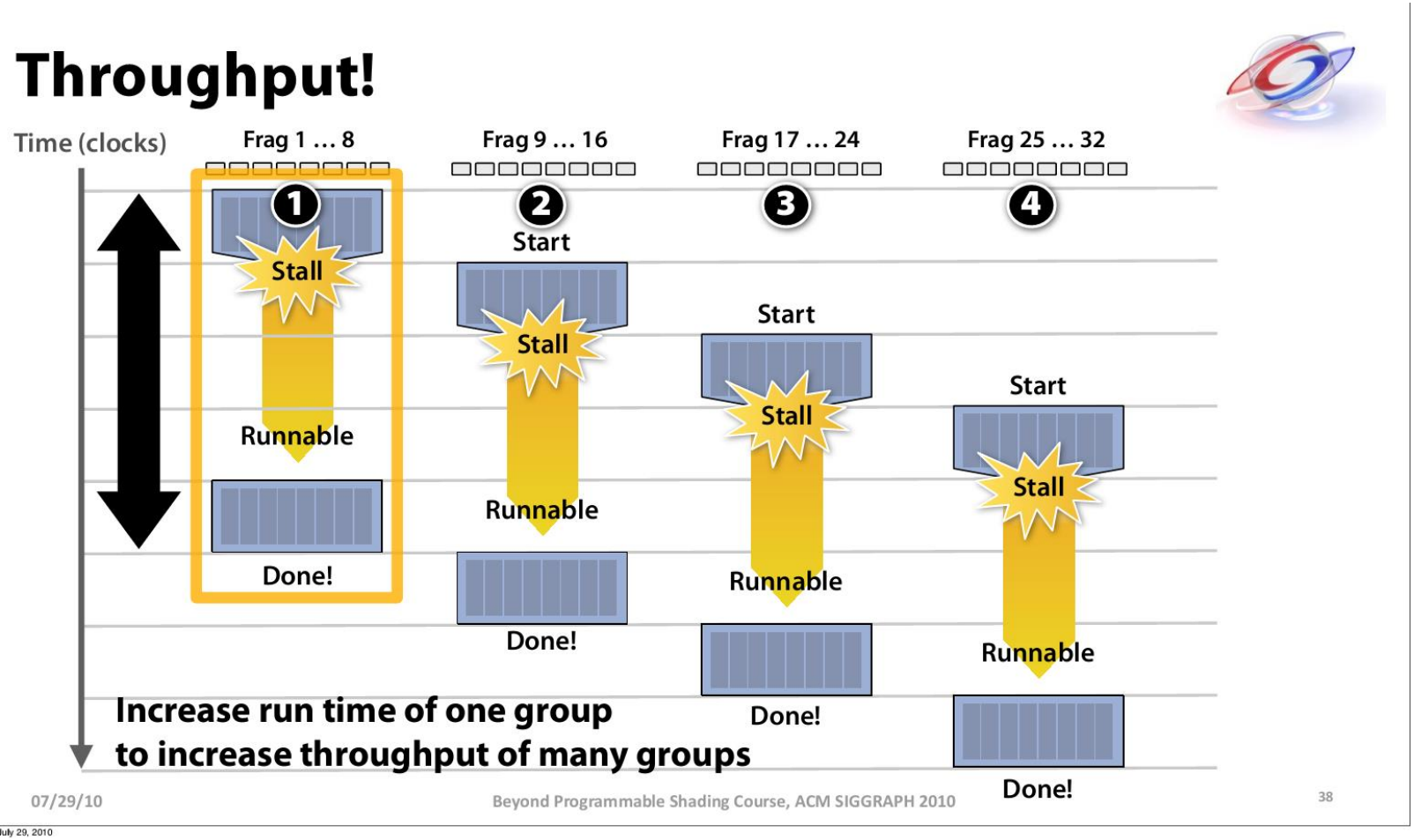


Thursday, July 29, 2010

Hiding shader stalls

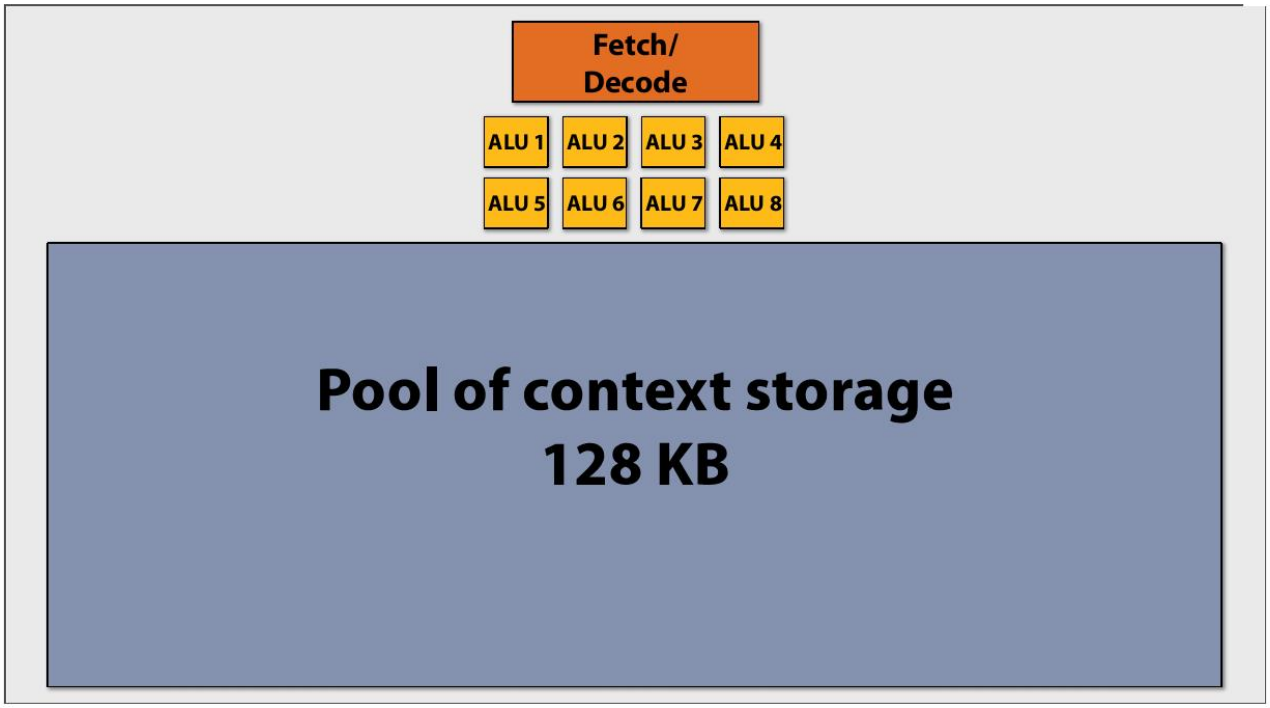


获得较高的吞吐


Throughput!
Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Runnable
Done!
Increase run time of one group
to increase throughput of many groups
07/29/10
Beyond Programmable Shading Course, ACM SIGGRAPH 2010
38
Thursday, July 29, 2010

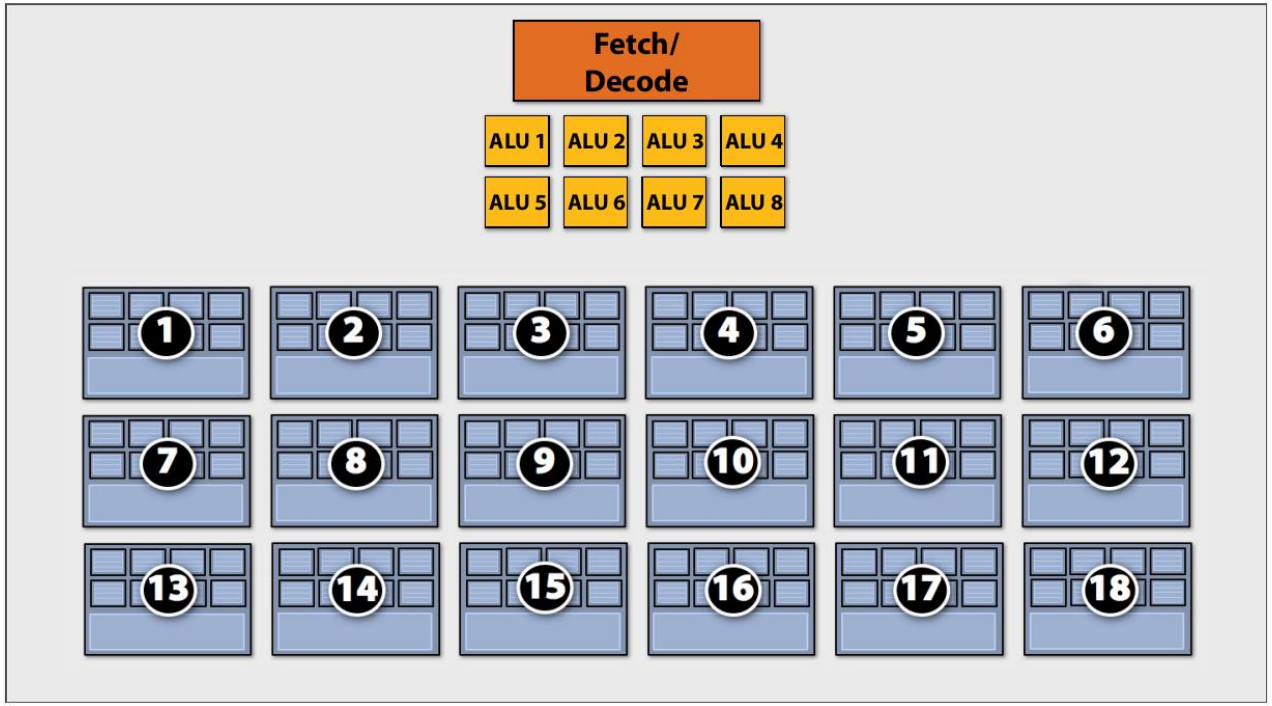
上下文存储空间

Storing contexts



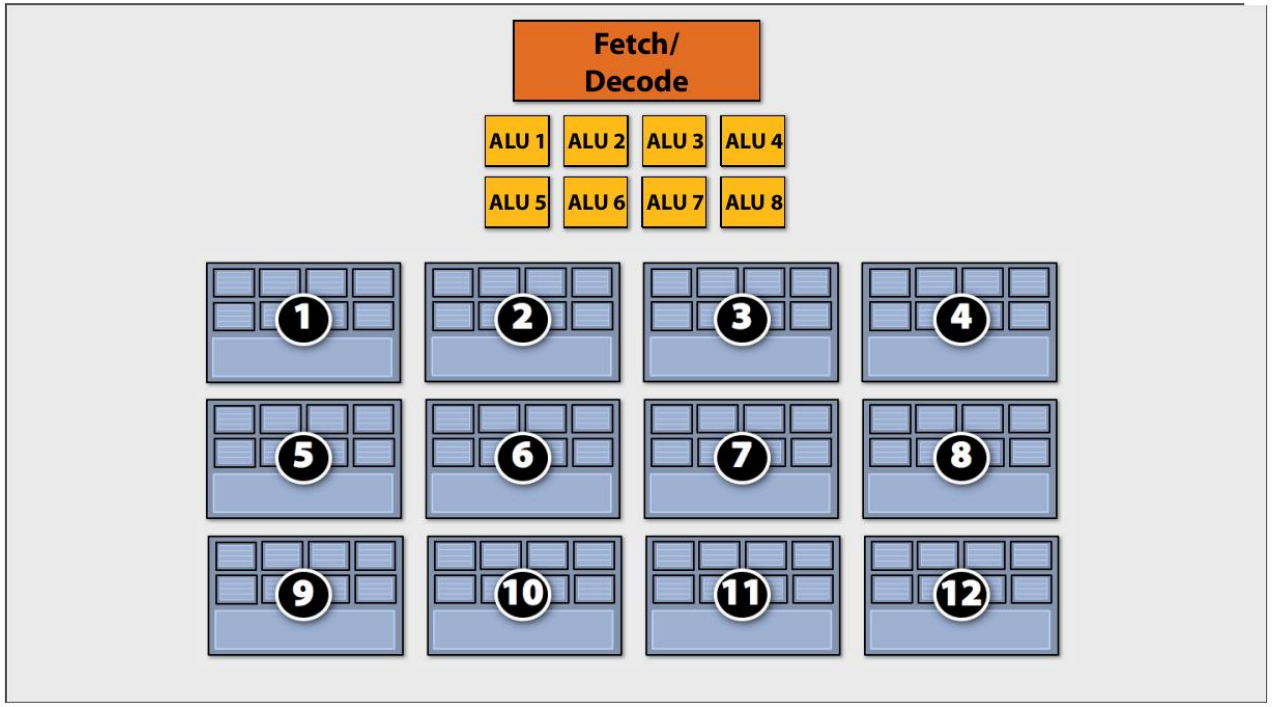
上下文存储池

18个小的上下文：好的延迟掩藏效果



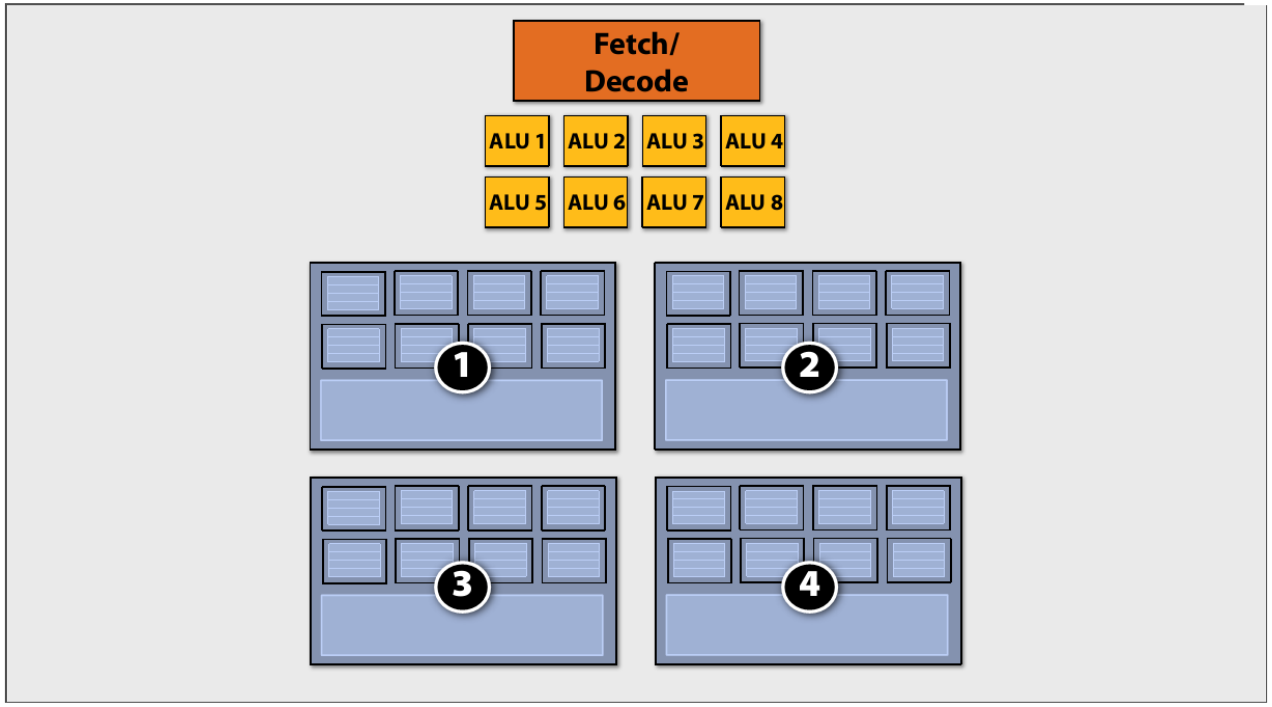
12个中等大小的上下文

Twelve medium contexts



4个大的上下文：延迟掩藏效果较差

Four large contexts (low latency hiding ability)



澄清：上下文切换可以软件也可以硬件管理！

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)

“我”的设计

My chip!

16 cores

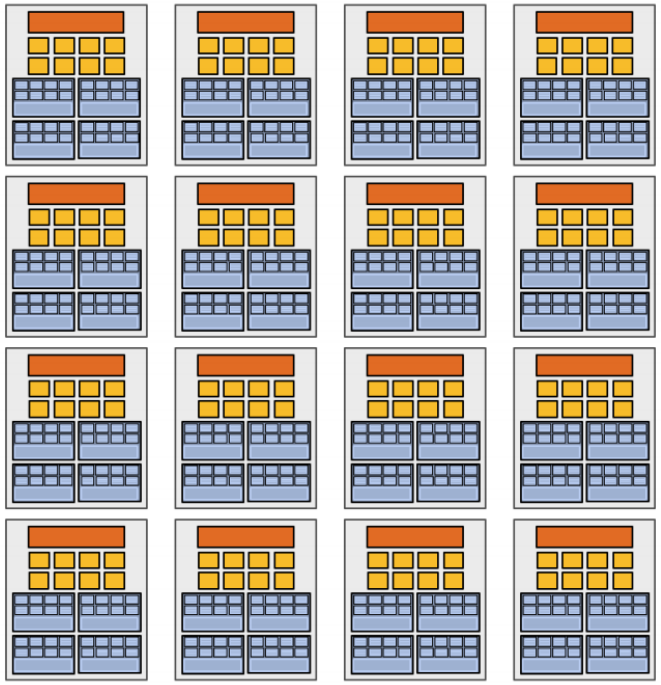
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

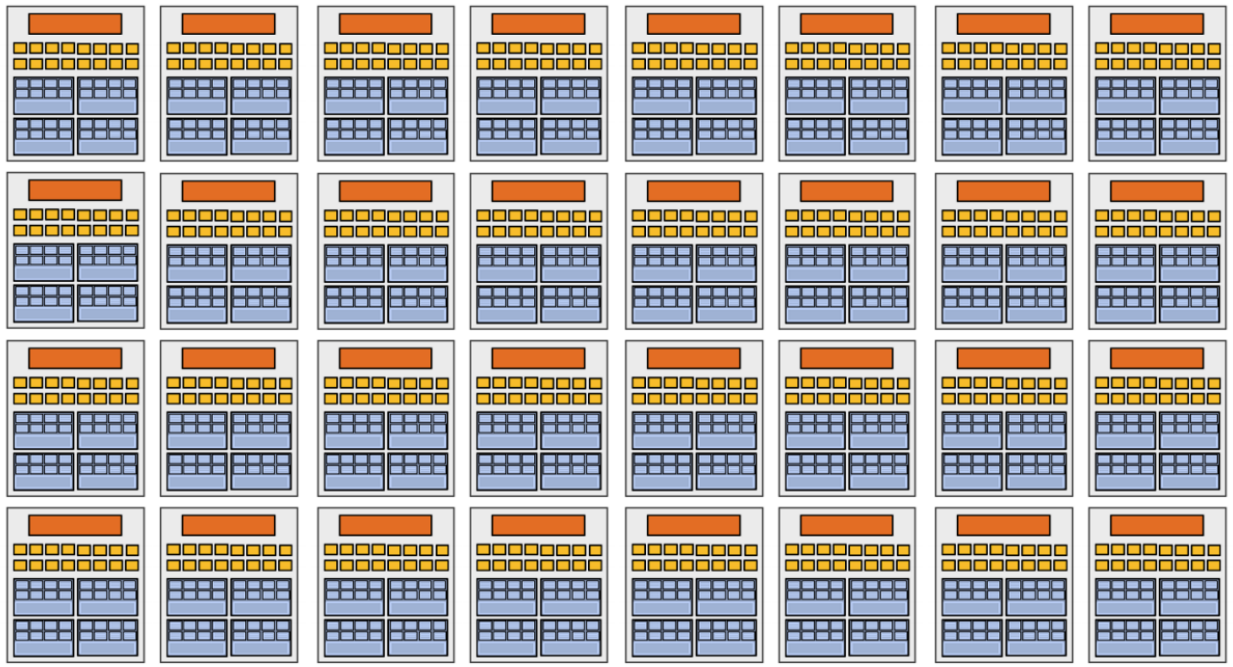
512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



“我”的“理想”设计

My “enthusiast” chip!



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

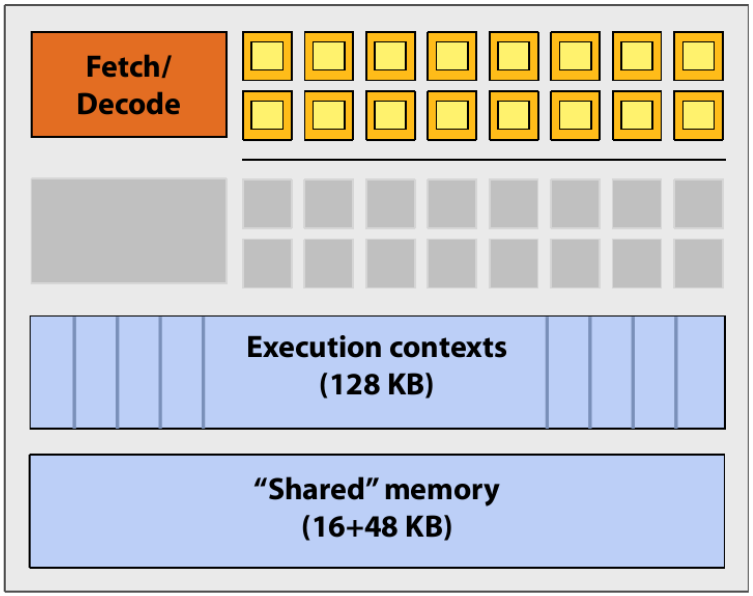
Fermi架构细节

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

NVIDIA GeForce GTX 480 “core”

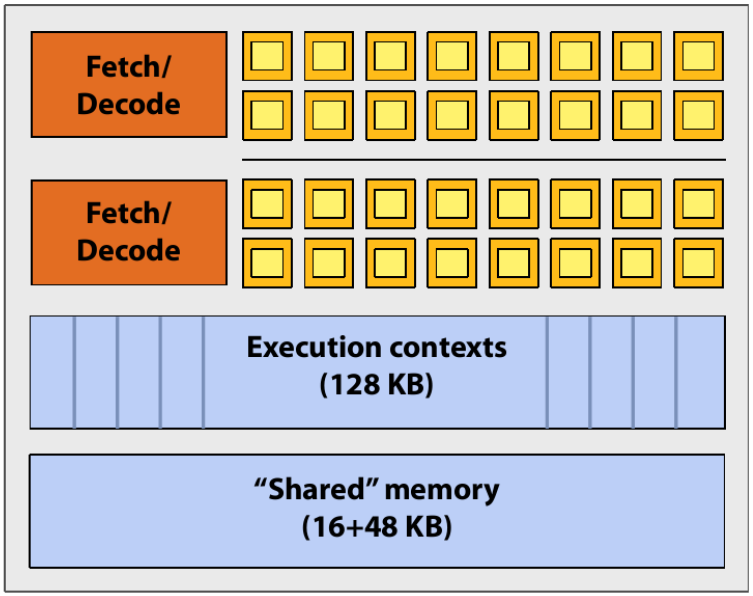


= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- 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"

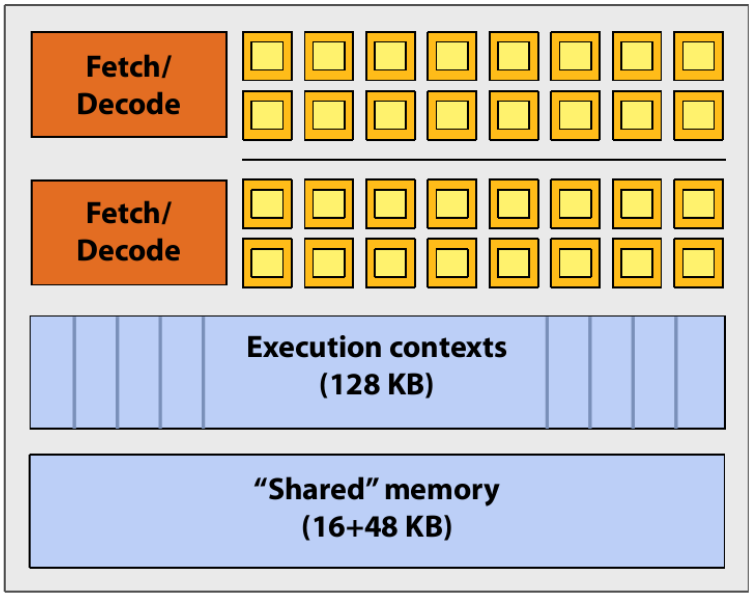


= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- 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"

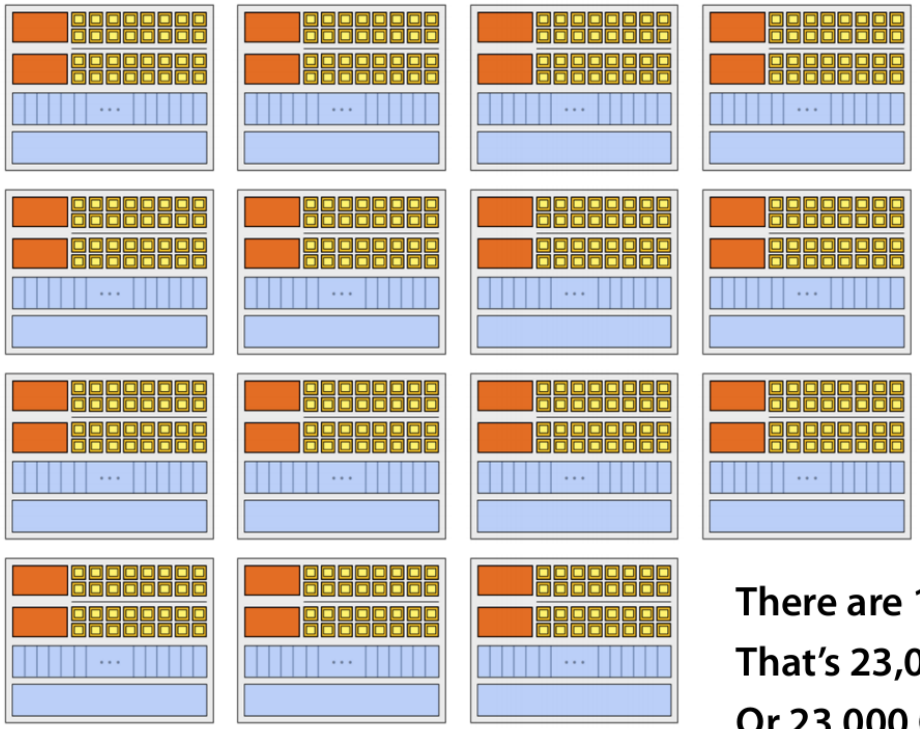


= **CUDA core**
(1 MUL-ADD per clock)

- 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

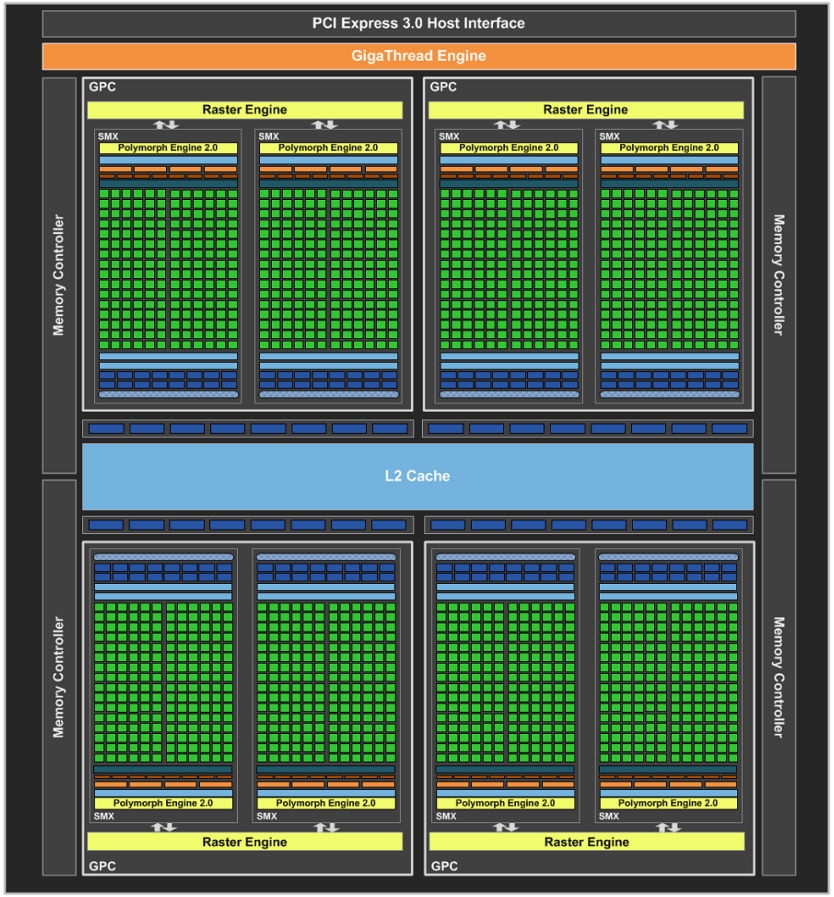


There are 15 of these things on the GTX 480:

That's 23,000 fragments!

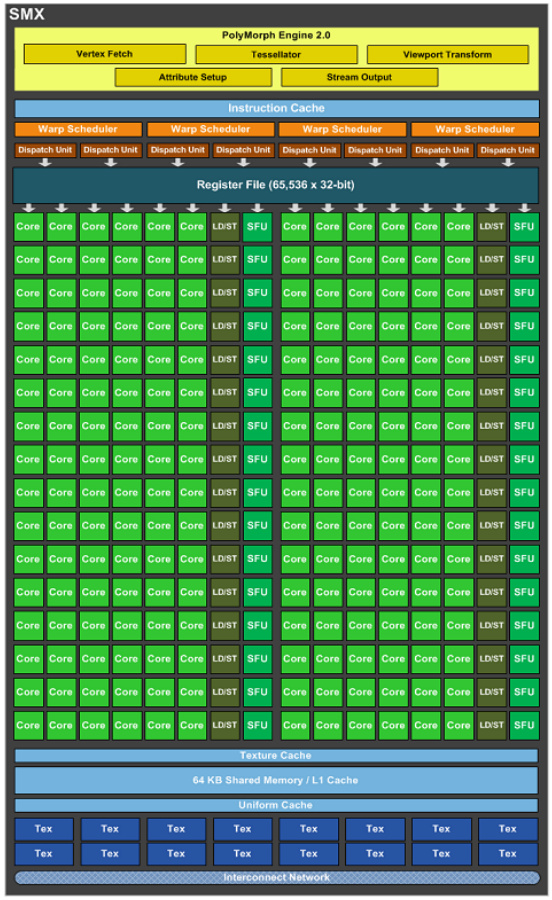
Or 23,000 CUDA threads!

Kepler 架构的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

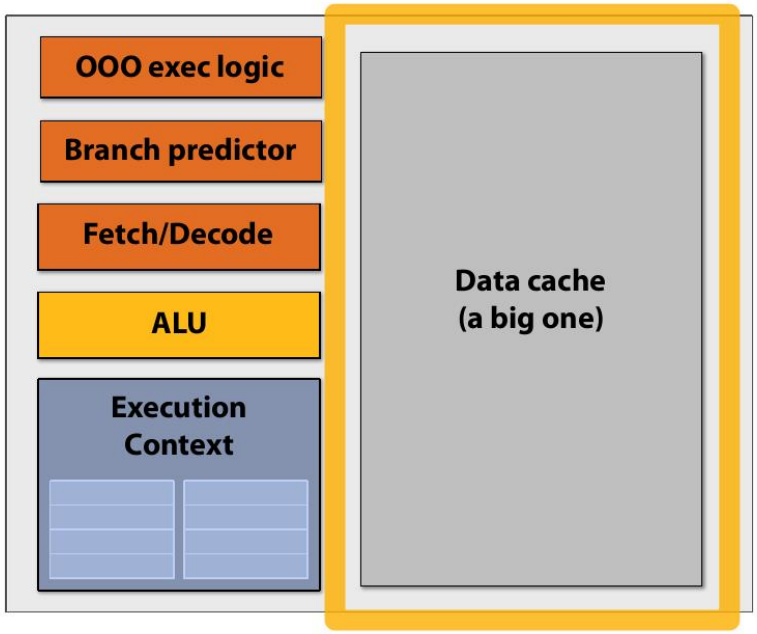
NVIDIA GK110架构

Architecture

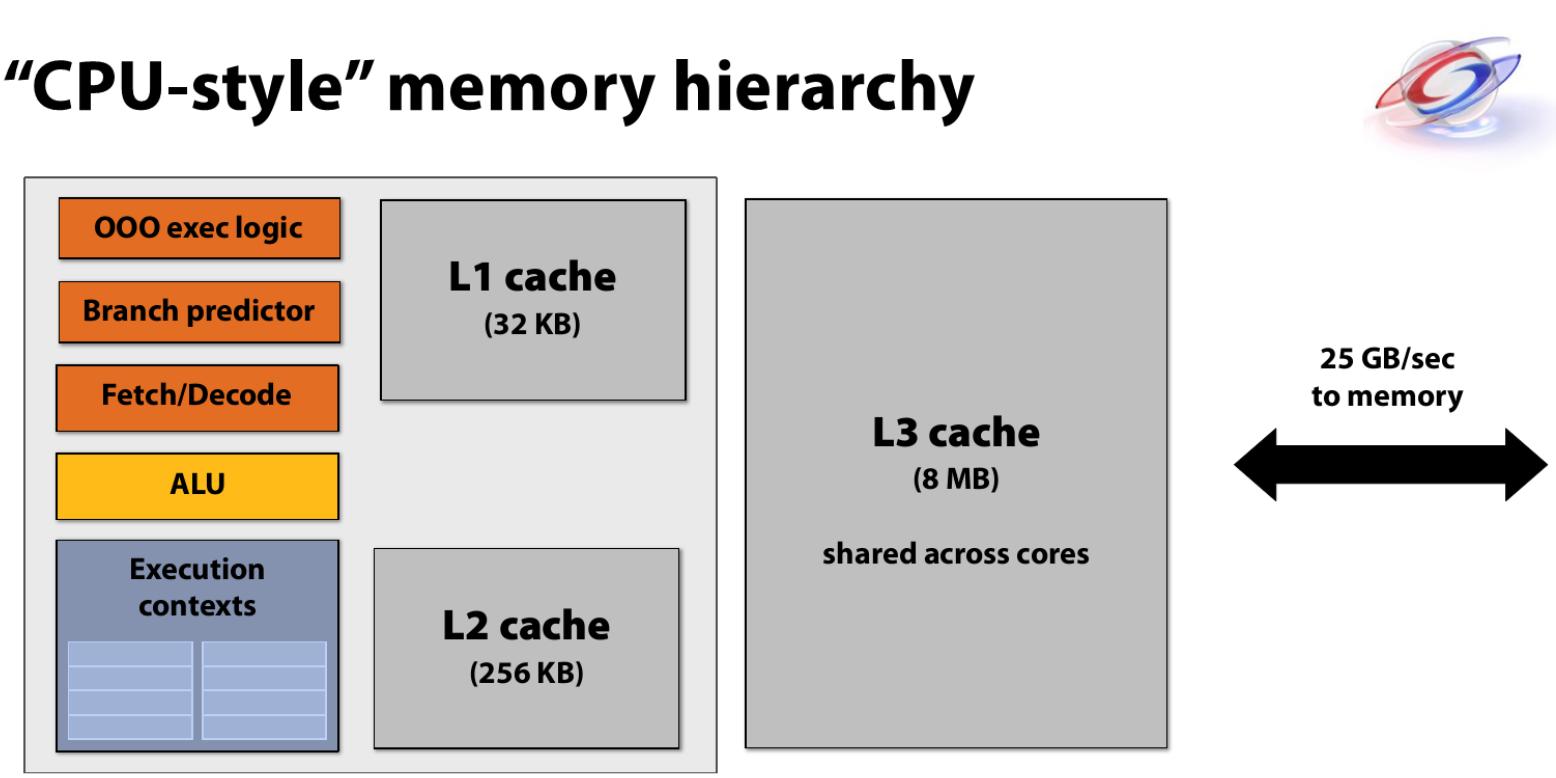
- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3

存储和数据访问

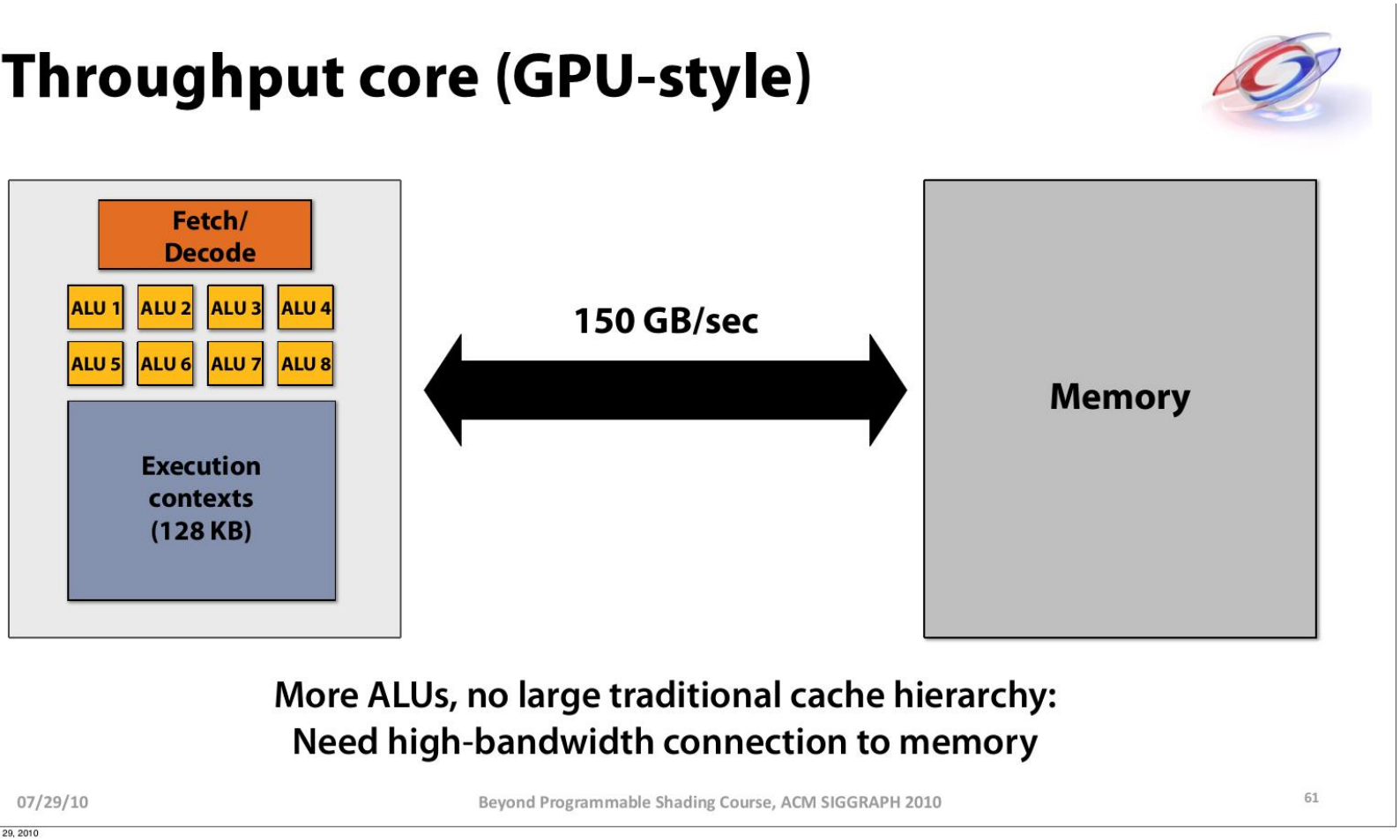
Recall: "CPU-style" core



CPU类型的存储器架构



GPU型的吞吐处理核


Throughput core (GPU-style)
Fetch/
Decode
ALU 1
ALU 2
ALU 3
ALU 4
ALU 5
ALU 6
ALU 7
ALU 8
Execution
contexts
(128 KB)
150 GB/sec
Memory
More ALUs, no large traditional cache hierarchy:
Need high-bandwidth connection to memory
07/29/10
Beyond Programmable Shading Course, ACM SIGGRAPH 2010
61
Thursday, July 29, 2010

带宽是非常宝贵的资源！

Bandwidth is a critical resource

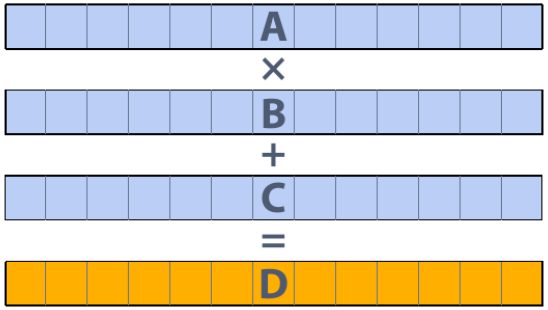
- A high-end GPU (e.g. Radeon HD 5870) has...
  - Over **twenty times** (2.7 TFLOPS) the compute performance of quad-core CPU
  - No large cache hierarchy to absorb memory requests
- GPU memory system is designed for throughput
  - Wide bus (150 GB/sec)
  - Repack/reorder/interleave memory requests to maximize use of memory bus
  - Still, this is only **six times** the bandwidth available to CPU

带宽测试

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!

带宽受限！！！

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.

减少带宽需求

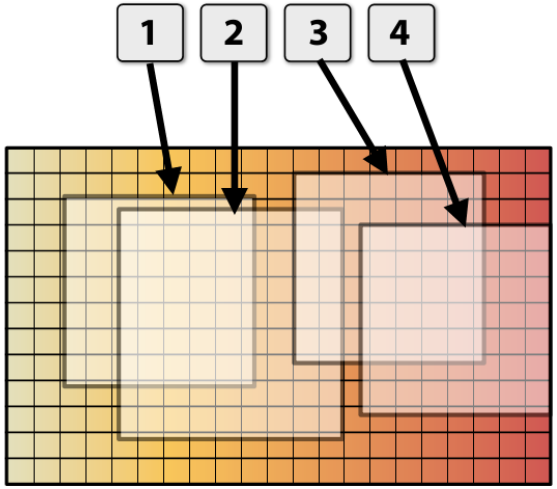
Reducing bandwidth requirements

- 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

Reducing bandwidth requirements

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

Texture data

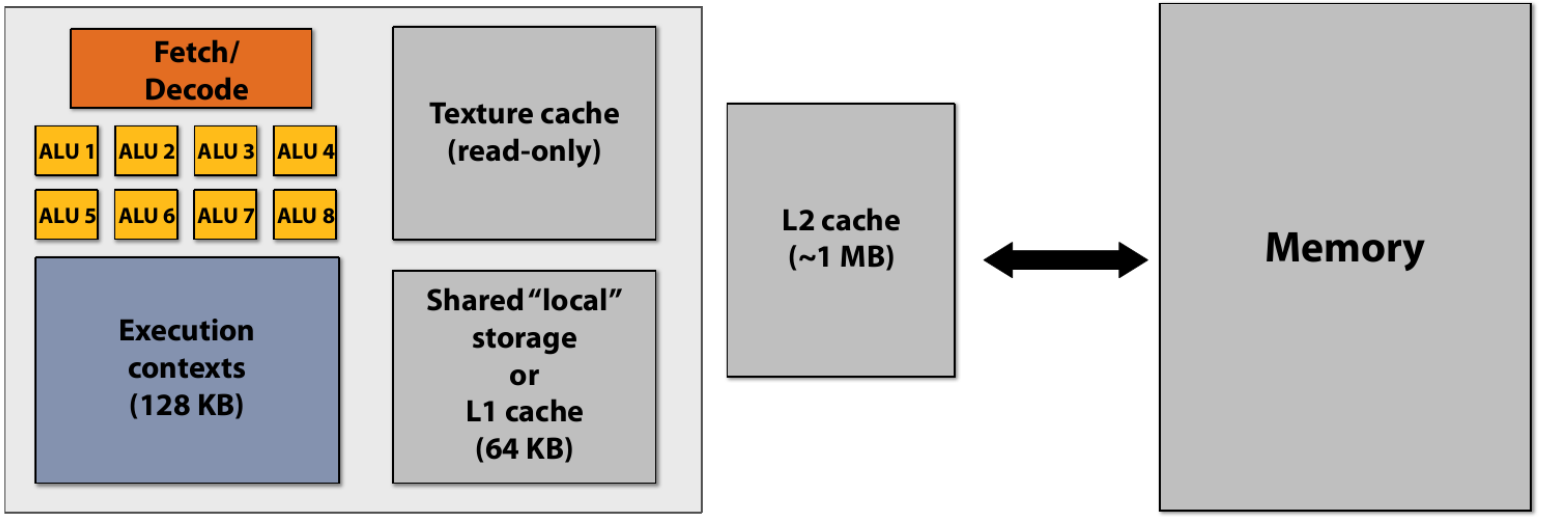


Texture caches:

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

现代GPU的存储器层次结构

Modern GPU memory hierarchy



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

GPU是异构 众核 处理器
针对吞吐优化

高效的GPU任务具备的条件

- 具有成千上万的独立工作
  - 尽量利用大量的ALU单元
  - 大量的片元切换掩藏延迟
- 可以共享指令流
  - 适用于SIMD处理
- 最好是计算密集的任务
  - 通信和计算开销比例合适
  - 不要受制于访存带宽