CUDA程序设计

Lec 2: GPU Architecture Overview

陈俊仕

2024 Fall

计算机科学与技术学院
School of Computer Science and Technology

一些术语

- **指令，Instruction**
  - 指示机器在某个步骤下执行什么操作
  - e.g. MUL、ADD、LD、ST、BR、JMP etc.
- **操作，Operation**
  - 计算单元为完成指令而执行的操作
  - e.g. FLT32 operation, INT operation etc.

一些术语

- **时钟周期，Clock Cycle**
  - 衡量指令时间的基本单位
  - e.g. MUL、ADD、LD、ST、BR、JMP etc.
- **时钟速度/频率，Clock Speed/Frequency**
  - 每秒（执行）的时钟周期数
  - e.g. 1 GHz = 10^9 cycles per second

一些术语

- 延迟，**Latency**（memory or instruction）
  - 衡量指令时间的基本单位
    - e.g. 200 cycles for global memory read
- 吞吐量，**Throughput**（memory or instruction）
  - 一定时间段内处理的项目/指令的数量
    - e.g. Instructions Per Cycle (IPC), or GB/s etc.
- 带宽，**Bandwidth**（memory）
  - 数据传输速率
    - e.g. 1008 GB/s for a RTX 4090

一些术语

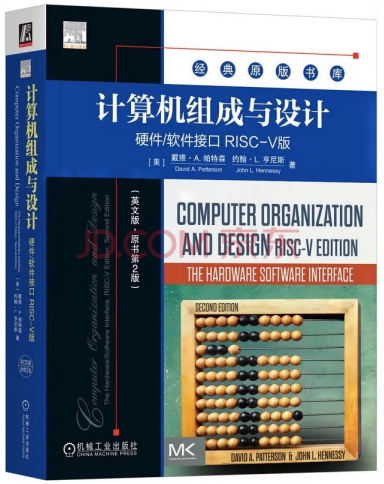
- **FLOPS** – Floating-point OPerations per Second
- **GFLOPS** – One billion (10^9) FLOPS
- **TFLOPS** – 1000 GFLOPS
- **PFLOPS** – 1000 TFLOPS
- **EFLOPS** – 1000 PFLOPS

了解现代CPU的架构

- 流水线 - Pipeling
- 分支预测 – Branch Prediction
- 超标量 - SuperScalar
- 乱序执行 – Out-of-Order Execution
- 存储器层次 – Memory Hierarchy
- 向量操作 – Vector Operations
- 多核处理 – Multi-core

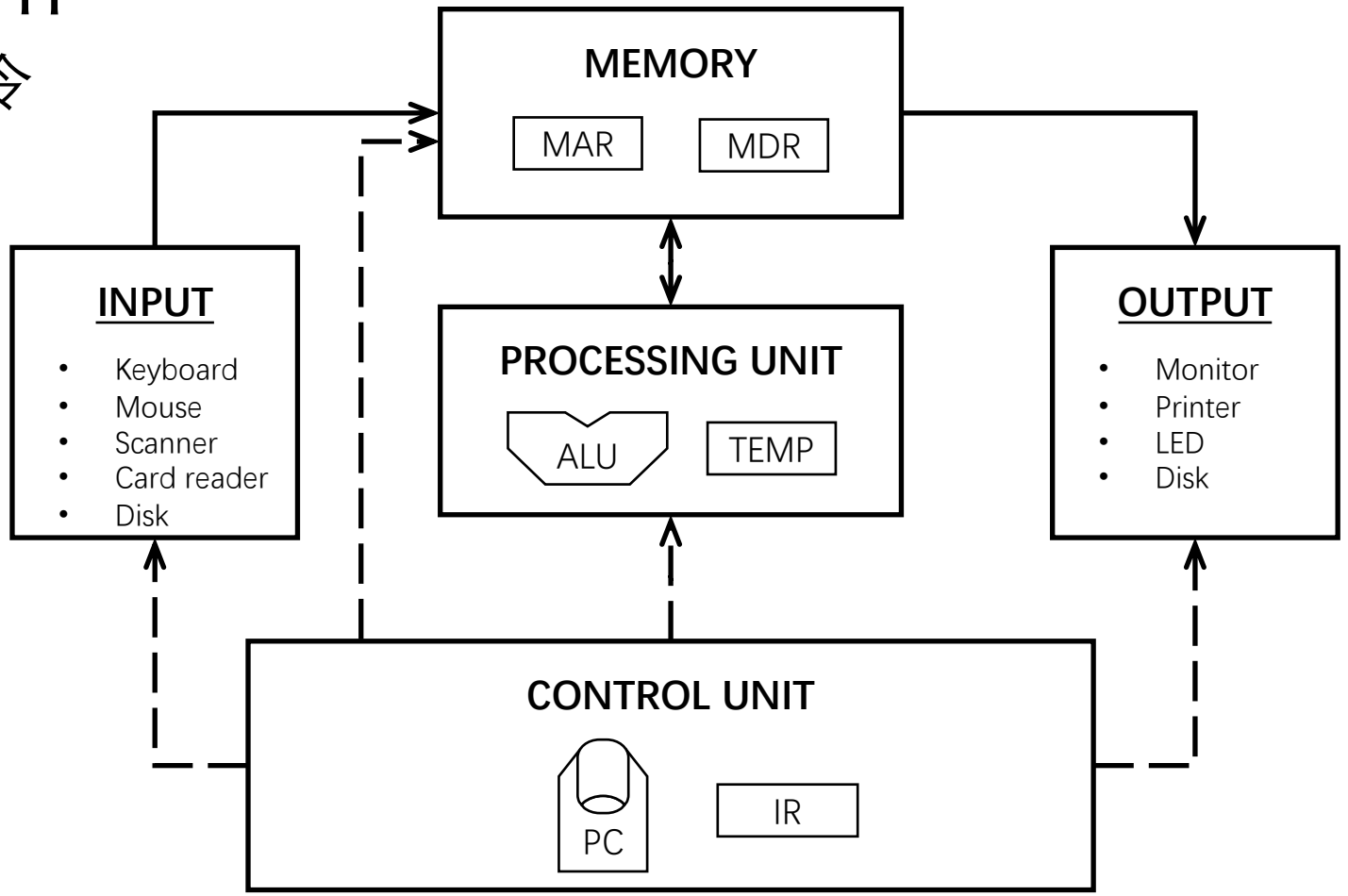
参考资料

- **参考书目：**
  - 《计算机组成与设计：硬件/软件接口》
  - 《计算机体系结构：量化研究方法》
- **参考课程：**
  - Joe Devietti, CIS 501, University of Pennsylvania



什么是CPU?

- **执行指令、处理数据的器件**
  - 完成基本的逻辑和算术指令
- **现在增加了复杂功能**
  - 内存接口
  - 外部设备接口
- **包含大量晶体管**
  - 非常多（上亿…）



指令

- **举例：**
  - 算术指令：**ADD r3, r4 → r4**
  - 访存指令：**LD [r4] → r7**
  - 控制指令：**JZ END**
  - 对于一个编译好的程序，最优化目标：

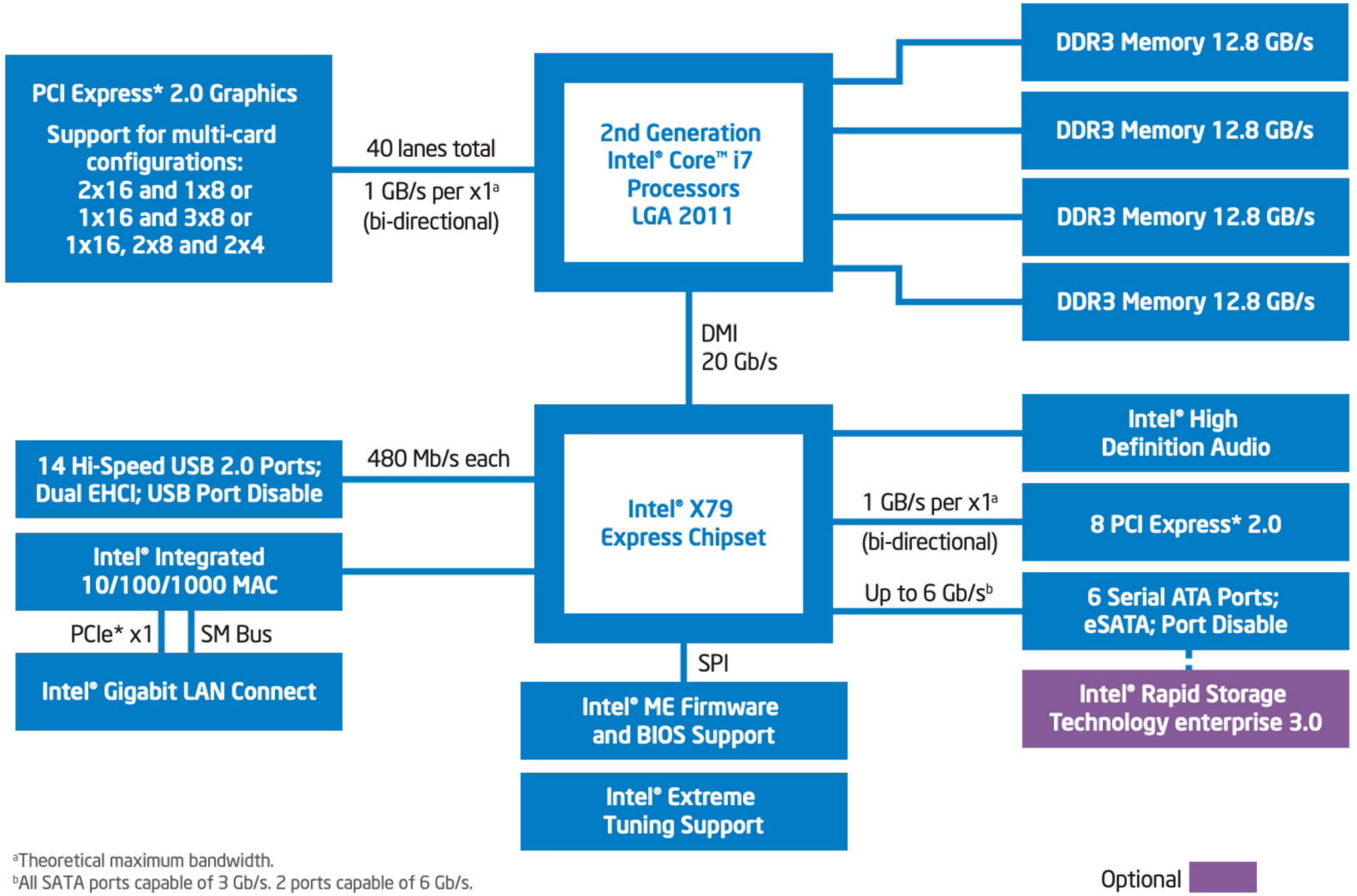
$$\frac{cycles}{instruction} \times \frac{seconds}{cycle}$$

CPI（每条指令的时钟数）& 时钟周期
注：这两个指标彼此并不独立

桌面应用 Desktop Programs

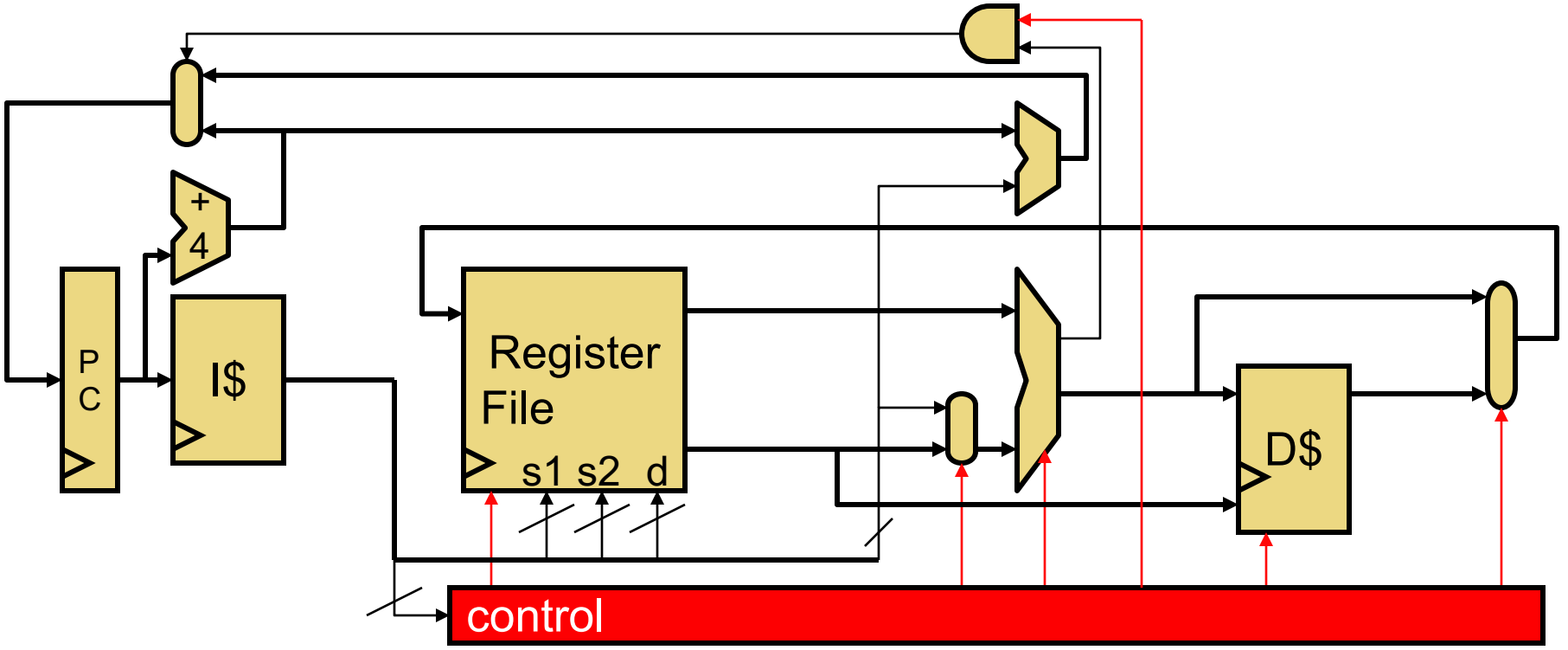
- 轻量级进程，少量线程 Lightly threaded
- 大量分支和交互操作 Lots of branches
- 大量的存储器访问 Lots of memory access
- 真正用于数值运算的指令很少

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



Intel® X79 Express Chipset Block Diagram

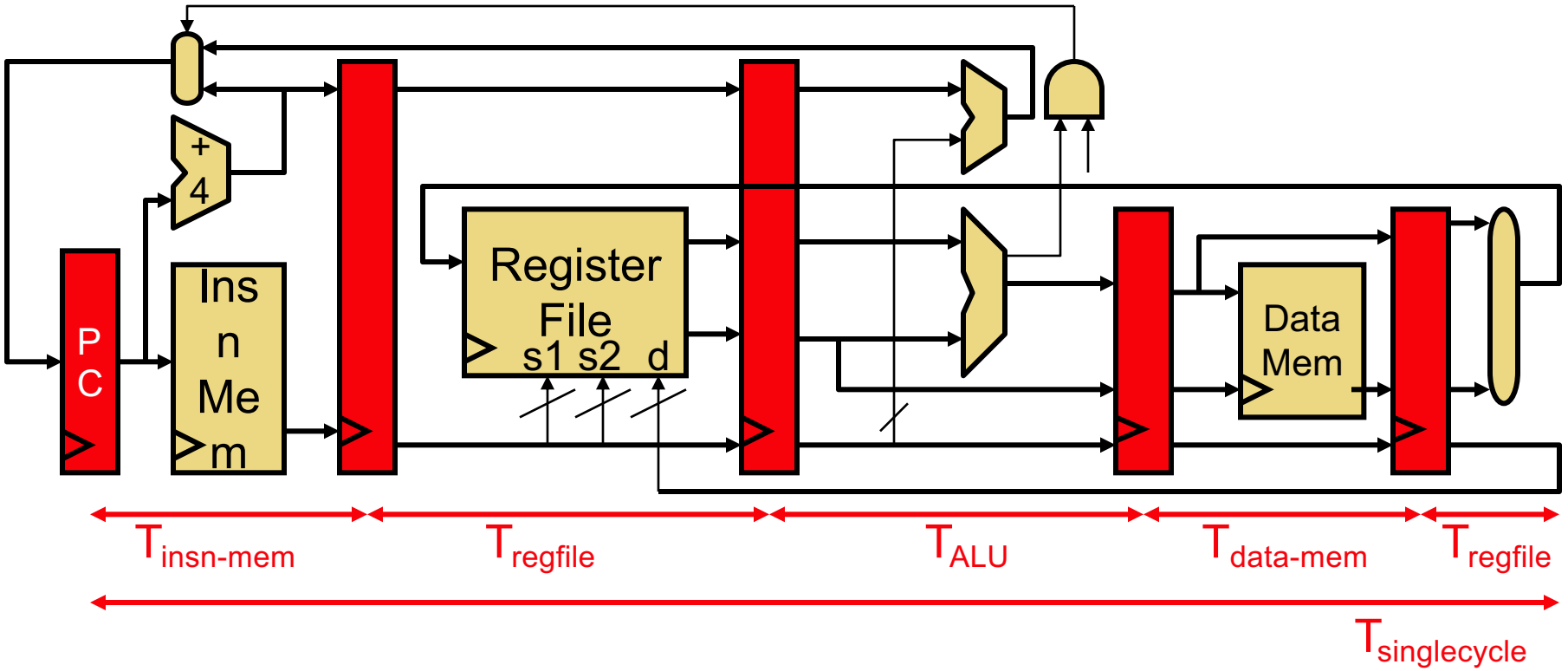
A Simple CPU Core



Fetch → Decode → Execute → Memory → Writeback

Image: Penn CIS501

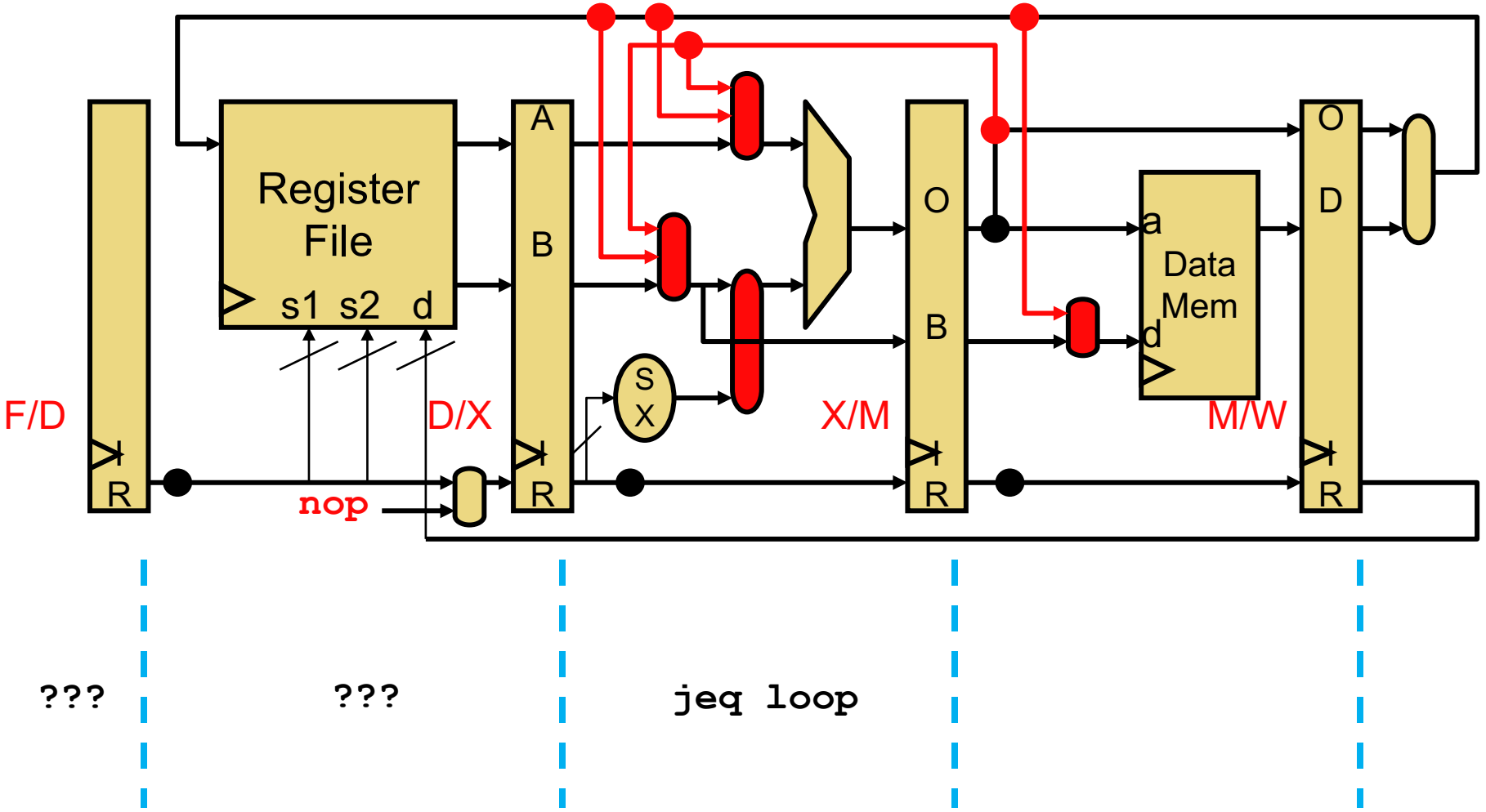
Pipelining


+
4
P
C
Ins
n
Me
m
Register
File
s1 s2 d
Data
Mem
T insn-mem
T regfile
T ALU
T data-mem
T regfile
T singlecycle

Pipelining

- 利用指令级并行（ILP，Instruction-Level Parallelism）
  - \+ 极大的减小时钟周期（Significantly reduced clock period）
  - – 增加一些延迟和芯片面积（Slight latency & area increase，pipeline latches）
  - ? 具有依赖关系的指令
  - ? 分支怎么处理
- 流水线长度：Alleged Pipeline Lengths
  - Core 2：14级
  - Pentium 4（Prescott）： 20级
  - Sandy Bridge：14 < ? < 20

Branches


Register File
s1 s2 d
A
B
S
X
O
B
a
Data Mem
d
O
D
F/D
D/X
X/M
M/W
R
R
R
R
nop
???
???
jeq loop

分支预测 Branch Prediction

- 猜测下一条指令
- **Guess what instruction comes next**
- 基于过去的分支记录
- **Based on branch history**
- 举例：基于全局记录的两层预测
- **Example: two-level predictor with global history**
  - Maintain history table of all outcomes for M successive branches
  - Compare with past N results (history register)
  - Sandy Bridge employs 32-bit history register

分支预测 Branch Prediction

+ 现代预测器准确度大于90%

Modern predictors > 90% accuracy

提升性能及能量效率

Raise performance and energy efficiency

– 面积增加，Area increase

– 可能会增加延迟，Potential fetch stage latency increase

– 带来安全问题？Spector/Meltdown

分支断定 Another option: Predication

- 用条件语句替换分支
- Replace branches with conditional instructions

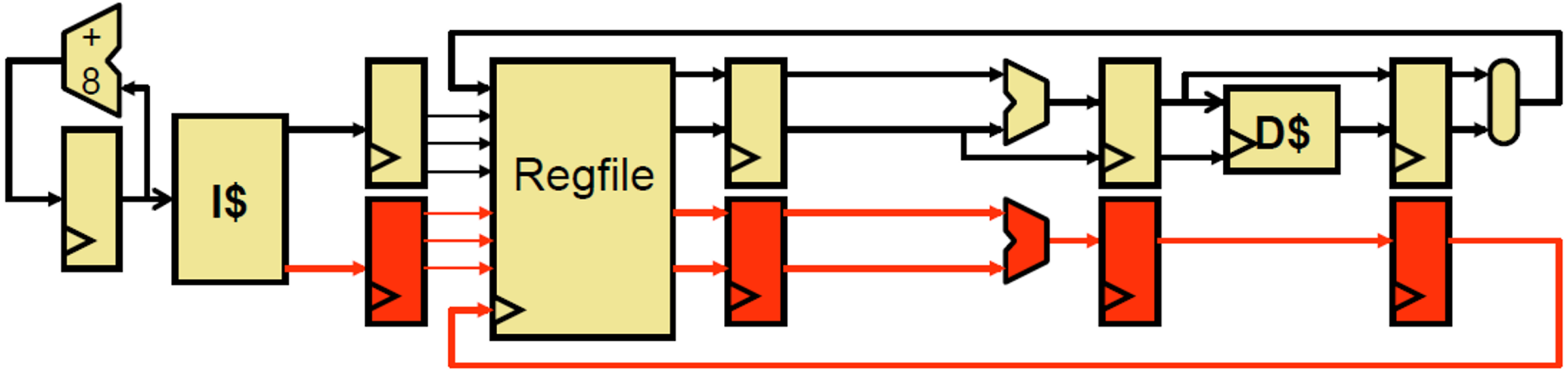
```
;  if (R1 == 0) R3 = R2
CMOVEQ R3, R1, R2
```

- 不使用分支预测器，Avoids branch predictor
  - + Avoids area penalty, misprediction penalty
  - 减少面积，减少错误预测
  - Introduces unnecessary `nop` if predictable branch
- 在GPU中使用分支断定（GPUs also use prediction）

提升IPC

- 常规IPC（instructions/cycle）受限于 I$
  - instruction/clock

- 超标量 Superscalar – 增加流水线宽度



提升IPC

+ **峰值IPC为N（N路超标量）**

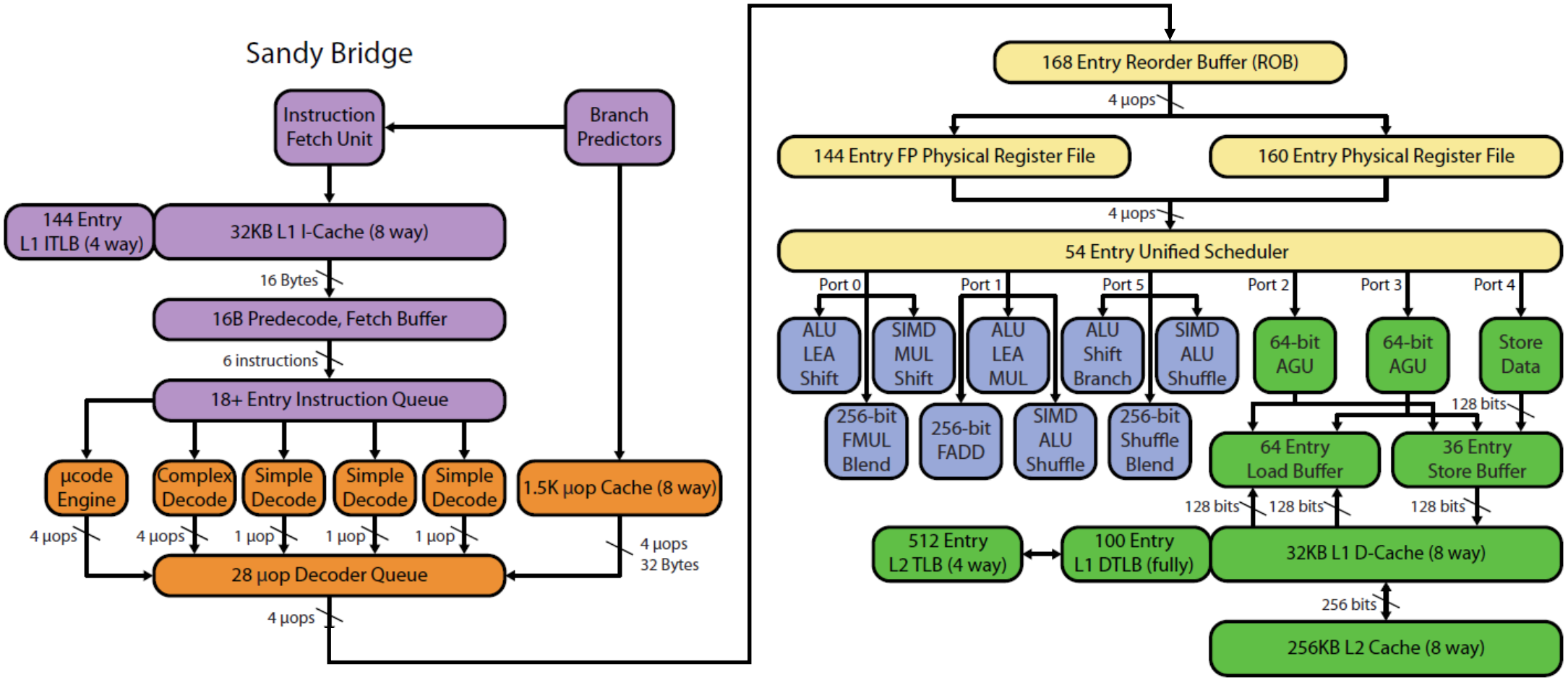
Peak IPC now at N (for N-way superscalar)

- 分支和调度会产生开销
- 需要一些技巧来逼近峰值

– **增加了面积**

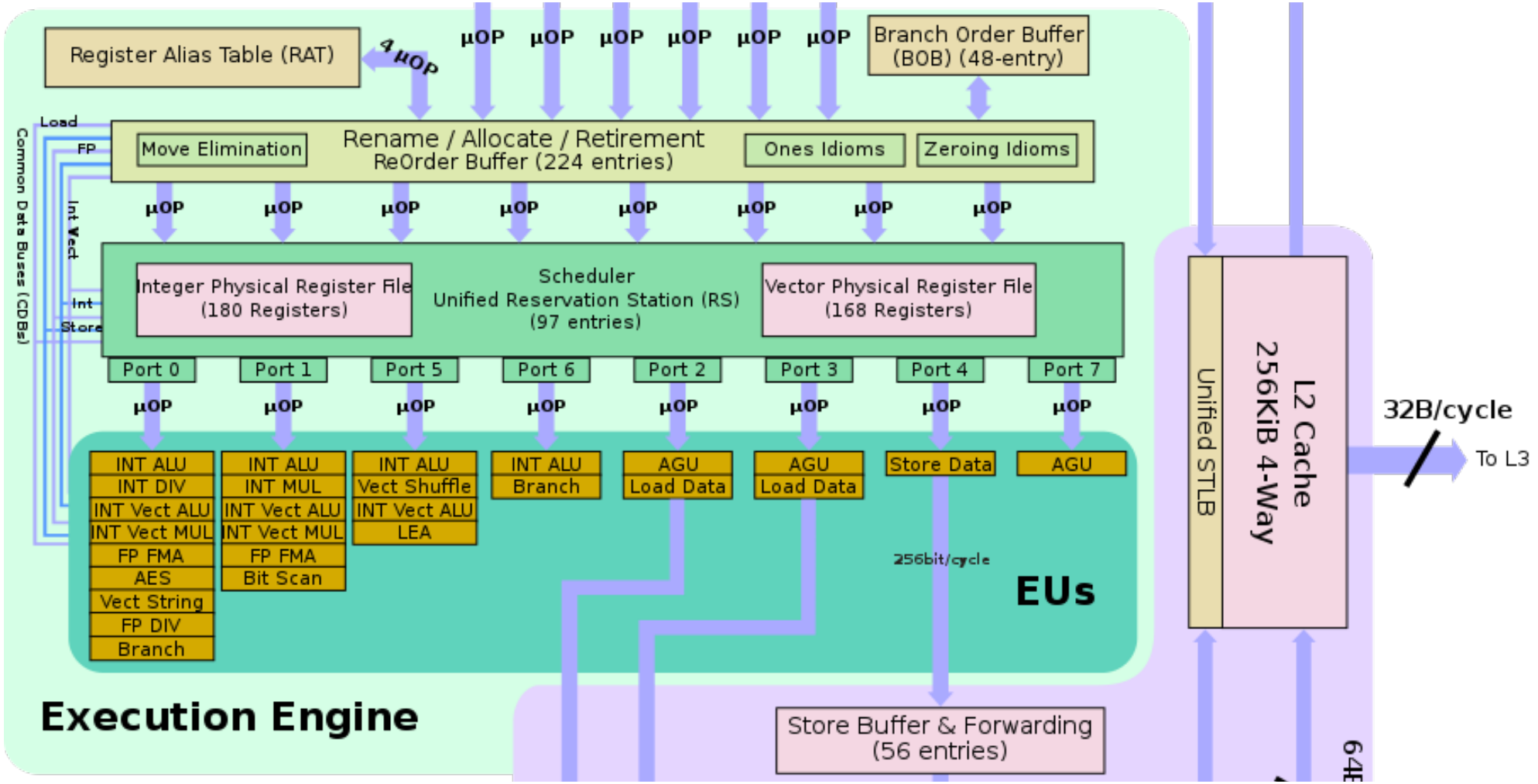
- N倍资源使用
- 旁路网络N^2
- 需要更多的寄存器和存储器带宽

Sandy Bridge Superscalar



https://www.realworldtech.com/sandy-bridge/10/

Coffee Lake Superscalar



https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake#Individual_Core

指令调度 Scheduling

- 考虑如下指令:

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

- `add` 依赖 `xor` (Read-After-Write, RAW)
- `addi` 依赖 `sub` (RAW)
- `xor` 和 `sub` 无依赖 (Write-After-Write, WAW)

寄存器重命名 Register Renaming

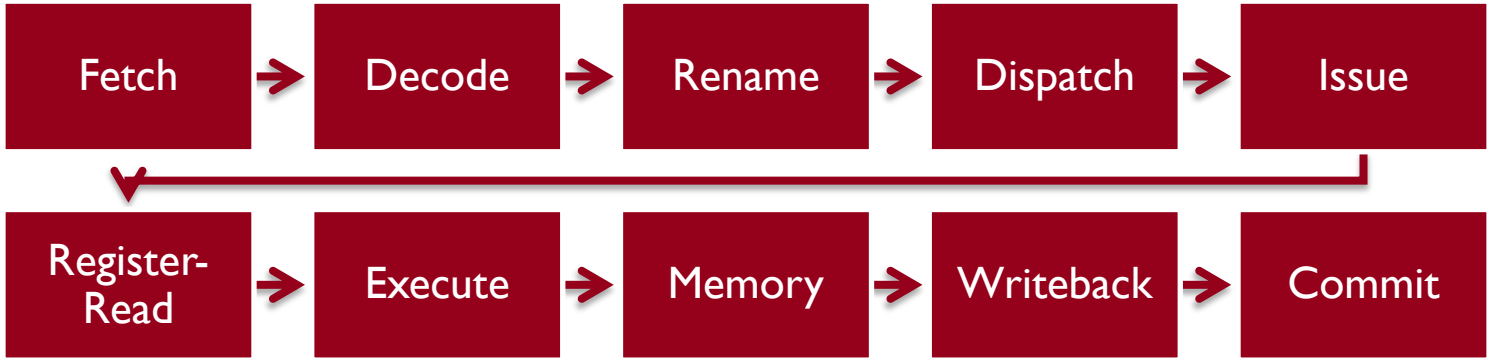
- 替换寄存器:

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

- xor 和 sub 可以并行执行

乱序执行 Out-of-Order Execution

- 重排指令，最大化吞吐率

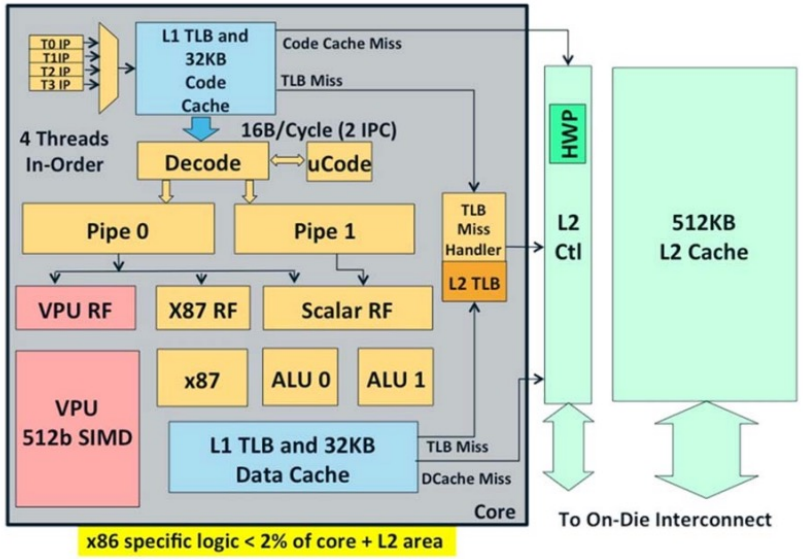


- 重排缓冲区，Reorder Buffer (ROB)
  - 记录所有执行中的指令状态
- 物理寄存器文件，Physical Register File (PRF)
- 发射队列/调度器，Issue Queue/Scheduler
  - 选择下一条执行的指令

乱序执行 Out-of-Order Execution

+ IPC接近理想状态
– 面积增加
– 功耗增加

- 顺序执行 Modern Desktop/Mobile In-order CPUs
  - Intel Atom, Intel Knights Corner
  - ARM Cortex-A8
- 乱序执行
  - Intel Pentium Pro and onwards
  - ARM Cortex-A9
  - Qualcomm Krait



Knights Corner Core

Memory Hierarchy

- 存储器越大越慢
- 粗略的估计:

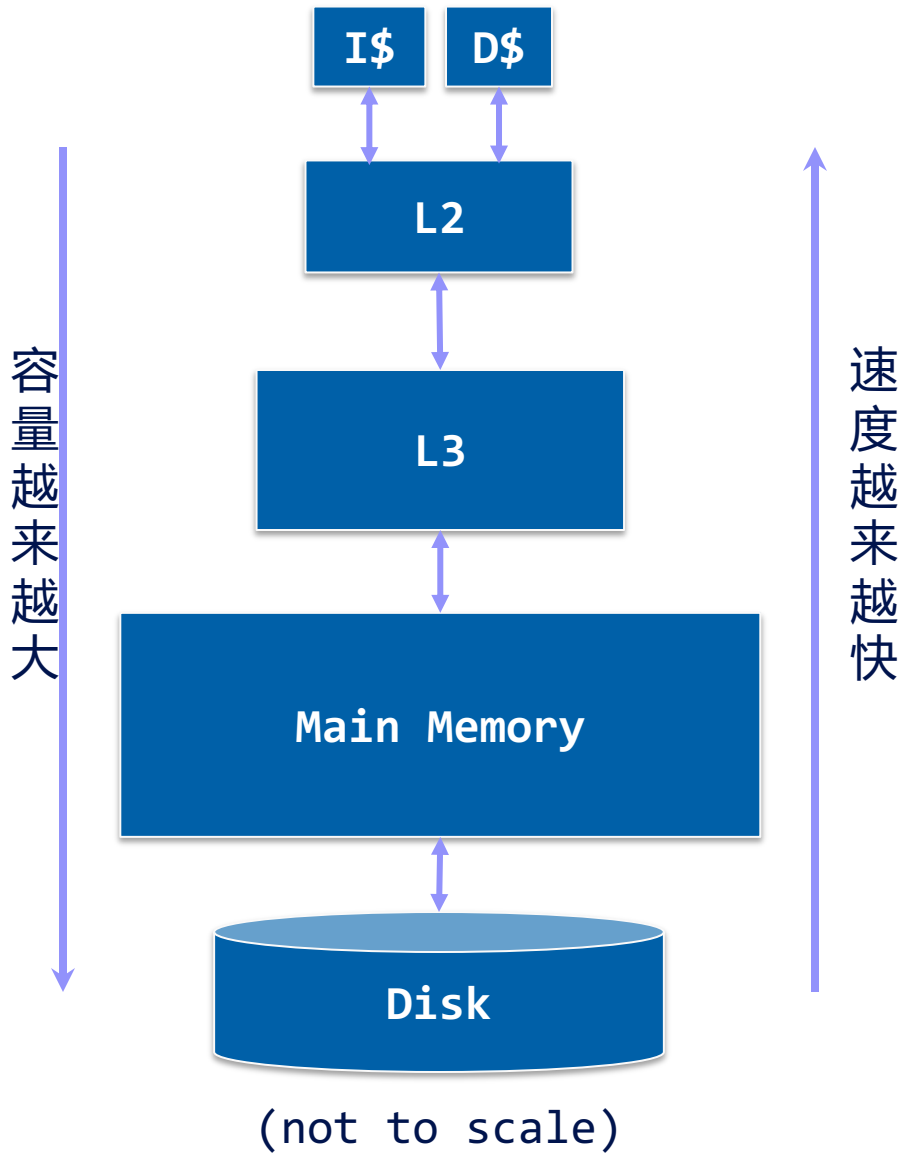
	延迟	带宽	大小
SRAM (L1, L2, L3)	1-2ns	200-3000GBps	1-20MB
DRAM (memory)	70ns	20GBps	1-20GB
Flash/SSD (disk)	70-90µs	200-500MBps	100-1000GB
HDD (disk)	10ms	1-150MBps	500-3000GB

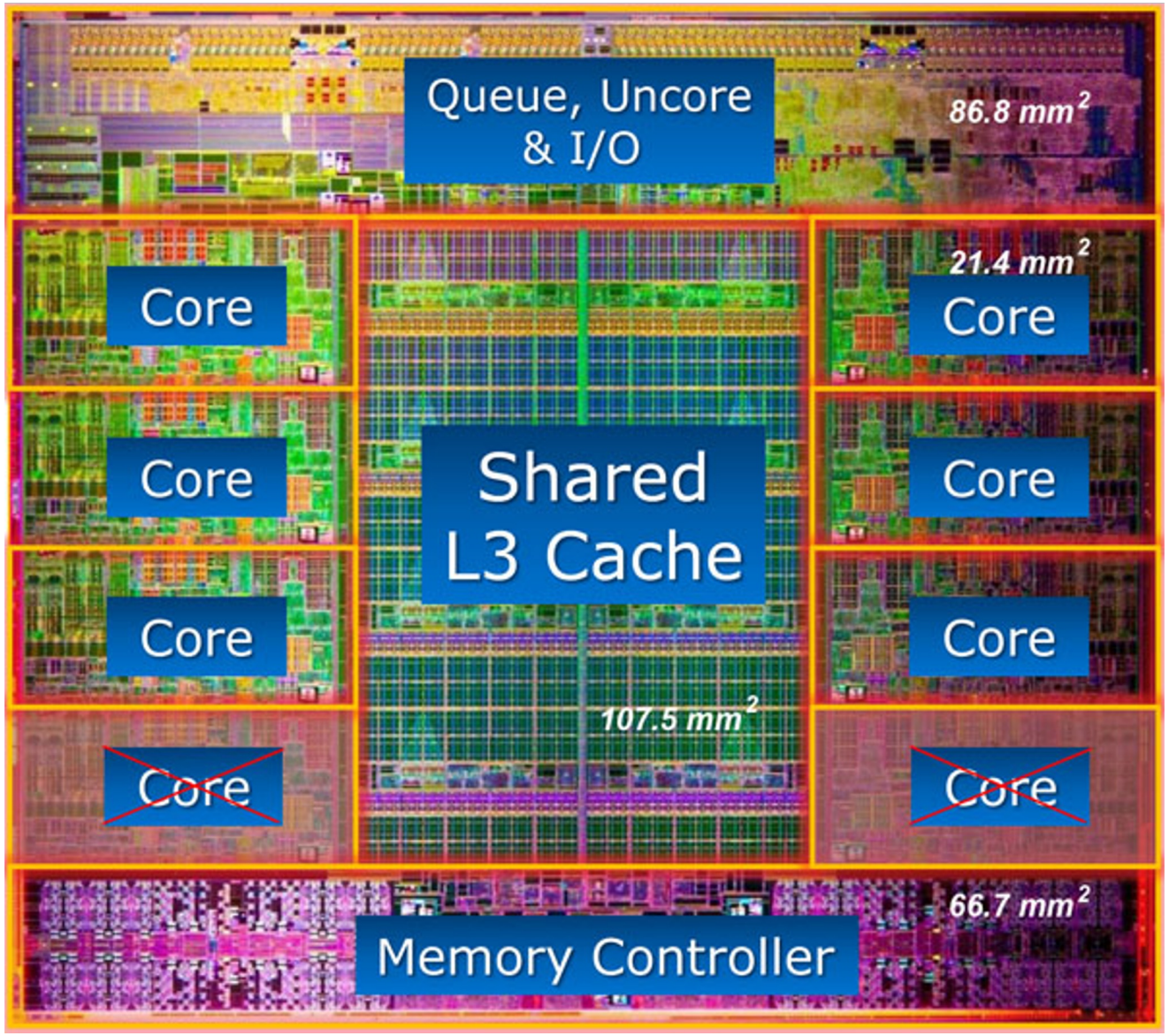
缓存Caching

- 将数据放在尽可能近的位置
- 利用:
  - **时间局部性，Temporal locality**
    - 刚刚使用过的数据很可能会被再次使用
  - **空间局部性，Spatial locality**
    - 倾向于使用周围的邻近的数据

Cache Hierarchy

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





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

Source: https://en.wikichip.org/wiki/intel/core_i7ee/i7-3960x

CPU内部的并行性

- **指令级并行 Instruction-Level (ILP)**
  - 超标量，Superscalar
  - 乱序执行，Out-of-order
- **数据级并行 Data-Level Parallelism (DLP)**
  - 向量计算，Vectors
- **Thread-Level Parallelism (TLP)**
  - 同时多线程，Simultaneous Multithreading (SMT)
  - 多核，Multicore

Vectors Motivation

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

数据级并行 Data-level Parallelism

- **单指令多数据 Single Instruction Multiple Data (SIMD)**
  - 加大执行单元（ALU）宽度
  - 同样地，也加大寄存器的宽度

```
for (int i = 0; i < N; i+= 4) {
      // in parallel
      A[i] = B[i] + C[i];
      A[i+0] = B[i+0] + C[i+0];
      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通道浮点/整数指令
  - Intel Pentium 4 onwards
  - AMD Athlon 64 onwards
- **AVX**
  - 8宽度浮点/整数指令
  - Intel Sandy Bridge
  - AMD Bulldozer

线程级并行 Thread-Level Parallelism

- **线程组成**
  - 指令流， Instruction streams
  - 私有寄存器： PC, registers, stack
  - 共享的全局变量、堆， Shared globals, heap
- **程序员可以创建和销毁**
- **程序员和操作系统都可以销毁**

同时多线程 - Simultaneous Multithreading

- 允许单个处理器核心在同一时间内执行多个线程
- 需要划分重排序缓冲区（ROB）和其他缓冲区

\+ 最小化硬件重复

\+ 为乱序执行（OoO）提供更多的调度自由度

– 缓存和执行资源的竞争可能会降低单线程性能

多核 Multicore

- 复制完整的流水线
- Coffee Lake: 8 cores

+ 完整的核，除了最后一级缓存，不共享其他资源
+ 很容易地可以继续保持Moore’s Law

– 多核程序和利用率问题

Locks，Coherence，and Consistency

- 问题：多线程读写同一块数据
- 解决办法：加锁

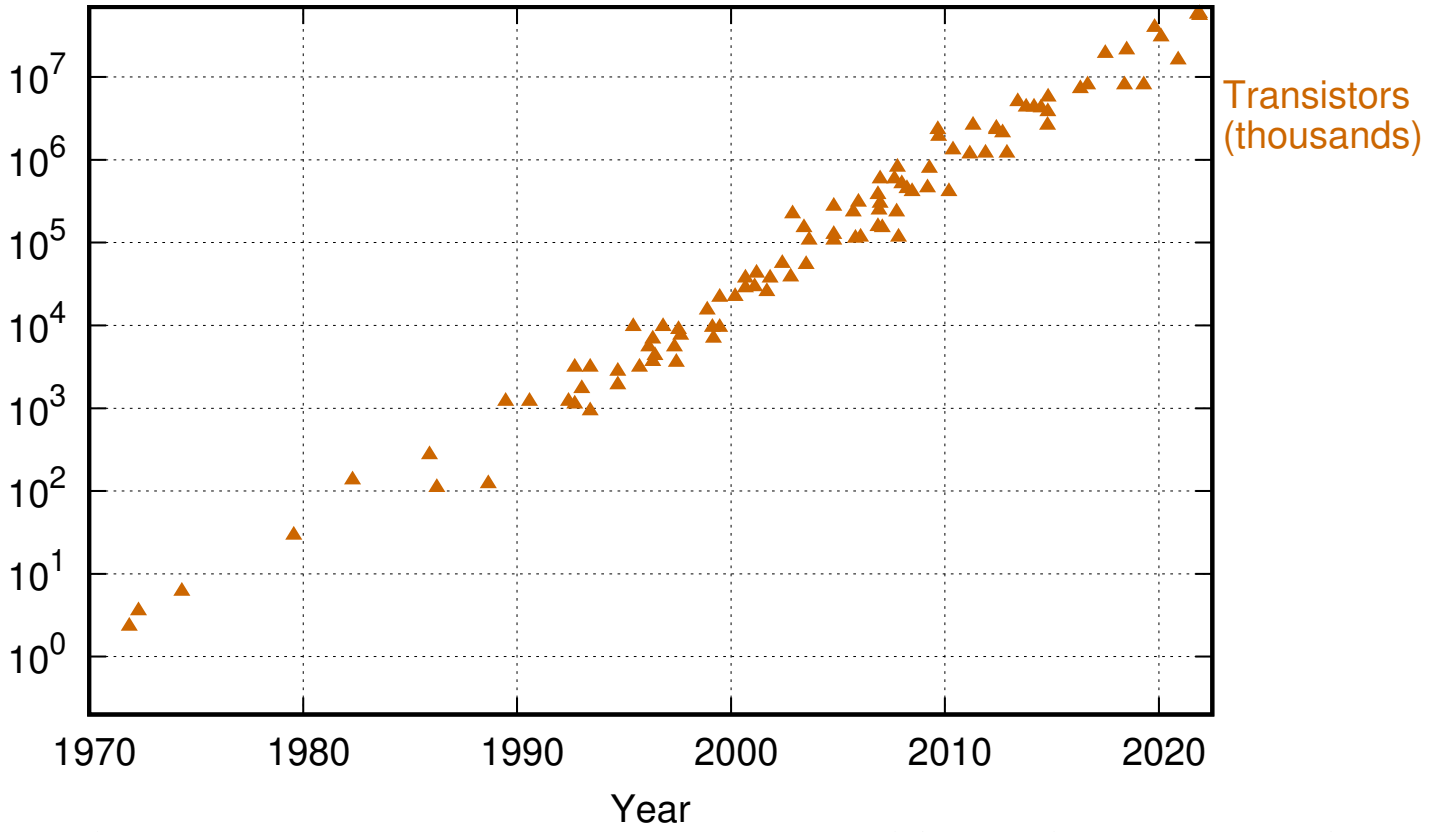
- 问题：谁的数据是正确的?
- 解决方法：缓存一致性协议Coherence

- 问题：什么样的数据是正确的？Consistency
- 解决方法：存储器同一性模型

CPU Conclusions

- **CPU为串行程序优化**
  - Pipelines, branch prediction, superscalar, OoO
  - 通过提高时钟速度和利用率来减少执行时间
- **缓慢的内存带宽将会是大问题**
- **并行处理是未来的方向**
  - Sandy Bridge-E great for 6-12 active threads
  - How about 32K threads?

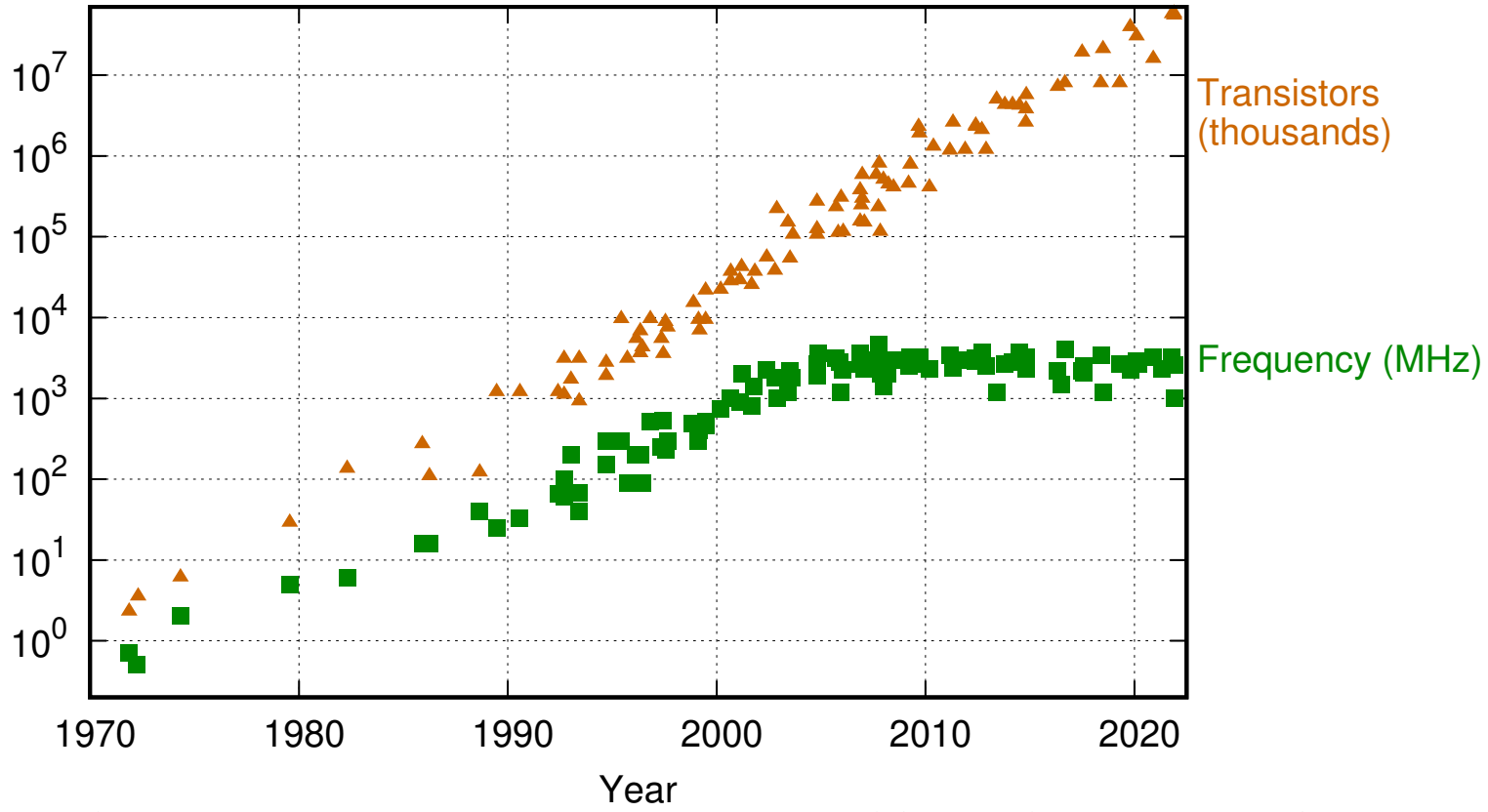
50 Years of Microprocessor Trend



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

“摩尔定律“预测微芯片上的**晶体管**数量每两年翻一番。

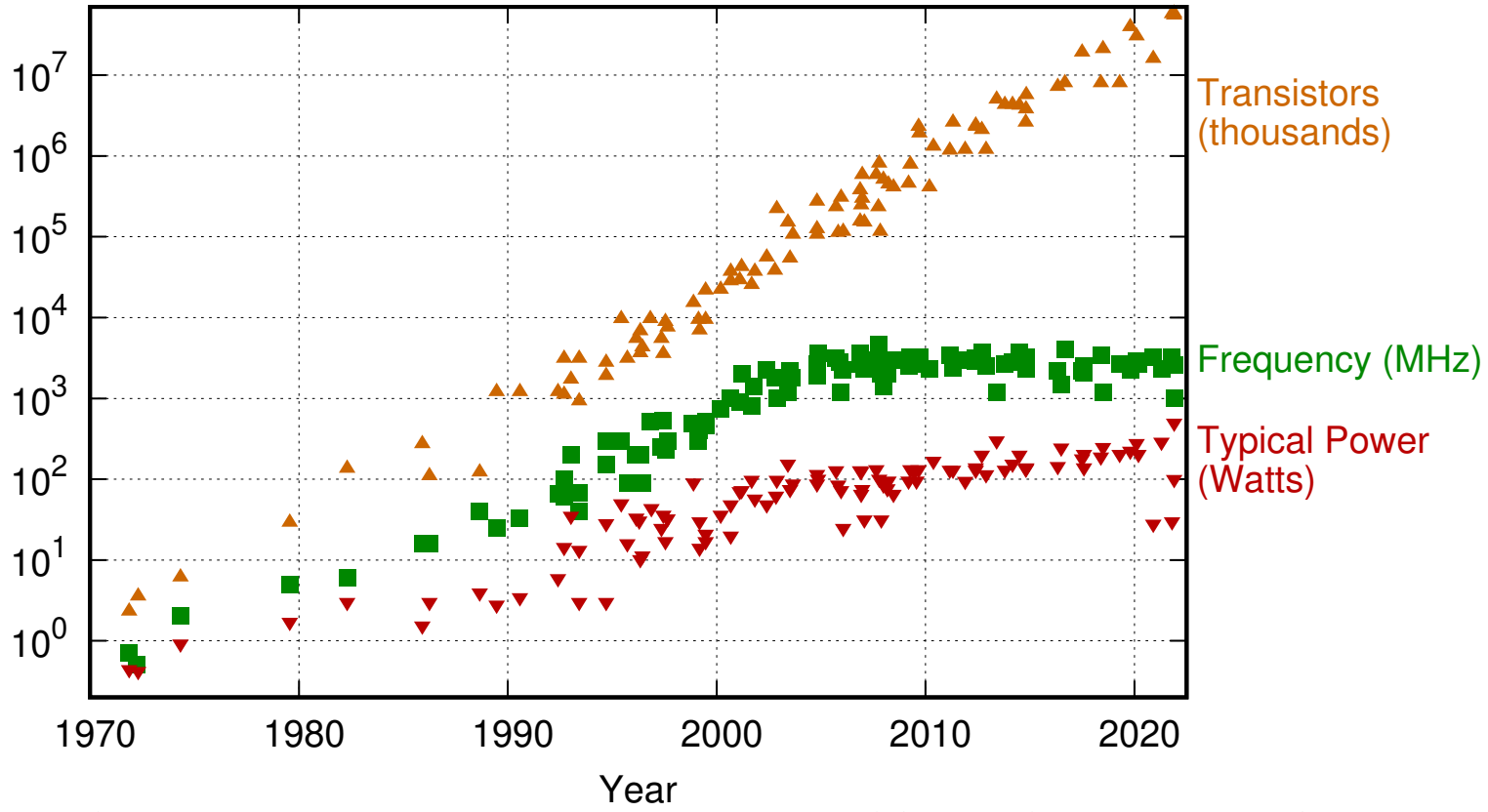
50 Years of Microprocessor Trend



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

处理器的**频率**保持了相同的趋势，因为更小的晶体管可以更快地切换……直到大约2005年。

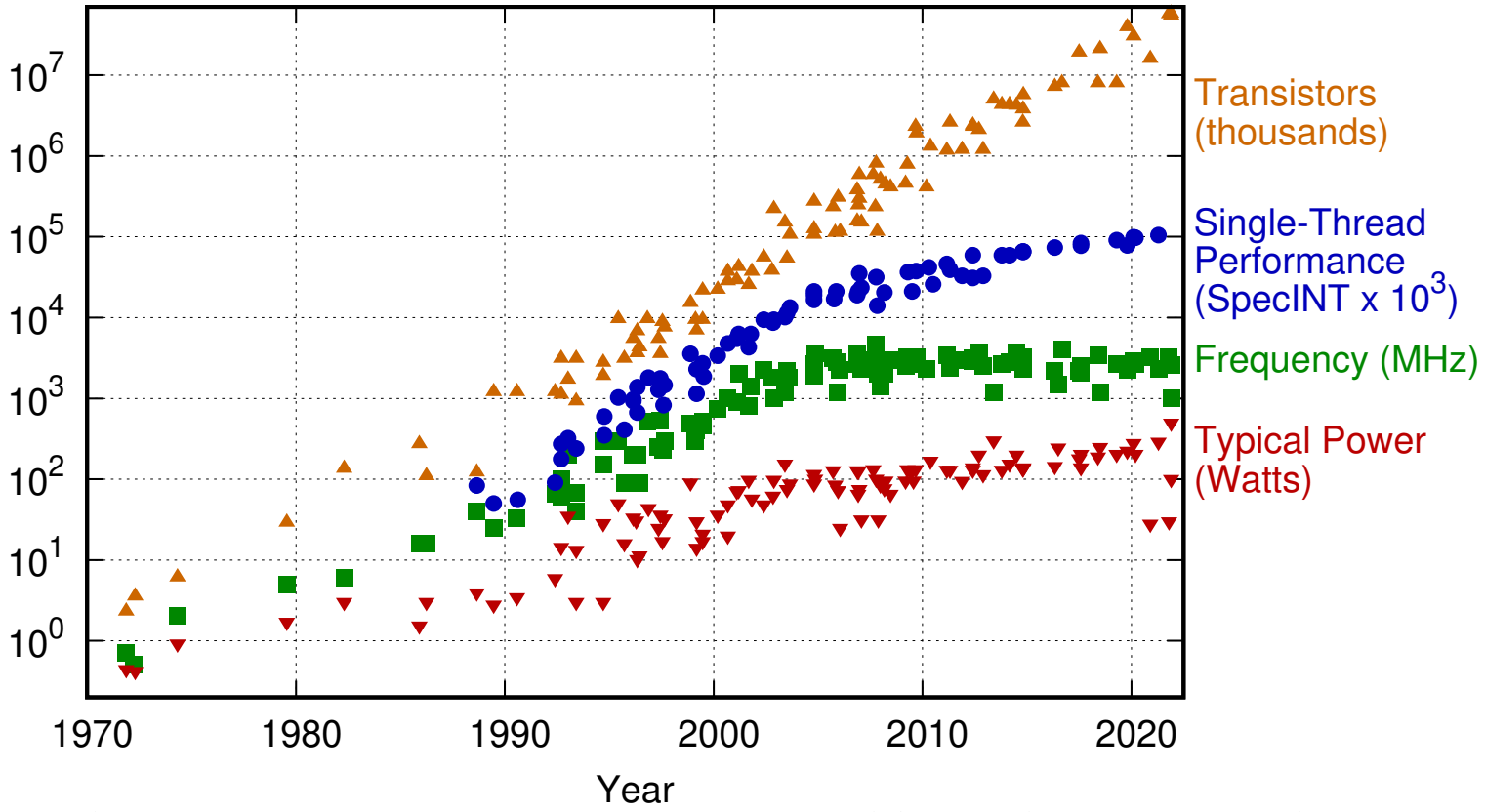
50 Years of Microprocessor Trend



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

从2005年开始，由于**功率墙**的限制，频率停止了增长。

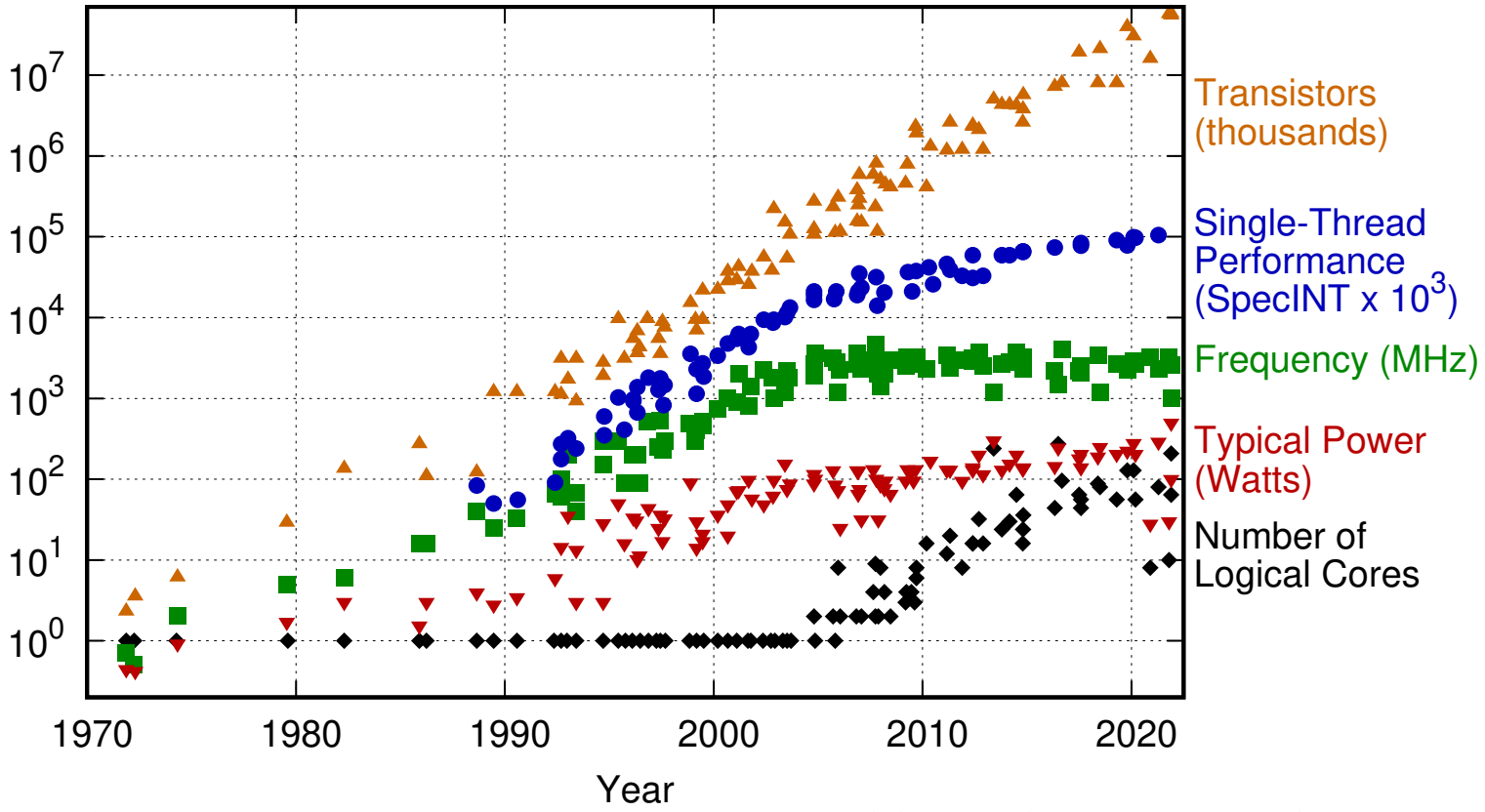
50 Years of Microprocessor Trend



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

频率的停滞导致了程序**单线程性能**的停滞，仅依靠架构和编译器进展带来改进。

50 Years of Microprocessor Trend



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

单线程性能的停滞使得并行计算成为主流，晶体管被用来在处理器中增加**更多的核心。**

Approaches to Processor Design

Latency-Oriented Design

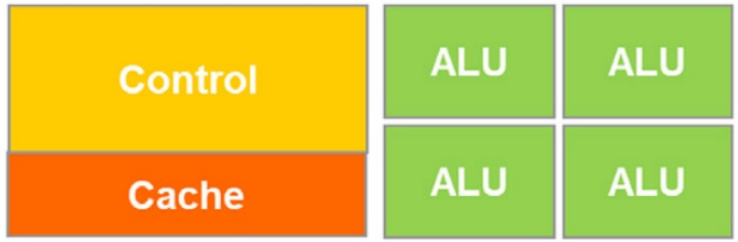
- 最小化单个任务的执行时间

Throughput-Oriented Design

- 最大化一定时间内执行的任务量

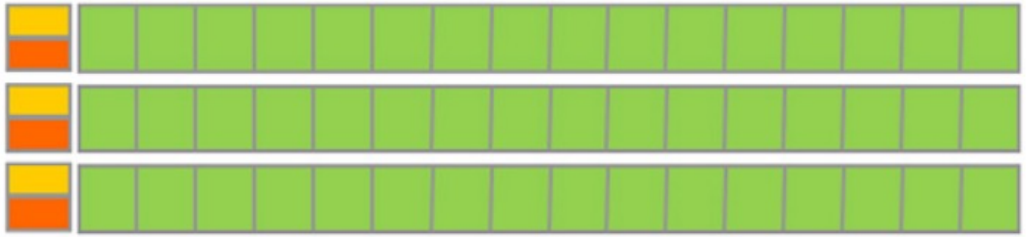
Approaches to Processor Design

Latency-Oriented Design



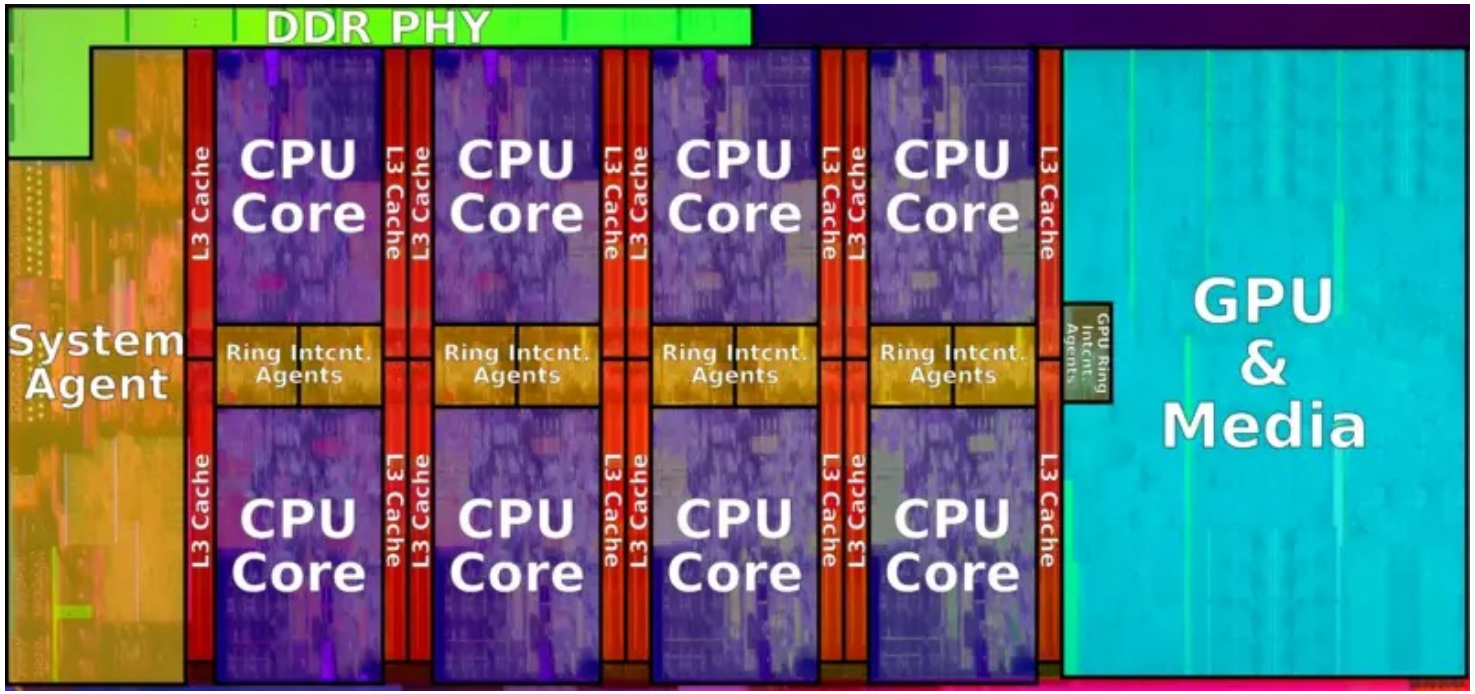
- A few powerful ALUs
  - 降低操作延迟
- Large caches
  - 将长延迟的内存访问转换为短延迟的缓存访问
- Sophisticated control
  - 使用分支预测减少控制冒险
  - 使用数据forwarding减少数据冒险
  - 乱序执行
- Modest multithreading to hide short latency
  - 同时多线程SMT，超线程HT
- High clock frequency

Throughput-Oriented Design



- Many small ALUs
  - 延迟较长，但高吞吐量
  - 高度流水化，进一步提高吞吐量
- Small caches
  - 使用片上内存，节省功耗
  - 更多的用于计算
- Simple control
  - 更多的面积用于计算
- Massive number of threads to hide the very high latency!
- Moderate clock frequency

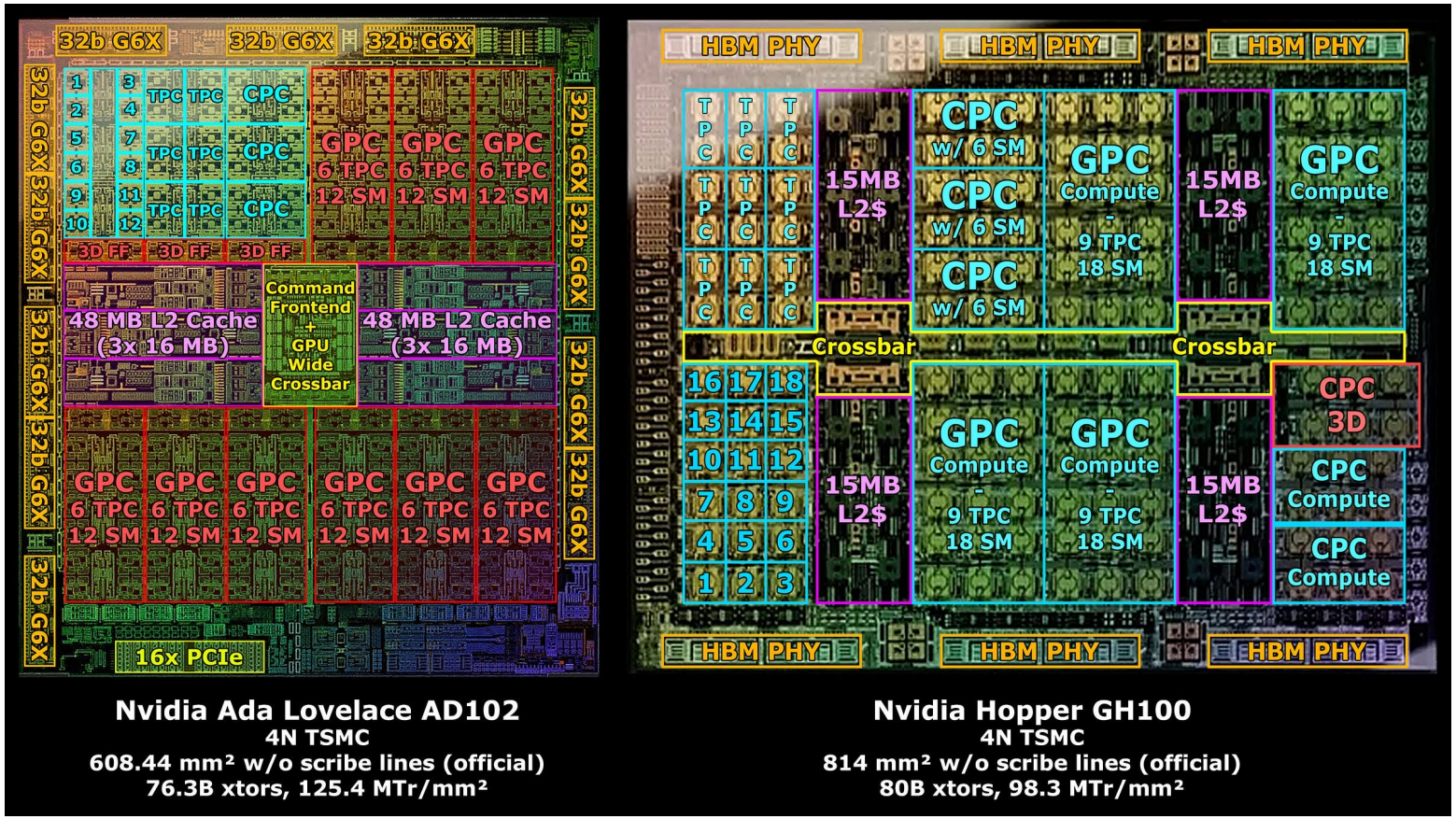
CPU Die Shot



- Intel Coffee-Lake
  - i9 9900 Series
  - 14 nm++ process
  - 11 metal layers
  - ~174 mm² die size
  - 8 CPU cores + 24 GPU EUs

https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake#Die

GPU Die Shot

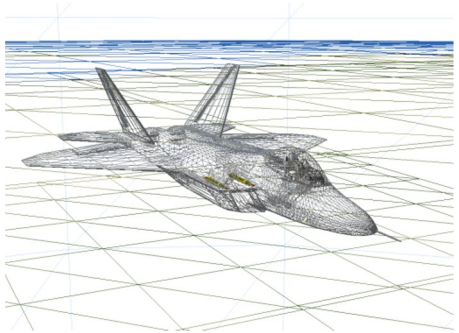


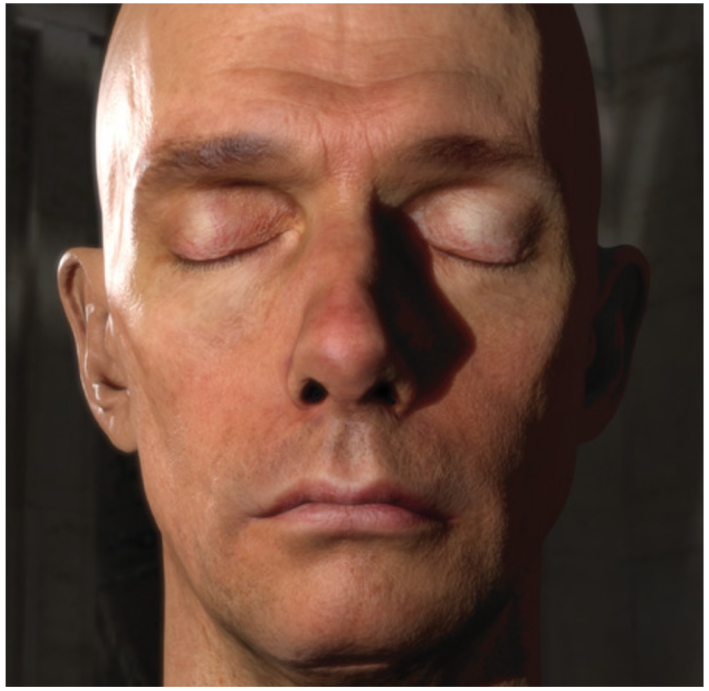
- NV AD102
  - 4N TSMC
  - 608 mm^2
- NV GH100
  - 4N TSMC
  - 814 mm^2

https://locuza.substack.com/p/nvidias-ad102-officially-revealed

Graphics Workloads

- Triangles/vertices and pixels/fragments





http://http.developer.nvidia.com/GPUGems3/gpugems3_ch14.html

Why GPUs?

- 图形学工作负载（Graphics workloads）具有高度的并行性
  - 数据并行
  - 流水线并行
- CPU 和 GPU 可以并行执行
- 硬件: 纹理过滤、光栅化、等

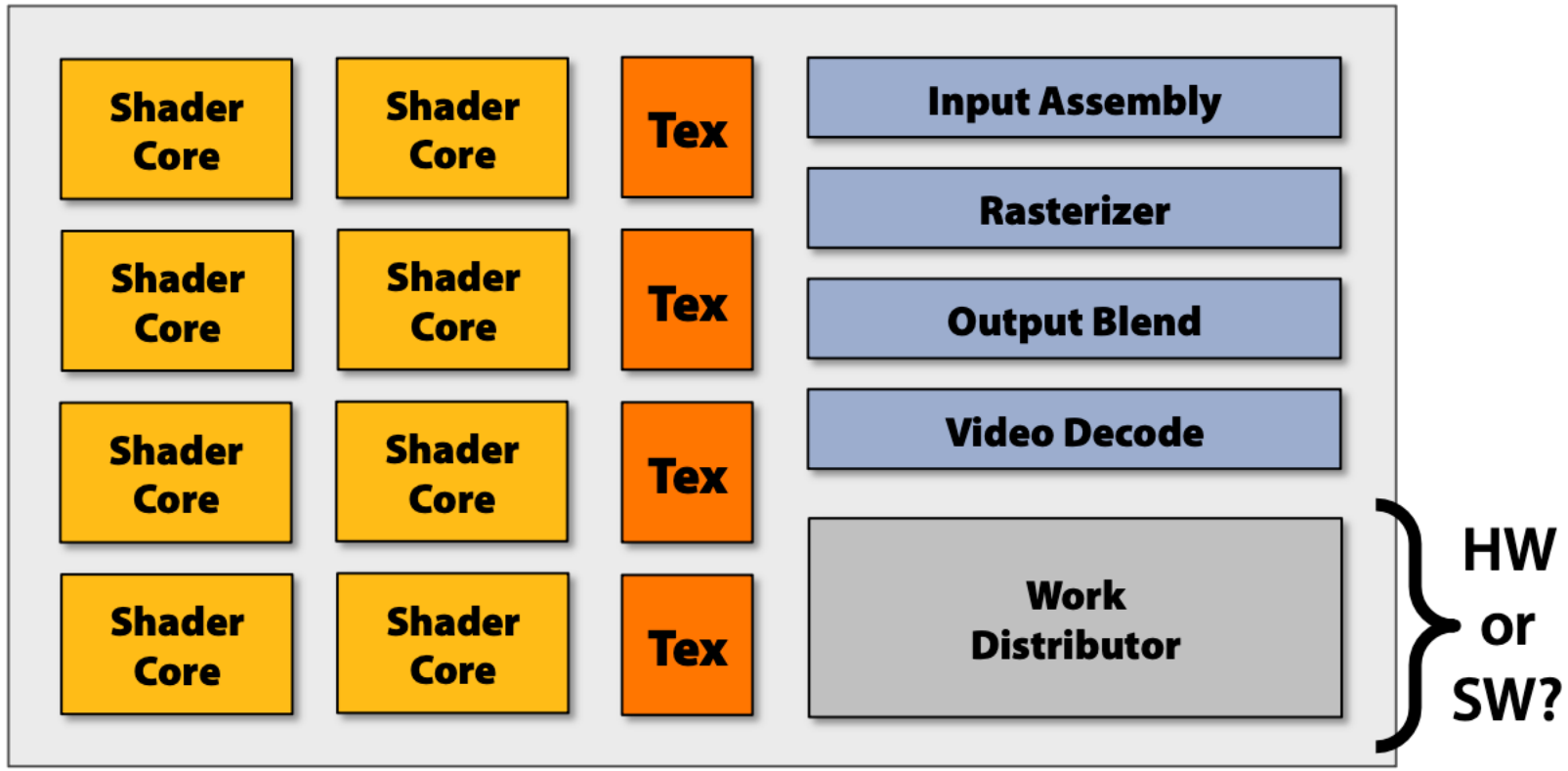
Part 1: throughput processing

- Three key concepts behind how modern GPU processing cores run code

- Knowing these concepts will help you :
  1. Understand space of GPU core (and throughput CPU processing core) designs
  2. Optimize shaders/compute kernels
  3. Establish intuition: what workloads might benefit from the design of these architectures?

What’s in a GPU?

- GPU (Graphic Processing Unit)
  - An Heterogeneous chip multi-processor (highly tuned for graphics)

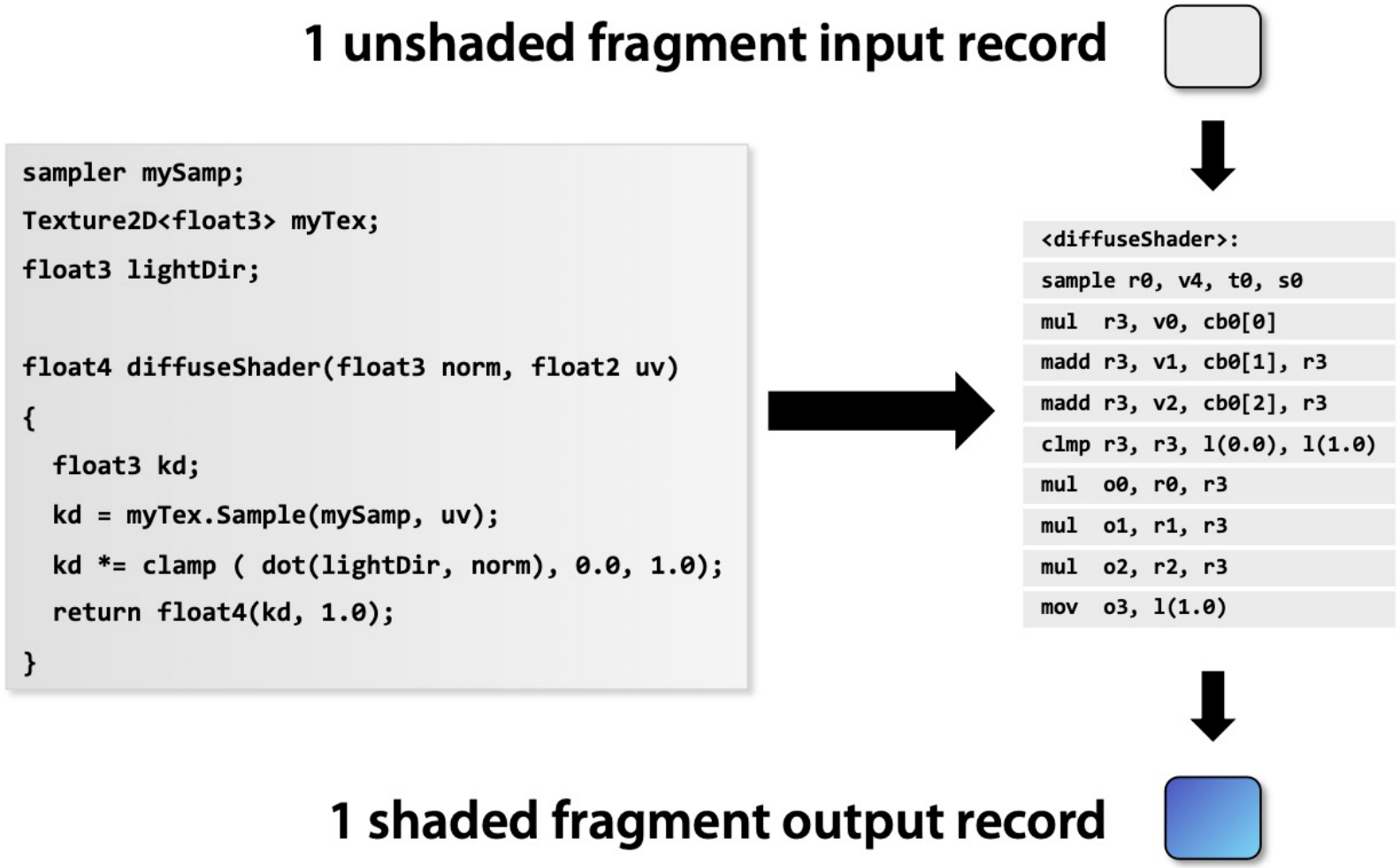


A diffuse reflectance shader

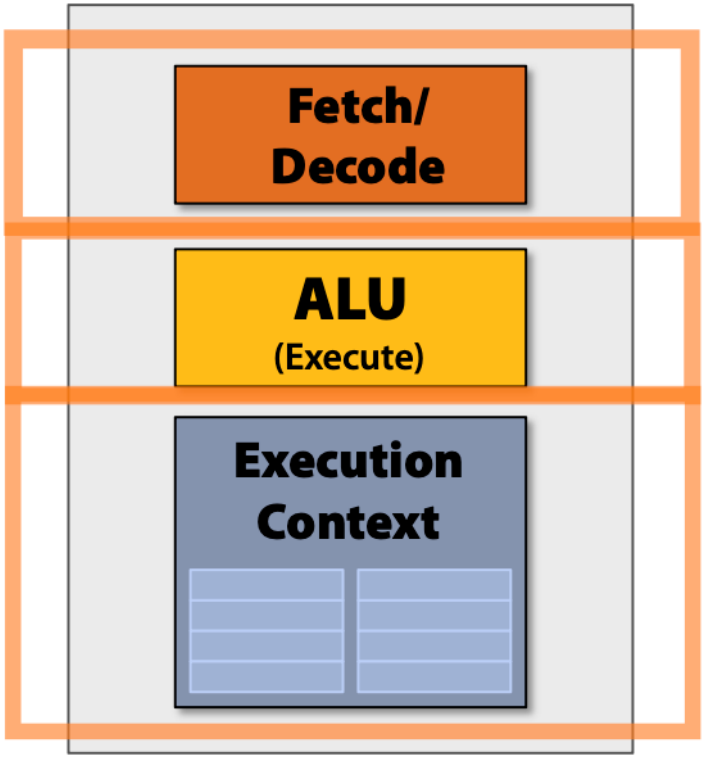
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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);
}
```

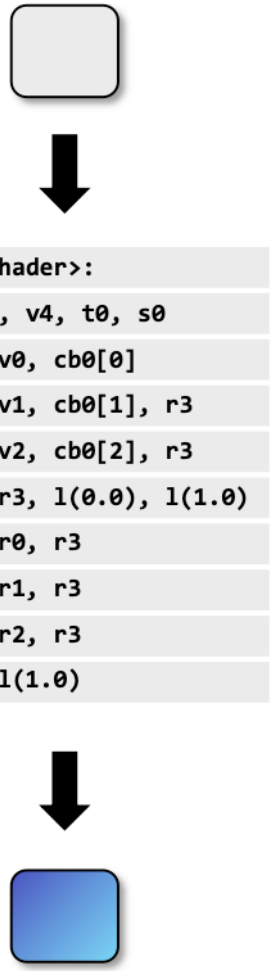
Independent, but no explicit parallelism

Compile shader



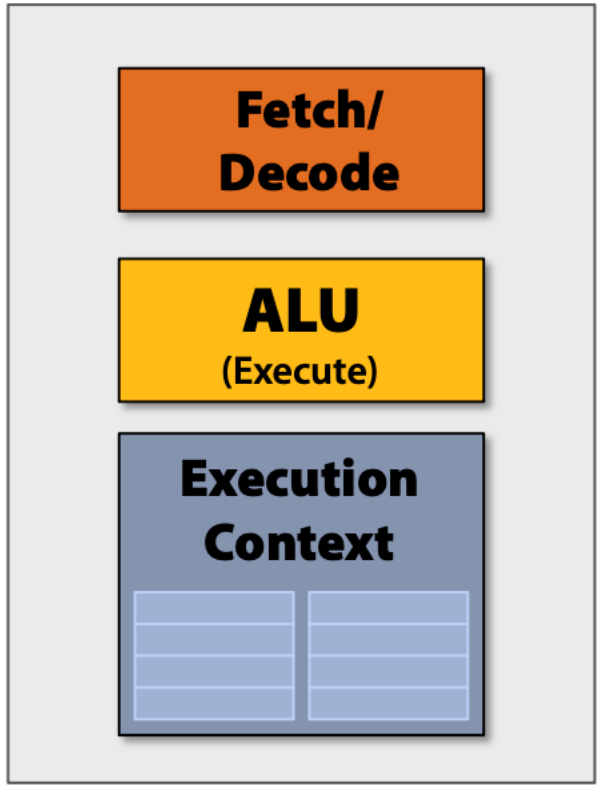
Execute Shader

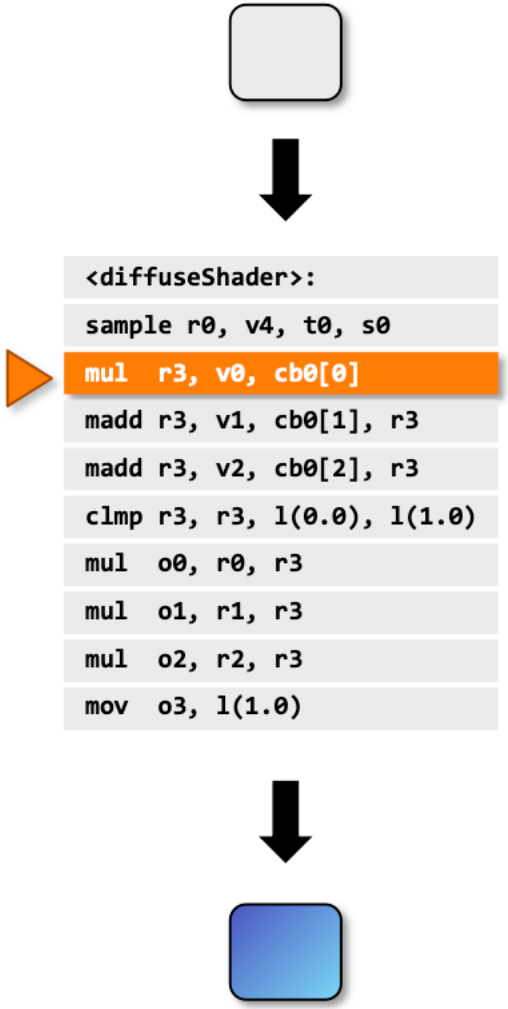




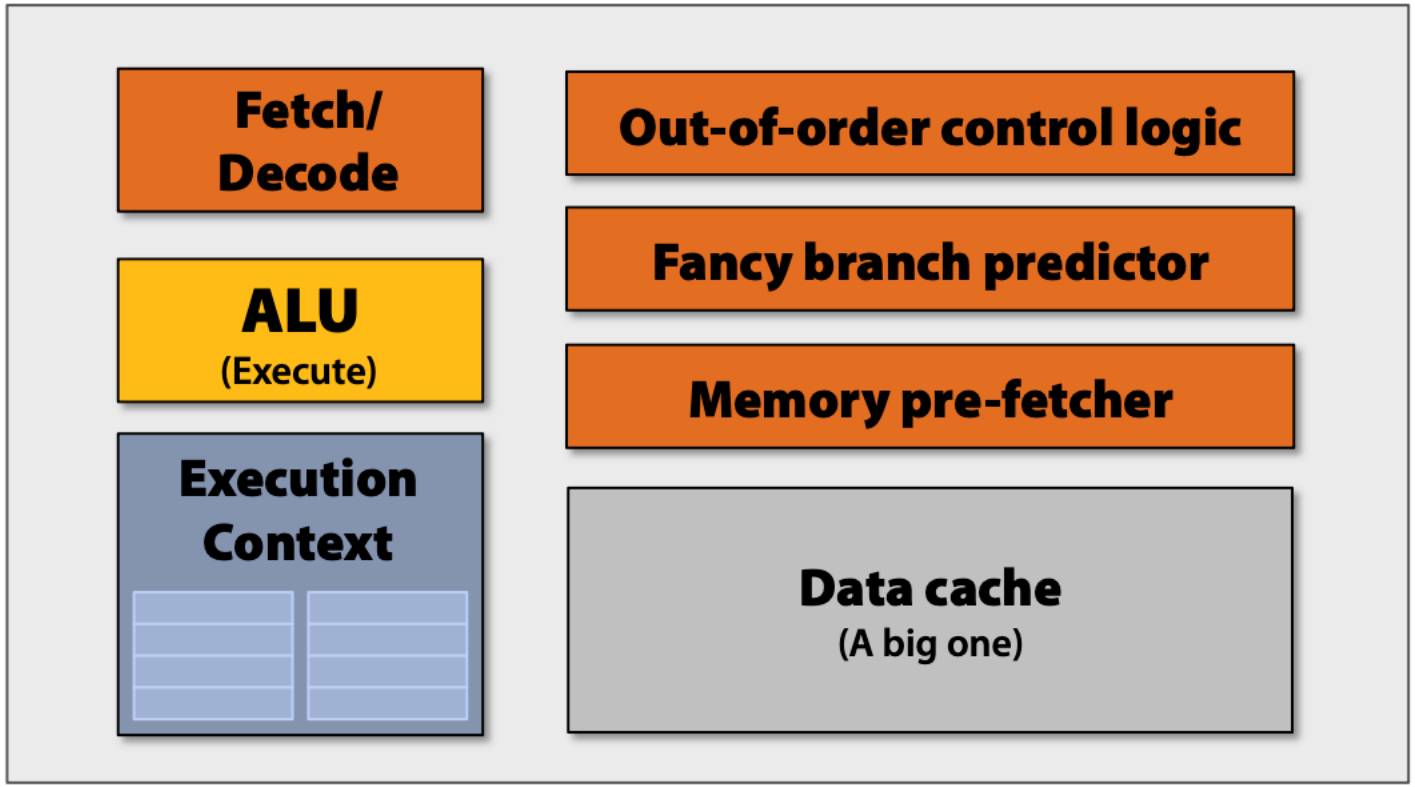
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Execute Shader

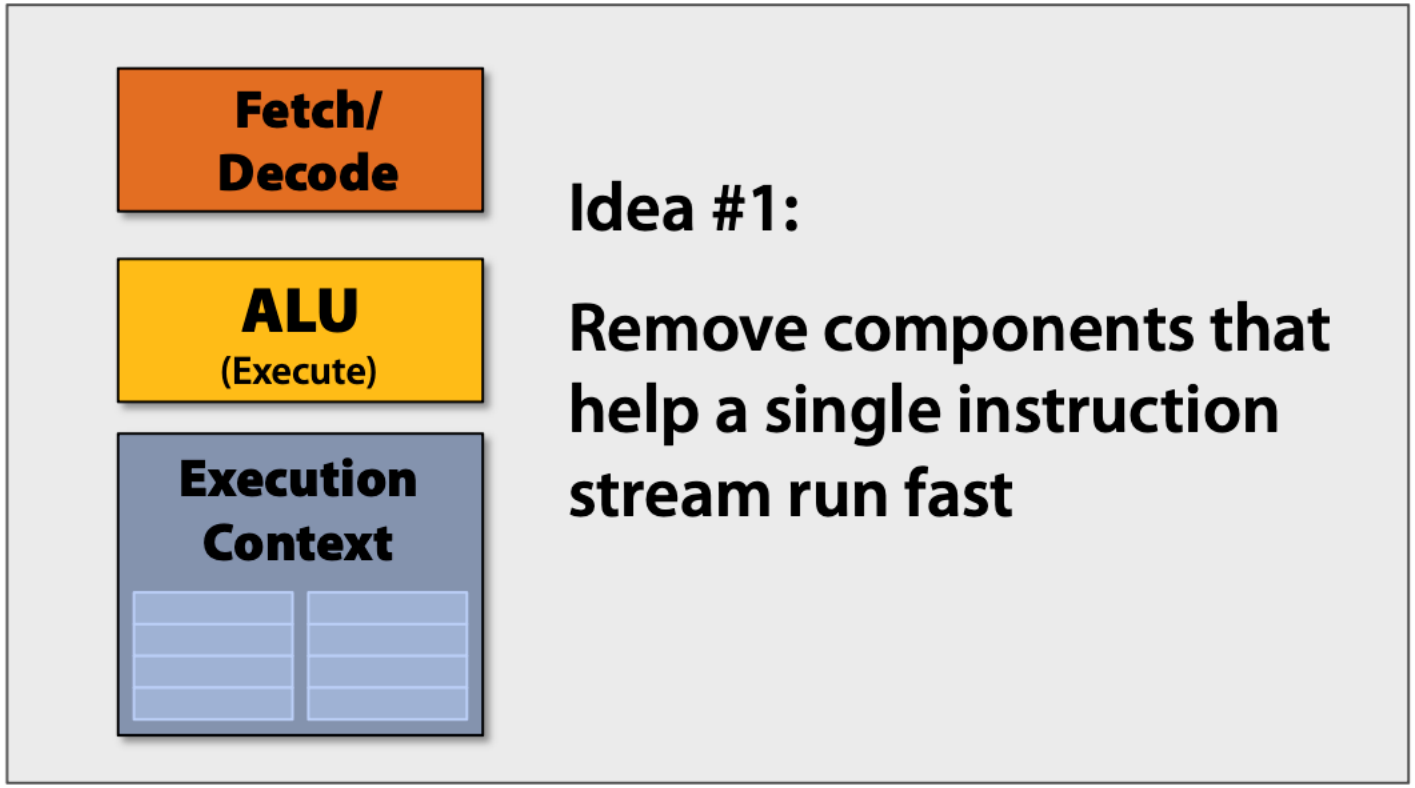




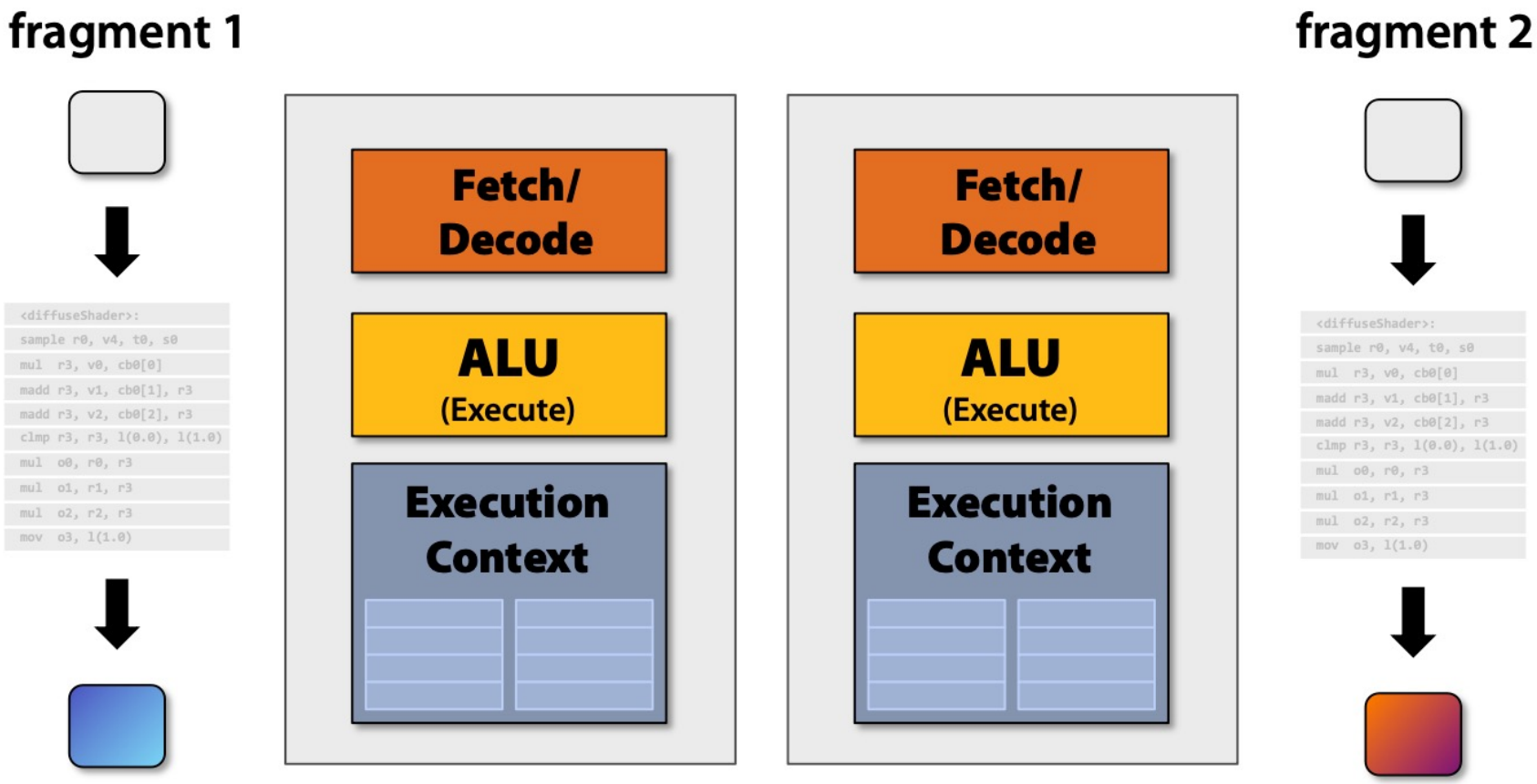
CPU- “style” cores



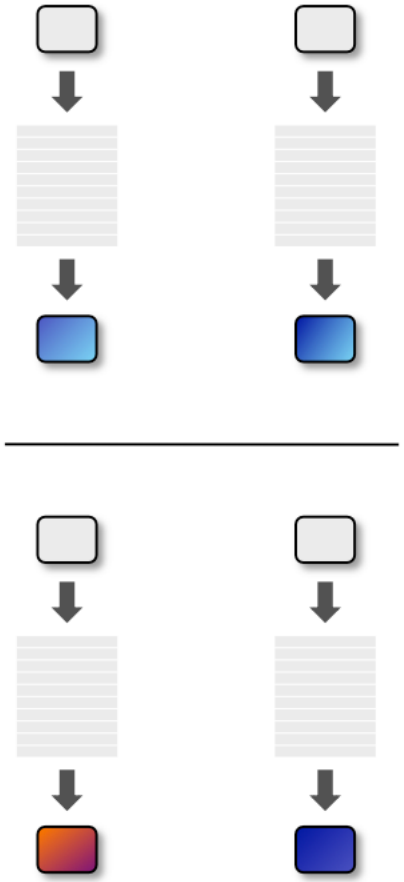
Slimming down “精简”

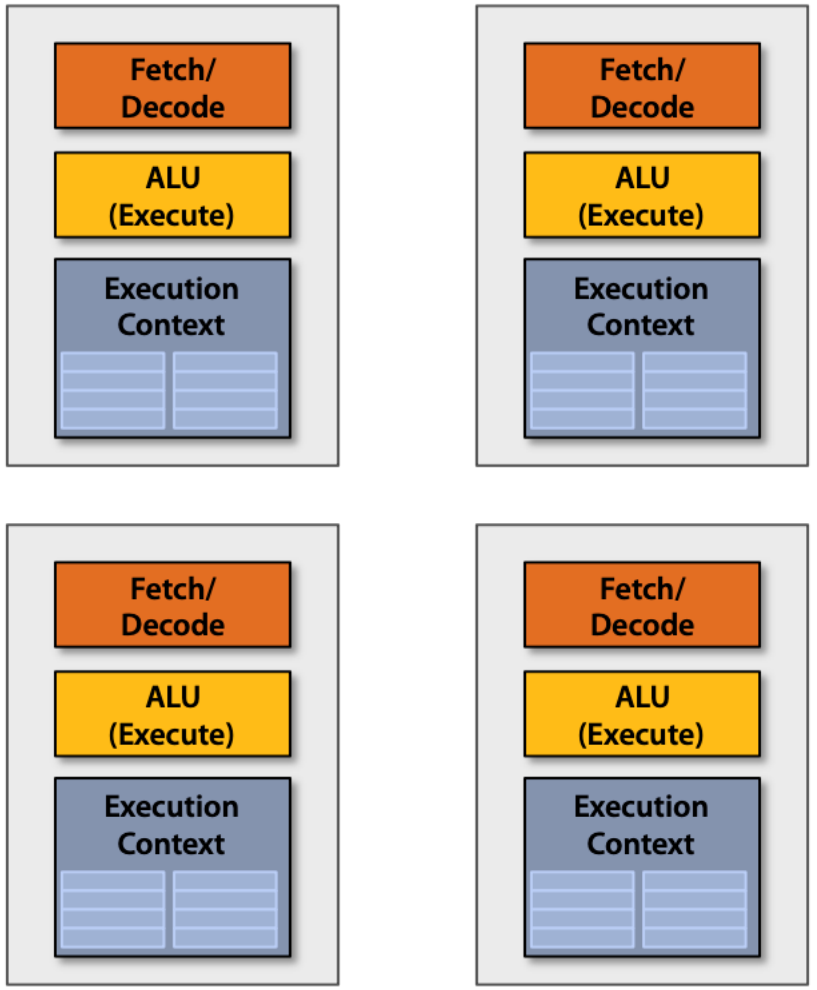


Two cores (two fragments in parallel)



Four cores (four fragments in parallel)

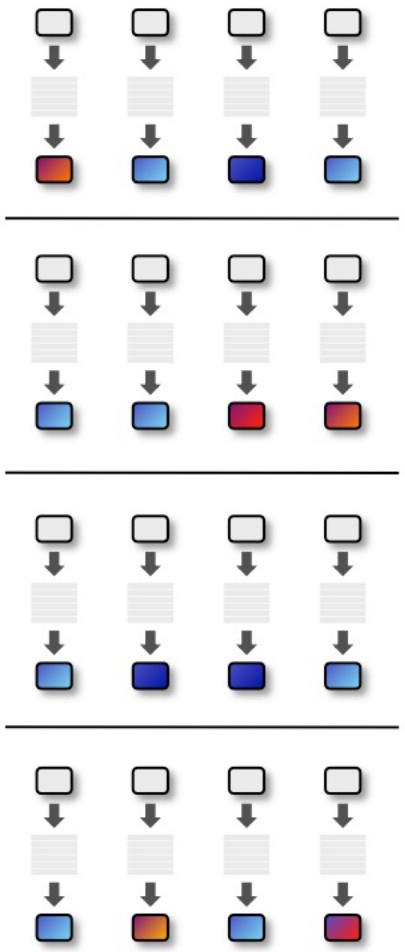




Sixteen cores (sixteen fragments in parallel)

16 cores = 16 simultaneous instruction streams

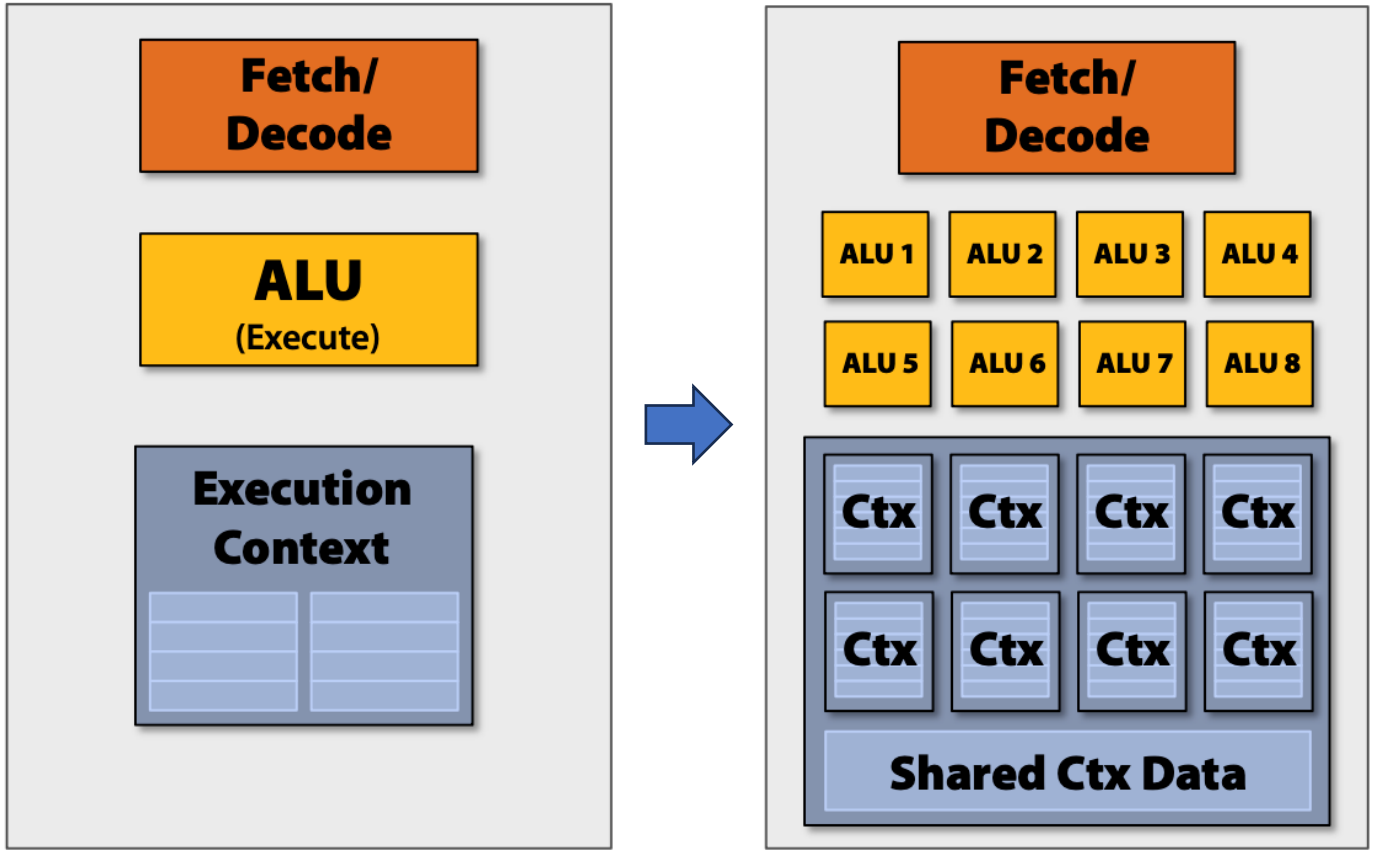
Instruction stream sharing



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

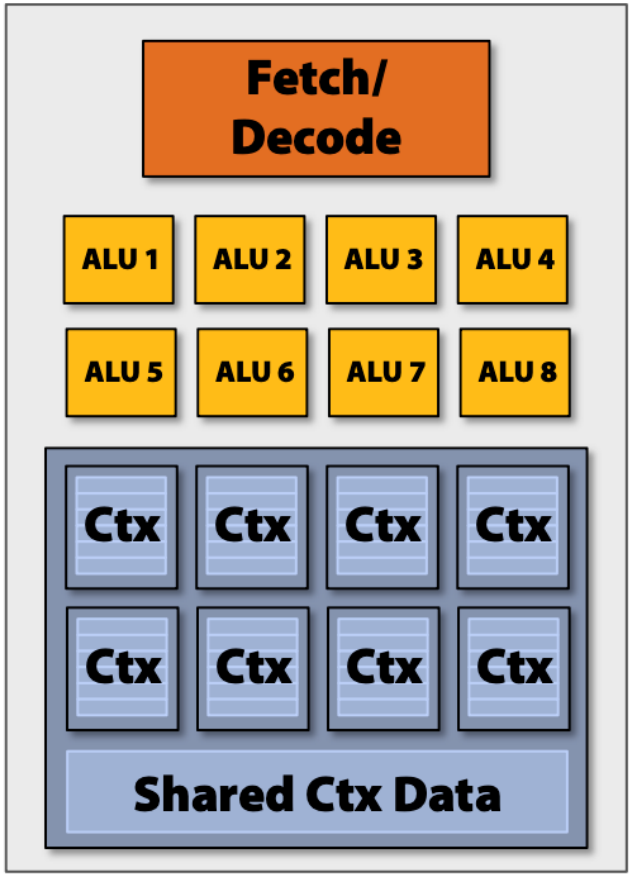


Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Modifying the shader

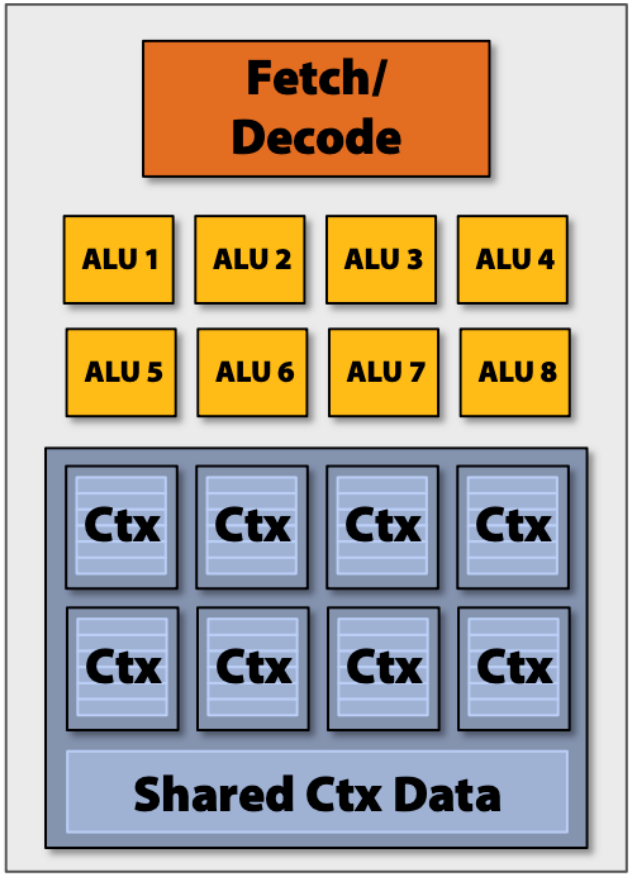


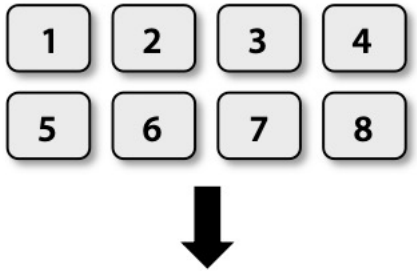

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
```

New compiled shader:

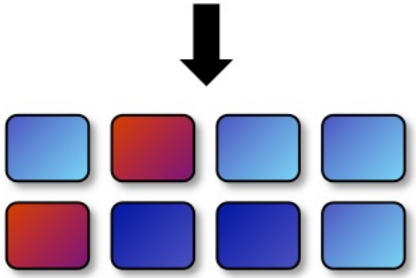
Processes 8 fragments using vector ops on vector registers

Modifying the shader

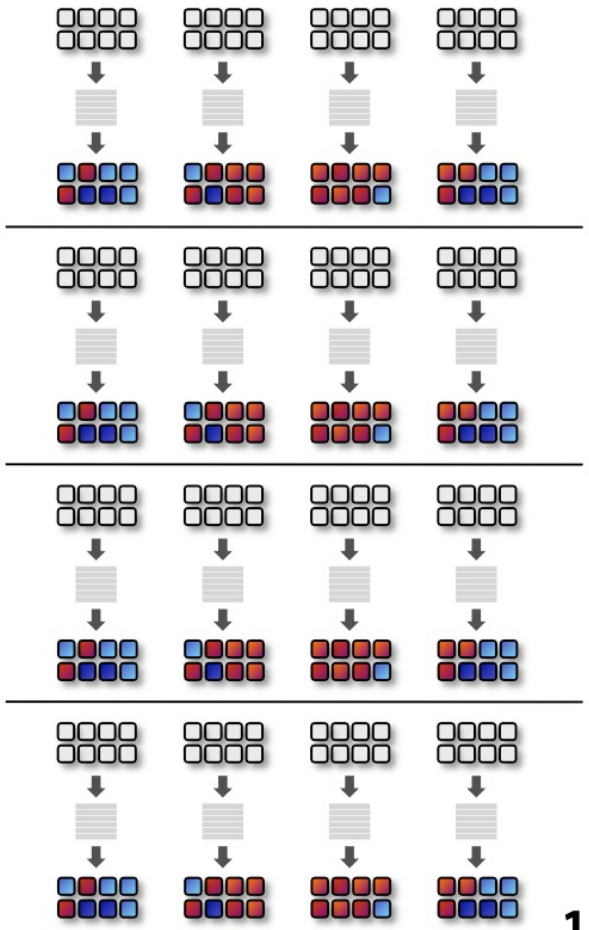


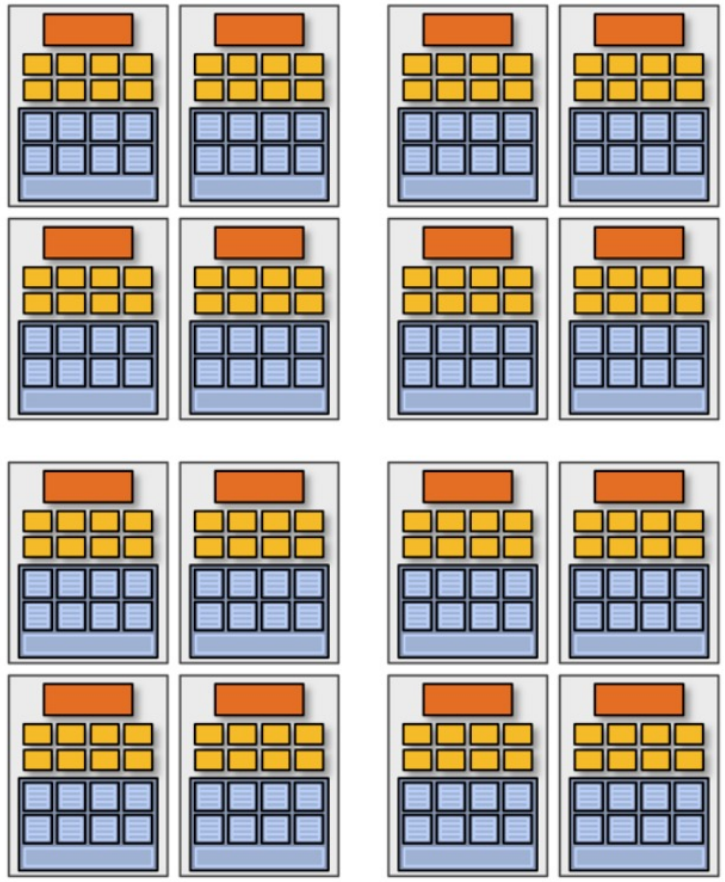



```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
```



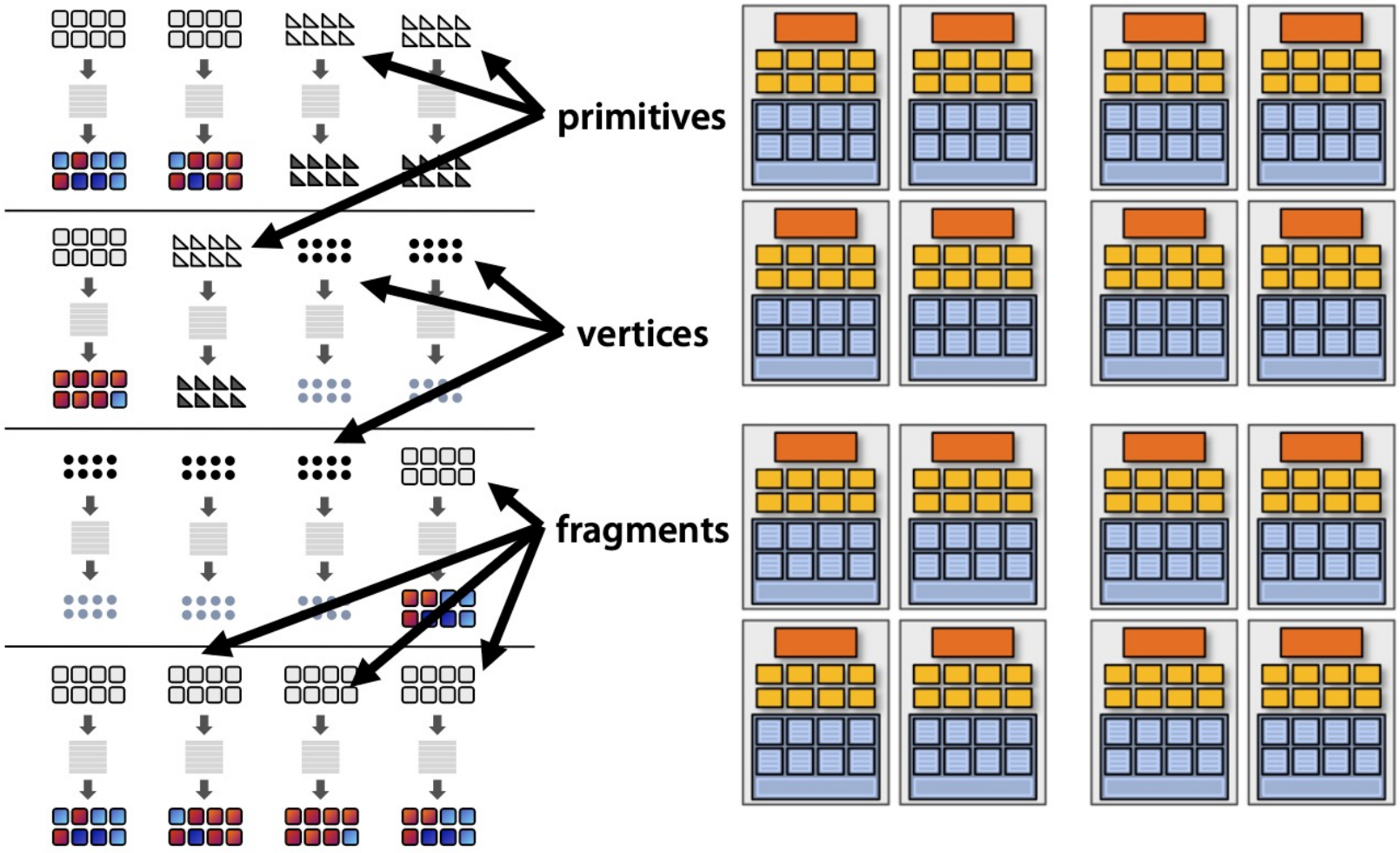
128 fragments in parallel



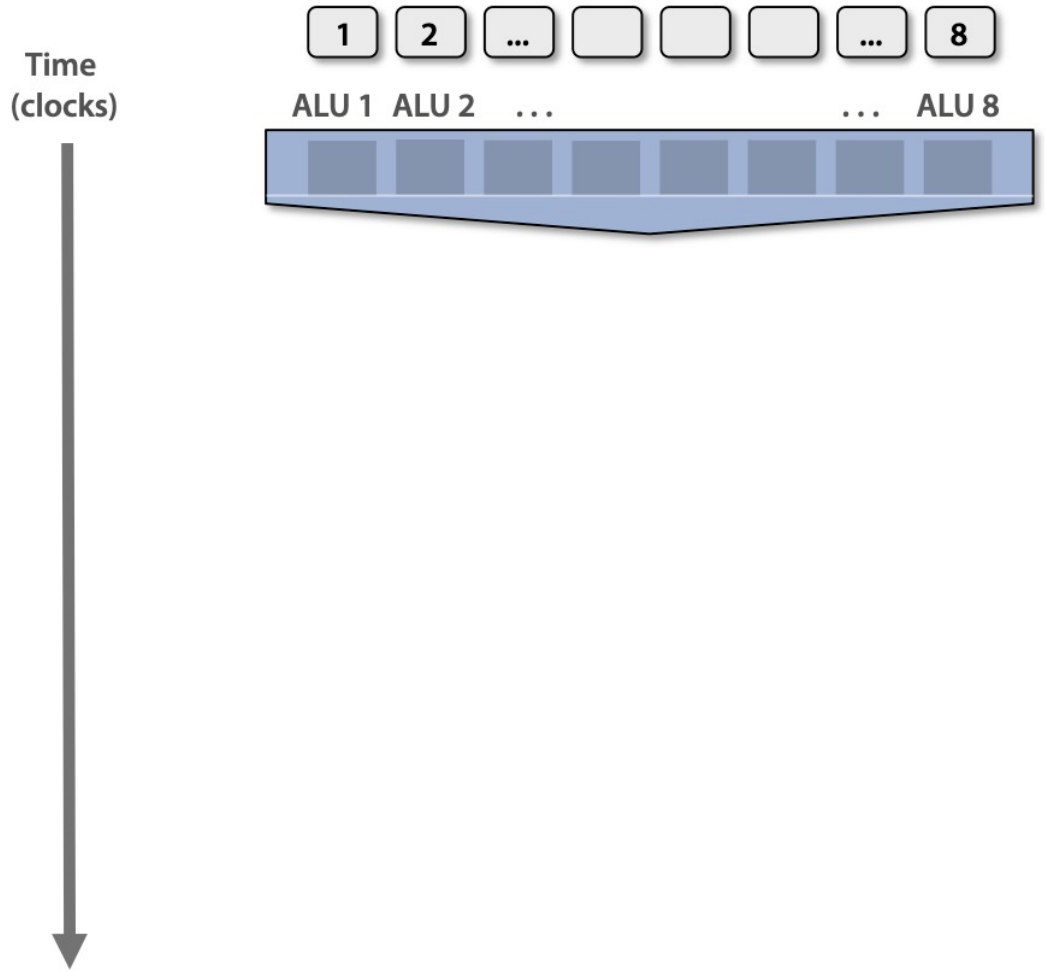


16 cores = 128 ALUs
= 16 simultaneous instruction streams

128 fragments in parallel

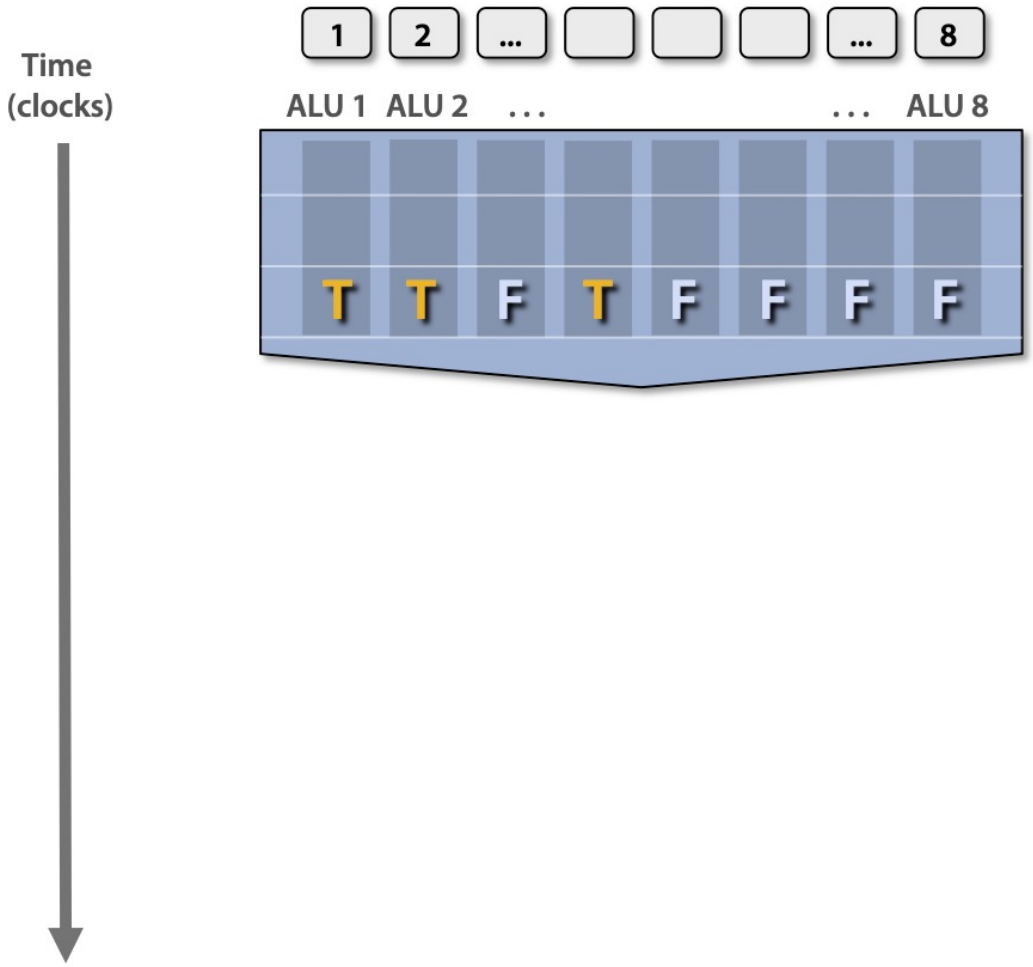


But what about branches?



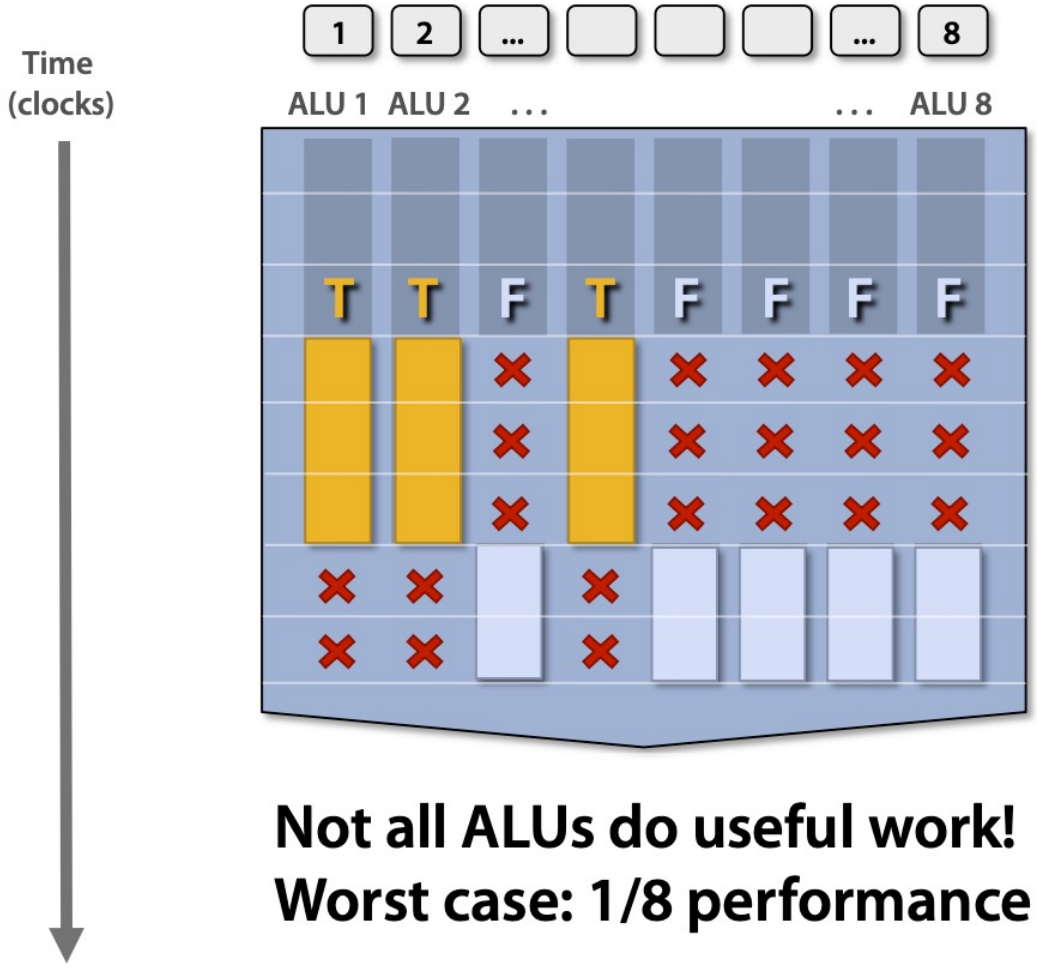

```
<unconditional
shader code>

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?



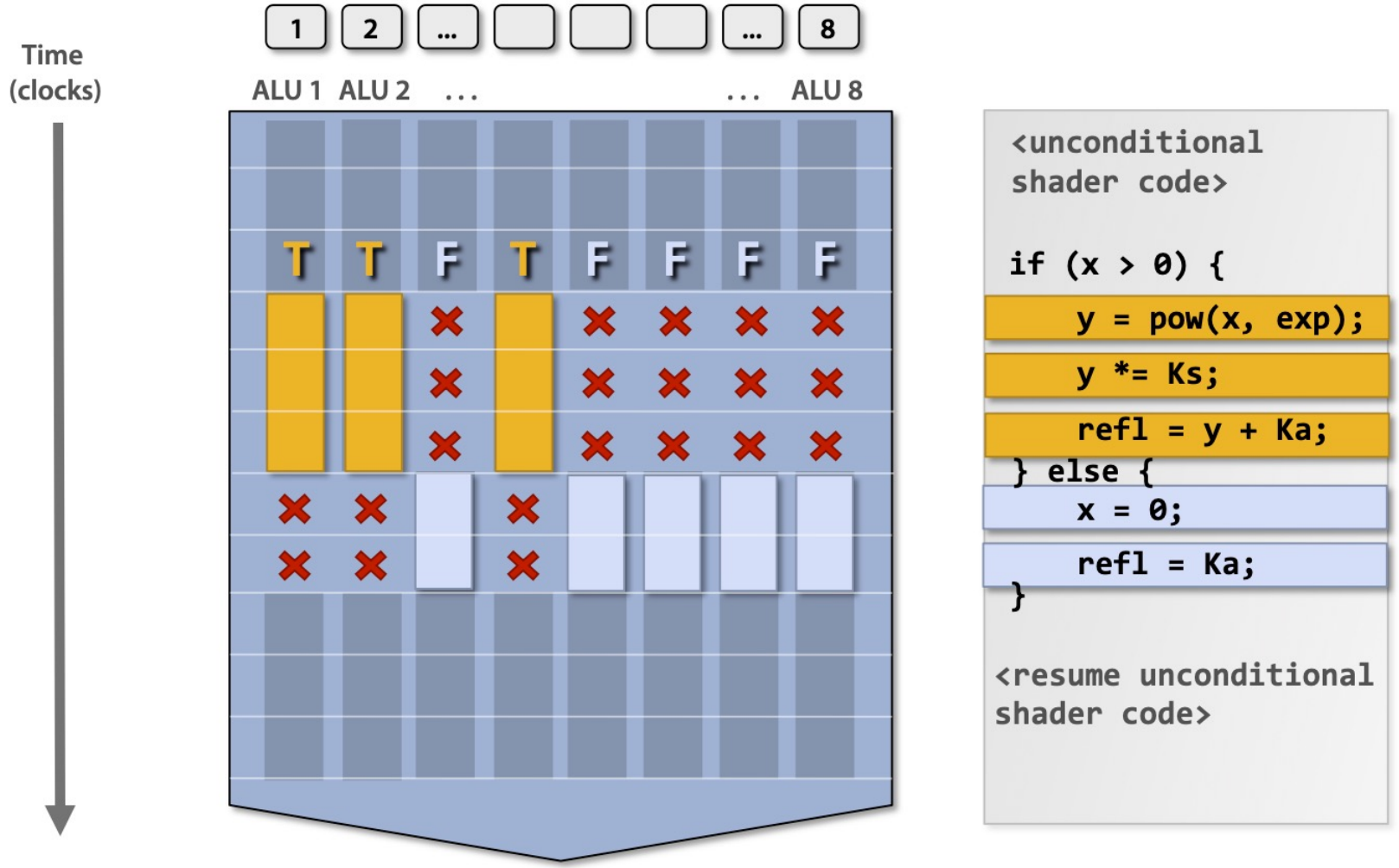

```
<unconditional
shader code>

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?



Clarification

- SIMD处理并不总是需要显式的SIMD指令
  - 选项 1: 显式的向量运算指令
    - Intel/AMD x86 SSE、AVX、AVX2
  - 选项 2: 标量指令、硬件隐式向量化
    - 硬件进行ALUs指令流共享 (amount of sharing hidden from software)
    - NVIDIA GeForce (“SIMT” warps), AMD Radeon architectures

In practice: 16 to 64 fragments share an instruction stream