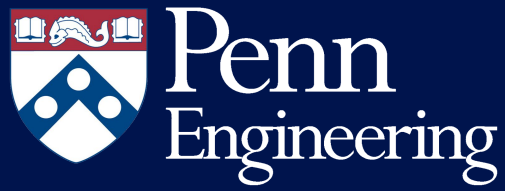
GPU Architecture Overview

Shehzan Mohammed

CIS 5650 - Fall 2024



Agenda

- Performance Terminologies
- How CPU is designed
- How GPU is designed
- GPU Architecture in the AI Era

Acknowledgements

- CPU slides
  - Varun Sampath, NVIDIA. Penn alumnus
- GPU slides
  - Kayvon Fatahalian, CMU
  - Mike Houston, NVIDIA

Performance Terminologies

- **Instruction**
  - Tells machine what to do at a given step
  - e.g. MUL, ADD etc.
- **Operation**
  - What a compute unit does to fulfill the instruction
  - e.g. FLT32 operation, INT operation etc.

Q: For a warp, how many operations are in one instruction?

Performance Terminologies

- **Instruction**
  - Tells machine what to do at a given step
  - e.g. MUL, ADD etc.
- **Operation**
  - What a compute unit does to fulfill the instruction
  - e.g. FLT32 operation, INT operation etc.

For a warp size of 32, 1 instruction = 32 operations

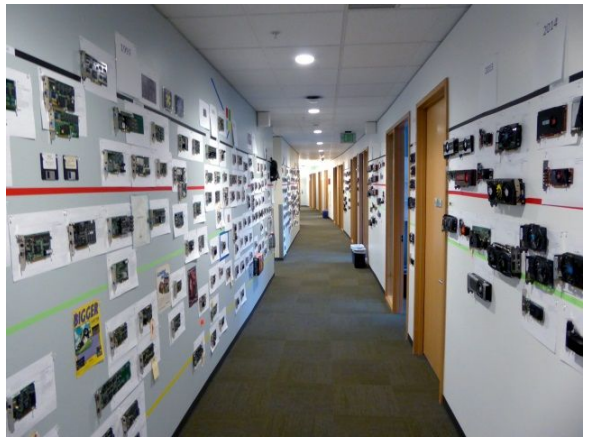
Performance Terminologies

- **Clock Cycle**
  - Basic unit that measures instruction time
- **Clock Speed**
  - Number of clock cycles (executed) per second
  - E.g. 1GHz = 1 billion cycles per second

Performance Terminologies

- **Latency** (memory or instruction)
  - Delay in processing data after the original instruction
    - e.g. 200 cycles for global memory read
- **Throughput** (memory or instruction)
  - Number of items/instructions processed within a time interval
    - e.g. Instructions Per Cycle (IPC), or GB/s etc.
- **Bandwidth** (memory)
  - The rate at which data can be transferred
    - e.g. 736 GB/s for a RTX 4080

Direct3D Wall of GPU History

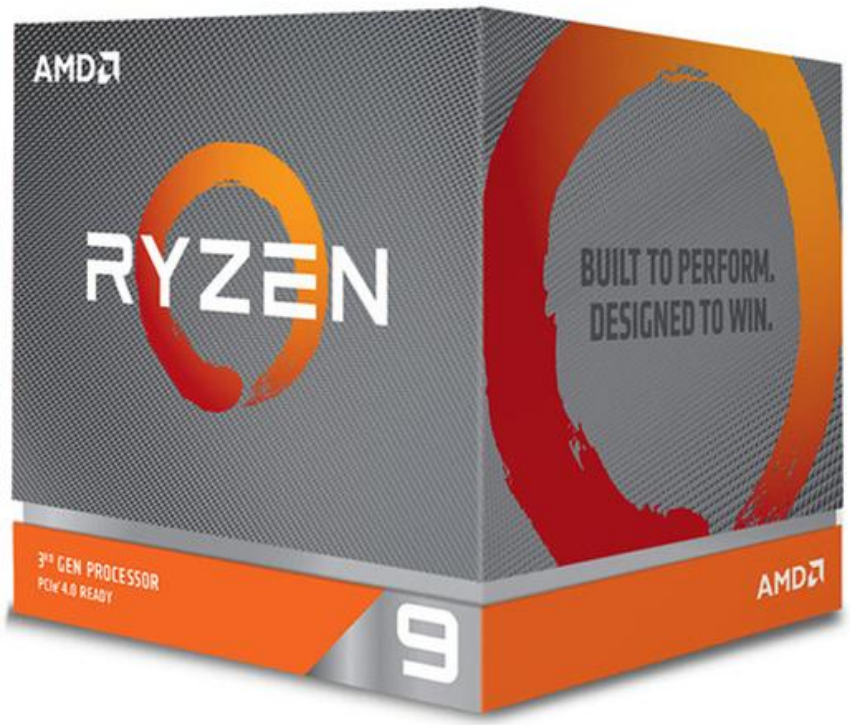


Penn Engineering

https://devblogs.microsoft.com/directx/wall-of-gpu-history/

CPU and GPU Trends

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



AMD RYZEN 9 3900X: 1.55 TFLOPS
Intel i9 9900K: 1.22 TFLOPS

NVIDIA Titan RTX GPU - 16.32 TFLOPS

7.5 TFLOPS in your car!

- NVIDIA DRIVE PX 2
  - Up to
    - 2 Tegra SoCs (2.5 TFLOPS)
    - 2 Pascal GPUs (5 TFLOPS)

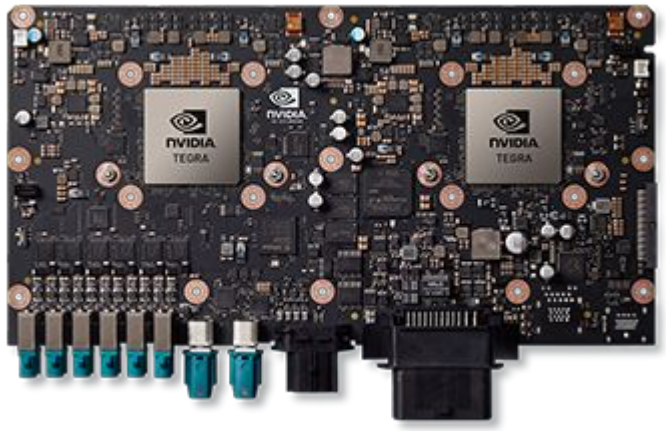
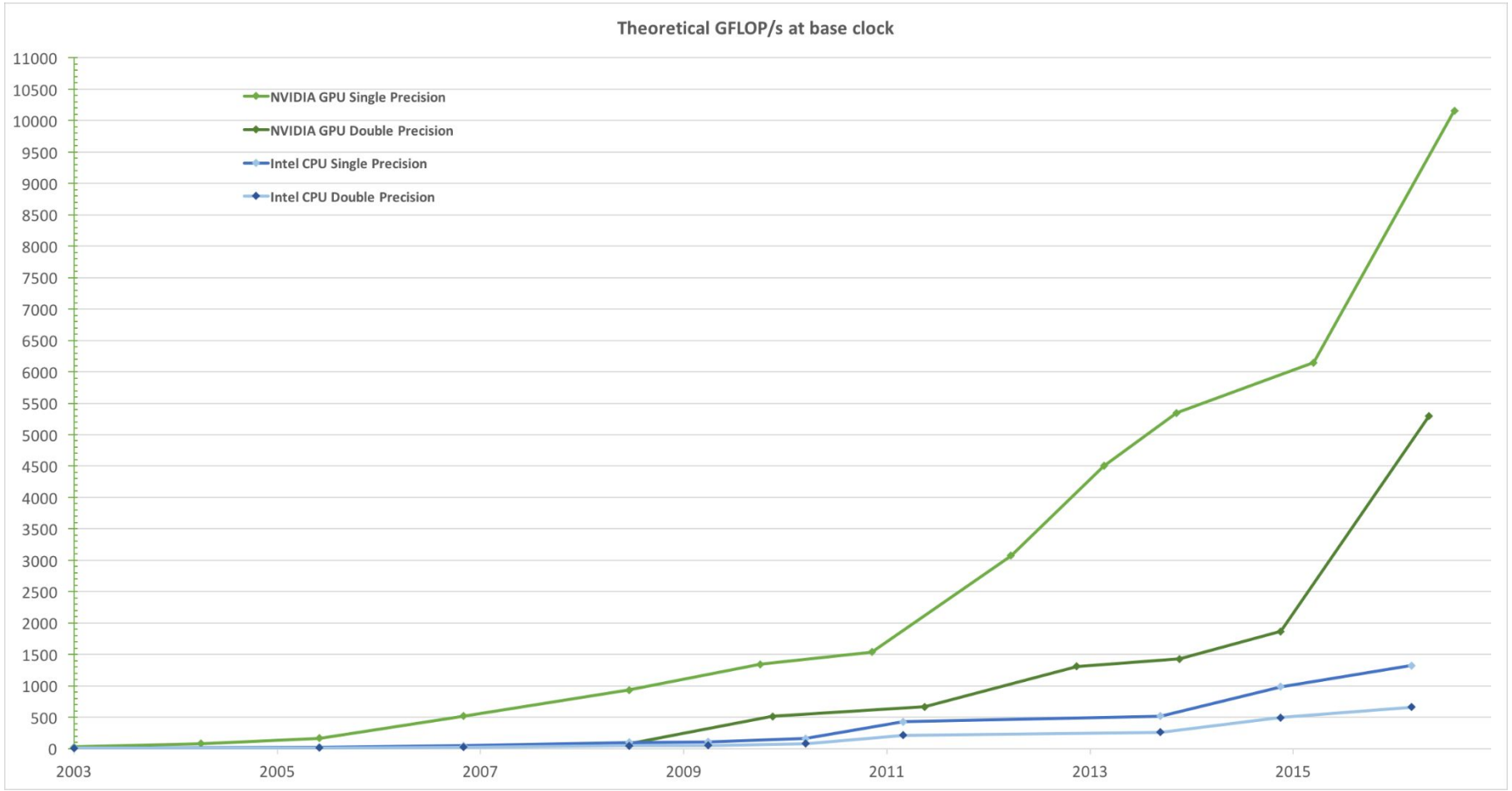


Image from: http://www.nvidia.com/object/drive-px.html

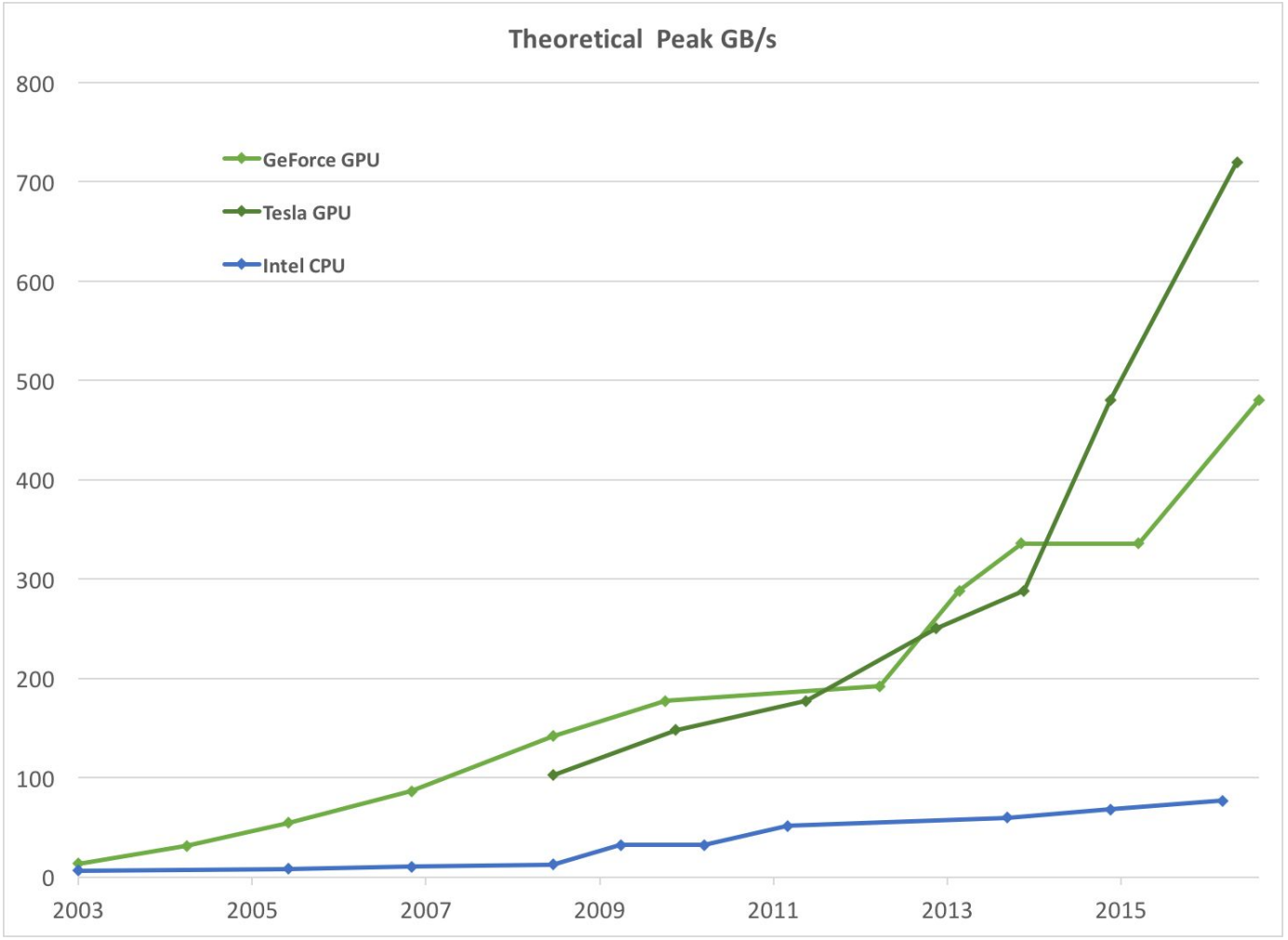
CPU and GPU Compute Trends



Penn Engineering

Chart from: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

CPU and GPU Memory Bandwidth Trends



Penn Engineering

Chart from: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

FLOP per…

- Single-core performance slowing since 2003
  - Power and heat limits
- GPUs delivery higher FLOP per
  - watt
  - mm
  - dollar

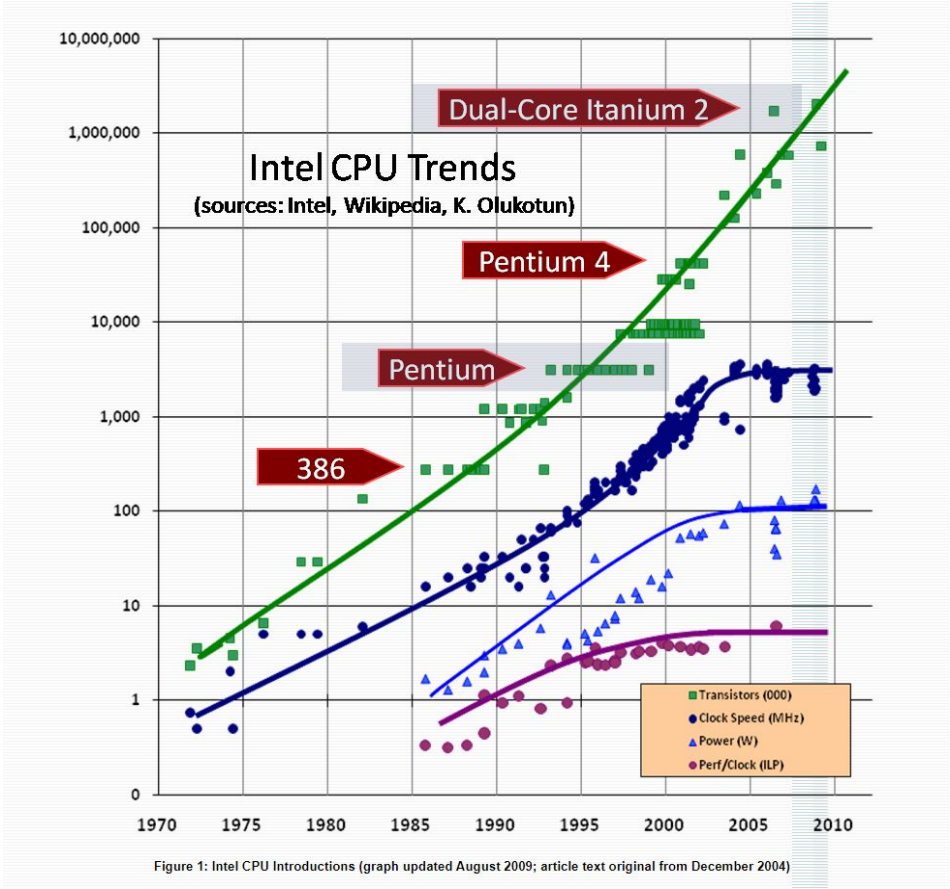


Figure 1: Intel CPU Introductions (graph updated August 2009; article text original from December 2004)

Penn Engineering

Chart from: http://www.gotw.ca/publications/concurrency-ddj.htm.

CPU Review

- What are the major components in a CPU die?

CPU Review

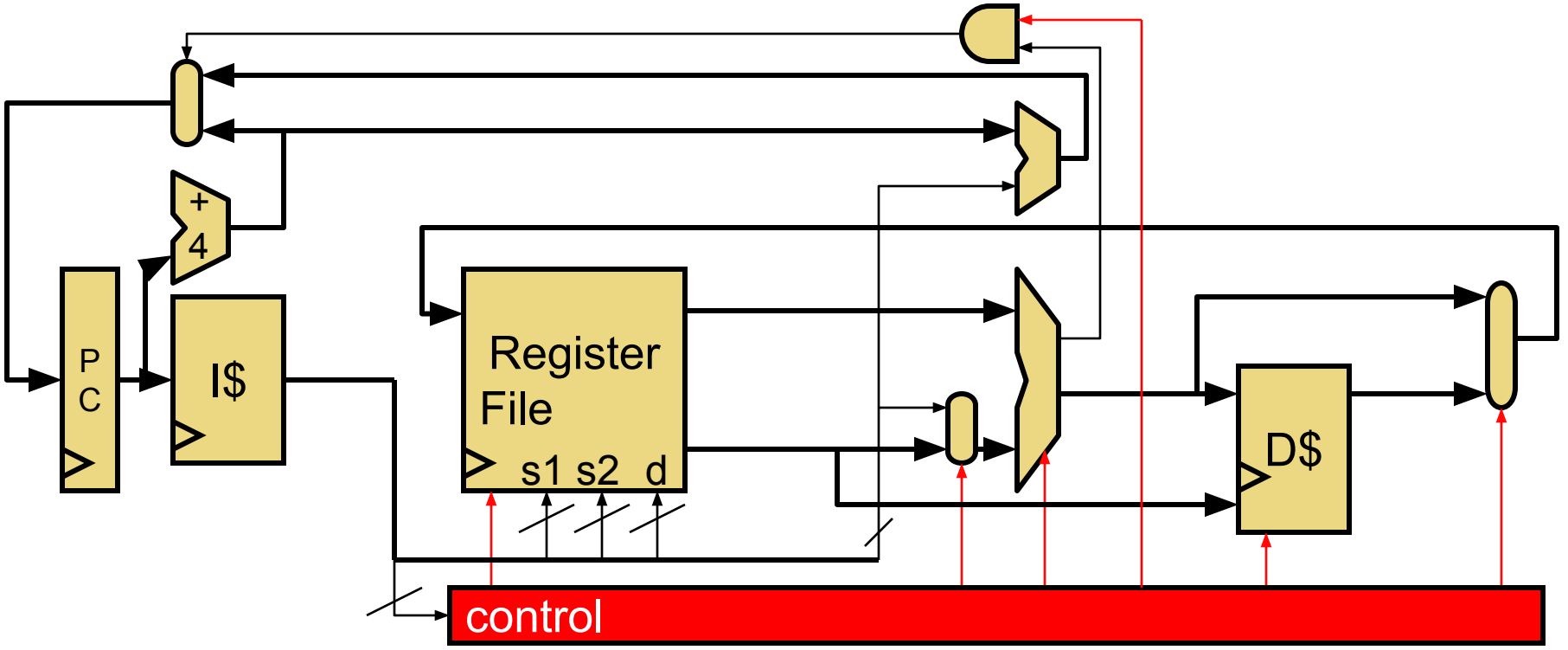
- Desktop Applications
  - Lightly threaded
  - Lots of branches
  - Lots of memory accesses

	`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

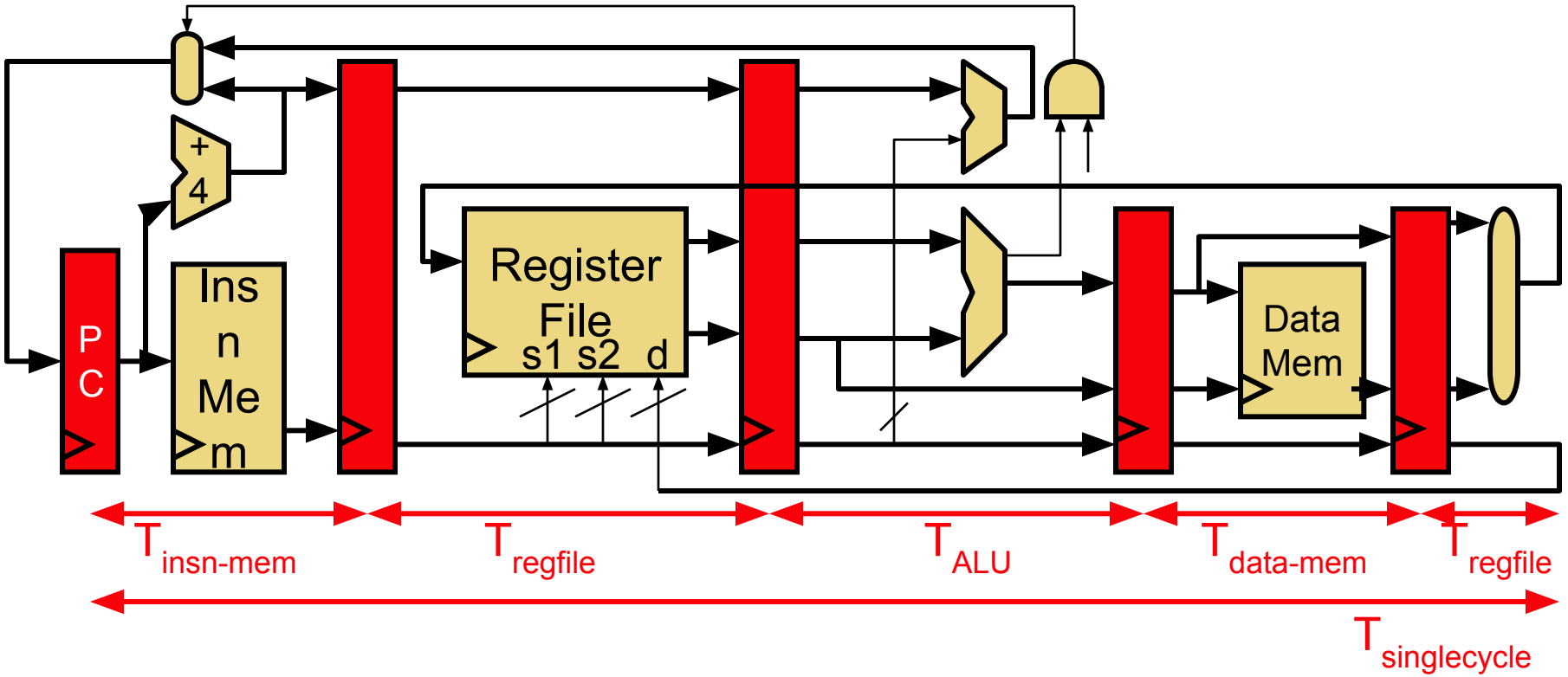
A Simple CPU Core



Fetch → Decode → Execute → Memory → Writeback

Image: Penn CIS501

Pipelining



Penn Engineering

Image: Penn CIS501

Branches

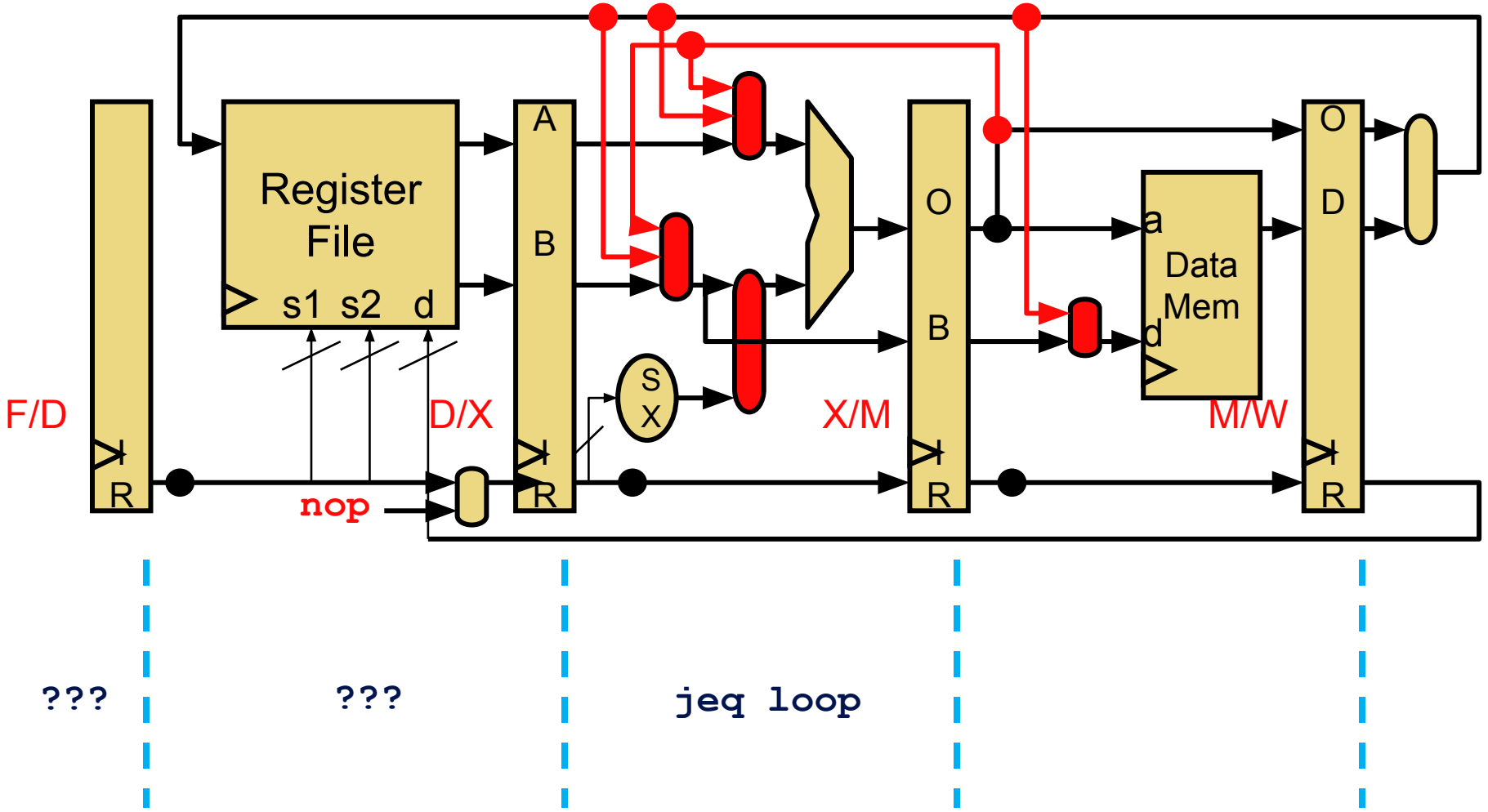


Image: Penn CIS501

Branch Prediction

+ Modern predictors > 90% accuracy
    - Raise performance and energy efficiency
- Area increase
- Potential fetch stage latency increase

- Security concerns? Spector/Meltdown

Memory Hierarchy

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

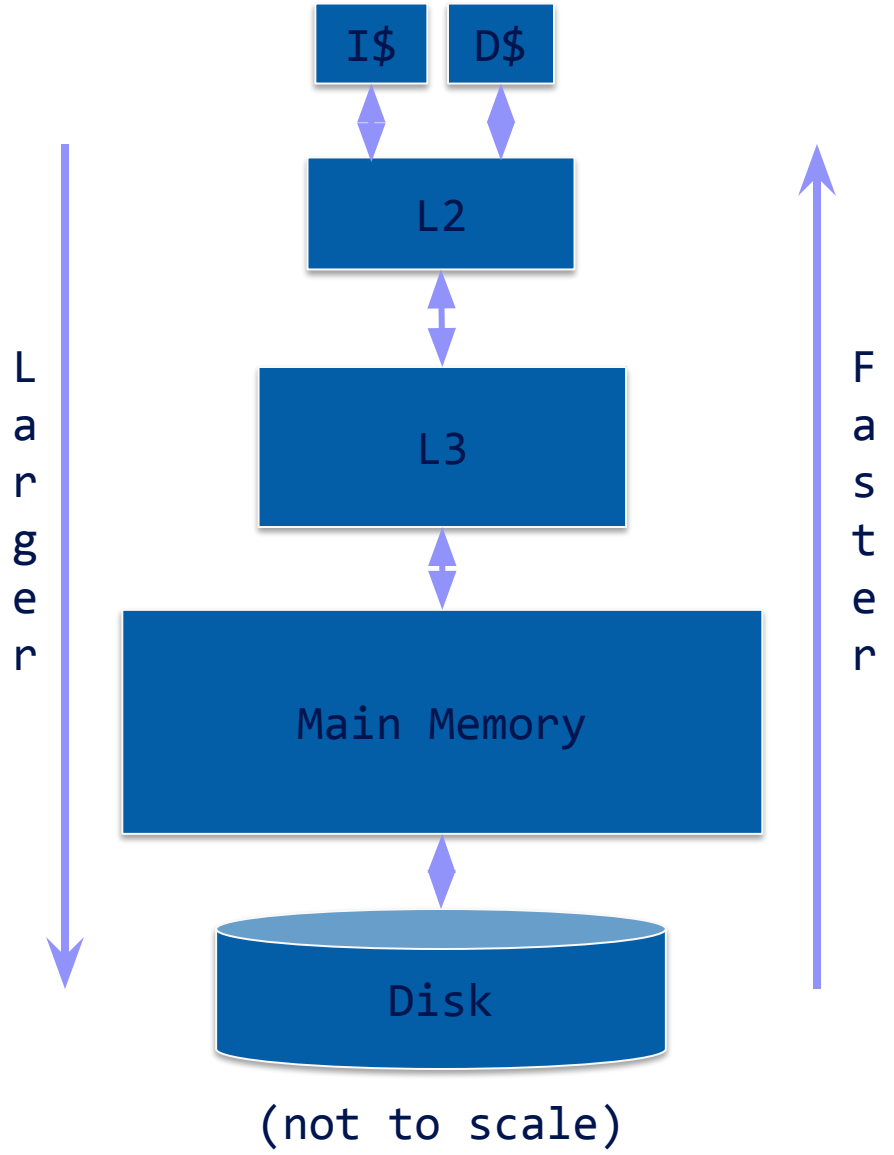
	Latency	Bandwidth	Size
SRAM (L1, L2, L3)	1-2ns	200-3000GBps	1-20MB
DRAM (memory)	70ns	20GBps	1-20GB
Flash/SSD (disk)	70-90µs	200-500MBps	100-1000GB
HDD (disk)	10ms	1-150MBps	500-3000GB

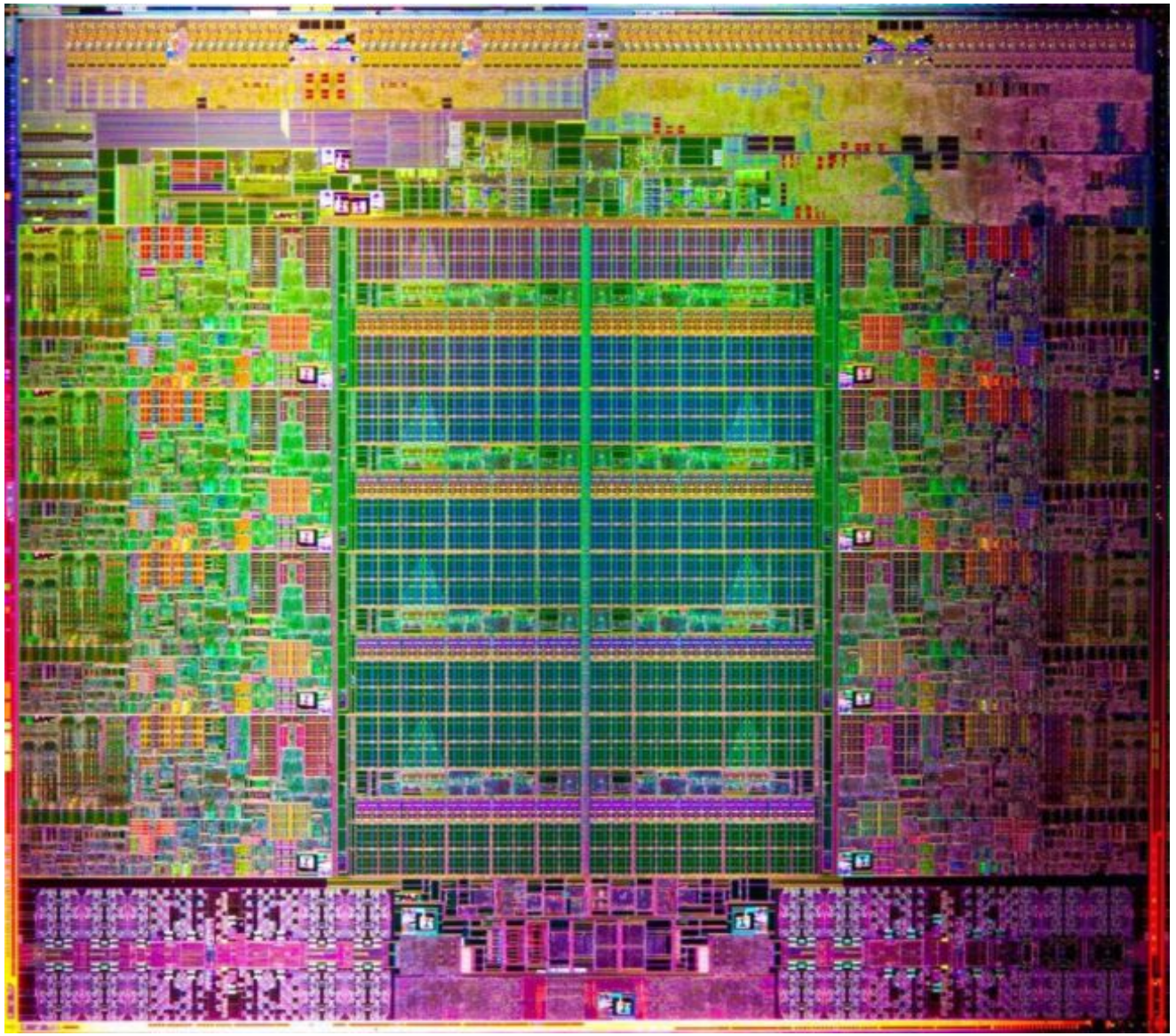
Caching

- Keep data we 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

Cache Hierarchy

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

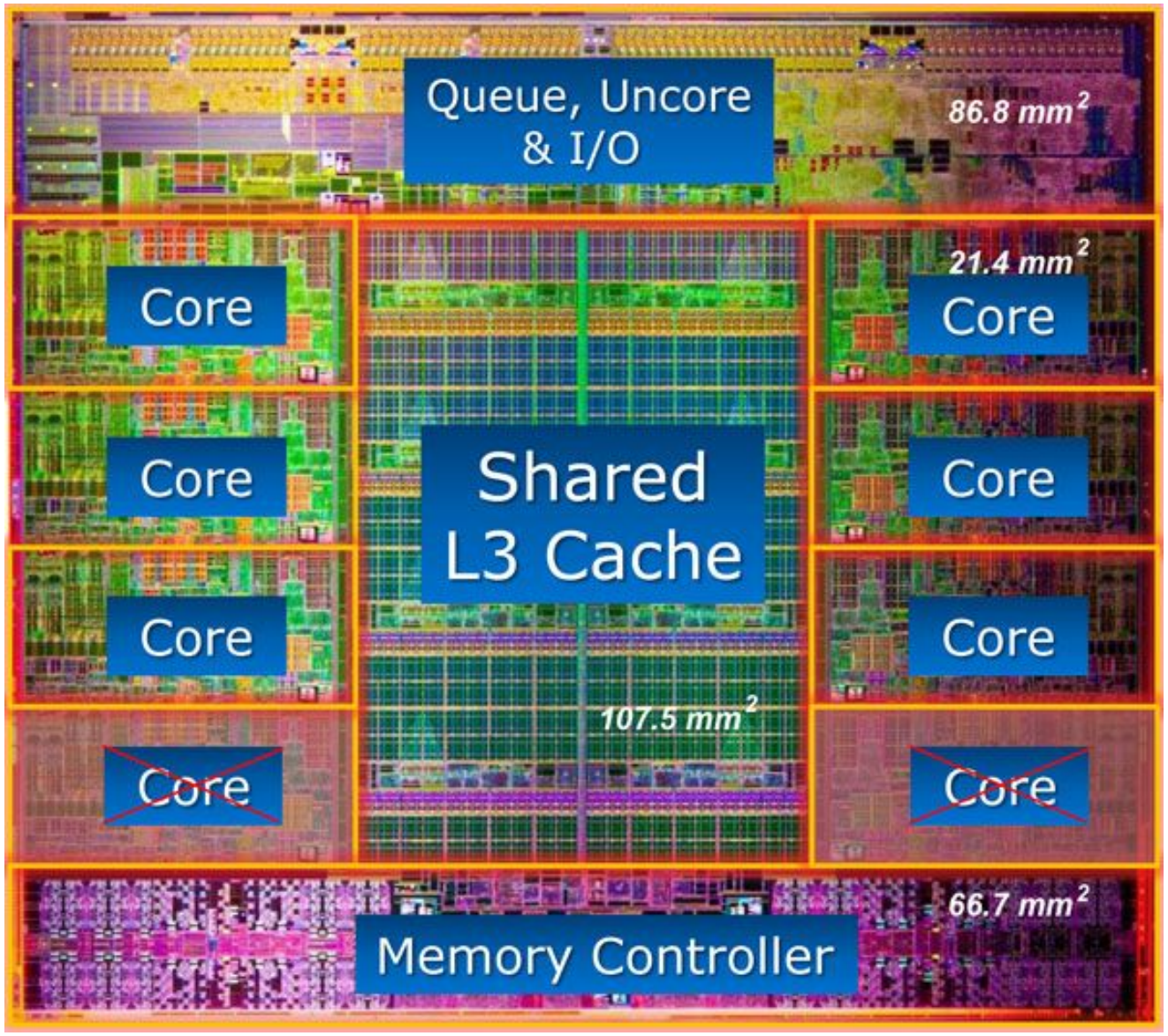




Intel Core i7 3960X (Sandy Bridge) – 15MB L3 (25% of die).
4-channel Memory Controller, 51.2GB/s total

Penn Engineering

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

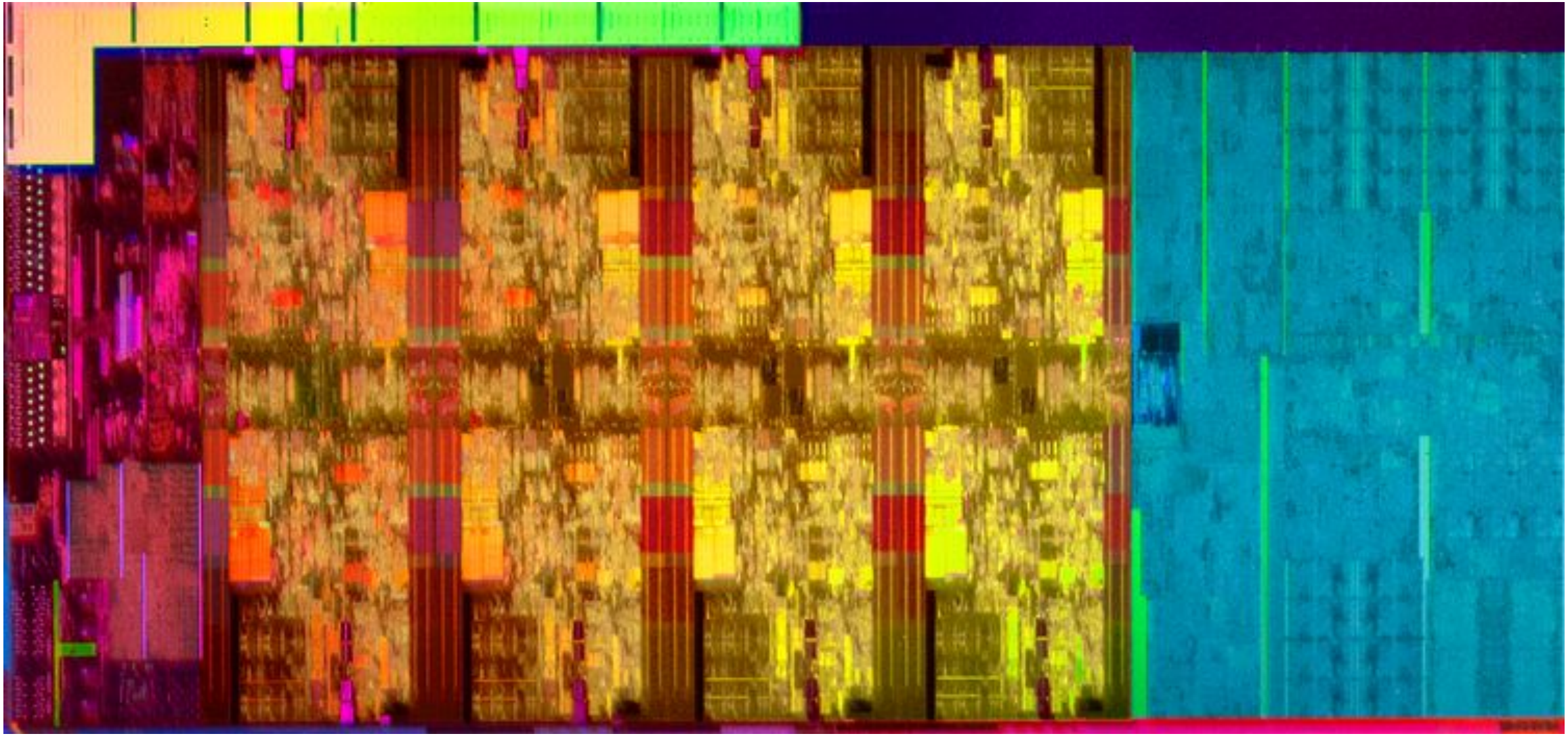


Intel Core i7 3960X (Sandy Bridge) – 15MB L3 (25% of die).

4-channel Memory Controller, 51.2GB/s total

Penn Engineering

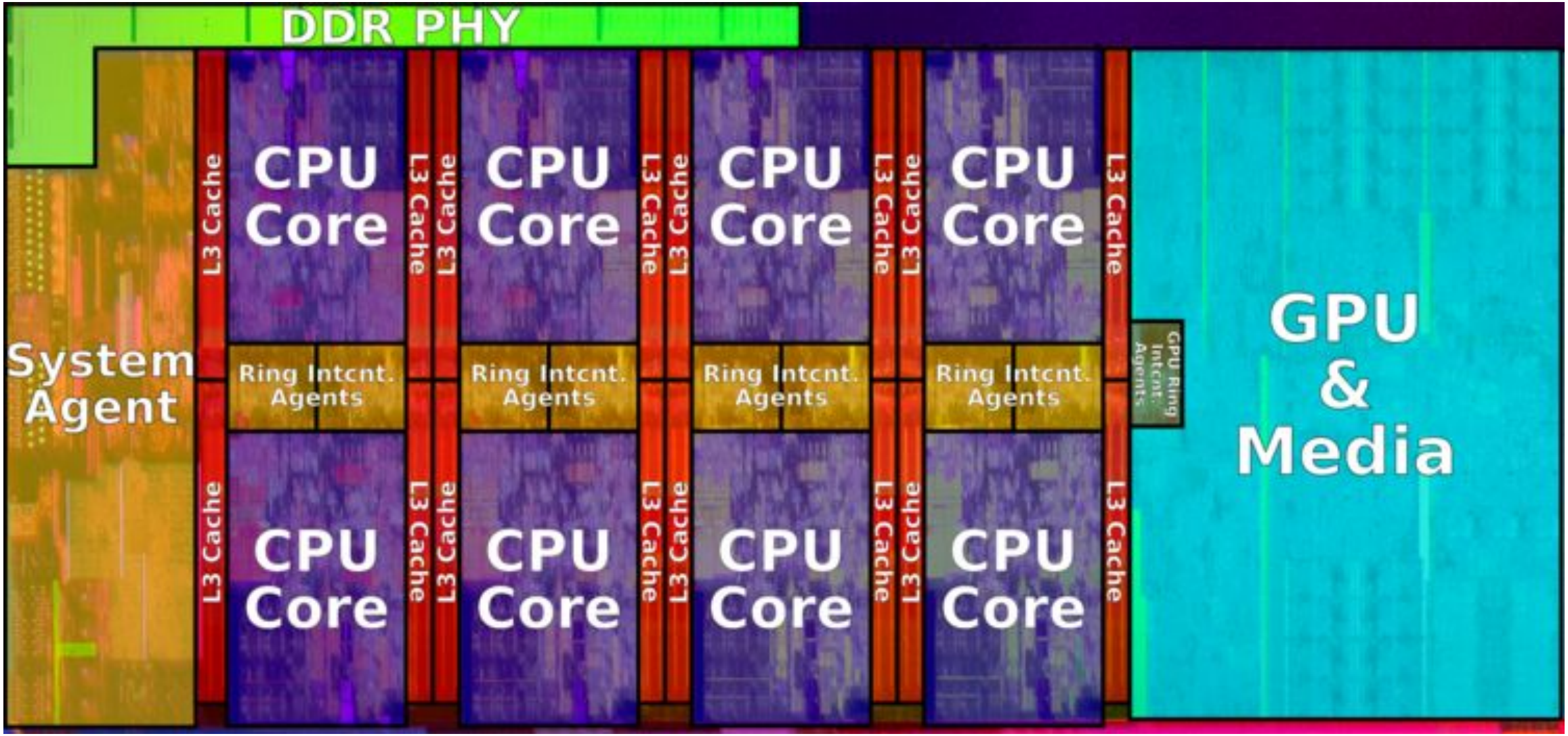
Source: https://en.wikichip.org/wiki/intel/core_i7ee/i7-3960x



Intel Coffee Lake Octa-Core CPU Die (i9 9900 Series): ~174 mm² die size, 2 channel memory 39.74 GB/s

8 CPU cores + 24 GPU EUs

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



Intel Coffee Lake Octa-Core CPU Die (i9 9900 Series): ~174 mm² die size, 2 channel memory 39.74 GB/s

8 CPU cores + 24 GPU EUs

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

Improving IPC

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

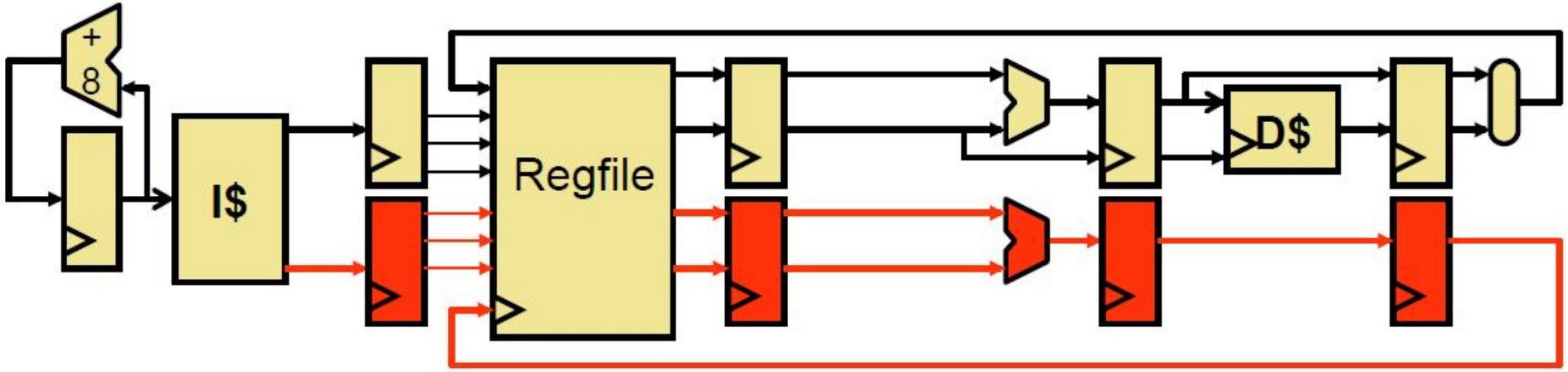


Image: Penn CIS371

Scheduling

- Consider instructions:

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

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

Register Renaming

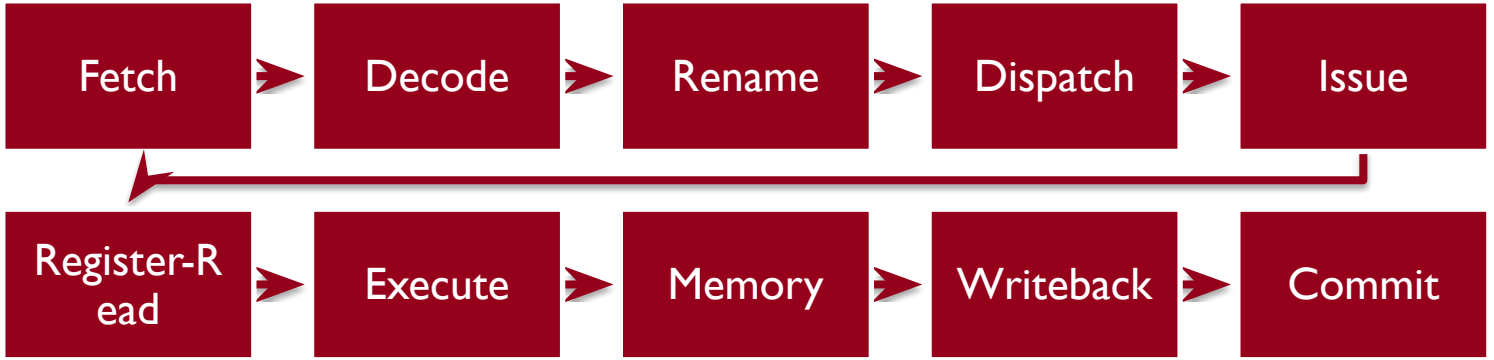
- How about this instead:

```
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



- Reorder Buffer (ROB)
  - Keeps track of status for in-flight instructions
- Physical Register File (PRF)
- Issue Queue/Scheduler
  - Chooses next instruction(s) to execute

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

Vectors Motivation

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

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+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-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

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

Simultaneous Multithreading

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

+ Minimal hardware duplication

+ More scheduling freedom for Out-of-Order Execution (OoO)

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

Multicore

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

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

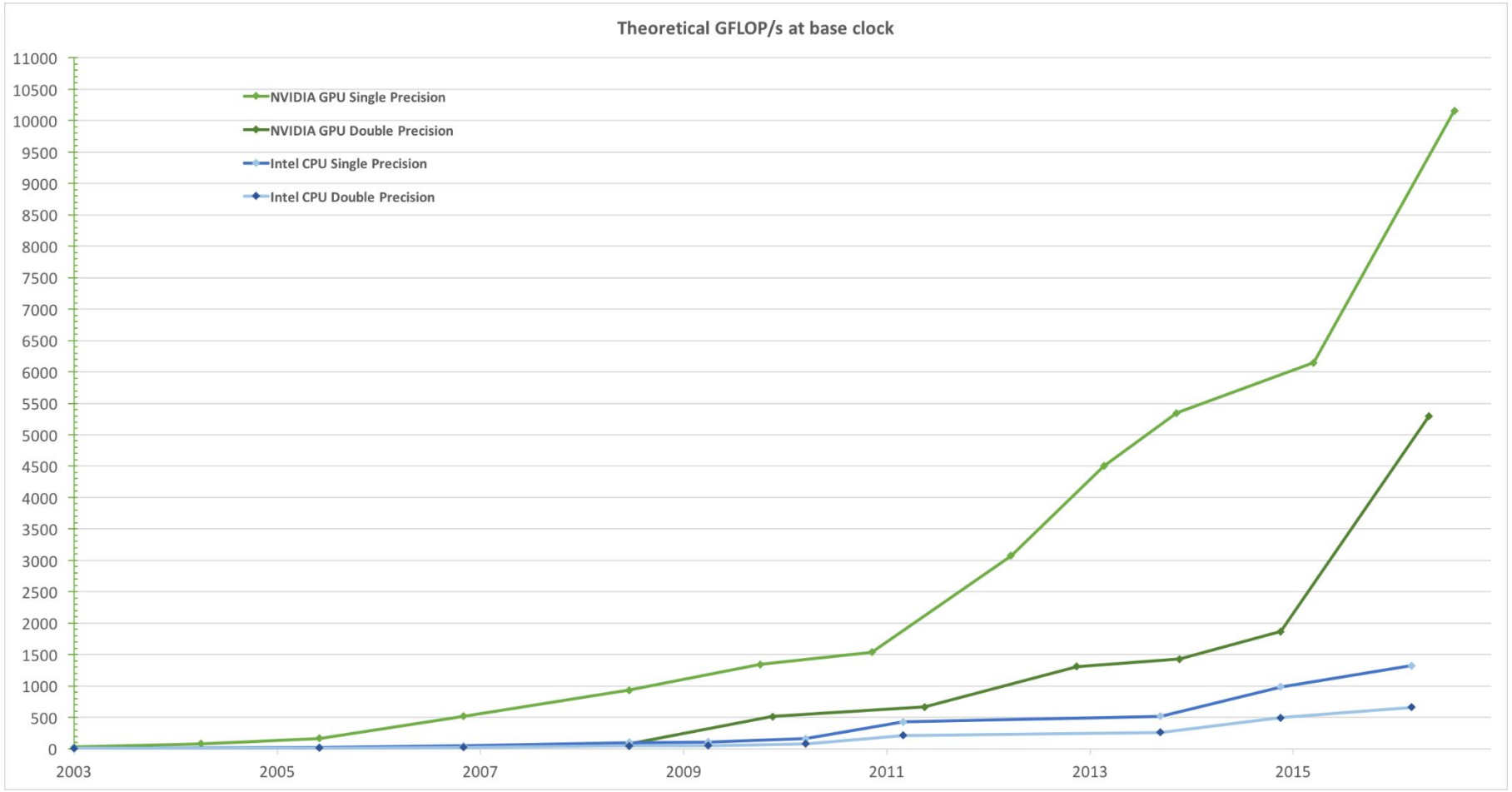
+ Easier way to take advantage of Moore's Law

– 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 32K threads?

How did this happen?

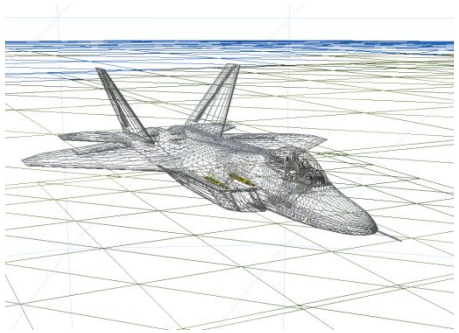


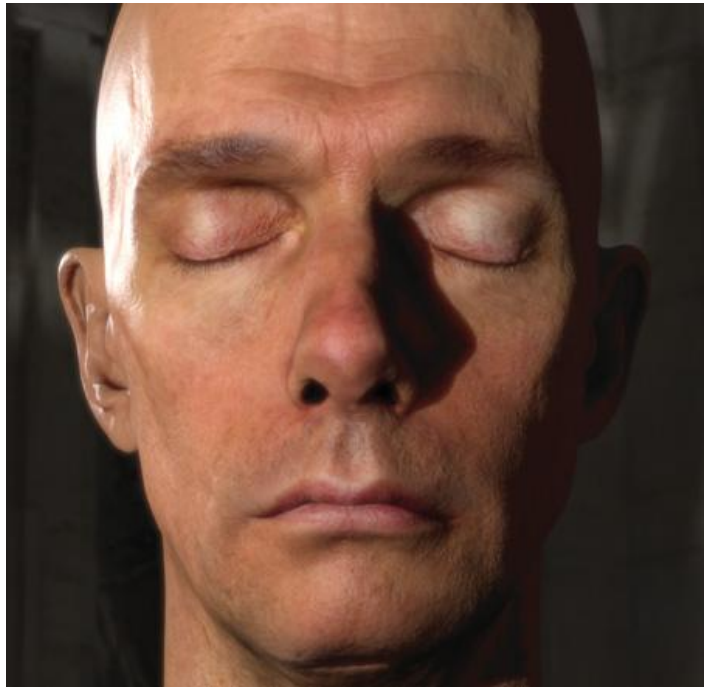
Penn Engineering

Chart from: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Graphics Workloads

- Triangles/vertices and pixels/fragments





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

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.

Data Parallel

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

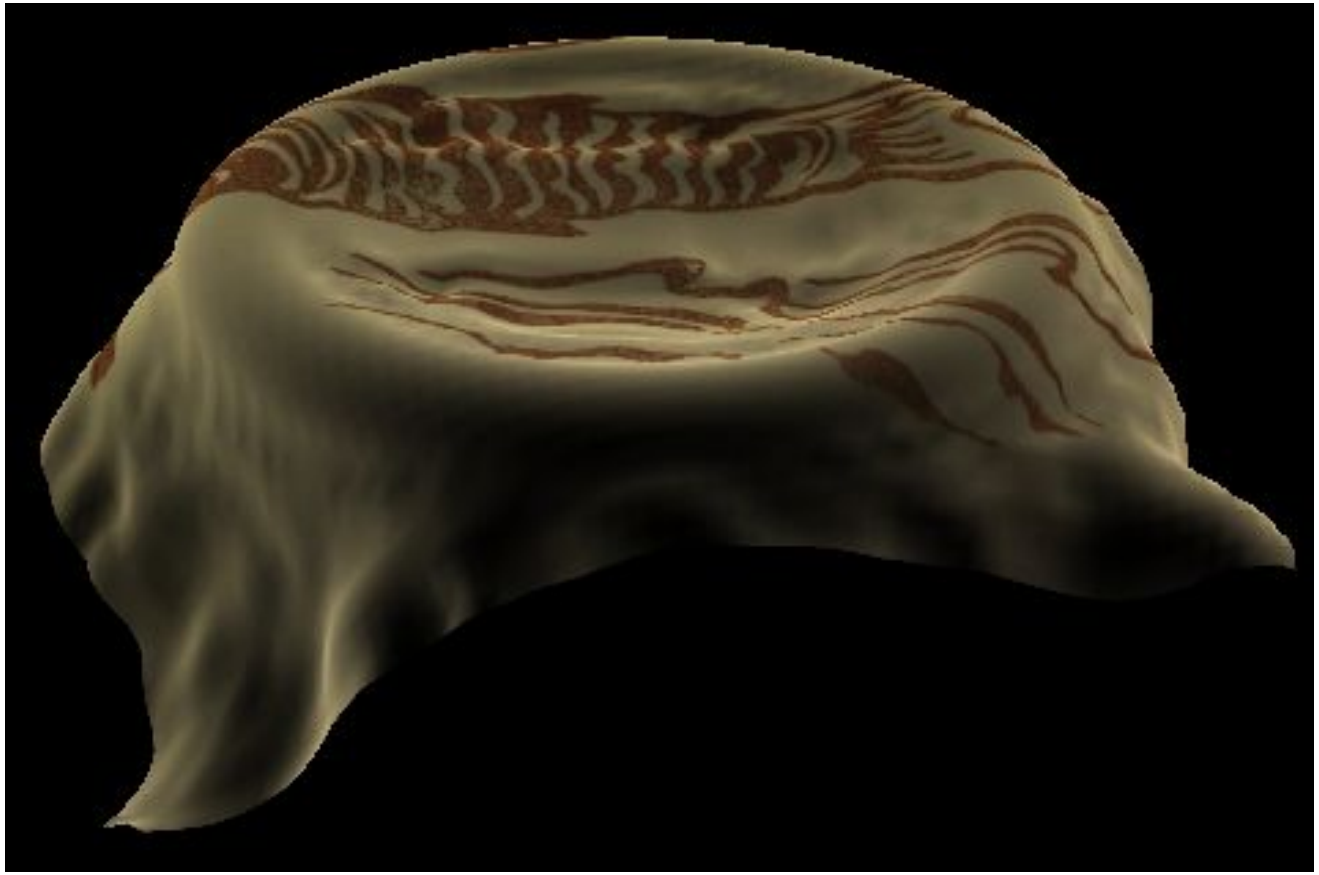


Image from: https://plus.google.com/u/0/photos/100838748547881402137/albums/5407605084626995217/5581900335460078306

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.

Independent logical sequence of control per fragment. *

Compile shader

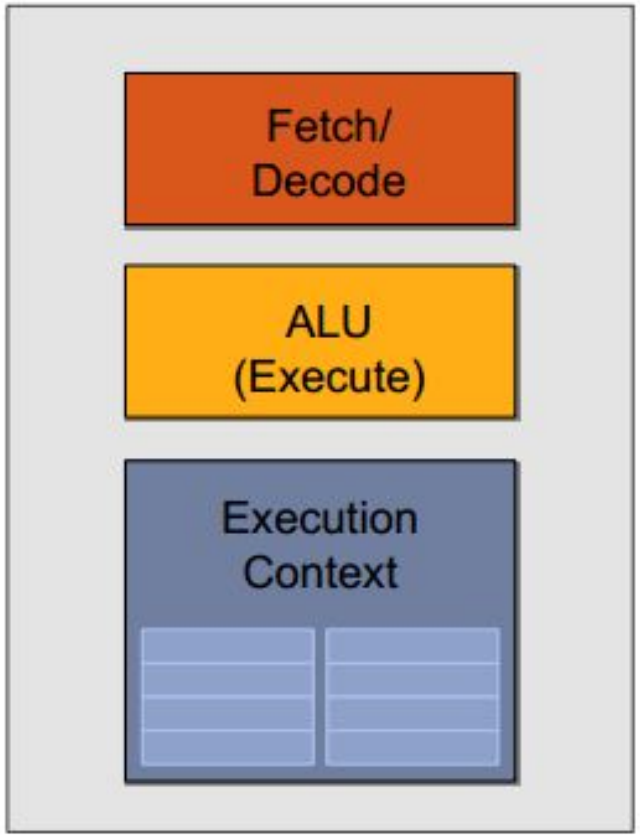
1 unshaded fragment input record

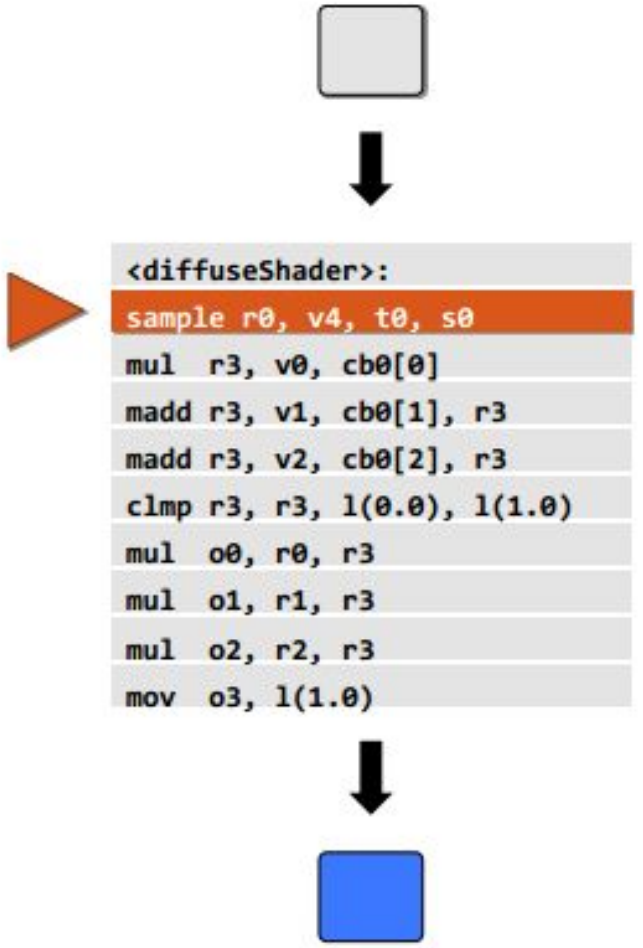
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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);
}
```

```
<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)
```

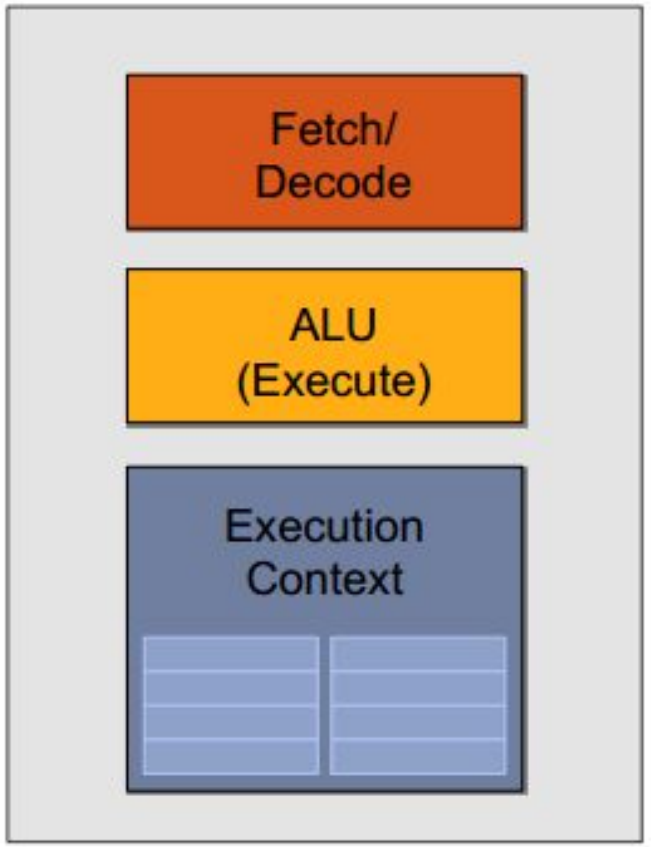
1 shaded fragment output record

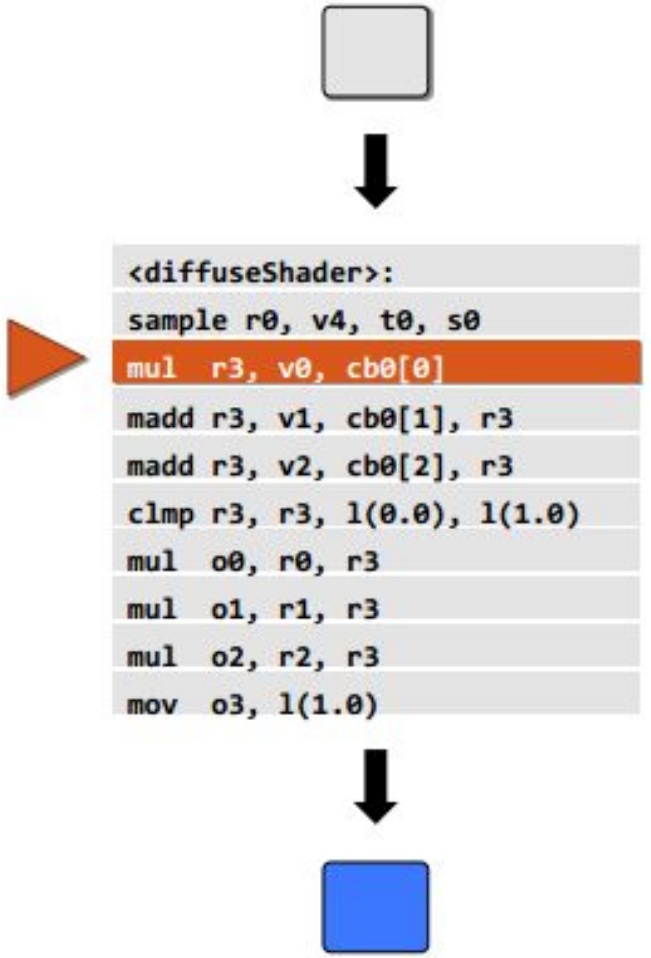
Execute shader





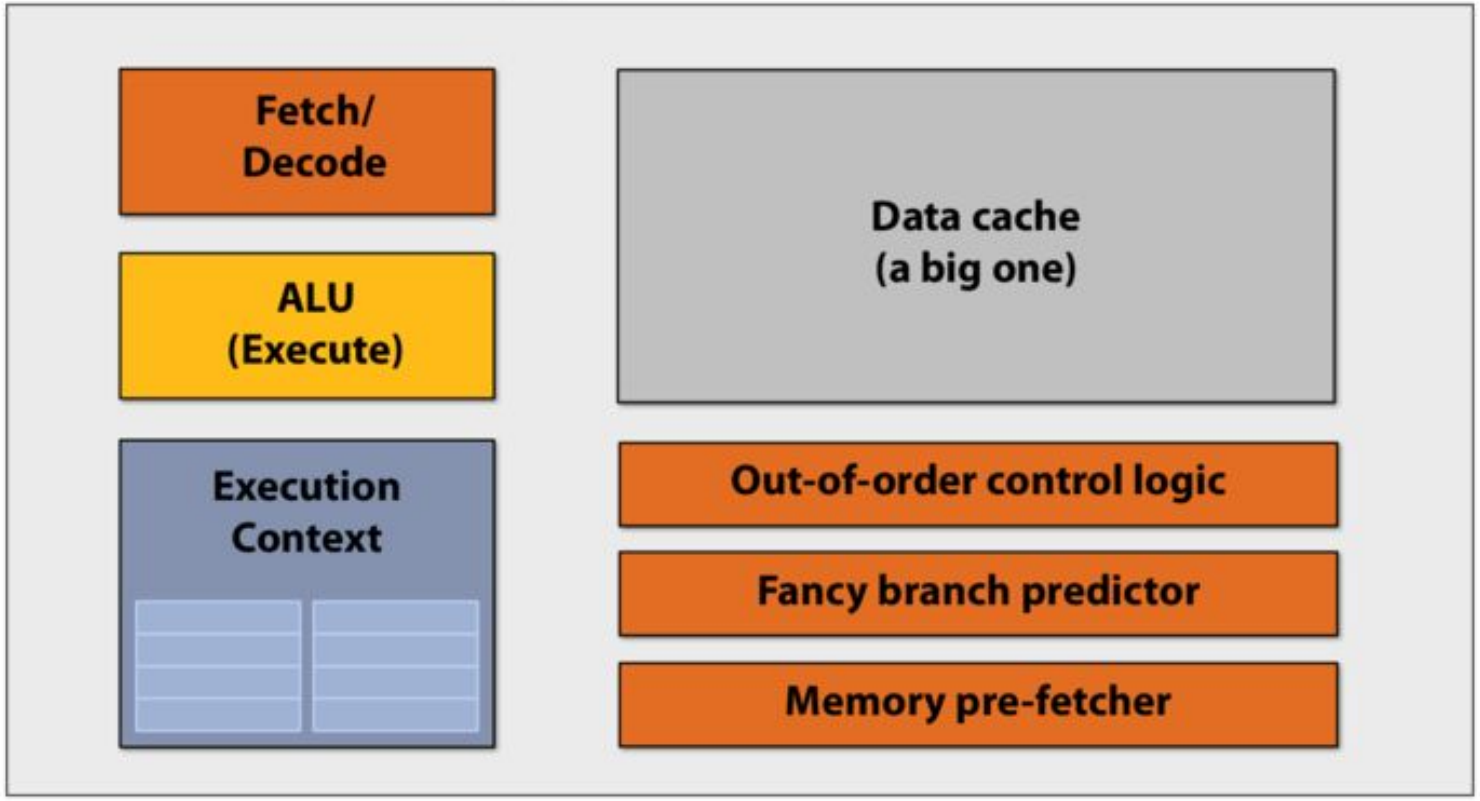
Execute shader





CPU: pre multi-core era

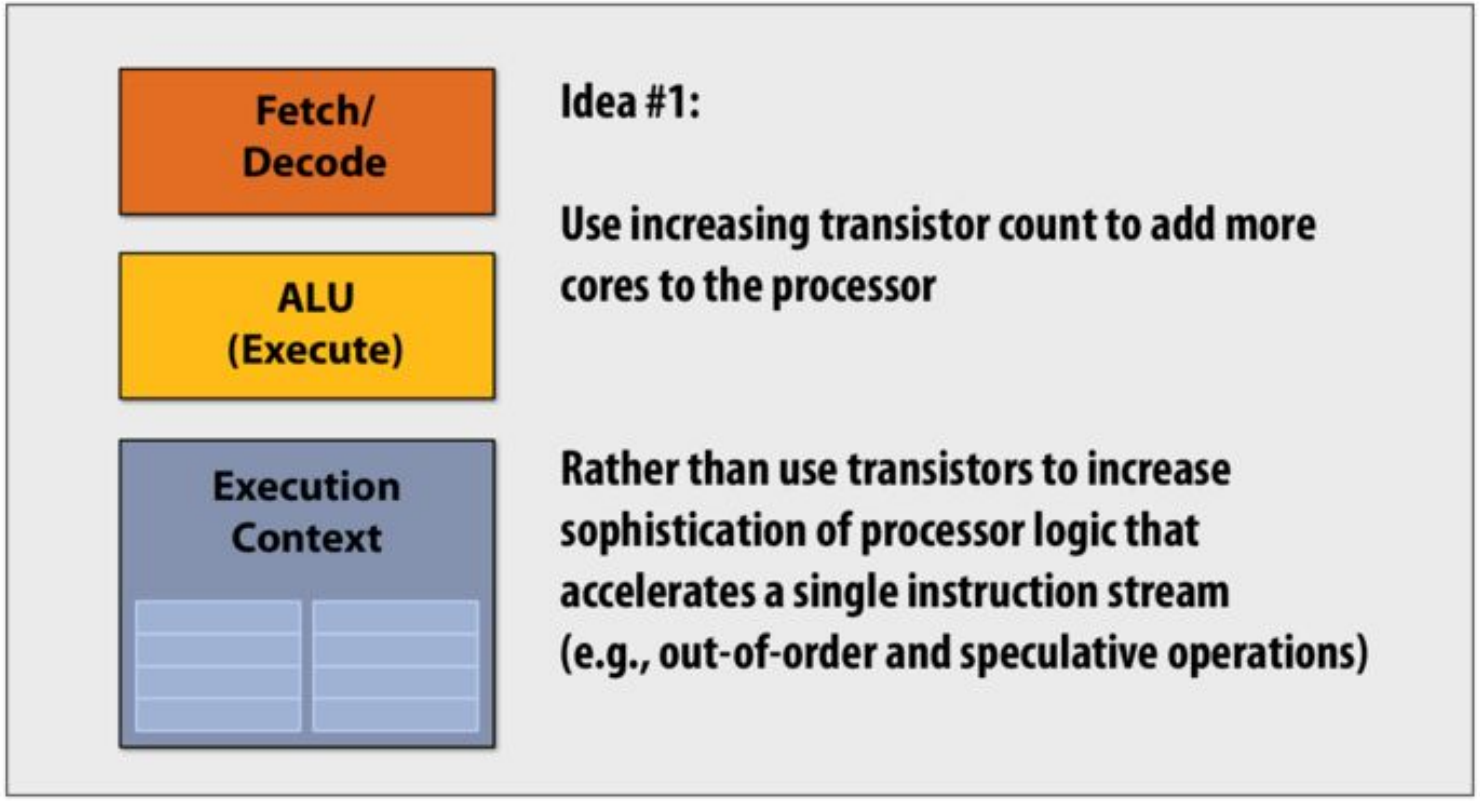
Majority of chip transistors perform operations that help a single instruction stream run fast.



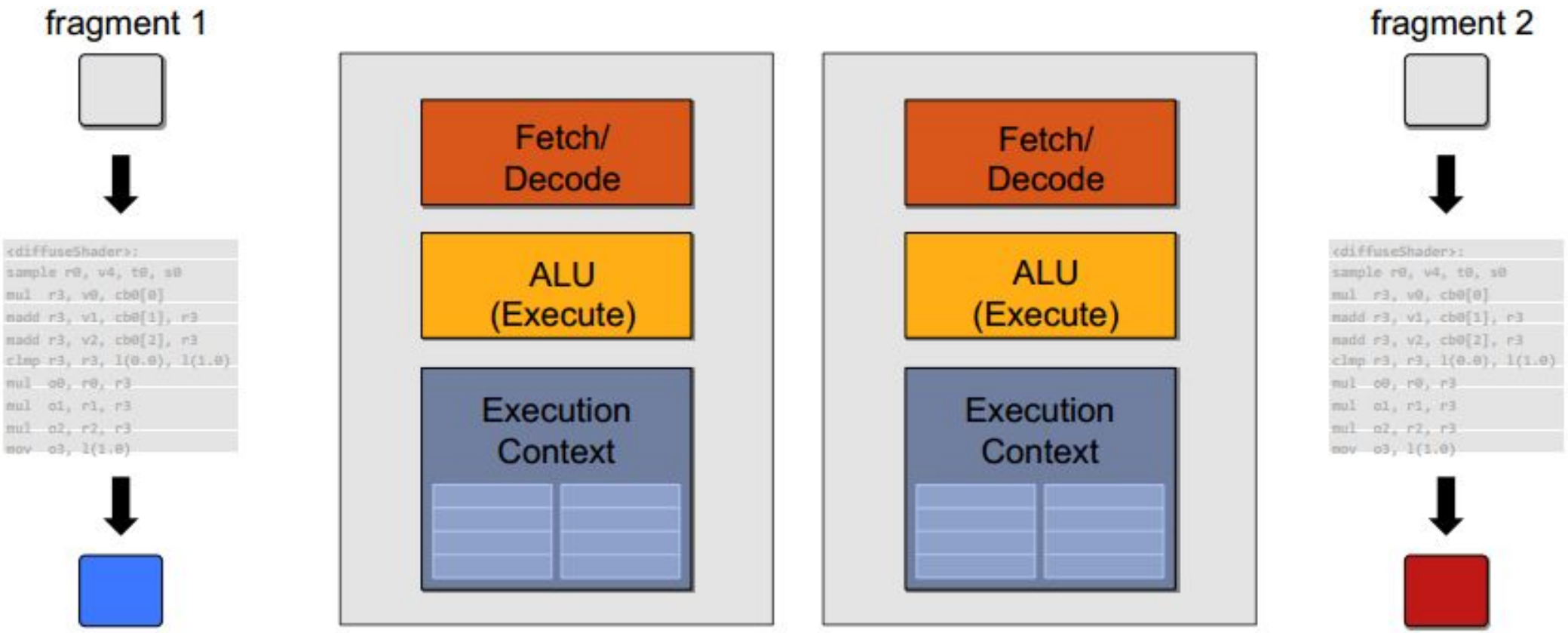
More transistors = larger cache, smarter out-of-order logic, smarter branch predictor, etc.

(Also: more transistors → smaller transistors → higher clock frequencies)

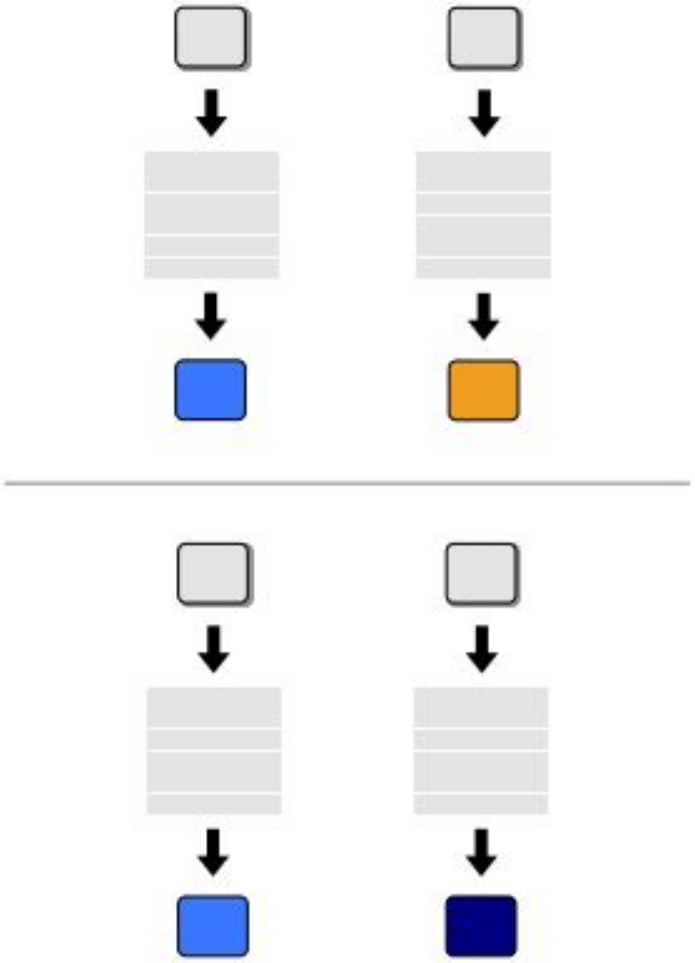
CPU: multi-core era

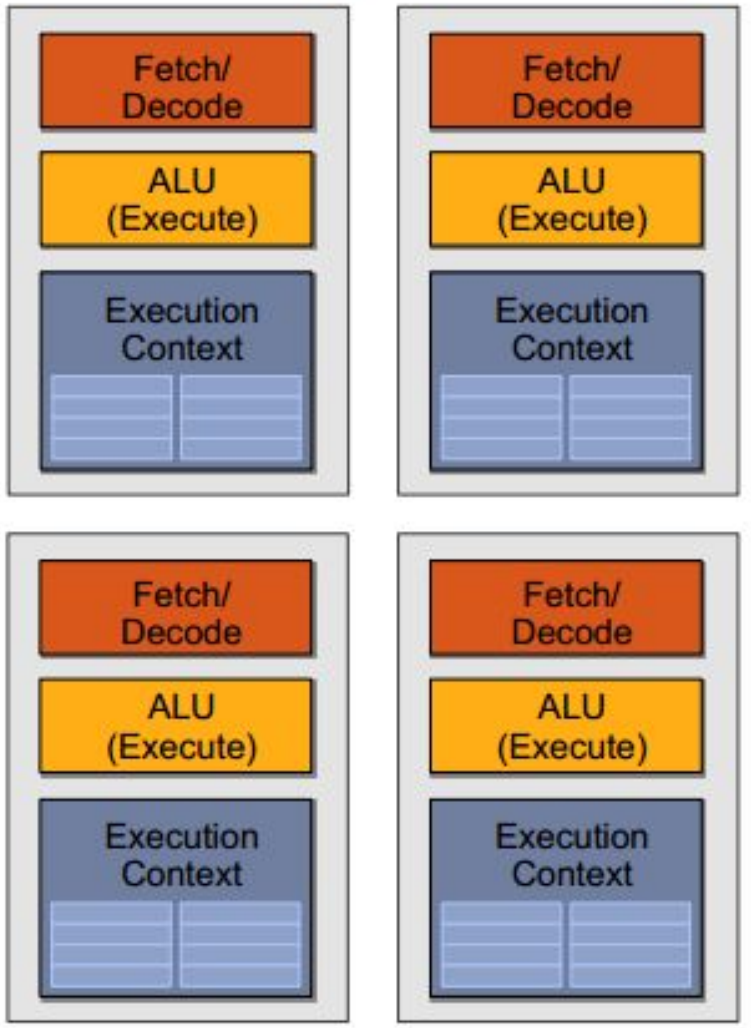


Two cores (two fragments in parallel)

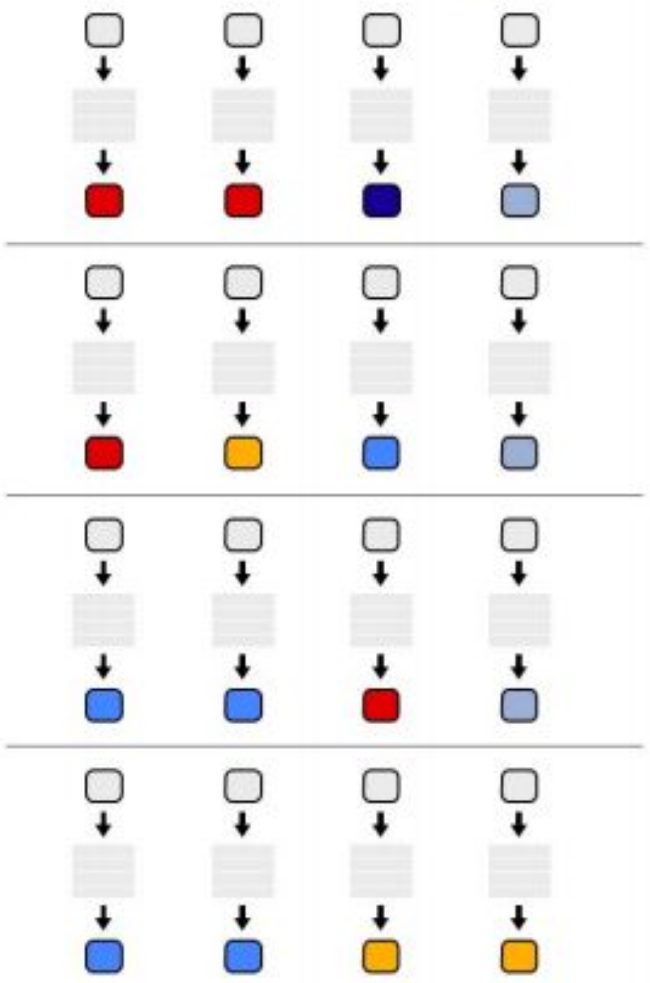


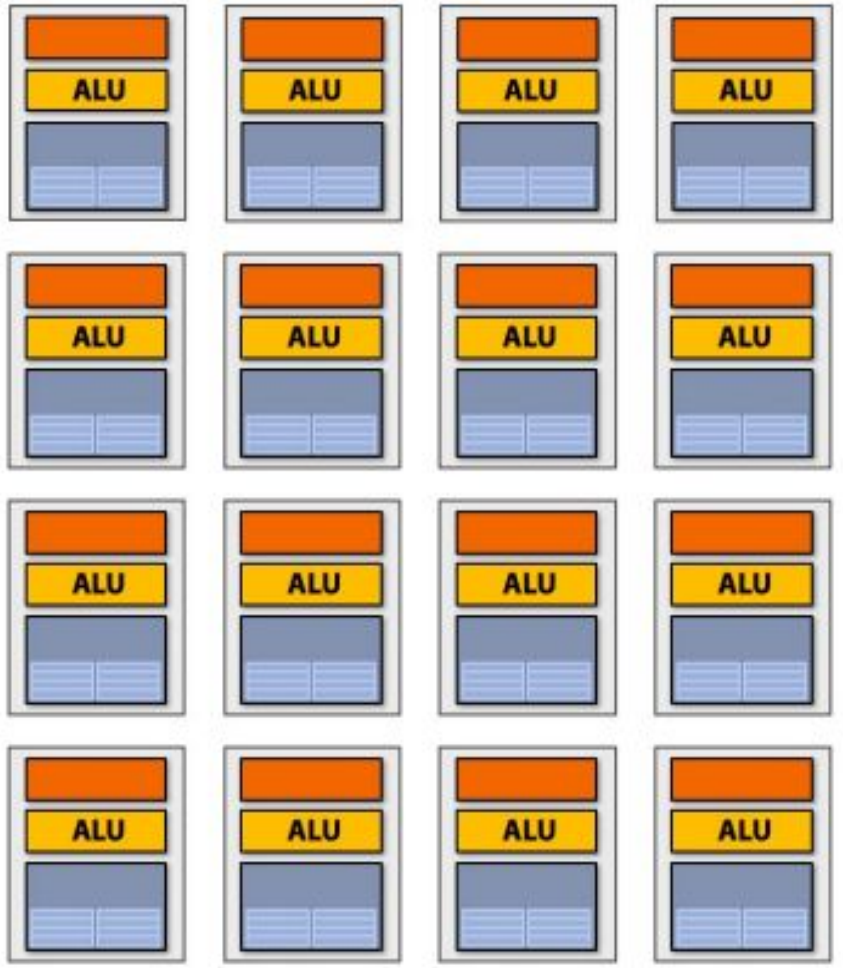
Four cores (four fragments in parallel)





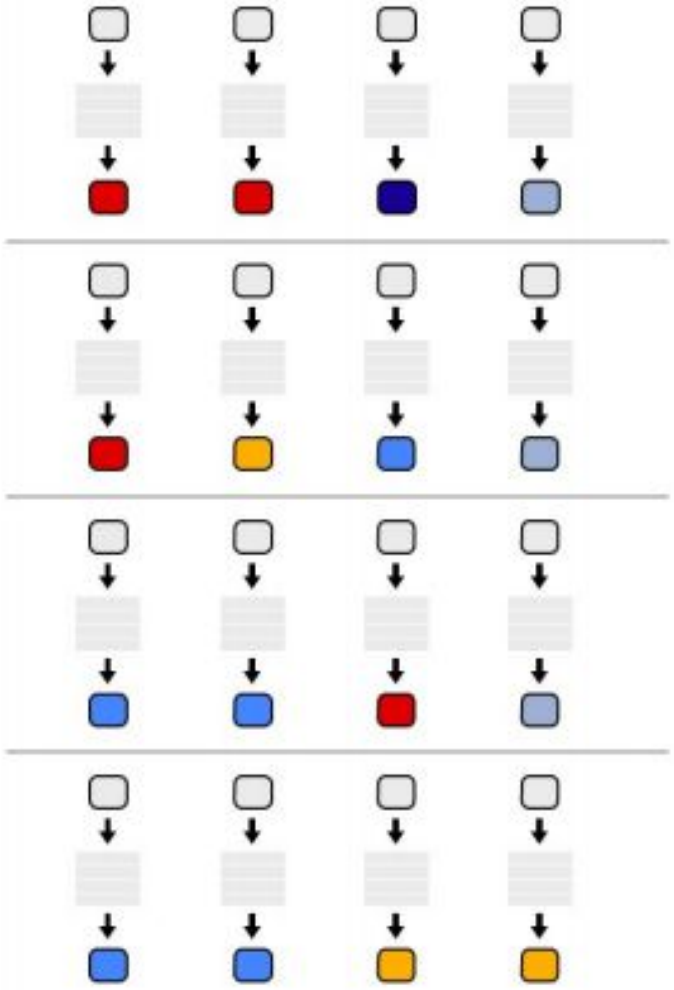
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

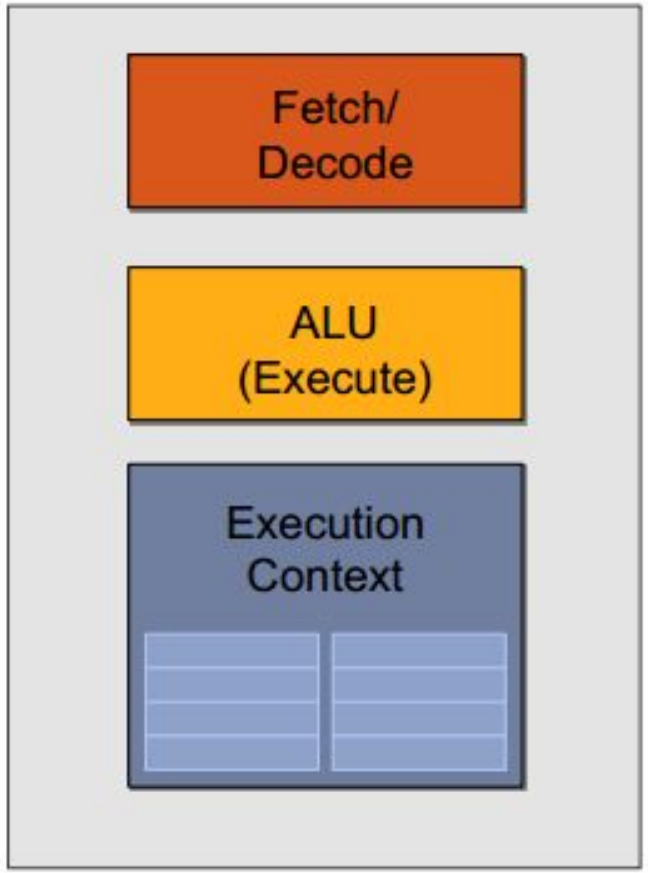
Instruction stream sharing



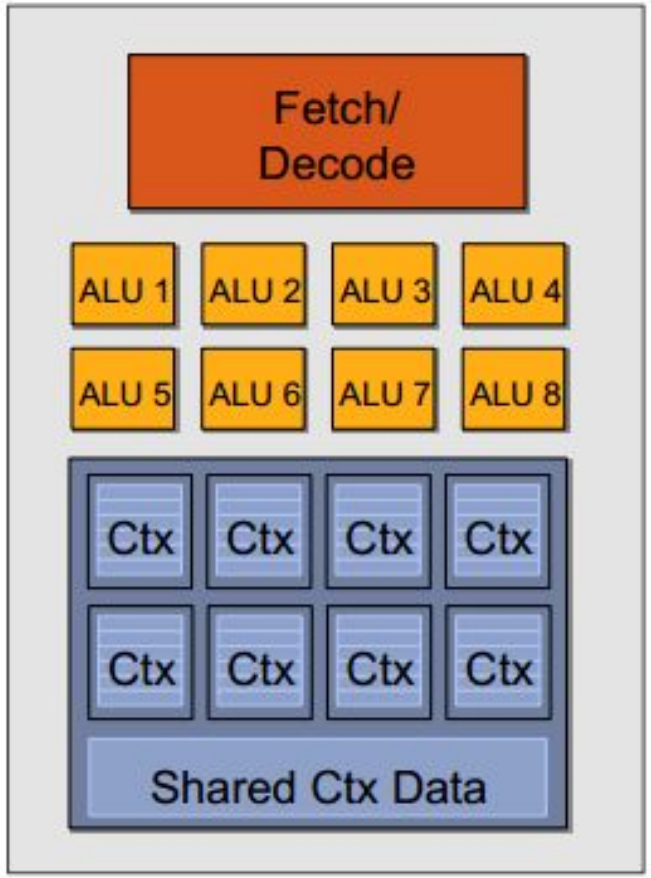
But ... many fragments should be able to share an instruction stream!

```
<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)
```

Recall: simple processing core



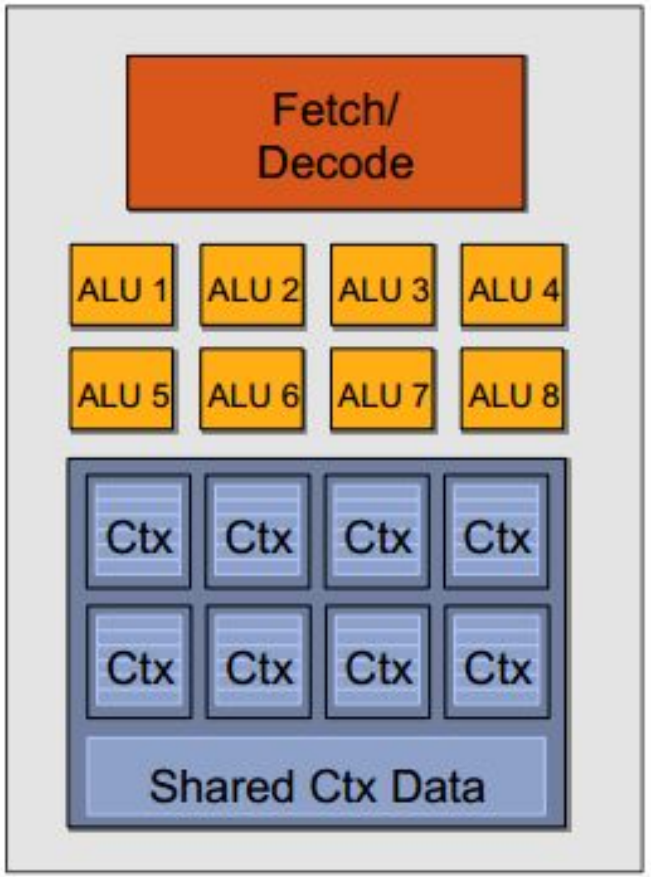
Add ALUs

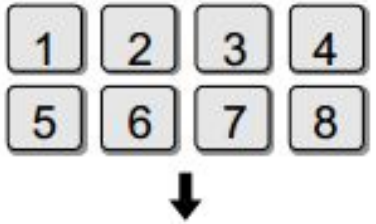


Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

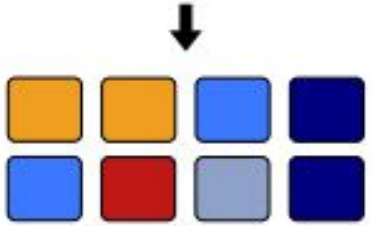
SIMD processing

Modifying the shader

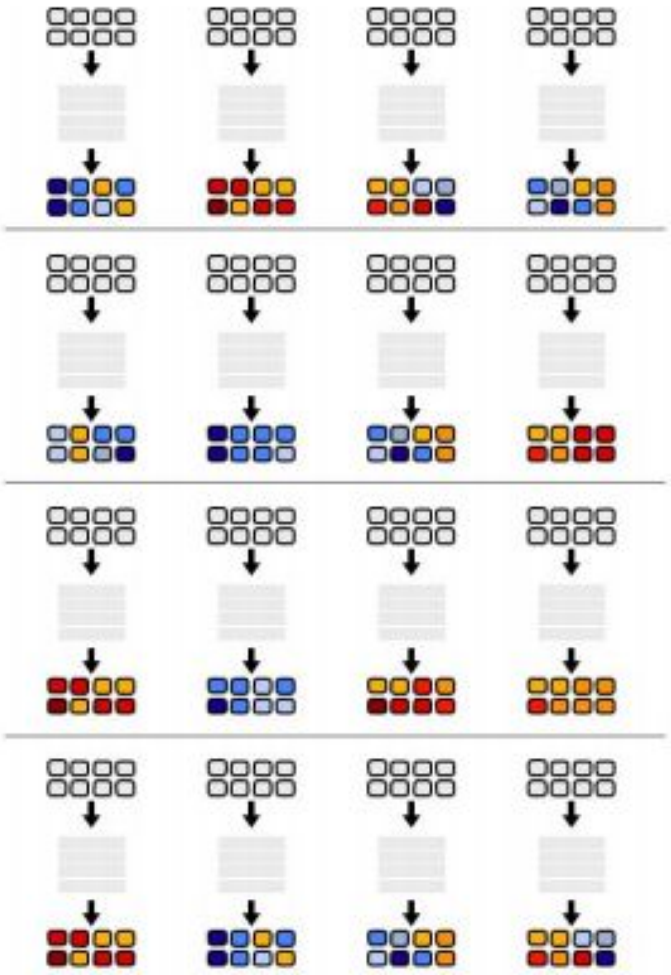




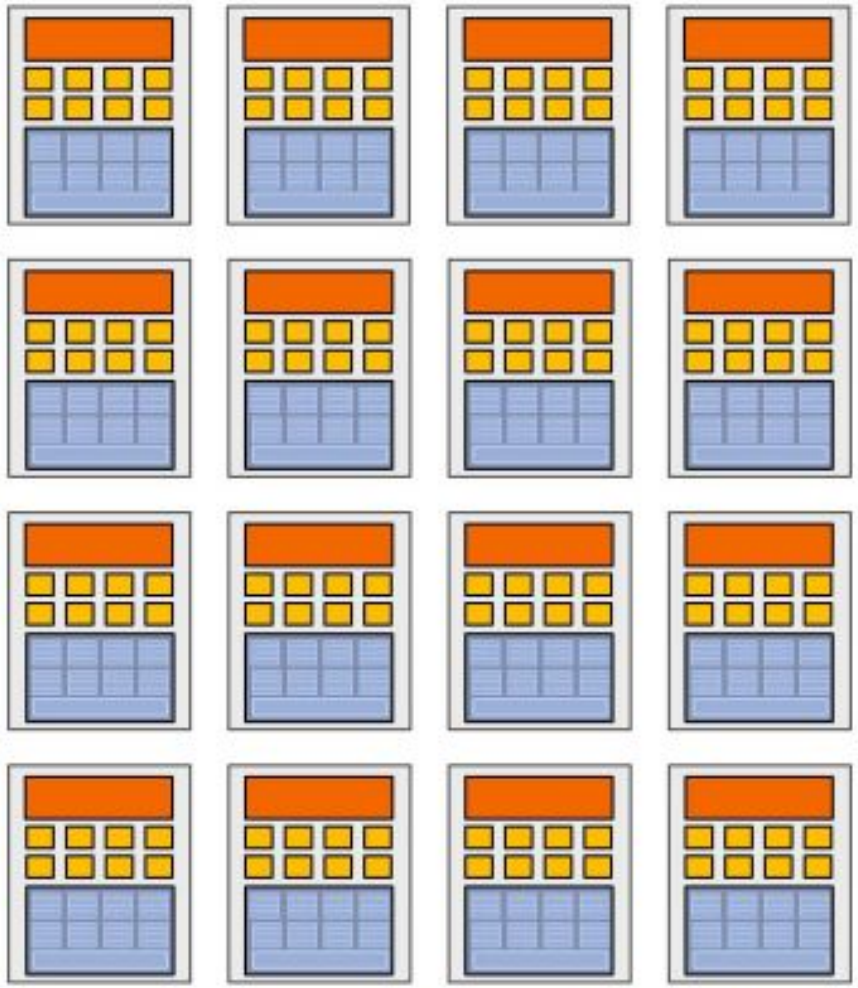

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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



128 fragments in parallel

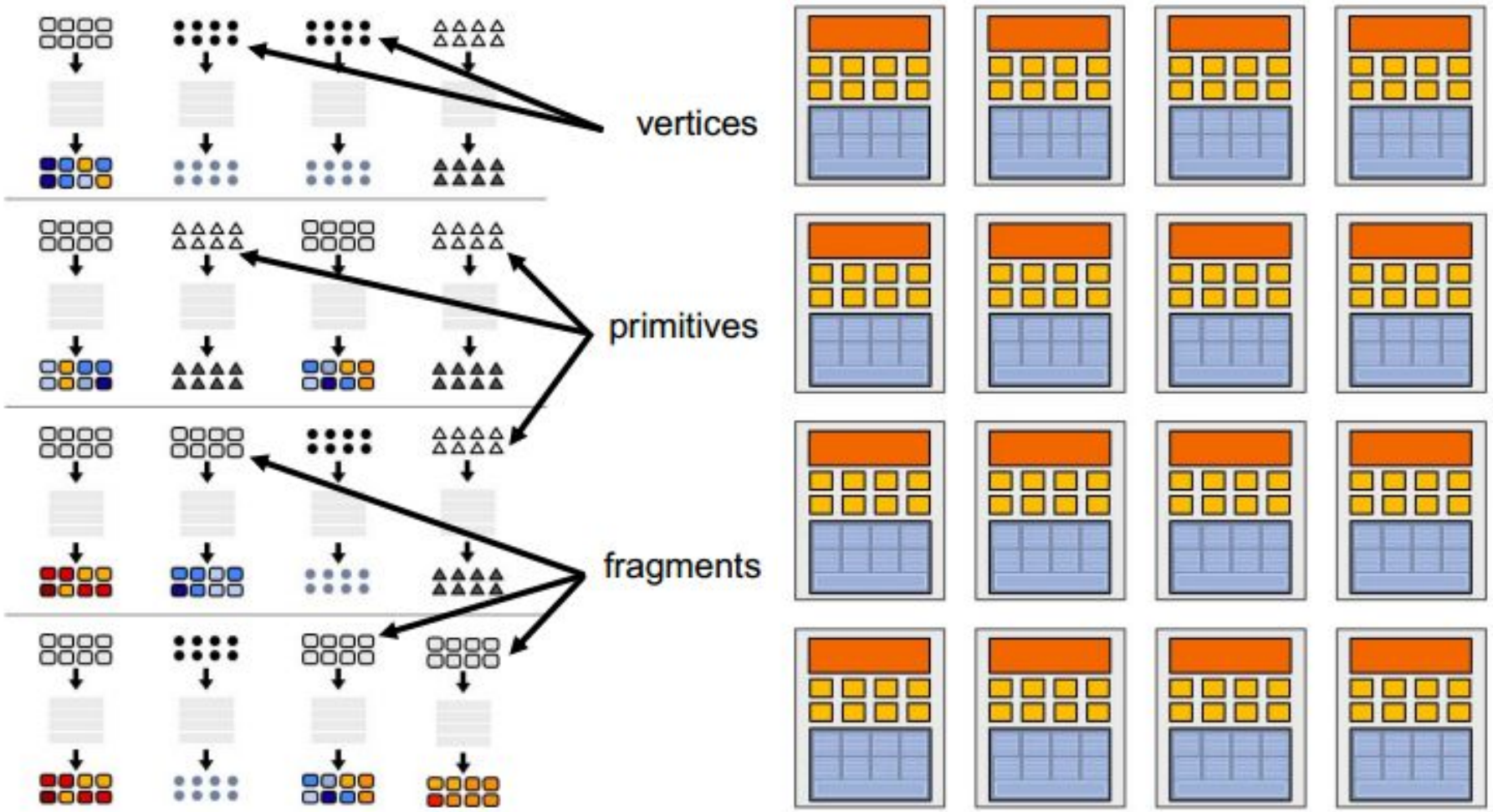


16 cores = 128 ALUs

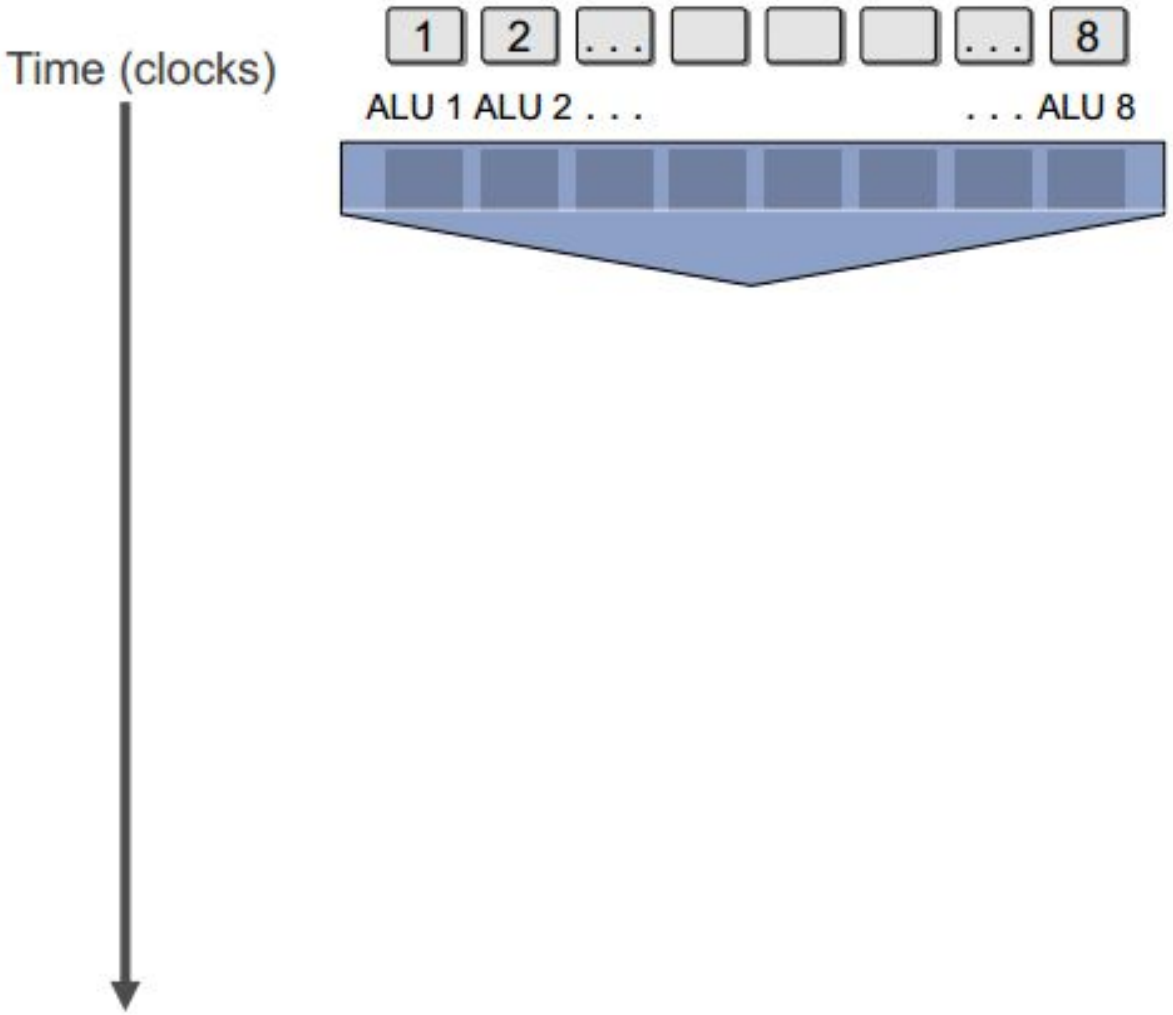


, 16 simultaneous instruction streams

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

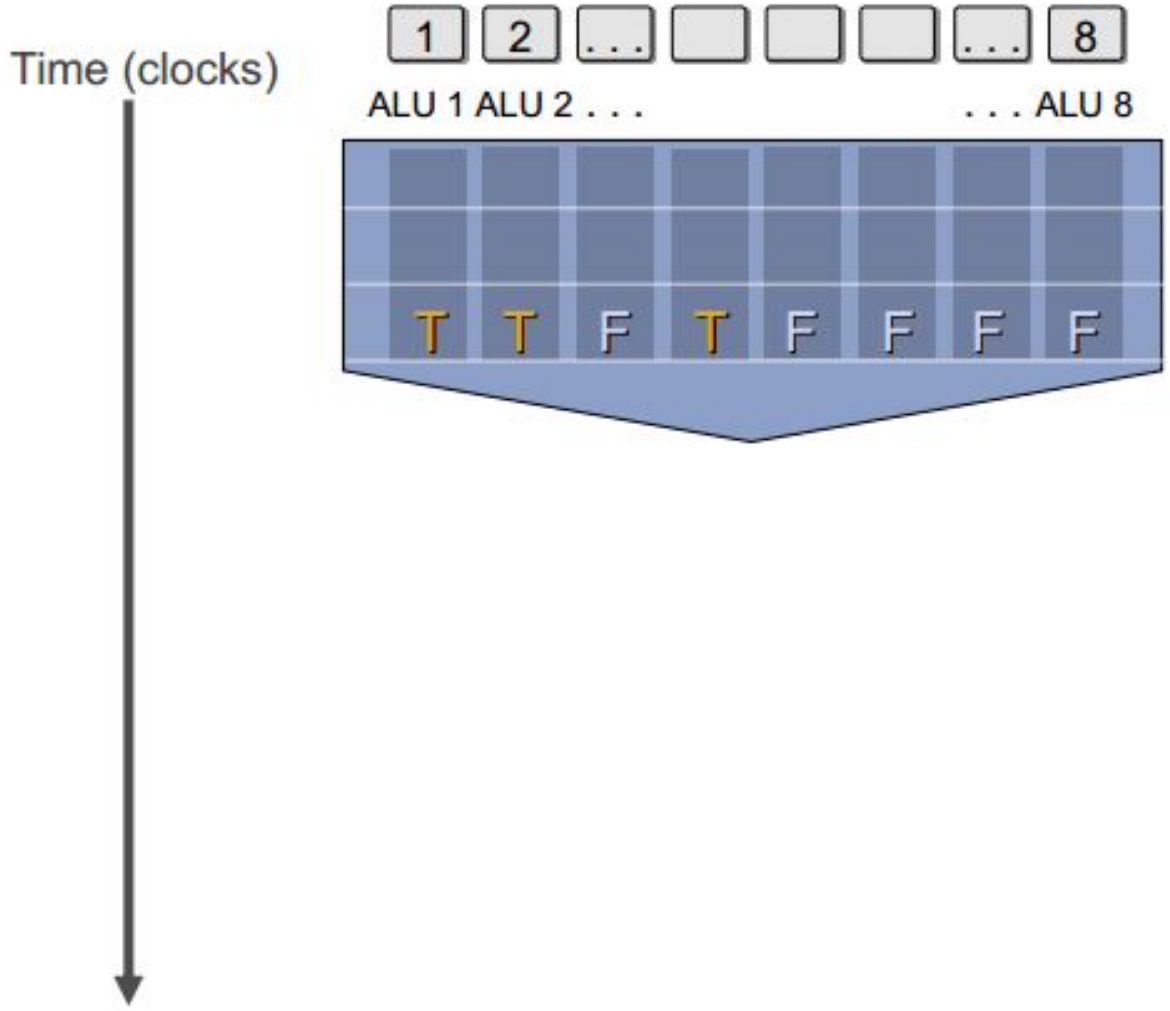


But what about branches?



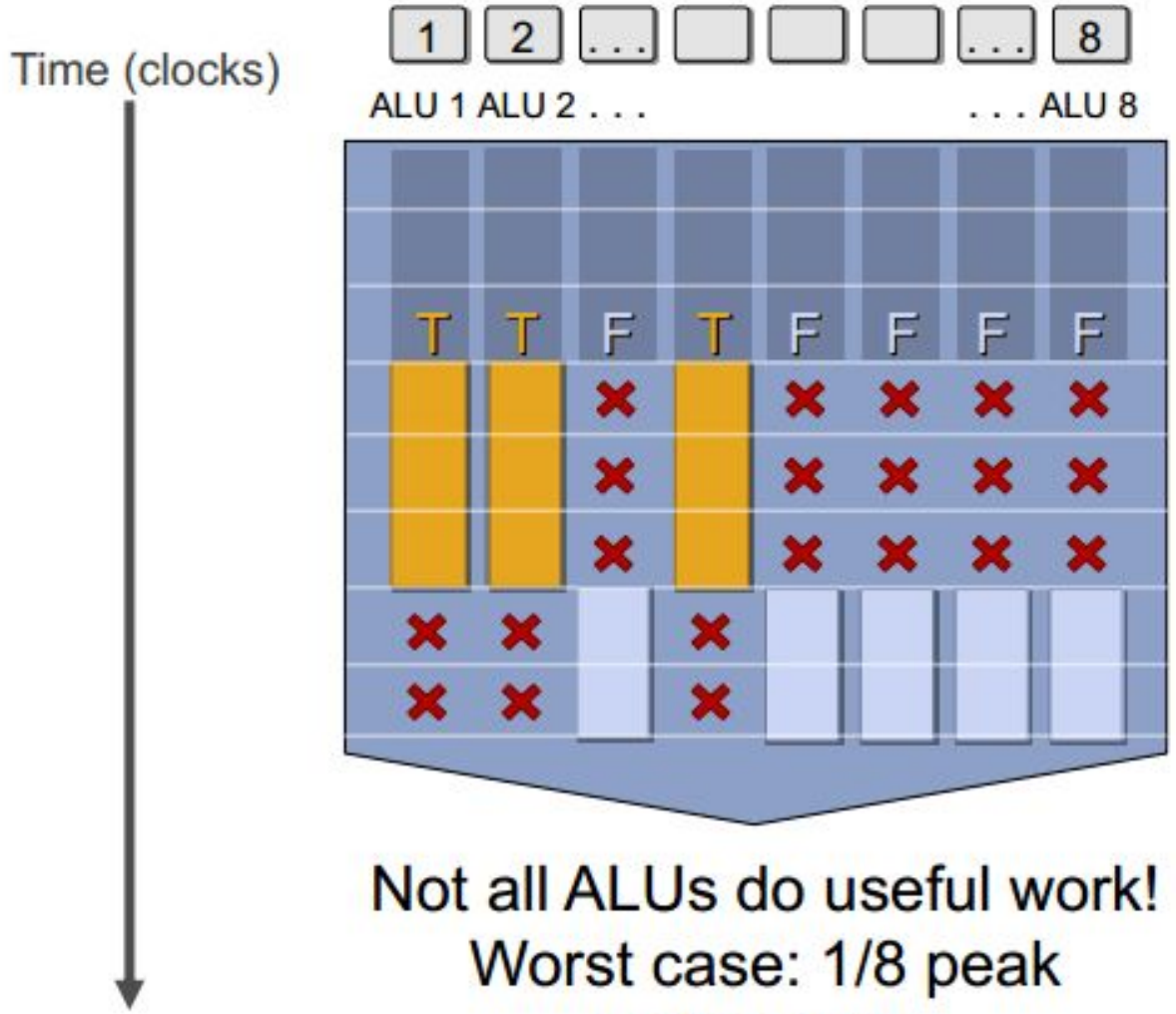

```
<unconditional
 shader code>

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?




```
<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?



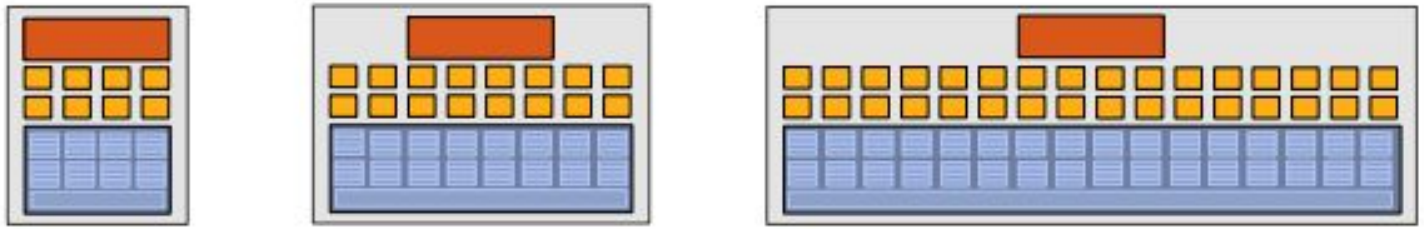

```
<unconditional
 shader code>

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

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.

Stalls!

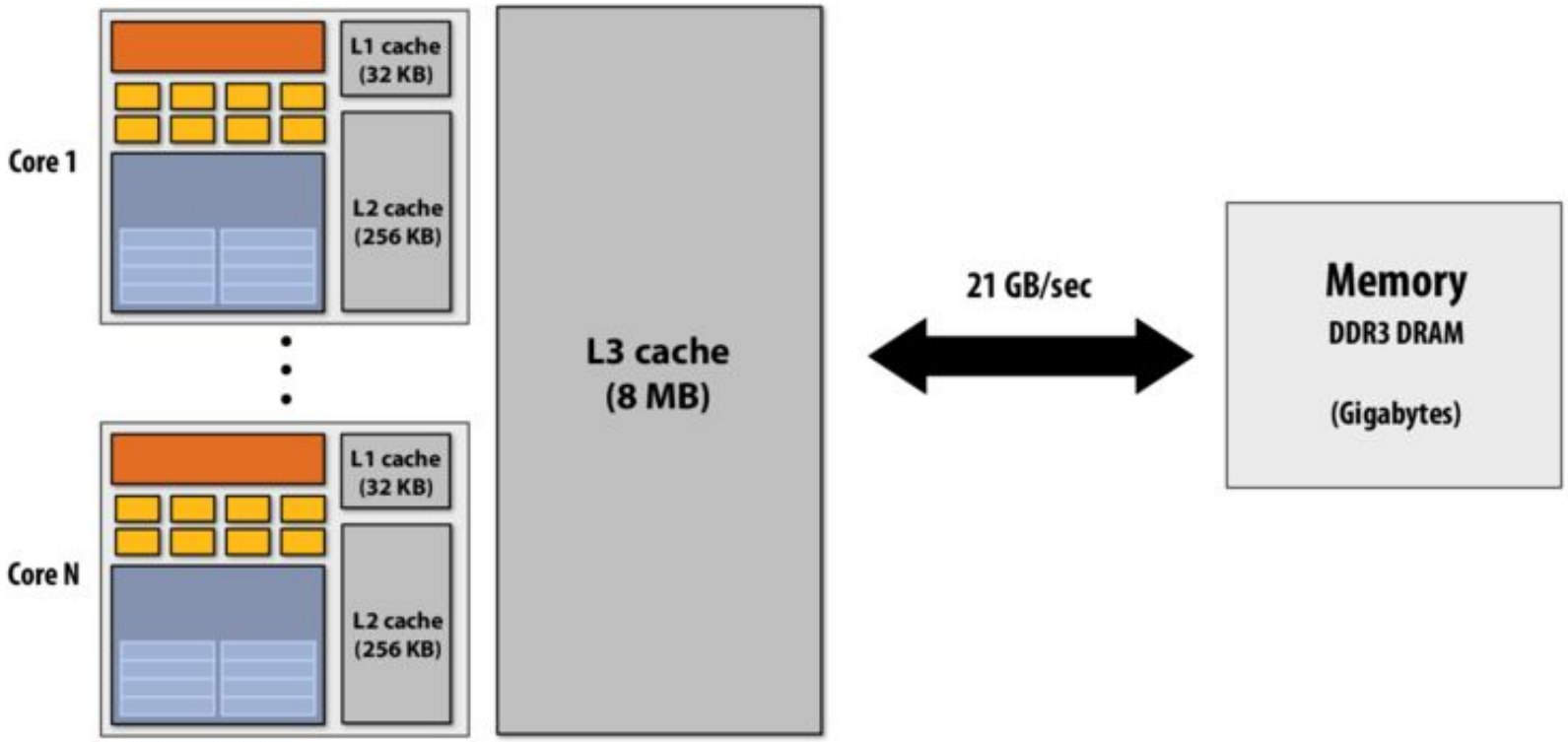
Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

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

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

Caches reduce length of stalls (reduce latency)

Processors run efficiently when data is resident in caches
(caches reduce memory access latency, also provide high bandwidth to CPU)

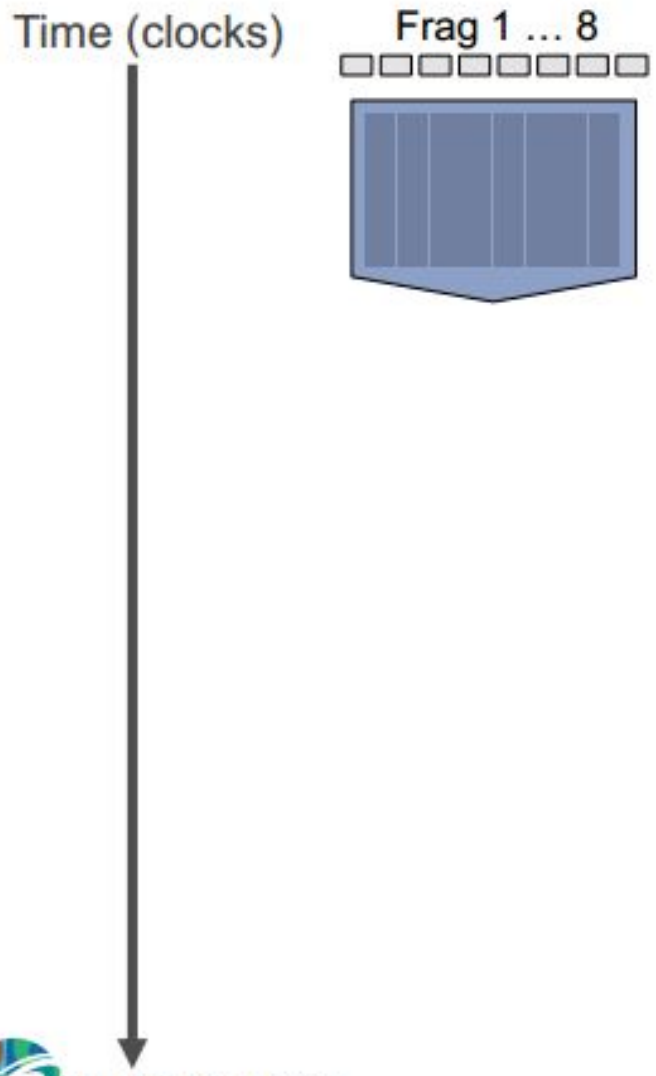


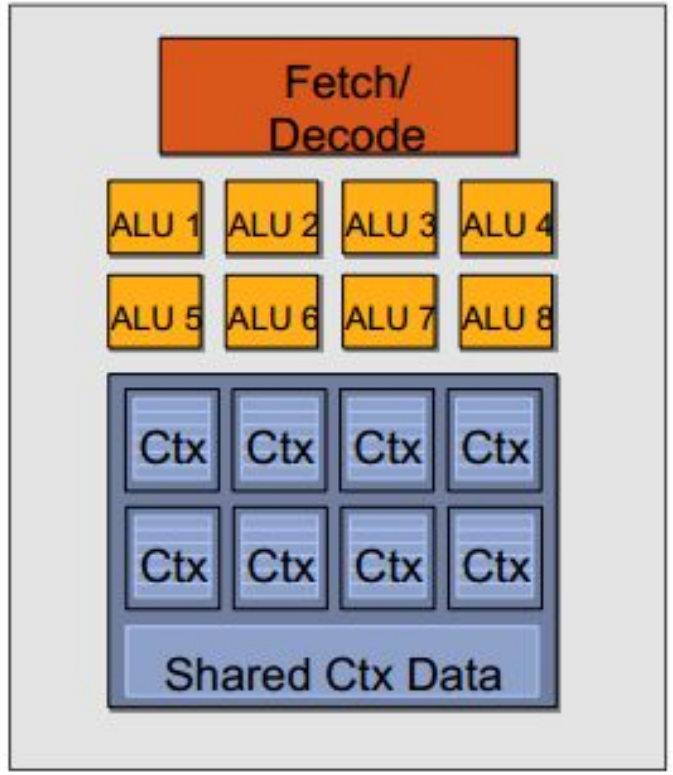
But we have LOTS of independent fragments.

Idea #3:

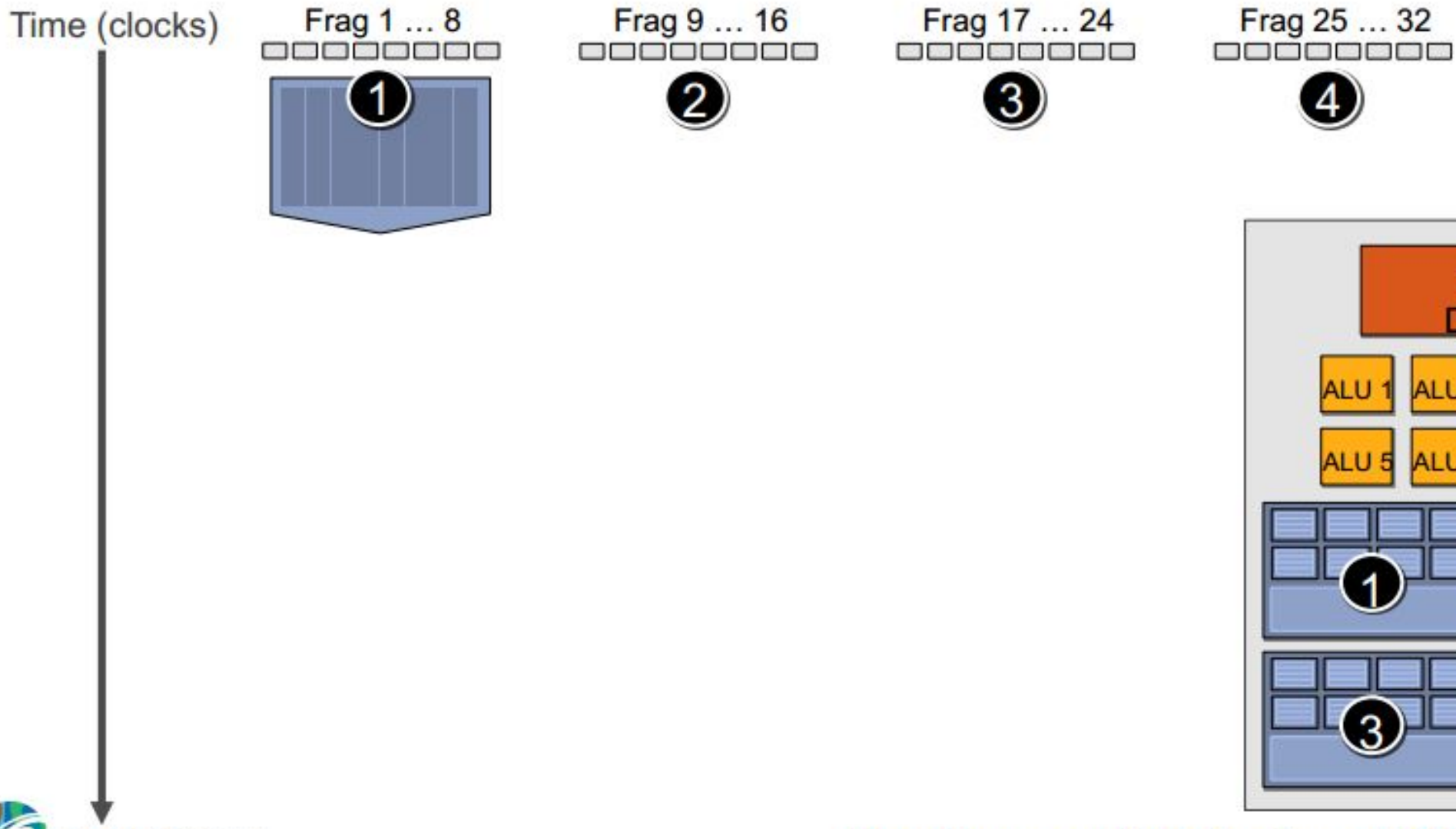
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

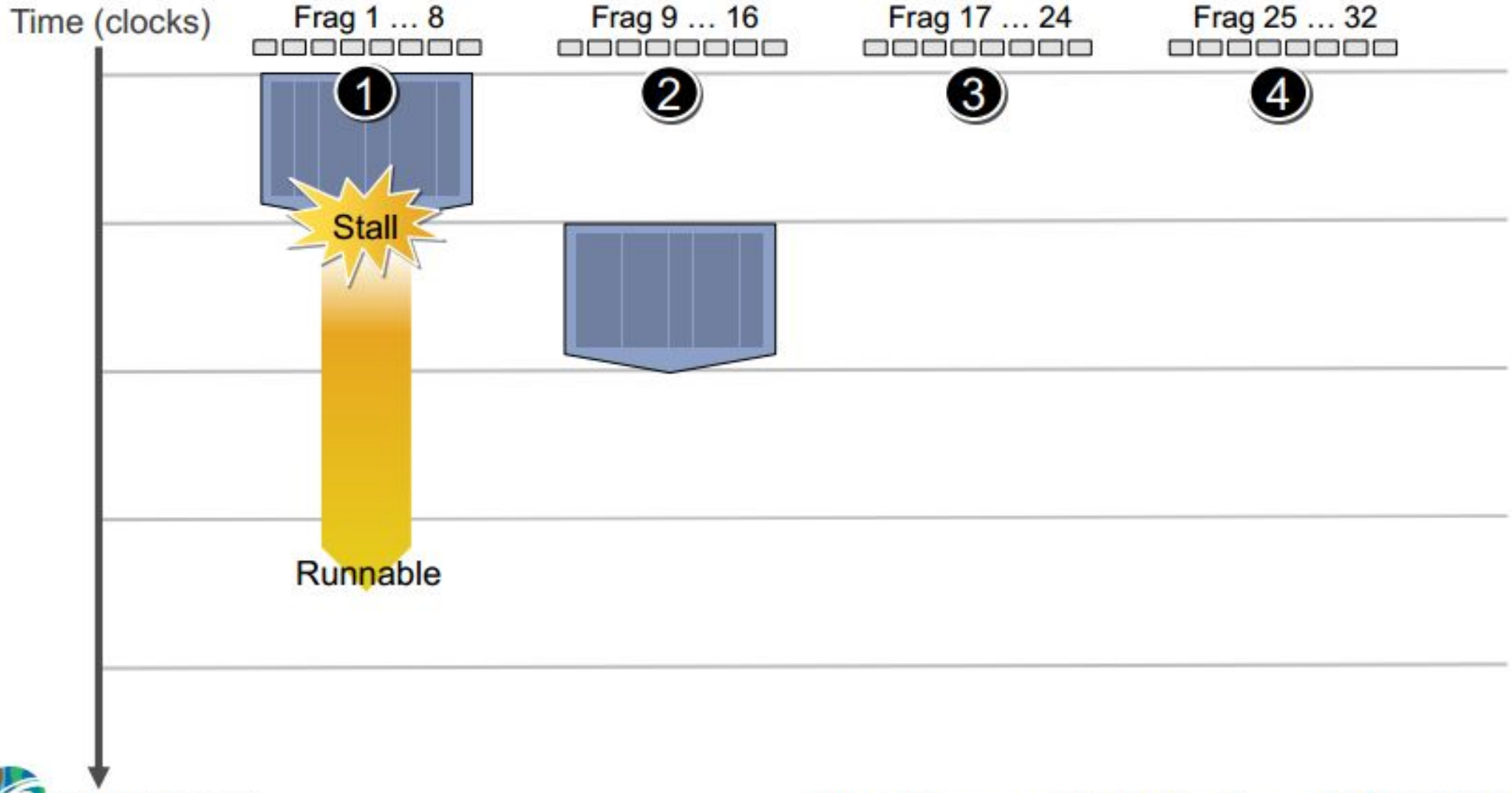




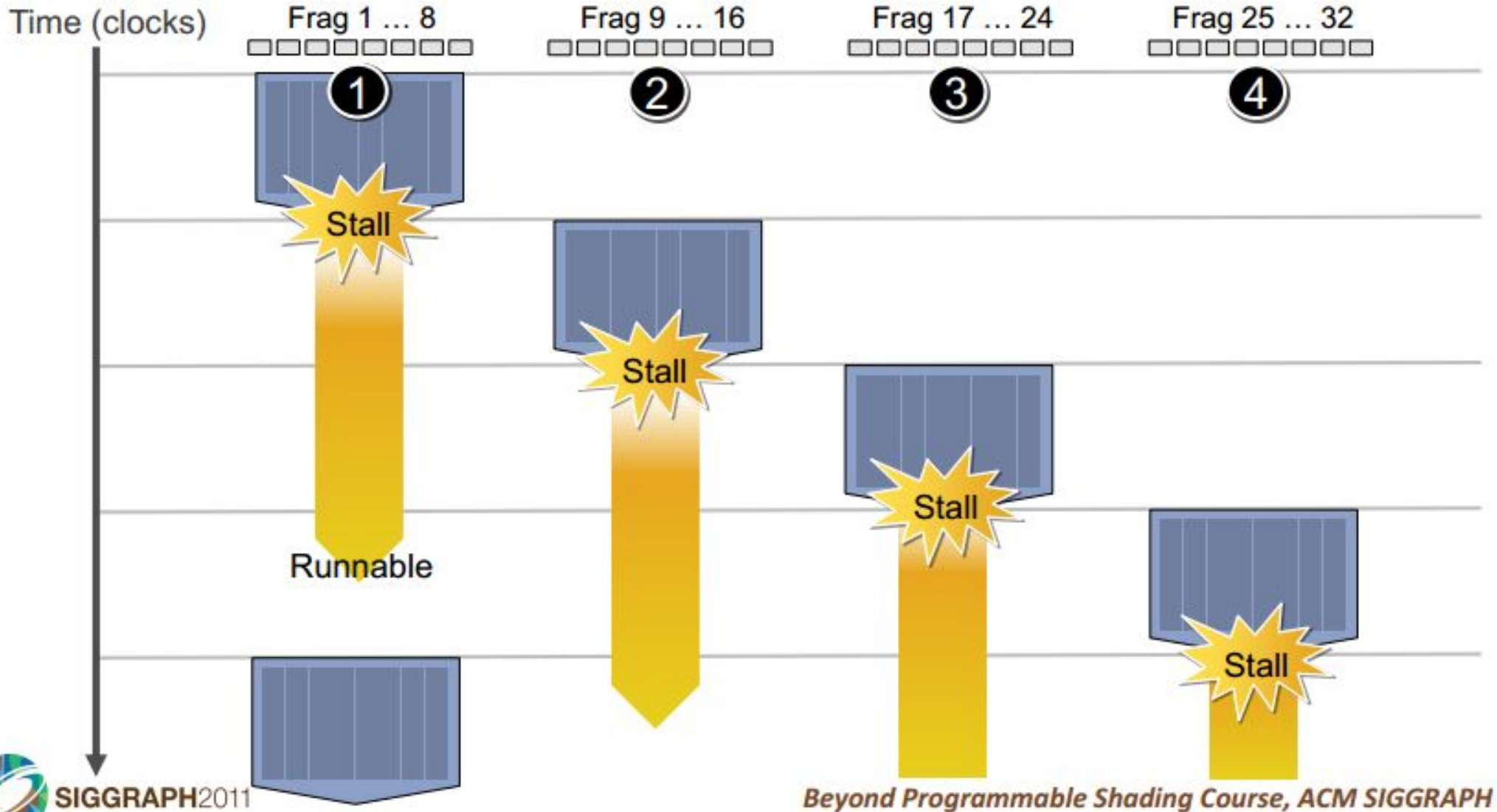
Hiding shader stalls



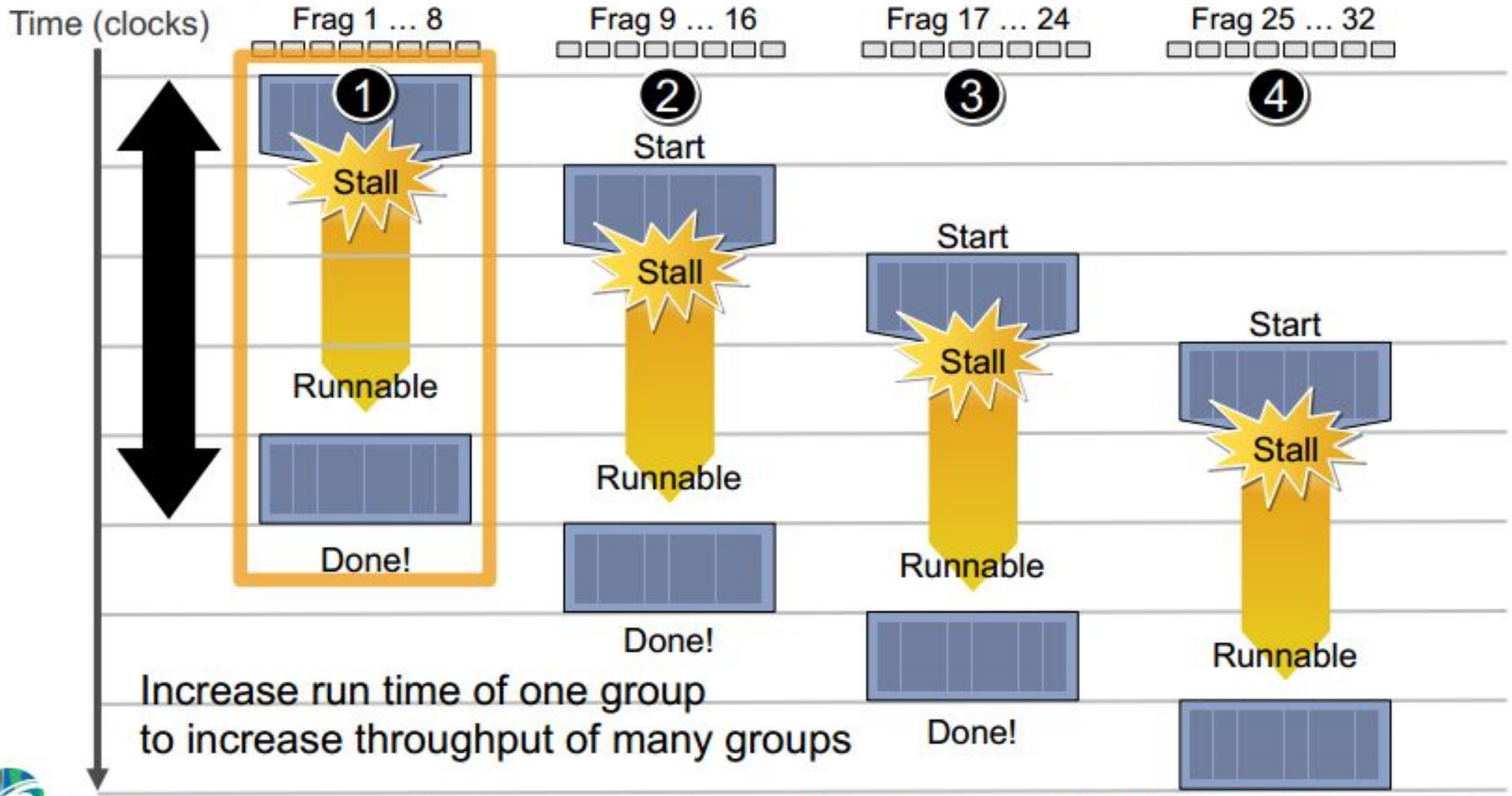
Hiding shader stalls



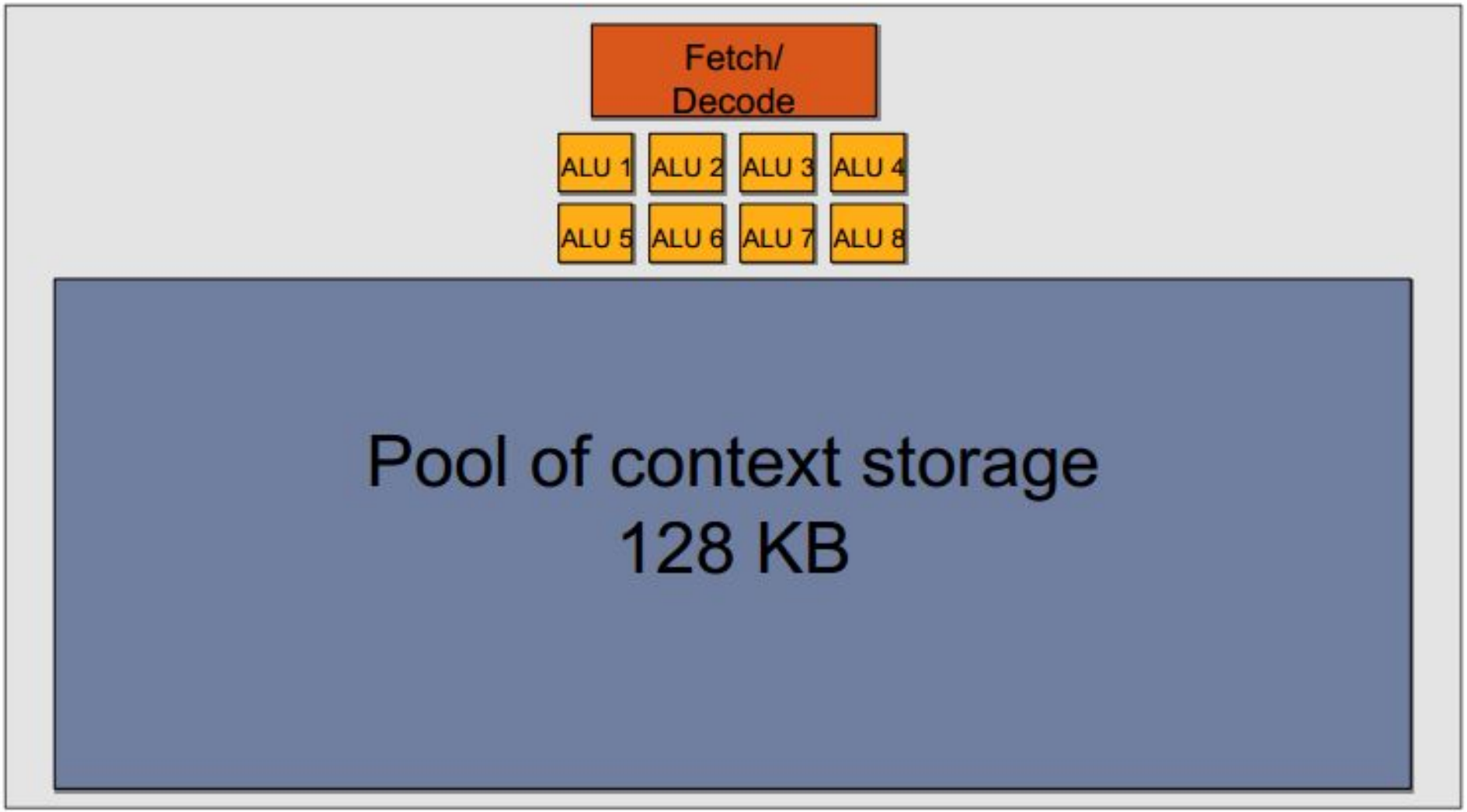
Hiding shader stalls



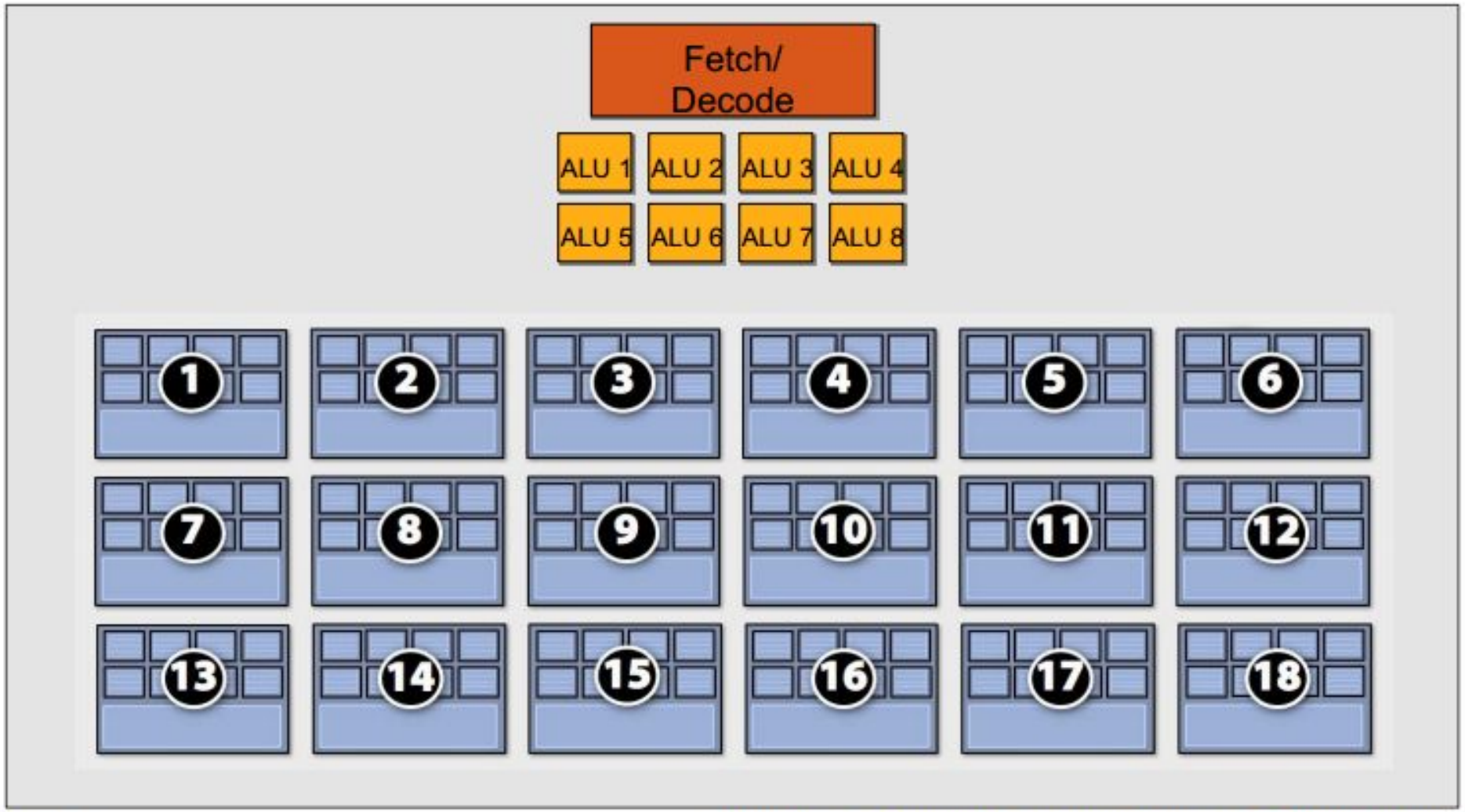
Throughput!


Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Stall
Start
Stall
Start
Stall
Runnable
Runnable
Runnable
Runnable
Done!
Done!
Done!
Increase run time of one group
to increase throughput of many groups

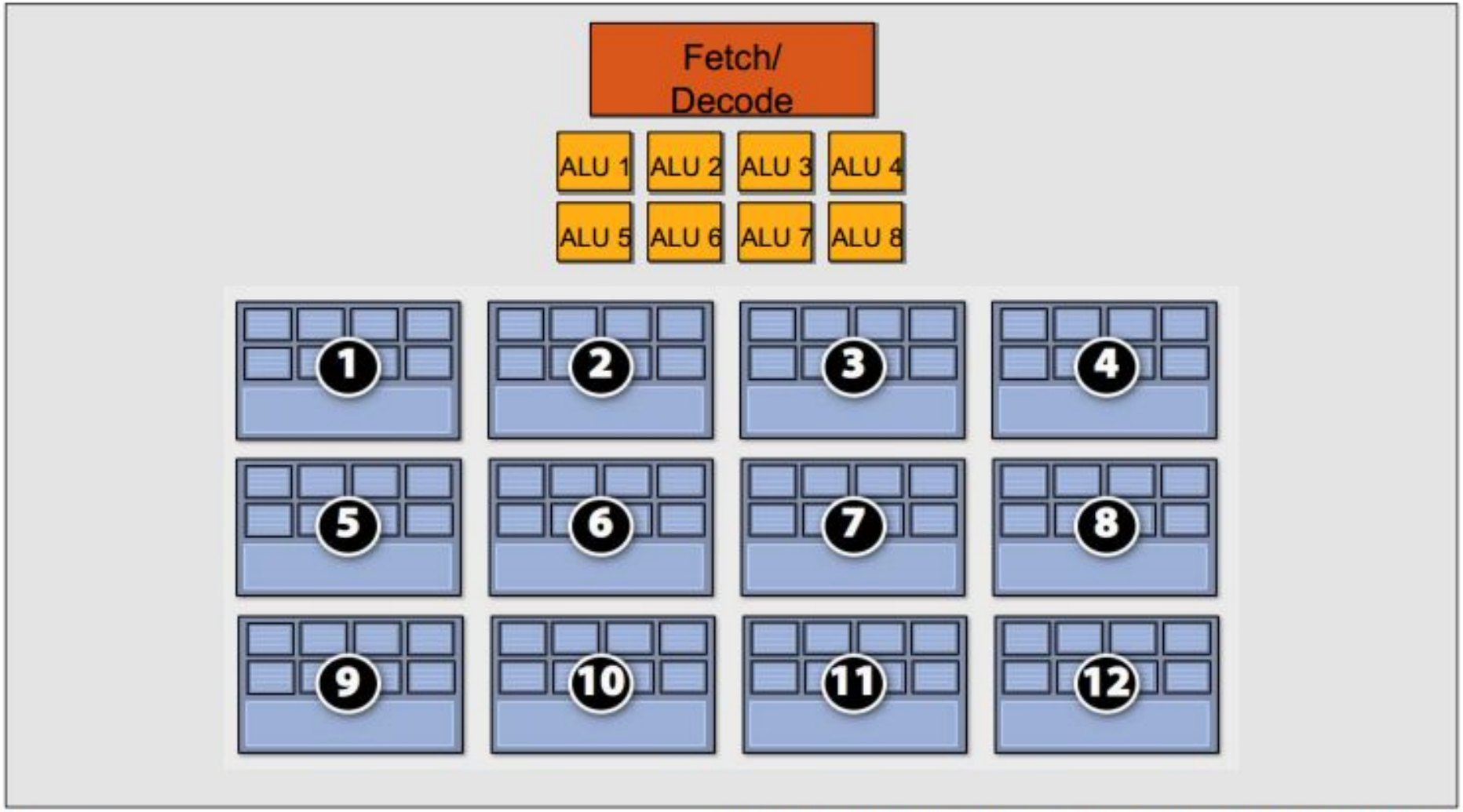
Storing contexts



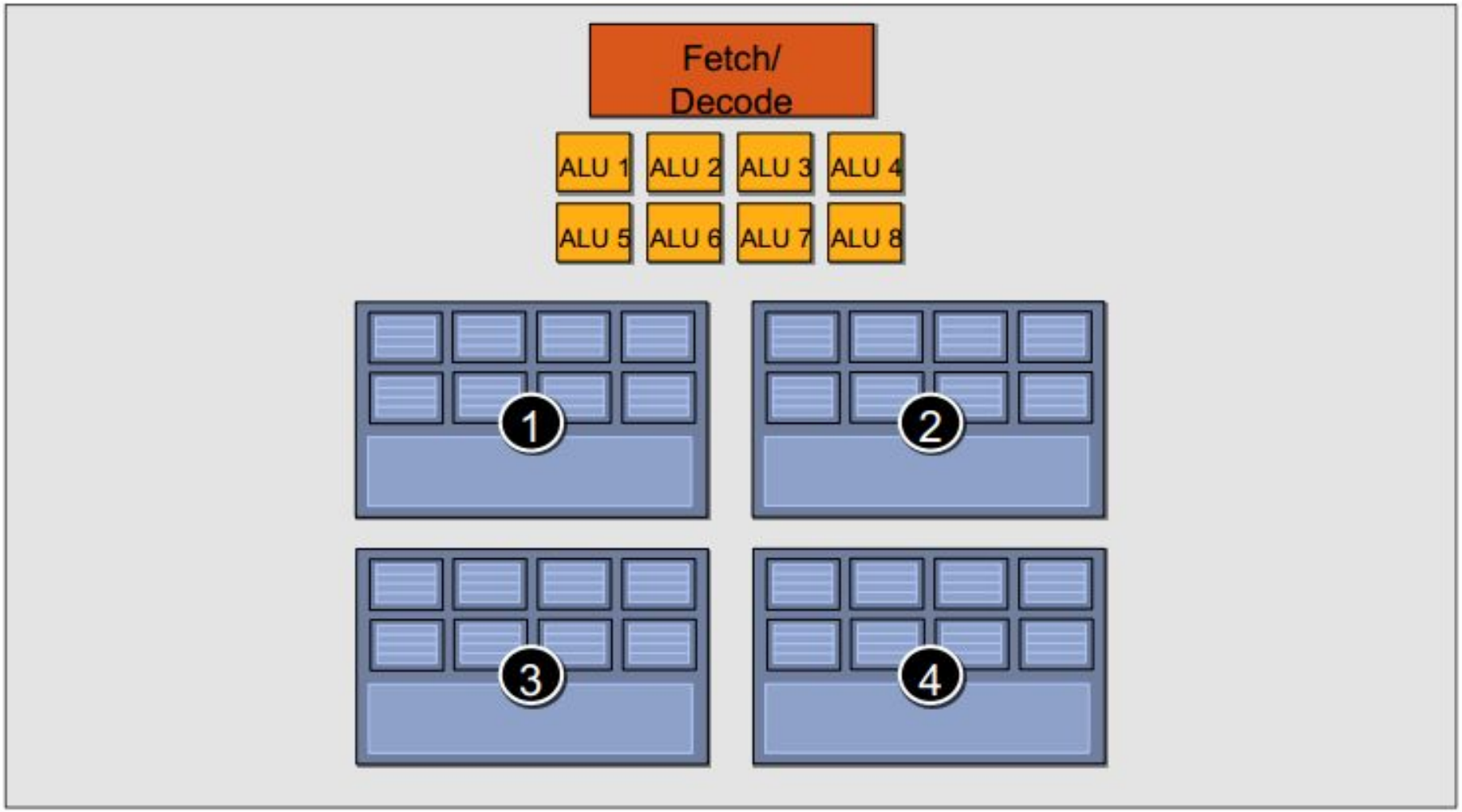
Eighteen small contexts (maximal latency hiding)



Twelve medium contexts



Four large contexts (low latency hiding ability)



Example chip

16 cores

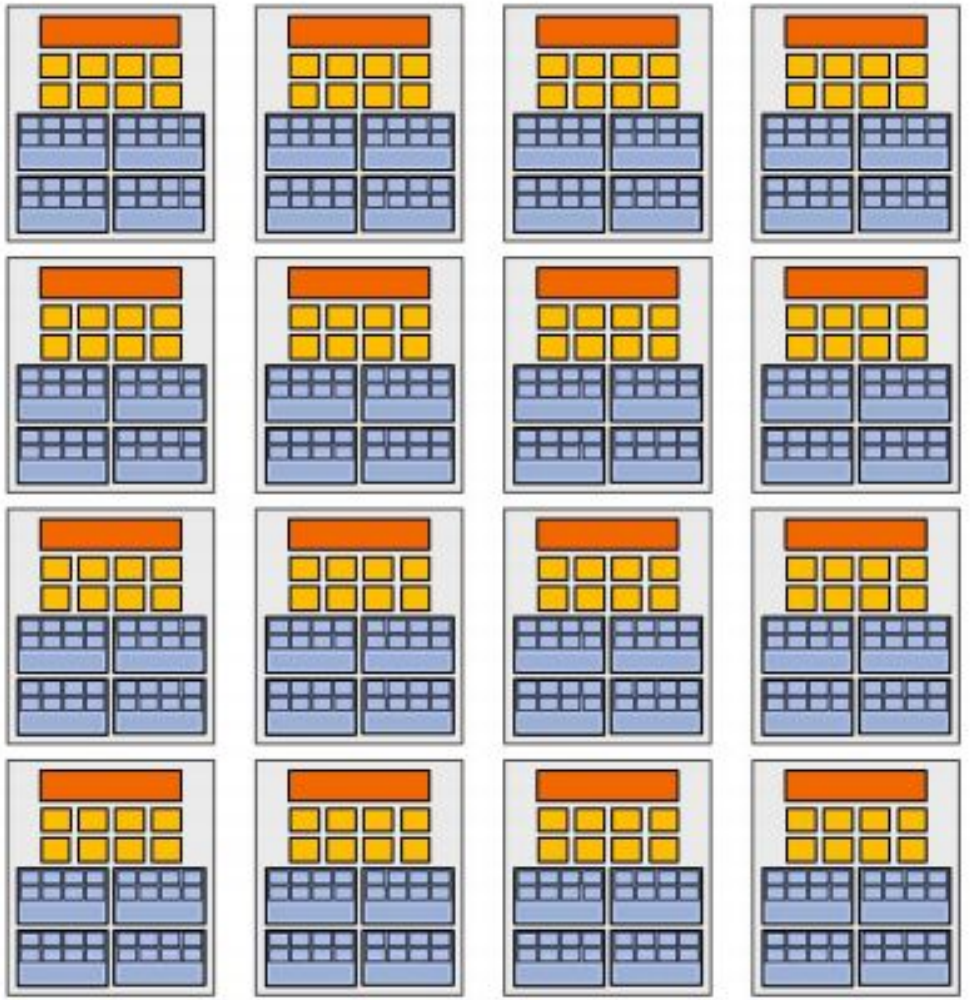
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

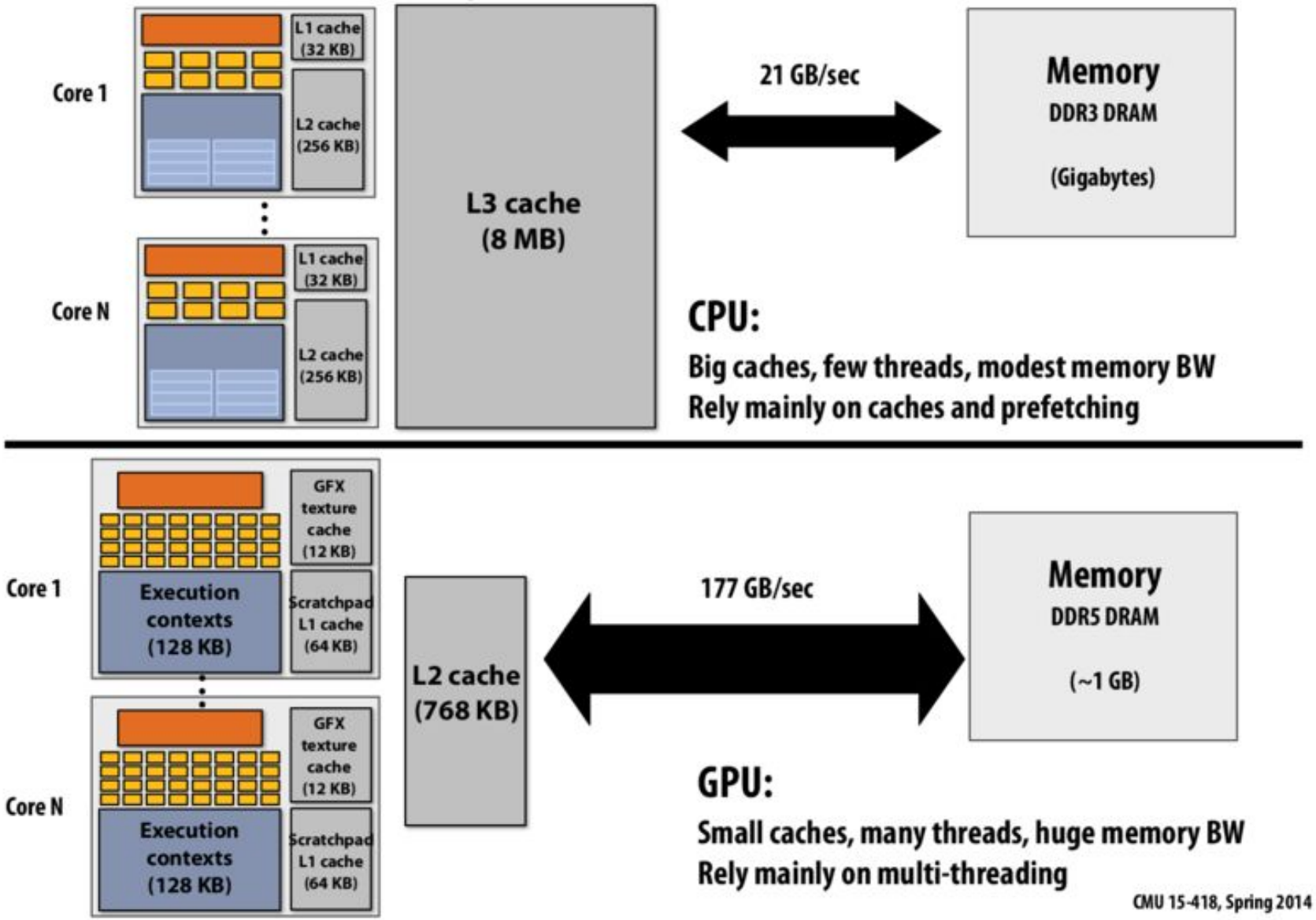
64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



CPU vs. GPU memory hierarchies



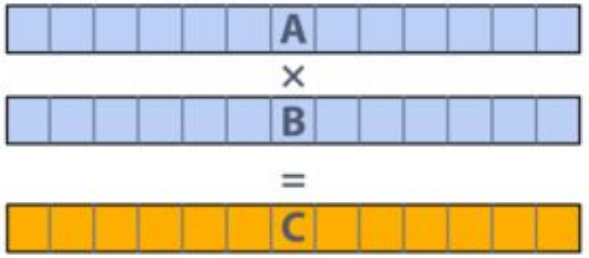
CMU 15-418, Spring 2014

Thought experiment

Task: element-wise multiplication of two vectors A and B

Assume vectors contain millions of elements

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



Three memory operations (12 bytes) for every MUL

NVIDIA GTX 480 GPU can do 480 MULs per clock (@ 1.4 GHz)

Need ~7.5 TB/sec of bandwidth to keep functional units busy (only have 177 GB/sec)

~ 2% efficiency… but 8x faster than CPU!

(3GHz Core i7 quad-core CPU connected to 25 GB/sec memory bus will exhibit similarly low efficiency on this computation)

Bandwidth limited!

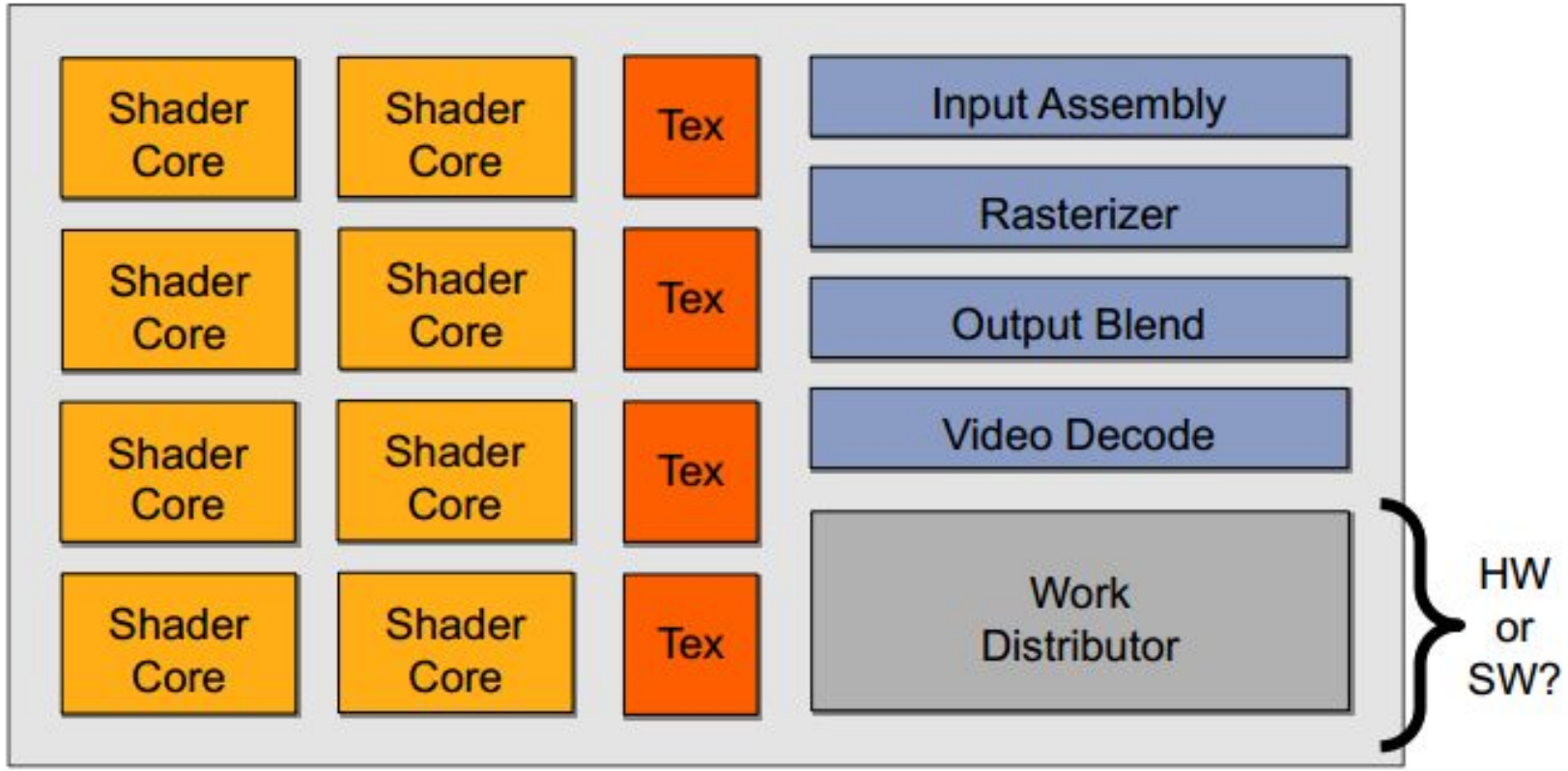
If processors request data at too high a rate, the memory system cannot keep up.

No amount of latency hiding helps this.

Overcoming bandwidth limits are a common challenge for application developers on throughput-optimized systems.

What's in a GPU?

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



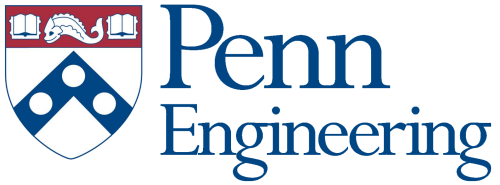
Summary: three key ideas

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

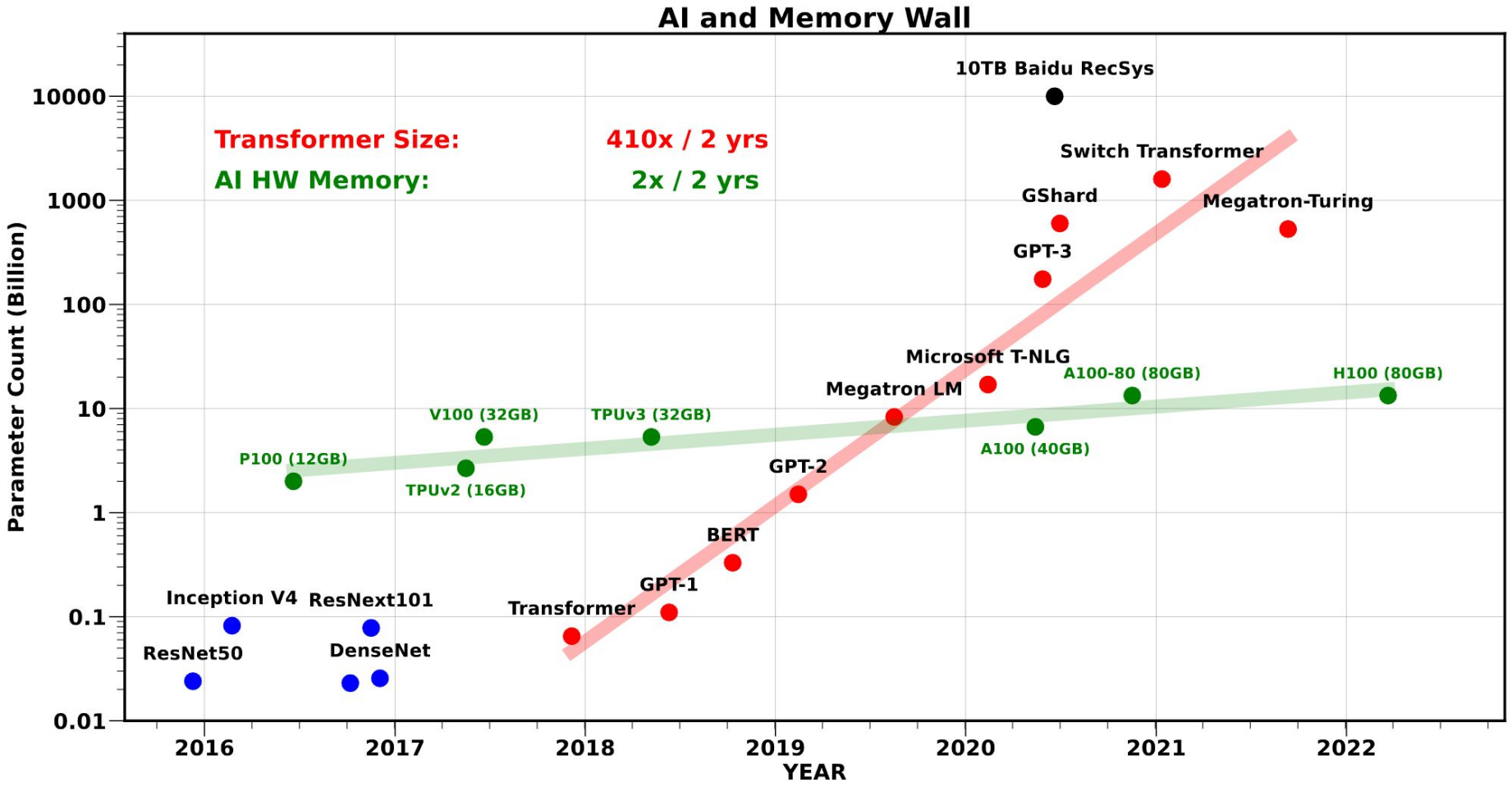
2. Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

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

GPU Architecture in the Age of AI



AI & The Memory Wall



Penn Engineering

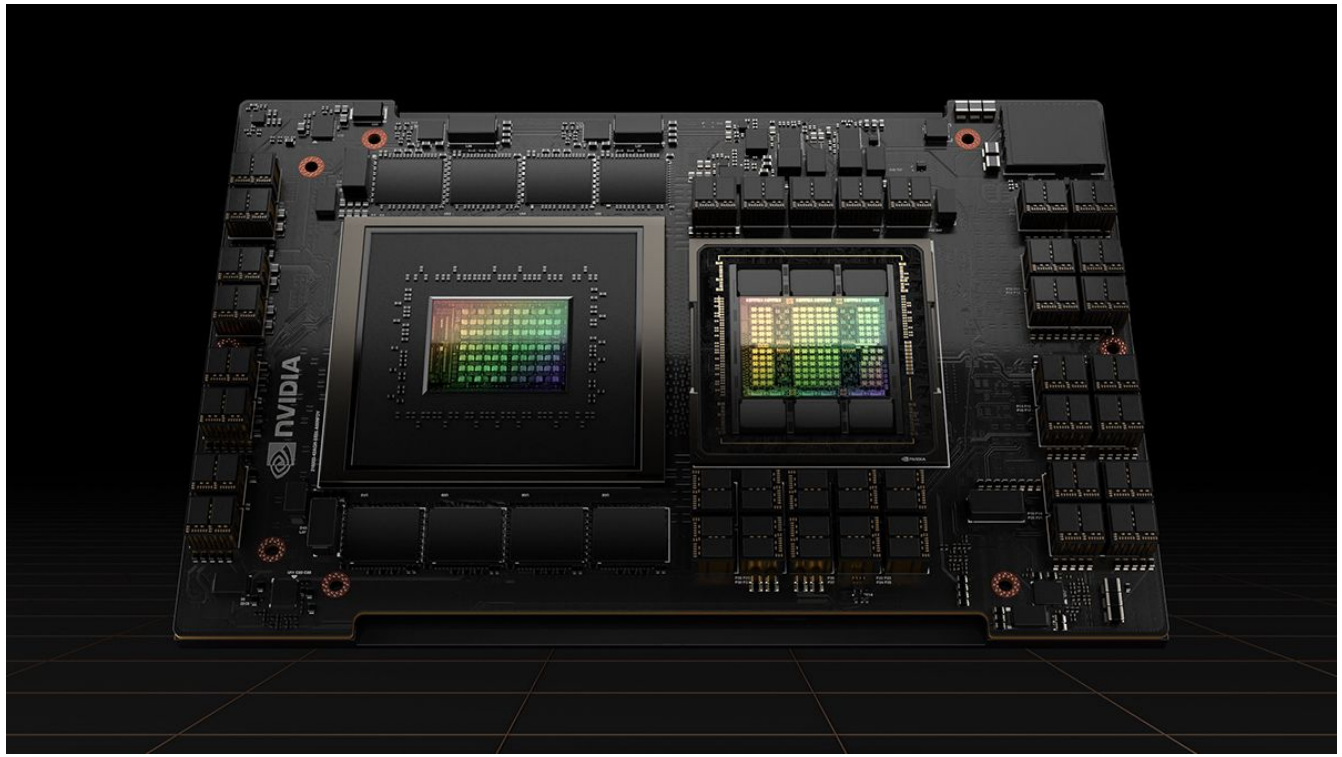
Image from: AI and Memory Wall. (This blogpost has been written in… | by Amir Gholami | riselab |Medium

Compute-in-Memory

- Deep learning requires a large amount of memory
  - Store the whole model for training and inference
- But we can't just keep increasing the memory per chip
  - Physical limitations
- **Keep lots of memory close to the compute unit**
  - So the data transfer is fastest possible

NVIDIA H100 GPU

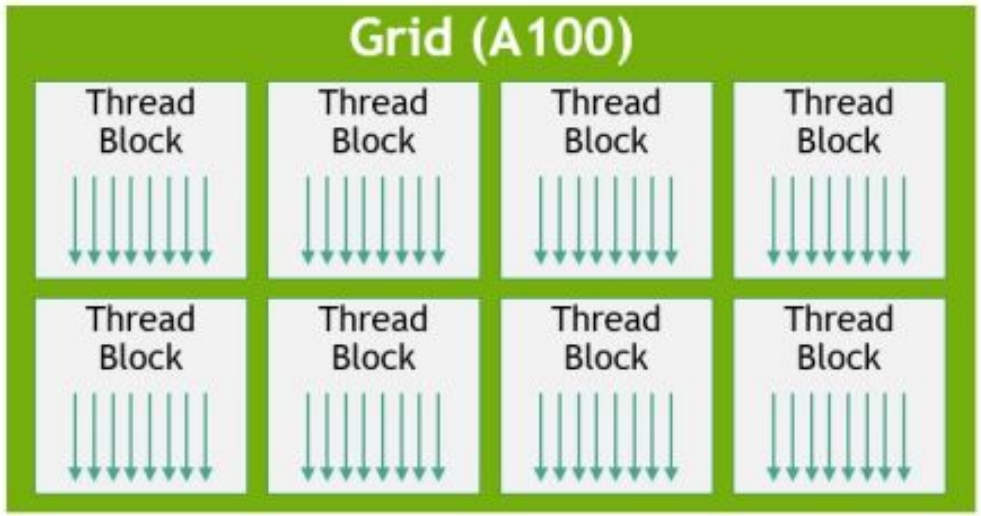
- **80GB** VRAM
- 80 SMs
- 18,432 CUDA Cores
- 67 TFLOPS (FP32)
- 4 PFLOPS (FP8)

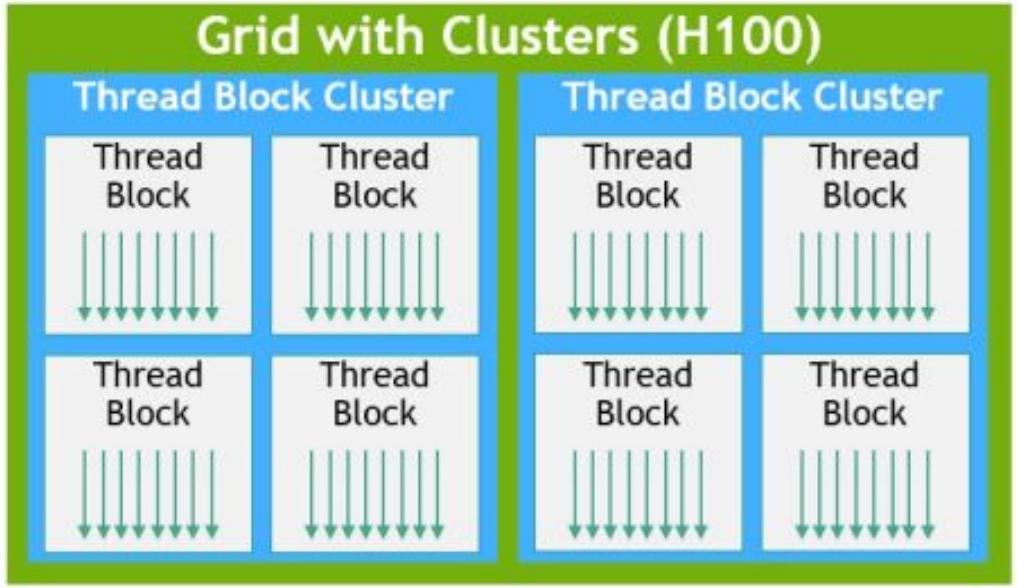


Source: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

GPU Processing Cluster

- A group of SMs in the hardware hierarchy that are always physically close together

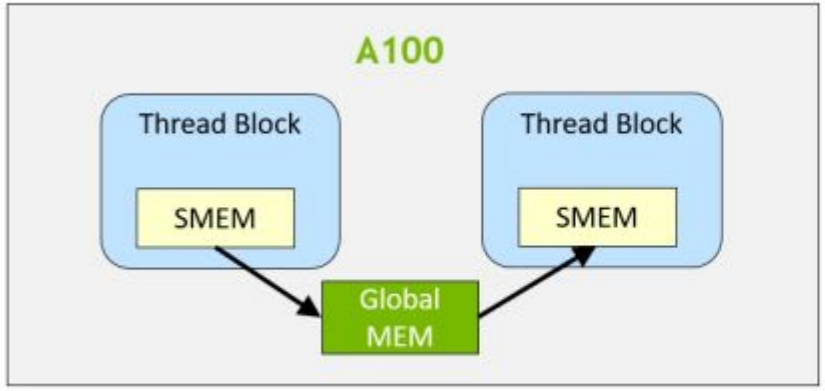


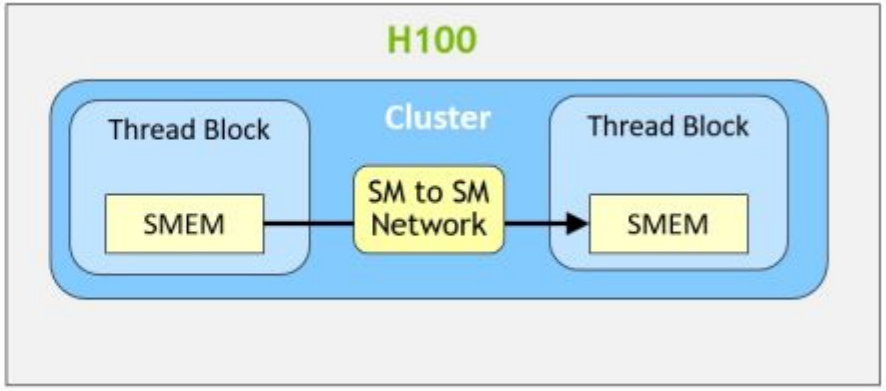


Source: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

Distributed Shared Memory

- SM-to-SM network within a cluster
- No longer need to transfer data to global memory

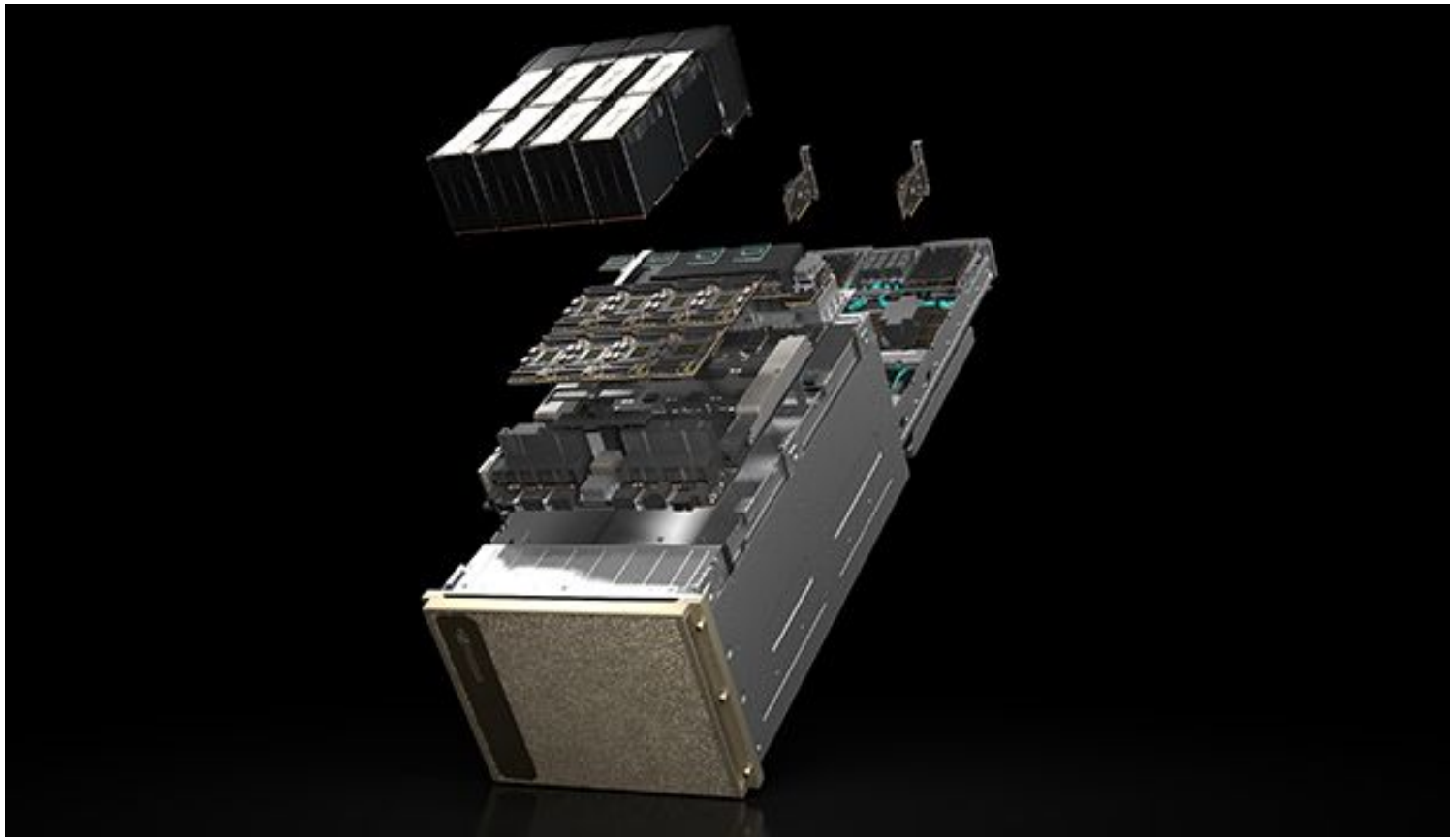




Source: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

NVIDIA DGX H100

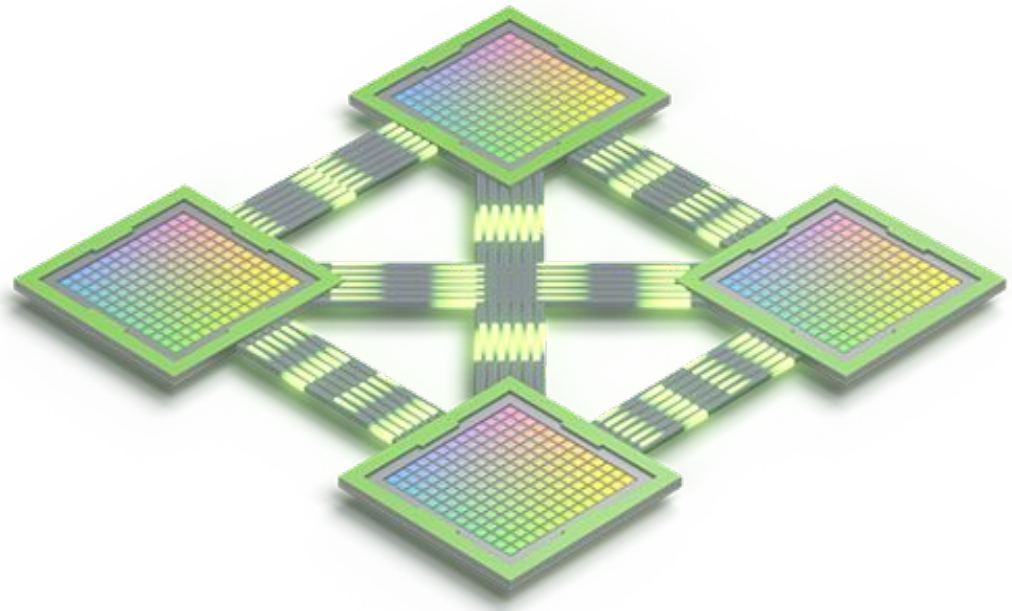
- 8 x H100 GPUs
- 640GB RAM
- 32 PFLOP (FP8)

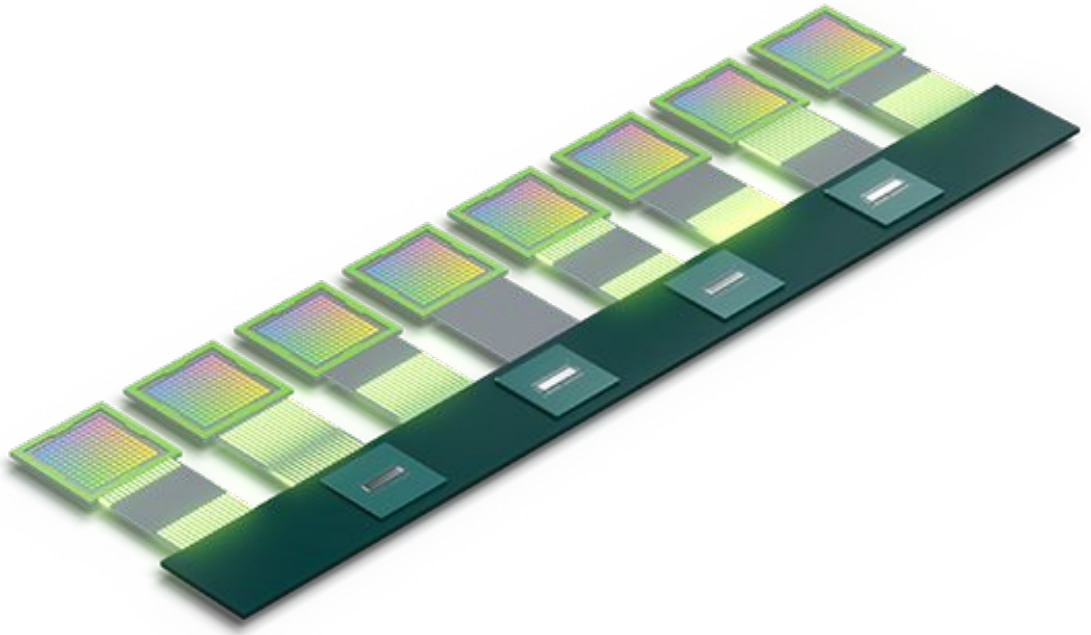


Source: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

Transferring data between GPUs

- **NVLink**: 900GB/s GPU-to-GPU throughput
  - 7x faster than PCIe Gen5
- **NVSwitch**: 7.2 TB/s total throughput

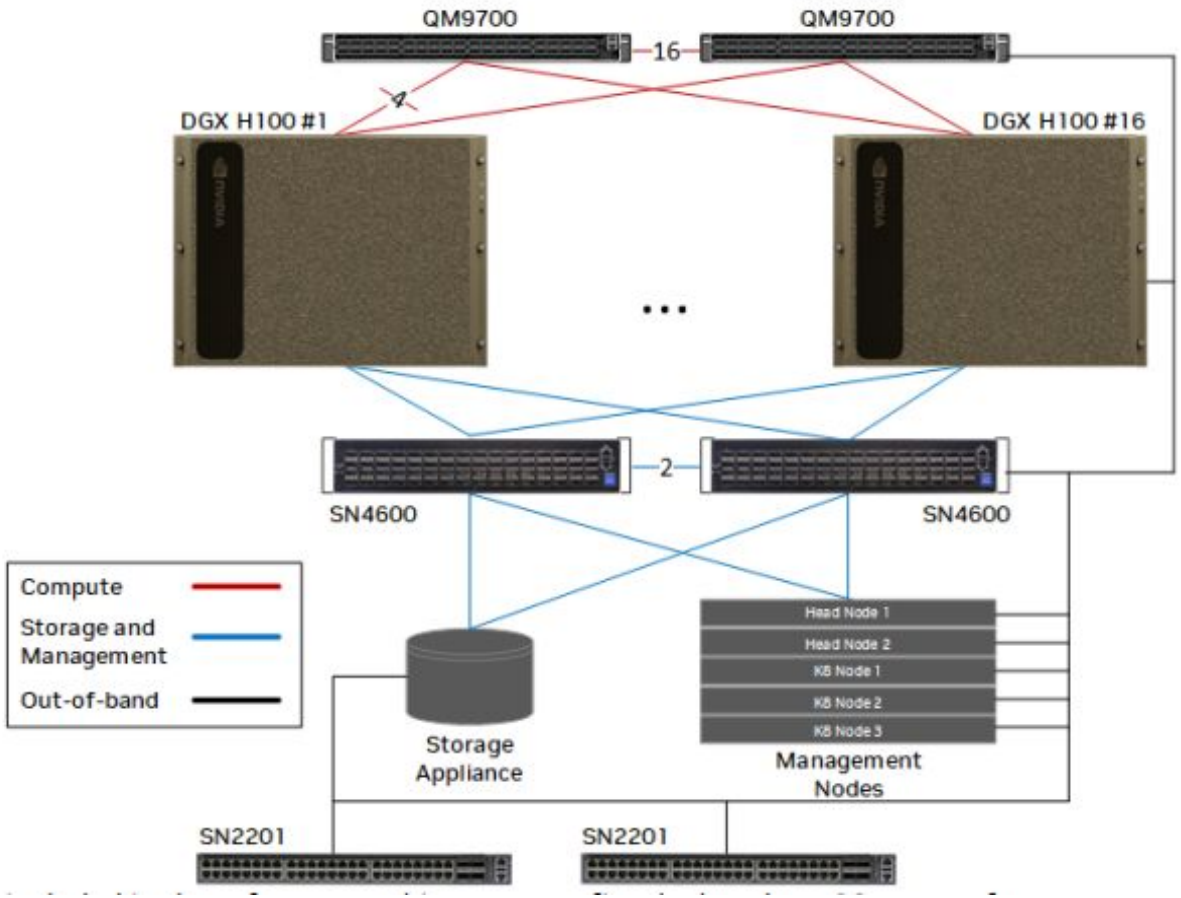




Source: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

NVIDIA DGX PODs

Figure 20. DGX BasePOD with up to 16 systems—DGX H100 NDR200



Penn Engineering

Source: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

Transferring data between servers

- InfiniBand Networking:
  - 400 GB/s data transfer

NVIDIA DGX SuperPODs

- **32+ DGX Systems**
- **1 exaFLOP** (FP8)

Source: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

The Best Part...

CUDA scales it automatically