CMPE 110: Computer Architecture

Week 11

GPU Memory Hierarchy

Jishen Zhao (http://users.soe.ucsc.edu/~jzhao/)

[Adapted in part from Jose Renau, Joe Devietti, Kayvon Fatahalian, Onur Mutlu, and others]

Reminders

- Midterm2 papers pickup
  - Rebecca: A-G
  - Xin: H-M
  - Aziz: N-S
  - Narendra: T-Z
- Homework 4 due on Thursday
- Review class on Friday
- Final exam
  - Dec. 7 (Wed) 12-3pm
  - DRC exam rooms
  - Overflow room (TBD)
  - This classroom (can fit rest of you)

GPU processor architecture review

- GPUs contain a collection of programmable processing cores: responsible for carrying out data-parallel stages of the graphics pipeline
- Three basic ideas
  - Remove components that help a single instruction stream fast
  - Amortize cost/complexity of managing an instruction across many ALUs
  - Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations

CPU

GPU

Each GPU core has many ALUs

Today

- Details about implementing thread interleaving
- GPU memory hierarchy

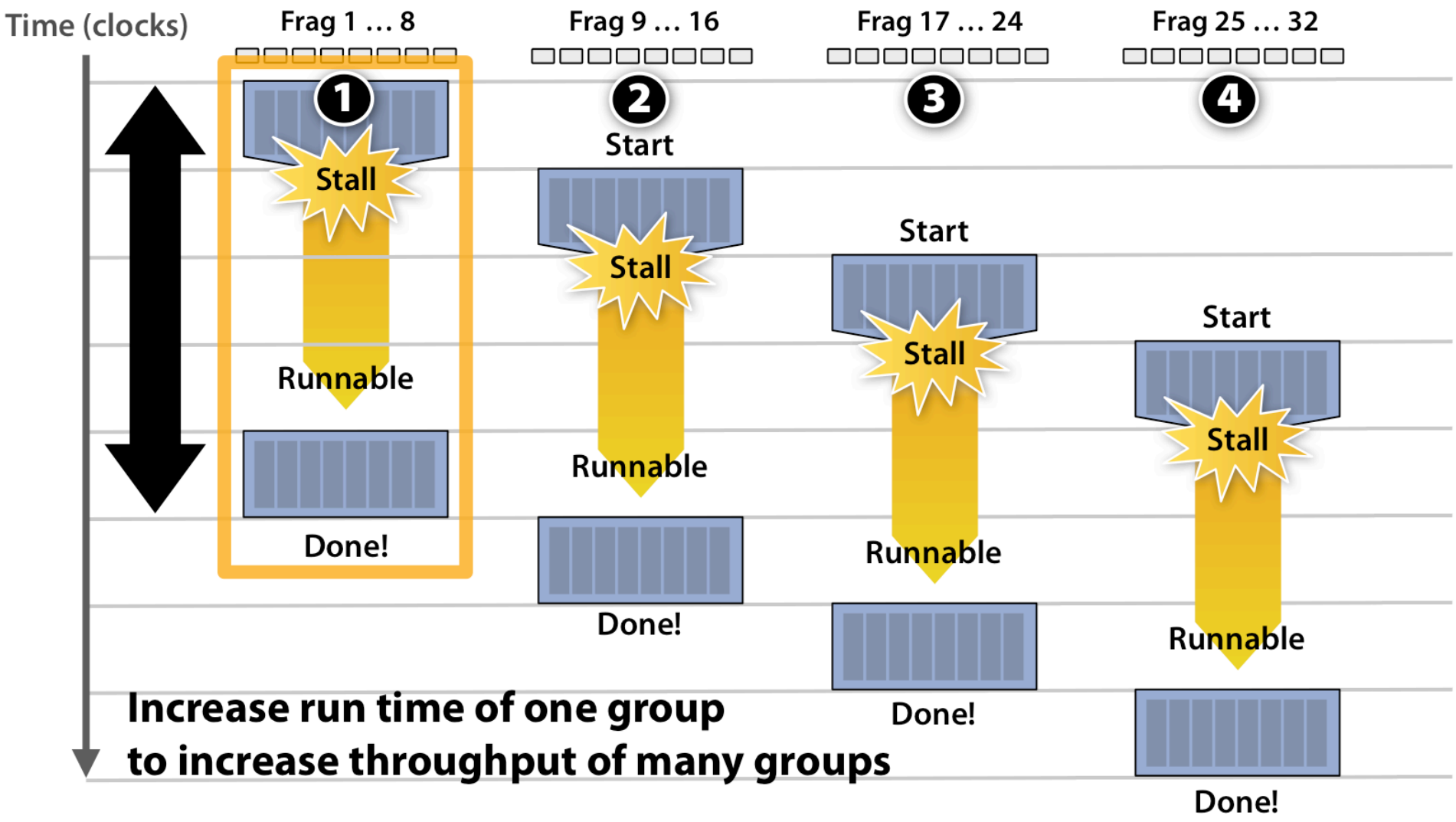
Stalls!

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

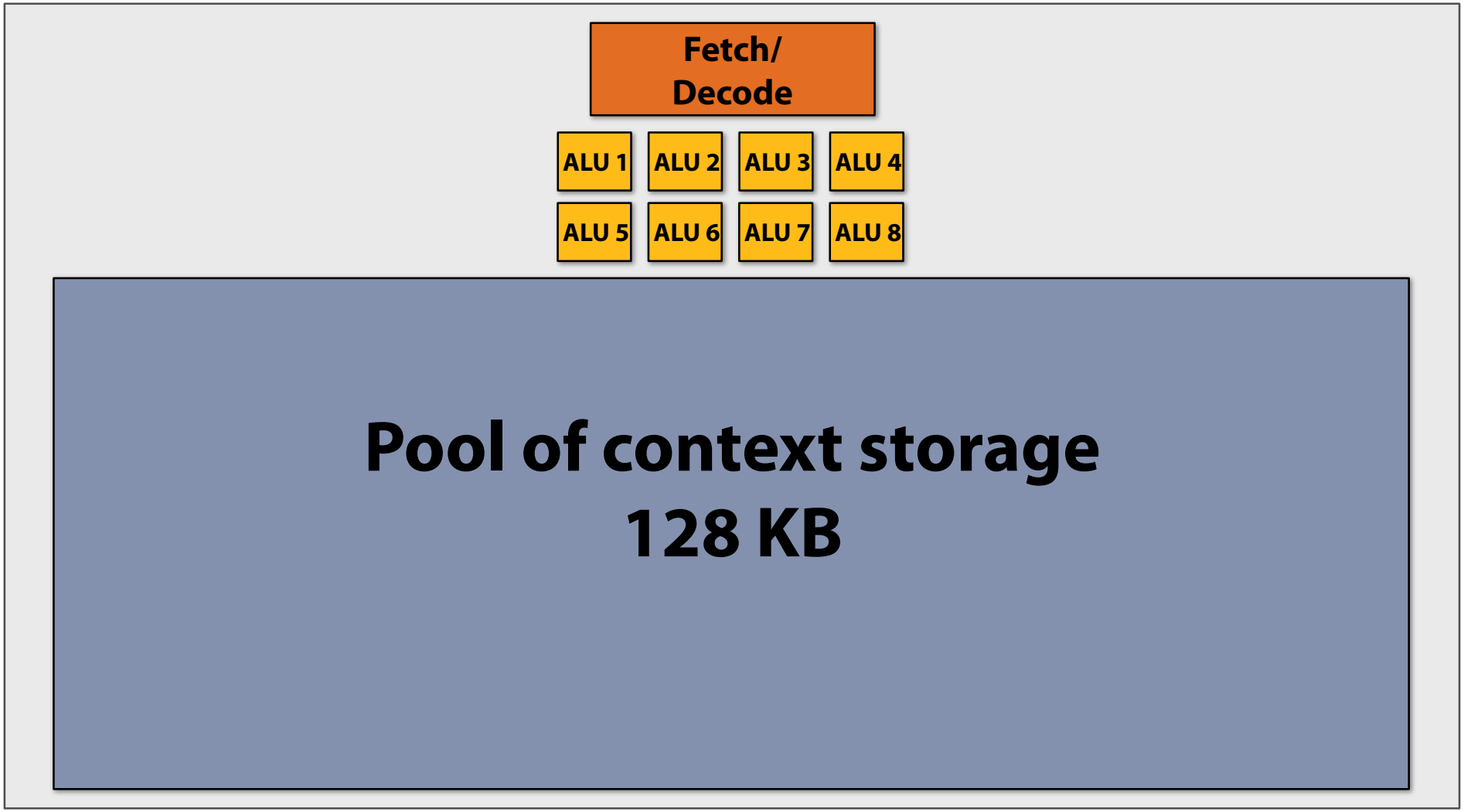
Texture access latency = 100's to 1000's of cycles

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

Throughput!

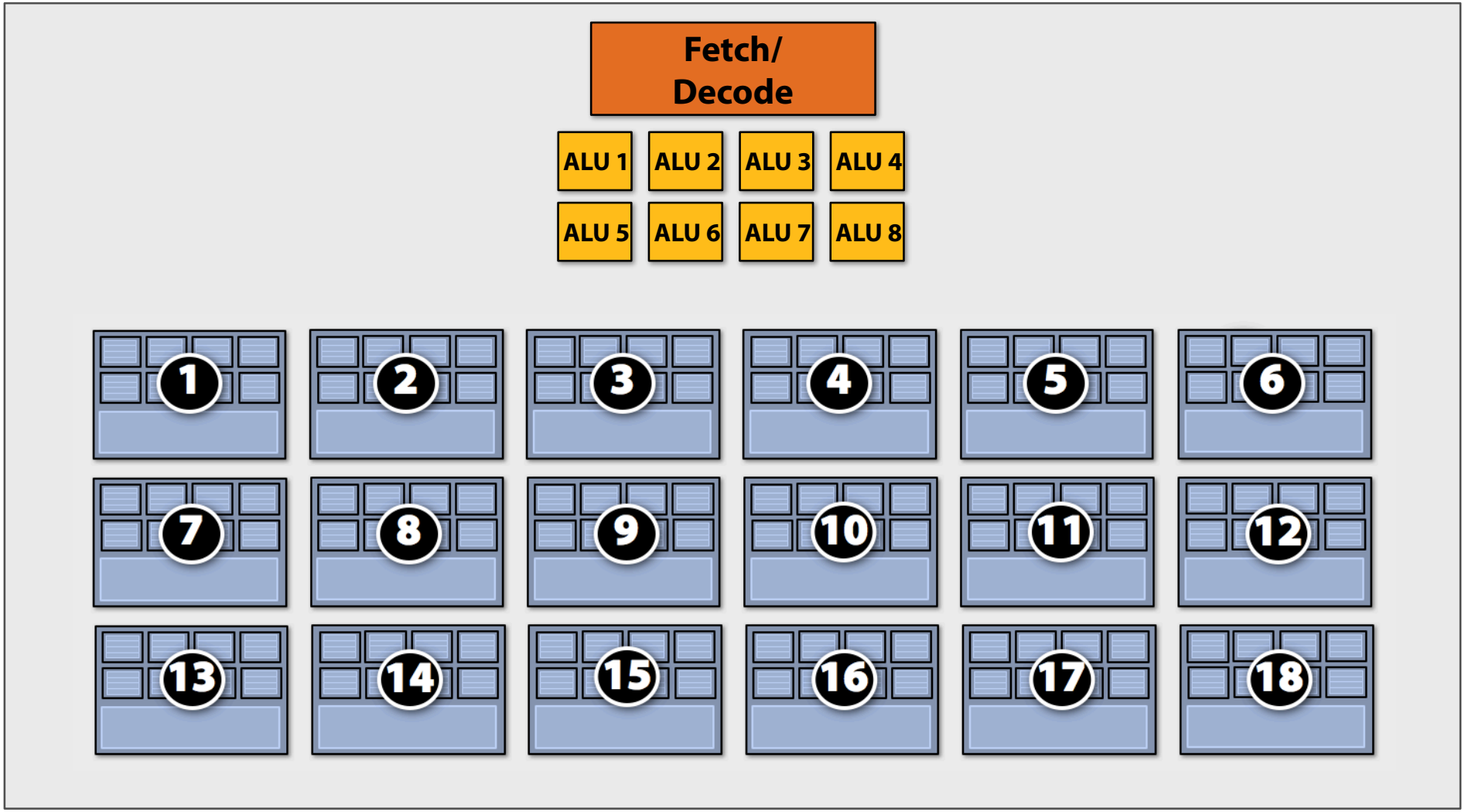

Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Runnable
Done!
Increase run time of one group
to increase throughput of many groups

Storing contexts

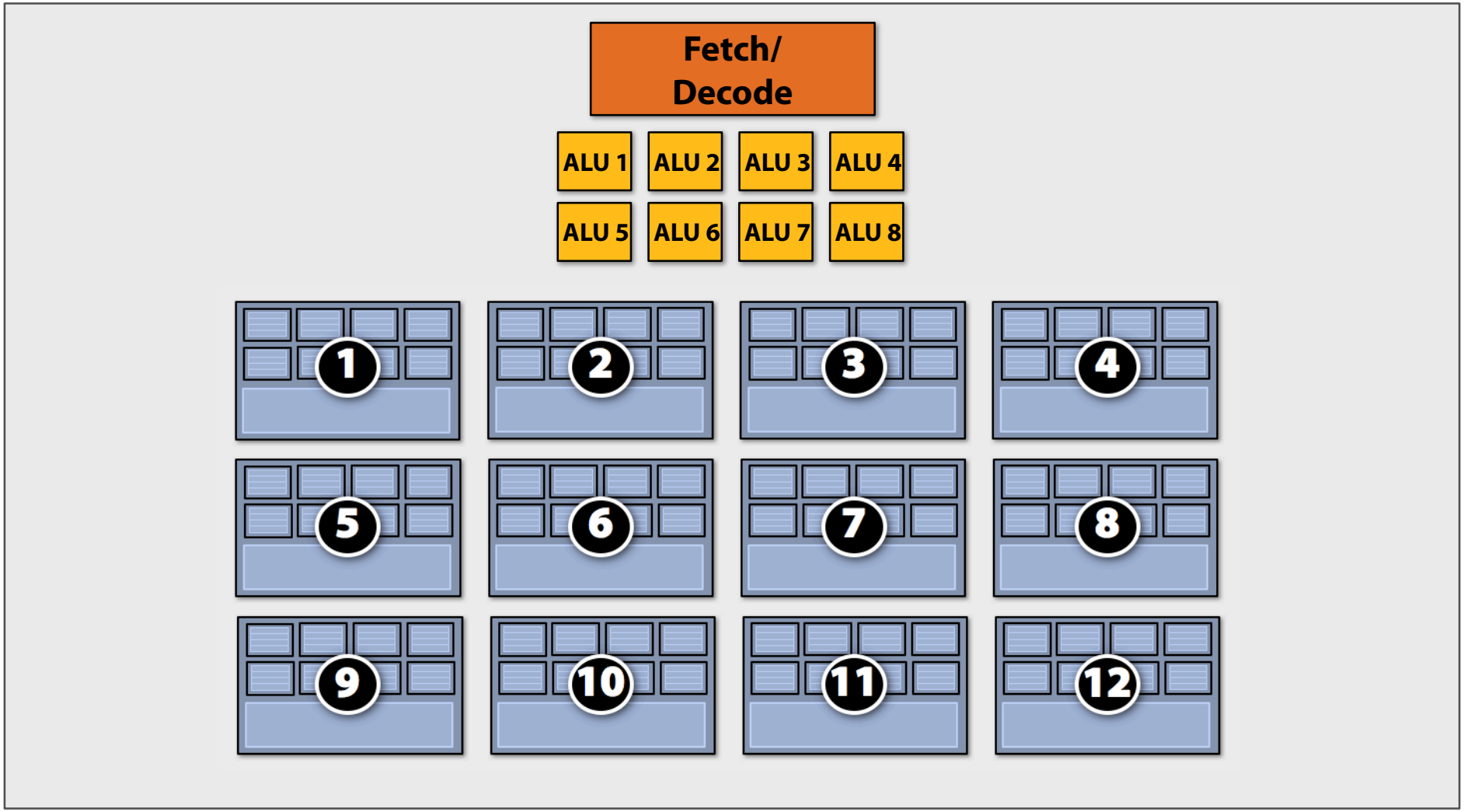


Eighteen small contexts

(maximal latency hiding)

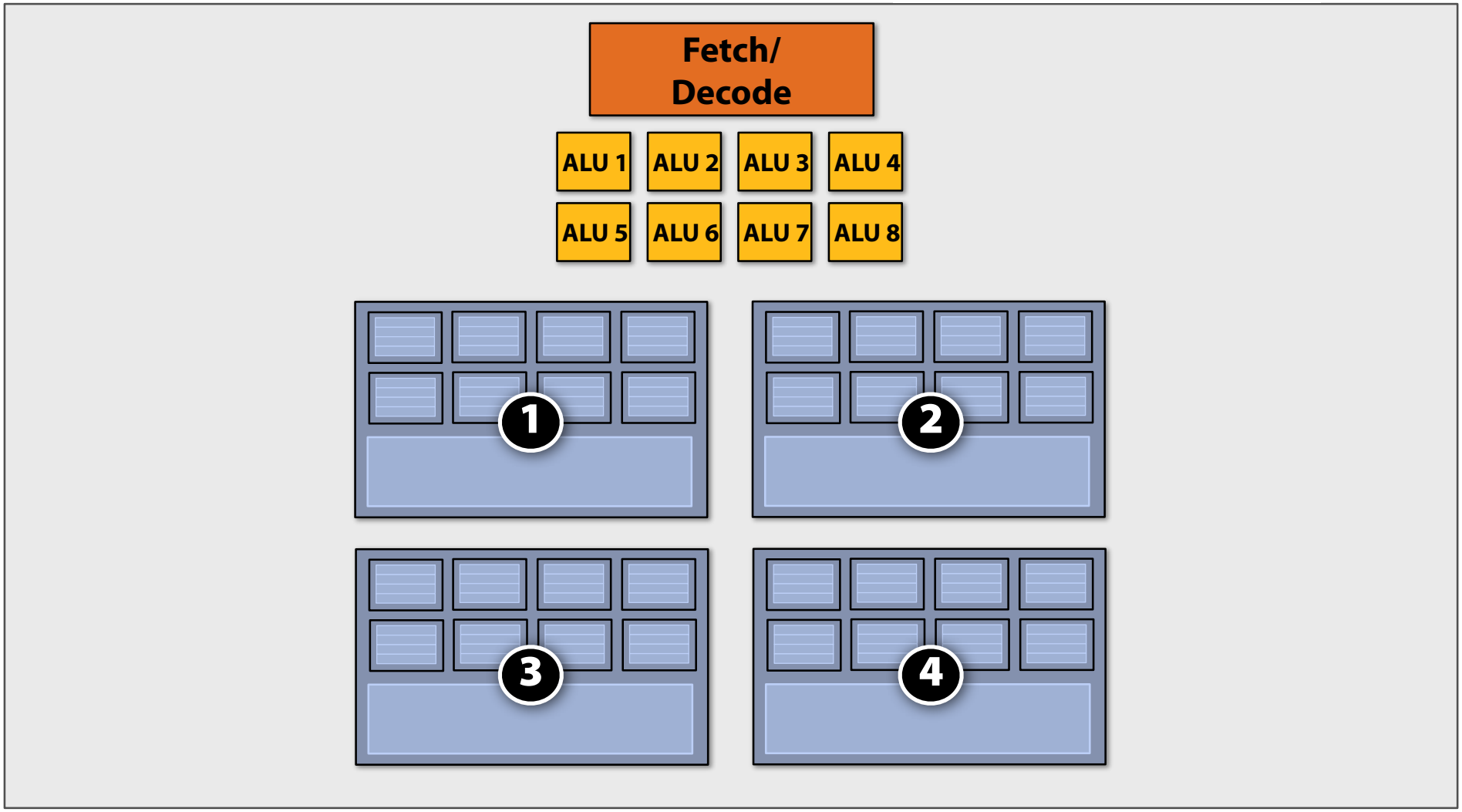


Twelve medium contexts



Four large contexts

(low latency hiding ability)



Clarification

- Interleaving between contexts can be managed by hardware or software (or both!)
- NVIDIA / AMD (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)

Idea 3: Interleave groups of threads

Put it all together: My chip!

16 cores

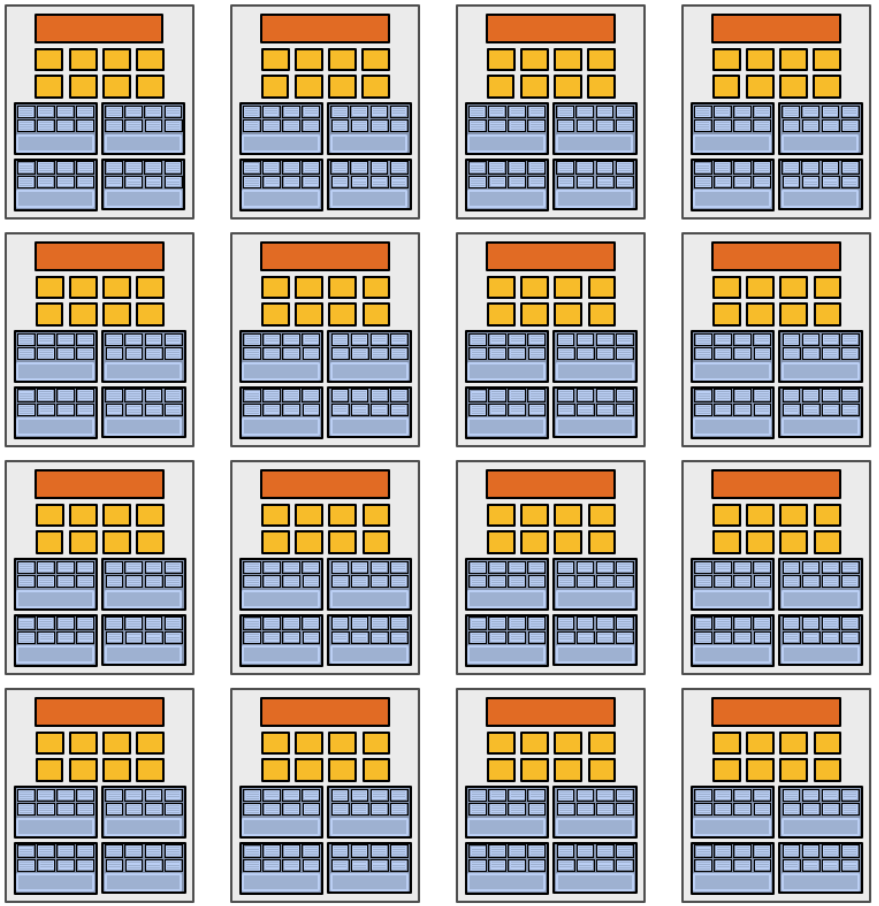
8 mul-add ALUs per core (128 total)

16 simultaneous instruction streams

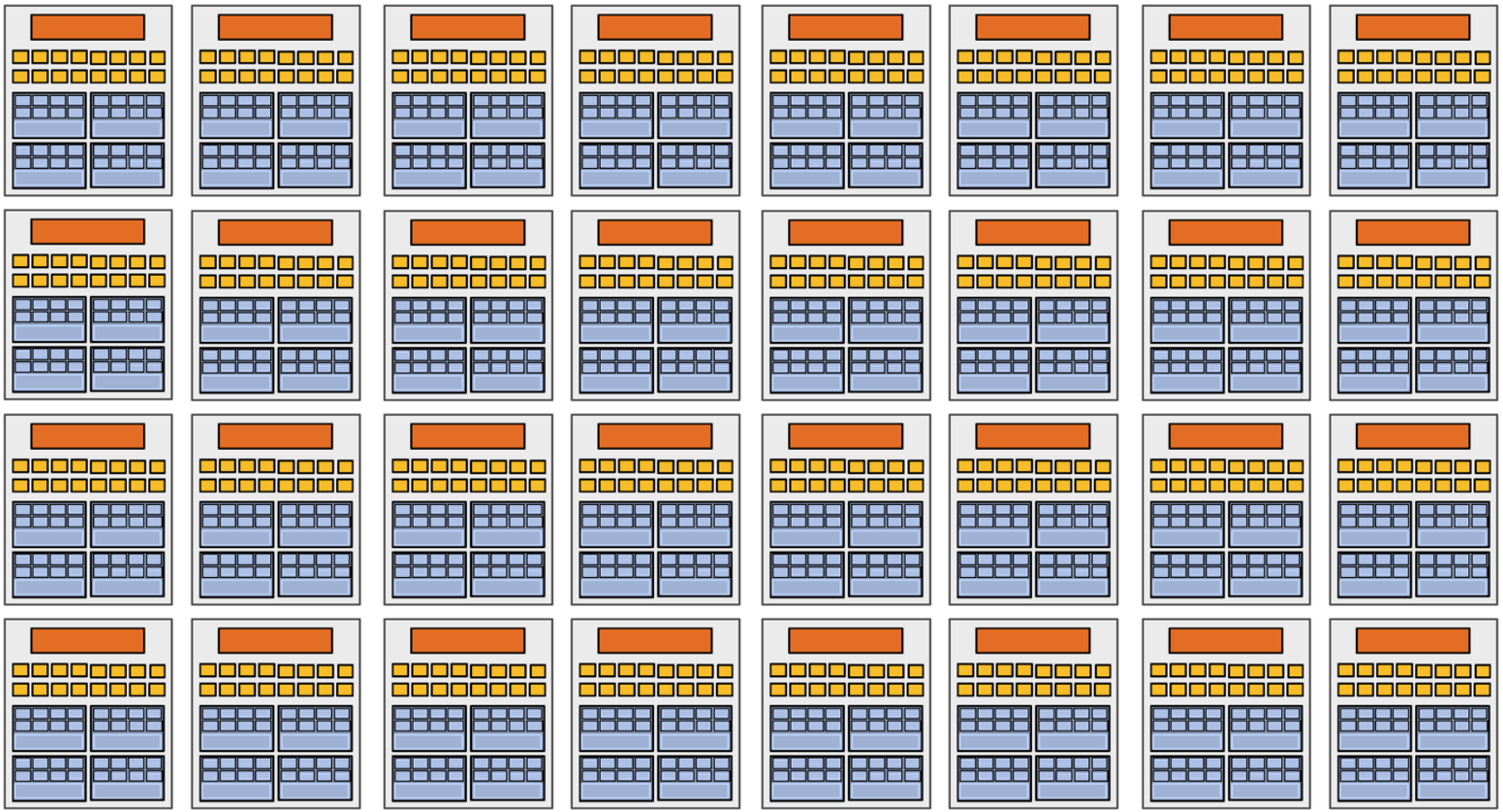
64 concurrent (but interleaved) instruction streams

512 concurrent fragments

256 GFLOPS @1GHZ



My "enthusiast" chip!



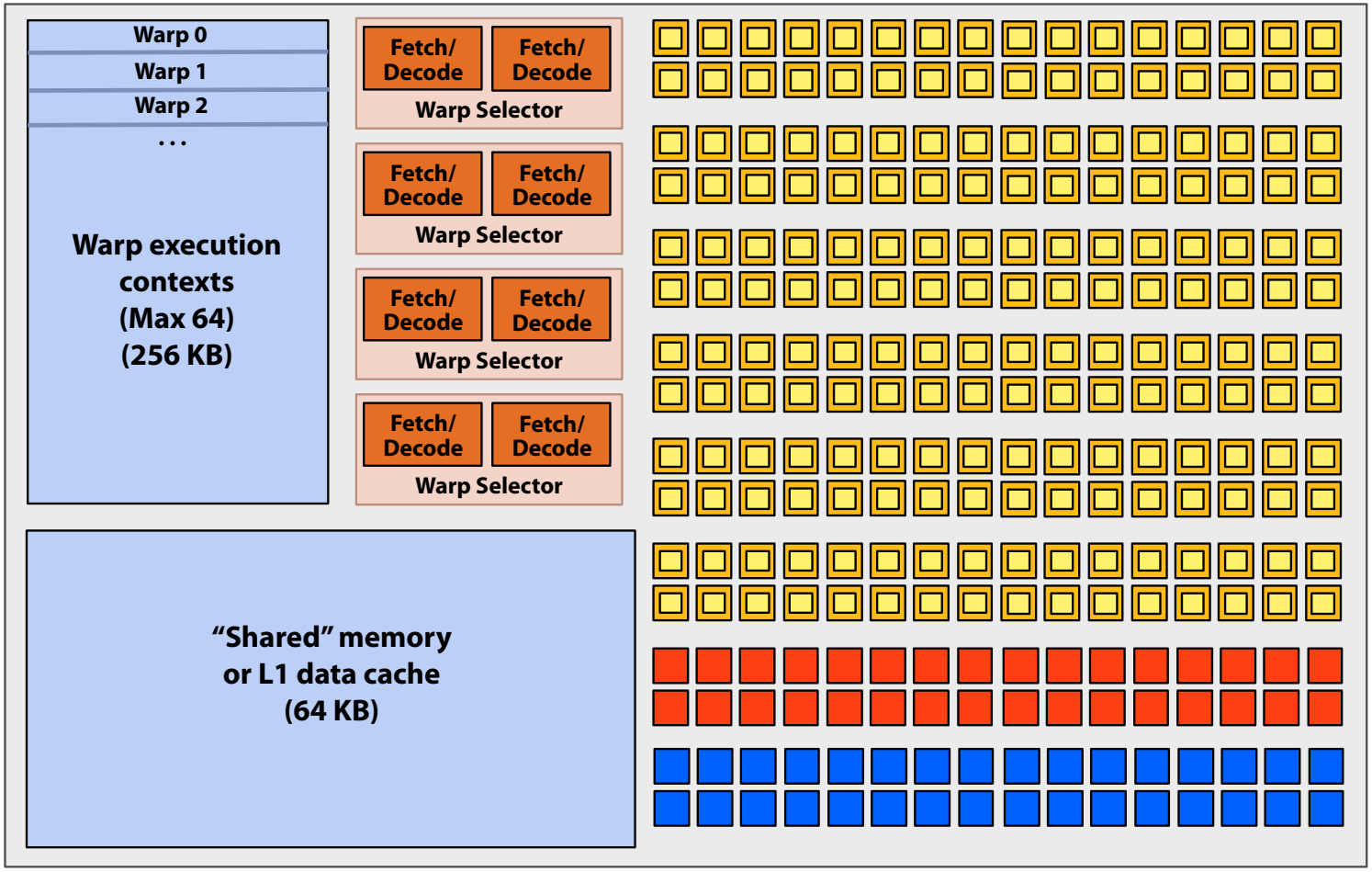
32 cores, 16 ALUs per core (512 total) = 1 TFLOPs (@ 1 GHz)

32 simultaneous instruction streams

Assume each instruction performs one single-precision floating-point operation

NVIDIA GTX 680 (2012)

This is one NVIDIA Kepler GK104 architecture SMX unit (one "core")



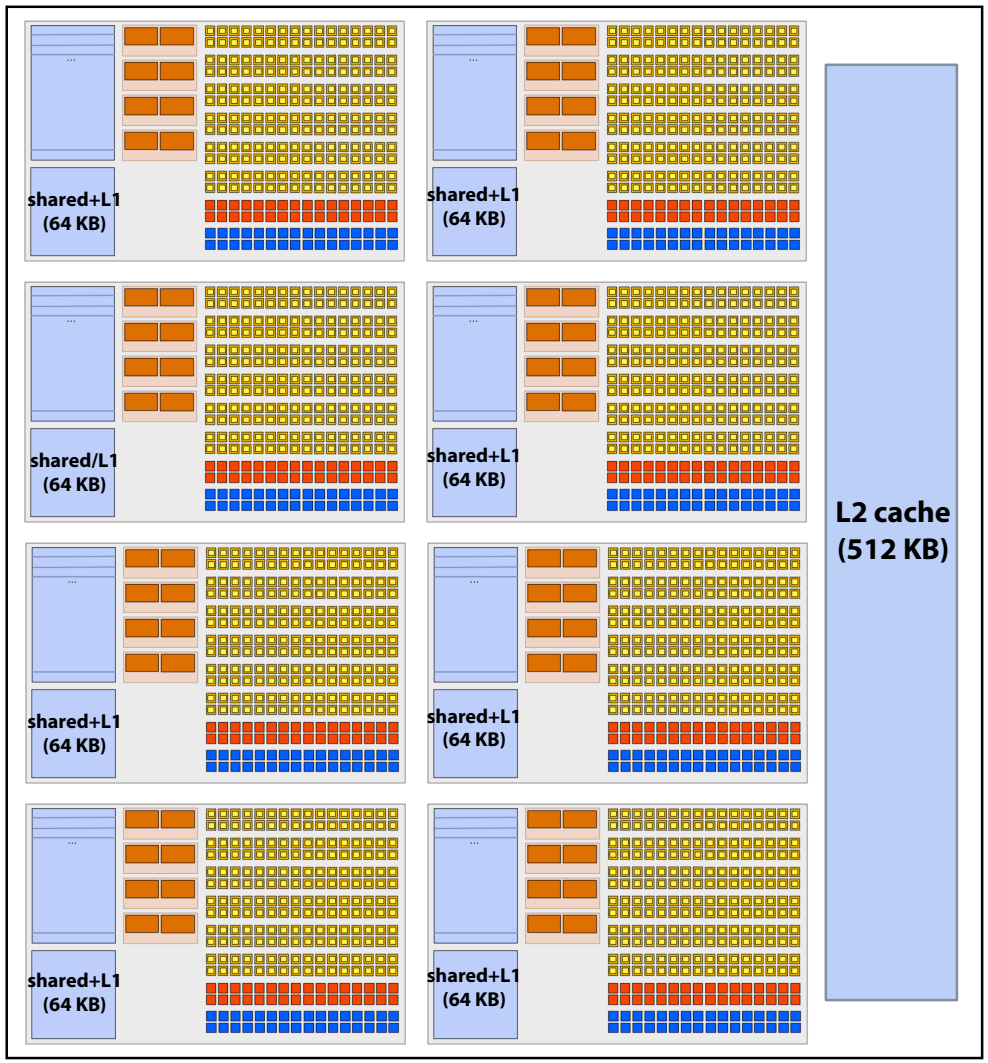
= SIMD functional unit, control shared across 32 units (1 MUL-ADD per clock)

= "special" SIMD functional unit, control shared across 32 units (operations like sin/cos)

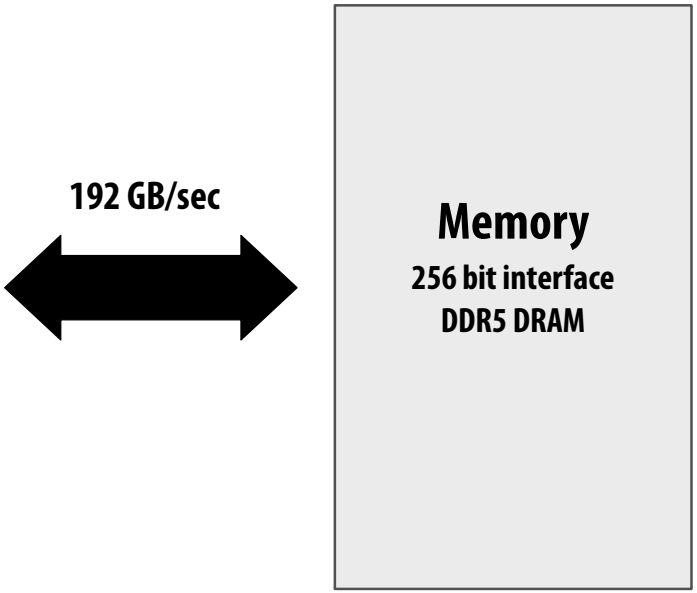
= SIMD load/store unit (handles warp loads/stores, gathers/scatters)

NVIDIA GTX 680 (2012)

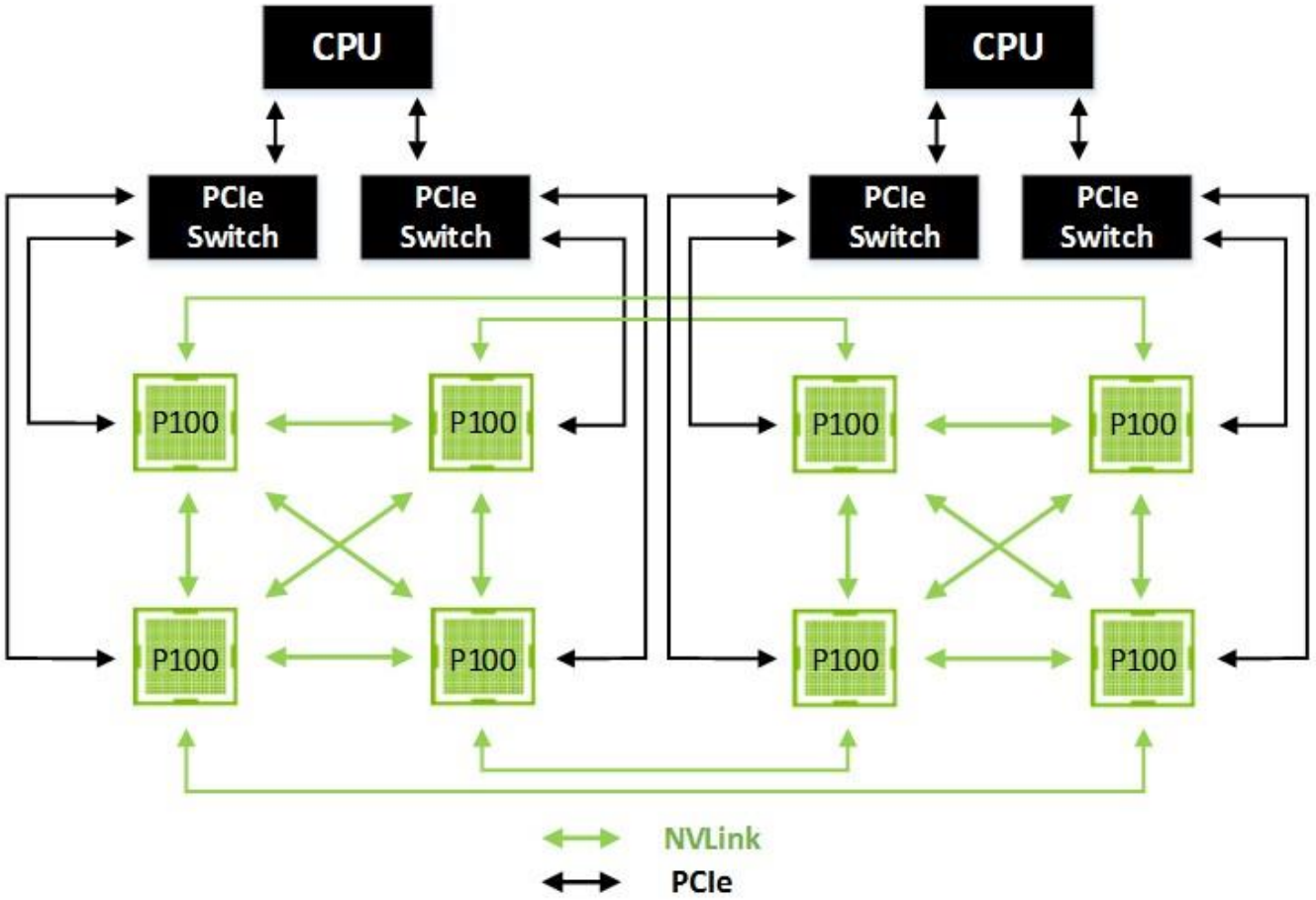
NVIDIA Kepler GK104 architecture



- 1 GHz clock
- Eight SMX cores per chip
- 8 x 192 = 1,536 SIMD mul-add ALUs
  = 3 TFLOPs
- Up to 512 interleaved warps per chip
  (16,384 CUDA threads/chip)
- TDP: 195 watts

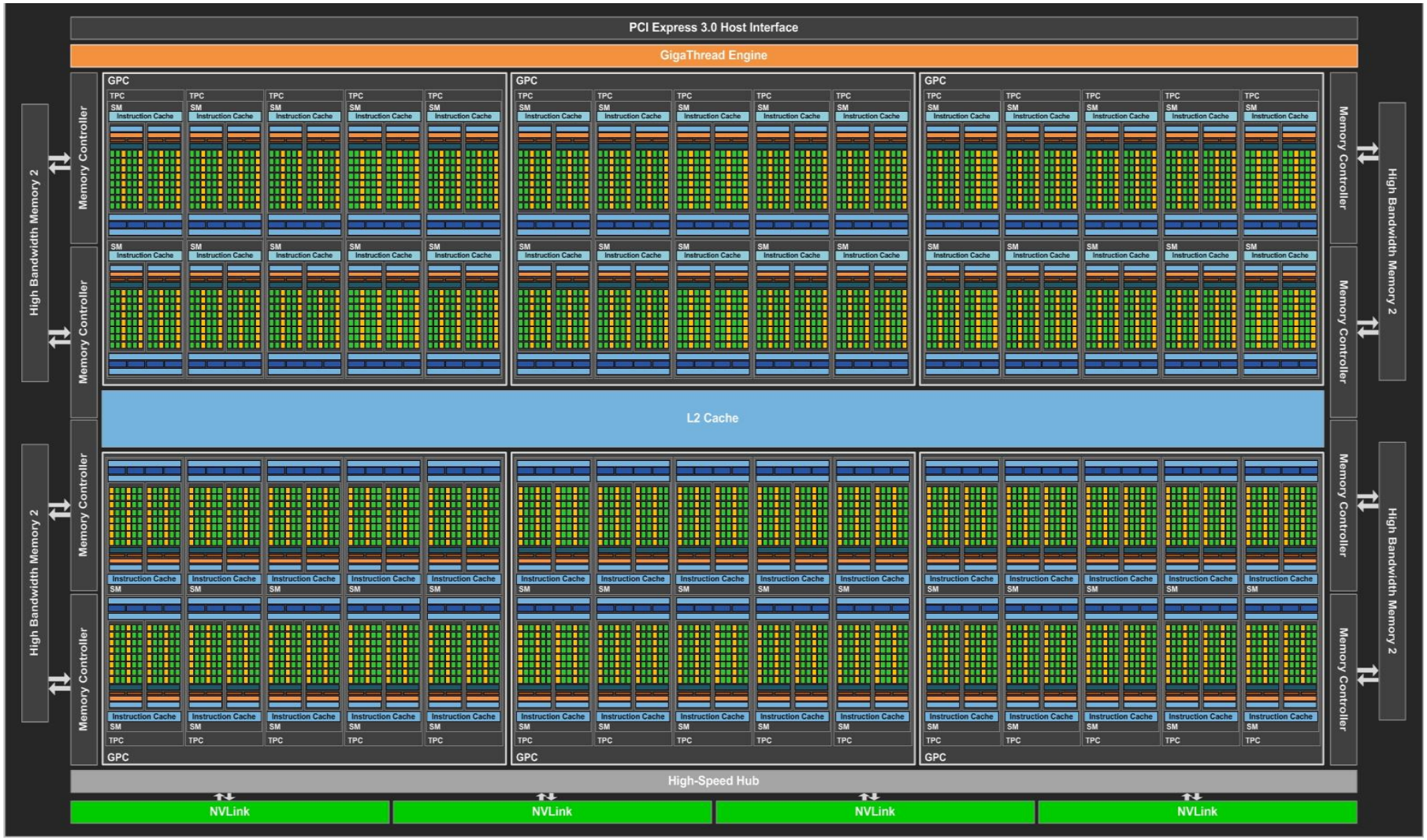


NVIDIA Tesla P100 (Pascal 2016)



NVIDIA Tesla P100 (Pascal 2016)

16nm FinFET

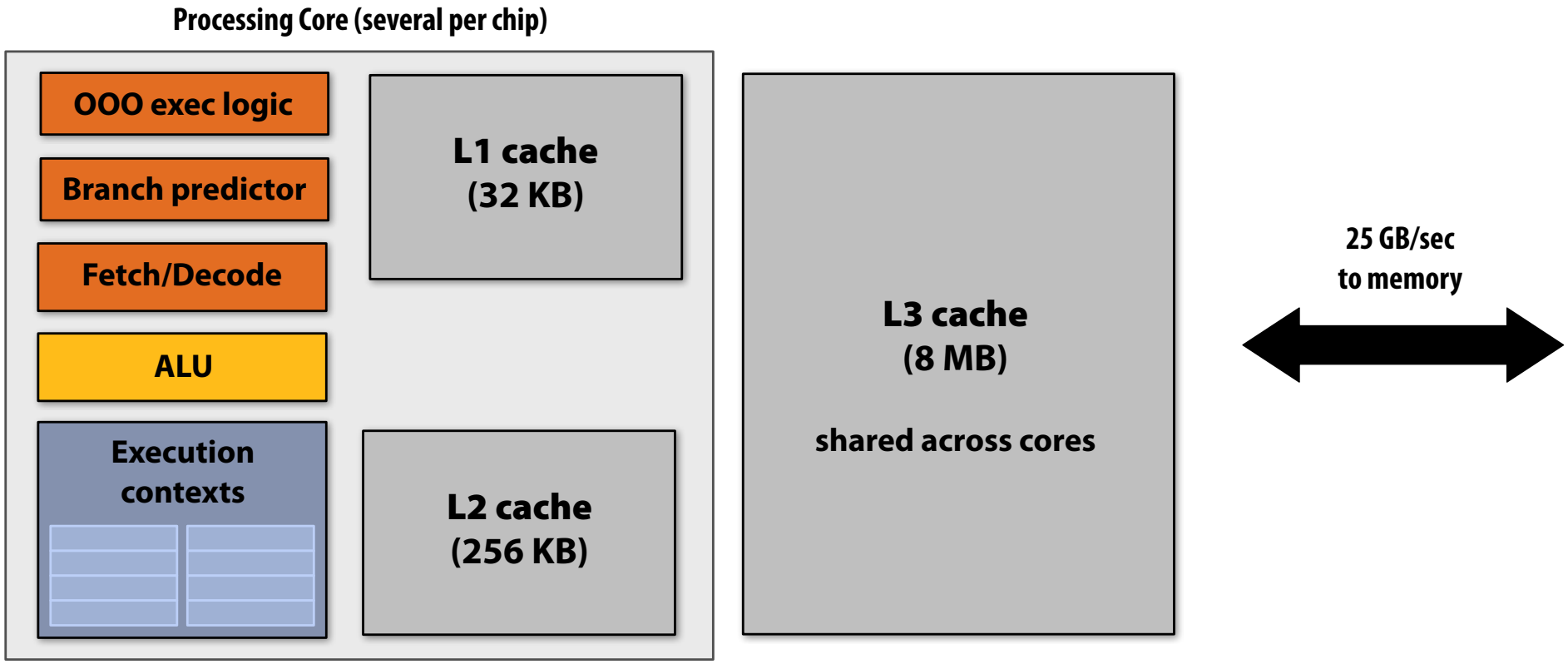


NVIDIA Tesla P100 (Pascal 2016)

Last time: processing data

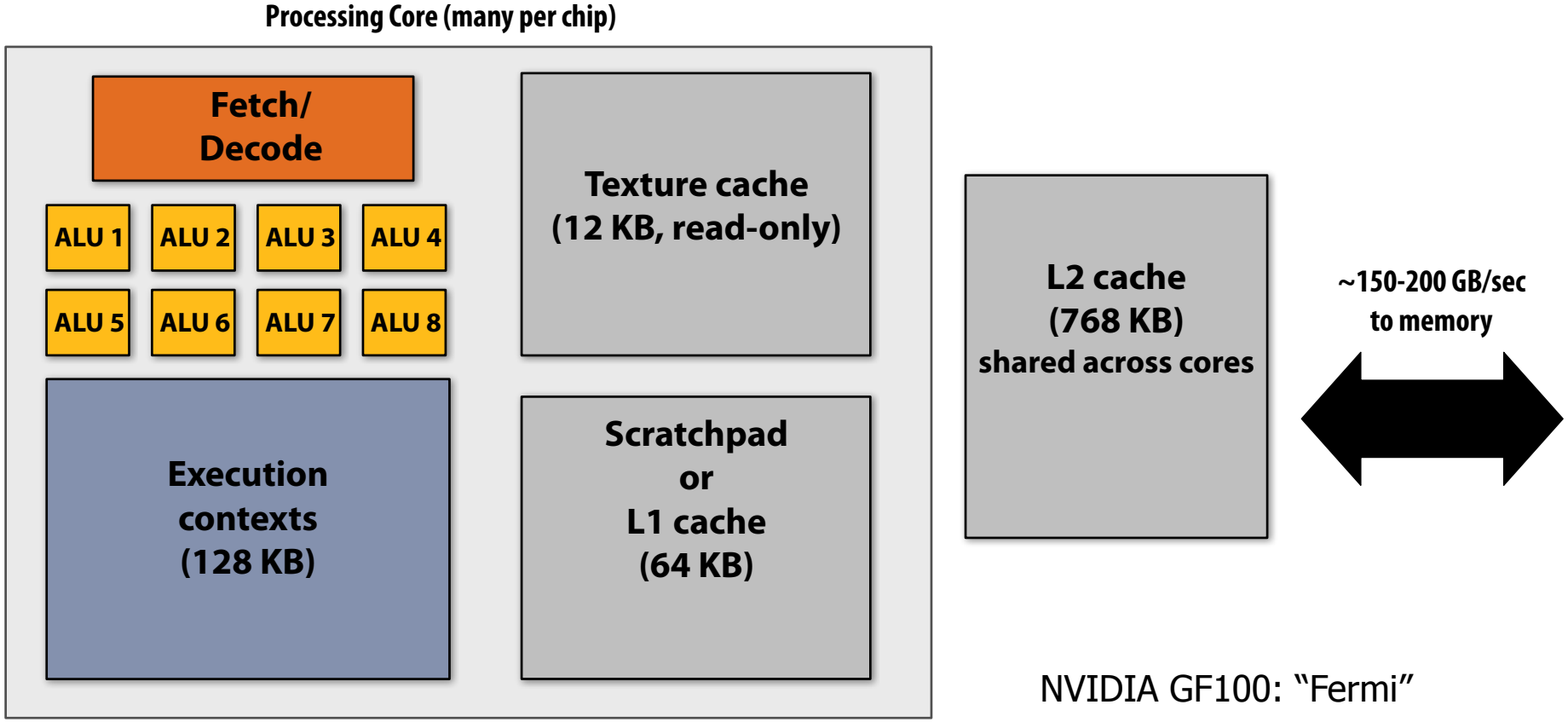
Today:
Moving data to processors

Recall: "CPU-style" memory hierarchy



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

"GPU-style" memory hierarchy



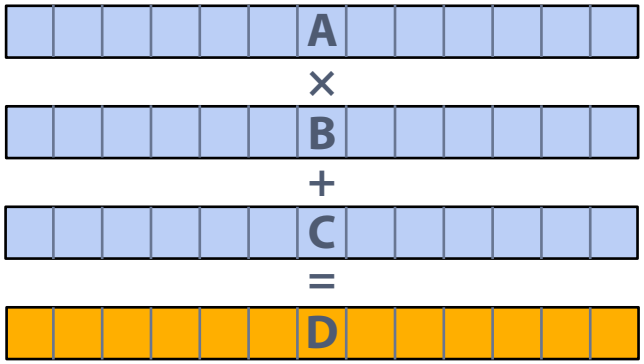
NVIDIA GF100: "Fermi"

More cores, more ALUs, no large traditional cache hierarchy (use threads to tolerate latency) Require high-bandwidth connection to memory

GPUs are memory bandwidth hungry

Task: element-wise multiplication of two 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!

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.

Bandwidth is a critical resource

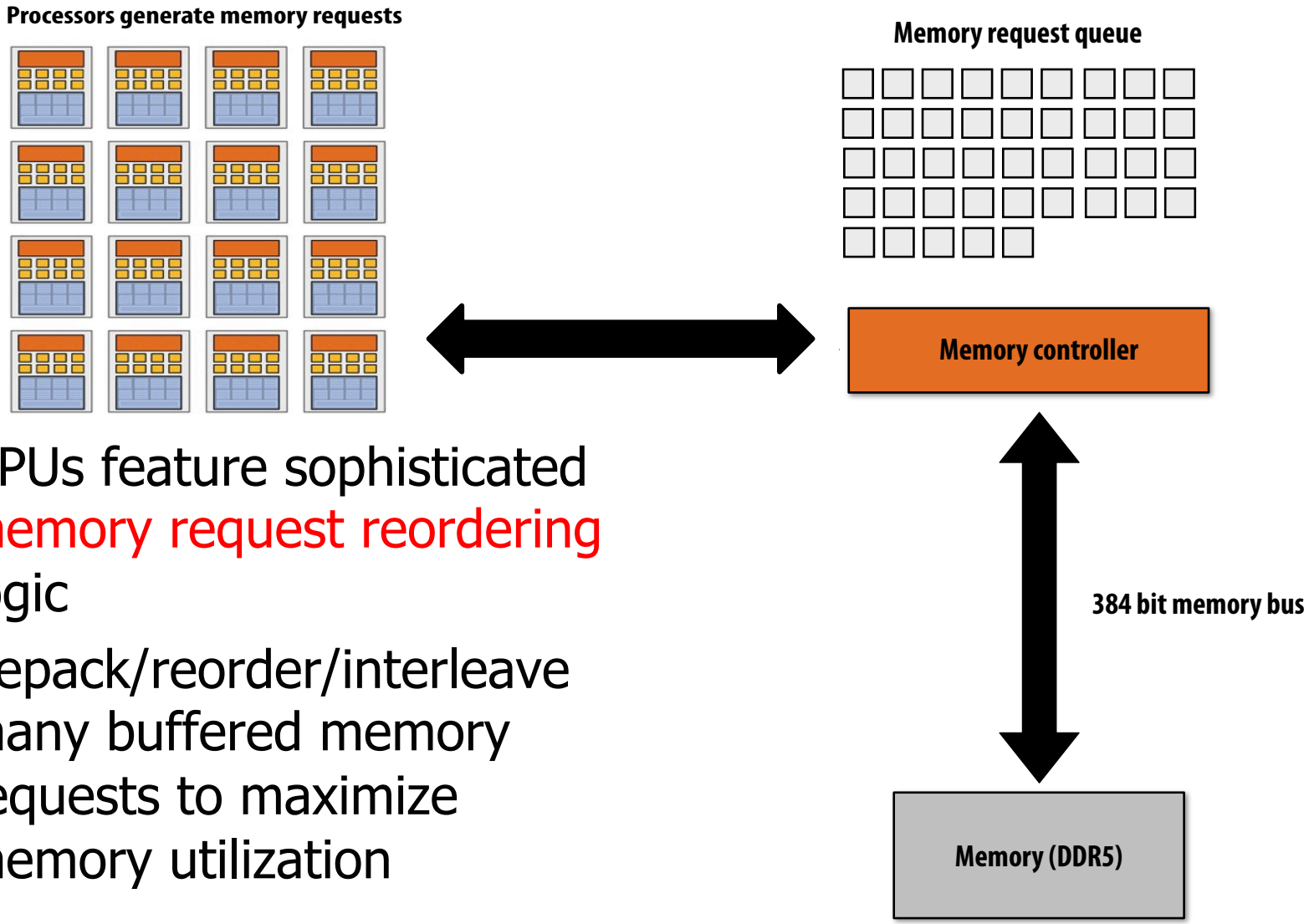
- A high-end GPU (e.g., Radeon HD 5870) has...
  - Over twenty times (2.7 TFLOPS) the compute performance of quad-core CPU
  - No large cache hierarchy to absorb memory requests

- GPU memory systems are designed for throughput
  - GDDR: today's version is GDDR5
  - Wide memory bus & high memory frequency
    - E.g., 384-bit memory bus
    - Bandwidth can achieve 150-200 GB/sec
  - Still, this is only six-to-eight times the bandwidth available to CPU

Deal with the bandwidth challenge

- Use available bandwidth well
- Fetch data from memory less often (share/reuse data)
- Request data less often (instead, do more math: it's "free")
  - "arithmetic intensity" : ratio of math to data access
- New memory technologies beyond GDDR5
  - HBM (3D/2.5D integrated memory), e.g., AMD Fury X, NVIDIA Pascal

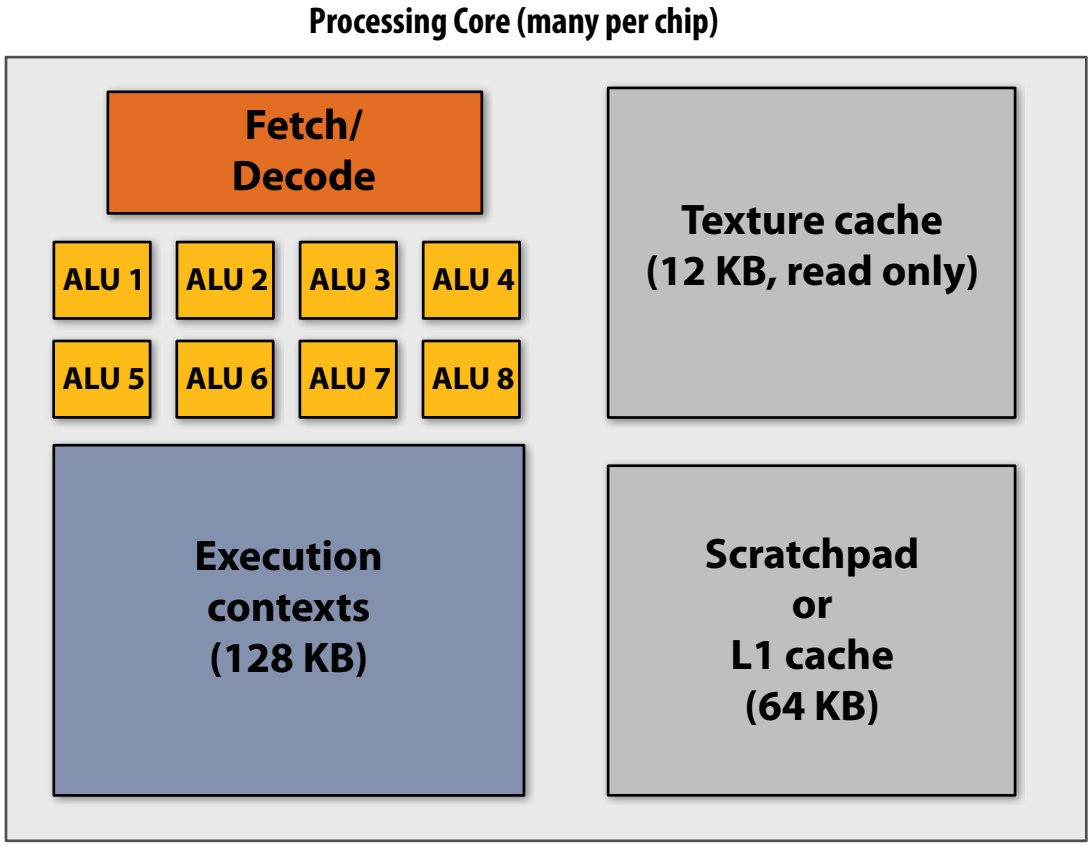
Using available bandwidth well



- GPUs feature sophisticated memory request reordering logic
- Repack/reorder/interleave many buffered memory requests to maximize memory utilization

Fetch data from memory less often

- Scratchpad for reuse known at compile-time
  - Intra-fragment reuse
  - Cross-fragment reuse
- Compression



Load-data into scratchpad (LD addr -> scratchpad addr)
Many fragments reuse data loaded into scratchpad once

Deal with the bandwidth challenge

- Use available bandwidth well
- Fetch data from memory less often (share/reuse data)
- Request data less often (instead, do more math: it's "free")
  - "arithmetic intensity" : ratio of math to data access
- New memory technologies

Shading often has high arithmetic intensity

```
sampler mySamp;
Texture2D<float3> myTex;
float3 ks;
float  shinyExp;
float3 lightDir;
float3 viewDir;

float4 phongShader(float3 norm, float2 uv)
{
  float result;
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  float spec = dot(viewDir, 2 * dot(-lightDir, norm) * norm + lightDir);
  result = kd * clamp(dot(lightDir, norm), 0.0, 1.0);
  result += ks * exp(spec, shinyExp);
  return float4(result, 1.0);
}
```

Image credit: http://caig.cs.nctu.edu.tw/course/CG2007

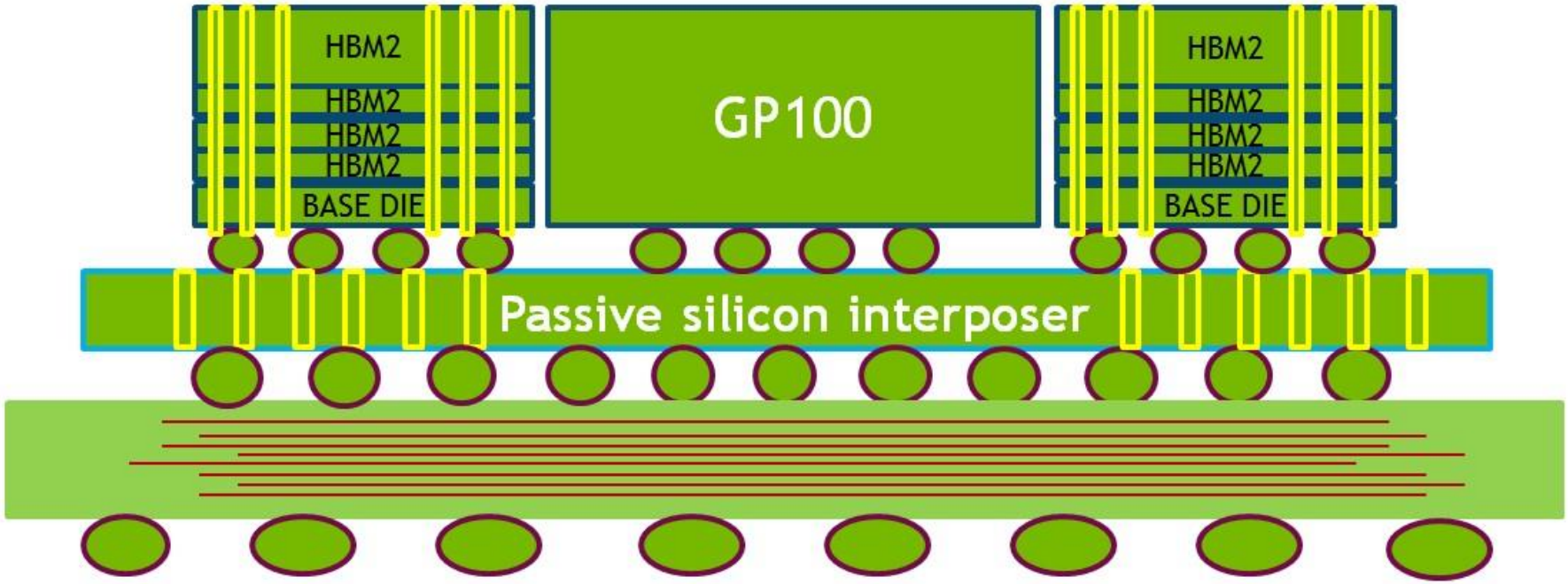
3 scalar float operations + 1 exp()
8 float3 operations + 1 clamp()
1 texture access (highlighted in red)

Vertex processing often has higher arithmetic intensity than fragment processing (less use of texturing)

Deal with the bandwidth challenge

- Use available bandwidth well
- Fetch data from memory less often (share/reuse data)
- Request data less often (instead, do more math: it's "free")
  - "arithmetic intensity" : ratio of math to data access
- New memory technologies
  - E.g., 3D integration – main memory can be "on-chip"

NVIDIA Tesla P100 (Pascal 2016)



Some cards feature 16 GB HBM2 in four stacks with a total of 4096bit bus with a memory bandwidth of 720 GB/s

Summary: workloads that run efficiently on a GPU's programming cores…

- Have thousands of independent pieces of work
  - Utilizes many ALUs on many cores
  - Have more more parallel work than numbers of GPU ALUs, enabling large-scale interleaving as a mechanism to hide stall latency, e.g., memory access and branch
- Are amenable to instruction stream sharing
  - Maps to SIMD execution well
- Are compute-heavy: the ratio of math operations to memory access is high
  - Not limited by memory bandwidth