CIS 5710: Computer Architecture

Unit 14: Data-Level Parallelism & Accelerators

Slides developed by Joe Devietti, Milo Martin & Amir Roth at UPenn
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood

How to Compute SAXPY Quickly?

- Performing the **same** operations on **many** data items
  - Example: SAXPY

```
for (I = 0; I < 1024; I++) {
  Z[I] = A*X[I] + Y[I];
}
```

```
L1: ldf [X+r1]->f1   // I is in r1
    mulf f0,f1->f2   // A is in f0
    ldf [Y+r1]->f3
    addf f2,f3->f4
    stf f4->[Z+r1}
    addi r1,4->r1
    blti r1,4096,L1
```

- Instruction-level parallelism (ILP) - fine grained
  - Loop unrolling with static scheduling –or– dynamic scheduling
  - Wide-issue superscalar (non-)scaling limits benefits
- Thread-level parallelism (TLP) - coarse grained
  - Multicore
- Can we do some "medium grained" parallelism?

Data-Level Parallelism

- **Data-level parallelism (DLP)**
  - Single operation repeated on multiple data elements
    - SIMD (**S**ingle-**I**nstruction, **M**ultiple-**D**ata)
  - Less general than ILP: parallel insns are all same operation
  - Exploit with **vectors**
- Old idea: Cray-1 supercomputer from late 1970s
  - Eight 64-entry x 64-bit floating point "vector registers"
    - 4096 bits (0.5KB) in each register! 4KB for vector register file
  - Special vector instructions to perform vector operations
    - Load vector, store vector (wide memory operation)
    - Vector+Vector or Vector+Scalar
      - addition, subtraction, multiply, etc.
    - In Cray-1, each instruction specifies 64 operations!
  - ALUs were expensive, so one operation per cycle (not parallel)

Example Vector ISA Extensions (SIMD)

- Extend ISA with vector storage …
  - **Vector register**: fixed-size array of FP/int elements
  - **Vector length**: For example: 4, 8, 16, 64, …
- … and example operations for vector length of 4
  - Load vector: **`ldf.v [X+r1]->v1`**

    `ldf [X+r1+0]->v1`$_0$

    `ldf [X+r1+1]->v1`$_1$

    `ldf [X+r1+2]->v1`$_2$

    `ldf [X+r1+3]->v1`$_3$
  - Add two vectors: **`addf.vv v1,v2->v3`**

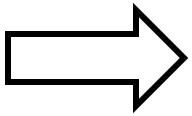
    `addf v1`$_i$**`,v2`**$_i$**`->v3`**$_i$ **`(where i is 0,1,2,3)`**
  - Add vector to scalar: **`addf.vs v1,f2,v3`**

    `addf v1`$_i$**`,f2->v3`**$_i$ **`(where i is 0,1,2,3)`**
- Today's vectors: short (128-512 bits), but fully parallel

Example Use of Vectors – 4-wide

```
ldf [X+r1]->f1
mulf f0,f1->f2
ldf [Y+r1]->f3
addf f2,f3->f4
stf f4->[Z+r1]
addi r1,4->r1
blti r1,4096,L1
```

7x1024 instructions



```
ldf.v [X+r1]->v1
mulf.vs v1,f0->v2
ldf.v [Y+r1]->v3
addf.vv v2,v3->v4
stf.v v4,[Z+r1]
addi r1,16->r1
blti r1,4096,L1
```

7x256 instructions
(4x fewer instructions)

- Operations
  - Load vector: **`ldf.v [X+r1]->v1`**
  - Multiply vector to scalar: **`mulf.vs v1,f2->v3`**
  - Add two vectors: **`addf.vv v1,v2->v3`**
  - Store vector: **`stf.v v1->[X+r1]`**
- Performance?
  - Best case: 4x speedup
  - But, vector instructions don't always have single-cycle throughput
    - Execution width (implementation) vs vector width (ISA)

Vector Datapath & Implementatoin

- Vector insn. are just like normal insn… only "wider"
  - Single instruction fetch (no extra N^2 checks)
  - Wide register read & write (not multiple ports)
  - Wide execute: replicate floating point unit (same as superscalar)
  - Wide bypass (avoid N^2 bypass problem)
  - Wide cache read & write (single cache tag check)
- Execution width (implementation) vs vector width (ISA)
  - Example: Pentium 4 and "Core 1" executes vector ops at half width
  - "Core 2" executes them at full width
- Because they are just instructions…
  - …superscalar execution of vector instructions
  - Multiple n-wide vector instructions per cycle

Vector Insn Sets for Different ISAs

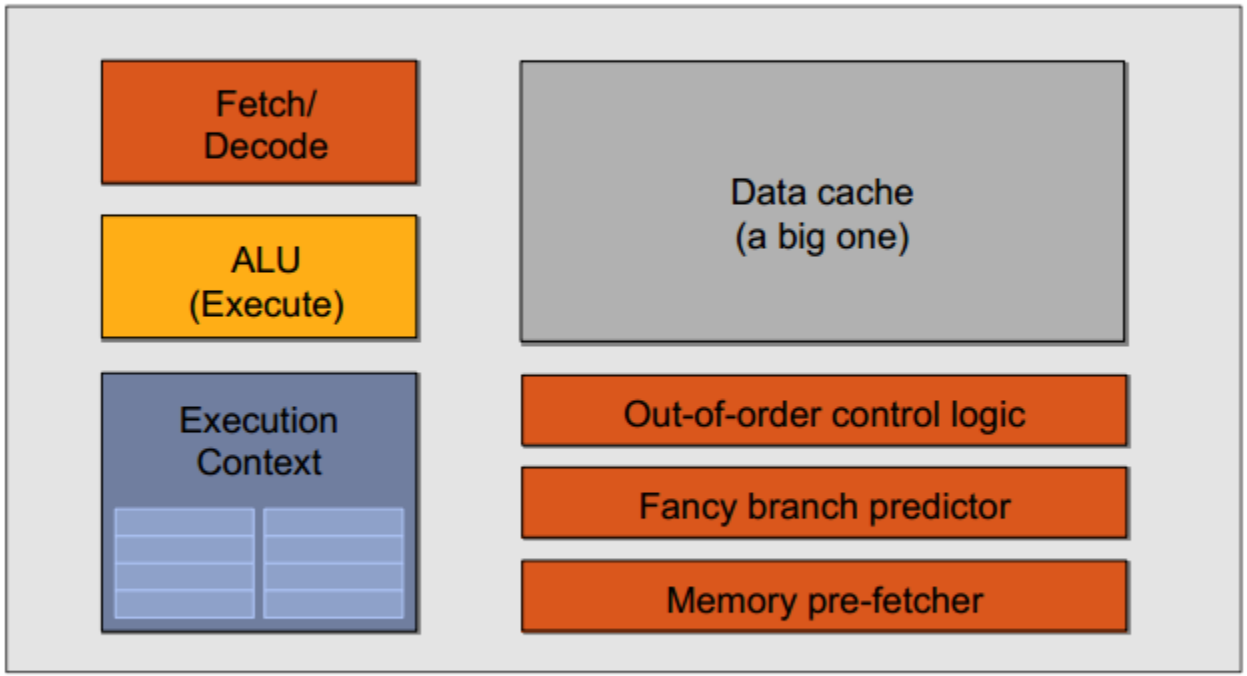
- x86
  - Intel and AMD: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2
  - currently: AVX 512 offers 512b vectors
- PowerPC
  - AltiVEC/VMX: 128b
- ARM
  - NEON: 128b
  - Scalable Vector Extensions (SVE): up to 2048b
    - implementation is narrower than this!
    - makes vector code portable

By the numbers: CPU vs GPU

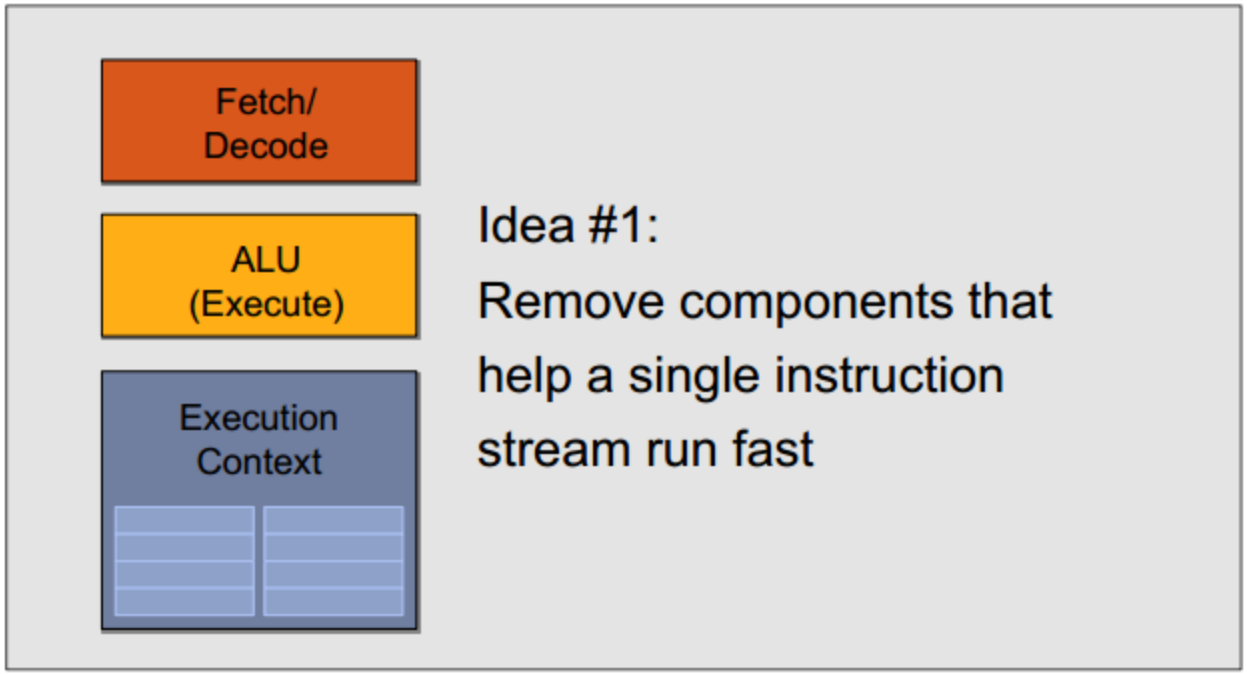
	Intel Xeon Platinum 8276L "Cascade Lake"	Nvidia Quadro GV100
frequency	2.2-4.0 GHz	1.1 GHz
cores / threads	28 / 56	80 ("5120") / 10Ks
RAM	4.5 TB	32 GB
DP TFLOPS	~1.5	5.8
Transistors	>5B ?	21.1B
Price	$11,700	$9,000

- following slides c/o Kayvon Fatahalian's "Beyond Programmable Shading" course
  - http://www.cs.cmu.edu/~kayvonf/

“CPU-style” cores



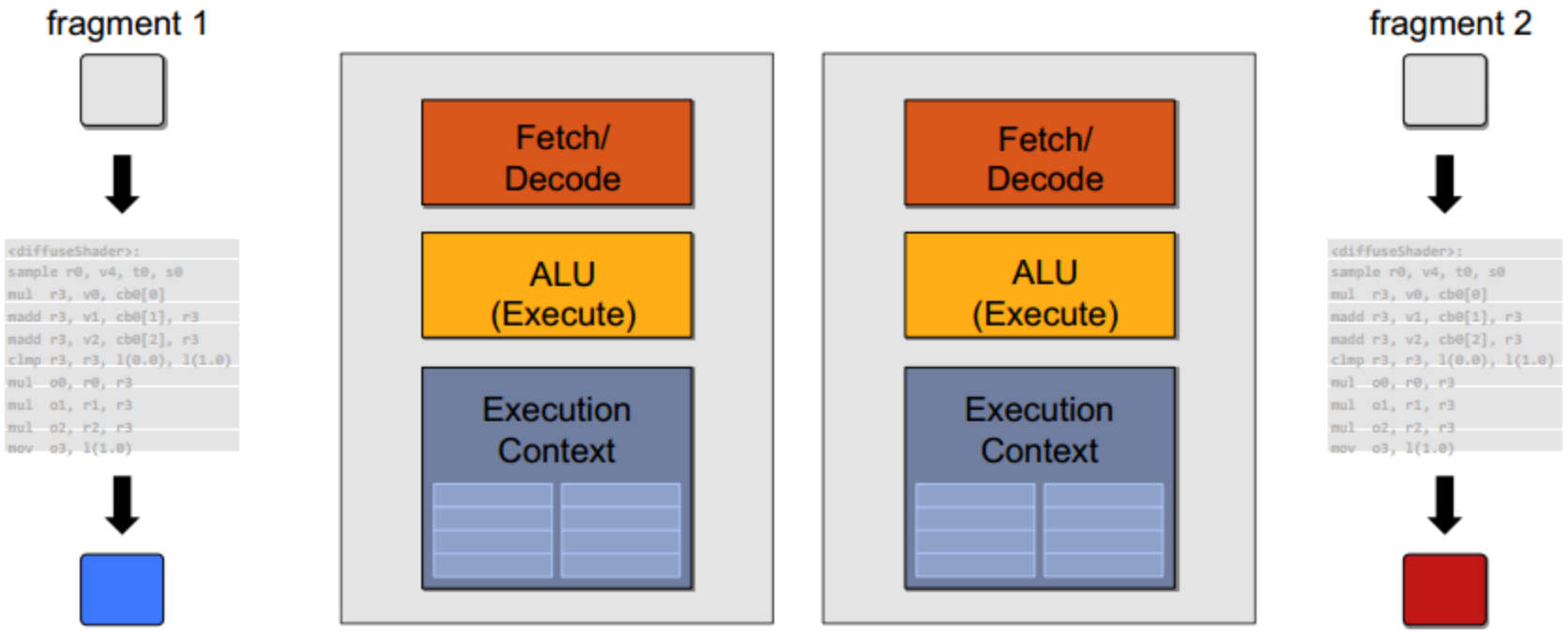
Slimming down



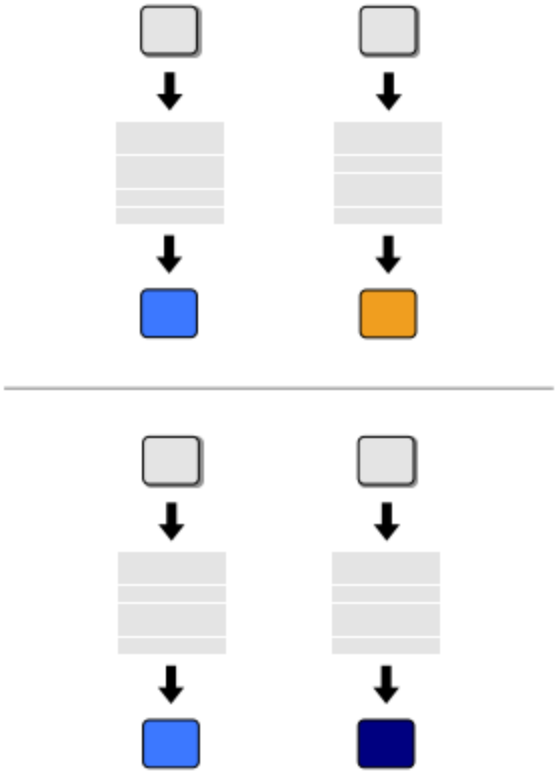
Idea #1:

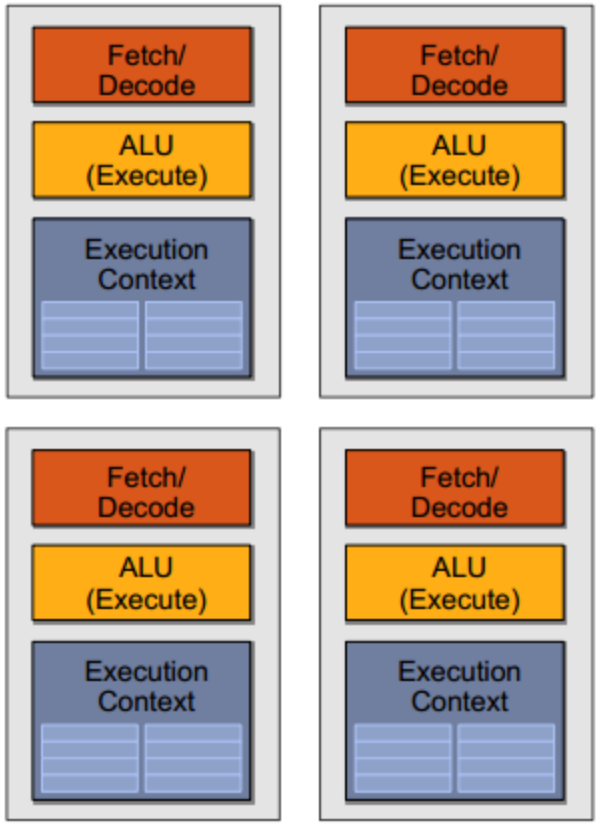
Remove components that help a single instruction stream run fast

Two cores (two fragments in parallel)

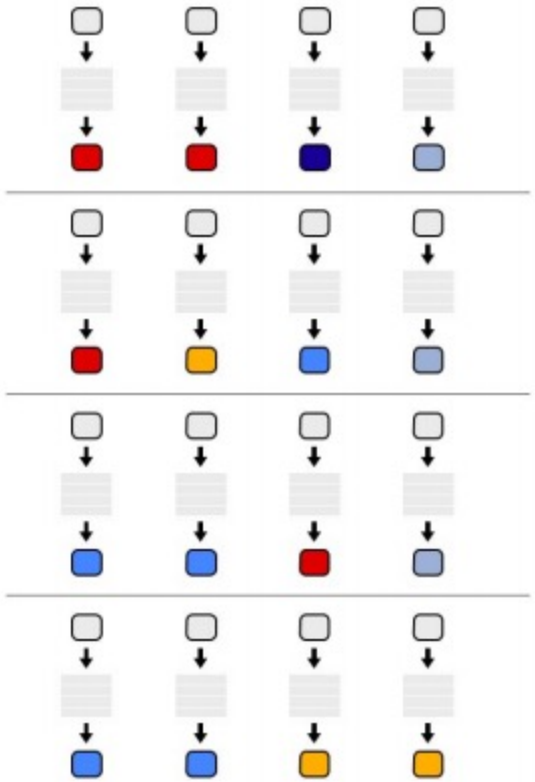


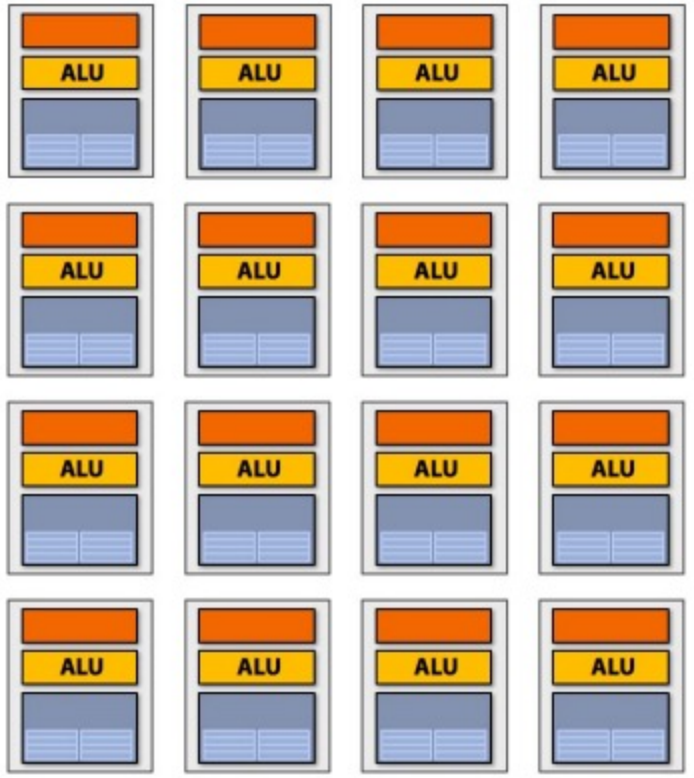
Four cores (four fragments in parallel)





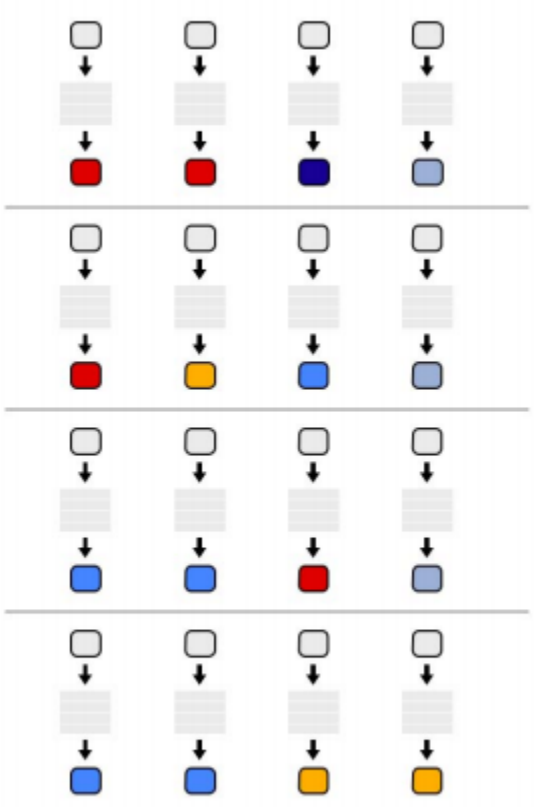
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

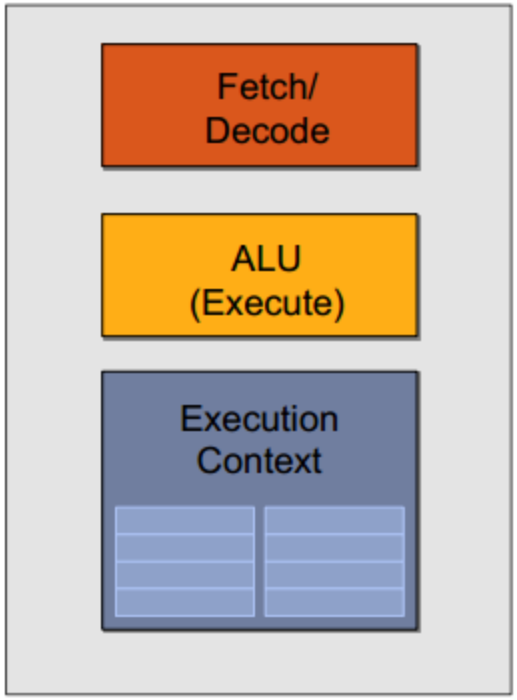
Instruction stream sharing



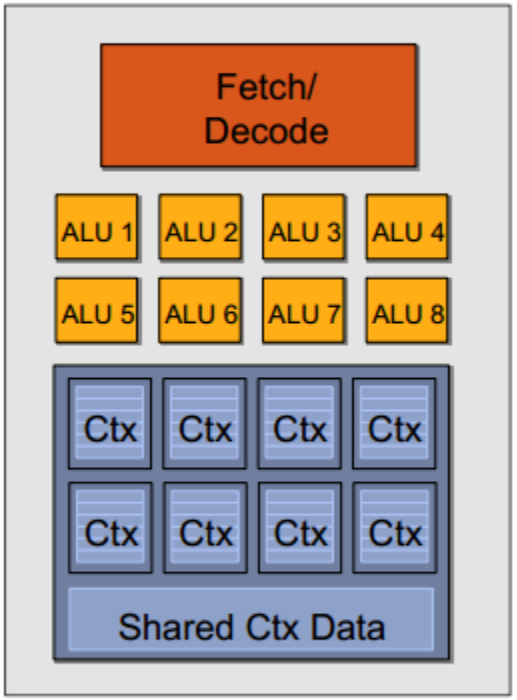
But ... many fragments should be able to share an instruction stream!

```
<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



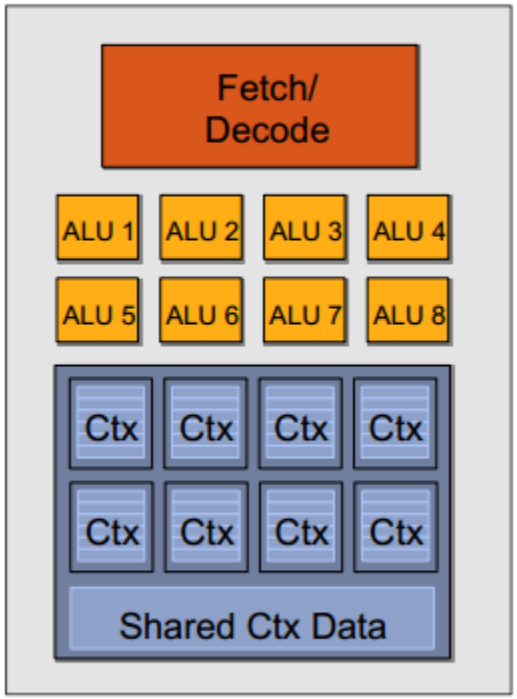
Add ALUs

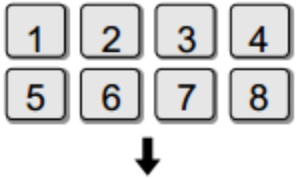


Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

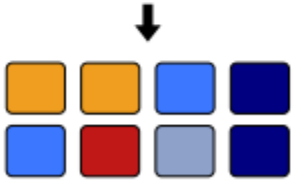
SIMD processing

Modifying the shader

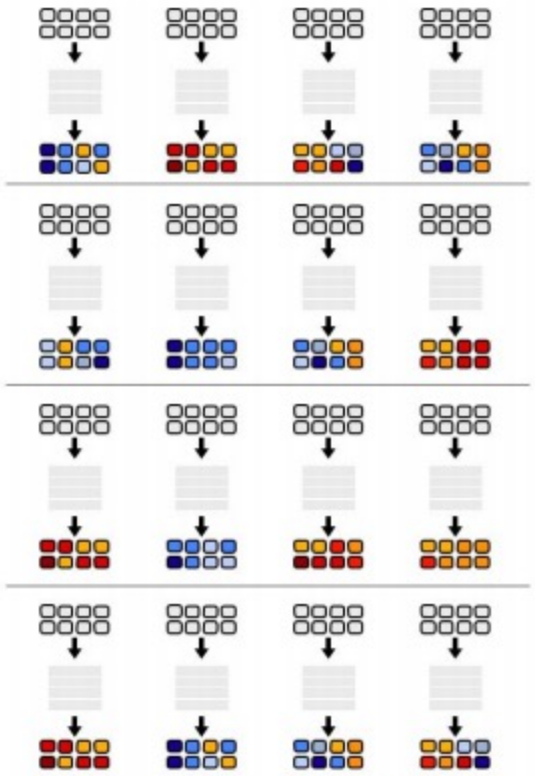


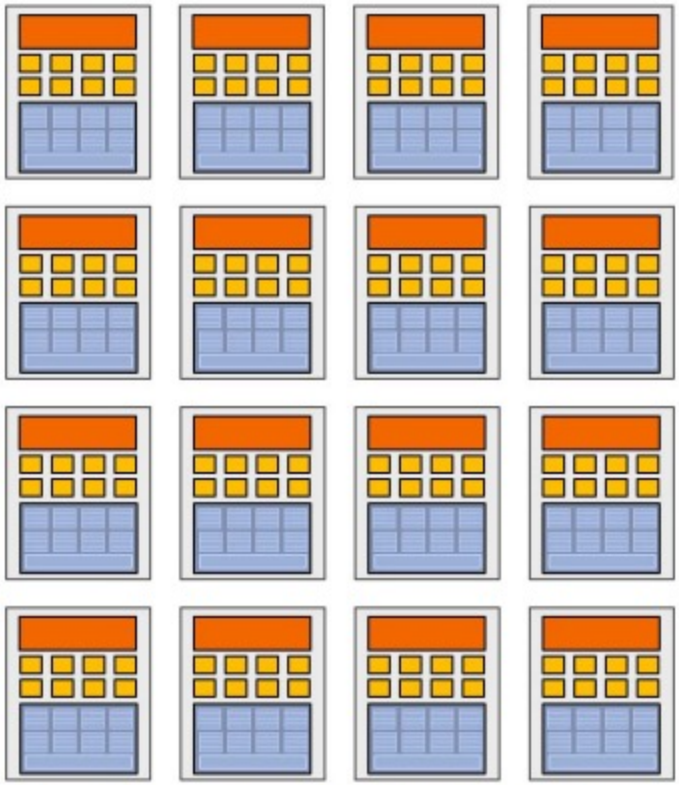



```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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



128 fragments in parallel

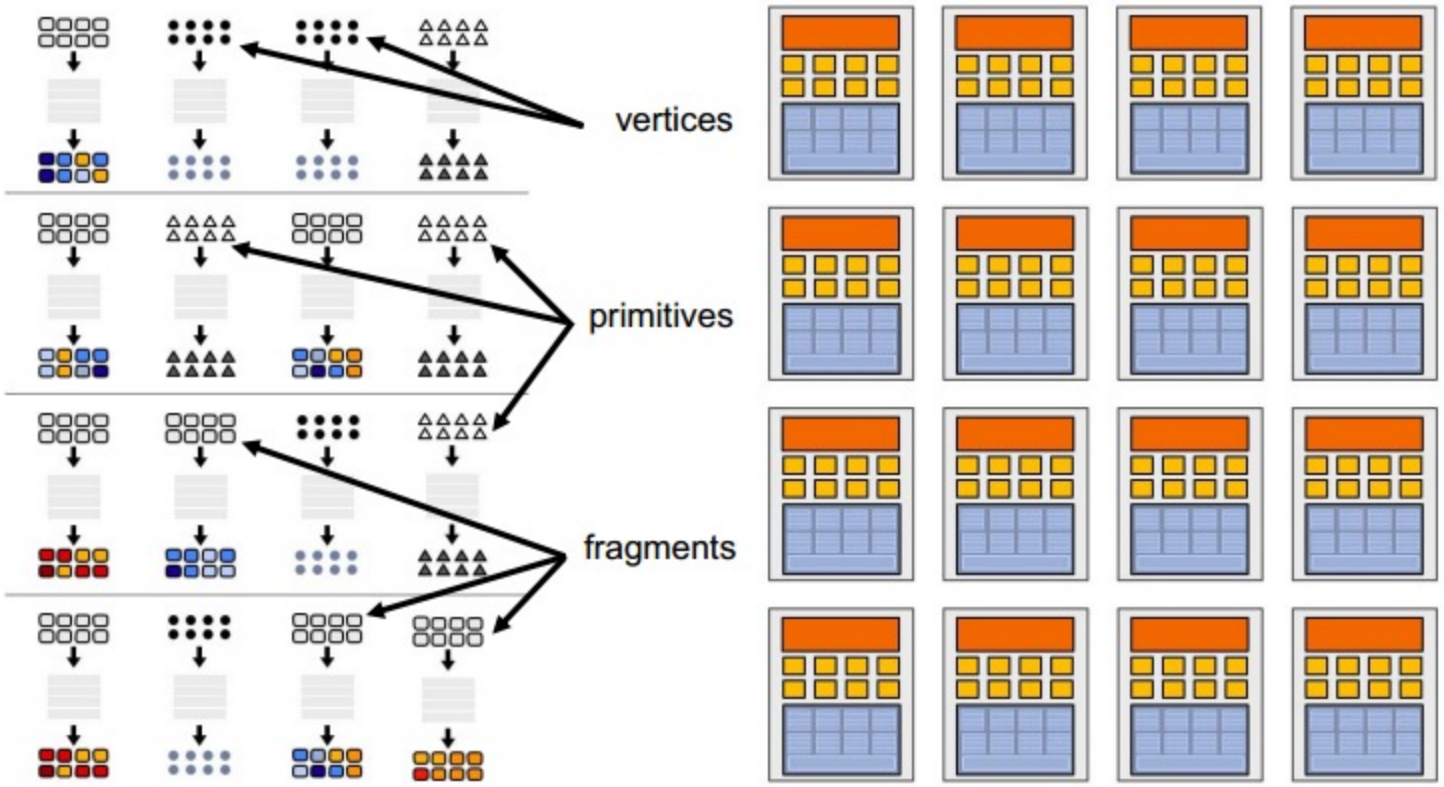




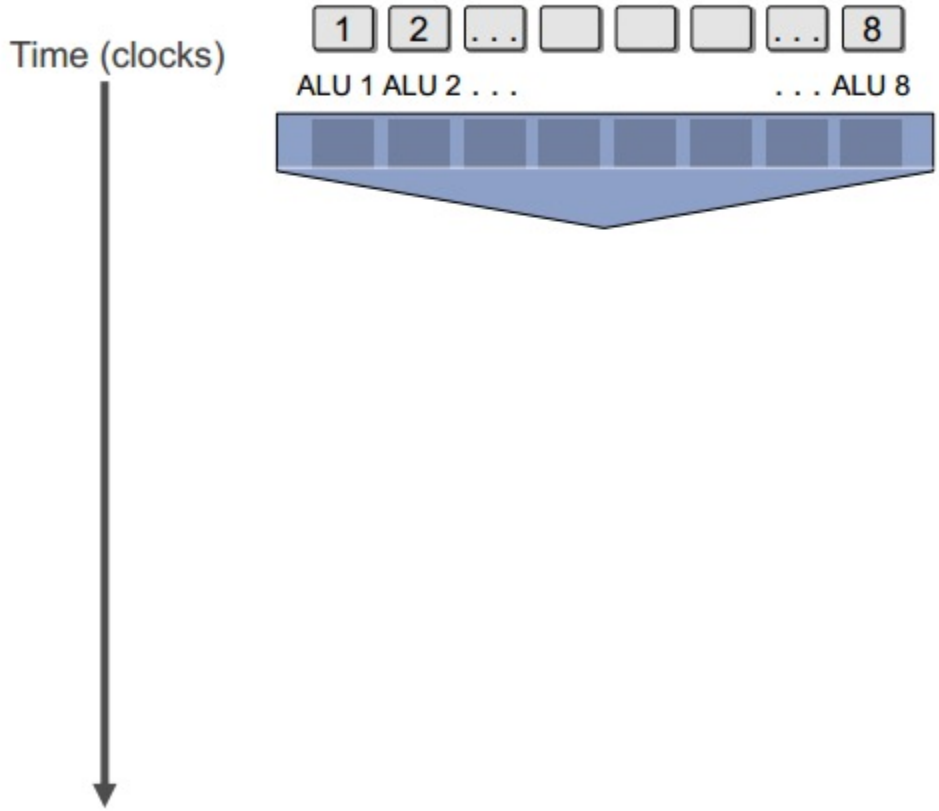
16 cores = 128 ALUs

, 16 simultaneous instruction streams

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



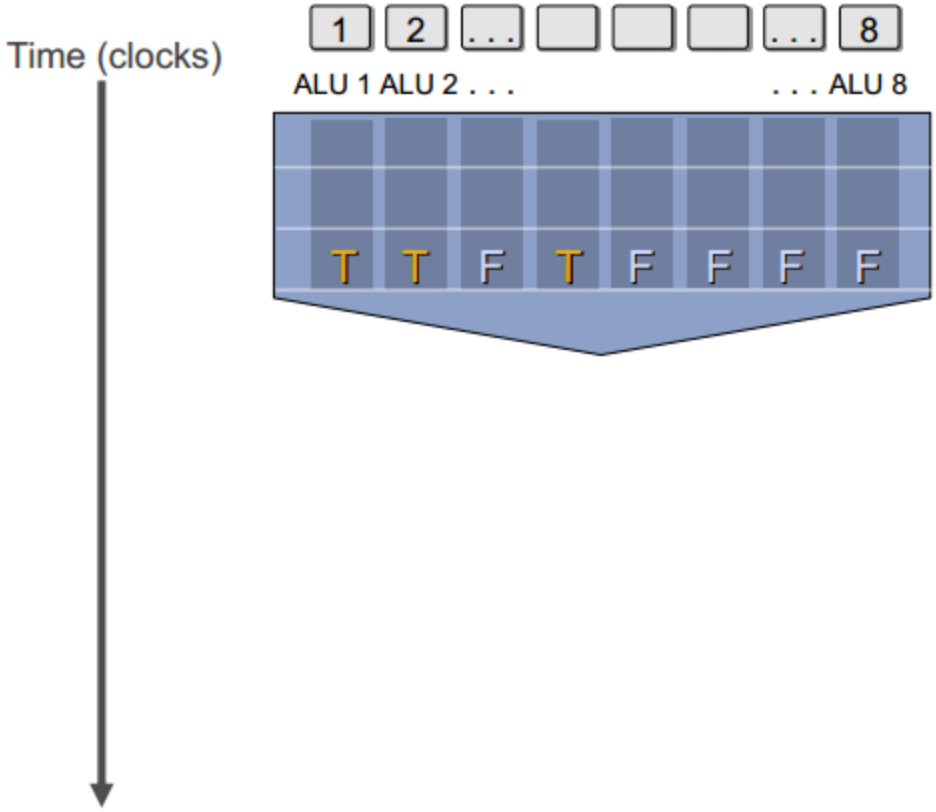
But what about branches?




```
<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?

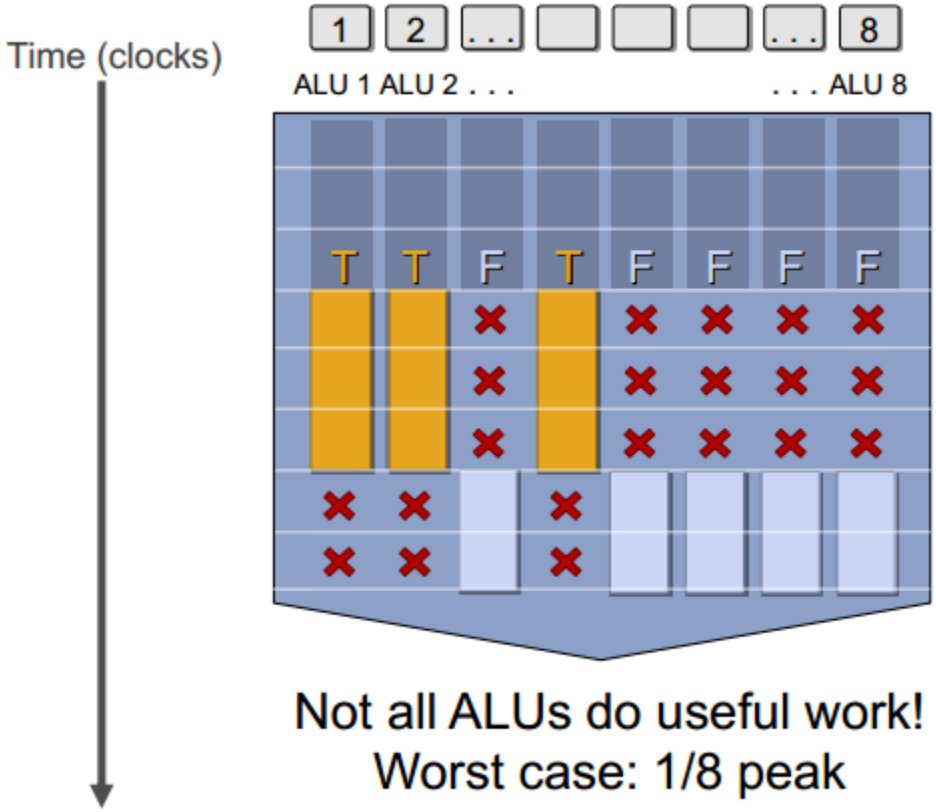



```
<unconditional
 shader code>

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.

SIMD vs SIMT

- SIMD: single insn multiple **data**
  - write 1 insn that operates on a vector of data
  - you handle control flow via explicit masking operations
- SIMT: single insn multiple **thread**
  - write 1 insn that operates on scalar data
  - each of many threads runs this insn
  - compiler+hw aggregate threads into groups that execute on SIMD hardware
  - compiler+hw handle masking for control flow

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

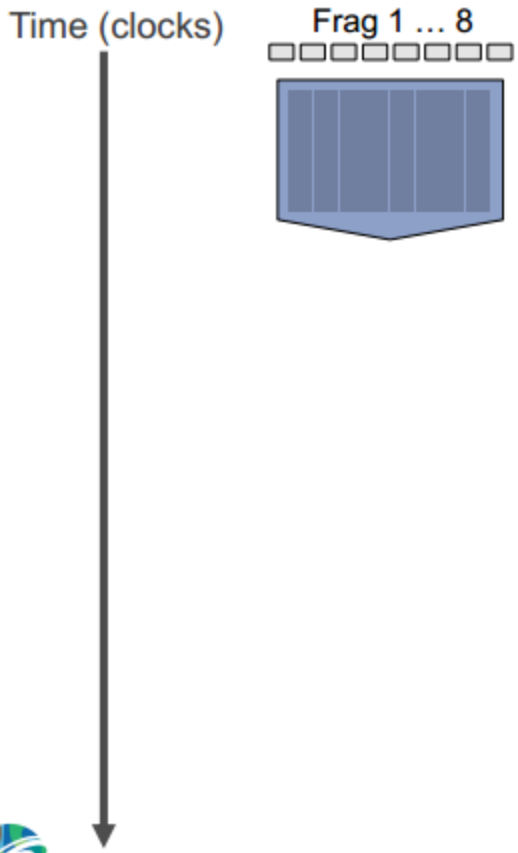
We've removed the fancy caches and logic that helps avoid stalls.

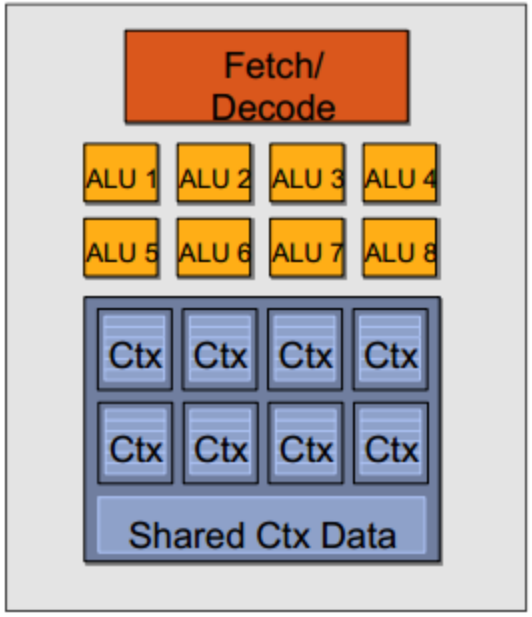
But we have LOTS of independent fragments.

Idea #3:

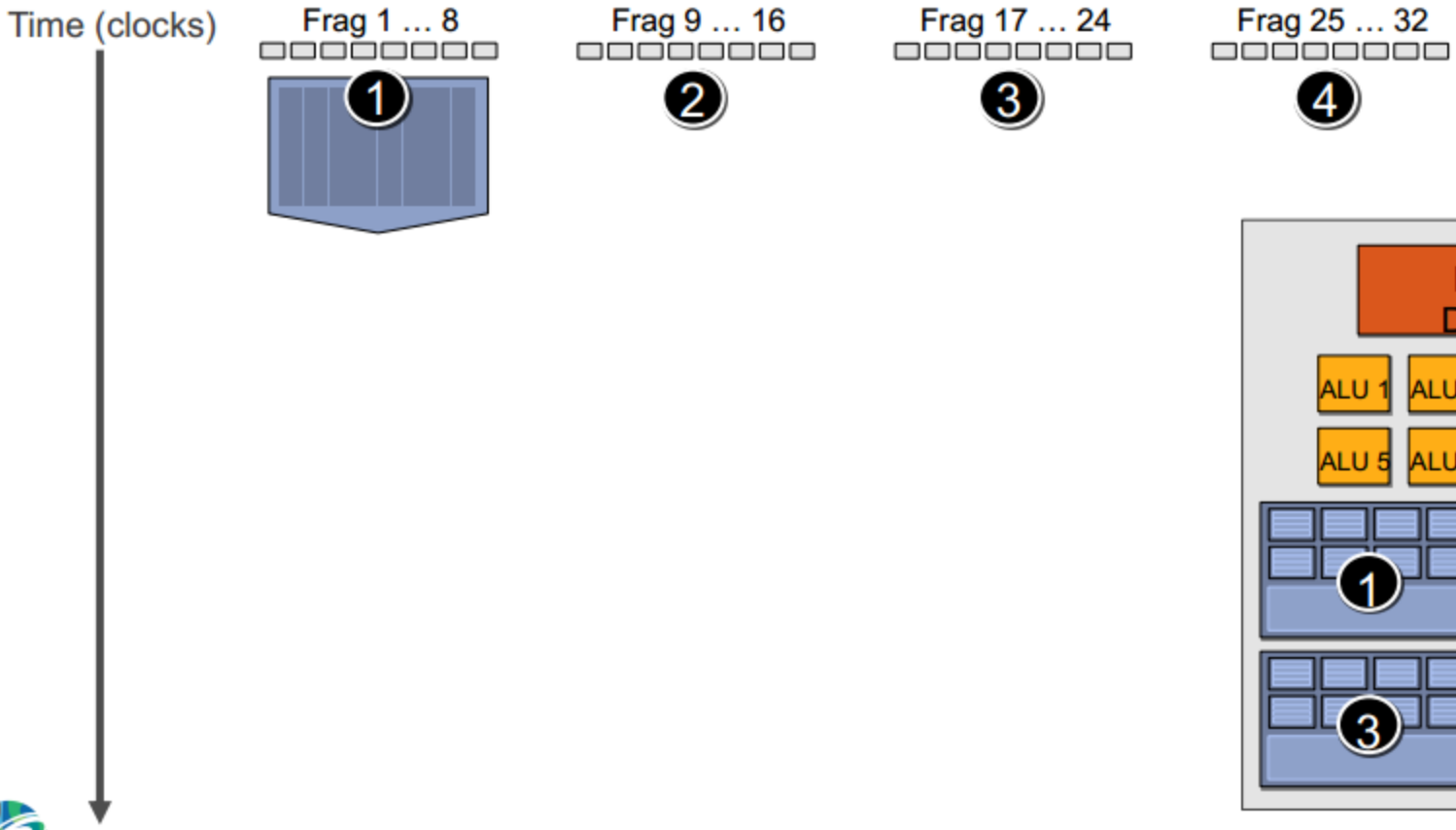
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

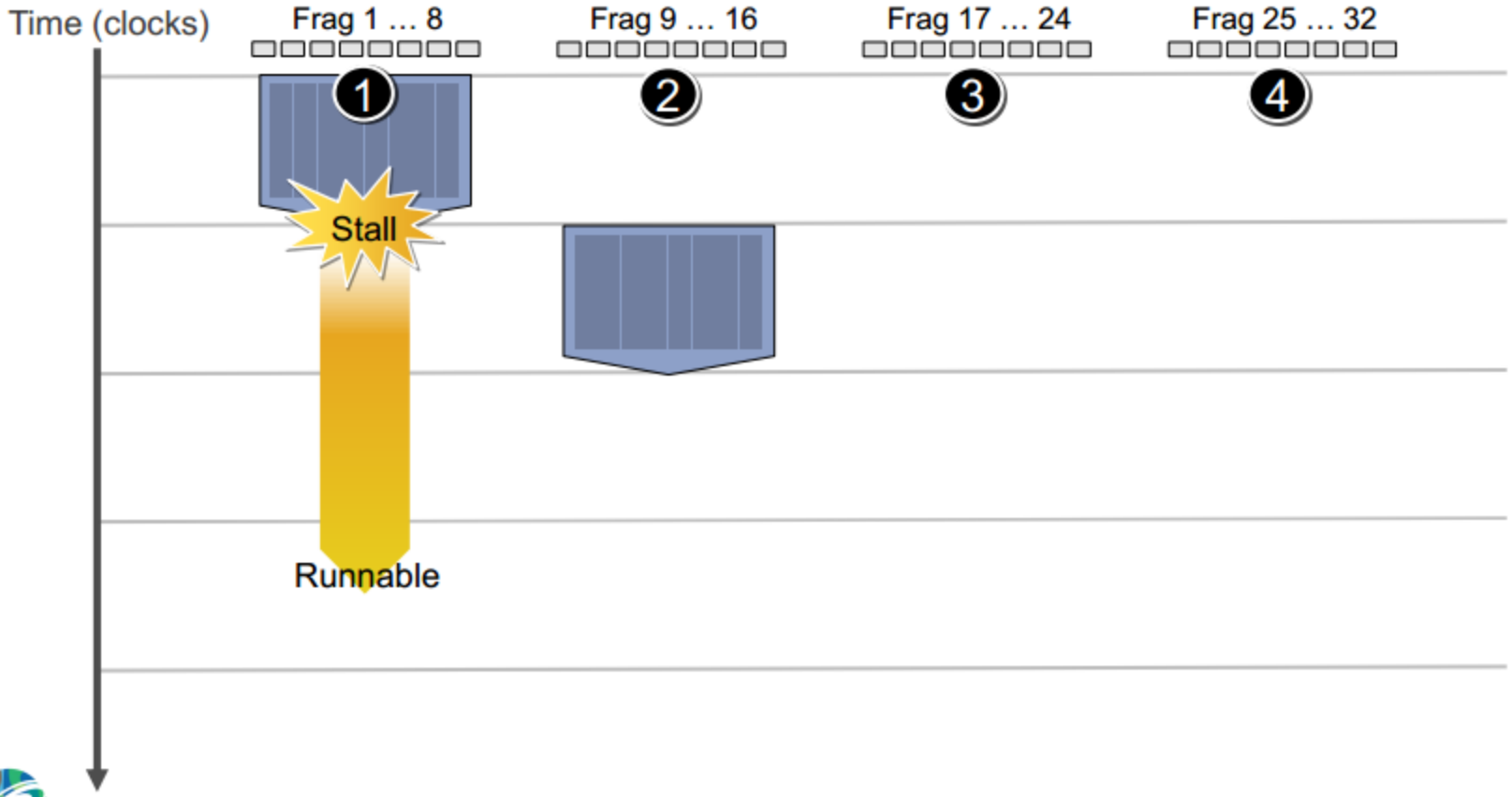




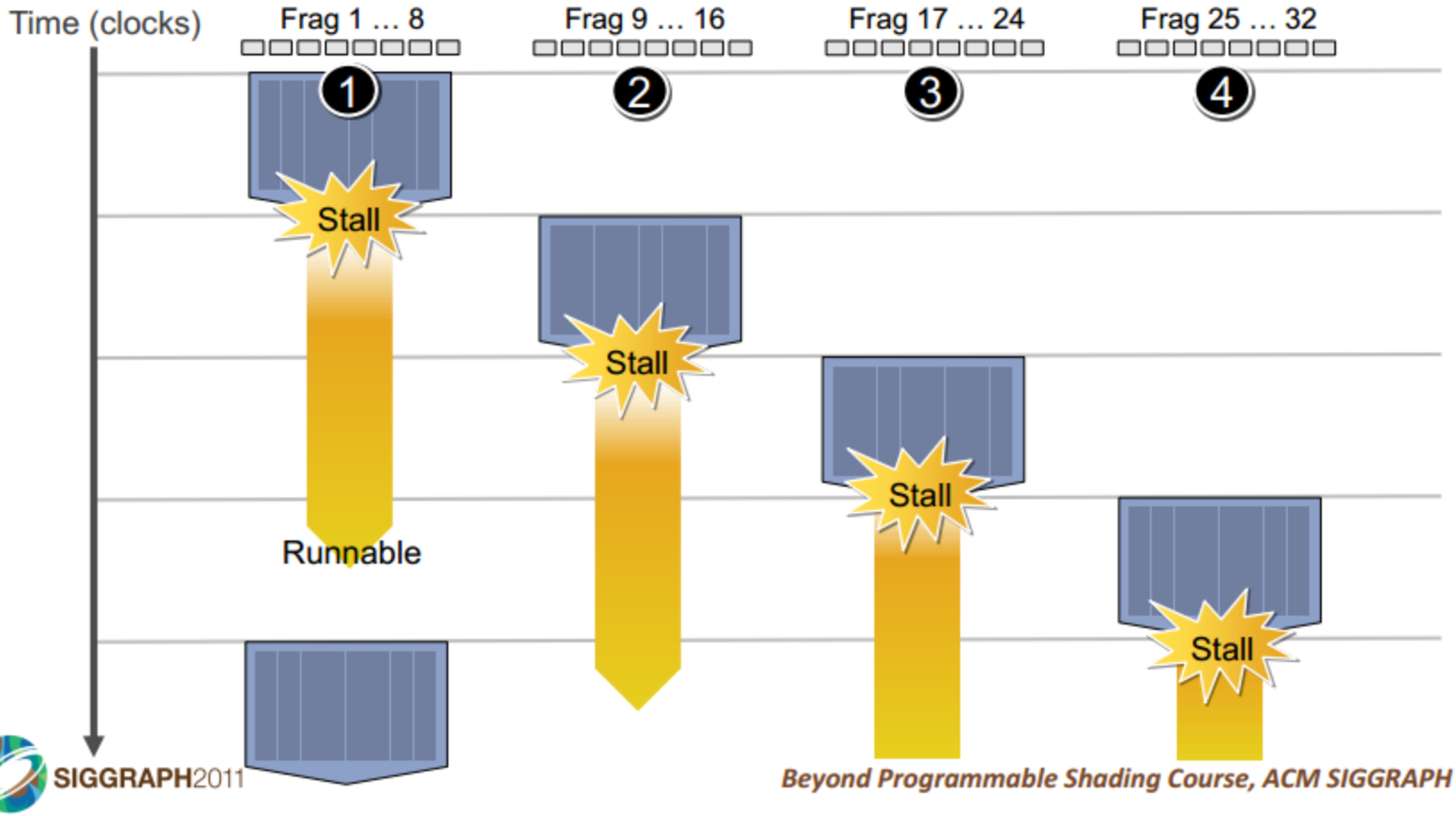
Hiding shader stalls



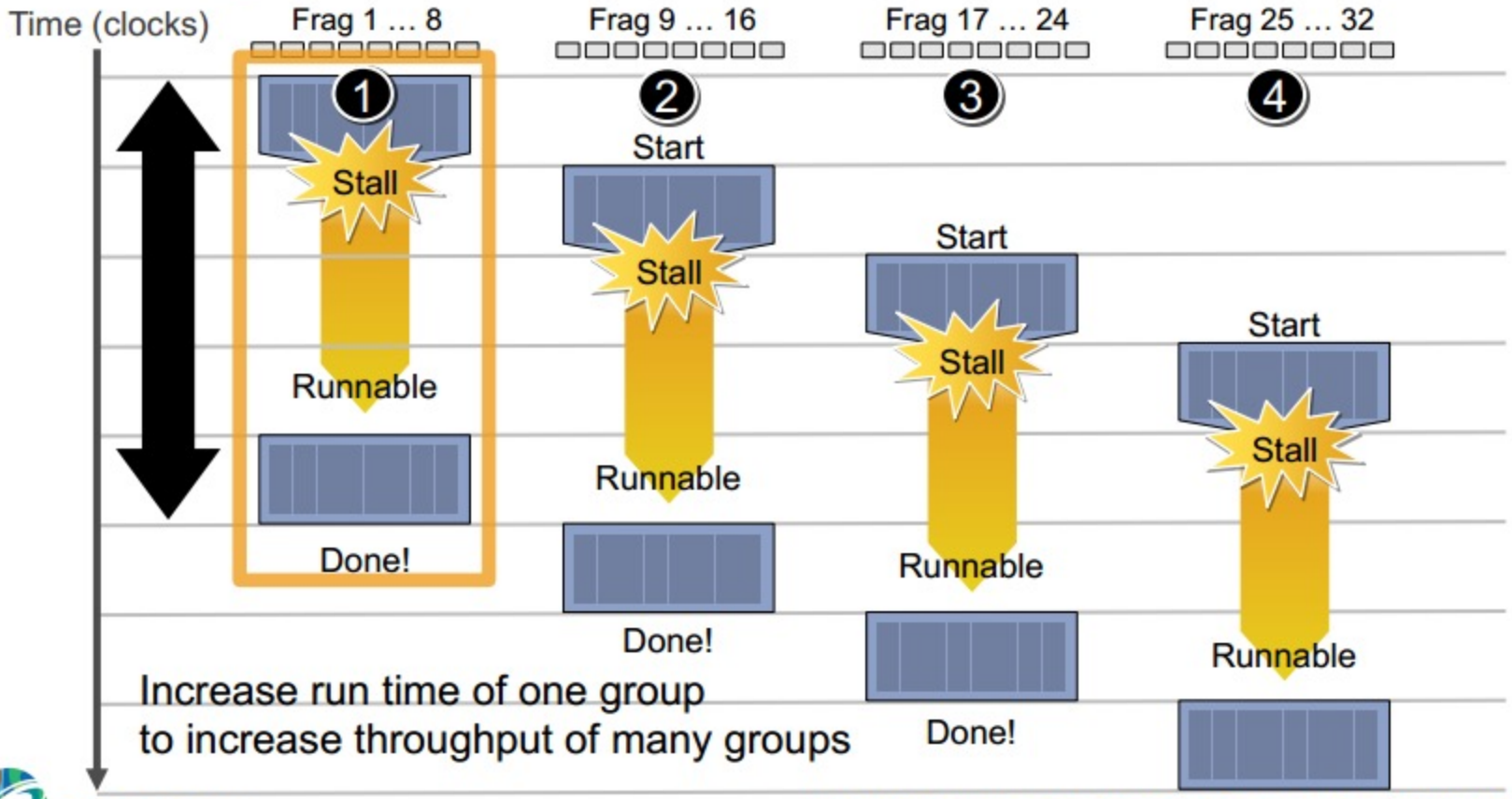
Hiding shader stalls



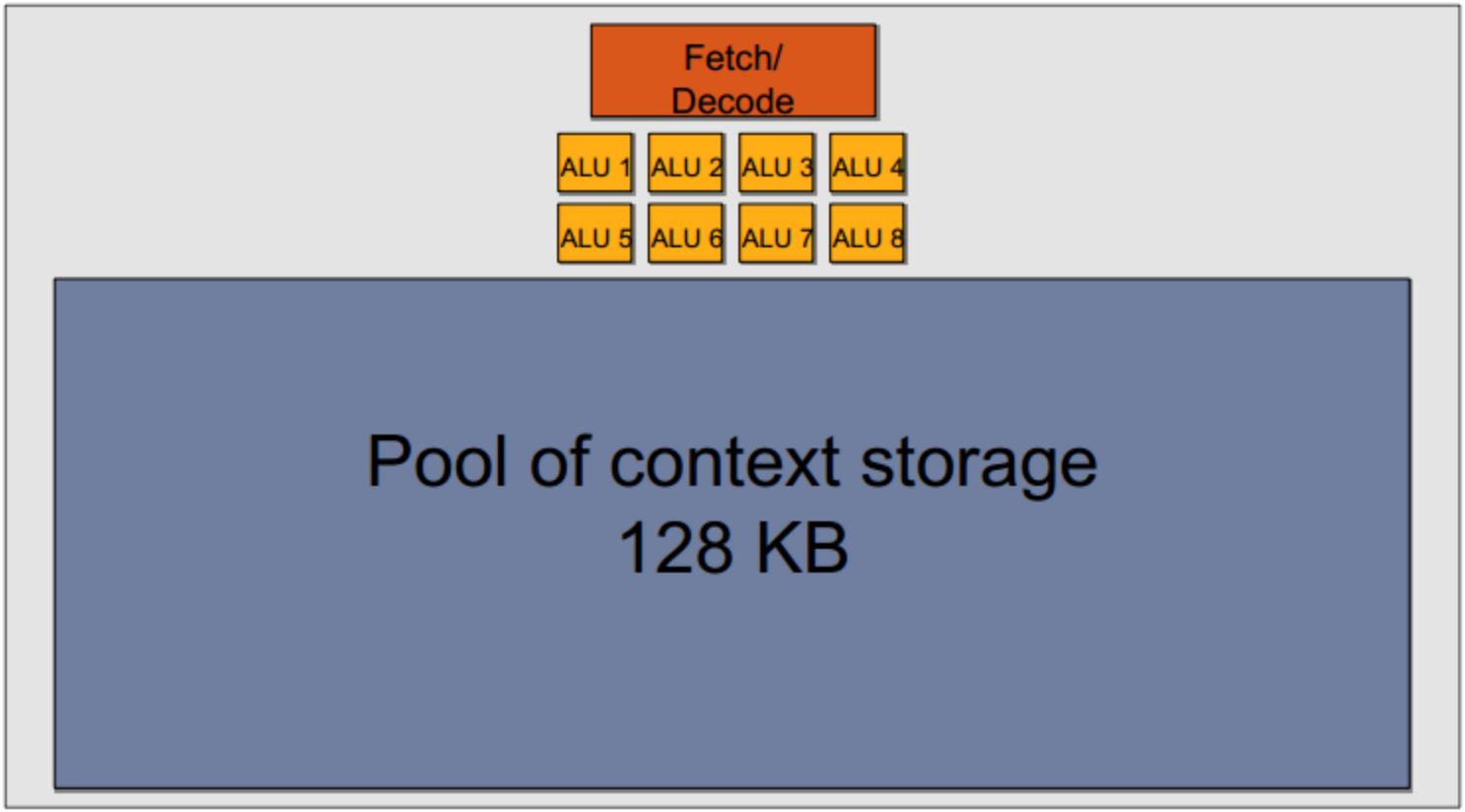
Hiding shader stalls



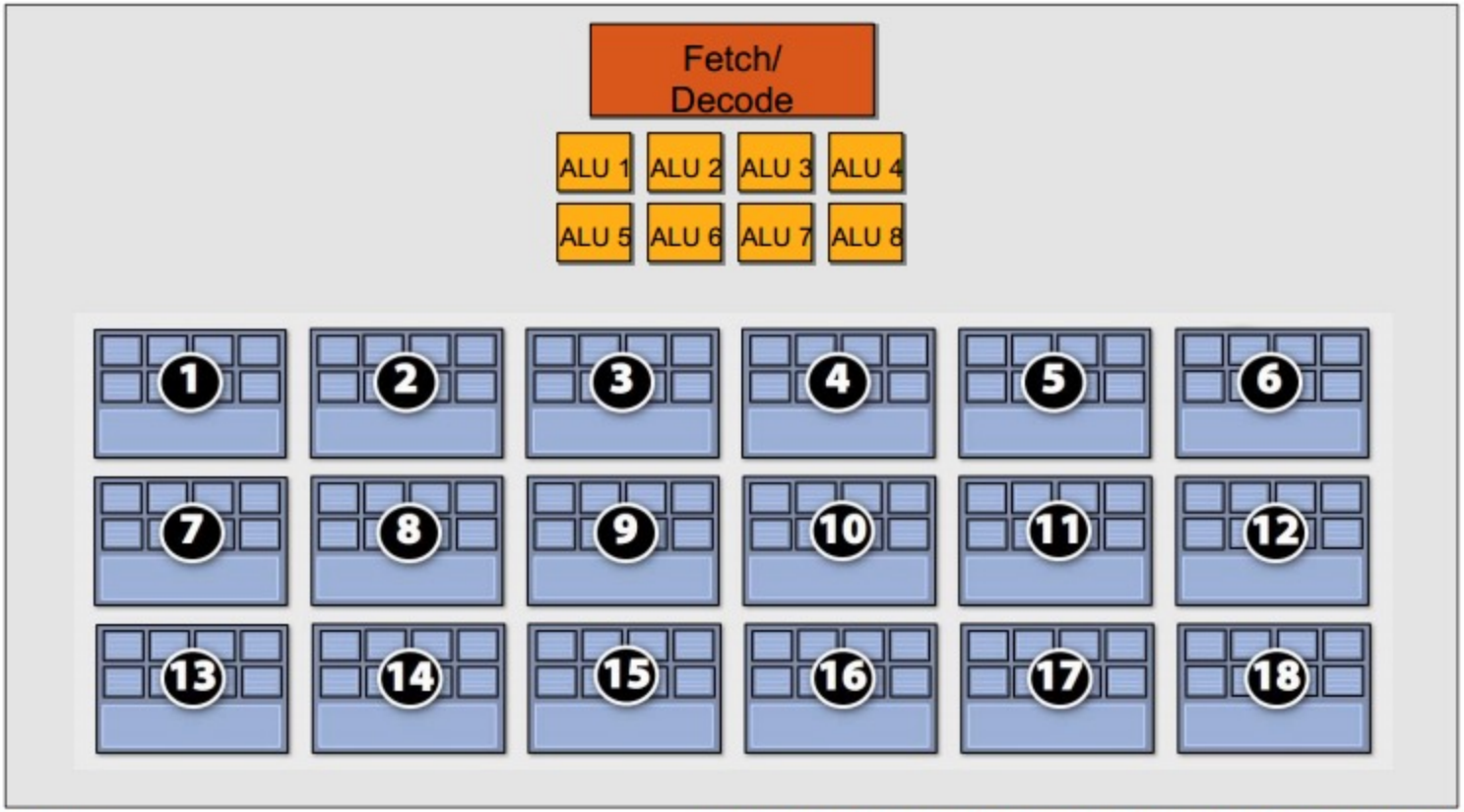
Throughput!


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

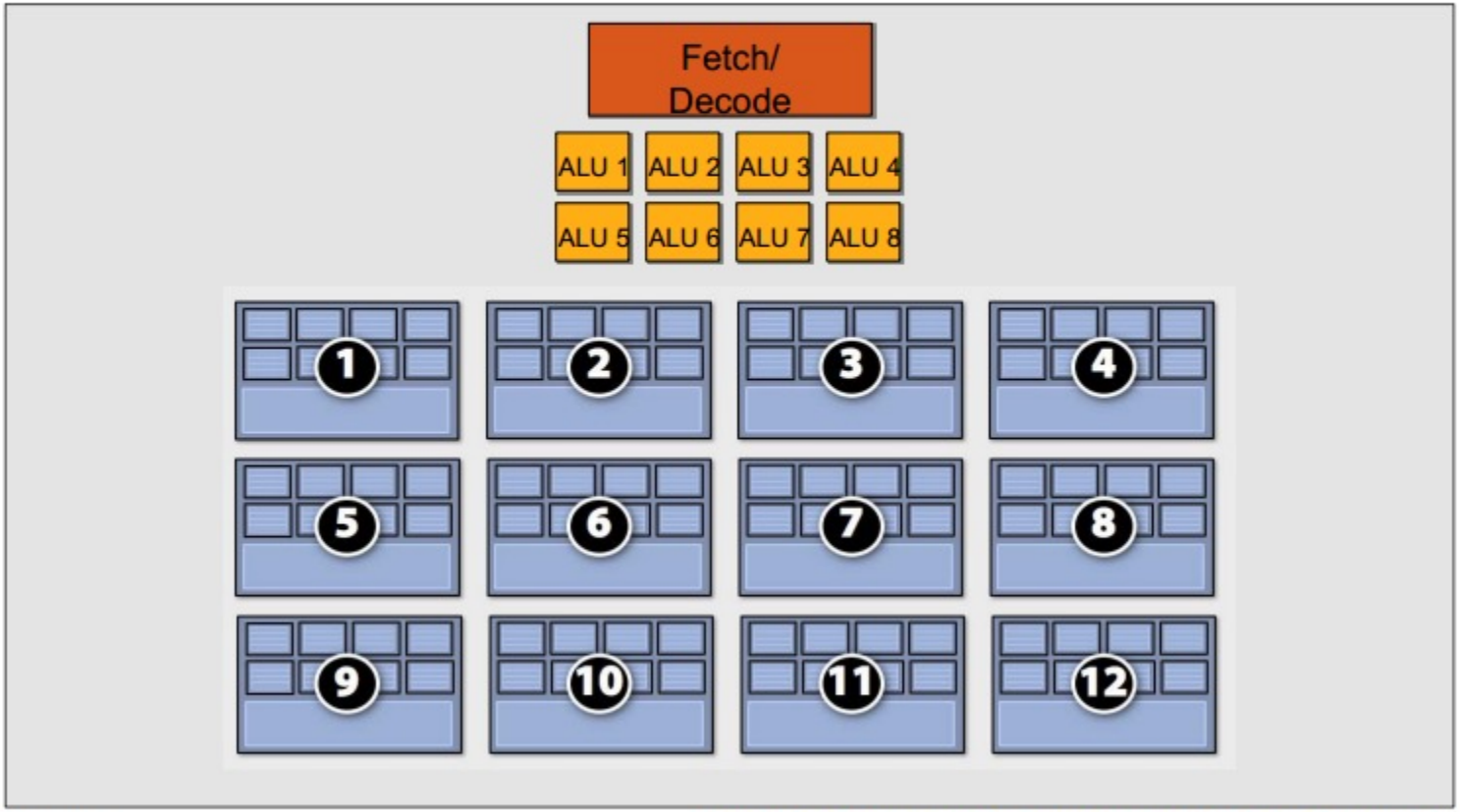
Storing contexts



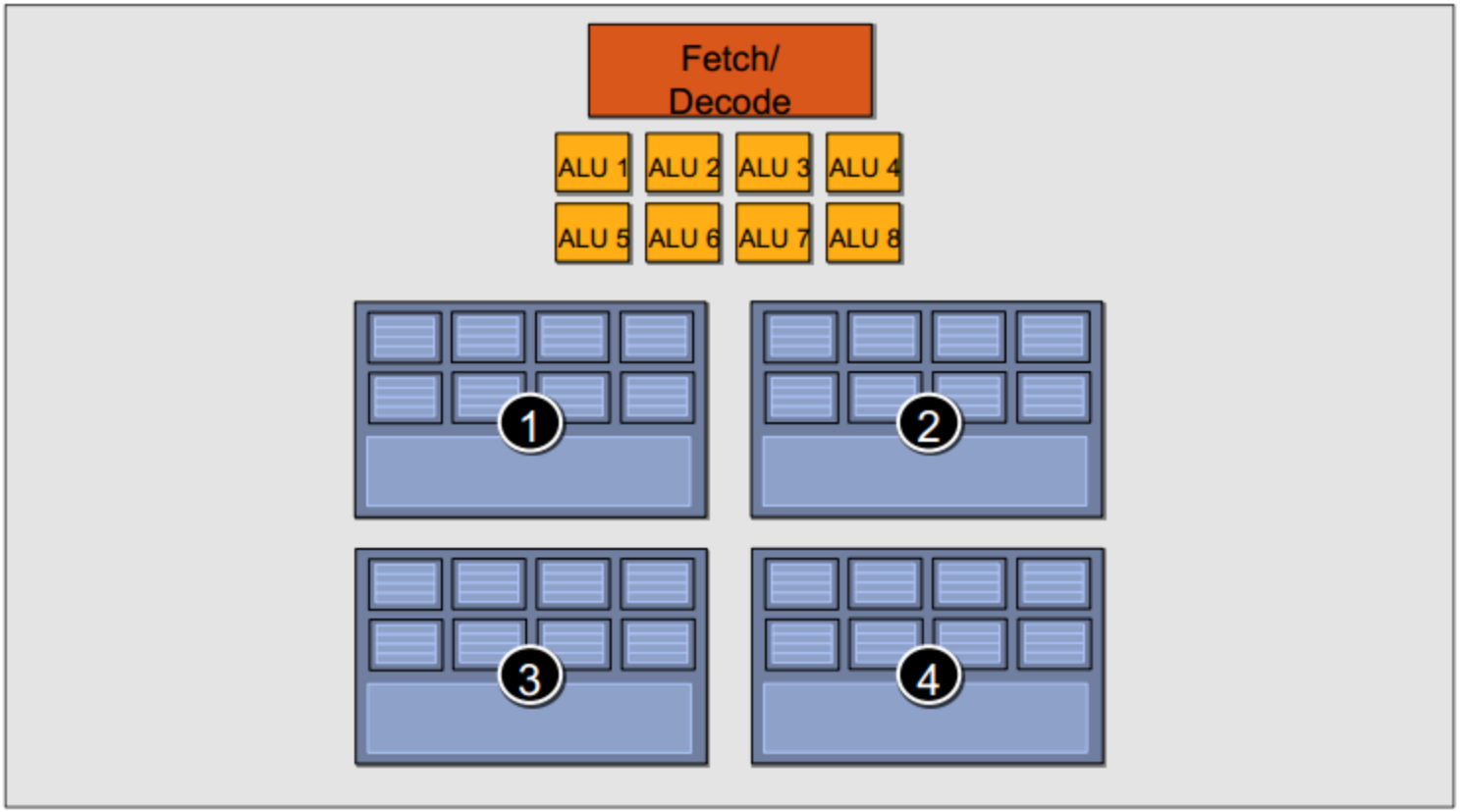
Eighteen small contexts (maximal latency hiding)



Twelve medium contexts



Four large contexts (low latency hiding ability)



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)

Example chip

16 cores

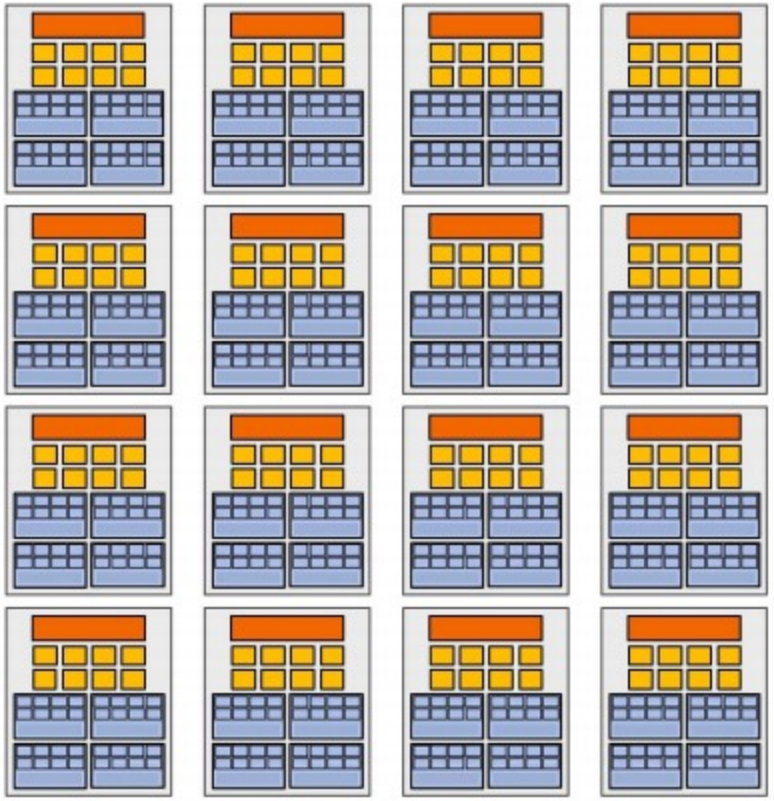
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



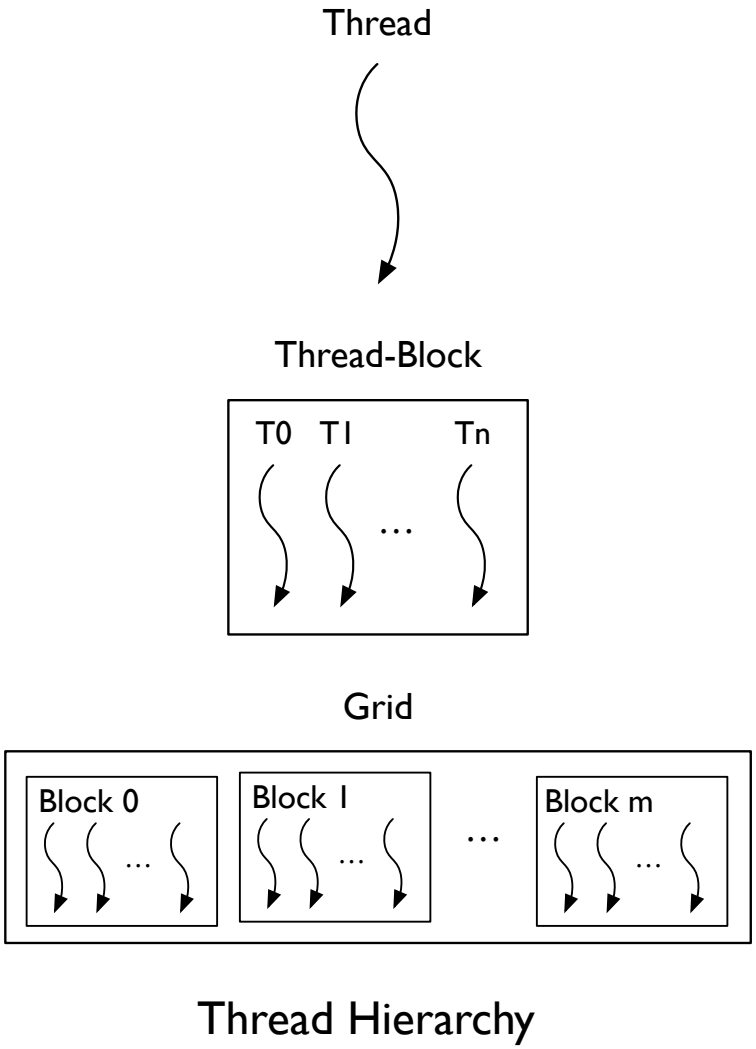
Summary: three key ideas

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

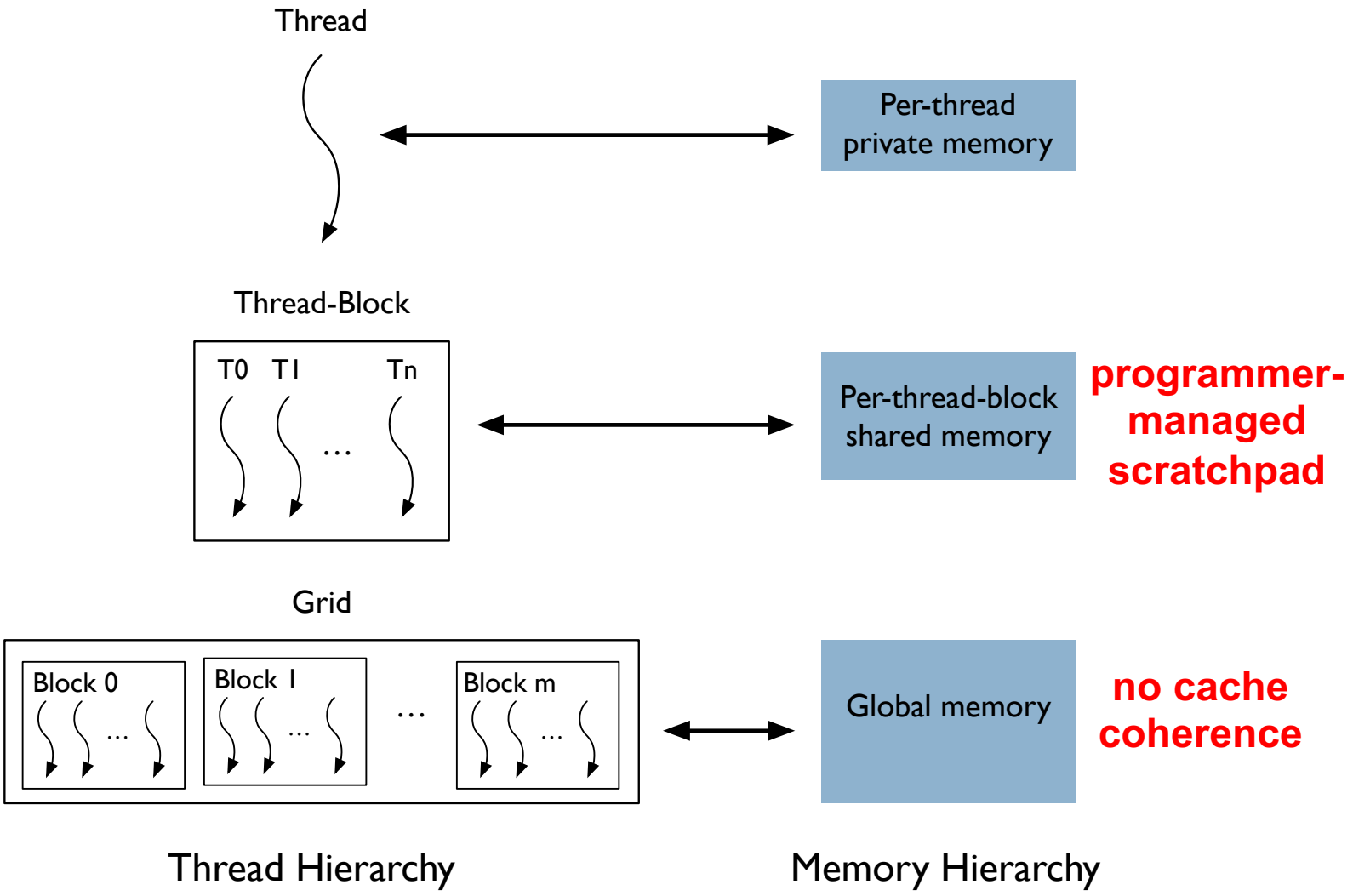
3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group

Nvidia CUDA Programming Model



Thread Hierarchy

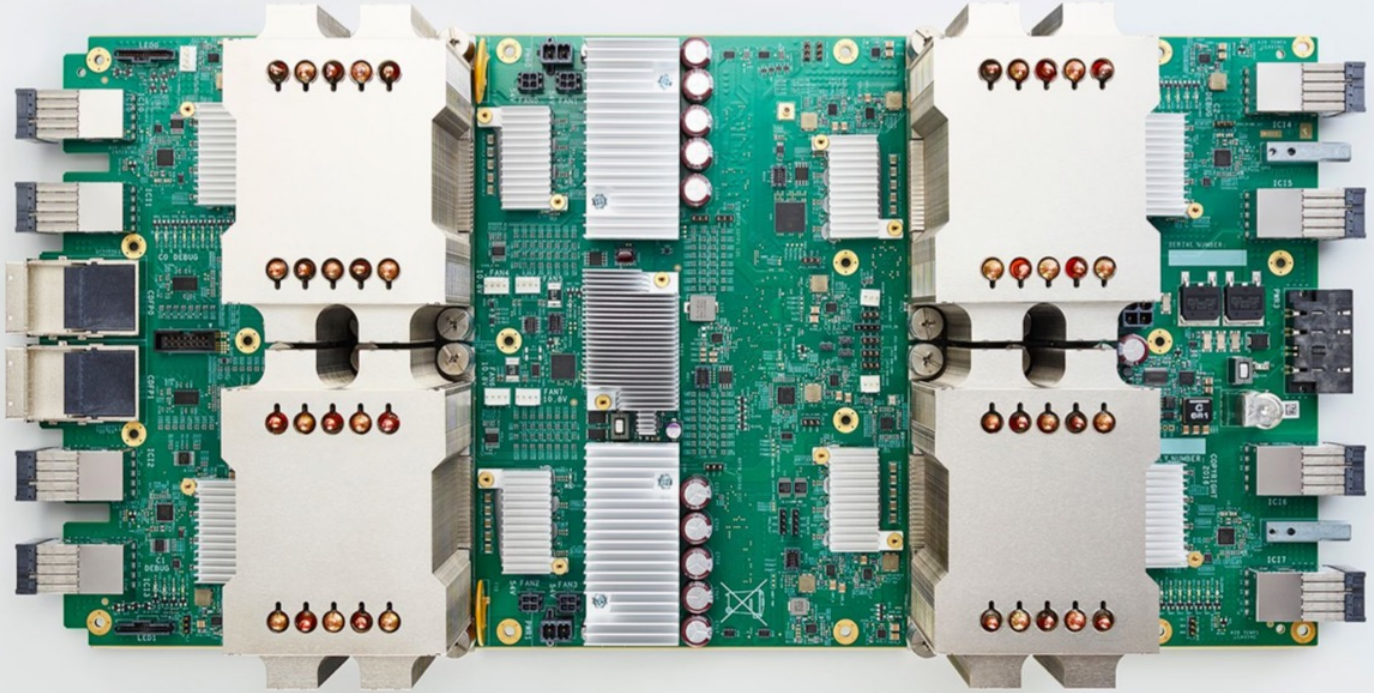
Nvidia CUDA Programming Model



Google Tensor Processing Unit (v2)

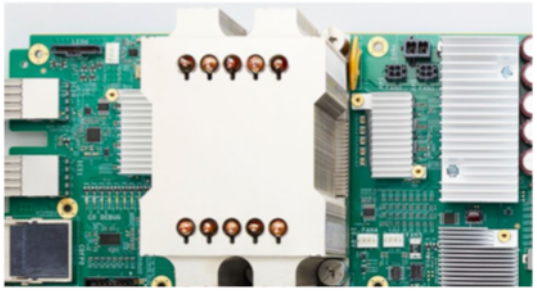
- Slides from HotChips 2017
  - https://www.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.22-Tuesday-Pub/HC29.22.69-Key2-AI-ML-Pub/HotChips%20keynote%20Jeff%20Dean%20-%20August%202017.pdf

Tensor Processing Unit v2

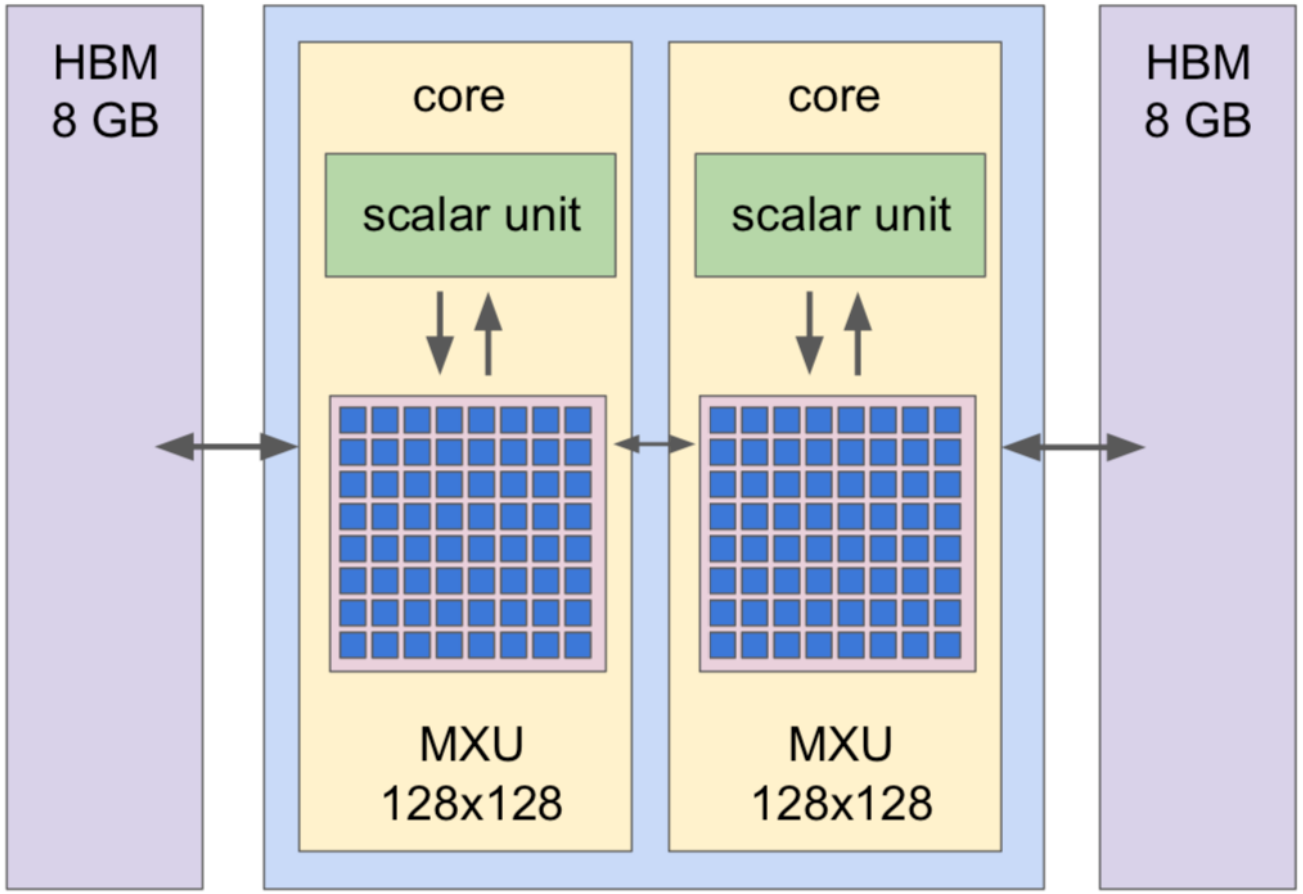


Google-designed device for neural net **training** and **inference**

TPUv2 Chip



- 16 GB of HBM
- 600 GB/s mem BW
- Scalar unit: 32b float
- MXU: 32b float accumulation but reduced precision for multipliers
- 45 TFLOPS



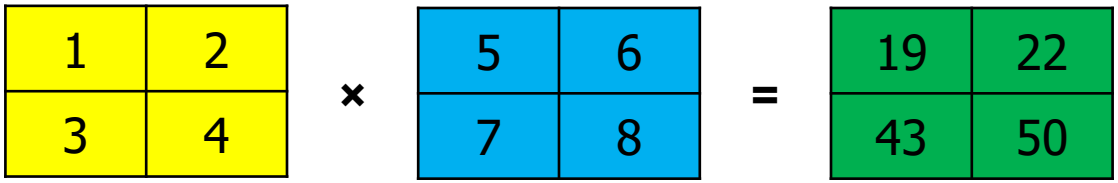
TPU v1 ISA

- 5 main (CISC) instructions
  `Read_Host_Memory`
  `Write_Host_Memory`
  `Read_Weights`
  `MatrixMultiply/Convolve`
  `Activate(ReLU,Sigmoid,Maxpool,LRN,…)`
- Average Clock cycles per instruction: >10
- 4-stage overlapped execution, 1 instruction type / stage
  - Execute other instructions while matrix multiplier busy
- Complexity in SW: No branches, in-order issue,
  SW controlled buffers, SW controlled pipeline synchronization

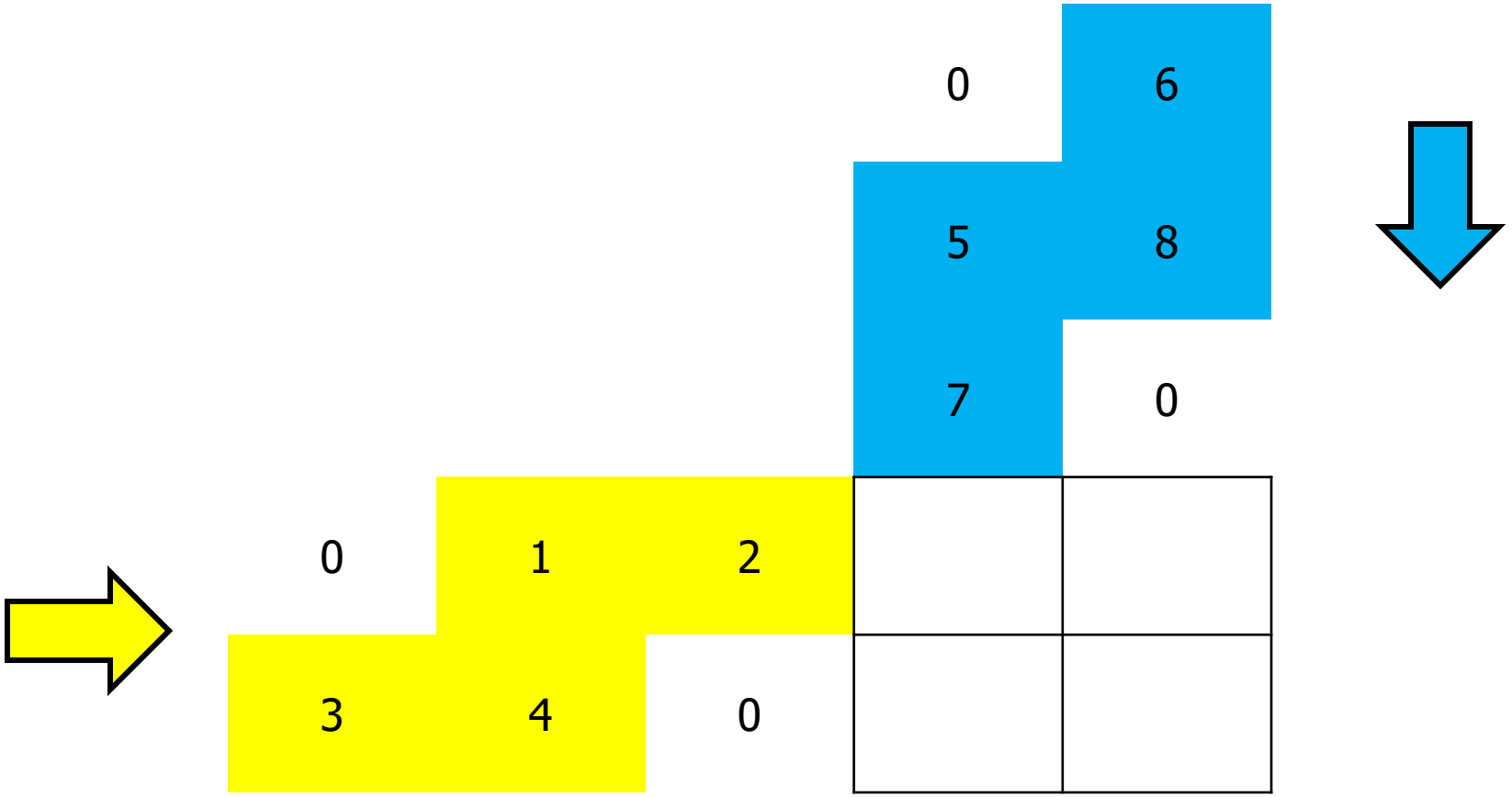
TPU Architecture, programmer's view

Systolic Array Matrix Multiply

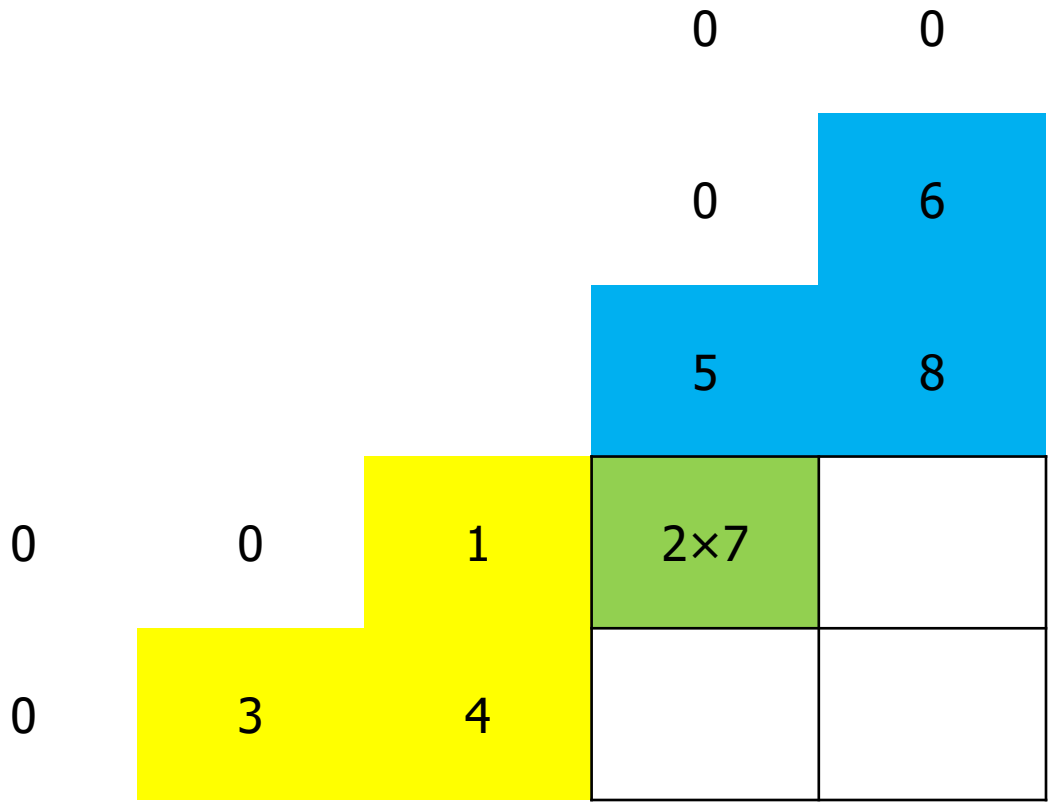
- H. T. Kung, C. E. Leiserson. *Algorithms for VLSI processor arrays*, 1979
- "pump" the data through the functional units, accumulating the dot products in place
- lots of parallel compute with low data movement



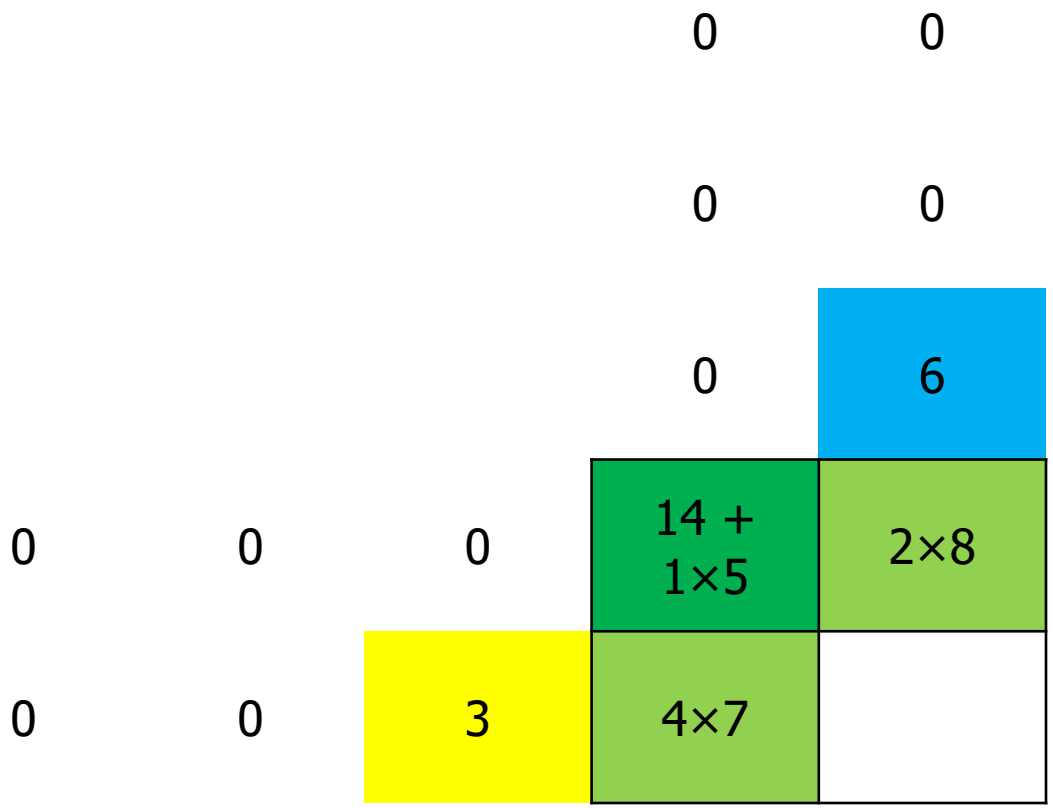
Systolic Array Matrix Multiply Example



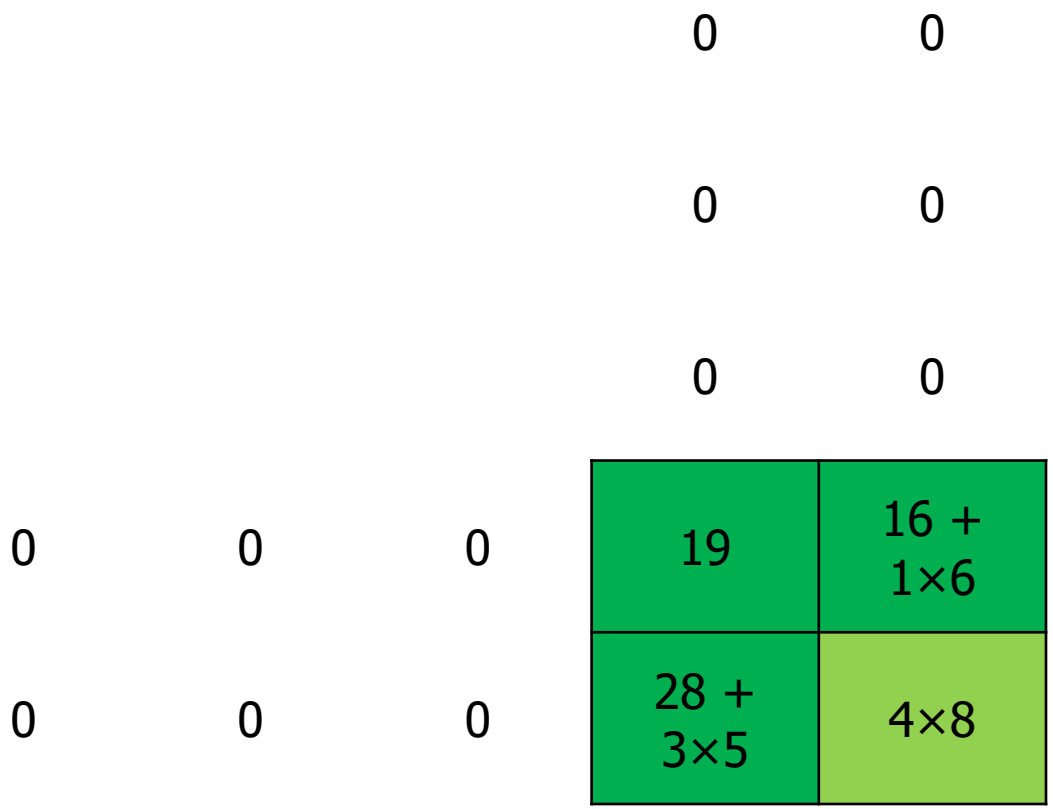
Systolic Array Matrix Multiply Example



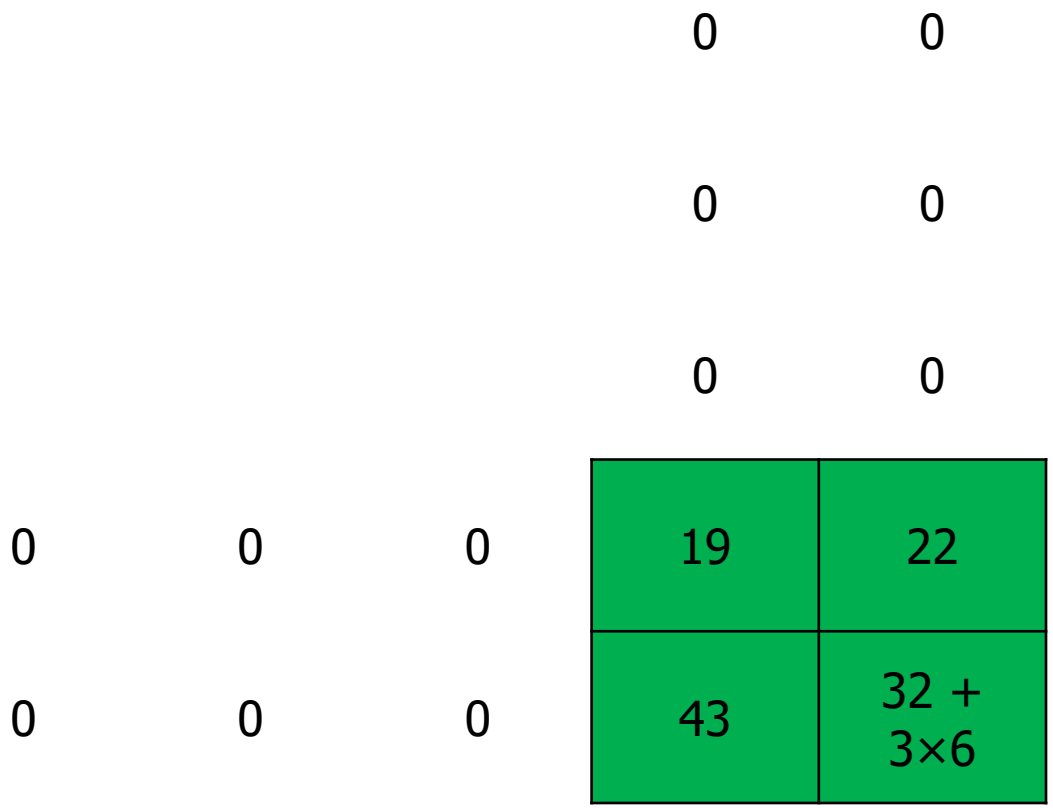
Systolic Array Matrix Multiply Example



Systolic Array Matrix Multiply Example



Systolic Array Matrix Multiply Example



Accelerators Summary

- Data Level Parallelism
  - "medium-grained" parallelism between ILP and TLP
  - Still one flow of execution (unlike TLP)
  - Compiler/programmer must explicitly express it (unlike ILP)
- GPUs
  - Embrace data parallelism via "SIMT" execution model
  - Becoming more programmable all the time
- TPUs
  - Neural network accelerator
  - Fast matrix multiply machine
- Slow growth in single-thread performance, Moore's Law drives adoption of accelerators