CS 6023 - GPU Programming

Parallel Computing Architecture

24/01/2019

Agenda

How do we contrast CPUs from GPUs?

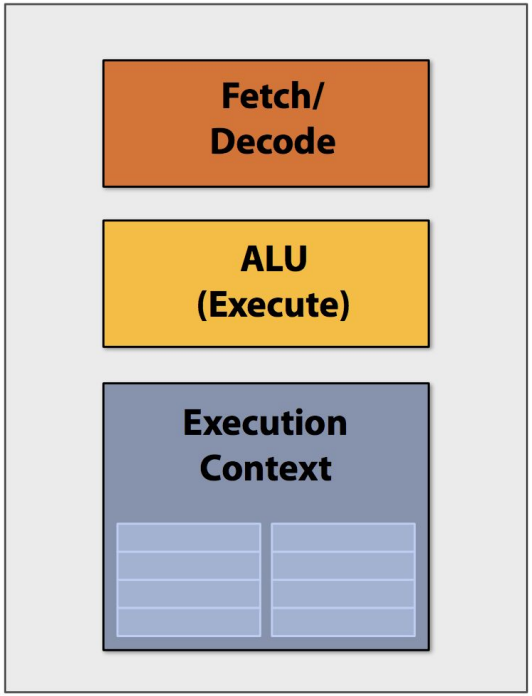
Part 1 of 2:

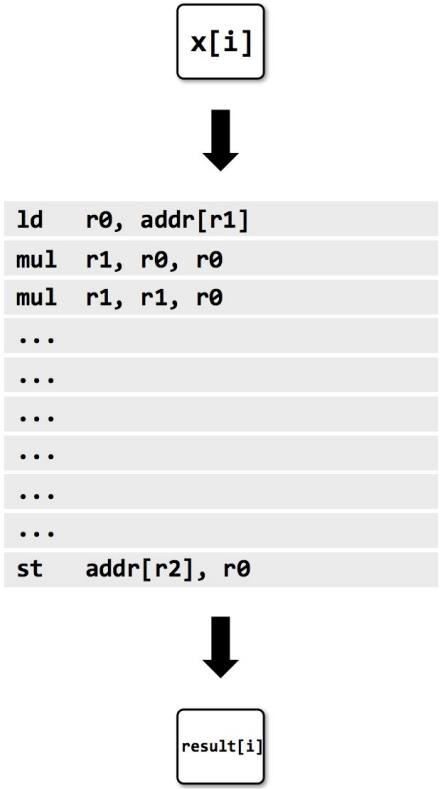
Last time: CPU architecture background

Part 2 of 2:

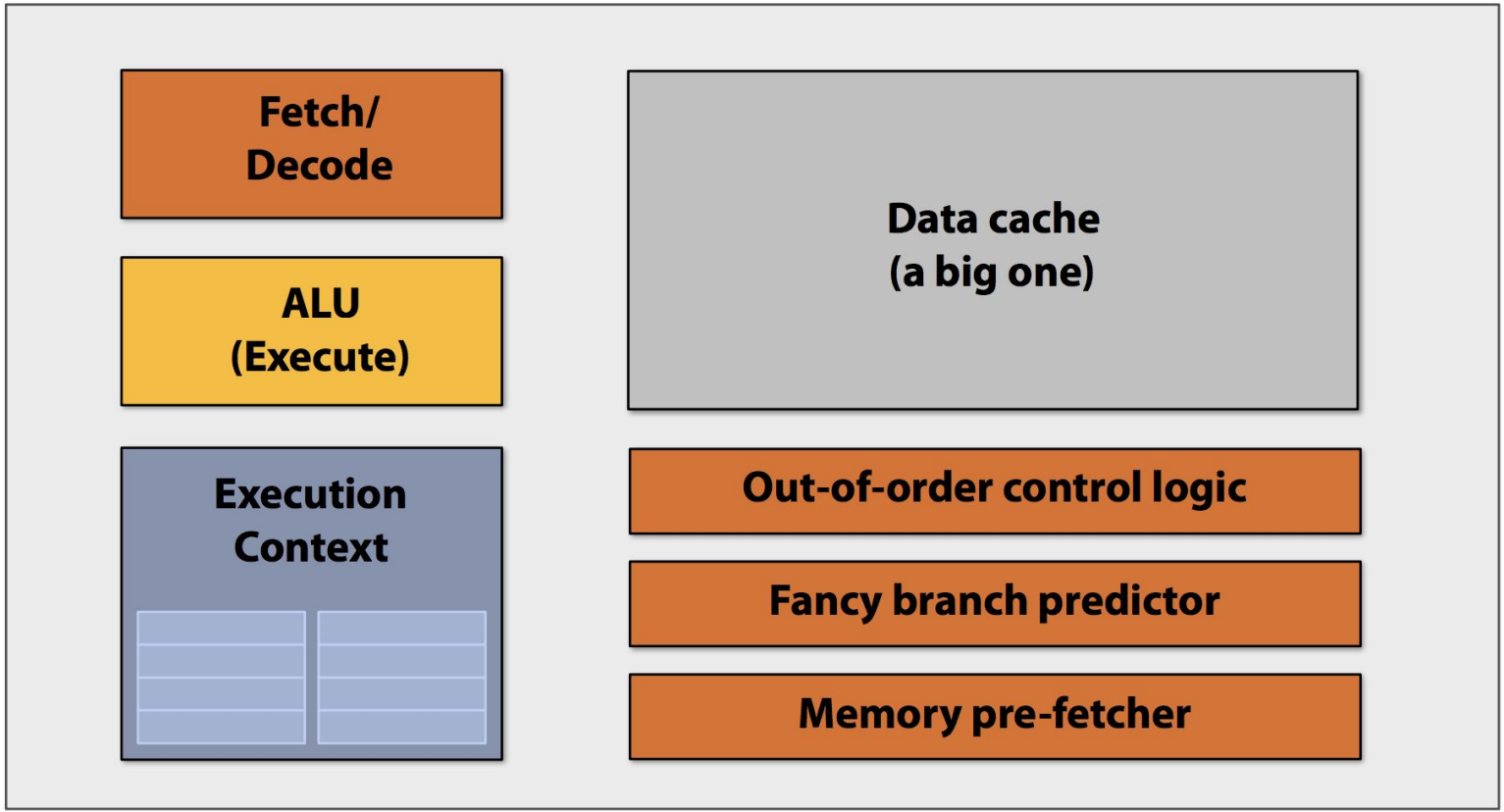
Today: Parallel computer architecture background

CPU - single core cartoon

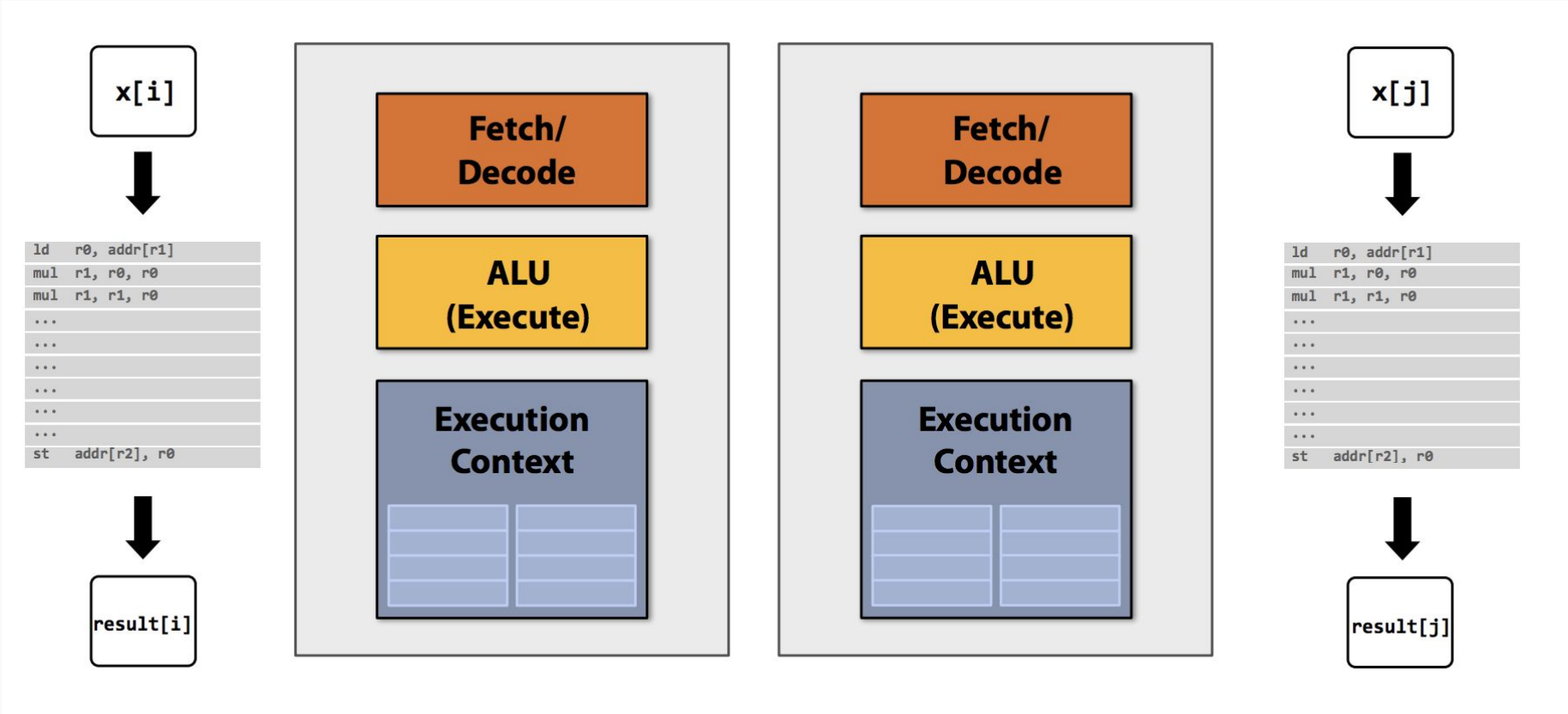




CPU cartoon - single core era

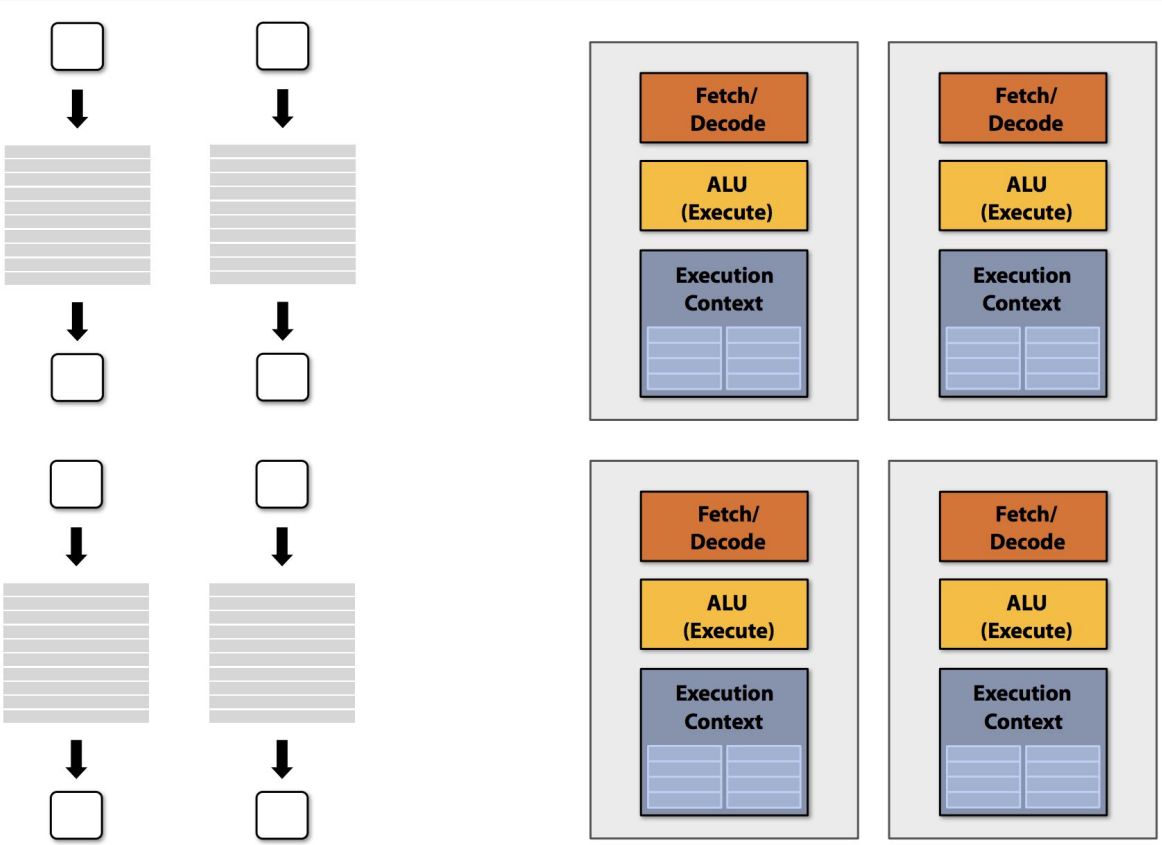


CPU cartoon - multi-core

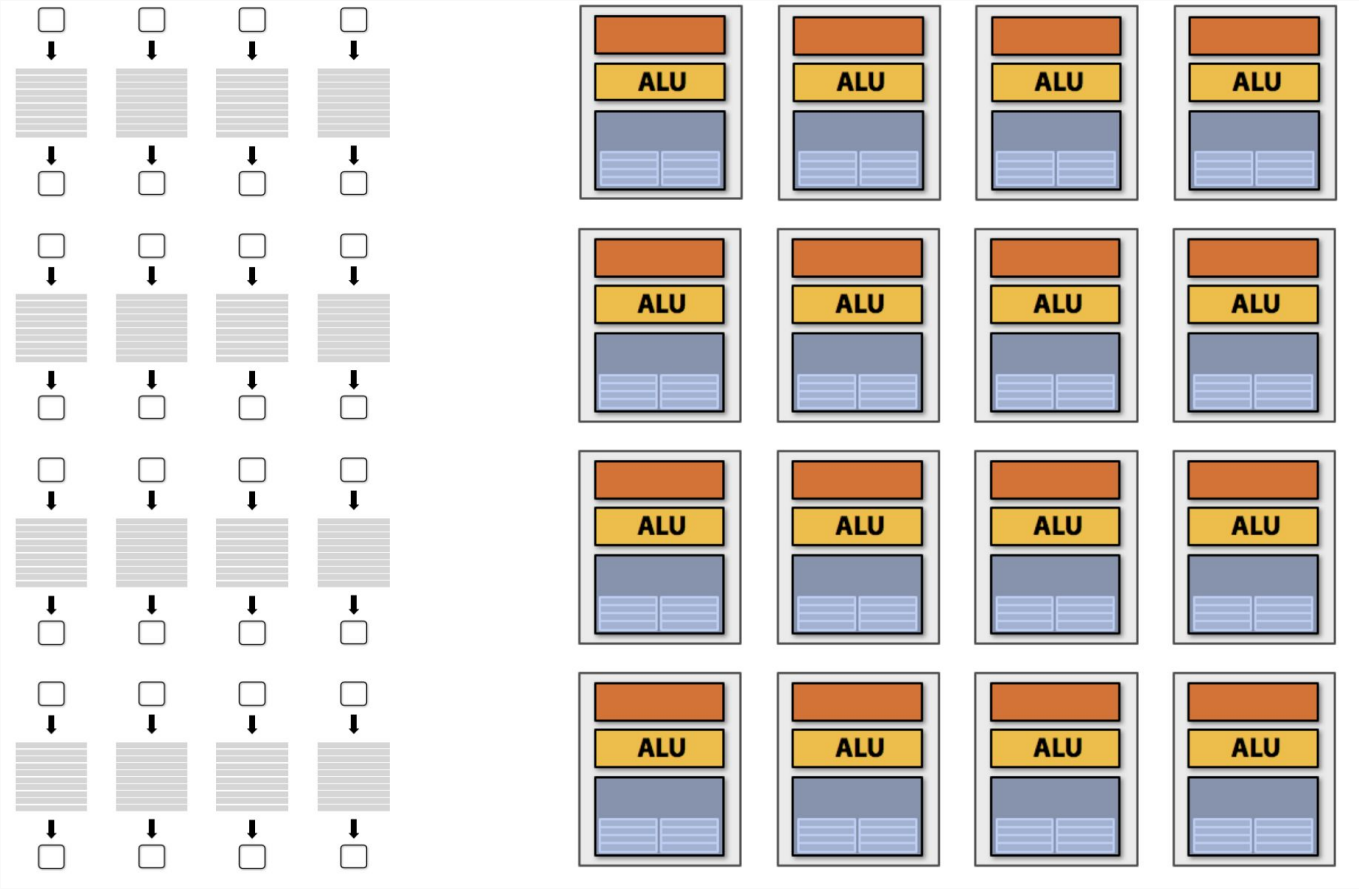


Make cores simpler, have multiple of them

CPU cartoon - many core

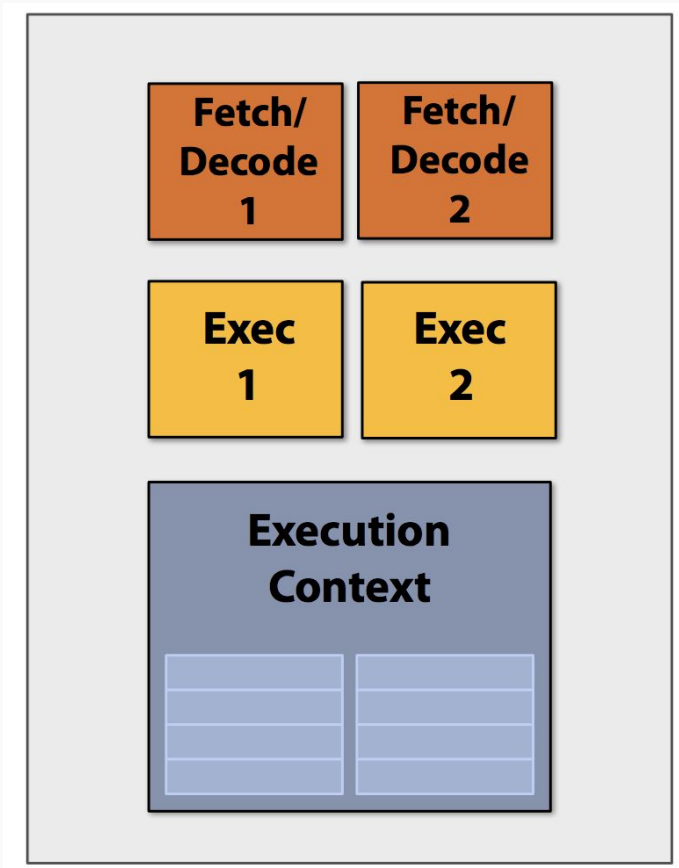


CPU cartoon - many core



Why don’t we keep doing this for better and better CPUs?

What is this?

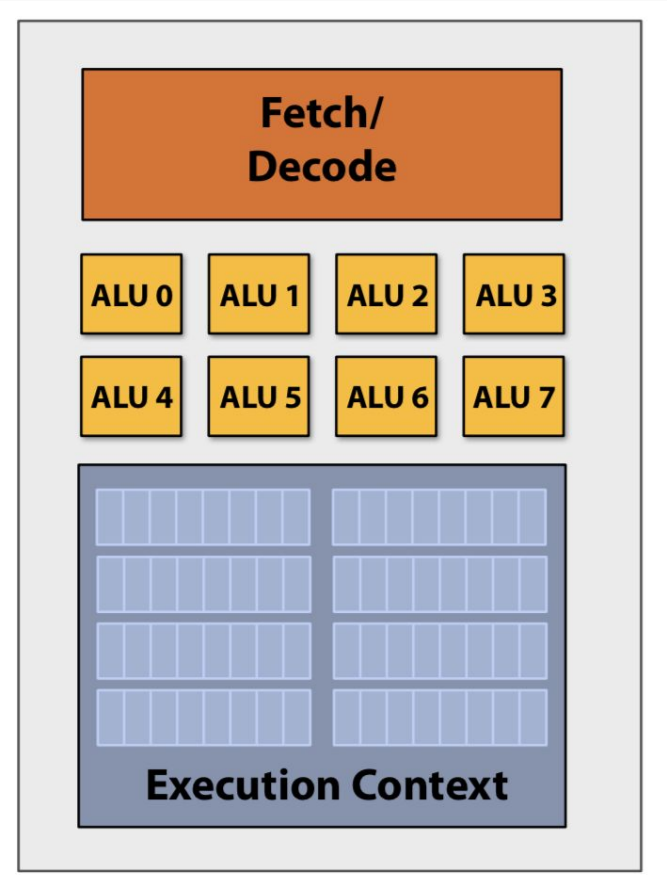


Superscalar

When is this useful?

- For exploiting Instruction Level Parallelism (ILP)

What is this?

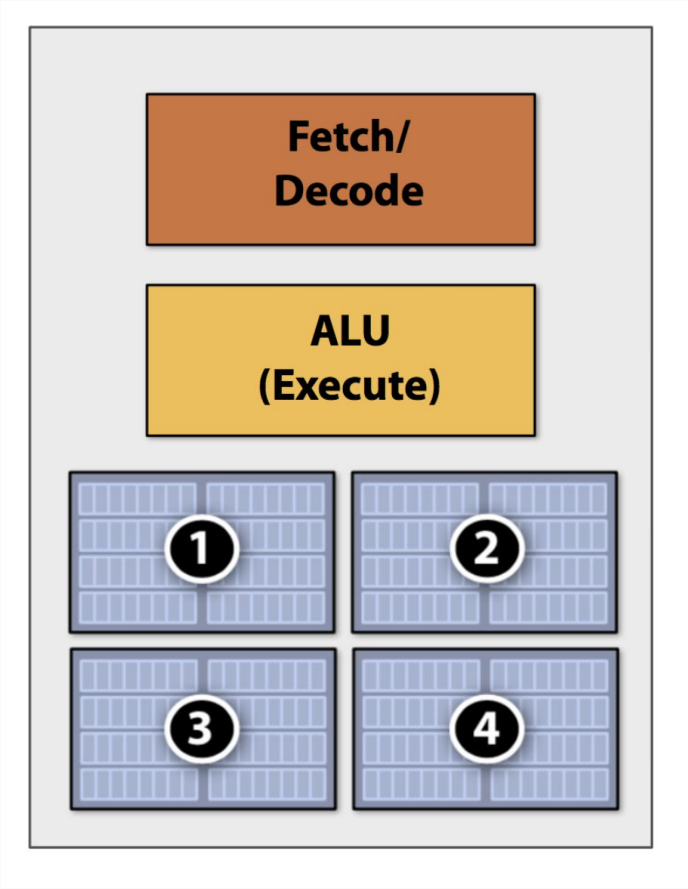


Single Instruction Multiple Data
SIMD processing

When is this useful?

- Explicit SIMD
- Vector processing

What is this?

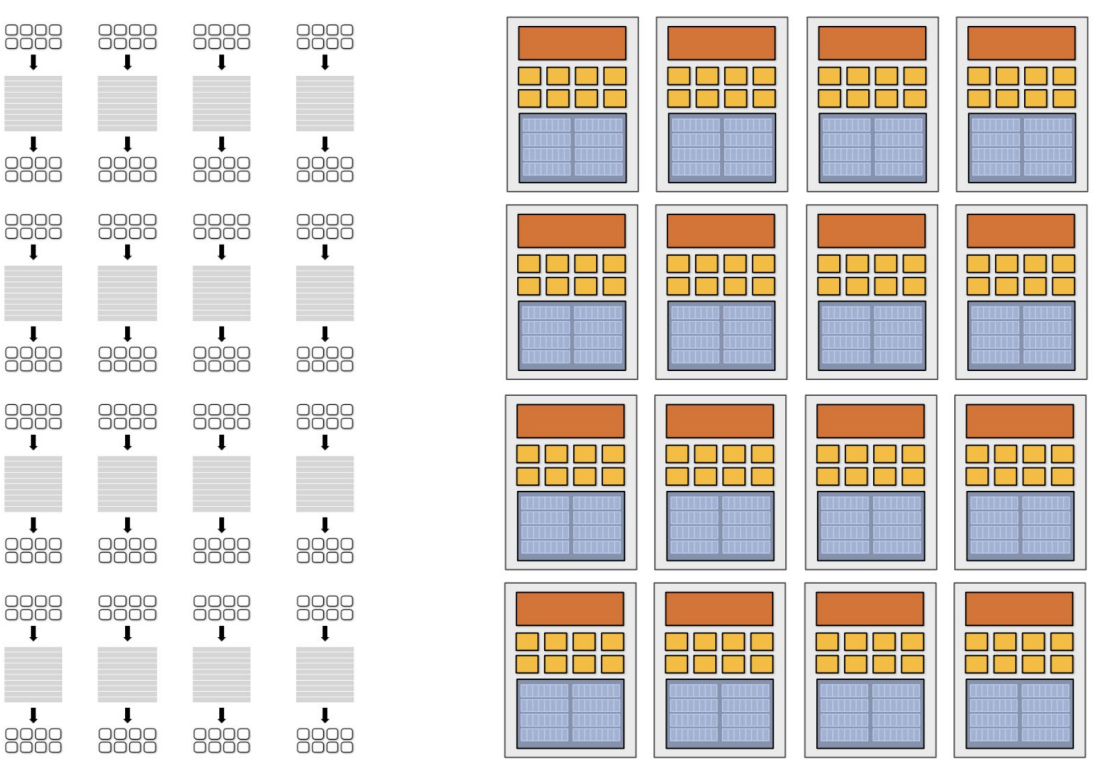


Multithreading

When is this useful

- For hiding latency during stalls

Combining multicore and SIMD



How many separate instruction streams does this process

16 cores, 16 simultaneous instruction streams

SIMD processing

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

SIMD processing

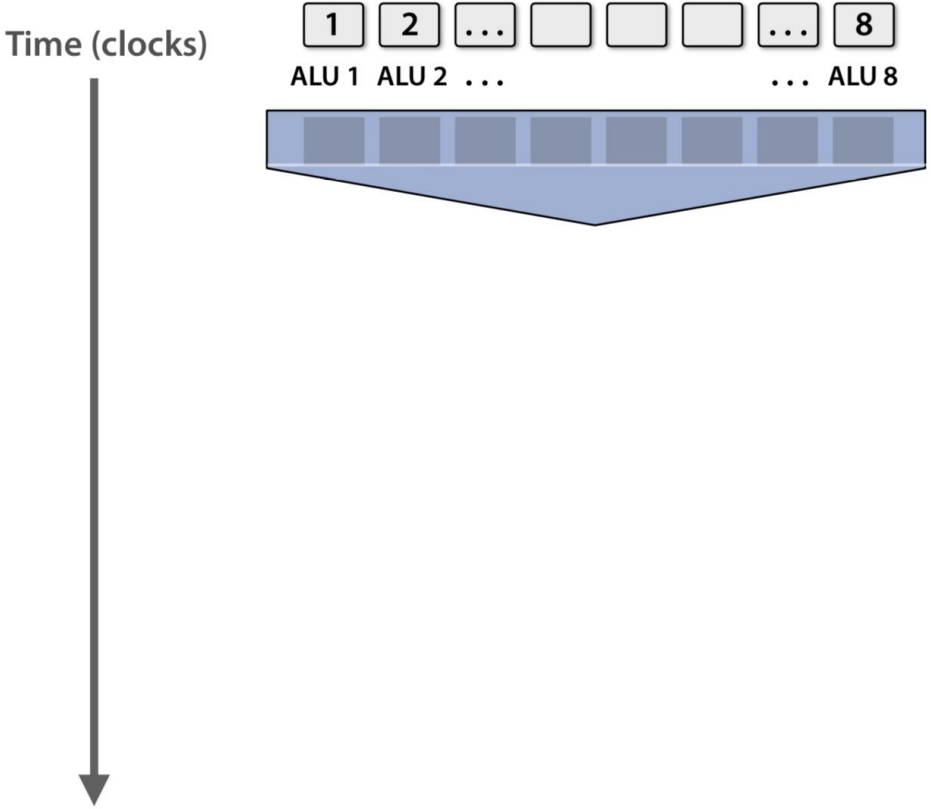
```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Each loop has same set of instructions to be applied independently on different sets of data

=> SIMD processing

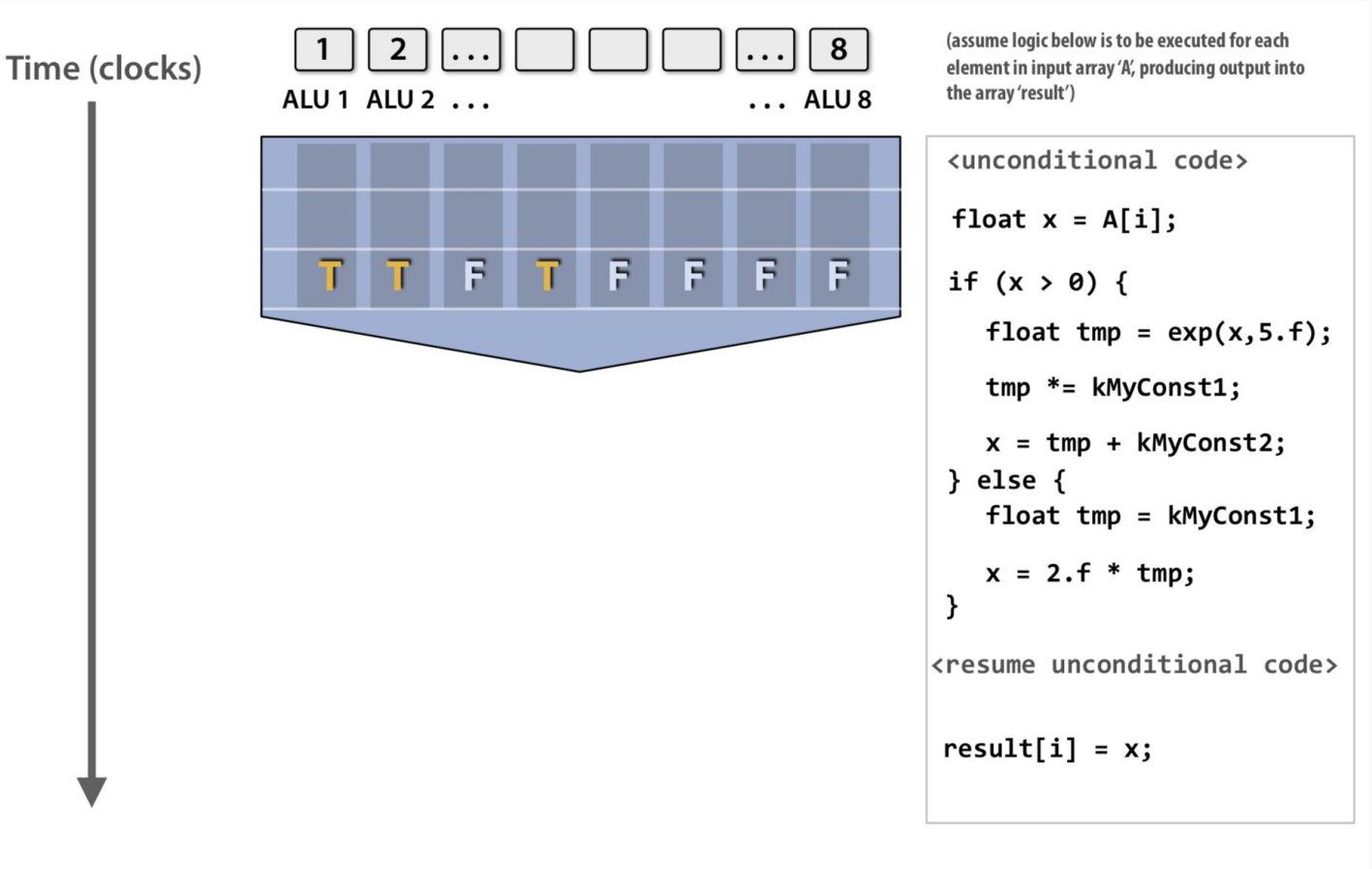
But we have a problem with conditions



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);

   tmp *= kMyConst1;

   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;

   x = 2.f * tmp;
}

<resume unconditional code>


result[i] = x;
```

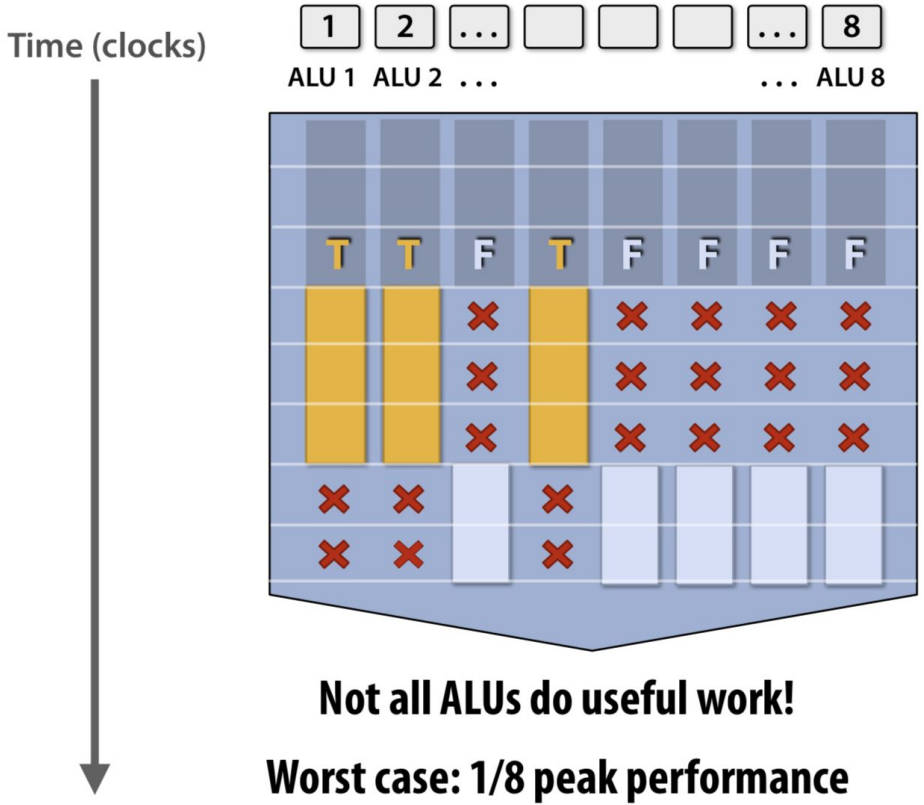
But we have a problem with conditions



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);

   tmp *= kMyConst1;

   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;

   x = 2.f * tmp;
}

<resume unconditional code>


result[i] = x;
```

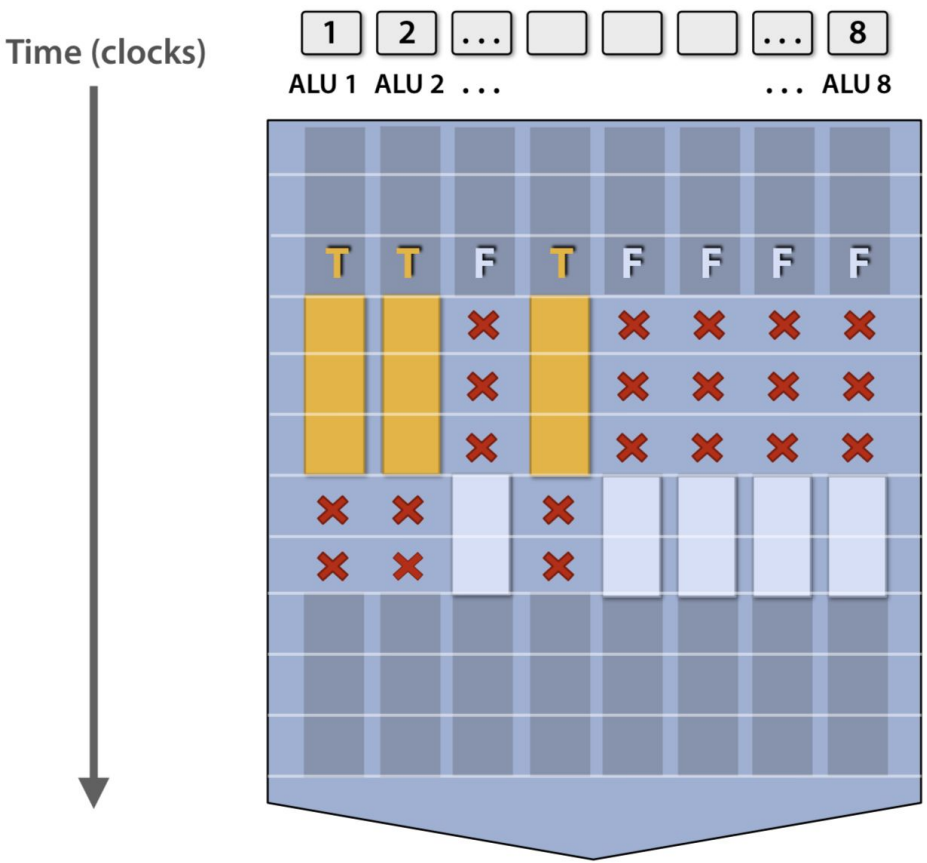
But we have a problem with conditions



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
    float tmp = exp(x,5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

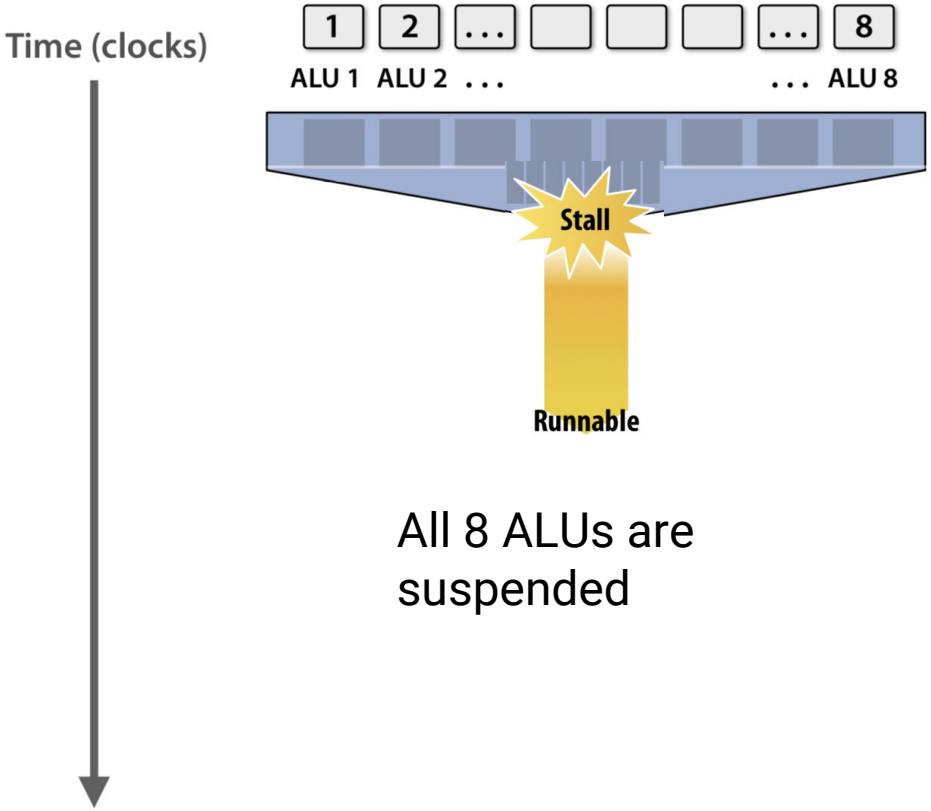
But we have a problem with conditions



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

Problem 2: What if there is a high latency stall



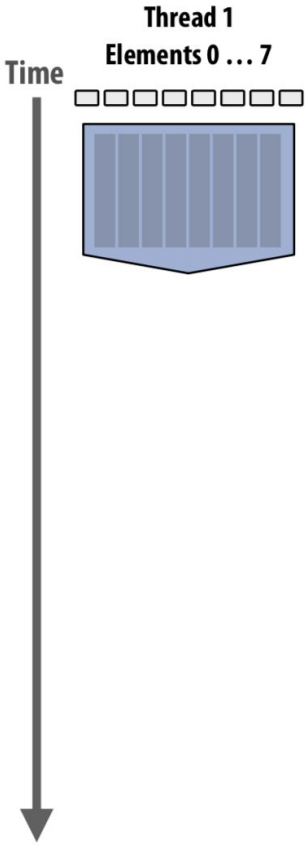
All 8 ALUs are suspended

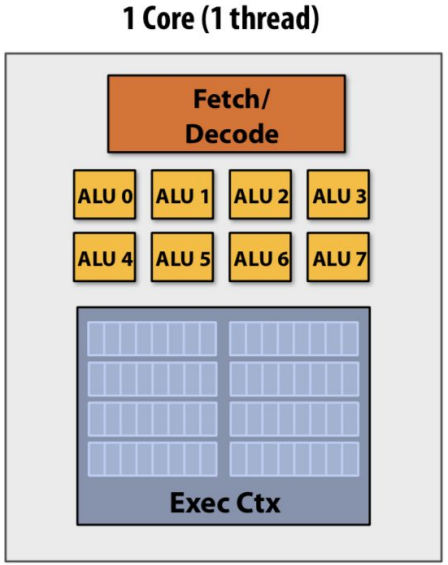
(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>
float x = A[i];
if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}
<resume unconditional code>

result[i] = x;
```

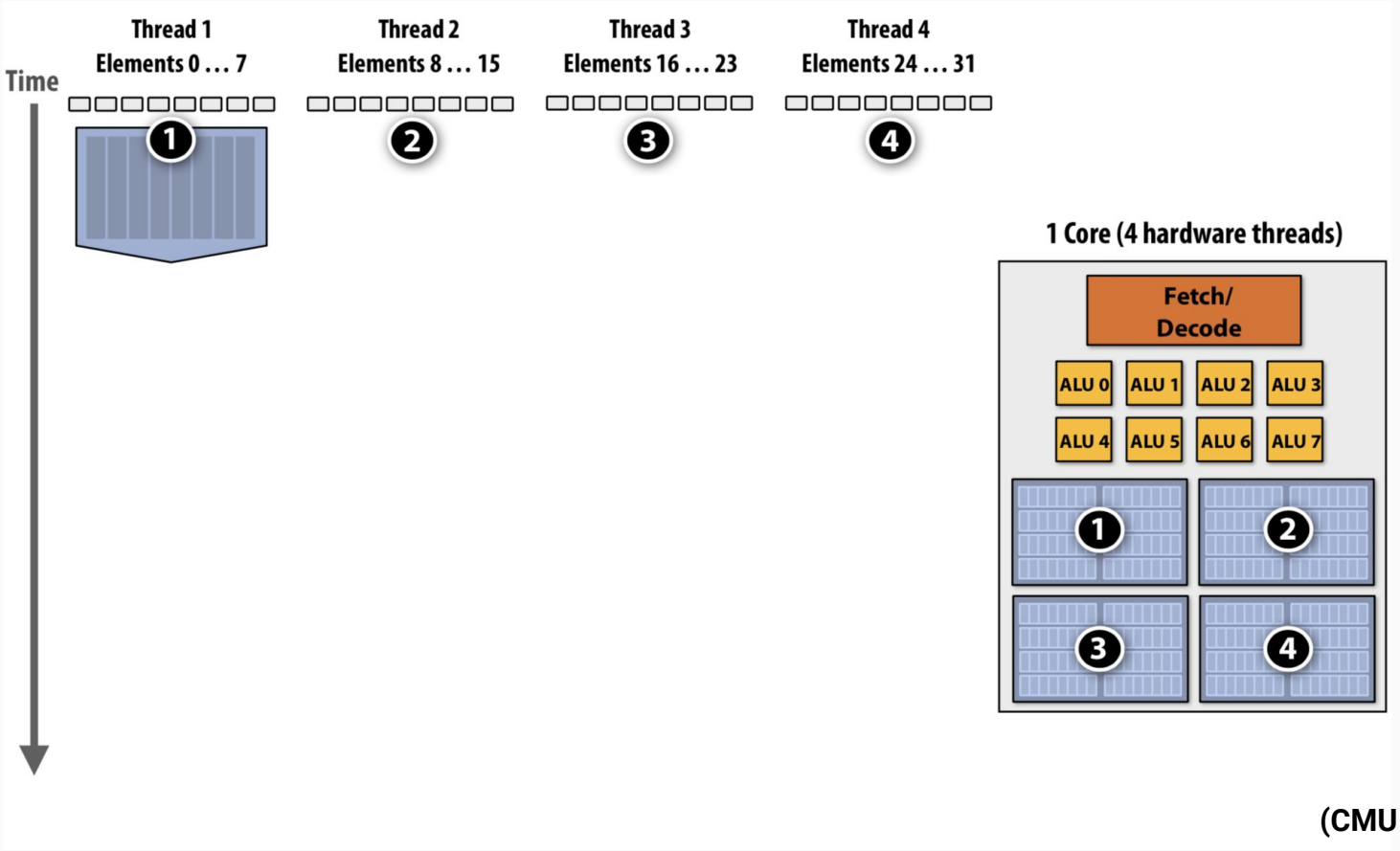
Assume a large latency in memory access

SIMD + Multithreading

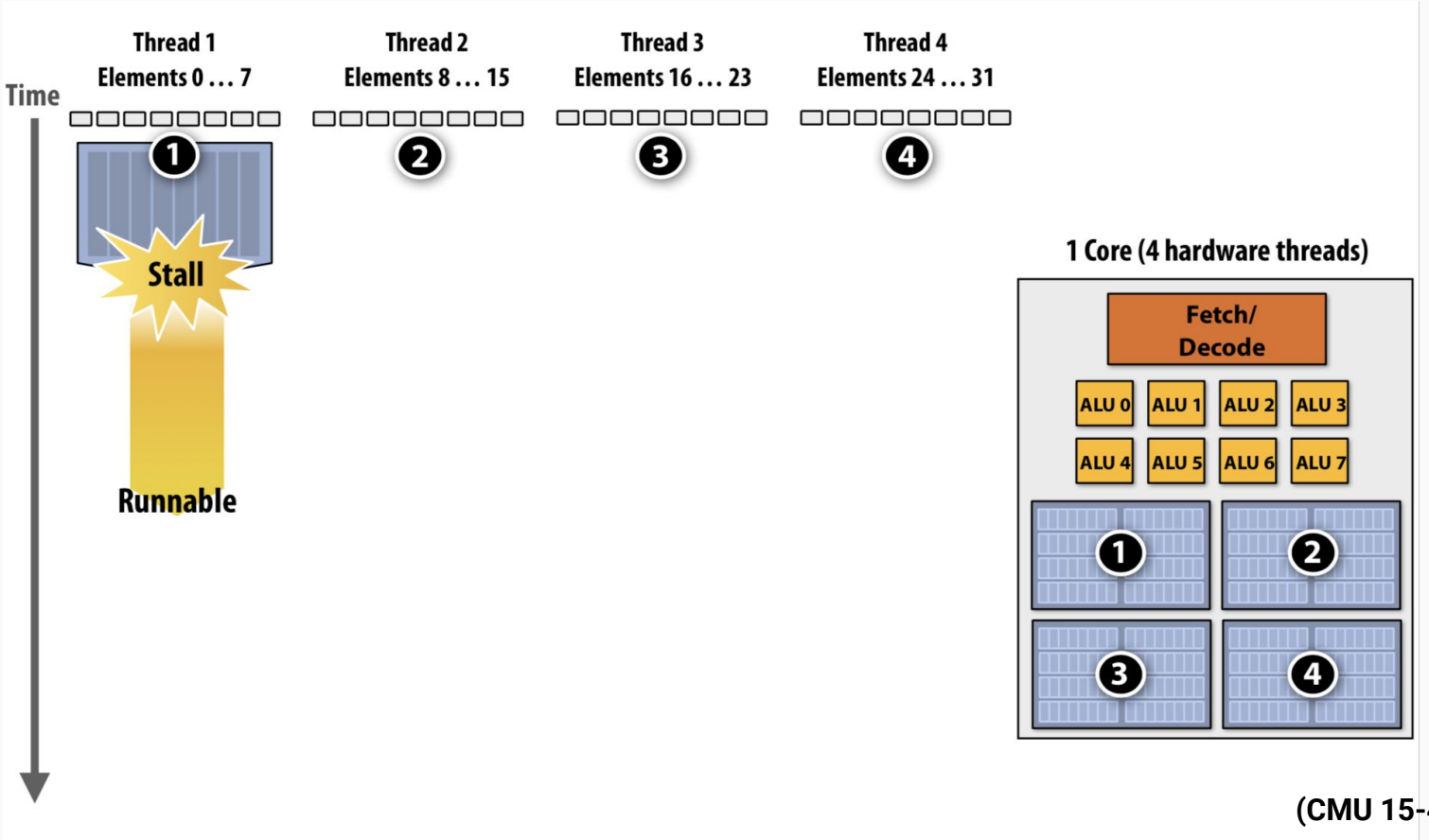




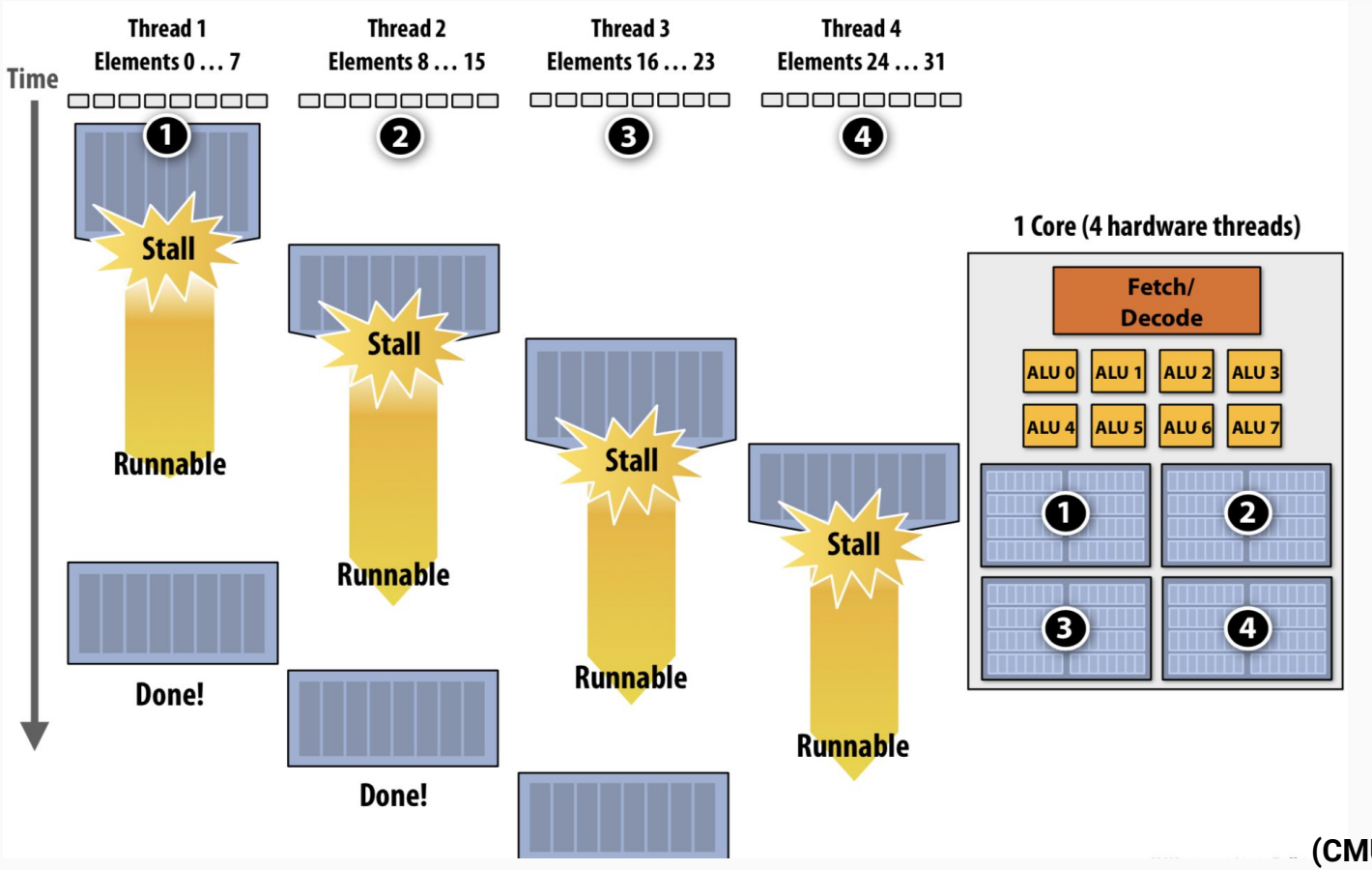
Partition context to support multiple threads



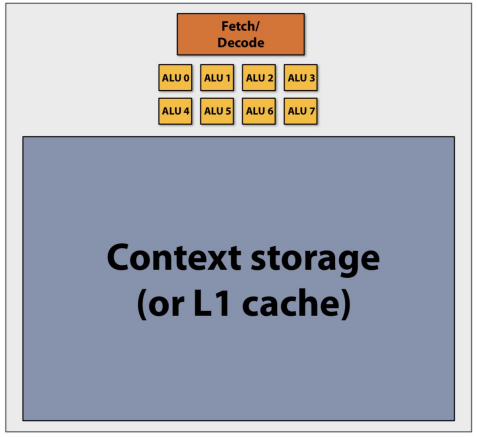
If one thread blocks, run the other



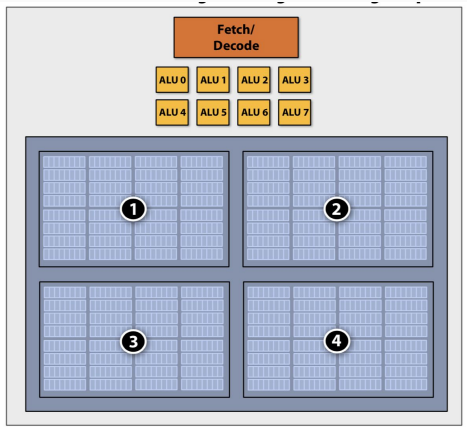
Have enough blocks to hide latency



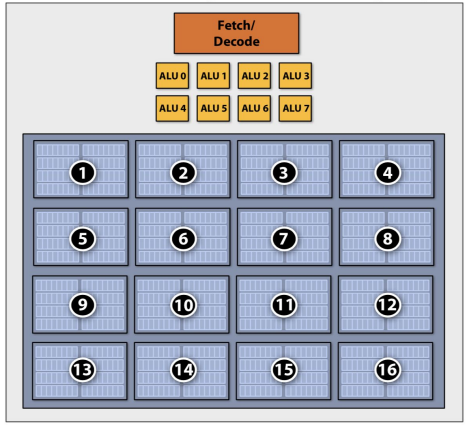
Trade off in context size



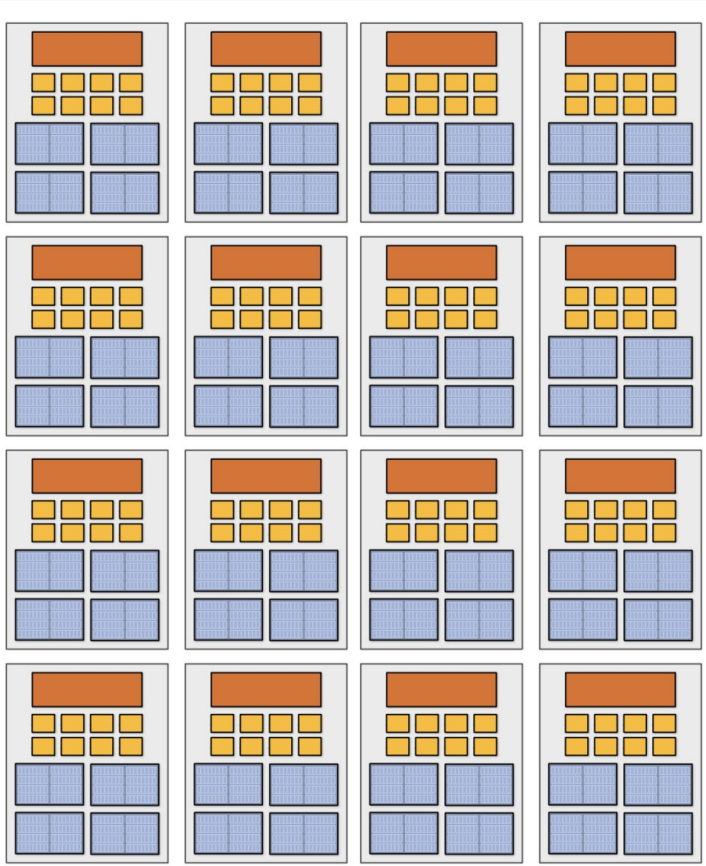
VS



VS

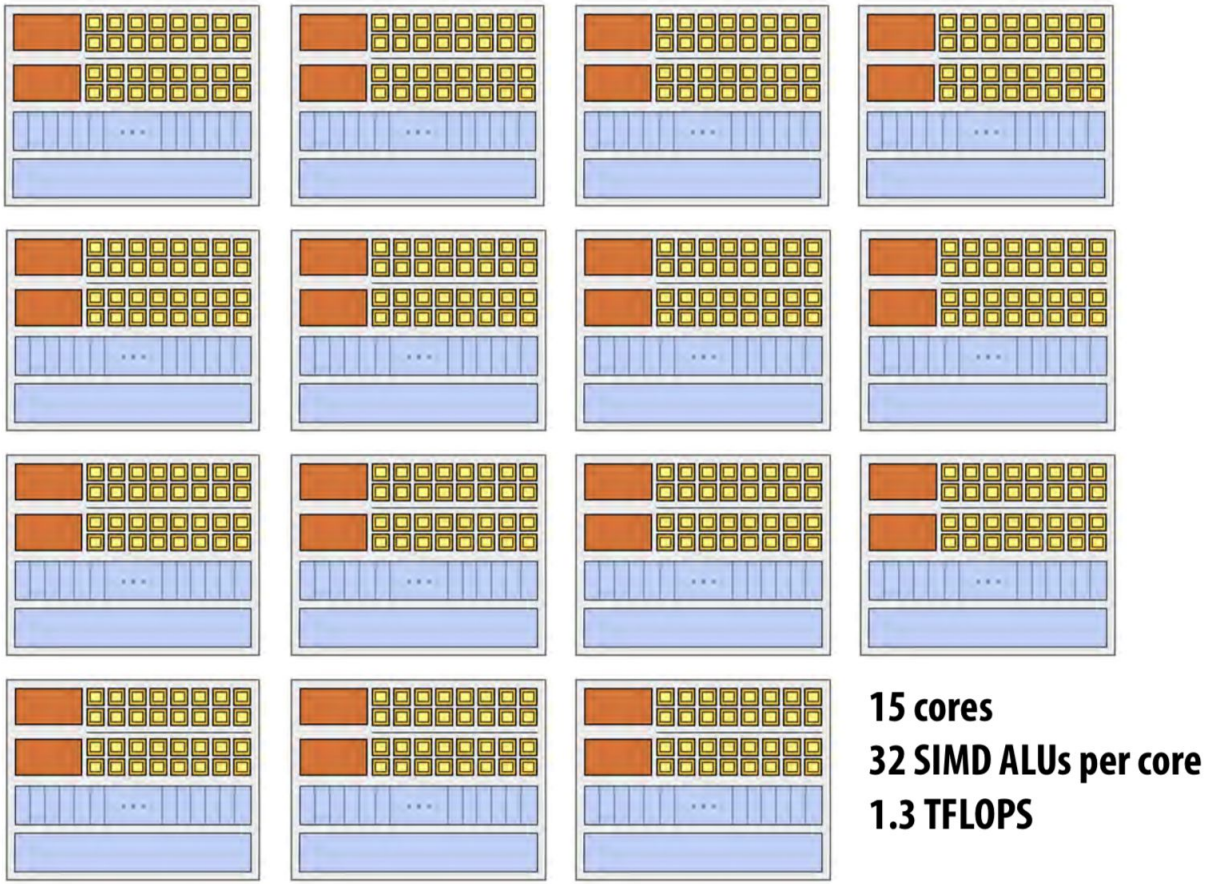


Fictitious chip



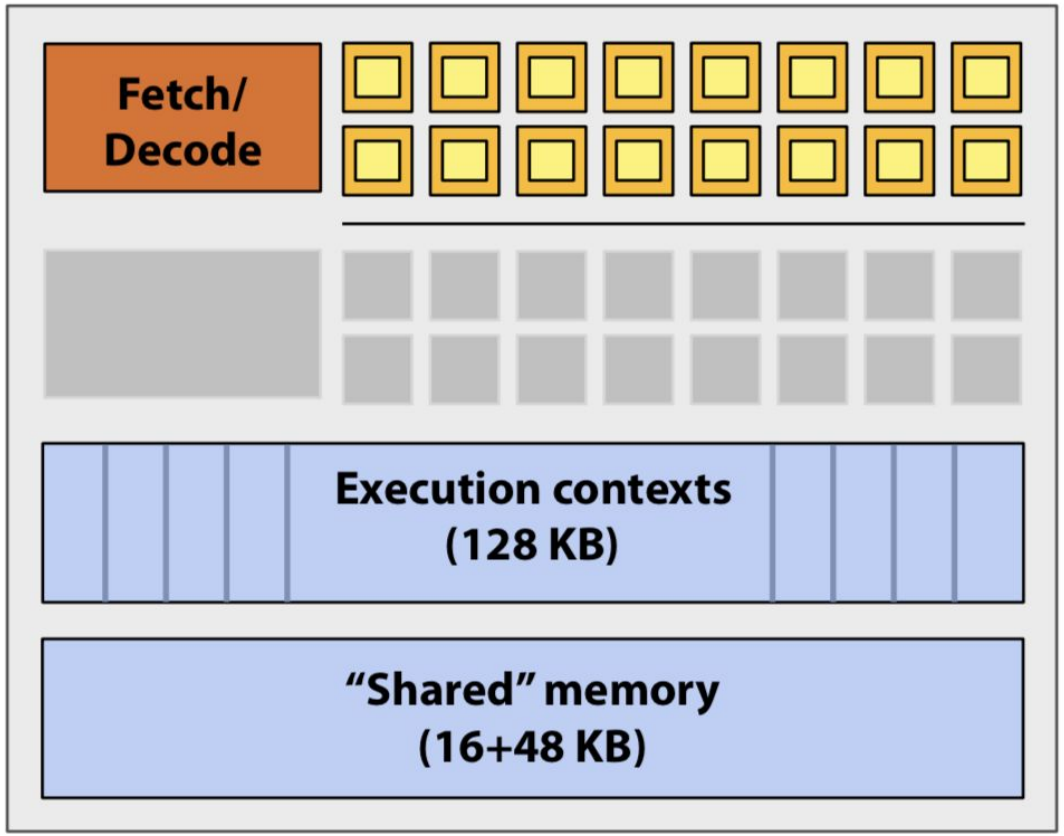
Number of cores	**16**
SIMD ALUs per core	**8**
Threads per core	**4**
Simultaneous instruction streams	**16**
Concurrent instruction streams	**64**
Total pieces of work to fully utilize processor	**512**

Nvidia GTX 480



15 cores
32 SIMD ALUs per core
1.3 TFLOPS

The GPU architecture

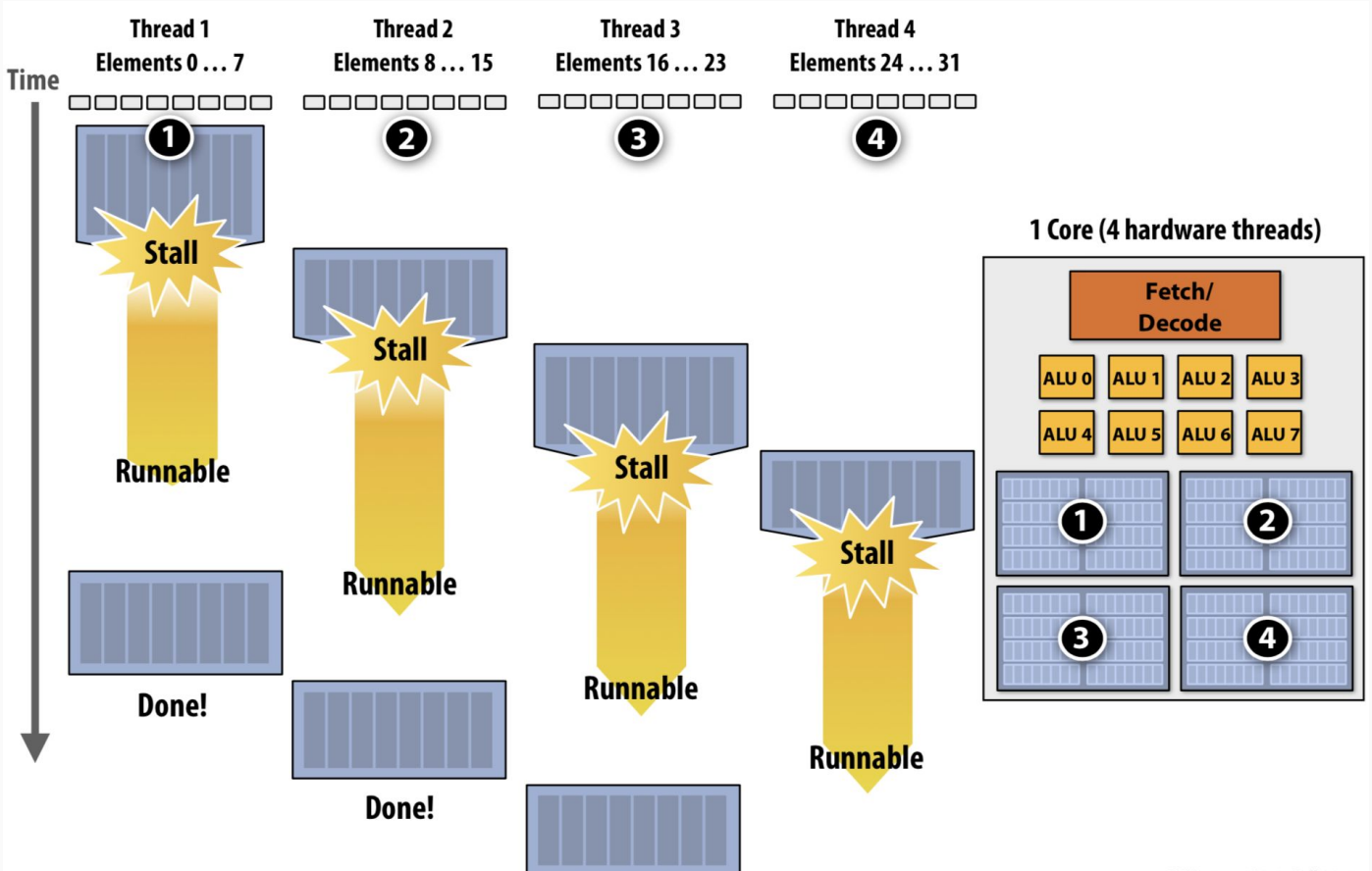


Each core has 16 ALUs running at twice the clock speed => SIMD instruction working on 32 data items

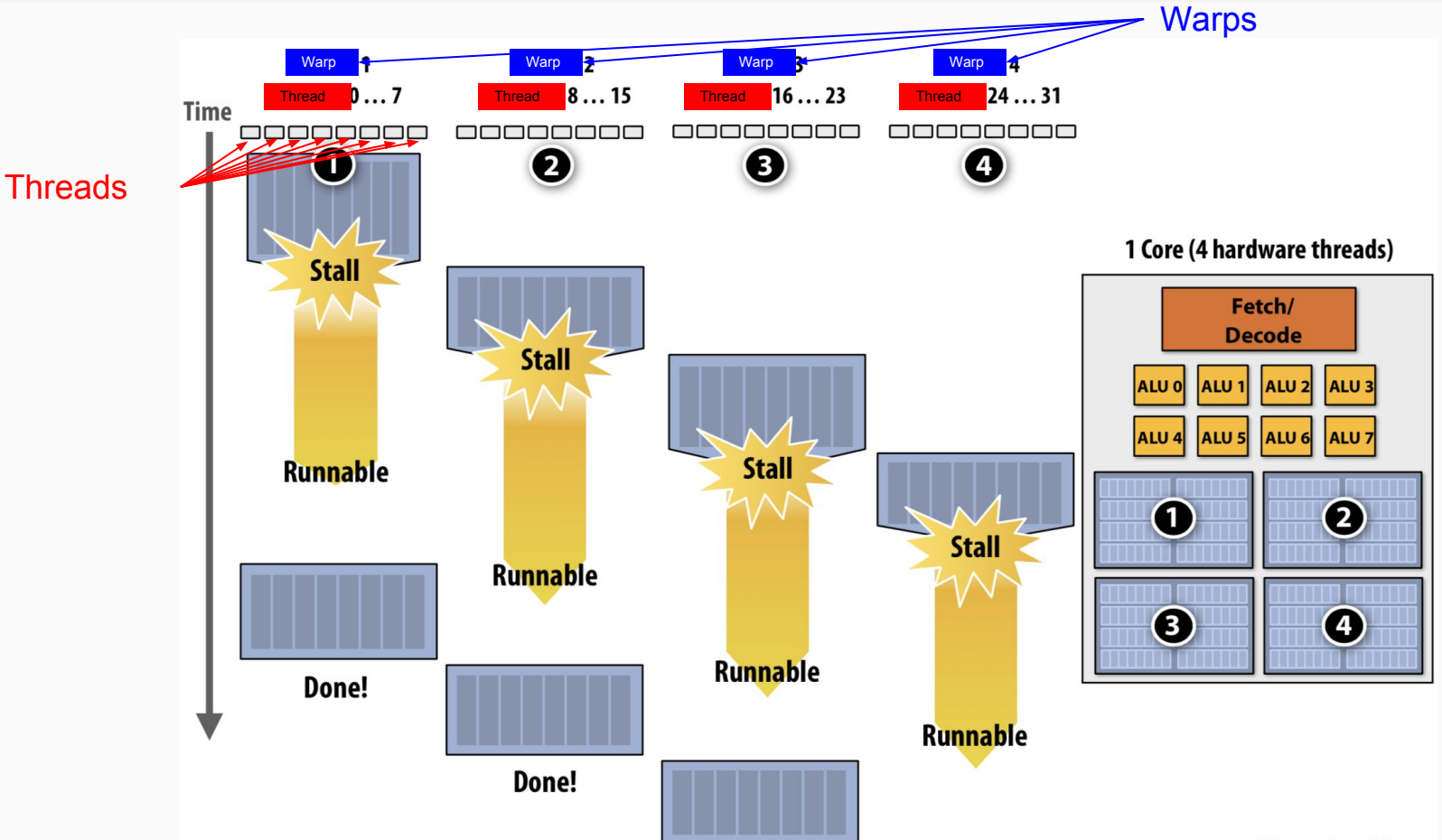
We will call each of these a **thread**
And a block of 32 threads a **warp**

Up to 48 warps are simultaneously scheduled => 1536 threads in flight

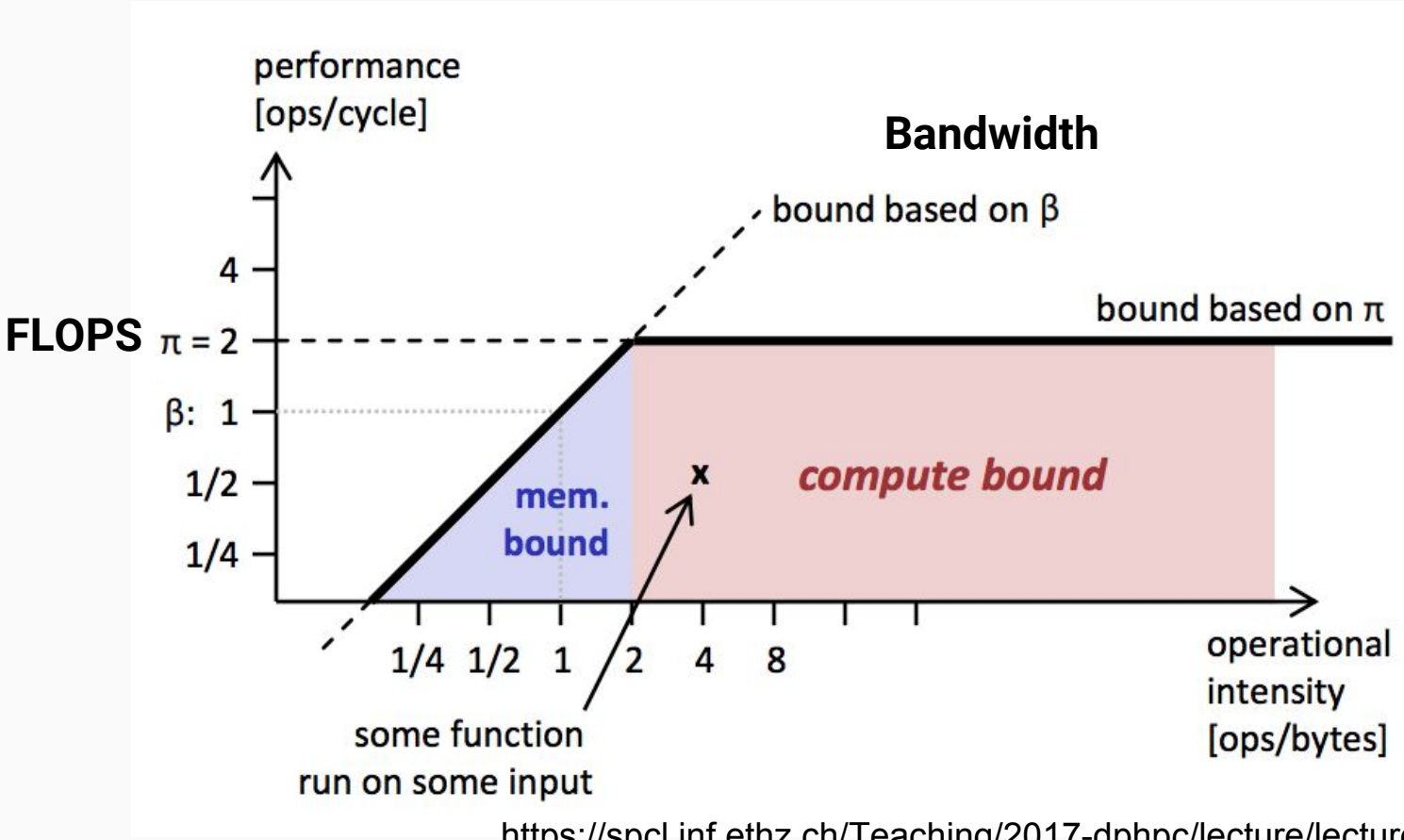
GPU terminology



GPU terminology



Roofline model



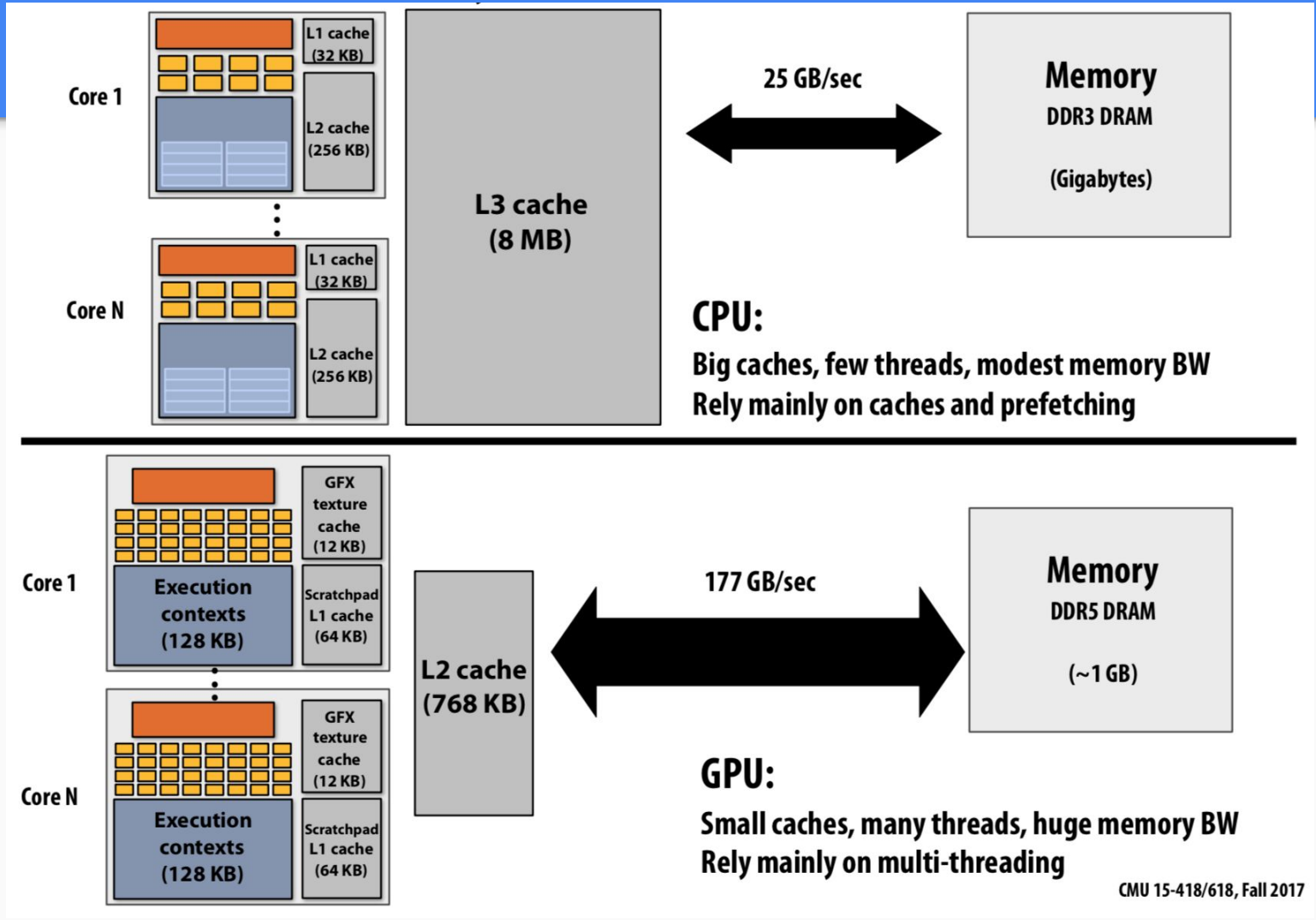
https://spcl.inf.ethz.ch/Teaching/2017-dphpc/lecture/lecture6-roofline.pdf

Exercise

Given a 1 TFLOPS machine, how much memory bandwidth is required to avoid being memory bound for

- vector point-wise multiplication (assume 4 bytes per word)
- [homework] FFT

Memory


Core 1
L1 cache (32 KB)
L2 cache (256 KB)
Core N
L1 cache (32 KB)
L2 cache (256 KB)
L3 cache (8 MB)
25 GB/sec
Memory
DDR3 DRAM
(Gigabytes)
CPU:
Big caches, few threads, modest memory BW
Rely mainly on caches and prefetching
Core 1
GFX texture cache (12 KB)
Execution contexts (128 KB)
Scratchpad L1 cache (64 KB)
Core N
GFX texture cache (12 KB)
Execution contexts (128 KB)
Scratchpad L1 cache (64 KB)
L2 cache (768 KB)
177 GB/sec
Memory
DDR5 DRAM
(~1 GB)
GPU:
Small caches, many threads, huge memory BW
Rely mainly on multi-threading
CMU 15-418/618, Fall 2017

Next time

Start with CUDA programming!