CS 6023 - GPU Programming

# Parallel Computing Architecture

03/08/2018

## Agenda

How do we contrast CPUs from GPUs?

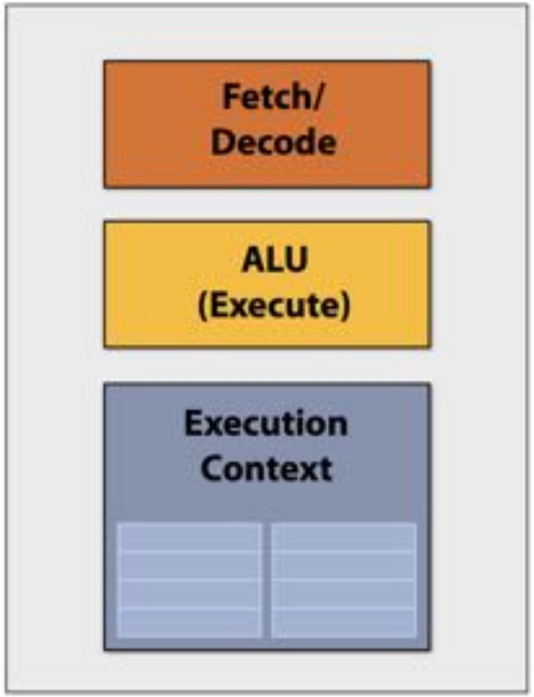
Part 1 of 2:

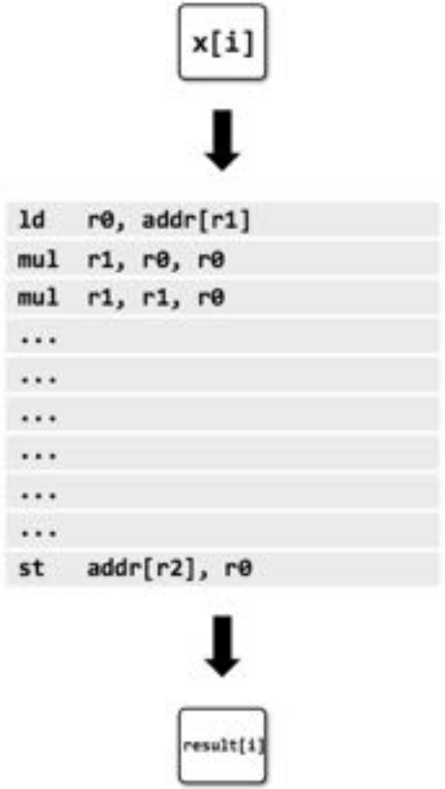
**Last time**: CPU architecture background

Part 2 of 2:

**Today:** Parallel computer architecture background

## CPU - single core cartoon

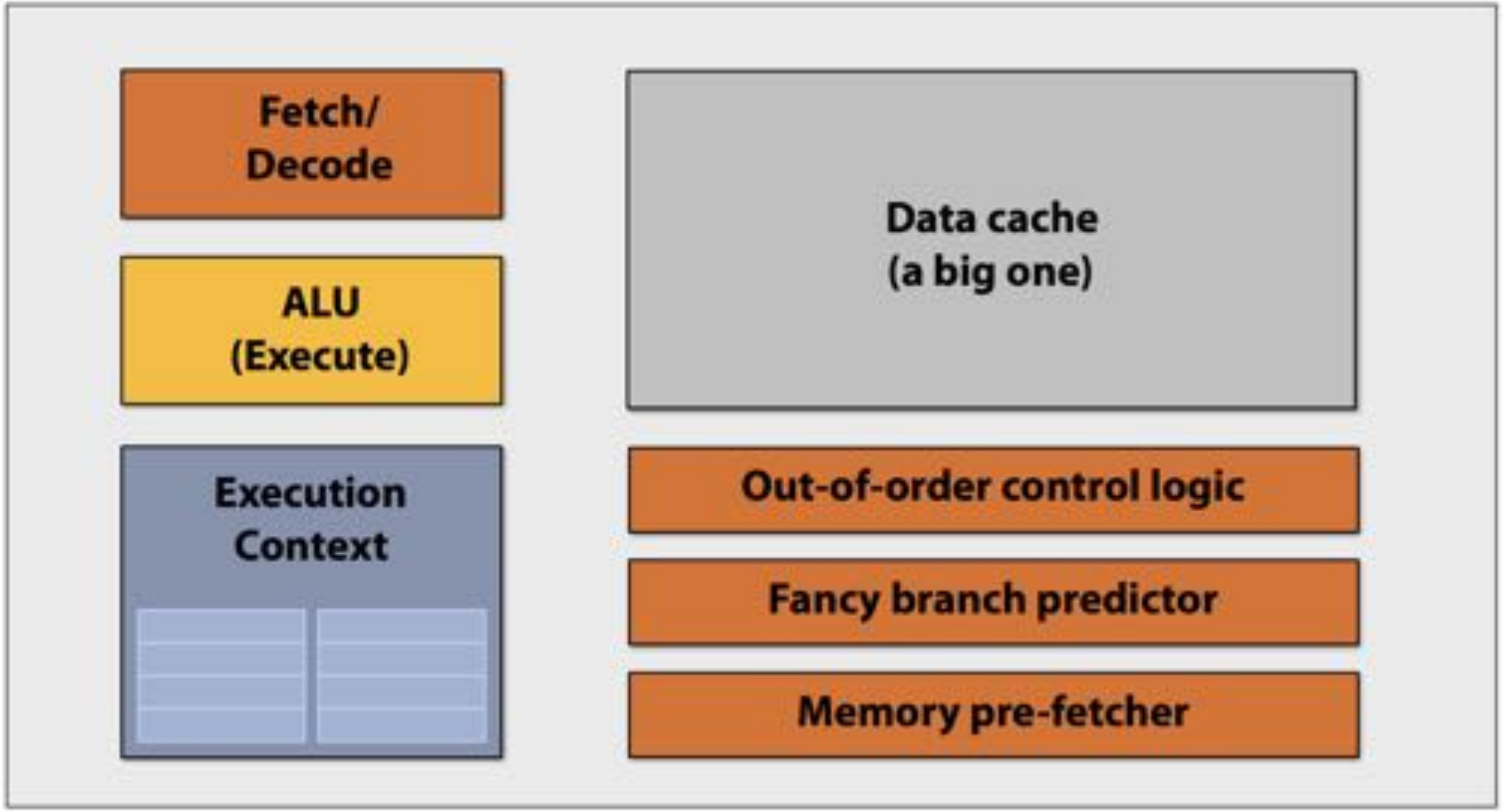




```
ld    r0, addr[r1]
mul   r1, r0, r0
mul   r1, r1, r0
...
...
...
...
...
...
st    addr[r2], r0
```

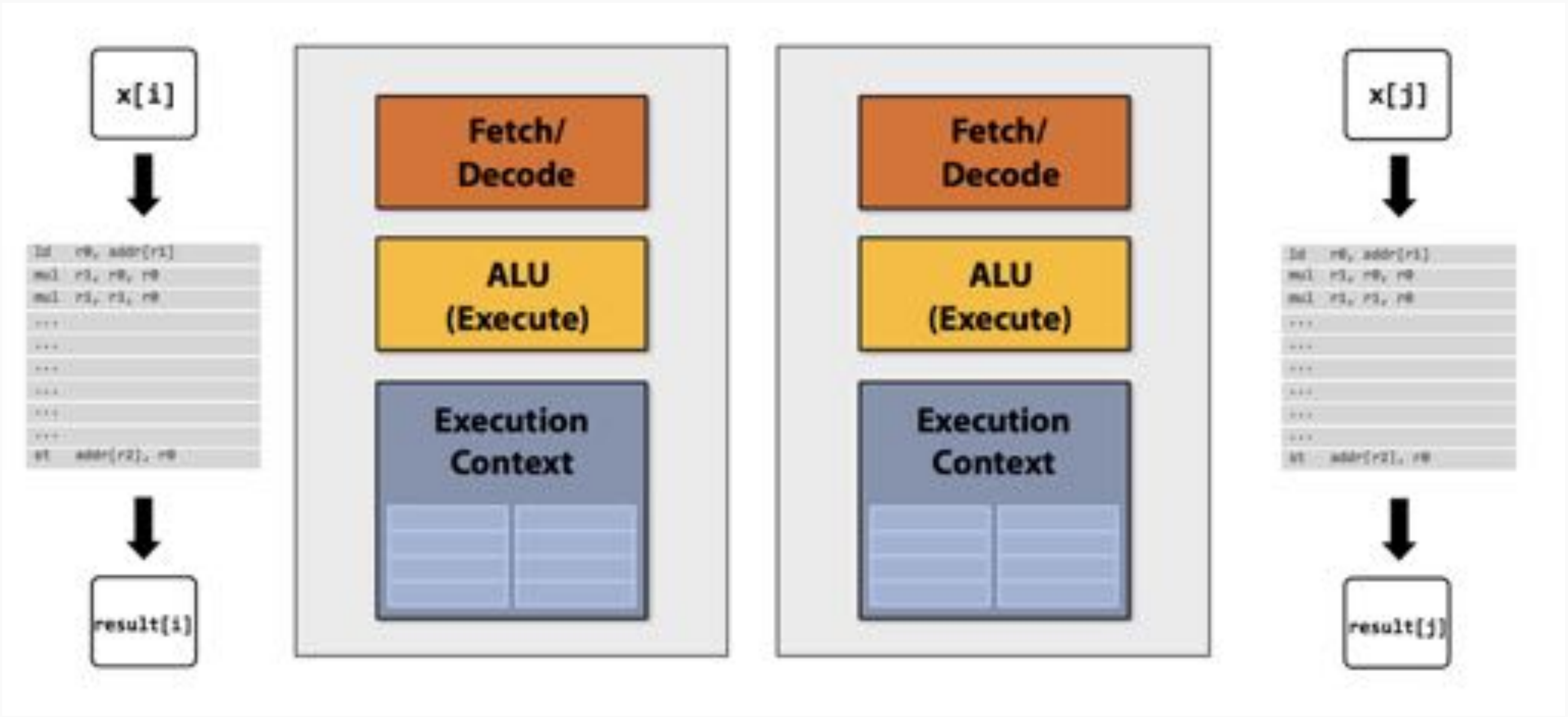
(CMU 15-418/618)

## CPU cartoon - single core era



(CMU 15-418/618)

# CPU cartoon - multi-core

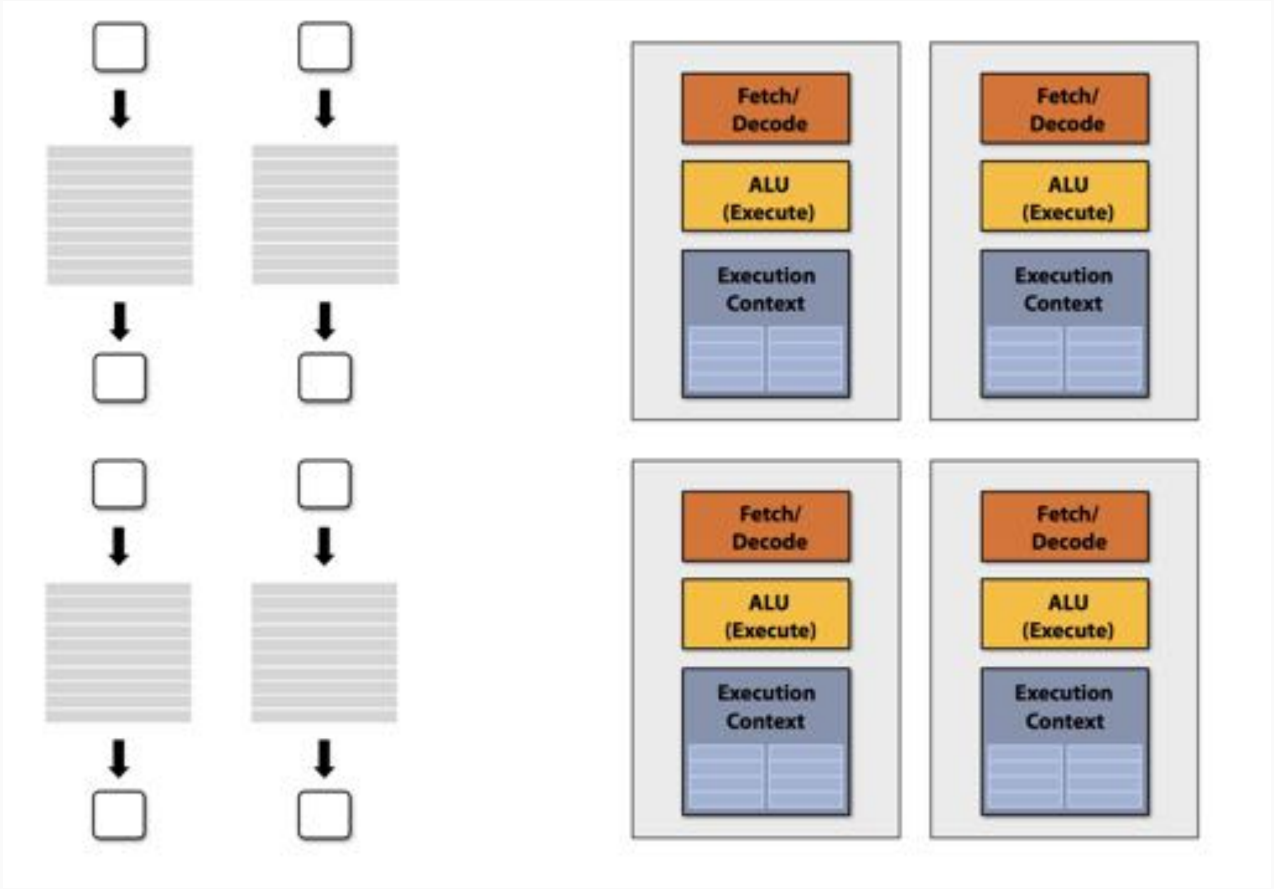


```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

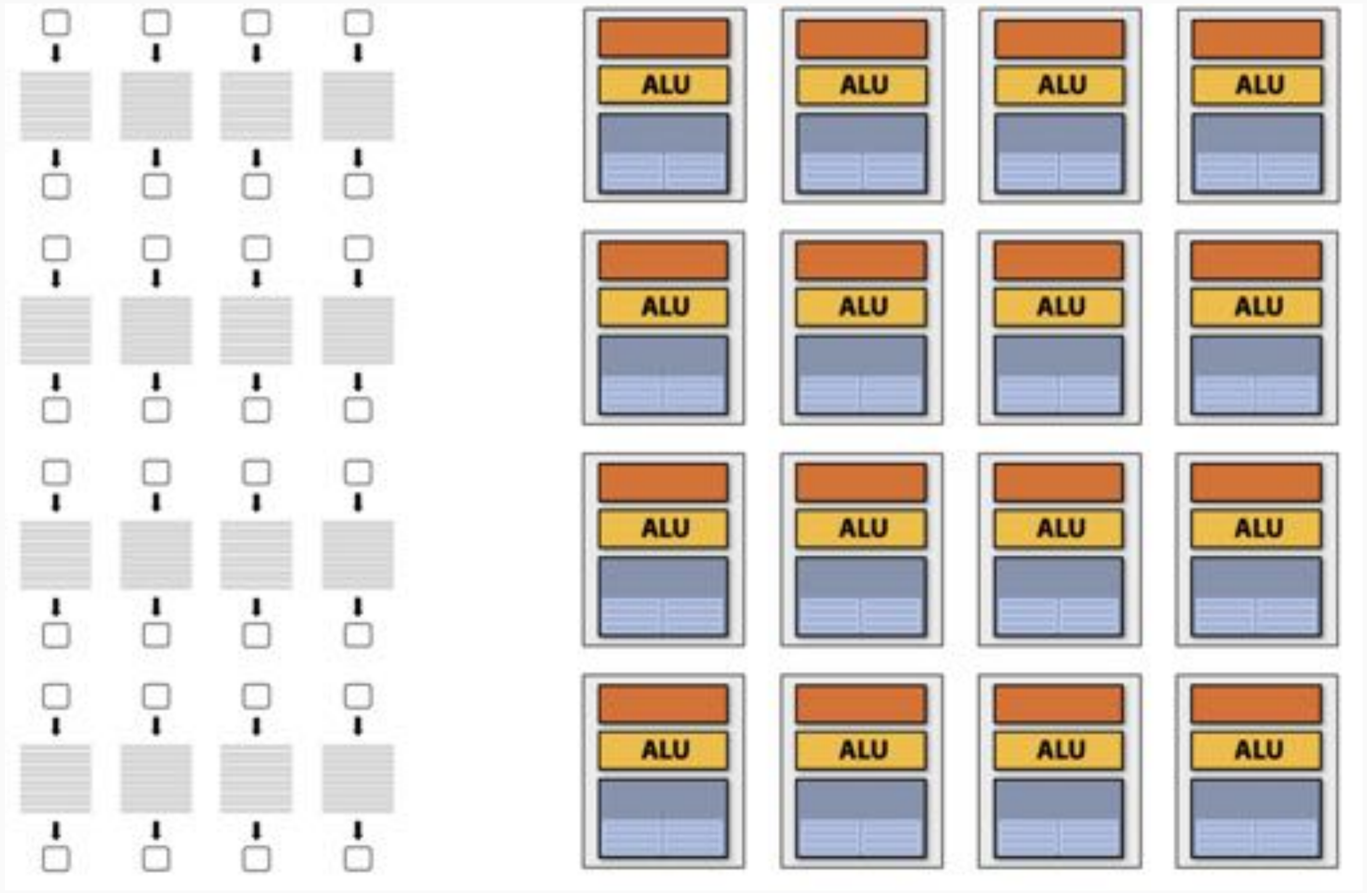
Make cores simpler, have multiple of them

## CPU cartoon - many core



(CMU 15-418/618)

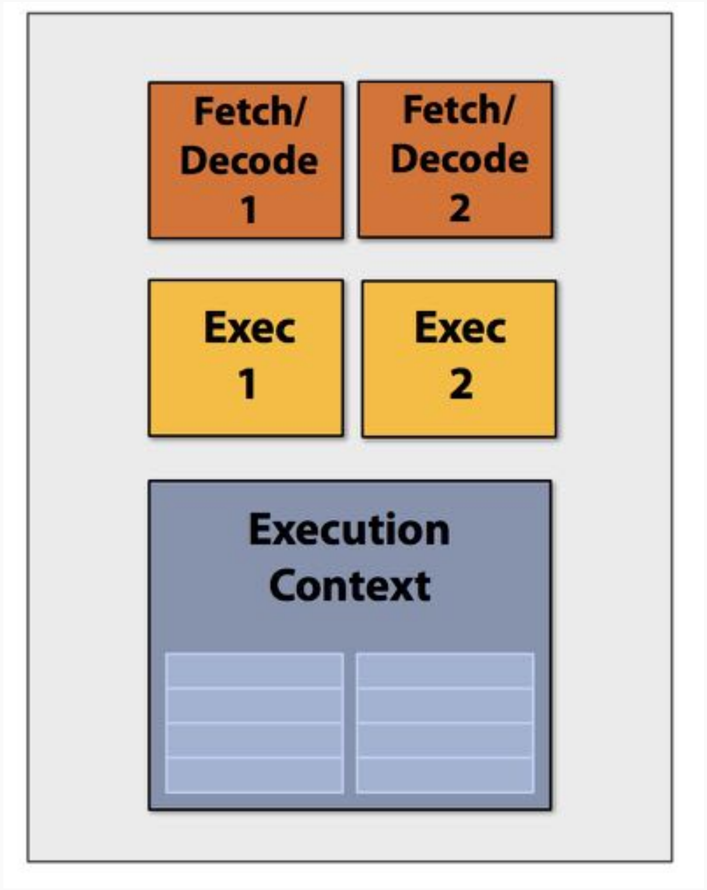
## CPU cartoon - many core



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

(CMU 15-418/618)

## What is this?

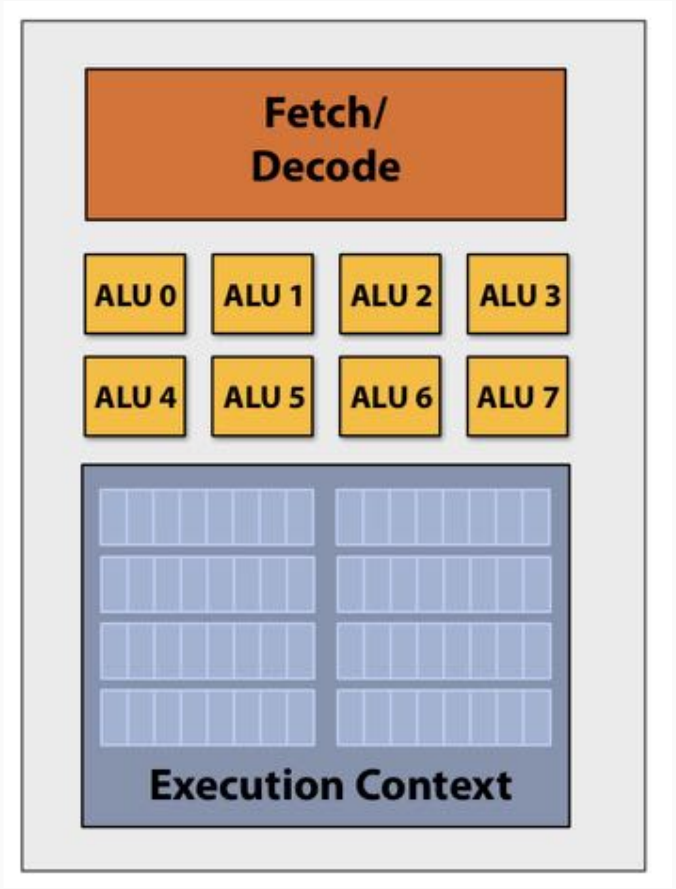


Superscalar

When is this useful?

- For exploiting ILP

## What is this?



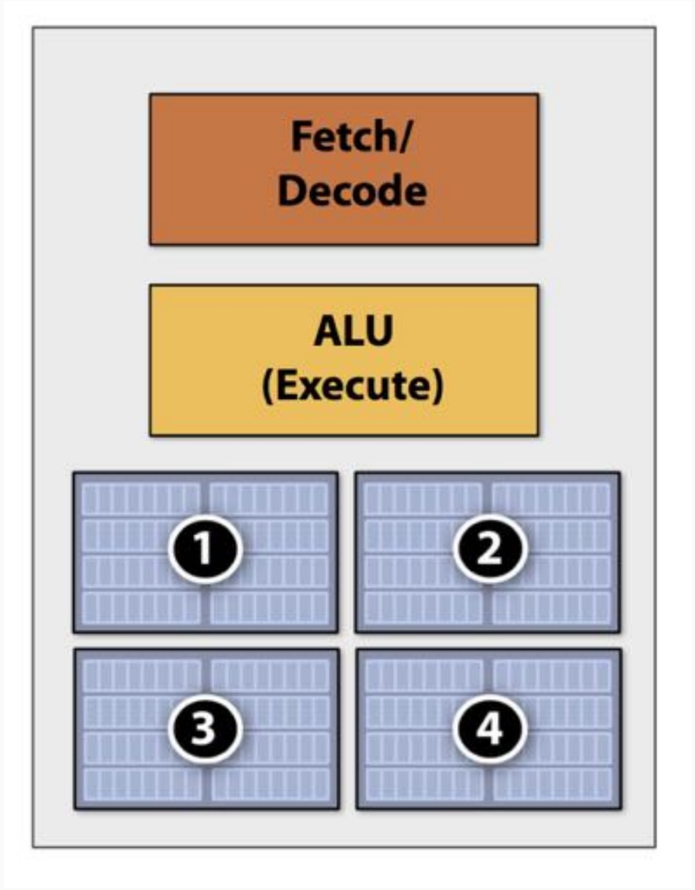
SIMD processing

When is this useful?

- Explicit SIMD
- Vector processing

(CMU 15-418/618)

## What is this?

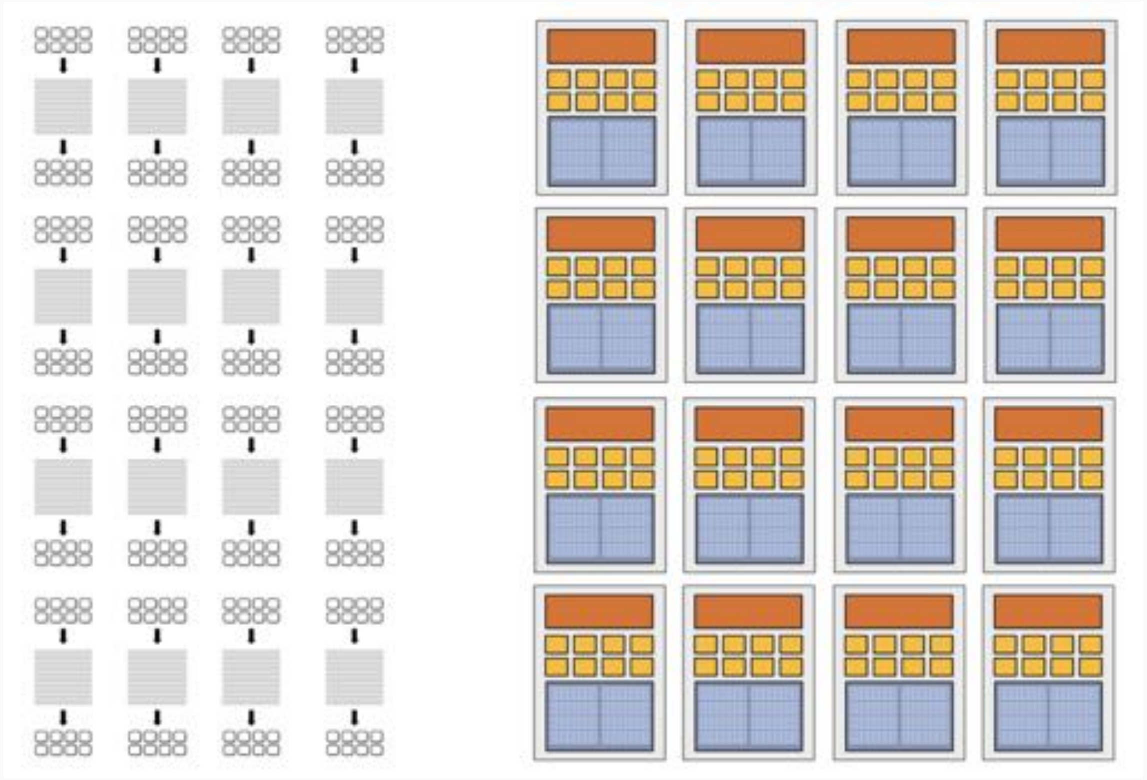


Multithreading

When is this useful

- For hiding latency during stalls

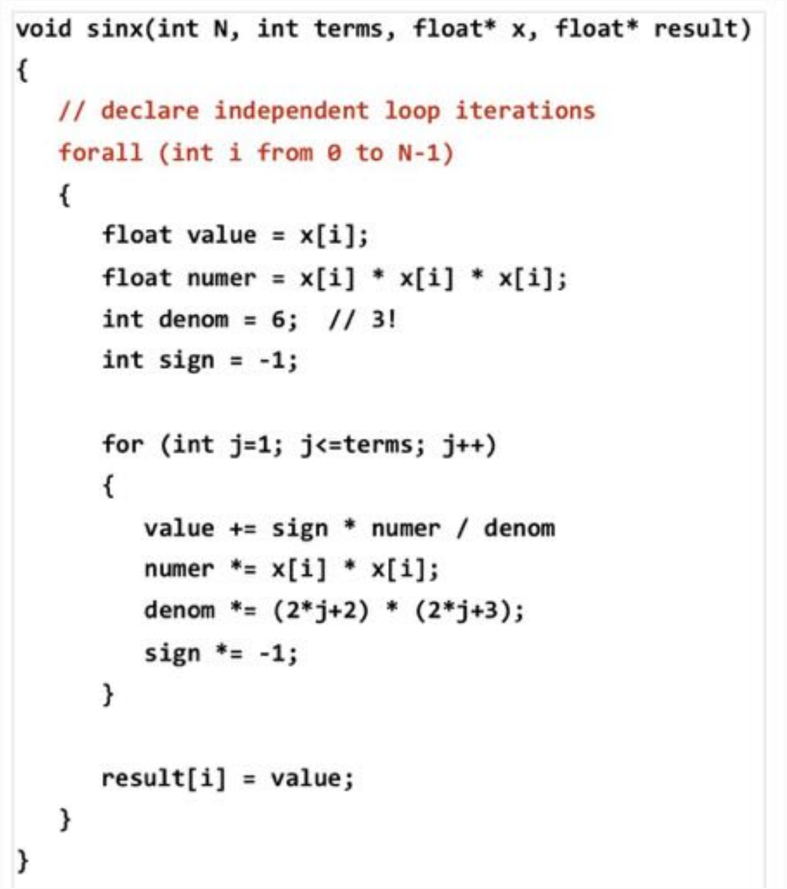
## Combining multicore and SIMD



How many separate instruction streams does this process

16 cores, 16 simultaneous instruction streams

## SIMD processing

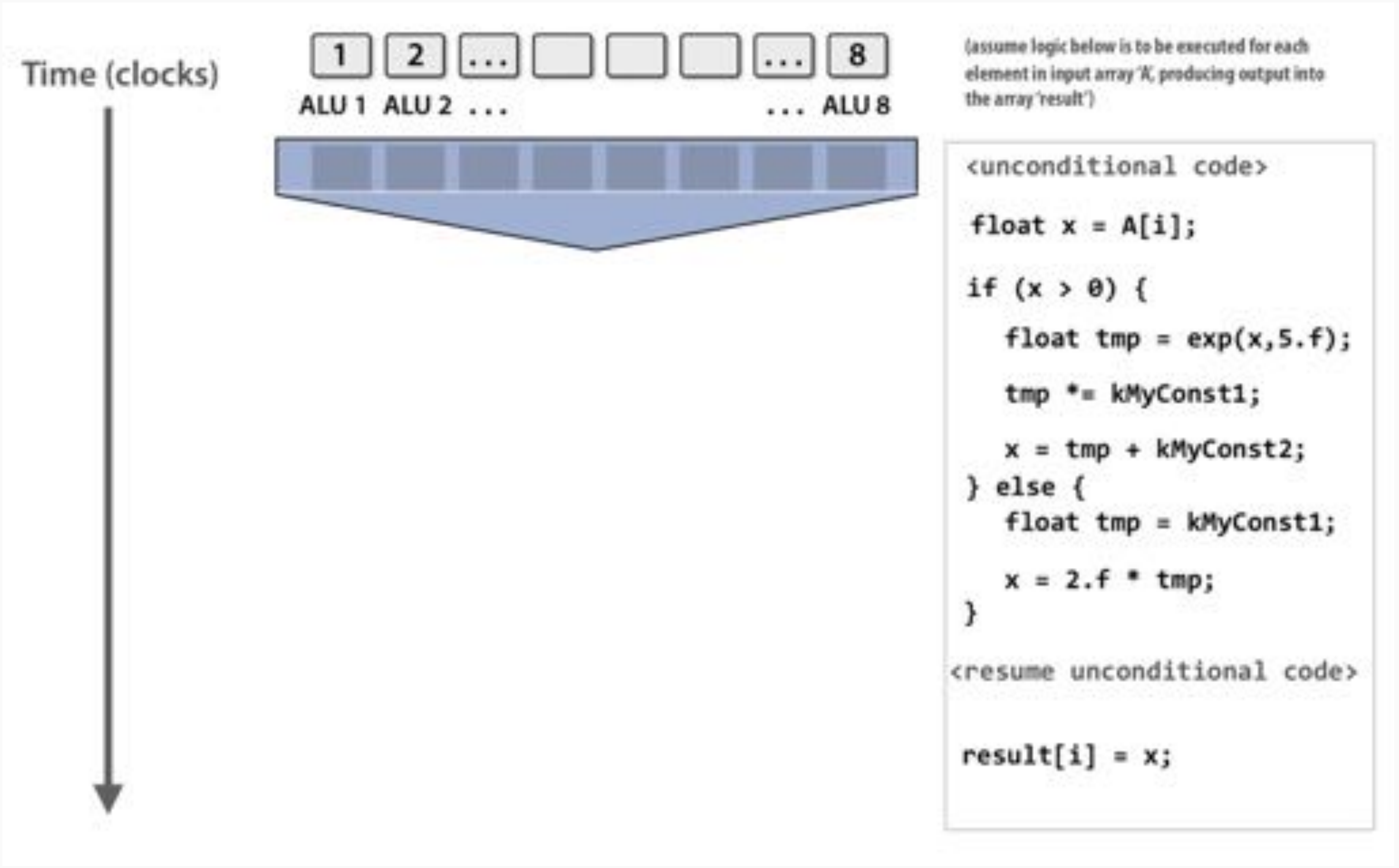


```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      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;
   }
}
```

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

=> SIMD processing

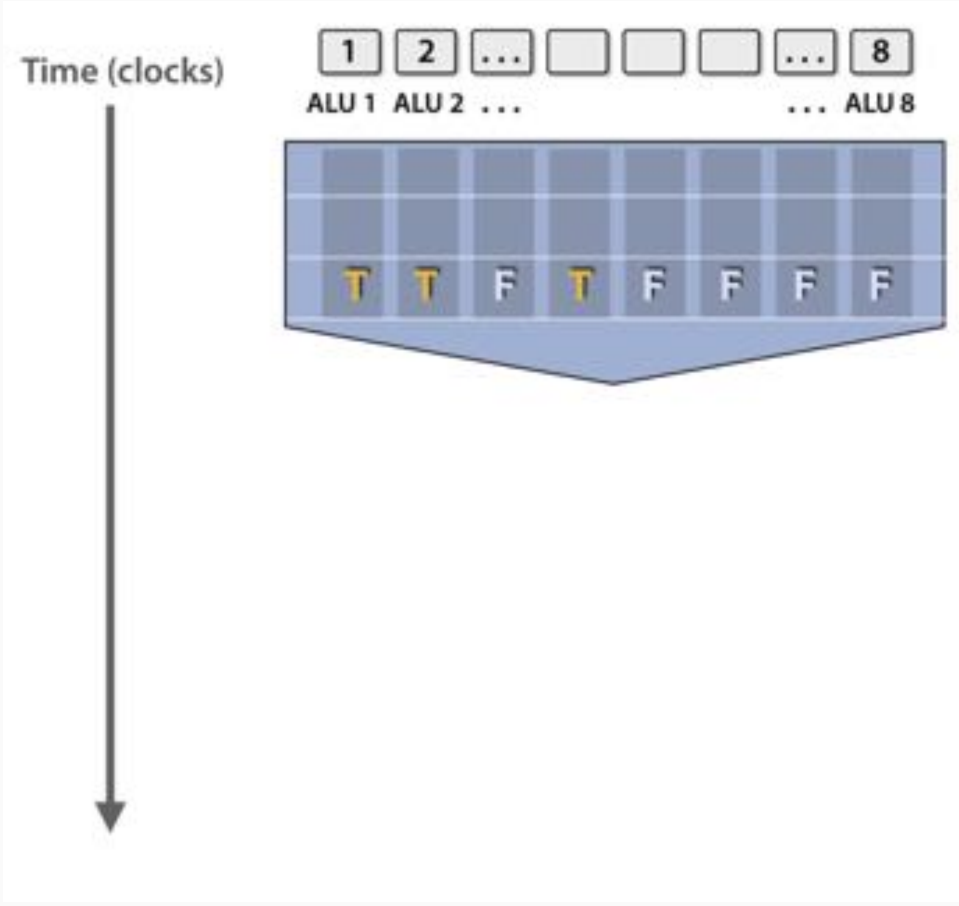
## But we have a problem with conditions



(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;
```

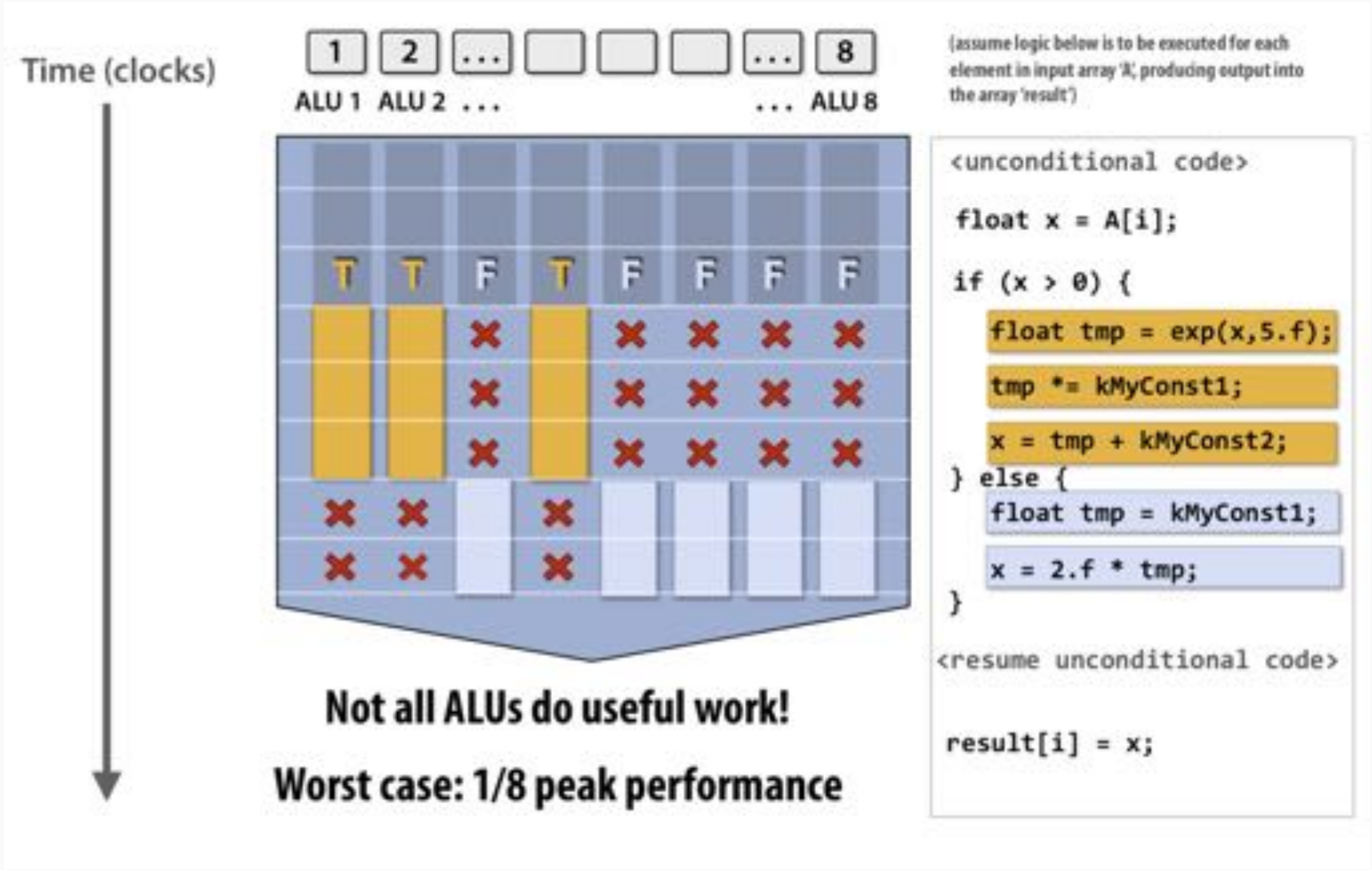
## But we have a problem with conditions



(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')

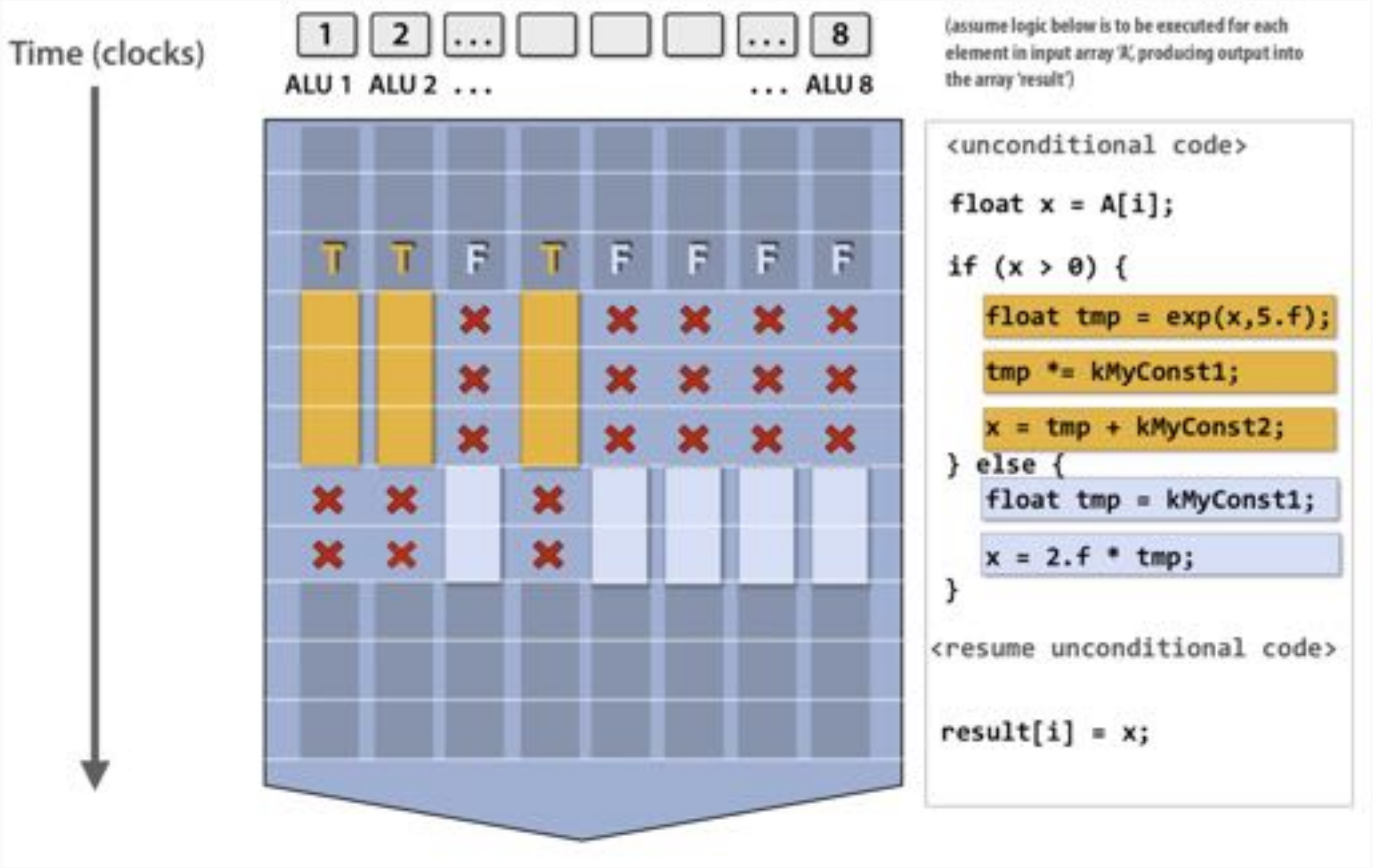
```
<unconditional code>

 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;
```

**Not all ALUs do useful work!**

**Worst case: 1/8 peak performance**

**(CMU 15-418/618)**

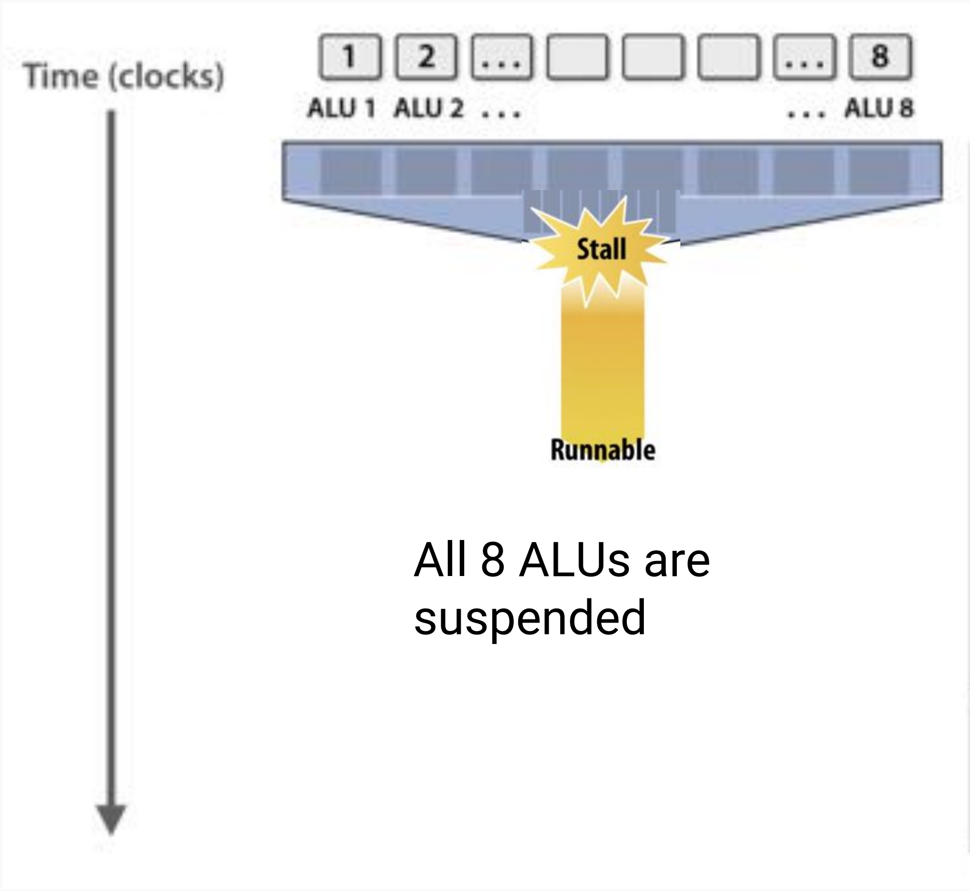
## But we have a problem with conditions



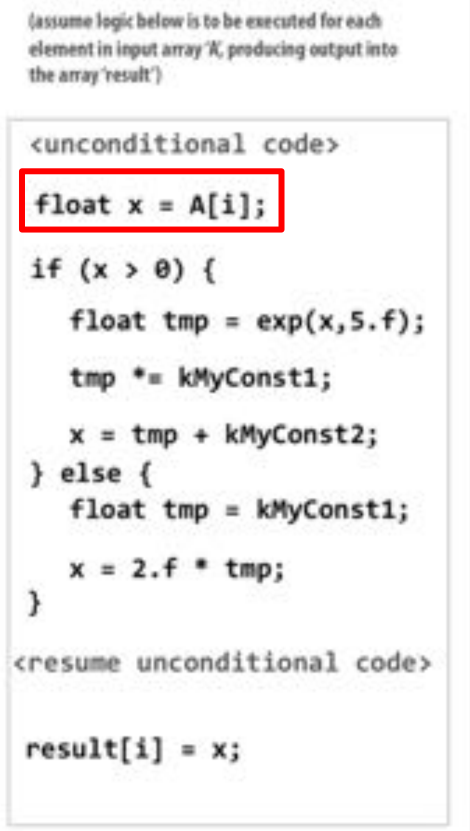
(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<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

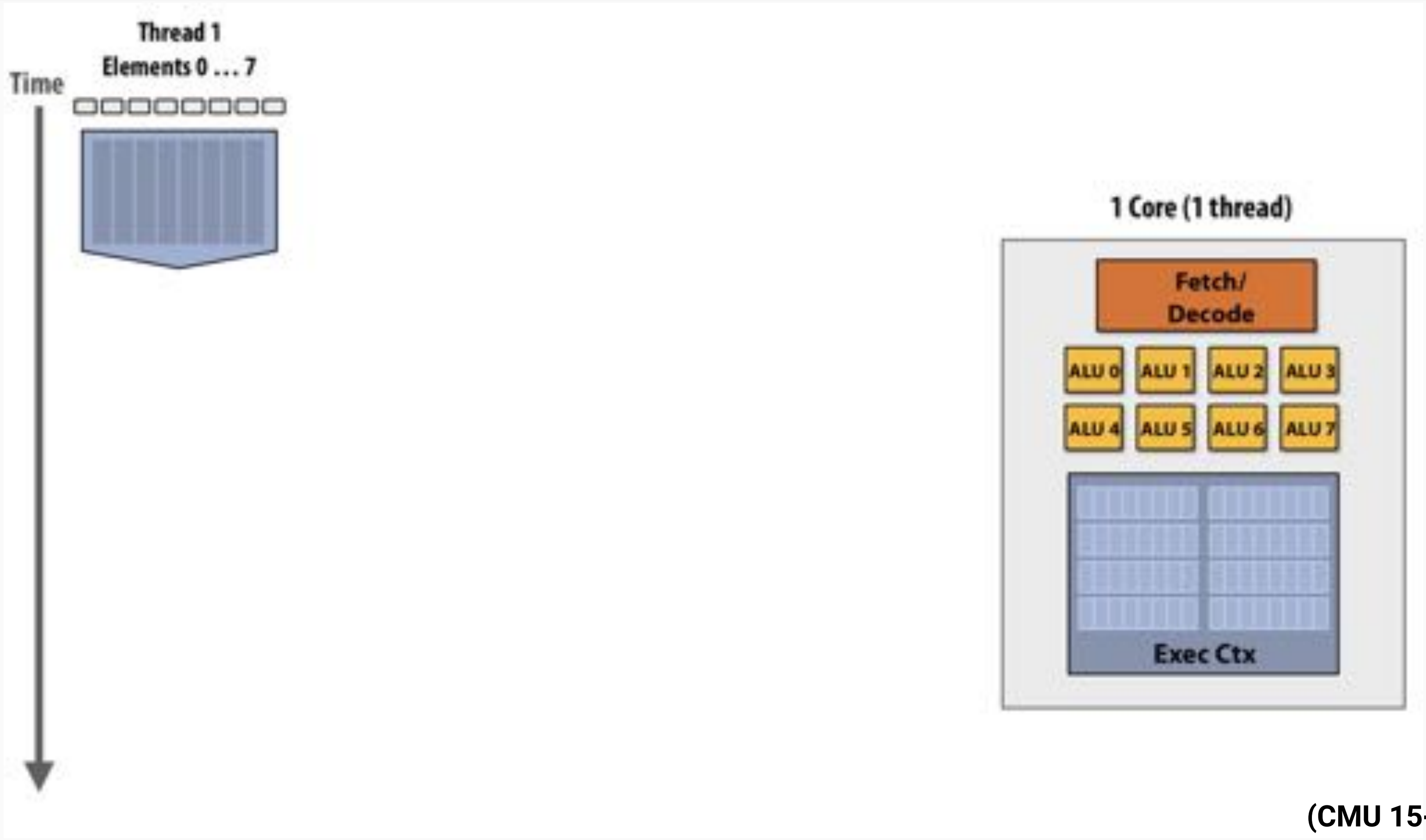

(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<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

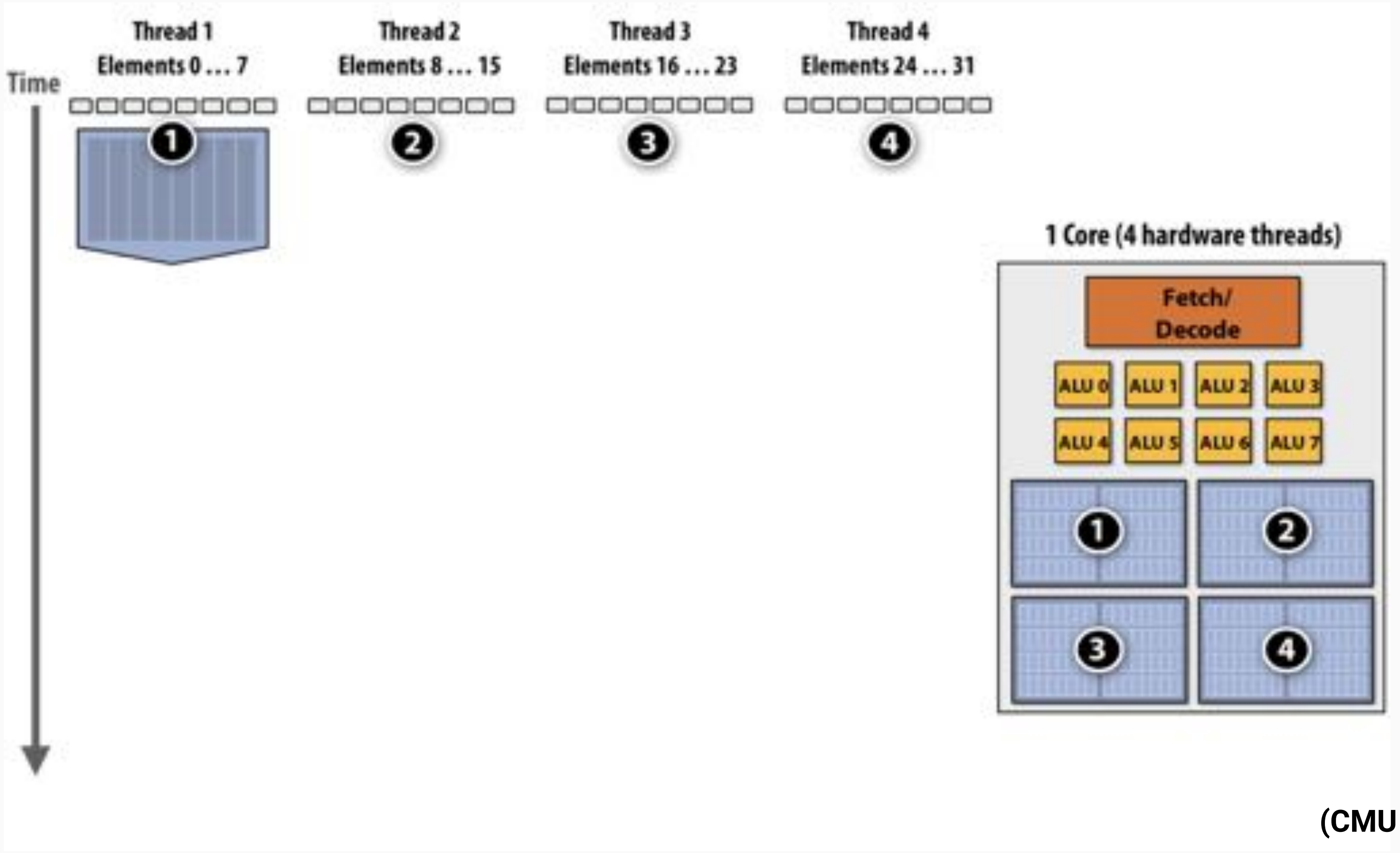
**(CMU 15-418/618)**

## SIMD + Multithreading



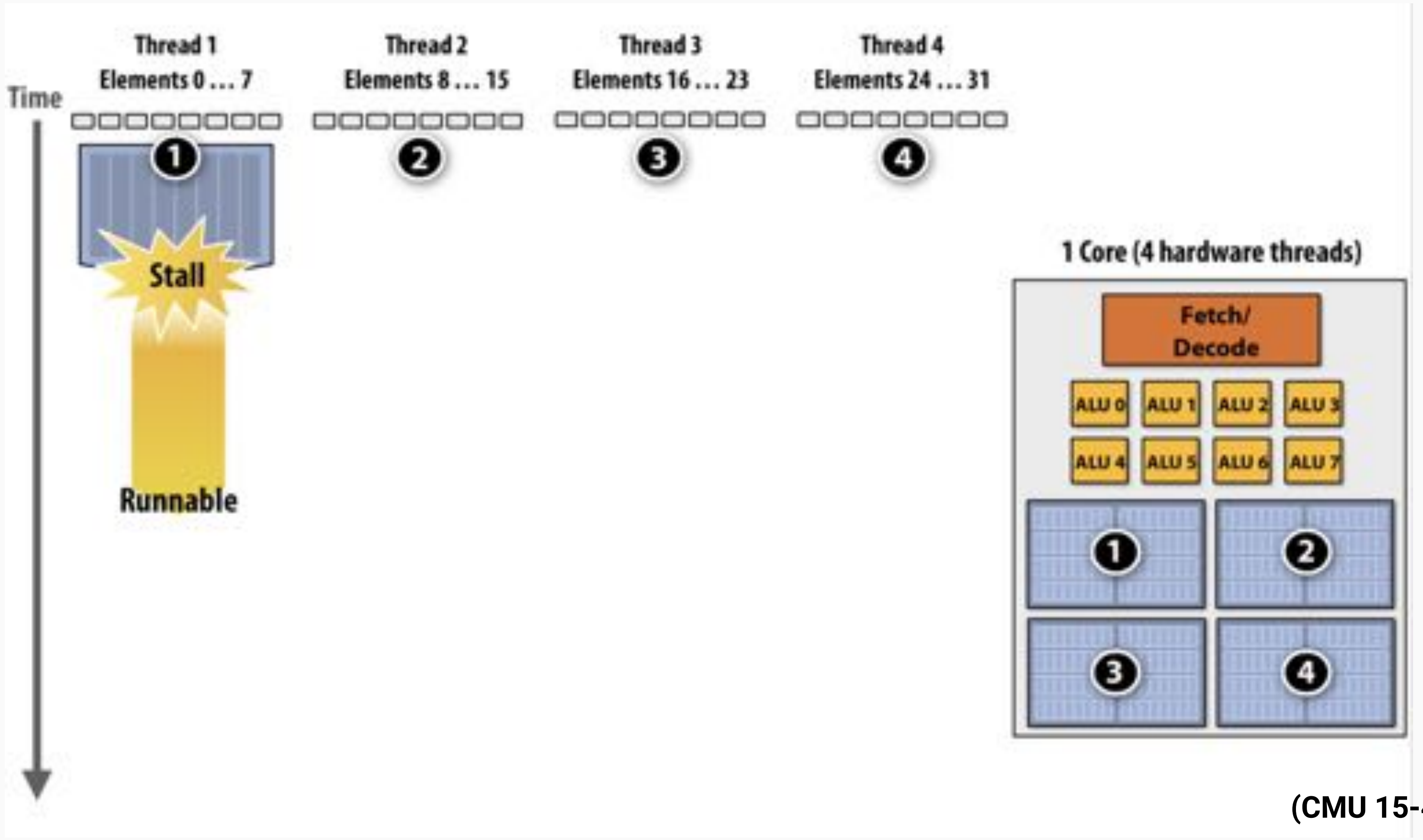
(CMU 15-418/618)

## Partition context to support multiple threads



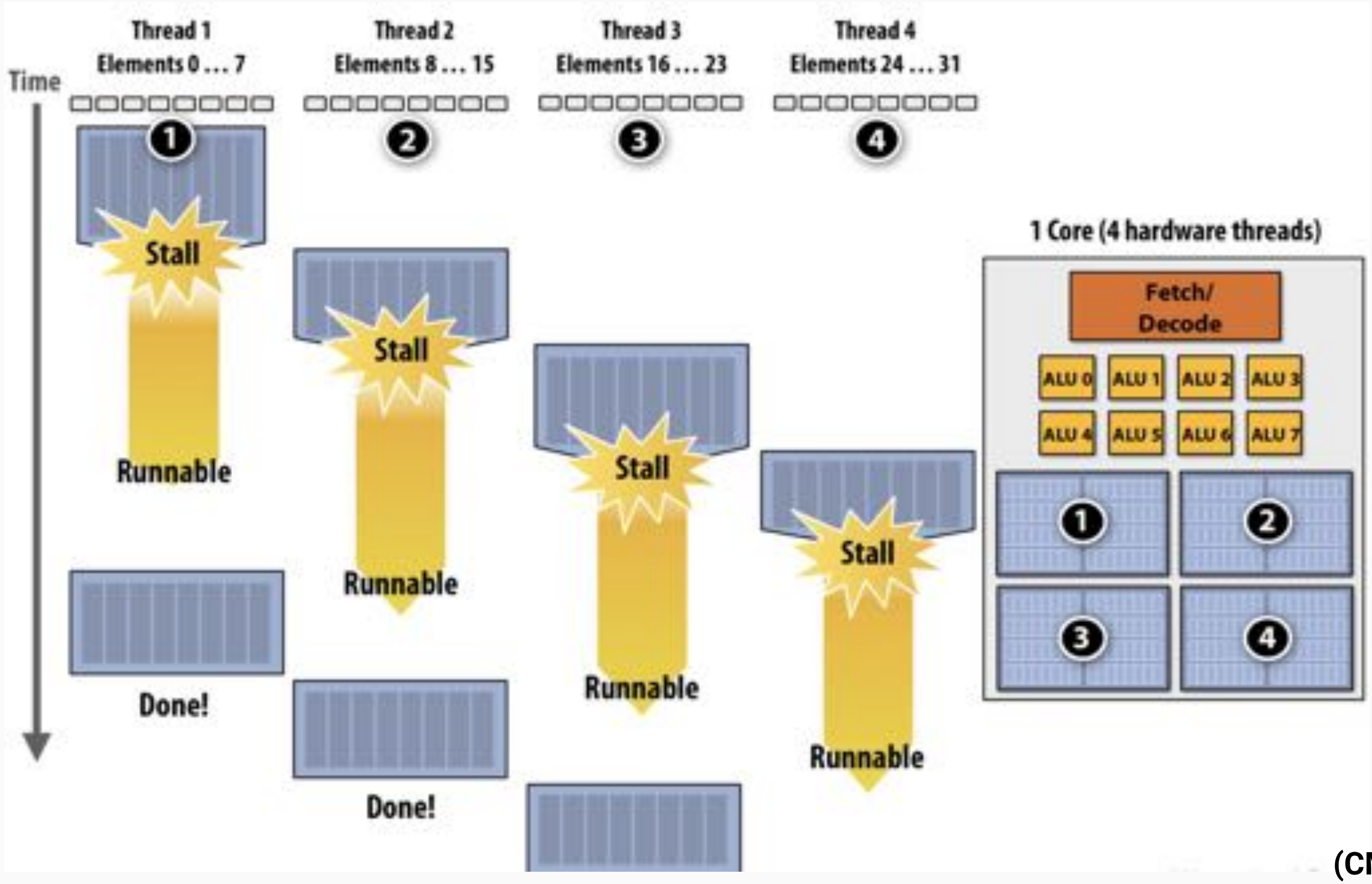
(CMU 15-418/618)

## If one thread blocks, run the other



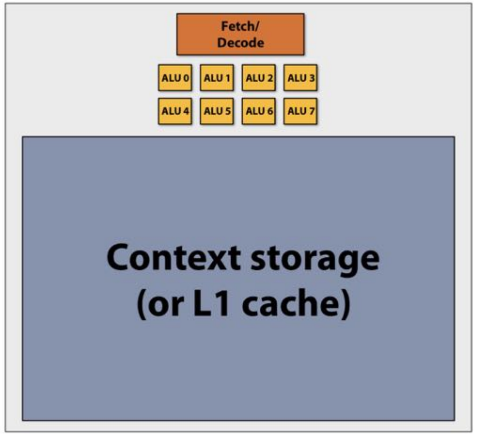
(CMU 15-418/618)

## Have enough blocks to hide latency

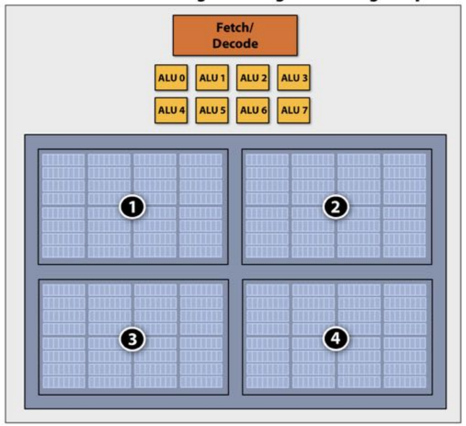


(CMU 15-418/618)

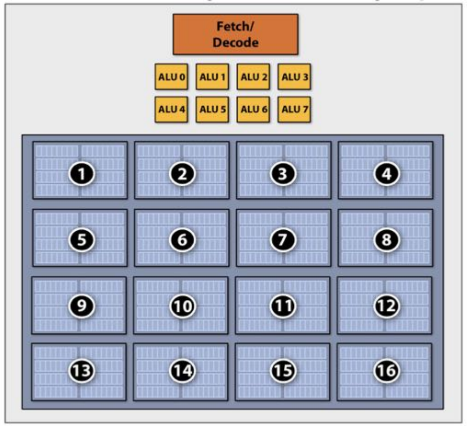
## Trade off in context size



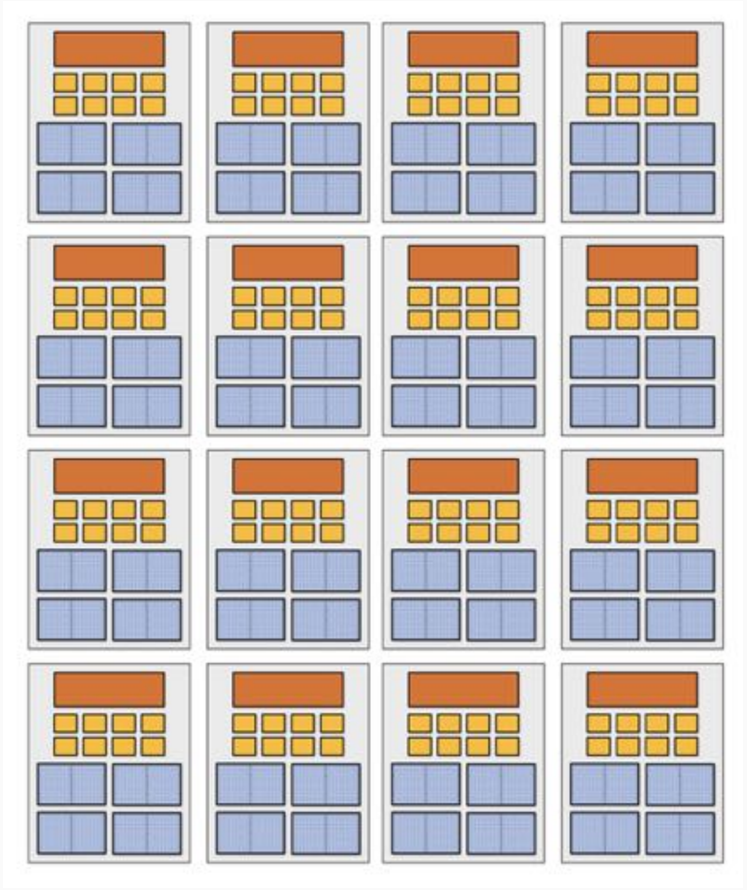
VS



VS

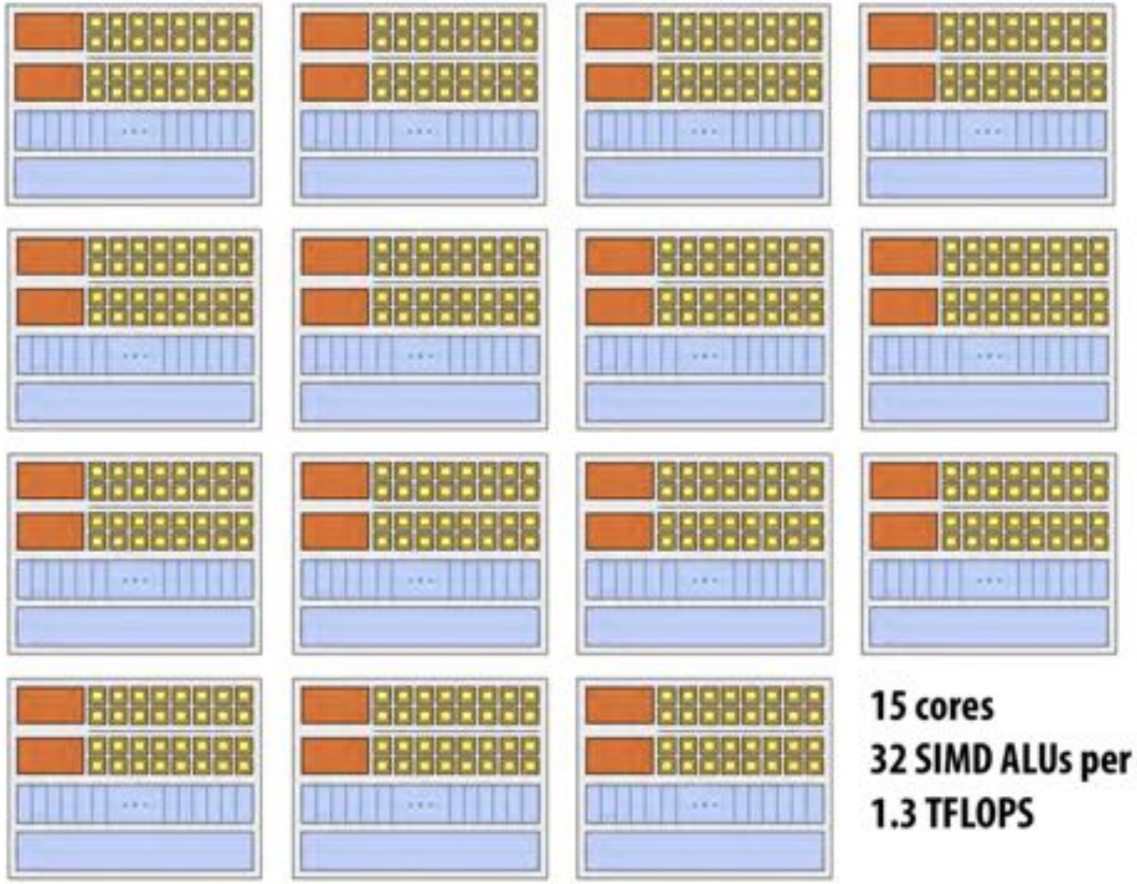


## Fictitious chip



| | |
|---|---|
| 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

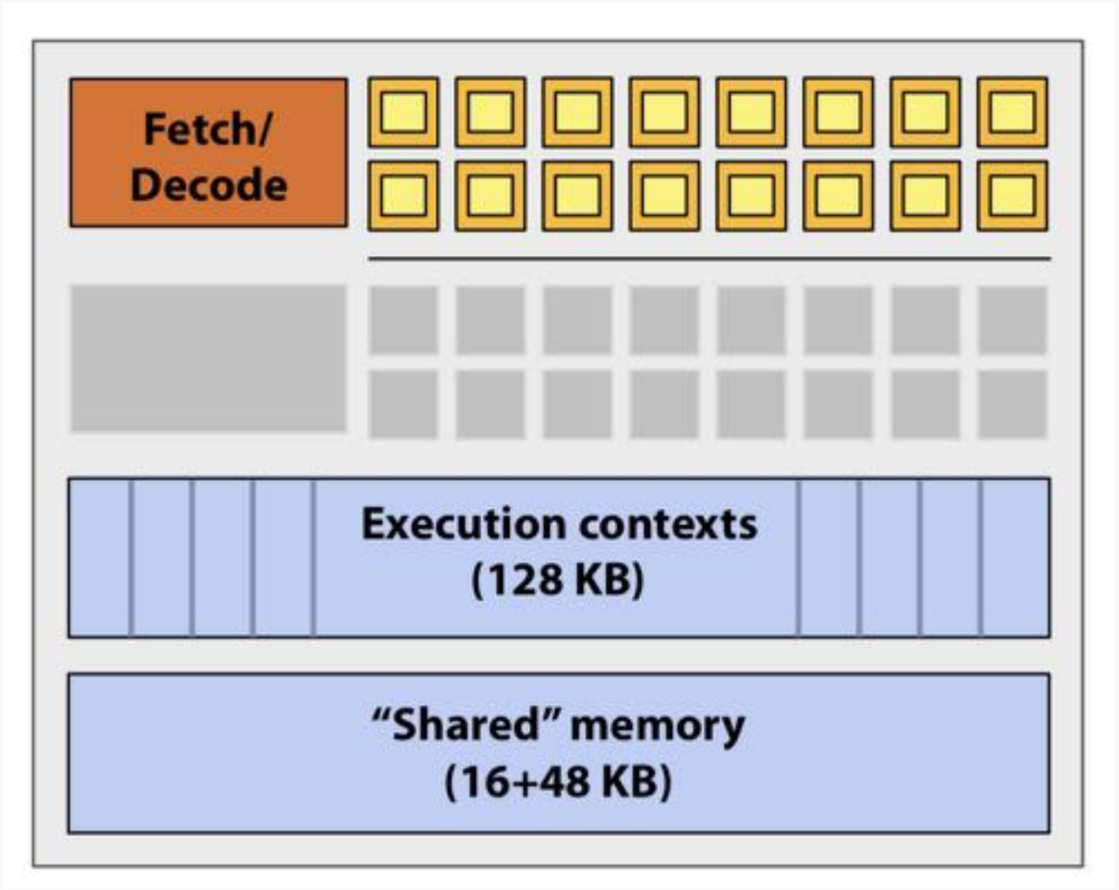


15 cores

32 SIMD ALUs per core

1.3 TFLOPS

## The GPU architecture



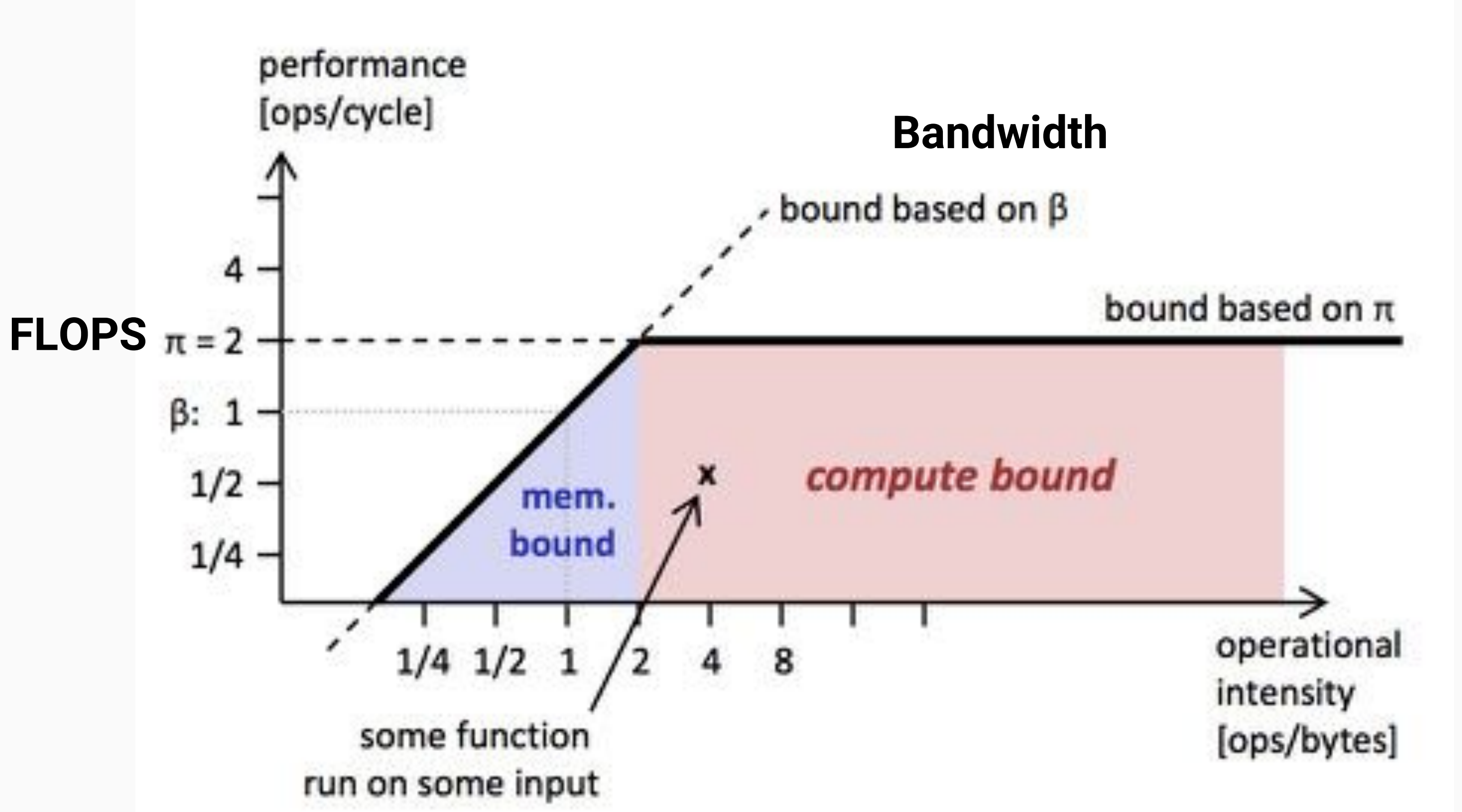
Each core has 16 ALUs running at twice the clock speed => SIMD instruction working on 32 data items

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

(CMU 15-418/618)

## Roofline model



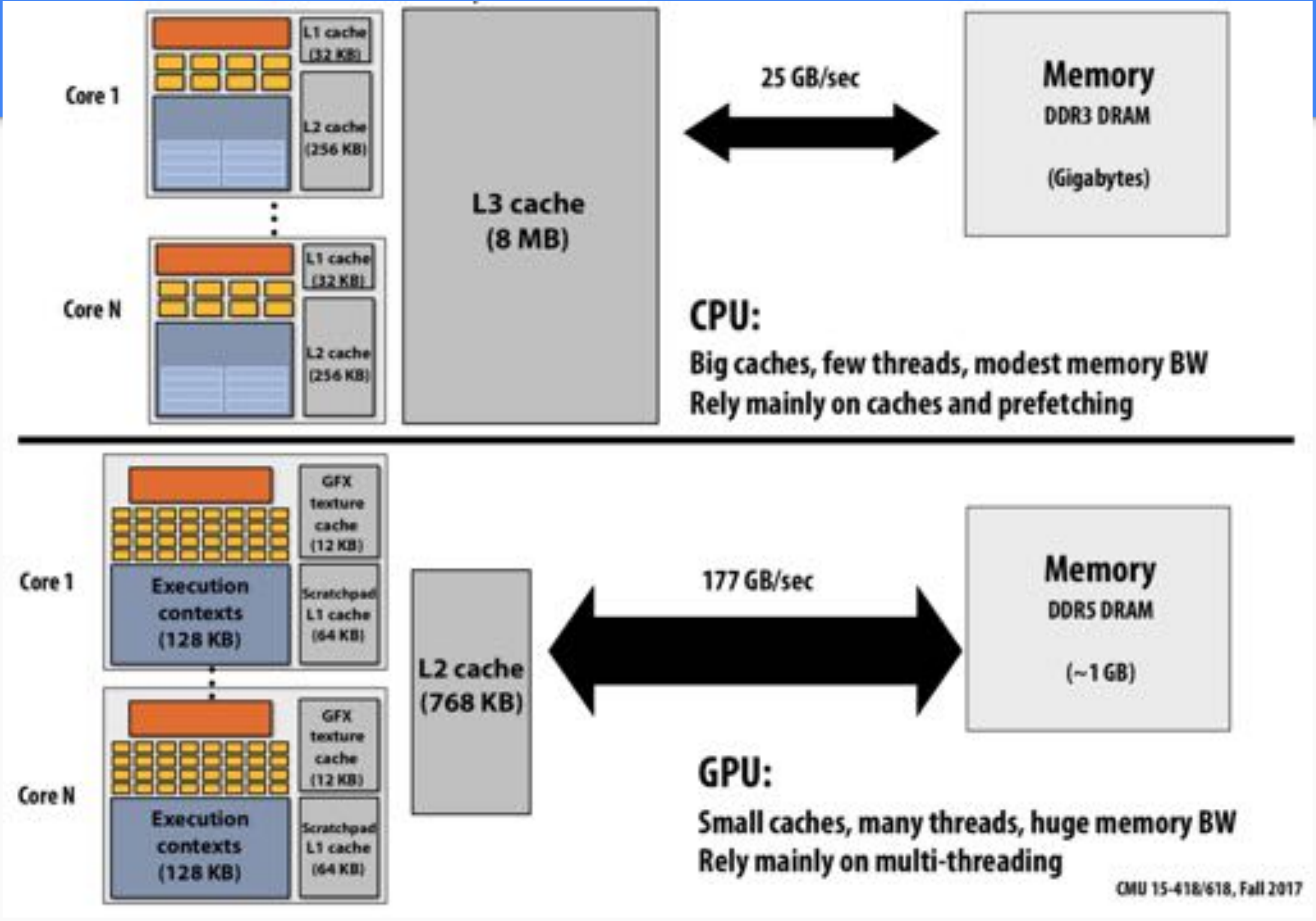
https://spcl.inf.ethz.ch/Teaching/2017-dphpc/lecture/lecture6-roofline.pdf

## 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



**CPU:**
Big caches, few threads, modest memory BW
Rely mainly on caches and prefetching

**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!