CS260 – Lecture 9

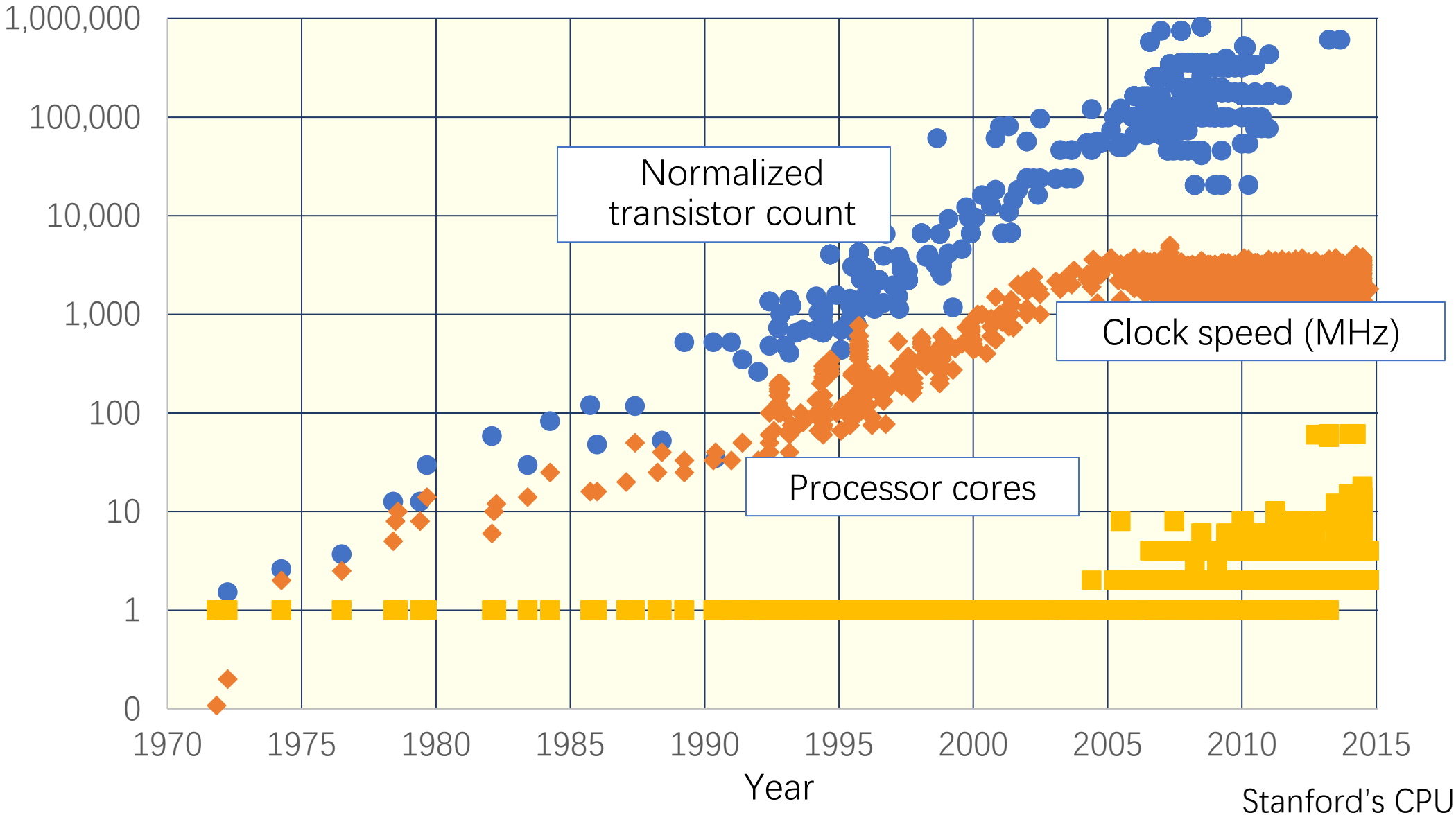
Yan Gu

# Algorithm Engineering (aka. How to Write Fast Code)

## An Overview of Computer Architecture

Many slides in this lecture are borrowed from the first and second lecture in Stanford CS149 Parallel Computing. The credit is to Prof. Kayvon Fatahalian, and the instructor appreciates the permission to use them in this course.

# Moore's law: #transistors doubles every 18 months




Stanford's CPU DB [DKM12]

# Until ~15 years ago: two significant reasons for processor performance improvement

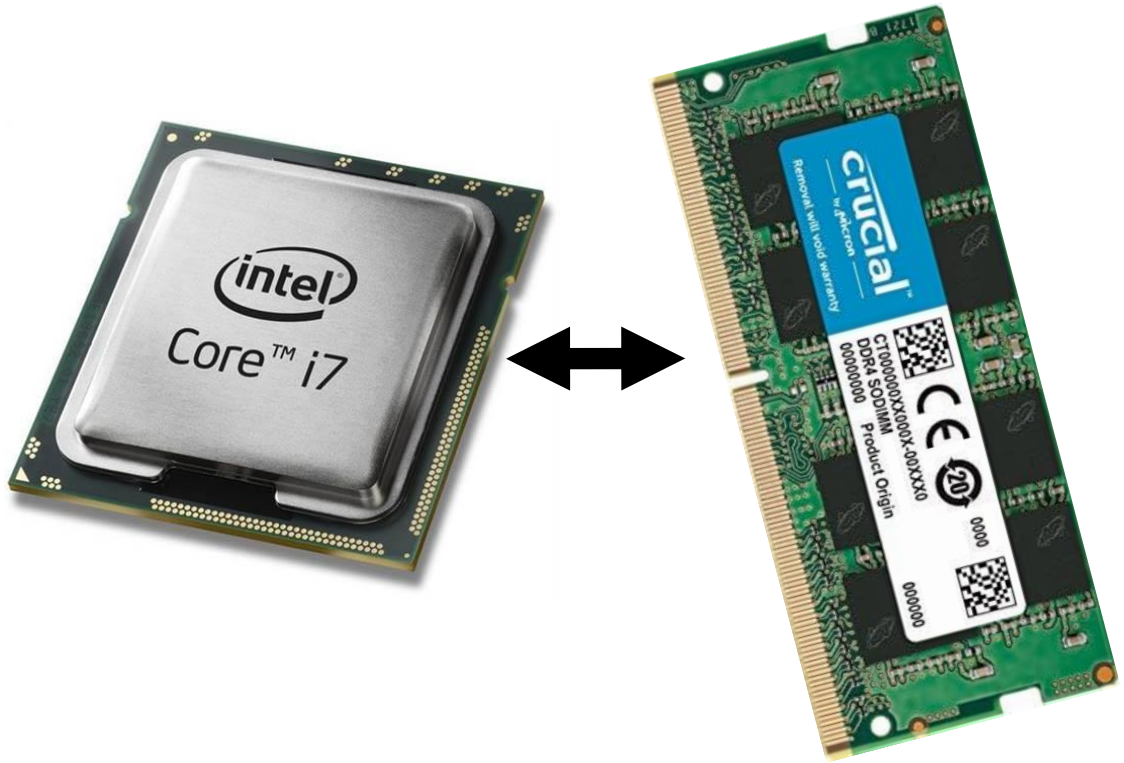
- **Increasing CPU clock frequency**
  - No longer work after 2005 due to energy issues
- **Exploiting instruction-level parallelism (superscalar execution)**
  - Have a ceiling on the parallelism we can get (<3)

# Part 1: Parallel Execution

# Summary: parallel execution

- **Several forms of parallel execution in modern processors**
  - **Superscalar: exploit ILP within an instruction stream. Process different instructions from the same instruction stream in parallel (within a core)**
    - Parallelism automatically and dynamically discovered by the hardware during execution (not programmer visible)
  - **Multi-core: use multiple processing cores**
    - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
    - Programmers/algorithms decide when to do so (e.g., via cilk_spawn, cilk_for)
  - **SIMD: use multiple ALUs controlled by same instruction stream (within a core)**
    - Efficient design for data-parallel workloads: control amortized over many ALUs
    - Vectorization usually declared by programmer, but can be inferred by loop analysis by advanced compiler

# Part 2: Accessing Memory



# Terminology

- **Memory latency**
  - The amount of time for a memory request (e.g., load, store) from a processor to be serviced by the memory system
  - Example: 100 cycles, 100 nsec

- **Memory bandwidth**
  - The rate at which the memory system can provide data to a processor
  - Example: 20 GB/s

# Stalls

- **A processor "stalls" when it cannot run the next instruction in an instruction stream because of a dependency on a previous instruction.**

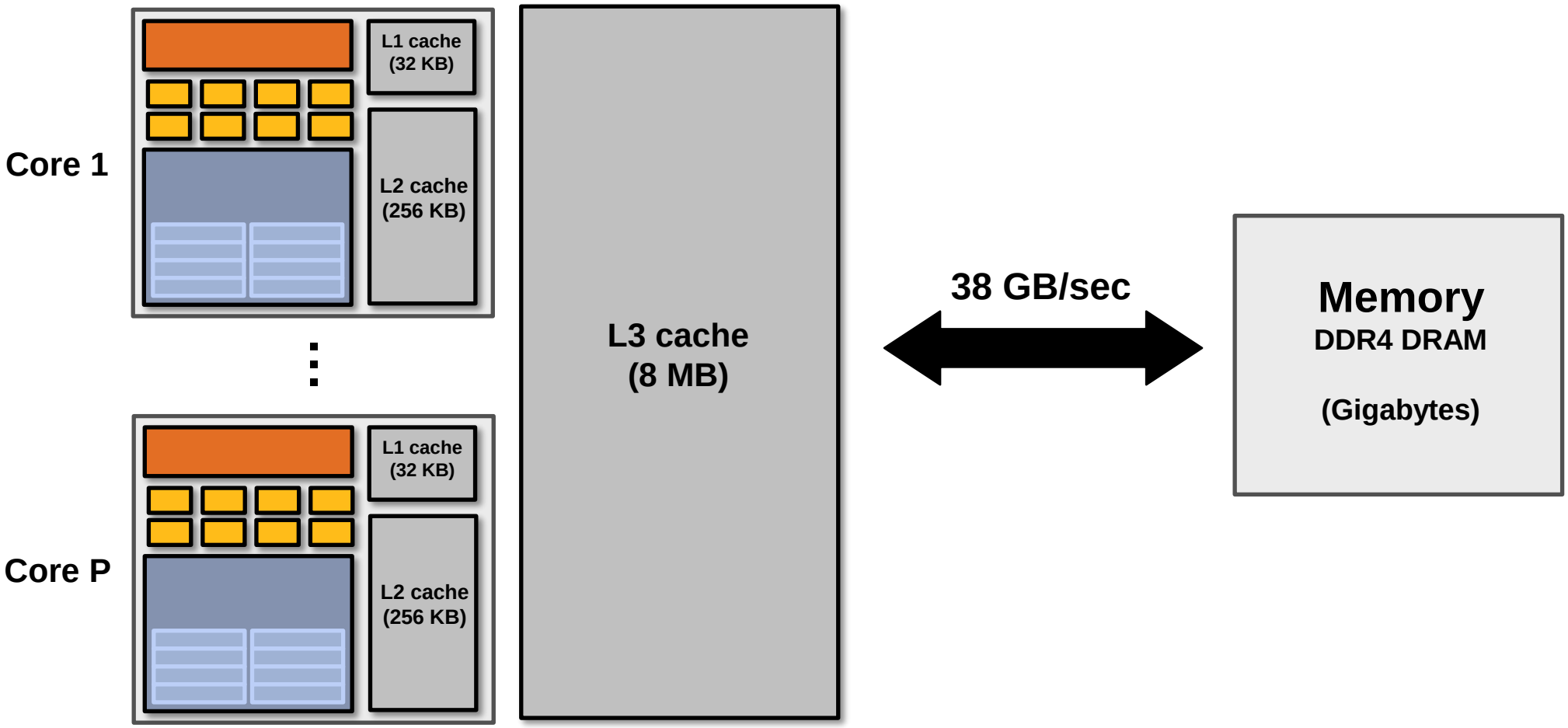
- **Accessing memory is a major source of stalls**

```
ld r0 mem[r2]
ld r1 mem[r3]
add r0, r0, r1
```

**Dependency: cannot execute 'add' instruction until data at mem[r2] and mem[r3] have been loaded from memory**

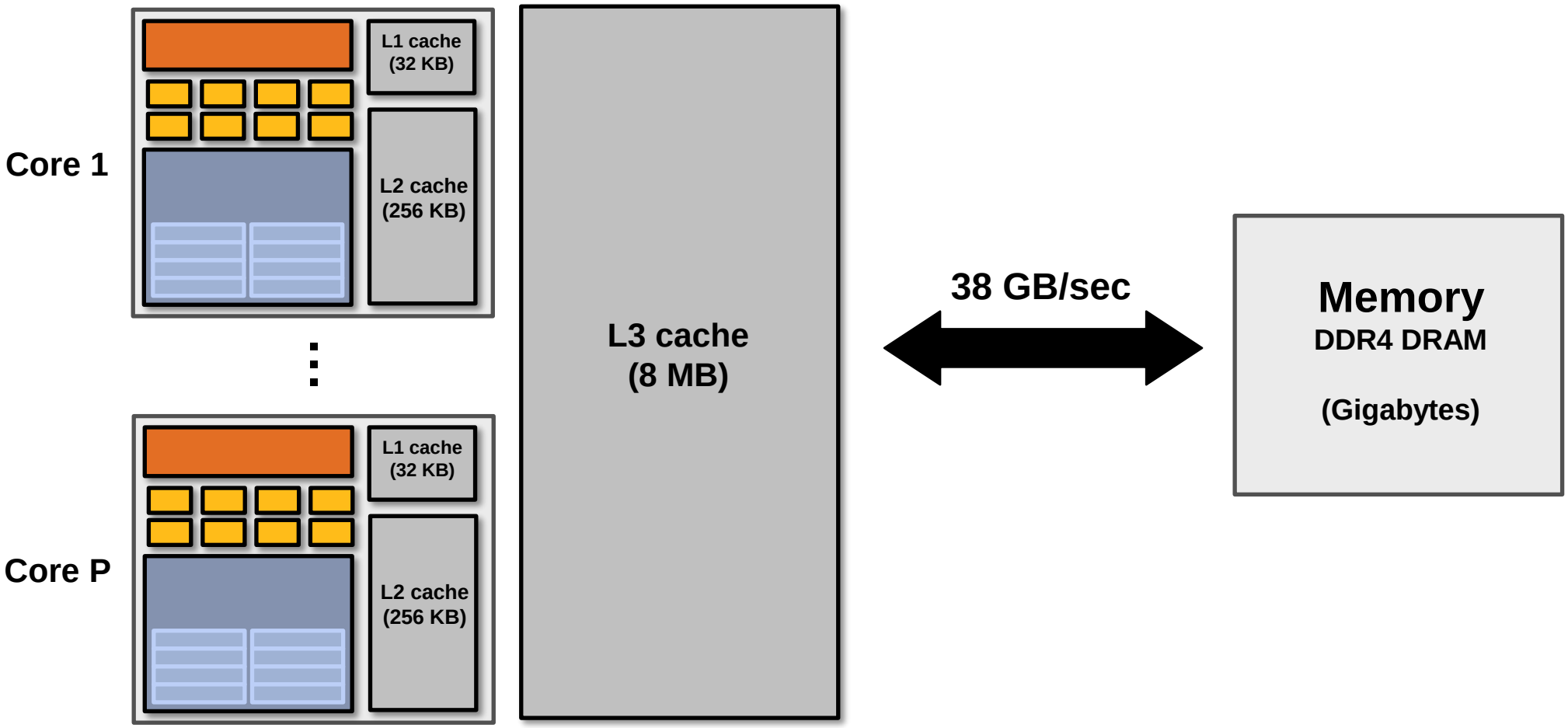
- **Memory access times ~ 100's of cycles**
  - Memory "access time" is a measure of latency

# Review: why do modern processors have caches?



# Caches reduce length of stalls (reduce latency)

**Processors run efficiently when data is resident in caches**
**Caches reduce memory access latency ***

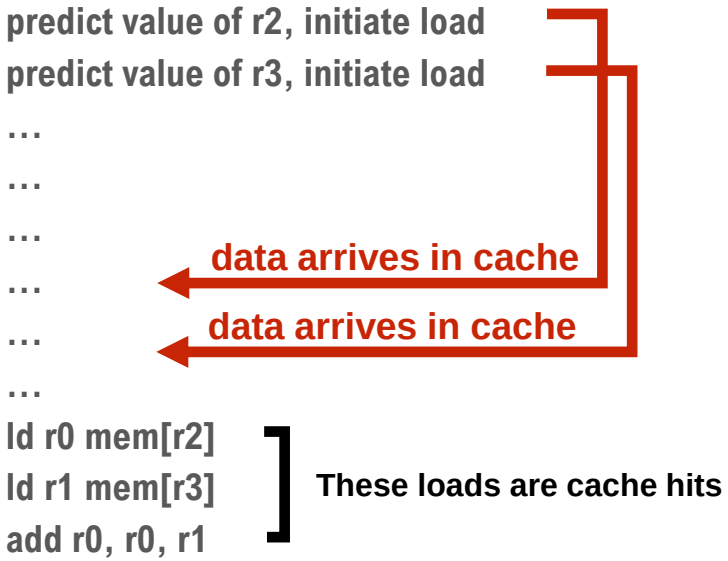



* Caches also provide high bandwidth data transfer to CPU

# Prefetching reduces stalls (hides latency)

- **All modern CPUs have logic for prefetching data into caches**
  - Dynamically analyze program's access patterns, predict what it will access soon

- **Reduces stalls since data is resident in cache when accessed**

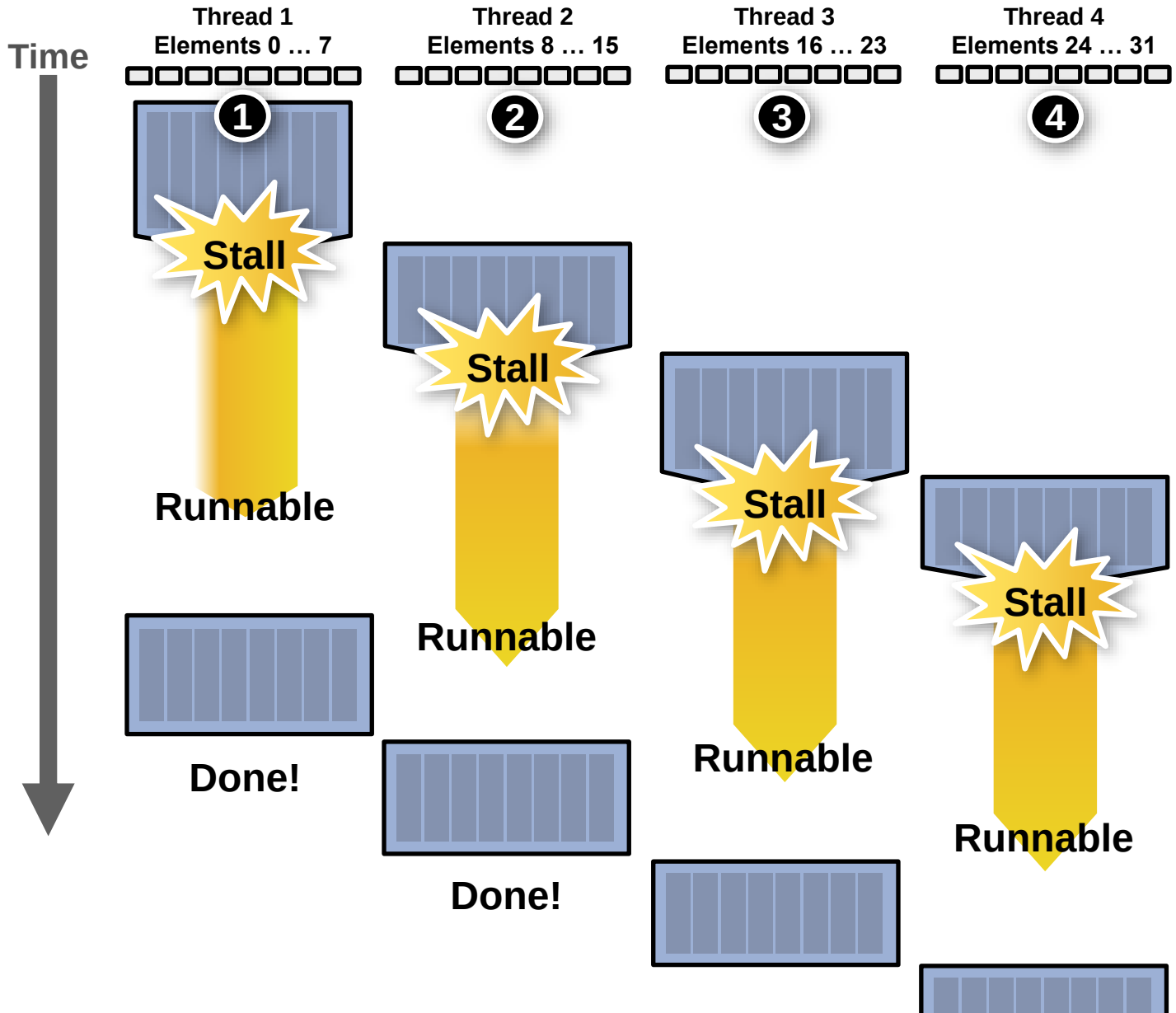


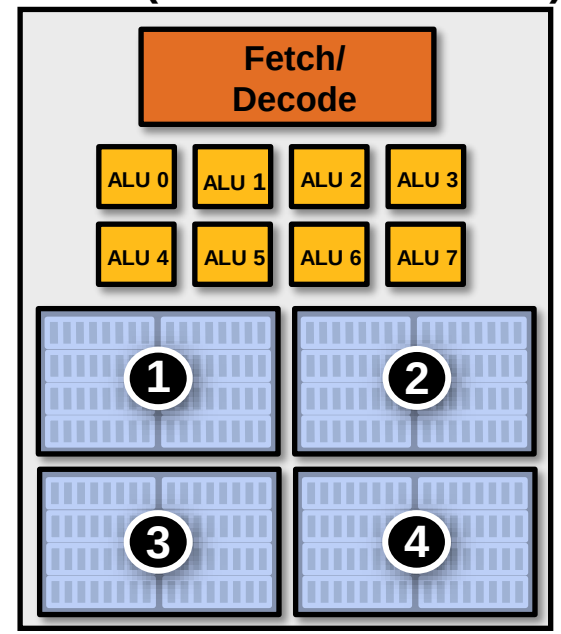

**Note: Prefetching can also reduce performance if the guess is wrong (hogs bandwidth, pollutes caches)**

# Multi-threading reduces stalls

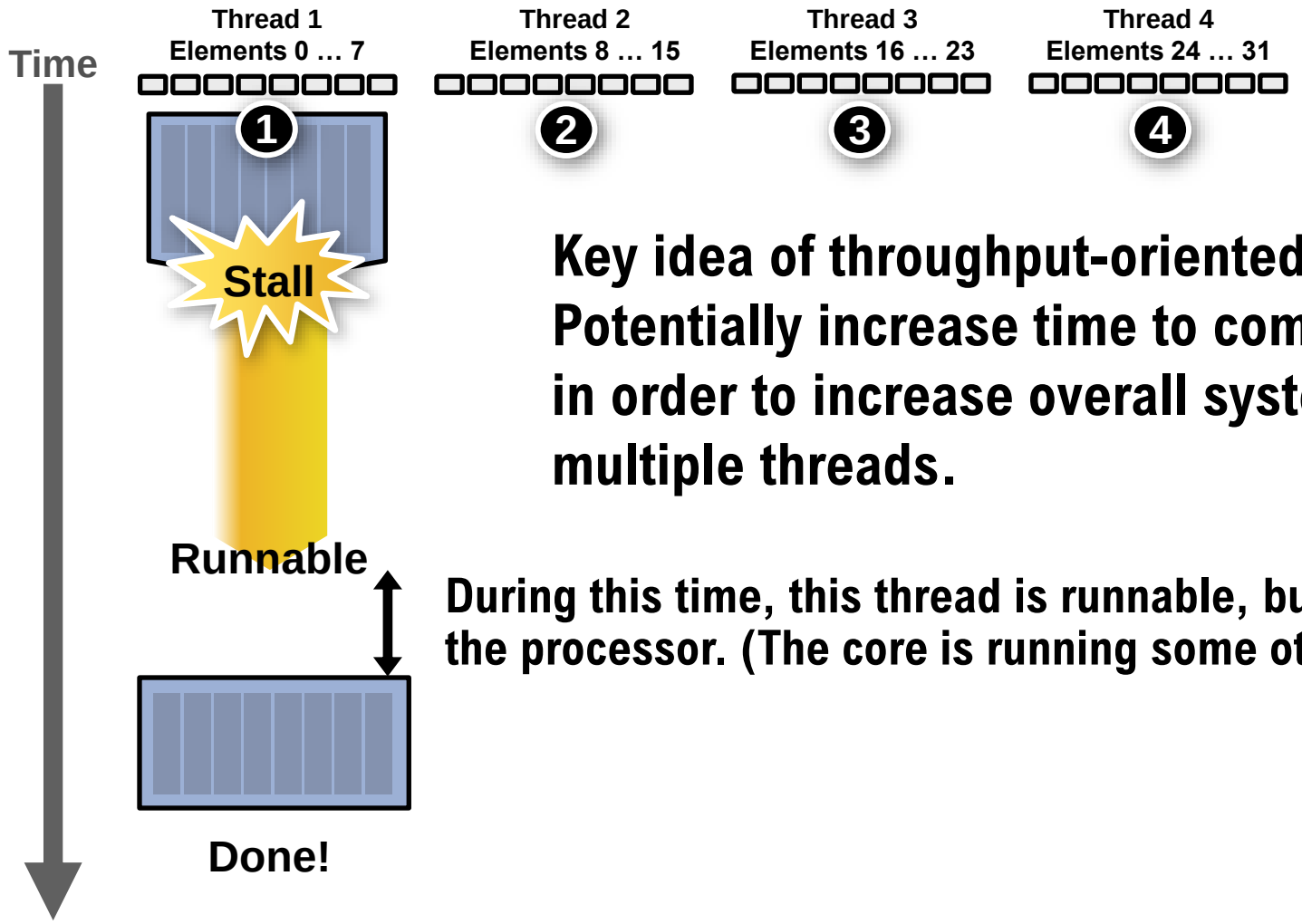
- Idea: interleave processing of multiple threads on the same core to hide stalls
- Like prefetching, multi-threading is a latency hiding, not a latency reducing technique

# Hiding stalls with multi-threading



# Throughput computing trade-off

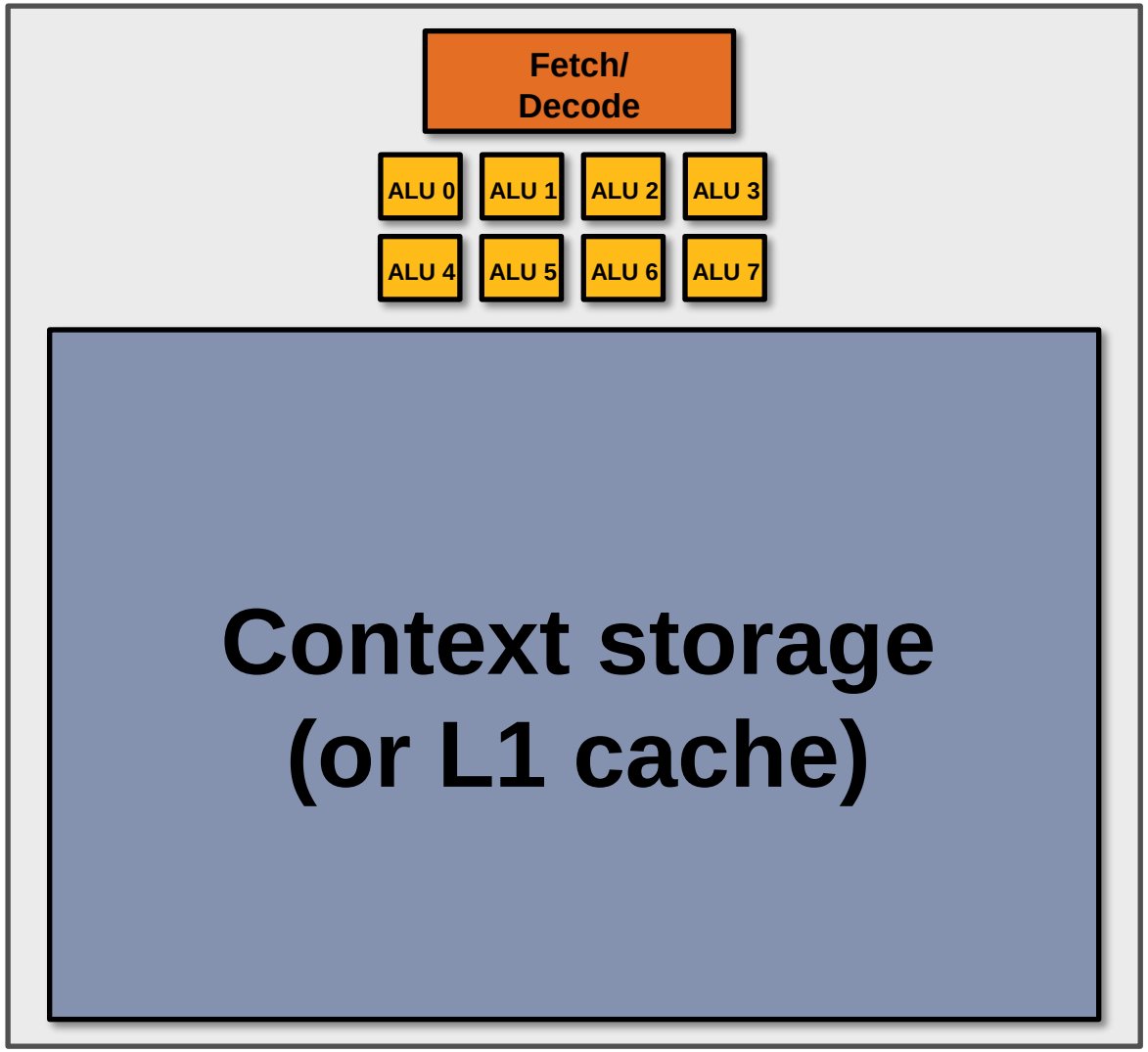



**Key idea of throughput-oriented systems:**
**Potentially increase time to complete work by any one thread, in order to increase overall system throughput when running multiple threads.**

**During this time, this thread is runnable, but it is not being executed by the processor. (The core is running some other thread.)**

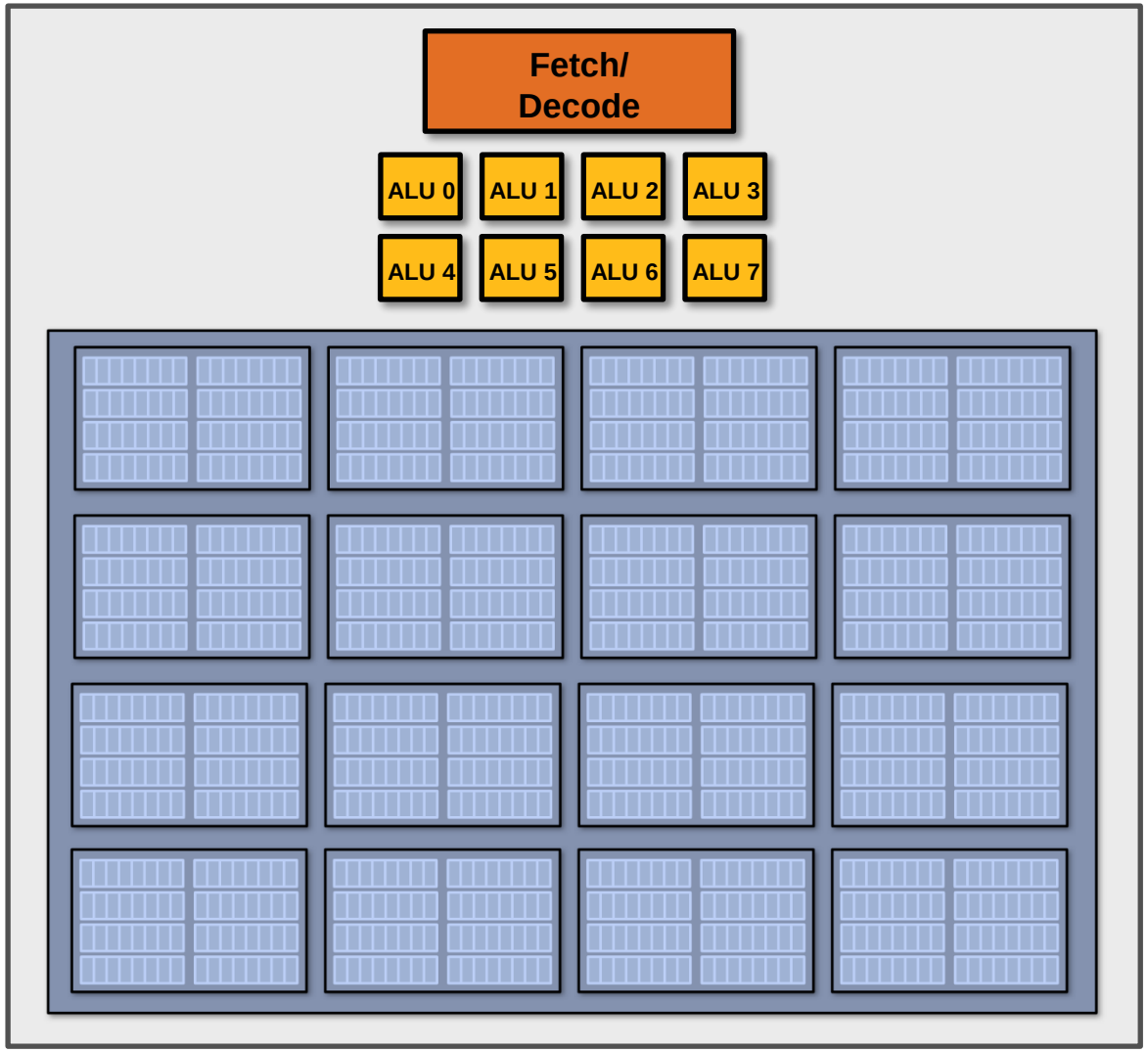
# Storing execution contexts

**Consider on-chip storage of execution contexts a finite resource.**



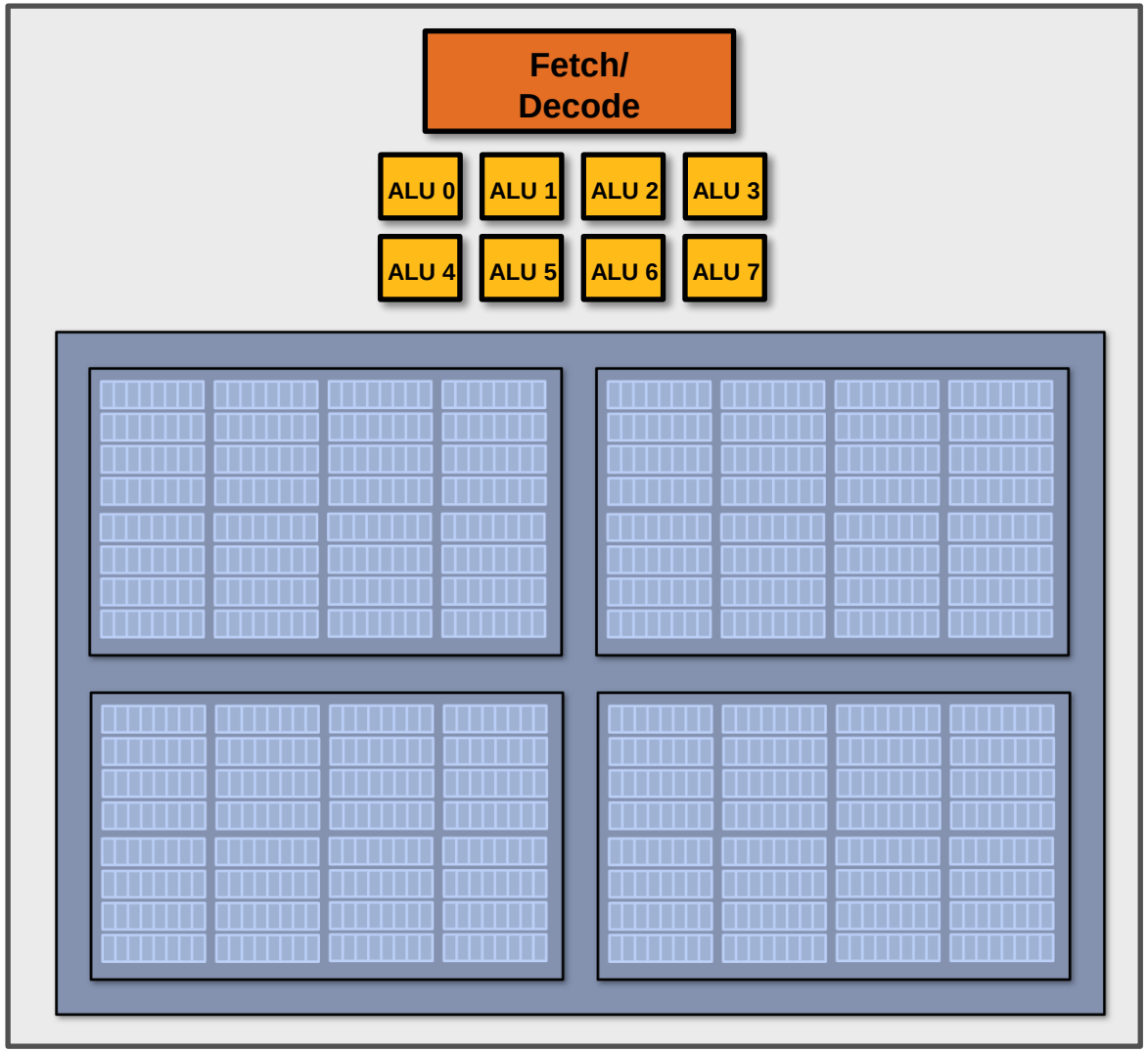
# Many small contexts (high latency hiding ability)

**1 core (16 hardware threads, storage for small working set per thread)**



# Four large contexts (low latency hiding ability)

**1 core (4 hardware threads, storage for larger working set per thread)**



# Hardware-supported multi-threading

- **Core manages execution contexts for multiple threads**
  - Runs instructions from runnable threads (processor makes decision about which thread to run each clock, not the operating system)
  - Core still has the same number of ALU resources: multi-threading only helps use them more efficiently in the face of high-latency operations like memory access
- **Interleaved multi-threading (a.k.a. temporal multi-threading)**
  - What I described on the previous slides: each clock, the core chooses a thread, and runs an instruction from the thread on the ALUs
- **Simultaneous multi-threading (SMT)**
  - Each clock, core chooses instructions from multiple threads to run on ALUs
  - Extension of superscalar CPU design
  - Example: Intel Hyper-threading (2 threads per core)

# Multi-threading summary

- **Benefit: use a core's execution resources (ALUs) more efficiently**
  - Hide memory latency
  - Fill multiple functional units of superscalar architecture
  - (when one thread has insufficient ILP)

- **Costs**
  - Requires additional storage for thread contexts
  - Increases run time of any single thread
    (often not a problem, we usually care about throughput in parallel apps)
  - Requires additional independent work in a program (more independent work than ALUs!)
  - Relies heavily on memory bandwidth
    - More threads → larger working set → less cache space per thread
    - May go to memory more often, but can hide the latency

# A fictitious multi-core chip

16 cores

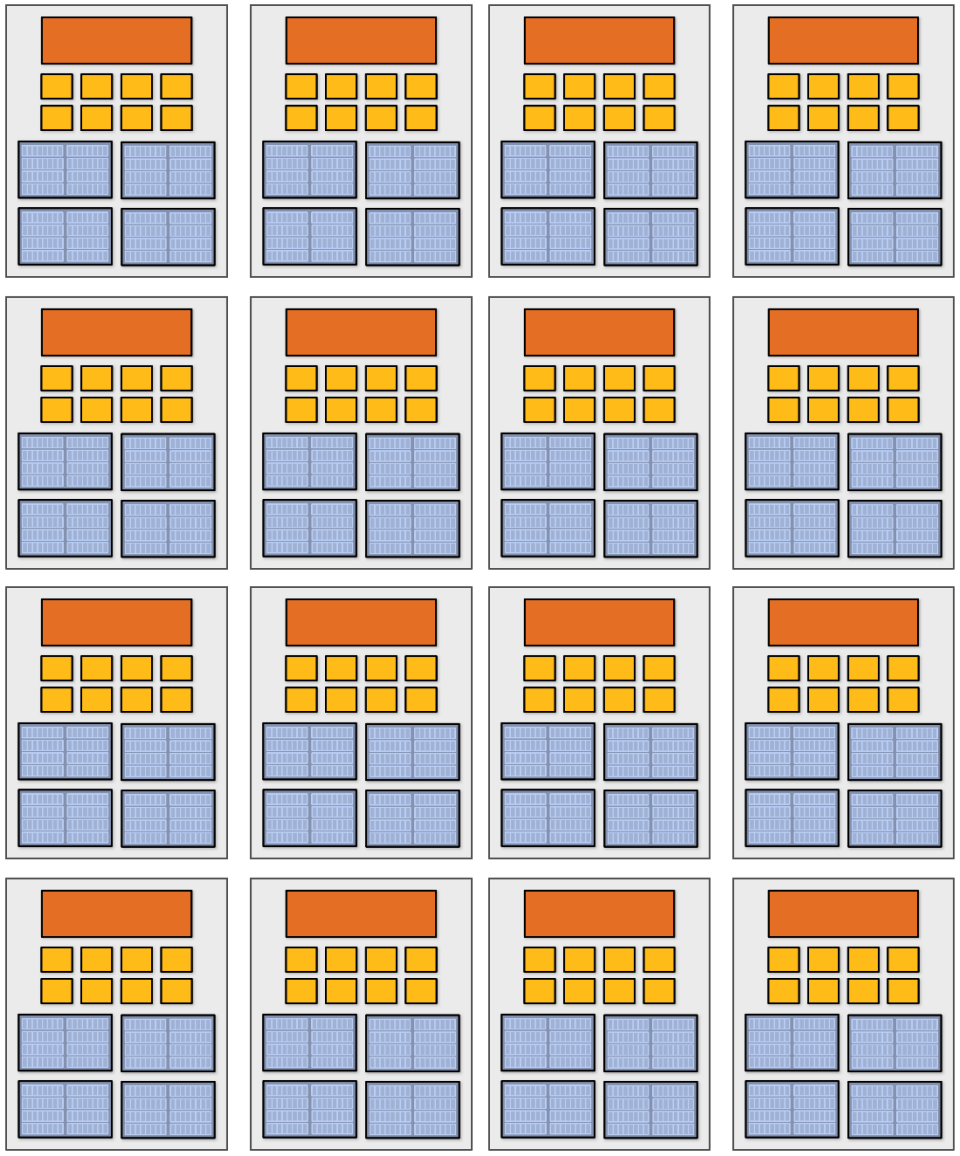
8 SIMD ALUs per core (128 total)

4 threads per core

16 simultaneous instruction streams

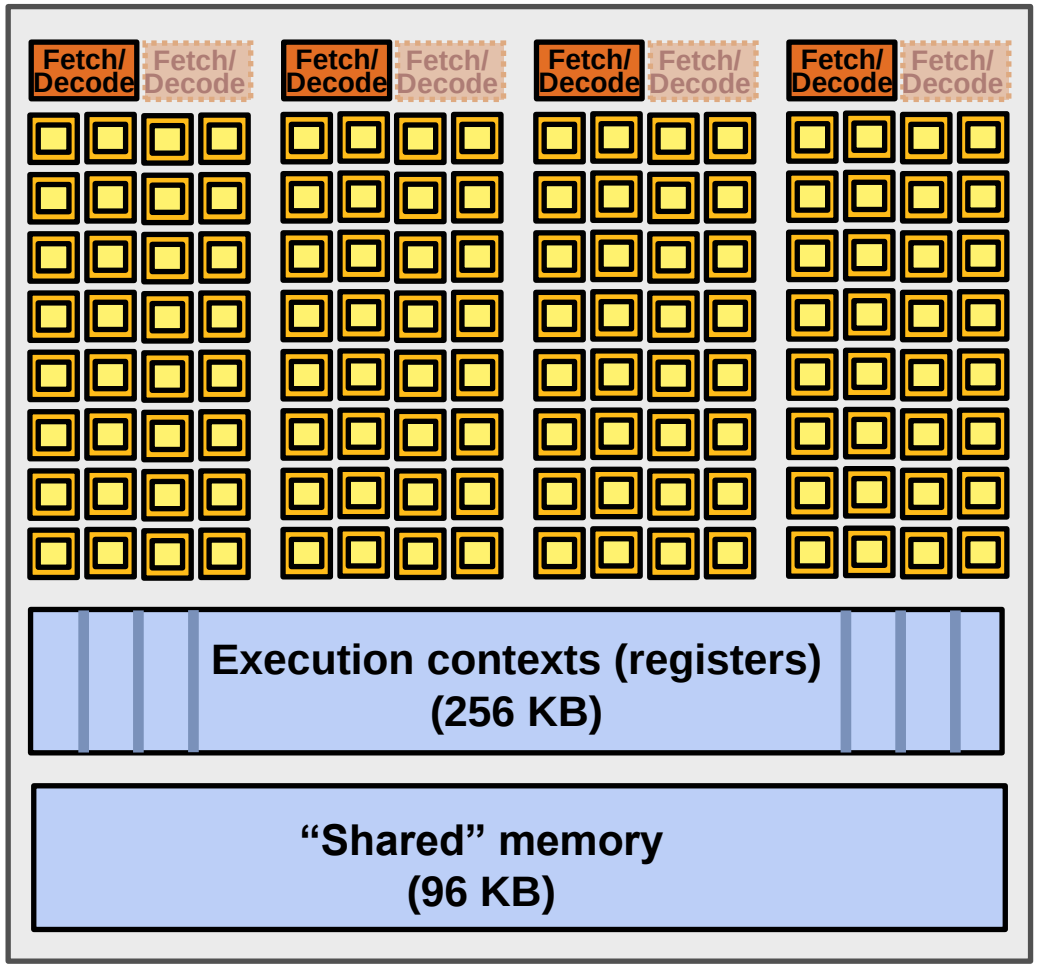
64 total concurrent instruction streams

512 independent pieces of work are needed to run chip with maximal latency hiding ability



# GPUs: extreme throughput-oriented processors

## NVIDIA GTX 1080 core ("SM")

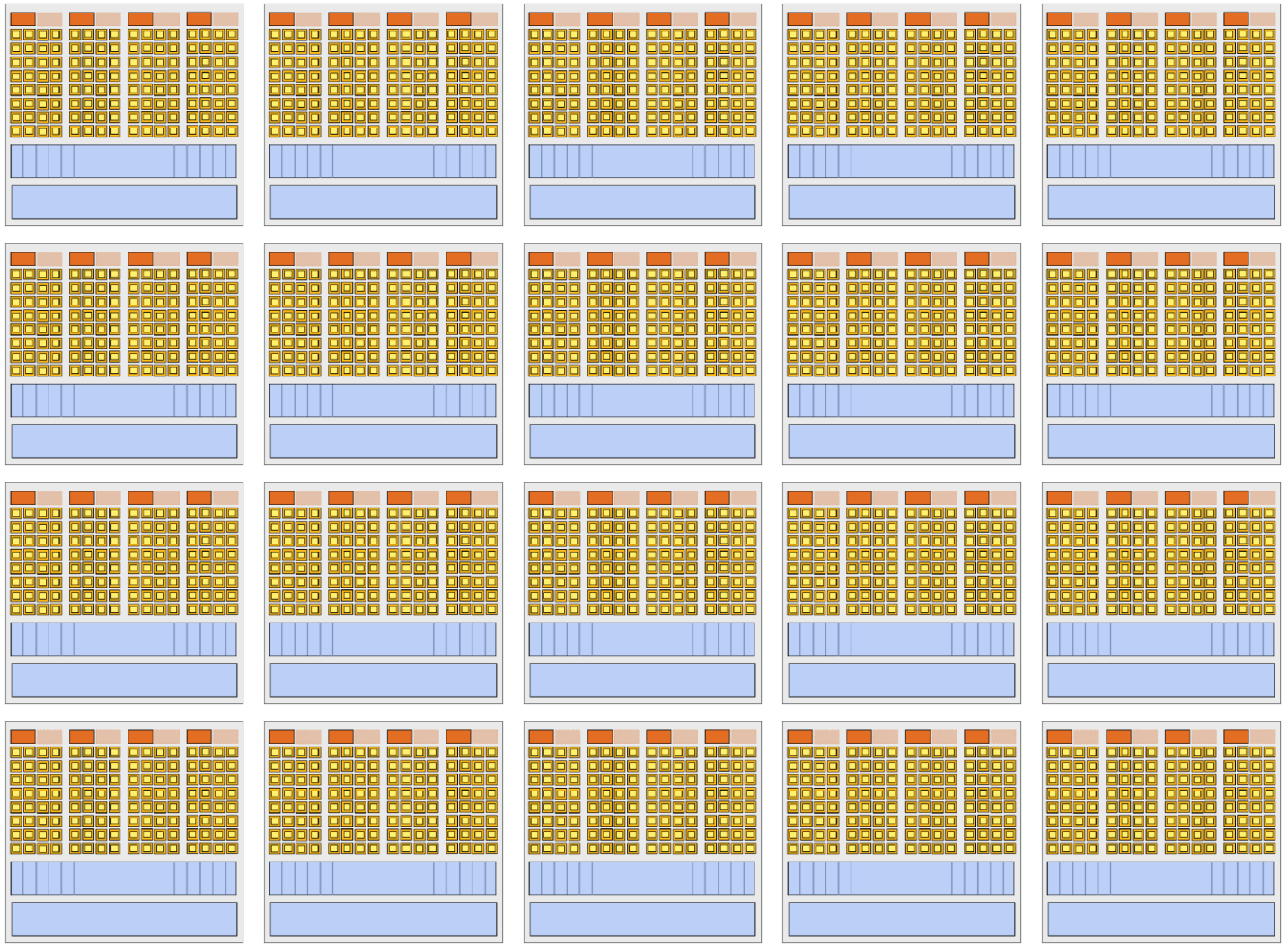



Source: NVIDIA Pascal Tuning Guide

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

- Instructions operate on 32 pieces of data at a time (instruction streams called "warps").
- Think: warp = thread issuing 32-wide vector instructions
- Different instructions from up to four warps can be executed simultaneously (simultaneous multi-threading)
- Up to 64 warps are interleaved on the SM (interleaved multi-threading)
- Over 2,048 elements can be processed concurrently by a core

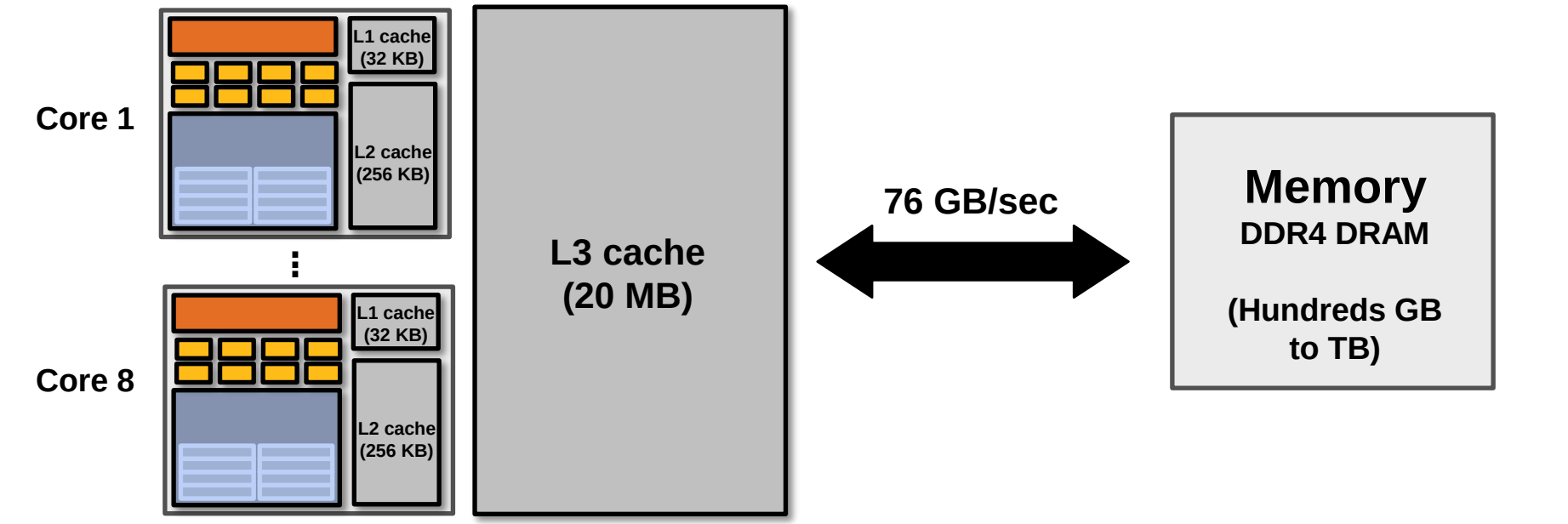
# NVIDIA GTX 1080



**There are 20 SM cores on the GTX 1080:**

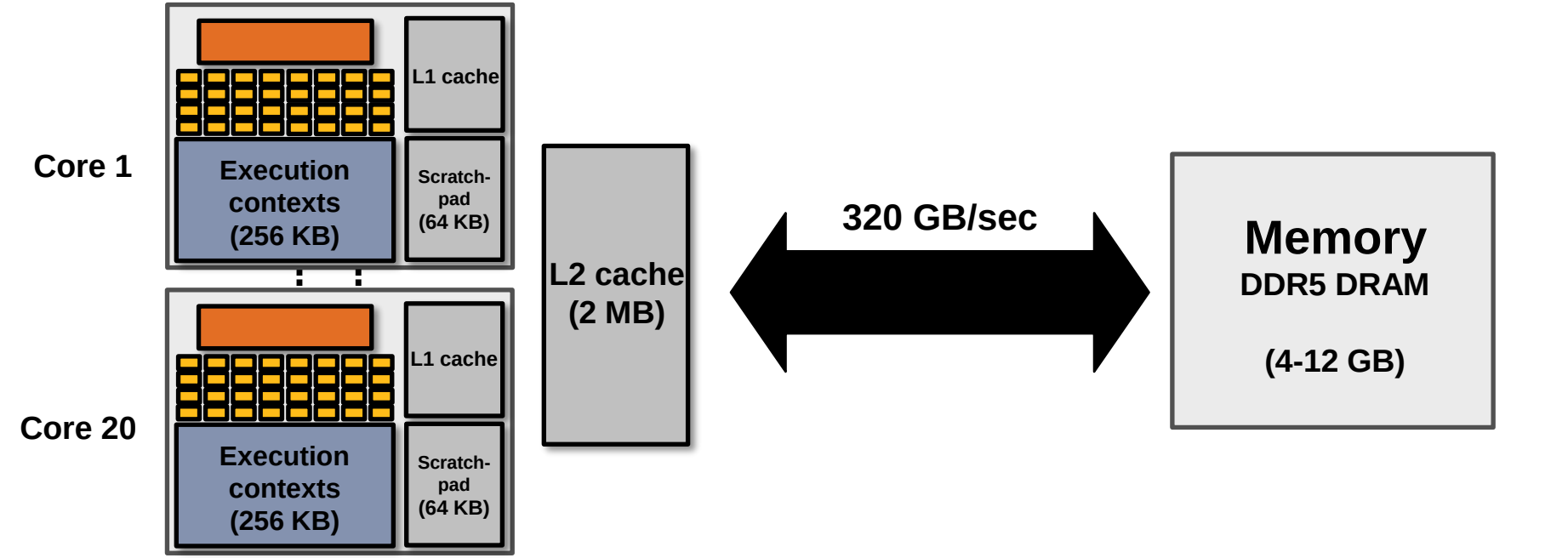
**That's 40,960 pieces of data being processed concurrently to get maximal latency hiding!**

# CPU vs. GPU memory hierarchies




**CPU:**

**Big caches, few threads per core, modest memory BW**
**Rely mainly on caches and prefetching (automatic)**




**GPU:**

**Small caches, many threads, huge memory BW**
**Rely heavily on multi-threading for performance (manual)**

# 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

**Performant parallel programs will:**

- **Organize computation to fetch data from memory less often**
  - Reuse data previously loaded by the same thread (traditional intra-thread temporal locality optimizations)
  - Share data across threads (inter-thread cooperation)

- **Request data less often (instead, do more arithmetic: it's "free")**
  - Useful term: "arithmetic intensity" — ratio of math operations to data access operations in an instruction stream
  - Main point: programs must have high arithmetic intensity to utilize modern processors efficiently

# Summary

- **Three major ideas that all modern processors employ to varying degrees**
  - **Provide multiple processing cores**
    - Simpler cores (embrace thread-level parallelism over instruction-level parallelism)
  - **Amortize instruction stream processing over many ALUs (SIMD)**
    - Increase compute capability with little extra cost
  - **Use multi-threading to make more efficient use of processing resources (hide latencies, fill all available resources)**
- **Due to high arithmetic capability on modern chips, many parallel applications (on both CPUs and GPUs) are bandwidth bound**
- **GPU architectures use the same throughput computing ideas as CPUs: but GPUs push these concepts to extreme scales**

# Review slides

(additional examples for review and to check our understanding)

# Putting together the concepts from the lectures:

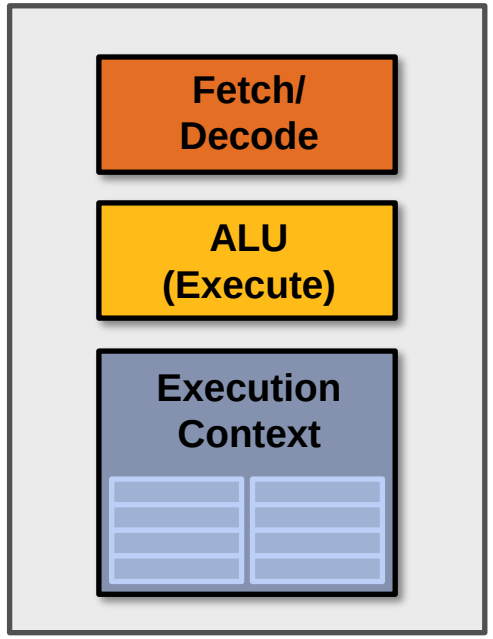
(if you understand the following sequence you understand this lecture)

# Running code on a simple processor

## My very simple program: compute sin(*x*) using Taylor expansion

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
         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;
   }
}
```

## My very simple processor: completes one instruction per clock



# Review: superscalar execution

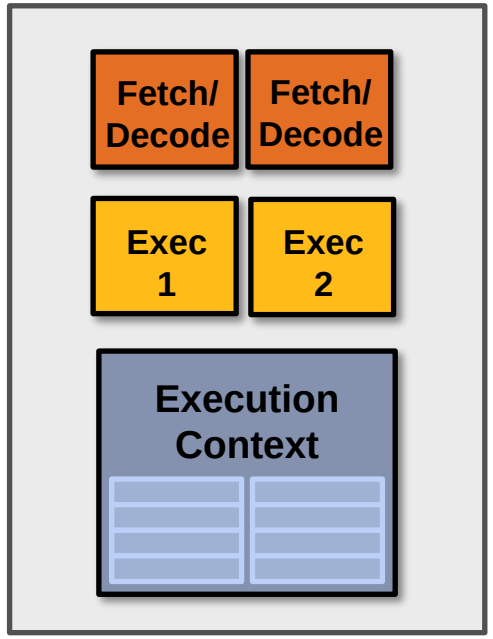
## Unmodified program

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
         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;
   }
}
```

Independent operations in instruction stream

(They are detected by the processor at run-time and may be executed in parallel on execution units 1 and 2)

My single core, superscalar processor: executes up to two instructions per clock from a single instruction stream.

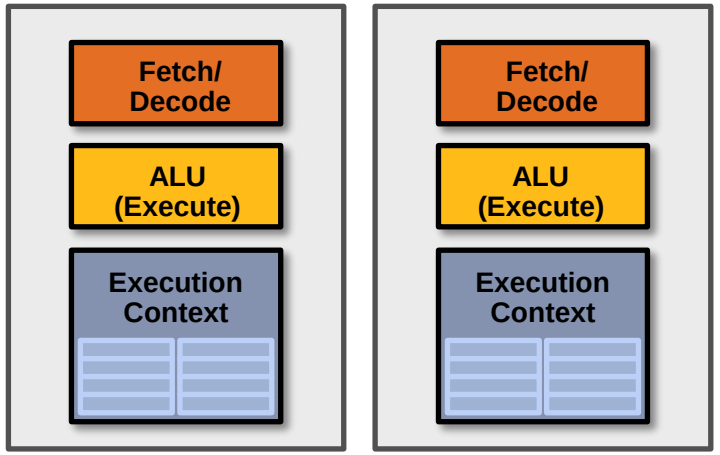


# Review: multi-core execution (two cores)

## Modify program to create multiple threads (multiple instruction streams)

```
void sinx(int N, int terms, float* x, float* result)
{
   cilk_for (int i=0; i<N; i++)
   {
         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;
   }
}
```

**My dual-core processor:**
**executes one instruction per clock from an instruction stream on each core.**

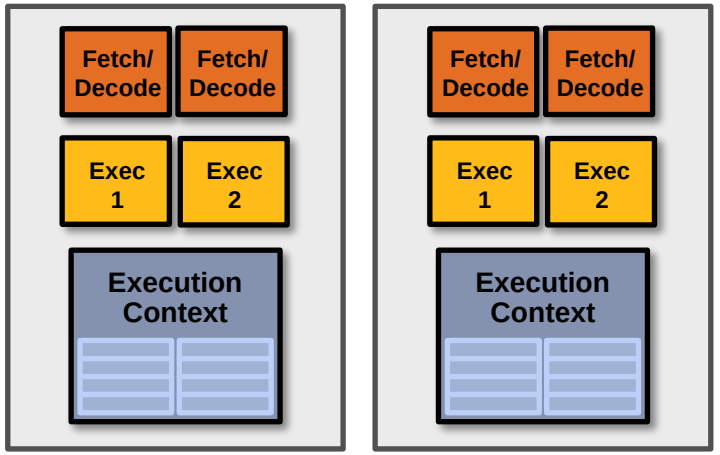


# Review: multi-core + superscalar execution

## Modify program to create multiple threads (multiple instruction streams)

```
void sinx(int N, int terms, float* x, float* result)
{
   cilk_for (int i=0; i<N; i++)
   {
         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;
   }
}
```

**My superscalar dual-core processor: executes up to two instructions per clock from an instruction stream on each core.**

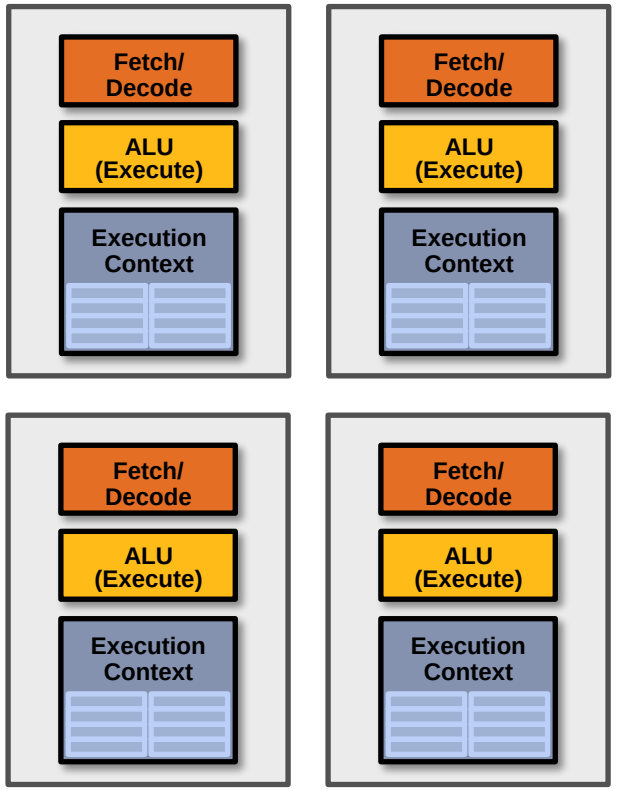


# Review: multi-core (four cores)

## Modify program to create multiple threads (multiple instruction streams)

```
void sinx(int N, int terms, float* x, float* result)
{
   cilk_for (int i=0; i<N; i++)
   {
         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;
   }
}
```

**My quad-core processor:**
**executes one instruction per clock**
**from an instruction stream on each core.**



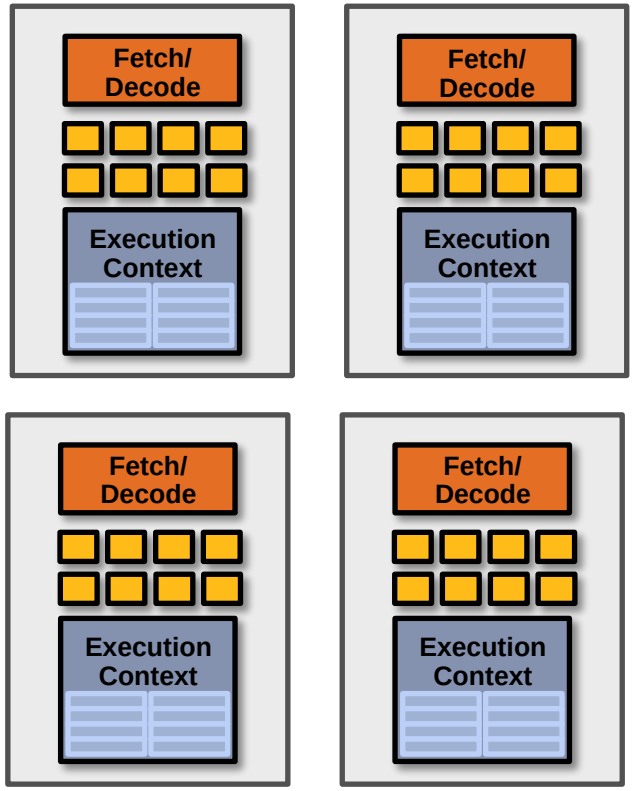
# Review: four, 8-wide SIMD cores

**Observation: program must execute many iterations of the same loop body.**

**Optimization: share instruction stream across execution of multiple iterations (single instruction multiple data = SIMD)**

```
void sinx(int N, int terms, float* x, float* result)
{
   cilk_for (int i=0; i<N; i++)
   {
         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;
   }
}
```

**My SIMD quad-core processor:**
**executes one 8-wide SIMD instruction per clock from an instruction stream on each core.**



# Review: four SIMD, multi-threaded cores

**Observation: memory operations have very long latency**

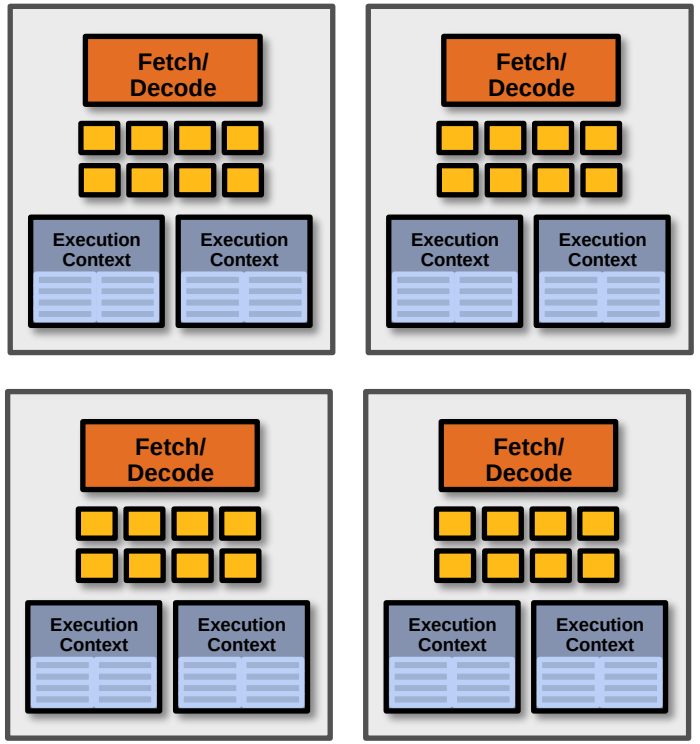
**Solution: hide latency of loading data for one iteration by executing arithmetic instructions from other iterations**

```
void sinx(int N, int terms, float* x, float* result)
{
   cilk_for (int i=0; i<N; i++)
   {
         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;
   }
}
```

Memory load

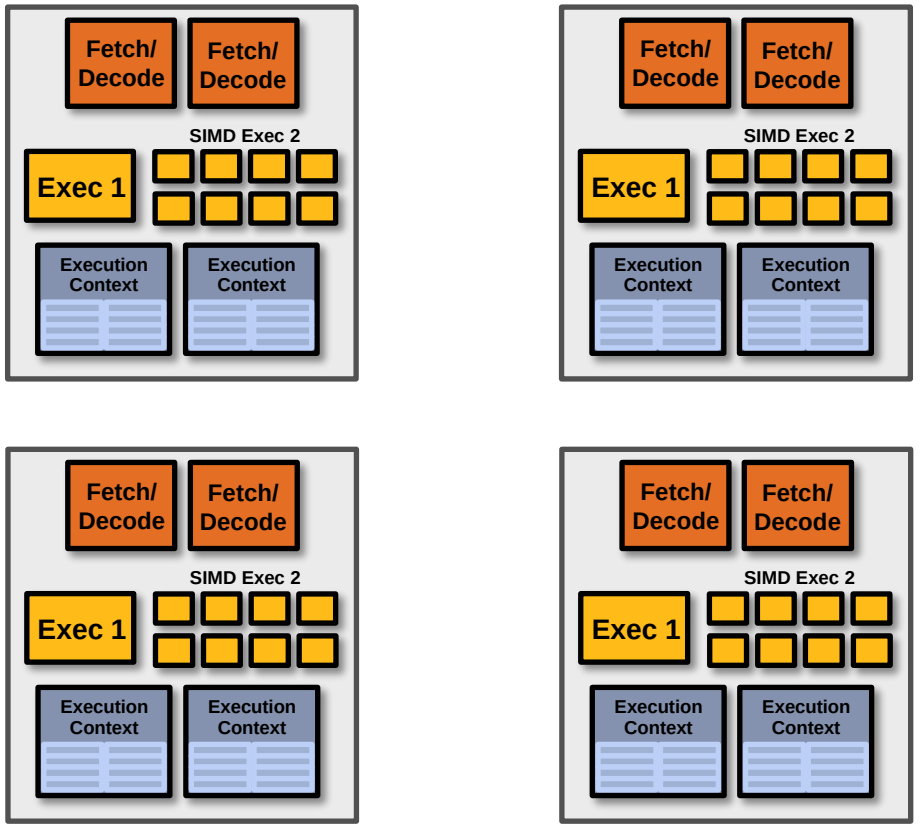
Memory store

**My multi-threaded, SIMD quad-core processor: executes one SIMD instruction per clock from one instruction stream on each core. But can switch to processing the other instruction stream when faced with a stall.**



# Summary: four superscalar, SIMD, multi-threaded cores

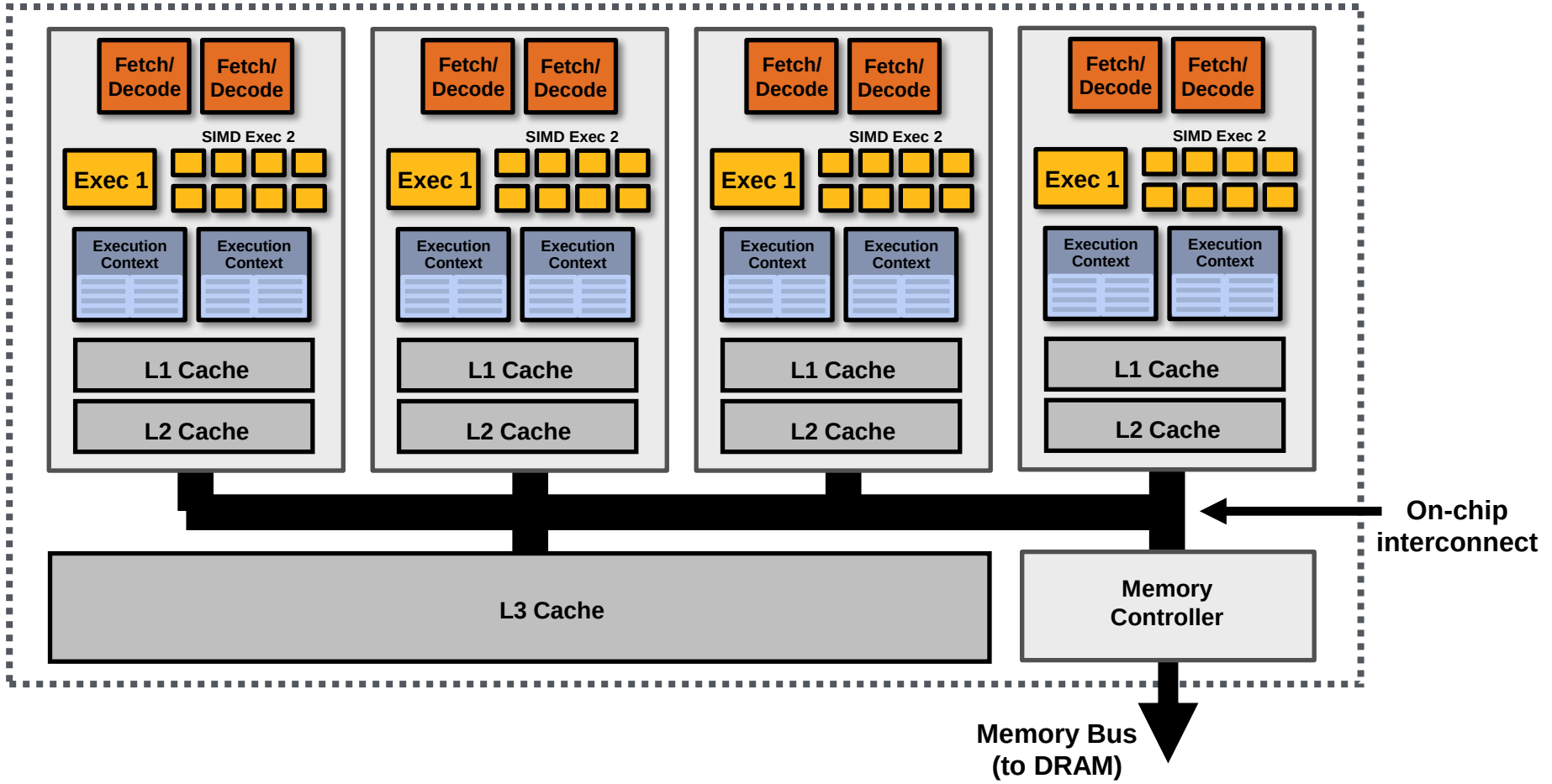
**My multi-threaded, superscalar, SIMD quad-core processor:**
**executes up to two instructions per clock from one instruction stream on each core (in this example: one SIMD instruction + one scalar instruction).**
**Processor can switch to execute the other instruction stream when faced with stall.**



# Connecting it all together

**A simple quad-core processor:**

**Four cores, two-way multi-threading per core (max eight threads active on chip at once), up to two instructions per clock per core (one of those instructions is 8-wide SIMD)**



# Lecture Review

- **In the lectures you have talked about a brief history of the evolution of architecture**

- **Instruction-level parallelism (ILP)**
- **Multiple processing cores**
- **Vector (superscalar, SIMD) processing**
- **Multi-threading (hyper-threading)**
- **Caching**

- **What we cover:**
  - Programming perspective of view

# Next lecture

- **I/O-efficient algorithms: algorithms incur fewer memory accesses**
  - How to model the cost
  - How to design efficient algorithms with fewer I/Os