

L2-A Modern Multi-Core Processor

Forms of parallelism: multi-core, SIMD, and multi-threading

HPC-study/cs-course/cs149 at main · iiinsight/HPC-study

1 review

1.1 computer program 程序

Review from class 1: What is a computer program?

? What is a computer program?

A program is a list of processor instructions

? What is an instruction?

An instruction is a list of commands, the commands modify the state in machine
程序是一系列处理器指令，一条指令是一串命令的列表，这些命令修改机器中的状态

```
int main(int argc, char** argv) {  
    int x = 1;  
  
    for (int i=0; i<10; i++) {  
        x = x + x;  
    }  
  
    printf("%d\n", x);  
  
    return 0;  
}
```



```
_main:  
100000f10: pushq   %rbp  
100000f11: movq %rsp, %rbp  
100000f14: subq $32, %rsp  
100000f18: movl $0, -4(%rbp)  
100000f1f: movl %edi, -8(%rbp)  
100000f22: movq %rsi, -16(%rbp)  
100000f26: movl $1, -20(%rbp)  
100000f2d: movl $0, -24(%rbp)  
100000f34: cmpl $10, -24(%rbp)  
100000f38: jge 23 <_main+0x45>  
100000f3e: movl -20(%rbp), %eax  
100000f41: addl -20(%rbp), %eax  
100000f44: movl %eax, -20(%rbp)  
100000f47: movl -24(%rbp), %eax  
100000f4a: addl $1, %eax  
100000f4d: movl %eax, -24(%rbp)  
100000f50: jmp -33 <_main+0x24>  
100000f55: leaq 58(%rip), %rdi  
100000f5c: movl -20(%rbp), %esi  
100000f5f: movb $0, %al  
100000f61: callq 14  
100000f66: xorl %esi, %esi  
100000f68: movl %eax, -28(%rbp)  
100000f6b: movl %esi, %eax  
100000f6d: addq $32, %rsp  
100000f71: popq %rbp  
100000f72: rets
```

program text →(compile code)→ a list of commands

程序文本 → (编译代码) → 一串命令

1.2 processor处理器

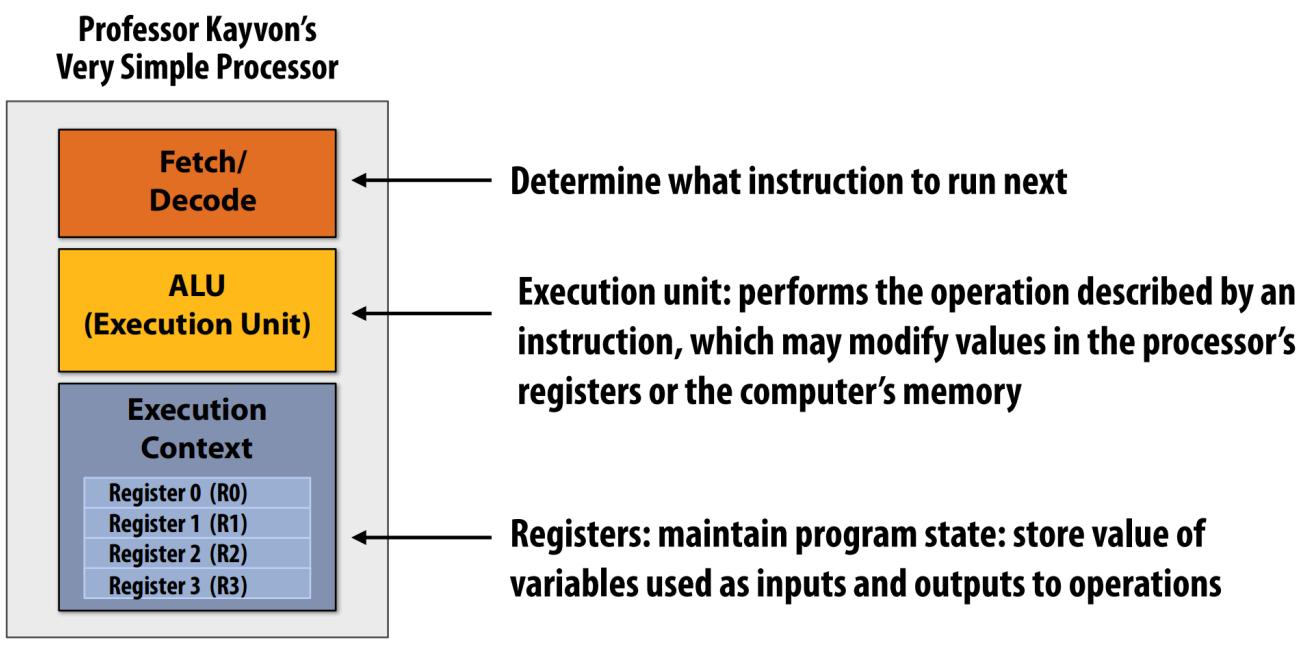
Review from class 1: What does a processor do?



？ What does a processor do？

A processor executes instructions

处理器执行指令

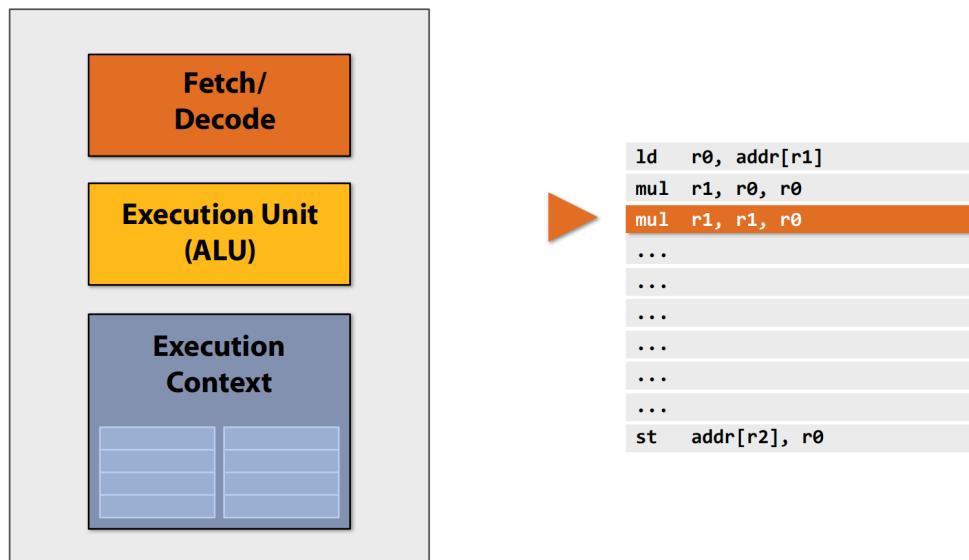


A processor executes instructions 处理器执行指令

- 取指、译码：决定下一次运行哪一个指令
- **ALU (Arithmetic Logic Unit 算术逻辑单元) (执行单元)**：执行指令的操作，可能会修改处理器的寄存器或者电脑的内存值
- 寄存器：维护程序状态，存储输入输出给操作的变量值

Execute program

My very simple processor: executes one instruction per clock



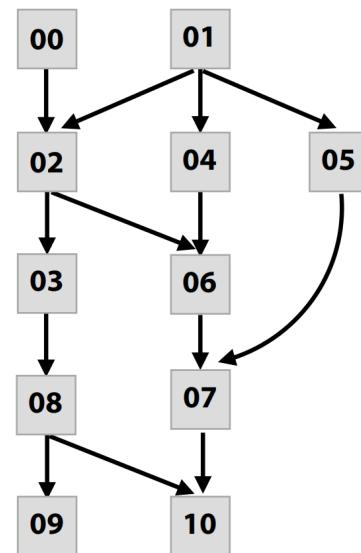
每个时钟周期执行一个指令

A program with instruction level parallelism

Program (sequence of instructions)

PC	Instruction	
00	a = 2	
01	b = 4	
02	tmp2 = a + b // 6	
03	tmp3 = tmp2 + a // 8	
04	tmp4 = b + b // 8	
05	tmp5 = b * b // 16	
06	tmp6 = tmp2 + tmp4 // 14	
07	tmp7 = tmp5 + tmp6 // 30	
08	if (tmp3 > 7)	
09	print tmp3	
10	else	
	print tmp7	

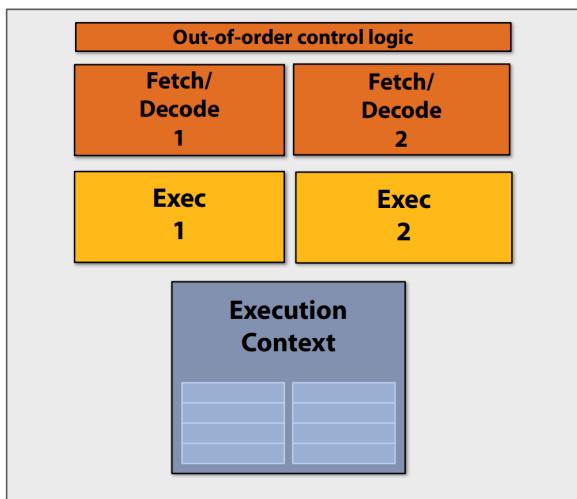
Instruction dependency graph



具有指令级并行 instruction level parallelism (ILP) 的程序

Superscalar processor

This processor can decode and execute up to two instructions per clock



Superscalar execution: processor automatically finds independent instructions in an instruction sequence and can execute them in parallel on multiple execution units.

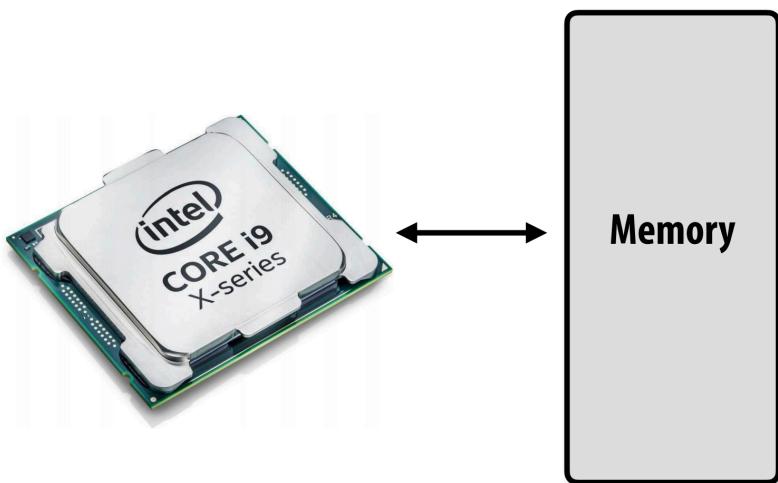
What does it mean for a superscalar processor to “respect program order”?

每个时钟周期中，处理器可以译码和执行至多两个指令

***Superscalar execution: 处理器自动找到在指令序列中的独立的指令，在多个执行单元上并行执行

1.3 memory 内存

Review from class 1:
What is memory?



A program's memory address space

- A computer's memory is organized as an array of bytes
- Each byte is identified by its "address" in memory
(its position in this array)
(We'll assume memory is byte-addressable)

"The byte stored at address 0x8 has the value 32."

"The byte stored at address 0x10 (16) has the value 128."

In the illustration on the right, the program's
memory address space is 32 bytes in size
(so valid addresses range from 0x0 to 0x1F)

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
:	:
0x1F	0

- 内存是一个字节数组
- 每个字节被它在内存中的地址定义
 - 地址是在这个数组中的位置
 - ***内存是 byte-addressable 按字节寻址的

右图中，这个程序的内存地址空间是32 bytes大小，合法地址是从0x0到0x1F

英	中	说明
byte-addressable	按字节寻址 / 字节可寻址	每个字节有唯一地址
word-addressable	按字寻址	每个字 (通常 2/4 字节) 有唯一地址
bit-addressable	按位寻址	每个位都有唯一地址

"Byte-addressable" 意思是 内存中的每一个字节 (8 位) 都有唯一的地址。
也就是说，处理器访问内存时，最小的可寻址单位是一个字节 (byte)。
在现代计算机中，主存通常是 **byte-addressable** 的。

比如，一个 32 位地址总线可以表示 2^{32} 个字节的地址空间，即 4 GB。

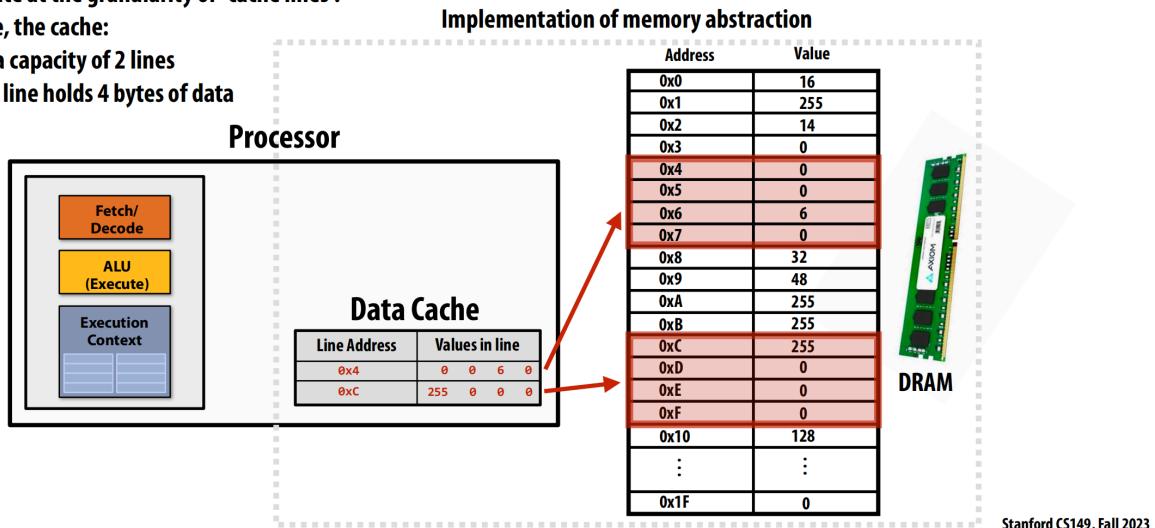
1.4 Cache缓存

What are caches?

- A cache is a hardware implementation detail that does not impact the output of a program, only its performance
- Cache is on-chip storage that maintains a copy of a subset of the values in memory
- If an address is stored “in the cache” the processor can load/store to this address more quickly than if the data resides only in DRAM
- Caches operate at the granularity of “cache lines”.

In the figure, the cache:

- Has a capacity of 2 lines
- Each line holds 4 bytes of data

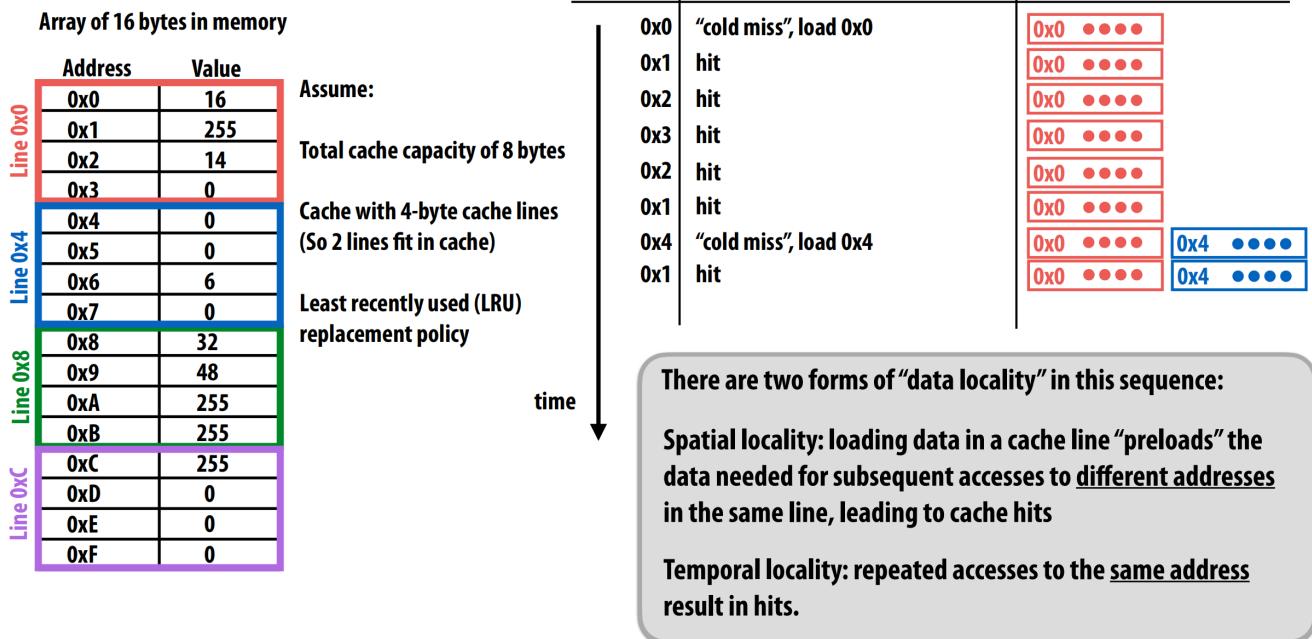


Stanford CS149, Fall 2023

- **cache**: 不影响一段程序输出，只影响性能的硬件实现
 - 是一段片上存储，包含了memory内存中的值的子集的拷贝
 - 如果地址存储在cache中，相比于存在DRAM，处理器可以更快地加载/存储这个地址
 - **cache** 缓存以“缓存行 (cache line)”为粒度进行操作
 - 图中，cache含有两个lines
 - 每一个line有4bytes数据

如果想在cache中访问0x5，需要请求memory，给我从0x4开始的整条cache line的所有信息

Cache example 1



- 假设：
 - 缓存 (cache) 的总容量为 **8 字节**;
 - 每个缓存行 (cache line) 的大小为 **4 字节**, 因此缓存中可以同时容纳 **2 个缓存行**;
 - 采用 **最近最少使用 (LRU, Least Recently Used)** 的替换策略。
- 数据局部性 (data locality)** 主要有两种形式:
 - 空间局部性 (Spatial locality)** : 当加载一个缓存行时, 相当于预先加载了该缓存行中其他地址的数据。因此, 后续访问同一缓存行中不同地址的数据时, 往往会命中缓存 (**cache hit**) 。
 - 时间局部性 (Temporal locality)** : 对同一地址进行多次重复访问时, 由于该数据仍在缓存中, 也会产生缓存命中。

第一次访问时, 会缓存未命中 (**cache miss**), “cold miss”为冷未命中

Cache example 2

Array of 16 bytes in memory

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
Line 0x4	
0x4	0
0x5	0
0x6	6
0x7	0
Line 0x8	
0x8	32
0x9	48
0xA	255
0xB	255
Line 0xC	
0xC	255
0xD	0
0xE	0
0xF	0

Assume:

Total cache capacity of 8 bytes

Cache with 4-byte cache lines
(So 2 lines fit in cache)

Least recently used (LRU) replacement policy

time ↓

Address accessed	Cache action	Cache state (after load is complete)
0x0	"cold miss", load 0x0	0x0 ••••
0x1	hit	0x0 ••••
0x2	hit	0x0 ••••
0x3	hit	0x0 ••••
0x4	"cold miss", load 0x4	0x0 •••• 0x4 ••••
0x5	hit	0x0 •••• 0x4 ••••
0x6	hit	0x0 •••• 0x4 ••••
0x7	hit	0x0 •••• 0x4 ••••
0x8	"cold miss", load 0x8 (evict 0x0)	0x8 •••• 0x4 ••••
0x9	hit	0x8 •••• 0x4 ••••
0xA	hit	0x8 •••• 0x4 ••••
0xB	hit	0x8 •••• 0x4 ••••
0xC	"cold miss", load 0xC (evict 0x4)	0x8 •••• 0xC ••••
0xD	hit	0x8 •••• 0xC ••••
0xE	hit	0x8 •••• 0xC ••••
0xF	hit	0x8 •••• 0xC ••••
0x0	"capacity miss", load 0x0 (evict 0x8)	0x0 •••• 0xC ••••

缓存未命中 (cache miss) 分为多种类型

- ***“cold miss”冷未命中：第一次访问地址
- ***“capacity miss”容量未命中：仅仅由于cache容量有限而发生的未命中
- ***“conflict miss”冲突未命中：由于cache组织方式而发生的未命中

Caches reduce length of stalls (reduce memory access latency)

- Processors run efficiently when they access data that is resident in caches
- Caches reduce memory access latency when processors accesses data that they have recently accessed! *

* Caches also provide high bandwidth data transfer

Stanford CS149, Fall 2023

***缓存减少停顿时间 (降低内存访问延迟)

- 当处理器访问缓存中已驻留的数据时，其运行效率更高；
- 当处理器访问最近使用过的数据时，缓存可以显著降低内存访问延迟；

- 此外，缓存还提供高带宽的数据传输能力。

stalls (停顿)：指 CPU 在等待数据从主存加载时的空转周期。

memory access latency (内存访问延迟)：指从请求数据到数据可用之间的时间。

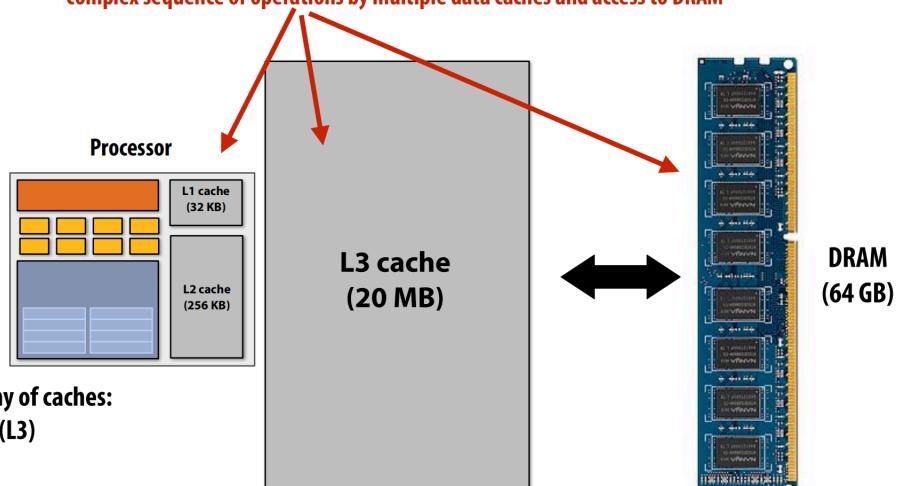
cache 的作用：

通过将最近访问的数据保留在更接近 CPU 的缓存中，减少数据取用时间，从而减少处理器空闲等待。

同时，缓存的高带宽也使得数据读写速度显著提升。

The implementation of the linear memory address space abstraction on a modern computer is complex

The instruction “load the value stored at address X into register R0” might involve a complex sequence of operations by multiple data caches and access to DRAM



Stanford CS149, Fall 2023

现代计算机中“线性内存地址空间抽象”的实现非常复杂

- 一条看似简单的指令：“将地址 X 处的值加载到寄存器 R0 中”，
- 实际上可能涉及多个缓存层次 (data caches) 的复杂操作序列，甚至访问主存 (DRAM) 。

常见的缓存层级结构：

第一级缓存 (L1)、第二级缓存 (L2)、第三级缓存 (L3)

***越靠近处理器的缓存容量越小 → 延迟越低

越远离处理器的缓存容量越大 → 延迟越高

现代 CPU 的“内存访问”并不是直接访问主内存 (DRAM)，而是依次经过多级缓存：

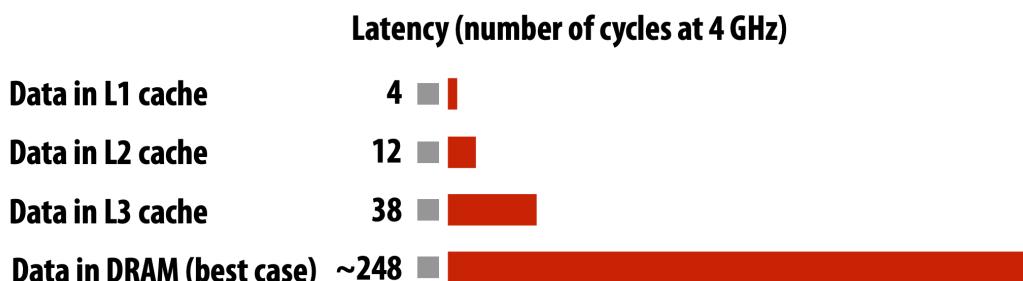
层级	典型容量	位置	访问延迟	特点
L1 Cache	~32 KB	最靠近 CPU 核心	~1ns	超高速, 存放最近使用的数据
L2 Cache	~256 KB	稍远	~3-5ns	比 L1 大但稍慢
L3 Cache	~10-30 MB	所有核心共享	~10-20ns	容量更大、延迟更高
DRAM	数 GB	远离 CPU	~100ns	访问最慢但容量最大

因此, 一个简单的“load”指令, 其实需要 CPU 从 L1 → L2 → L3 → DRAM 逐级查找数据。

这也是为什么 缓存层次结构 (**cache hierarchy**) 对性能如此关键。

Data access times

(Kaby Lake CPU)



2. 并行计算的三个主要思想

Today

- Today we're talking computer architecture... from a software engineer's perspective

- Key concepts about how modern parallel processors achieve high throughput
 - Two concern parallel execution (multi-core, SIMD parallel execution)
 - One addresses the challenges of memory latency (multi-threading)

- Understanding these basics will help you
 - Understand and optimize the performance of your parallel programs
 - Gain intuition about what workloads might benefit from fast parallel machines

从软件工程师的角度来谈谈计算机体系结构

关于现代并行处理器如何实现高吞吐量的关键概念

- 两个关注点是并行执行(多核 multi-core、 SIMD 并行执行)
- 一种方法解决了内存延迟的挑战(多线程 multi-threading)

了解这些基础知识将对你有所帮助

- 理解并优化并行程序的性能
- 深入了解哪些工作可能从快速并行机器中受益

Today's example program

```
void sinx(int N, int terms, float* x, float* y)
{
    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;

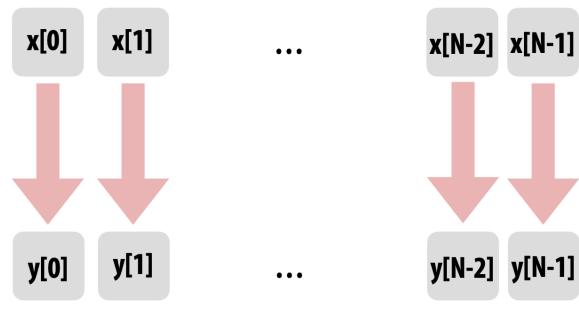
        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

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

Compute $\sin(x)$ using Taylor expansion:

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$$

for each element of an array of N floating-point numbers



使用泰勒展开式计算 $\sin(x)$

对于浮点数数组的每个元素，输入 x_i ，输出 y_i

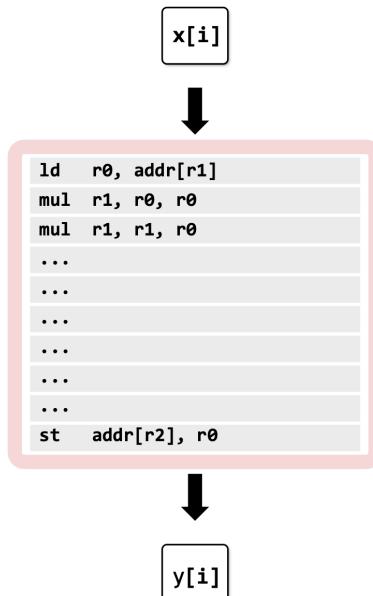
Compile program

```
void sinx(int N, int terms, float* x, float* y)
{
    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;
        }

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

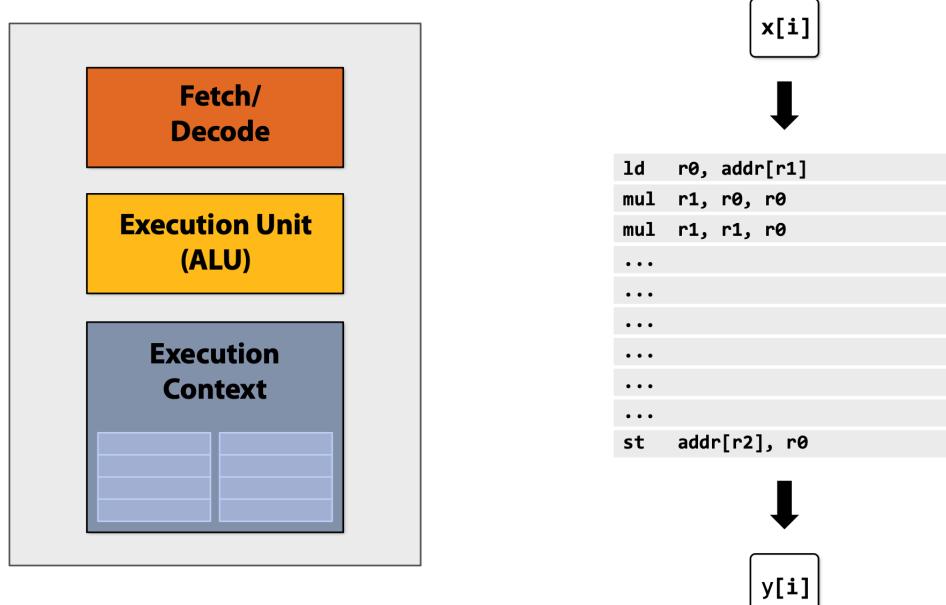
Compiled instruction stream
(scalar instructions)



编译后的指令流 instruction stream
(标量指令 scalar instructions)

Execute program

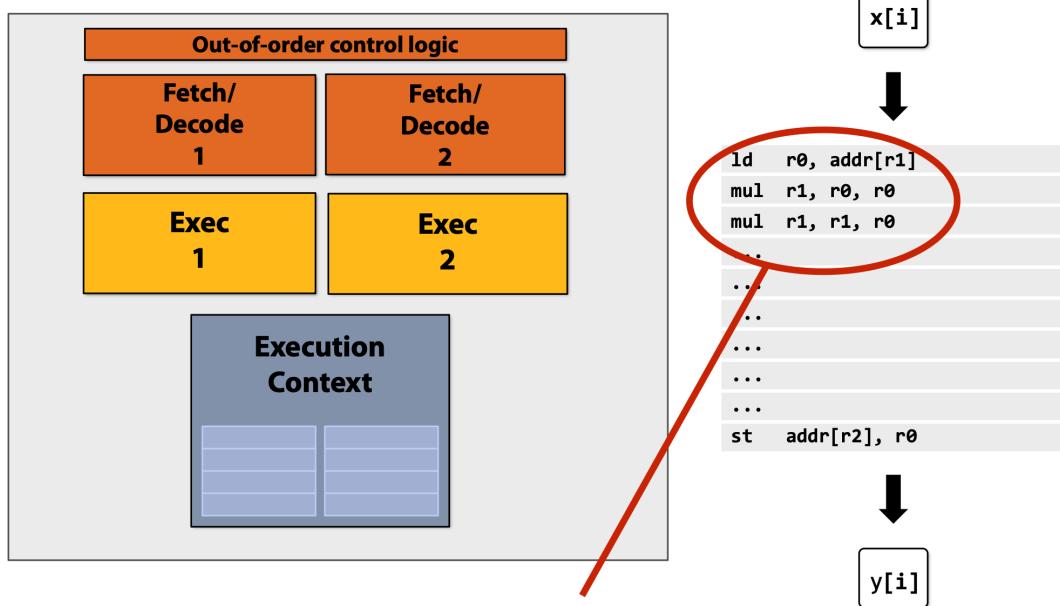
My very simple processor: executes one instruction per clock



非常简单的处理器：每个时钟周期执行一条指令

Superscalar processor

The processor shown here can decode and execute two instructions per clock
(if independent instructions exist in an instruction stream)



Note: No ILP exists in this region of the program

超标量处理器 superscalar processor

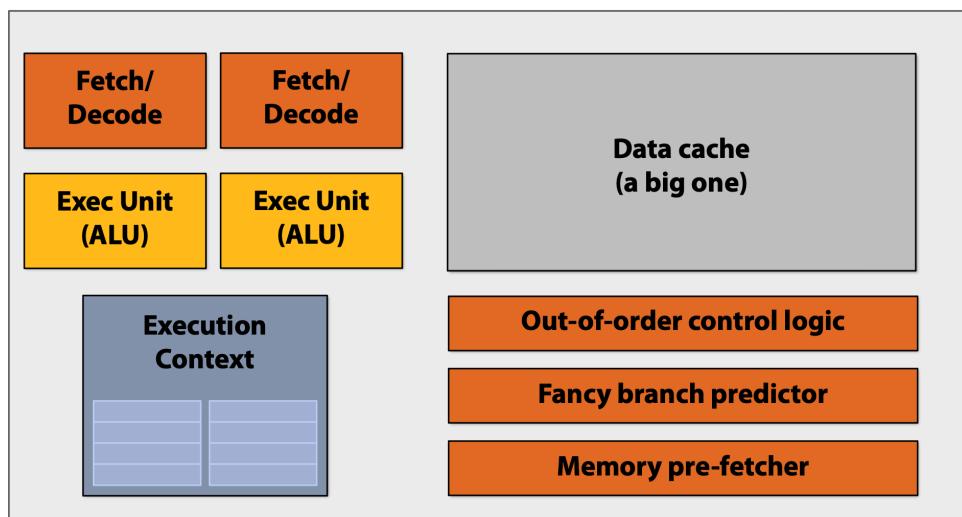
这里展示的处理器可以在每个时钟周期内译码并执行两条指令(如果是指令流中存在独立指令)

乱序控制逻辑

注意：程序画框区域不存在指令级并行 instruction level parallelism (ILP) , 是相互依赖的

Pre multi-core era processor

Majority of chip transistors used to perform operations that help make a single instruction stream run fast



More transistors = larger cache, smarter out-of-order logic, smarter branch predictor, etc.

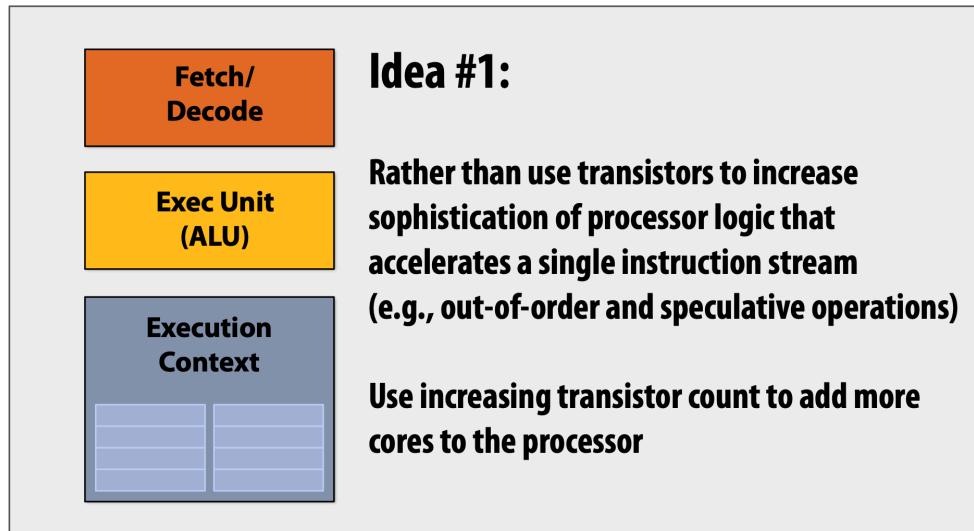
多核时代之前的处理器

大多数芯片晶体管用于执行操作,这些操作帮助使单个指令流快速运行

更多晶体管=更大的缓存、更智能的乱序逻辑、更智能的分支预测器等。

2.1 Idea1: multi-core (多核)

Multi-core era processor

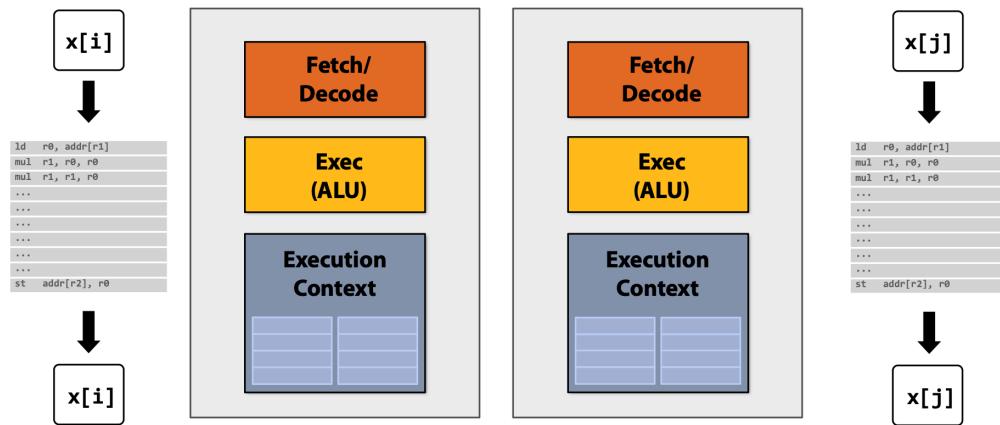


多核时代处理器 multi-core era processor

与其使用晶体管来增加处理器逻辑的复杂性以加速单一指令流（例如，乱序执行和推测性操作）

不如通过增加晶体管数量来为处理器增加更多核心core。

Two cores: compute two elements in parallel



Simpler cores: each core may be slower at running a single instruction stream than our original “fancy” core (e.g., 25% slower)

But there are now two cores: $2 \times 0.75 = 1.5$ (potential for speedup!)

两个核心core：并行计算两个元素

更简单的核心：每个核心在运行单一指令流时可能会比我们原先的“高级”核心慢一些（例如，慢25%）。

但现在有两个核心： $2 \times 0.75 = 1.5$ $2 \times 0.75 = 1.5$ (有加速的潜力！)

But our program expresses no parallelism

```
void sinx(int N, int terms, float* x, float* y)
{
    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;
        }

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

This C program will compile to an instruction stream that runs as one thread on one processor core.

If each of the simpler processor cores was 25% slower than the original single complicated one, our program now runs 25% slower than before.



但我们的程序没有表达并行性

这段 C 程序将被编译成一个指令流，它在一个处理器核心上以一个线程运行。

如果每个简单的处理器核心比原来的复杂核心慢 25%，那么我们的程序现在运行的速度比之前慢 25%。

Example: expressing parallelism using C++ threads

```
typedef struct {
    int N;
    int terms;
    float* x;
    float* y;
} my_args;

void my_thread_func(my_args* args)
{
    sinx(args->N, args->terms, args->x, args->y); // do work
}

void parallel_sinx(int N, int terms, float* x, float* y)
{
    std::thread my_thread;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.y = y;

    my_thread = std::thread(my_thread_func, &args); // launch thread
    sinx(N - args.N, terms, x + args.N, y + args.N); // do work on main thread
    my_thread.join(); // wait for thread to complete
}
```

```
void sinx(int N, int terms, float* x, float* y)
{
    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;
        }

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

Stanford CS149, F

使用 C++ 线程表达并行性

1 定义结构体参数

```
typedef struct {
    int N;
    int terms;
    float* x;
    float* y;
} my_args;
```

C++

定义一个结构体 `my_args`，用于封装线程函数需要的参数：

- `N`: 计算的数据量 (例如数组长度)
- `terms`: 计算展开的项数
- `x`、`y`: 输入输出数组指针

2 线程函数

```
void my_thread_func(my_args* args)
{
    sinx(args->N, args->terms, args->x, args->y); // do work
}
```

C++

定义线程要执行的函数 `my_thread_func`，内部调用 `sinx` 函数进行计算。

3 并行函数

C++

```
void parallel_sinx(int N, int terms, float* x, float* y)
{
    std::thread my_thread;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.y = y;

    my_thread = std::thread(my_thread_func, &args); // 启动线程
    sinx(N - args.N, terms, x + args.N, y + args.N); // 主线程做另一半
    my_thread.join(); // 等待线程结束
}
```

定义一个并行版本的 `sinx`：

1. 把任务分成两半：
 - 前半部分交给子线程；
 - 后半部分由主线程自己完成。
2. 启动一个子线程执行 `my_thread_func`；
3. 主线程同时计算另一部分；
4. 最后用 `join()` 等待子线程结束，保证所有结果都计算完。

4 sinx 函数本体

```

void sinx(int N, int terms, float* x, float* y)
{
    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;
        }

        y[i] = value;
    }
}

```

sinx 实现对 **x** 数组中每个元素用泰勒展开近似计算 $\sin(x)$ 。

即: $[\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots]$

每个循环计算一个输入的 \sin 值并存入 **y[i]**。

#为什么要创建线程?

创建线程的目的是并行计算。

原始的 **sinx()** 是串行执行的 (一次只算一个元素)。

parallel_sinx() 把任务一分为二, 让:

- 一个线程 (子线程) 计算前半部分;
- 主线程计算后半部分;

这样两个 CPU 核心可以同时工作, 理论上计算时间减半 (理想情况下)。

模块	作用
sinx()	串行计算 $\sin(x)$ 的近似值
my_thread_func()	封装线程的工作内容 (调用 sinx)
parallel_sinx()	创建线程并并行计算两部分数据
my_thread.join()	等待子线程结束, 确保所有结果已写入

Data-parallel expression

(in Kayvon's fictitious programming language with a "forall" construct)

```
void sinx(int N, int terms, float* x, float* y)
{
    // declares that loop iterations are independent
    forall (int i from 0 to N)
    {
        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;
        }

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

In this code, loop iterations are declared by the programmer to be independent (see the 'forall')

With this information, you could imagine how a compiler might automatically generate threaded code for you.

数据并行表达式 (在 Kayvon 虚构的编程语言中, 使用 `forall` 结构)

在这段代码中, 程序员通过 `forall` 声明循环中每次 `i` 的迭代之间互不依赖, 因此可以并行执行。

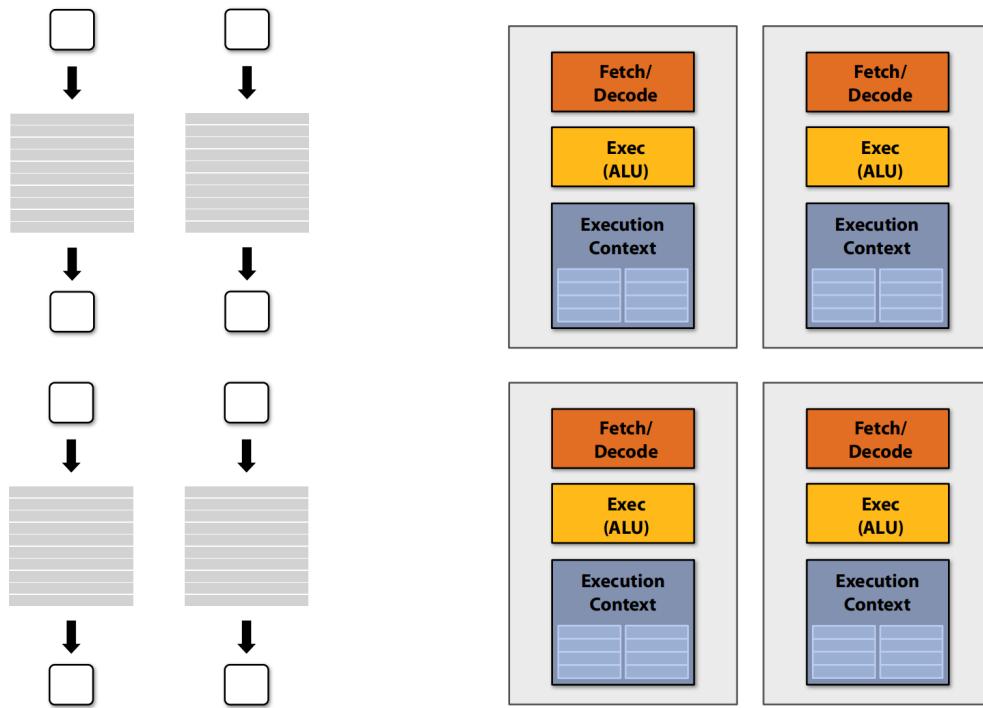
```
// 声明循环的各次迭代是相互独立的
forall (int i from 0 to N)
```

有了这个信息, 编译器就可以自动为你生成多线程代码 (即自动并行化执行)。

代码逻辑:

- 外层循环: `forall (int i from 0 to N)`
→ 表示要对每个输入 `x[i]` 分别计算 `sin(x[i])`。
每个 `i` 的计算彼此独立, 因此可以并行。
- 内层循环: `for (int j=1; j<=terms; j++)`
→ 对单个 `x[i]` 进行多项式展开计算。
- 最终结果: `y[i] = value;`
→ 把计算结果存入 `y` 数组。

Four cores: compute four elements in parallel



四核：并行计算四个元素

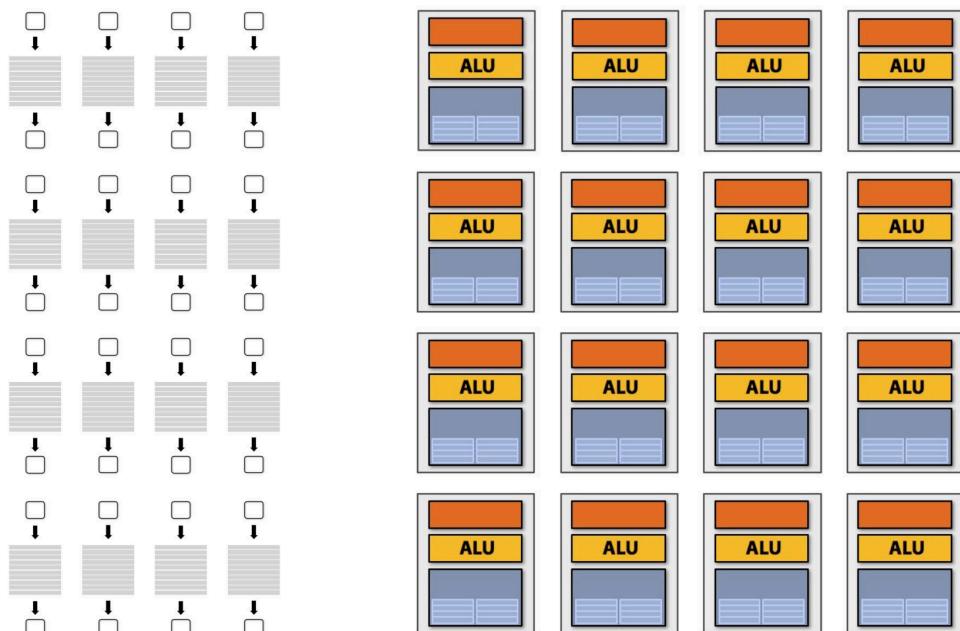
四个独立的处理核心（cores）。每个核心都有：

- 取指/译码（Fetch/Decode）：从内存中取出指令并解析。
- 执行单元（Exec/ALU）：算术逻辑单元，用来执行计算。
- 执行上下文（Execution Context）：存放寄存器、变量、中间结果等。
- 四个核心各自独立地处理自己的输入数据（每个小方框代表输入/输出，灰色条代表数据块）。
- 每个核心可以同时（并行）执行一部分任务，因此一次可以计算四个数据元素。

多核 CPU 的基本概念：

- 每个核心都能独立执行一条指令流。
- 当任务可以被分解为多个独立部分时（例如处理数组中的每个元素），四个核心就能同时执行四个子任务。
- 理论上，性能可以提升至单核的 4 倍（取决于任务是否完全可并行）。

Sixteen cores: compute sixteen elements in parallel



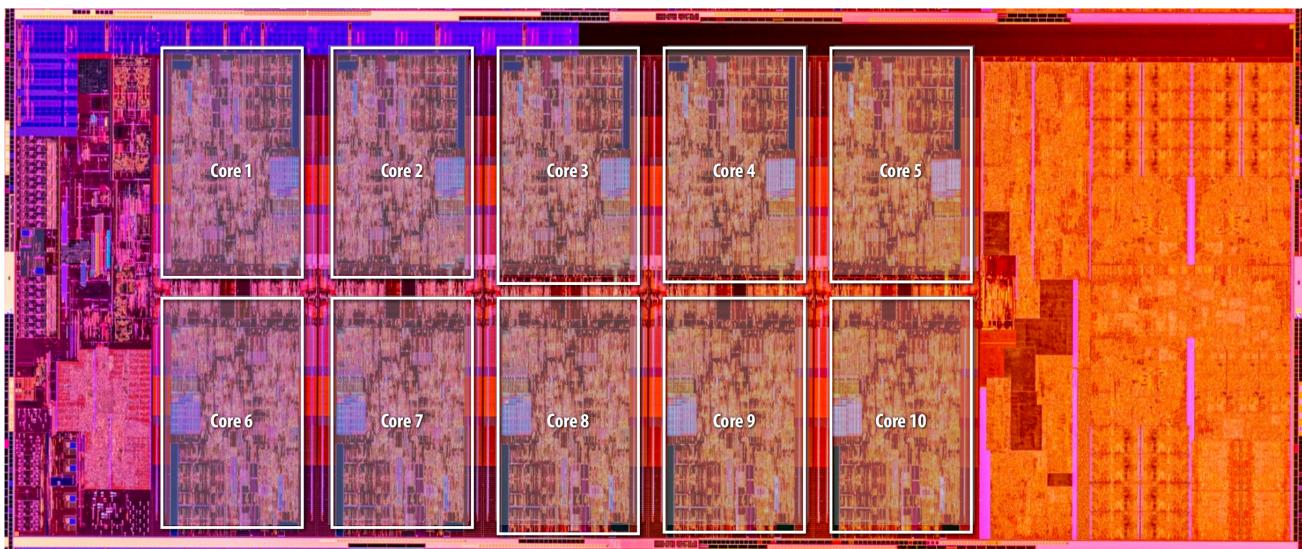
Sixteen cores, sixteen simultaneous instruction streams

十六核：并行计算十六个元素

十六个核心，十六条同时执行的指令流

Example: multi-core CPU

Intel “Comet Lake” 10th Generation Core i9 10-core CPU (2020)



多核 CPU (Intel “Comet Lake” i9-10900, 2020)

- 10 个矩形区域 (Core 1~Core 10) , 每个区域都是一个 独立的 **CPU** 核心 (**core**) 。
- 每个核心都能独立执行指令、拥有自己的缓存、寄存器和执行单元。
- 这是一颗典型的高性能 通用处理器 (**CPU**) , 用于执行复杂的逻辑运算、系统管理任务、以及串行计算。

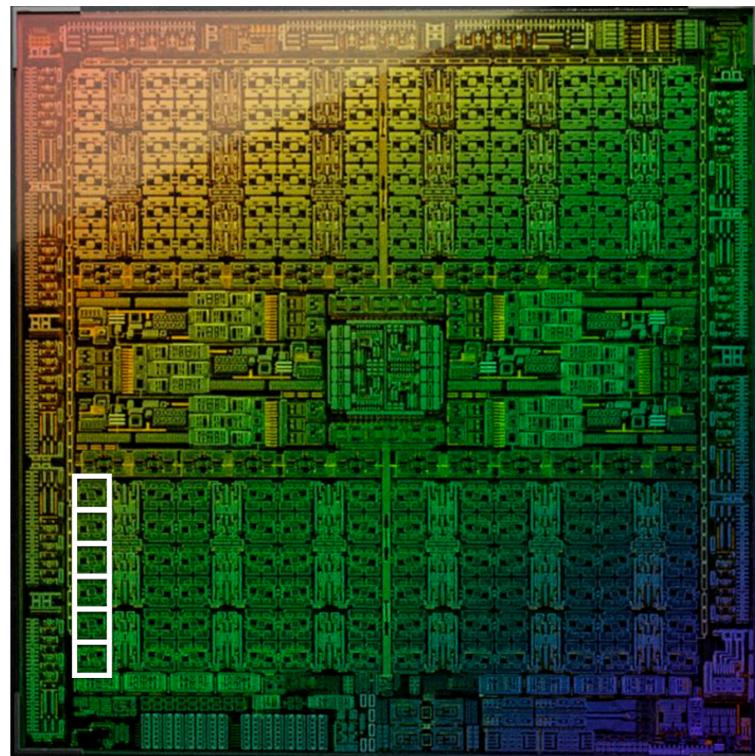
特点：

- 核心数量较少（10个），但每个核心性能强。
- 每个核心结构复杂，具备分支预测、乱序执行、多级缓存。
- 适合处理顺序逻辑、条件判断多、延迟敏感的任务（如操作系统调度、应用程序主线程等）。

Multi-core GPU

GeForce RTX 4090 (2022)

144 processing blocks (called SMs)

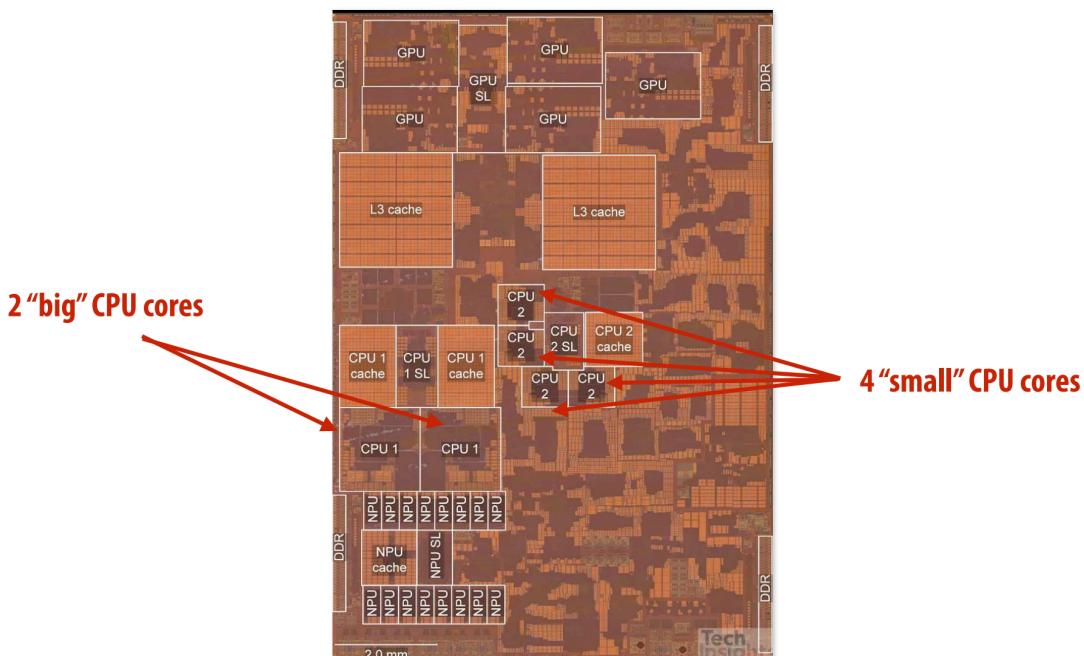


多核 GPU (NVIDIA GeForce RTX 4090, 2022)

- 该 GPU 拥有 **144** 个处理单元 (**Streaming Multiprocessors, SM**)。
 - 每个 SM 内部再包含数十个小的运算核心 (CUDA cores)，专门处理并行数据。
- 特点：
- 极高的并行度：数千个小计算单元可以同时执行同一种运算。
 - 每个核心相对简单，但总数量巨大，专为 并行计算 而设计。
 - 适合处理 图形渲染、AI推理、科学计算 等需要同时处理大量相似数据的任务。
 - 与 CPU 不同，GPU 不擅长逻辑复杂或分支多的任务。

Apple A15 Bionic

Two “big cores” + four “small” cores



苹果手机中的 SoC (System on Chip) , 即“系统级芯片”。

- 采用 异构多核 (**heterogeneous multi-core**) 架构：
 - 2 个“大核”：高性能核心，用于运行高负载任务。
 - 4 个“小核”：高能效核心，用于轻量任务，节省功耗。
- 同时集成了 **GPU**、**NPU** (神经网络处理单元) 和缓存区。
特点：
- 性能与功耗兼顾：通过智能调度，“大核”处理高性能任务，“小核”处理后台任务。
- 是一种 移动设备常用架构 (**big.LITTLE** 设计) 。
- 适合手机和嵌入式场景，能在有限功耗下保持高性能。

芯片类型	典型用途	核心数量	核心类型	特点
CPU	通用计算、系统调度	少 (4-16)	复杂核心	高灵活性，低并行度
GPU	图形渲染、AI计算	多 (上千)	简单核心	极高并行度，低控制能力
SoC (A15)	移动设备、混合任务	中等 (6 CPU核 + GPU + NPU)	异构核心	性能与能效平衡

2.1 Idea2: SIMD (单指令多数据)

Data-parallel expression (in Kayvon's fictitious programming language with a "forall" construct)

```
void sinx(int N, int terms, float* x, float* result)
{
    // declares that loop iterations are independent
    forall (int i from 0 to N)
    {
        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;
    }
}
```

Another interesting property of this code:

Parallelism is across iterations of the loop.

All the iterations of the loop carry out the exact same sequence of instructions (defined by the loop body), but on different input data given by $x[i]$

(the loop body computes $\sin(x[i])$)

数据并行：

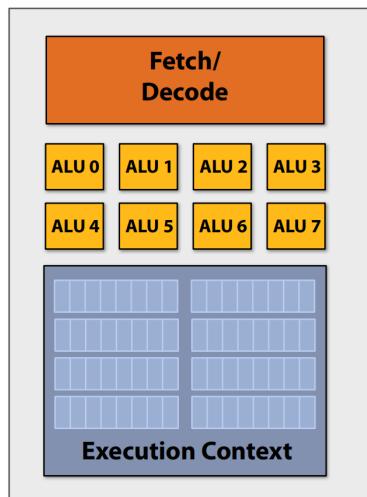
- 每次循环（迭代）执行相同的计算逻辑；
- 不同迭代之间的数据 ($x[i]$) 彼此独立；
- 因此可以并行处理每个元素。
- 数据并行 = 同样的计算过程 + 不同的数据输入 + 可同时执行。

GPU 并行计算 或 SIMD (单指令多数据) 的核心思想：

***一条程序控制流（同样的代码）同时处理多个不同的数据输入。

1、增加执行单元 (ALUs)

Add execution units (ALUs) to increase compute capability



Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Single instruction, multiple data

Same instruction broadcast to all ALUs

This operation is executed in parallel on all ALUs

通过增加执行单元 (ALUs) 提升计算能力

主要思想 (Idea # 2) :

将指令流的管理成本/复杂度分摊到多个 ALU 上。 → 除了ALU其他部分执行一次，增加 ALUs 相比于 multi cores 节省成本

这意味着多个 ALU 共享同一条指令流，从而节省控制开销。

***SIMD** (单指令多数据) 处理:

- 一条指令 (Single Instruction)
- 同时处理多组数据 (Multiple Data)
- 相同的指令被广播到所有 ALU
- 每个 ALU 在不同数据上并行执行相同操作

CPU 内部的简化结构:

- **Fetch/Decode** (取指/译码) : 统一获取并解析指令。
- **ALU (Arithmetic Logic Unit)** : 算术逻辑单元，是执行实际运算的核心。
- **Execution Context**: 控制和寄存器状态，用于协调 ALU 的并行执行。

CPU 通过添加多个 ALU，在同一时间执行同一指令的多个数据版本。

例如：8 个 ALU 同时对 8 个数执行加法或乘法。

Recall our original scalar program

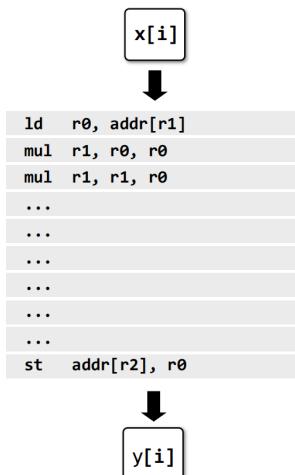
```
void sinx(int N, int terms, float* x, float* y)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        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;
        }

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

Original compiled program:

Processes one array element using scalar instructions
on scalar registers (e.g., 32-bit floats)



Stanford CS149, Fa

原始编译器生成的程序是标量程序，
每次只能处理一个数组元素。
使用标量寄存器（如 32 位浮点寄存器）执行运算。

即：

- 每次循环处理一个 **x[i]**；
- 加载数据 → 运算 → 存回；
- 无并行化。

2、SIMD (AVX 指令) 向量化

Vector program (using AVX intrinsics)

```
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* y)
{
    float three_fact = 6; // 3!
    for (int i=0; i<N; i+=8)
    {
        __m256 origx = _mm256_load_ps(&x[i]);
        __m256 value = origx;
        __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
        __m256 denom = _mm256_broadcast_ss(&three_fact);
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            // value += sign * numer / denom
            __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1ps(sign), numer), denom);
            value = _mm256_add_ps(value, tmp);

            numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
            denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
            sign *= -1;
        }
        _mm256_store_ps(&y[i], value);
    }
}
```

Intrinsic datatypes and functions available to C programmers

Intrinsic functions operate on vectors of eight 32-bit values (e.g., vector of 8 floats)

Stanford CS149, Fall 2014

AVX 提供了特殊的数据类型与函数，
允许 C 程序员操作向量数据 (**vector datatypes**)。
这些函数可在一条指令中处理**8 个 32 位浮点数** (256 位寄存器)。

```

#include <immintrin.h>

void sinx(int N, int terms, float* x, float* y) {
    float three_fact = 6; // 3!
    for (int i = 0; i < N; i += 8) { // 一次处理 8 个元素
        __m256 origx = _mm256_load_ps(&x[i]); // 加载8个float
        __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx)); // x^3
        __m256 denom = _mm256_broadcast_ss(&three_fact);
        int sign = -1;

        for (int j = 1; j <= terms; j++) {
            __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1_ps(sign), numer), denom);
            value = _mm256_add_ps(value, tmp);

            numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
            denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2)*
(2*j+3)));
            sign *= -1;
        }

        _mm256_store_ps(&y[i], value); // 存储8个结果
    }
}

```

Vector program (using AVX intrinsics)

```

#include <immintrin.h>

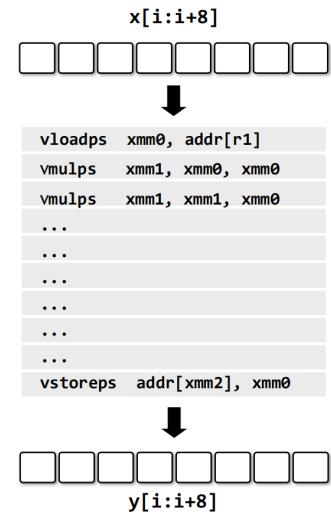
void sinx(int N, int terms, float* x, float* y)
{
    float three_fact = 6; // 3!
    for (int i=0; i<N; i+=8)
    {
        __m256 origx = _mm256_load_ps(&x[i]);
        __m256 value = origx;
        __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
        __m256 denom = _mm256_broadcast_ss(&three_fact);
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            // value += sign * numer / denom
            __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1_ps(sign), numer), denom);
            value = _mm256_add_ps(value, tmp);

            numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
            denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
            sign *= -1;
        }

        _mm256_store_ps(&y[i], value);
    }
}

```



Compiled program:

Processes eight array elements simultaneously using vector instructions on 256-bit vector registers

Stanford CS149,

执行流程：

- 输入： $x[i : i+8]$ ——一次取出8个连续元素

- 寄存器：使用 256 位向量寄存器 (`ymm`)
- 执行：一条指令同时在这 8 个数上执行加、乘、除等操作
- 输出：`y[i : i+8]` ——一次存入8个结果

在一条向量指令中处理**8** 个 **32** 位浮点数 (256 位向量寄存器)。

💡 标量&向量程序：

类型	执行方式	每次操作的数据量	示例
标量程序 (Scalar)	一次处理一个元素	$1 \times 32\text{-bit}$	传统 CPU 循环
向量程序 (SIMD/AVX)	一次处理 8 个元素	$8 \times 32\text{-bit}$	256-bit AVX 寄存器

💡 总结：

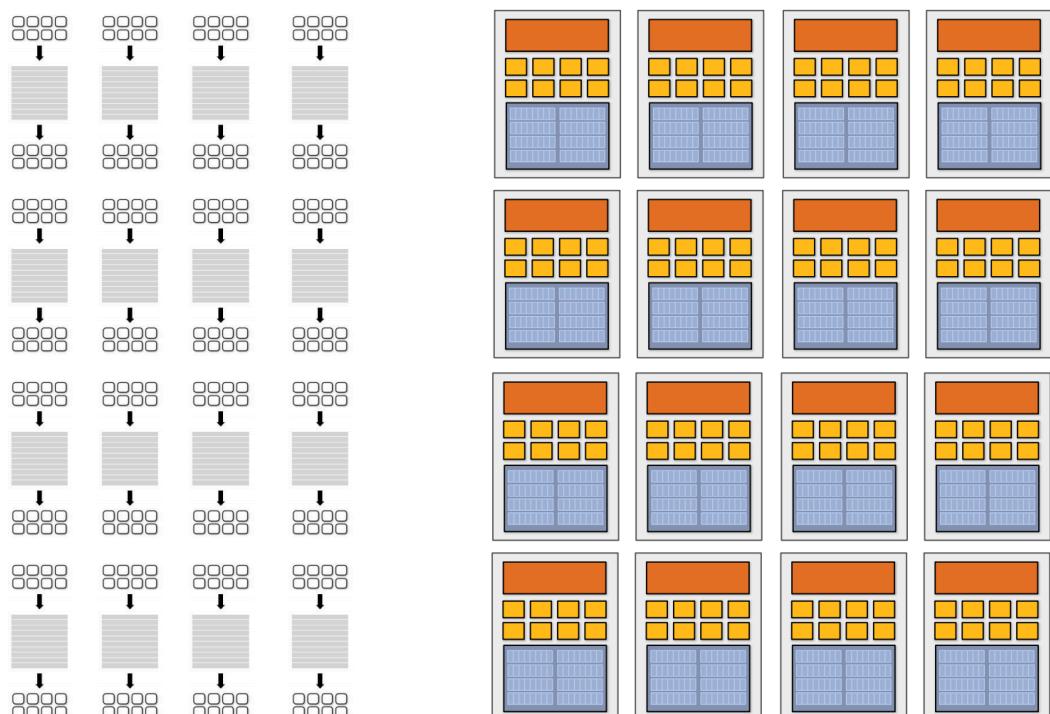
标量程序：一次计算一个元素。

向量程序 (**SIMD**)：一次计算多个元素。

通过 增加 **ALU** 数量 + 广播同一条指令，CPU 实现了高效的数据并行计算，这正是现代向量化编程（如 AVX、NEON、CUDA）的基础。

3、多核 multi cores + SIMD 并行

16 SIMD cores: 128 elements in parallel



16 cores, 128 ALUs, 16 simultaneous instruction streams

16个 SIMD 核心：128 个元素并行执行

16 个核心，128 个 ALU（算术逻辑单元），

16 条同时执行的指令流（instruction streams）（线程）

- 16个核心（core）。➡ 多个核心之间同时执行不同任务（多线程或多指令流）（16个线程）。
- 每个核心内部有8个 ALU（SIMD 单元）。➡ 每个核心内部的 SIMD 单元在一条指令下处理多组数据（8-wide vectors 8宽向量）。
- 因此系统可以同时执行 **16 × 8 = 128** 个数据操作。➡ 在16核芯片上速度快了128倍（与最初的1core 1ALU相比）

即“**多核 + 向量化 (SIMD)**”并行结构：

*核之间并行（多指令流）+核内并行（单指令多数据）

Data-parallel expression

(in Kayvon's fictitious programming language with a "forall" construct)

```
void sinx(int N, int terms, float* x, float* result)
{
    // declares that loop iterations are independent
    forall (int i from 0 to N)
    {
        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;
    }
}
```

The program's use of "forall" declares to the compiler that loop iterations are independent, and that same loop body will be executed on a large number of data elements.

This abstraction can facilitate automatic generation of both multi-core parallel code, and vector instructions to make use of SIMD processing capabilities within a core.

程序使用 **forall** 告诉编译器：

循环迭代之间相互独立，

并且相同的循环体会在大量数据元素上重复执行。

这种抽象可以让编译器自动生成：

➡ 多核并行代码（每个核执行不同的迭代区间）

➡ SIMD 向量指令（每个核内部利用多 ALU 同步执行）

这段代码的逻辑使用了“forall”语句来显式声明循环是数据独立的，这意味着编译器可以：

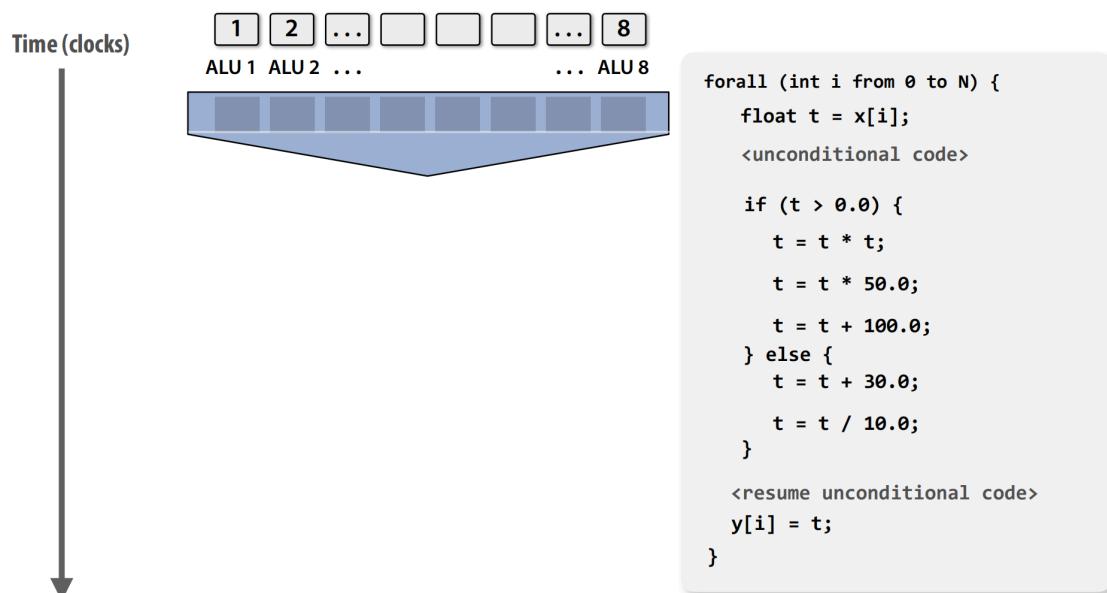
- 将不同的 **i** 值分配给不同的核心 (**Core**) → 多核并行;
- 在单个核心中进一步用 **SIMD** 指令并行处理多个 **i** 值 → 核内向量化;
- 最终实现多层次并行 (**hierarchical parallelism**)。

💡 从单核到多核 + SIMD:

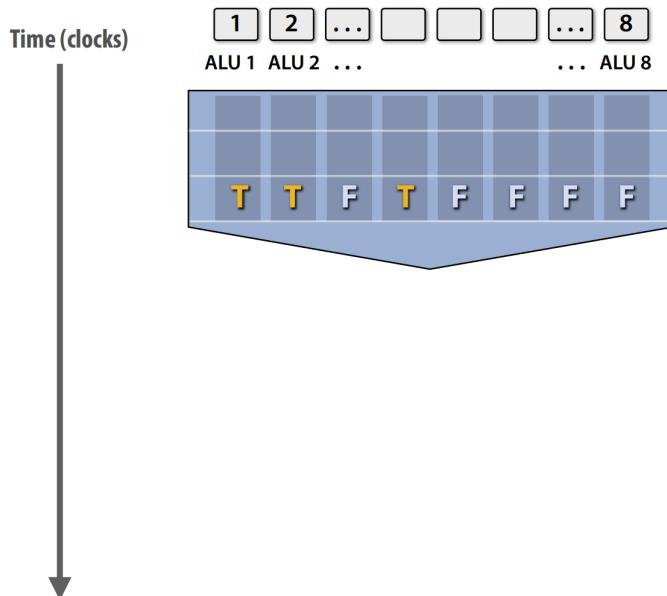
级别	并行类型	并行单位	示例
单核标量	无并行	每次执行 1 个数据	原始 <code>for</code> 循环
SIMD 核内并行	数据并行 (单指令多数据)	每次执行 8 个数据	使用 AVX 向量指令
多核并行	任务并行 (多线程)	每个核心处理一段数据	OpenMP / 多线程
多核 + SIMD	混合并行	16 核 × 8 元素 = 128 元素并行	GPU / HPC 芯片架构

4、条件执行 (Conditional Execution)

What about conditional execution?



What about conditional execution?

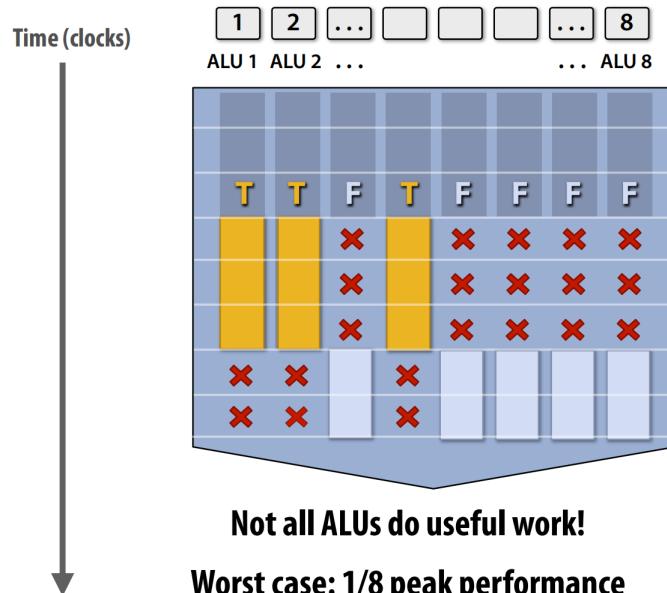


```
forall (int i from 0 to N) {  
    float t = x[i];  
    <unconditional code>  
  
    if (t > 0.0) {  
        t = t * t;  
        t = t * 50.0;  
        t = t + 100.0;  
    } else {  
        t = t + 30.0;  
        t = t / 10.0;  
    }  
  
<resume unconditional code>  
    y[i] = t;  
}
```

SIMD 处理器一次能并行执行多个相同的操作（图中 8 个 ALU：ALU1 ~ ALU8）。但当出现条件语句时，例如 `if (t > 0)`，每个 ALU 的输入数据不同，可能导致有的执行 if 分支、有的执行 else 分支。

- 在第一个图中：所有线程执行相同的代码（效率高）。
 - 在第二个图中：有的线程条件为真（T），有的为假（F），就导致分歧（divergence）。
- 这种情况称为 *“branch divergence”（分支发散）”。

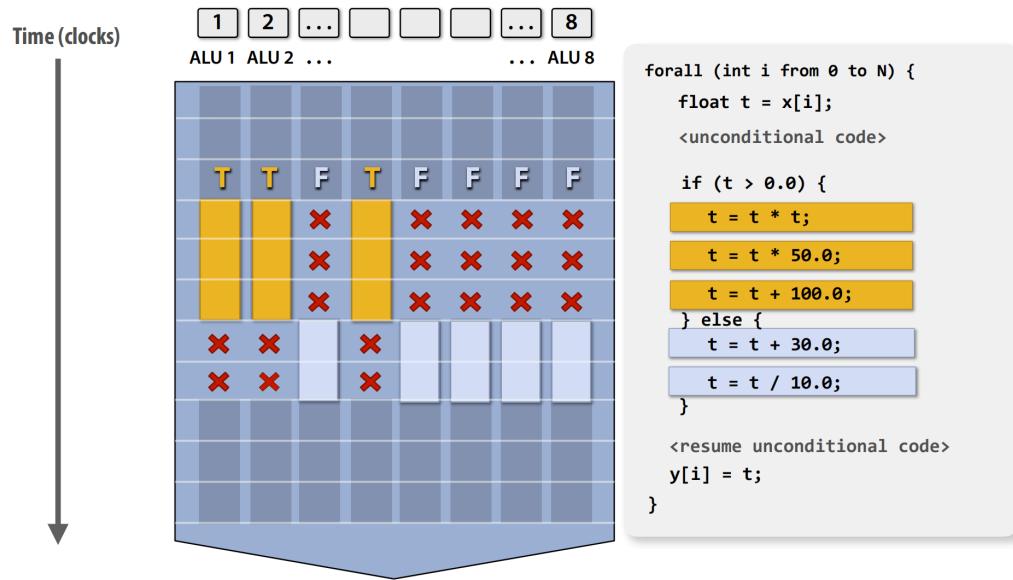
Mask (discard) output of ALU



```
forall (int i from 0 to N) {  
    float t = x[i];  
    <unconditional code>  
  
    if (t > 0.0) {  
        t = t * t;  
        t = t * 50.0;  
        t = t + 100.0;  
    } else {  
        t = t + 30.0;  
        t = t / 10.0;  
    }  
  
<resume unconditional code>  
    y[i] = t;  
}
```

掩盖或丢弃部分 ALU 的输出

After branch: continue at full performance



分支结束后恢复满速执行

- 当分支存在时，SIMD 并行处理器会为每个元素建立一个“掩码（mask）”来指示哪些 ALU 参与计算。
- 例如：只有部分 ALU 执行 if 块中的代码，其他的被“闲置”（标红叉）。
- 如果 **if** 和 **else** 块都很耗时，处理器需要先执行 **if** 分支，再执行 **else** 分支，即每个分支都执行两次，只是屏蔽掉一半的结果。

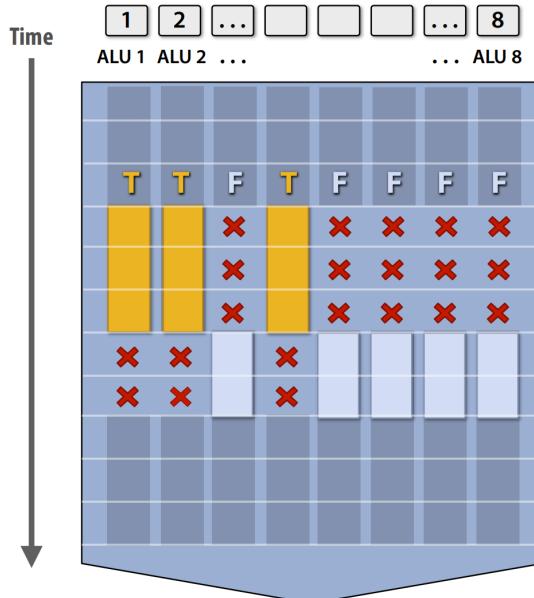
结果：

- 在最坏情况下，只有 **1/8** 的 **ALU** 在干活（其余空闲）。
- 执行完分支后，才恢复全部 ALU 的工作。
这就是 SIMD 架构下条件语句导致性能损失的原因。

Breakout question

Can you think of piece of code that yields the worst case performance on a processor with 8-wide SIMD execution?

Hint: can you create it using only a single "if" statement?



```
forall (int i from 0 to N) {  
    float t = x[i];  
    <unconditional code>  
    if (t > 0.0) {  
        ???  
    } else {  
        ???  
    }  
    <resume unconditional code>  
    y[i] = t;  
}
```

让芯片run at 1/8 的性能:

- 把1/8迭代送到if, 需要十亿条指令, expensive
- 7/8次迭代送到else, cheap path

其他理解:

- if (i % 8 == 0) {每八条数据执行一次if}
- else 留空

5、常见术语

Some common jargon

- **Instruction stream coherence (“coherent execution”)**
 - Property of a program where the same instruction sequence applies to many data elements
 - Coherent execution IS NECESSARY for SIMD processing resources to be used efficiently
 - Coherent execution IS NOT NECESSARY for efficient parallelization across different cores, since each core has the capability to fetch/decode a different instructions from their thread's instruction stream
- **“Divergent” execution**
 - A lack of instruction stream coherence in a program

■ ***指令流一致性 (Instruction Stream Coherence / Coherent Execution)

- 指的是：同一段指令序列被应用到多个数据元素上。
换句话说，不同数据执行的操作是一样的，只是操作的数不同。
- *Coherent Execution 是 SIMD 高效执行的必要条件。
因为 SIMD 需要所有通道（ALUs）执行相同指令。
- *但在多核（multi-core）并行中，不需要保持一致性 Coherent Execution。
各个核心可以执行不同指令流（不同线程），因为每个核心都有自己的取指和解码能力。

***分歧执行 (Divergent Execution)

- 指程序中缺乏指令流一致性。
- 例如： 语句导致部分线程执行 if 分支、部分执行 else 分支。
⇒ SIMD 并行度降低。

6. 现代 CPU 上的 SIMD 执行

SIMD execution: modern CPU examples

- Intel AVX2 instructions: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)
- Intel AVX512 instruction: 512 bit operations: 16x32 bits...
- ARM Neon instructions: 128 bit operations: 4x32 bits...
- Instructions are generated by the compiler
 - Parallelism explicitly requested by programmer using intrinsics
 - Parallelism conveyed using parallel language semantics (e.g., `forall` example)
 - Parallelism inferred by dependency analysis of loops by “auto-vectorizing” compiler
- Terminology: “explicit SIMD”: SIMD parallelization is performed at compile time
 - Can inspect program binary and see SIMD instructions (`vstoreps`, `vmulps`, etc.)

SIMD 指令集示例

- **Intel AVX2** 指令: 256位操作 (8×32位 或 4×64位) (8-wide float vectors)
👉 一次能并行计算 8 个浮点数或 4 个双精度数。
- **Intel AVX512** 指令: 512位操作 (16×32位)
- **ARM Neon** 指令: 128位操作 (4×32位)

指令的来源

- 编译器生成 SIMD 指令的三种方式：

1. 程序员直接使用 **intrinsics** (内建函数) ;
2. 使用支持并行语义的语言结构 (如 **forall**) ;
3. 编译器自动识别循环中的独立性, 进行 自动向量化 (**auto-vectorization**) 。

■ ***“显式 SIMD” (Explicit SIMD)

- 编译阶段完成并行化;
- 可通过查看编译后的二进制程序找到 SIMD 指令, 如:
`vstoreps`, `vmulps` 等。

7、现代 GPU 上的 SIMD 执行

SIMD execution on many modern GPUs

TL;DR — see “going farther” video

- “Implicit SIMD”
 - Compiler generates a binary with scalar instructions
 - But N instances of the program are always run together on the processor
 - Hardware (not compiler) is responsible for simultaneously executing the same instruction from multiple program instances on different data on SIMD ALUs
- SIMD width of most modern GPUs ranges from 8 to 32
 - Divergent execution can be a big issue
(poorly written code might execute at 1/32 the peak capability of the machine!)

■ ***“隐式 SIMD” (Implicit SIMD)

- 编译器仍生成普通的标量指令;
- 但硬件在执行时会让多个实例 (N 个线程) 同步执行同一条指令;
- 即: GPU 的硬件负责将同一指令作用于不同数据 (SIMD ALUs) 。

■ GPU SIMD 特性

- 多数 GPU 的 SIMD 宽度 (SIMD width) 在 **8 到 32** 之间;
- 如果发生分歧执行 (**divergent execution**) , 性能可能骤降;
例如在 32-lane SIMD 上, 最坏情况下仅 1/32 的 ALU 有效。

2.3 三种并行执行形式总结

Summary: three different forms of parallel execution

- **Superscalar:** exploit ILP within an instruction stream. Process different instructions from the same instruction stream in parallel (within a core)
 - Parallelism automatically discovered by the hardware during execution
- **SIMD:** multiple ALUs controlled by same instruction (within a core)
 - Efficient for data-parallel workloads: amortize control costs over many ALUs
 - Vectorization done by compiler (explicit SIMD) or at runtime by hardware (implicit SIMD)
- **Multi-core:** use multiple processing cores
 - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
 - Software creates threads to expose parallelism to hardware (e.g., via threading API)

正交概念：

■ *超标量 (Superscalar)

同一个核心中，利用指令级并行 (ILP) 在同一条指令流中实现并行执行多条不同的指令。

- 这种并行性由硬件自动发现和调度，程序员或编译器无需显式标明。
- 🤝 例：现代 CPU (如 Intel Core 系列) 可以在单个时钟周期同时发射多条不同的指令到不同功能单元执行，比如一条加法、一条乘法、一条内存访问。

■ *单指令多数据 (SIMD)

同一个核心中，由一条指令同时控制多个 ALU，对多个数据执行相同操作。

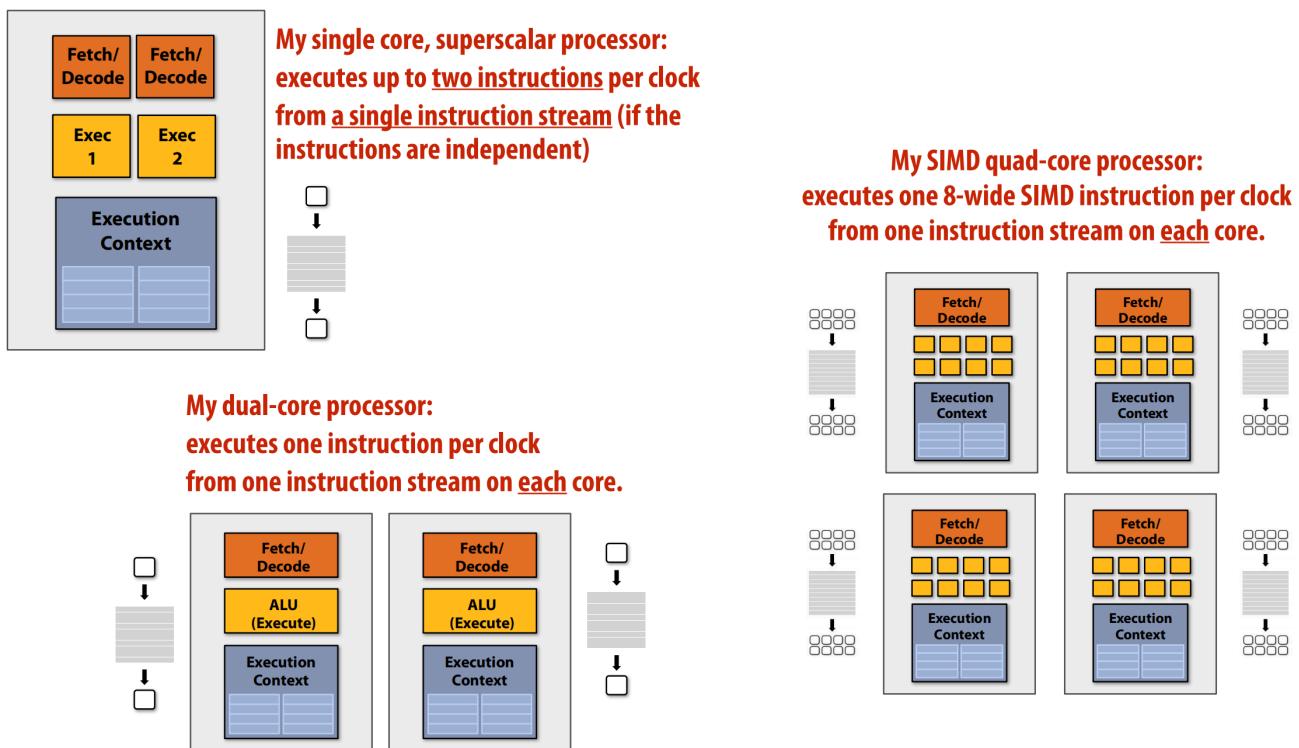
- 对数据并行任务非常高效（如向量运算、图像处理、矩阵计算等），因为控制成本被分摊到多个数据上。
- 向量化执行可以：
 - 由编译器显式完成 (Explicit SIMD, 例如 AVX、Neon 指令集)；
 - 或由硬件在运行时自动完成 (Implicit SIMD, 例如 GPU 的 warp 机制)。
- 🤝 例：当计算 $y[i] = x[i] * 2$ 时，SIMD 可以同时对 8 个元素执行乘法操作，而不是逐个循环

■ *多核 (Multi-core)

多个处理核心，每个核心都有自己独立的指令流。

- 提供线程级并行性 (**Thread-level Parallelism, TLP**)，即每个核心可同时执行不同的指令流。
- 由软件创建线程（如使用线程库、OpenMP、CUDA 等）来显式地向硬件暴露并行性。
- 🤝 例：四核 CPU 可以同时运行四个线程，每个线程执行不同任务或数据片段，从而实现加速。

对比维度	Superscalar	SIMD	Multi-core
指令流数量	1	1	多个
数据流数量	1	多个	多个
并行粒度	指令级并行 (ILP)	数据级并行 (DLP)	线程级并行 (TLP)
控制方式	由硬件自动发现	同一指令控制多个 ALU	软件创建多线程
特点	同一核心执行多条不同指令	适合批量数据运算	多核心独立运行不同指令流
示例	CPU (乱序执行)	AVX / NEON / GPU warp	多核 CPU / GPU SM



■ 单核超标量处理器 (**1 core, superscalar**) :

每个时钟周期可以从单一指令流 (single instruction stream) 中执行最多两条指令 (前提是这些指令之间相互独立)。

同时取出并解码多条独立指令，并发送到不同的执行单元（如加法器、乘法器等）。

- 每条指令来自同一线程（单指令流）。
- 并行性由硬件自动检测（称为 ILP：Instruction-Level Parallelism）。

■ 双核处理器（**2 cores, no vector, no superscalar**）：

每个时钟周期在每个核心上，从各自的指令流中执行一条指令。

多核（multi-core）并行。

- 每个核心都有自己的取指、解码、执行单元和上下文（即独立线程）。
- 两个核心可以并行执行不同程序（不同指令流）。
- 并行性由软件或操作系统提供（例如多线程、任务分配）。

■ 四核 SIMD 处理器（**4 cores, 8-wide vector, no superscalar**）：

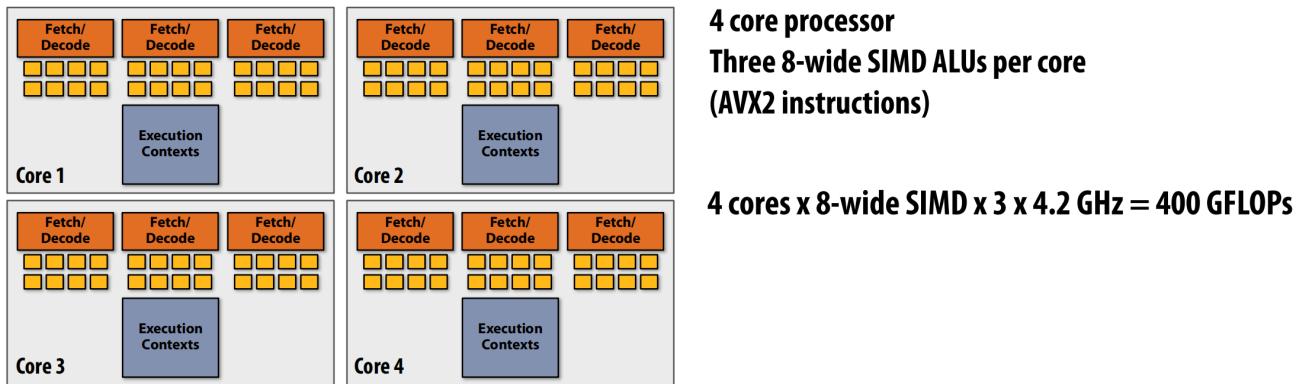
每个时钟周期在每个核心上，从各自的指令流执行一条8通道宽（8-wide）SIMD 指令。

结合了多核（multi-core）与 SIMD（单指令多数据）：

- 每个核心内部执行 SIMD 指令（即一次对 8 个数据并行操作）；
- 同时有 4 个核心并行运行，形成“核间并行 + 核内并行”的双层结构；
- 每个核心独立取指、解码、执行，但共享同样的程序结构。

结构	核心数量	指令流数量	每周期执行	并行粒度	并行来源
Superscalar （超标量）	1	1	最多 1 条指令流上的 2 条独立指令	指令级 (ILP)	硬件自动发现
Dual-core （双核）	2	2	每核 1 条指令流	线程级 (TLP)	软件线程调度
SIMD Quad-core （四核 + SIMD）	4	4	每核 1 条指令流上的 1 条 8-wide SIMD 指令	数据级 + 线程级 (DLP + TLP)	编译器 + 软件

Example: four-core Intel i7-7700K CPU (Kaby Lake)



* Showing only AVX math units, and fetch/decode unit for AVX (additional capability for integer math)

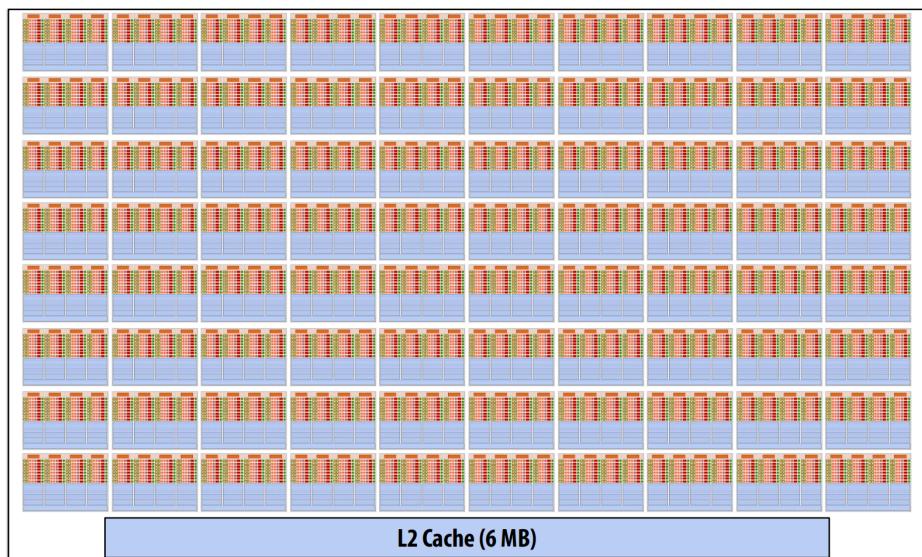
四核 Intel i7-7700K CPU (Kaby Lake)

4 cores, per core (3-way superscalar, 8 vector)

- 四核心处理器 (**multi-cores**)，每个核心包含3个8路 SIMD (AVX2 指令) 运算单元 (ALU)
- 每个时钟周期每个核心可以运行3条并行指令 (**superscalar**)，这些指令是 vector instructions 向量指令 (**SIMD**)
- 理论峰值计算能力为：
 $4\text{核心} \times 8\text{路 SIMD} \times 3\text{单元} \times 4.2\text{GHz} = 400\text{GFLOPs}$
最多 $4 \times 8 \times 3 = 96$ 次 operations

图中只显示 AVX 浮点运算单元及其取指/译码模块 (AVX 还具备额外的整数运算能力)

Example: NVIDIA V100 GPU



80 “SM” cores

128 SIMD ALUs per “SM” (@1.6 GHz) = 16 TFLOPs (~250 Watts)

NVIDIA V100 GPU

80 cores, per core (128 ALUs, organized in 32-wide SIMD instructions)

- 80 个“SM” cores (Streaming Multiprocessor, 流式多处理器) (**multi-cores**) ;
- 每个 SM 包含 128 个 SIMD 执行单元 (ALU) ;
- 以 1.6 GHz 频率运行, 总理论性能约为:
 $80 \times 128 \times 1.6\text{GHz} = 16\text{TFLOPs}$
最多 80×128 次 operations
- 功耗约为 **250 瓦**;
- L2 缓存为 **6 MB**.