
Parallel Architecture and Programming

现代处理器的并行工作原理及编程示例

(**processor perspective + understanding latency and bandwidth**)

中国科学院计算技术研究所

高性能计算机研究中心

邵恩

今天的授课内容

- 今天我们主要从从处理器架构角度来认识并行
- 主要讲现代处理器的并行工作原理及编程示例
 - **第一部分：处理器实现并行计算的两种方式：多核和SIMD**
 - **第二部分：访存已经成为处理器提升算力的瓶颈**
- 了解这些处理器架构基础知识将帮助你
 - 了解并优化并行程序的性能
 - 直观了解哪些工作负载可能会从这些并行架构中受益

第一部分： 处理器实现并行计算的两种方式

首先我们来看一个程序样例

对包含N个浮点数的数组的每个元素

使用泰勒展开计算 $\sin(x)$: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$

```
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;

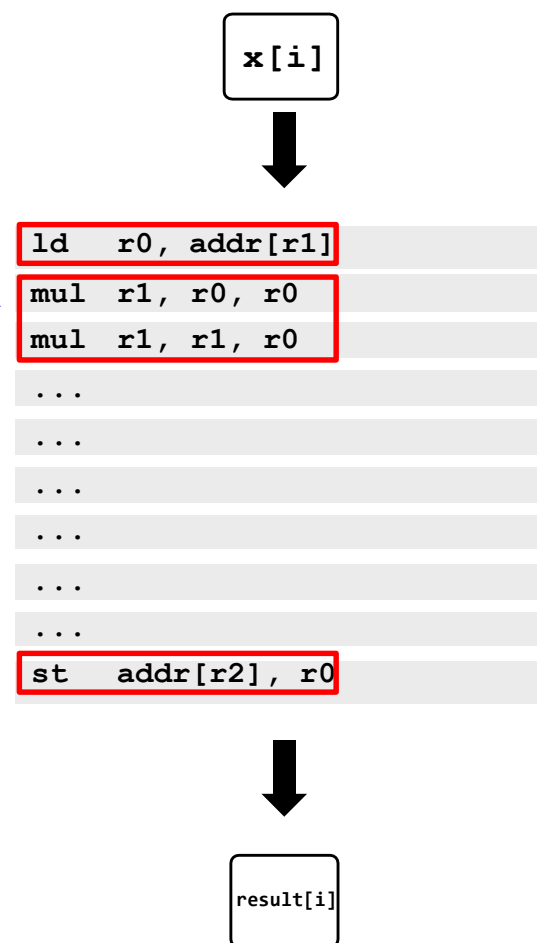
        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

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

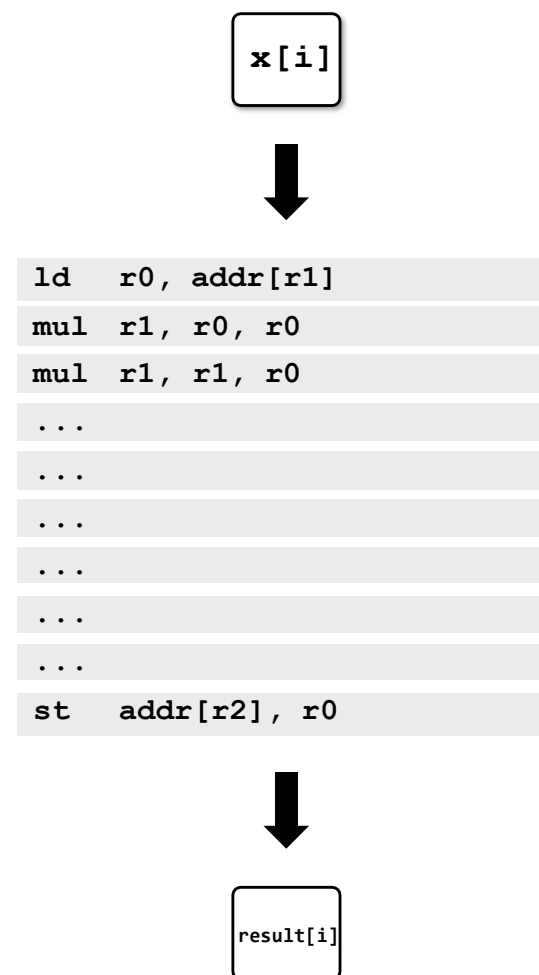
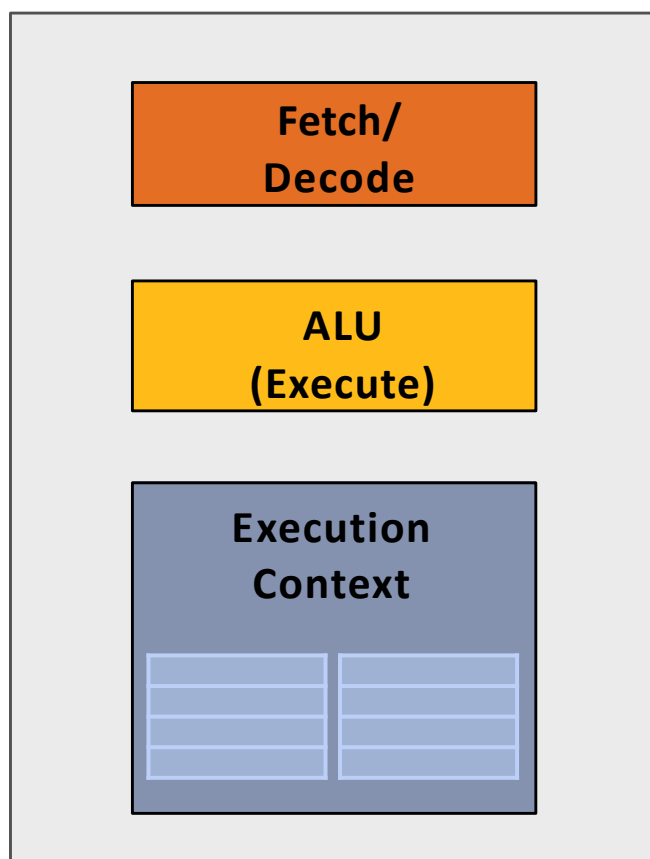
编译程序

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

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

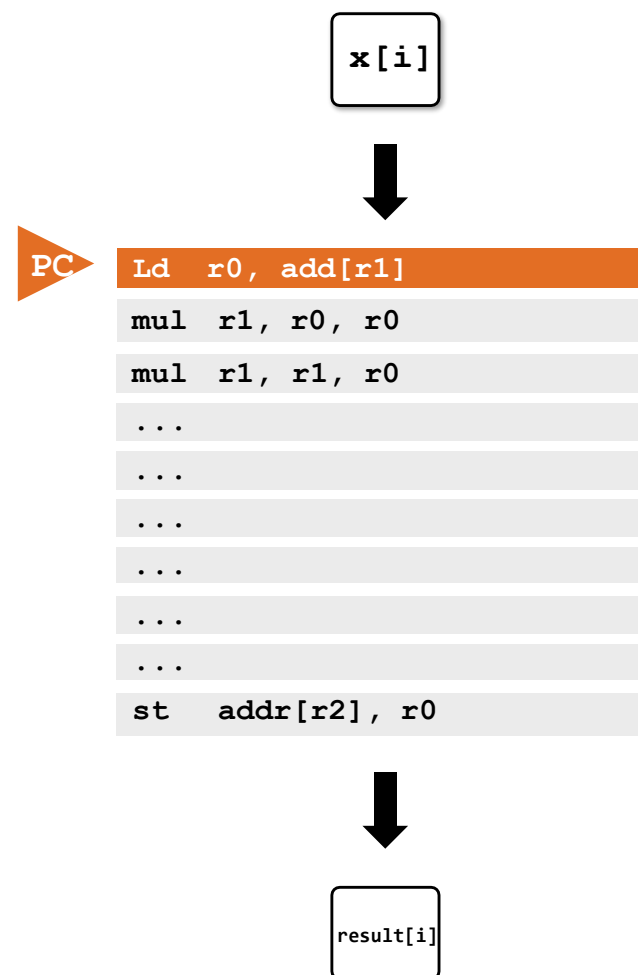
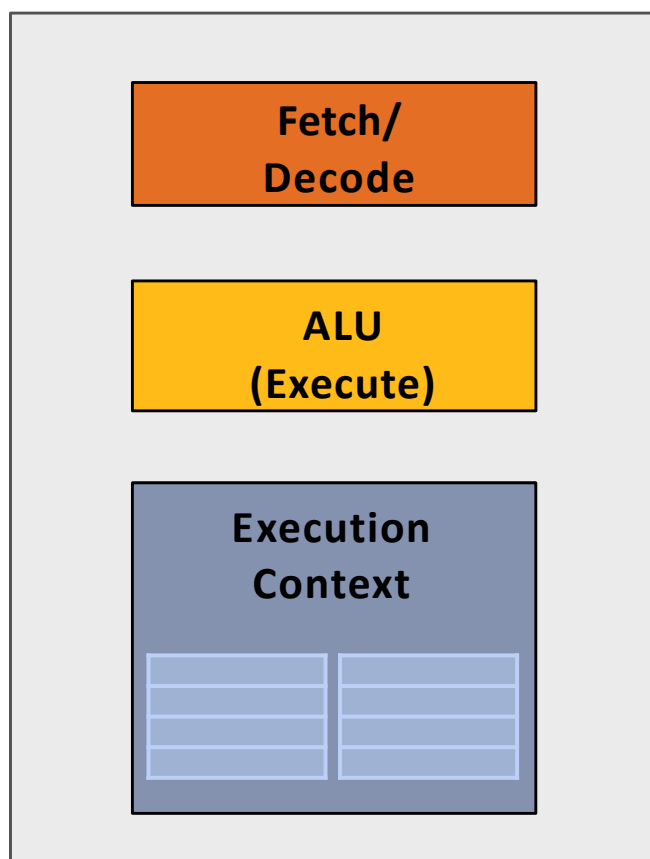


执行程序



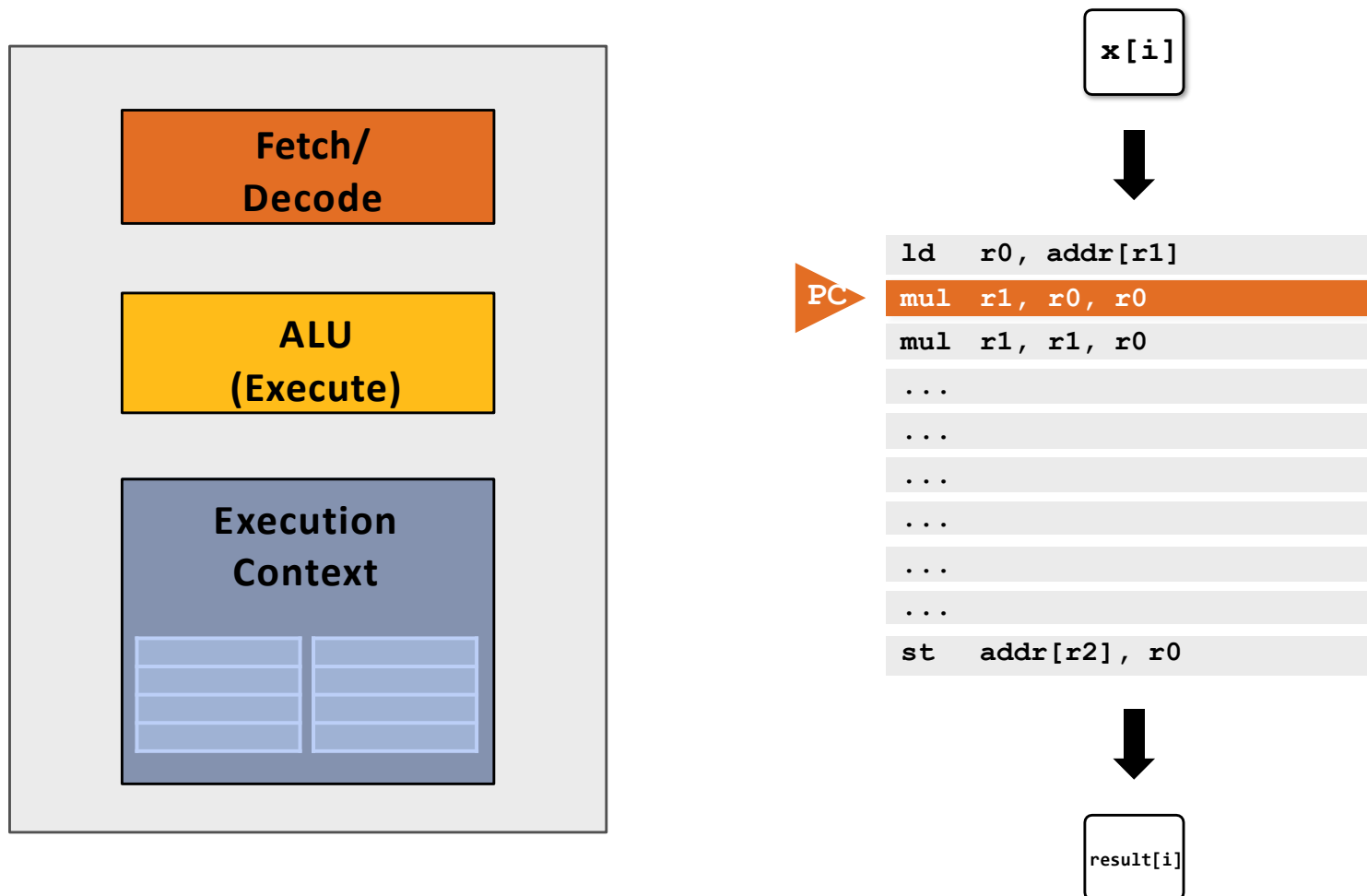
执行程序

写一个非常简单的处理器：每个时钟执行一条指令



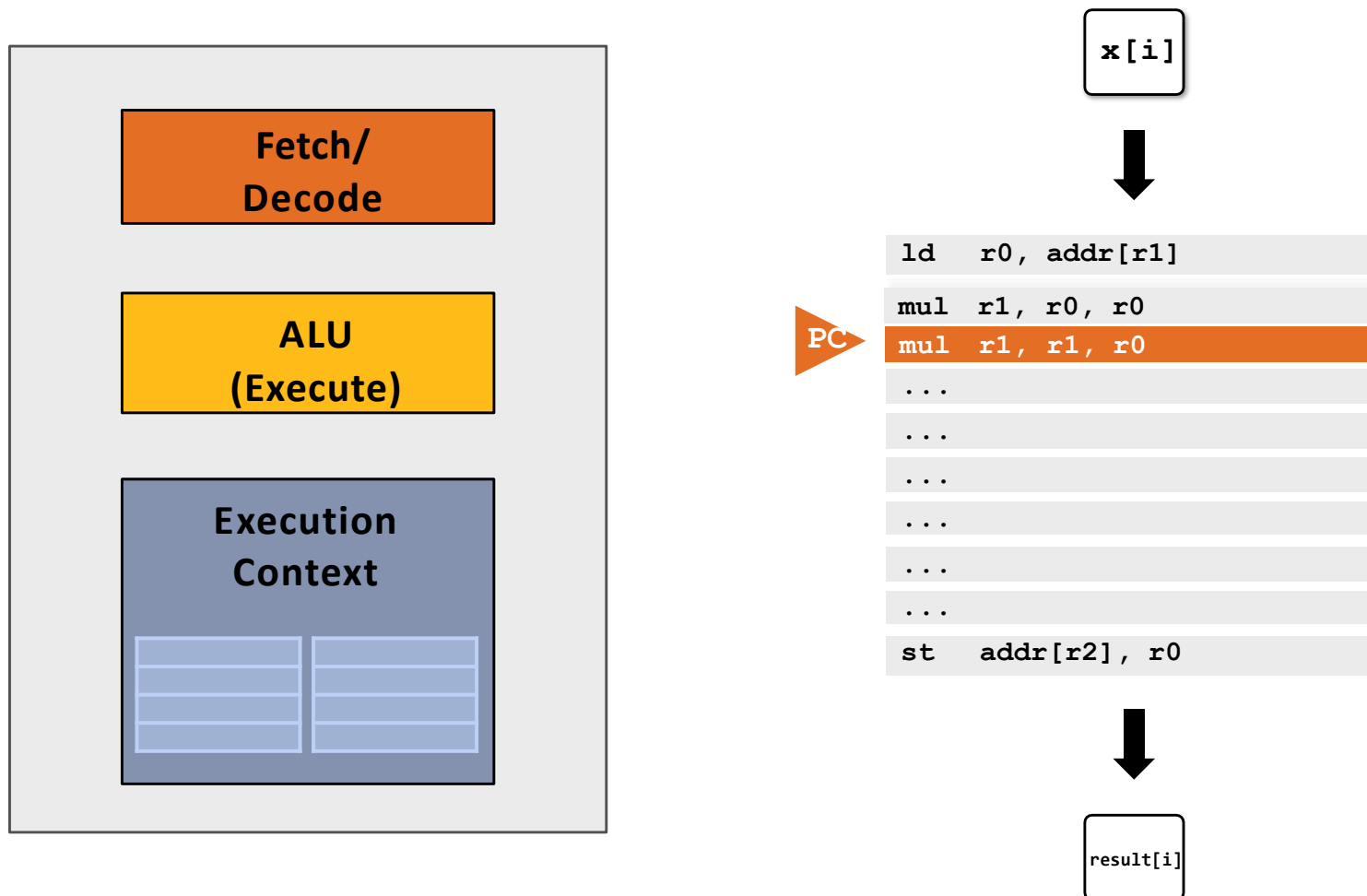
执行程序

写一个非常简单的处理器：每个时钟执行一条指令



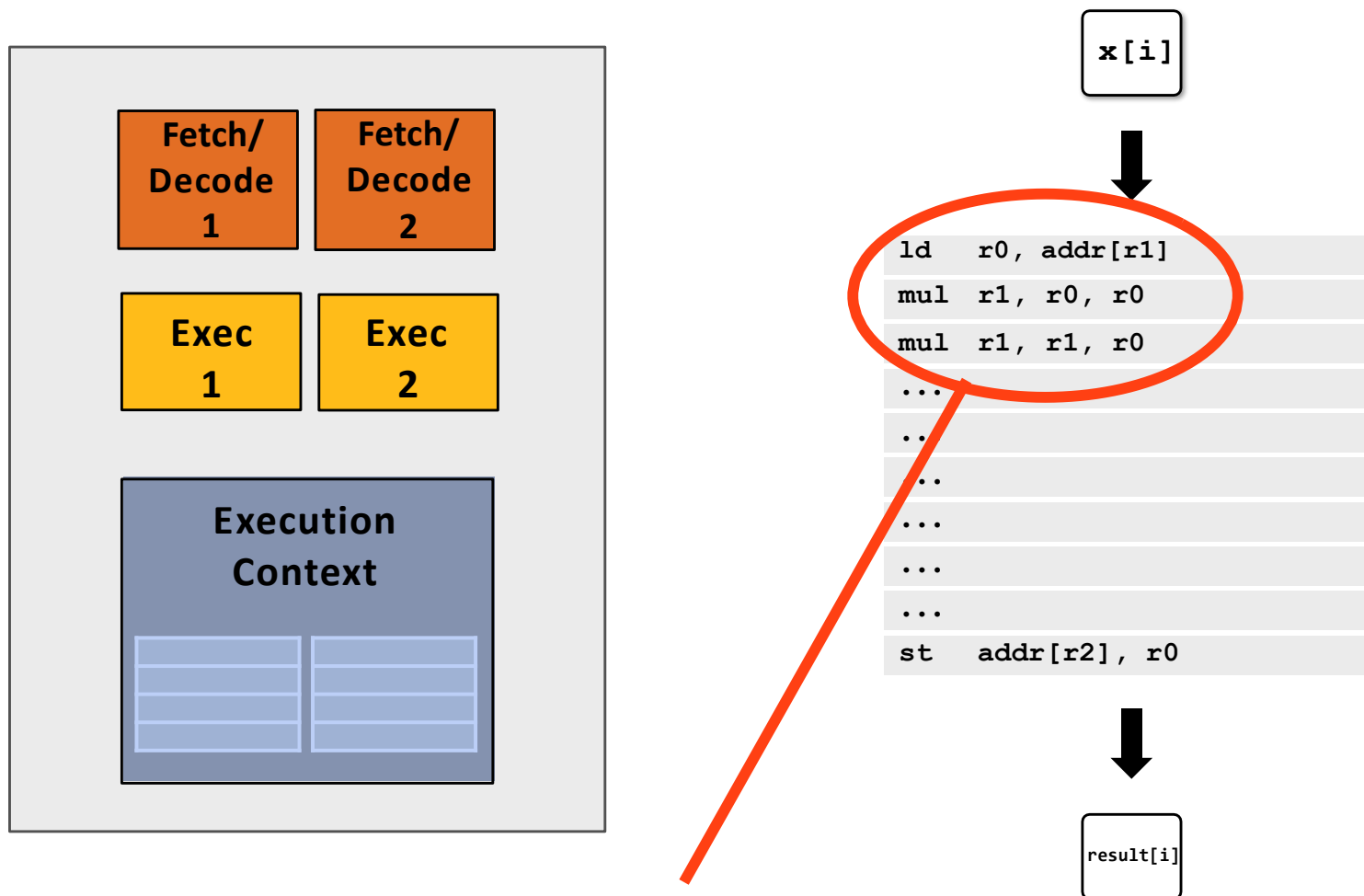
执行程序

写一个非常简单的处理器：每个时钟执行一条指令



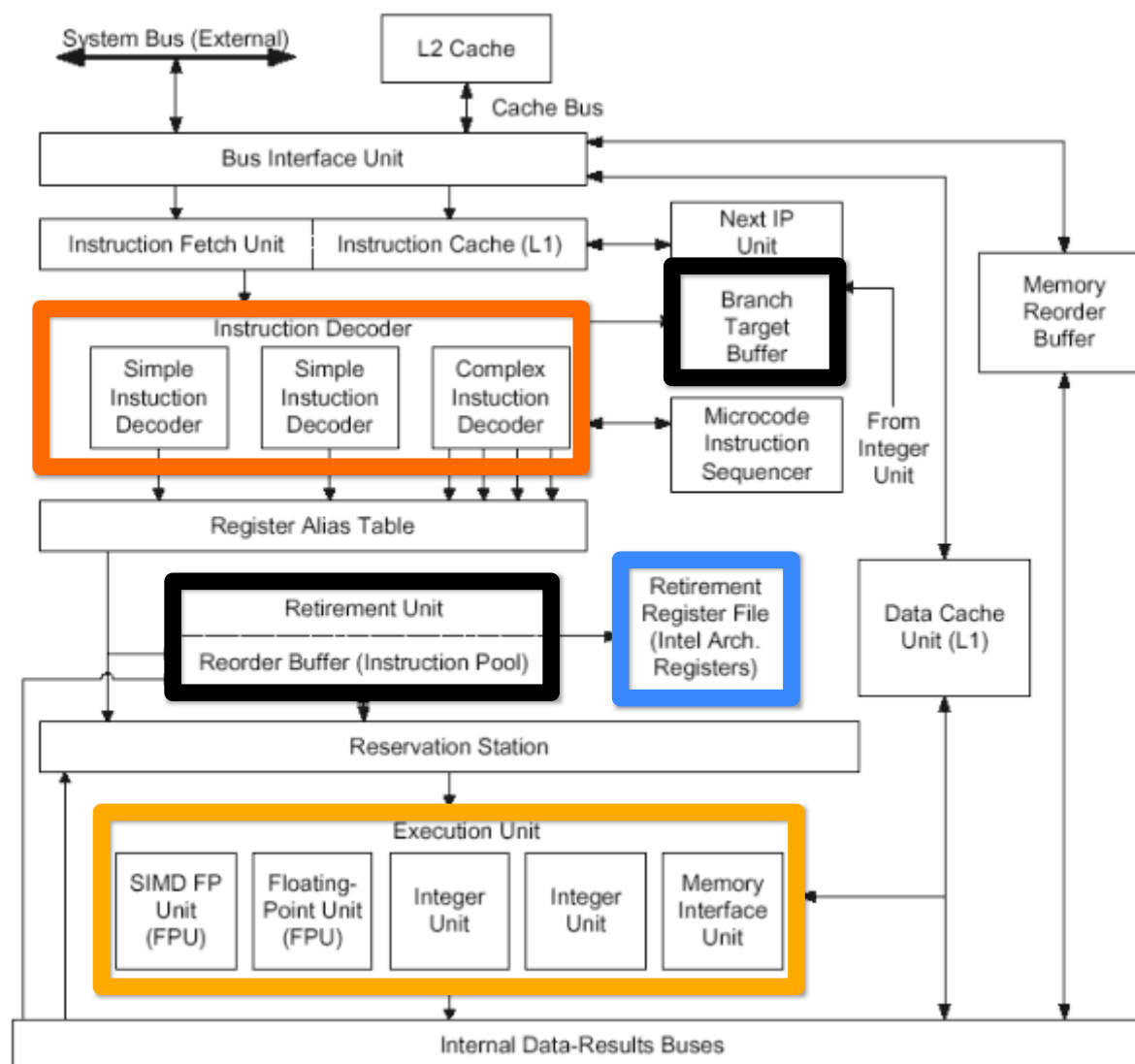
超标量处理器

指令级并行 (ILP) 有可能实现每个时钟解码并执行两条指令



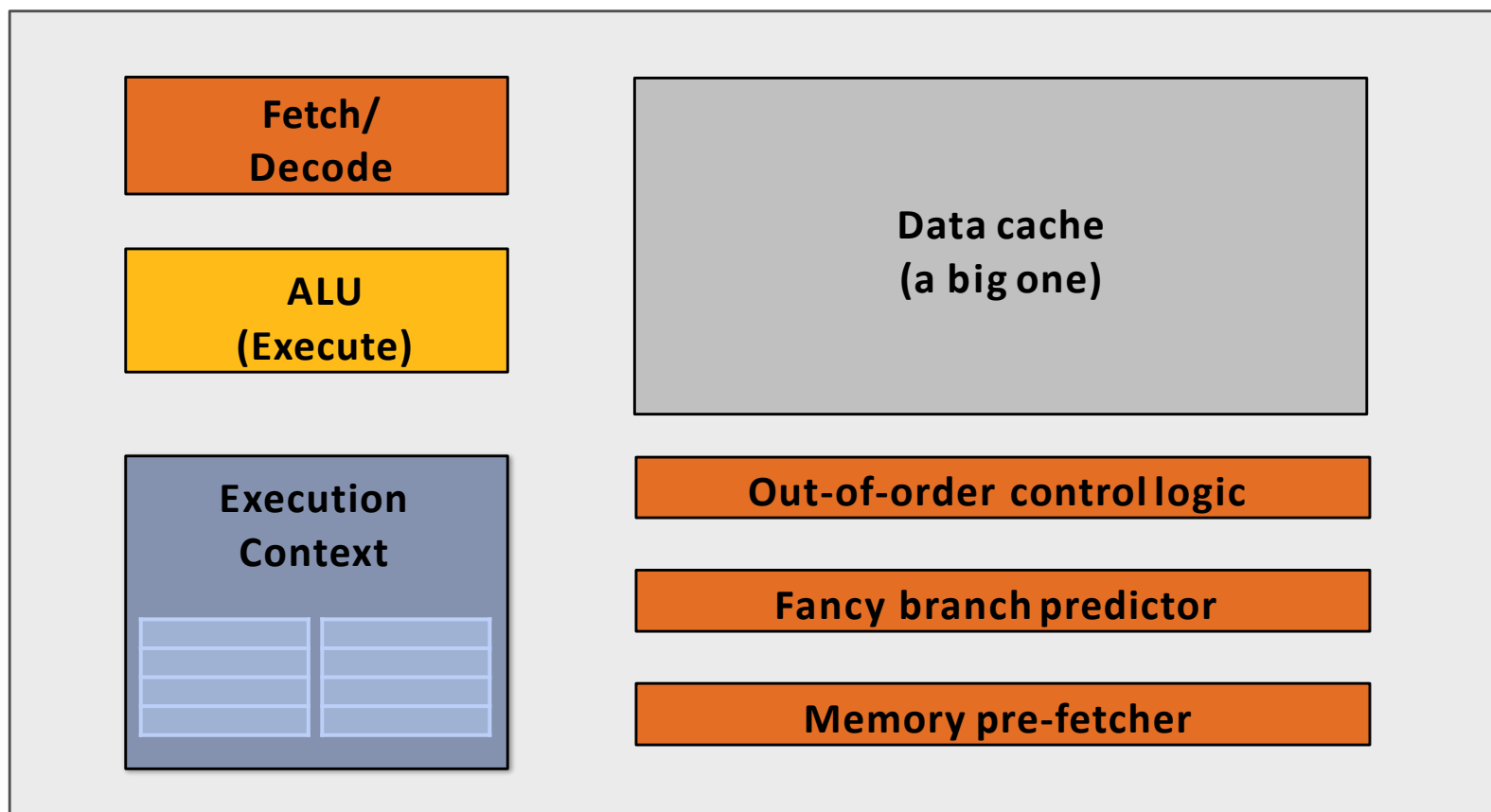
Note: No ILP exists in this region of the program

例如: Pentium 4



处理器：前多核时代

遵循摩尔定律而增加的大多数晶体管都用于使单个指令流更加快速地运行



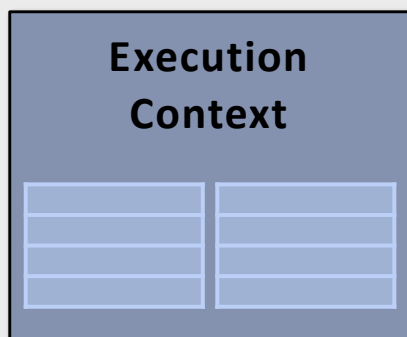
更多的晶体管=更大的cache, 更加高效的乱序执行逻辑,更智能的分支预测器,etc.

(Also: more transistors → smaller transistors → higher clock frequencies)

处理器：多核时代

Fetch/
Decode

ALU
(Execute)

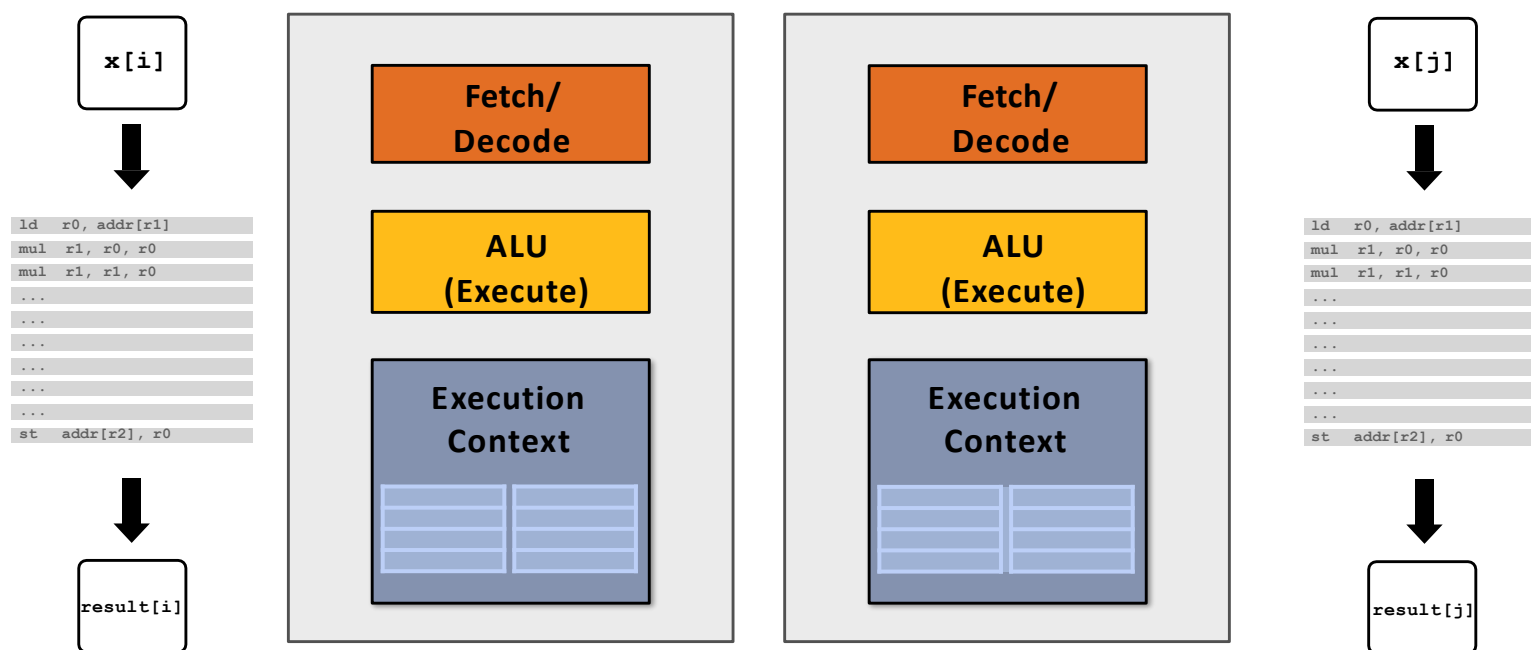


想法 #1:

使用增加的晶体管来为处理器添加更多内核

而不是使用晶体管来增加加速单个指令流的
处理器逻辑的复杂性 (e.g., out-of-order
and speculative operations)

双核：并行计算两个元素



更简单的内核：每个内核在运行单个指令流时都比我们原来的“花式（Fancy）”内核慢 (e.g., 25% slower)

但是现在有两个核心： $2 \times 0.75 = 1.5$ (potential for speedup!)

但是我们的程序没有表现出并行性

```
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;
    }
}
```

这个用gcc编译的程序将作为一个线程在处理器内核之一上运行.

如果每个较简单的处理器内核都比原来的单个复杂处理器内核慢**25%**，那么我们的程序现在运行速度会慢**25%.** :-(

使用 pthreads 表达并行性

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

void parallel_sinx(int N, int terms, float* x, float* result)
{
    pthread_t thread_id;
    my_args args;

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

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    sinx(N - args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL); // wait for thread end
}

void my_thread_start(void* thread_arg)
{
    my_args* thread_args = (my_args*)thread_arg;
    sinx(thread_args->N, thread_args->terms, thread_args->x, thread_args->result); // do work
}
```

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

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

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

Posix标准线程库pthreads

数据并行表达

(在我们虚构的数据并行语言中)

```
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;

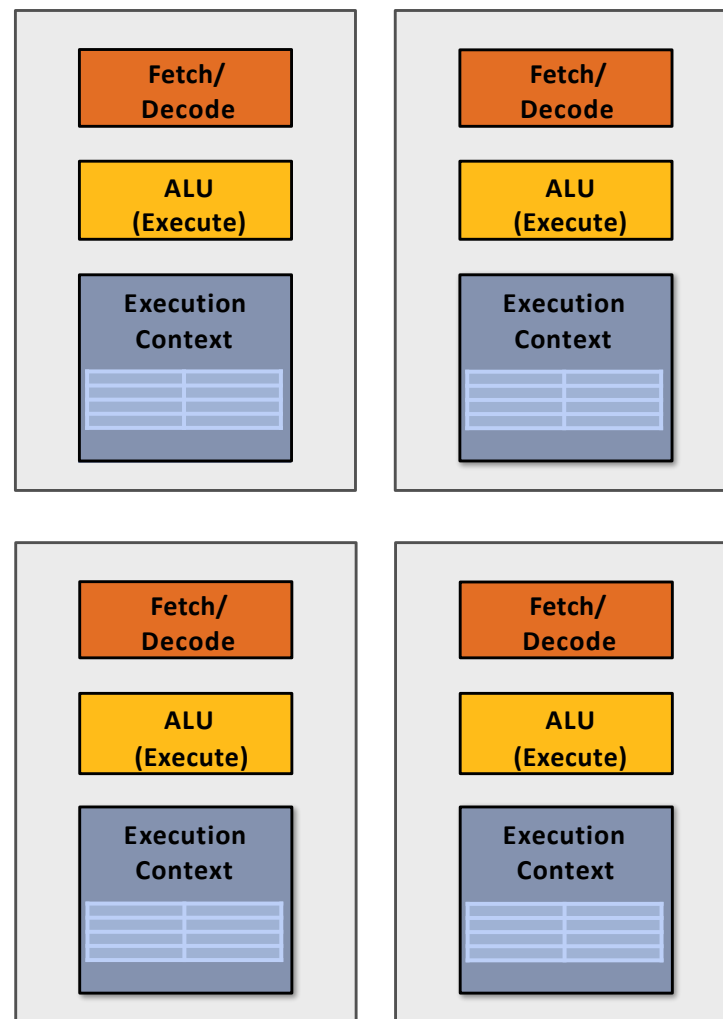
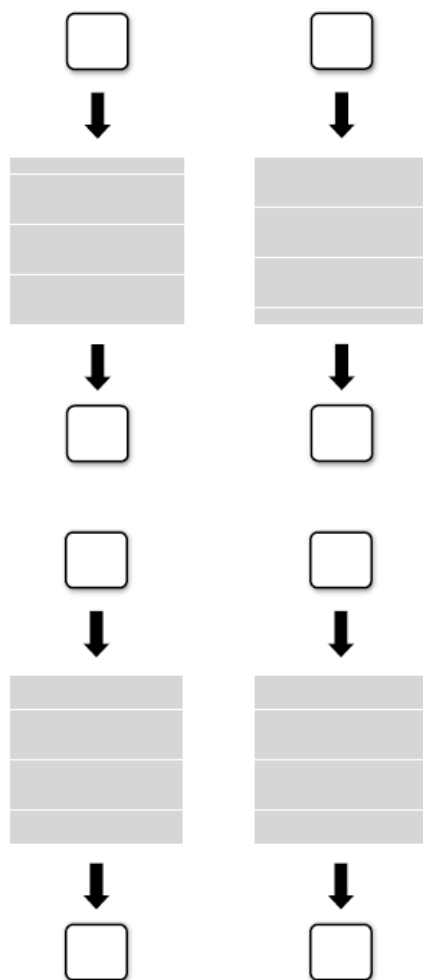
        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

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

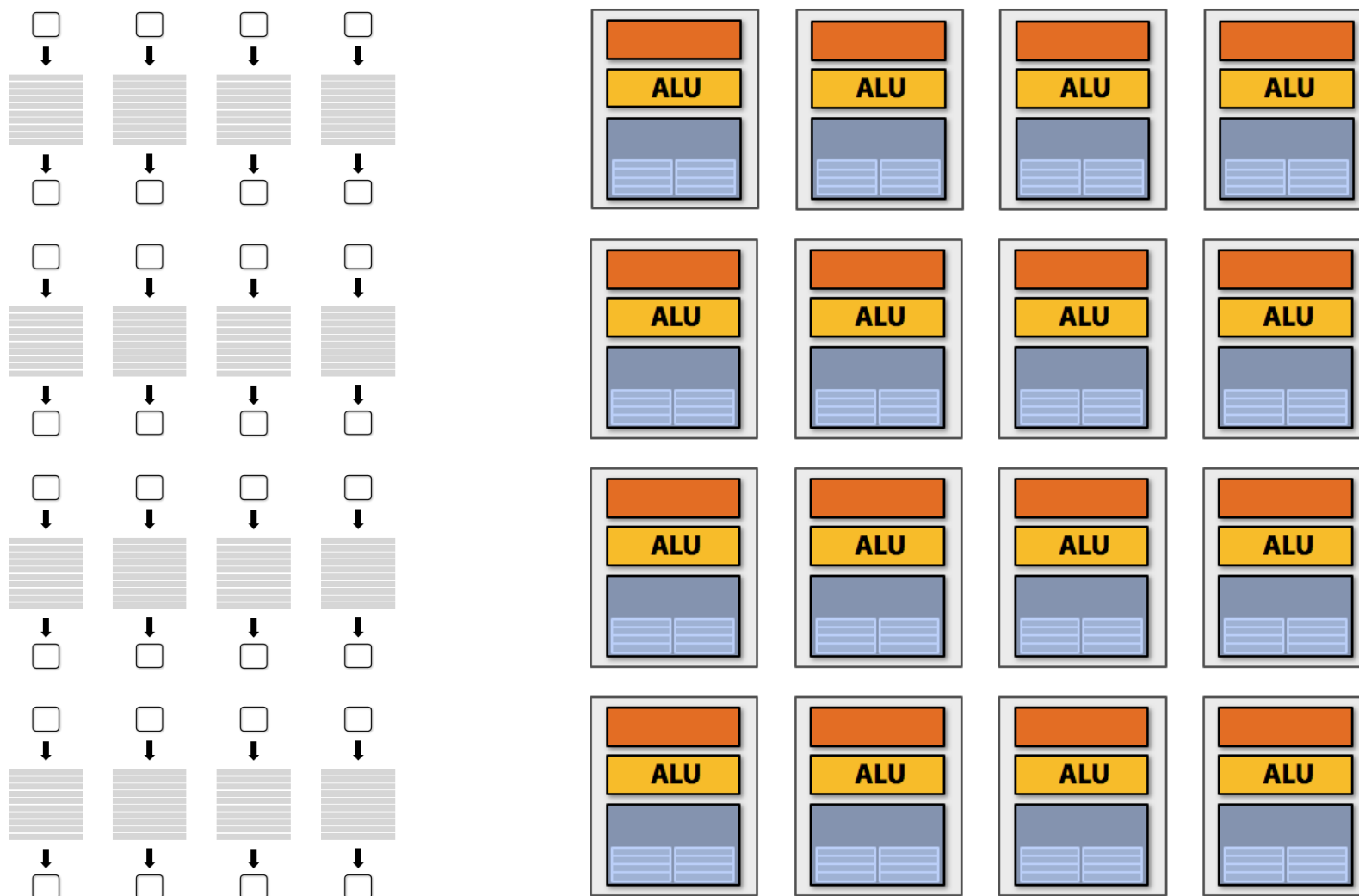
程序员声明的循环迭代是独立的

有了这些信息，你可以想象编译器如何自动生成并行线程代码

四核：并行计算四个元素

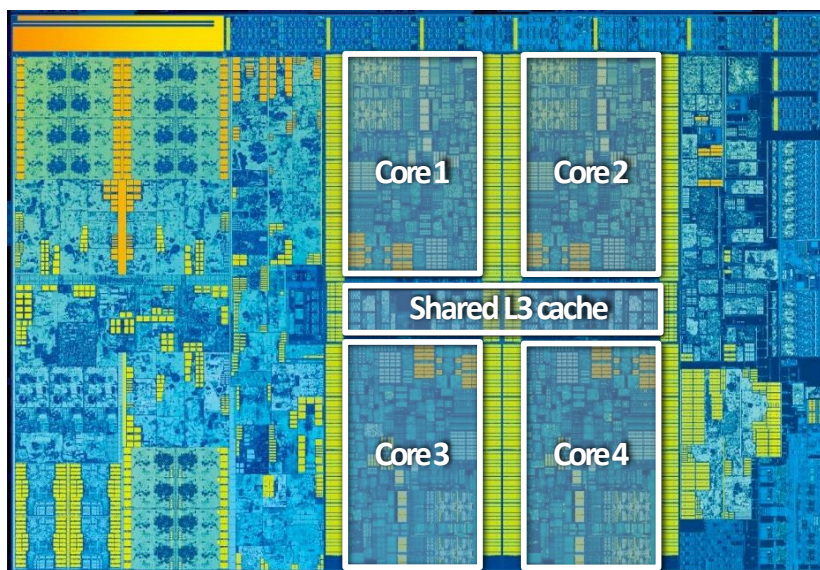


十六核：并行计算十六个元素

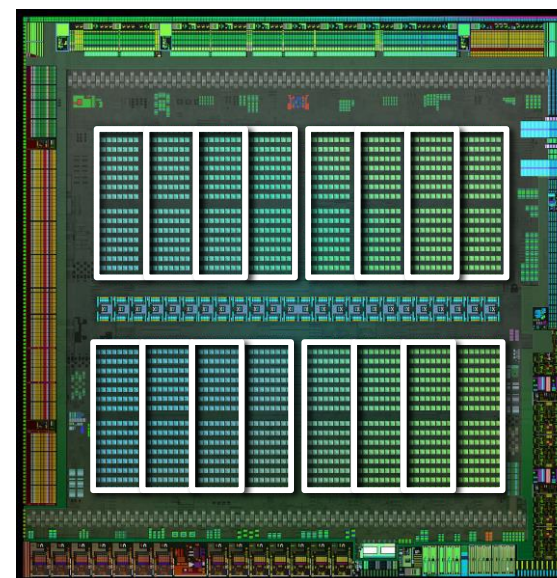


Sixteen cores, sixteen simultaneous instruction streams

多核示例

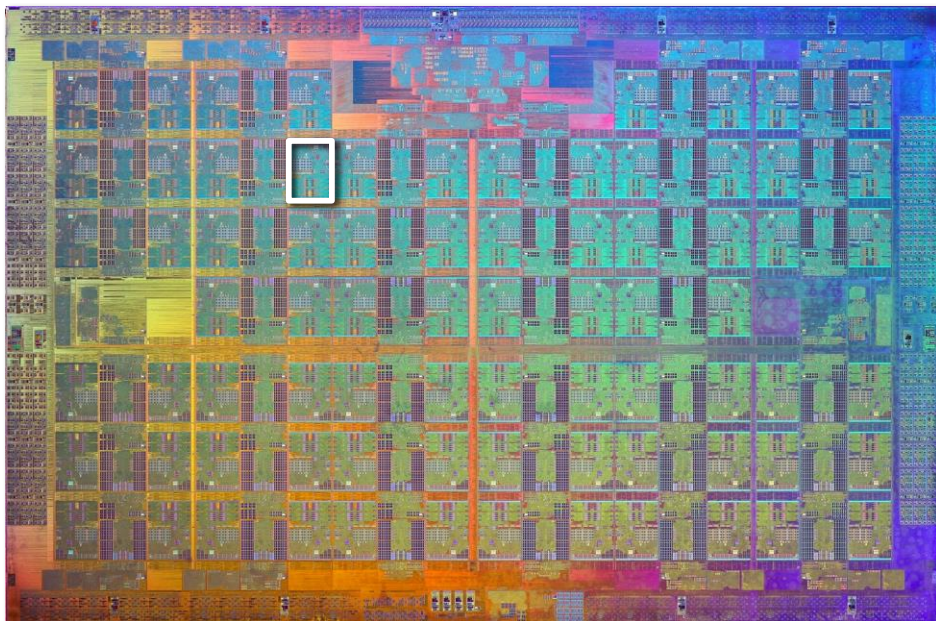


Intel "Skylake" Core i7 quad-core
CPU (2015)

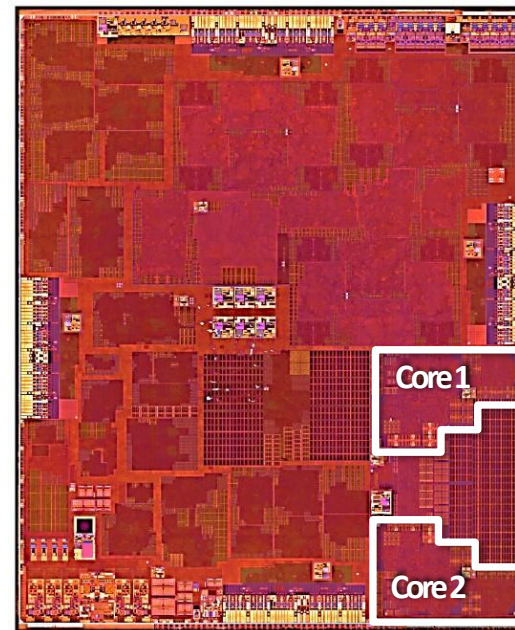


NVIDIA GTX 980 GPU
16 replicated processing cores
("SM") (2014)

更多的多核示例



Intel Xeon Phi "Knights Landing" 76-core
CPU (2015)



Apple A9 dual-core
CPU (2015)

数据并行表达

(在我们虚构的数据并行语言中)

```
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;
        }

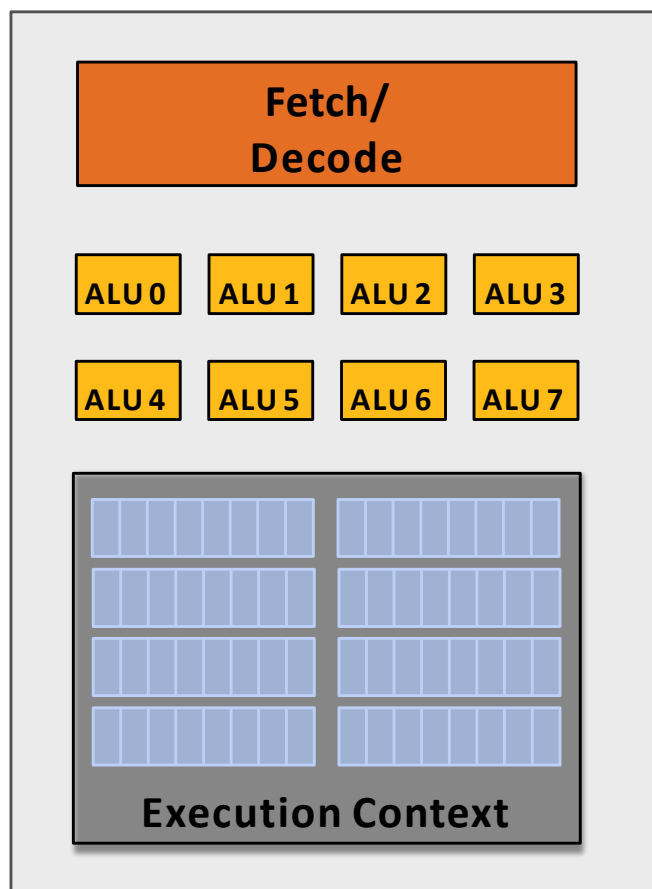
        result[i] = value;
    }
}
```

此代码的另一个有趣属性是：

并行性跨越循环的迭代。

循环的所有迭代都做同样的事情：计算单个输入数的正弦值

添加 ALU 以提高计算能力



想法 #2:

分摊管理跨多个 ALU 的指令流的成本/
复杂性

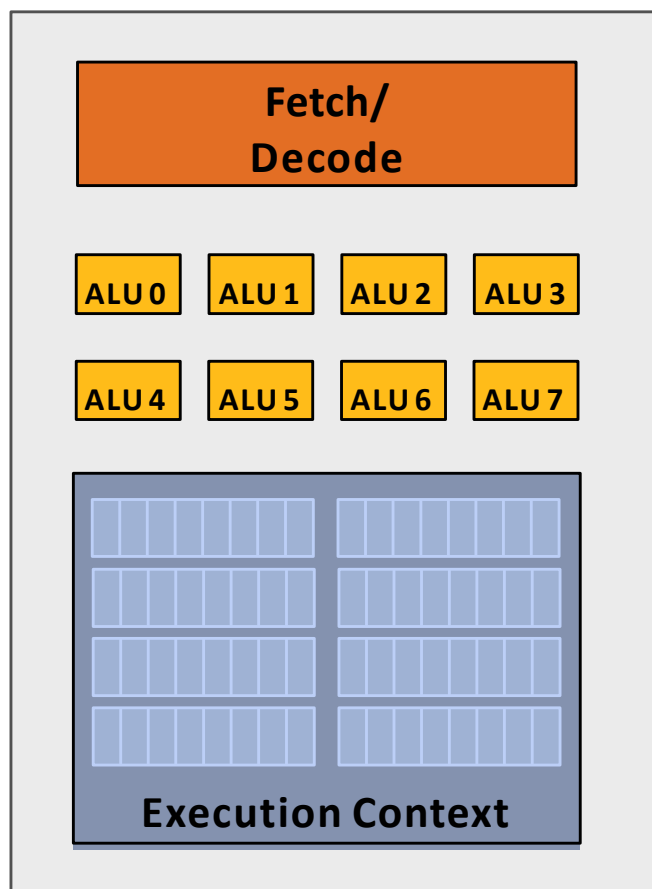
SIMD处理

Single instruction, multiple data

单指令，多数据

向所有 ALU 广播相同的指令 在所有 ALU 上
并行执行

添加 ALU 以提高计算能力



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

回顾之前的编译程序:

指令流使用标量寄存器上的标量指令一次
处理一个**数组元素(e.g., 32-bit floats)**

Vector program (using AVX intrinsics)

```
#include <immintrin.h>
```

```
void sinx(int N, int terms, float* x, float* result)
```

```
{
```

```
    float three_fact = 6; // 3!
```

```
    for (int i=0; i<N; i+=8)
```

```
    {
```

```
        __m256 origx = _mm256_load_ps(&x[i]);
```

```
        __m256 value = origx;
```

```
        __m256 number = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
```

```
        __m256 denom = _mm256_set1ps(three_fact); float
```

```
        sign = -1;
```

```
        for (int j=1; j<=terms; j++)
```

```
        {
```

```
            // value += sign * number / denom
```

```
            __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1ps(sign), number), denom);
```

```
            value = _mm256_add_ps(value, tmp);
```

```
            number = _mm256_mul_ps(number, _mm256_mul_ps(origx, origx));
```

```
            denom = _mm256_mul_ps(denom, _mm256_set1ps((2*j+2)*(2*j+3)));
```

```
            sign *= -1;
```

```
        }
```

```
        _mm256_store_ps(&result[i], value);
```

```
    }
```

```
}
```

Intel提供的SIMD函数接口

1.一种纯C语言，良好的交互接口

2.手工向量化实现并行操作

3.Intel intrinsics Guide官网查询

Vector program (using AVX intrinsics)

```
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* result)
{
    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_set1ps(three_fact);
        float 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);

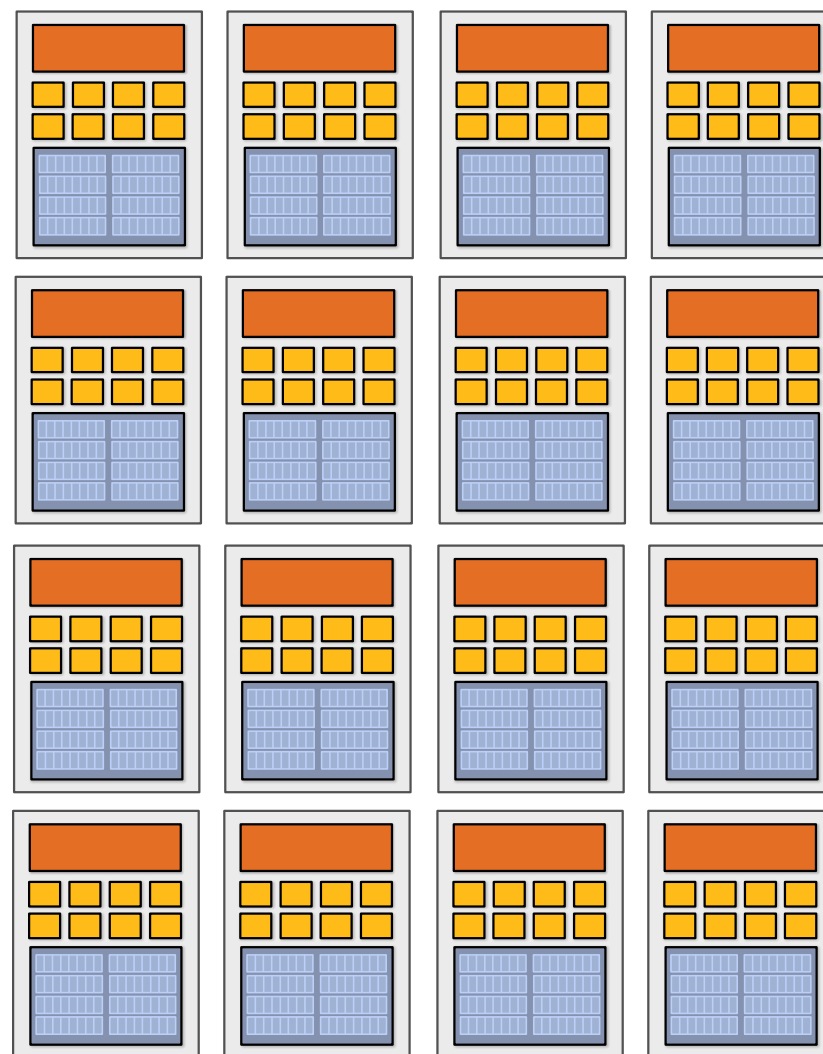
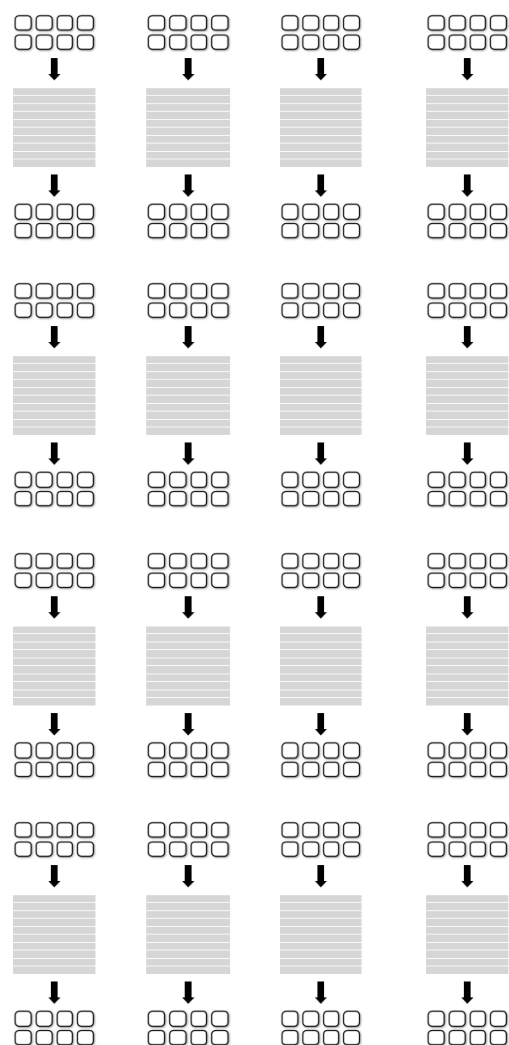
            numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
            denom = _mm256_mul_ps(denom, _mm256_set1ps((2*j+2)*(2*j+3)));
            sign *= -1;
        }
        _mm256_store_ps(&result[i], value);
    }
}
```

```
vloadps  xmm0, addr[r1]
vmulps   xmm1, xmm0, xmm0
vmulps   xmm1, xmm1, xmm0
...
...
...
...
...
...
vstoreps addr[xmm2], xmm0
```

Compiled program:

使用 256 位向量寄存器上的向量指令同时处理 8 个数组元素

16 SIMD cores: 128 elements in parallel



16 cores, 128 ALUs, 16 simultaneous instruction streams

Data-parallel expression

(in our fictitious data-parallel language)

```
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;

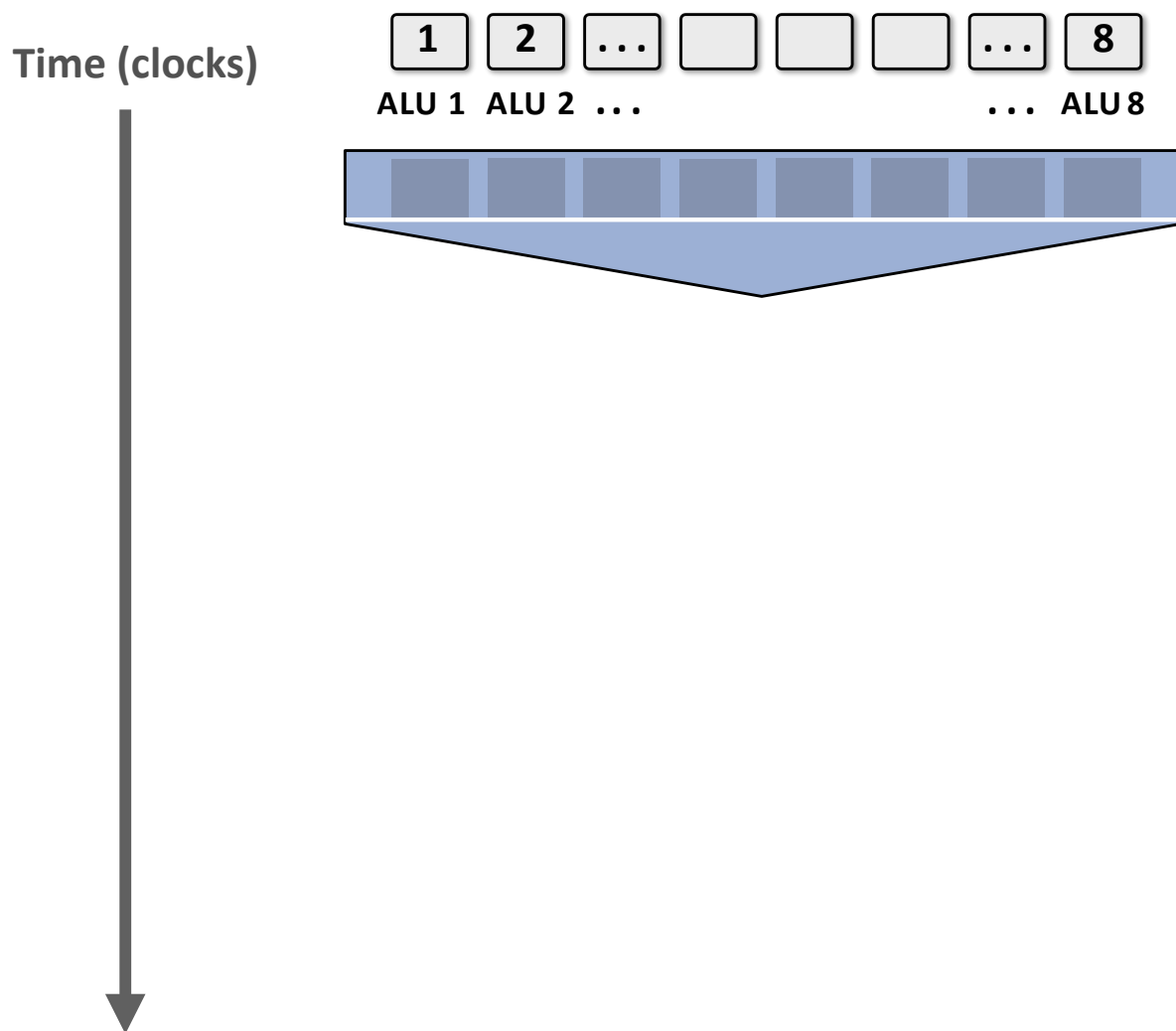
        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

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

编译器理解循环迭代是独立的，并且相同的循环体将在大量数据元素上执行。

Abstraction facilitates automatic generation of both multi-core parallel code, and vector instructions to make use of SIMD processing capabilities within a core.

如果遇到条件执行会怎样?



(假设下面的逻辑将对输入数组“A”中的每个元素执行，产生输出到数组“result”中)

```
<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;
}

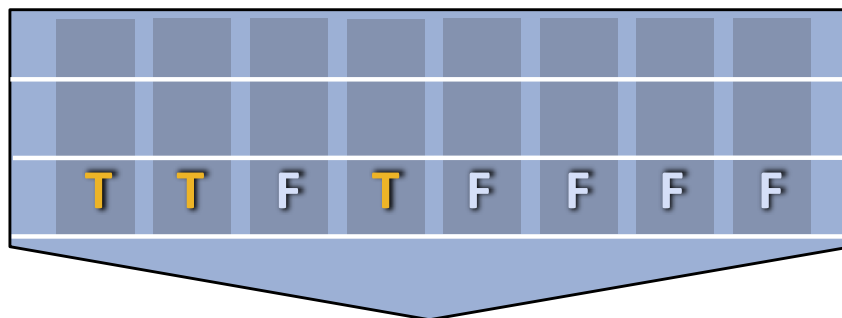
<resume unconditional code>

result[i] = x;
```

如果遇到条件执行会怎样?

Time (clocks)

1 2 ... ALU 1 ALU 2 ... ALU 8



(假设下面的逻辑将对输入数组“A”中的每个元素执行，产生输出到数组“result”中)

```
<unconditional code>

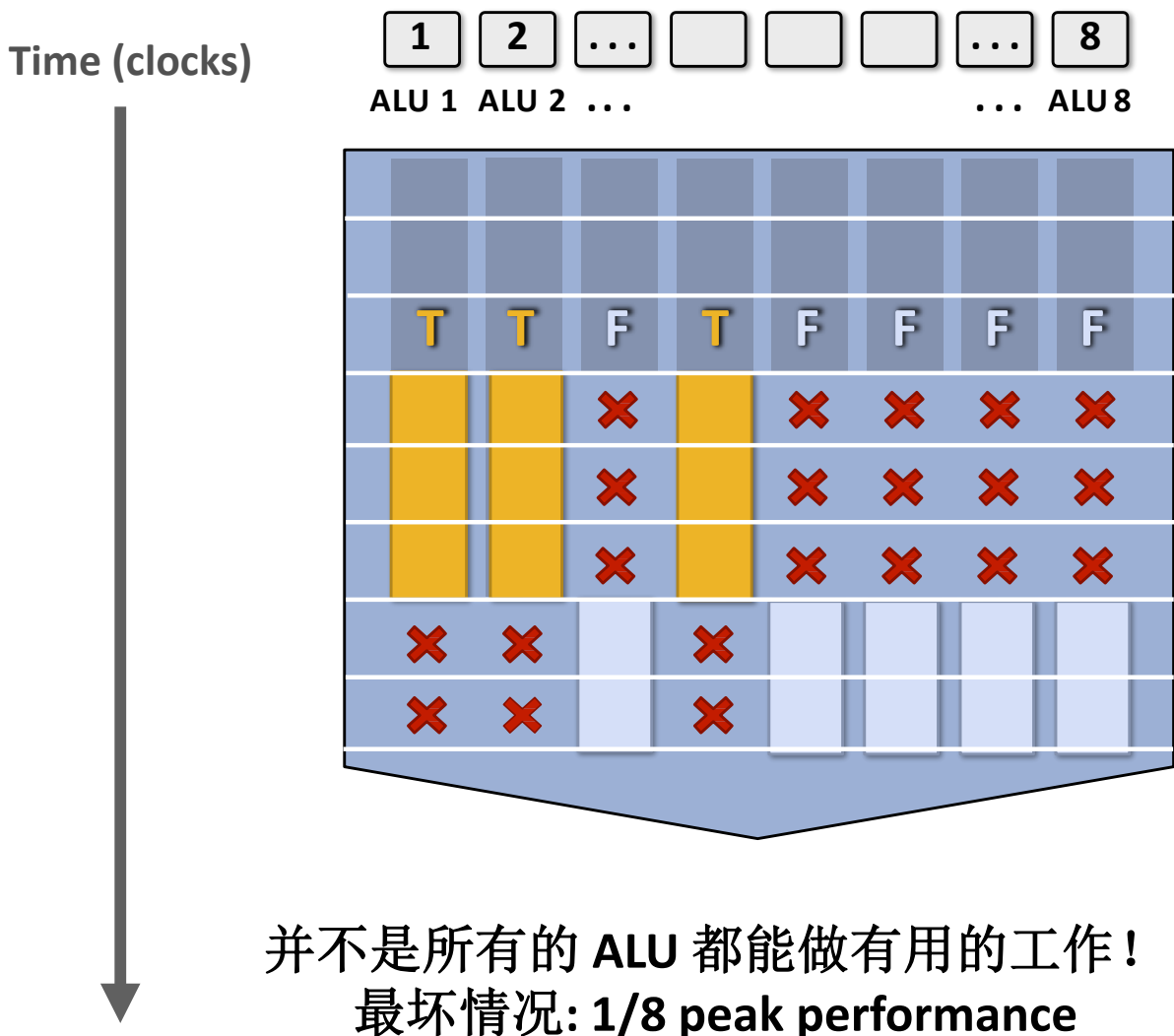
float x = A[i];

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;
```

ALU 的屏蔽（丢弃）输出



(假设下面的逻辑将对输入数组 “A” 中的每个元素执行，产生输出到数组 “result”中’)

```
<unconditional code>

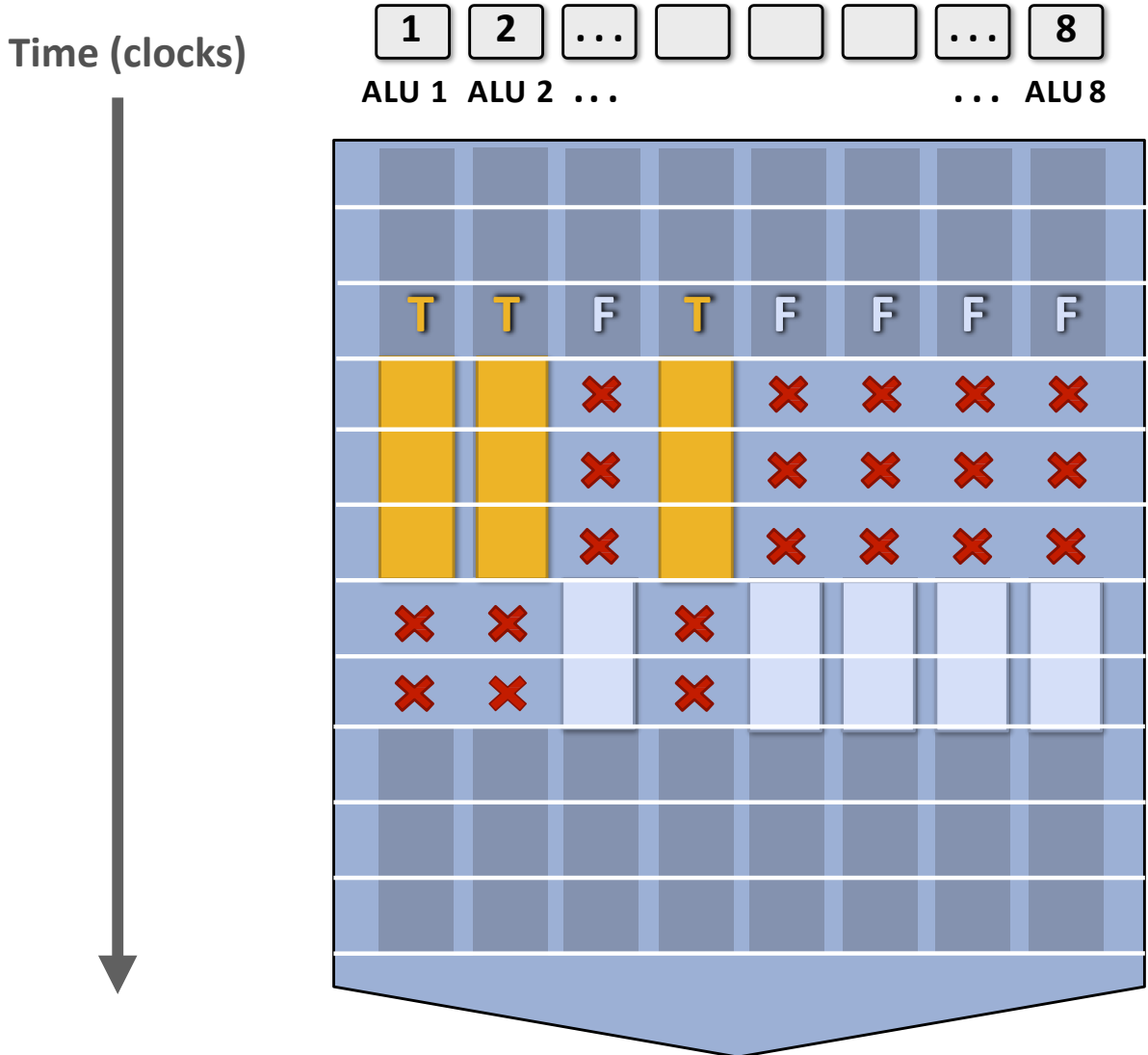
float x = A[i];

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;
```

分支后：继续全性能运行



(假设下面的逻辑将对输入数组 “A” 中的每个元素执行，产生输出到数组 “result”中’)

```
<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;
```


术语

- **指令流一致性Instruction stream coherence (一致执行“coherent execution”)**
 - 相同的指令序列应用于同时操作的所有元素
 - 一致性执行对于有效使用 SIMD 处理资源是必要的
 - 一致性执行对于跨内核的高效并行化不是必需的，因为每个内核都有能力获取/解码不同的指令流
- **“分歧”执行 “Divergent” execution**
 - 缺乏指令流一致性
- **注意：不要将指令流一致性与“缓存一致性”（本课程后面的一个主要主题）混淆**

在现代 CPU 上执行 SIMD

- SSE instructions: 128-bit operations: 4x32 bits or 2x64 bits (4-wide float vectors)
- AVX instructions: 256-bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)
- AVX512 instructions: 512-bit operations: 16x32 bits or 8x64 bits (16-wide float vectors)
- 指令由编译器生成，但有三种方法
 - 程序员使用内建函数明确请求的并行性 (intrinsics)
 - 使用并行语言语义传达并行性(e.g., forall example)
 - 通过循环的依赖分析推断的并行性 (hard problem, even best compilers are not great on arbitrary C/C++ code)
- 术语：“显式SIMD (explicit SIMD)”：在编译时执行SIMD并行化
 - 可以检查程序二进制文件并查看说明(vstoreps, vmulps, etc.)

在现代GPUs的SIMD

■ “隐式SIMD (Implicit SIMD)”

- 编译器生成标量二进制表达(scalar instructions)

- 但是程序的 N 个实例总是在处理器上一起运行

```
execute(my_function, N) // execute my_function N times
```

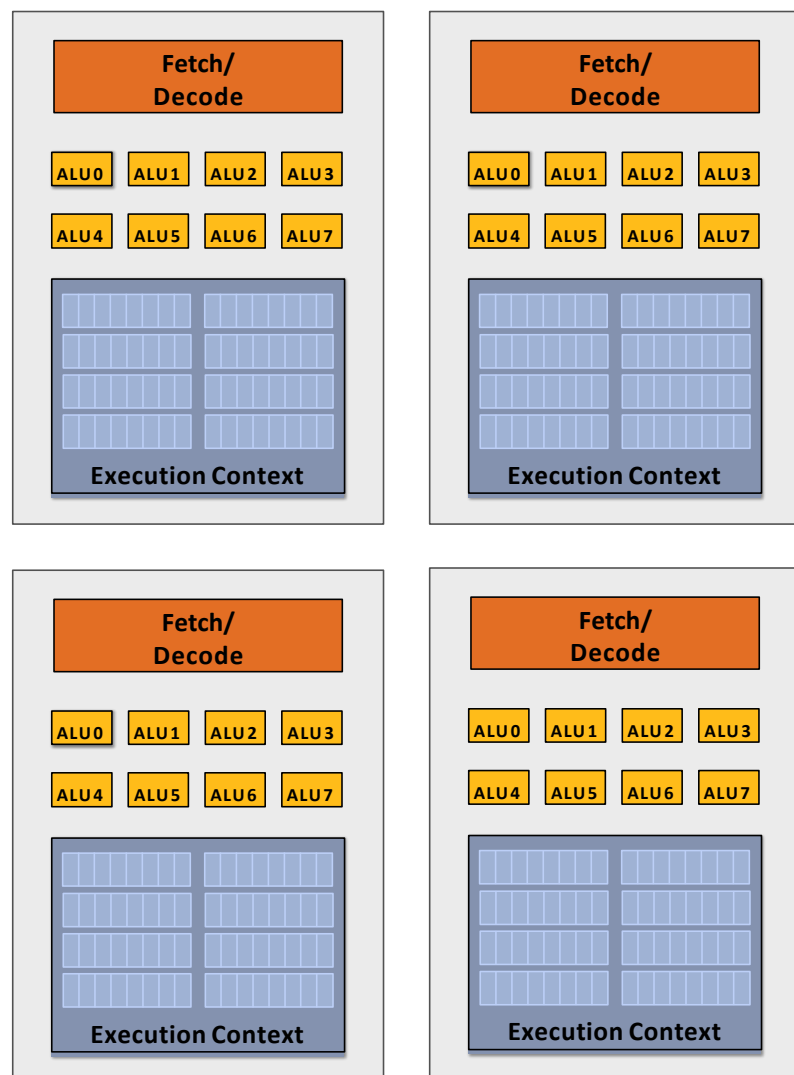
- 换句话说，硬件本身的接口是数据并行的
- 硬件（不是编译器）负责在 SIMD ALU 上的不同数据上同时执行来自多个实例的相同指令

■ 大多数现代 GPU 的 SIMD 宽度范围为 8 到 32

- Divergence can be a big issue

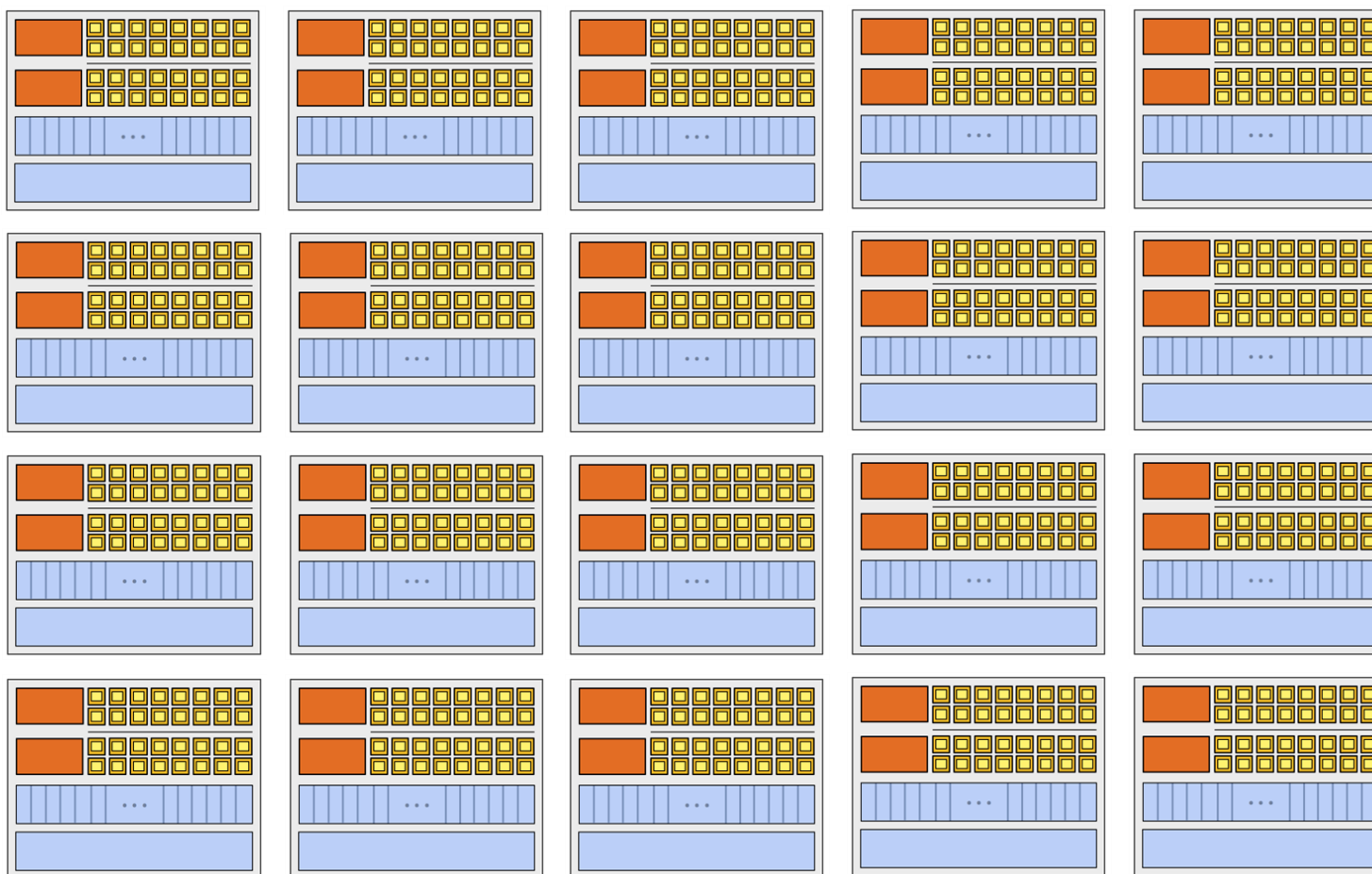
(写得不好的代码可能会以机器峰值能力的 1/32 执行!)

Example: Intel Core i7



4 cores
8 SIMD ALUs per core
(AVX instructions)

Example: NVIDIA GTX 1080



- 20 cores
- 32 SIMD ALUs per core
- 8.2 TFLOPS

总结: 处理器的并行处理介绍

■ 现代处理器并行执行的三种形式

① 多核: 使用多个处理核心

- 提供线程级并行性: 在每个内核上同时执行完全不同的指令流
- 软件决定何时创建线程 (e.g., via pthreads API)

② SIMD: 使用由相同指令流控制的多个 ALU (在一个内核中)

- 数据并行工作负载的高效设计: 控制分摊到多个 ALU
- 向量化可以由编译器 (显式 SIMD) 或在运行时由硬件完成
- [缺乏] 依赖性在执行前已知 (通常由程序员声明, 但可以通过高级编译器的循环分析推断)

③ 超标量: 在指令流中利用 ILP。并行处理来自同一指令流的不同指令 (在一个内核中)

- 硬件在执行期间自动和动态地发现并行性 (程序员不可见)
- 本课程不再进一步讨论 (计算机系统结构)。

第二部分： 访存已经成为处理器提升算力的瓶颈

术语

■ 访存延迟 **Memory latency**

- 内存服务一个来自处理器的内存请求(e.g., load, store)所需的时间
- 例如: 100 cycles, 100 nsec

■ 访存带宽 **Memory bandwidth**


- 内存系统向处理器提供数据的速率
- 例如: 20 GB/s

停顿 Stalls

- 当处理器的运行指令流中出现了：后一条指令依赖于前一条指令，前一条指令迟迟无法完成，处理器就会“停顿”（**Stall**）。

- 访问内存是停顿的主要来源

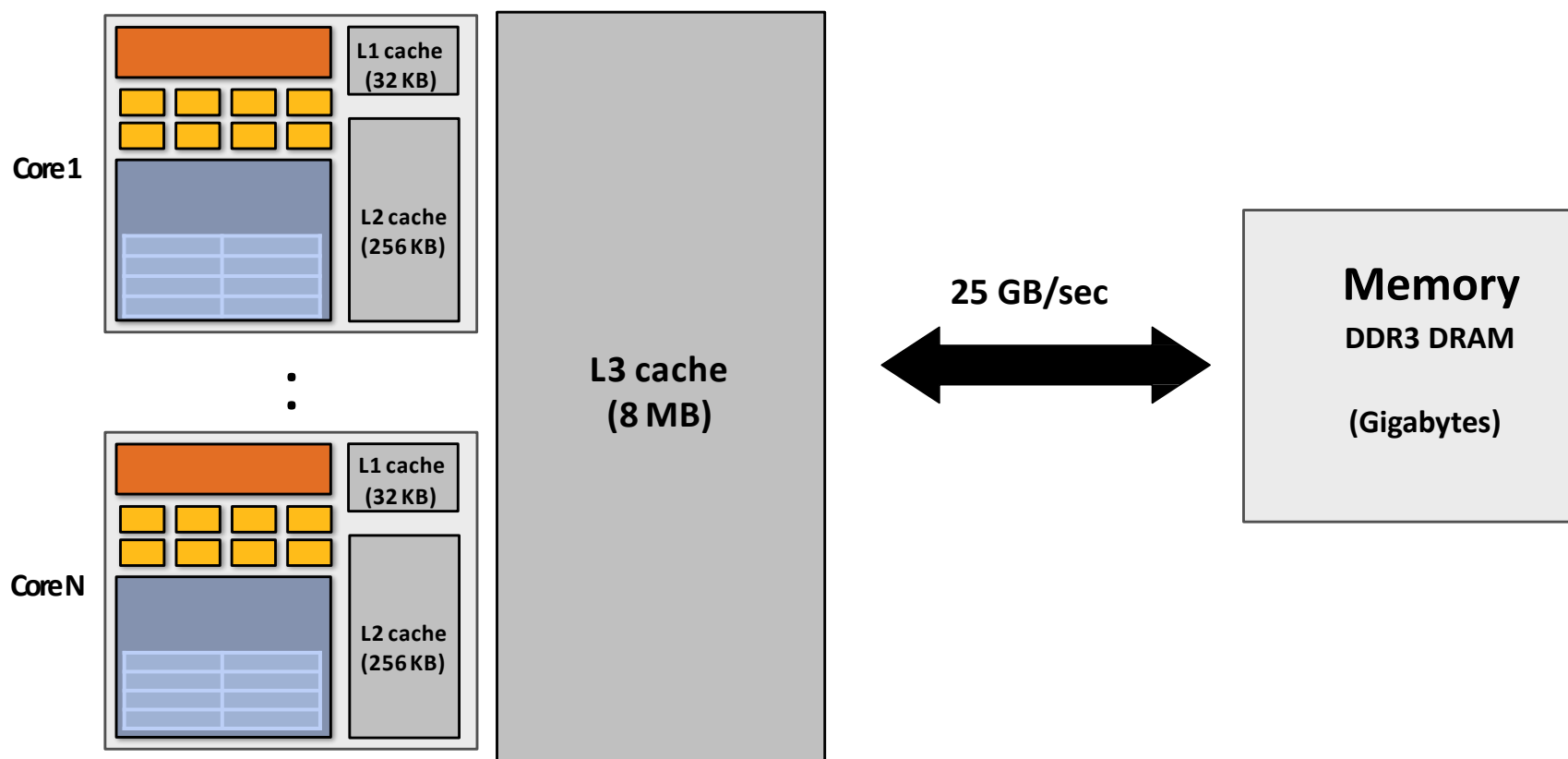
```
ld  r0  mem[r2]  
ld  r1  mem[r3]  
add r0, r0, r1
```



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

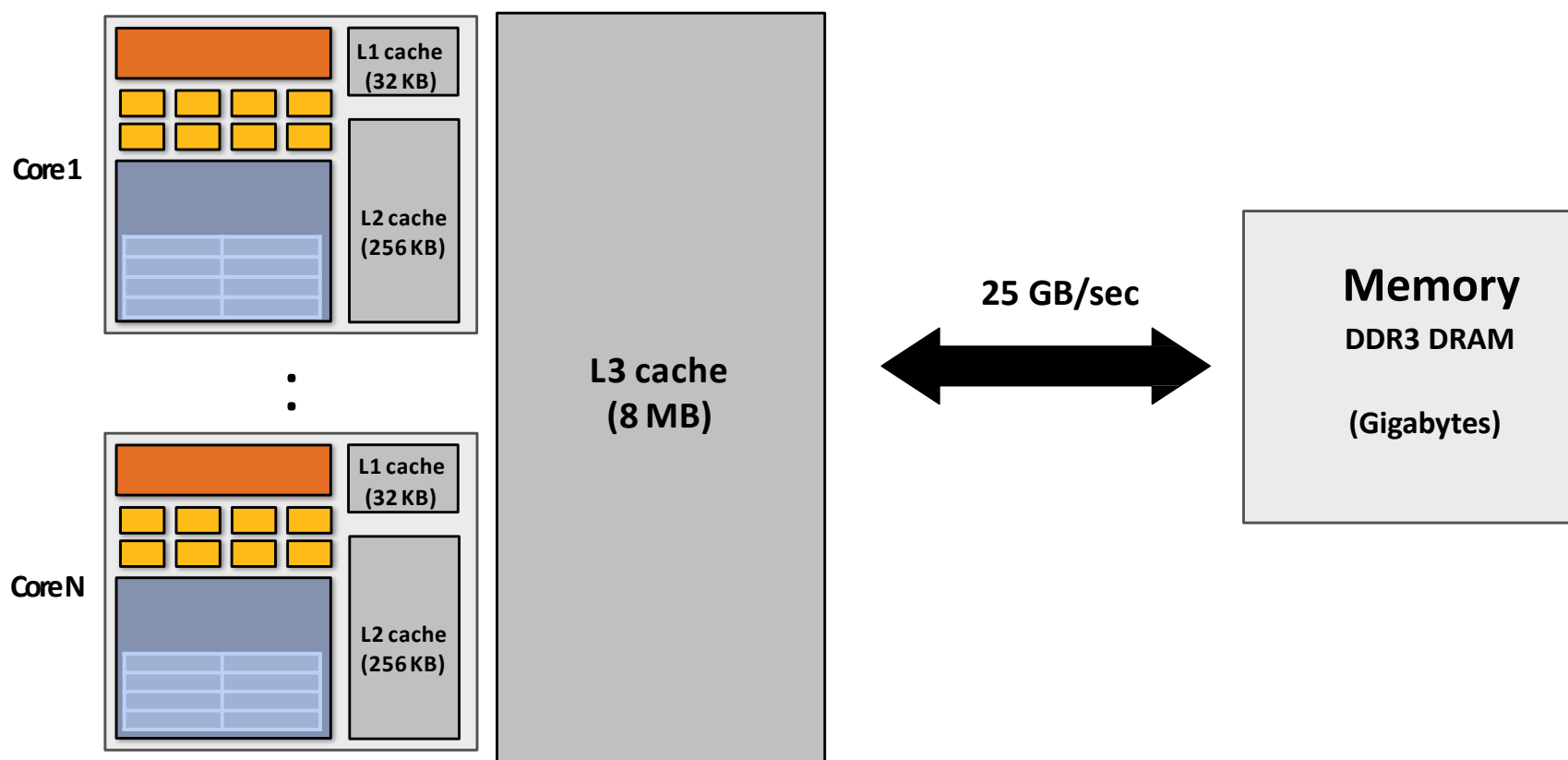
- 访存所需的时间（访存延迟） ~ 100's of cycles
- 计算所需的时间 ~ 1 of cycles

回顾：为什么现代处理器有缓存？



Caches能够减少处理器停顿的时间(reduce latency)

当数据驻留在高速缓存中时，可减少内存访问延迟，
从而使处理器更高效运行*



*缓存还能**为 CPU 提供更高的数据传输带宽**

通过预取减少处理器停顿(hides latency)

- 所有现代 **CPU** 都有将数据预取到缓存中的处理逻辑
 - 动态分析程序的访问模式，预测即将访问的内存区域
- 当**CPU**使用数据时，数据已经在高速缓存中，从而减少了停顿

predict value of r2, initiate load

predict value of r3, initiate load

...

...

...

...

...

...

ld r0 mem[r2]

ld r1 mem[r3]

add r0, r0, r1

data arrives in cache

data arrives in cache

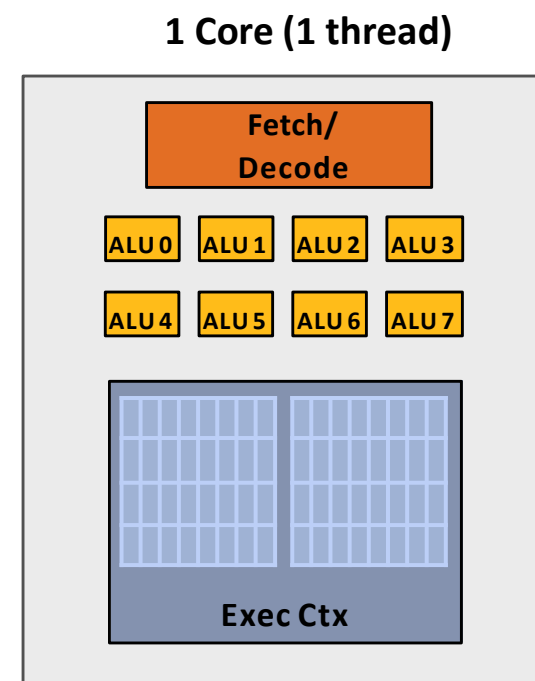
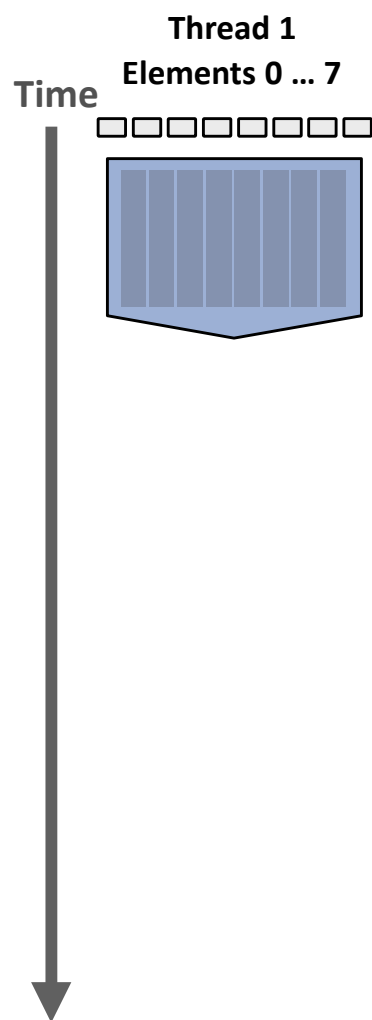
These loads are cache hits

Note:如果猜测错误（占用带宽、污染缓存），预取也会降低性能

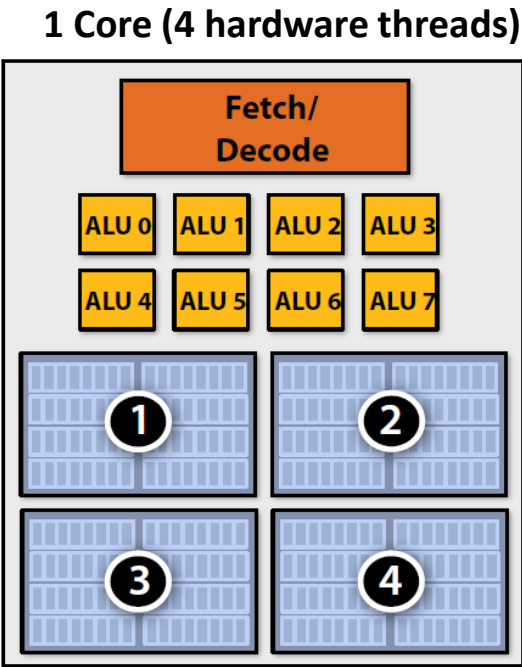
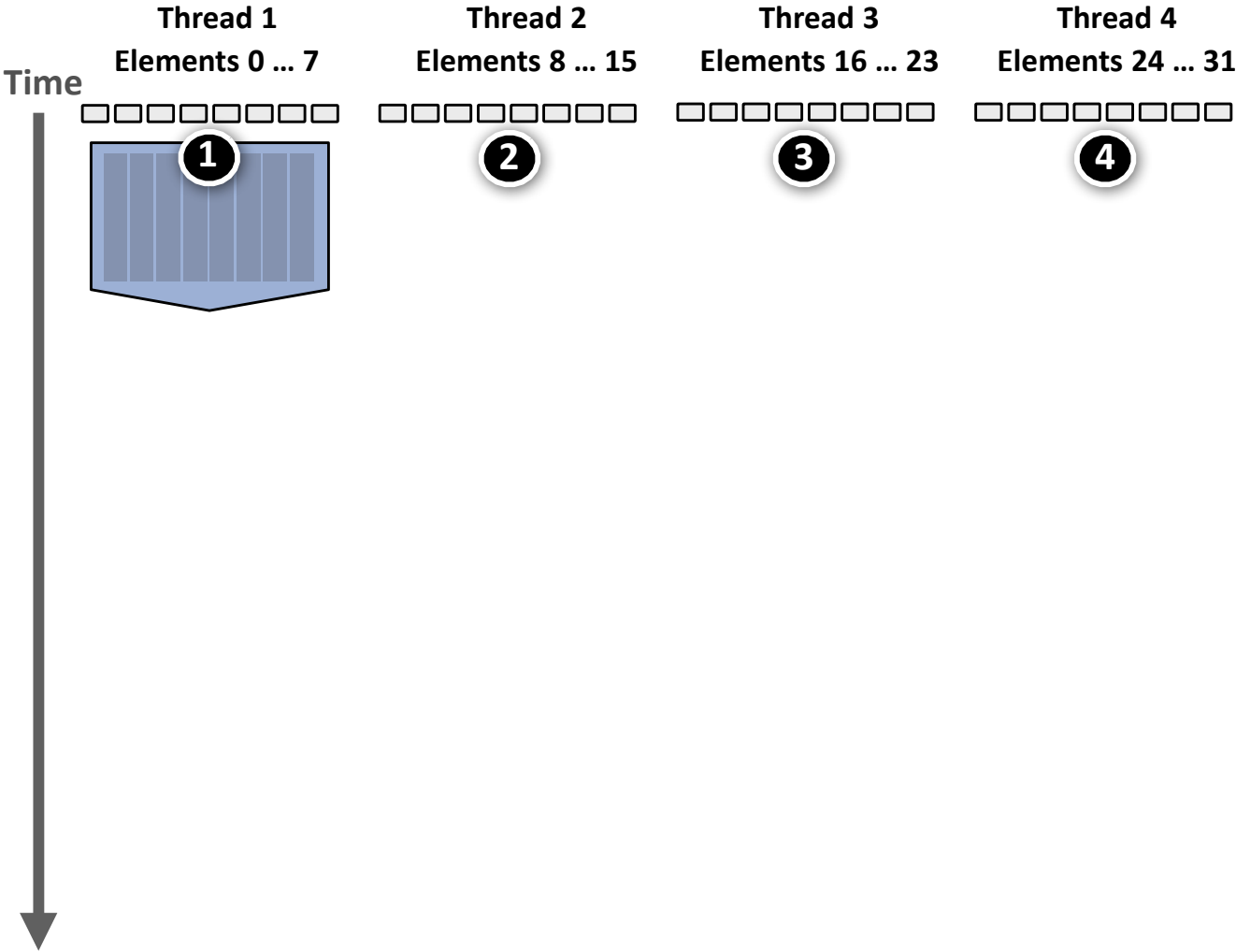
多线程减少CPU停顿

- **Idea:**在同一核心上交错处理多个线程以隐藏停顿
- 与预取一样，多线程是一种延迟隐藏，而不是延迟减少技术

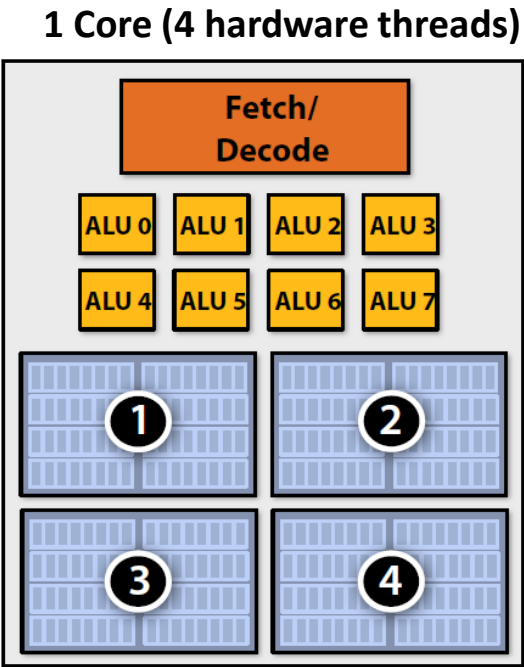
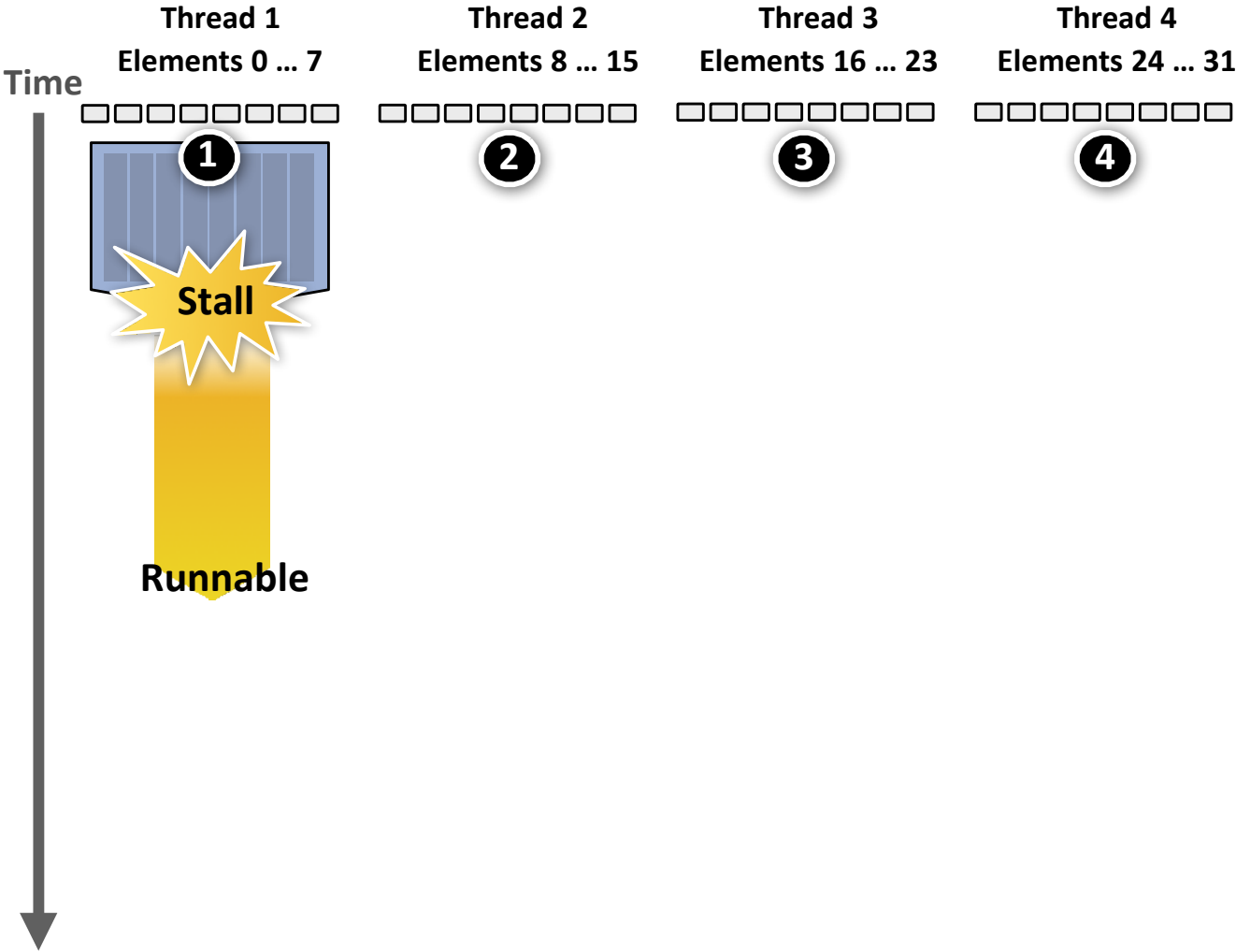
多线程减少CPU停顿



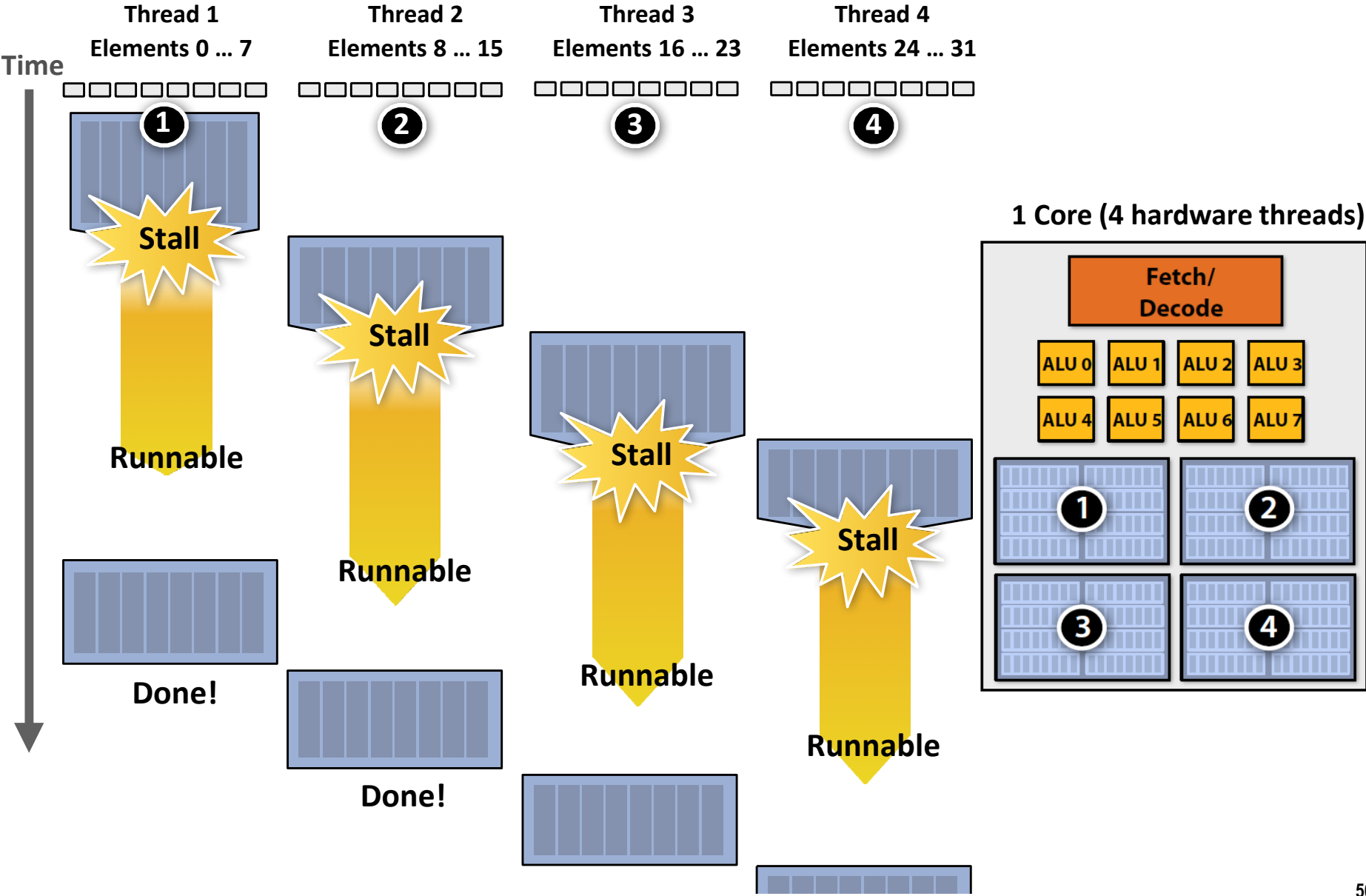
多线程减少CPU停顿



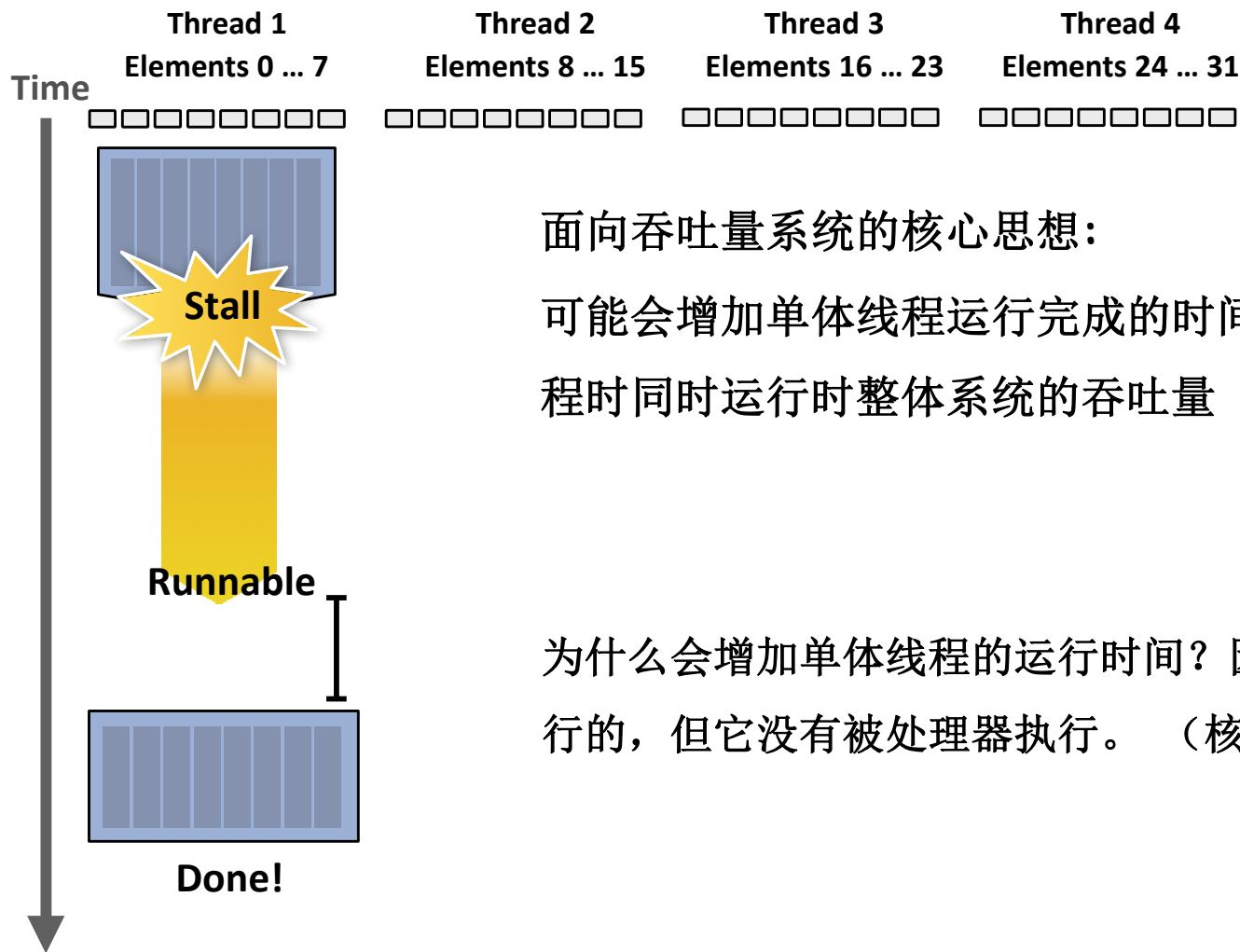
多线程减少CPU停顿



多线程减少CPU停顿



面向计算吞吐量的权衡



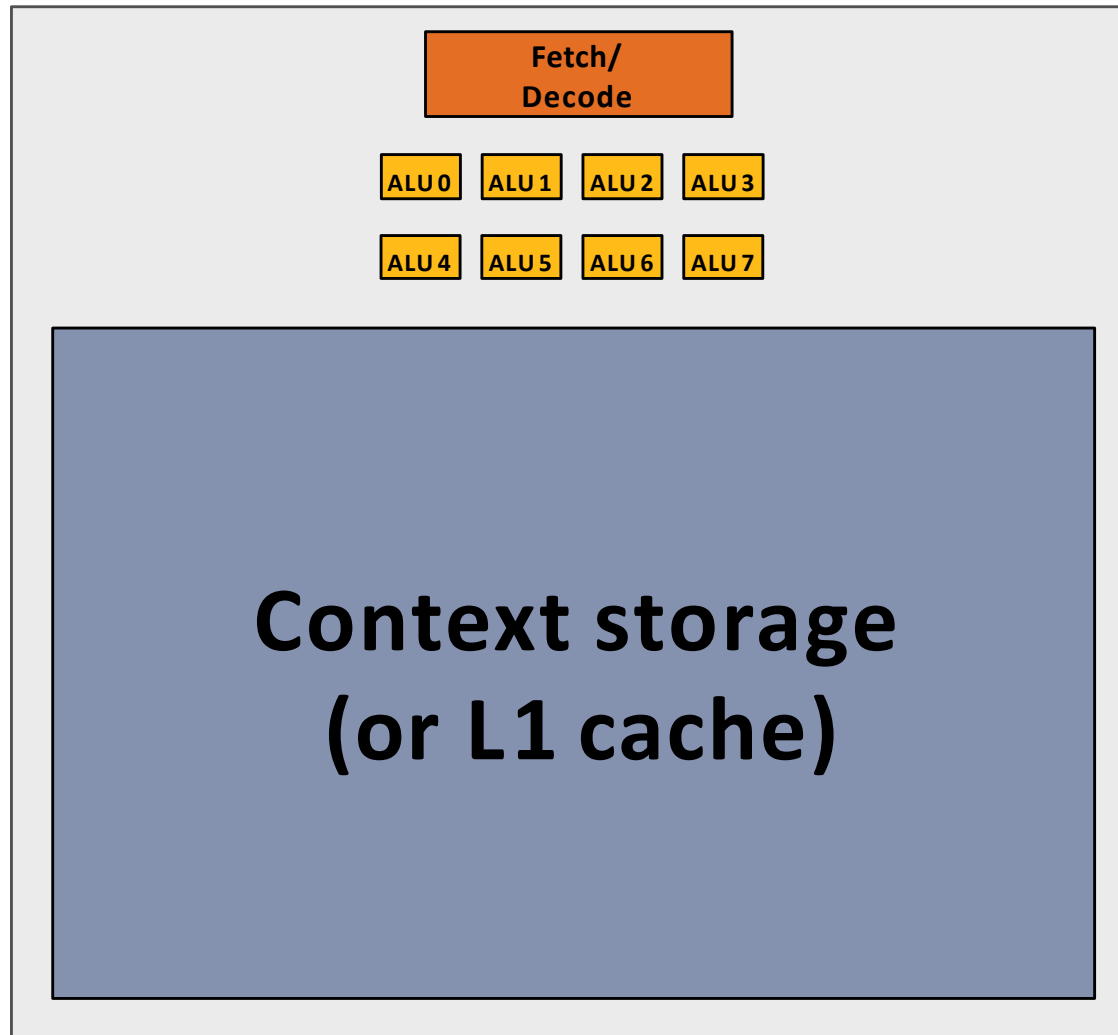
面向吞吐量系统的核心思想：

可能会增加单体线程运行完成的时间，但能够提升多个线程同时运行时整体系统的吞吐量

为什么会增加单体线程的运行时间？因为可能这个线程是可运行的，但它没有被处理器执行。（核心正在运行其他线程。）

Storing execution contexts

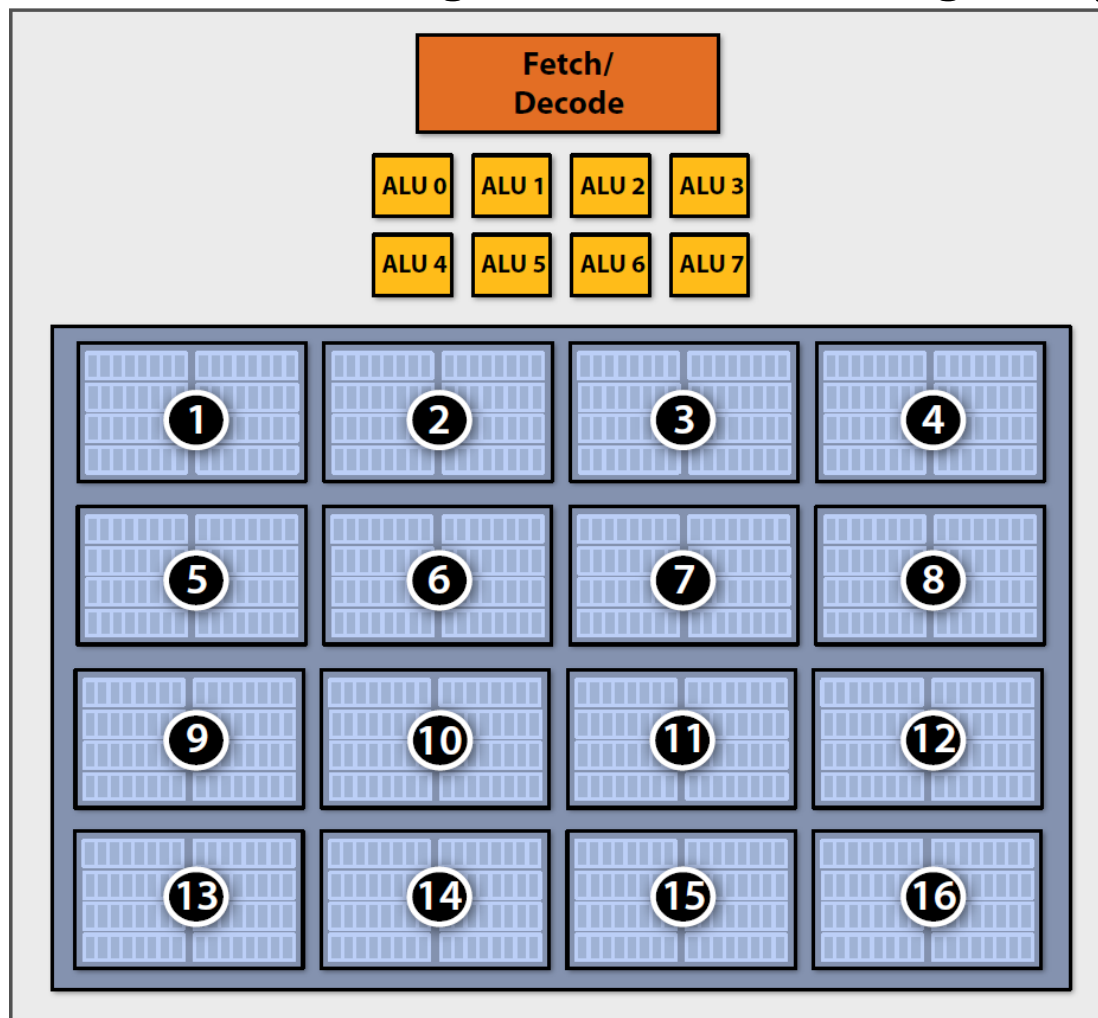
Consider on ship storage of execution contexts a finite resource.



Many small contexts (high latency hiding ability)

1 core

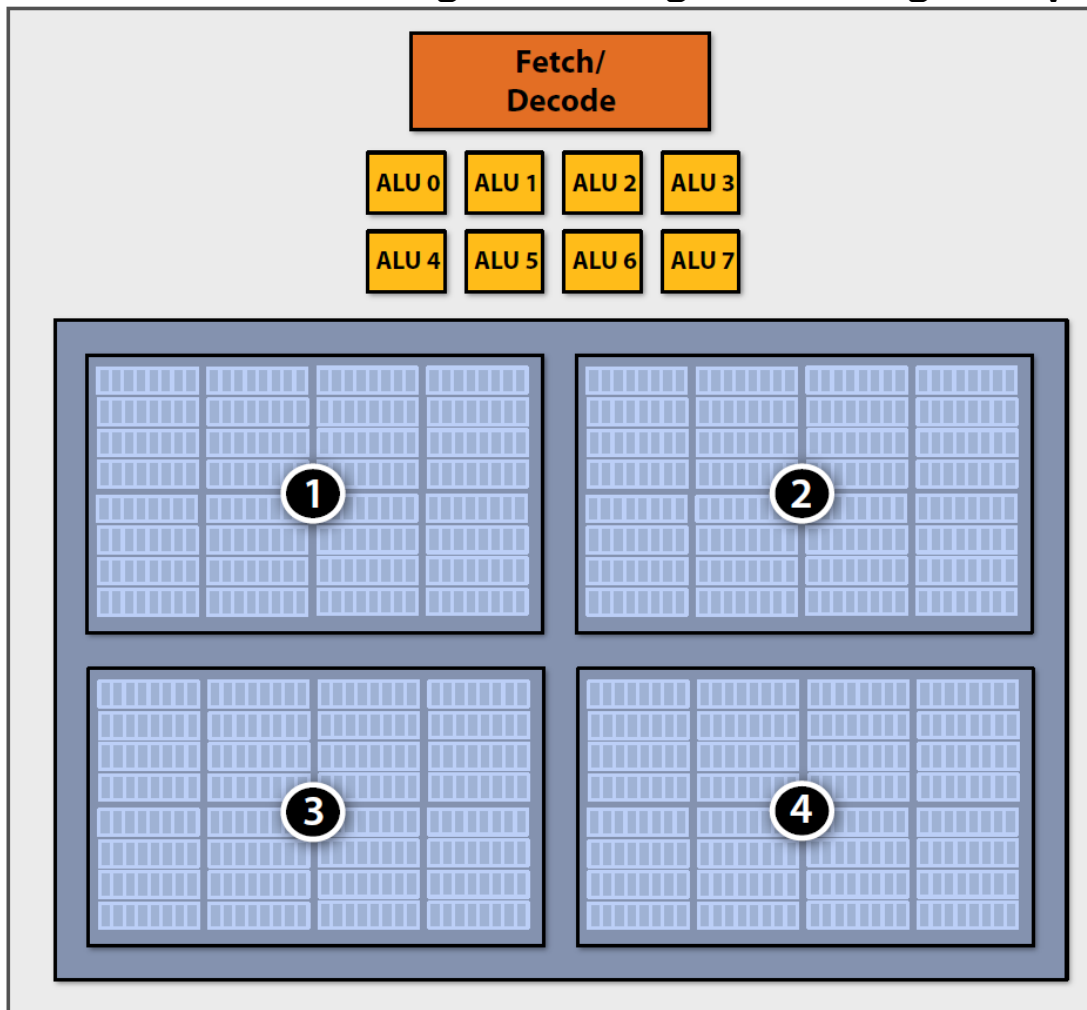
(16 hardware threads, storage for small working set per thread)



Four large contexts (low latency hiding ability)

1 core

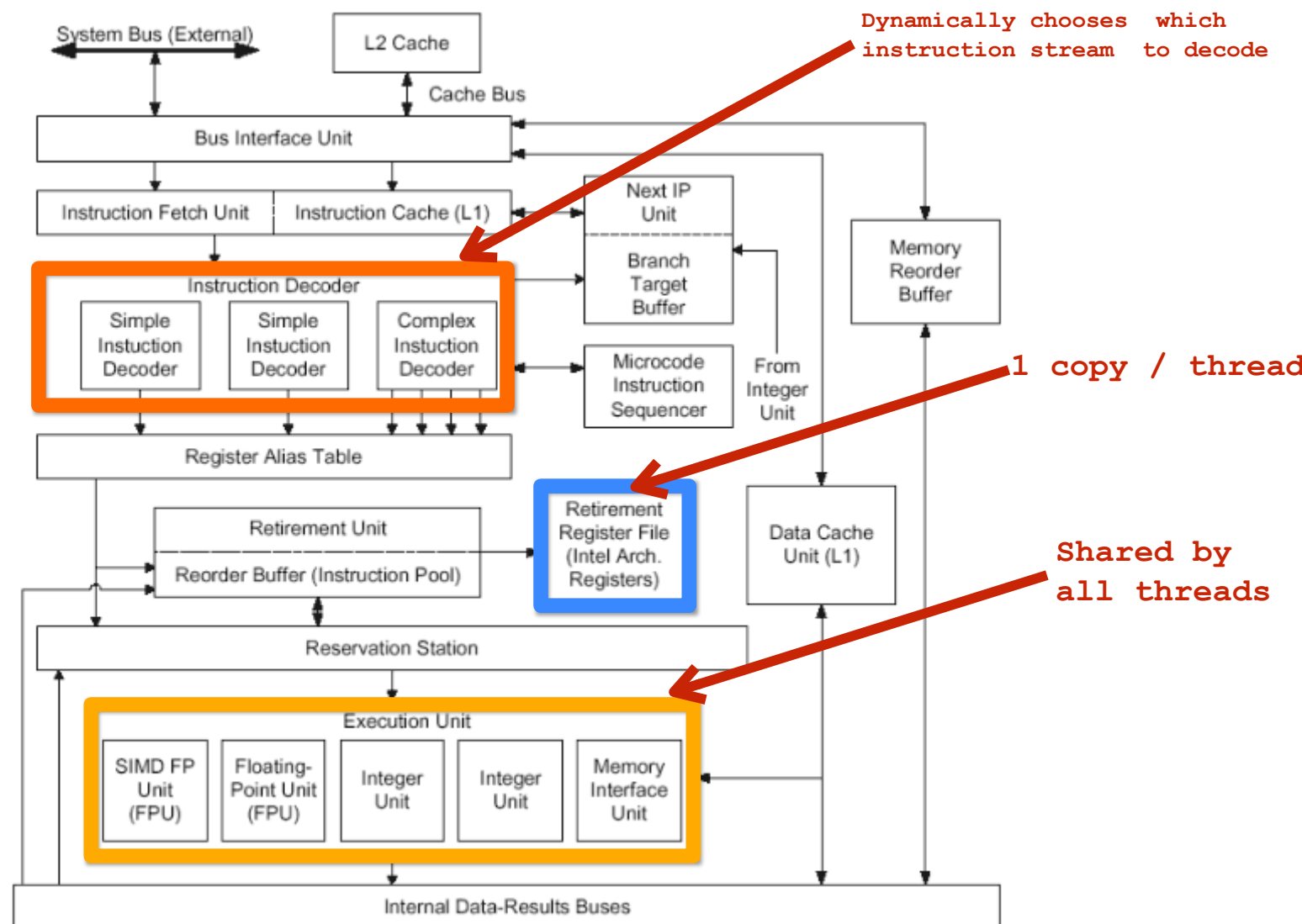
(4 hardware threads, storage for larger working set per thread)



Hardware-supported multi-threading

- **处理器核心管理多个线程的执行上下文**
 - 从可运行的线程运行指令（处理器决定每个时钟运行哪个线程，而不是操作系统）
 - 处理器核拥有的 **ALU** 资源没有变化：多线程仅在面对内存访问等高延迟操作时，能使用户更有效地使用**ALU**资源
- **交错的多线程 Interleaved multi-threading (a.k.a. temporal multi-threading)**
 - 之前的幻灯片所描述的：每个时钟，处理器核选择一个线程，并在 **ALU** 上的运行该线程的一条指令
- **同时多线程 Simultaneous multi-threading (SMT)**
 - 每个时钟周期，处理器核允许多个线程的指令同时执行在其**ALU**上
 - 是超标量处理器设计的一种扩展
 - 例如: **Intel** 超线程技术 (2 threads per core)

同时多线程的硬件实现



多线程总结

■ 收益：更有效地使用内核的 **ALU** 资源

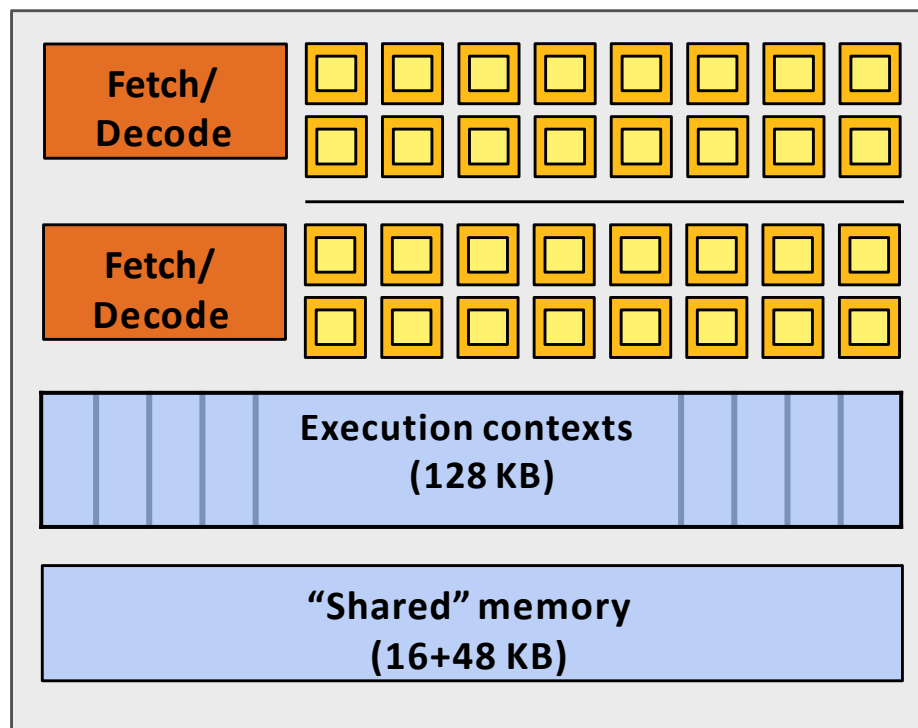
- 隐藏访存延迟
- 填充超标量架构的多个功能单元（当一个线程的 **ILP** 不足时）

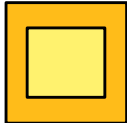
■ 代价

- 需要额外的线程上下文存储
- 增加任何单体线程的运行时间
(通常不是问题，我们通常关心并行应用程序的吞吐量)
- 严重依赖于内存带宽
 - 更多的线程 → 更大的工作数据集 → 每线程分到的 **cache** 空间更小
 - 所以有可能会需要经常去内存访问数据, 但也可以通过预取隐藏访存延迟

GPUs: Extreme throughput-oriented processors

NVIDIA GTX 480 core



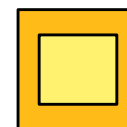
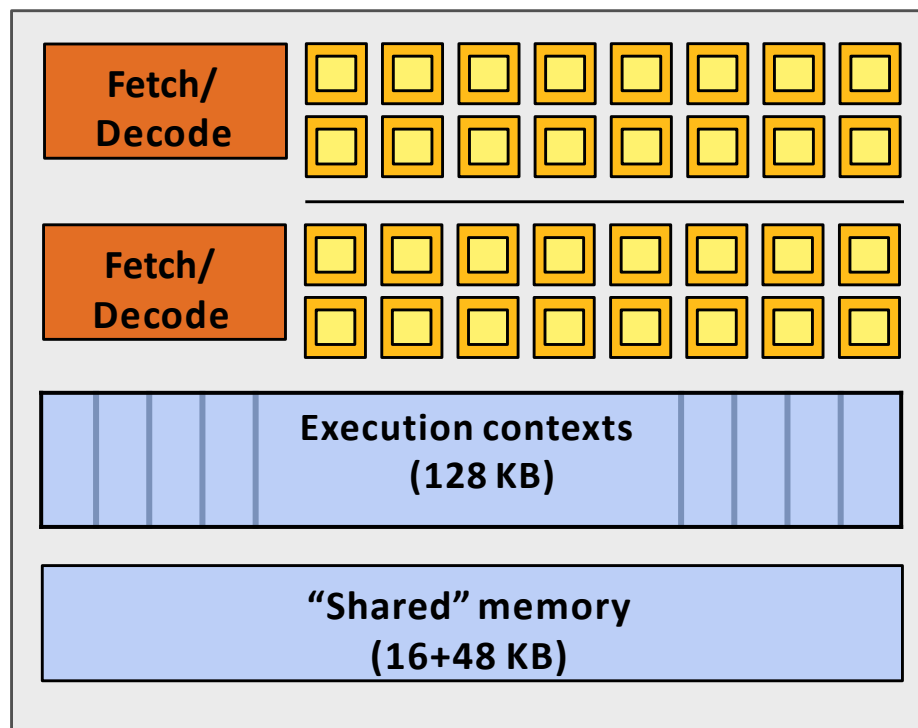
 = SIMD 功能单元，一次处理 16 个数据
(1 MUL-ADD per clock)

- 指令操作32条数据(called "warps").
- warp = 一个线程每次发出 32 宽度的向量指令
- 最多可以48 warps (线程) 同时执行
- $48 \times 32 = 1536$ 个元素可以并行处理

Source: Fermi Compute Architecture Whitepaper CUDA
Programming Guide 3.1, Appendix G

NVIDIA GTX 480: more detail (just for the curious)

NVIDIA GTX 480 core



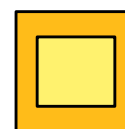
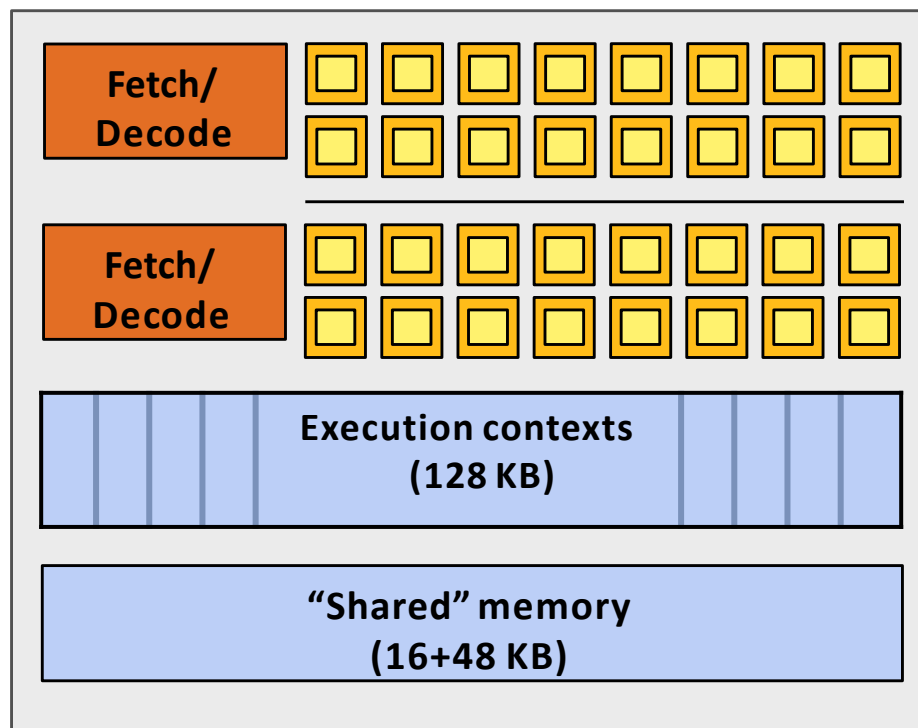
= SIMD 功能单元，一次处理 16 个数据
(1 MUL-ADD per clock)

- 为什么每个 warp 每个时钟周期处理 32 个数据，而这里使用的是 16 的 SIMD ALUs?
- 这有点复杂：**ALU** 的运行时钟速率是芯片其余部分的两倍。因此，每条解码指令在 16 SIMD ALU 运行两个时钟就可以处理 32 条数据。（但对程序员来说，它的行为就像一个 32 位宽的 SIMD 操作）

Source: Fermi Compute Architecture Whitepaper CUDA
Programming Guide 3.1, Appendix G

NVIDIA GTX 480: more detail (just for the curious)

NVIDIA GTX 480 core

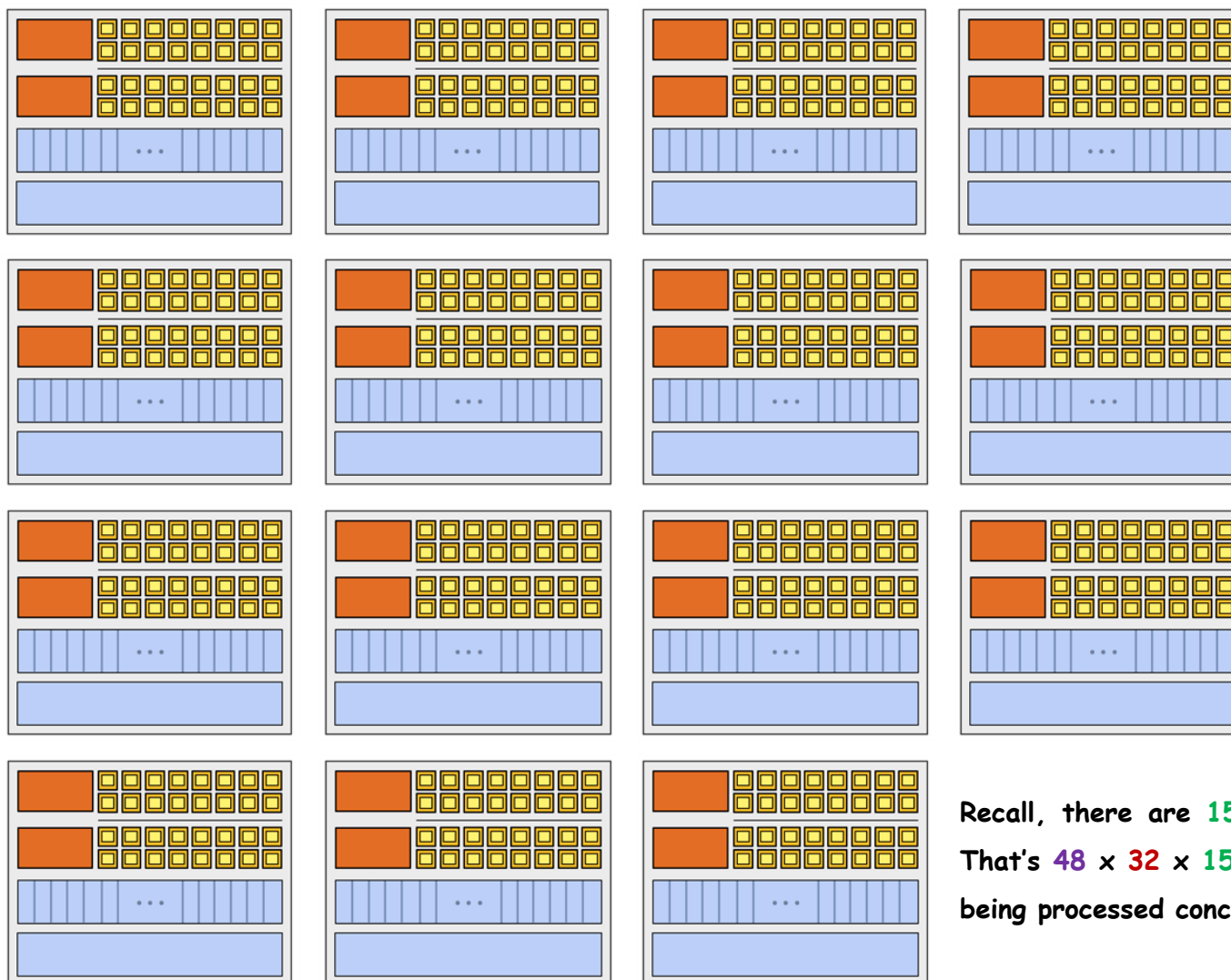


= SIMD 功能单元，一次处理 16 个数据(1 MUL-ADD per clock)

- 每个核当中还有一组 16个ALU
- 整个芯片有 15 cores \times 32 = 480 ALU

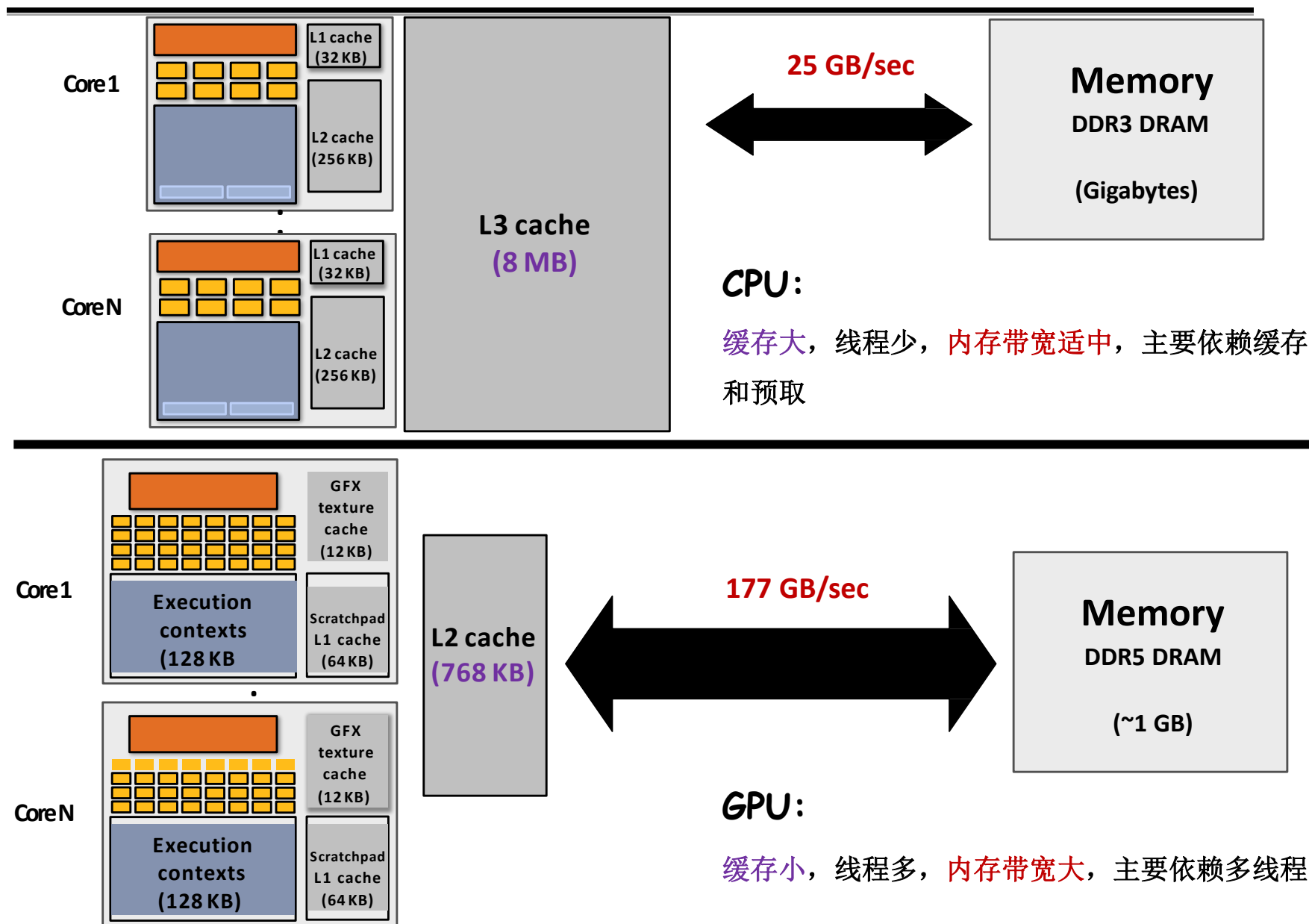
Source: Fermi Compute Architecture Whitepaper CUDA Programming Guide 3.1, Appendix G

NVIDIA GTX 480



Recall, there are 15 cores on the GTX 480:
That's $48 \times 32 \times 15 = 23,040$ pieces of data
being processed concurrently!

CPU vs. GPU memory hierarchies

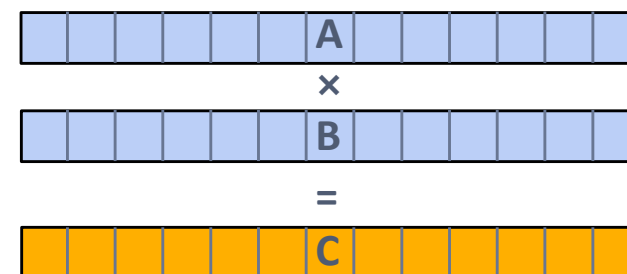


思想实验

Task: 两个向量 **A** 和 **B** 的逐元素乘法

假设向量包含数百万个元素

- Load input **A[i]**
- Load input **B[i]**
- Compute $A[i] \times B[i]$
- Store result into **C[i]**



Three memory operations (12 bytes) for every MUL

NVIDIA GTX 480 GPU can do 23,040 MUL per clock
(@1.2 GHz)

Need ~**6.4 TB/sec** of bandwidth to keep functional units **busy** (only have **177 GB/sec**)

~ **3% efficiency**... but **7x faster** than quad-core CPU !

(2.6 GHz Core i7 Gen 4 quad-core CPU connected to 25 GB/sec memory bus will exhibit similar efficiency on this computation)

Bandwidth limited!

如果处理器以过高的速率请求数据，则内存系统无法跟上.

再多的延迟隐藏也无济于事.

克服带宽限制是应用程序开发人员在吞吐量优化系统上面临的共同挑战.

访存带宽是关键资源

高性能并行程序需要：

- 更好地组织计算以**减少**从内存中**获取数据的频率**
 - 重用先前由同一线程加载的数据(传统的线程内时间局部性优化)
 - 跨线程共享数据(线程间合作)
- **更少**地请求数据**总量**（相反，做更多的计算：它是“免费的”）
 - 有用的术语：“**运算强度，AI**”——指令流中数学运算与数据访问运算的比率
 - 要点：程序必须具有高运算强度才能有效地利用现代处理器

总结

- 第一部分：所有现代处理器都不同程度采用的**三种并行设计**
 - ① 使用多个处理核心
 - **Simpler cores** (不断地增强线程级并行性)
 - ② 在多个 **ALU** 上分摊指令流处理(**SIMD**)
 - 以很少的额外成本提高计算能力
 - ③ 使用多线程更有效地利用处理资源（隐藏延迟，填充所有可用资源）
- 第二部分：由于现代芯片的高算术能力，许多并行应用程序（在 **CPU** 和 **GPU** 上）都受到**访存带宽限制**
- **GPU** 架构使用与 **CPU** 相同的提高计算吞吐量的思想：但 **GPU** 将这些概念推向了极致