Data-Level Parallelism

From CPU to GPU

主讲人: 邓倩妮
上海交通大学

部分思路、图片参考 CMU15-418/618 课程

本节

- 总结：现代计算机体系结构的重要概念
  - 性能指标
    - 延迟
    - 吞吐量
  - 并行
    - 超标量（superscalar）：开发ILP
    - 并发多线程（SMT）、 多核（multi-core）：开发TLP
    - 单指令多数据（SIMD） ： 开发DLP
  - 存储器
    - 多线程切换：访存延迟的隐藏
    - 访存带宽/吞吐量
- 现代GPU：设计思想总结

论文：云计算系统相关
Profiling a warehouse-scale computer， I*SCA'15*

- Group 1
- 孙雪涵
- 周新博
- 朱晨旭
- 王晨
- 王华宇

上海交通大学PPT模板

2016年4月

- Group 2
- 朱煜烨
- 边逸翔
- 龚政
- 左思成
- 杨屿杰

标题页第二版以及应用实例（学术）

主讲人姓名

2016年4月

Svilen Kaven (Harvard), Juan Pablo Darago (Universidad de Buenos Aires), Kim Hazelwood (Yahoo Labs), Parthasarathy Ranganathan (Google), Tipp Moseley (Google), Gu-Yeon Wei (Harvard), David Brooks (Harvard),
Profiling a warehouse-scale computer， I*SCA'15*

性能：两个常用的指标

执行时间（Execution Time）

- 又称为：响应时间（Response Time)
- 或：延迟(Latency)
- 处理器：完成某任务的总时间

吞吐量（Throughput)

- 又称为：带宽（bandwidth)
- 处理器：单位时间内完成的任务数量

性能：术语

- 访存延迟
  - 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
- 存储器带宽
  - The rate at which the memory system can provide data to a processor
  - Example: 20 GB/s

Part 1： 并行

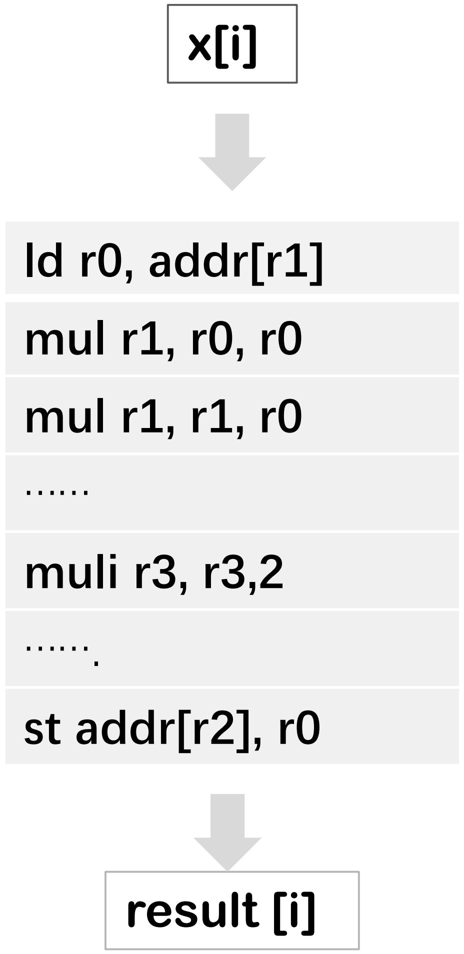
开发各类并行性：ILP、TLP、 DLP

例如： 泰勒级数展开计算 $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

```
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：

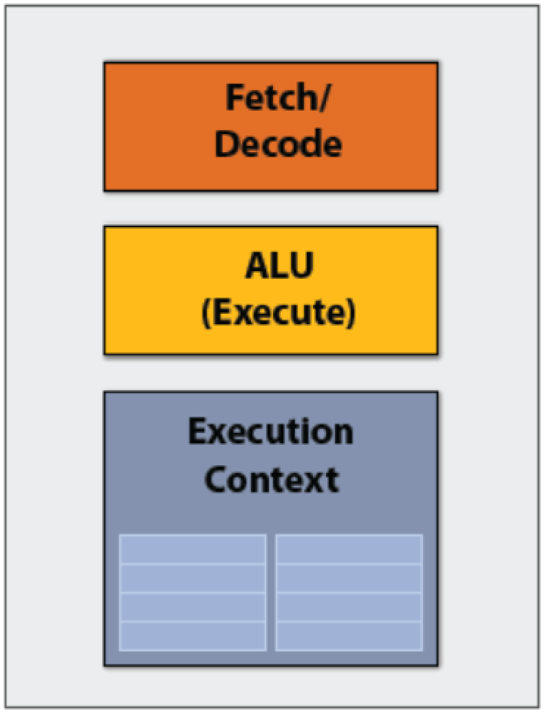
```
void sinx(int N, int terms, float* x, float* result)
{
  for (int i=0; i<N; i++)
  {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;       // 3!
      int sign = -1;

      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;
  }
}
```

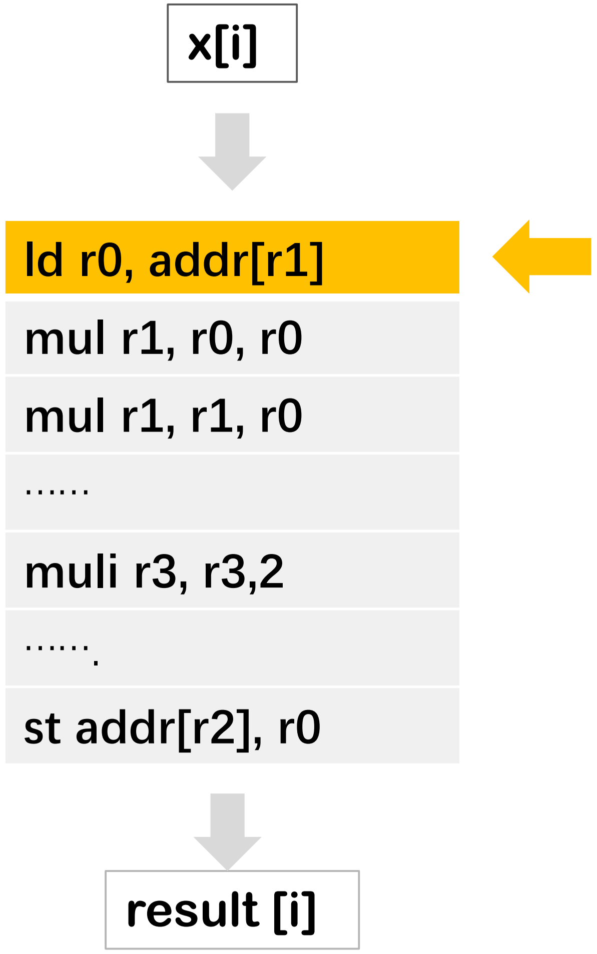


```
x[i]

ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
……
muli r3, r3,2
…….
st addr[r2], r0

result [i]
```

执行程序

简单的流水线处理器,
每周期执行一条指令：

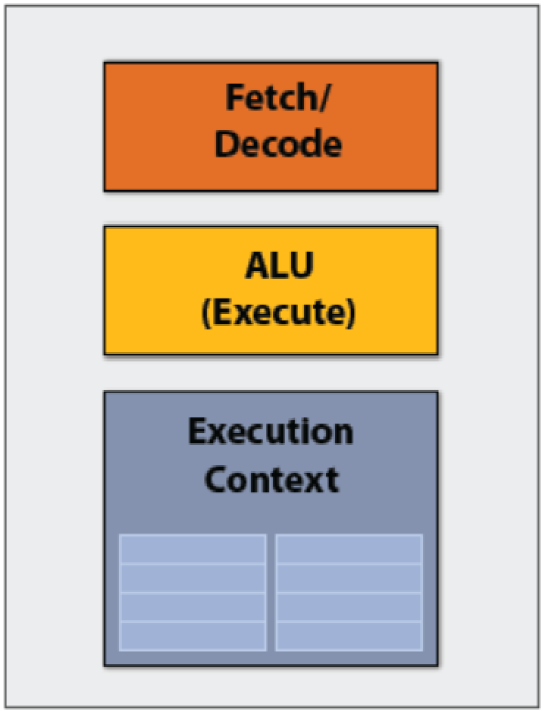


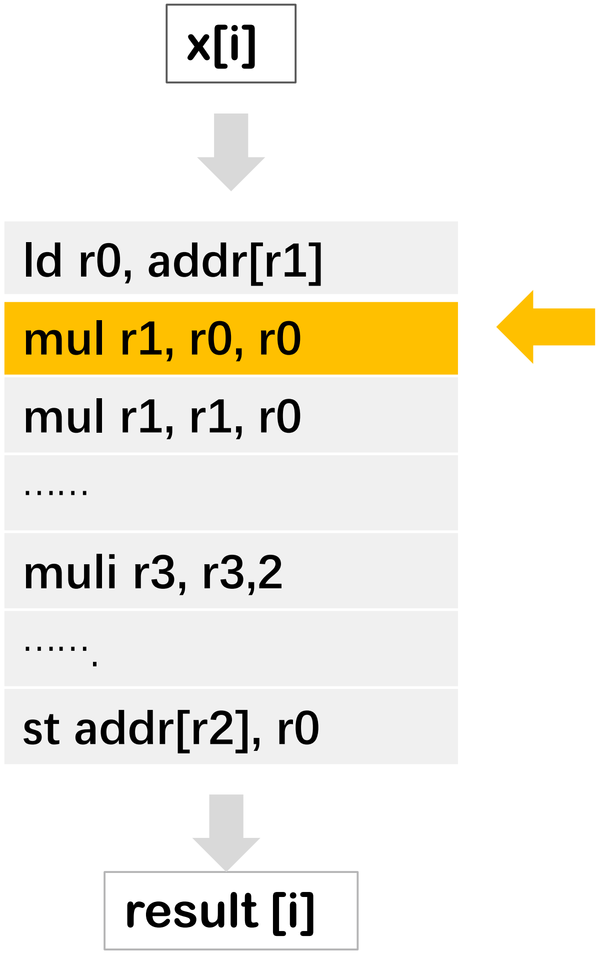


```
ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
......
muli r3, r3,2
.......
st addr[r2], r0
```

执行程序

简单的流水线处理器，
每周期执行一条指令：

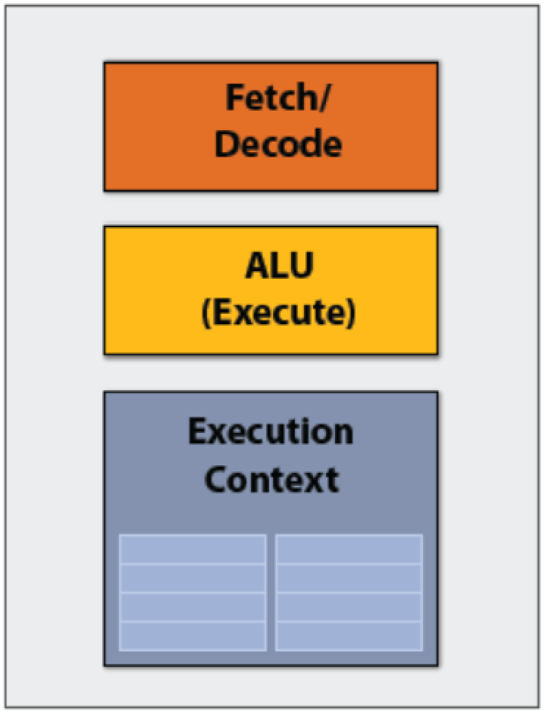


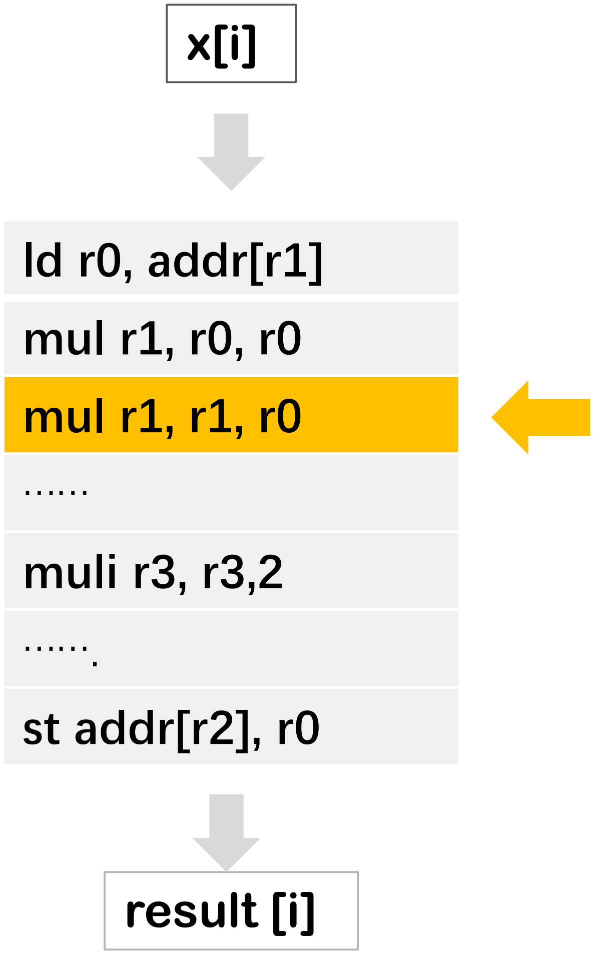


```
ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
......
muli r3, r3,2
.......
st addr[r2], r0
```

执行程序

简单的流水线处理器,
每周期执行一条指令：

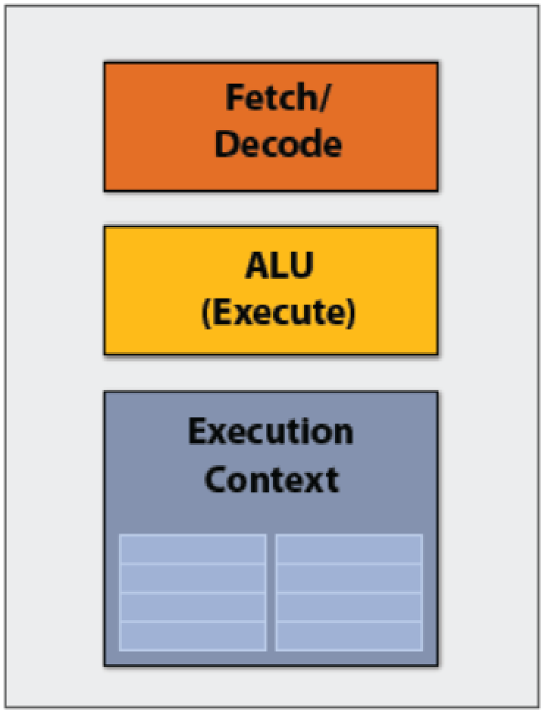


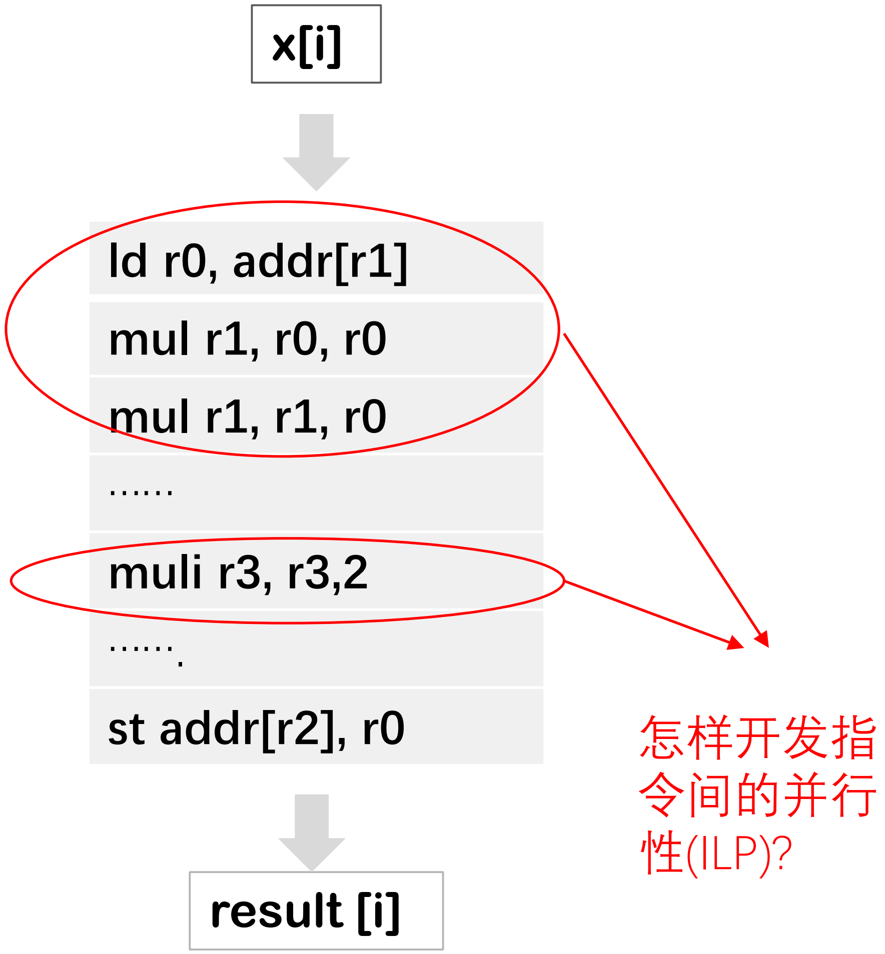


```
ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
……
muli r3, r3,2
…….
st addr[r2], r0
```

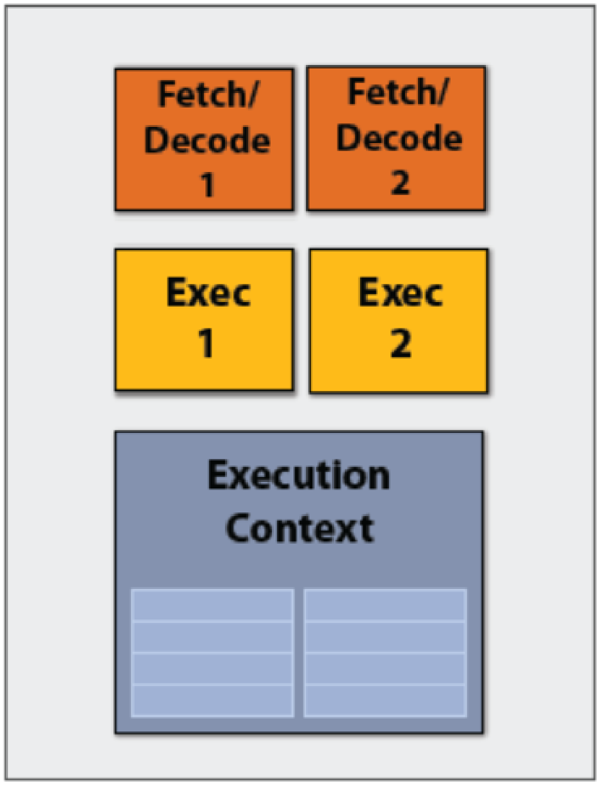
执行程序

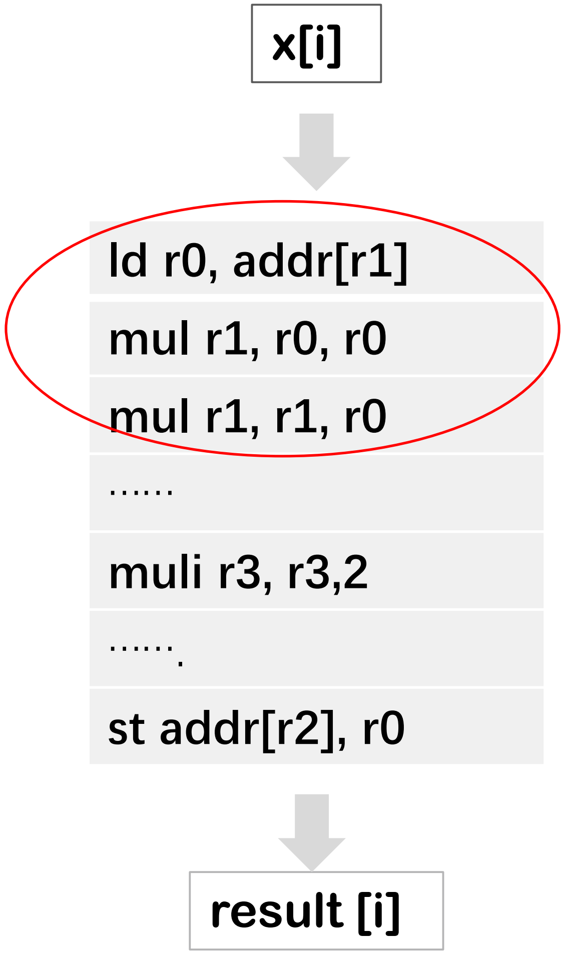
简单的流水线处理器，
每周期执行一条指令：





超标量处理器（superscalar）

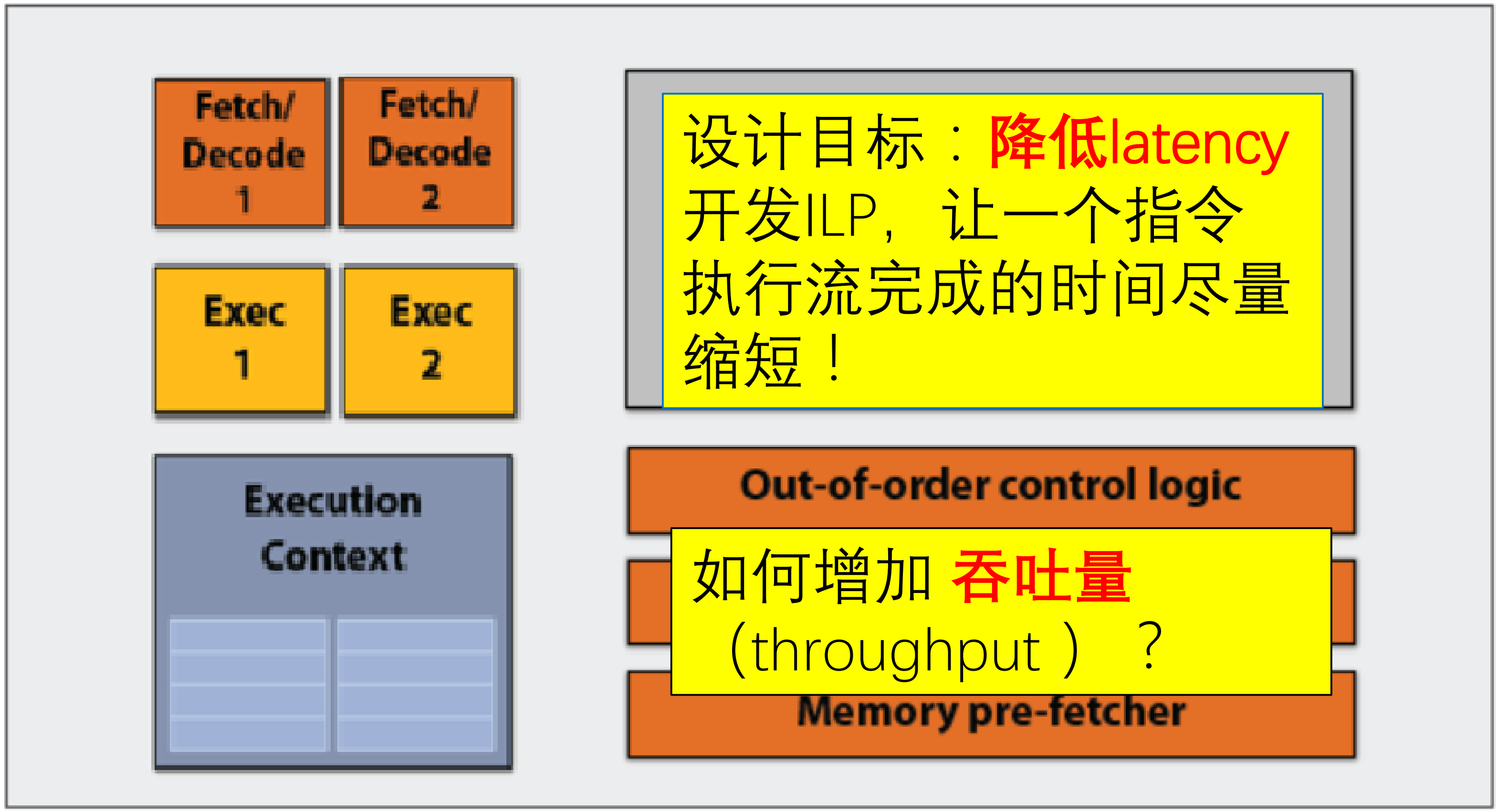




```
ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
……
muli r3, r3,2
…….
st addr[r2], r0
```

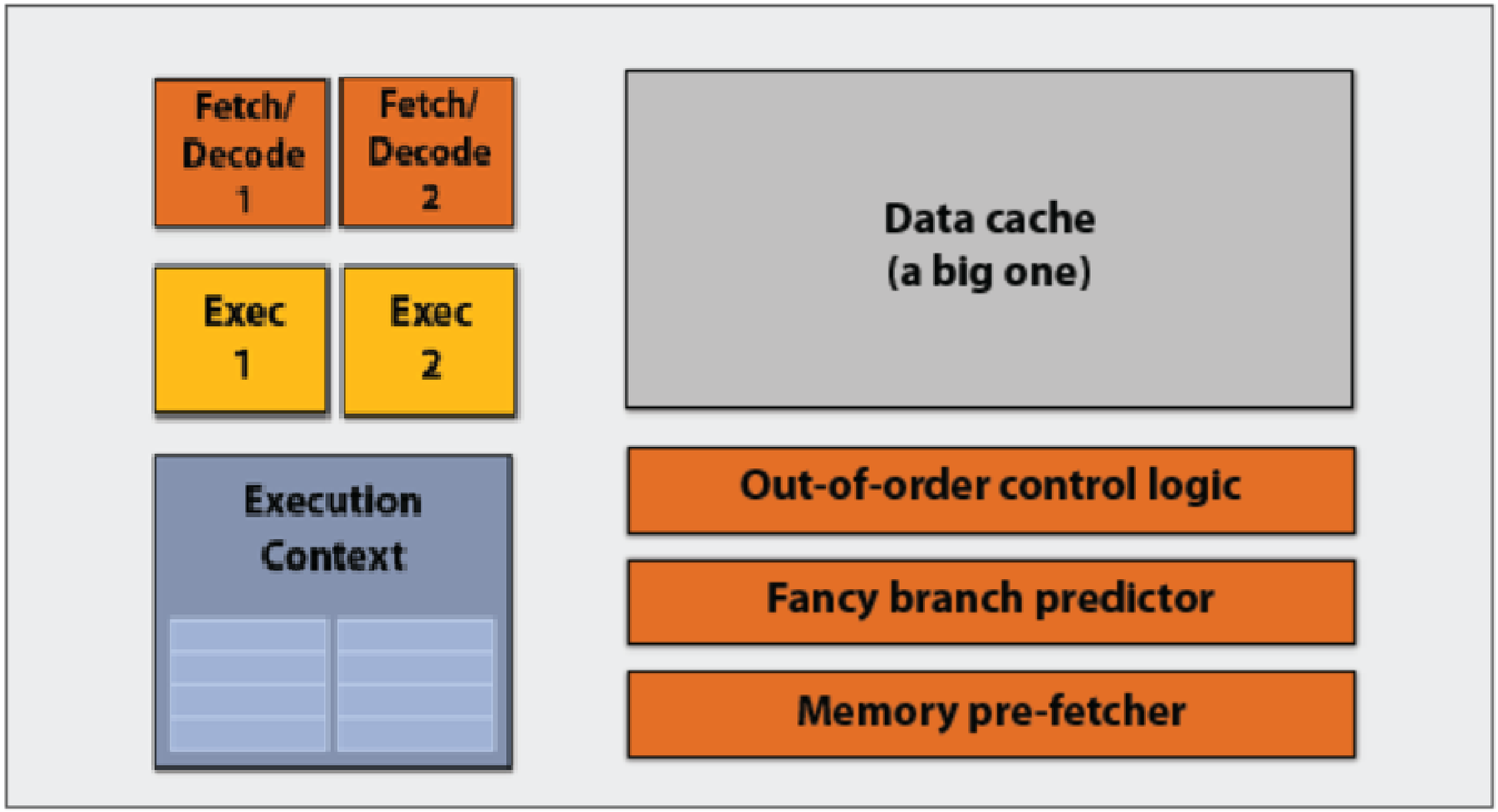
这段代码中没有指令间的并行性(ILP)

乱序超标量（out of order superscalar）— pre multi-core era


Fetch/ Decode 1
Fetch/ Decode 2
Exec 1
Exec 2
Execution Context
设计目标：降低latency
开发ILP，让一个指令执行流完成的时间尽量缩短！
Out-of-order control logic
如何增加 吞吐量（throughput）？
Memory pre-fetcher

More transistors = larger cache, smarter out-of-order logic, smarter branch predictor, etc. (Also: more transistors → smaller transistors → higher clock frequencies)

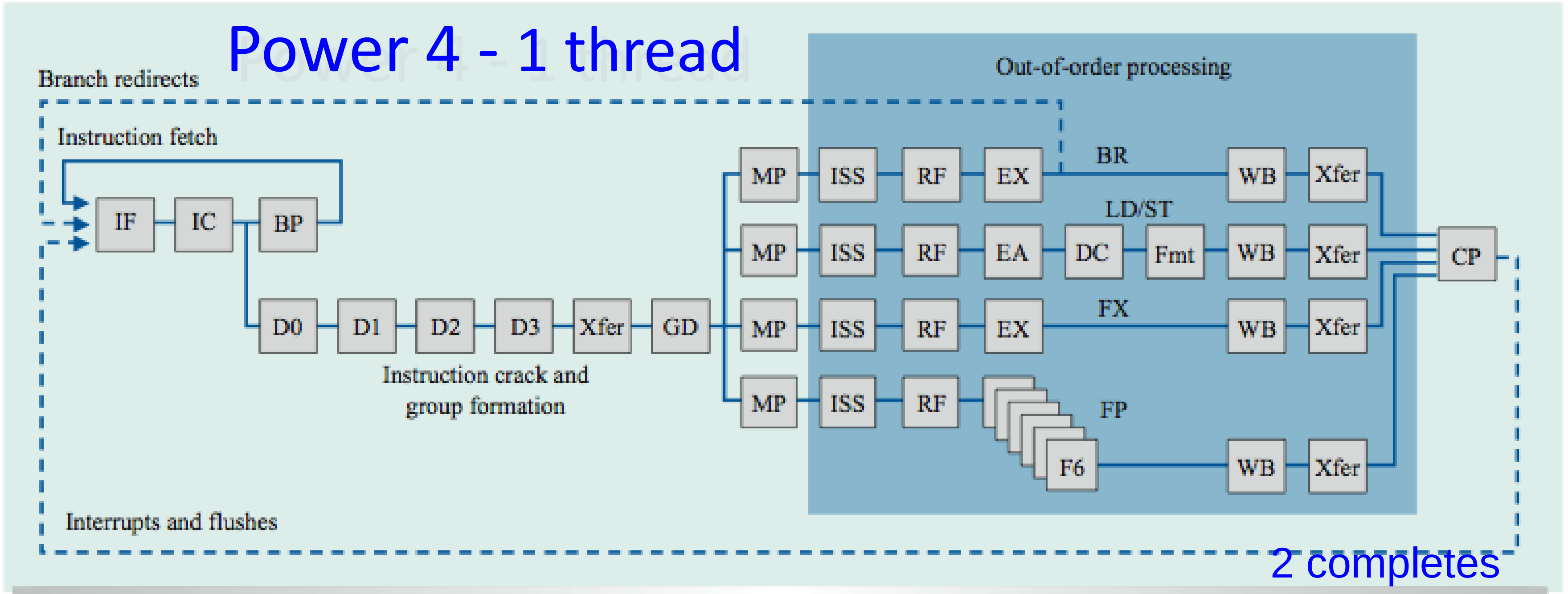
乱序超标量（out of order superscalar）— pre multi-core era



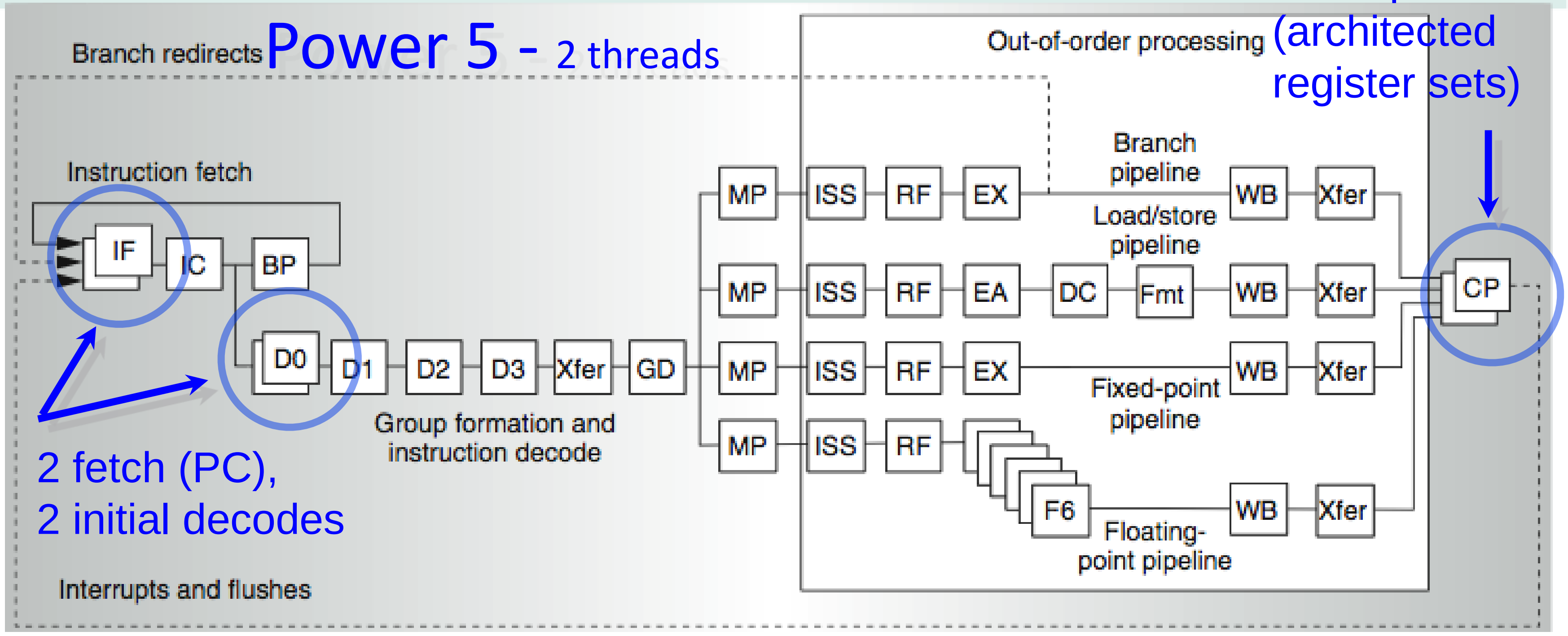
Idea #1 **：超线程技术（硬件多线程）**

多个线程同时在一个处理器上执行，提高执行部件的利用率，提高吞吐量

Power 4 - 1 thread

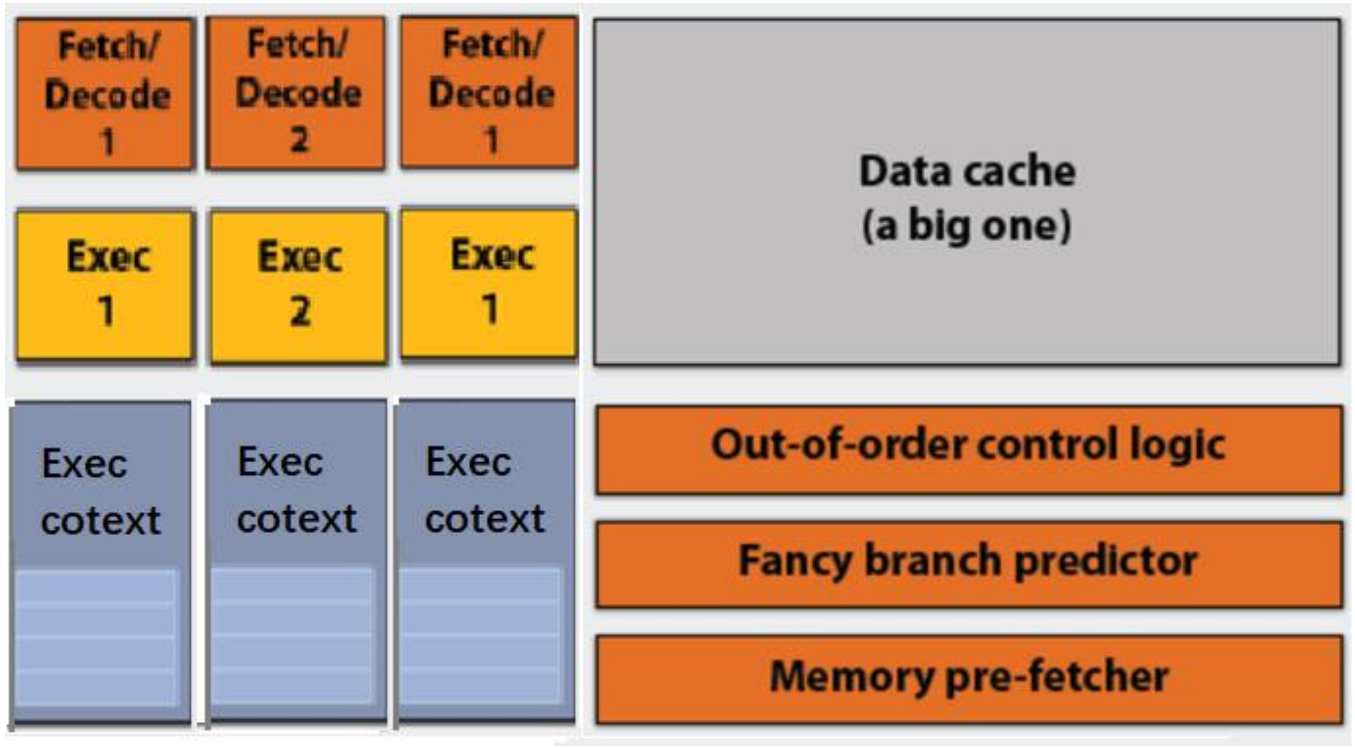


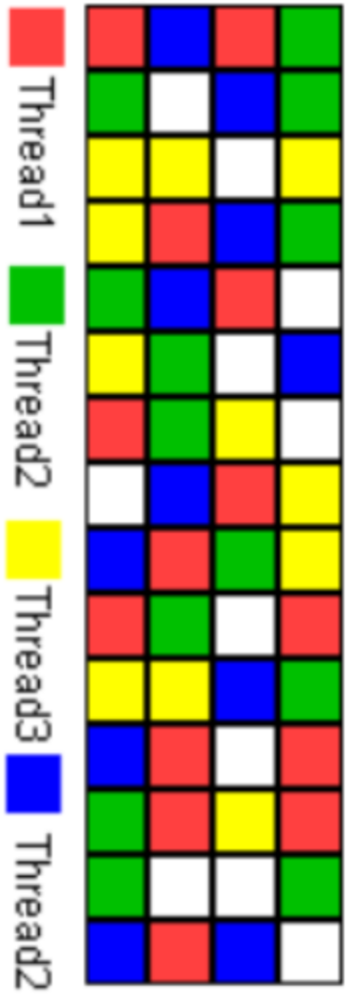
Power 5 - 2 threads



See www.ibm.com/servers/eserver/pseries/news/related/2004/m2040.pdf

并发多线程（SMT）

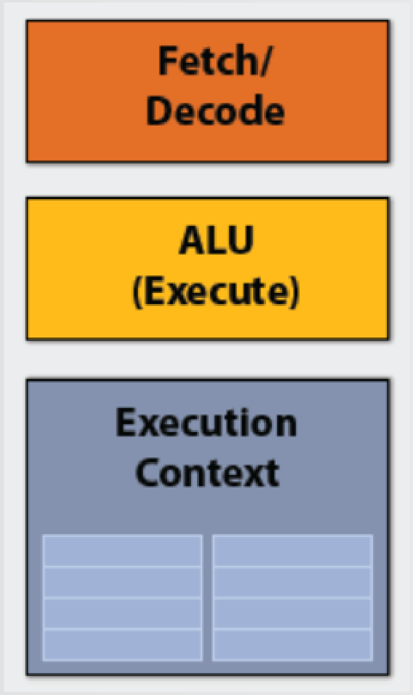




Simultaneous Multithreading： 49/60= 81.67%

提高吞吐量、但会影响单个任务的执行时间

处理器 — multi-core era



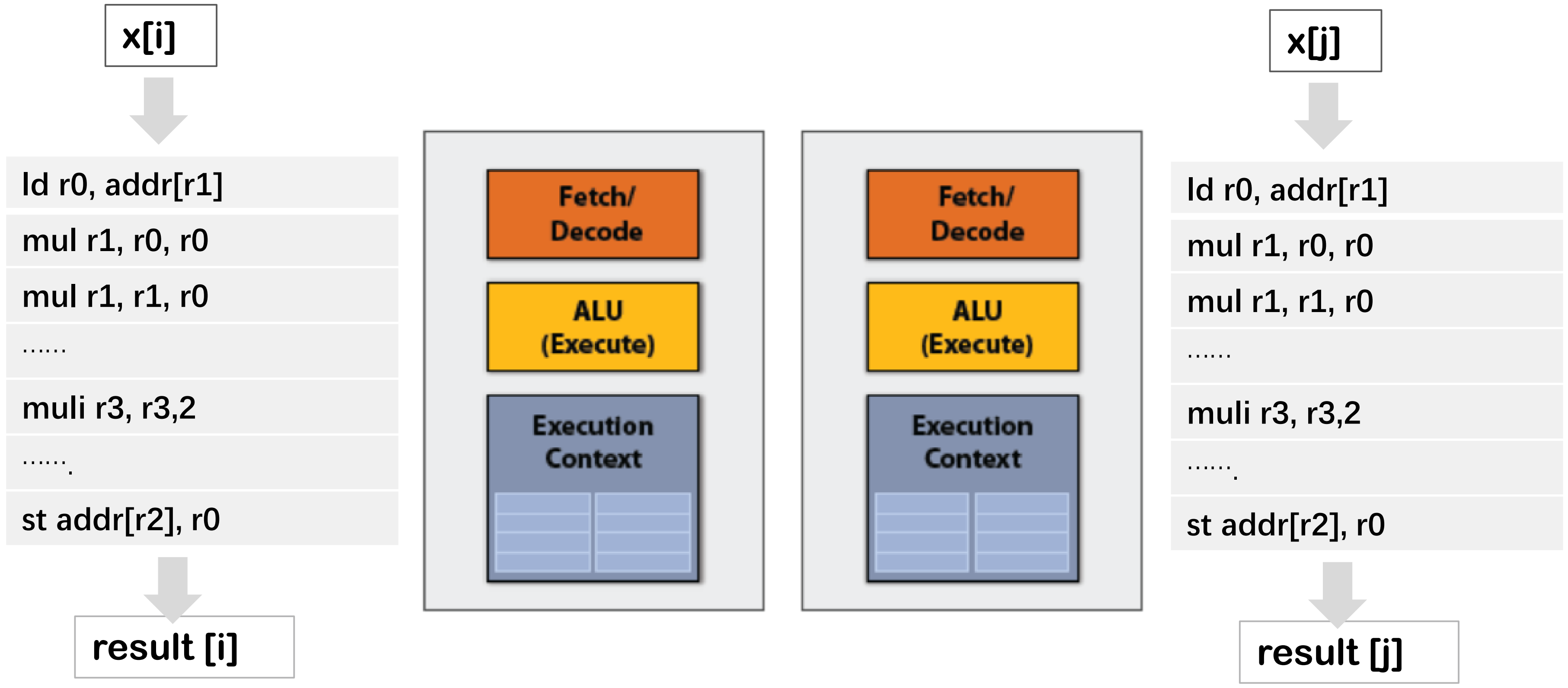
如何支持更多的并发多线程？

Idea #2:
加入更多的核心数到处理器中。

处理器中的晶体管，与其用于复杂的控制逻辑、转移预测、乱序执行、猜测执行，以降低单个指令流的延迟；

不如用来增加并行计算能力、以提高系统整体的吞吐量。

双核处理器：compute two elements in parallel



```
ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
......
muli r3, r3,2
.......
st addr[r2], r0
```

```
ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
......
muli r3, r3,2
.......
st addr[r2], r0
```

简单的核，比复杂的核慢25%
双核的整体性能： 2 × 0.75 = 1.5 (potential for speedup!)
通过开发TLP， 提升整体的吞吐量

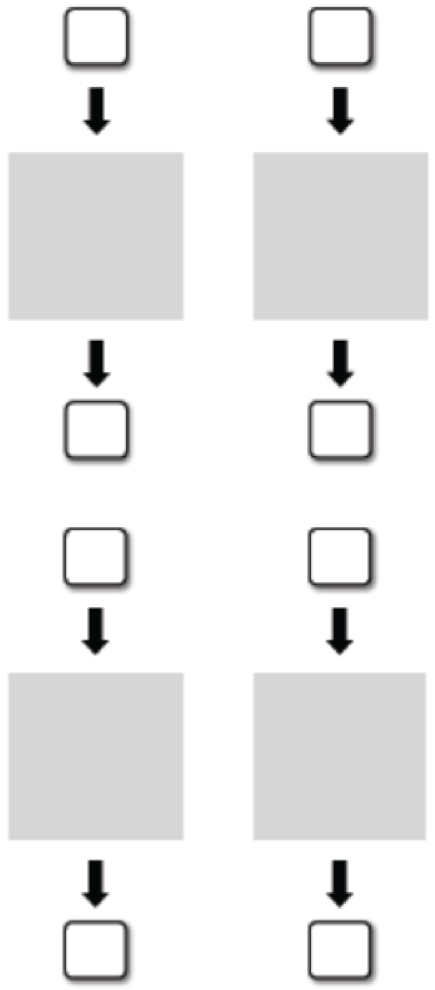
线程级并行(TLP) — 显式表达并行性

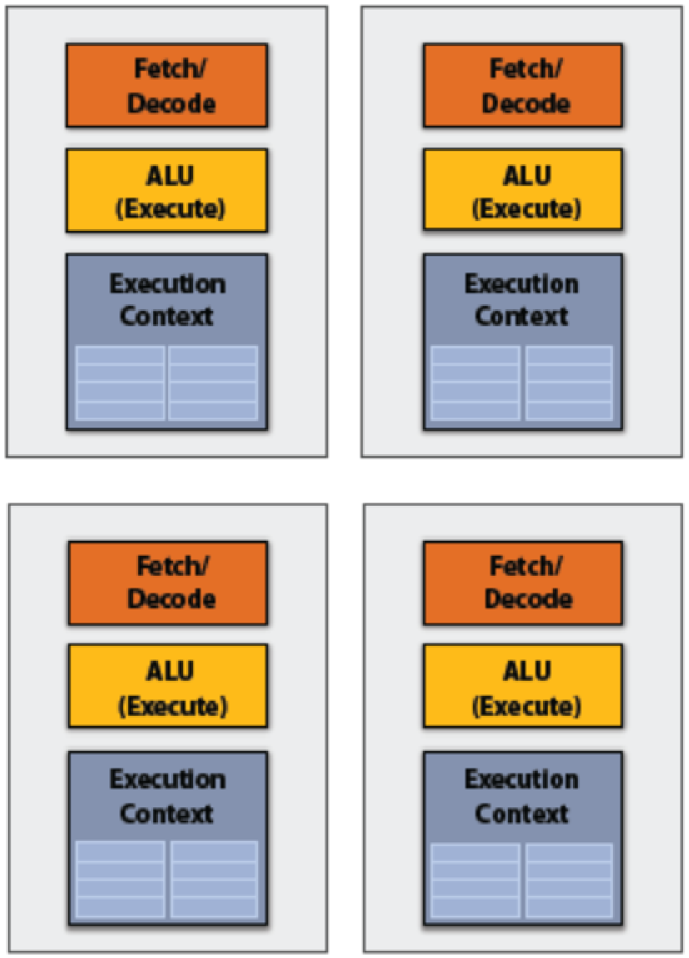
```
void parallel_sinx(int N, int terms, float* x, float* result)
{
  pthread_t thread_id;
  my_args args;
  args.N = N/2;
  args.terms = terms;
  args.x = x;
  args.result = result;

   pthread_create(&thread_id, NULL, my_thread_start,
&args);   // launch thread

sinx(N - args.N, terms, x + args.N, result + args.N);

  pthread_join(thread_id, NULL);
}

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

```
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;
  }
}
```

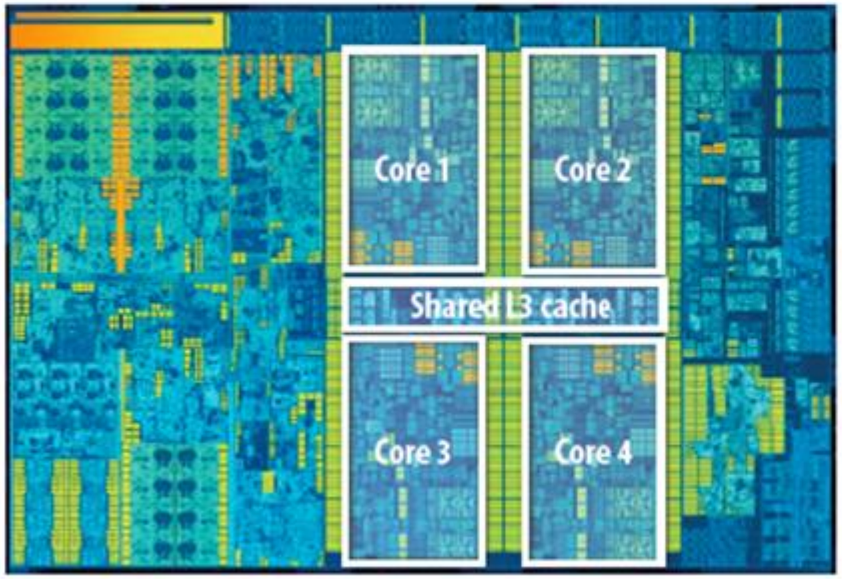
```
typedef struct
{    int N;    int terms;
      float* x;     float* result;
  }  my_args;
```

四核：同时执行四个指令流（异步）

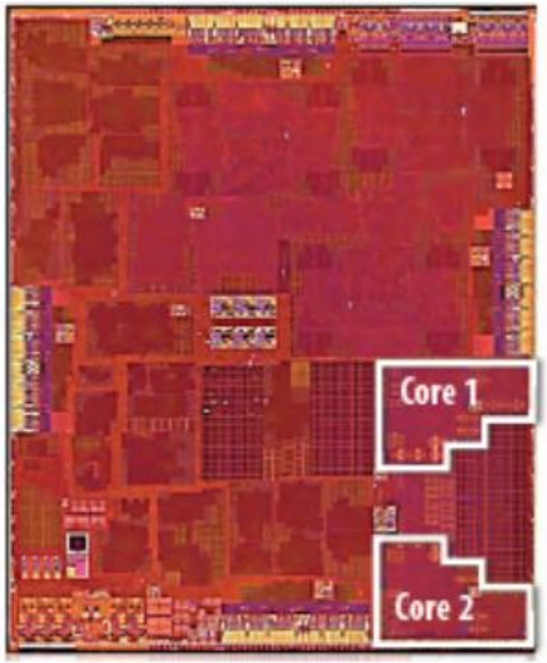




多核处理器实例



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



Apple A9 dual-core CPU
(2015)

数据级并行性（DLP）

```
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;
  }
}
```

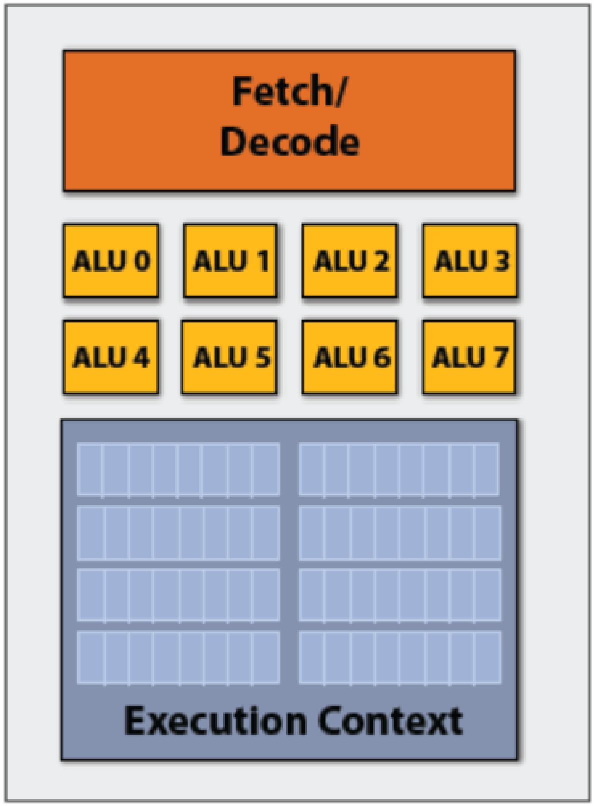
代码还具备以下特点：

循环级的并行性.：

多次迭代是对不同的数据完成相同的工作；
（N 可能很大，远大于16）

——— **数据级并行（DLP）**

SIMD — 单指令多数据



Idea #3:

增加多个ALU提高计算能力

SIMD processing

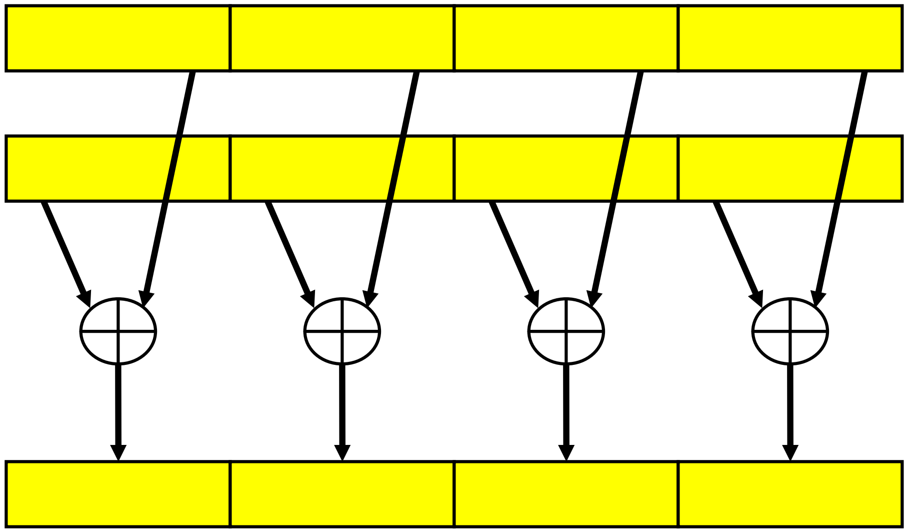
Single instruction, multiple data

将一条指令广播给所有ALU

所有 ALUs 同步执行一条指令，处理不同的数据

SIMD on modern CPUs

- **scalar. vector 指令都支持**
- 对原有指令集进行SIMD扩展
- 例如 x86 SSE 指令: 对128-bit 的数据并行操作:
    - 4x32 bits or 2x64 bits (4-wide float vectors)



SIMD on modern CPUs

- x86
  - Intel and AMD: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2
  - AVX instructions: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors) - 2011，Sandy Bridge架构开始支持
  - 目前: AVX 512 提供 512位长的 vectors， 2016.10 宣布
  - “It is made easier by the fact that companies like Google have open sourced their code such as Tensorflow.”
- PowerPC
  - AltiVEC/VMX: 128b
- ARM
  - NEON: 128b
  - Scalable Vector Extensions (SVE): up to 2048b

```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float* sinx)
{
  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_broadcast_ss(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(&sinx[i], value);
}
}
```

“显式的 SIMD”:

由程序员显式声明数据并行性，调用 intrinsics 中的库函数

编译器产生对应的SIMD指令

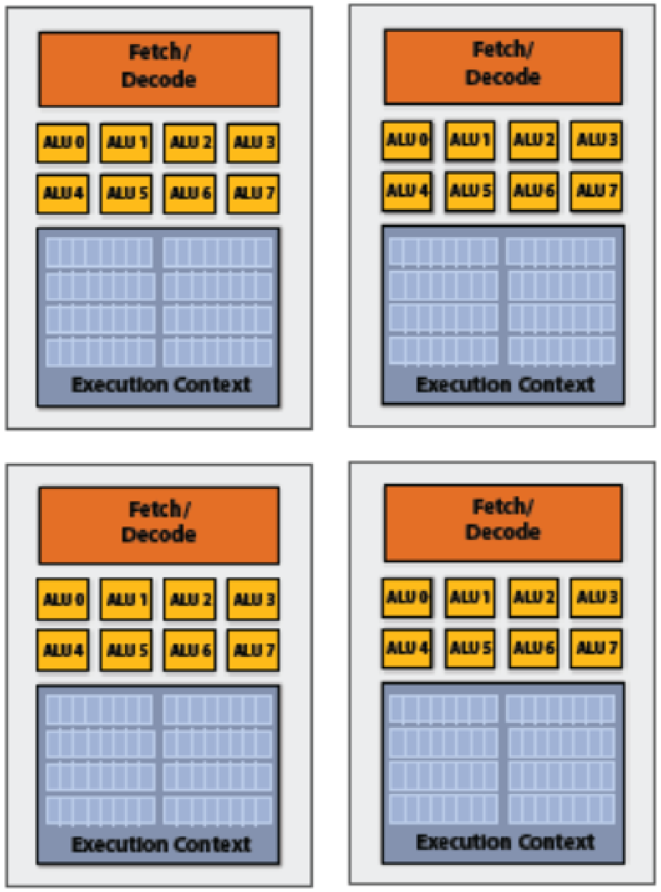
```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float* sinx)
{
  float three_fact = 6; // 3!
  for (int i=0; i<N; i+=8)
  {
    __m256 origx = _mm256_load_ps(&x[i]);
    __m256 value = origx;
    __m256 numer = _mm256_mul_ps(origx, _mm256_mul_p
    __m256 denom = _mm256_broadcast_ss(&three_fact);
    int sign = -1;
   for (int j=1; j<=terms; j++)
   {    // value += sign * numer / denom
         __m256 tmp =
_mm256_div_ps(_mm256_mul_ps(_mm256_broadcast_ss(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(&sinx[i], value);
 }
}
```

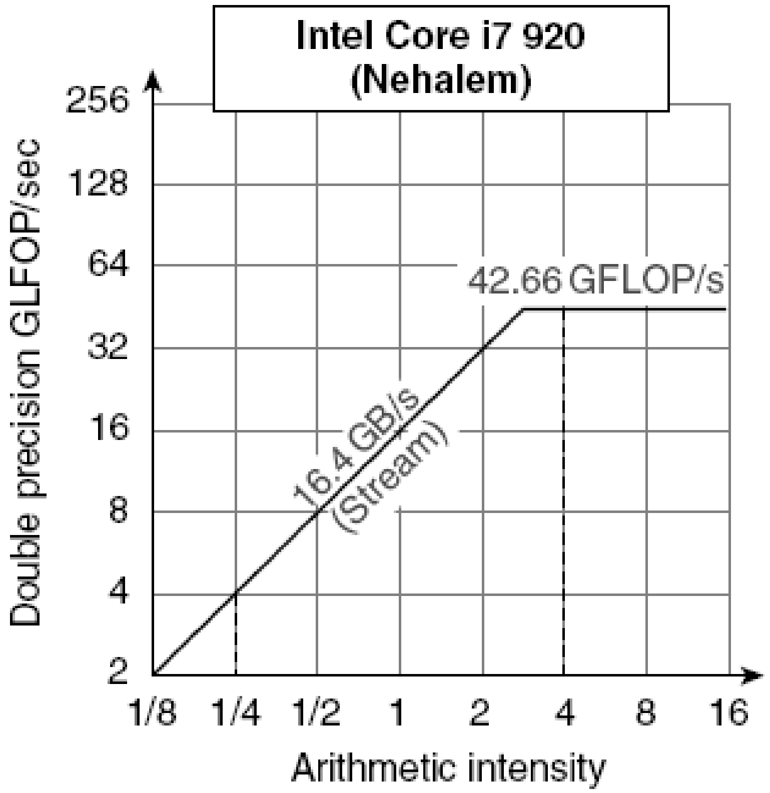
```
vloadps xmm0, addr[r1]
vmulps xmm1, xmm0, xmm0
vmulps xmm1, xmm1, xmm0
……
……
…….
vstoreps addr[xmm2], xmm0
```

编译程序: 使用一个256-位的向量寄存器，同时处理八个数据单元

SIMD CPU Example: Intel Core i7

4 cores， 8 SIMD ALUs per core （TLP， DLP 都支持）

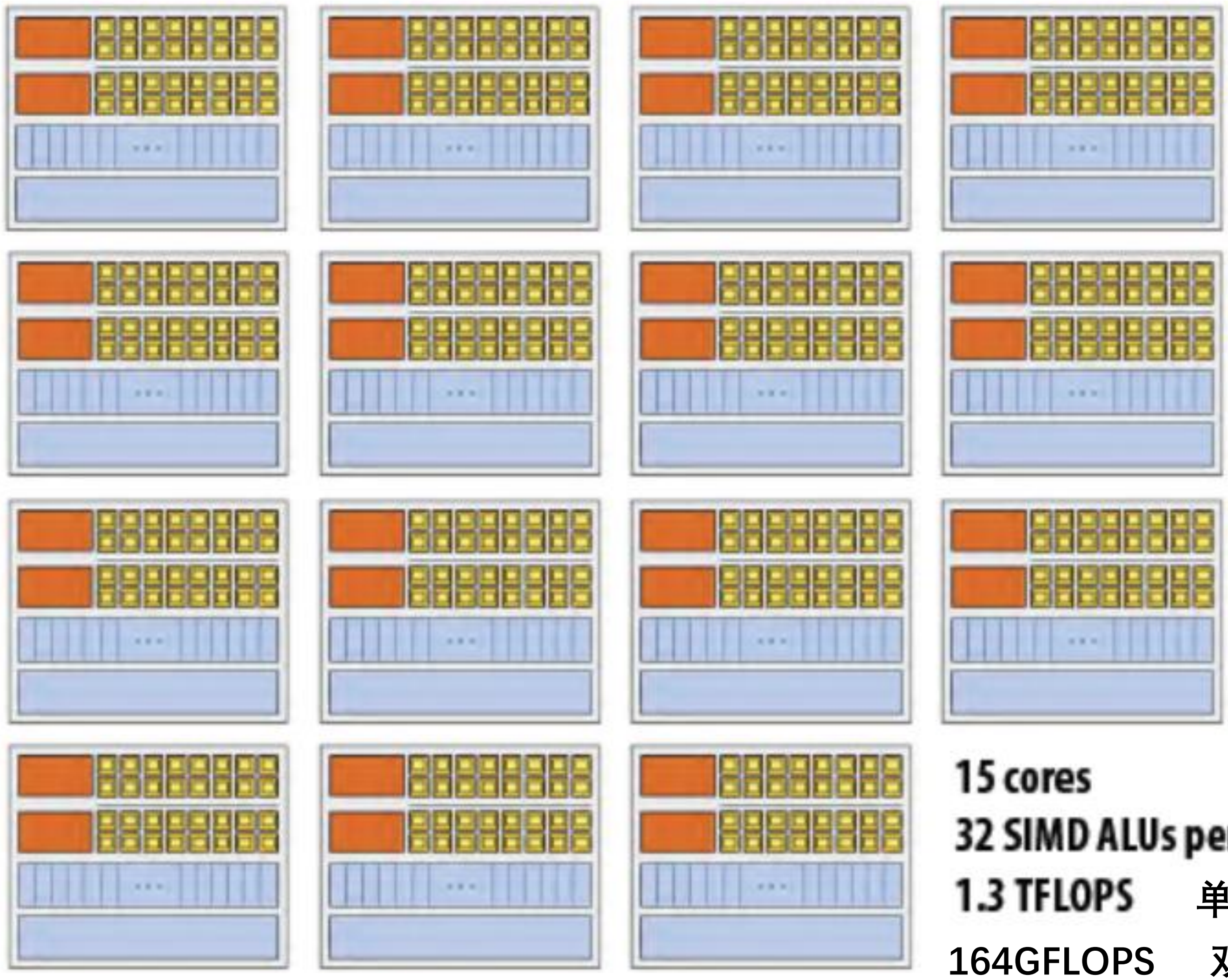




I7 875K，单精度：86.83， 双精度：45.46 GFLops

“arithmetic intensity” — ratio of math operations to data access operations

SIMD on GPU : NVIDIA GTX 480 （2010）



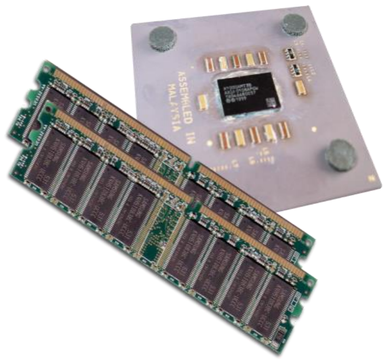
15 cores
32 SIMD ALUs per core
1.3 TFLOPS 单精度浮点计算能力
164GFLOPS 双精度计算能力

SIMD on many modern GPUs

- 隐式的SIMD
- 硬件接口本身就是“数据级并行的”
- GPU中的 SIMD 宽度： ranges from 8 to 32
- 编译器生成标量指令 (scalar instructions)
- 硬件会自动将标量指令广播给所有ALU，自动同步并行执行 N instances of the program
- 各个ALU中的的instance 执行相同指令，但处理的是不同的数据

异构计算 on GPU

- 术语:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)


Host

Device

异构计算

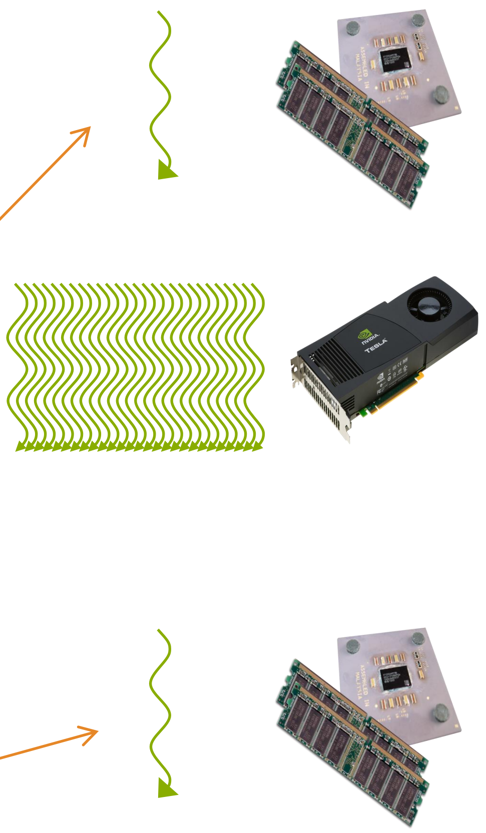
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N        1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;           // host copies of a, b, c
        int *d_in, *d_out;       // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

serial code

parallel code

serial code



CUDA Threads

- Terminology: a block can be split into parallel threads
- SIMT： single instruction multiple thread
- Let’s change `add()` to use parallel *threads*

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- We use **`threadIdx.x`**
- Need to make one change in **`main()`**…

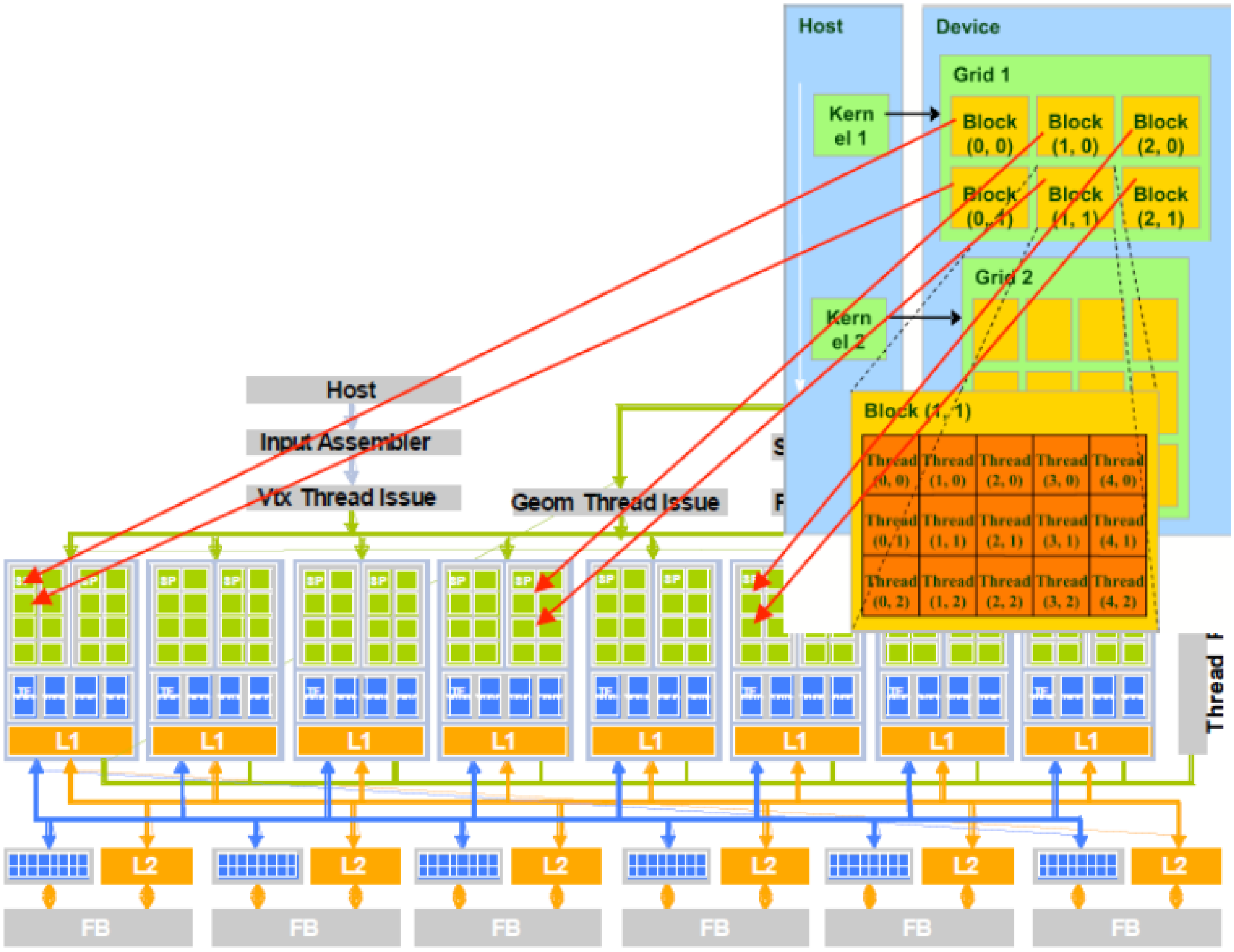
Vector Addition Using Threads: `main()`

```
#define N 512
int main(void) {
    int *a, *b, *c;               // host copies of a, b, c
    int *d_a, *d_b, *d_c;         // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

// Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

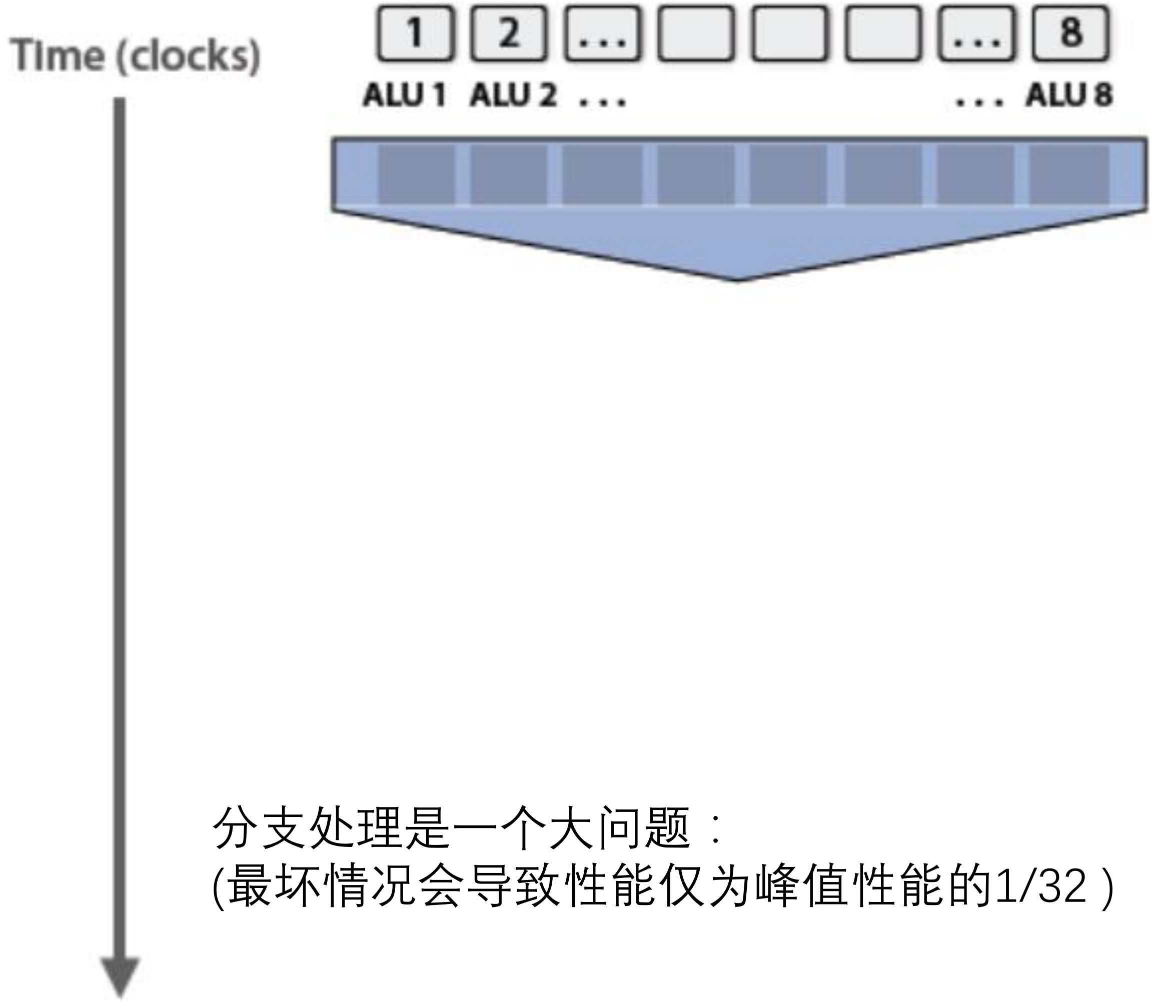
Vector Addition Using Threads: `main()`

```
        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU with N threads
        add<<<1,N>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

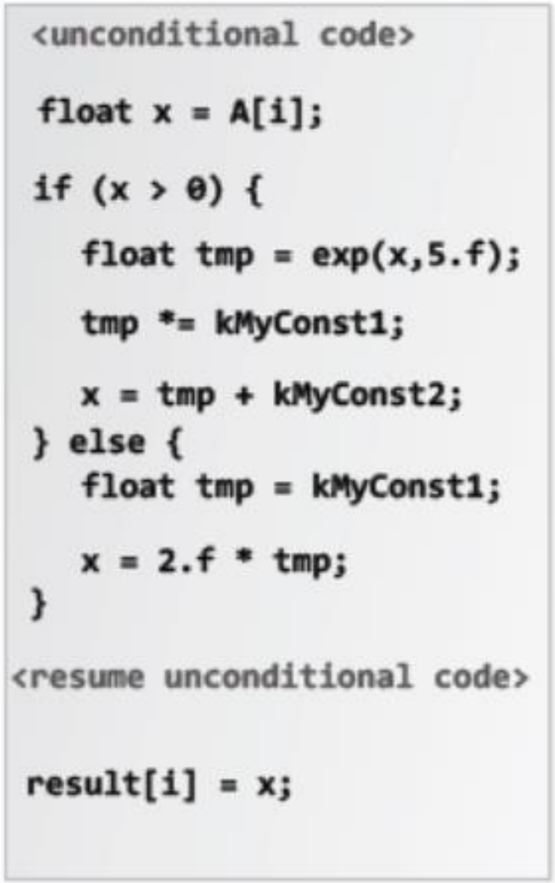
线程并行模型


Host
Device
Grid 1
Kernel 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kernel 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)
Host
Input Assembler
Vtx Thread Issue
Geom Thread Issue
SP
TF
L1
L2
FB
Thread

如何处理条件分支？



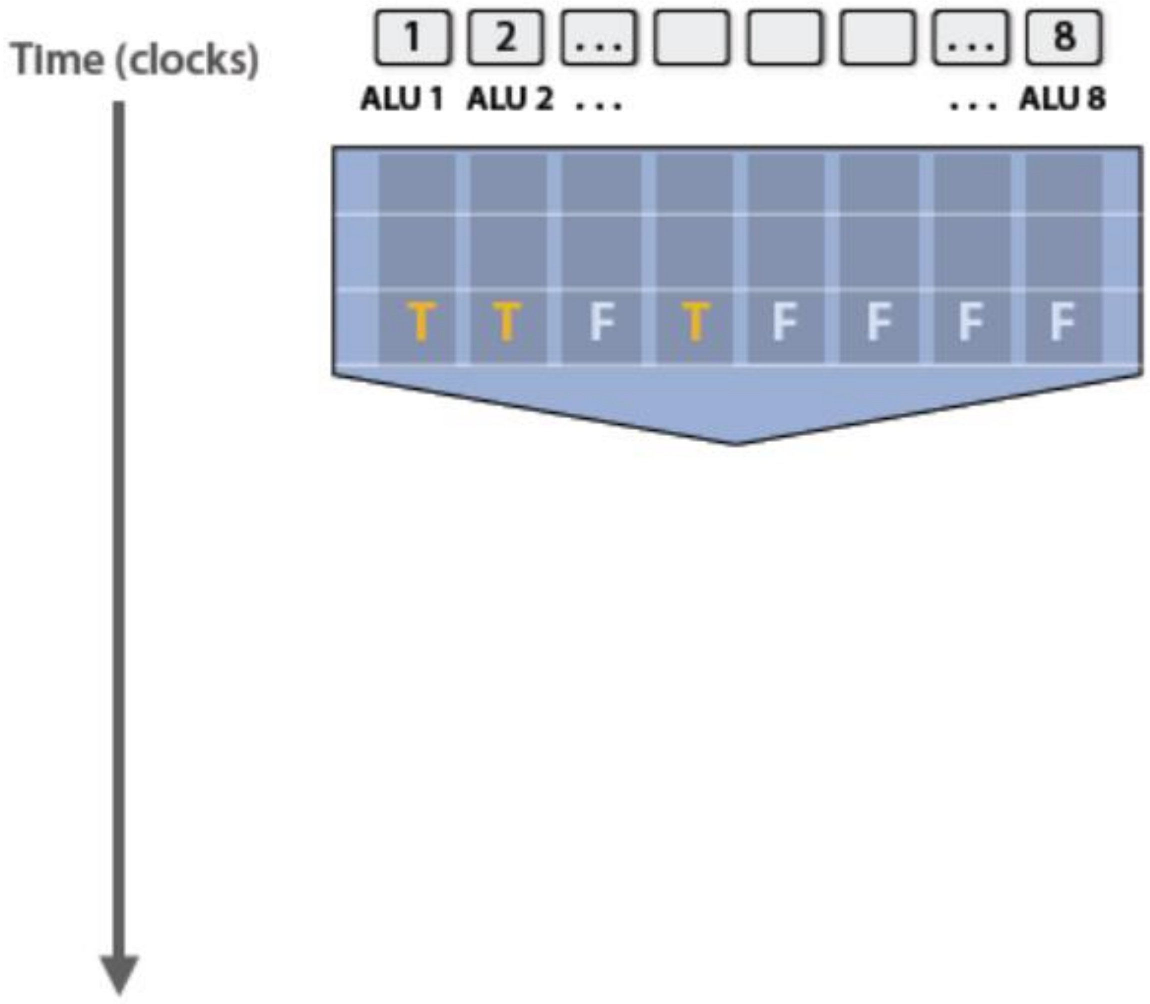
(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')



```
<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;
```

分支处理是一个大问题：
(最坏情况会导致性能仅为峰值性能的1/32)

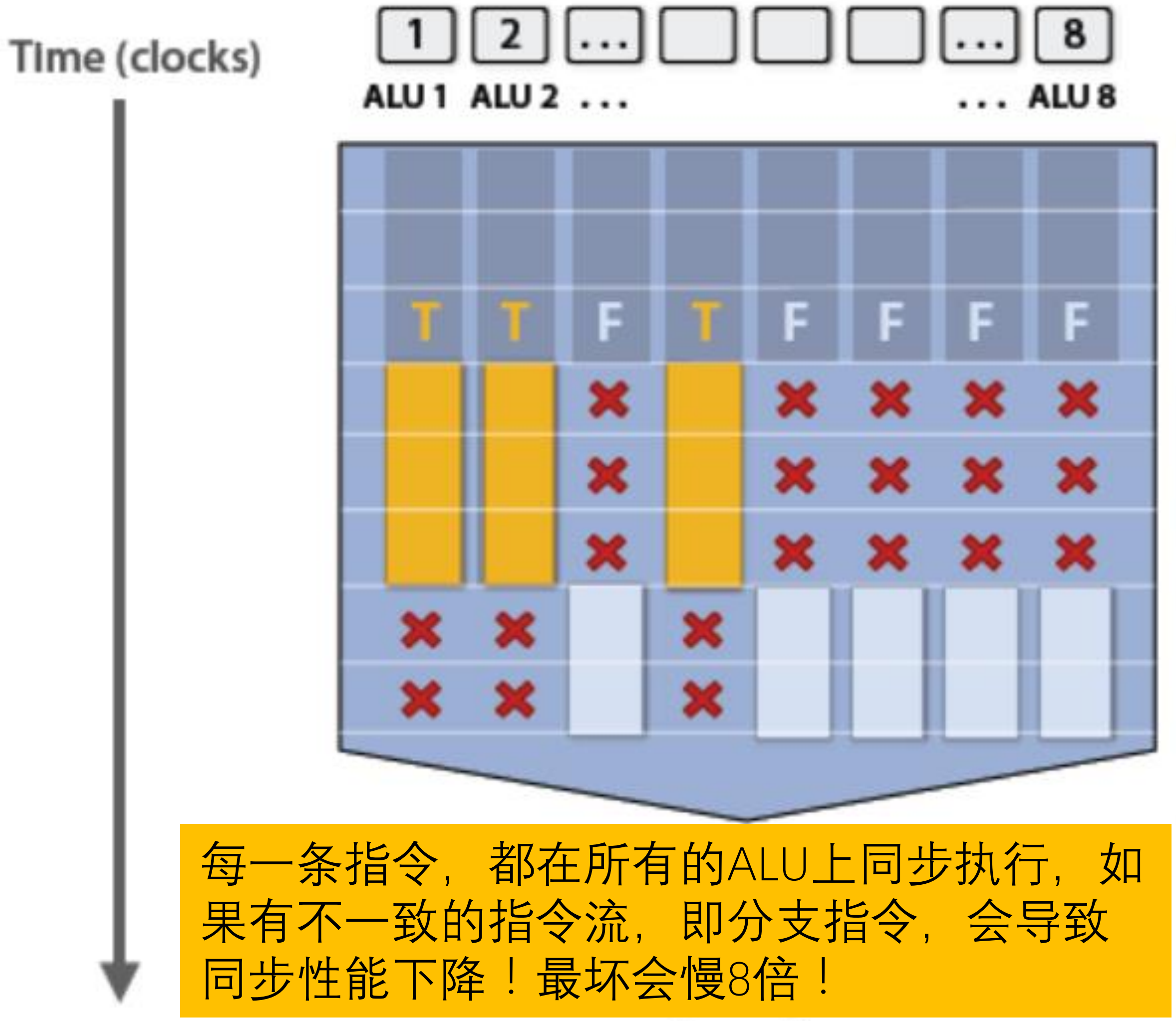
如何处理条件分支？



(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;
```

设置掩码（Mask）寄存器，屏蔽部分ALU的输出



(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;
```

每一条指令，都在所有的ALU上同步执行，如果有不一致的指令流，即分支指令，会导致同步性能下降！最坏会慢8倍！

举例：if (X[i] != 0)
X[i] = X[i] – Y[i];
else X[i] = Z[i];

分支指令，编译后得到的Nvidia GPU“Parallel Thread Execution (PTX)” 指令

```
        ld.global.f64  RD0, [X+R8]       ; RD0 = X[i]
        setp.neq.s32  P1, RD0, #0        ; P1 is predicate register 1

        @!P1, bra      ELSE1,            ; if P1 false, got ELSE1

        ld.global.f64   RD2, [Y+R8]      ; RD2 = Y[i]
        sub.f64 RD0, RD0, RD2            ; Difference in RD0
        st.global.f64   [X+R8], RD0      ; X[i] = RD0

        @P1, bra       ENDIF1            ; if P1 true, go to ENDIF1

ELSE1:  ld.global.f64 RD0, [Z+R8]        ; RD0 = Z[i]
        st.global.f64 [X+R8], RD0        ; X[i] = RD0

ENDIF1: <next instruction>,              ;
```

并行执行-summary

- 多核处理器 multi core
  - 完全不同的指令流在不同处理器上异步执行；
  - 线程级并行 thread-level parallelism:
  - 程序中显式说明
- 单指令多数据 SIMD
  - 一条指令控制一个核内的多个ALU同步执行相同的动作
  - 数据级并行
  - 可以是显式（程序中说明，CPU）或隐式的（硬件自动并行，GPU）
- 超标量 Superscalar
  - 开发一个指令执行流中的指令间的并行性，加快执行速度
  - exploit ILP 指令级并行
  - 硬件动态检测，无须程序员干预

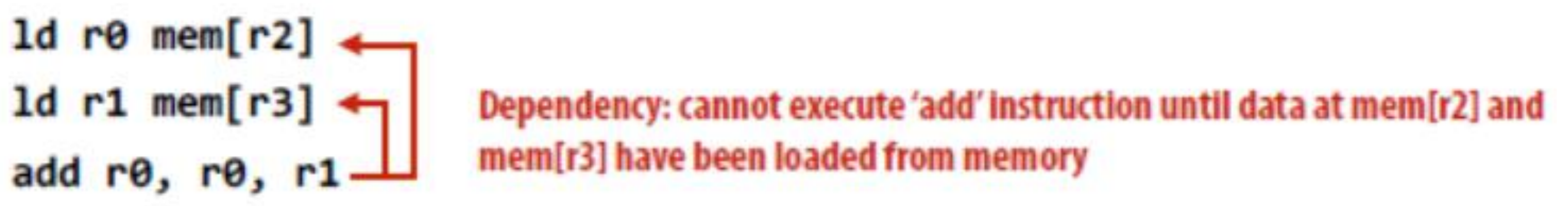
Part 2： 存储器访问

降低、隐藏访存延迟

提升存储带宽

“停顿”

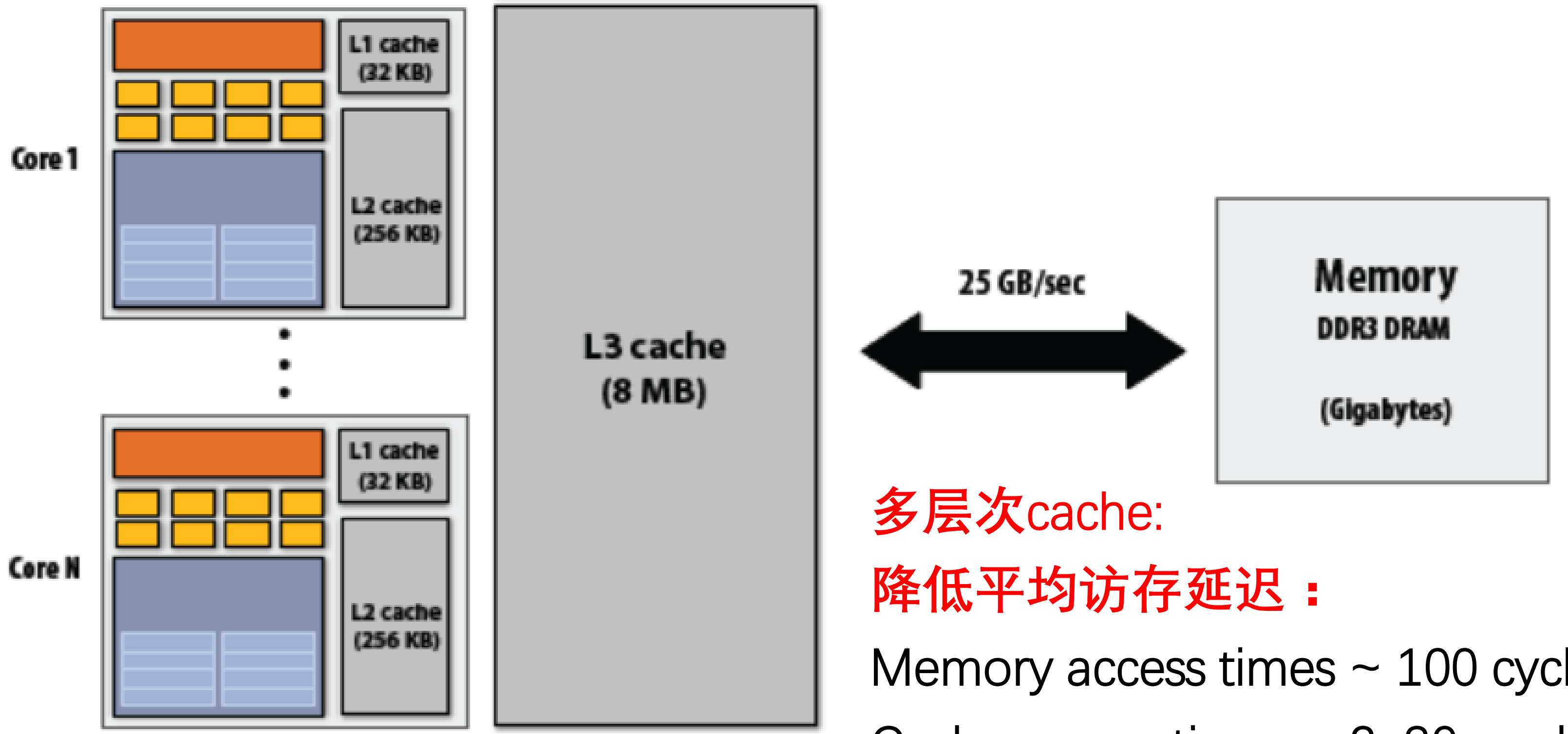
- 处理器 “停顿”的原因
- 指令流中的指令与前面的指令有依赖关系
  - Data Dependency
  - Structure Dependency
  - Control Dependency
- 存储器访问时 “停顿”产生的主要原因之一
  - 存储器的访存延迟一般为100个时钟周期



```
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

CPU Memory hierarchies : 多层次cache



多层次cache:

降低平均访存延迟：

Memory access times ~ 100 cycles

Cache access times ~ 2-30 cycles

计算题

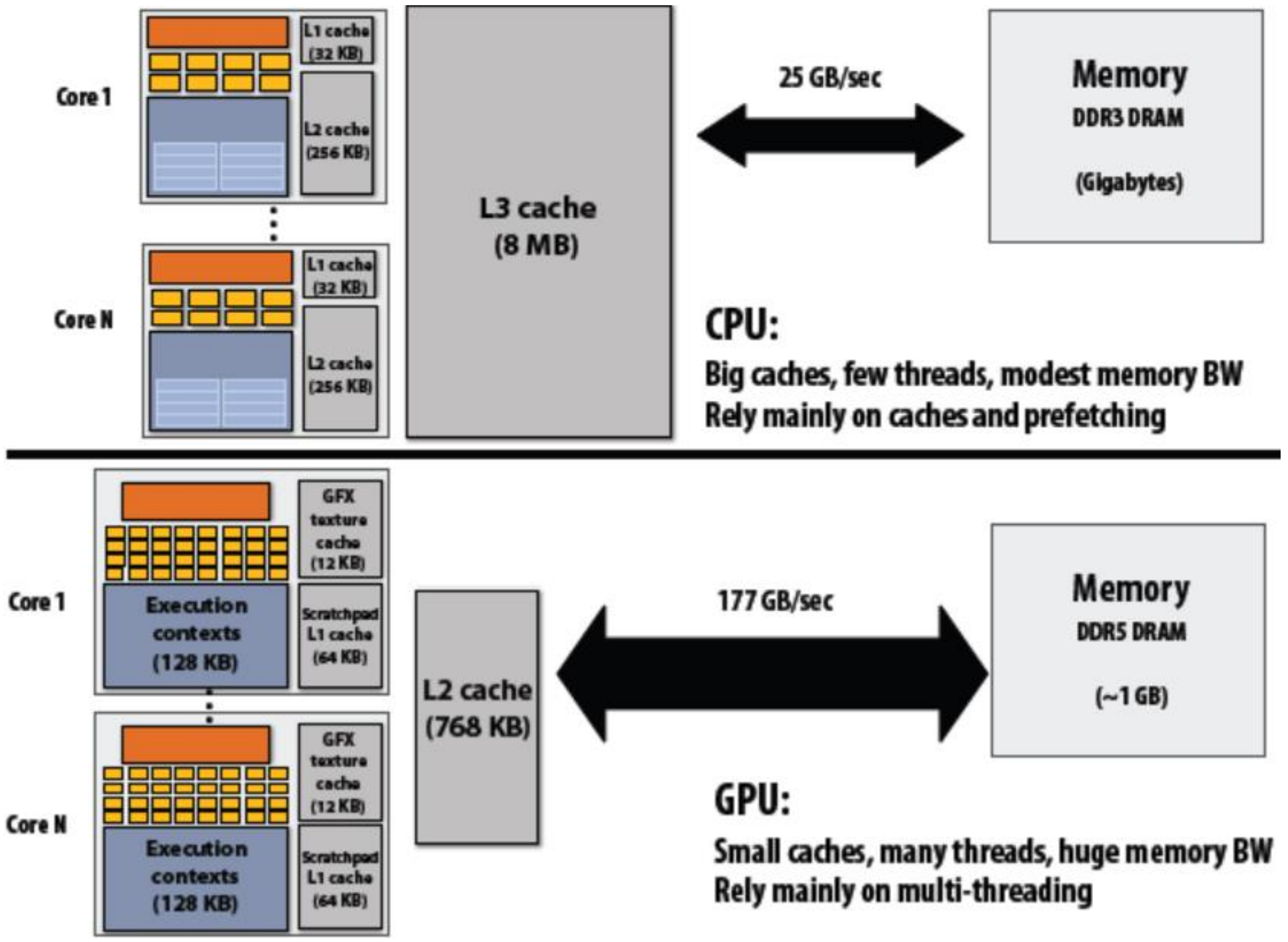
- 有一个 八核 3 GHz 的 有L2 cache 的服务器；
- 每个核执行某工作负载的平均 CPI 为2.0；
- L2 cache的数据块（line size）大小为32B，
- 工作负载平均每1K条指令会产生6.67次L2 miss,
- 存储器的带宽一般为平均带宽需求的两倍
- 请问存储器带宽至少为多少？
- 一个单通道的DDR2-667 DIMM是否满足该系统的带宽需求？

- 8cores × 3GHz / 2.0 CPI = 12 billion instructions per second.
- 12 × 0.00667 = 80 million level-2 misses per second.
- 80 × 32B = 2560MB/sec； 2X this, it would be 5120MB/sec.
- 667M * 2 * 8B = 10672 MB/sec， DDR2-667 DIMM 满足带宽需求

如何减少（reduce）或隐藏（hide）访存延迟

- 多层次的cache
  - 减小cache miss rate
- 预取（ Prefetch ）
  - 动态分析程序访问数据的规律，提前将数据或指令装入cache.
- 乱序执行隐藏cache miss 的开销
  - 发生 cache misses时，可以寻找不相关的指令继续执行
  - hiding memory latency
    - Especially good at hiding L2 hits (~12 cycles in Core i7)

CPU vs. GPU memory hierarchies



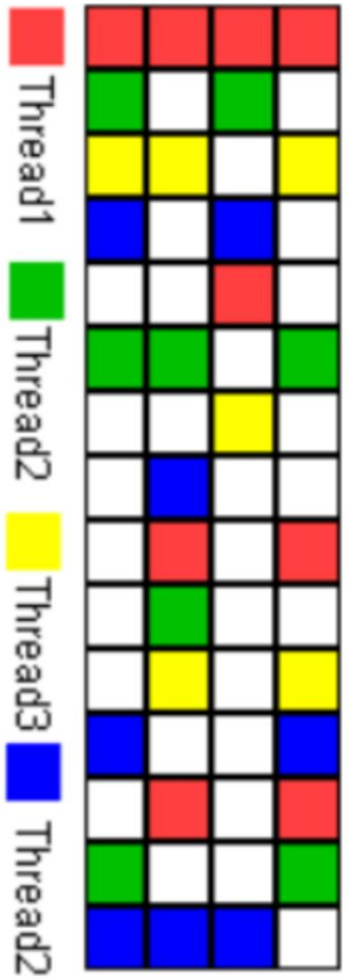
CPU:
Big caches, few threads, modest memory BW
Rely mainly on caches and prefetching

GPU:
Small caches, many threads, huge memory BW
Rely mainly on multi-threading

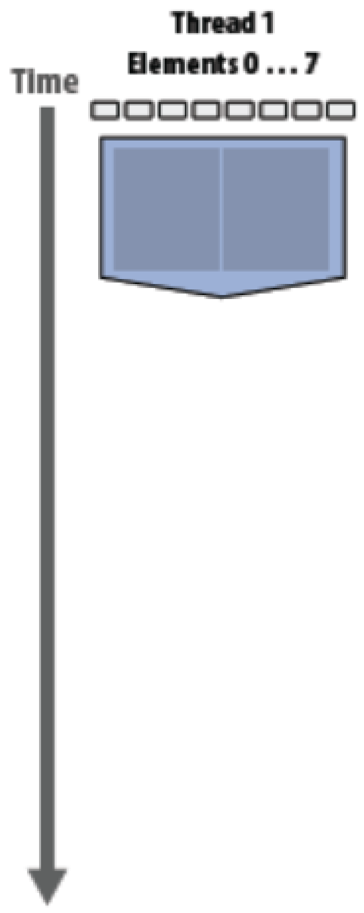
GPU: 利用多线程隐藏访存延迟

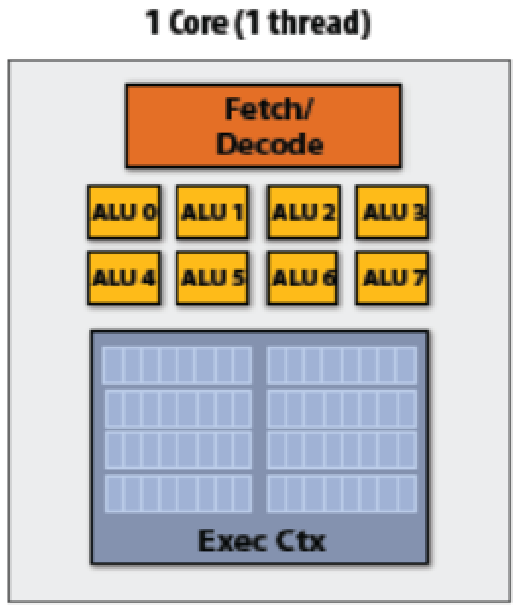
基本思想：

- 多个线程交替执行，隐藏较长的访存延迟
- **interleave processing of multiple threads on the same core to hide stalls**

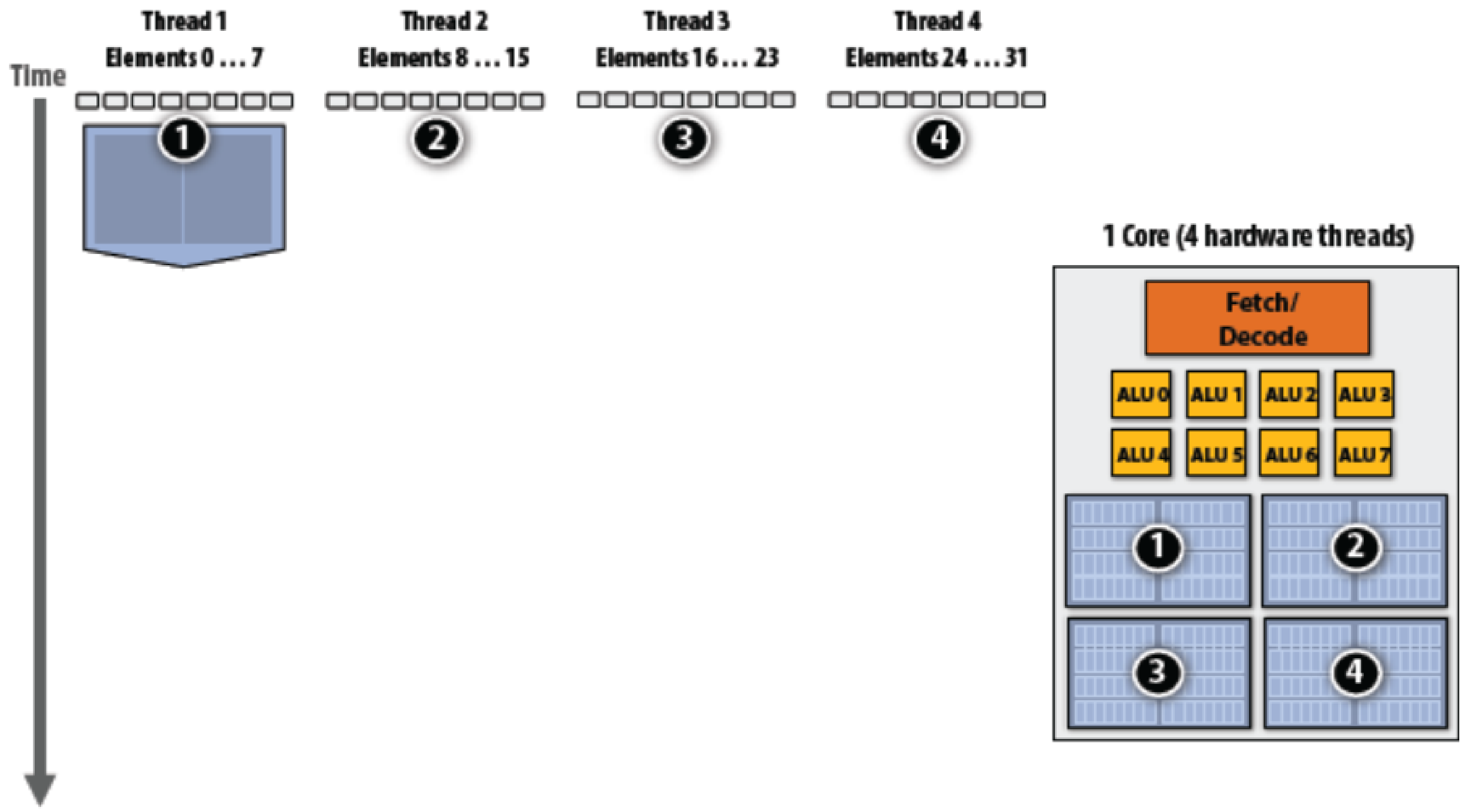


利用多线程隐藏访存延迟

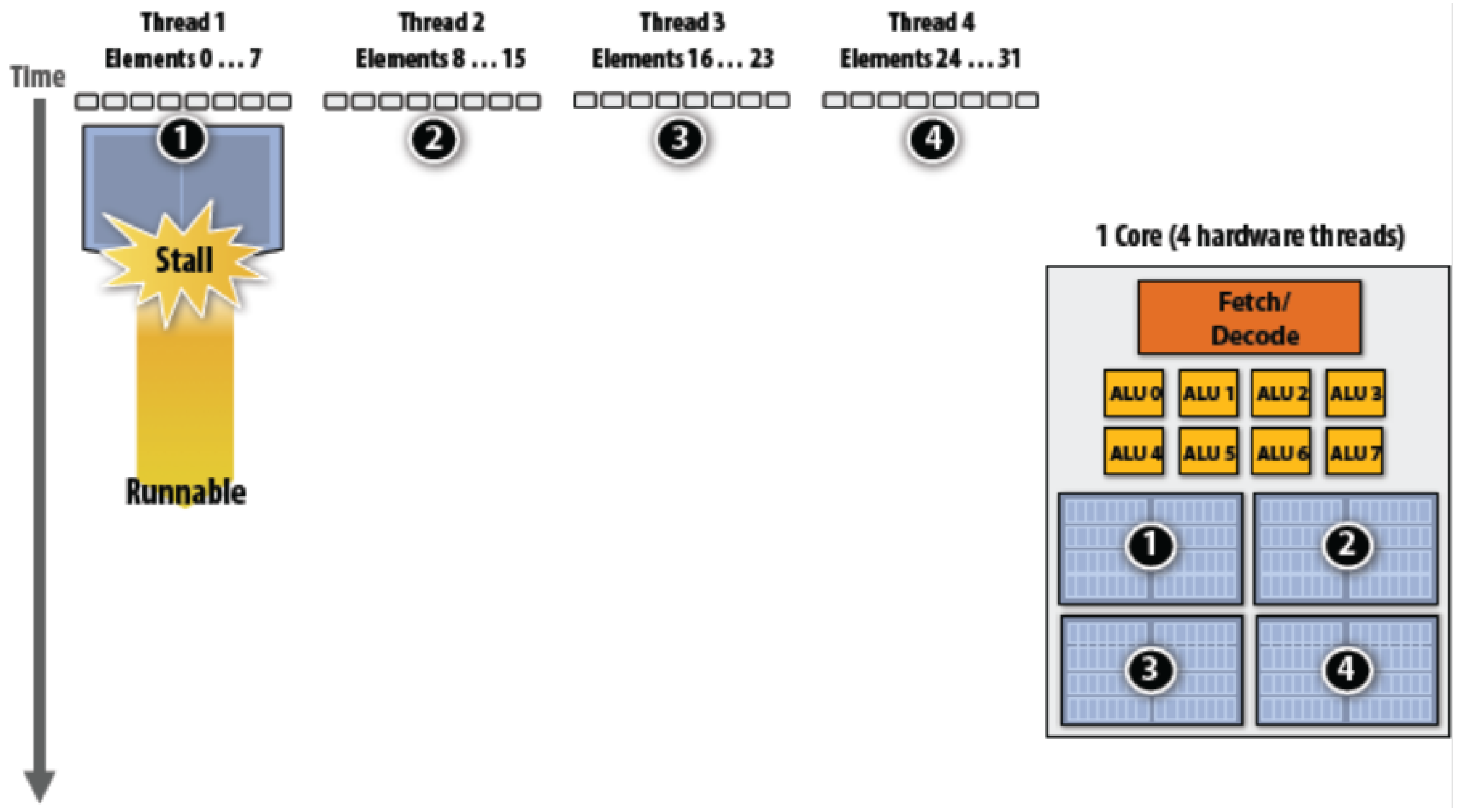




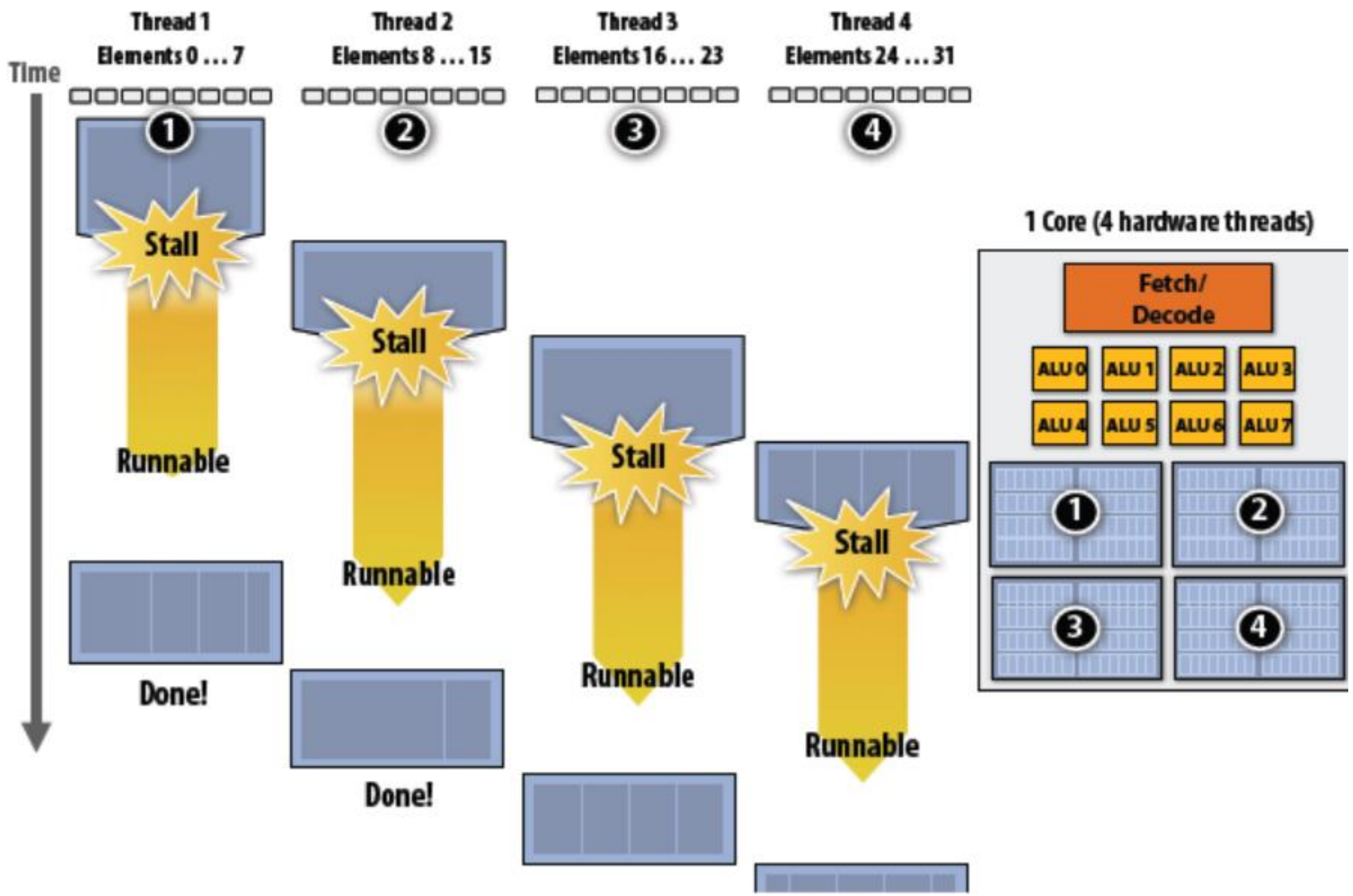
利用多线程隐藏访存延迟



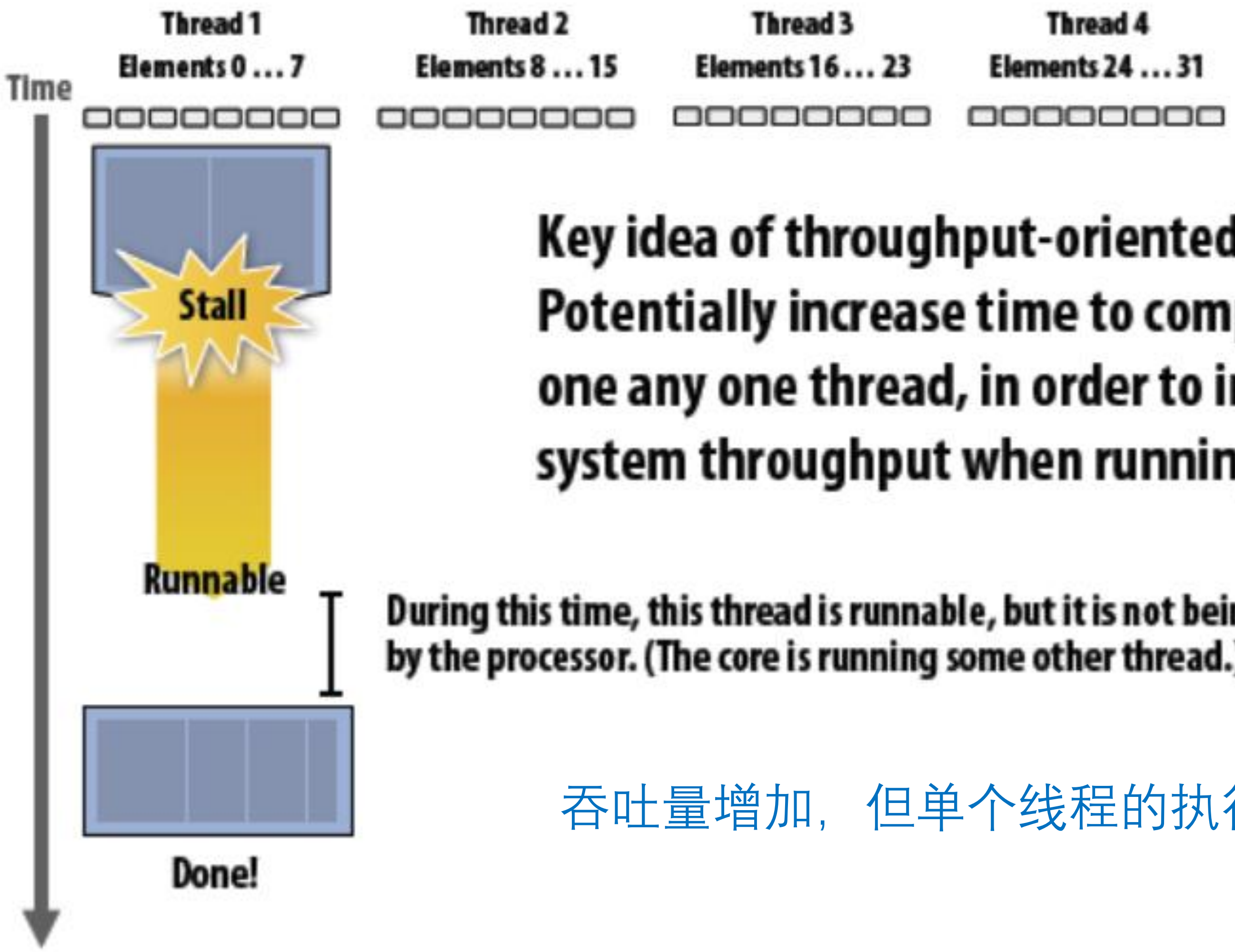
利用多线程隐藏访存延迟



利用多线程隐藏访存延迟



吞吐量增加引起的trade off



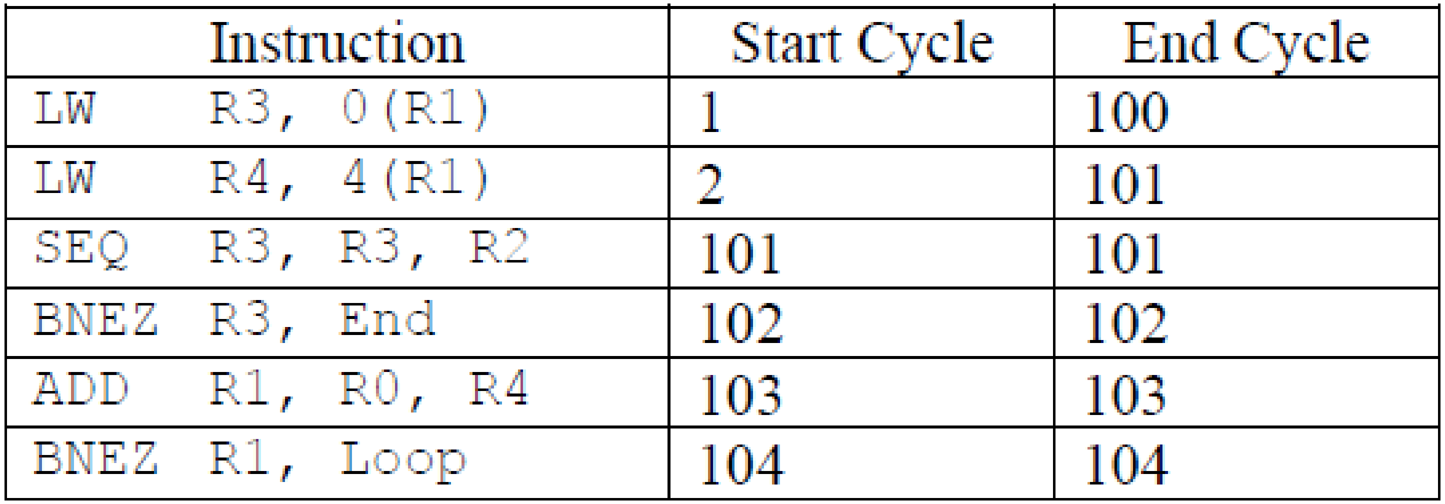
Key idea of throughput-oriented systems:
Potentially increase time to complete work by any one any one thread, in order to increase overall system throughput when running multiple threads.

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

吞吐量增加，但单个线程的执行时间变长

问题：

- 假设系统没有 cache
- 直接存储器访问需要 100 CPU cycles.
- load/store unit 全流水化，非阻塞.
- 执行一个线程所需要的延迟如下表：
- 如果轮转式的交替执行多个线程，将延迟隐藏，至少需要多少个线程？

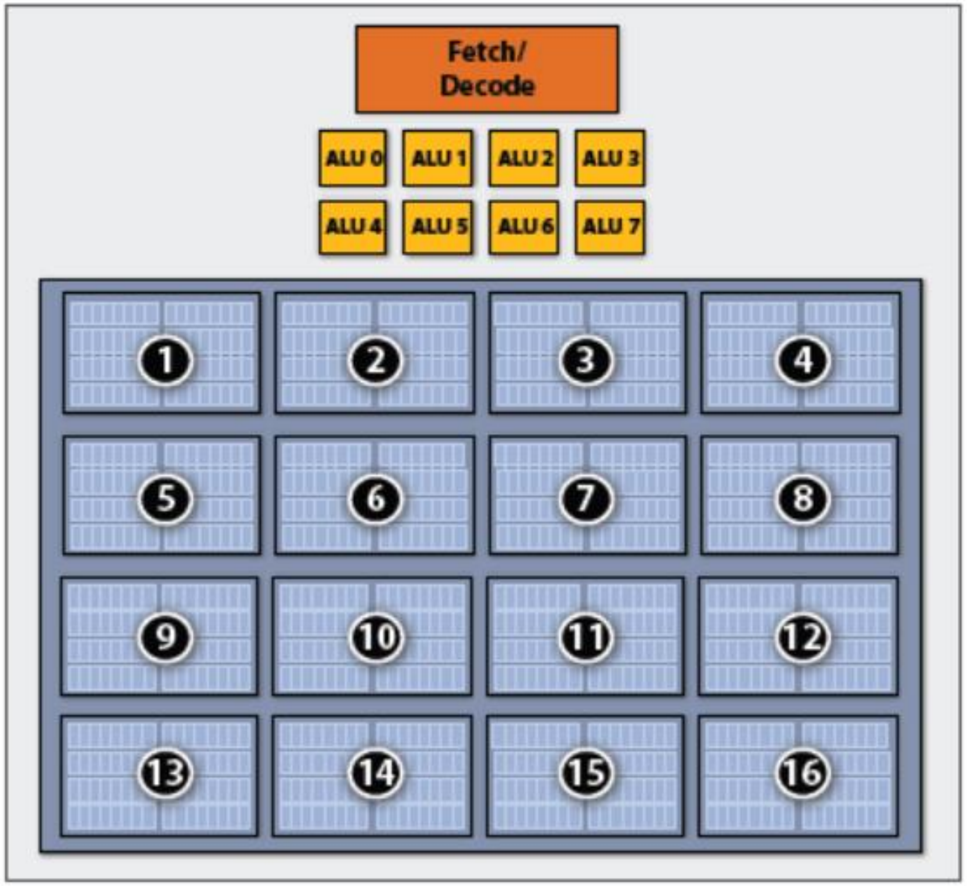


Instruction	Start Cycle	End Cycle
LW R3, 0(R1)	1	100
LW R4, 4(R1)	2	101
SEQ R3, R3, R2	101	101
BNEZ R3, End	102	102
ADD R1, R0, R4	103	103
BNEZ R1, Loop	104	104

,2N + 1 ⩾ 101. The minimum number of thread needed is 50.

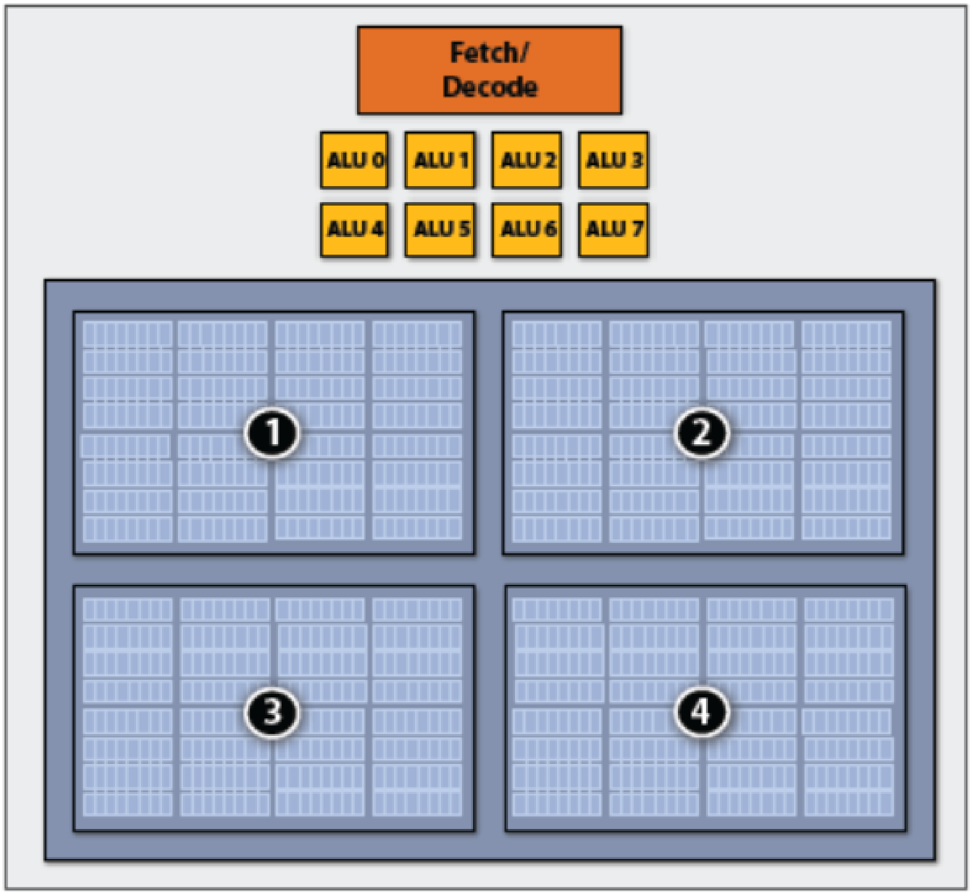
多线程：多个上下文 (high latency hiding ability)

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



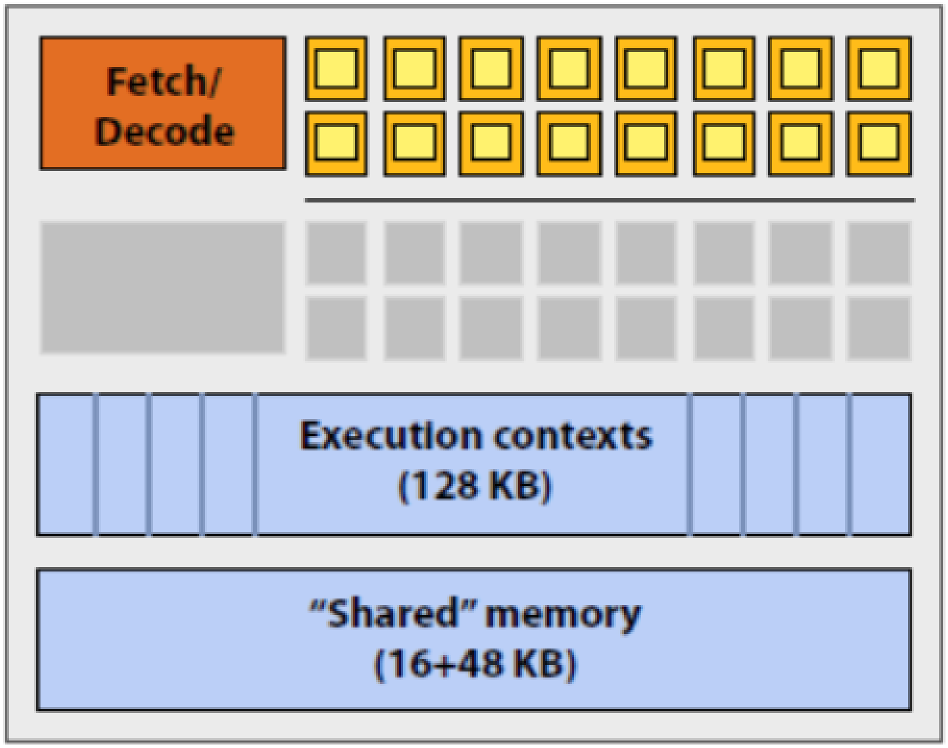
少量的上下文：(low latency hiding ability)

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

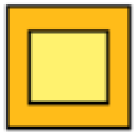


GPUs: Extreme throughput-oriented processors

NVIDIA GTX 480 core



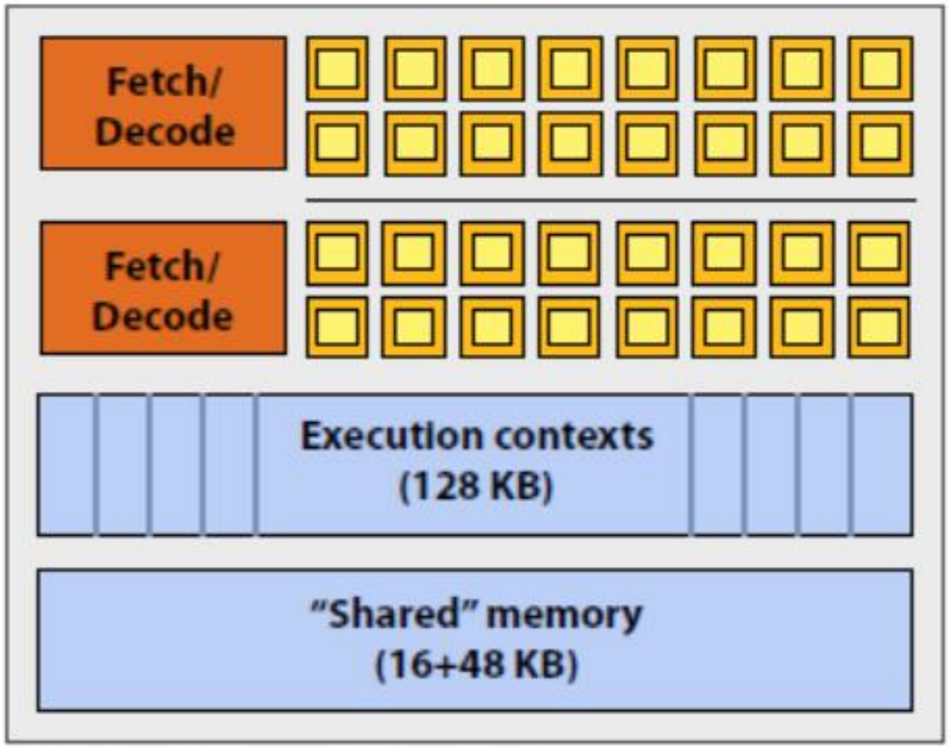
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G


= SIMD function unit, control shared across 16 units (1 MUL-ADD per clock)

- Why is a warp 32 elements and there are only 16 SIMD ALUs?
- It's a bit complicated: ALUs run at twice the clock rate of rest of chip. So each decoded instruction runs on 32 pieces of data on the 16 ALUs over two ALU clocks. (but to the programmer, it behaves like a 32-wide SIMD operation)

NVIDIA GTX 480: more detail

NVIDIA GTX 480 core

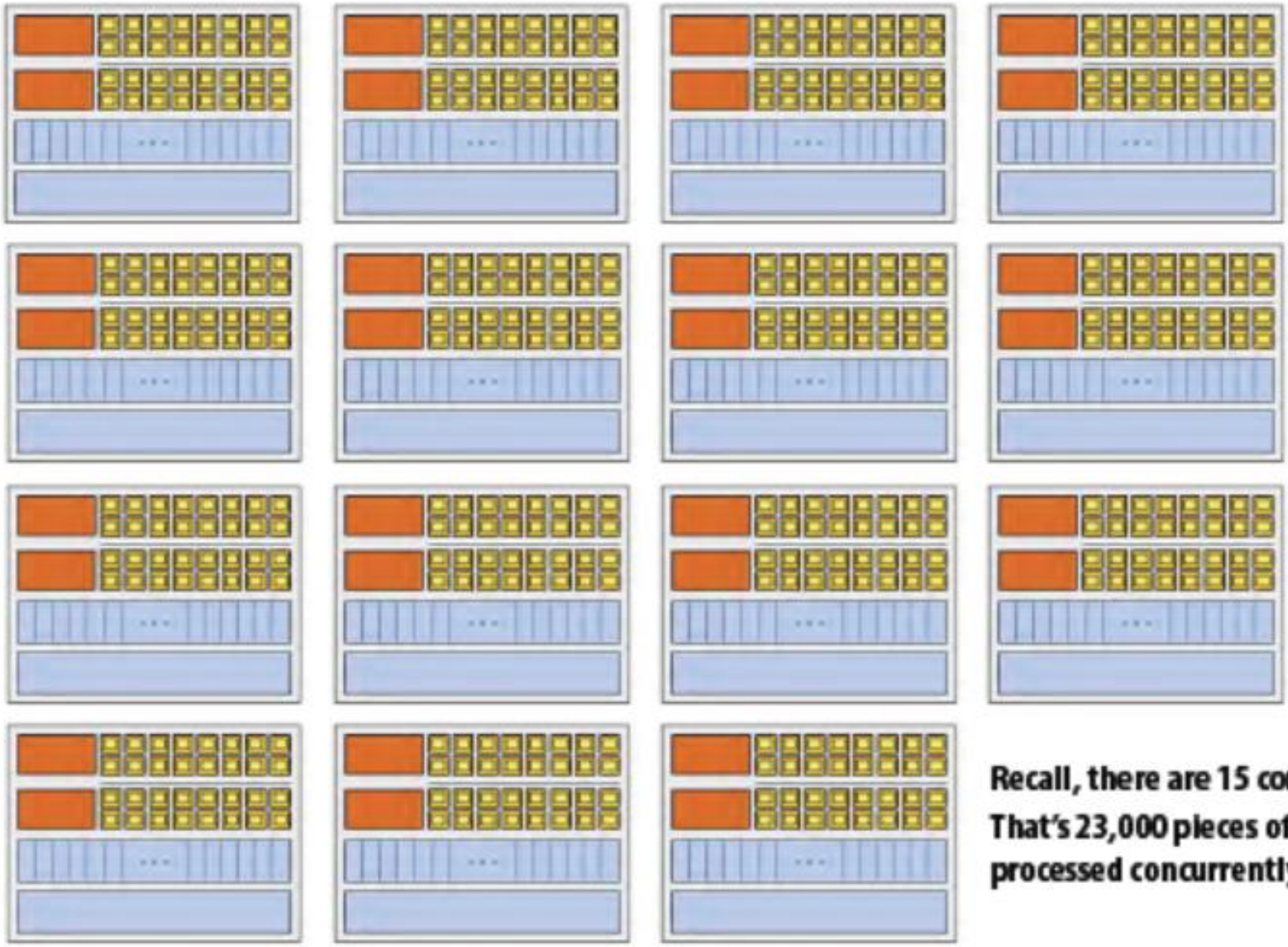


Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

▨ = SIMD function unit, control shared across 16 units (1 MUL-ADD per clock)

- This process occurs on another set of 16 ALUs as well
- So there are 32 ALUs per core
- 15 cores × 32 = 480 ALUs per chip

NVIDIA GTX 480



Recall, there are 15 cores on the GTX 480:
That's 23,000 pieces of data being processed concurrently!

总结：GPU的三个key idea

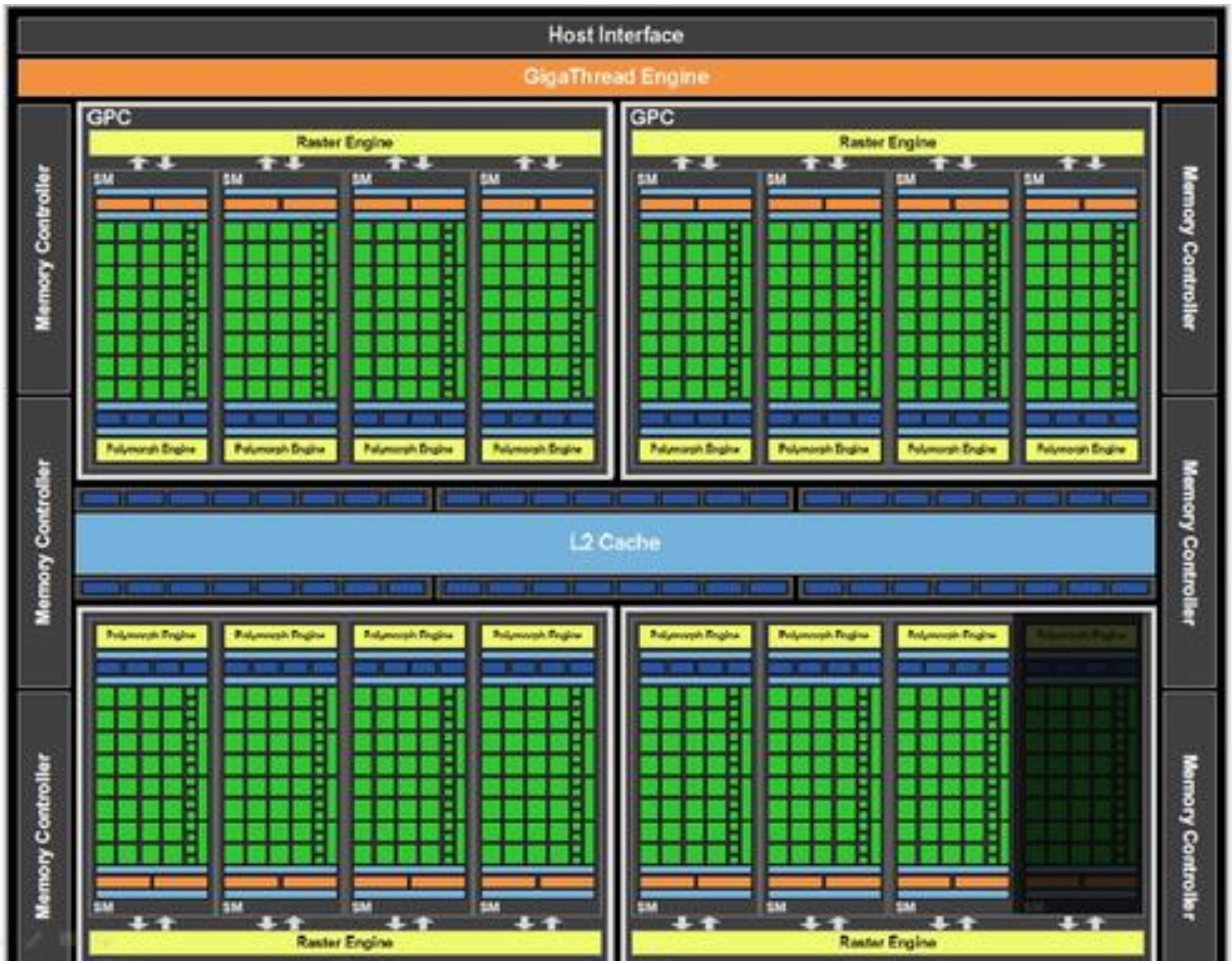
- Employ multiple processing cores
  - 简单的核 (比起ILP，更注重TLP ：thread-level parallelism)
- Pack cores full of ALU
  - 开发数据级的并行性 (SIMD)
  - 一条指令执行时，多个ALU同步处理不同的数据
  - 芯片面积额外增加的开销不大，但大大增强了计算能力
- 利用多线程提高系统的处理能力（吞吐量）
  - 轮流交替执行不同线程的代码段以隐藏延迟

Nvidia 通用图形加速单元体系结构

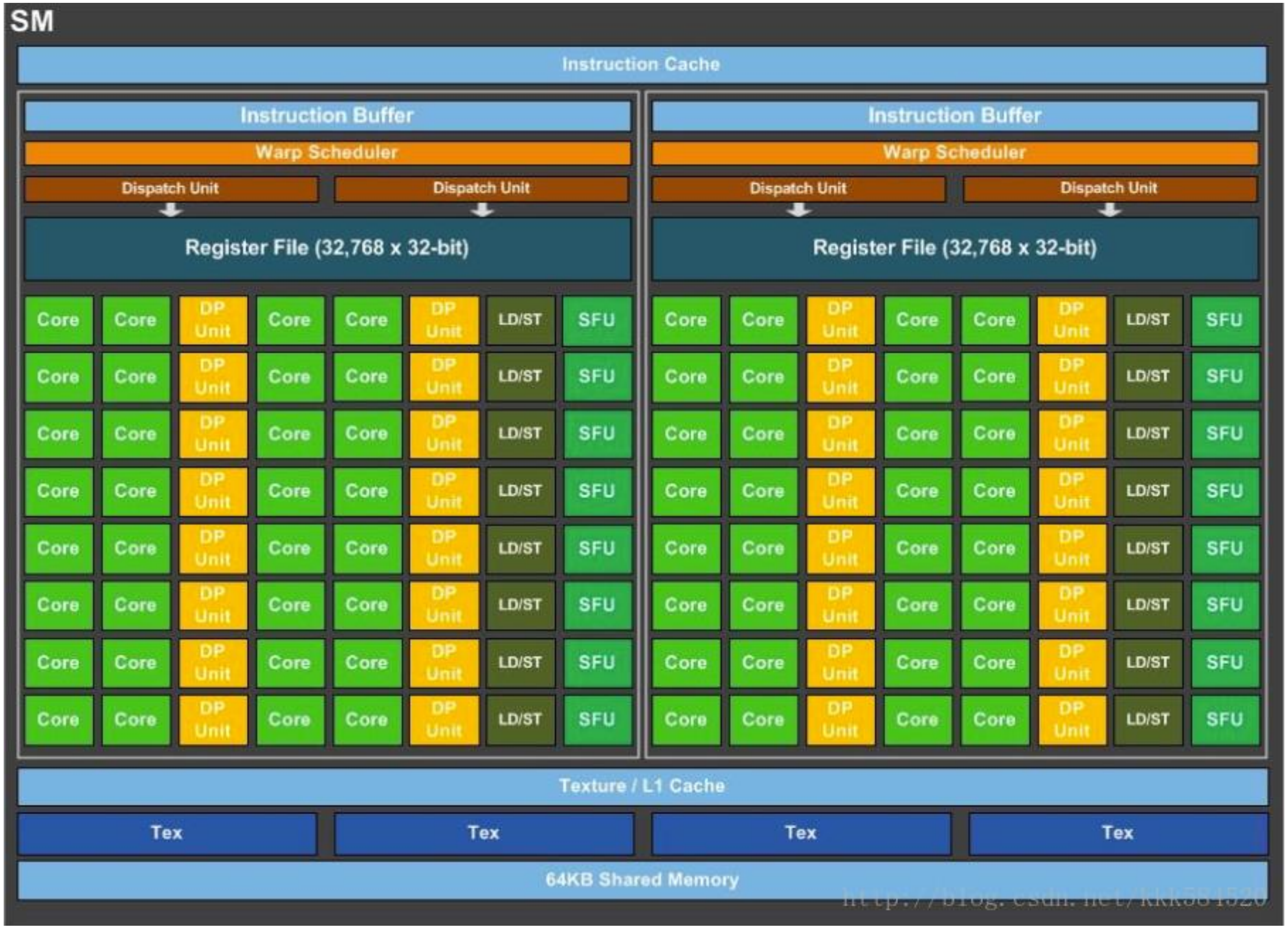
- 2008 Tesla
- 2010 Fermi
- 2012 Kepler
- 2014 Maxwell
- 2016 Pascal

- 每个SM包含的SP（GPU core）数量依据GPU架构而不同， Fermi架构GF100是32个， GF10X是48个， Kepler架构都是192个， Maxwell都是128个，Pascal64个。

Floorplan of Fermi GTX480



Pascal 架构(Tesla P100 2016)



Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
Peak FP32 GFLOPs[1]	5040	6840	10600
Peak FP64 GFLOPs[1]	1680	210	5300
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm^2	601 mm^2	610 mm^2
Manufacturing Process	28 nm	28 nm	16 nm FinFET

By the numbers: CPUs vs GPUs

	Intel Xeon Platinum 8168 “Skylake”	Nvidia Tesla P100	Intel Xeon Phi 7290F
frequency	2.7 GHz	1.3 GHz	1.5 GHz
cores / threads	24 / 48	56 (“3584”) / 10Ks	72 / 288
RAM	768 GB	16 GB	384 GB
DP TFLOPS	1.0	4.7	3.5
Transistors	>5B ?	15.3B	>5B ?
Price	$5,900	$6,000	$3,400

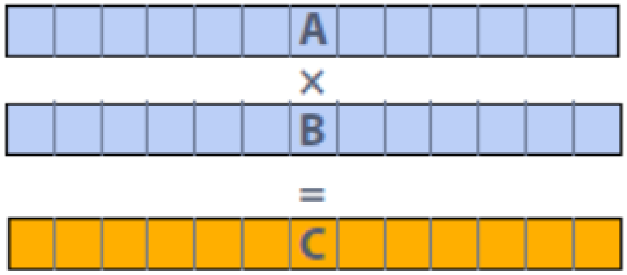
带宽受限!

- 如果处理器高速的请求数据，存储系统的带宽能力可能跟不上.
- 即使是多线程隐藏延迟，也无法解决此问题
- Overcoming bandwidth limits are a common challenge for application developers on throughput-optimized systems.

计算

- 假设GPU ：Clock rate 1.5 GHz,
- 16 个SIMD 处理器，each containing 16 个单精度浮点运算单元
- 如果不考虑带宽限制、并假设访存延迟全部被隐藏,
- what is the peak single-precision， floating-point throughput for this GPU in GLFOP/sec？

- 1.5 × 16 × 16 = 384 GFLOPS/s
- 如果每次操作需要输入两个单精度浮点数、输出一个单精度浮点数,
- 支持这个浮点数吞吐量 (assuming no temporal locality) 需要：
- 12 bytes/FLOP × 384 GFLOPs/s = 4.6 TB/s of memory bandwidth.



如果存储器带宽：100 GB/sec， Is this throughput sustainable given the memory bandwidth limitation?

回答：无法支持

带宽是一个关键资源

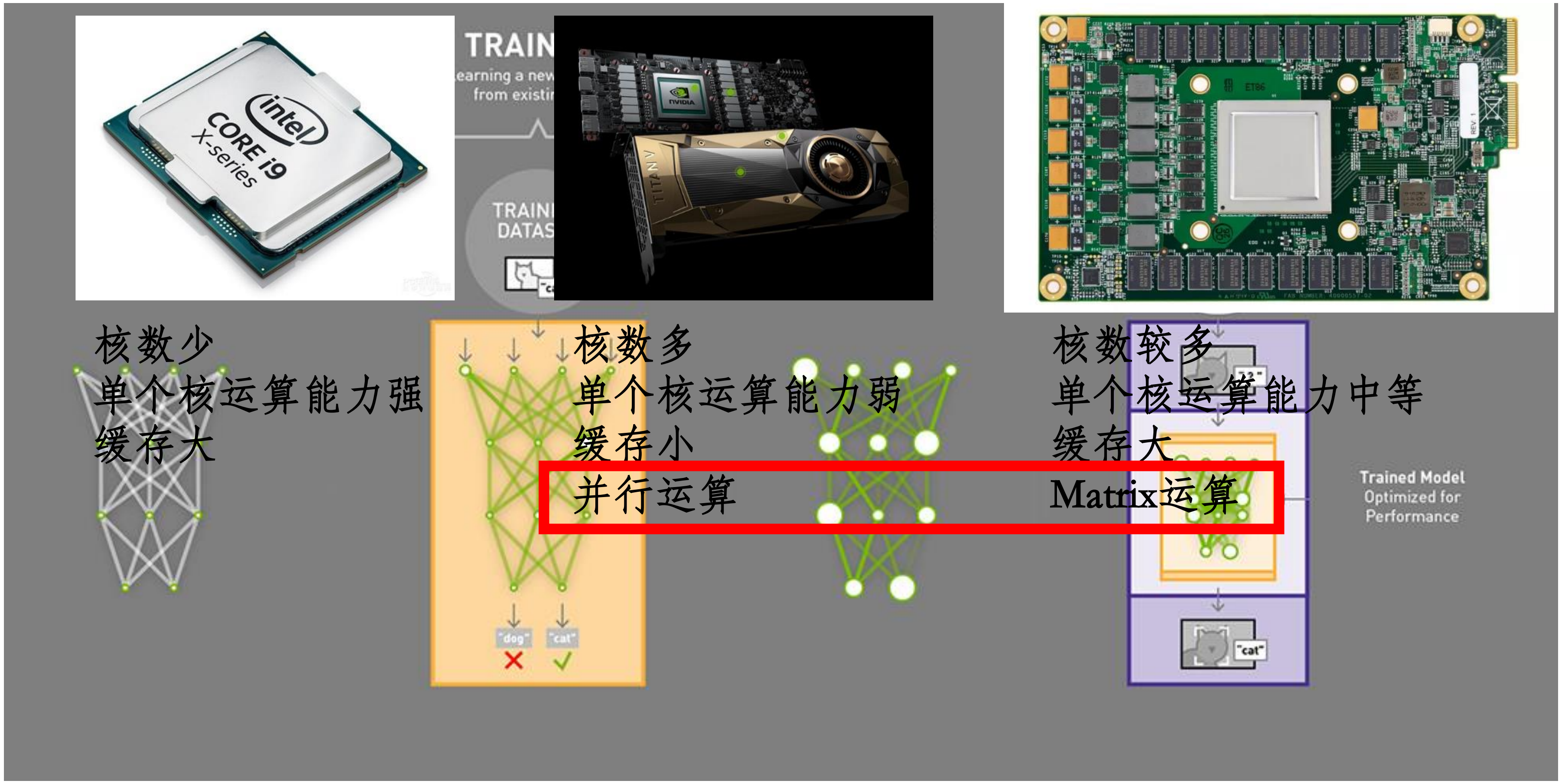
- 对程序员的要求：
  - ▪将计算组织为尽量少的从memory 读取数据
  - 相同线程加载的数据尽可能的重用 (traditional intra-thread temporal locality optimizations)
  - -多个线程之间共享数据 (inter-thread cooperation)
- arithmetic intensity” 密集度高的应用更适合
  - “arithmetic intensity” — ratio of math operations to data access operations in an instruction stream
  - 程序应该有 high arithmetic intensity，这样才能有效利用处理器的计算能力

Data Parallelism Summary

- Data Level Parallelism
  - "medium-grained" parallelism between ILP and TLP
  - Still one flow of execution (unlike TLP)
  - Compiler/programmer must explicitly expresses it (unlike ILP)
- Hardware support: new "wide" instructions (SIMD)
  - Wide registers, perform multiple operations in parallel
- Trends
  - Wider: 64-bit (MMX, 1996), 128-bit (SSE2, 2000),
    256-bit (AVX, 2011), 512-bit (Xeon Phi, 2013)
  - More advanced and specialized instructions
- GPUs
  - Embrace data parallelism via "SIMT" execution model
  - Becoming more programmable all the time
- Today's chips exploit parallelism at all levels: ILP, DLP, TLP

CPU vs GPU vs TPU

CPU vs GPU vs TPU



核数少
单个核运算能力强
缓存大

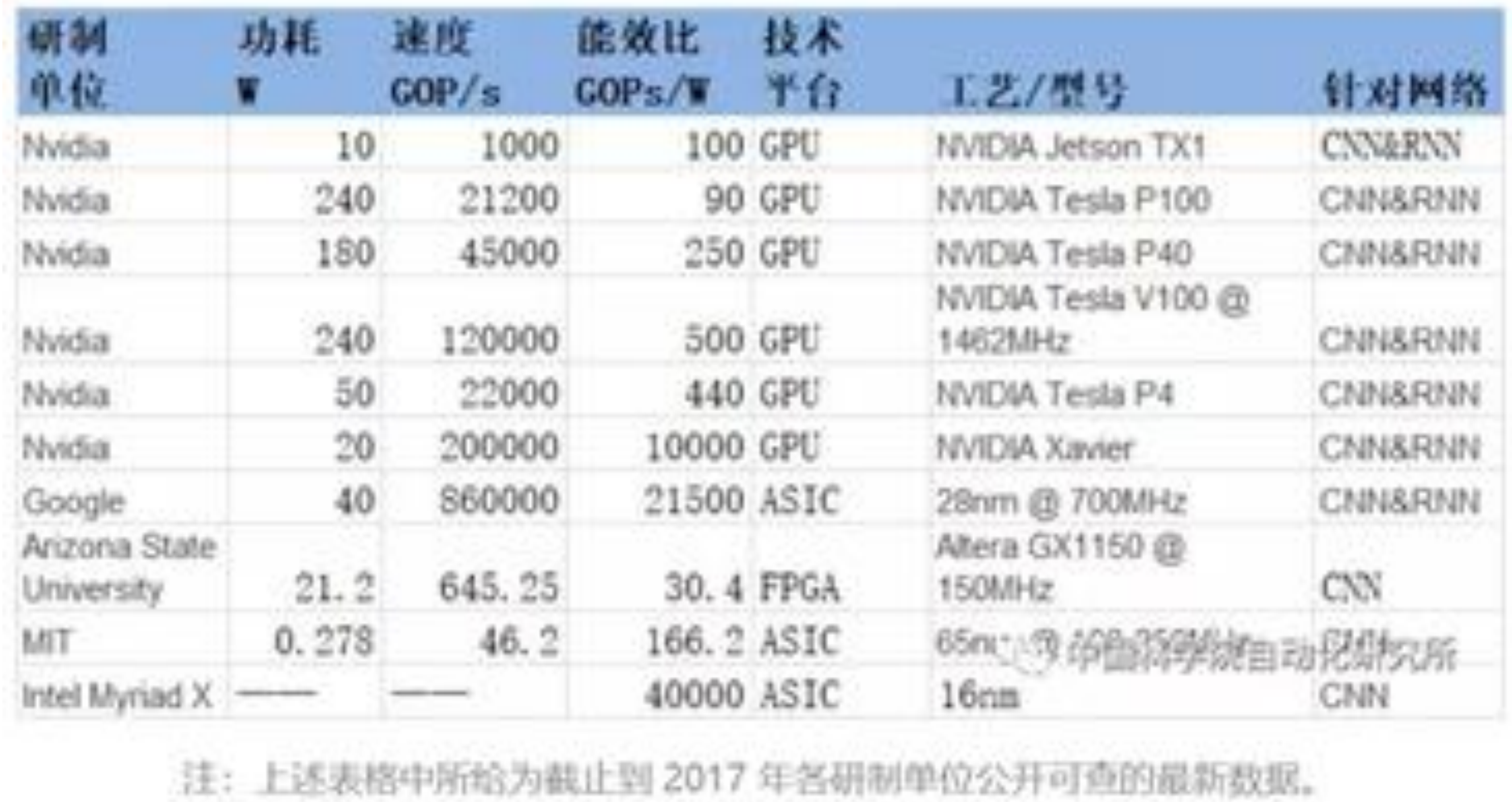
核数多
单个核运算能力弱
缓存小
并行运算

核数较多
单个核运算能力中等
缓存大
Matrix运算

Deep neural network训练时，GPU一定比CPU好么？ ——不一定

From： 柯晓荣 陈沈威 章志凌 秦炜 王辰(TPU group 1)

TPU 与GPU 处理神经网络的计算能力对比

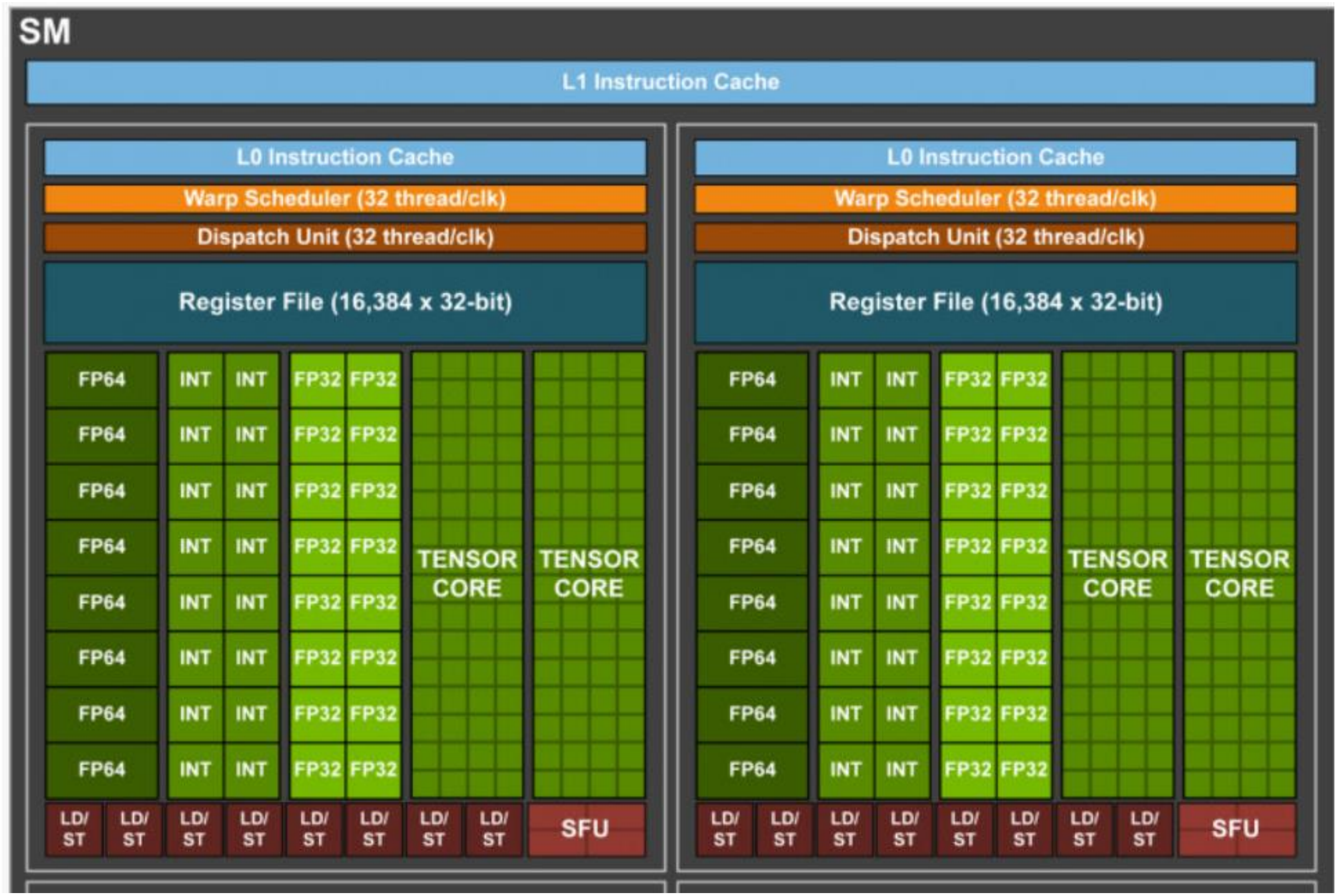


研制单位	功耗 W	速度 GOP/s	能效比 GOPs/W	技术平台	工艺/型号	针对网络
Nvidia	10	1000	100	GPU	NVIDIA Jetson TX1	CNN&RNN
Nvidia	240	21200	90	GPU	NVIDIA Tesla P100	CNN&RNN
Nvidia	180	45000	250	GPU	NVIDIA Tesla P40	CNN&RNN
Nvidia	240	120000	500	GPU	NVIDIA Tesla V100 @ 1462MHz	CNN&RNN
Nvidia	50	22000	440	GPU	NVIDIA Tesla P4	CNN&RNN
Nvidia	20	200000	10000	GPU	NVIDIA Xavier	CNN&RNN
Google	40	860000	21500	ASIC	28nm @ 700MHz	CNN&RNN
Arizona State University	21.2	645.25	30.4	FPGA	Altera GX1150 @ 150MHz	CNN
MIT	0.278	46.2	166.2	ASIC	65nm @ [illegible]250MHz	[illegible]
Intel Myriad X	——	——	40000	ASIC	16nm	CNN

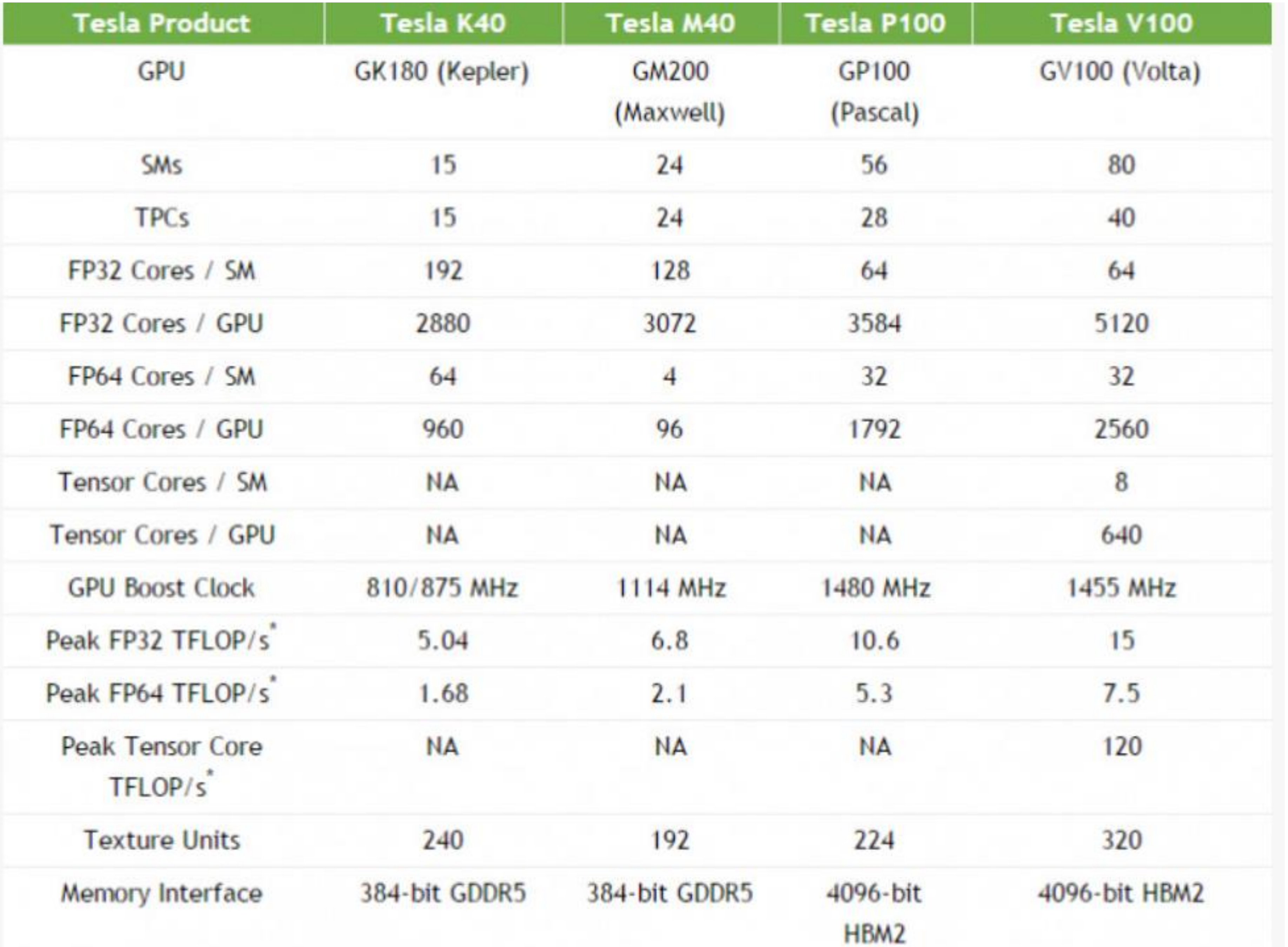
注：上述表格中所给为截止到 2017 年各研制单位公开可查的最新数据。

From Diannao group1 李赵睿 卢熠辉 孙健华 张宇鹏 周睿锋

Tesla V100 (2017)


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU

Tesla V100 与过去五年历代 Tesla 系列加速器的参数对比



Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1455 MHz
Peak FP32 TFLOP/s*	5.04	6.8	10.6	15
Peak FP64 TFLOP/s*	1.68	2.1	5.3	7.5
Peak Tensor Core TFLOP/s*	NA	NA	NA	120
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2

谢谢！