计算机体系结构概览

周学海
xhzhou@ustc.edu.cn
0551-63492149
中国科学技术大学

Summary-计算部分

- **Part1： 指令级并行**
- **Part2： 数据级并行**
- **Part3： 任务级并行**

- **核心目标：**
  - **如何提高计算机信息处理的效率?**
    - **信息处理： 计算、访存、通信**

  - **如何有效提高单条/多条指令流的执行效率?**

Part2：数据级并行

- **DLP技术基础：研究动机及种类**
  - 从效率瓶颈到DLP解决方案
  - DLP核心定义与并行范式对比
- **向量处理机：DLP的基础形态**
  - 向量处理器硬件结构
  - 向量指令集与代码示例
  - 向量处理器性能计算与优化
- **SIMD指令集扩展：DLP的轻量化落地**
- **GPU：DLP的高性能实战形态**
  - GPU硬件架构与SIMT模型
  - GPU编程模型与AI任务适配
- **AI专用处理器：DLP的AI场景定制形态**
  - AI专用处理器的发展历程：从学术探索到产业爆发
  - 典型AI专用处理器架构与DLP实现
- **四类DLP技术综合对比**

DLP技术基础：研究动机

- **提高性能的传统方法（挖掘ILP）的主要缺陷:**
  - 受到程序内在的并行性的限制
  - 提高流水线的时钟频率:
    - 提高时钟频率，有时导致CPI随着增加 (branches, other hazards)
  - 指令预取和译码:
    - 有时在每个时钟周期很难预取和译码多条指令
  - 提高Cache命中率 :
    - 在有些计算量较大的应用中（科学计算）需要大量的数据，其局部性较差，有些程序处理的是连续的媒体流(multimedia),其局部性也较差。

动机：DLP的兴起

- 应用需求和技术发展推动着体系结构的发展
- 图形、机器视觉、语音识别、机器学习等新的应用均需要大量的数值计算，其**算法通常具有数据并行特征**

- SIMD-based 结构 (vector-SIMD, subword-SIMD, SIMT/GPUs，专用AI处理器) 是执行这些算法的最有效途径

动机：SIMD结构的优势

- **SIMD 结构可有效地挖掘数据级并行:**
  - 基于矩阵运算的科学计算
  - 图像和声音处理
  - .....
- **SIMD比MIMD更节能**
  - 针对每组数据操作仅需要取指一次
  - SIMD对PMD(personal mobile devices)更具吸引力
- **SIMD 允许程序员继续以串行模式思维**

SIMD 结构的种类

- **向量体系结构**
- **多媒体SIMD指令集扩展**
- **Graphics Processor Units (GPUs)**

- **For x86 processors:**
  - 每2年增加2cores/chip
  - SIMD 宽度每4年翻一番
  - SIMD潜在加速比是MIMD的2倍

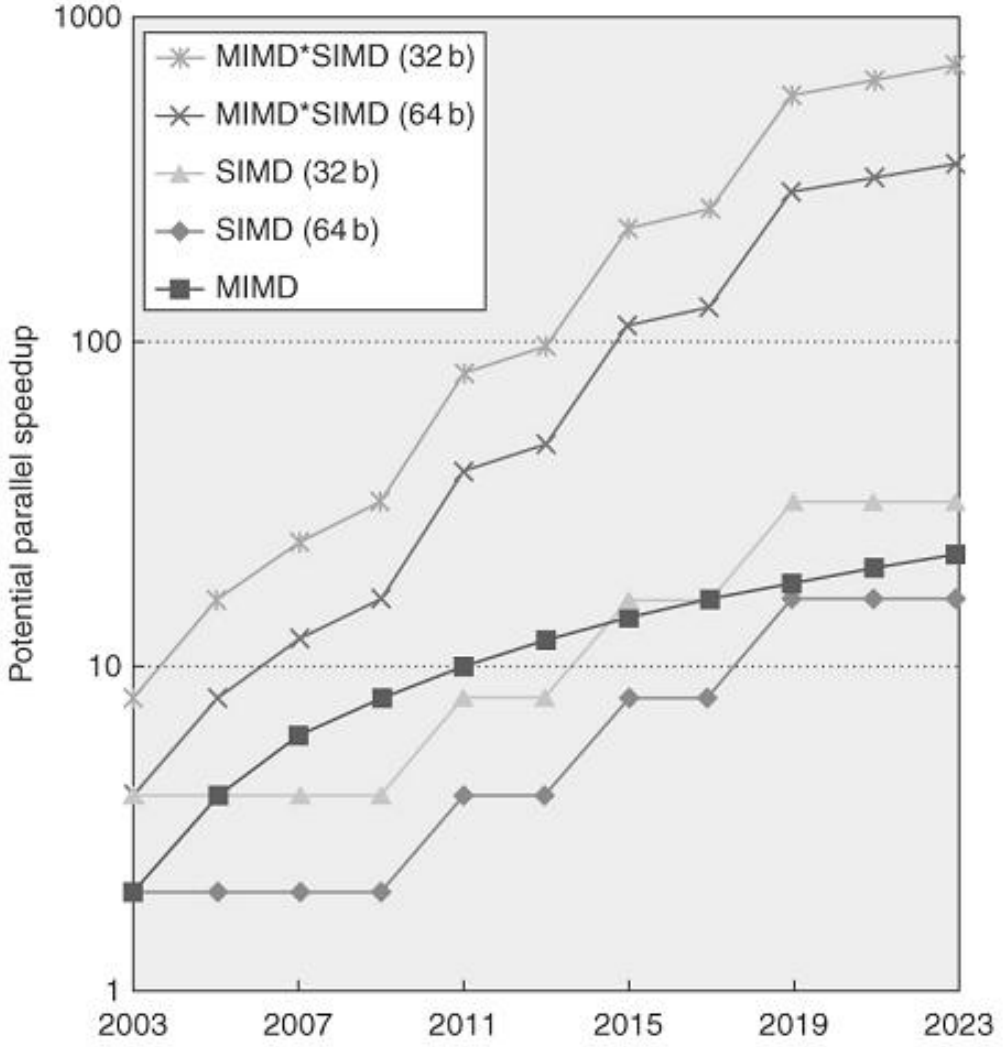


Figure 4.1 Potential speedup via parallelism from MIMD, SIMD, and both MIMD and SIMD over time for x86 computers. This figure assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD will double every four years.

Part2：数据级并行

- **DLP技术基础：研究动机及种类**
  - 从效率瓶颈到DLP解决方案
  - DLP核心定义与并行范式对比
- **向量处理机：DLP的基础形态**
  - 向量处理器硬件结构
  - 向量指令集与代码示例
  - 向量处理器性能计算与优化
- **SIMD指令集扩展：DLP的轻量化落地**
- **GPU：DLP的高性能实战形态**
  - GPU硬件架构与SIMT模型
  - GPU编程模型与AI任务适配
- **AI专用处理器：DLP的AI场景定制形态**
  - AI专用处理器的发展历程：从学术探索到产业爆发
  - 典型AI专用处理器架构与DLP实现
- **四类DLP技术综合对比**

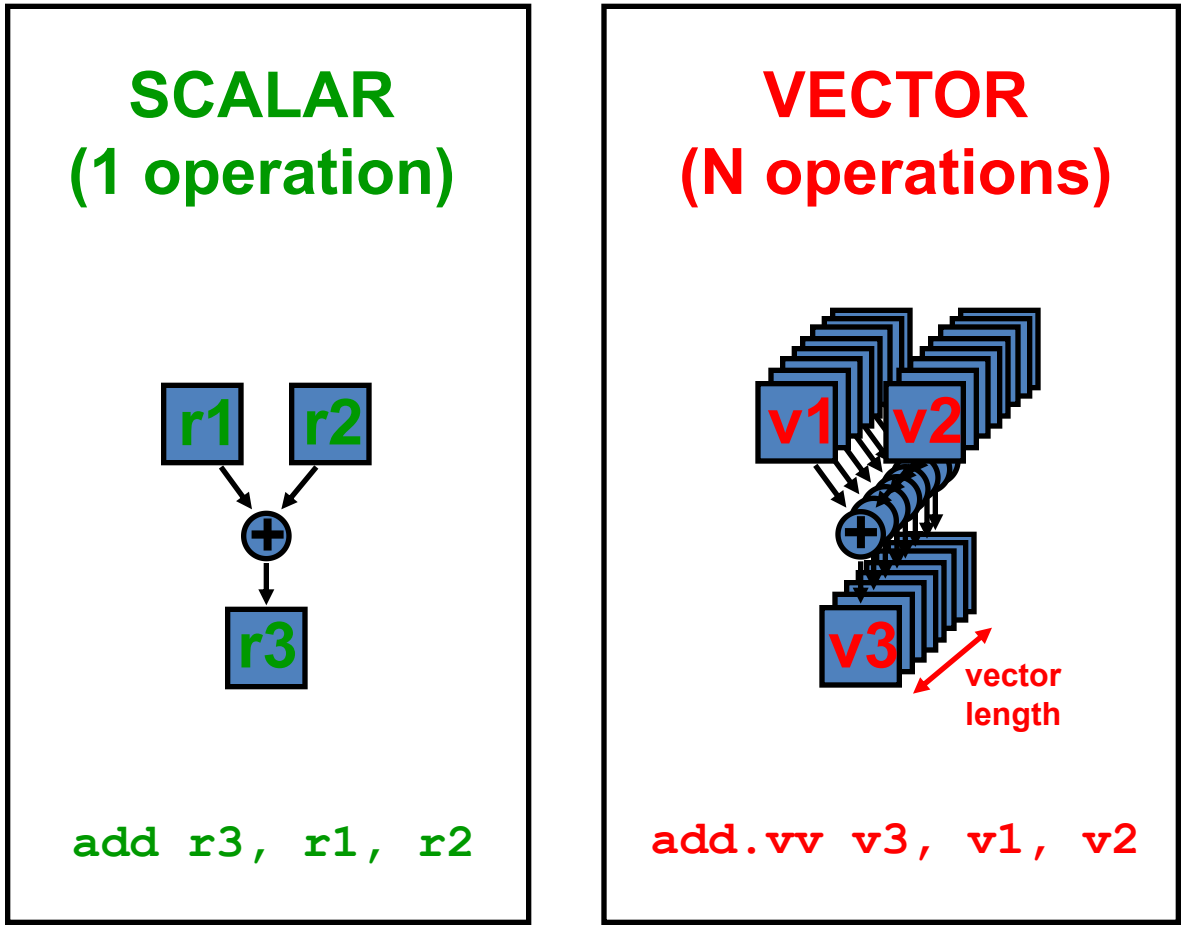
向量处理机模型

向量处理机模型

性能评估

向量处理模型

- **向量处理机具有更高层次的操作，一条向量指令可以处理N个或N对操作数（处理对象是向量）**



向量处理机的基本特性

- **基本思想：两个向量的对应分量进行运算，产生一个结果向量**
- **简单的一条向量指令包含了多个操作=> fewer instruction fetches**
- **每一结果独立于前面的结果**
  - 长流水线，编译器保证操作间没有相关性
  - **硬件仅需检测两条向量指令间的相关性**
  - 较高的时钟频率
- **向量指令以已知的模式访问存储器**
  - 可有效发挥多体交叉存储器的优势
  - 可通过重叠减少存储器操作的延时 -（例如：一个向量包含 64个元素）
  - 不需要数据Cache! (仅使用指令cache)
- **在流水线控制中减少了控制相关**

向量处理机的基本结构

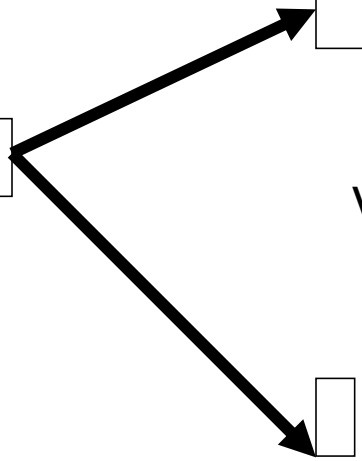
- ***memory-memory vector processors:*** **所有的向量操作是存储器到存储器**
- ***vector-register processors:*** **除了load 和store操作外，所有的操作是向量寄存器与向量寄存器间的操作**
  - 向量机的Load/Store结构
  - 1980年以后的所有的向量处理机都是这种结构： Cray, Convex, Fujitsu, Hitachi, NEC
  - 我们也主要针对这种结构

Vector Memory-Memory versus Vector Register Machines

- **存储器-存储器型向量机所有指令操作的操作数来源于存储器**
- **第一台向量机 CDC Star-100 ('73) and TI ASC ('71), 是存储器-存储器型机器**
- **Cray-1 ('76) 是第一台寄存器型向量机**

Example Source Code

```
for (i=0; i<N; i++)
{
   C[i] = A[i] + B[i];
   D[i] = A[i] - B[i];
}
```



Vector Memory-Memory Code

```
ADDV C, A, B
SUBV D, A, B
```

Vector Register Code

```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

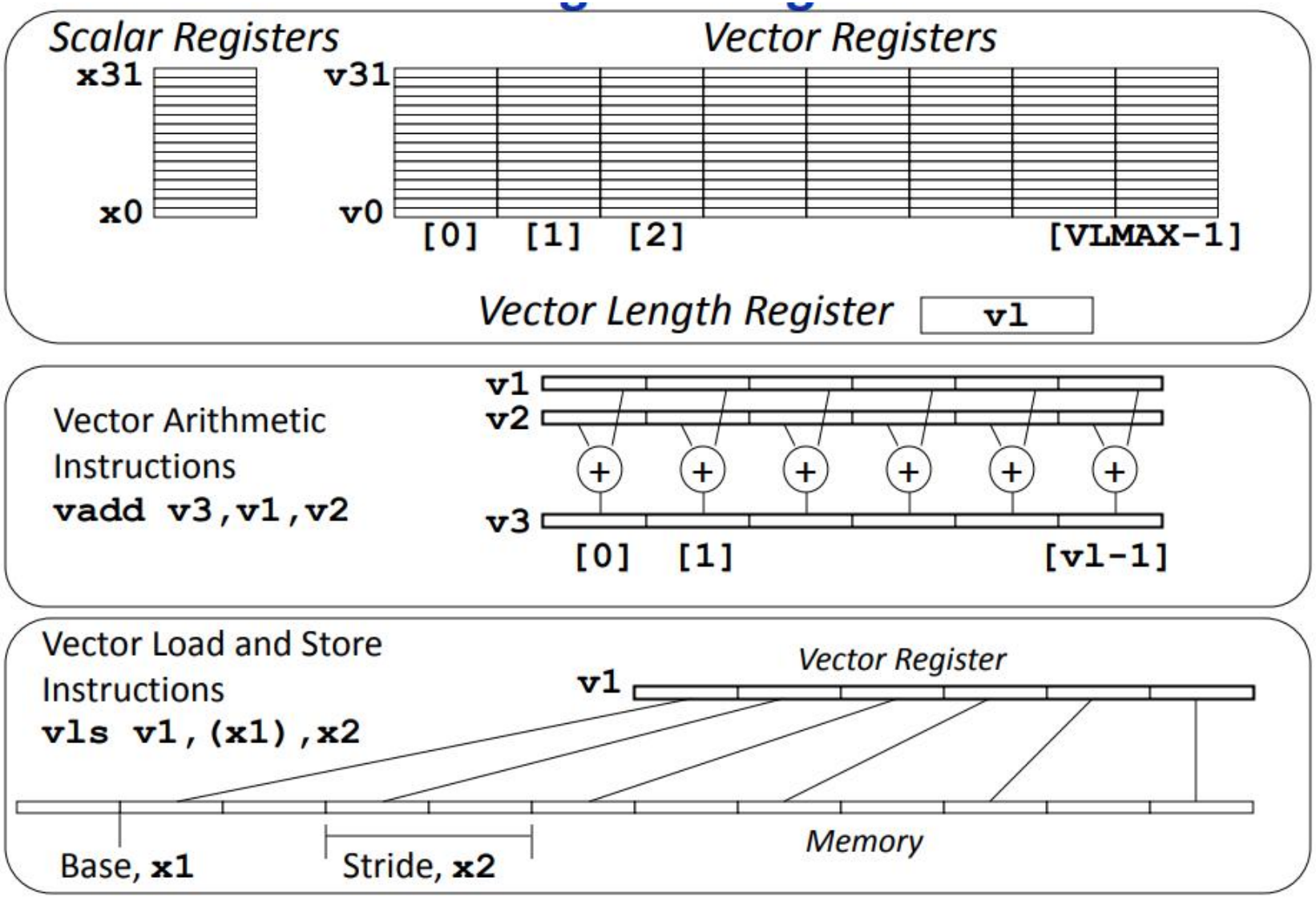
Vector Memory-Memory vs. Vector Register Machines

- **存储器-存储器型向量机 (VMMA) 需要更高的存储器带宽**
  - All operands must be read in and out of memory
- **VMMA结构使得多个向量操作重叠执行较困难**
  - Must check dependencies on memory addresses
- **VMMA启动时间更长**
  - CDC Star-100 在向量元素小于100时，标量代码的性能高于向量化代码
- **CDC Cray-1后续的机器 (Cyber-205, ETA-10) 都是寄存器型向量机**

Vector Instruction Set Advantages

- **格式紧凑**
  - 一条指令包含N个操作
- **表达能力强，一条指令能告诉硬件:**
  - N个操作之间无相关性
  - 使用同样的功能部件
  - 访问不相交的寄存器
  - 与前面的操作以相同模式访问寄存器
  - 访问存储器中的连续块 (unit-stride load/store)
    或以已知的模式访问存储器 (strided load/store)
- **可扩展性好**
  - 可以在多个并行的流水线上运行同样的代码 (lanes)

Vector Programming Model



Vector Code Example

```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  li x4, 64
loop:
  fld f1, 0(x1)
  fld f2, 0(x2)
  fadd.d f3,f1,f2
  fsd f3, 0(x3)
  addi x1, x1, 8
  addi x2, x2, 8
  addi x3, x3, 8
  subi x4, x4, 1
  bnez x4, loop
```

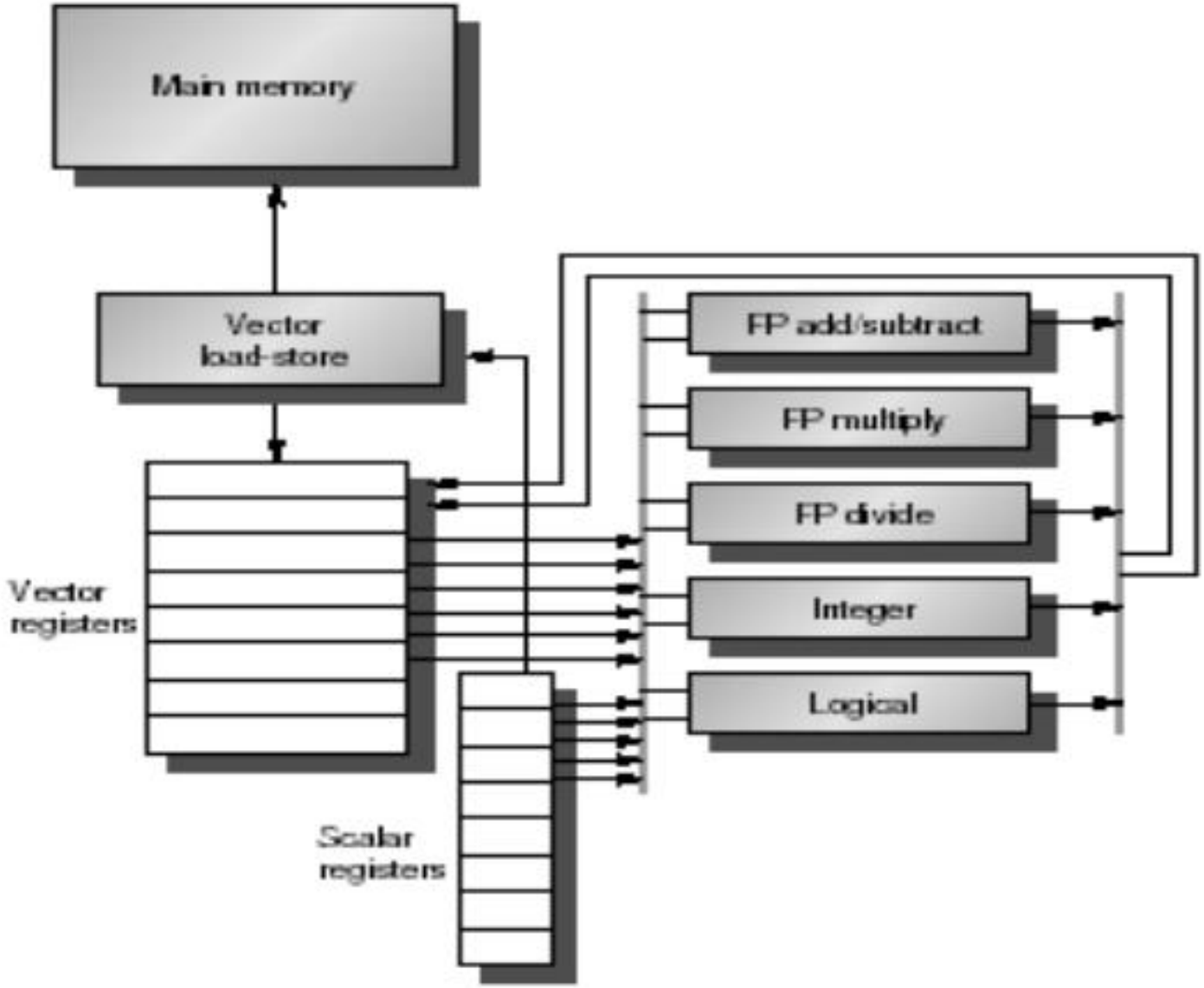
```
# Vector Code
  li x4, 64
  vsetvl x4  ;设置向量长度
  vld v1, (x1)
  vld v2, (x2)
  vadd v3,v1,v2
  vst v3, (x3)
```

Vector Instructions (DLXV)

	Instr.	Operands	Operation Comment
ADDV	V1,V2,V3	V1=V2+V3	vector + vector
ADDSV	V1,F0,V2	V1=F0+V2	scalar + vector
MULTV	V1,V2,V3	V1=V2xV3	vector x vector
MULSV	V1,F0,V2	V1=F0xV2	scalar x vector
LV	V1,R1	V1=M[R1..R1+63]	load, stride=1
LVWS	V1,R1,R2	V1=M[R1..R1+63*R2]	load, stride=R2
LVI	V1,R1,V2	V1=M[R1+V2i,i=0..63]	indir.("gather")
CeqV	VM,V1,V2	VMASKi = (V1i=V2i)?	comp. setmask
MOV	VLR,R1	Vec. Len. Reg. = R1	set vector length
MOV	VM,R1	Vec. Mask = R1	set vector mask

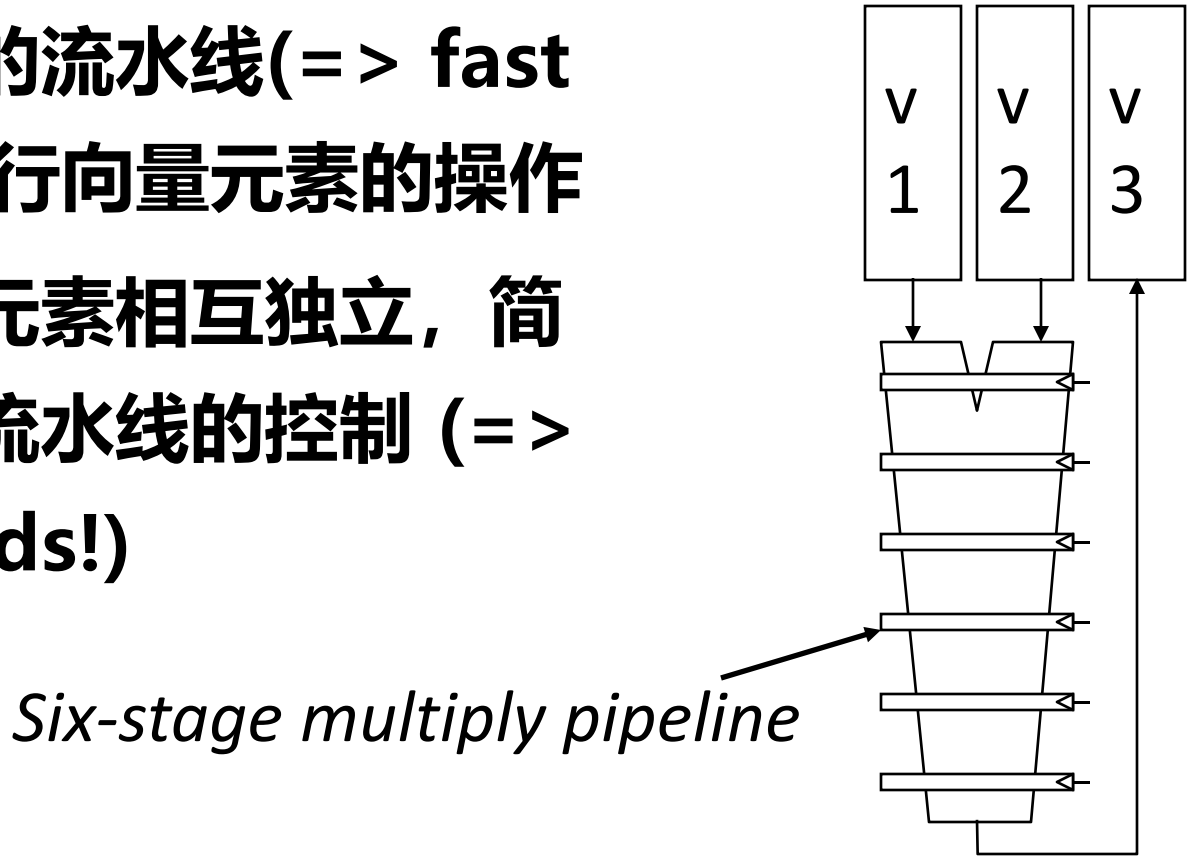
向量处理机的基本组成单元

- ***Vector Register*: 固定长度的一块区域，存放单个向量**
  - 至少2个读端口和一个写端口（一般最少16个读端口，8个写端口）
  - 典型的有8-32 向量寄存器，每个寄存器存放64到128个64位元素
- ***Vector Functional Units (FUs)*: 全流水化的，每一个clock启动一个新的操作**
  - 一般4到8个FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical，shift；可能有些重复设置的部件
- ***Vector Load-Store Units (LSUs)*: 全流水化地load 或store一个向量，可能会配置多个LSU部件**
- ***Scalar registers*: 存放单个元素用于标量处理或存储地址**
- **用交叉开关连接(Cross-bar) FUs , LSUs, registers**

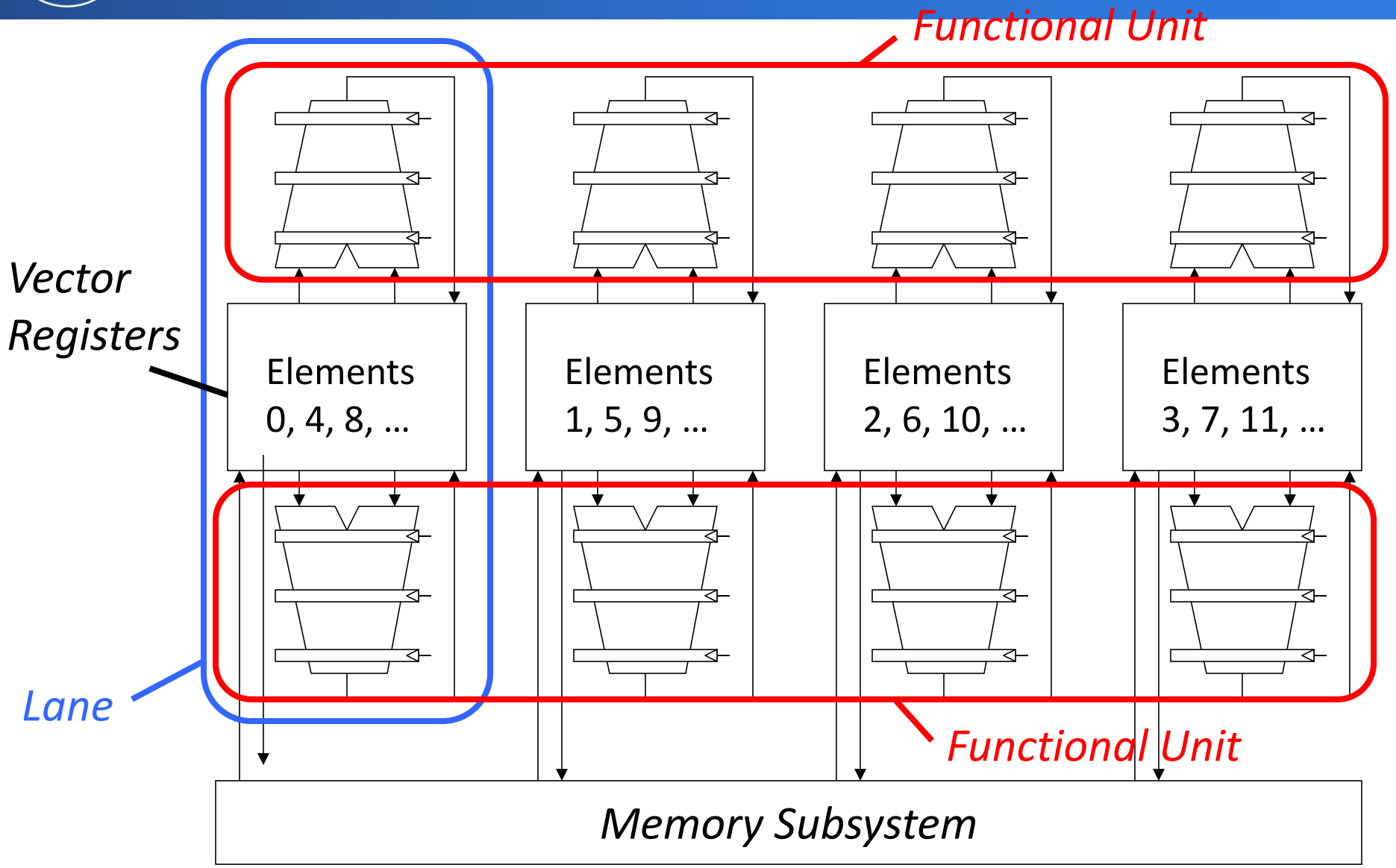

Main memory
Vector load-store
FP add/subtract
FP multiply
FP divide
Integer
Logical
Vector registers
Scalar registers

Vector Arithmetic Execution

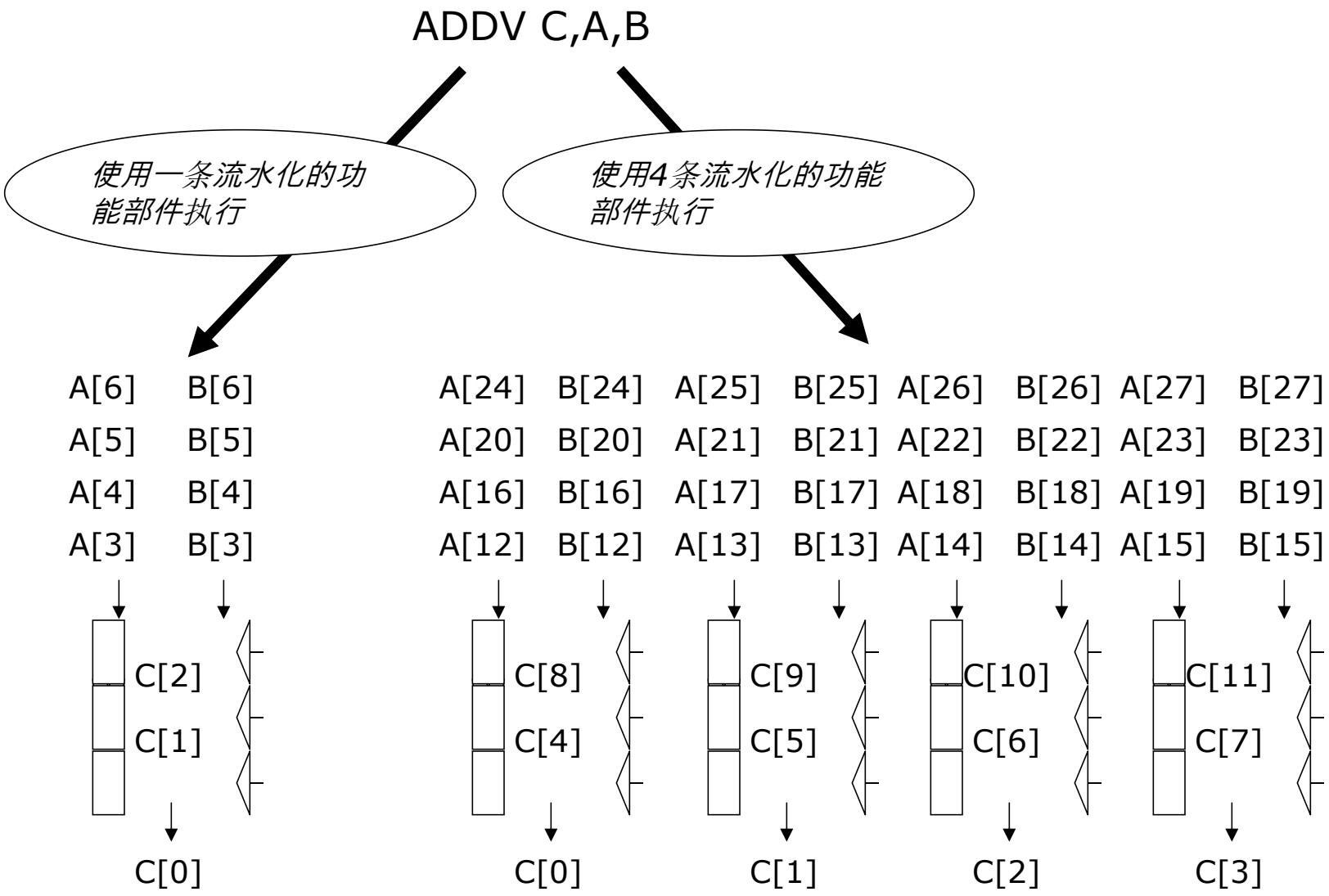
- **使用较深的流水线(=> fast clock) 执行向量元素的操作**
- **由于向量元素相互独立，简化了深度流水线的控制 (=> no hazards!)**



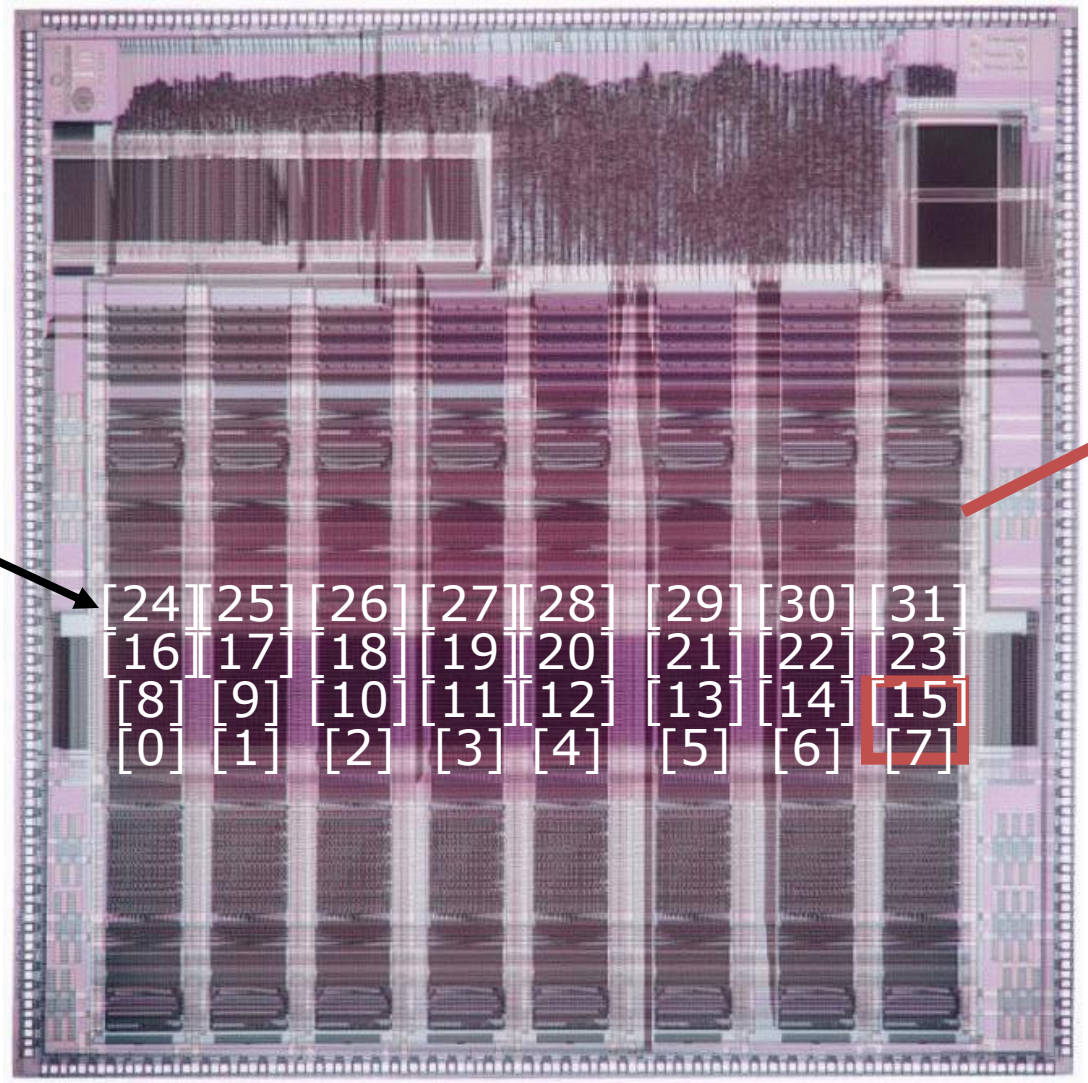
Vector Unit Structure



Vector Instruction Execution



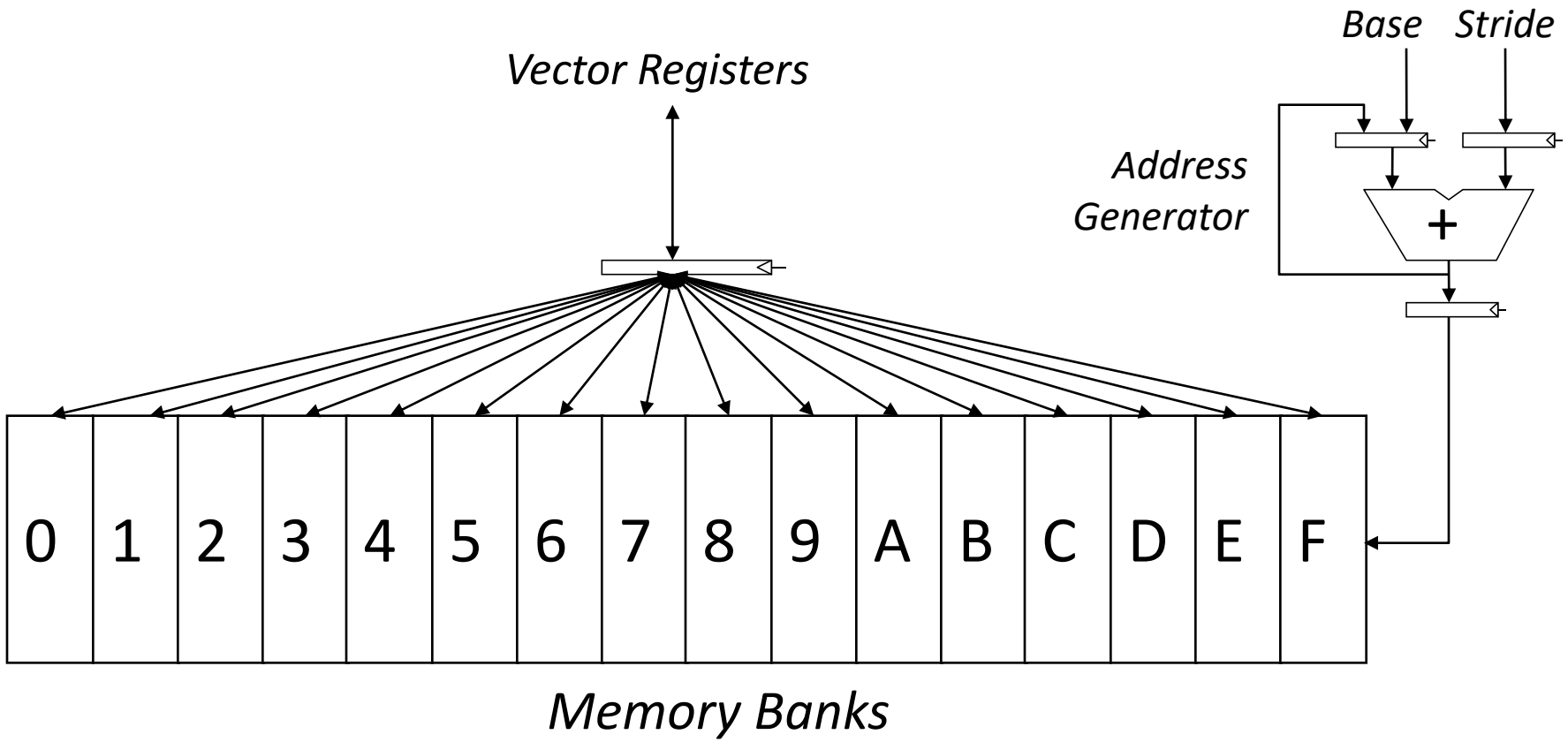
T0 Vector Microprocessor (UCB/ICSI, 1995)



Interleaved Vector Memory System

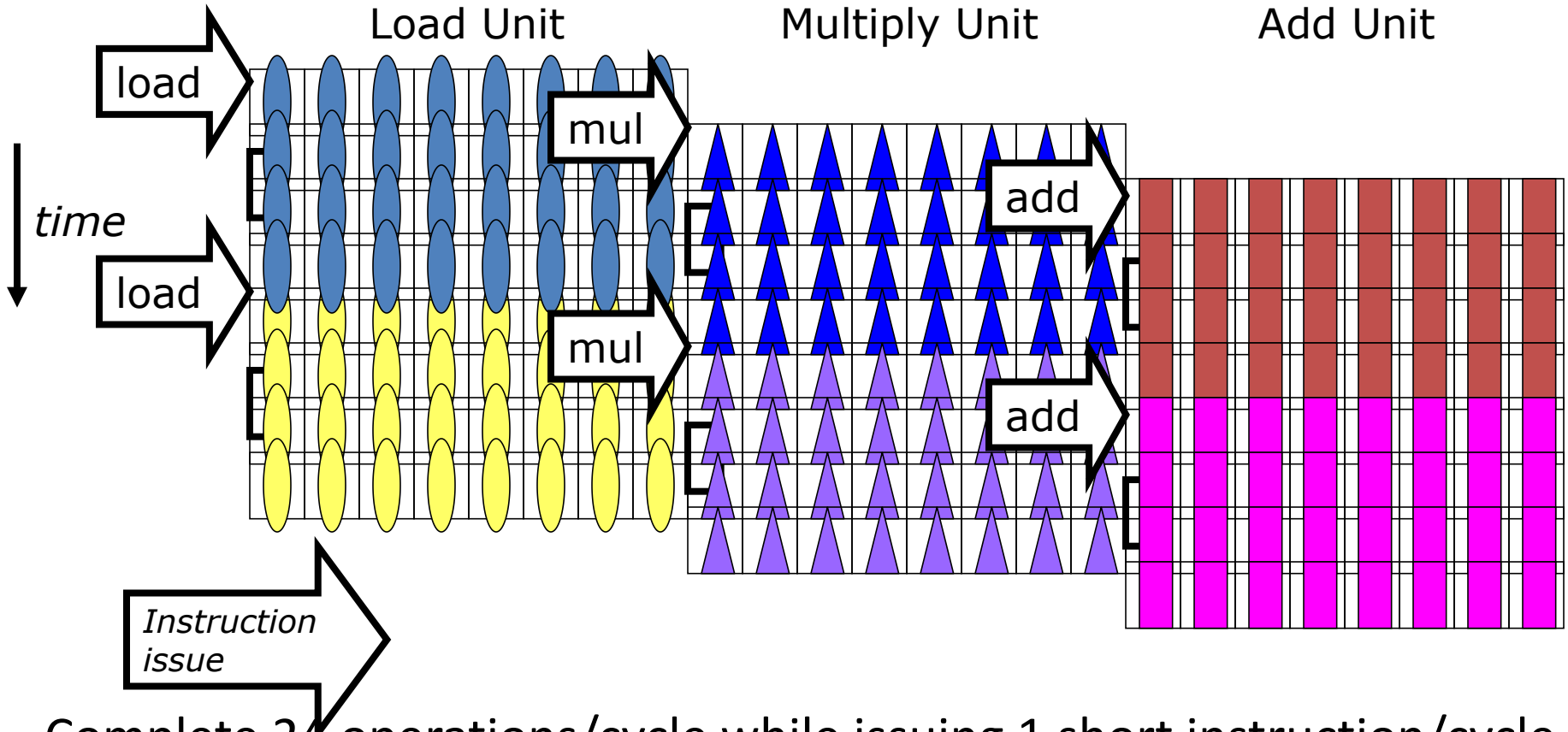
Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- *Bank busy time*: Time before bank ready to accept next request



Vector Instruction Parallelism

- **多条向量指令可重叠执行(链接技术)**
  - 例如：每个向量 32 个元素，8 lanes（车道）



Complete 24 operations/cycle while issuing 1 short instruction/cycle

向量处理机模型

向量处理机模型

性能评估

Vector Execution Time

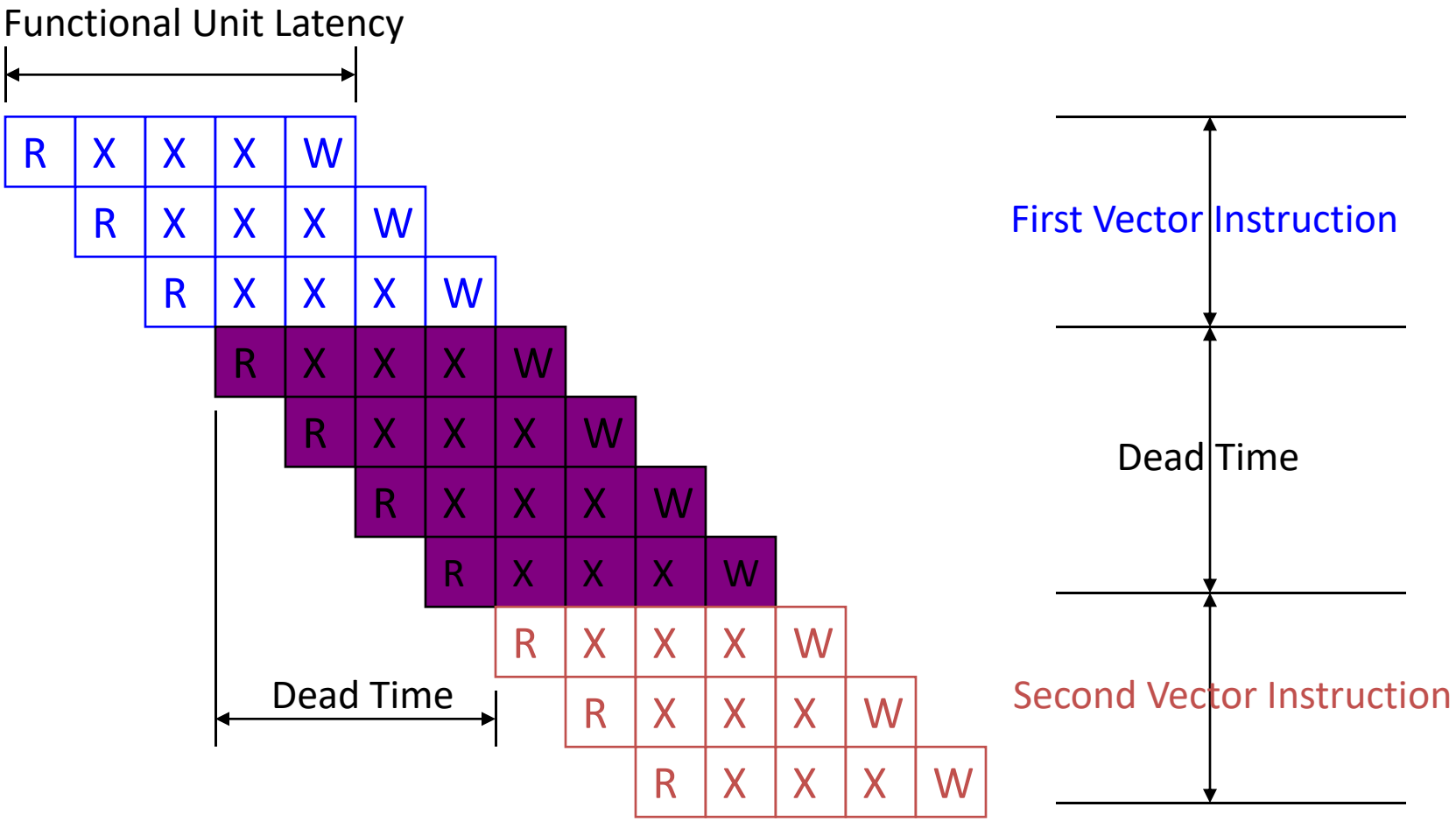
- *Time* = f(vector length, data dependencies, struct. hazards)
- *Initiation rate*: 功能部件消耗向量元素的速率
- *Convoy:* 可在同一时钟周期开始执行的指令集合 (no structural or data hazards)
- *Chime:* 执行一个*convoy*所花费的大致时间（approx. time)
- *m convoys take m chimes*;
  - 如果每个向量长度为*n*, 那么*m*个*convoys* 所花费的时间是*m*个*chimes*
  - 每个*chime*所花费的时间是*n*个*clocks*，该程序所花费的总时间大约为 ***m x n clock cycles*** (忽略额外开销; 当向量长度较长时这种近似是合理的)

```
1: LV    V1,Rx     ;load vector X
2: MULV  V2,F0,V1  ;vector-scalar mult.
   LV    V3,Ry     ;load vector Y
3: ADDV  V4,V2,V3  ;add
4: SV    Ry,V4     ;store the result
```

4 convoys, 1 lane, VL=64
=> 4 x 64 = 256 clocks
(or 4 clocks per result)

Vector Startup

- **向量启动时间由两部分构成**
  - 功能部件延时：一个操作通过功能部件的时间
  - 截止时间或恢复时间（dead time or recovery time）：运行下一条向量指令的间隔时间



VMIPS Start-up Time

Start-up time: FU 部件流水线的深度

Operation	Start-up penalty (from CRAY-1)
Vector load/store	12
Vector multiply	7
Vector add	6

Assume convoys don't overlap; vector length = n

Convoy	*Start*	*1st result*	*last result*	
1. LV	0	12	11+n	(12+n-1)
2. MULV, LV	12+n	12+n+7	18+2n	*Multiply startup*
	12+n	12+n+12	23+2n	*Load start-up*
3. ADDV	24+2n	24+2n+6	29+3n	*Wait convoy 2*
4. SV	30+3n	30+3n+12	41+4n	*Wait convoy 3*

review：向量处理机结构

- **向量处理机基本概念**
  - 基本思想：两个向量的对应分量进行运算，产生一个结果向量
- **向量处理机基本特征**
  - VSIW-一条指令包含多个操作
  - 单条向量指令内所包含的操作相互独立
  - 以已知模式访问存储器-多体交叉存储系统
  - 控制相关少
- **向量处理机基本结构**
  - **向量指令并行执行**
  - **向量运算部件的执行方式-流水线方式**
  - **向量部件结构-多“道”结构-多条运算流水线**
- **向量处理机性能评估**
  - **向量指令流执行时间: Convey, Chimes, Start-up time**
  - **其他指标：** R_∞ , $N_{1/2}$, N_V

Vector Length

- 当向量的长度不是64时（假设向量寄存器的长度是64）怎么办?
- vector-length register (VLR) 控制特定向量操作的长度, 包括向量的load/store. (当然一次操作的向量的长度不能 > 向量寄存器的长度) 例如:

```
      do 10    i = 1, n
10    Y(i) = a * X(i) + Y(i)
```

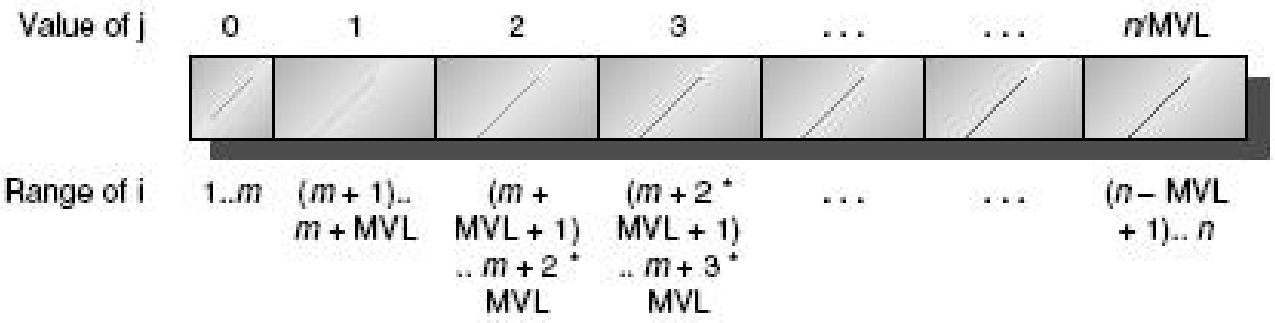
n的值只有在运行时才能知道

n > Max. Vector Length (MVL)怎么办?

Strip Mining（分段开采）

- 假设Vector Length > Max. Vector Length (MVL)?
- Strip mining: 产生新的代码，使得每个向量操作的元素数 ≤ MVL
- 第一次循环做最小片(n mod MVL), 以后按VL = MVL操作

```
low = 1
VL = (n mod MVL)  /*find the odd size piece*/
do 1  j = 0, (n / MVL)  /*outer loop*/
    do 10 i = low, low+VL-1  /*runs for length VL*/
            Y(i) = a*X(i) + Y(i)  /*main operation*/
10  continue
    low = low+VL  /*start of next vector*/
    VL = MVL  /*reset the length to max*/
1   continue
```



Strip Mining的向量执行时间计算

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

试计算A=B×s，其中A,B为长度为200的向量（每个向量元素占8个字节），s是一个标量。向量寄存器长度为64。各功能部件的启动时间如前所述，求总的执行时间，（T_{loop} = 15)

```
        ADDI R2,R0,#1600    ;total # bytes in vector
        ADD R2,R2,Ra        ;address of the end of A vector
        ADDI R1,R0,#8       ;loads length of 1st segment
        MOVI2S  VLR,R1      ;load vector length in VLR
        ADDI R1,R0,#64      ;length in bytes of 1st segment
        ADDI R3,R0,#64      ;vector length of other segments
Loop:   LV  V1,Rb           ;load B
        MULSV V2,V1,Fs      ;vector * scalar
        SV Ra,V2            ;store A
        ADD Ra,Ra,R1        ;address of next segment of A
        ADD  Rb,Rb,R1       ;address of next segment of B
        ADDI R1,R0,#512     ;load byte offset next segment
        MOVI2S VLR,R3       ;set length to 64 elements
        SUB R4,R2,Ra        ;at the end of A?
        BNEZ R4,Loop        ;if not, go back
```

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

Tstart = 12 + 7 + 12 = 31

T200 = 660+4*31 = 784

每一元素的执行时间 = 784/200 = 3.9

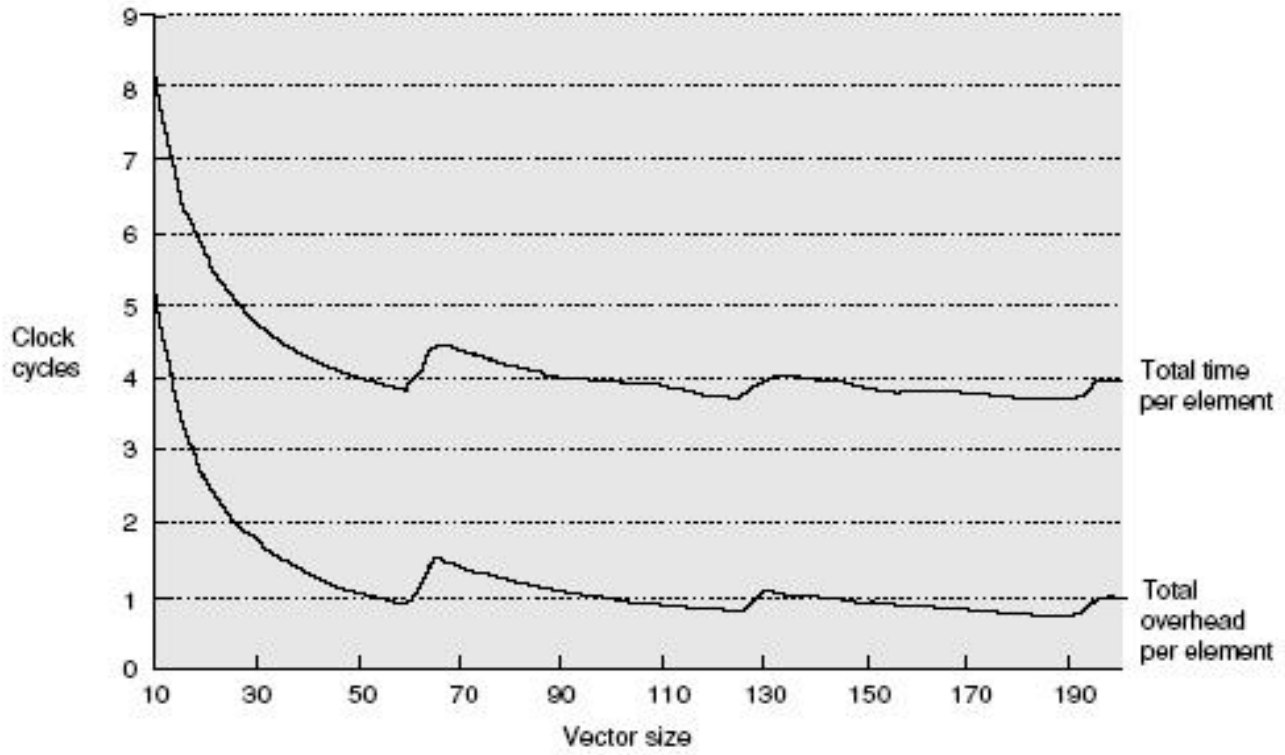


Figure G.9 The total execution time per element and the total overhead time per element versus the vector length for the example on page G-19. For short vectors the total start-up time is more than one-half of the total time, while for long vectors it reduces to about one-third of the total time. The sudden jumps occur when the vector length crosses a multiple of 64, forcing another iteration of the strip-mining code and execution of a set of vector instructions. These operations increase T_n by $T_{loop} + T_{start}$.

Common Vector Metrics

- **R_∞: 当向量长度为无穷大时的向量流水线的最大性能。常在评价峰值性能时使用，单位为MFLOPS**
  - 实际问题是向量长度不会无穷大，start-up的开销还是比较大的
  - R_n 表示向量长度为n时的向量流水线的性能
- **$N_{1/2}$: 达到R∞ 一半的值所需的向量长度，是评价向量流水线start-up 时间对性能的影响。**
- **N_V: 向量流水线方式的工作速度优于标量串行方式工作时所需的向量长度临界值。**
  - 该参数既衡量建立时间，也衡量标量、向量速度比对性能的影响

DAXPY (Y = a × X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector

```
LD      F0,a
ADDI    R4,Rx,#512 ;last address to load
loop: LD      F2, 0(Rx)   ;load X(i)
      MULTD   F2,F0,F2    ;a*X(i)
      LD      F4, 0(Ry)   ;load Y(i)
      ADDD    F4,F2,F4    ;a*X(i) + Y(i)
      SD      F4,0(Ry)    ;store into Y(i)
      ADDI    Rx,Rx,#8    ;increment index to X
      ADDI    Ry,Ry,#8    ;increment index to Y
      SUB     R20,R4,Rx   ;compute bound
      BNZ     R20,loop    ;check if done
```

```
LD      F0,a      ;load scalar a
LV      V1,Rx     ;load vector X
MULTS   V2,F0,V1  ;vector-scalar mult.
LV      V3,Ry     ;load vector Y
ADDV    V4,V2,V3  ;add
SV      Ry,V4     ;store the result
```

578 (2+9*64) vs.
321 (1+5*64) ops (1.8X)

578 (2+9*64) vs.
6 instructions (96X)

64 operation vectors + no loop overhead

also 64X fewer pipeline hazards

Summary：向量体系结构

- **向量处理机基本概念**
  - 基本思想：两个向量的对应分量进行运算，产生一个结果向量
- **向量处理机基本特征**
  - VSIW-一条向量指令包含多个操作
  - 单条向量指令内所包含的操作相互独立
  - 以已知模式访问存储器-多体交叉存储系统
  - 控制相关少
- **向量处理机基本结构**
  - **向量指令并行执行**
  - **向量运算部件的执行方式-流水线方式**
  - **向量部件结构-多“道”结构-多条运算流水线**
- **向量处理机性能评估**
  - **向量指令流执行时间: Convey, Chimes, Start-up time**
  - **其他指标：** R_∞ , $N_{1/2}$, N_V
- **向量处理机性能优化**
  - 链接技术
  - 条件执行
  - 稀疏矩阵

Part2：数据级并行

- **DLP技术基础：研究动机及种类**
  - 从效率瓶颈到DLP解决方案
  - DLP核心定义与并行范式对比
- **向量处理机：DLP的基础形态**
  - 向量处理器硬件结构
  - 向量指令集与代码示例
  - **向量处理器性能计算与优化**
- **SIMD指令集扩展：DLP的轻量化落地**
- **GPU：DLP的高性能实战形态**
  - GPU硬件架构与SIMT模型
  - GPU编程模型与AI任务适配
- **AI专用处理器：DLP的AI场景定制形态**
  - AI专用处理器的发展历程：从学术探索到产业爆发
  - 典型AI专用处理器架构与DLP实现
- **四类DLP技术综合对比**

向量处理器性能优化

- 存储器访问
- 链接技术
- 条件执行
- 稀疏矩阵

Vector Stride

- 假设数组元素为双字，处理顺序相邻的元素在存储器中不顺序存储。例如

```
      do 10 i = 1,100
        do 10 j = 1,100
           A(i,j) = 0.0
           do 10 k = 1,100
10            A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

- B 或 C 的两次访问不会相邻 (相隔800 bytes)
- ***stride***: 向量中相邻元素间的距离
  => **LVWS** (load vector with stride) instruction
- Strides => 会导致体冲突
  (e.g., stride = 32 and 16 banks)

Memory operations

- **Load/store 操作成组地在寄存器和存储器之间移动数据**
- **三类寻址方式**
  - Unit stride (单步长)
    - Fastest

    LV V1,R1 //V1=M[R1..R1+63] load, stride=1
  - Non-unit (constant) stride (常数步长)

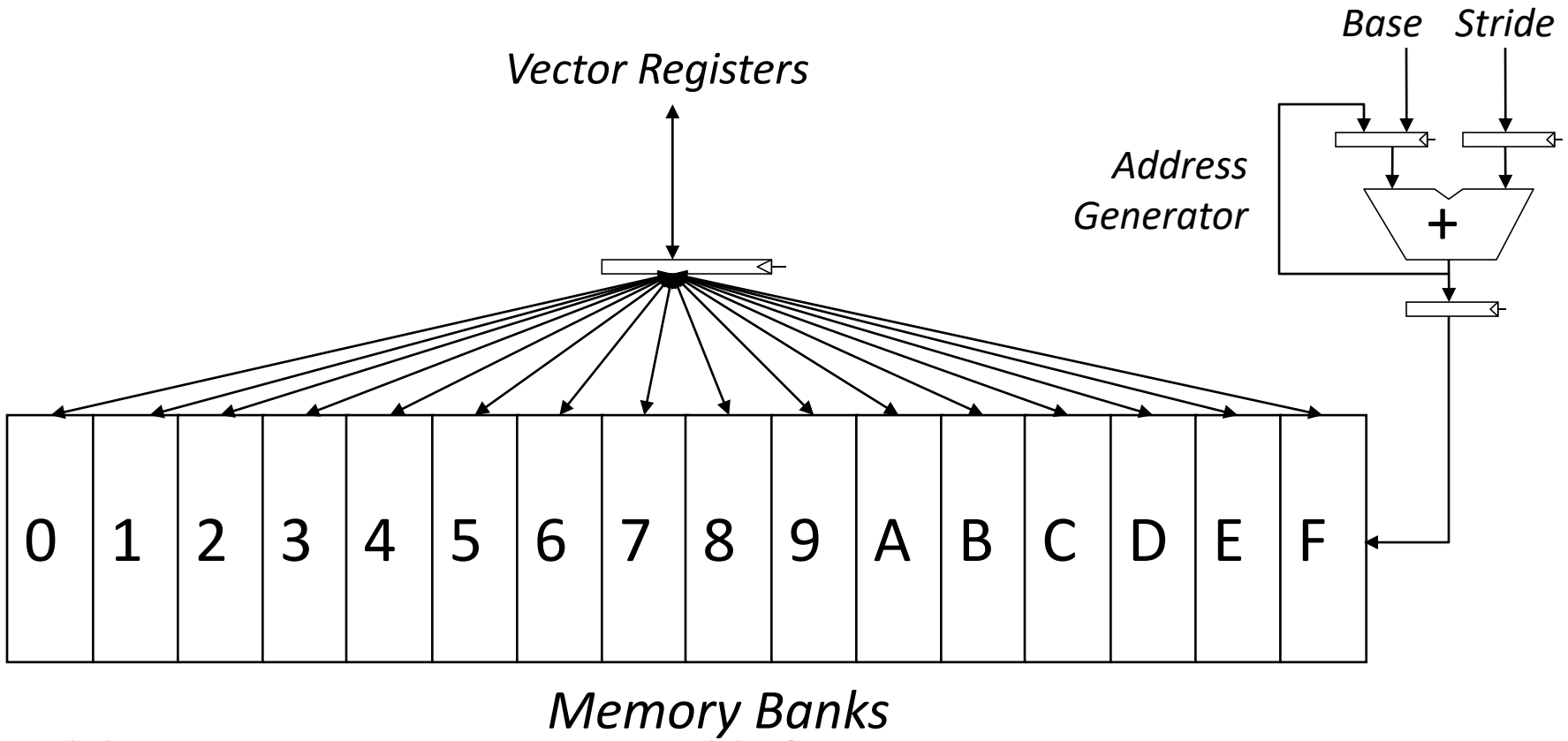
    LVWS V1,R1,R2 //V1=M[R1..R1+63*R2] load, stride=R2
  - Indexed (gather-scatter) (间接寻址)

    LVI V1,R1,V2 //V1=M[R1+V2i,i=0..63] indir.("gather")

    - 等价于寄存器间接寻址方式
    - 对稀疏矩阵有效
    - 用于向量化操作的指令增多

Interleaved Vector Memory System

- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency
  - *Bank busy time*: 存储体准备接收下一请求的间隔时间
  - ***If stride = 1 & 连续的向量元素交叉存储在不同存储体中 & 存储体的数量 >= 存储体的访问时间（Bank busy time + latency),*** *则该存储器组织的吞吐率可达到：* ***1 element/cycle***



Example(AppF F-15)

假设我们要从字节地址为136处开始获取一个包含（连续存储）64个元素(DW)的向量，并且一次内存访问需要6个时钟。我们必须有多少存储体才能支持每个时钟周期存取一个元素?访问每个存储体的地址是多少？每个元素何时到达CPU?

	Bank							
Cycle no.	0	1	2	3	4	5	6	7
0		136						
1		busy	144					
2		busy	busy	152				
3		busy	busy	busy	160			
4		busy	busy	busy	busy	168		
5		busy	busy	busy	busy	busy	176	
6			busy	busy	busy	busy	busy	184
7	192			busy	busy	busy	busy	busy
8	busy	200			busy	busy	busy	busy
9	busy	busy	208			busy	busy	busy
10	busy	busy	busy	216			busy	busy
11	busy	busy	busy	busy	224			busy
12	busy	busy	busy	busy	busy	232		
13		busy	busy	busy	busy	busy	240	
14			busy	busy	busy	busy	busy	248
15	256			busy	busy	busy	busy	busy
16	busy	264			busy	busy	busy	busy

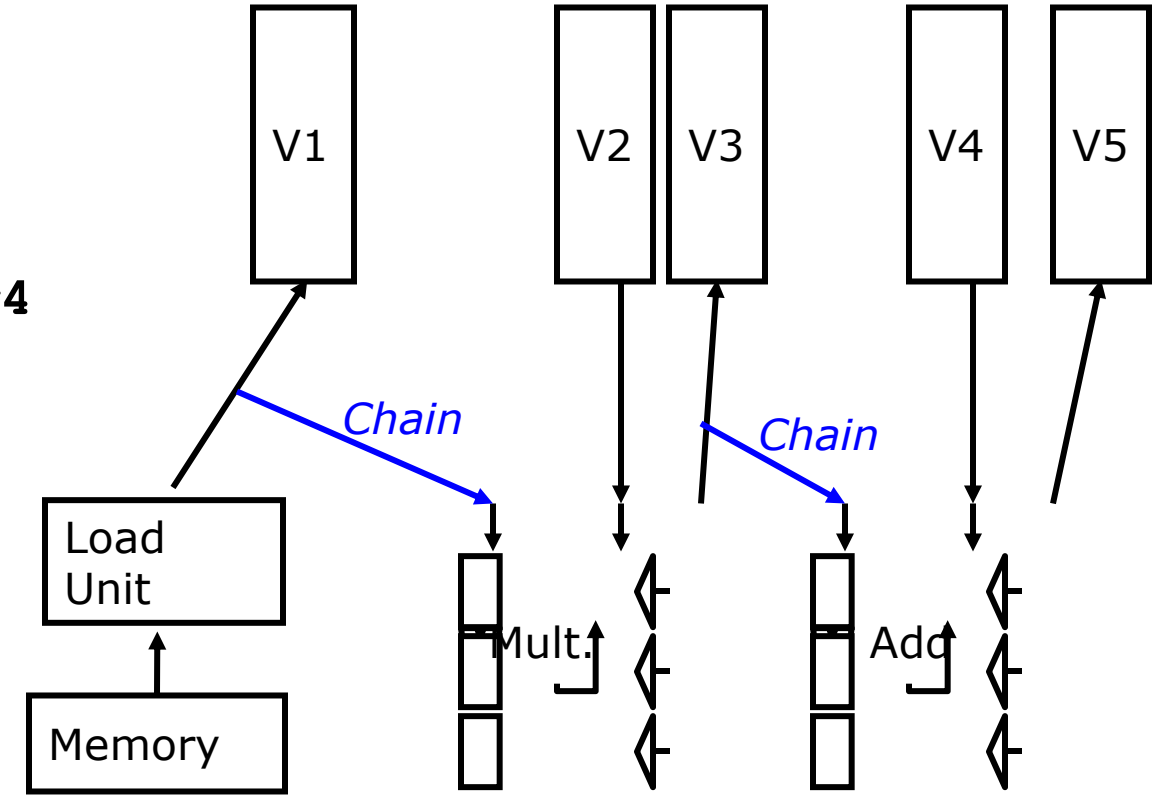
Figure F.7 Memory addresses (in bytes) by bank number and time slot at which access begins. Each memory bank latches the element address at the start of an access and is then busy for 6 clock cycles before returning a value to the CPU. Note that the CPU cannot keep all eight banks busy all the time because it is limited to supplying one new address and receiving one data item each cycle.

Bank# = (address/8) mod 8

Vector Opt#1: Vector Chaining

- 寄存器定向路径的向量机版本
- 首次在Cray-1上使用

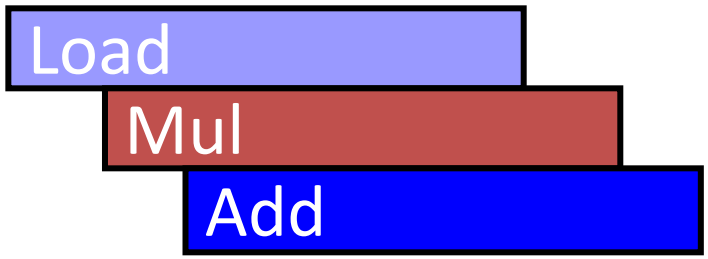
```
LV   v1
MULV v3,v1,v2
ADDV v5,v3, v4
```



Vector Chaining Advantage

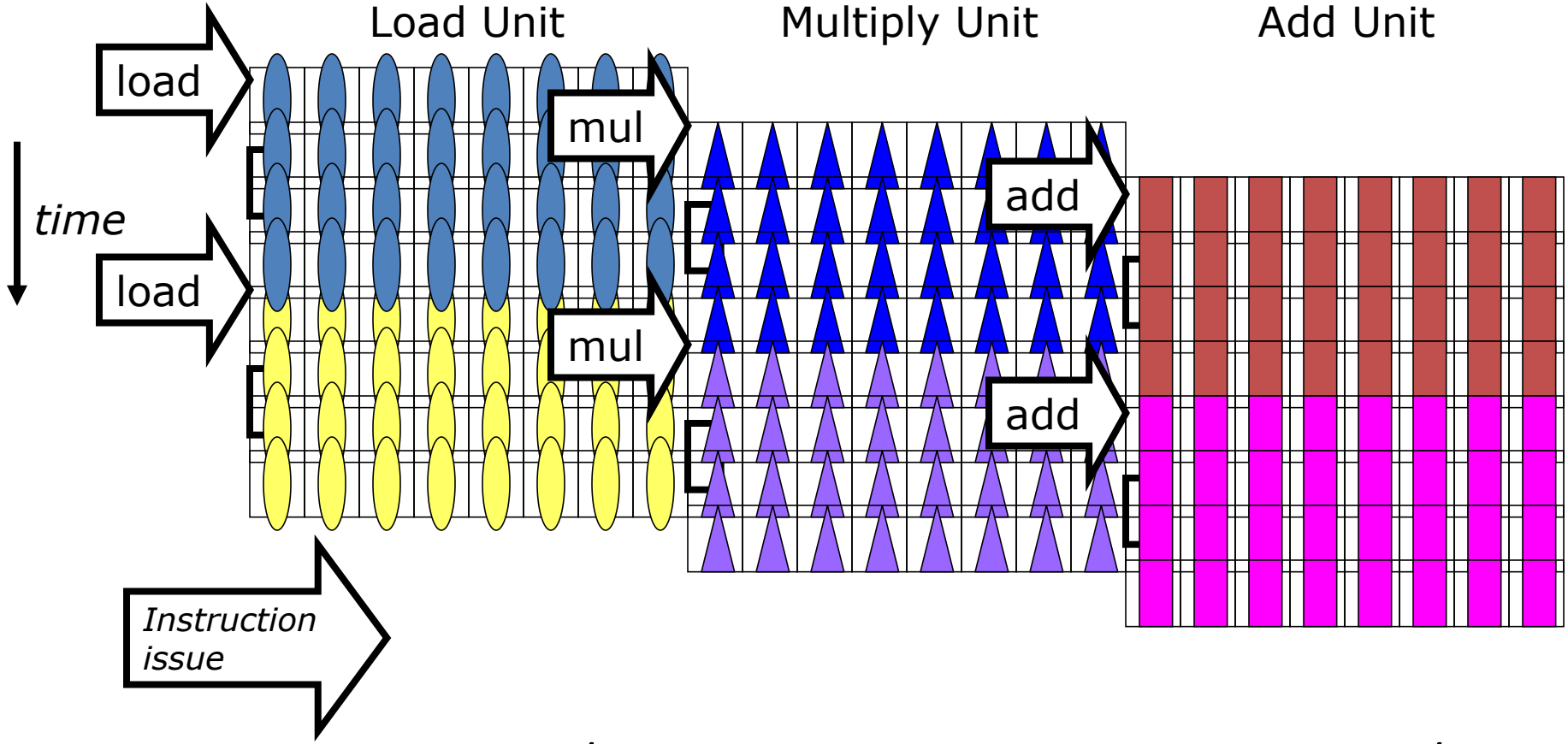
- 不采用链接技术，必须处理完前一条指令的最后一个元素，才能启动下一条相关的指令

- 采用链接技术，前一条指令的第一个结果出来后，就可以启动下一条相关指令的执行



Vector Instruction Parallelism

- **多条向量指令可重叠执行(链接技术)**
  - 例如：每个向量 32 个元素，8 lanes（车道）



Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Opt #2: Conditional Execution

- **Suppose:**

```
    do 100 i = 1, 64
        if (A(i) .ne. 0) then
            A(i) = A(i) - B(i)
        endif
100 continue
```

- ***vector-mask control*** **使用长度为MVL的布尔向量控制向量指令的执行**
- ***当vector-mask register*** **使能时，向量指令操作仅对vector-mask register中 对应位为1的分量起作用**

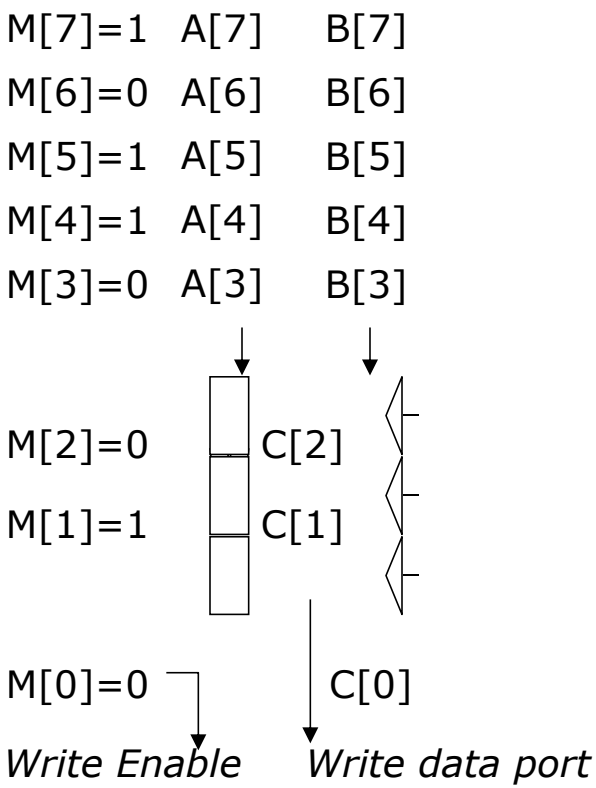
```
LV   V1,Ra          ;  load vector A into V1
LV V2,Rb            ;  load vector B
L.D F0,#0           ; load FP zero into F0
SNEVS.D V1,F0       ;sets VM(i) to 1 if V1(i)!=F0
SUBV.D V1,V1,V2     ;subtract under vector mask
CVM                  ;set the vector mask to all 1s
SV Ra,V1            ;store the result in A
```

- **使用vector-mask寄存器的缺陷**
  - 简单实现时，条件不满足时向量指令仍然需要花费时间
  - 有些向量处理器带条件的向量执行仅控制向目标寄存器的写操作，可能会有除法错。

Masked Vector Instructions

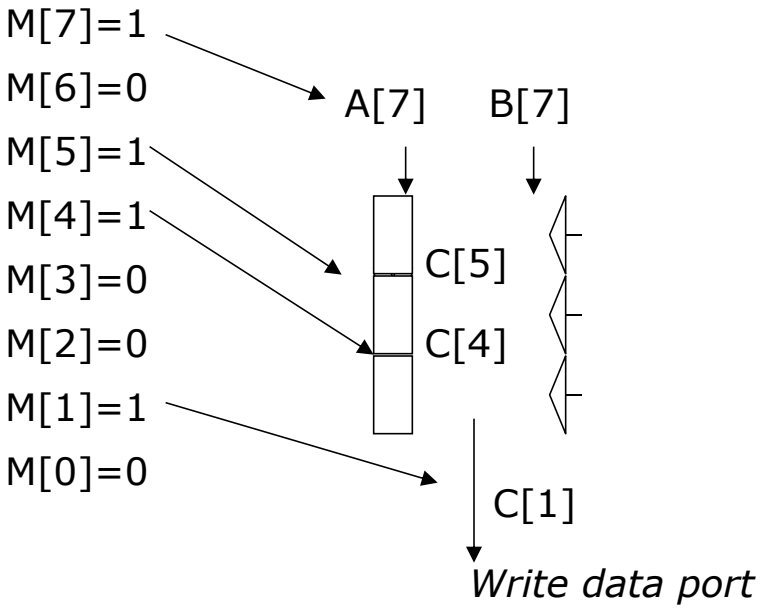
Simple Implementation

- execute all N operations, turn off result writeback according to mask



Density-Time Implementation

- scan mask vector and only execute elements with non-zero masks



Vector Opt #3: Sparse Matrices

- **Suppose:**

```
                do      100 i = 1,n
100             A(K(i)) = A(K(i)) + C(M(i))
```

- ***gather*** **(LVI) operation 使用*index vector*中给出的偏移再加基址来读取 => a nonsparse vector in a vector register**
- **这些元素以密集的方式操作完成后，再使用同样的index vector存储到稀疏矩阵的对应位置**

```
LV Vk, Rk               ; load index  K
LVI  Va, (Ra+Vk)        ; gather  load A(K(i))
LV Vm,Rm                ;  load index M
LVI Vc, (Rc+Vm)         ;  gather  load C(M(i))
ADDVV.D Va,Va,Vc        ;  ADDVV
SVI (Ra+Vk), Va         ; scatter store A(K(i))
```

- **VMIP还提供了CVI 创建索引向量，(index 0, 1xm, 2xm, ..., 63xm)**
  - CVI V1，R1 //创建索引向量V1，将0，1*R1，2*R1，....63*R1存储到V1
- **这些操作编译时可能无法完成。主要原因：编译器无法预知K(i)以及是否有数据相关**

Sparse Matrix Example

- Cache (1993) vs. Vector (1988)

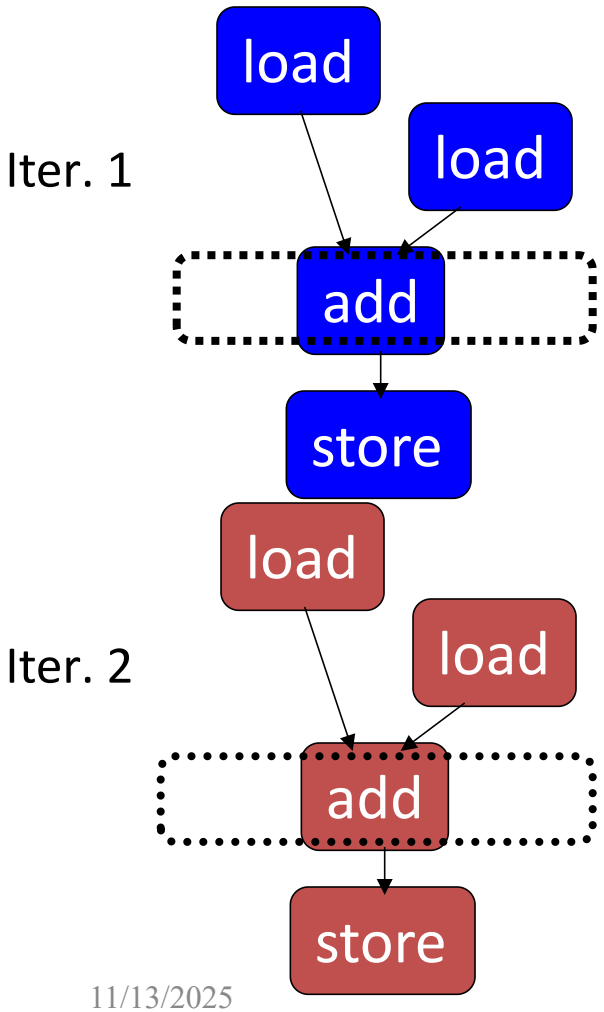
	IBM RS6000	Cray YMP
Clock	72 MHz	167 MHz
Cache	256 KB	0.25 KB
Linpack	140 MFLOPS	160 (1.1)
Sparse Matrix (Cholesky Blocked)	17 MFLOPS	125 (7.3)

- Cache: 1 address per cache block (32B to 64B)
- Vector: 1 address per element (4B)

Automatic Code Vectorization

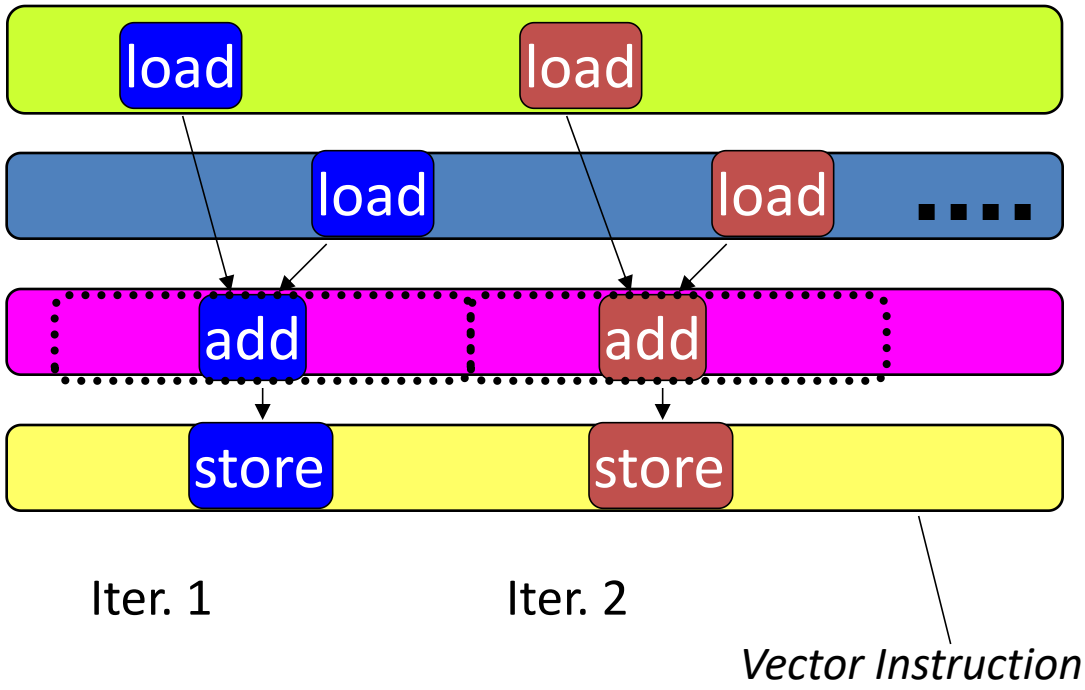
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Scalar Sequential Code



Time

Vectorized Code



向量化是指在编译期间对操作重定序⇒ 需要进行大量的循环相关分析

Summary：向量体系结构

- **向量处理机基本概念**
  - 基本思想：两个向量的对应分量进行运算，产生一个结果向量
- **向量处理机基本特征**
  - VSIW-一条指令包含多个操作
  - 单条向量指令内所包含的操作相互独立
  - 以已知模式访问存储器-多体交叉存储系统
  - 控制相关少
- **向量处理机基本结构**
  - **向量指令并行执行**
  - **向量运算部件的执行方式-流水线方式**
  - **向量部件结构-多“道”结构-多条运算流水线**
- **向量处理机性能评估**
  - **向量指令流执行时间: Convey, Chimes, Start-up time**
  - **其他指标：** R_∞ , $N_{1/2}$, N_V
- **向量处理机性能优化**
  - 链接技术
  - 条件执行
  - 稀疏矩阵

Vector/SIMD Processing Summary

- **Vector/SIMD 机器适合挖掘数据级并行**
  - 同样的操作作用于不同的数据元素
  - 向量内的元素操作独立，可有效提高性能，简化设计
- **性能的提升受限于代码的向量化**
  - 标量操作限制了向量机的性能
  - Amdahl's Law
- **很多ISA包含SIMD操作指令**
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

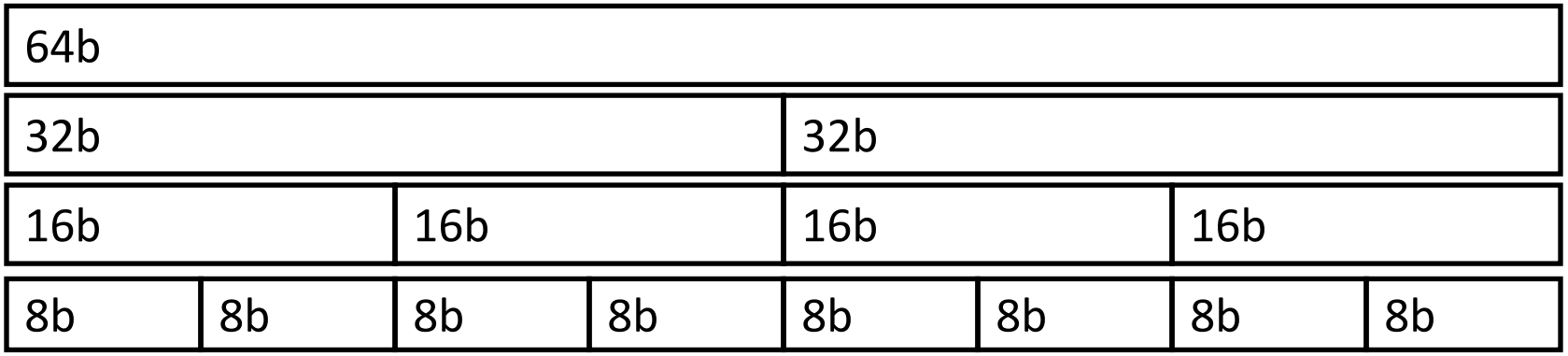
Part2：数据级并行

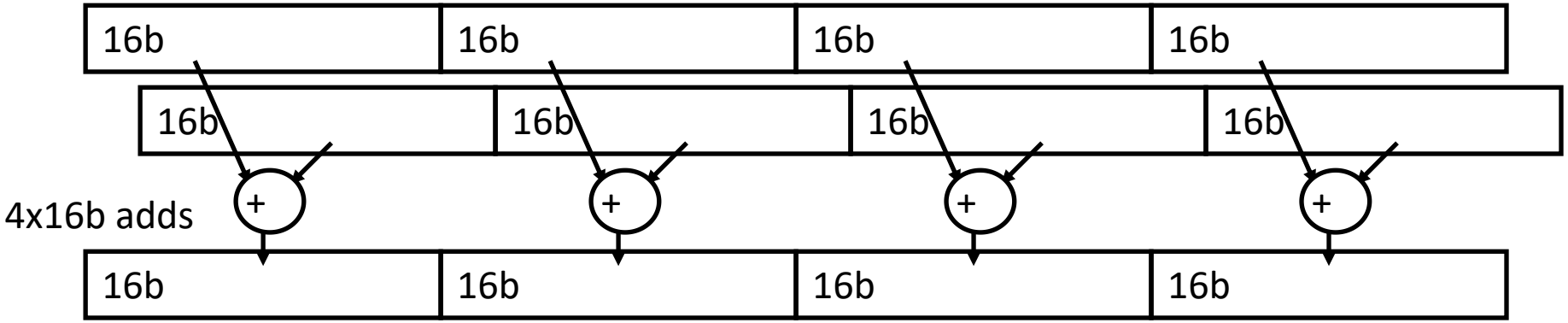
- **DLP技术基础：研究动机及种类**
  - 从效率瓶颈到DLP解决方案
  - DLP核心定义与并行范式对比
- **向量处理机：DLP的基础形态**
  - 向量处理器硬件结构
  - 向量指令集与代码示例
  - 向量处理器性能计算与优化
- **SIMD指令集扩展：DLP的轻量化落地**
- **GPU：DLP的高性能实战形态**
  - GPU硬件架构与SIMT模型
  - GPU编程模型与AI任务适配
- **AI专用处理器：DLP的AI场景定制形态**
  - AI专用处理器的发展历程：从学术探索到产业爆发
  - 典型AI专用处理器架构与DLP实现
- **四类DLP技术综合对比**

Multimedia Extensions (aka SIMD extensions)

- **在已有ISA中添加一些向量长度很短的向量操作指令**
- **将已有的 64-bit 寄存器拆分为 2x32b or 4x16b or 8x8b**
  - 1957年，Lincoln Labs TX-2 将36bit datapath 拆分为2x18b or 4x9b
  - 新的设计具有较宽的寄存器
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b for Intel AVX (Advanced Vector Extensions)
- **单条指令可实现寄存器中所有向量元素的操作**

Multimedia Extensions (aka SIMD extensions)





Intel Pentium MMX Operations

- **idea: 一条指令操作同时作用于不同的数据元**
  - 全阵列处理
  - 用于多媒体操作

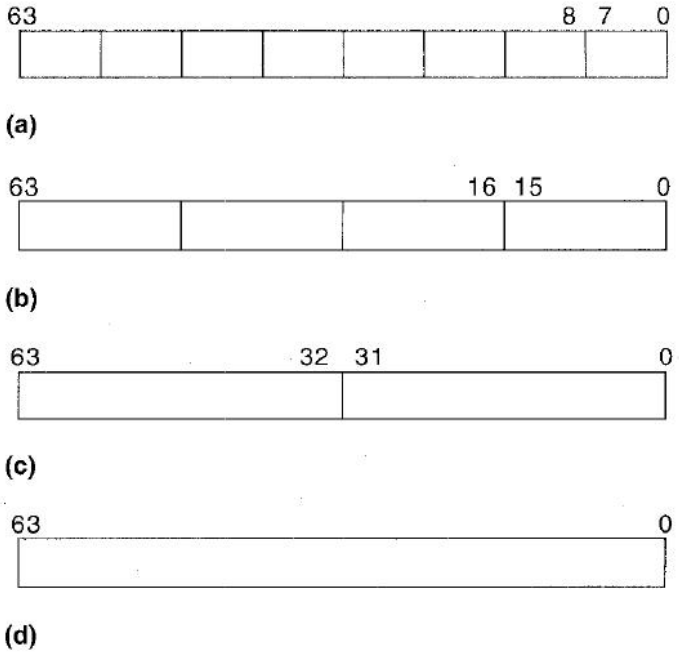


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

- No VLEN register
- Opcode determines data type:
  - 8 8-bit bytes
  - 4 16-bit words
  - 2 32-bit doublewords
  - 1 64-bit quadword
- Stride always equal to 1.

MMX Example: Image Overlaying (I)

Figure 8. Chroma keying: image overlay using a background color.

PCMPEQB MM1, MM3

MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF

Bitmask

Figure 9. Generating the selection bit mask.

MMX Example: Image Overlaying (II)

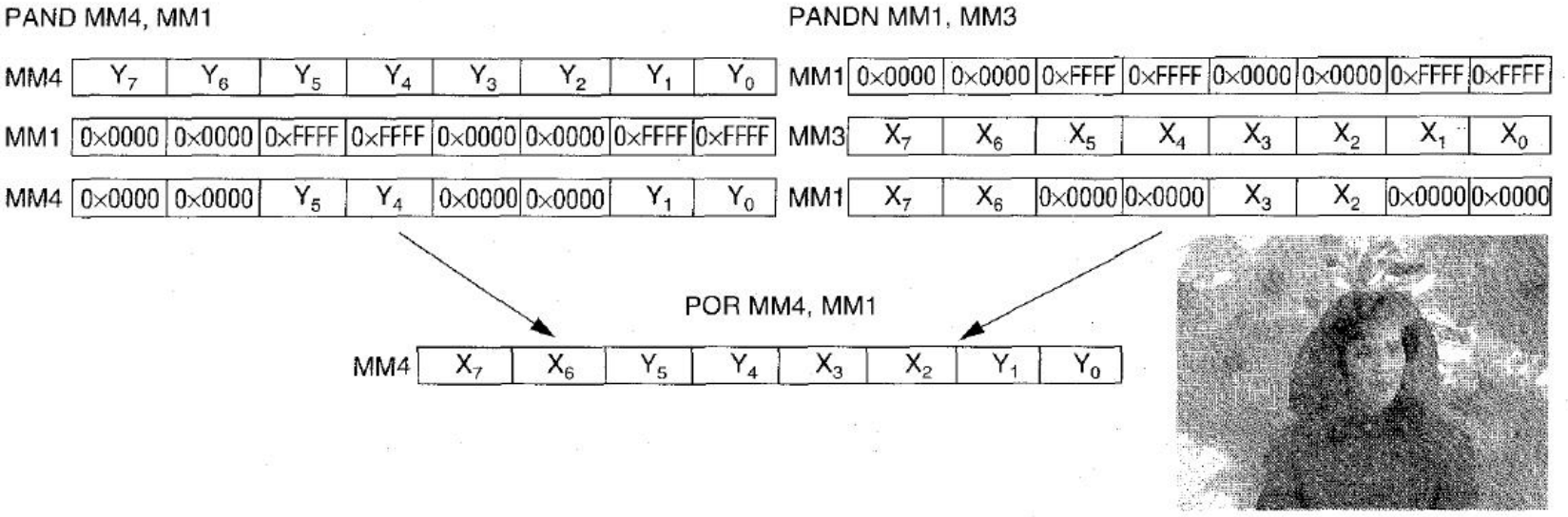


Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
Movq    mm3, mem1    /* Load eight pixels from
                        woman's image
Movq    mm4, mem2    /* Load eight pixels from the
                        blossom image
Pcmpeqb mm1, mm3
Pand    mm4, mm1
Pandn   mm1, mm3
Por     mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.

Multimedia Extensions versus Vectors

- **受限的指令集:**
  - 无向量长度控制
  - Load/store操作无 常数步长寻址和 scatter/gather操作
  - loads 操作必须64/128-bit 边界对齐
- **受限的向量寄存器长度:**
  - 需要超标量发射以保持multiply/add/load 部件忙
  - 通过循环展开隐藏延迟增加了寄存器读写压力
- **在微处理器设计中向全向量化发展**
  - 更好地支持非对齐存储器访问
  - 支持双精度浮点数操作 (64-bit floating-point)
  - Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)

Summary

- **向量机的存储器访问**
  - 存储器组织：独立存储体、多体交叉方式
  - Stride : 固定步长（1 or 常数）, 非固定步长（index）
- **基于向量机模型的优化**
  - 链接技术
  - 有条件执行
  - 稀疏矩阵的操作
- **多媒体扩展指令**
  - 扩展的指令类型较少
  - 向量寄存器长度较短

并行的类型

- **指令级并行(ILP)**
  - 以并行方式执行某个指令流中的独立无关的指令 (pipelining, superscalar, VLIW)
- **数据级并行(DLP)**
  - 以并行方式执行多个相同类型的操作 (vector/SIMD execution)
  - Array Processor 、 Vector Processor
- **线程级并行 (TLP)**
  - 以并行方式执行多个独立的指令流 (multithreading, multiple cores)

- **Which is easiest to program?**
- **Which is most flexible form of parallelism?**
  - i.e., can be used in more situations
- **Which is most efficient?**
  - i.e., greatest tasks/second/area, lowest energy/task

Part2：数据级并行

- **DLP技术基础：研究动机及种类**
  - 从效率瓶颈到DLP解决方案
  - DLP核心定义与并行范式对比
- **向量处理机：DLP的基础形态**
  - 向量处理器硬件结构
  - 向量指令集与代码示例
  - 向量处理器性能计算与优化
- **SIMD指令集扩展：DLP的轻量化落地**
- **GPU：DLP的高性能实战形态**
  - GPU硬件架构与SIMT模型
  - GPU编程模型与AI任务适配
- **AI专用处理器：DLP的AI场景定制形态**
  - AI专用处理器的发展历程：从学术探索到产业爆发
  - 典型AI专用处理器架构与DLP实现
- **四类DLP技术综合对比**

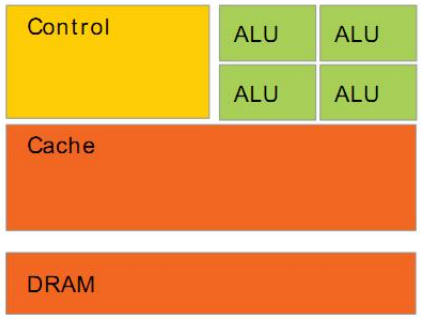
GPU I

Graphic Processing Unit (GPU) 起源

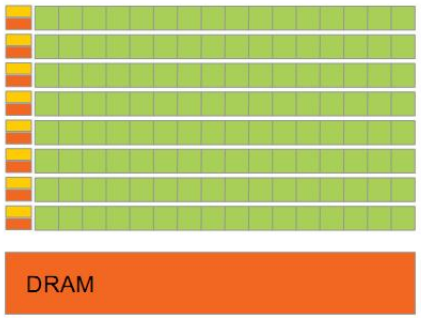
- **源于计算机图形学的发展：计算机图形学用于电影动画离线渲染，游戏中实时渲染**
- **显卡：硬件支持渲染**
  - 显卡始于81年 IBM 单色显示器 (MDA)，开始仅支持文本，后来支持 2D, 3D加速；
- **功能固定的GPU：用来渲染图形的专用设备（3D加速卡）**
  - 1999年 NVIDIA公司在发布其标志性产品**GeForce256**时，**首次提出了GPU的概念**
    - 首次实现基于硬件的变换与光照（transform and lighting）
  - 早期GPU：指带有高性能浮点运算部件、可高效生成3D图形的功能固定的专用设备
- **可编程的GPU：**
  - 2001年 NVIDIA 公司推出第一个可编程GPU（**GeForce 3**）
    - GPU 可以接受用户的顶点渲染程序（vertex shader）和片元级渲染程序（fragment shader）
  - 2003年 学术界研究将矩阵数据转换为纹理数据并使用shaders来完成线性代数运算
    - 激发了GPU制造商直接支持通用计算。
  - 2005 年 GPU 引入了统一渲染内核（unified shader processor）概念
    - 一组完全相同的、具有较强编程能力的内核，根据任务情况在顶点和片元处理任务之间动态分配
  - 2006年 NVDIA推出第一个支持通用计算的GPU（**GeForce 8** 系列）
  - G80 之后又推出 Fermi、Kepler 和 Maxwell 等多代 GPU，其通用计算能力越来越强大
- **GPU可编程性的提高，促进了其在通用计算领域的应用**
  - 2006 年 **CUDA 编程语言**和工具的推出，极大简化了 GPU 通用编程（只支持 NVIDIA的 GPU）
  - 2008年 **OpenCL** 推出，支持多家厂商的GPU（Intel、AMD、NVIDIA）

GPU基本硬件结构（1/2）

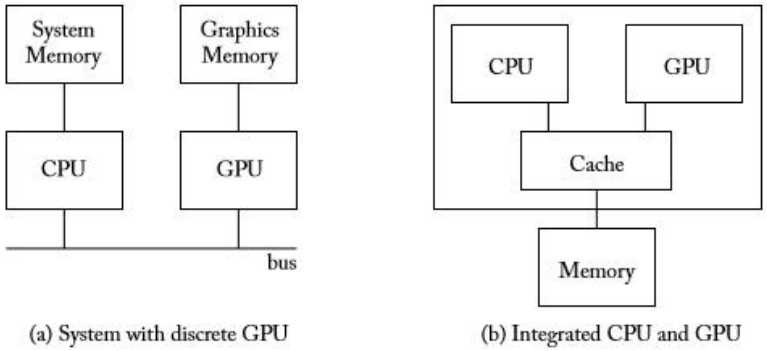
- CPU+GPU异构多核系统
  - 按照任务特点分配计算资源
  - GPU作为加速器（协处理器）
- CPU 核通常适合执行单一指令流
  - cache容量大，访问存储器延时低
- GPU 通常适合执行大量并行线程
  - 可扩放的并行执行
  - 高带宽的并行存取



强控制、弱计算



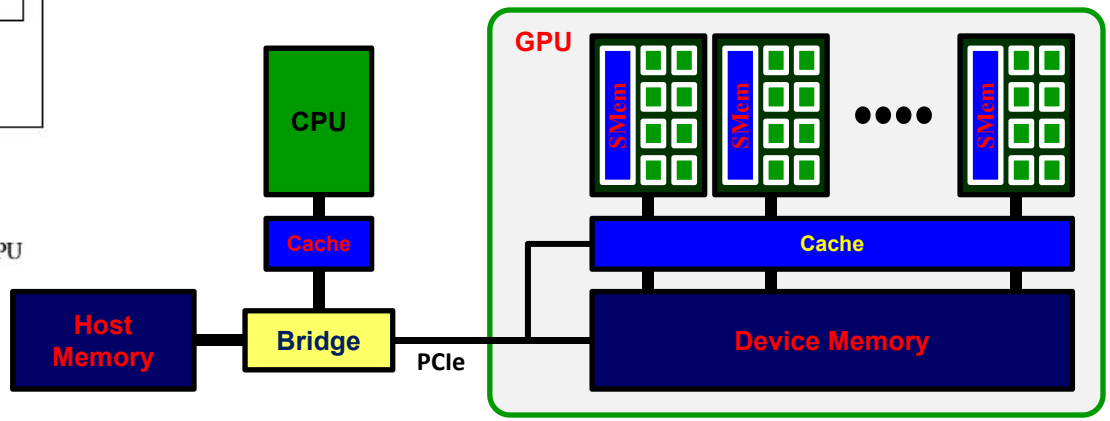
弱控制、强计算



(a) System with discrete GPU

(b) Integrated CPU and GPU

Figure 1.1: GPU computing systems include CPUs.



GPU硬件结构 (2/2)

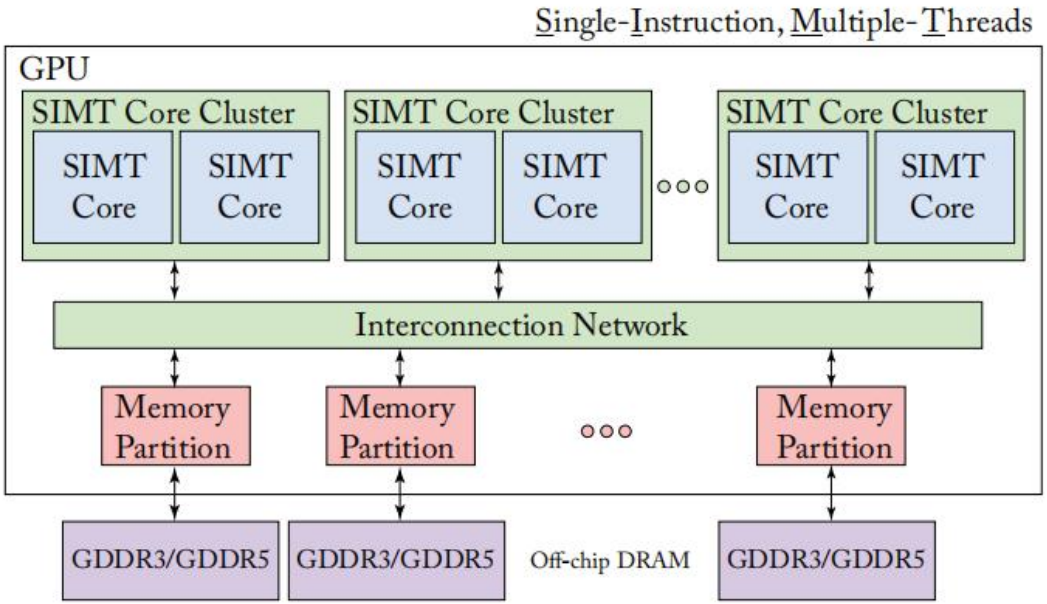


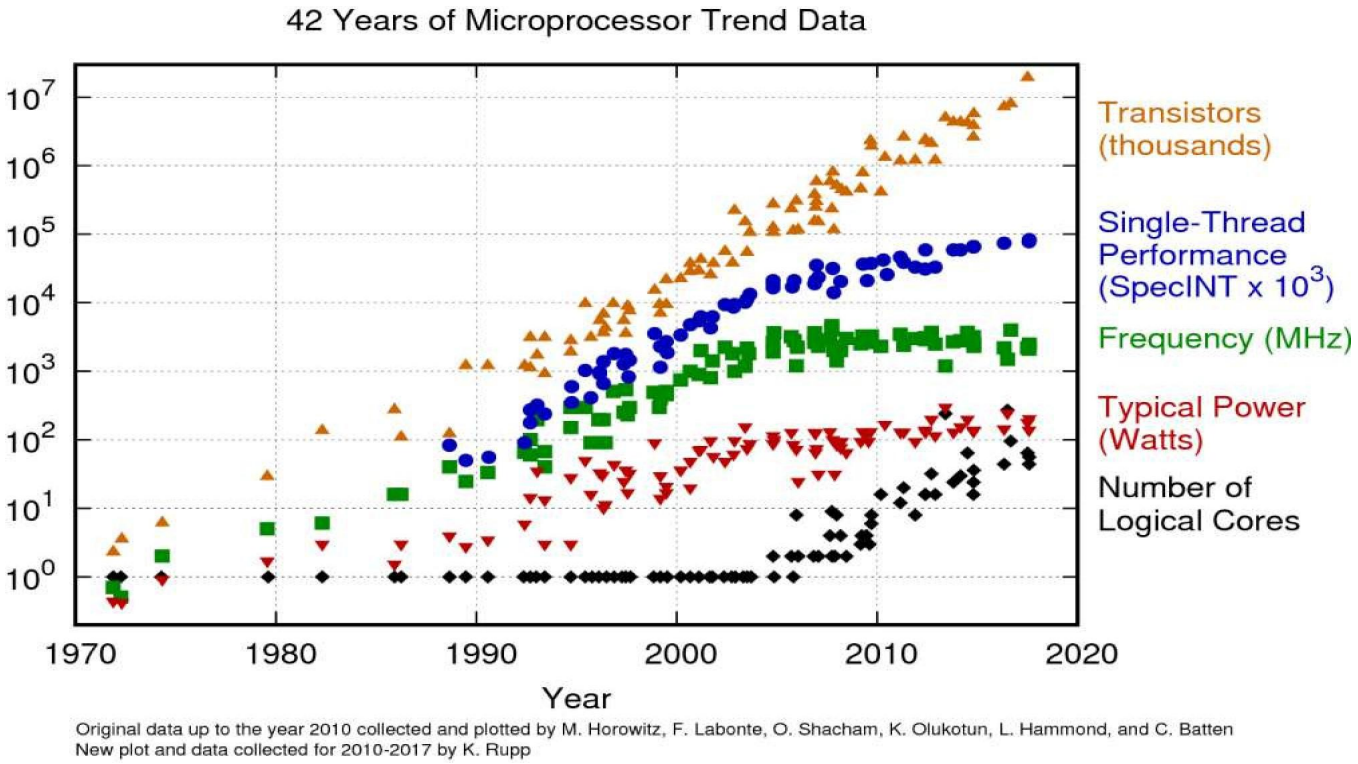
Figure 1.2: A generic modern GPU architecture.

- **现代GPU由许多（逻辑）核心组成，NVIDIA称其硬件计算单元为Streaming multiprocessor**
- **每个核心执行一个单指令多线程(SIMT)程序，该程序对应于发送到GPU上运行的kernel**
- **每个核心可以运行上千个线程。线程可通过Scratchpad通信，使用barrier操作进行同步**

- **每个核心通常还包含一级指令和数据缓存，以减少与较低级别的内存系统通信量。**
- **当在第一级缓存中未命中时，核心上运行的大量线程通过线程切换来隐藏访问内存的延迟**

- **GPU的并行性要求更高的存储器带宽，通常要求存储系统是多通道存储系统。**

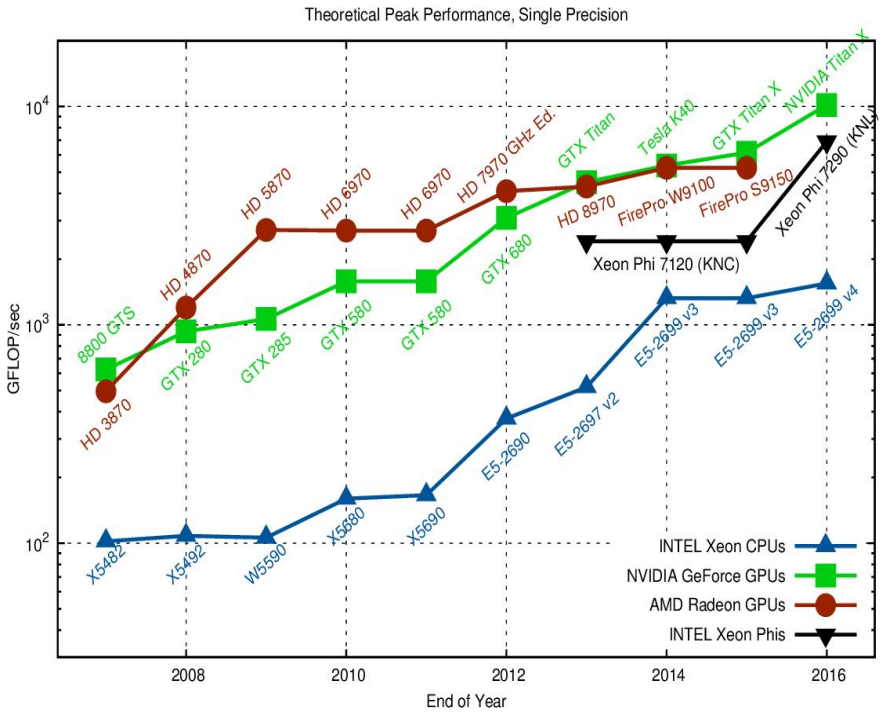
GPU得到普遍关注，why?

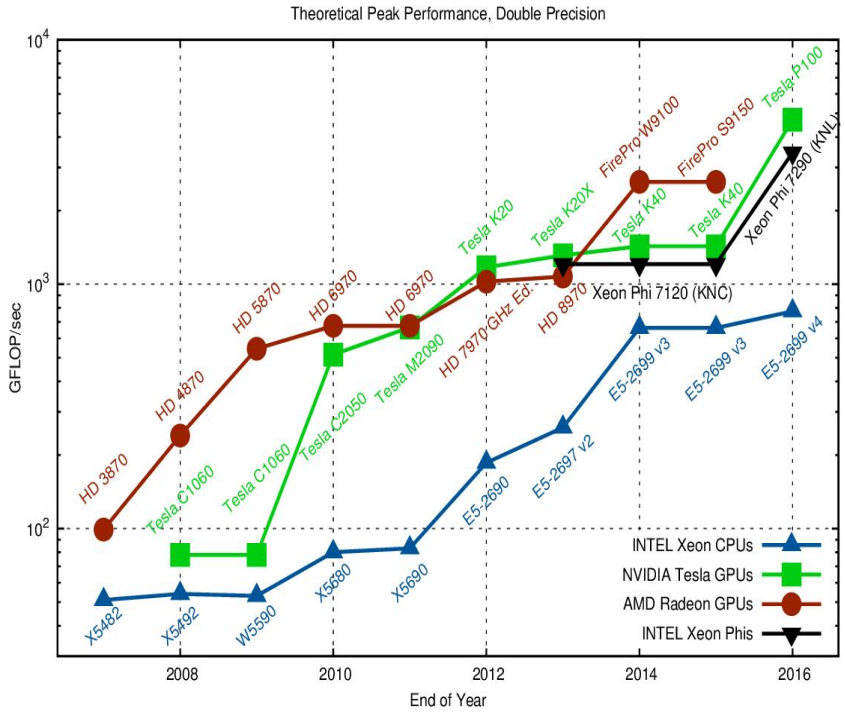


- **Moore's law：早期:IC中晶体管的数量每两年2X；目前：趋势放缓**
- **21世纪以来：时钟频率、单核的性能增加有限；性能提升主要依赖于单片上的“core”的数量；必须设计并行执行的代码**

Source: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

GPU得到普遍关注，why?

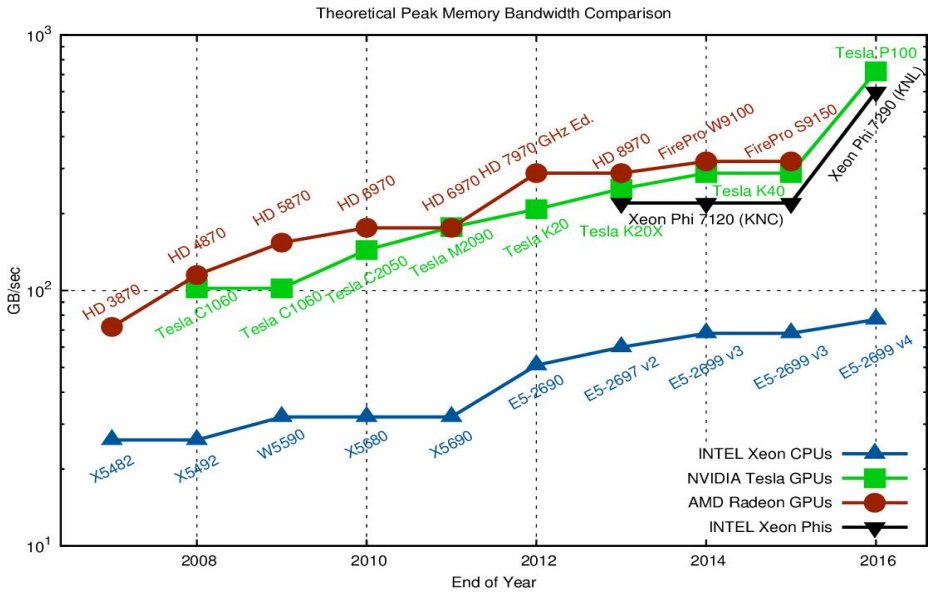


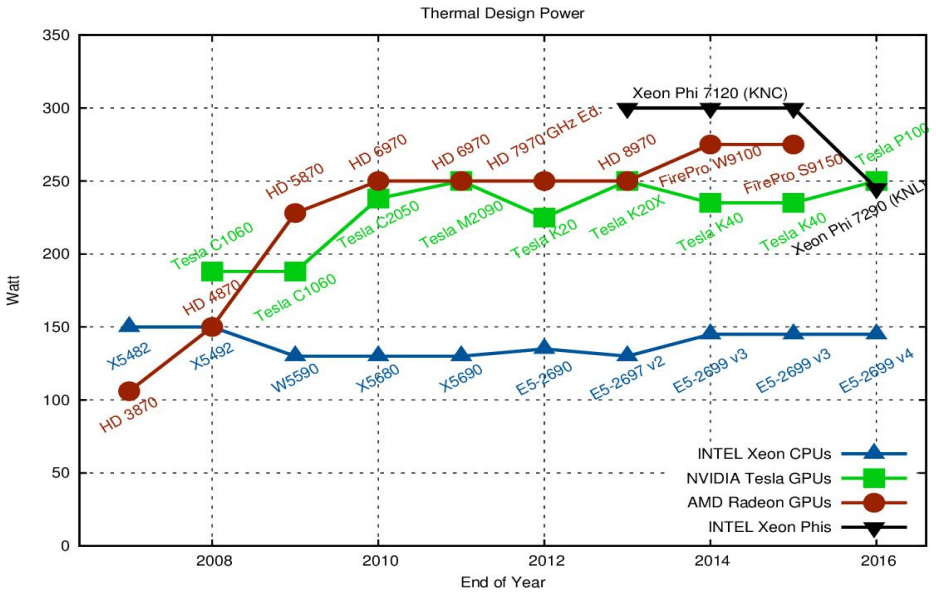


与cpu相比，gpu提供了更高的32位浮点数性能

数据中心GPU还提供比cpu更高的64位浮点性能

Figures source: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/



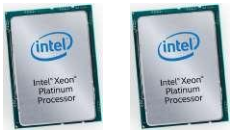


GPU的访存带宽明显高于cpu

GPU与CPU的散热设计功耗（TDP）比较

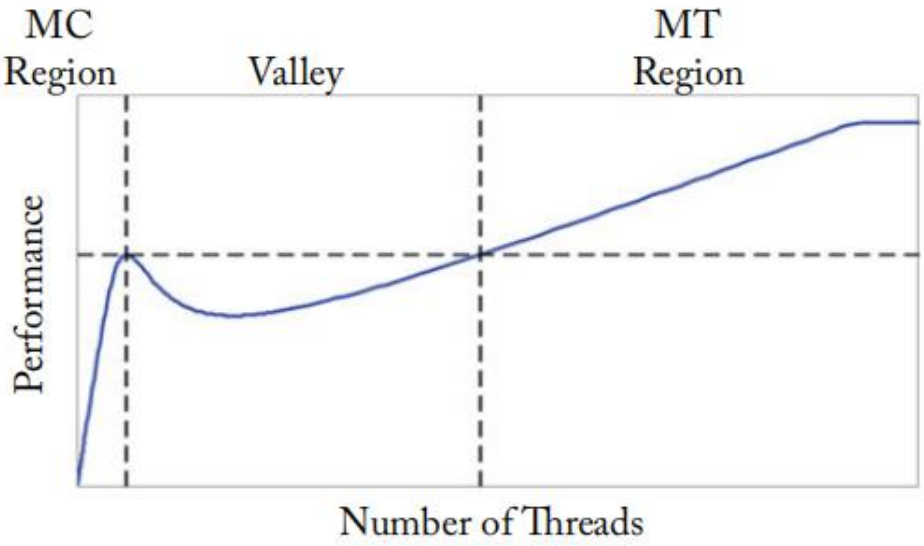
Figures source: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

X86 Intel8180 vs Nvidia V100 GPU



Aggregate performance numbers (FLOPs, BW)	**Dual socket Intel 8180 28-core (56 cores per node)**	**Nvidia Tesla V100, dual cards in an x86 server**
Peak DP FLOPs	4 TFLOPs	14 TFLOPs (3.5x)
Peak SP FLOPs	8 TFLOPs	28 TFLOPs (3.5x)
Peak HP FLOPs	N/A	224 TFLOPs
Peak RAM BW	~ 200 GB/sec	~ 1,800 GB/sec (9x)
Peak PCIe BW	N/A	32 GB/sec
Power / Heat	~ 400 W	2 x 250 W (+ ~ 400 W for server) (~ 2.25x)
Code portable?	Yes	Yes (OpenACC, OpenCL)

Multicore vs. Multithreaded



为了保持模型简单，假设一个简单的缓存模型：

线程不共享数据，并且片外内存带宽足够大

Figure 1.3: An analytical model-based analysis of the performance tradeoff between multicore (MC) CPU architectures and multithreaded (MT) architectures such as GPUs shows a "performance valley" may occur if the number of threads is insufficient to cover off-chip memory access latency (based on Figure 1 from Guz et al. [2009]).

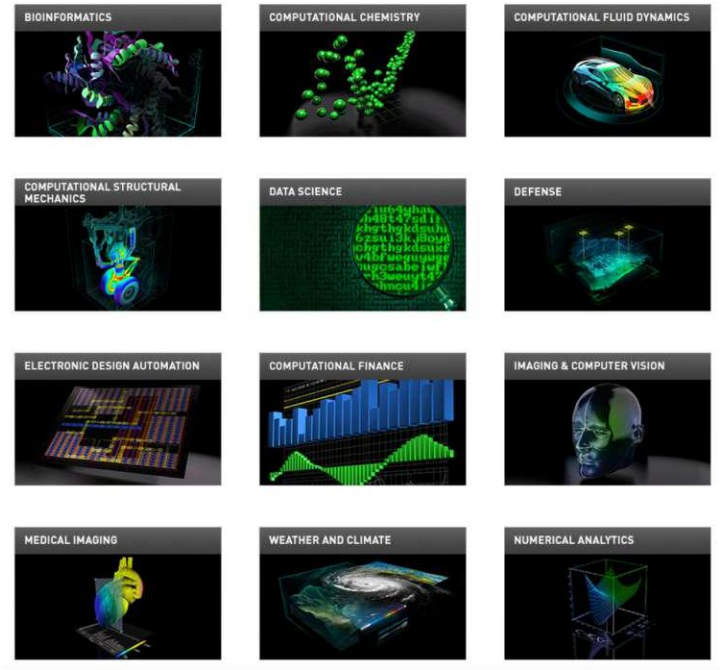
- **多核处理器结构（CPU）随着线程数的增加会落入性能低谷**
  - Cache容量不足时，性能就会下降。
- **当线程数较少时多线程结构（GPU）会落入性能低估**
  - 无法抵消对片外存储访问的延时

Zvika Guz et al. Many-core vs. many-thread machines: Stay away from the valley. IEEE Computer Architecture Letters, 8(1):25–28, 2009. DOI: 10.1109/l-ca.2009.4. 5

GPU 应用领域

一些主要应用领域

- Exhautive list on https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/
- 化学
- 生命科学
- 生物信息学
- 天体物理学
- 金融
- 医学图像
- 自然语言处理
- 社会科学
- 天气与气候
- 计算流体动力学
- **机器学习**
- etc...



人工智能应用

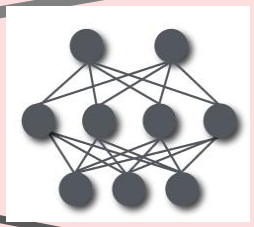
超级计算机

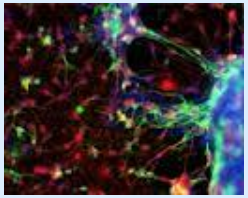
数据中心

智能手机

嵌入式设备

机器学习
神经网络





药物研制

自动翻译

图像分析

机器人

Source: Yunji Chen (ICT)

提升算力的有效途径=>加速器

基于GPU的加速器
- NVIDIA P100--V100--A100

专用智能加速器
- DianNao 寒武纪
- Google TPU
- 华为昇腾910


基于FPGA的加速器
- Intel+Altera DLIA
- Microsoft Catapult
- DeepHi

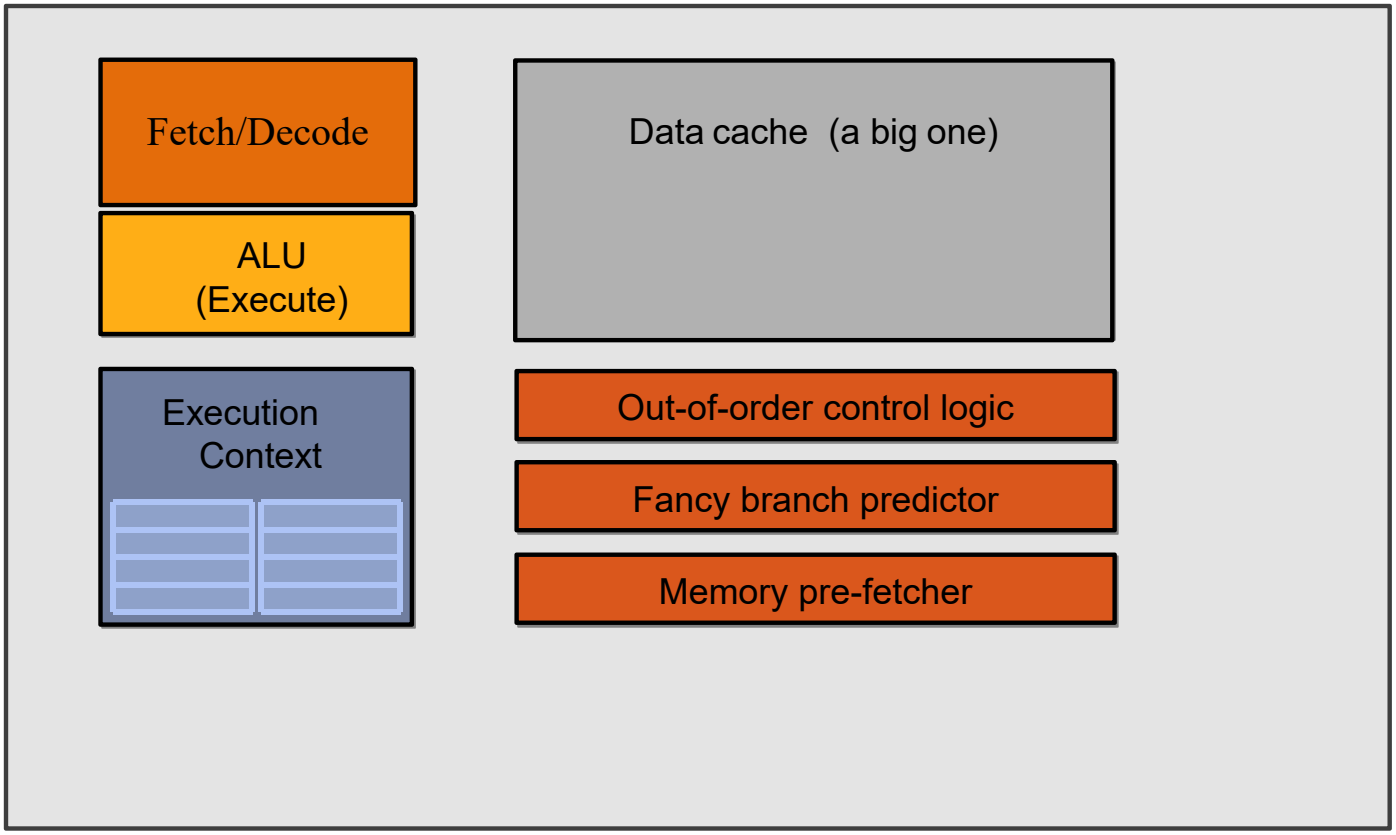
GPU I

GPU: throughput processing

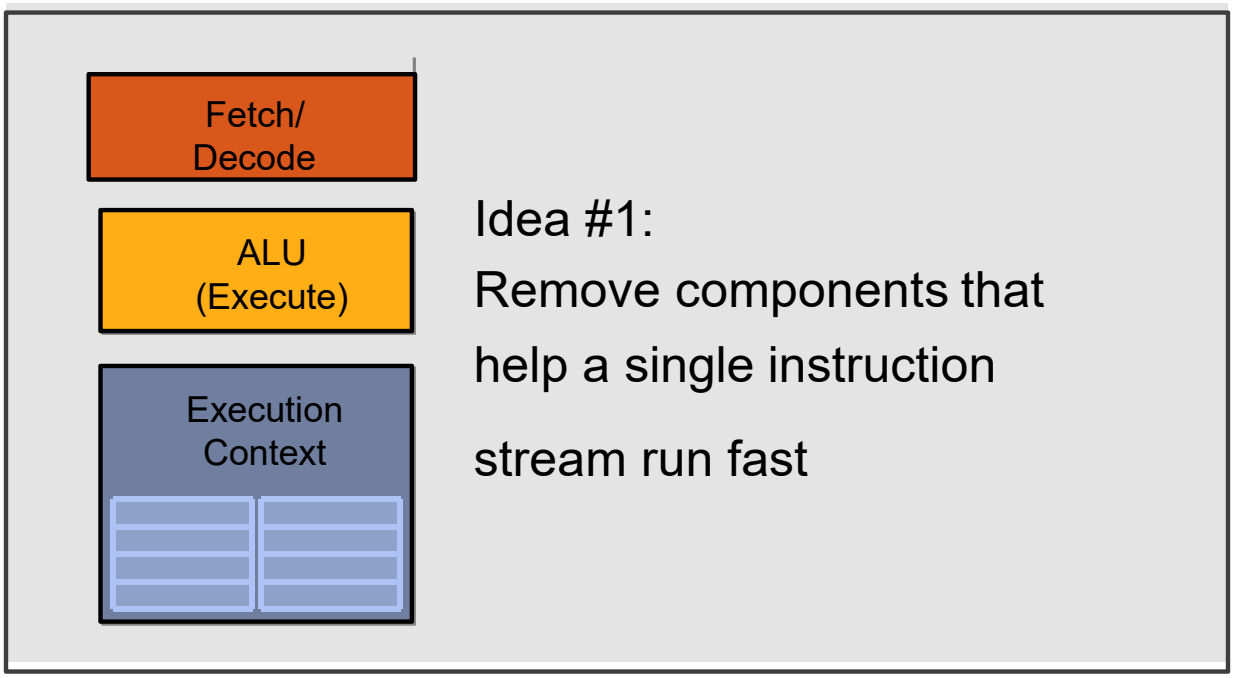
- **现代GPU能够高效运行背后的三个关键概念**
  - 使用许多“精简的core”并行运行
  - “装满ALU的core” 形成SIMD处理模式
  - 通过交叉执行不同组线程来避免长延迟(数据访问)

- **了解这些概念的作用：**
  - 理解GPU结构设计的基本思想
  - 优化我们的应用程序
  - 建立直观印象:哪些工作负载可能会适合在这些体系结构（的机器上运行）？

“CPU-style” cores

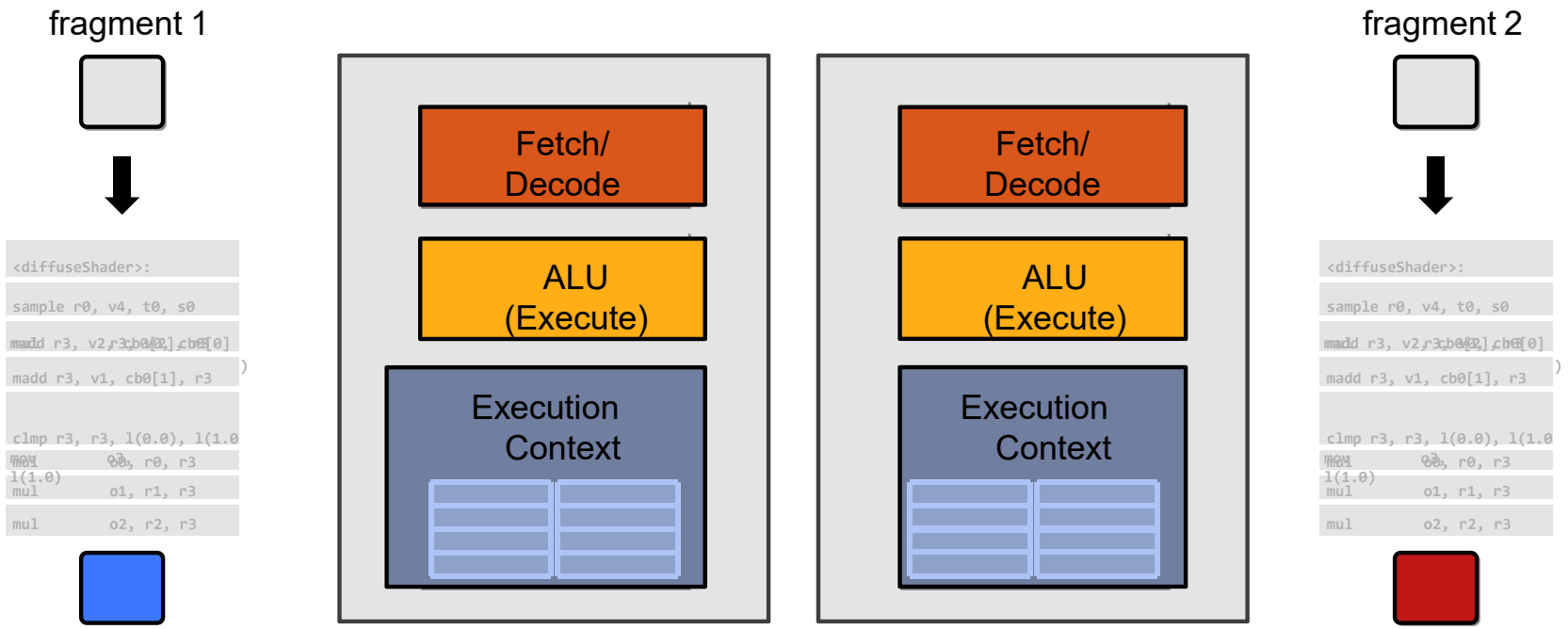


Slimming down

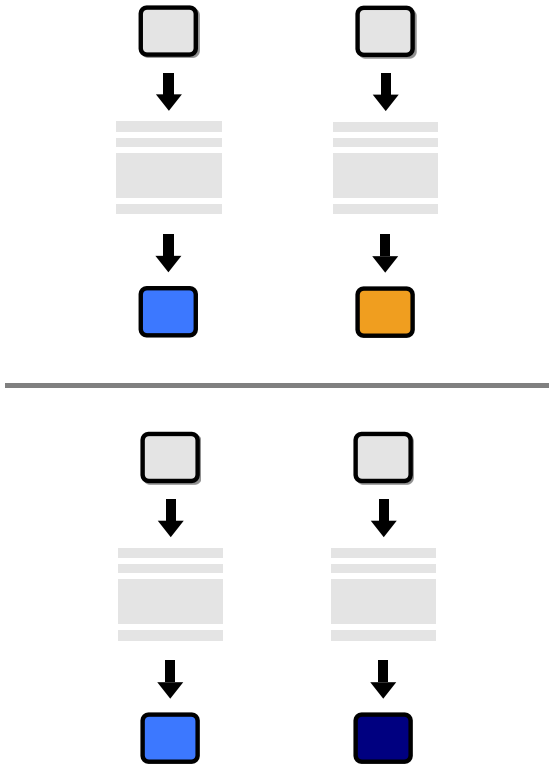


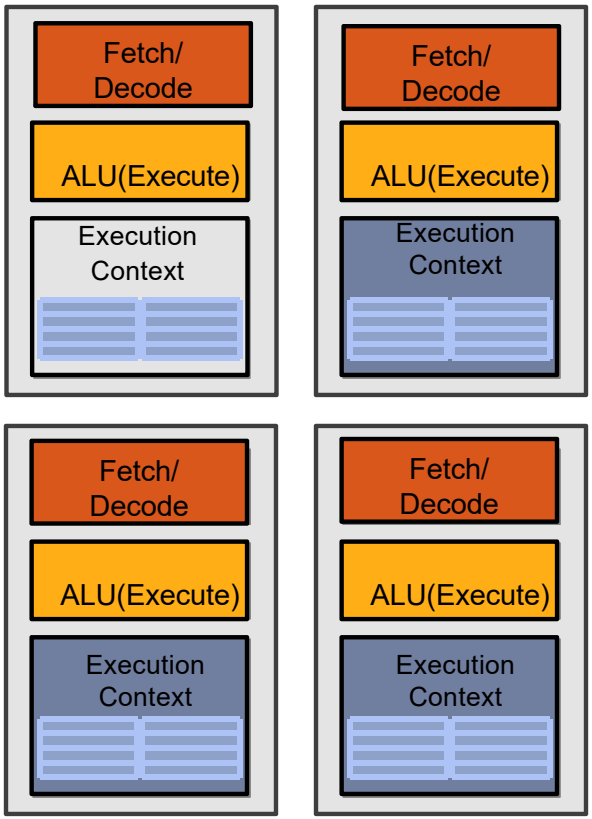
去除CPU中加速单指令流运行的组件，构建简单的core
由简单的core构成多核系统，并行执行多条指令流

Two cores (two fragments in parallel)

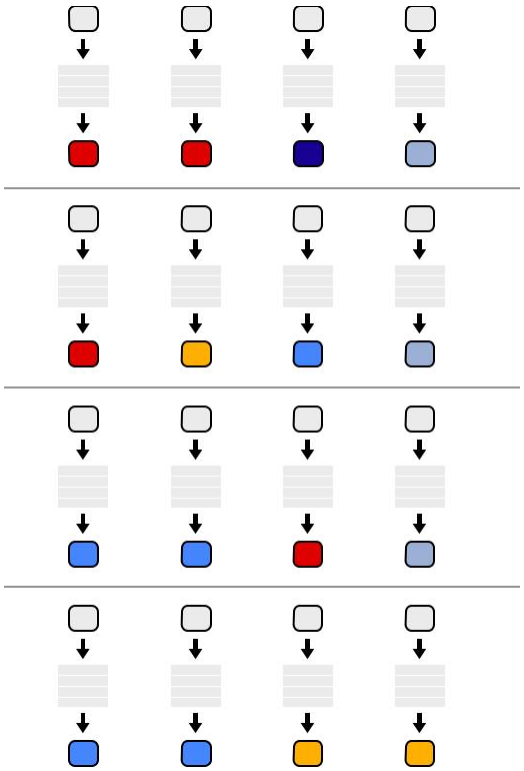


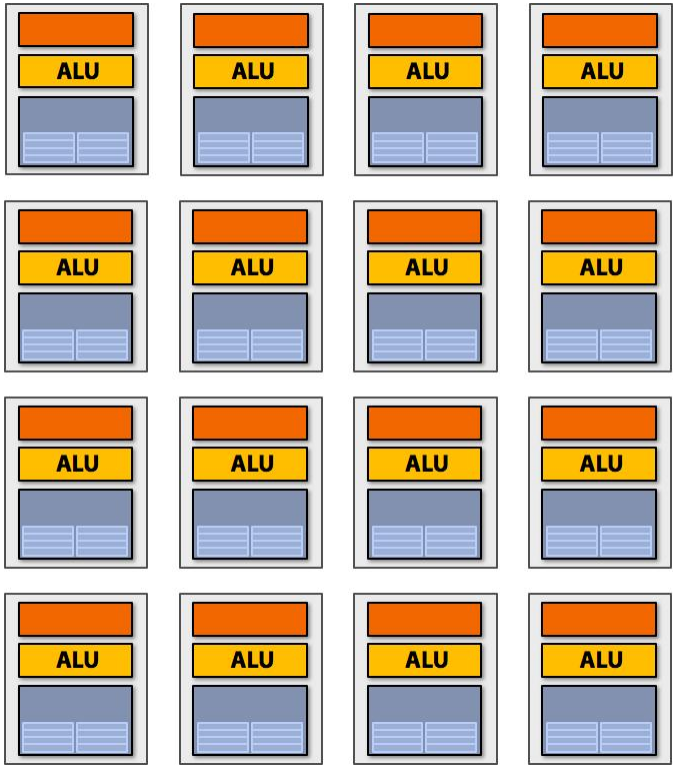
Four cores (four fragments in parallel)





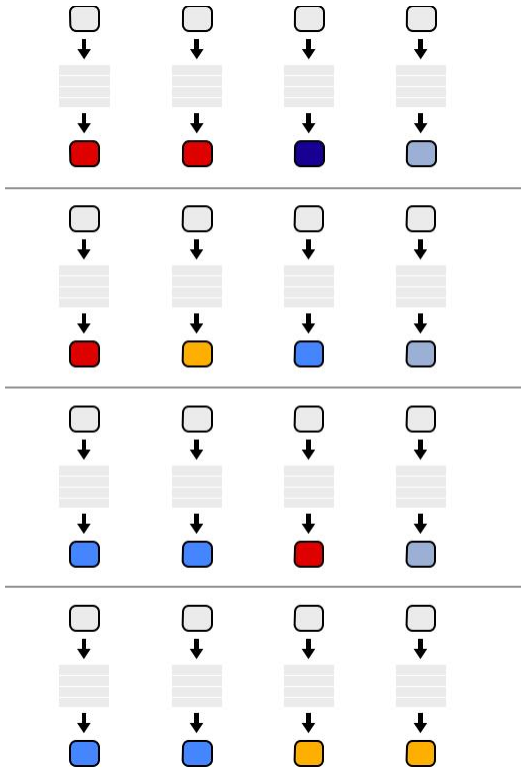
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

Instruction stream sharing

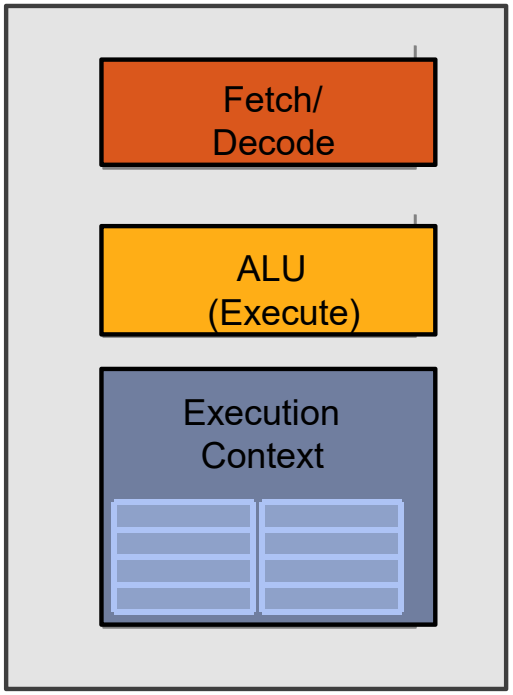


But ... many fragments should be able to share an instruction stream!

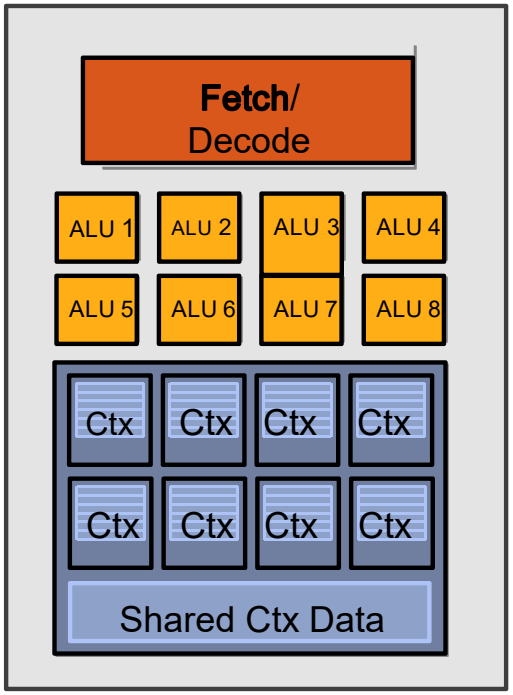
```
<diffuseShader>:
sample r0, v4, t0, s0
mul     r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul     o0, r0, r3
mul     o1, r1, r3
mul     o2, r2, r3
mov     o3, l(1.0)
```

多组不同的数据应该还可以共享一条指令流，即单个指令流处理多组数据 SIMD?

Recall: simple processing core



Add ALUs



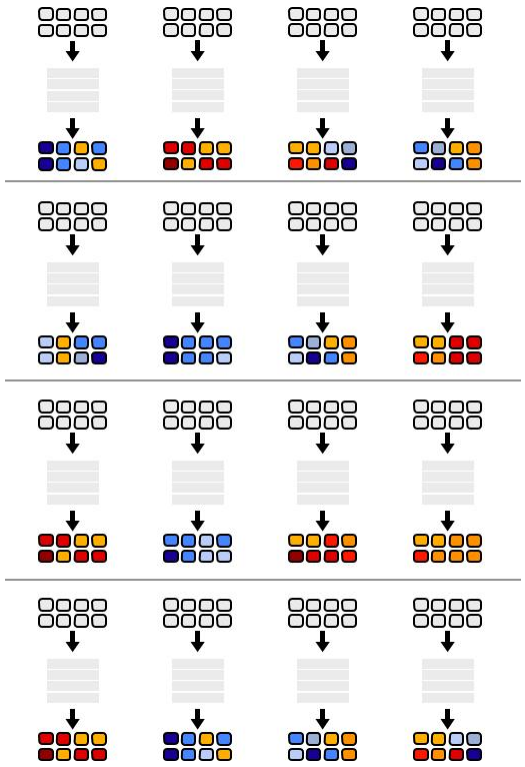
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

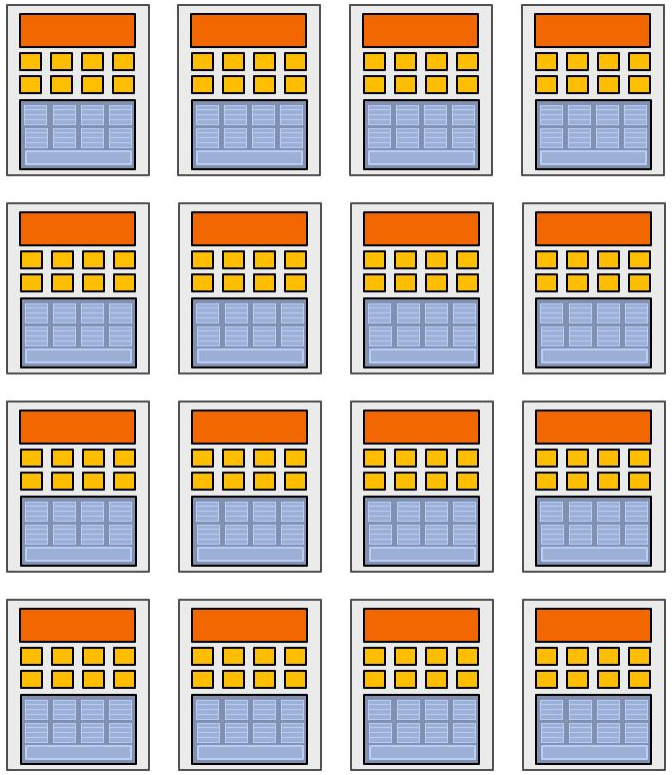
增加ALU的数量，形成SIMD处理模式。即：

多组不同的数据共享一条指令流，即单个指令流处理多组数据

128 fragments in parallel



16 cores = 128 ALUs

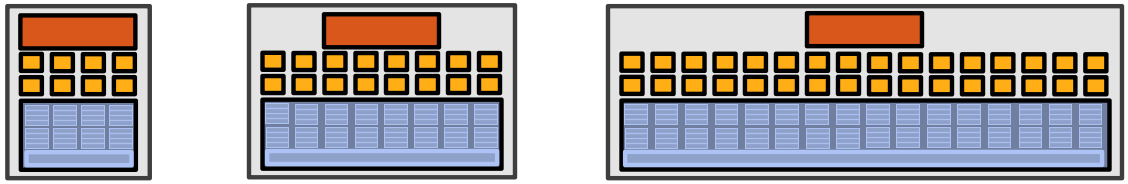


16 simultaneous instruction streams

SIMD 处理的方式

SIMD处理并不意味着一定是执行SIMD指令

- Option 1: 显式的向量指令
  - x86 SSE, AVX, Intel Larrabee
- **Option 2: 标量指令，隐式HW向量化**
  - 由HW 确定ALU共享的指令流 (多少ALU共享一条指令流对软件隐藏)
  - NVIDIA GeForce (“SIMT” warps), ATI Radeon architectures (“wavefronts”)



In practice: 16 to 64 fragments share an instruction stream.

如何隐藏程序执行中的停顿?

- **当CPU核由于正在运行的指令依赖于前面未完成的指令时会导致指令运行停顿(Stalls)**
  - Texture access latency = 100's to 1000's of cycles
- **GPU减少了那些避免或减少stall的cache以及其他支持挖掘程序内在并行性的组件**

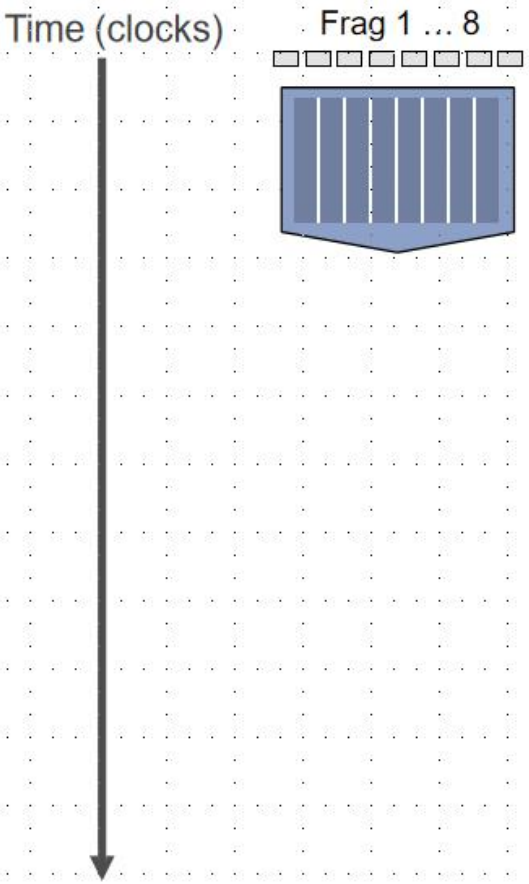
- **如何避免或减少Stall?**

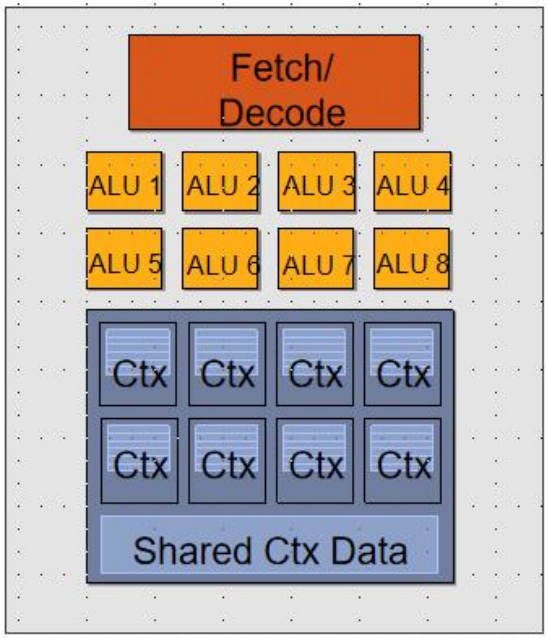
Idea #3:

Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations

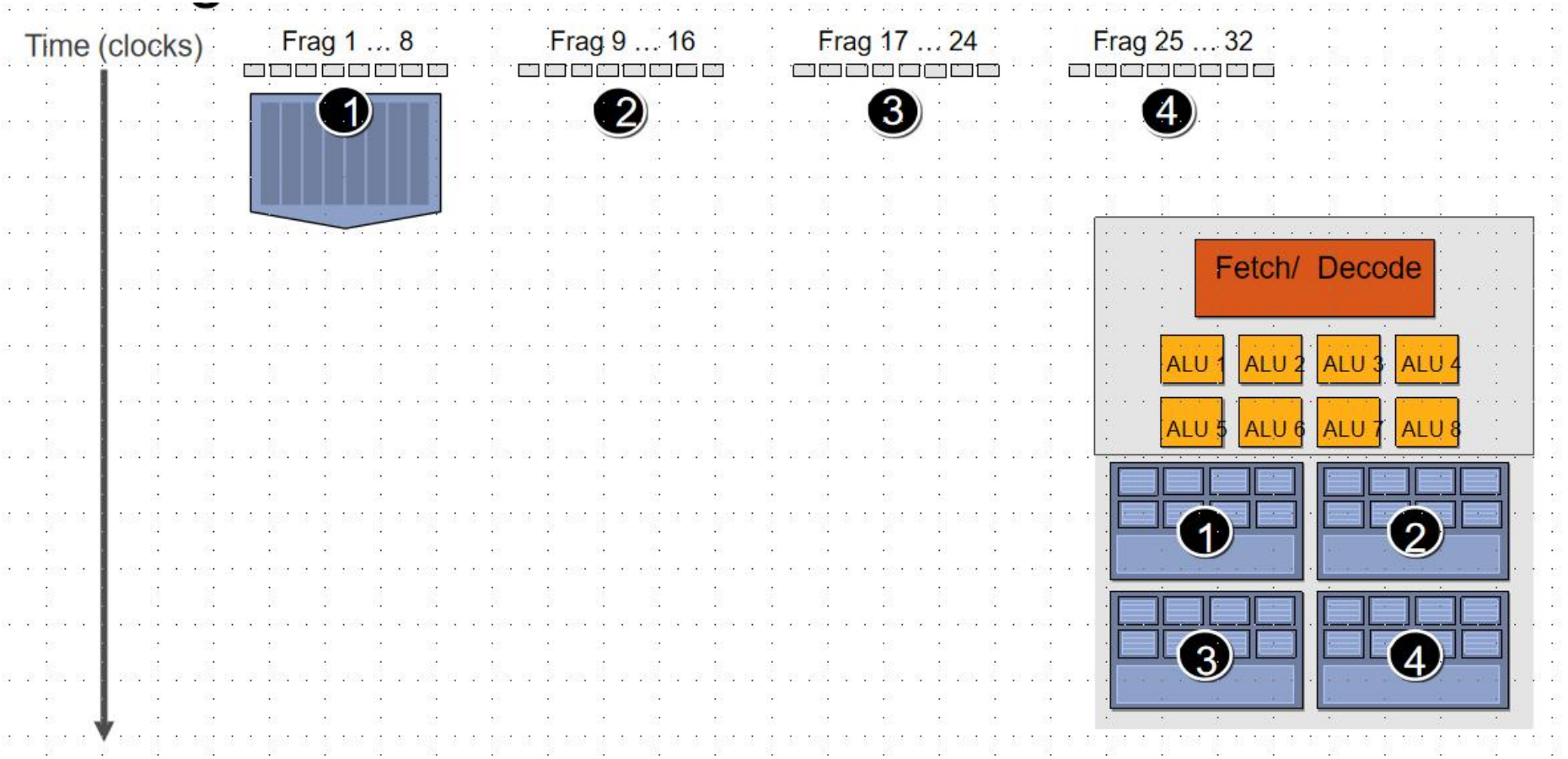
(在单个核上通过指令流切换交错处理不同数据，以避免长延迟操作造成的停顿.)

Hiding shader stalls

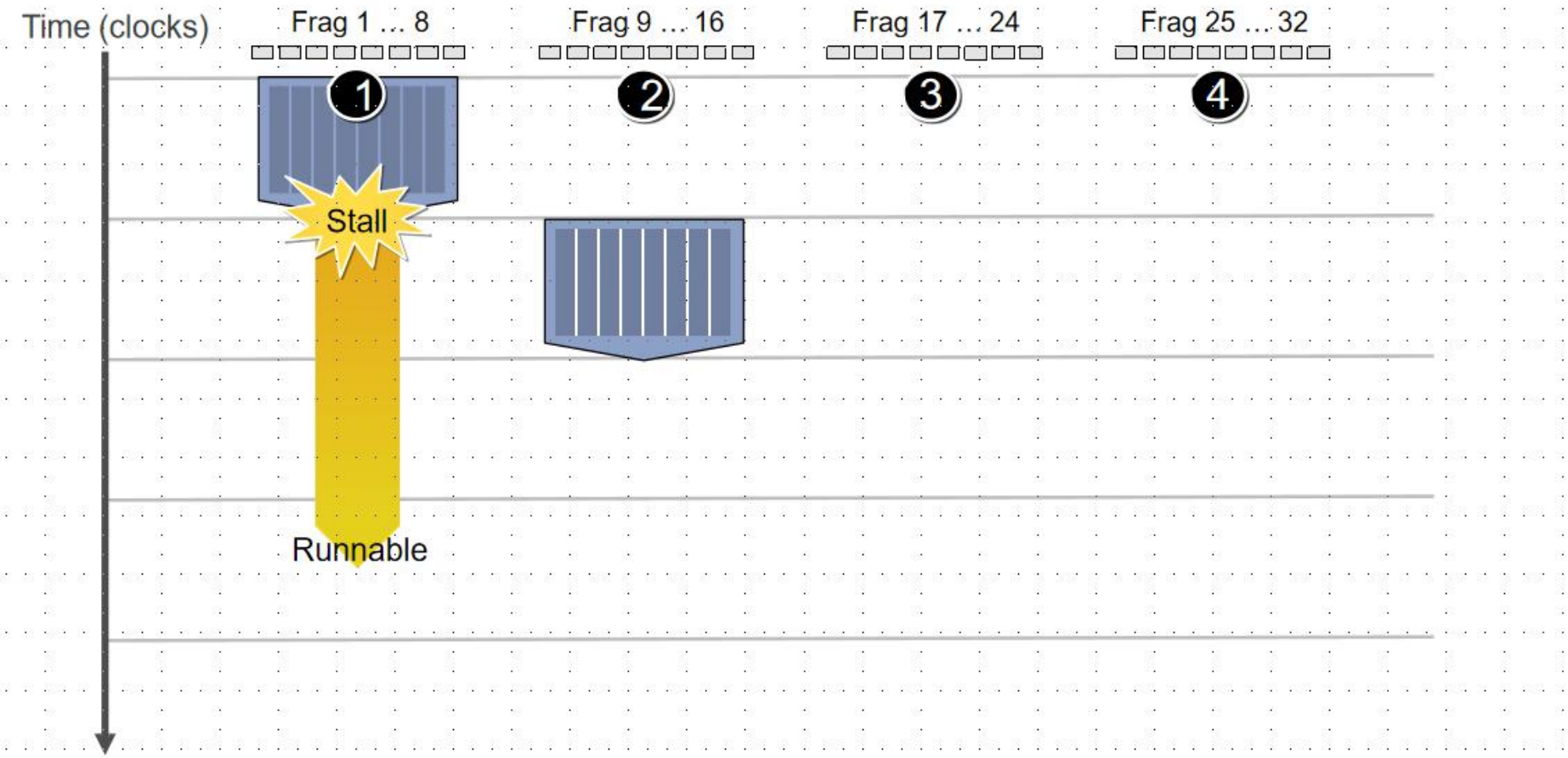




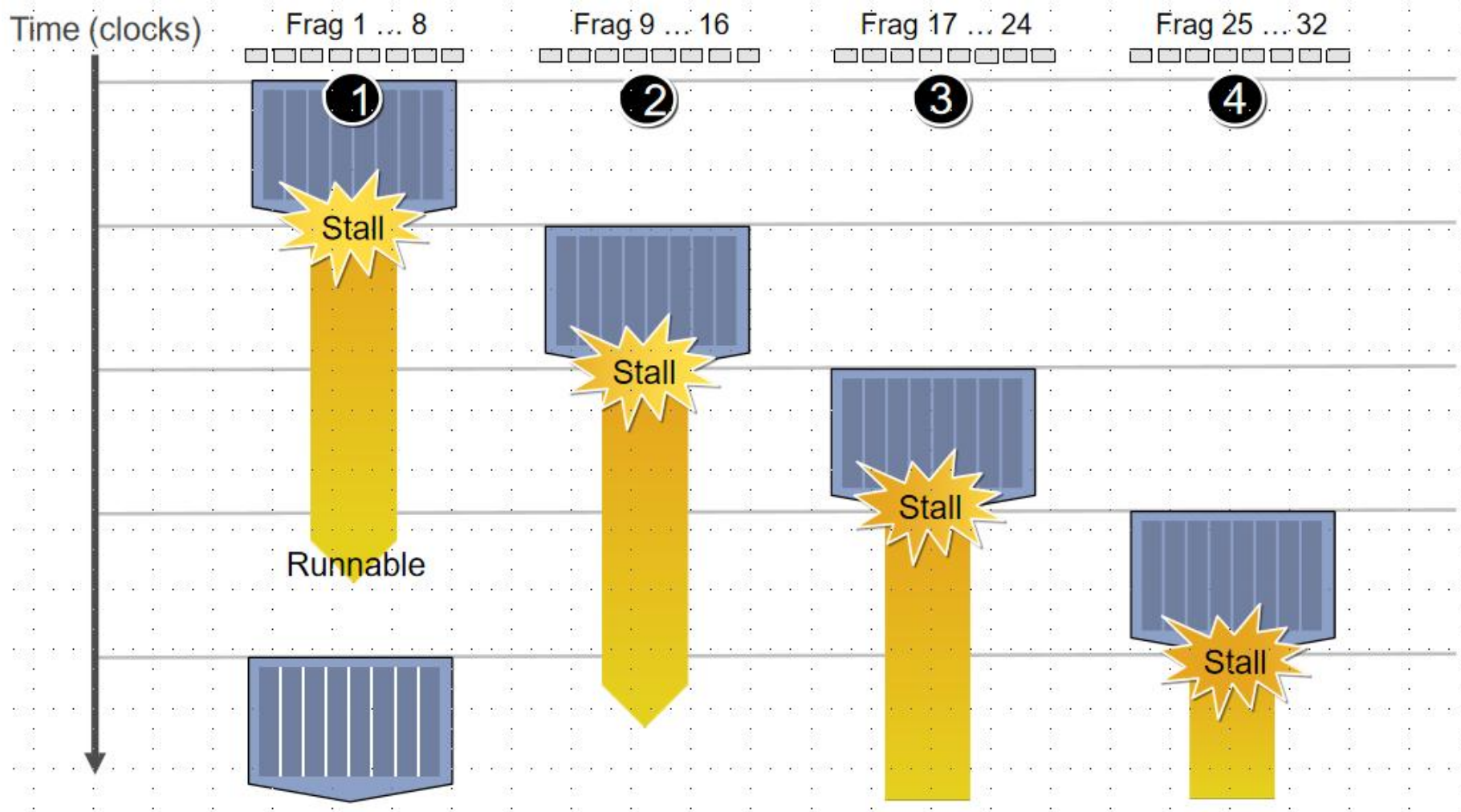
Hiding shader stalls



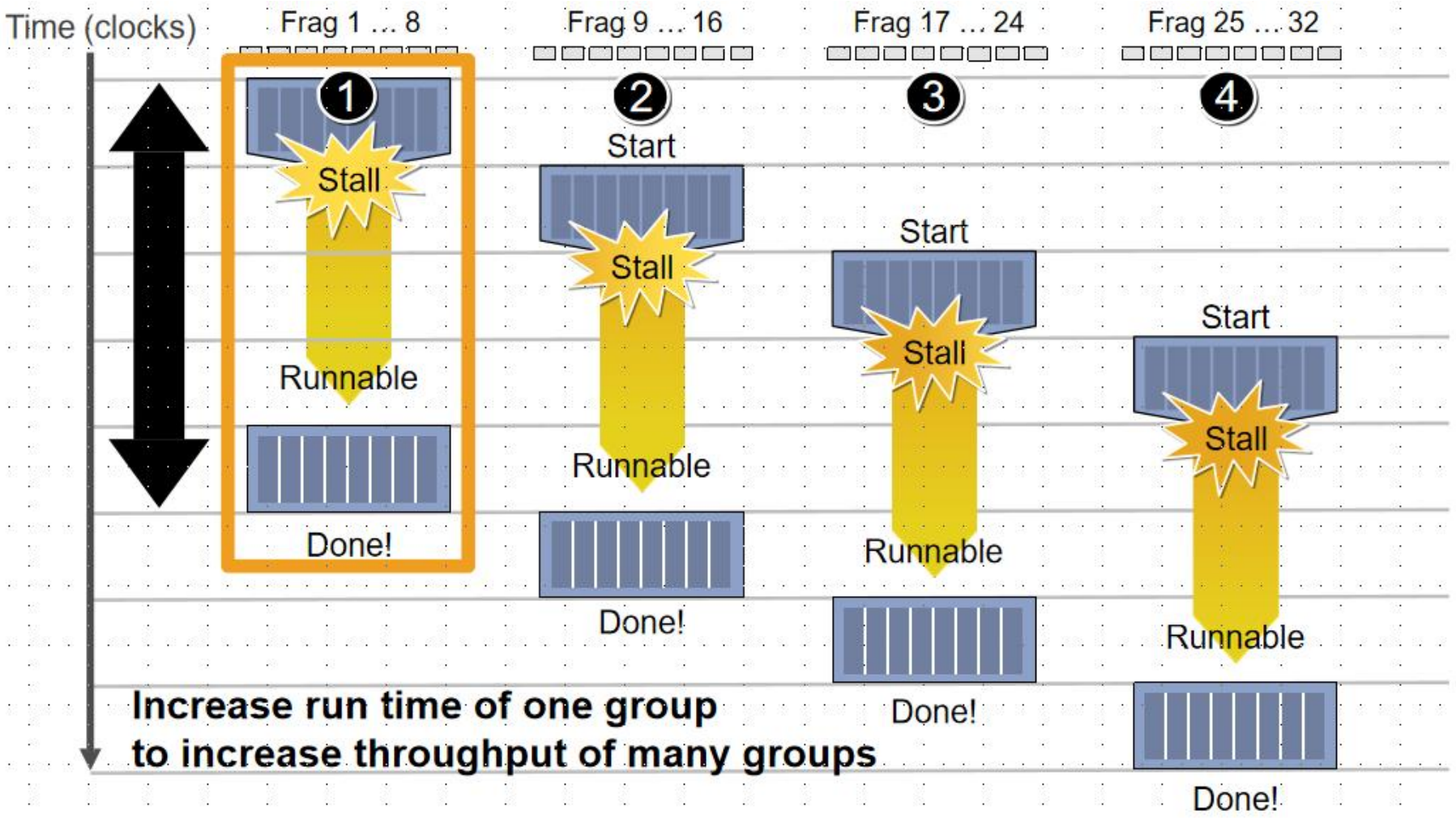
Hiding shader stalls



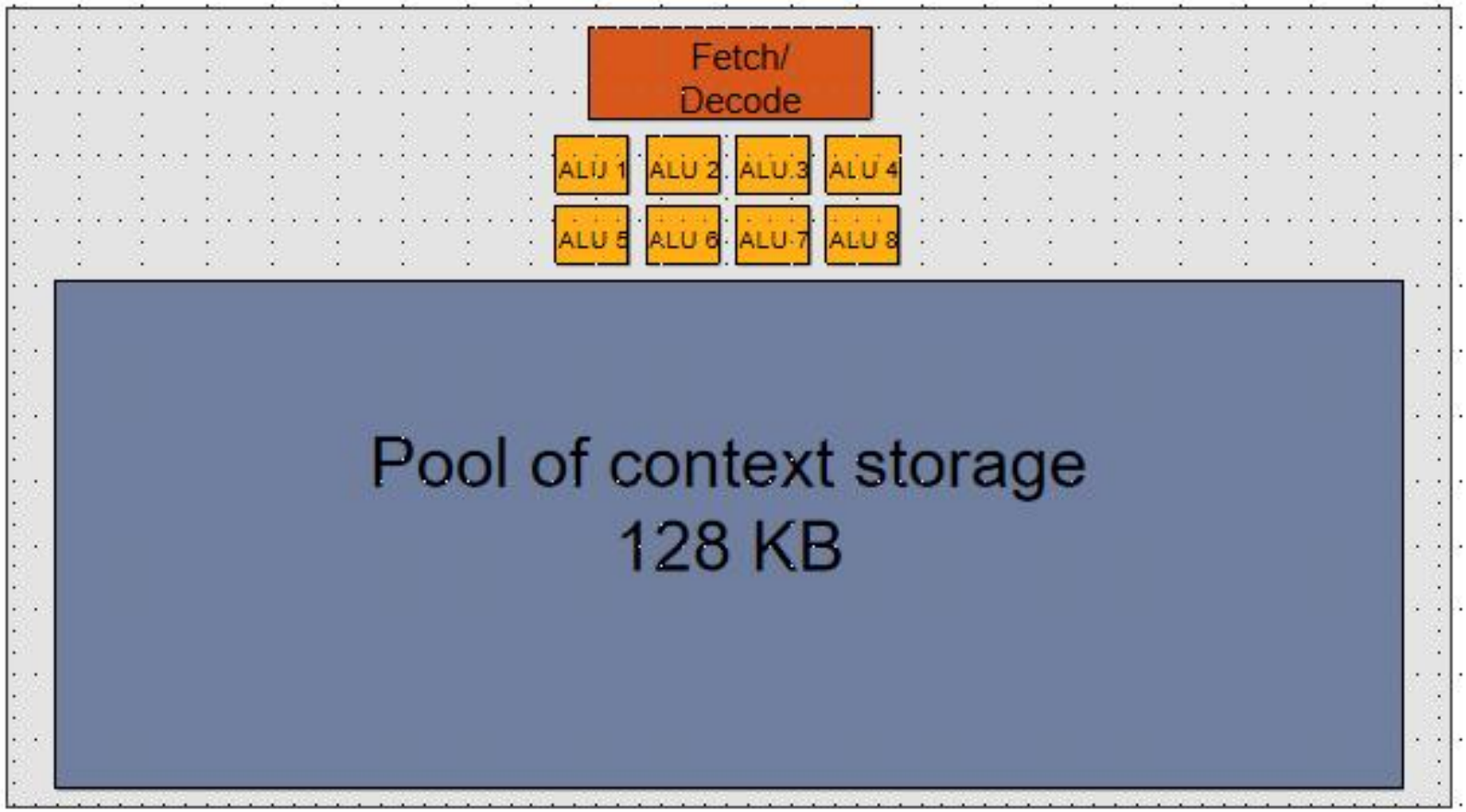
Hiding shader stalls



Throughput!

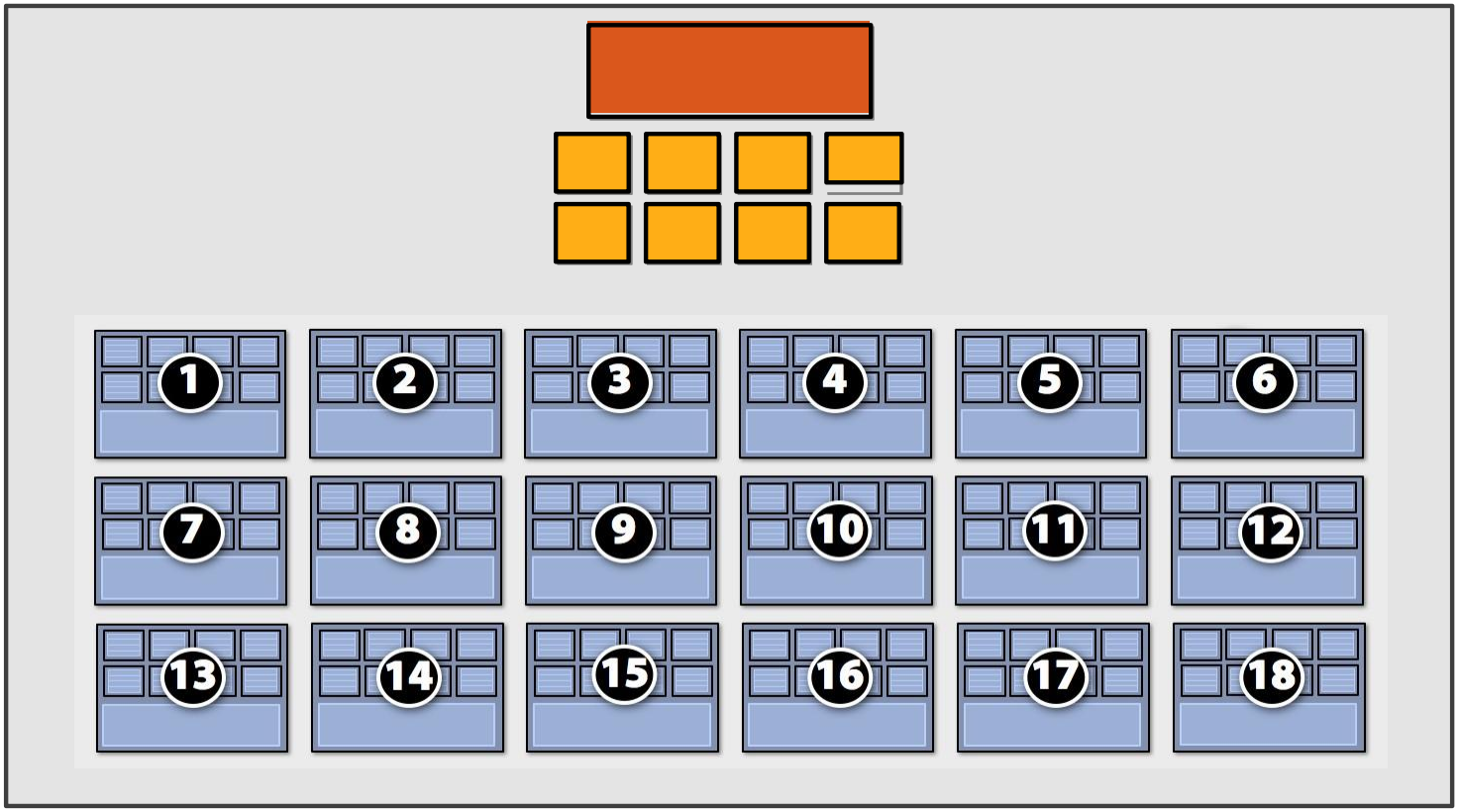

Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Stall
Start
Stall
Start
Stall
Runnable
Runnable
Runnable
Runnable
Done!
Done!
Done!
Done!
Increase run time of one group
to increase throughput of many groups

Storing contexts



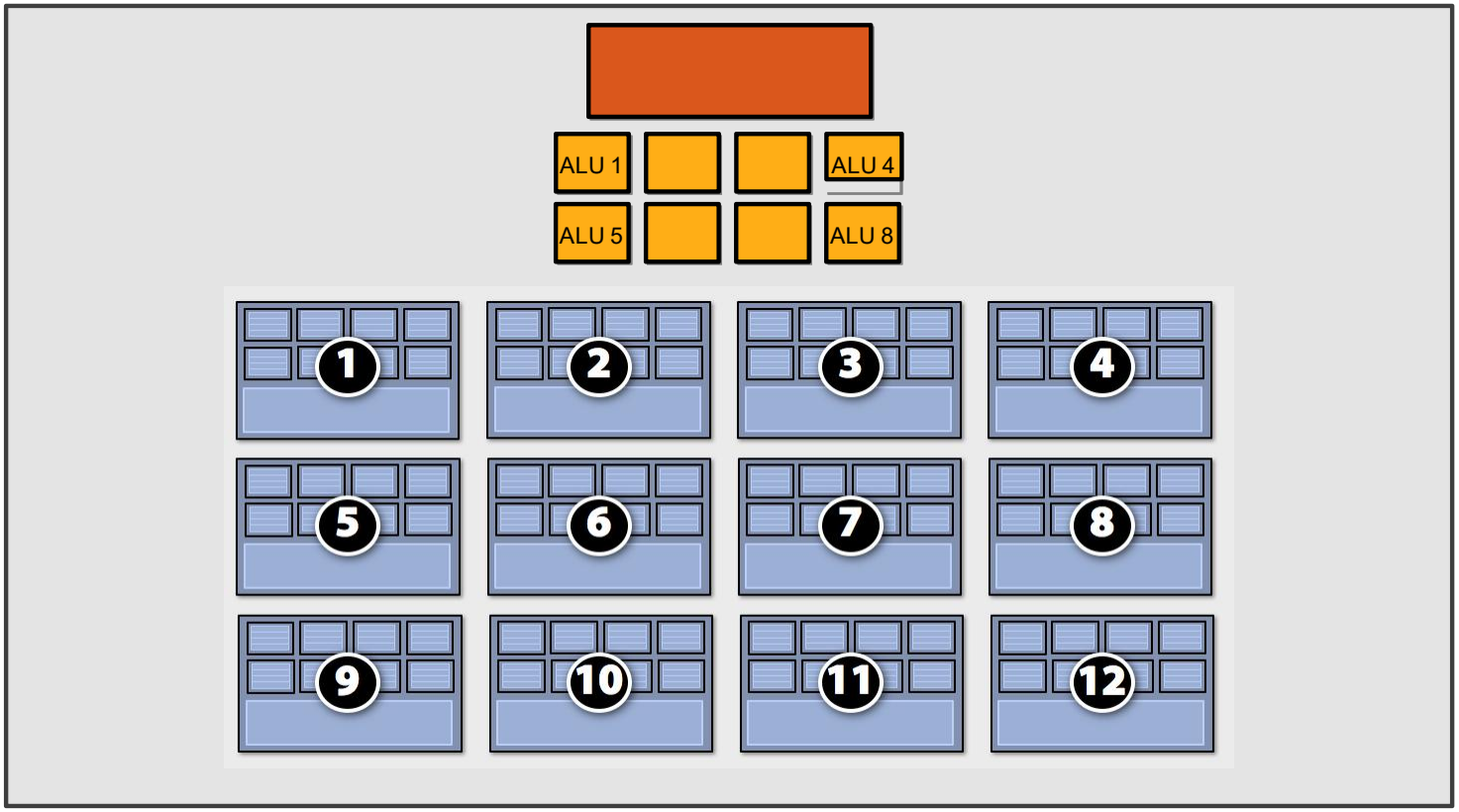
Eighteen small contexts

(maximal latency hiding)



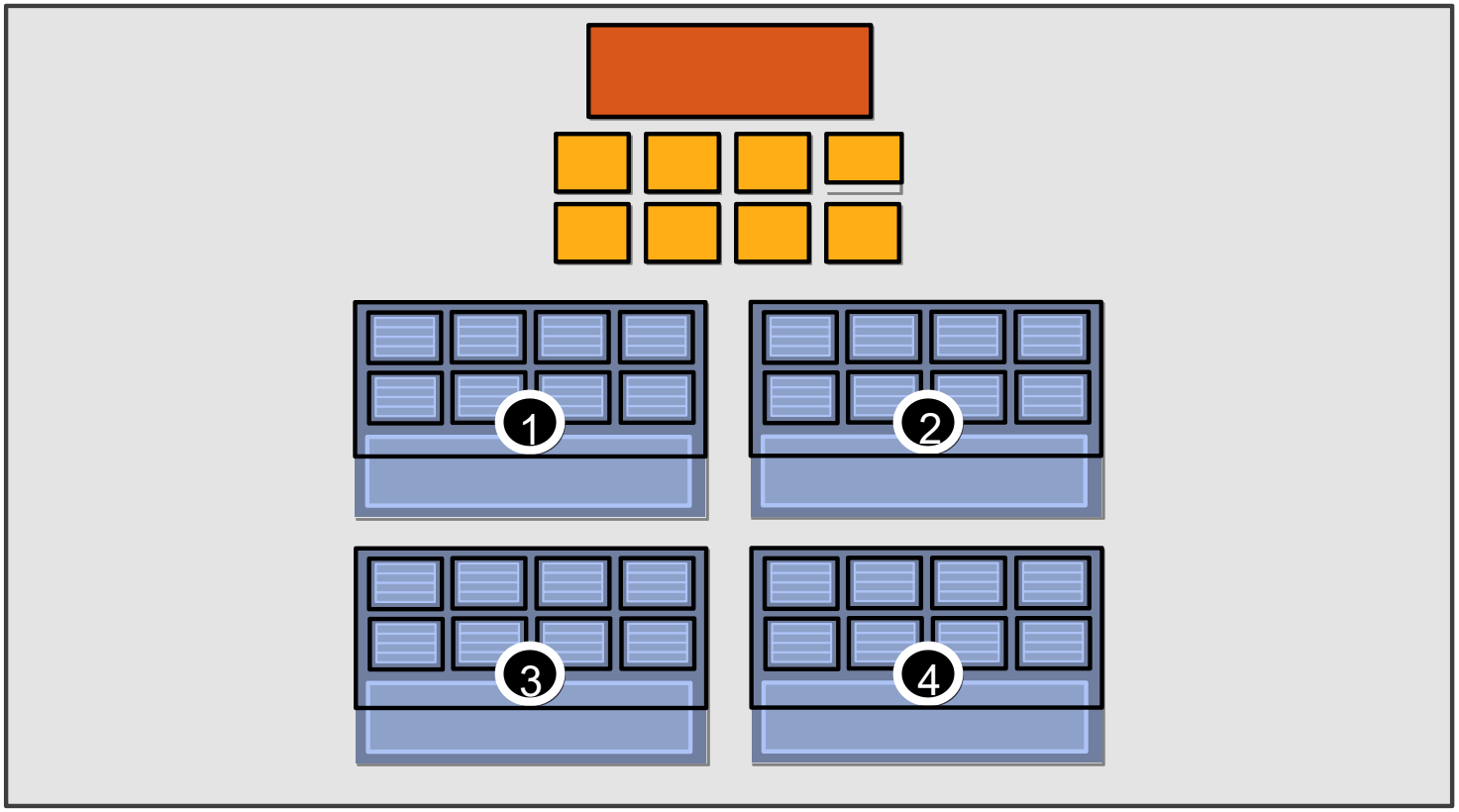
Twelve medium contexts

(medium latency hiding)



Four large contexts

(low latency hiding ability)

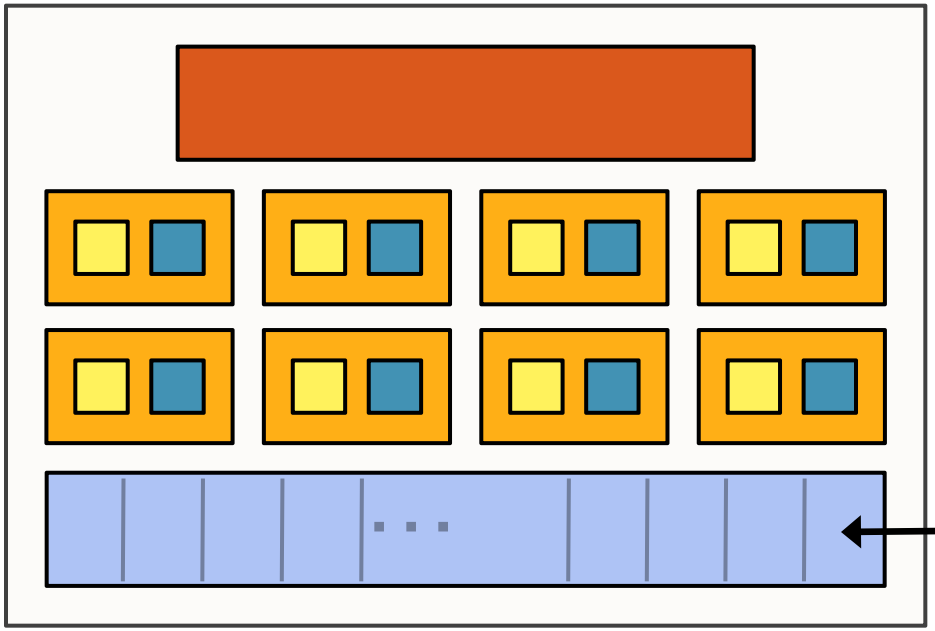


NVIDIA GeForce GTX 285

- **NVIDIA-speak:**
  - 240 stream processors
  - "SIMT execution"
  -

- **Generic speak:**
  - 30 cores
  - 8 SIMD functional units per core

NVIDIA GeForce GTX 285 "core"



64 KB of storage for thread contexts (registers)

= SIMD functional unit, control shared across 8 units

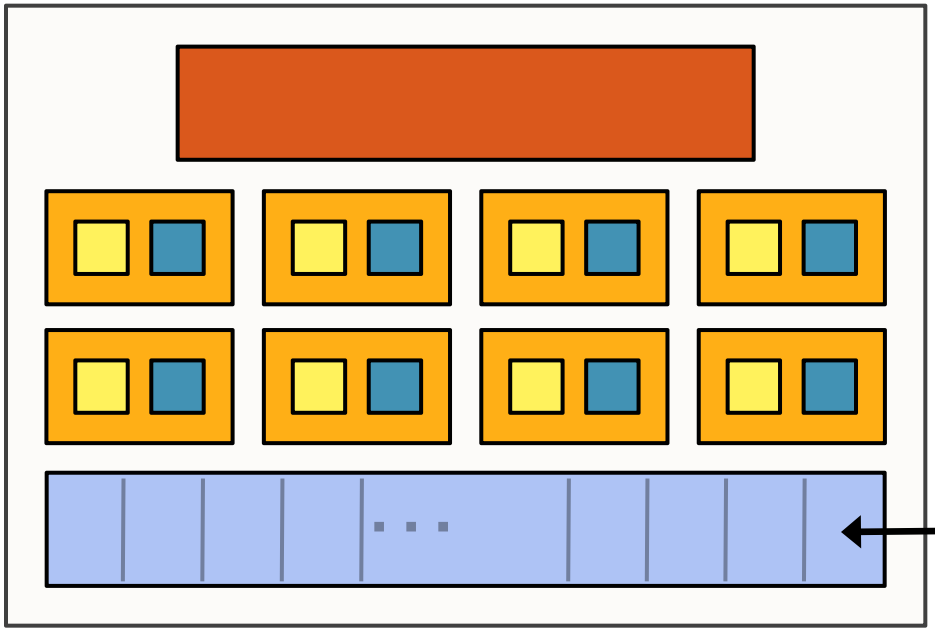
= multiply-add

= multiply

= instruction stream decode

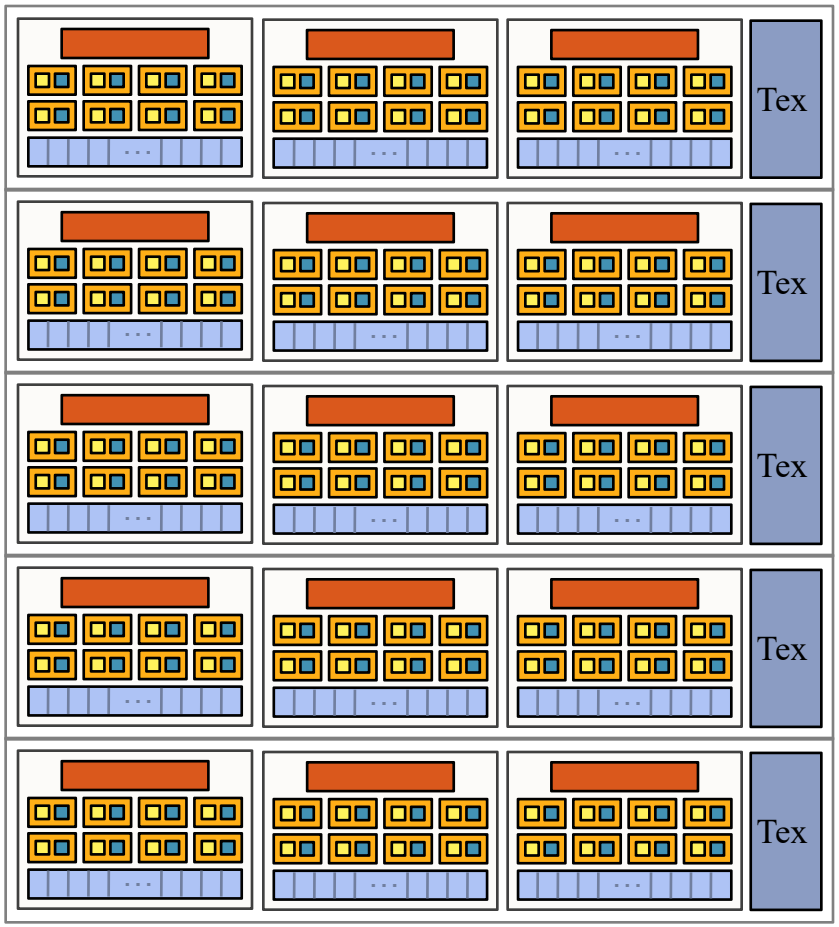
= execution context storage

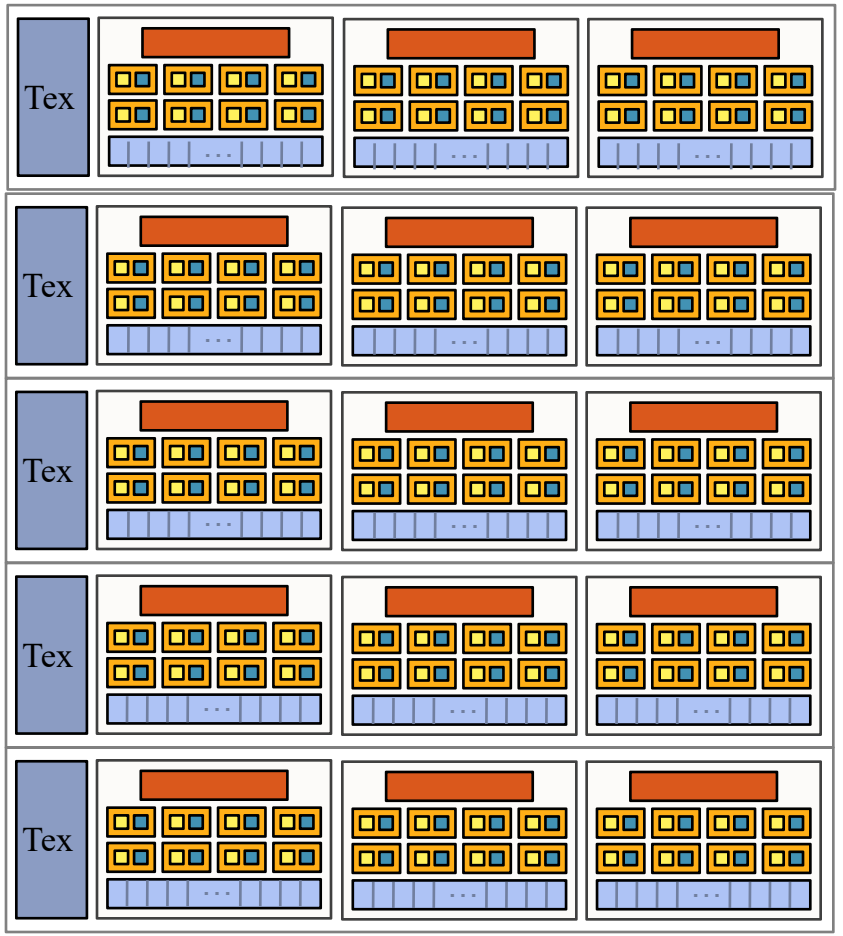
NVIDIA GeForce GTX 285 “core”



- **32个线程一组(Warp）共享一条指令流**
- **每个core可交叉执行32个 warps**
- **即每个core需存储1024个线程的上下文**
- **每个线程16个words(4B) = 64 Bytes**

NVIDIA GeForce GTX 285





- GTX 285包含30个core
- 每个core支出32个warps交叉执行，每个warps包含32个线程，共计：1024个线程
- 总计：可支持 30,720 threads （30×1024）

GP100 GPU

- 56 SMs
- 64 Lanes/SM

Or

- 3584 Stream Processors

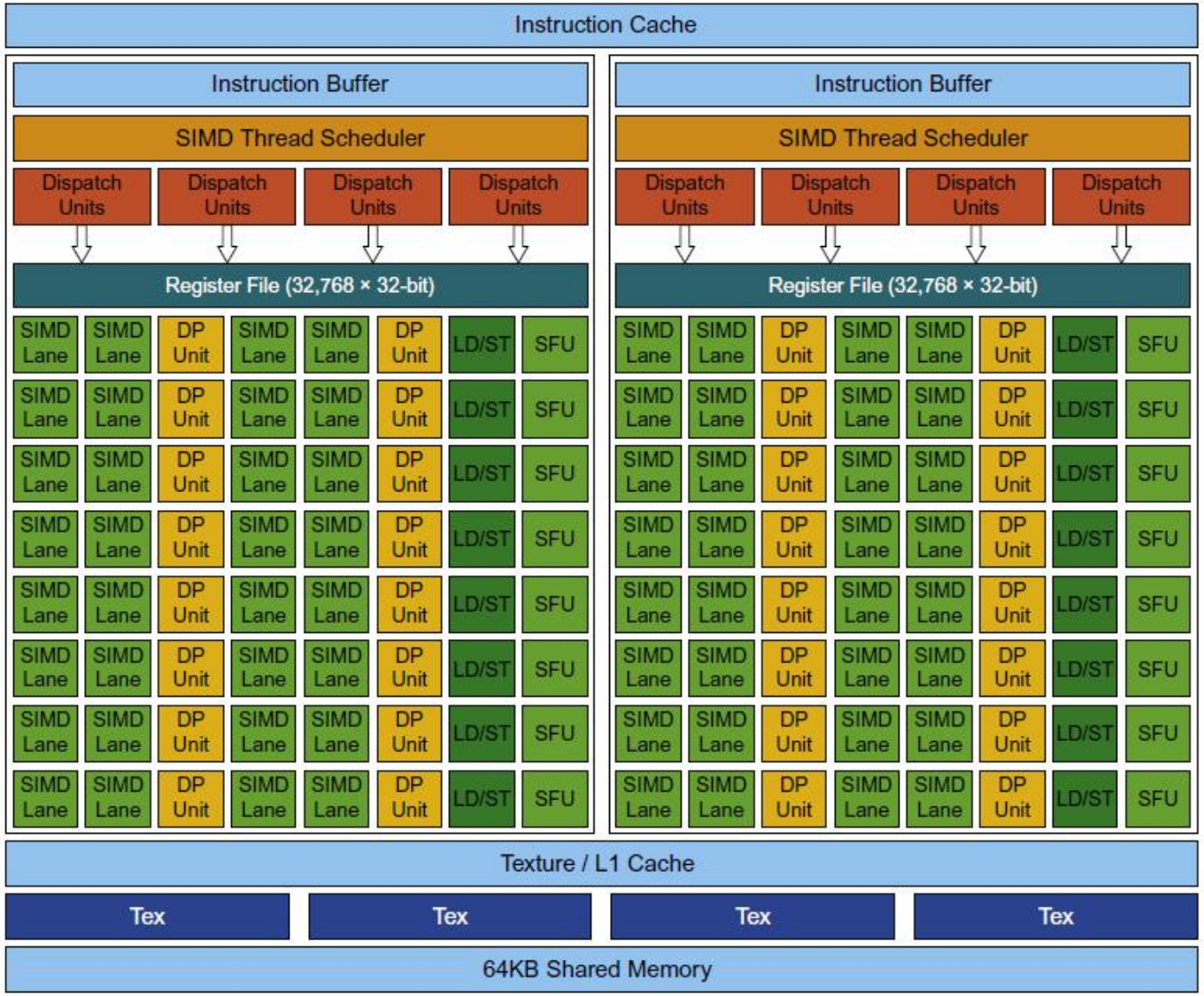


Figure 4.20 Block diagram of the multithreaded SIMD Processor of a Pascal GPU. Each of the 64 SIMD Lanes (cores) has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The 64 SIMD Lanes interact with 32 double-precision ALUs (DP units) that perform 64-bit floating-point arithmetic, 16 load-store units (LD/STs), and 16 special function units (SFUs) that calculate functions such as square roots, reciprocals, sines, and cosines.

Summary: three key ideas

- **GPU使用大量”简单核心”（多核）并行执行**
- **核心中配置大量ALU部件形成SIMD处理模式**
  - Option 1: Explicit SIMD vector instruction
  - Option 2: Implicit sharing managed by hardware
- **通过交叉执行不同线程组处理不同数据片段避免指令流运行时的长延时（Stall)**
  - When one group stalls, work on another group
- **GPU如何处理程序中的分支？vs. 向量处理器模型中的Conditional Execution**

Summary：General-Purpose GPUs

- **随着Dennard Scaling定律的失效，通过提高主频提升性能越来越困难，需要探索更有效的硬件结构-通过专用硬件结构有可能使得能效提升500+**
  - 通过配置支持向量操作的部件（例如GPU），**减少指令处理的额外开销**
  - 通过引入较复杂的操作完成多个ALU运算避免大量访问存储器 **减少数据搬移**
  - **主要挑战：**平衡专用硬件加速器带来的性能提升与系统的适用性之间的矛盾
- **基于硬件加速器（DSA）的系统正成为体系结构发展的重要方向之一**
  - 寒武纪的智能处理器：思源270、思元290、思源370、思元590
  - Google： Tensor Processing Unit
  - 机器学习应用中大量的计算任务迁移到专用加速器上运行
- **现代GPU具备图灵完备性，具有更好的实用性**
  - 支持在足够的存储空间内，用足够的时间可以完成的计算
  - 现代超级计算机大量采用GPU，以提高系统的能效 （性能/瓦）
- **GP-GPU的基本思想**
  - 发挥GPU**计算的高性能**和**存储器的高带宽**来加速数据并行性高的任务
  - 是一种协处理器（GPU作为附加设备）：CPU将数据并行的kernels迁移到GP-GPU上运行

Review：General-Purpose GPUs

- **随着Dennard Scaling定律的失效，通过提高主频提升性能越来越困难，需要探索更有效的硬件结构-通过专用硬件结构有可能使得能效提升500+**
  - 通过配置支持向量操作的部件（例如GPU），**减少指令处理的额外开销**
  - 通过引入较复杂的操作完成多个ALU运算避免大量访问存储器 **减少数据搬移**
  - **主要挑战：**平衡专用硬件加速器带来的性能提升与系统的适用性之间的矛盾
- **基于硬件加速器（DSA）的系统正成为体系结构发展的重要方向之一**
  - 寒武纪的智能处理器：思源370、思源270、思源290
  - Google： Tensor Processing Unit
  - 机器学习应用中大量的计算任务迁移到专用加速器上运行
- **现代GPU具备图灵完备性**
  - 支持在足够的存储空间内，用足够的时间可以完成的计算
  - 现代超级计算机大量采用GPU，以提高系统的能效 （性能/瓦）
- **GP-GPU的基本思想**
  - 发挥GPU**计算的高性能**和**存储器的高带宽**来加速数据并行性高的任务
  - 是一种协处理器（GPU作为附加设备）：CPU将数据并行的kernels迁移到GP-GPU上运行

Review: three key ideas

- **GPU使用大量”简单核心”（多核）并行执行**
- **核心中配置大量ALU部件形成SIMD处理模式**
  - Option 1: Explicit SIMD vector instruction
  - Option 2: Implicit sharing managed by hardware

- **通过交叉执行不同线程组处理不同数据片段避免指令流运行时的长延时（Stall)**
  - When one group stalls, work on another group

- **GPU如何处理程序中的分支？vs. 向量处理器模型中的Conditional Execution**

- **从用户的角度出发，GPU的编程模型?**

GPU II

GPU
编程模型

GPU
分支处理

GPUs are SIMD Engines Underneath

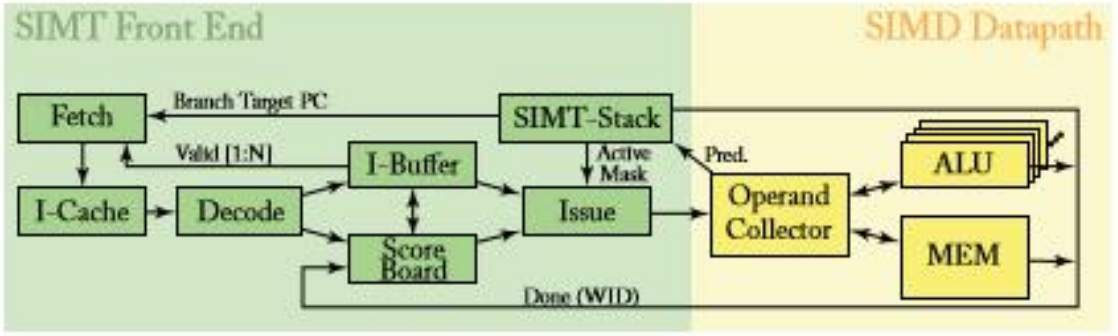


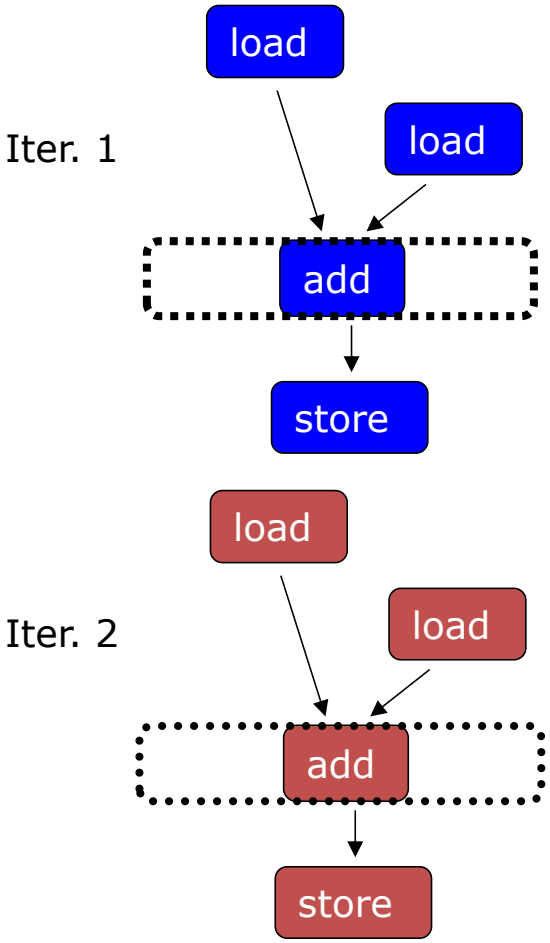
Figure 3.1: Microarchitecture of a generic GPGPU core.

- **指令流水线类似于SIMD的流水线。**
  - 不是用SIMD指令（向量指令）编程
  - 基于一般的标量指令，应用由一组线程构成。
- **两个概念**
  - Programming Model (Software) vs Execution Model (Hardware)
- **编程模型：指程序员如何描述应用（更侧重方法论）**
  - 例如, 顺序模型 (von Neumann), 数据并行, 数据流模型、多线程模型 (MIMD, SPMD), …
- **执行模型：指硬件底层如何执行代码**
  - 例如, 乱序执行、向量机、数据流处理机、多处理机、多线程处理机等
- **执行模型与编程模型可以差别很大**
  - 例如，顺序模型可以在乱序执行的处理器上执行。 SPMD 模型可以用SIMD处理器实现 (a GPU)

如何挖掘程序的并行性?

Scalar Sequential Code

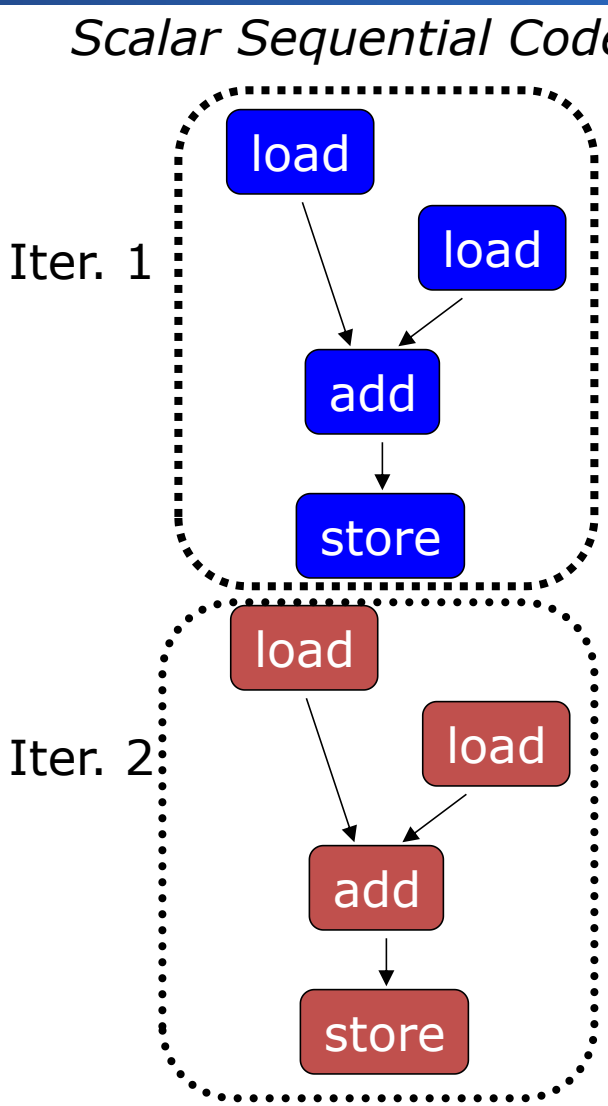
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



三种编程模式来挖掘程序的并行性:

1. Sequential (SISD)
2. Data-Parallel (SIMD)
3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)



编程模型

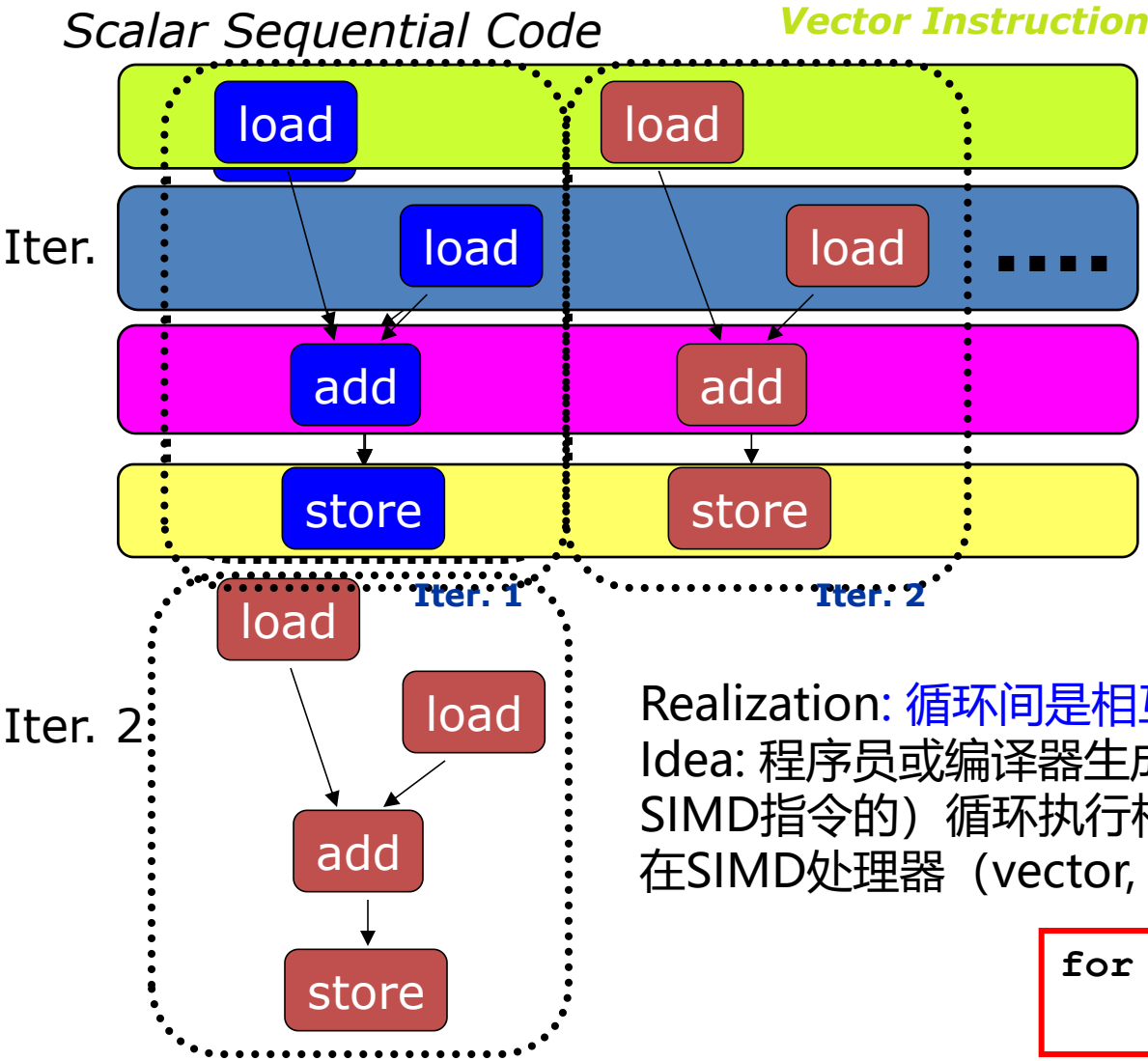
执行模型

可以采用如下不同类型的处理器执行

- **Pipelined processor**
- **Out-of-order execution processor**
  - 就绪的相互无关的指令
  - 不同循环的指令缓存在指令窗口中，多个功能部件可以并行执行
  - 即：硬件做循环展开
- **Superscalar or VLIW processor**
  - 每个cycle可以存取和执行多条指令

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Prog. Model 2: Data Parallel (SIMD)



Vectorized Code

VLD A → V1

VLD B → V2

VADD V1 + V2 → V3

VST V3 → C

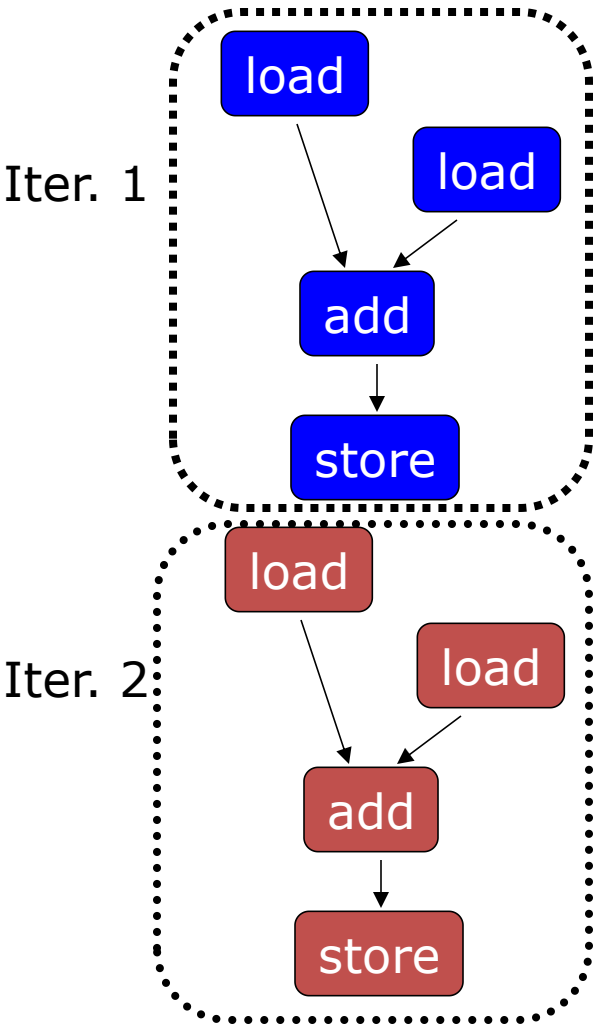
Realization: 循环间是相互独立的，循环间没有数据依赖

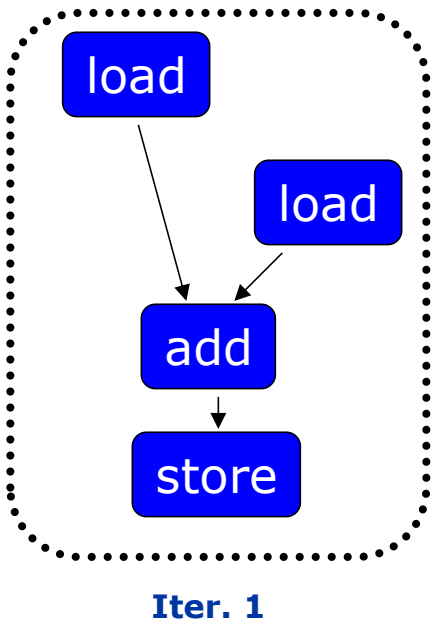
Idea: 程序员或编译器生成SIMD指令，所有的（聚合为SIMD指令的）循环执行相同的指令，处理不同的数据在SIMD处理器（vector, array）上运行

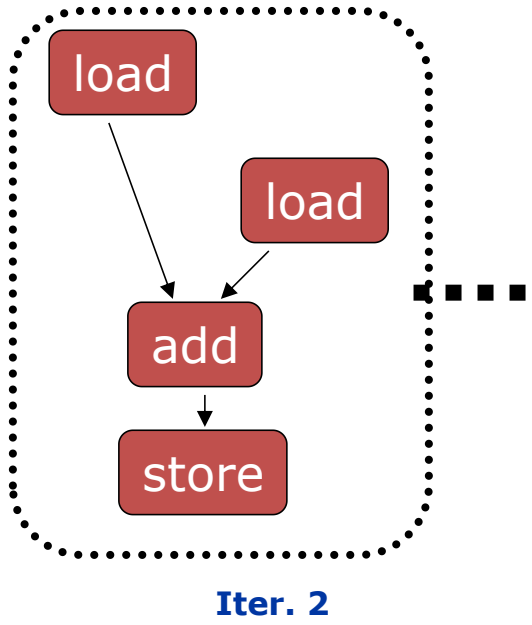
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Prog. Model 3: Multithreaded

Scalar Sequential Code







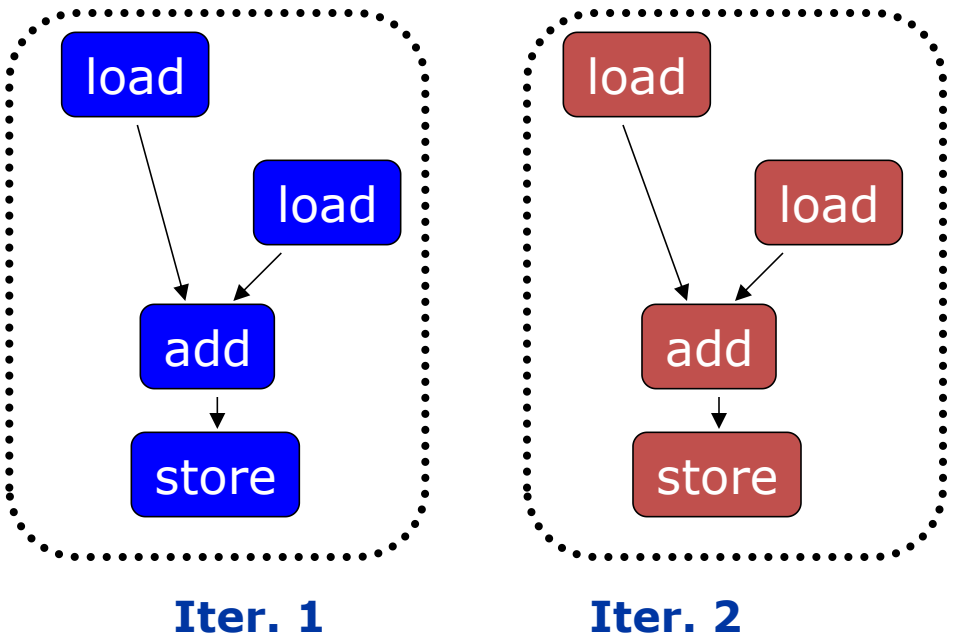
Realization: 循环间是相互独立的，循环间没有数据依赖

Idea: 程序员或编译器为每次循环生成一个线程。每个线程执行同样的指令流，处理不同的数据

可以以MIMD模式运行

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Prog. Model 3: Multithreaded




```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

....

Reali
Idea:
行同

这种模式也称为:

SPMD: Single Program Multiple Data

丸

可以

可以在SIMT 机器上运行
Single Instruction Multiple Thread

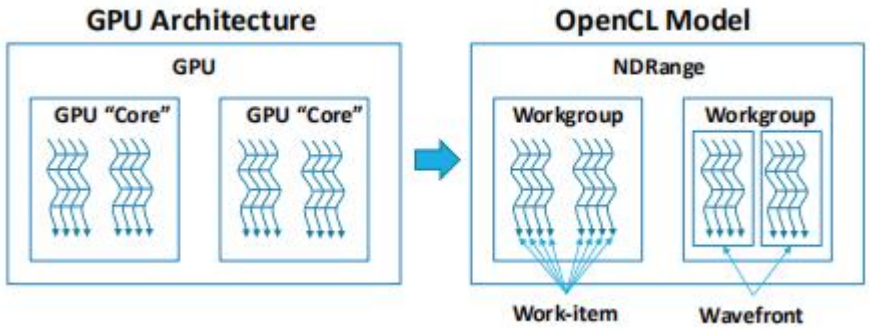
SPMD

- **Single procedure/program, multiple data**
  - 它是一种编程模型而不是计算机组织
- **每个逻辑处理单元执行同样的过程，处理不同的数据**
  - 这些过程可以在程序中的某个点上同步，例如 barriers

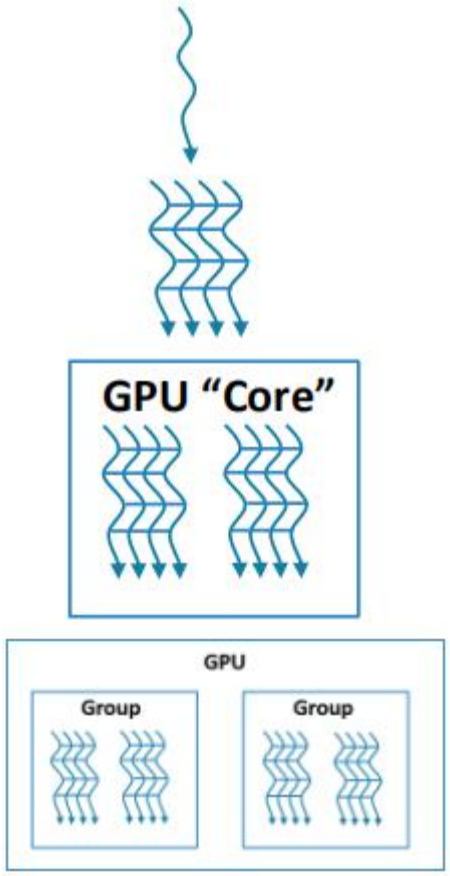
- **多条指令流执行相同的程序**
  - 每个程序/过程
    - 操作不同的数据
    - 运行时可以执行不同的控制流路径
  - 许多科学计算应用以这种方式编程，运行在MIMD硬件结构上(multiprocessors)
  - 现代 GPUs 以这种类似的方式编程，运行在SIMD硬件上

GPU 编程框架

- **编程框架：编程模型的一种实现框架。目的：方便用户，提高生产率**
- **CUDA – Compute Unified Device Architecture**
  - Nvidia 研制开发的专用模型，第一个GPGPU编程环境
- **C++ AMP – C++ Accelerated Massive Parallelism**
  - 微软研制开发，CUDA/OpenCL的更高层抽象
- **OpenACC – Open Accelerator**
  - Like OpenMP for GPUs (semi-auto-parallelize serial code)，CUDA/OpenCL的更高层抽象
  - 2011年11月，Cray、PGI、CAPS和英伟达4家公司联合推出OpenACC 1.0编程标准
- **OpenCL**
  - 最早由苹果公司研制开发，后来形成开放的异构平台编程规范
  - 异构平台编程框架，OpenCL 用来编写设备端程序
  - OpenCL工作组的成员包括：3Dlabs、AMD、苹果、ARM、Codeplay、爱立信、飞思卡尔、华为、HSA基金会、GraphicRemedy、IBM、Imagination Technologies、Intel、诺基亚、NVIDIA、摩扌 布里斯托尔大学、瑞典Ume大学



一些术语



CUDA/Nvidia	OpenCL/AMD	Henn&Patt
Thread	Work-item	Sequence of SIMD Lane Operations
Warp	Wavefront	Thread of SIMD Instructions
Block	Workgroup	Body of vectorized loop
Grid	NDRange	Vectorized loop

Threads and Blocks

- **一个线程对应一个数据元素**
- **大量的线程组织成很多线程块（Block）**
- **许多线程块组成一个网格（Grid）**

- **GPU 由硬件对线程进行管理**
  - 两级调度
    - Thread Block Scheduler
    - SIMD Thread Scheduler
  - Warp（线程束，Nvidia：32个线程）
    - SIMD线程
    - 线程调度的基本单位

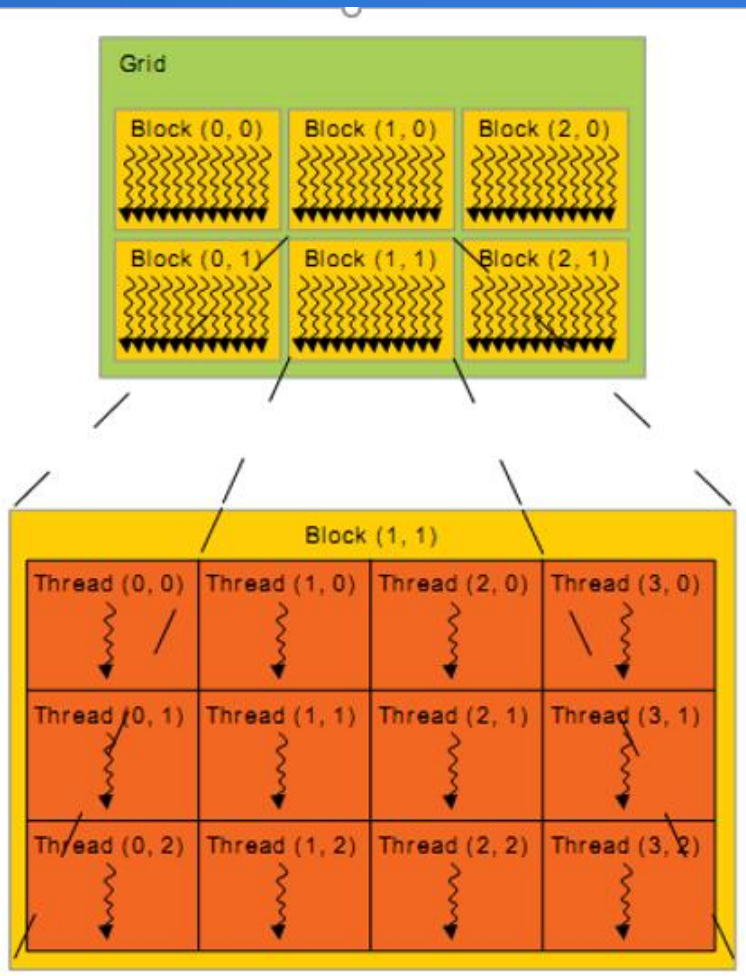
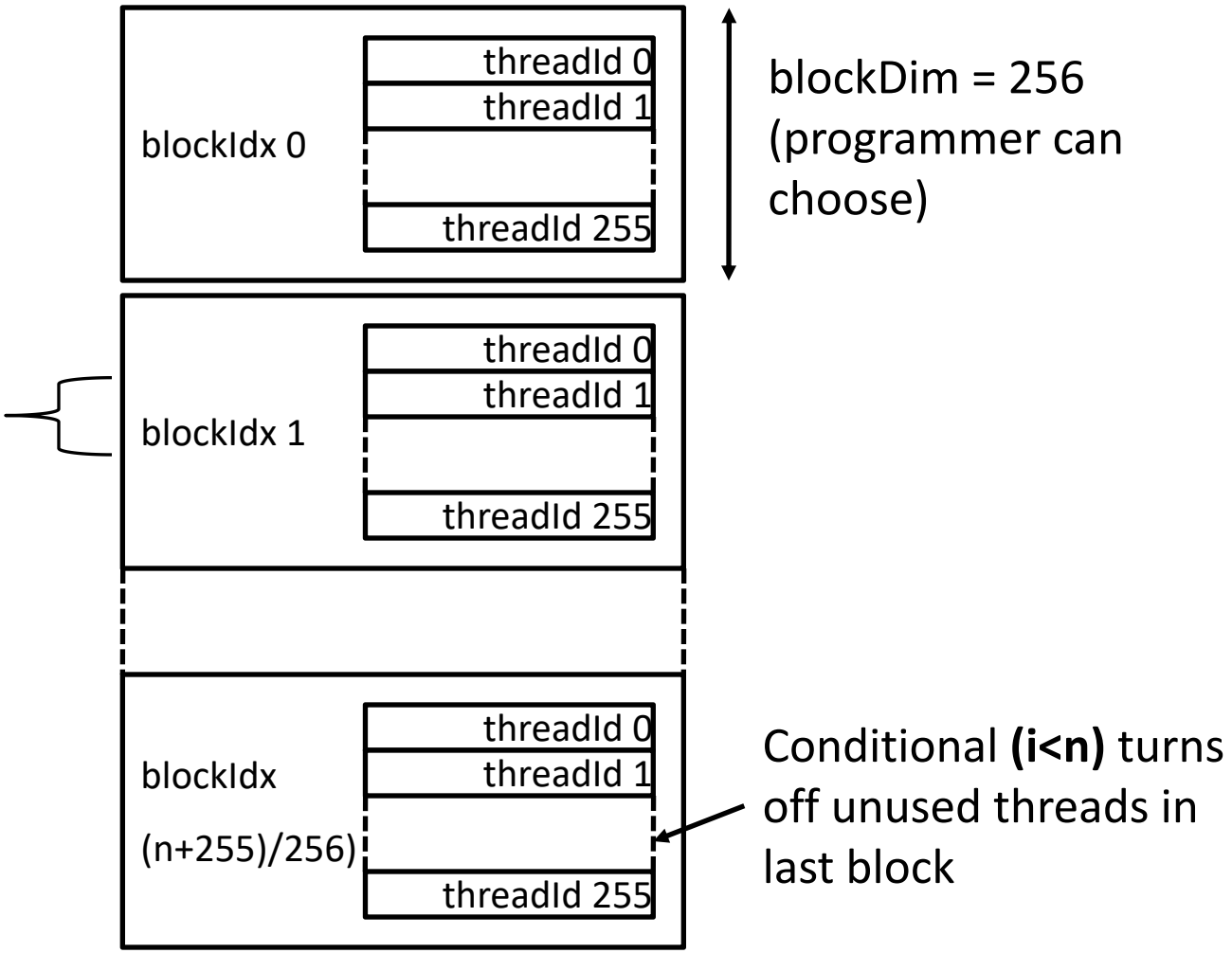


Figure 6 Grid of Thread Blocks

Programmer's View of Execution

创建足够的线程块以适应输入向量(Nvidia 中将由多个线程块构成的、在GPU上运行的代码 称为*Grid*, *Grid可以是2维的*)



Simplified CUDA Programming Model

- **计算由大量的相互独立的线程(*CUDA threads* or *microthreads*) 完成，这些线程组合成线程块（*thread blocks)***

```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{  for (int i=0; i<n; i++)
       y[i] = a*x[i] + y[i]; }

// CUDA version.
__host__   // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__   // Piece run on GP-GPU.
void daxpy(int n, double a, double*x, double*y)
{  int i = blockIdx.x*blockDim.x + threadId.x;
   if (i<n) y[i]=a*x[i]+y[i]; }
```

NVIDIA Instruction Set Arch.

- ISA 是硬件指令集的抽象
  - Parallel Thread Execution (PTX)
  - 使用虚拟寄存器
  - 用软件将其翻译成机器码
  - Example: 512threads/Block

```
shl.s32  R8, blockIdx, 9         ; Thread Block ID * Block size (512 or 2^9)
add.s32  R8, R8, threadIdx       ; R8 = i = my CUDA thread ID
ld.global.f64    RD0, [X+R8]     ; RD0 = X[i]
ld.global.f64    RD2, [Y+R8]     ; RD2 = Y[i]
mul.f64   RD0, RD0, RD4          ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64   RD0, RD0, RD2          ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64    [Y+R8], RD0     ; Y[i] = sum (X[i]*a + Y[i])
```

处理向量长度为8192的程序组织：

Grid：

1、16个block/Grid

2、512个向量元素/Block

Block：

3、16个SIMD线程/Block

4、32个CUDA线程/SIMD线程

5、处理一个元素/CUDA线程

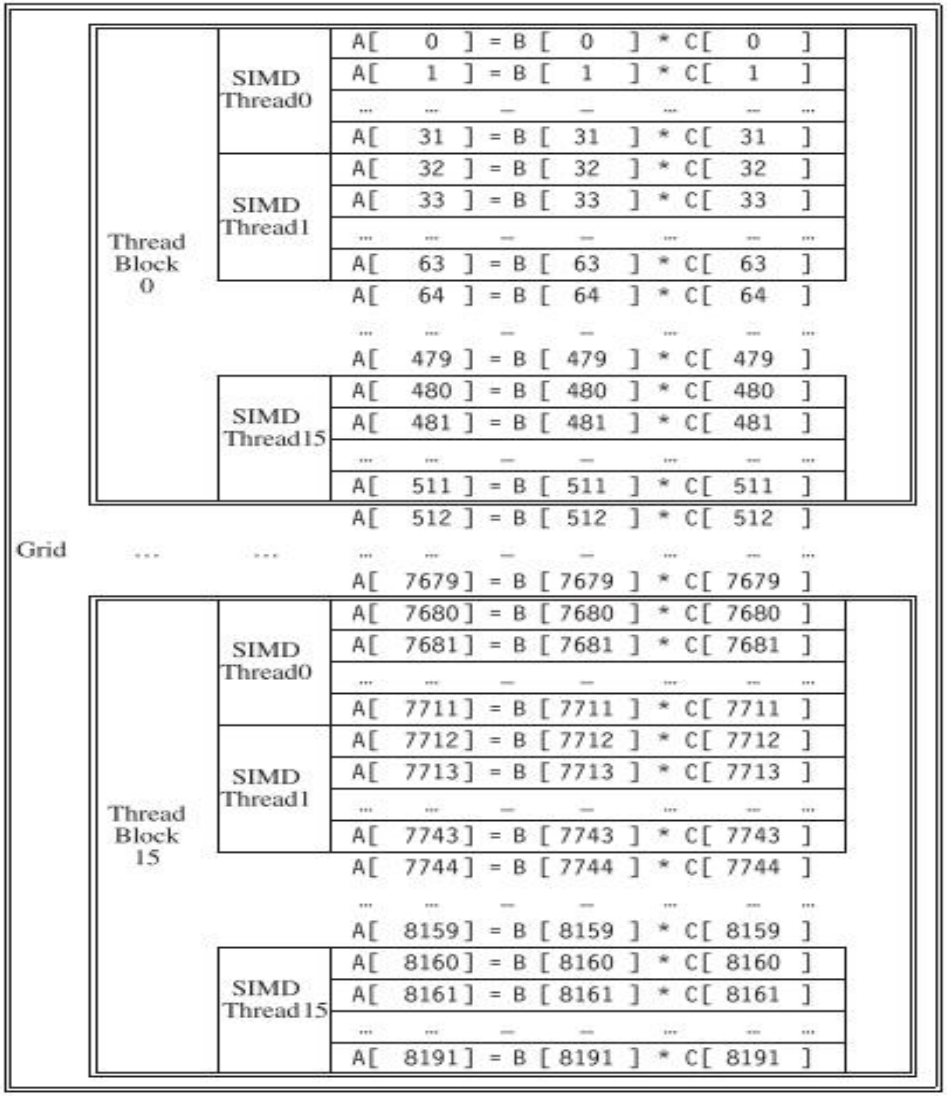
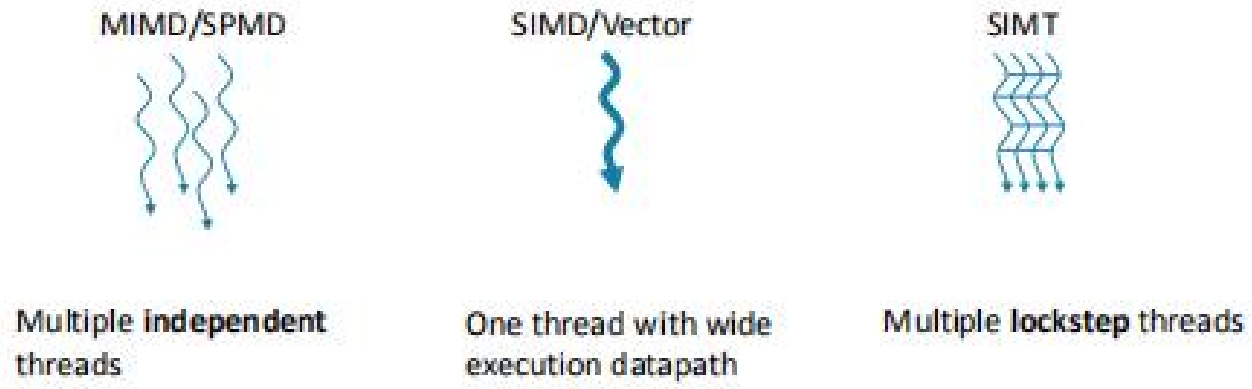


Figure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long. Each thread of SIMD instructions calculates 32 elements per instruction, and in this example, each Thread Block contains 16 threads of SIMD instructions and the Grid contains 16 Thread Blocks. The hardware Thread Block Scheduler assigns Thread Blocks to multithreaded SIMD Processors, and the hardware Thread Scheduler picks which thread of SIMD instructions to run each clock cycle within a SIMD Processor. Only SIMD Threads in the same Thread Block can communicate via local memory. (The maximum number of SIMD Threads that can execute simultaneously per Thread Block is 32 for Pascal GPUs.)

A GPU is a SIMD (SIMT) Machine

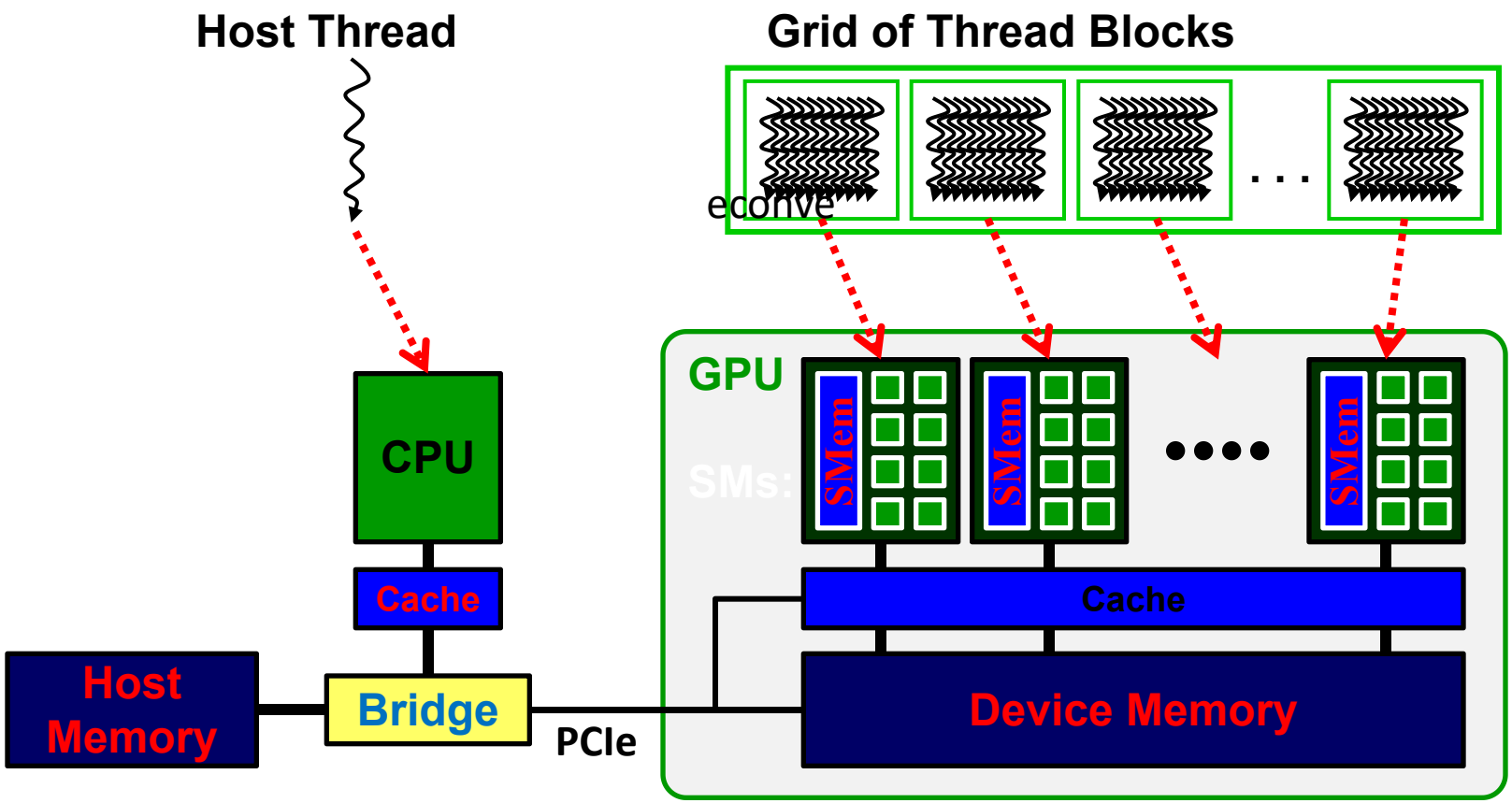
- **GPU不是用SIMD指令编程**
- **使用多线程模型 (一种SPMD 编程模型)**
  - 每个线程执行同样的代码，但操作不同的数据元素
  - 每个线程有自己的上下文(即可以独立地启动/执行等)
- **一组执行相同指令的线程由硬件动态组织成warp**
  - 一个warp是由硬件形成的SIMD操作
  - lockstep模式执行



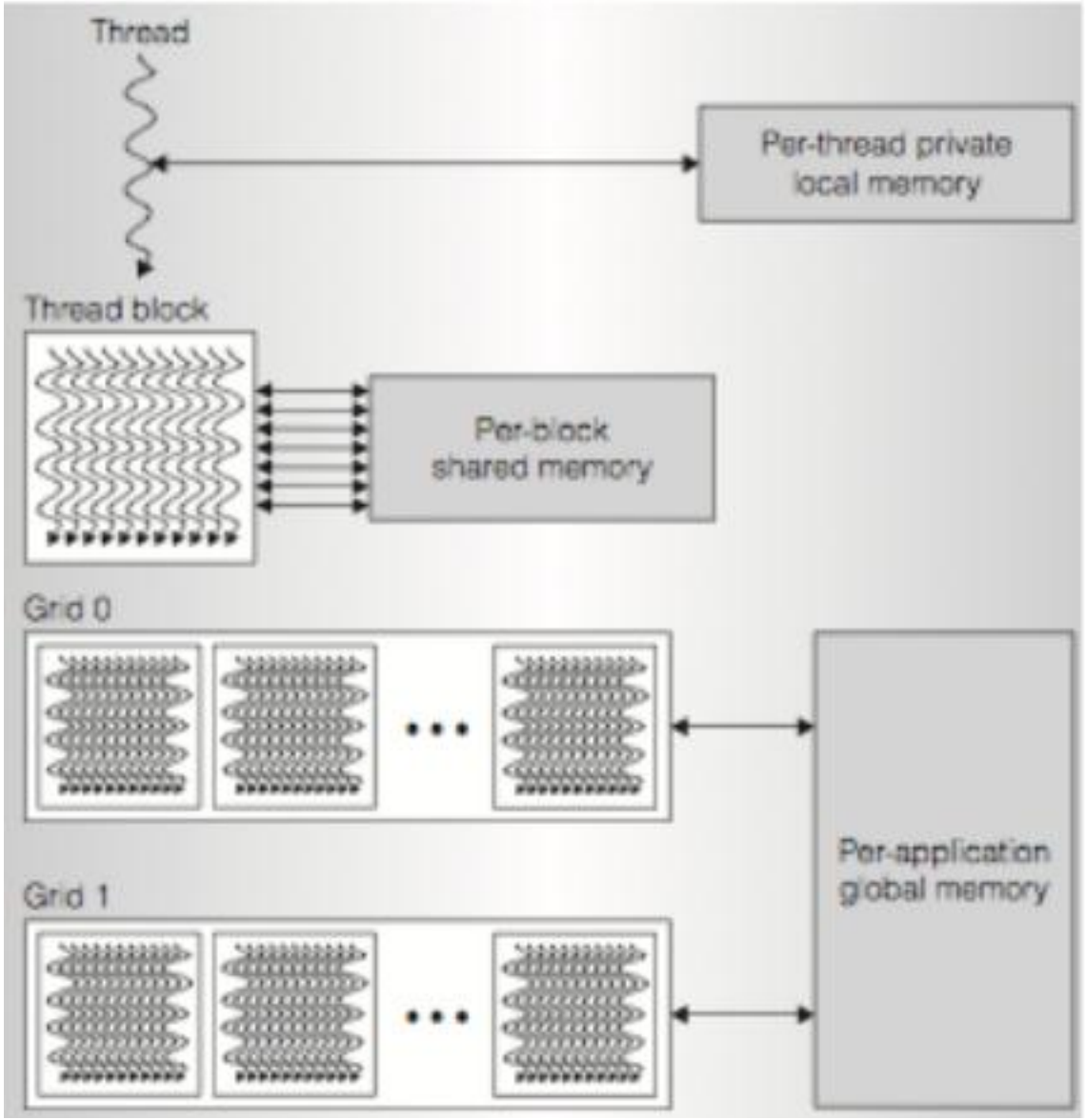
数据并行不同的执行模式

CUDA kernel maps to Grid of Blocks

```
kernel_func<<<nblk, nthread>>>(param, … );
//nblk: 线程块的总数
//nthread: 每个线程块所包含的线程数
```



GPU Memory Hierarchy



[Nvidia, 2010]

Summary- 向量处理机 vs. GPU

- **不同层次相近的术语比较**

Type	Vector term	Closest CUDA/NVIDIA GPU term	Comment
Program abstractions	Vectorized Loop	Grid	Concepts are similar, with the GPU using the less descriptive term
	Chime	—	Because a vector instruction (PTX instruction) takes just 2 cycles on Pascal to complete, a chime is short in GPUs. Pascal has two execution units that support the most common floating-point instructions that are used alternately, so the effective issue rate is 1 instruction every clock cycle
Machine objects	Vector Instruction	PTX Instruction	A PTX instruction of a SIMD Thread is broadcast to all SIMD Lanes, so it is similar to a vector instruction
	Gather/ Scatter	Global load/store (ld. global/st.global)	All GPU loads and stores are gather and scatter, in that each SIMD Lane sends a unique address. It's up to the GPU Coalescing Unit to get unit-stride performance when addresses from the SIMD Lanes allow it
	Mask Registers	Predicate Registers and Internal Mask Registers	Vector mask registers are explicitly part of the architectural state, while GPU mask registers are internal to the hardware. The GPU conditional hardware adds a new feature beyond predicate registers to manage masks dynamically

Summary-向量处理机 vs. GPU

Type	Vector term	Closest CUDA/NVIDIA GPU term	Comment
Processing and memory hardware	Vector Processor	Multithreaded SIMD Processor	These are similar, but SIMD Processors tend to have many lanes, taking a few clock cycles per lane to complete a vector, while vector architectures have few lanes and take many cycles to complete a vector. They are also multithreaded where vectors usually are not
	Control Processor	Thread Block Scheduler	The closest is the Thread Block Scheduler that assigns Thread Blocks to a multithreaded SIMD Processor. But GPUs have no scalar-vector operations and no unit-stride or strided data transfer instructions, which Control Processors often provide in vector architectures
	Scalar Processor	System Processor	Because of the lack of shared memory and the high latency to communicate over a PCI bus (1000s of clock cycles), the system processor in a GPU rarely takes on the same tasks that a scalar processor does in a vector architecture
	Vector Lane	SIMD Lane	Very similar; both are essentially functional units with registers
	Vector Registers	SIMD Lane Registers	The equivalent of a vector register is the same register in all 16 SIMD Lanes of a multithreaded SIMD Processor running a thread of SIMD instructions. The number of registers per SIMD Thread is flexible, but the maximum is 256 in Pascal, so the maximum number of vector registers is 256
	Main Memory	GPU Memory	Memory for GPU versus system memory in vector case

Figure 4.21 GPU equivalent to vector terms.

Summary-Multimedia SIMD Computers and GPUs

Feature	Multicore with SIMD	GPU
SIMD Processors	4–8	8–32
SIMD Lanes/Processor	2–4	up to 64
Multithreading hardware support for SIMD Threads	2–4	up to 64
Typical ratio of single-precision to double-precision performance	2:1	2:1
Largest cache size	40 MB	4 MB
Size of memory address	64-bit	64-bit
Size of main memory	up to 1024 GB	up to 24 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	Yes
Integrated scalar processor/SIMD Processor	Yes	No
Cache coherent	Yes	Yes on some systems

Figure 4.23 Similarities and differences between multicore with multimedia SIMD extensions and recent GPUs.

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Program abstractions	Vectorizable loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more "Thread Blocks" (or bodies of vectorized loop) that can execute in parallel. OpenCL name is "index range." AMD name is "NDRange"	A Grid is an array of Thread Blocks that can execute concurrently, sequentially, or a mixture
	Body of Vectorized loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. These SIMD Threads can communicate via local memory. AMD and OpenCL name is "work group"	A Thread Block is an array of CUDA Threads that execute concurrently and can cooperate and communicate via shared memory and barrier synchronization. A Thread Block has a Thread Block ID within its Grid
	Sequence of SIMD Lane operations	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask. AMD and OpenCL call a CUDA Thread a "work item"	A CUDA Thread is a lightweight thread that executes a sequential program and that can cooperate with other CUDA Threads executing in the same Thread Block. A CUDA Thread has a thread ID within its Thread Block
Machine object	A thread of SIMD instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results are stored depending on a per-element mask. AMD name is "wavefront"	A warp is a set of parallel CUDA Threads (e.g., 32) that execute the same instruction together in a multithreaded SIMT/SIMD Processor
	SIMD instruction	PTX instruction	A single SIMD instruction executed across the SIMD Lanes. AMD name is "AMDIL" or "FSAIL" instruction	A PTX instruction specifies an instruction executed by a CUDA Thread

Figure 4.24 Conversion from terms used in this chapter to official NVIDIA/CUDA and AMD jargon. OpenCL names are given in the book's definitions.

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Processing hardware	Multithreaded SIMD processor	Streaming multiprocessor	Multithreaded SIMD Processor that executes thread of SIMD instructions, independent of other SIMD Processors. Both AMD and OpenCL call it a "compute unit." However, the CUDA programmer writes program for one lane rather than for a "vector" of multiple SIMD Lanes	A streaming multiprocessor (SM) is a multithreaded SIMT/SIMD Processor that executes warps of CUDA Threads. A SIMT program specifies the execution of one CUDA Thread, rather than a vector of multiple SIMD Lanes
	Thread Block Scheduler	Giga Thread Engine	Assigns multiple bodies of vectorized loop to multithreaded SIMD Processors. AMD name is "Ultra-Threaded Dispatch Engine"	Distributes and schedules Thread Blocks of a grid to streaming multiprocessors as resources become available
	SIMD Thread scheduler	Warp scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution. AMD name is "Work Group Scheduler"	A warp scheduler in a streaming multiprocessor schedules warps for execution when their next instruction is ready to execute
	SIMD Lane	Thread processor	Hardware SIMD Lane that executes the operations in a thread of SIMD instructions on a single element. Results are stored depending on mask. OpenCL calls it a "processing element." AMD name is also "SIMD Lane"	A thread processor is a datapath and register file portion of a streaming multiprocessor that executes operations for one or more lanes of a warp

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Memory hardware	GPU Memory	Global memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU. OpenCL calls it "global memory"	Global memory is accessible by all CUDA Threads in any Thread Block in any grid; implemented as a region of DRAM, and may be cached
	Private memory	Local memory	Portion of DRAM memory private to each SIMD Lane. Both AMD and OpenCL call it "private memory"	Private "thread-local" memory for a CUDA Thread; implemented as a cached region of DRAM
	Local memory	Shared memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors. OpenCL calls it "local memory." AMD calls it "group memory"	Fast SRAM memory shared by the CUDA Threads composing a Thread Block, and private to that Thread Block. Used for communication among CUDA Threads in a Thread Block at barrier synchronization points
	SIMD Lane registers	Registers	Registers in a single SIMD Lane allocated across body of vectorized loop. AMD also calls them "registers"	Private registers for a CUDA Thread; implemented as multithreaded register file for certain lanes of several warps for each thread processor

Figure 4.25 Conversion from terms used in this chapter to official NVIDIA/CUDA and AMD jargon. Note that our descriptive terms "local memory" and "private memory" use the OpenCL terminology. NVIDIA uses SIMT (single-instruction multiple-thread) rather than SIMD to describe a streaming multiprocessor. SIMT is preferred over SIMD because the per-thread branching and control flow are unlike any SIMD machine.

Summary- 向量处理机 vs. GPU

不同层次相近的术语比较

Type	Descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Short explanation
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via local memory
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it only contains SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes

Summary-向量处理机 vs. GPU

Type	Descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Short explanation
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors
	SIMD Thread Scheduler	Thread Scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full Thread Block (body of vectorized loop)

Figure 4.12 Quick guide to GPU terms used in this chapter. We use the first column for hardware terms. Four groups cluster these 11 terms. From top to bottom: program abstractions, machine objects, processing hardware, and memory hardware. Figure 4.21 on page 312 associates vector terms with the closest terms here, and Figure 4.24 on page 317 and Figure 4.25 on page 318 reveal the official CUDA/NVIDIA and AMD terms and definitions along with the terms used by OpenCL.

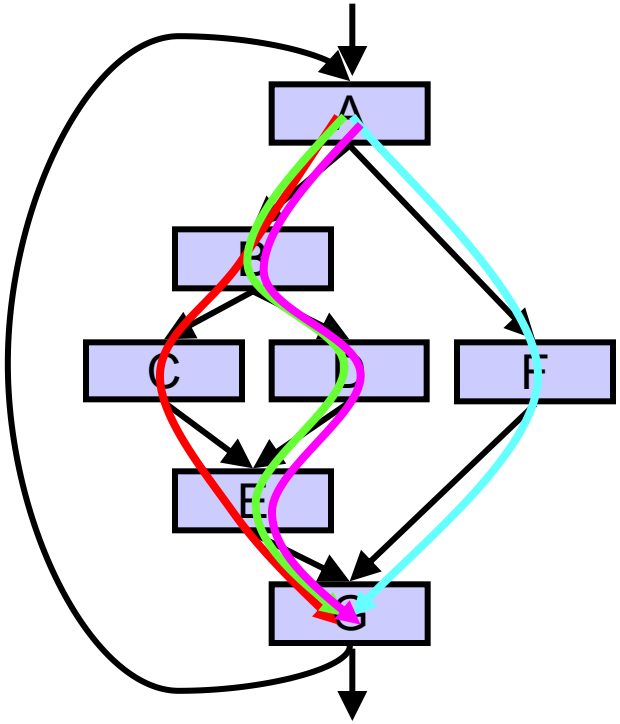
GPU II

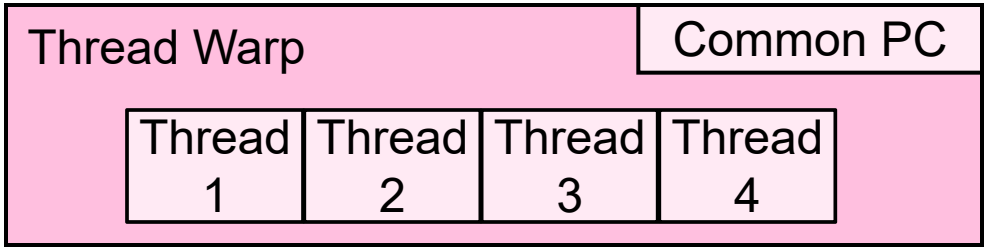
GPU
编程模型

GPU
分支处理

Threads Can Take Different Paths in Warp-based SIMD

- 每个线程可以包含控制流指令
- 这些线程可以执行不同的控制流路径



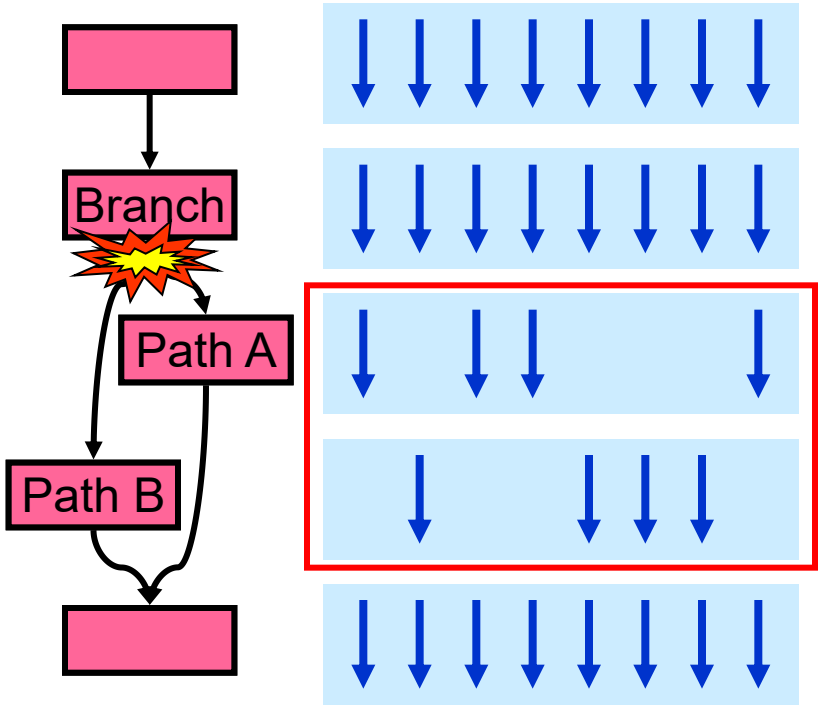


Slide credit: Tor Aamodt

Control Flow Problem in GPUs/SIMT

- GPU 控制逻辑使用 SIMD 流水线 以节省资源
  - 这些标量线程构成 warp

- 当一个WARP中的线程分支到不同的执行路径时，产生分支发散（Branch divergence）



与向量处理机模型的条件执行类似
（Vector Mask and Masked Vector Operations?）

Conditional Branching

- **与向量结构类似, GPU 使用内部的屏蔽字(masks)**
- **还使用了 (SIMT-Stack)**
  - 分支同步堆栈
    - 保存分支的不同路径地址
    - 保存该路径的SIMD lane 屏蔽字(mask)
      - 即指示哪些车道可以提交结果
  - 指令标记(instruction markers)
    - 管理何时分支（divergence）到多个执行路径，何时路径汇合(converge)
- **PTX层**
  - CUDA线程的控制流由PTX分支指令(branch、call、return and exit）控制
  - 每个线程车道包含由程序员指定的1-bit谓词寄存器
- **GPU硬件指令层,控制流包括:**
  - 分支指令(branch,jump call return)
  - **特殊的指令用于管理分支同步栈**
  - GPU硬件为每个SIMD thread 提供堆栈 保存分支的路径
  - GPU硬件指令带有控制每个线程车道的1-bit谓词寄存器

Example

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

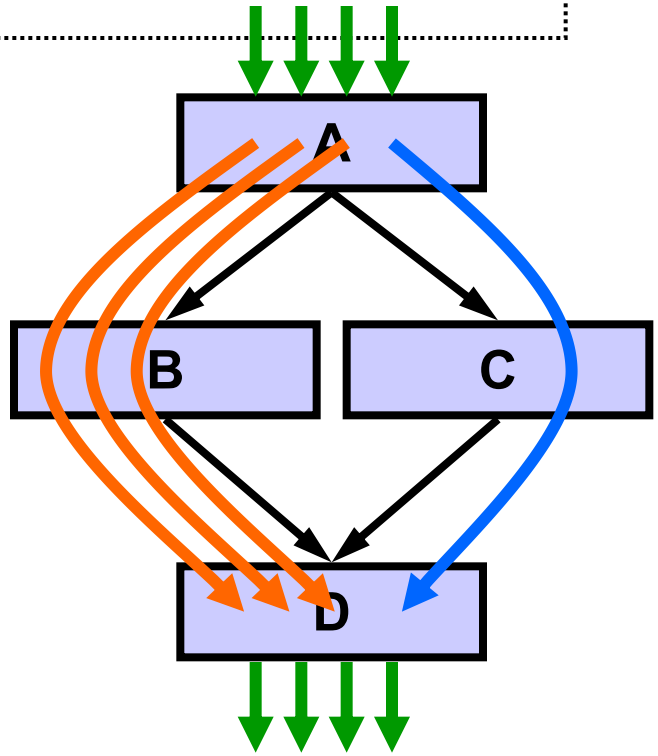
```
        ld.global.f64  RD0, [X+R8]  ; RD0 = X[i]
        setp.neq.s32  P1, RD0, #0   ; P1 is predicate register 1
        @!P1, bra  ELSE1, *Push     ; Push old mask, set new mask bits
                                    ; if P1 false, go to ELSE1
        ld.global.f64 RD2, [Y+R8]  ; RD2 = Y[i]
        sub.f64 RD0, RD0, RD2      ; Difference in RD0
        st.global.f64 [X+R8], RD0  ; X[i] = RD0
       @P1, bra  ENDIF1, *Comp     ; complement mask bits
                                   ; 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>, *Pop  ; pop to restore old mask
```

Branch divergence

- **硬件跟踪各μthreads转移的方向（判定哪些是成功的转移，哪些是失败的转移）**
- **如果所有线程路径相同，则保持这种 SIMD 执行模式**
- **如果各线程选择的方向不一致，那么在SIMT-Stack中保存分支分叉地址、对应的屏蔽字、以及分支汇聚点**
- **SIMD 车道何时执行分支同步堆栈中的路径?**
  - 通过执行pop操作，弹出执行路径以及屏蔽字，执行该转移路径
  - SIMD lane完成整个分支路径执行后再执行下一条指令 称为converge(汇聚)
  - 对于相同长度的路径，IF-THEN-ELSE 操作 的 效率平均为50%

Branch Divergence Handling

```
A;
if (some condition) {
   B;
} else {
   C;
}
D;
```

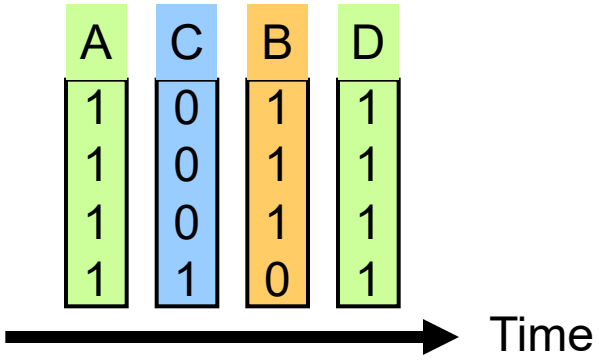


One per warp

Control Flow Stack

	Next PC	Recv PC	Active Mask
TOS →	D	--	1111
	B	D	1110
	C	D	0001

Execution Sequence



SIMT stack operation

```
do {
  t1 = tid*N;          // A
  t2 = t1 + i;
  t3 = data1[t2];
  t4 = 0;
  if( t3 != t4 ) {
    t5 = data2[t2]; // B
    if( t5 != t4 ) {
      x += 1;          // C
    } else {
      y += 2;          // D
    }
  } else {
    z += 3;            // F
  }
  i++;                 // G
} while( i < N );
```

Example CUDA C source code for illustrating SIMT stack operation

```
A:    mul.lo.u32      t1, tid, N;
      add.u32         t2, t1, i;
      ld.global.u32   t3, [t2];
      mov.u32         t4, 0;
      setp.eq.u32     p1, t3, t4;
@p1   bra             F;
B:    ld.global.u32   t5, [t2];
      setp.eq.u32     p2, t5, t4;
@p2   bra             D;
C:    add.u32         x, x, 1;
      bra             E;
D:    add.u32         y, y, 2;
E:    bra             G;
F:    add.u32         z, z, 3;
G:    add.u32         i, i, 1;
      setp.le.u32     p3, i, N;
@p3   bra             A;
```

Example PTX assembly code for illustrating SIMT stack operation.

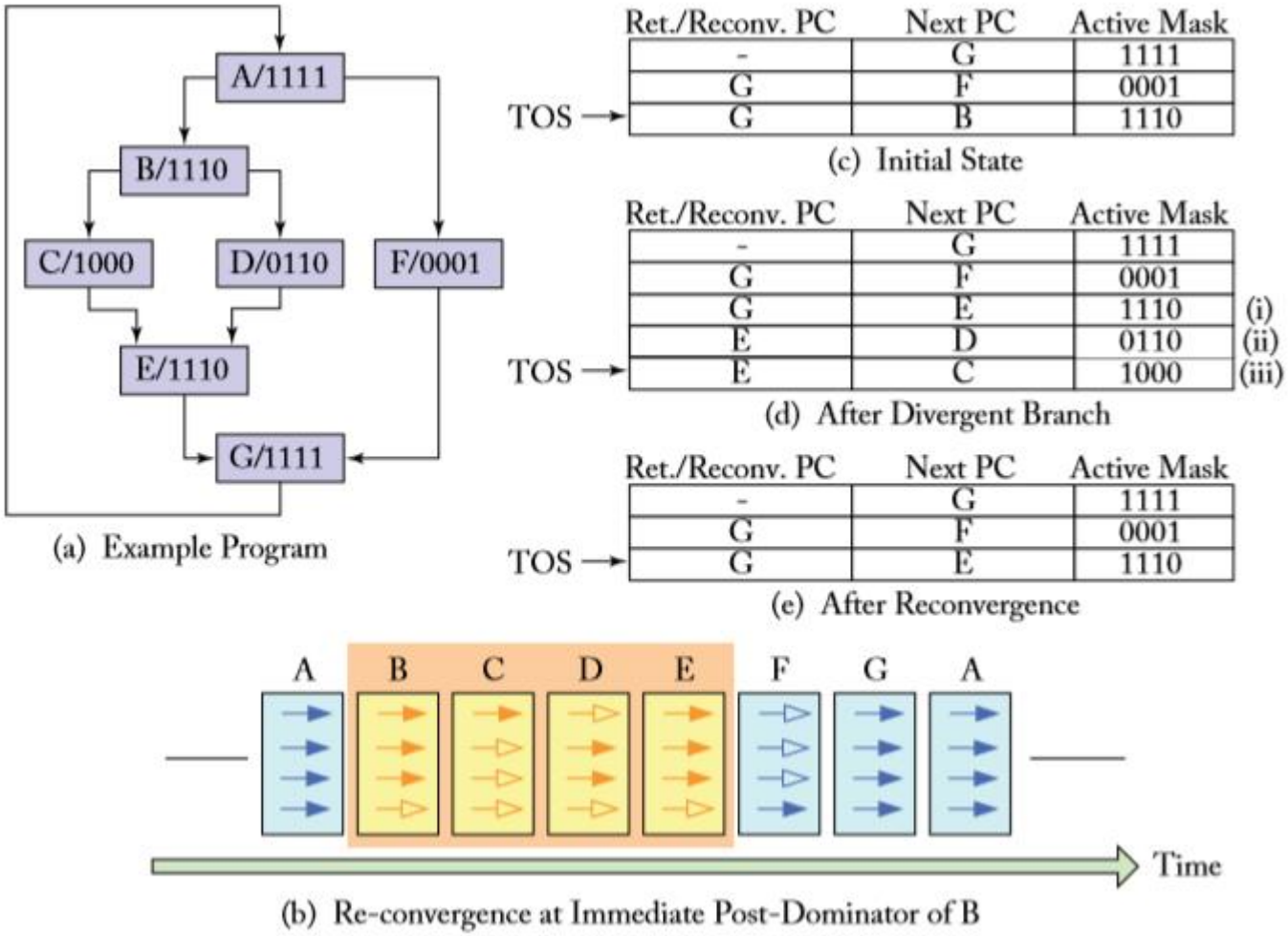


Figure 3.4: Example of SIMT stack operation (based on Figure 5 from Fung et al. [2007]).

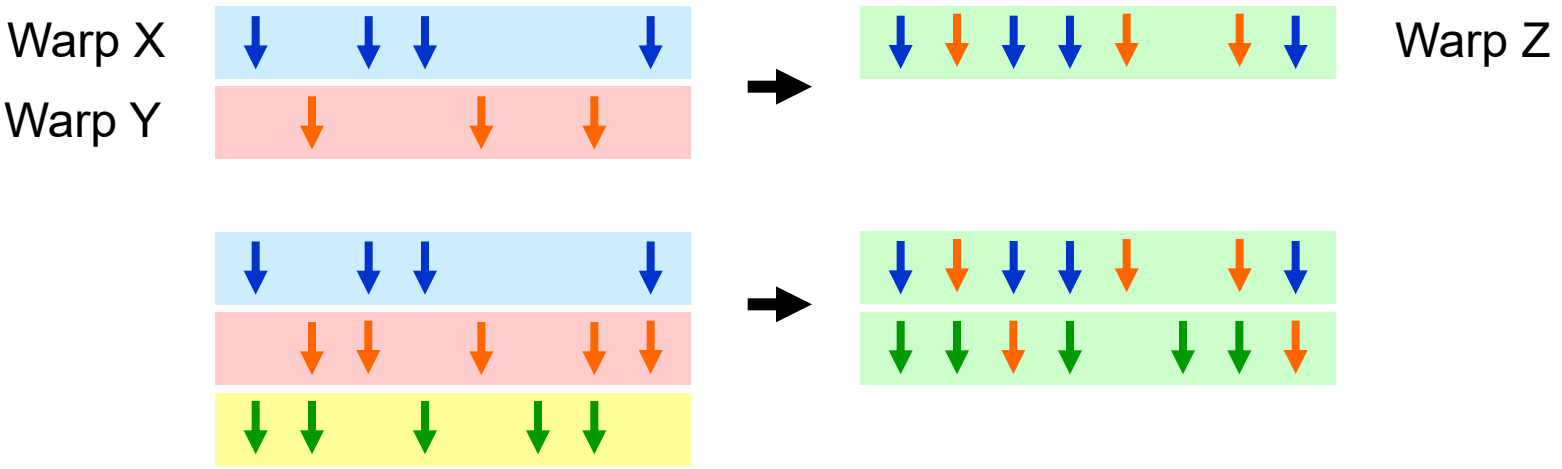
Remember: Each Thread Is Independent

- **SIMT 主要优点：**
  - 可以独立地处理线程,即每个线程可以在任何标量流水线上单独执行（ MIMD 处理模式）
  - 可以将线程组织成warp，即可以将执行相同指令流的线程构成warp，形成SIMD 处理模式, 以发挥SIMD处理的优势

- **如果有许多线程，对具有相同PC值的线程可以将它们动态组织到一个warp中**
- **这样可以减少“分支发散” →提高SIMD 利用率**
  - SIMD 利用率: 执行有用操作的SIMD lanes的比例 (即, 执行活动线程的比例)

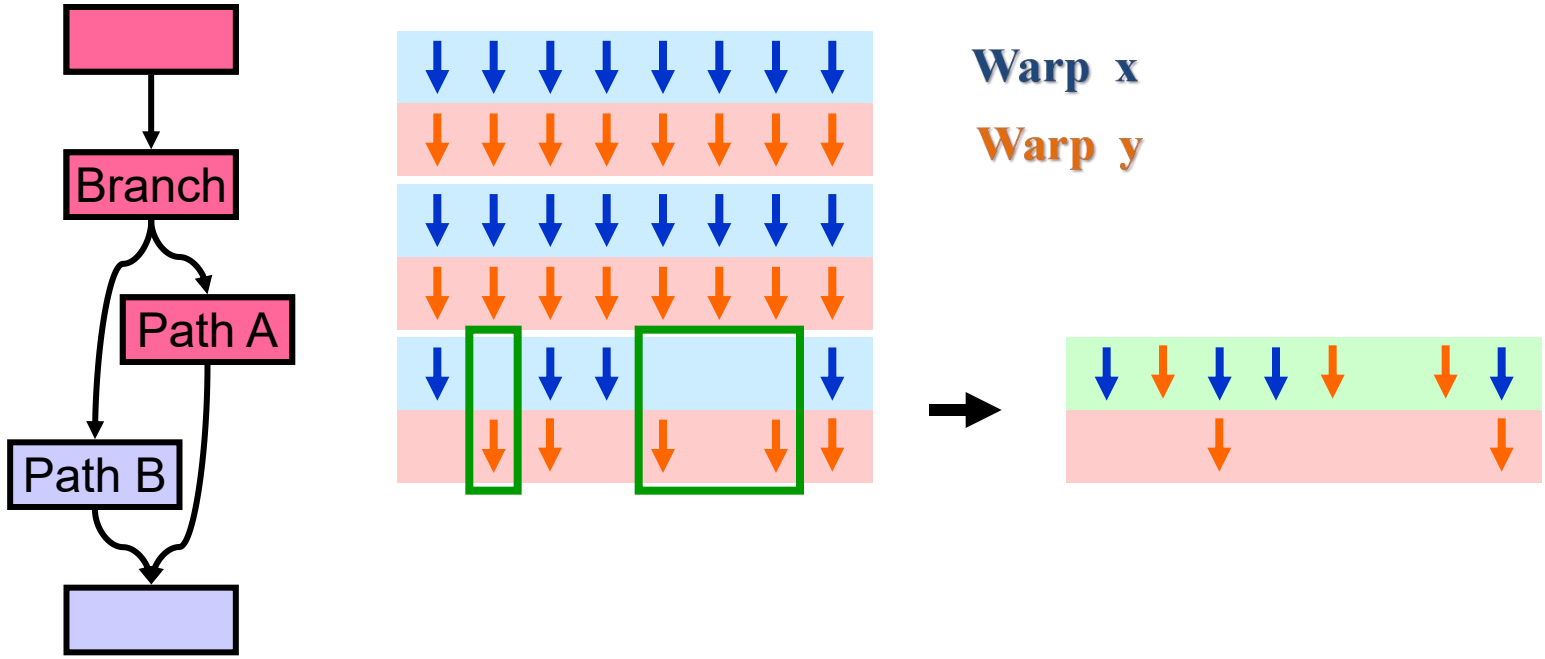
Dynamic Warp Formation/Merging

- **Idea:** 分支发散之后，动态合并执行相同指令的线程
  - 从那些等待的warp中形成新的warp
  - 足够多的线程分支到每个路径，有可能创建完整的新warp



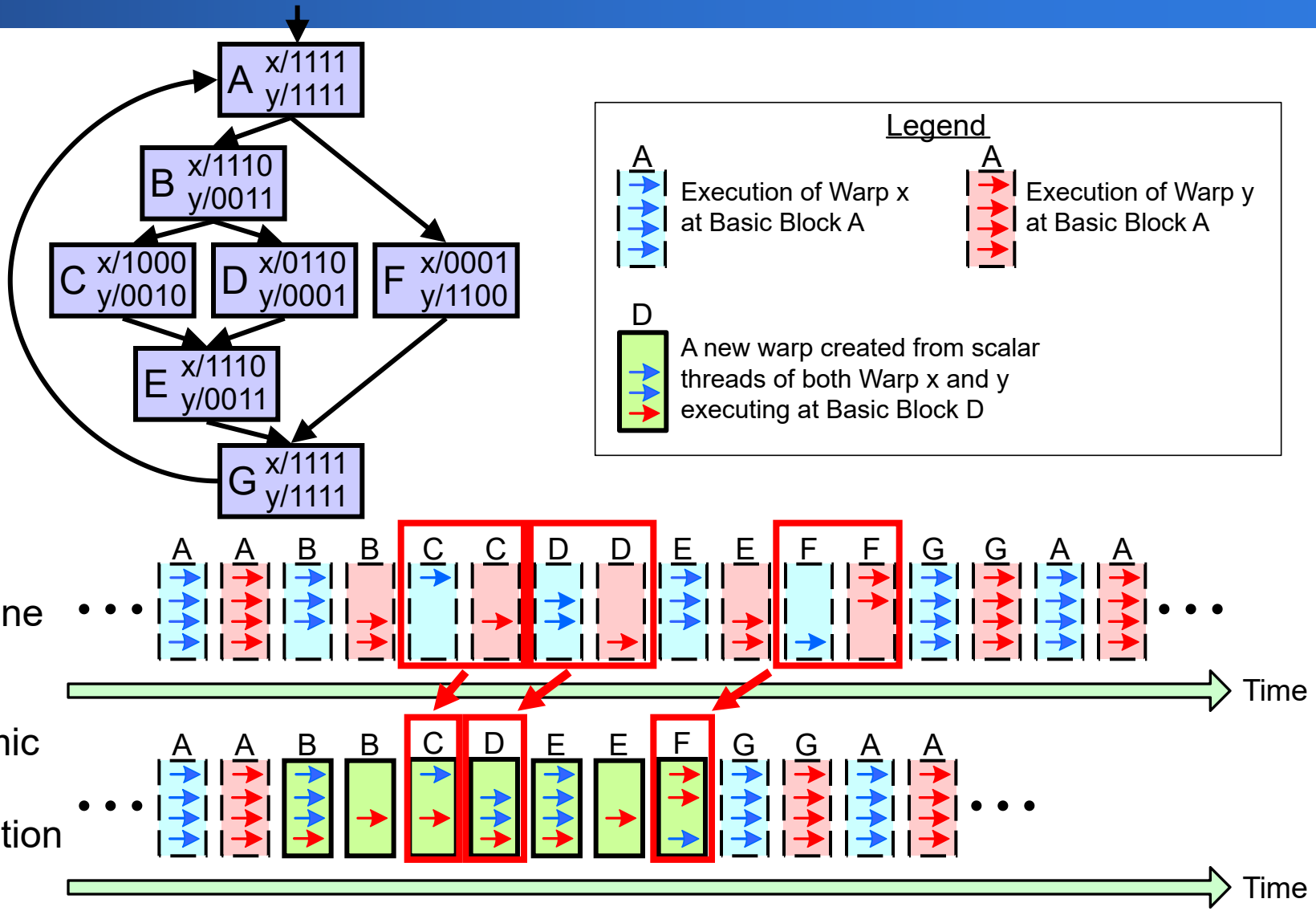
Dynamic Warp Formation/Merging

- Idea:
  - 分支发散之后，动态合并执行相同指令的线程

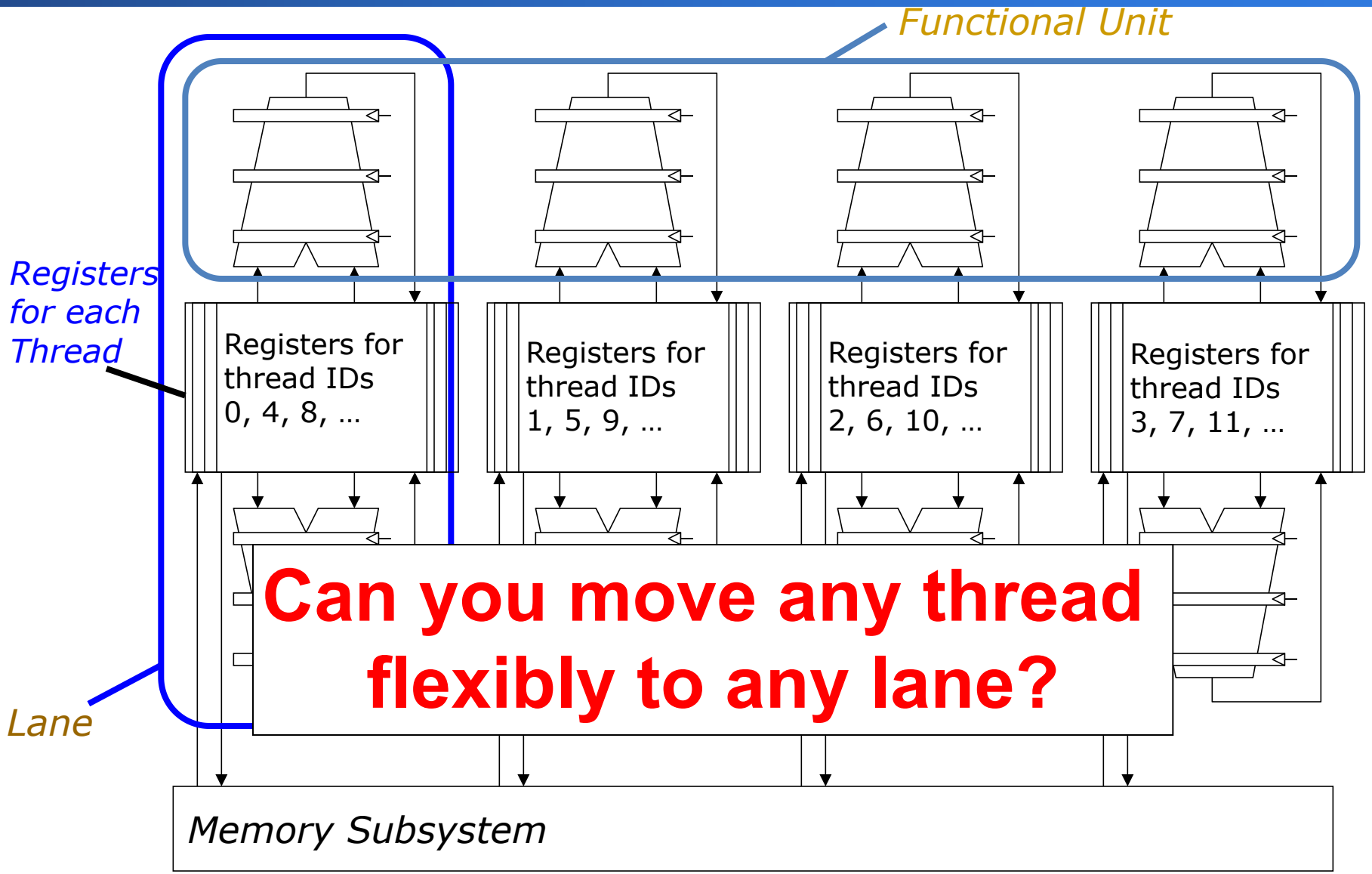


- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

Dynamic Warp Formation Example



Hardware Constraints Limit Flexibility of Warp Grouping



When You Group Threads Dynamically

- **存储器访问如何处理?**

- **固定模式的存储器访问相对简单，当动态构成warp时，使得访问模式具有随机性，使得问题变得复杂。→ 降低存储器访问的局部性**
  - → 导致存储器带宽利用率的下降

What About Memory Divergence?

- 现代 GPUs 包括高速缓存，减少对存储器的访问
- Ideally: 一个warp中的所有线程的存储器访问都命中 (互相没有冲突)
- Problem: 一个Warp中有些命中，有些失效
- Problem: 一个线程的stall导致整个warp停顿

- 需要有相关技术来解决存储器发散访问问题

Part2：数据级并行

- **DLP技术基础：研究动机及种类**
  - 从效率瓶颈到DLP解决方案
  - DLP核心定义与并行范式对比
- **向量处理机：DLP的基础形态**
  - 向量处理器硬件结构
  - 向量指令集与代码示例
  - 向量处理器性能计算与优化
- **SIMD指令集扩展：DLP的轻量化落地**
- **GPU：DLP的高性能实战形态**
  - GPU硬件架构与SIMT模型
  - GPU编程模型与AI任务适配
- **AI专用处理器：DLP的AI场景定制形态**
  - AI专用处理器的发展历程：从学术探索到产业爆发
  - 典型AI专用处理器架构与DLP实现
- **四类DLP技术综合对比**

AI专用处理器的发展历程：从学术探索到产业爆发

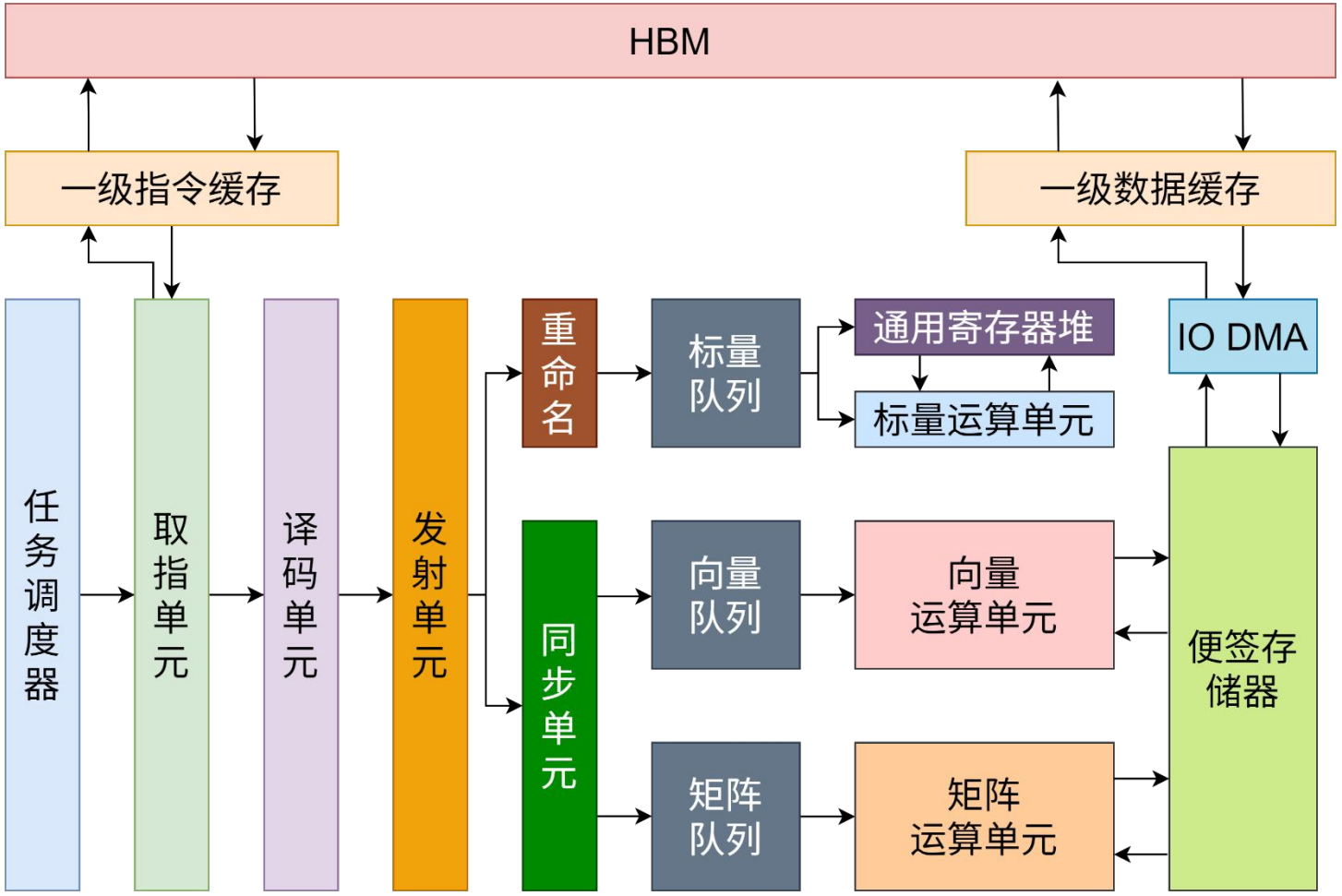
- **阶段一：学术奠基期（2012-2015 年）—DLP 技术的 AI 适配探索**
  - 技术背景：2012 年 AlexNet 证明深度神经网络的可行性，但 CPU/GPU 能效比仅 10-50 FLOPS/W，无法满足大规模部署需求
  - 关键突破：
    - 2013 年，中科院计算所提出 “Diannao” 架构（寒武纪前身）
    - 2014 年，斯坦福大学提出 Eyeriss 架构
  - 产业特征：
    - 以学术论文为主，未形成商用产品，核心贡献是验证 “AI 专用 DLP 架构” 的可行性。
- **阶段二：商用启动期（2016-2019 年）—— 云端专用芯片崛起**
  - 技术里程碑：
    - 从2016年开始：MLU 端侧（寒武纪1A ,220)，MLU云侧（100，290, 590)
    - 2016年谷歌推出商用AI处理器 （TPU v1） （脉动阵列）
    - 2017年华为麒麟970集成寒武纪NPU，首次实现手机端AI专用DLP加速
    - 2018 年，寒武纪发布思元 100，国内首款云端 AI 专用处理器，支持多精度计算；
    - 2019 年，华为发布昇腾 310/910，基于自研达芬奇架构，支持大规模模型训练；
  - 技术特征：以云端训练 / 推理为核心场景，采用 DLP 架构，能效比较 GPU 提升 5-10 倍。
- **阶段三：全域普及期(2020-2024年) —— 云端边协同发展**
  - 产品：
    - 谷歌：TPU v4 （2020), TPU v5e (2024)，AMD, Intel, Apple
    - 寒武纪：思元590 （2024），华为，阿里、百度.....
  - 产业特征：
    - 形成 “云端训练（TPU / 昇腾）+ 边缘推理（思元 220）+ 端侧智能（手机 / PC NPU）” 的全场景布局，DLP 技术从 “单一张量并行” 升级为 “稀疏计算 + 算子融合 + 低精度” 的综合优化。

- **阶段四：未来演进期（2025 年及以后）—— 存算融合与自适应架构**
  - 技术趋势：
    - 基于新型存储器（如ReRAM）的存算融合架构（如 HP 实验室原型）
    - 如何进一步降低访存开销。如将计算与存储集成
    - 如何平衡通用性与效率；如支持多种张量尺寸，
  - 应用预判：
    - 适配千亿参数大模型的分布式 DLP 架构、边缘设备的纳瓦级能效比芯片成为研发重点。

2012	2016	2019	2024	2025+
学术探索	商用启动	端元协同	全域普及	存算融合
Diannao	TPU v1	昇腾910	TPU v5e	自适应架构
	麒麟970	思元220	思元590	存算一体芯片

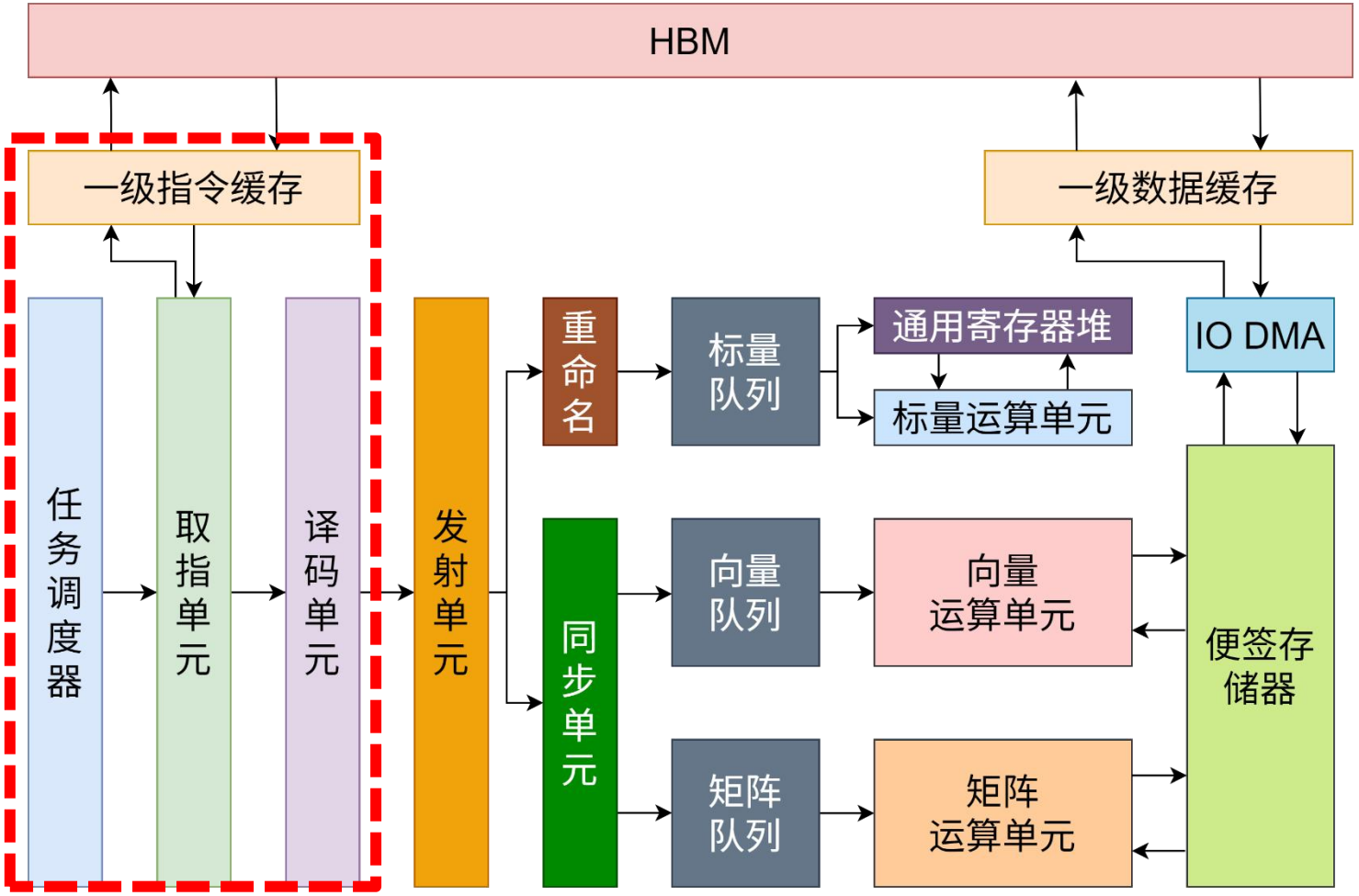
人工智能处理器总体架构

- **单核架构（Cambricon架构为例）**



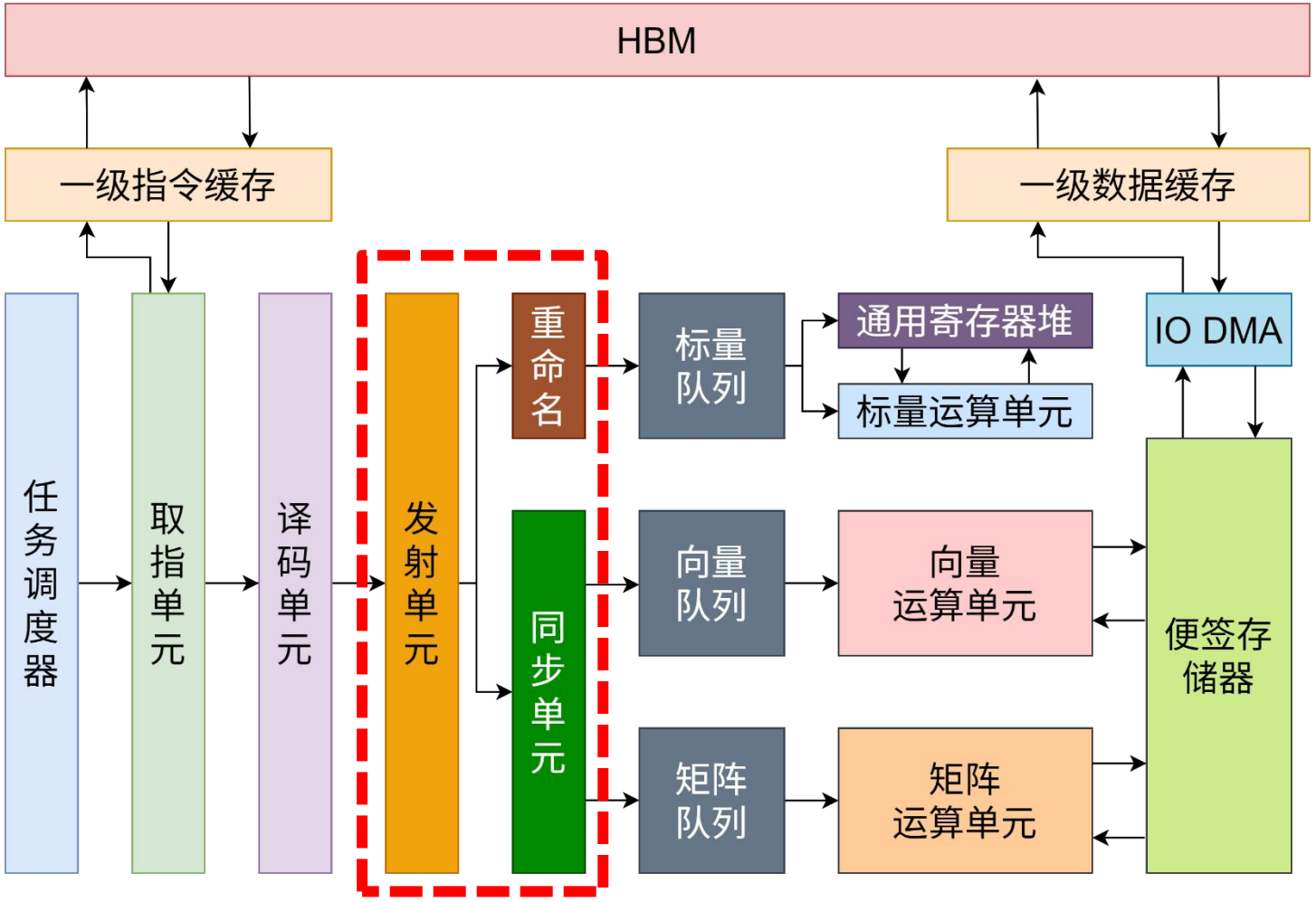
人工智能处理器总体架构

- **取指译码**



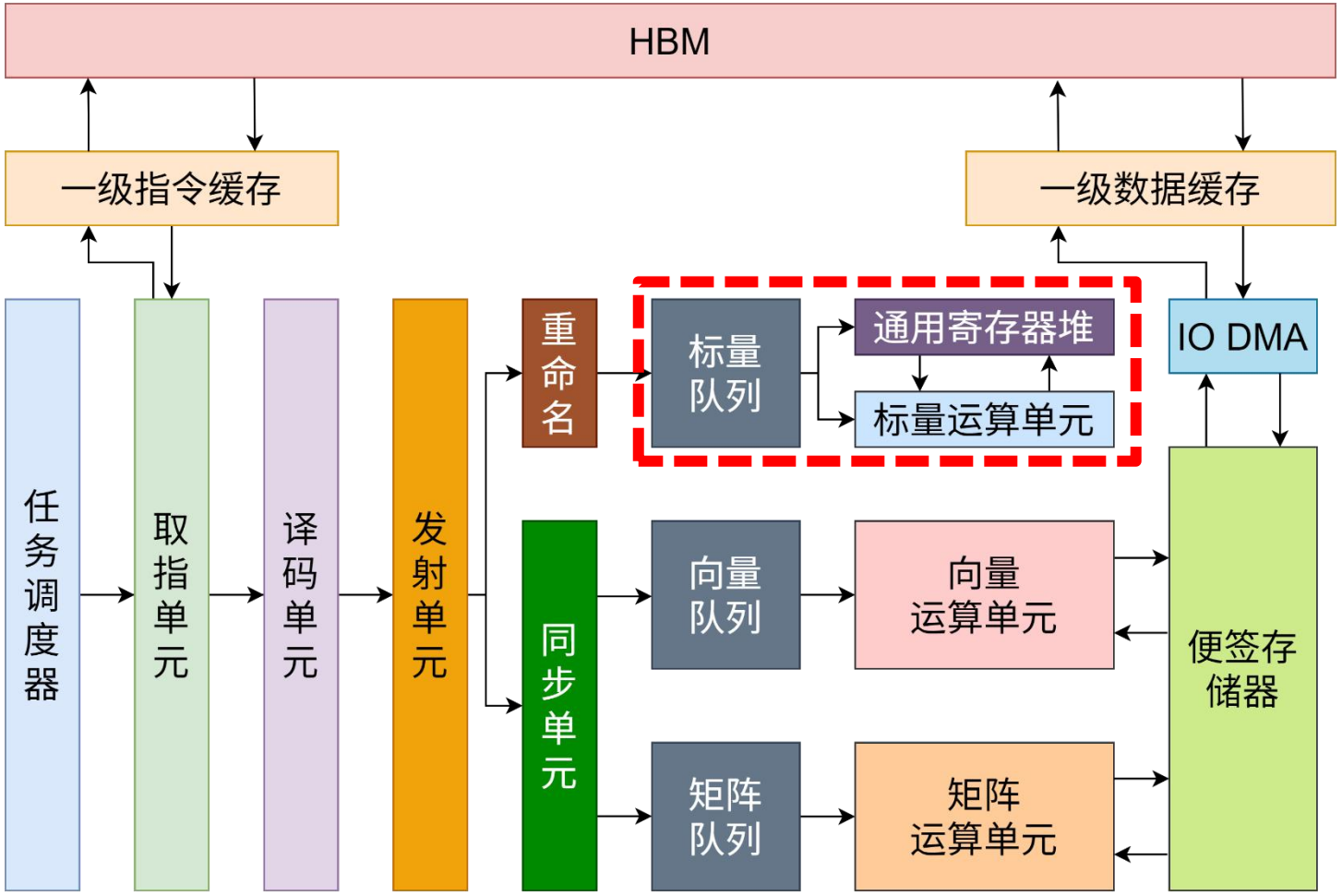
人工智能处理器总体架构

- **解依赖分发**



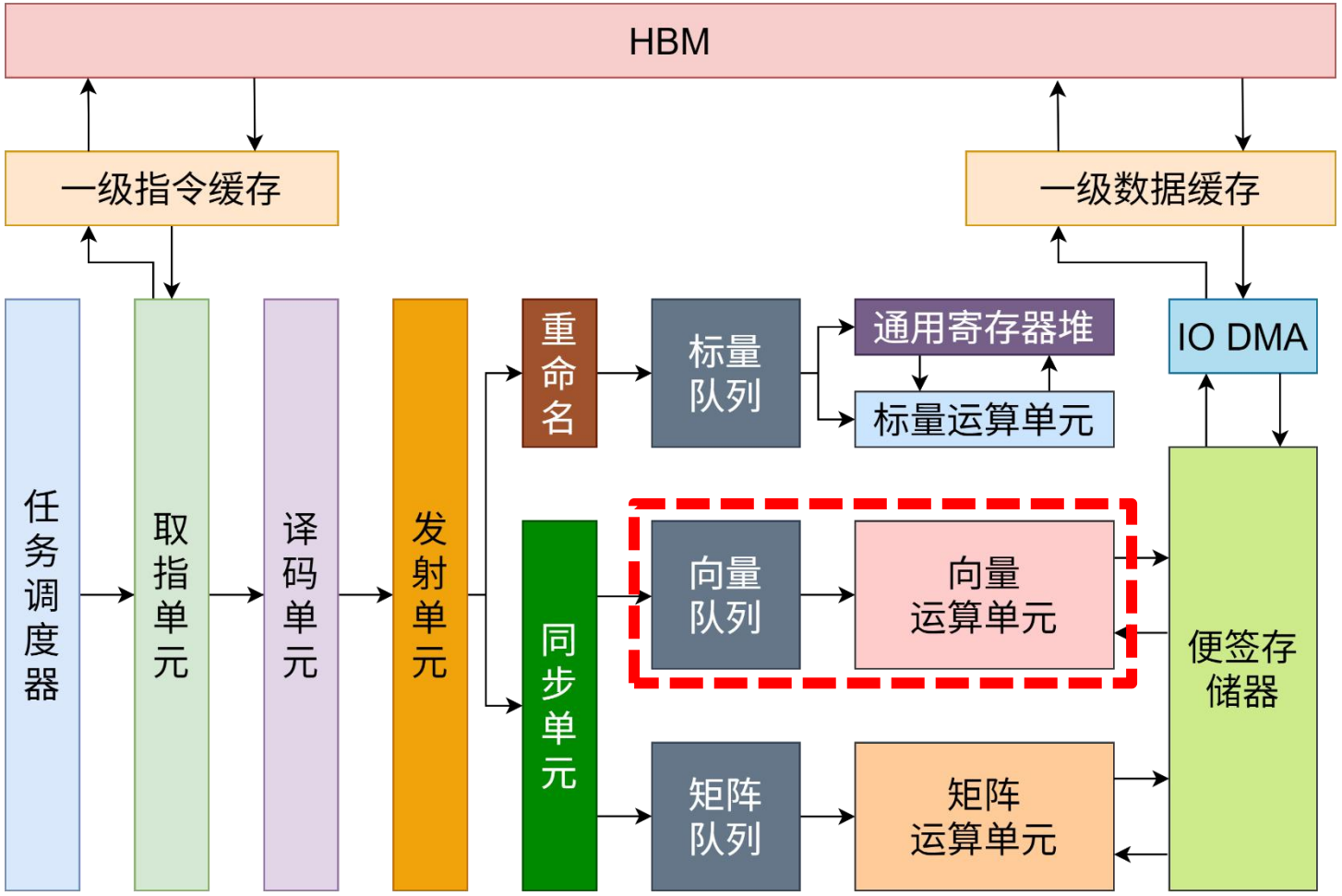
人工智能处理器总体架构

- **标量处理系统**



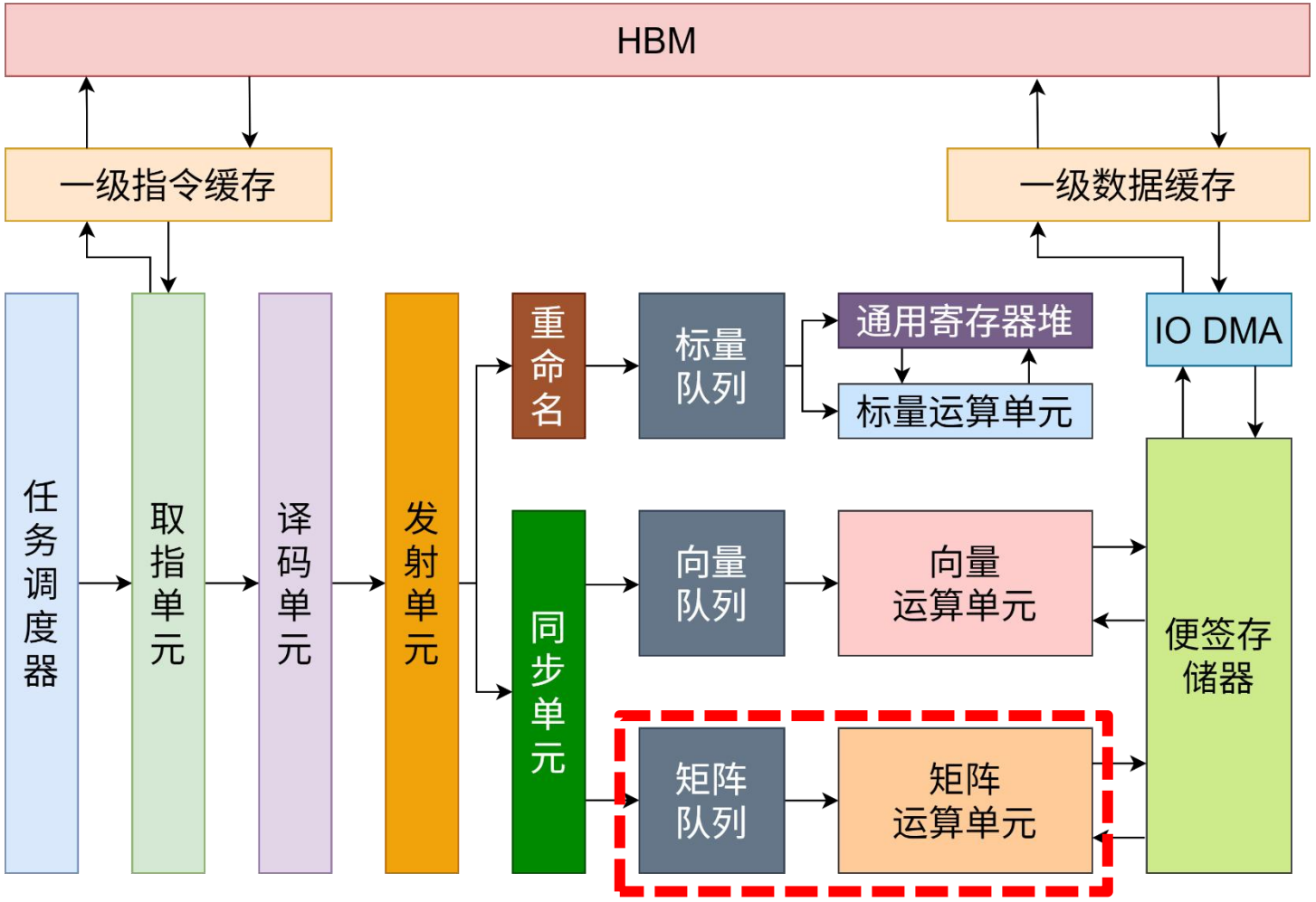
人工智能处理器总体架构

- **向量处理系统**



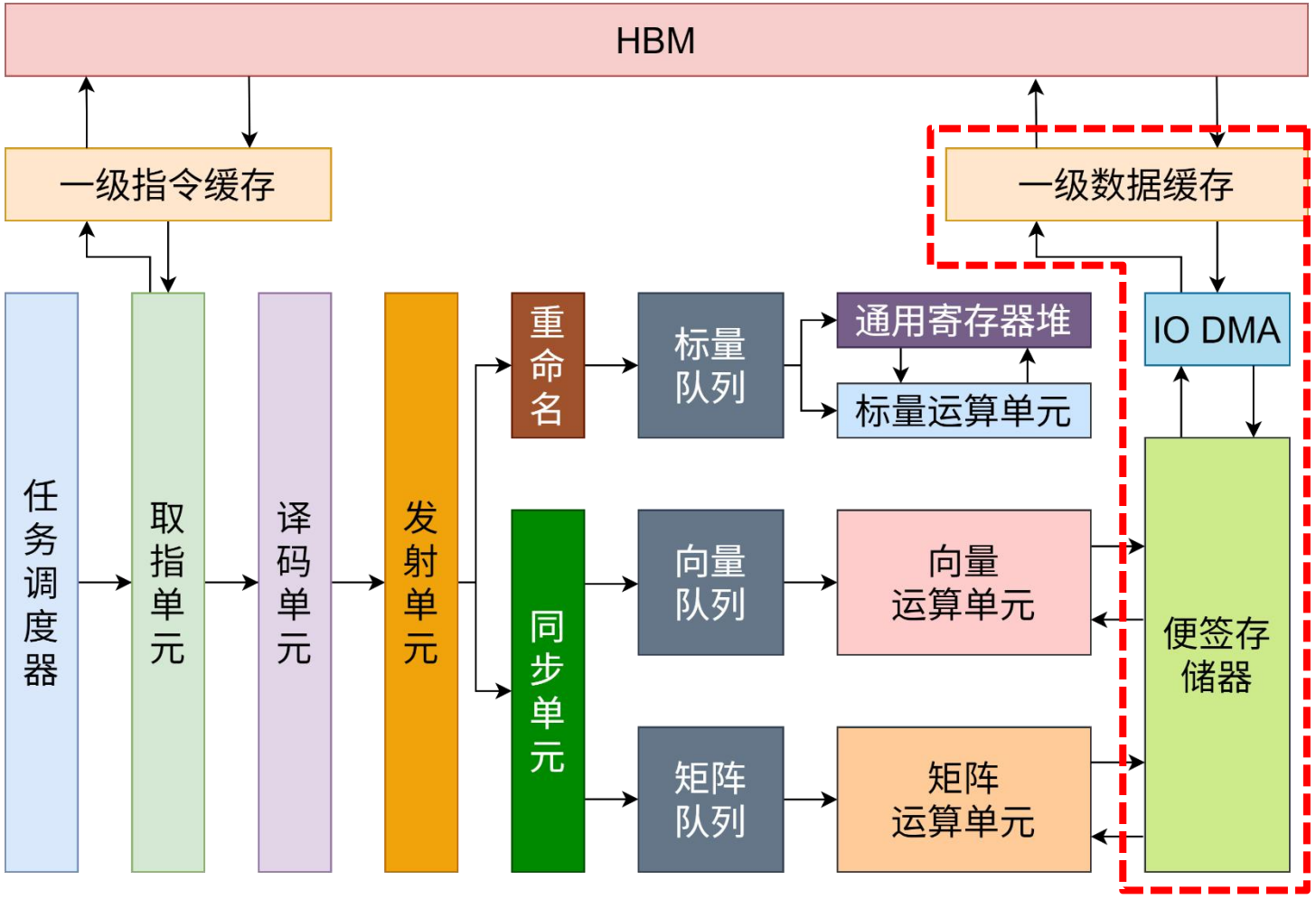
人工智能处理器总体架构

- **矩阵处理系统**



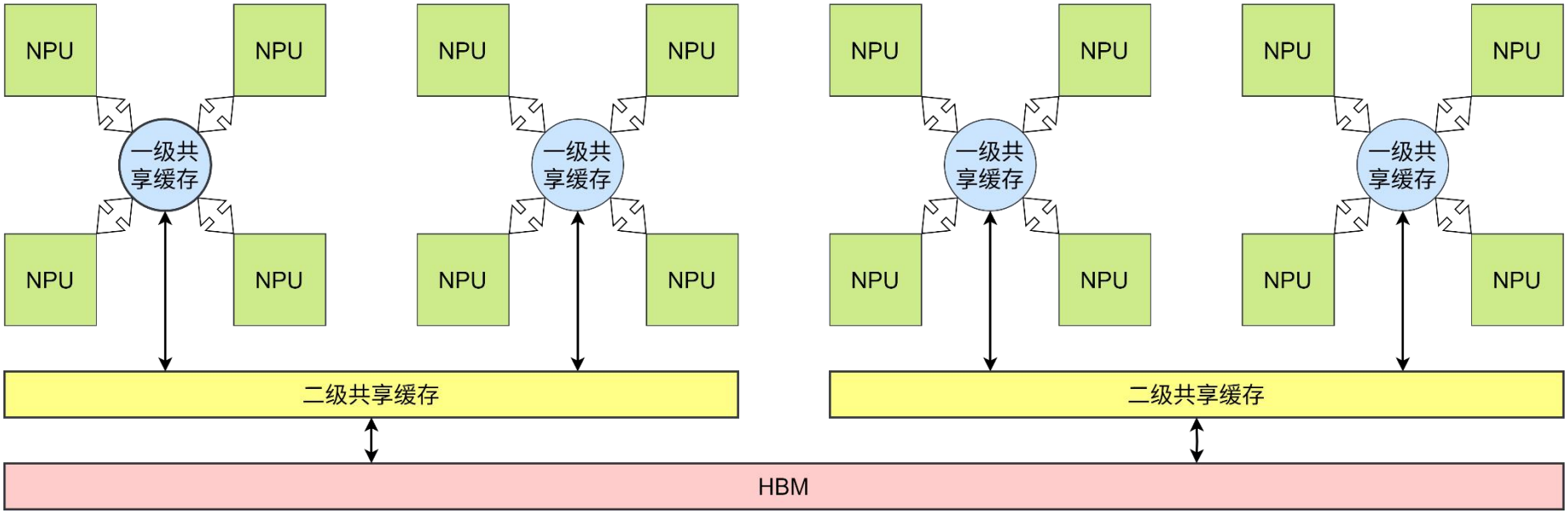
人工智能处理器总体架构

- **存储系统**



人工智能处理器总体架构

- **多核架构**
  - 多级集成
  - 提升资源利用率和灵活度
  - 降低EDA优化复杂度



Part2：数据级并行

- **DLP技术基础：研究动机及种类**
  - 从效率瓶颈到DLP解决方案
  - DLP核心定义与并行范式对比
- **向量处理机：DLP的基础形态**
  - 向量处理器硬件结构
  - 向量指令集与代码示例
  - 向量处理器性能计算与优化
- **SIMD指令集扩展：DLP的轻量化落地**
- **GPU：DLP的高性能实战形态**
  - GPU硬件架构与SIMT模型
  - GPU编程模型与AI任务适配
- **AI专用处理器：DLP的AI场景定制形态**
  - AI专用处理器的发展历程：从学术探索到产业爆发
  - 典型AI专用处理器架构与DLP实现
- **四类DLP技术综合对比**

并行范式对比

并行范式	核心逻辑	硬件支撑	适用场景	演进阶段
SISD	单指令 单数据	分支密集型任务	标量 CPU	基础计算时代
ILP	多指令 并行执行	超标量 CPU	指令级独立的串行任务	通用计算优化期
DLP (通用)	单指令 多数据并行	向量机 / SIMD/GPU	科学计算 多媒体	并行计算发展期
DLP (AI 专用)	张量级并行 (SIMD 扩展)	TPU / 寒武纪 / 昇腾	深度学习 (卷积、矩阵乘)	智能计算爆发期

四类DLP技术对比

对比维度	向量处理机	SIMD指令集扩展	GPU	AI专用处理器
并行粒度	长向量	短向量	线程级	张量级
硬件成本	高	低	较高	高
计算密度	中	低	高	极高
适用场景	科学计算	多媒体处理	AI+通用计算	全场景AI
发展脉络	1970s-1990s	1990至今 (CPU集成)	2010至今 (AI主力)	2016至今 (爆发增长)
技术传承	奠定 “数据并行”基础	实现 “轻量级并行”落地	开创 “Thread级并行”范式	升级 “张量级专用并行”

Acknowledgements

- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - David Patterson (UCB)
  - Chenxi Zhang (Tongji)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**