计算机体系结构

周学海

xhzhou@ustc.edu.cn

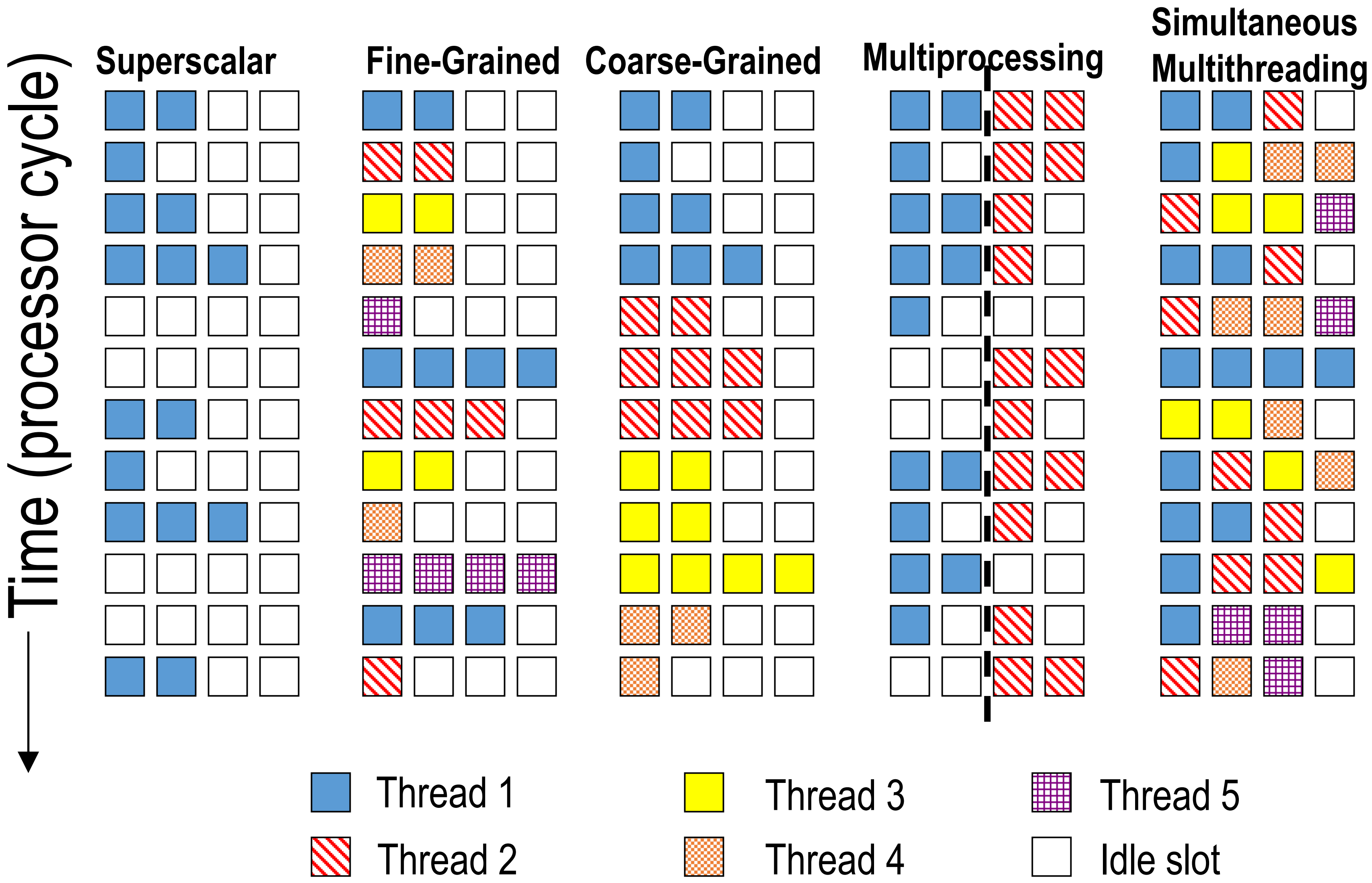
0551-63601556, 63492271

中国科学技术大学

第6章 Data-Level Parallelism in Vector, SIMD, and GPU Architectures

- SIMD的变体
- 向量体系结构
- GPU

review: Multithreading



传统指令级并行技术的问题

挖掘ILP的传统方法的主要缺陷:

- *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。

Introduction

- 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 Parallelism

- 向量体系结构
- SIMD 扩展
- Graphics Processor Units (GPUs)

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

Supercomputers

- Definition of a supercomputer:
    - 对于给定任务而言世界上最快的机器
    - 将计算密集型转化为I/O密集型的设备
    - 任何造价超过3千万美元的机器
- 由Seymour Cray设计的机器
- CDC6600 (Cray, 1964) 被认为是第一台超级计算机`

CDC 6600 *Seymour Cray, 1963*

- A fast pipelined machine with 60-bit words
  - 128 Kword main memory capacity, 32 banks
- Ten functional units (parallel, unpipelined)
  - Floating Point: adder, 2 multipliers, divider
  - Integer: adder, 2 incrementers, ...
- Hardwired control (no microcoding)
- *Scoreboard* for dynamic scheduling of instructions
- Ten Peripheral Processors for Input/Output
  - a fast multi-threaded 12-bit integer ALU
- Very fast clock, 10 MHz (FP add in 4 clocks)
- >400,000 transistors, 750 sq. ft., 5 tons, 150 kW, novel freon-based technology for cooling
- Fastest machine in world for 5 years (until 7600)
  - over 100 sold ($7-10M each)

IBM Memo on CDC6600

Thomas Watson Jr., IBM CEO, August 1963:

> *“Last week, Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer.”*

To which Cray replied: *“It seems like Mr. Watson has answered his own question.”*

Supercomputer Applications

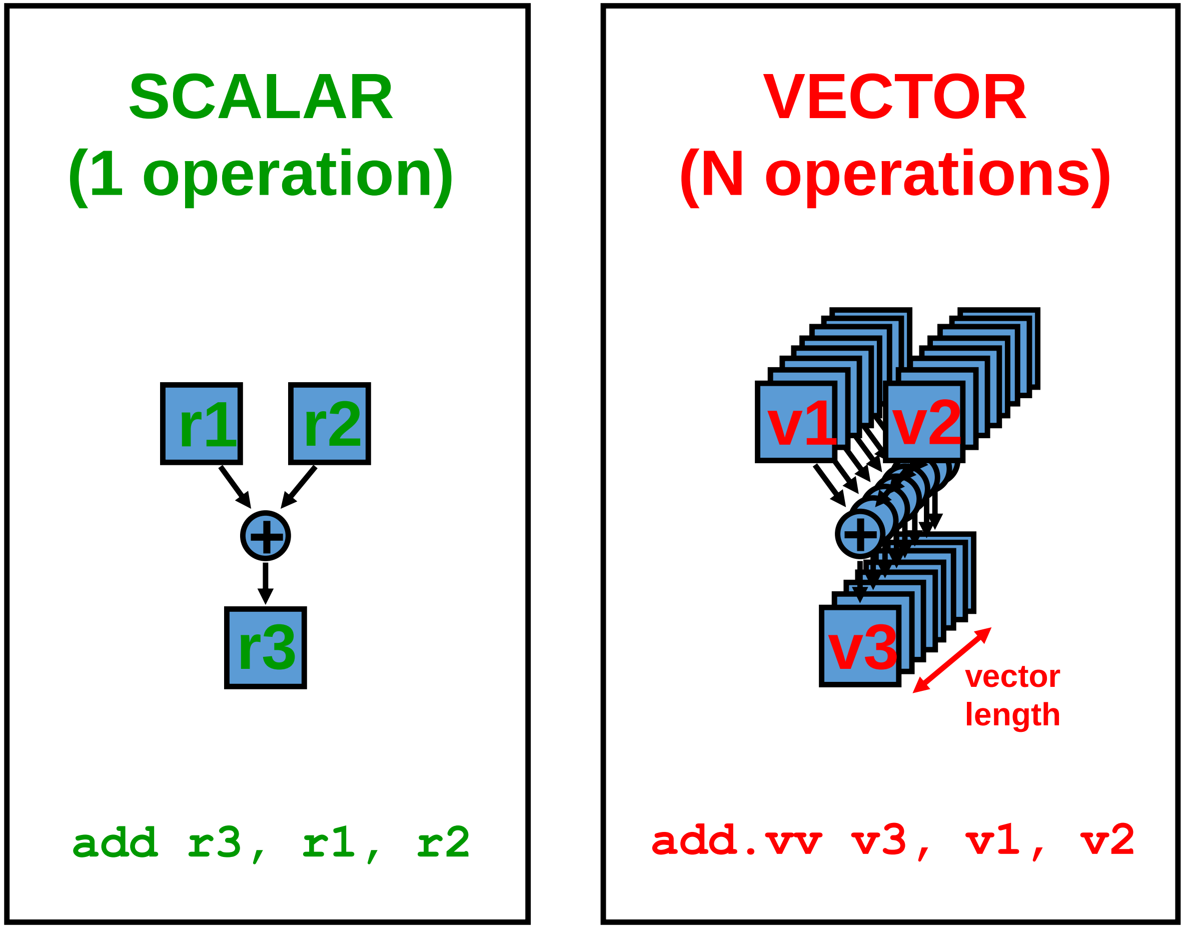
- 典型应用领域
  - 军事研究领域（核武器研制、密码学）
  - 科学研究
  - 天气预报
  - 石油勘探
  - 工业设计 (car crash simulation)
  - 生物信息学
  - 密码学

- 均涉及大量的数据集处理

- 70-80年代Supercomputer = Vector Machine

Alternative Model:Vector Processing

- 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）



SCALAR
(1 operation)

```
add r3, r1, r2
```

VECTOR
(N operations)

```
add.vv v3, v1, v2
```

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 64 elements
  - 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

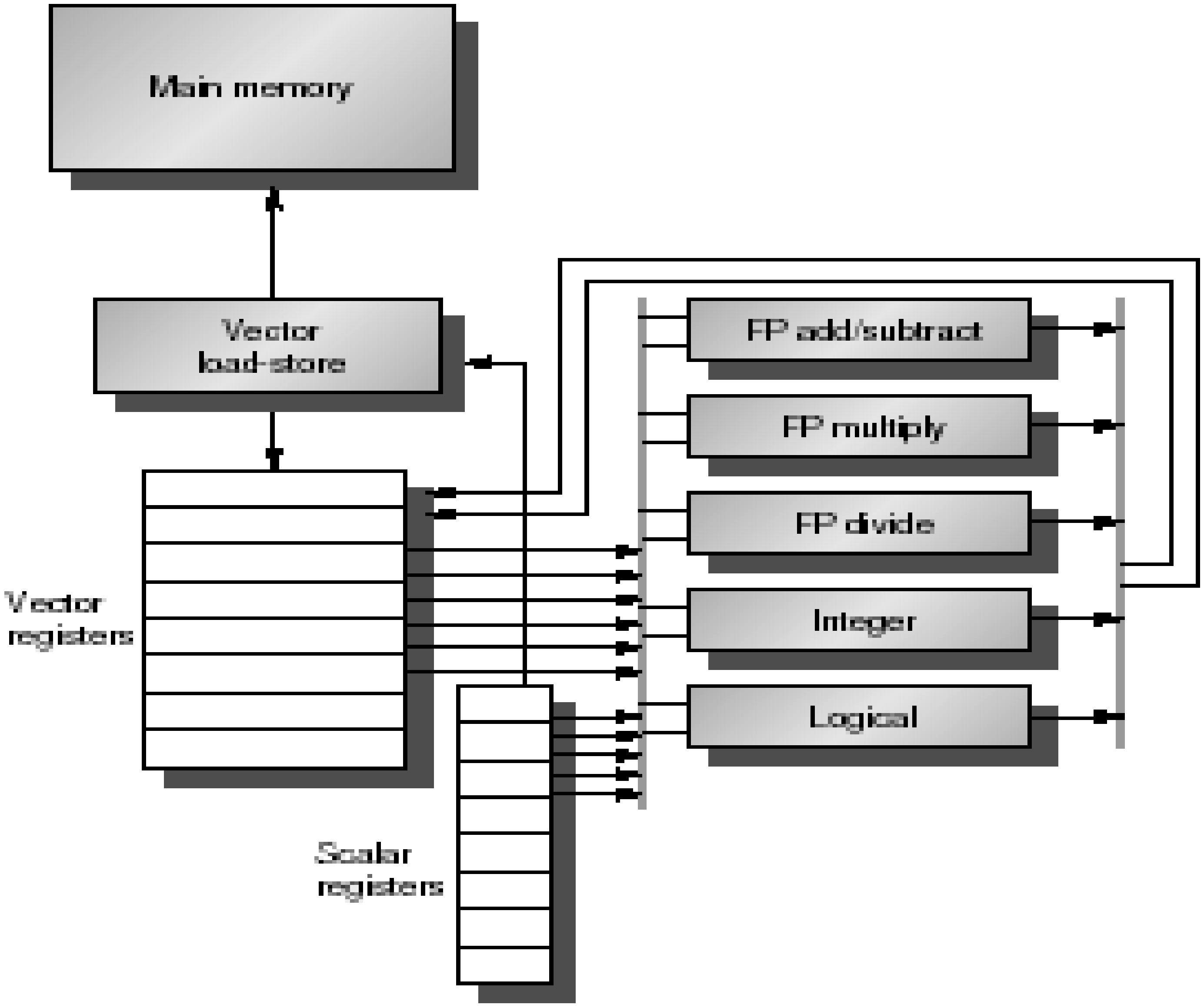
向量处理机的基本结构

- *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

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
  - 典型的有8-32 向量寄存器，每个寄存器存放64到128个64位的元素
- *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 to connect) FUs , LSUs, registers


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

Vector Supercomputers

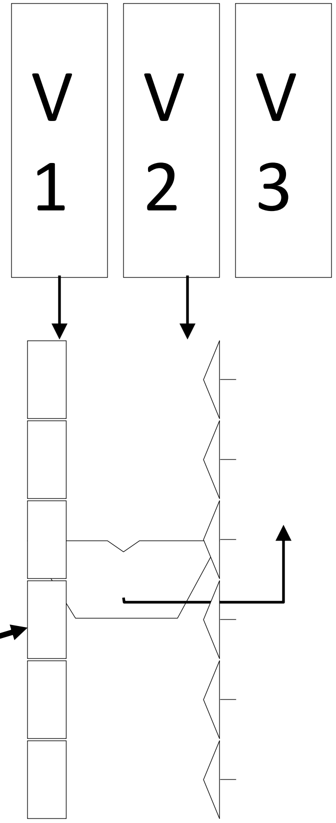
- Cray-1的变体（1976）：
- Scalar Unit： Load/Store Architecture
- Vector Extension
  - Vector Registers
  - Vector Instructions
- Implementation
  - 硬布线逻辑控制
  - 高效流水化的功能部件
  - 多体交叉存储系统
  - 无Data Cache
  - 不支持 Virtual Memory

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)
  - 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 Arithmetic Execution

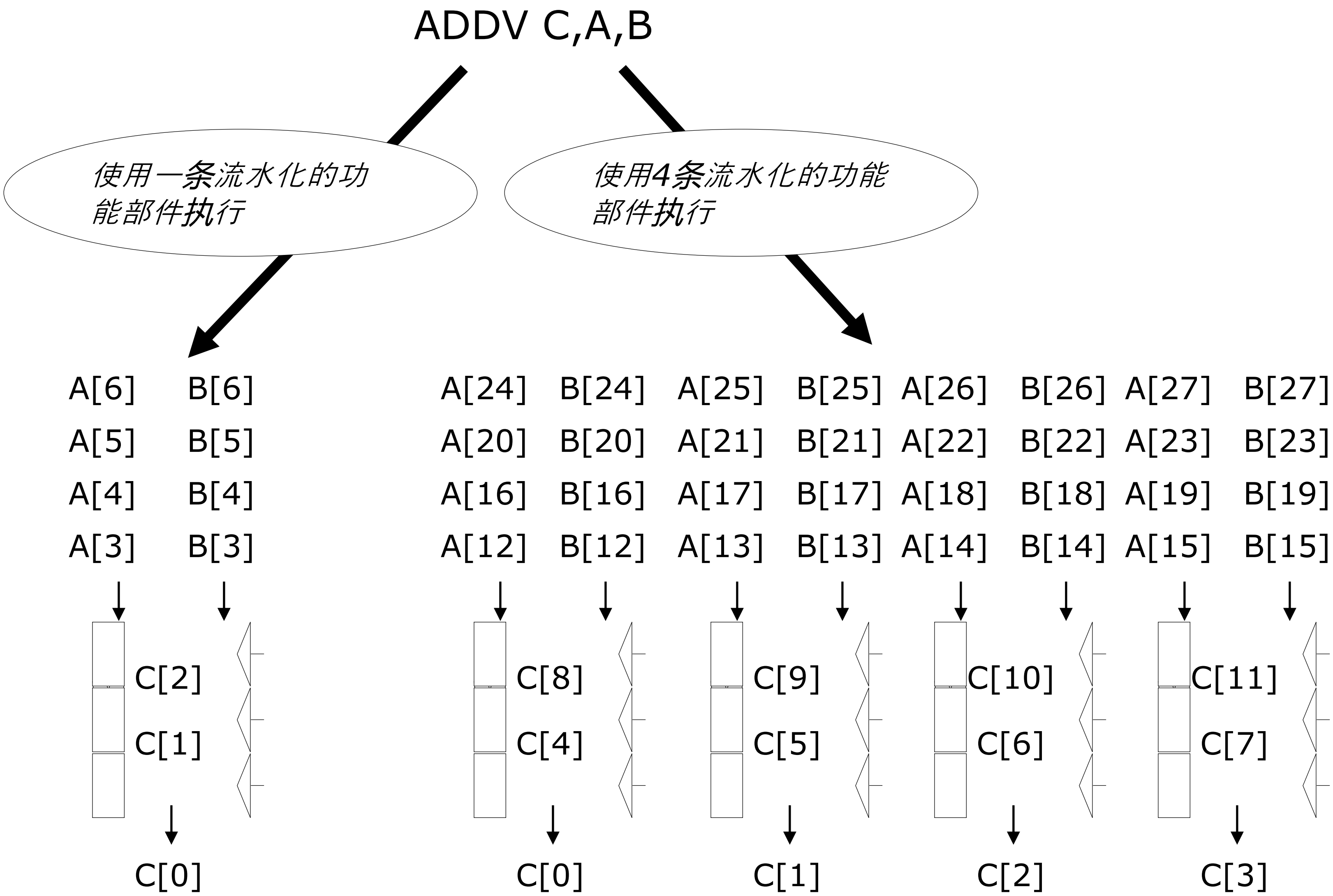
- 使用较深的流水线(=> fast clock)执行向量元素的操作
- 由于向量元素相互独立，简化了的深度流水线的控制 (=> no hazards!)



Six stage multiply pipeline

V3 <- v1 * v2

Vector Instruction Execution



ADDV C,A,B

使用一条流水化的功能部件执行

使用4条流水化的功能部件执行

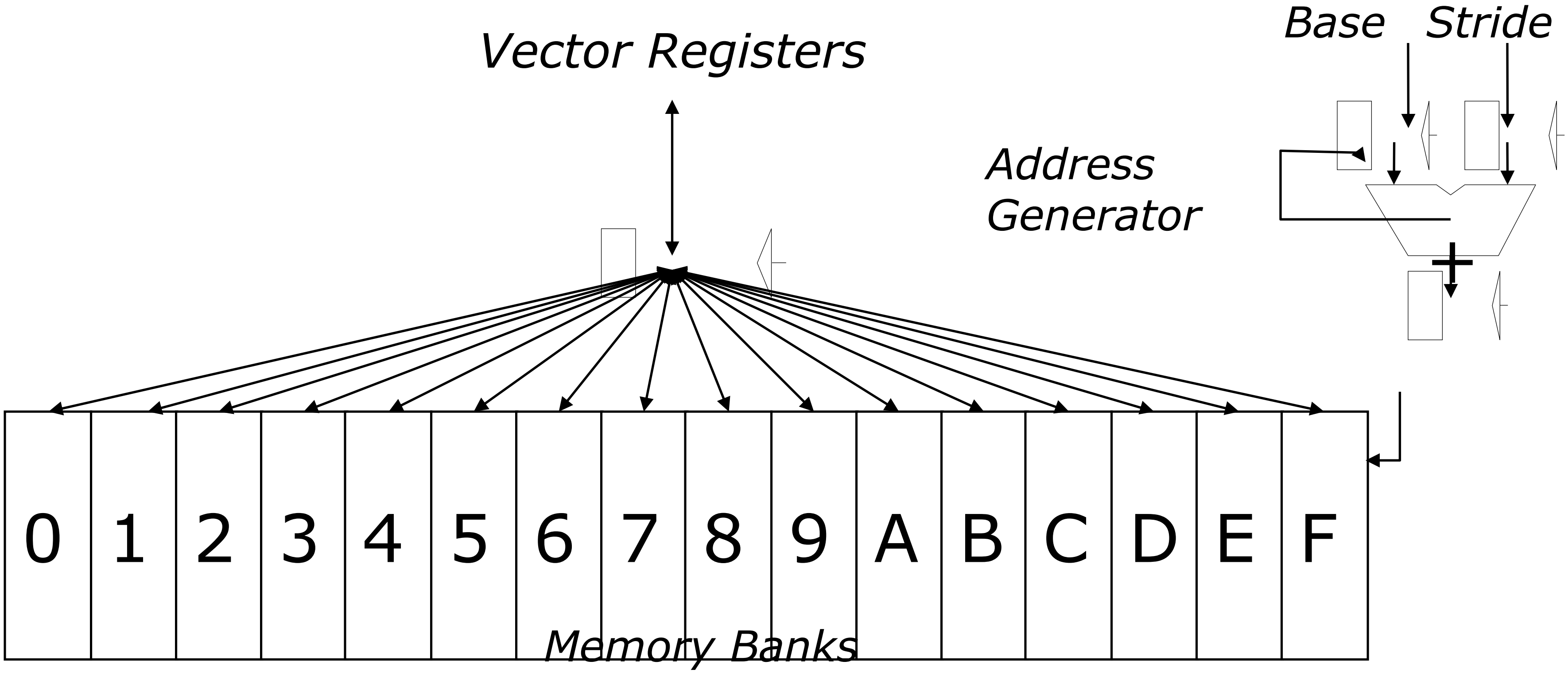
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]

A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]

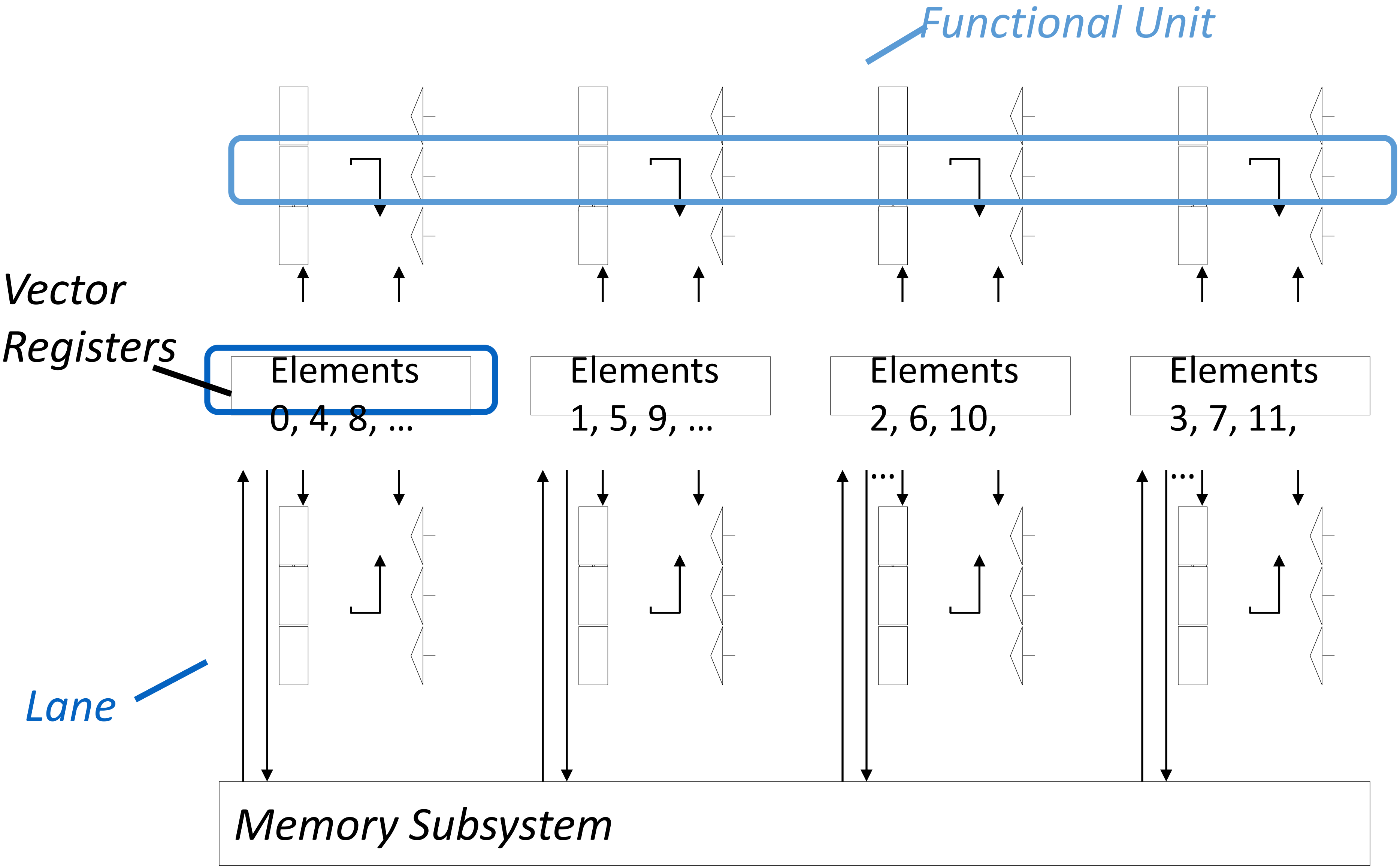
Interleaved Vector Memory System

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

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

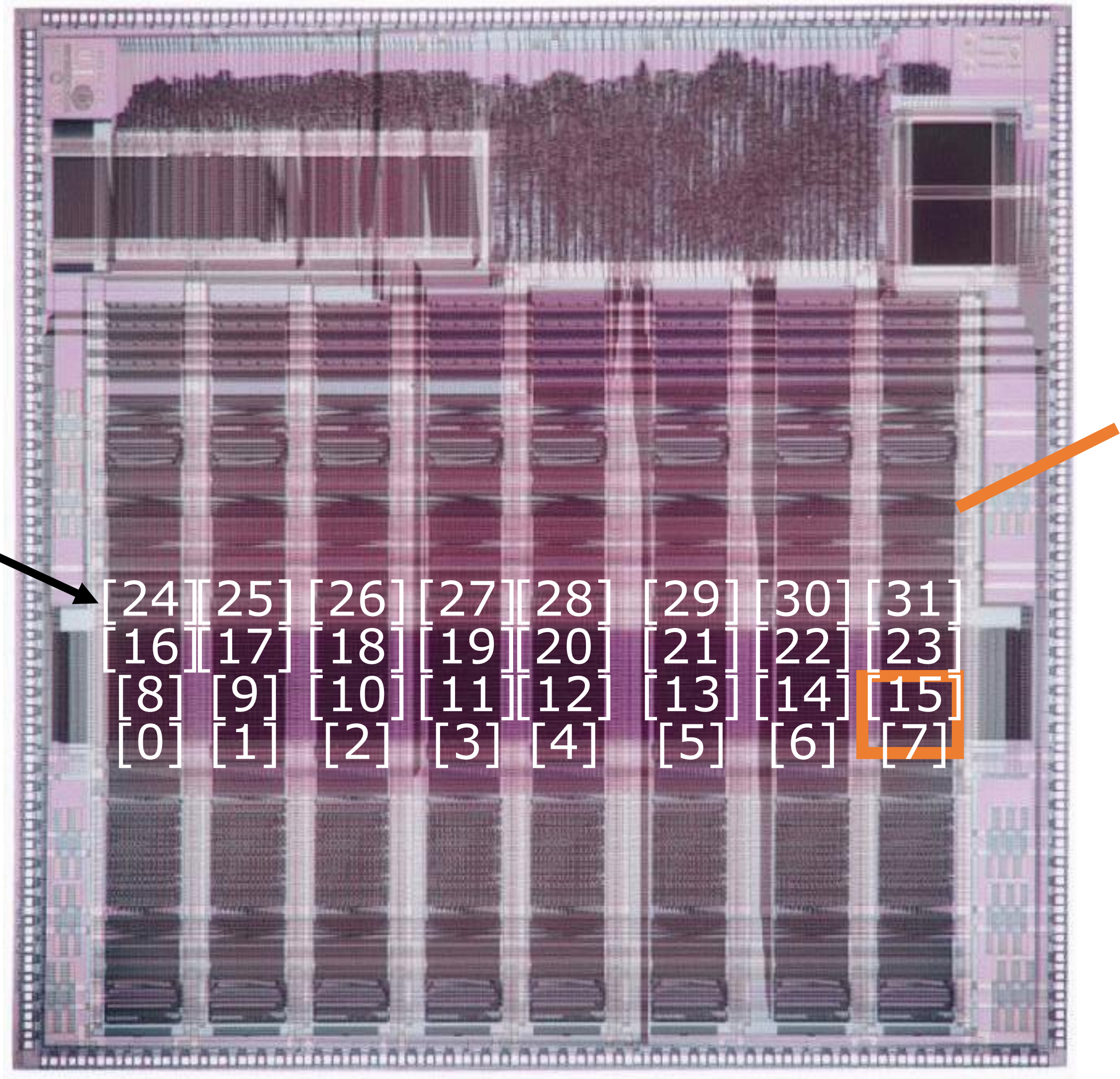


Vector Unit Structure



T0 Vector Microprocessor (UCB/ICSI, 1995)

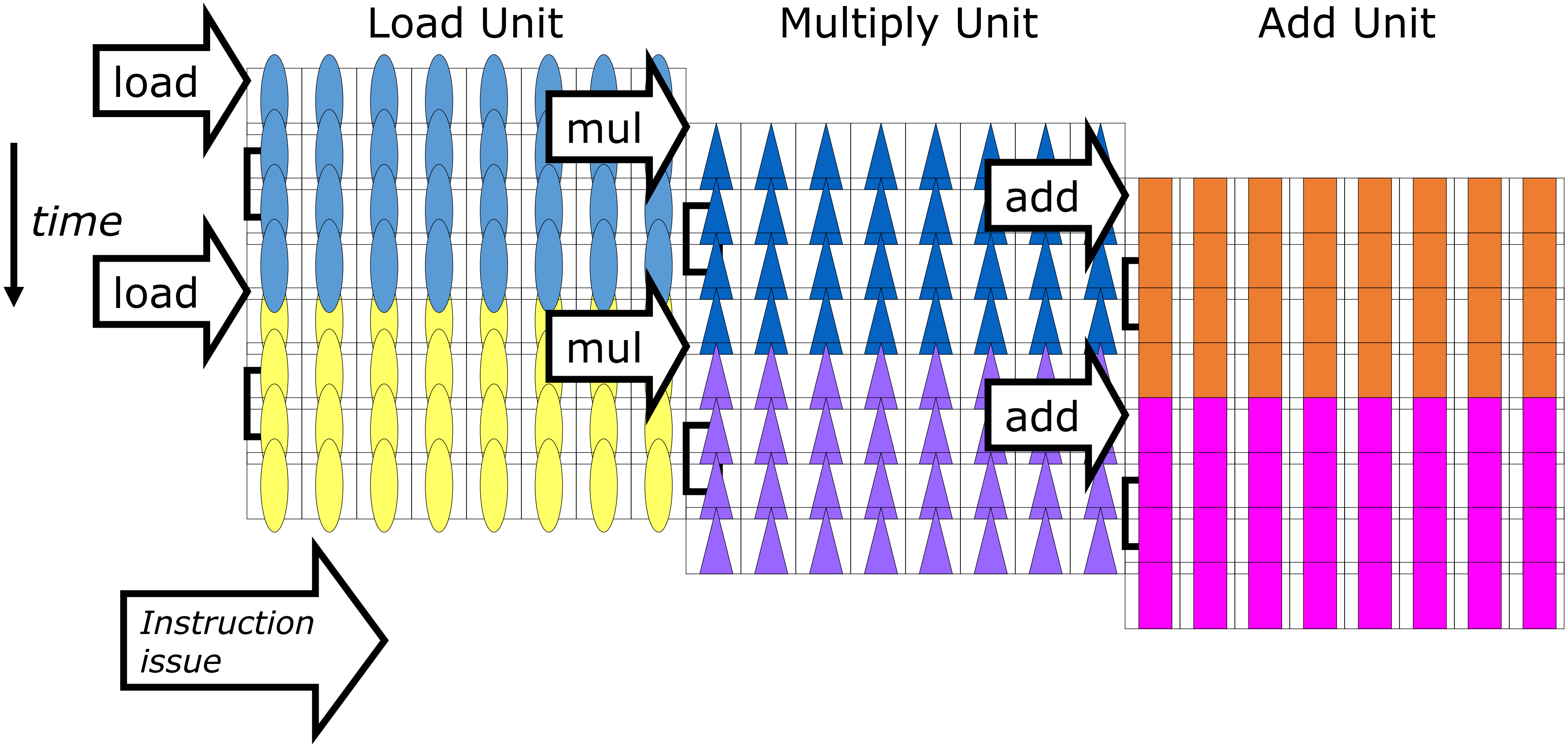
Vector register elements striped over lanes



Lane

Vector Instruction Parallelism

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



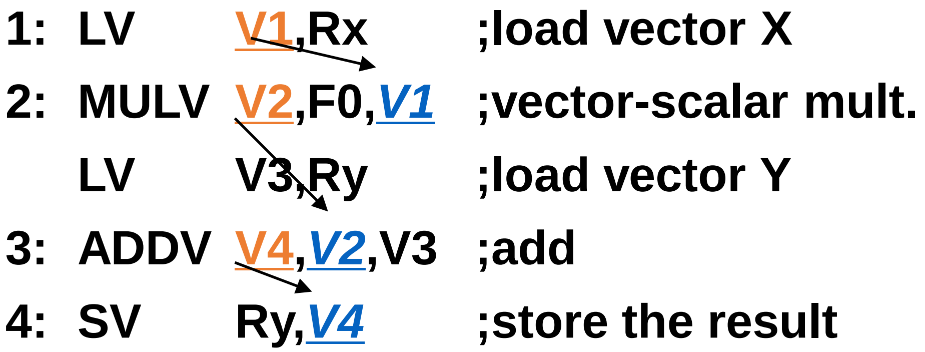
Complete 24 operations/cycle while issuing 1 short instruction/cycle

“DLXV” Vector Instructions

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

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 (ignores overhead; good approximization for long vectors)

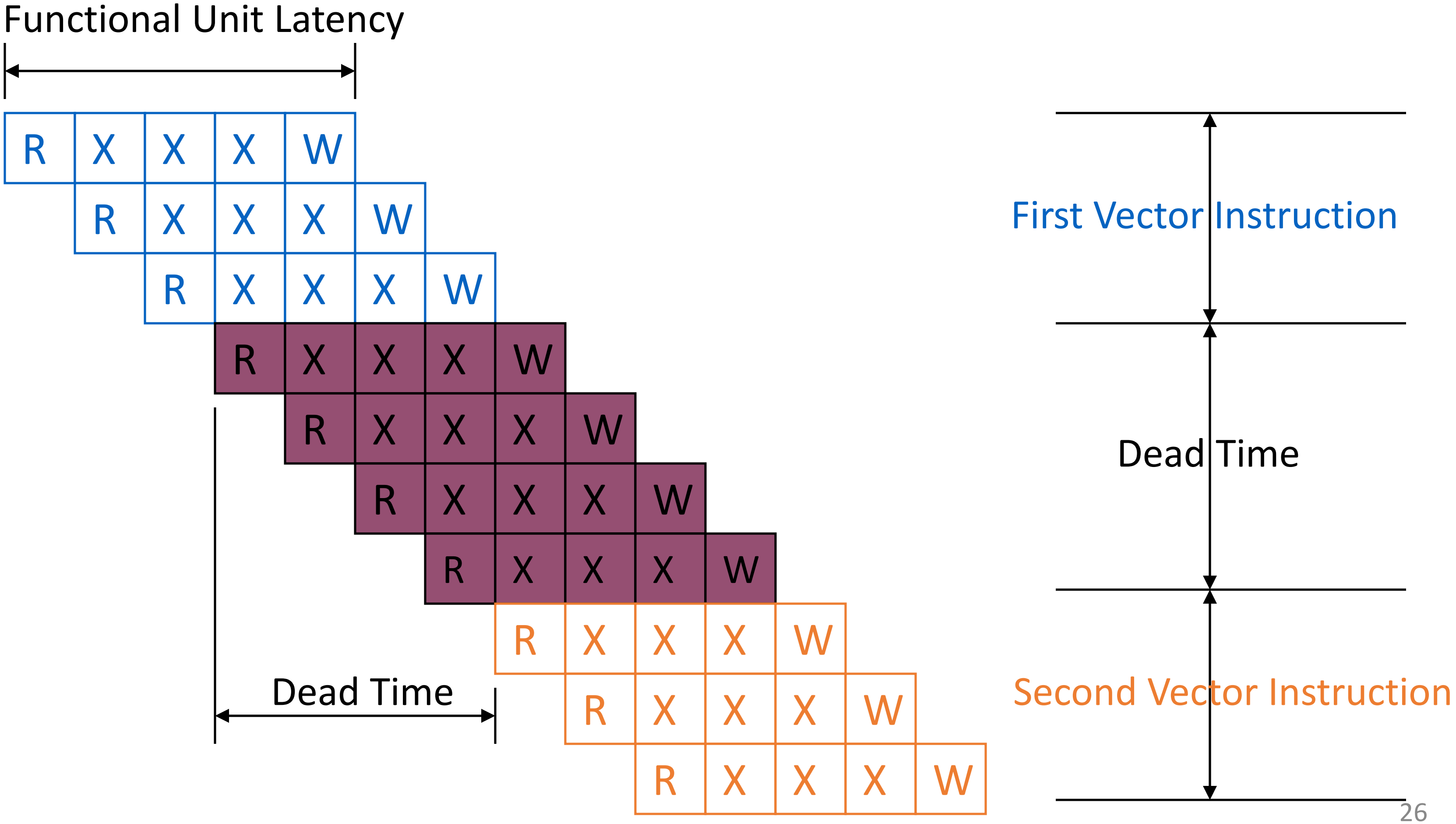


```
1:  LV      V1,Rx       ;load vector X
2:  MULV    V2,F0,V1    ;vector-scalar mult.
    LV      V3,Ry       ;load vector Y
3:  ADDV    V4,V2,V3    ;add
4:  SV      Ry,V4       ;store the result
```

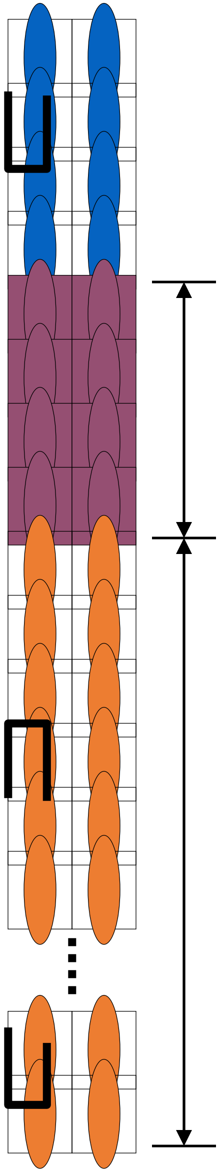
4 convoys, 1 lane, VL=64
=> 4 x 64 = 256 clocks
(or 4 clocks per result)

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
  - 截止时间或回复时间（dead time or recovery time）：运行下一条向量指令的间隔时间



Dead Time and Short Vectors



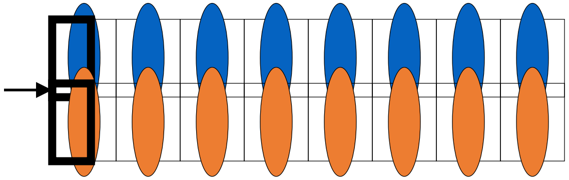
4 cycles dead time

64 cycles active

Cray C90, Two lanes

4 cycle dead time

Maximum efficiency 94% with 128 element vectors



No dead time

T0, Eight lanes

No dead time

100% efficiency with 8 element vectors

DLXV 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***

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:

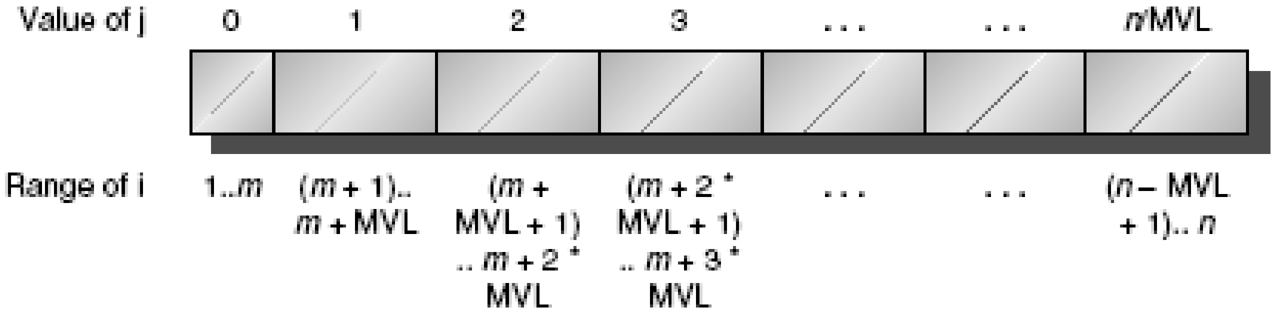
```
        do 10 i = 1, n
10            Y(i) = a * X(i) + Y(i)
```

- 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
```

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

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

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

Tstart = 12 + 7 + 12 = 31

T200 = 660+4*31 = 784

每一元素的执行时间 = 784/200 = 3.9

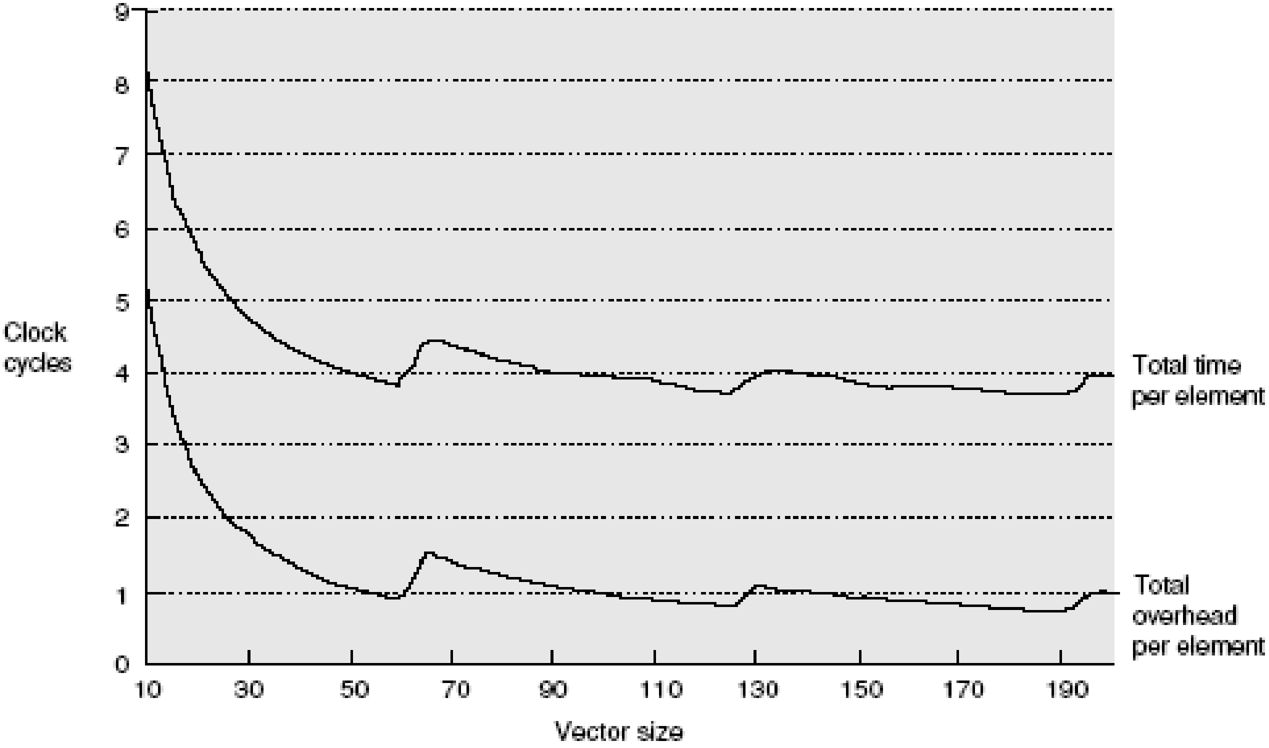


Figure G.9 The total execution time per element and the total overhead time per element versus the vector length for the example on page G-19. For short vectors the total start-up time is more than one-half of the total time, while for long vectors it reduces to about one-third of the total time. The 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

Vector Stride

- 假设处理顺序相邻的元素在存储器中不是顺序存储。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
  - Non-unit (constant) stride (常数步长)
  - Indexed (gather-scatter) (间接寻址)
    - 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

DAXPY (Y = a * X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector

```
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
```

```
        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
```

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

Example Vector Machines

Machine	Year	Clock	Regs	Elements	FUs	LSUs
Cray 1	1976	80 MHz	8	64	6	1
Cray XMP	1983	120 MHz	8	64	8	2 L, 1 S
Cray YMP	1988	166 MHz	8	64	8	2 L, 1 S
Cray C-90	1991	240 MHz	8	128	8	4
Cray T-90	1996	455 MHz	8	128	8	4
Conv. C-1	1984	10 MHz	8	128	4	1
Conv. C-4	1994	133 MHz	16	128	3	1
Fuj. VP200	1982	133 MHz	8-256	32-1024	3	2
Fuj. VP300	1996	100 MHz	8-256	32-1024	3	2
NEC SX/2	1984	160 MHz	8+8K	256+var	16	8
NEC SX/3	1995	400 MHz	8+8K	256+var	16	8

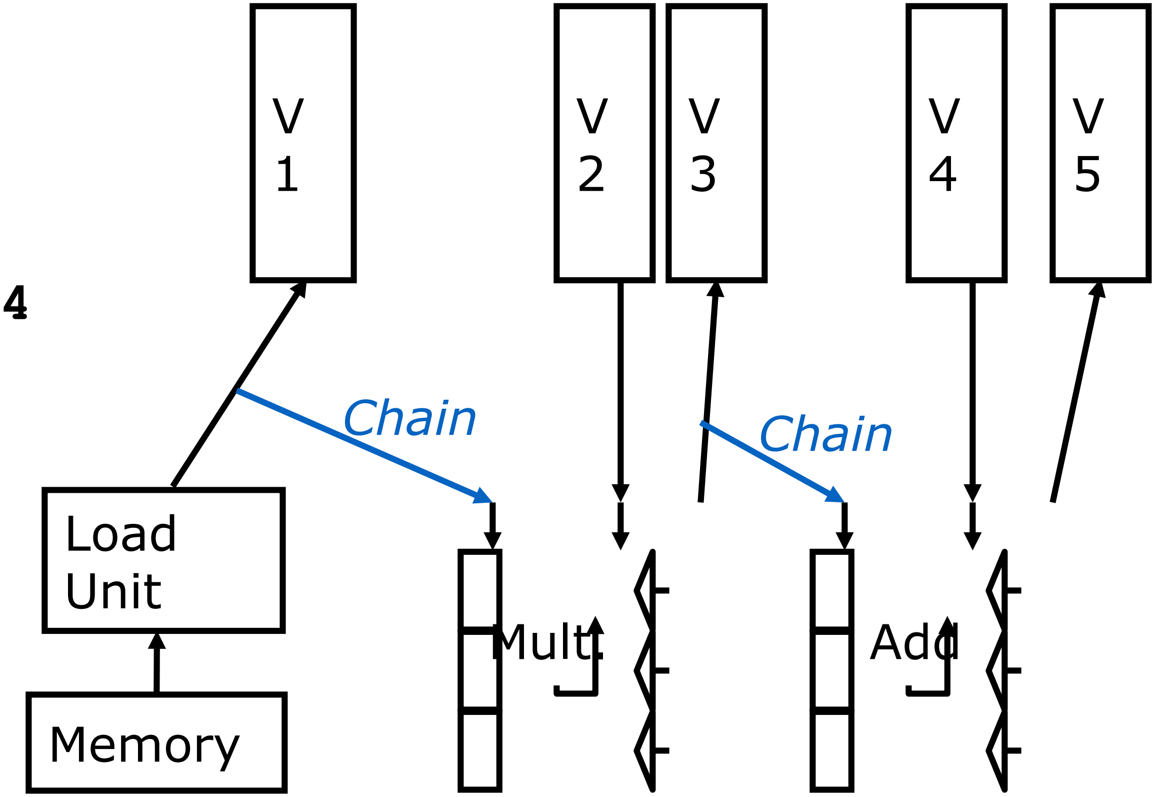
Vector Linpack Performance (MFLOPS)

			Matrix Inverse (gaussian elimination)		
Machine	Year	Clock	100x100	1kx1k	Peak(Procs)
• Cray 1	1976	80 MHz	12	110	160(1)
• Cray XMP	1983	120 MHz	121	218	940(4)
• Cray YMP	1988	166 MHz	150	307	2,667(8)
• Cray C-90	1991	240 MHz	387	902	15,238(16)
• Cray T-90	1996	455 MHz	705	1603	57,600(32)
• Conv. C-1	1984	10 MHz	3	--	20(1)
• Conv. C-4	1994	135 MHz	160	2531	3240(4)
• Fuj. VP200	1982	133 MHz	18	422	533(1)
• NEC SX/2	1984	166 MHz	43	885	1300(1)
• NEC SX/3	1995	400 MHz	368	2757	25,600(4)

Vector Opt#1: Vector Chaining

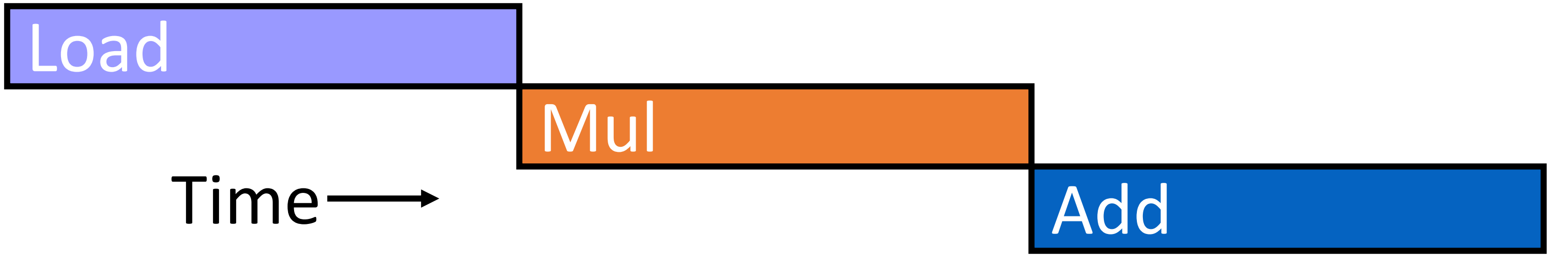
- 寄存器定向路径的向量机版本
- 首次在Cray-1上使用

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

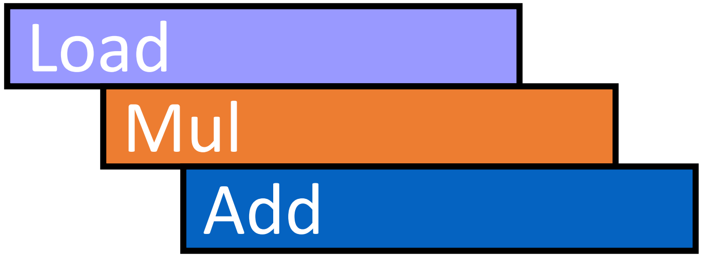


Vector Chaining Advantage

- 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 Opt #2: Conditional Execution

- Suppose:

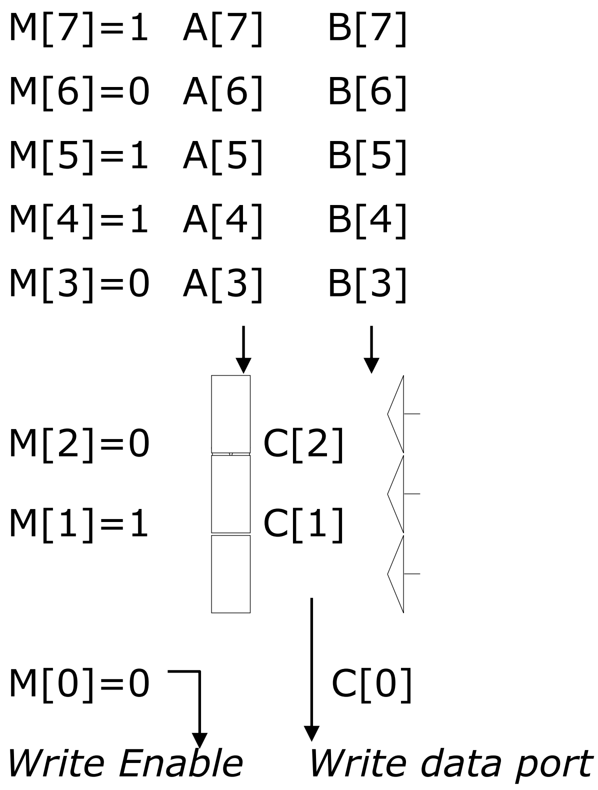
```
    do 100 i = 1, 64
        if (A(i) .ne. 0) then
            A(i) = A(i) – B(i)
        endif
100 continue
```

- *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

Masked Vector Instructions

Simple Implementation

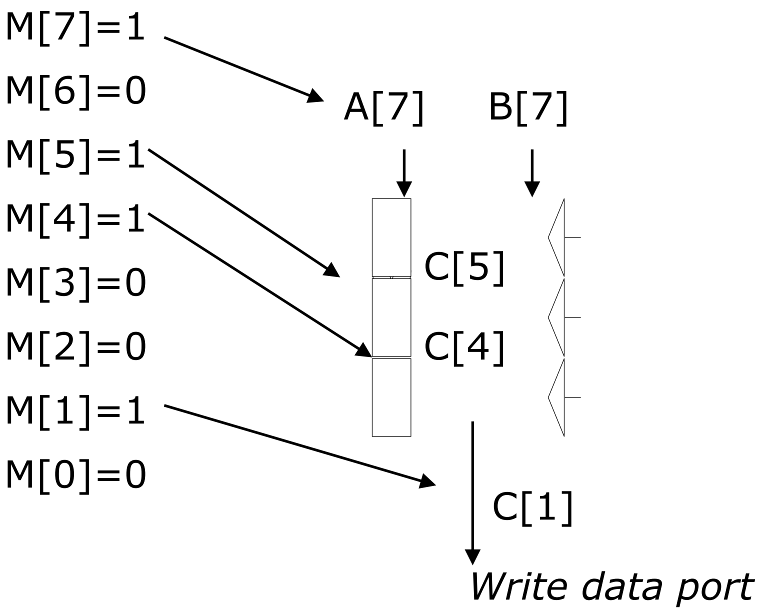
- execute all N operations, turn off result writeback according to mask



M[7]=1 A[7] B[7]
M[6]=0 A[6] B[6]
M[5]=1 A[5] B[5]
M[4]=1 A[4] B[4]
M[3]=0 A[3] B[3]
M[2]=0 C[2]
M[1]=1 C[1]
M[0]=0 C[0]
Write Enable *Write data port*

Density-Time Implementation

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



M[7]=1
M[6]=0
M[5]=1
M[4]=1
M[3]=0
M[2]=0
M[1]=1
M[0]=0
A[7] B[7]
C[5]
C[4]
C[1]
Write data port

```
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。

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
- 这些操作编译时可能无法完成。主要原因：编译器无法预知Ki以及是否有数据相关
- 使用CVI 设置步长（ index 0, 1xm, 2xm, ..., 63xm）

Sparse Matrix Example

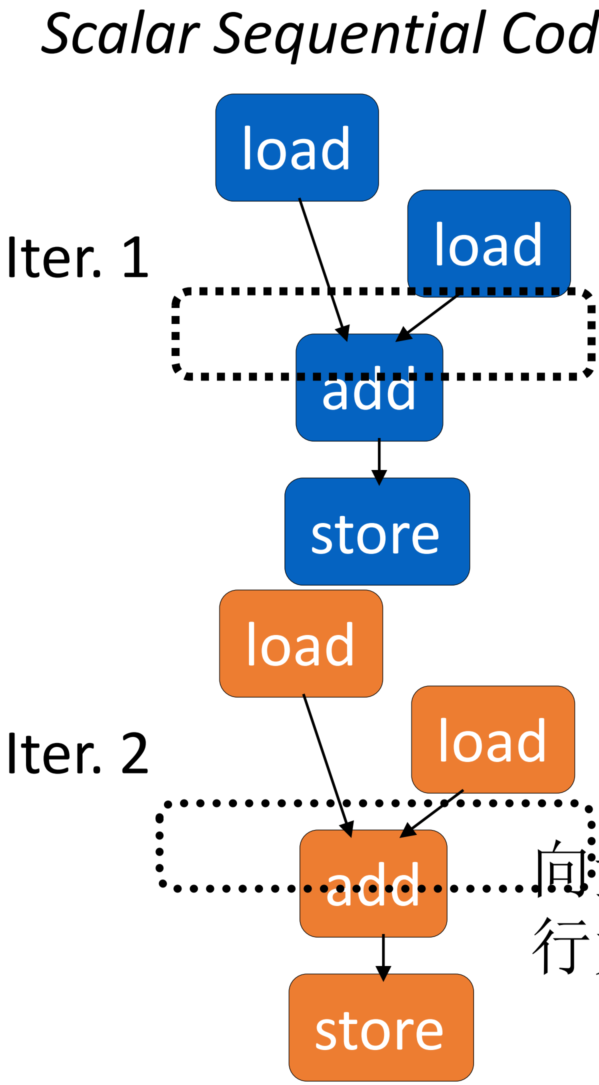
- Cache (1993) vs. Vector (1988)

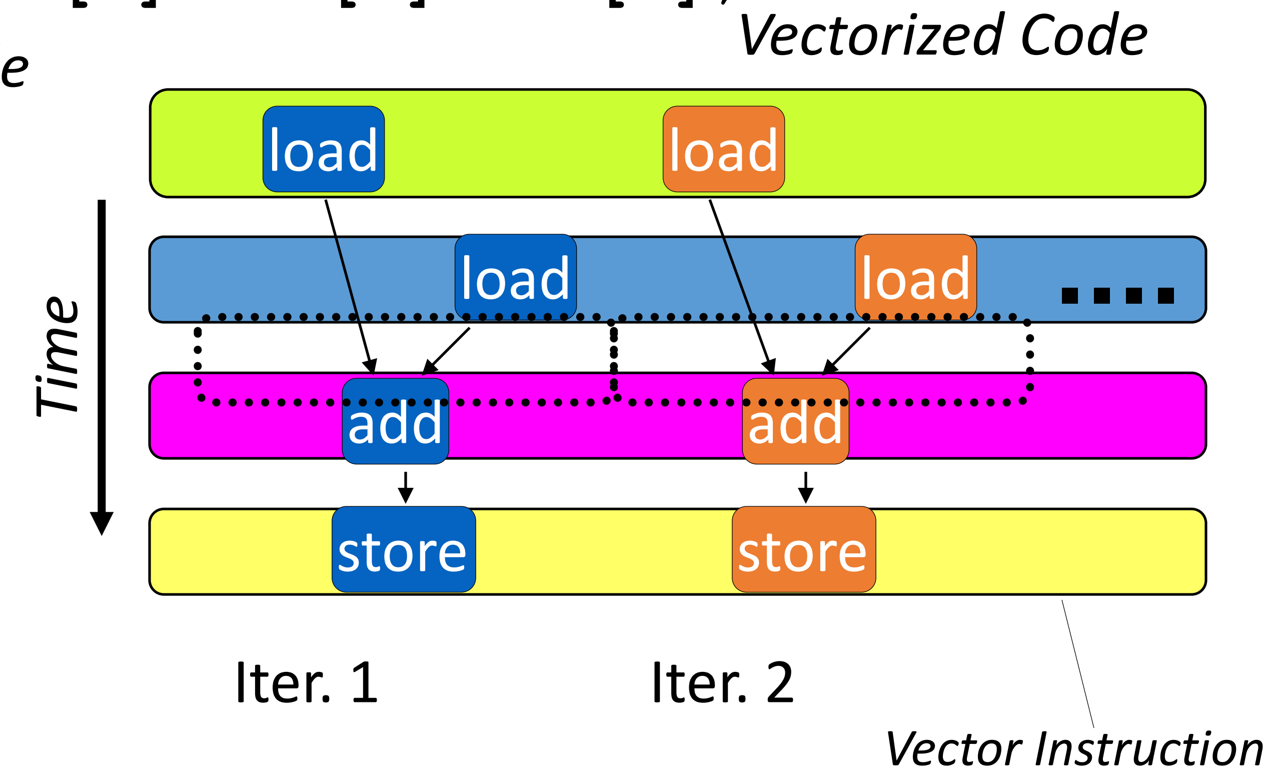
	IBM RS6000	Cray YMP
Clock	72 MHz	167 MHz
Cache	256 KB	0.25 KB
Linpack	140 MFLOPS	160 (1.1)
Sparse Matrix (Cholesky Blocked)	17 MFLOPS	125 (7.3)

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

Automatic Code Vectorization

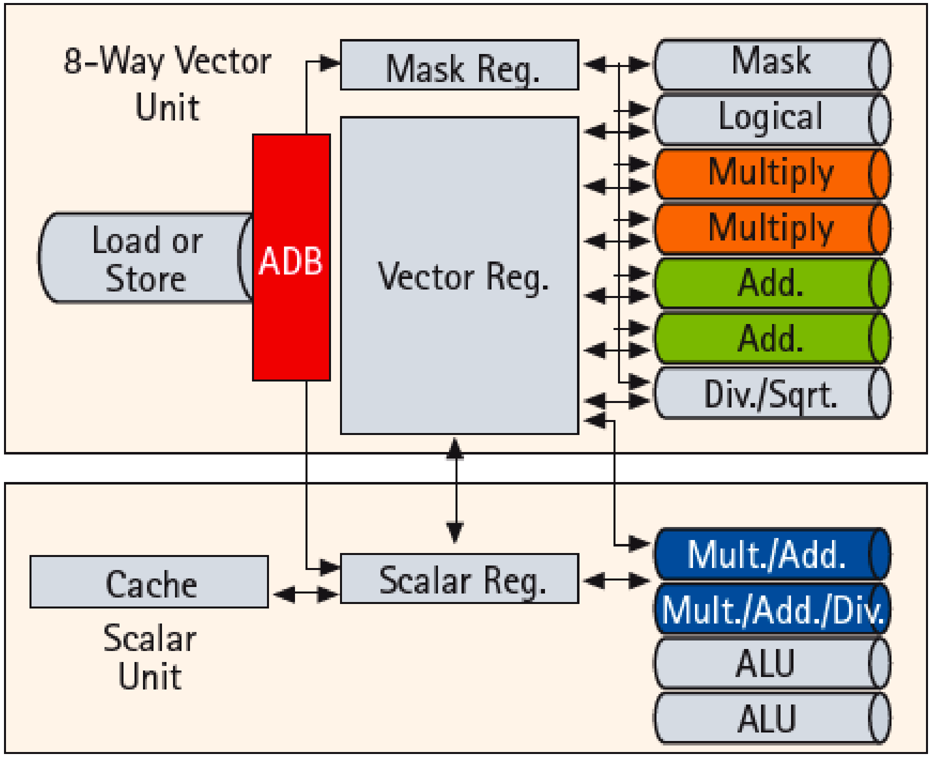
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```





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

A Modern Vector Super: NEC SX-9 (2008)

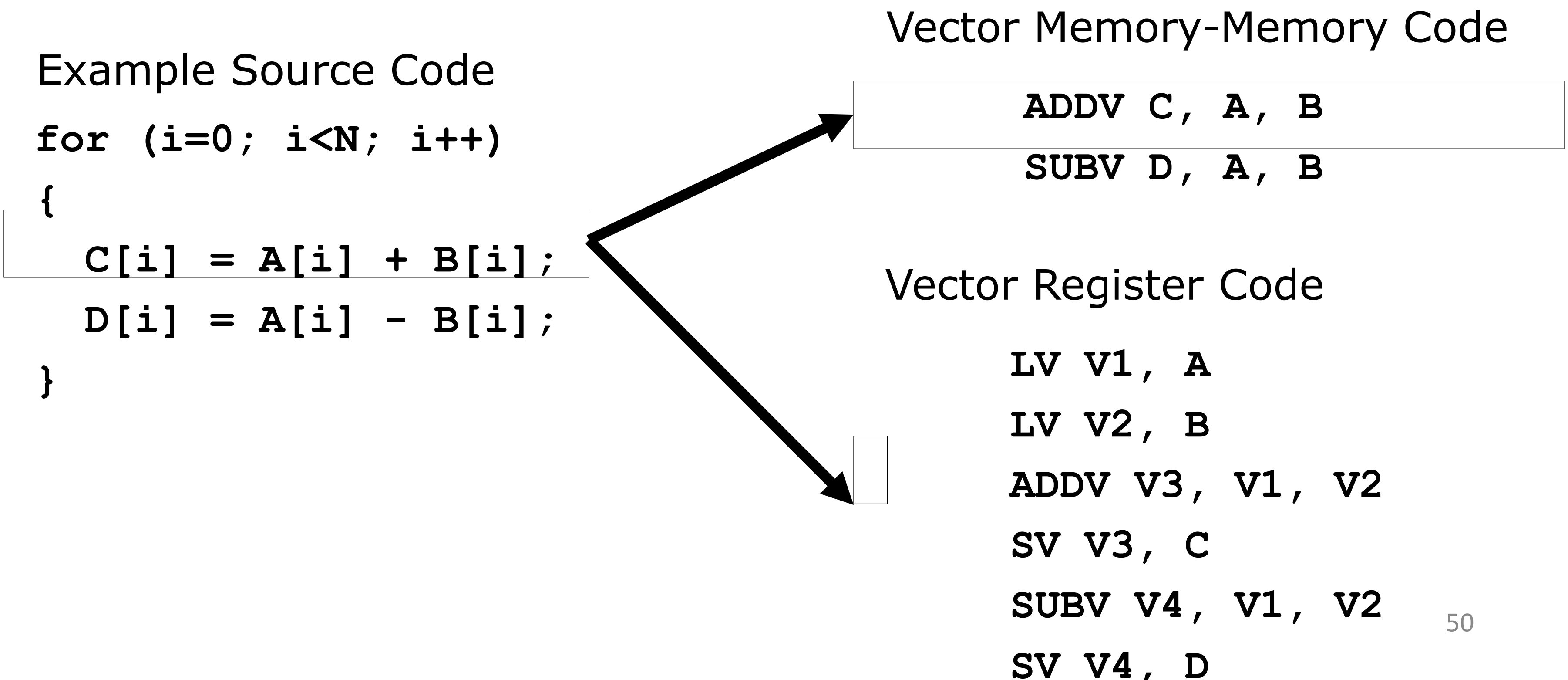


- Memory system provides 256GB/s DRAM bandwidth per CPU
- Up to 16 CPUs and up to 1TB DRAM form shared-memory *node*
  – total of 4TB/s bandwidth to shared DRAM memory
- Up to 512 nodes connected via 128GB/s network links (message passing between nodes)

- 65nm CMOS technology
- Vector unit (3.2 GHz)
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 64-bit functional units: 2 multiply, 2 add, 1 divide/sqrt, 1 logical, 1 mask unit
  - 8 lanes (32+ FLOPS/cycle, 100+ GFLOPS peak per CPU)
  - 1 load or store unit (8 x 8-byte accesses/cycle)
- Scalar unit (1.6 GHz)
  - 4-way superscalar with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache

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were memory-memory machines
- 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
```

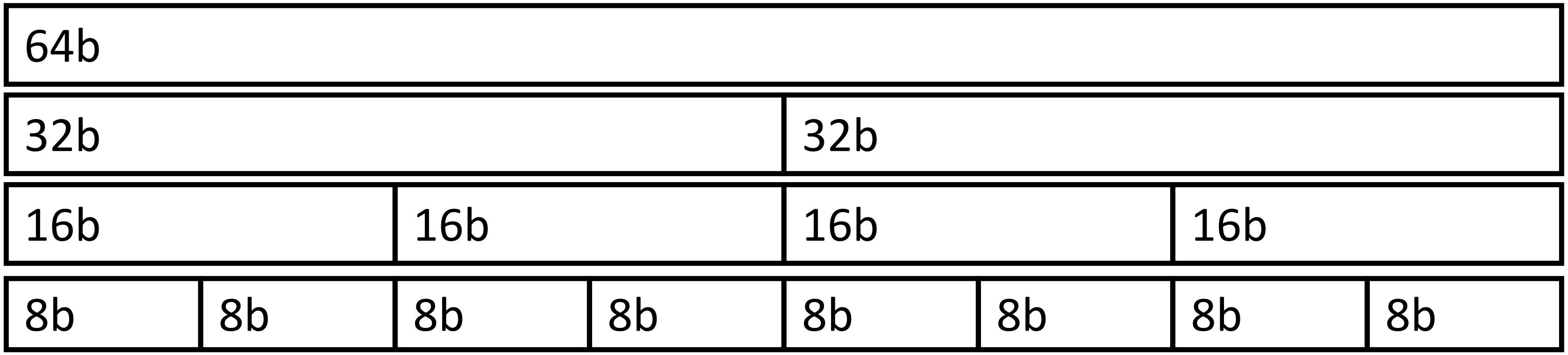
Vector Register Code

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

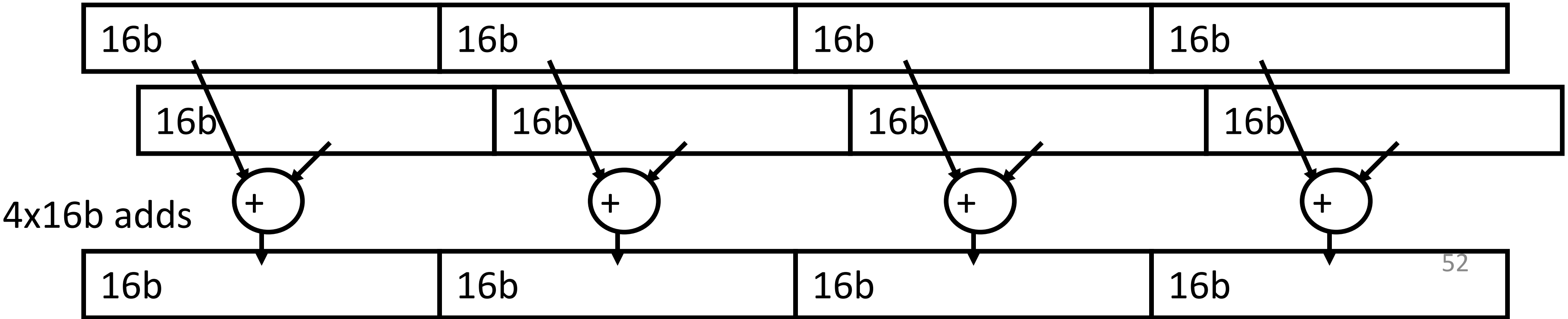
Vector Memory-Memory vs. Vector Register Machines

- 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
- VMMAs结构使得多个向量操作重叠执行更困难
  - Must check dependencies on memory addresses
- VMMAs 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
  - For Cray-1, vector/scalar 均衡点在2个元素
- CDC Cray-1后续的机器 (Cyber-205, ETA-10) 都是寄存器型向量机

Multimedia Extensions (aka SIMD extensions)



- 在已有的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
- 单条指令可实现寄存器中所有向量元素的操作



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)

Types of Parallelism

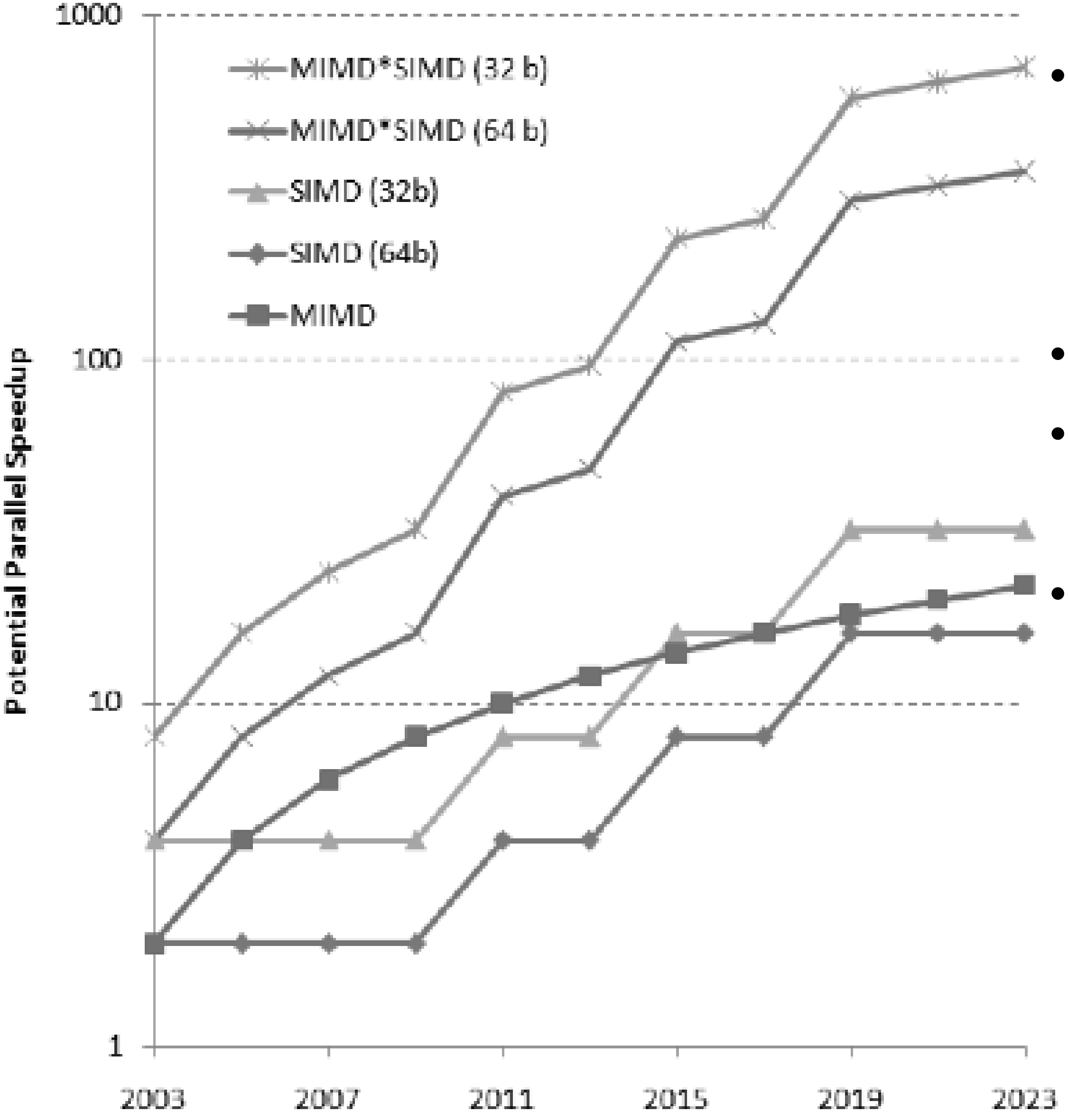
- Instruction-Level Parallelism (ILP)
  - Execute independent instructions from one instruction stream in parallel (pipelining, superscalar, VLIW)
- Thread-Level Parallelism (TLP)
  - Execute independent instruction streams in parallel (multithreading, multiple cores)
- Data-Level Parallelism (DLP)
  - Execute multiple operations of the same type in parallel (vector/SIMD execution)

- 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

Resurgence of DLP

- 应用需求和技术约束推动着体系结构的发展和选择
- 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
- SIMD-based 结构 (vector-SIMD, subword-SIMD, SIMT/GPUs) 是执行这些算法的最有效途径

DLP important for conventional CPUs too



- Prediction for x86 processors, from Hennessy & Patterson, 5th edition
  - ***Note: Educated guess, not Intel product plans!***
- TLP: 2+ cores / 2 years
- DLP: 2x width / 4 years
- DLP will account for more mainstream parallelism growth than TLP in next decade.
  - SIMD –single-instruction multiple-data (DLP)
  - MIMD- multiple-instruction multiple-data (TLP)

Graphics Processing Units (GPUs)

- 原来的GPU是指带有高性能浮点运算部件、可高效生成3D图形的具有固定功能的专用设备 (mid-late 1990s)
  - 让PC机具有类似工作站的图形功能
  - 用户可以配置图形处理流水线，但不是真正的对其编程
- 随着时间的推移，GPU加入了越来越多的可编程性 (2001-2005)
  - 例如新的语言 Cg可用来编写一些小的程序处理图形的顶点或像素，是Windows DirectX的变体
  - 大规模并行（针对每帧上百万顶点或像素）但非常受限于编程模型
- 有些用户注意到通过将输入和输出数据映射为图像，并对顶点或像素渲染计算 可进行通用计算
  - 因为不得不使用图形流水线模型，这对完成通用计算来说是个非常难用的编程模型

General-Purpose GPUs (GP-GPUs)

- 2006年, Nvidia 的 GeForce 8800 GPU 支持一种新的编程语言: CUDA
  - “Compute Unified Device Architecture”
  - 随后工业界推出OpenCL，与CUDA具有相同的ideas, 但独立于供应商
- Idea: 针对通用计算，发挥GPU的计算的高性能和存储器的高带宽来加速一些通用计算中的核心（Kernels）
- 一种附加处理器模型（GPU作为附加设备）： Host CPU发射数据并行的kernels 到GP-GPU上运行
- 我们仅讨论Nvidia CUDA样式的简化版本，仅考虑GPU的计算核部分，不涉及图形加速部分

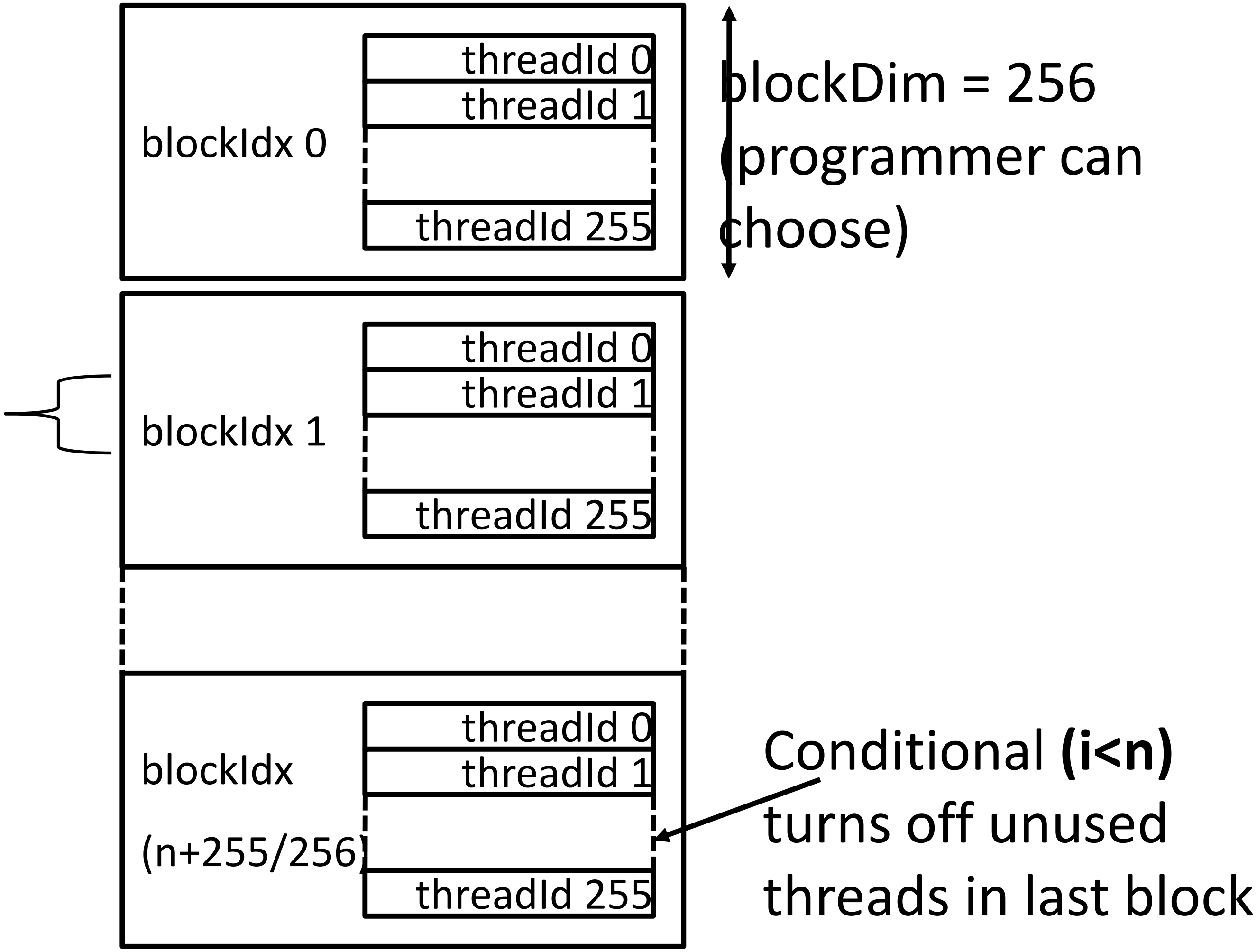
Simplified CUDA Programming Model

- 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*）

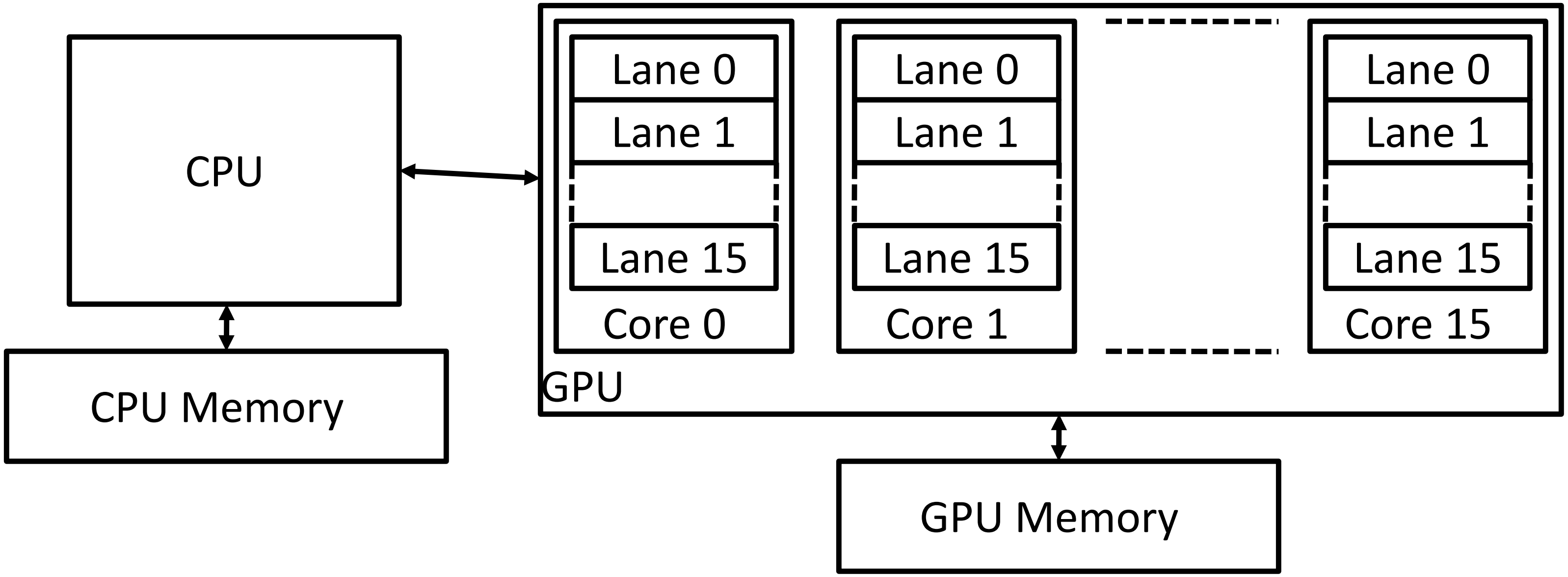
```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{ for (int i=0; i<n; i++)
      y[i] = a*x[i] + y[i]; }

// 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]; }
```

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*)



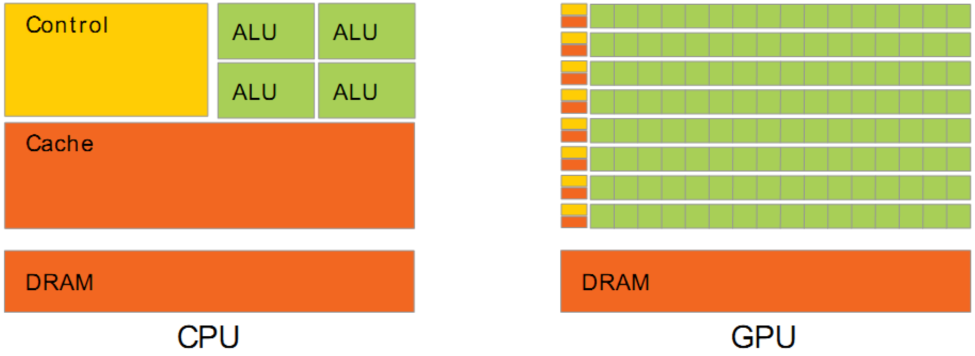
Hardware Execution Model

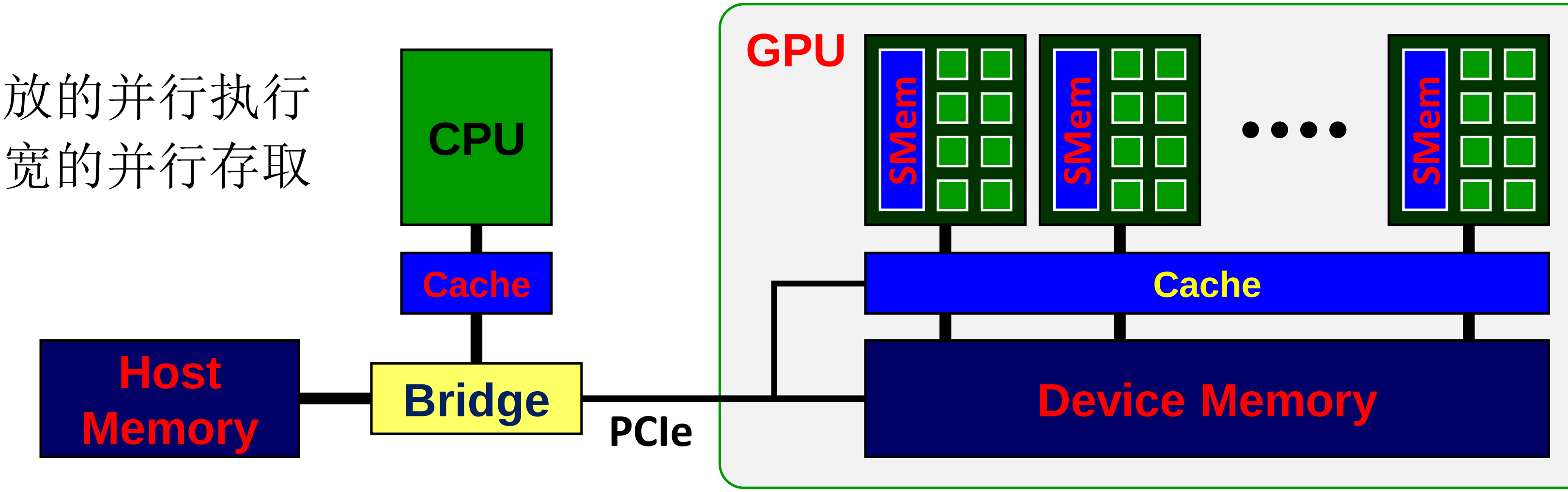


- GPU 由多个并行核构成，每个核包含一个多线程SIMD处理器（包含多个车道）
- CPU 发送整个 “grid”到GPU，由GPU将这些线程块分发到多个核上（每个线程块在一个核上运行）
  - GPU上核的数量对程序员而言是透明的

Using CPU+GPU Architecture

- 异构系统（异构多核）
- 针对每个任务选择合适的处理器和存储器
- 通用CPU 适合执行一些串行的线程
  - 串行执行快
  - 带有cache，访问存储器延时低
- GPU 适合执行大量并行的线程
  - 可扩放的并行执行
  - 高带宽的并行存取





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
- 许多线程块组成一个网格

- GPU 由硬件对线程进行管理
  - Thread Block Scheduler
  - SIMD Thread Scheduler

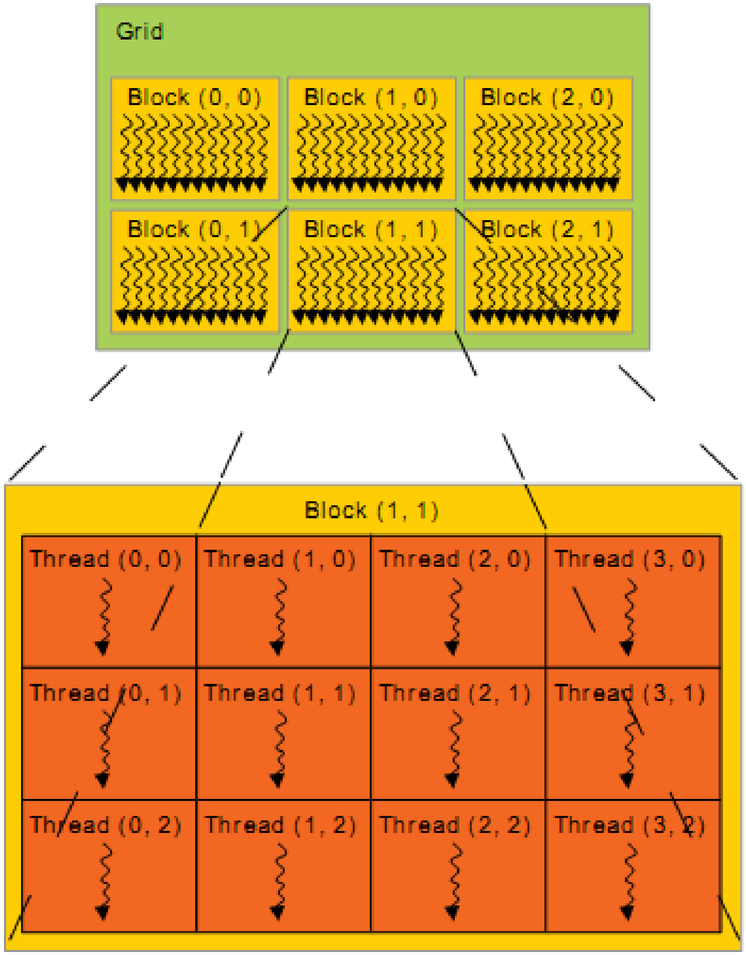
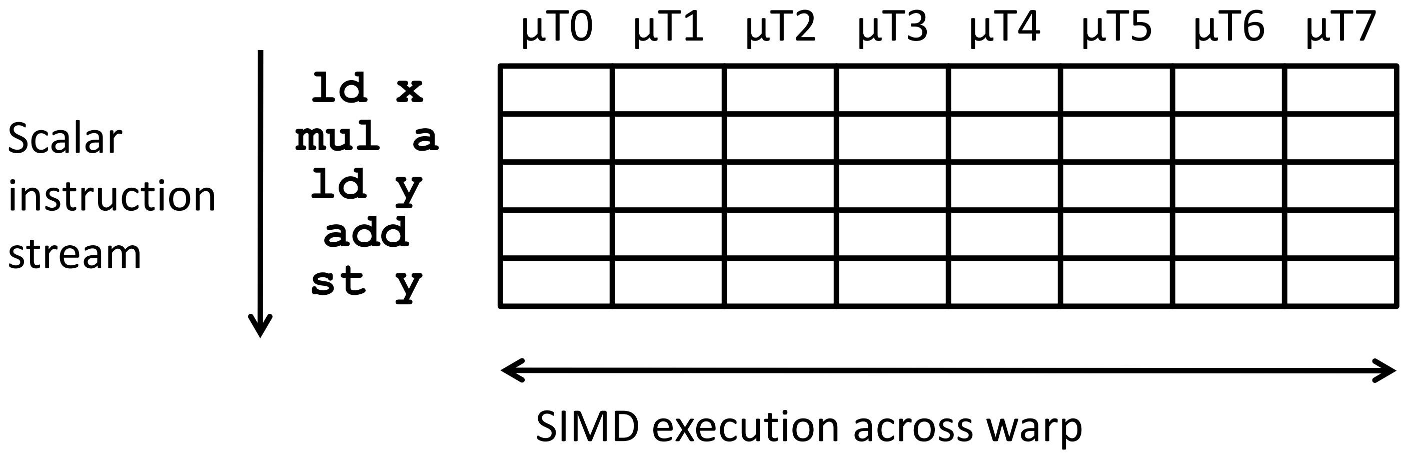


Figure 6 Grid of Thread Blocks

CUDA：“Single Instruction, Multiple Thread”

- GPUs 使用 SIMT模型, 每个CUDA线程的标量指令流汇聚在一起在硬件上以SIMD方式执行 (Nvidia groups 32 CUDA threads into a *warp*)



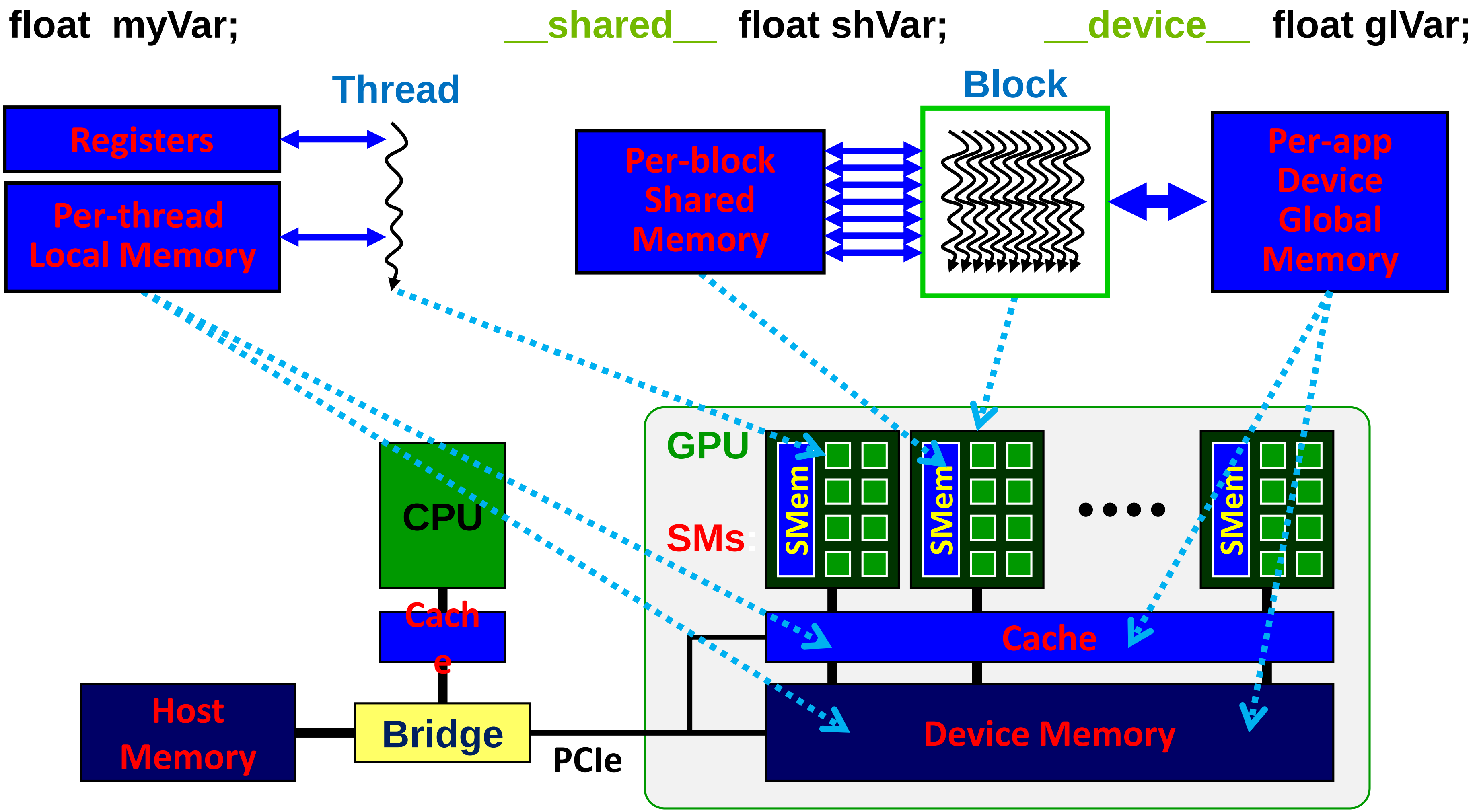
Scalar instruction stream

```
ld x
mul a
ld y
add
st y
```

µT0 µT1 µT2 µT3 µT4 µT5 µT6 µT7

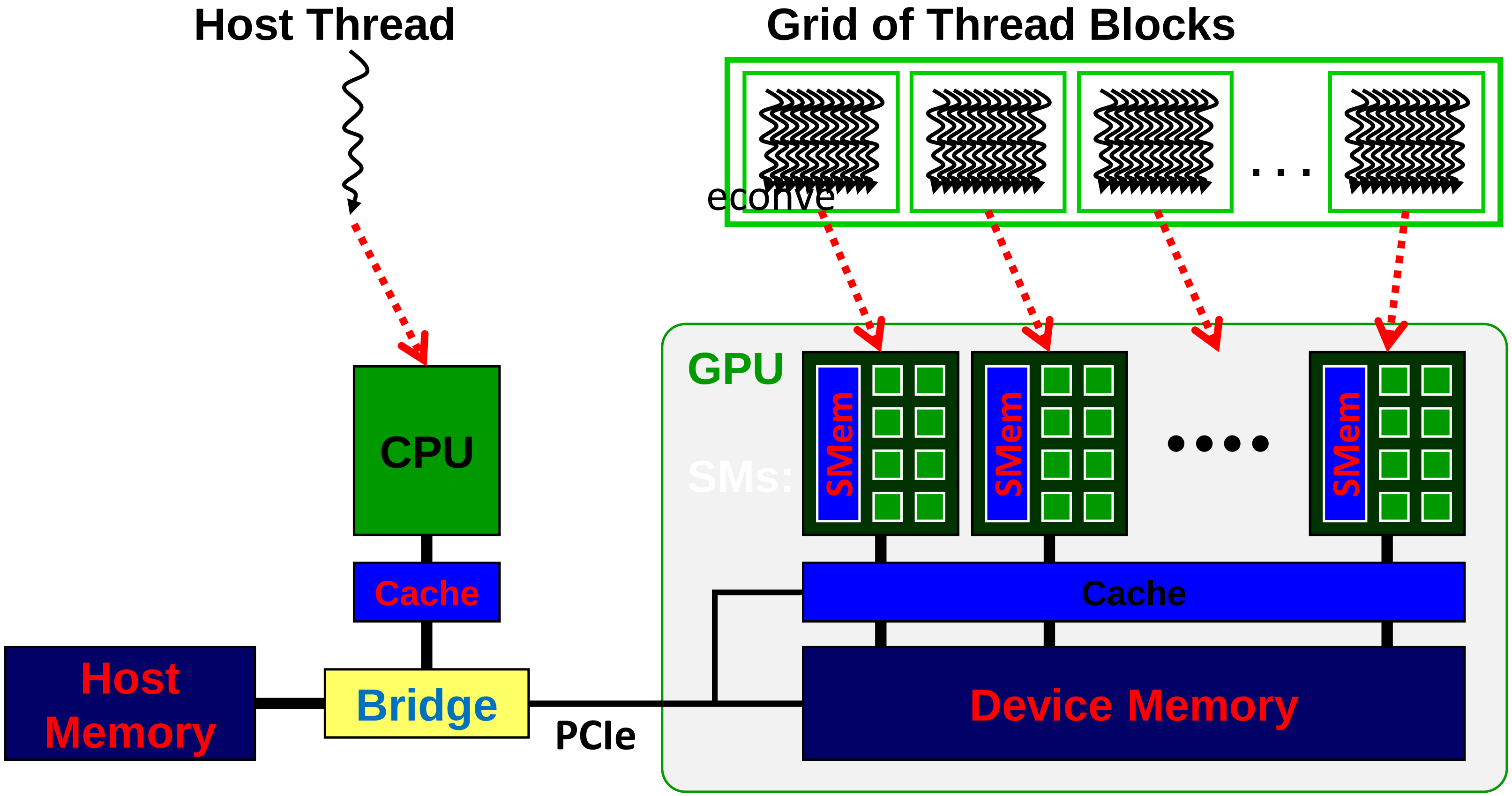
SIMD execution across warp

Thread blocks execute on an SM
Thread instructions execute on a core



CUDA kernel maps to Grid of Blocks

```
kernel_func<<<nblk,
 nthread>>>(param, … );
```



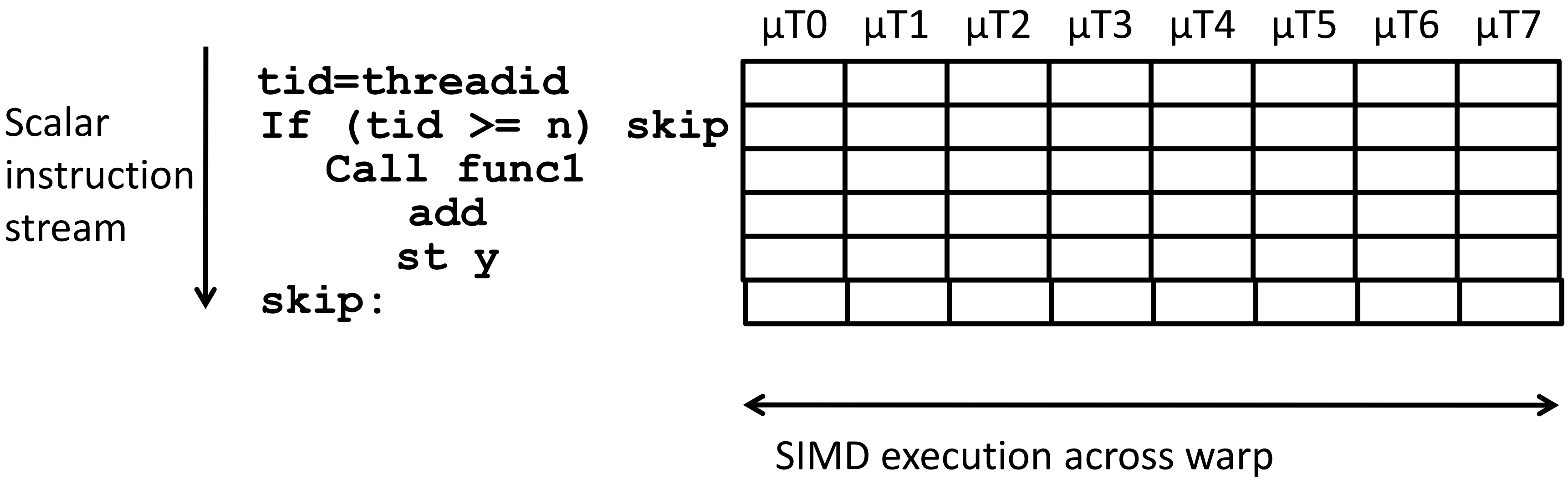
NVIDIA Instruction Set Arch.

- 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:

```
shl.s32         R8, blockIdx, 9         ; Thread Block ID * Block size (512 or 29)
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 R0D, RD0, RD4   ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2   ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0               ; Y[i] = sum (X[i]*a + Y[i])
```

Conditionals in SIMT model

- 简单的 if-then-else 编译为谓词（有条件）执行等价于在向量屏蔽寄存器的作用下的向量运算
- 比较复杂的控制流编译生成分支
- 如何处理分支？



Scalar instruction stream

```
tid=threadid
If (tid >= n) skip
   Call func1
       add
      st y
skip:
```

µT0 µT1 µT2 µT3 µT4 µT5 µT6 µT7

SIMD execution across warp

Conditional Branching

- 与向量结构类似, GPU 使用内部的屏蔽字(masks)
- 还使用了
  - 分支同步堆栈
    - 保存分支成功的路径地址
    - 保存该路径的每个SIMD lane 的屏蔽字(mask)
      - 即指示哪些车道可以提交结果 (all threads execute)
  - 指令标记(instruction markers）
    - 管理何时分支到多个执行路径，何时路径汇合
- PTX层
  - CUDA线程的控制流由PTX分支指令(branch、call、return and exit）
  - 由程序员指定的每个线程车道的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 提供堆栈保存分支成功的路径
  - GPU硬件指令带有控制每个线程车道1-bit谓词寄存器

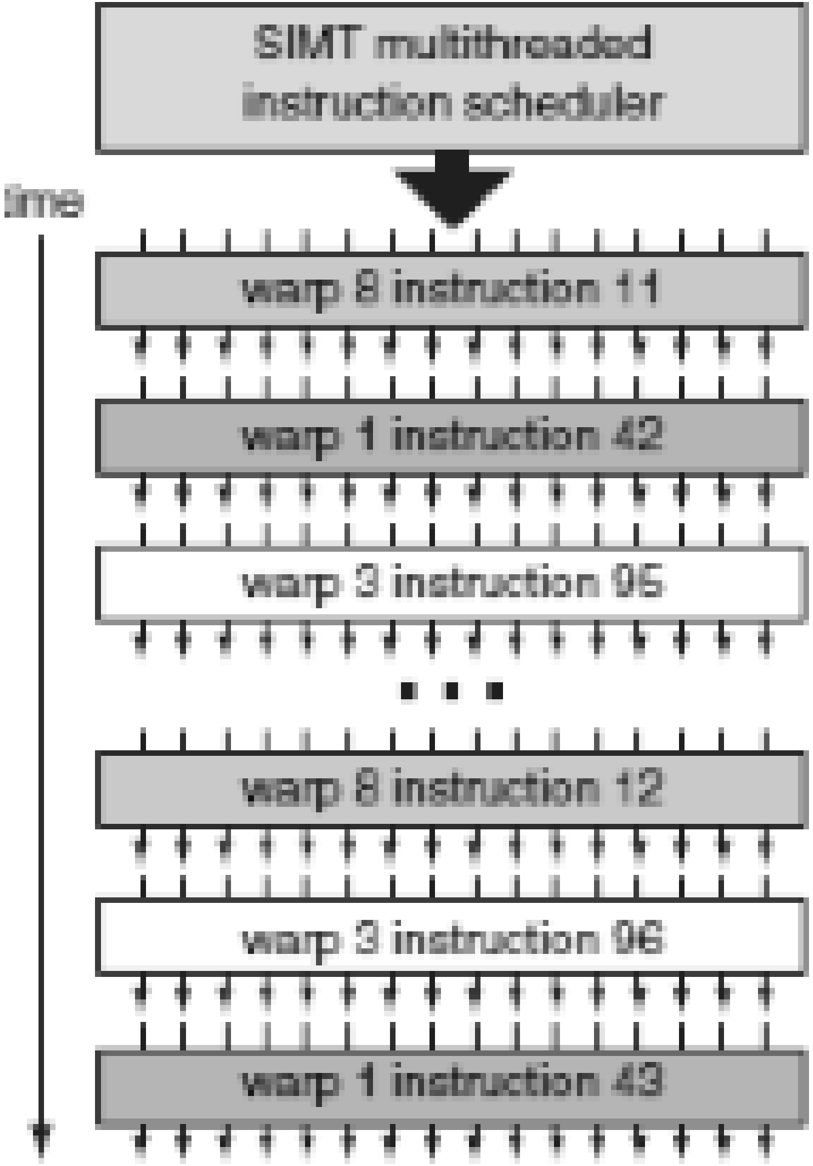
Branch divergence

- 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 线程所走的路径相同，那么可以保持这种SIMD fashion
- 如果各线程选择的方向不一致，那么创建一个屏蔽（mask） 向量来指示各线程的转移方向（成功、失败）
- 继续执行分支失败的路径，将分支成功的路径压入硬件堆栈（分支同步堆栈），待后续执行
- 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，执行转移成功路径
  - 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）

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
```

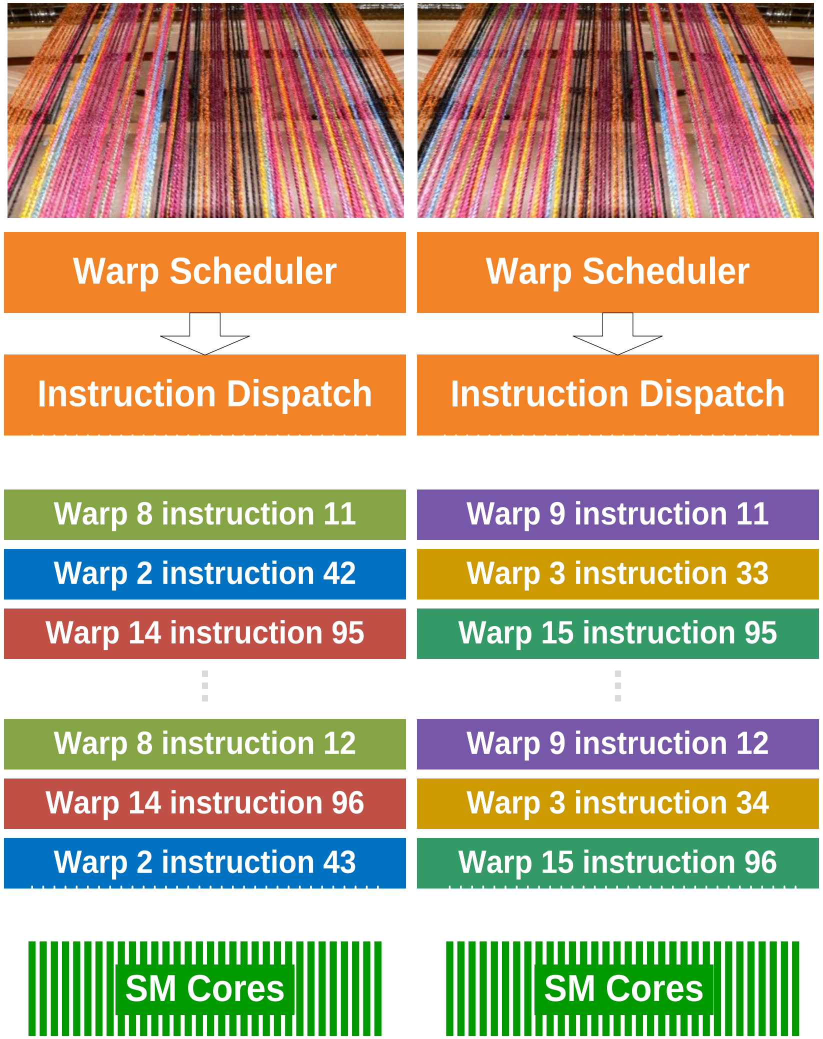
Warps are multithreaded on core



[Nvidia, 2010]

- 一个 warp 由 32个 µthreads 构成
- 多个warp线程在单个核上交叉运行，以隐藏存储器访问呢和功能部件的延迟
- 单个线程块包含多个warp (up to 512 µT max in CUDA), 这些warp都映射到同一个核上
- 多个线程块也可以在同一个核上运行

SIMT Warp Execution in the SM



Warp Scheduler

Instruction Dispatch

Warp 8 instruction 11

Warp 2 instruction 42

Warp 14 instruction 95

Warp 8 instruction 12

Warp 14 instruction 96

Warp 2 instruction 43

SM Cores

Warp Scheduler

Instruction Dispatch

Warp 9 instruction 11

Warp 3 instruction 33

Warp 15 instruction 95

Warp 9 instruction 12

Warp 3 instruction 34

Warp 15 instruction 96

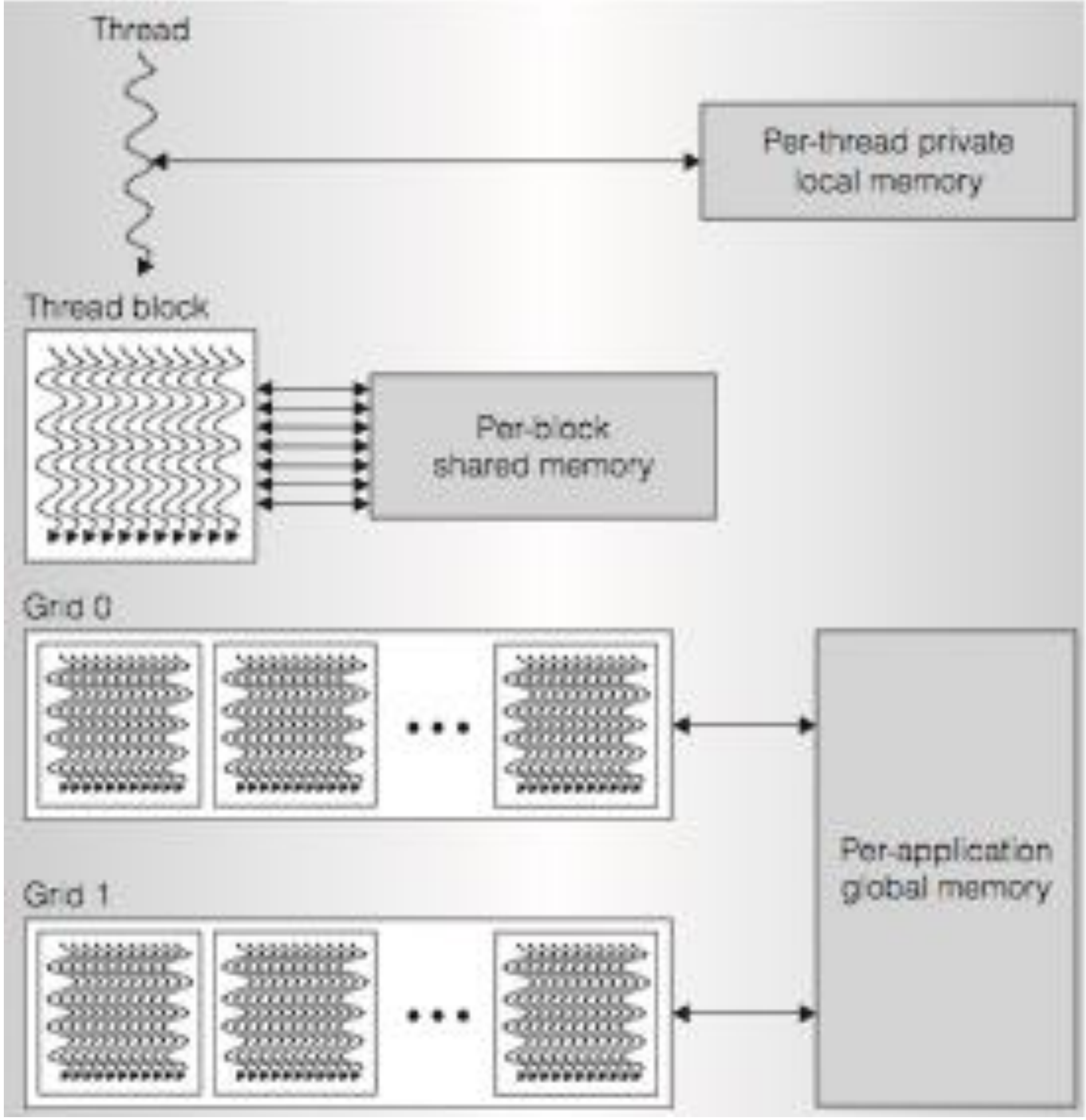
SM Cores

Warp: 一组32个CUDA 线程，构成一条SIMD指令

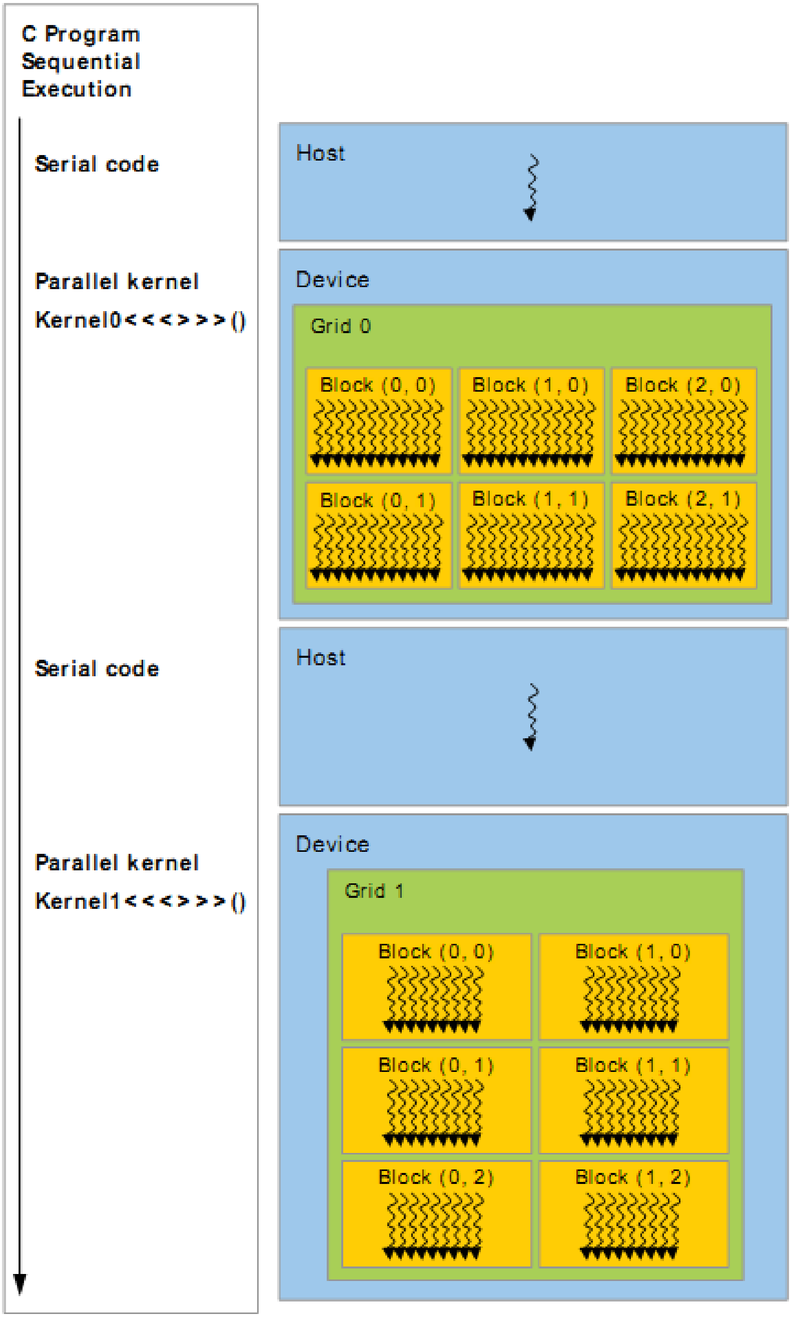
SIMT: Single-Instruction Multi-Thread 建立适用于warp的并行线程的指令

- SM 双发射流水线选择两个warp 发射到并行的核上
- SIMT warp 执行warp中的指令（SIMD指令）(32个CUDA thread）
- 通过谓词寄存器控制单个线程 (CUDA thread) 执行结果 (enable/disable)
- 通过同步堆栈管理线程的分支
- 通过规整的一致的计算来处理分支速度高于非规整的分支转移

GPU Memory Hierarchy


Thread
Per-thread private local memory
Thread block
Per-block shared memory
Grid 0
Grid 1
Per-application global memory

[Nvidia, 2010]

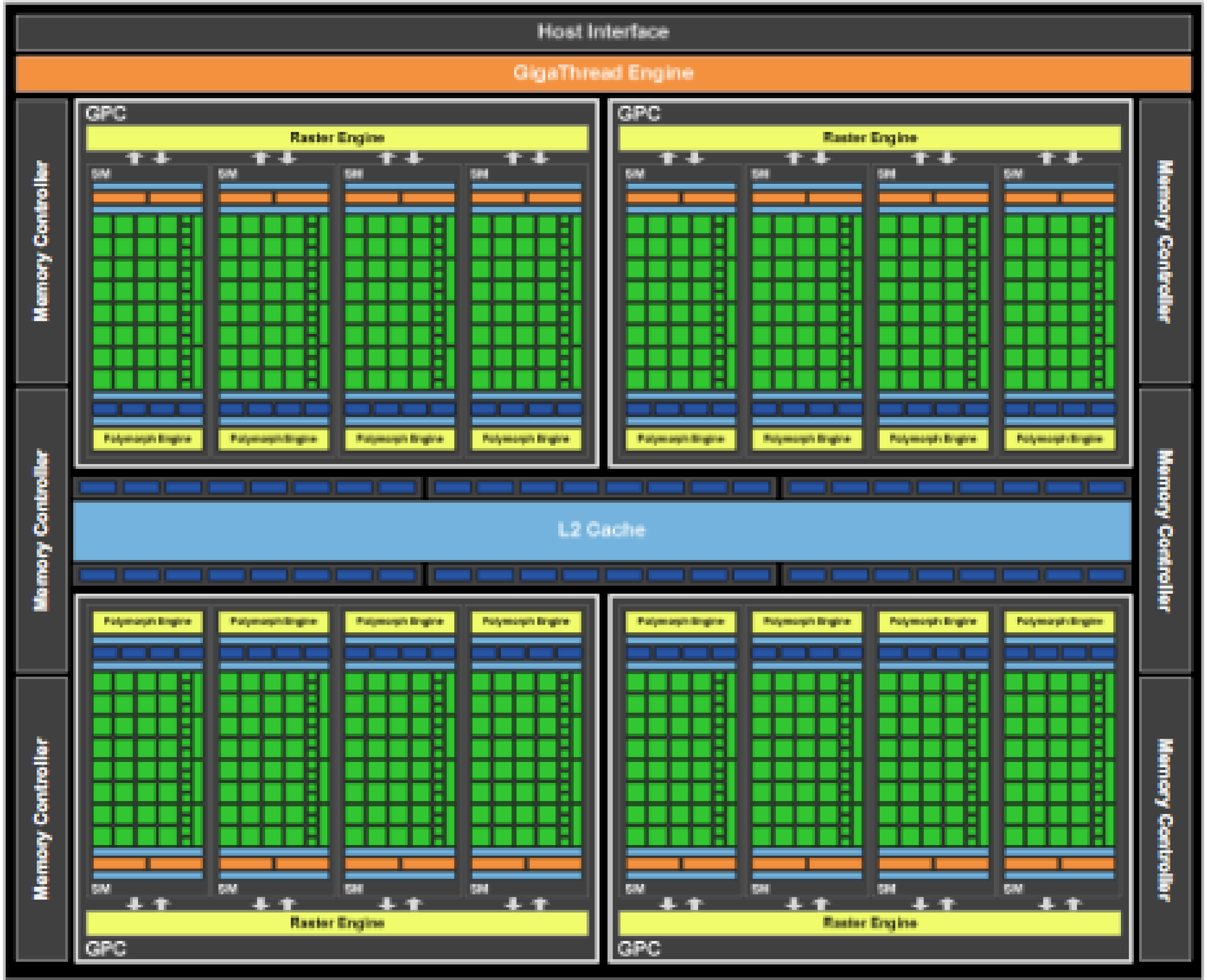


Serial code executes on the host while parallel code executes on the device.

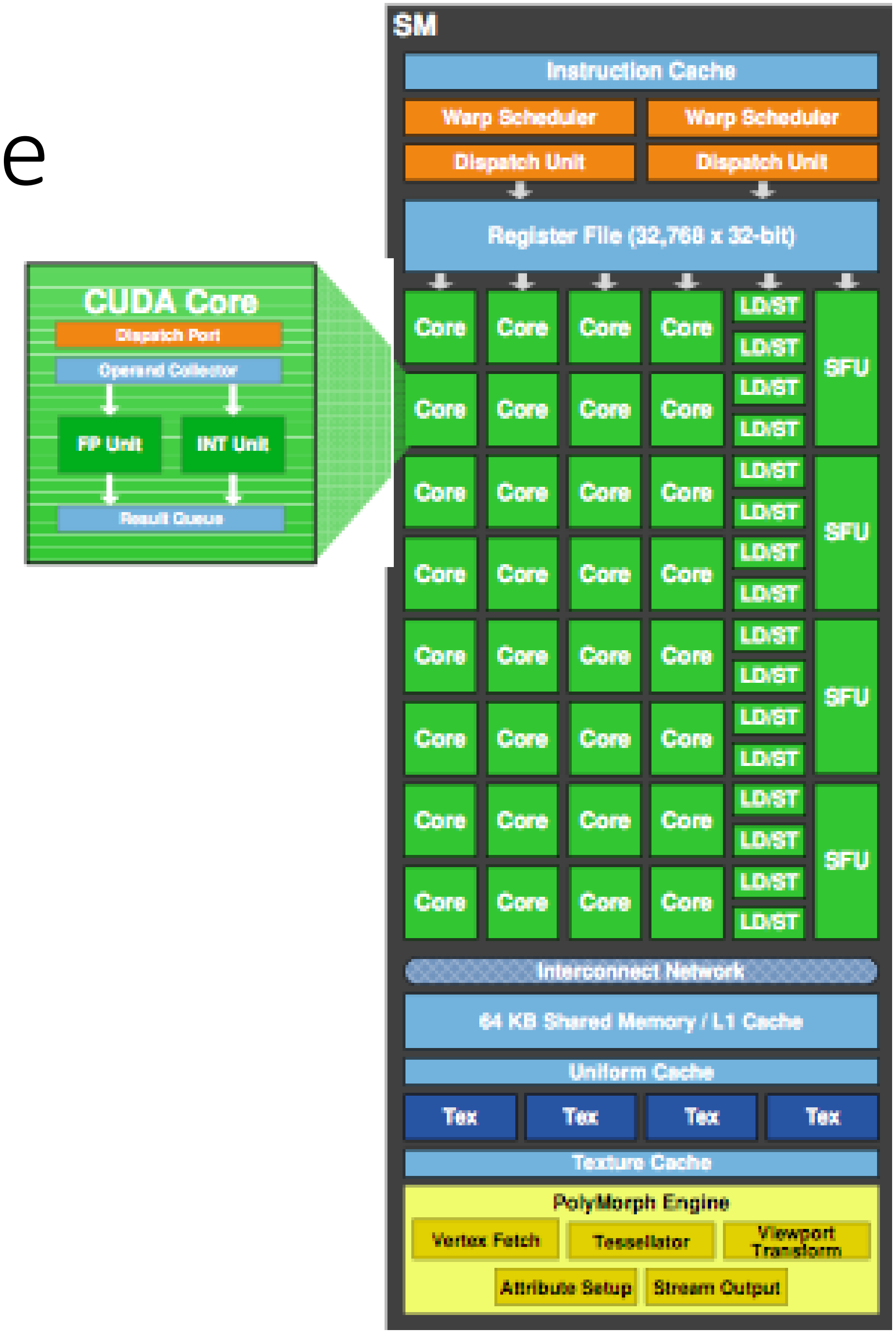
Figure 8 Heterogeneous Programming

Nvidia Fermi GF100 GPU

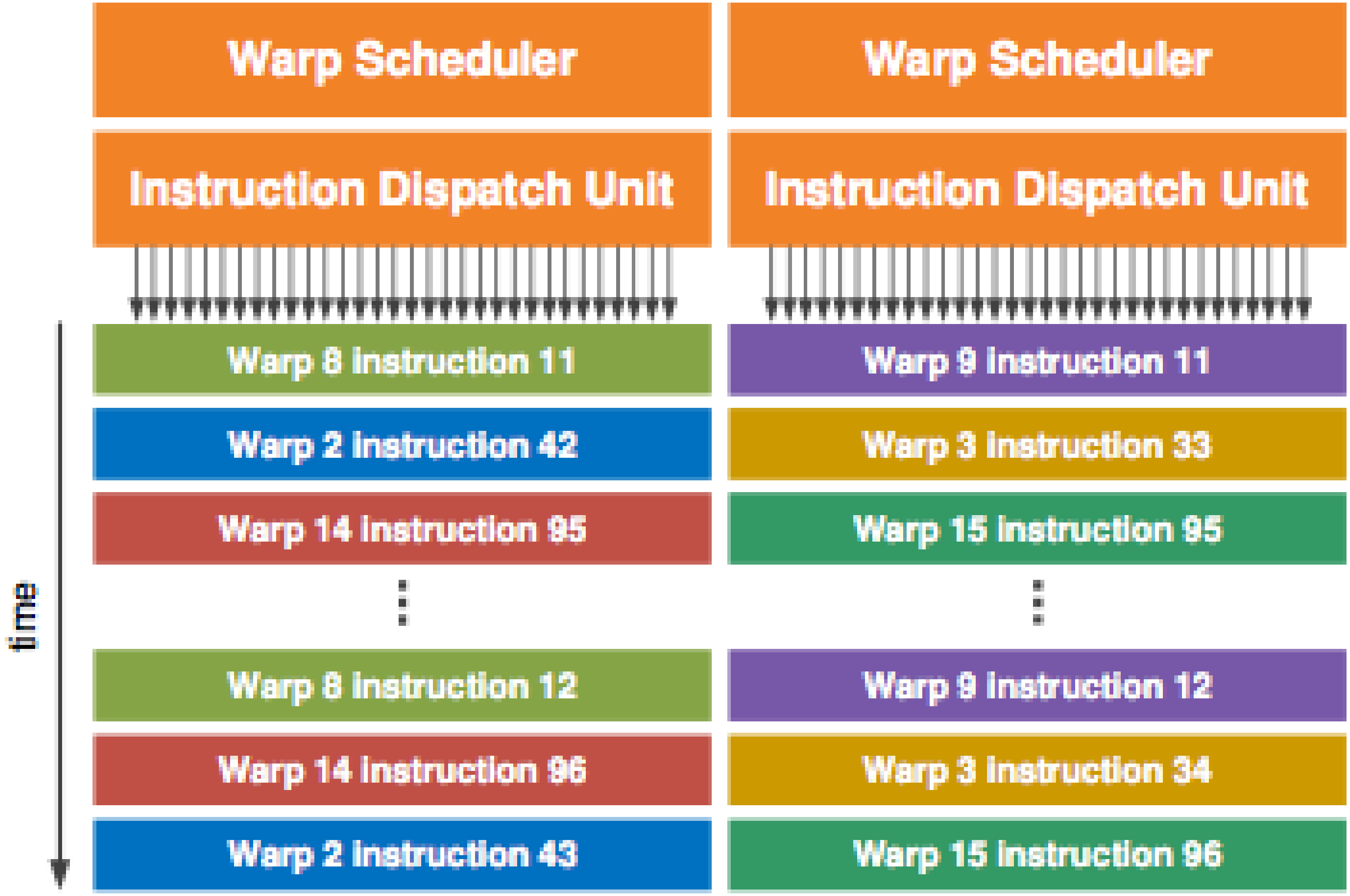
[Nvidia, 2010]



Fermi “Streaming Multiprocessor” Core

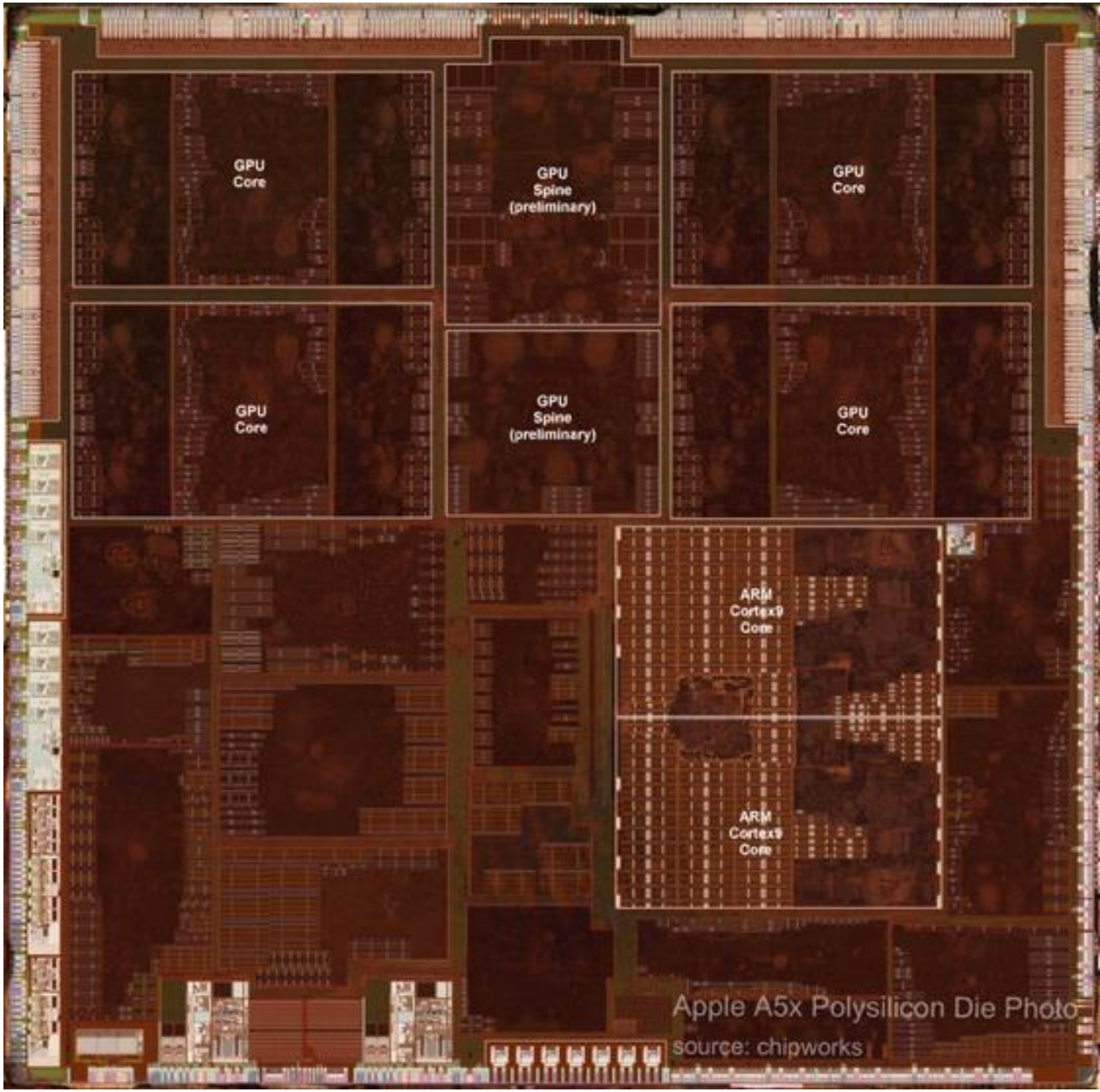


Fermi Dual-Issue Warp Scheduler



Apple A5X Processor for iPad v3 (2012)

- 12.90mm x 12.79mm
- 45nm technology



[Source: Chipworks, 2012]

Loop-Level Parallelism

- 研究的焦点问题是：在循环结构中，后面的循环是否依赖于前面的循环
  - Loop-carried dependence

- Example 1:

```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

- 循环级并行：No loop-carried dependence

Loop-Level Parallelism

- Example 2:

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i]; /* S1 */
        B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

- S1 and S2 均使用了前一次循环的结果
- S2 使用同一循环中的S1的结果

Loop-Level Parallelism

- Example 3:

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i]; /* S1 */
        B[i+1] = C[i] + D[i]; /* S2 */
}
```

- S1 使用了S2前一次循环的结果，但数据依赖没有形成环路，也是循环级并行。
- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Loop-Level Parallelism

- Example 4:

```
for (i=0;i<100;i=i+1)  {
        A[i] = B[i] + C[i];
        D[i] = A[i] * E[i];
}
```

- Example 5:

```
for (i=1;i<100;i=i+1)  {
        Y[i] = Y[i-1] + Y[i];   //递推关系
}
```

通常循环间相关表现形式为：递推关系

Finding dependencies

- Assume indices are affine:
  - a x i + b (i is loop index)

- Assume:
  - Store to a x i + b, then
  - Load from c x i + d
  - i runs from m to n
  - Dependence exists if:
    - Given j, k such that $m \le j \le n$, $m \le k \le n$
    - Store to a x j + b, load from a x k + d, and a x j + b = c x k + d

Finding dependencies

- 通常在编译时无法确定是否相关
- 数据依赖关系的测试
  - GCD test:
    - 如果存在相关，那么 GCD(*c*,*a*) 必须能整除 (*d*-*b*)

- Example:

```
for (i=0; i<100; i=i+1) {
   X[2*i+3] = X[2*i] * 5.0;
}
```

Finding dependencies

- Example 2:

```
for (i=0; i<100; i=i+1) {
   Y[i] = X[i] / c; /* S1 */
   X[i] = X[i] + c; /* S2 */
   Z[i] = Y[i] + c; /* S3 */
   Y[i] = c - Y[i]; /* S4 */
}
```

- 反相关和输出相关
- 通过重命名消除这两类相关

Reductions

- Reduction Operation:

  ```
  for (i=9999; i>=0; i=i-1)
              sum = sum + x[i] * y[i];
  ```

- Transform to…

  ```
  for (i=9999; i>=0; i=i-1)
              sum [i] = x[i] * y[i];
  for (i=9999; i>=0; i=i-1)
              finalsum = finalsum + sum[i];   //该循环存在循环间相关
  ```

- Do on p processors:

  ```
  for (i=999; i>=0; i=i-1)
              finalsum[p] = finalsum[p] + sum[i+1000*p];
  ```

- Note: assumes associativity!

GPU Future

- High-end desktops have separate GPU chip, but trend towards integrating GPU on same die as CPU (already in laptops, tablets and smartphones)
  - Advantage is shared memory with CPU, no need to transfer data
  - Disadvantage is reduced memory bandwidth compared to dedicated smaller-capacity specialized memory system
    - Graphics DRAM (GDDR) versus regular DRAM (DDR3)
- Will GP-GPU survive? Or will improvements in CPU DLP make GP-GPU redundant?
  - On same die, CPU and GPU should have same memory bandwidth
  - GPU might have more FLOPS as needed for graphics anyway

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