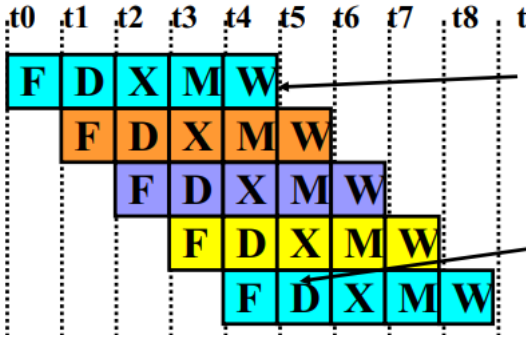
CA 数据集并行

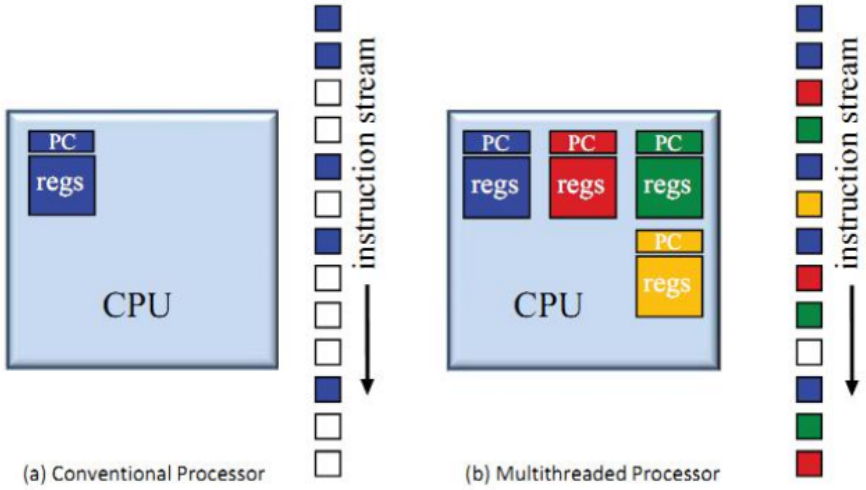
1 多线程（指令集并行）

- 如何保证流水线中指令间无数据依赖关系?
- 一种办法：在相同的流水线中交叉执行来自不同线程的指令

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5
T1: LW r5, 12(r1)

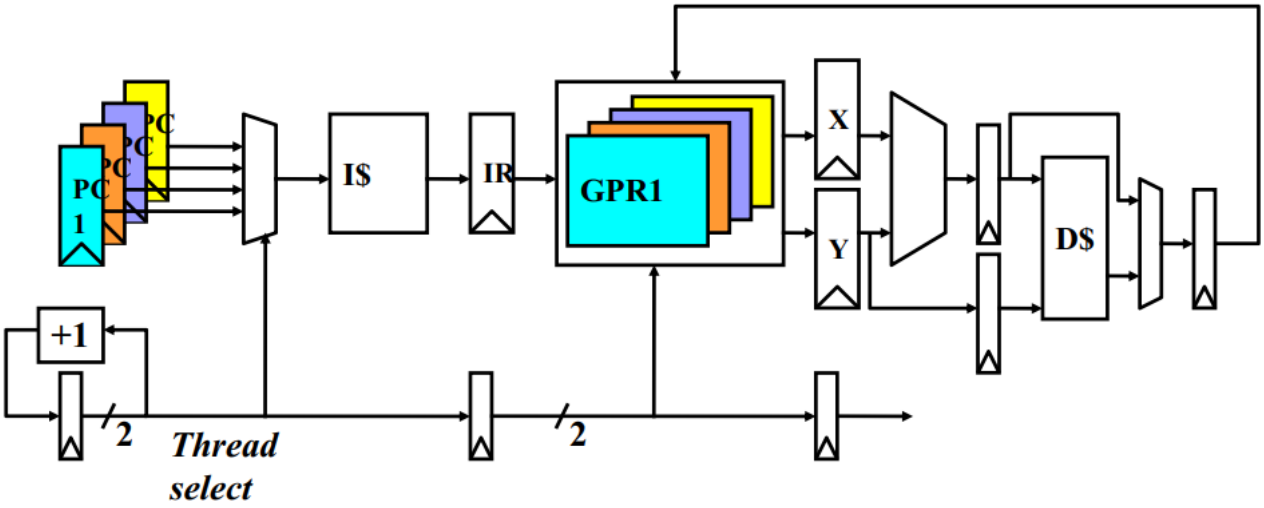




（传统）单线程处理器与多线程处理器

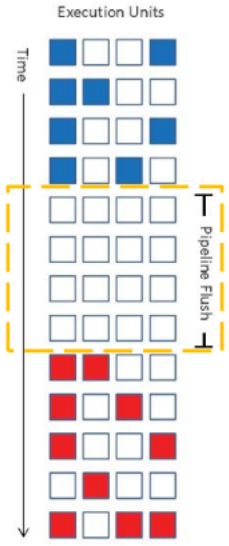
Simple Multithreaded Pipeline

- **必须传递线程选择信号以保证各流水段读写的正确性**
- **从软件（包括OS）的角度看 好像存在多个CPU（针对每个线程，CPU似乎运行的慢一些)**



Vertical(Coarse-Grain) Multithreading

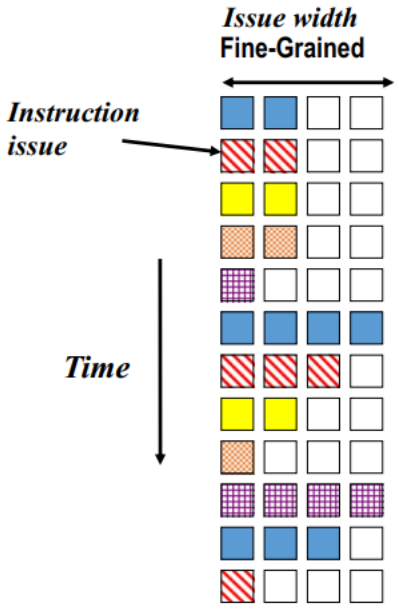
- **粗粒度多线程方式**
  - 基本思想源于分时系统
  - 当线程运行时存在较长时间延时时切换到另一线程
    - 例如：Cache失效时
    - 等待同步结束
  - 同一线程指令间的较短延时不切换
- **如果基于粗粒度的时钟周期交叉运行模式，结果怎样?**
  - 减少了垂直方向的浪费，但仍然存在垂直方向的浪费
  - 减少水平方向的浪费? X
- **最早的CGMT系统:**
  - DYSEAC [Leiner, 1954]
    - 美国国家标准局为美军通信兵团设计建造
  - TX-2 [Forgie, 1957]
    - MIT Lincoln Laboratory.



A *Coarse-Grain Multithreaded* architecture has the ability to switch between threads with only a short hardware context switch latency < 10 cycles

线程间切换速度快！<10 cycles

Fine-Grain Multithreading



- **细粒度多线程**
  - 多个线程的指令交叉执行
- **如果基于细粒度的时钟周期交叉运行模式，结果怎样?**
  - 减少垂直方向的浪费
    - 当线程数足够多时，可消除垂直方向的浪费
  - 仍然存在水平方向的浪费
- **CDC 6600：第一次实现FGMT**
  - ALU部件100ns vs. 存储器访问1000ns

Summary: Multithreaded Categories

多线程技术	线程间共享的资源	上下文切换机制
None	共享：所有资源	操作系统负责切换
FGMT	独占：寄存器文件，控制逻辑/状态	每个Cycle切换
CGMT	独占：I-fetch buffer, 寄存器文件，控制逻辑/状态	流水线较长stall时切换
SMT	独占：I-fetch buffer，return address stack, 寄存器文件，控制逻辑/状态，reorder buffer，store queue等	所有上下文处于活跃状态，没有切换问题
CMP	共享：二级Cache，系统互联	所有上下文处于活跃状态，没有切换问题

(a) CMP	**空间维度划分发射宽度；静态划分执行所需资源**
(b) FGMT	时间维度划分发射宽度；静态划分执行所需资源
(c) CGMT	长延时时切换线程；每个cycle动态划分执行所需的资源
(d) SMT	每个cycle发射的指令可来自不同线程，动态划分执行所需的资源

向量处理机

1 SIMD向量处理机

- **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**

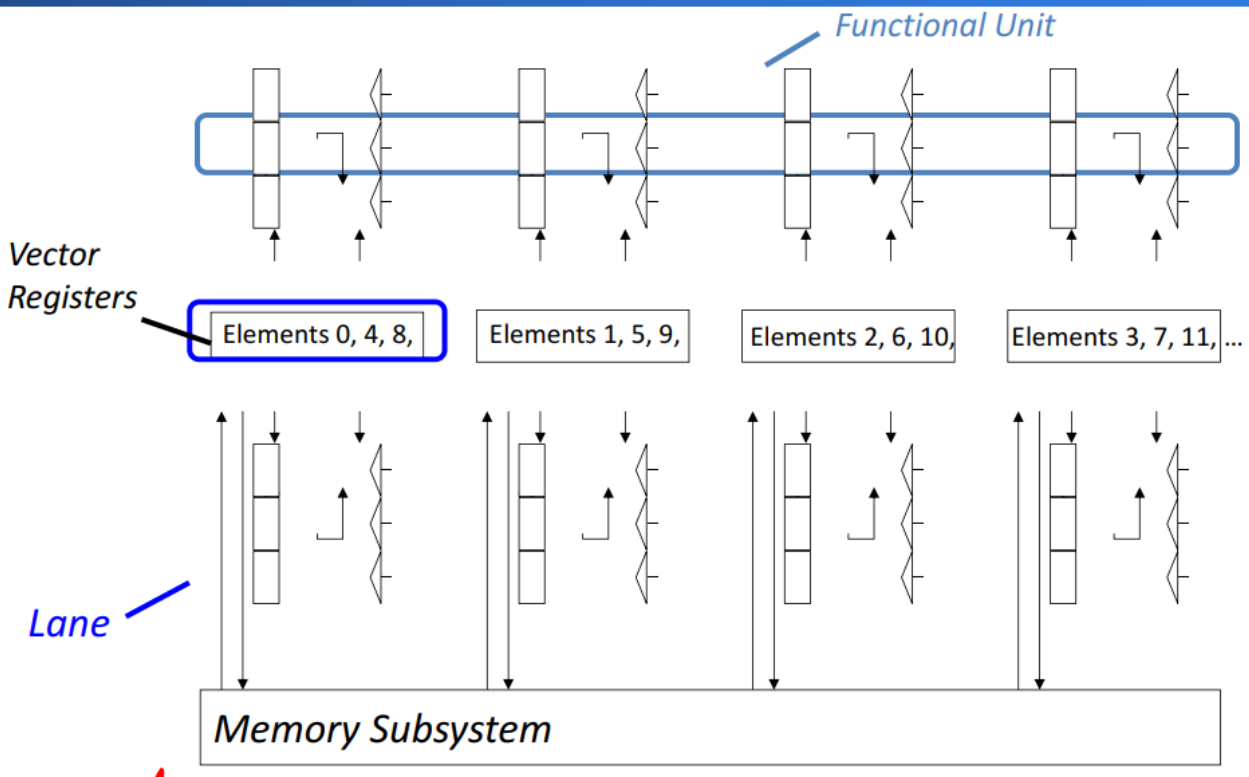
Vector Memory-Memory vs. Vector Register Machines

- **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

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**

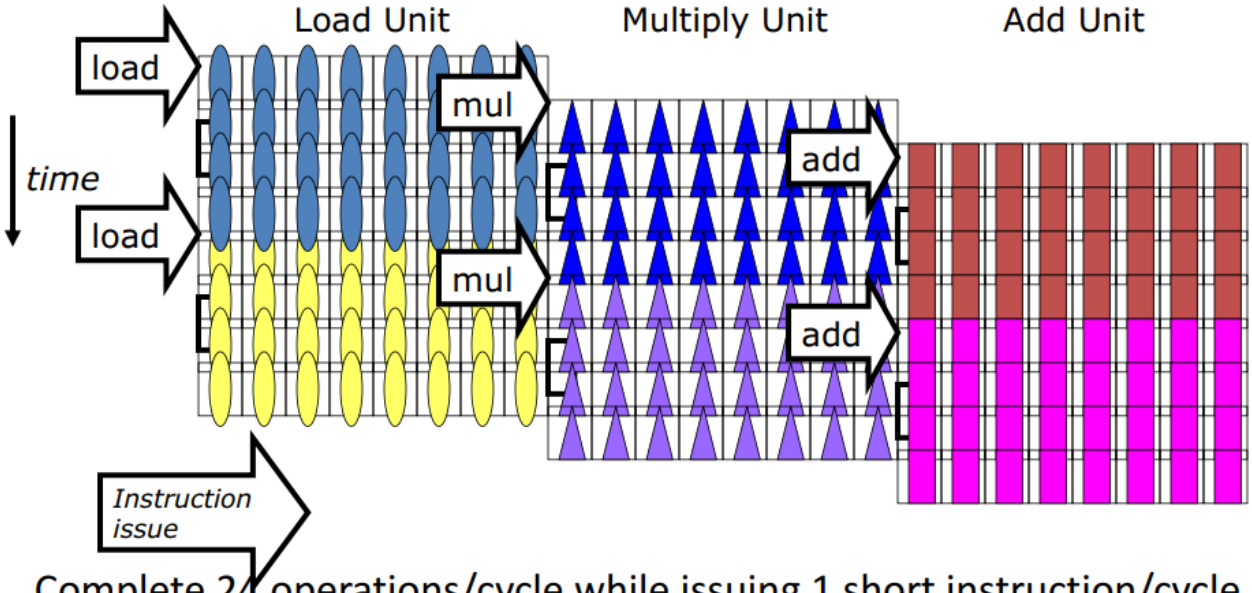
Vector Unit Structure



每个cycle 完成4个vector处理

4个lane车道

- **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

性能平菇（很重要）

1 Convoy及总执行时间

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)

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

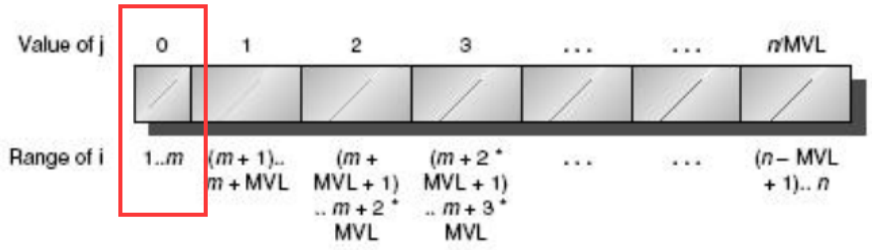
Assume convoys don't overlap; vector length = n

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*
LV	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*

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
```



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

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）

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

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

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

Tstart = 12 + 7 + 12 = 31

T200 = 660+4*31 = 784

每一元素的执行时间 = 784/200 = 3.9

```
        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
```

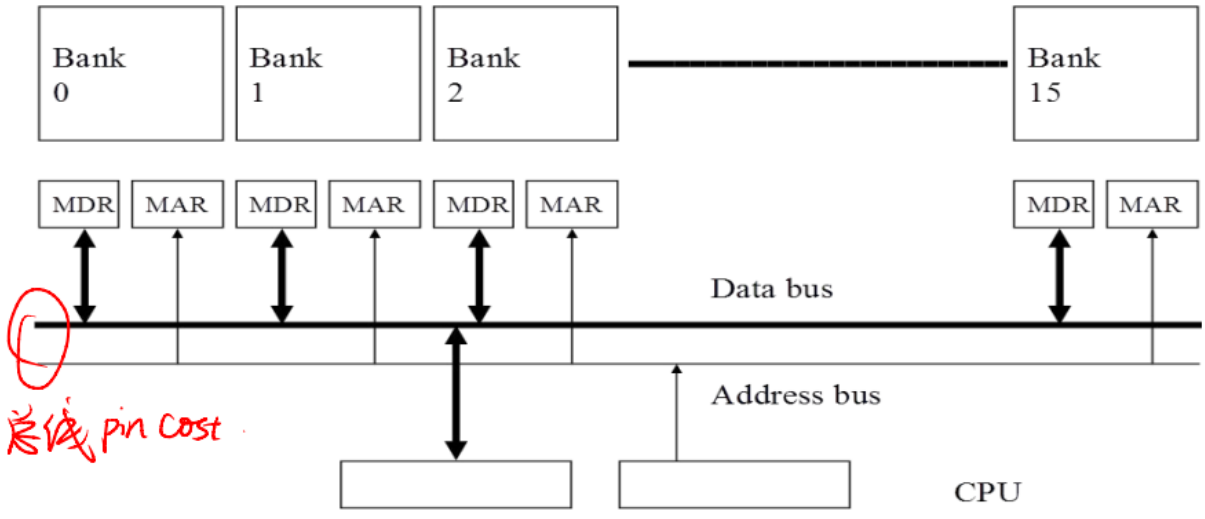
Common Vector Metrics

- **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

2 模性优化06-02

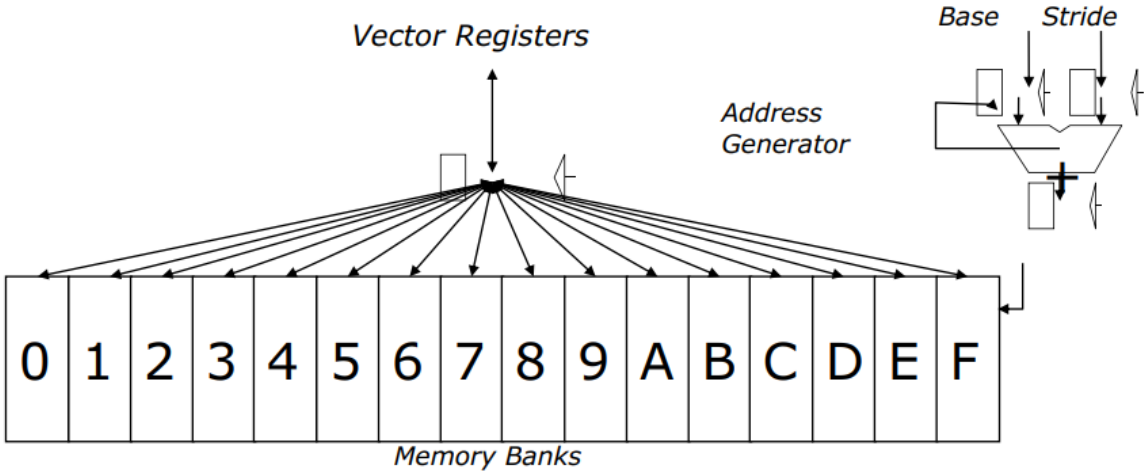
Memory Banking

- **独立存储体方式：由多个相互独立的存储体（Bank）构成存储器组织。可独立访问存储体，各存储体共享数据和地址总线 (minimize pin cost)**
- **每个周期可以启动和完成一个bank的访问**
- **如果N个存储器访问不同的bank可以并行执行**



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?

存储体 bank ≥ 6 即可

	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

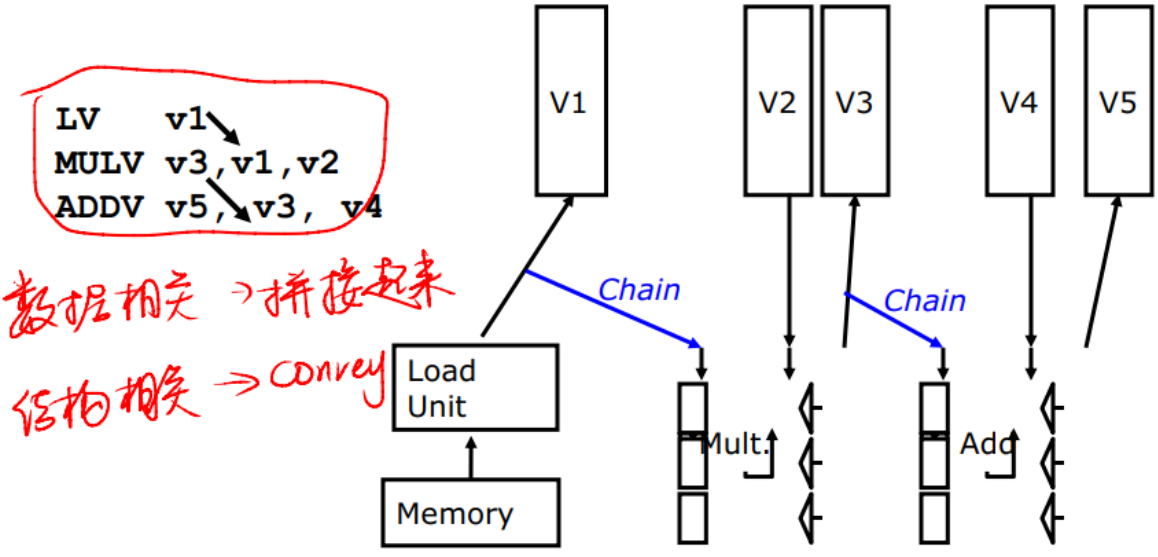
Figure F.7 Memory addresses (in bytes) by bank number and time slot at which access begins. Each memory bank latches the element address at the start of an access and is then busy for 6 clock cycles before returning a value to the CPU. Note that the CPU cannot keep all eight banks busy all the time because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

Bank# = (address/8) mod 8 ⇒ bank号. 双字对齐

Vector Opt#1: Vector Chaining

链接技术 → 定向路径

- 寄存器定向路径的向量机版本
- 首次在Cray-1上使用



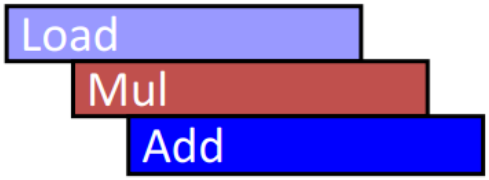
数据相关 → 拼接起来

结构相关 → Convey

并发执行

- 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**

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K(i)以及是否有数据相关**
- **使用CVI 设置步长（index 0, 1xm, 2xm, ..., 63xm）（见P266）**
  - CVI V1，R1　//创建索引向量V1

MMX

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

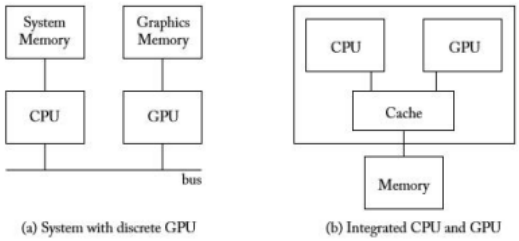
GPU

1 GPU简介

GPU基本硬件结构（1/2）

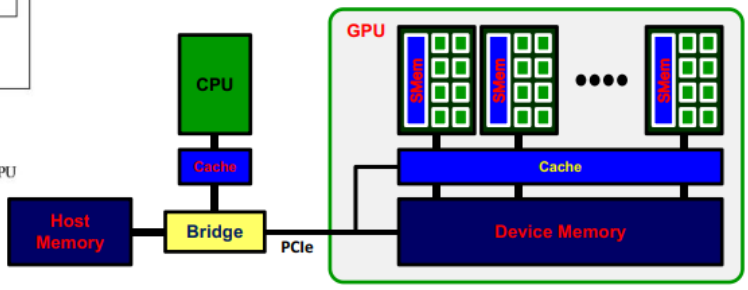
- CPU+GPU异构多核系统
  - 按照任务特点分配计算资源
  - GPU作为加速器（协处理器）
- CPU 核适合执行单一指令流
  - cache容量大，访问存储器延时低
- GPU 适合执行大量并行线程
  - 可扩放的并行执行
  - 高带宽的并行存取

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.



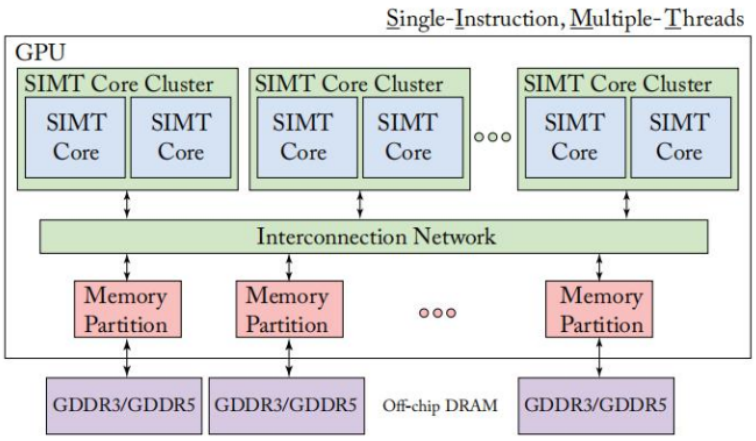
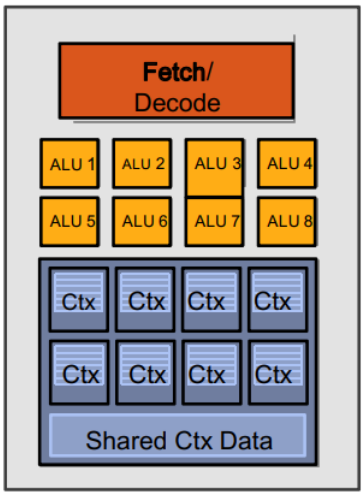


Figure 1.2: A generic modern GPU architecture.

- **现代GPU由许多核心组成，NVIDIA称这些核心为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。**



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

SIMD 处理的方式

SIMD处理并不意味着一定是执行SIMD指令

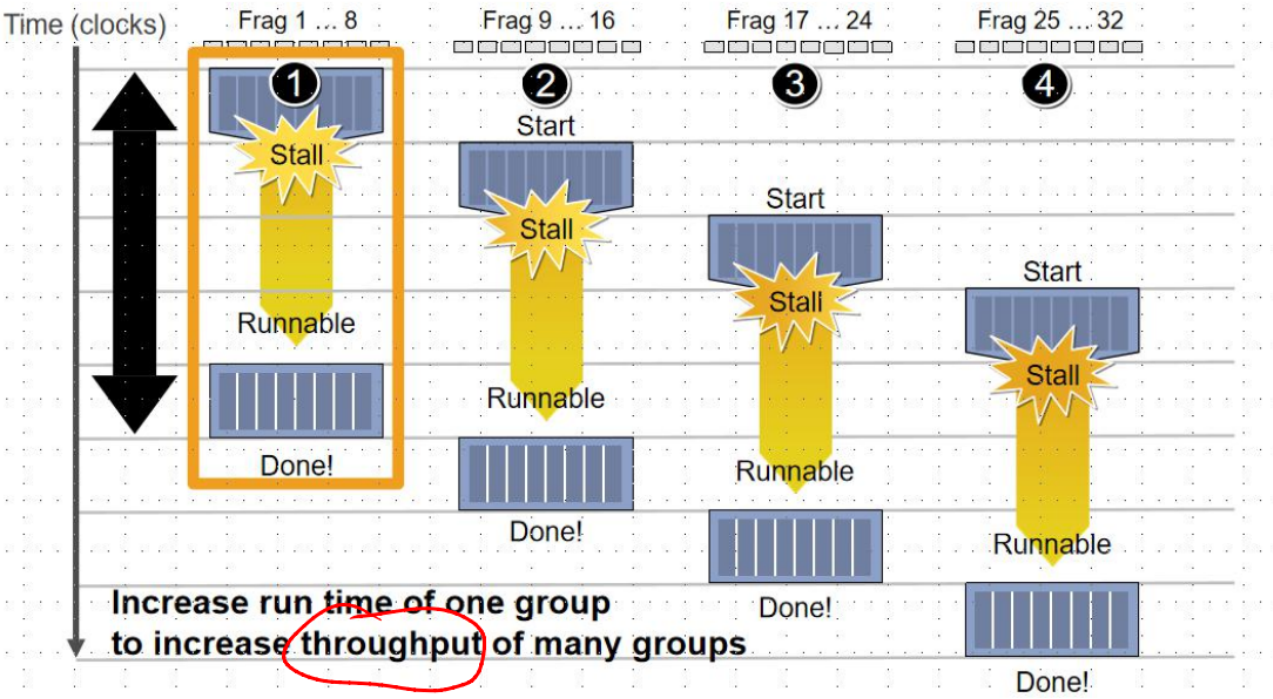
- Option 1: 显式的向量指令
  - x86 SSE, AVX, Intel Larrabee
- **Option 2: 标量指令，隐式HW向量化**
  - 由HW 确定ALU共享的指令流 (多少ALU共享一条指令流对软件隐藏)
  - NVIDIA GeForce ("SIMT" warps), ATI Radeon architectures ("wavefronts")

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

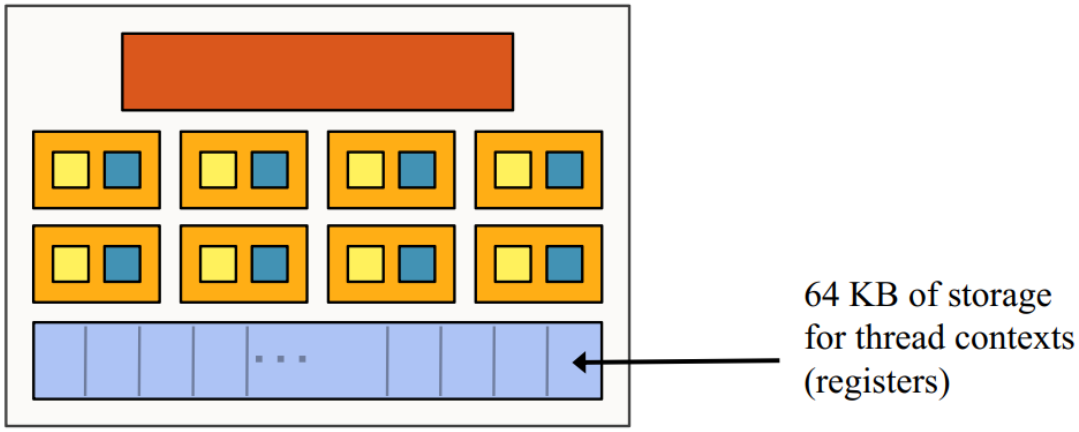
Idea #3:

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.)



NVIDIA GeForce GTX 285 “core”



- **32个线程一组(Warp）共享一条指令流**
- **每个core可同时交叉执行32个 warps**
- **即每个core可存储1024个线程的上下文**

Summary: three key ideas

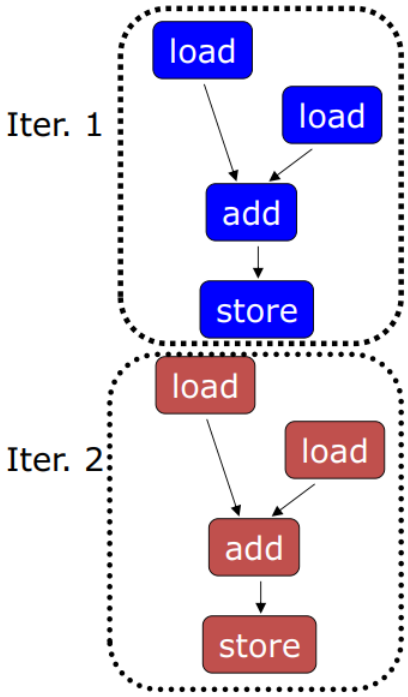
- **GPU使用大量”简单核心”（多核）并行执行**
- **核心中配置大量ALU部件形成SIMD处理模式**
  - Option 1: Explicit SIMD vector instruction
  - Option 2: Implicit sharing managed by hardware
- **通过交叉执行不同线程组处理不同数据片段避免指令流运行时的长延时（Stall）**
  - When one group stalls, work on another group

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)

Scalar Sequential Code



编程模型

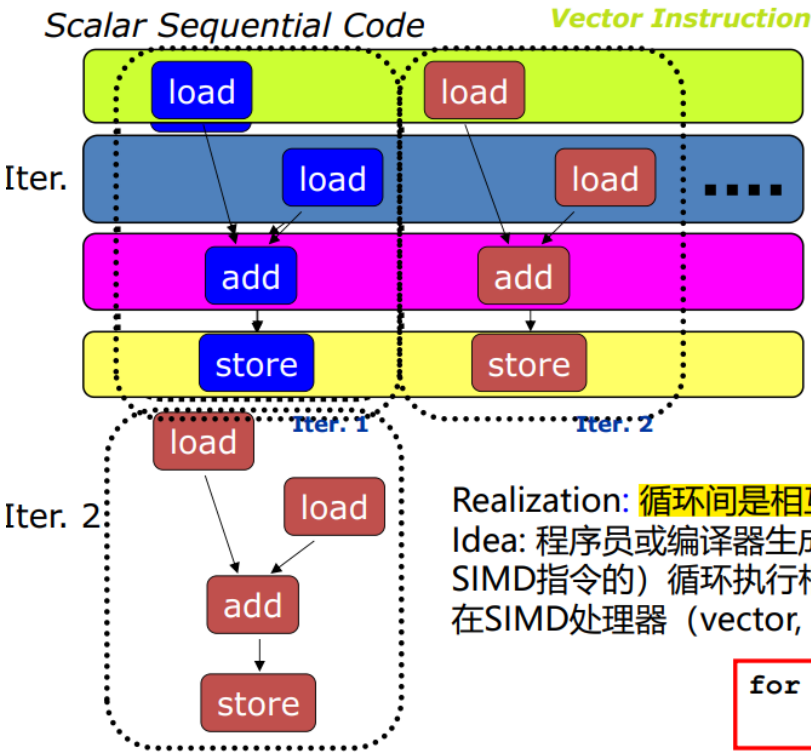
执行模型

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];
```

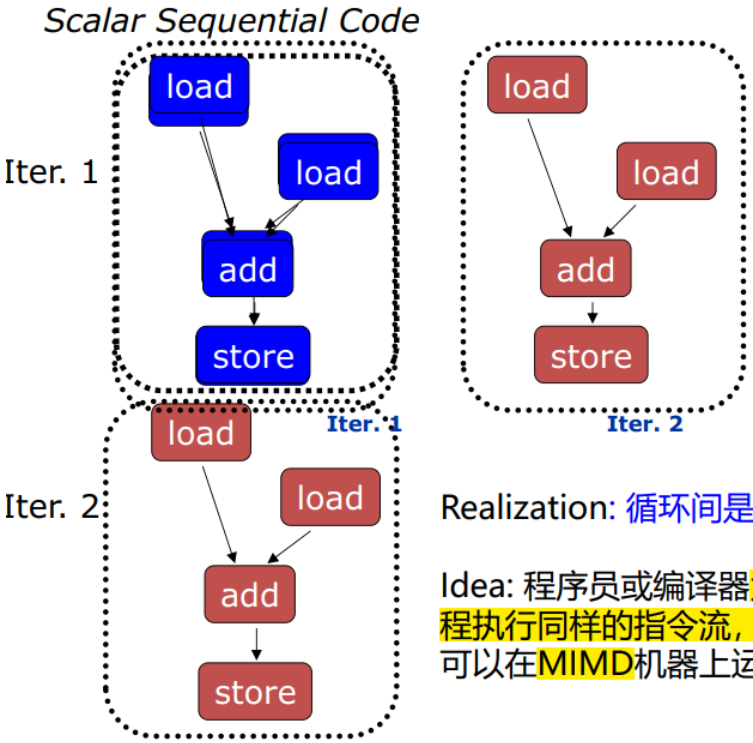
Prog. Model 2: Data Parallel (SIMD)



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




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

....

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
可以在MIMD机器上运行

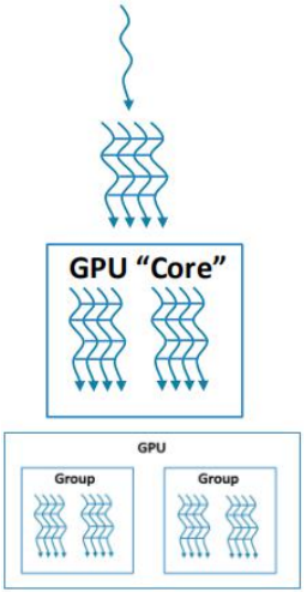
这种模式也称为: SPMD: Single Program Multiple Data

可以在SIMD机器上运行

可以在SIMT机器上运行

Single Instruction Multiple Thread

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

- **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）**

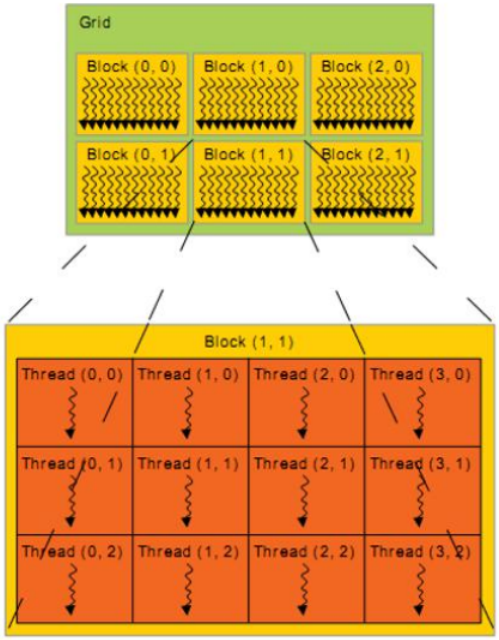


Figure 6 Grid of Thread Blocks

- **GPU 由硬件对线程进行管理**
  - 两级调度
    - Thread Block Scheduler
    - SIMD Thread Scheduler
  - Warp
    - SIMD线程
    - 线程调度的基本单位

- **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*)**

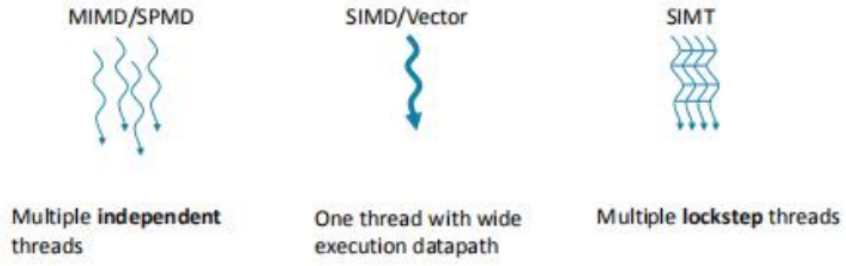
```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{  for (int i=0; i<n; i++)
       y[i] = a*x[i] + y[i]; }

// 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]; }
```

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

3　GPU分支处理

Conditional Branching

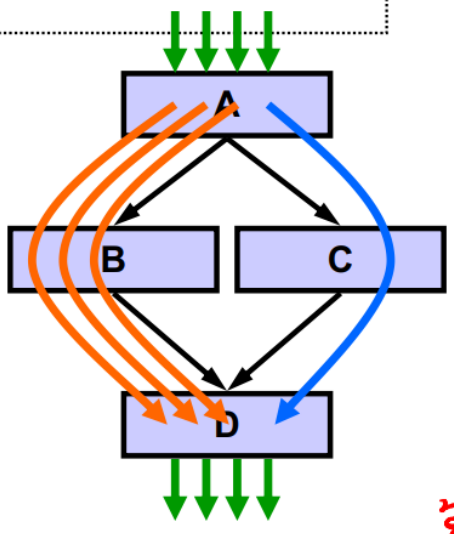
- **与向量结构类似, GPU 使用内部的屏蔽字(masks)**
- **还使用了 (SIMT-Stack)**
  - 分支同步堆栈
    - 保存分支的路径地址
    - 保存该路径的SIMD lane 屏蔽字(mask)
      - 即指示哪些车道可以提交结果
  - 指令标记(instruction markers)
    - 管理何时分支（divergence）到多个执行路径，何时路径汇合(converge)
- **PTX层**
  - CUDA线程的控制流由PTX分支指令(branch、call、return and exit）控制
  - 每个线程车道包含由程序员指定的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

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，那么创建一个屏蔽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

A	C	B	D
1	0	1	1
1	0	1	1
1	0	1	1
1	1	0	1

Time

先pop C，再pop B，再pop D

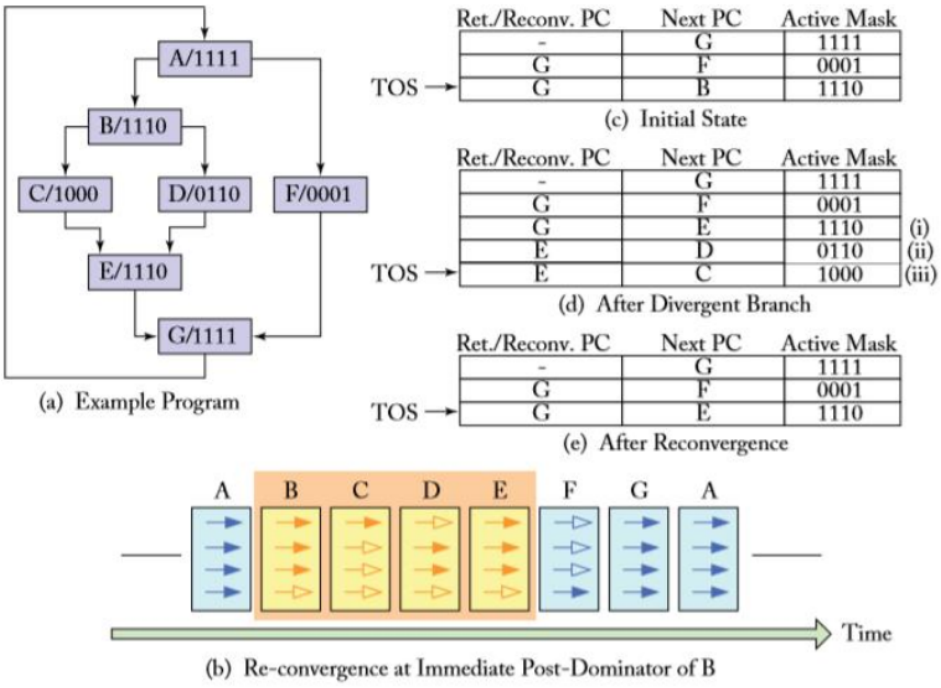
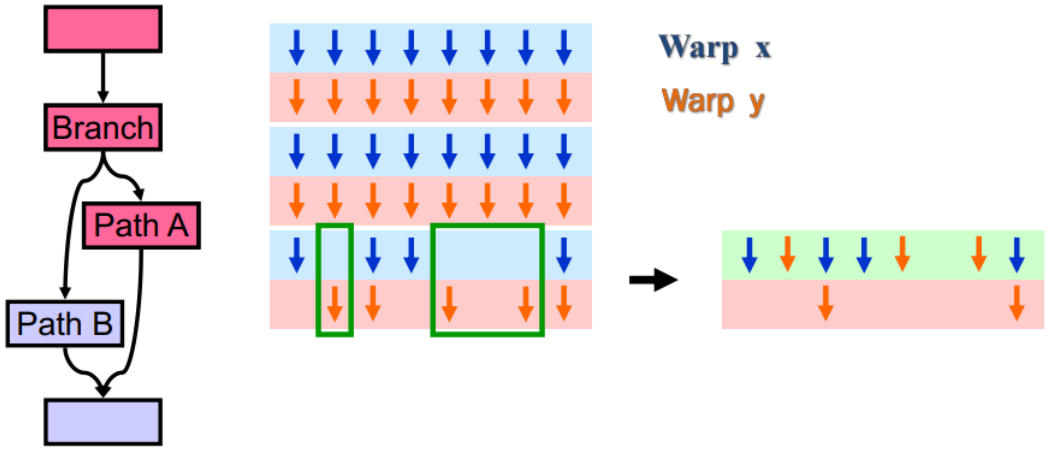


Figure 3.4: Example of SIMT stack operation (based on Figure 5 from Fung et al. [2007]).

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