数据级并行性(2)

图形处理单元

- GPU 和 CPU 没有共同的祖先
  - GPU 的前辈是图形加速器，做好图形处理是其存在的原因；
  - GPU 在走向主流计算的同时，仍然不能放弃自己在图形处理上的责任。
- 从技术术语和硬件特性上看，GPU 与矢量处理器和 SIMD 体系结构是十分不同的。

- 要点
  - 异构计算模式需要协作良好的计算调度：CPU 是主机，GPU 是设备
  - 编程环境具备多种类型的并行性：多线程，MIMD，SIMD 和 指令级
  - GPU 专用的类 C 编程语言——CUDA(Compute Unified Device Architecture)
  - 将所有类型的并行性统一为 CUDA 线程(CUDA Thread)
  - 编程模型为单指令多线程(single instruction multiple thread, SIMT)

GPU编程

- 线程块(Thread Block)：被限制在一起执行的线程
- 网格(Grid)：线程块的集合
- 多线程 SIMD 处理器：执行全部线程块的硬件

- 技术术语
  - dimGrid：网格维度（以线程块为单位）
  - dimBlock：线程块维度（以线程为单位）
  - blockIdx：线程块索引
  - threadIdx：线程索引（线程块中）
  - blockDim：线程数目（线程块中）
- GPU运行函数的扩展函数调用语法

```
function_name<<<dimGrid, dimBlock>>>(…parameter list…)
```

程序示例

- 传统 C 语言代码

```
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代码
  - 使用每个线程块256个线程的多线程SIMD处理器，启动 n 个线程，每个矢量元素一个线程。

```
__host__
int nblocks = (n + 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x, y)

__device__
void daxpy(int n, double a, double *x, double *y){
    int i = blockIdx.x * blockDim.x + threadIdx.x
    if (i < n)  y[i] = a*x[i] + y[i];
}
```

程序示例

- 在 C 语言代码中，循环不同迭代之间是相互独立的（循环承载相关性），可以直接转换为独立线程的并行代码。
- 在 CUDA 代码中，程序员通过说明网格维度（以线程块为单位）和每个 SIMD 处理器中线程数目（以线程为单位）来决定并行性。
  - 单个线程与每个元素对应
  - 当写结果到内存时，不需要线程之间的同步
- GPU 中的并行执行和线程管理由硬件处理，而不是由应用程序或者操作系统来完成。

GPU编程语言——CUDA

- CUDA 要求线程块能够以任何顺序独立地执行，且不同线程块之间不能直接通信，而是通过全局存储器交换数据。
- 关注性能的程序员需要牢记 CUDA 的硬件结构
- 对于 CUDA 来说，编程效率和性能之间的折中方案是让程序员能够通过代码来控制硬件。

主流计算术语与CUDA GPU术语

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

主流计算术语与CUDA GPU术语

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)

NVIDIA GPU 计算结构与矢量体系结构的对比

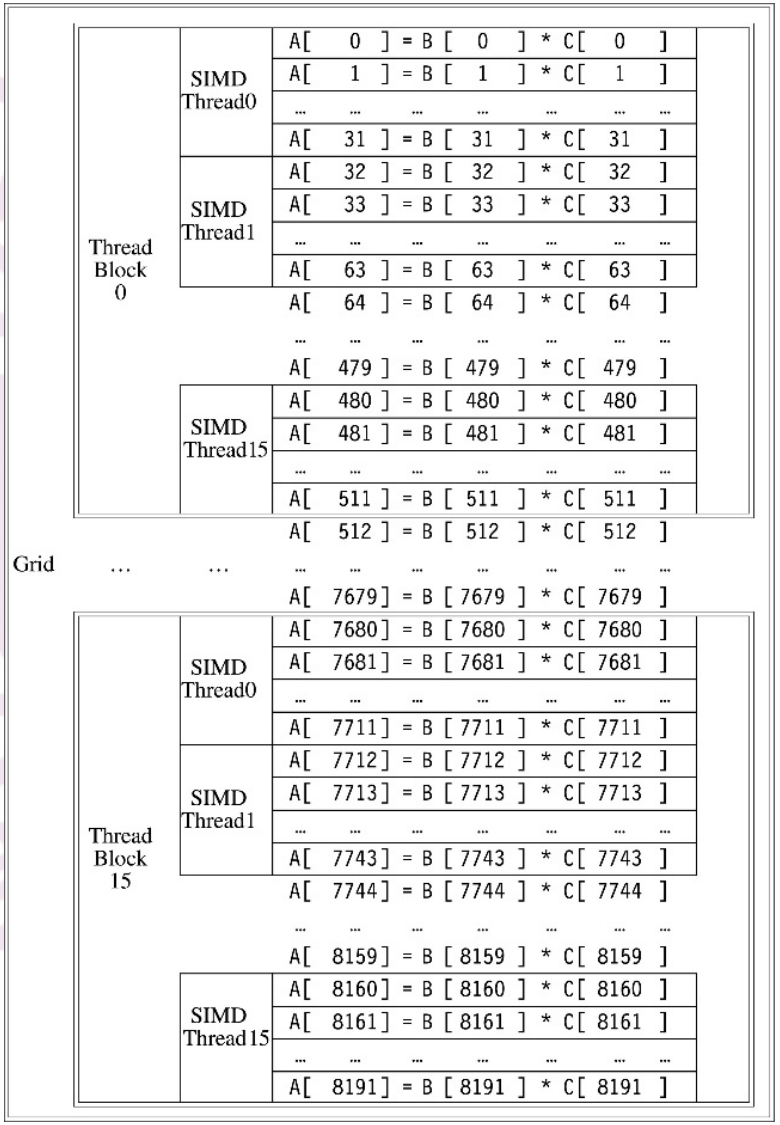
- 相似点
  - 很好地处理数据级并行性问题
  - 收集-分散(gather-scatter)数据传输
  - 掩码寄存器
  - 更大的寄存器文件

- 不同点
  - 没有标量处理器
  - 使用多线程来隐藏存储器延迟
  - 具有许多功能单元，而不是深度流水单元

NVIDIA GPU 计算结构

- 网格(Grid)：运行在 GPU 上的代码，由一组线程块(Thread Blocks)组成。
  - 网格 ⇔ 矢量化循环
  - 线程块 ⇔ 循环体

- 例：两个矢量 8192 个元素相乘
  - 网格负责 8192 个元素乘法操作
  - 网格由可管理的线程块组成
  - 每个线程块最多处理 512 个元素
  - 网格包含 $8192 \div 512 = 16$ 线程块
  - 一条 SIMD 指令同时计算 32 个元素
  - 线程块包括 $512 \div 32 = 16$ SIMD线程



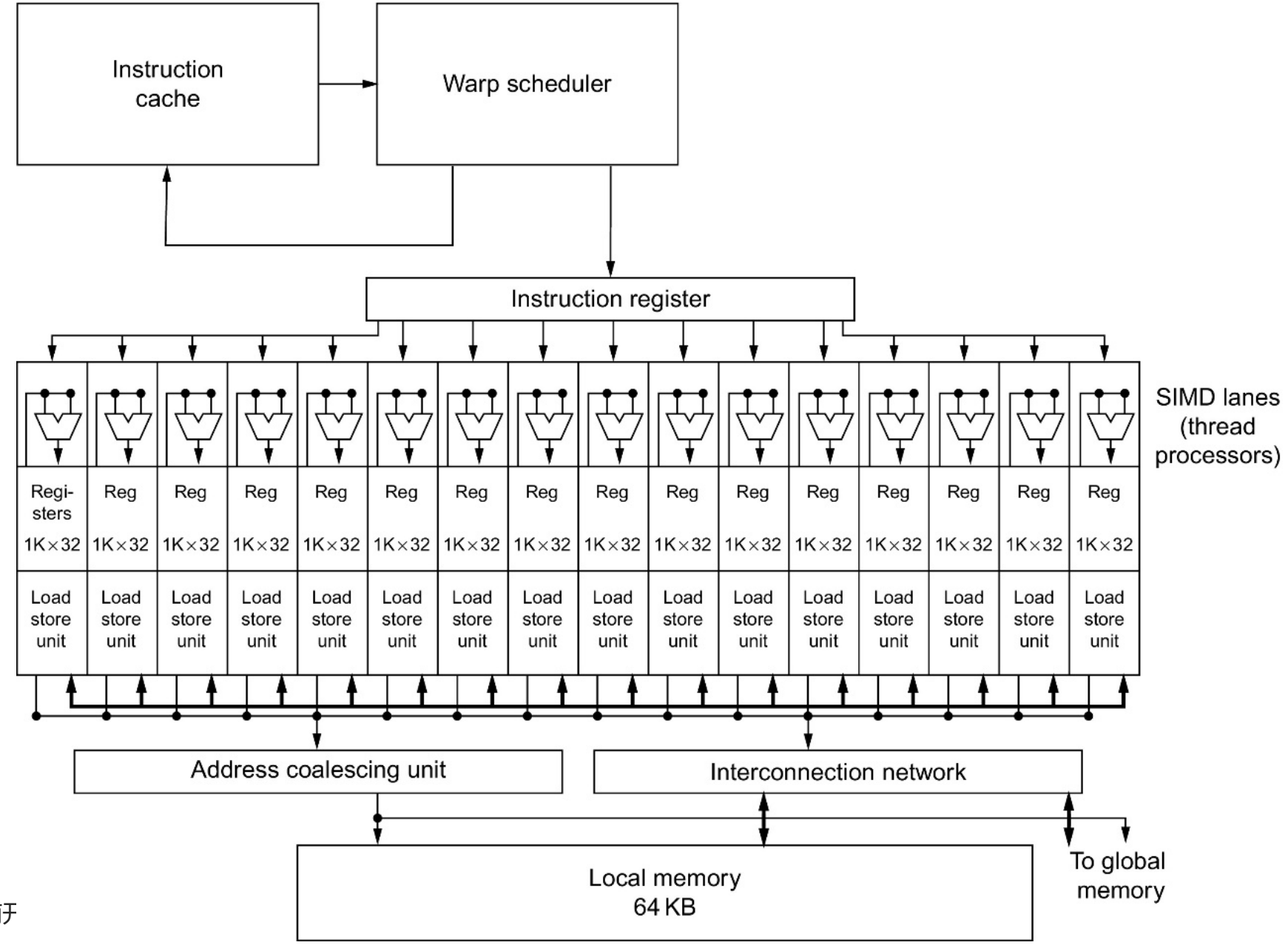
NVIDIA GPU 计算结构

- 线程块 (Thread Block) 通过线程块调度器 (Thread Block Scheduler) 分配给执行该代码的多线程 SIMD 处理器 (multithreaded SIMD processor)。
- 线程块调度器类似矢量体系结构中的控制处理器
  - 确定循环所需要的线程块数量
  - 不断将线程块分配给不同的多线程 SIMD 处理器，直到完成循环
- GPU 可以有一到几十个多线程 SIMD 处理器
  - Pascal P100系统具有 56 个多线程 SIMD 处理器
  - GPU 就是由多线程 SIMD 处理器组成的多处理器(multiprocessor)
- GPU 硬件创建、管理、调度和执行的对象是 SIMD 指令组成的线程(SIMD Thread)
  - 这是主流计算层面的线程，只包含 SIMD 指令

Pascal P100 GPU芯片框图

- 56 个多线程 SIMD 处理器（设计 60 个，剩余 4 个为备件）
- 每个多线程 SIMD 处理器拥有：1 个 L1 Cache和局部内存，32 个 L2 单元，内存总线宽度为 4096 条数据线
- 具有 4 个 HBM2 端口支持 $16GB$ 容量

多线程 SIMD 处理器简单框图



多线程 SIMD 处理器

- SIMD 通道(SIMD Lanes)：SIMD 处理器的并行功能单元
  - 与矢量体系结构中的矢量通道相似
- 32-元素宽的 SIMD 指令组成 SIMD 线程(SIMD Thread)或“warp”
  - 映射到 16 个物理通道（SIMD通道）
  - 每条 SIMD 指令（线程）需要 2 个时钟周期
- 每个 SIMD 处理器上最多调度 32 个warp
  - 每个 warp 拥有自己的 PC（程序计数器）
  - 线程调度程序使用计分板来调度 warp 中准备好运行的 SIMD 指令
  - 根据定义，warp之间不存在数据相关性
  - 将 warp 调度到流水线中，隐藏存储器延迟

多线程 SIMD 处理器的两级（硬件）调度器

- 线程块调度器：将线程块（矢量化循环体）分配给多线程SIMD 处理器
- SIMD线程调度器：在多线程 SIMD 处理器内，调度 SIMD 指令线程执行
  - 假设存在足够多的 SIMD 指令线程来隐藏 DRAM 延迟和提高多线程 SIMD 处理器的利用率

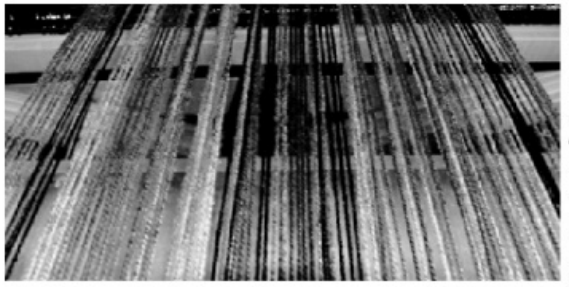
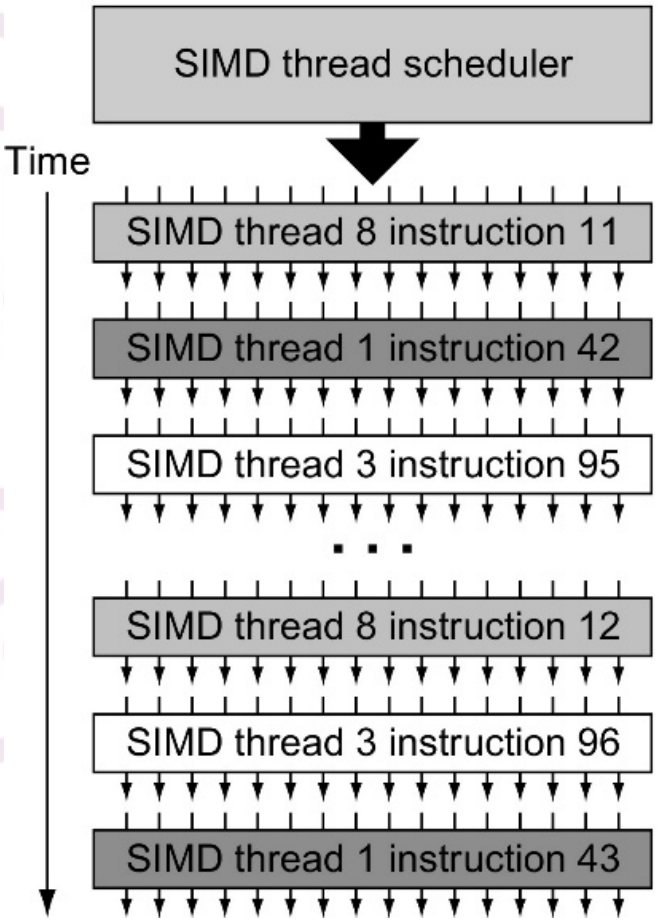


Photo: Judy Schoonmaker



多线程 SIMD 处理器

- 寄存器
  - SIMD 处理器包含 $32K$~$64K$ 个 32 比特寄存器
  - 逻辑上划分给每个 SIMD 通道
  - 每个 SIMD 线程不超过 256 个矢量寄存器，如：
    - 256 个矢量寄存器，每个矢量寄存器 32 个元素，每个元素 32 比特；或
    - 128 个矢量寄存器，每个矢量寄存器 32 个元素，每个元素 64 比特。

- 寄存器数目和最大线程数目之间的权衡
  - 每个线程使用较少的寄存器，意味着可以有更多的线程；
  - 每个线程使用较多的寄存器，意味着可以有较少的线程。
- CUDA 线程只是 SIMD 指令线程的一个纵切面(vertical cut)，对应着 SIMD 通道执行的一个元素。

NVIDIA指令集体系结构

- ISA 是硬件指令集的抽象
  - PTX(Parallel Thread Execution) 给编译器提供指令集（硬件指令集对程序员透明）
    - `opcode.type d,a,b,c;`
  - 使用虚拟寄存器
  - 在软件中完成机器代码转换
  - 例：

```
shl.s32         R8,blockIdx,9    # Thread Block ID*Block size
                                 # (512 or 2^8)
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])
```

基本 PTX GPU 线程指令

Group	Instruction	Example	Meaning	Comments
Arithmetic	arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
	add.type	add.f32 d, a, b	d = a + b;	
	sub.type	sub.f32 d, a, b	d = a - b;	
	mul.type	mul.f32 d, a, b	d = a * b;	
	mad.type	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.type	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.type	rem.u32 d, a, b	d = a % b;	integer remainder
	abs.type	abs.f32 d, a	d = |a|;	
	neg.type	neg.f32 d, a	d = 0 - a;	
	min.type	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.type	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	selp.cmp.type	selp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.type	mov.b32 d, a	d = a;	move
	selp.type	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype
Special function	special .type = .f32 (some .f64)			
	rcp.type	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.type	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.type	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.type	sin.f32 d, a	d = sin(a);	sine
	cos.type	cos.f32 d, a	d = cos(a);	cosine
	lg2.type	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.type	ex2.f32 d, a	d = 2 ** a;	binary exponential

基本 PTX GPU 线程指令（续）

Logical	logic.type = .pred, .b32, .b64			
	and.type	and.b32 d, a, b	d = a & b;	
	or.type	or.b32 d, a, b	d = a | b;	
	xor.type	xor.b32 d, a, b	d = a ^ b;	
	not.type	not.b32 d, a, b	d = ~a;	one's complement
	cnot.type	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.type	shl.b32 d, a, b	d = a << b;	shift left
	shr.type	shr.s32 d, a, b	d = a >> b;	shift right
Memory access	memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory space
	st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory space
	tex.nd.dtyp.btype	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup
	atom.spc.op.type	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b, c	atomic { d = *a; *a = op(*a, b); }	atomic read-modify-write operation
	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32			
Control flow	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

条件分支

- 与矢量体系结构一样，GPU 分支硬件也使用内部掩码
- 使用
  - 分支同步堆栈
    - 内容由每个 SIMD 通道掩码组成
  - 当分支指令分叉为多个执行路径时，由指令标记进行管理
    - 将分叉分支压栈
  - ...当路径交汇时
    - 充当栅栏
    - 弹栈
- 由程序员指定每个线程通道 1-比特谓词寄存器
  - GPU 设置谓词指令（`setp`）计算 IF 语句的条件部分

例

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

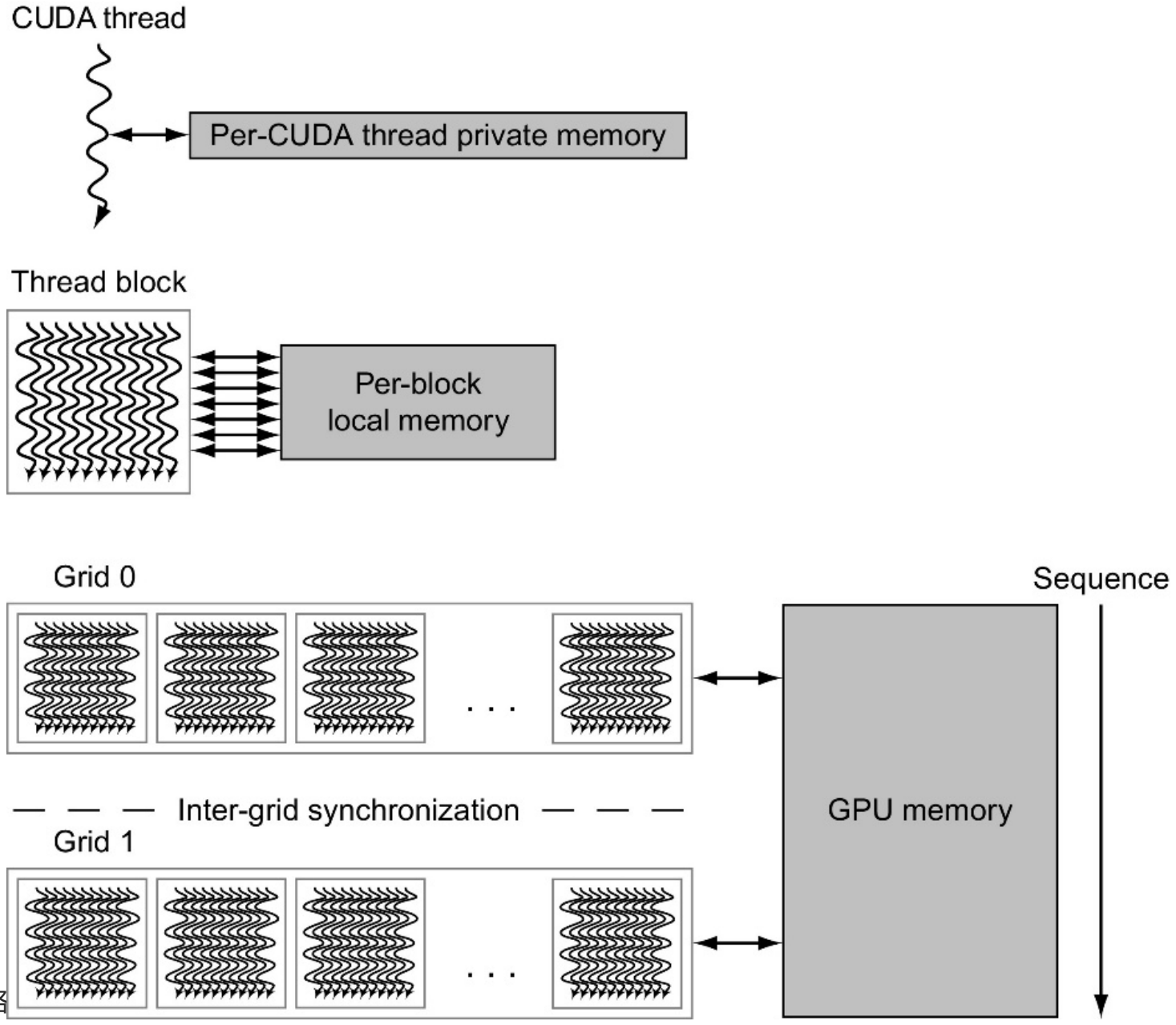
```
        ld.global.f64   RD0,[X+R8]    # RD0 = X[i]
        setp.neq.s32    P1,RD0,#0     # P1 is predicate reg. 1
        @!P1,bra        ELSE1,*Push   # Push old mask, set new
                                      # mask bits, if P1 false,
                                      # go to ELSE1
        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 inst.>,*Pop             # pop to restore old mask
```

条件分支

- 在一个时钟周期内，SIMD 通道执行 PTX 指令操作或者空闲
- 两条 SIMD 通道不能同时执行不同的指令
- 每个 CUDA 线程不是执行与线程块中每个其它线程相同的指令，就是空闲。

- 条件执行
  - GPU 在硬件运行时处理
  - 矢量体系结构在编译时处理

NVIDIA GPU 存储器结构



NVIDIA GPU 存储器结构

- 每个 SIMD 通道都有片外 DRAM 的私有部分
  - 私有存储器(private memory)
  - 用途：堆栈帧，溢出寄存器和（不适合放在寄存器中的）私有变量
  - SIMD 通道不共享私有存储器
  - GPU 在 L1 和 L2 缓存器中缓存这个私有存储器
- 每个多线程 SIMD 处理器也有局部存储器
  - 低延迟，高带宽，小型暂存存储器—— $48\ KiB$
  - 由块内的 SIMD 通道 / 线程共享
  - 多线程 SIMD 处理器之间不共享
  - 创建线程块时动态申请，线程块退出时释放
- 由 SIMD 处理器共享的存储器是 GPU 存储器
  - 主机可以读 / 写GPU存储器

Pascal 体系结构创新

- 每个 SIMD 处理器拥有
  - $2\sim4$ 个 SIMD 线程调度器，两个指令调度单元。
  - 16个 SIMD 通道（SIMD宽度$=32$，定时$=2$ 个周期），16 个加载-存储单元，4 个特殊功能单元。
  - 每两个时钟周期调度两个SIMD指令线程
- 快速的单精度、双精度和半精度浮点数运算
- 高带宽（$732\ GB/s$）存储器（HBM2）
  - HBM: High Bandwidth Memory
- 多个 GPU 间互联的 NVLink（每个方向 $20GB/s$）
- 统一的虚拟内存和分页支持

Pascal 多线程 SIMD 处理器

GPU 与矢量体系结构的对应

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

GPU 与矢量体系结构的对应

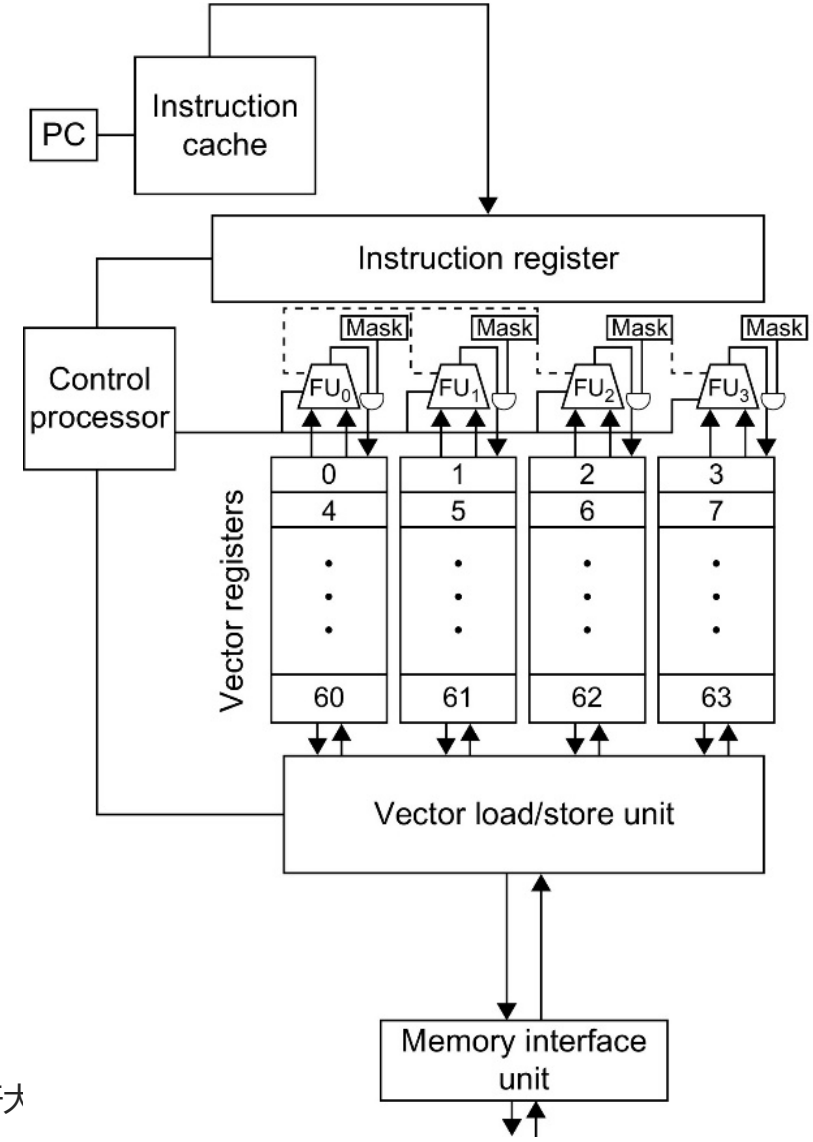
Processing and memory hardware	Vector Processor	Multithreaded SIMD Processor	These are similar, but SIMD Processors tend to have many lanes, taking a few clock cycles per lane to complete a vector, while vector architectures have few lanes and take many cycles to complete a vector. They are also multithreaded 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

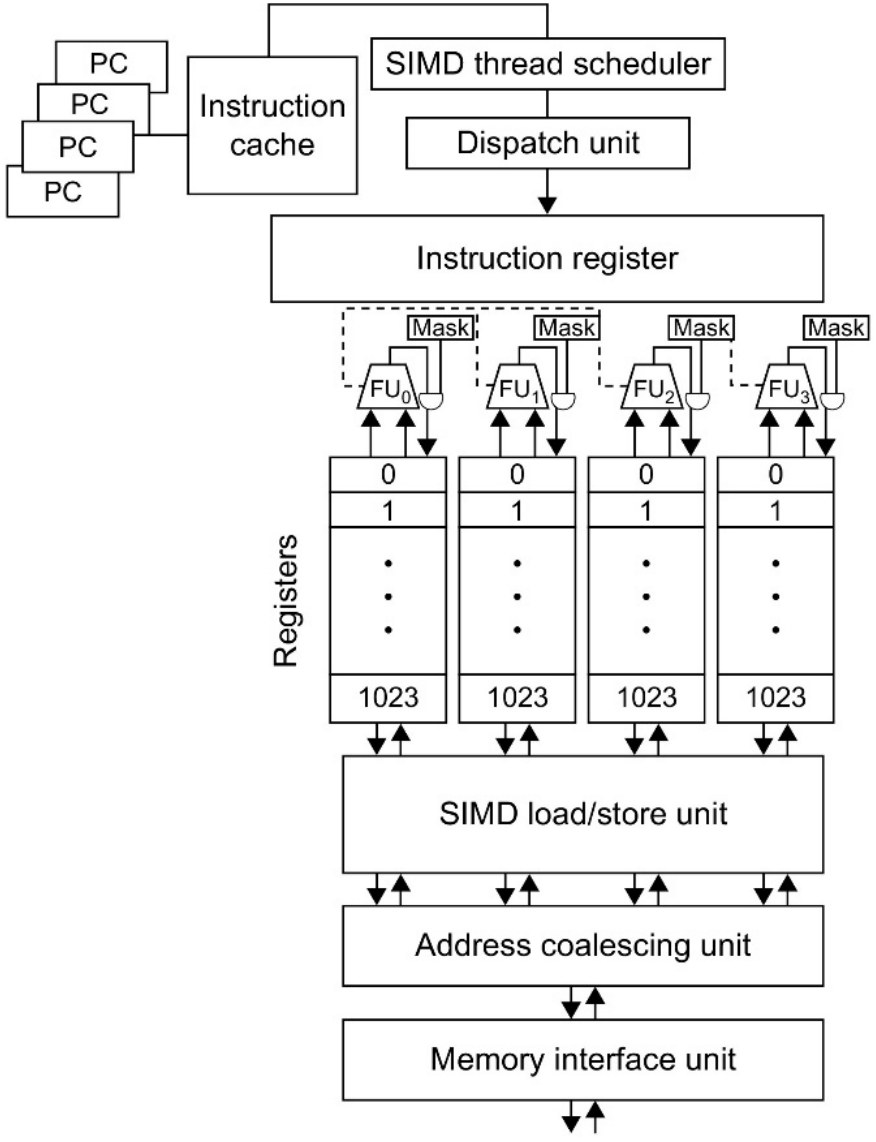
GPU 与矢量体系结构的对应

- SIMD 处理器类似于矢量处理器，GPU中多个 SIMD 处理器做为独立的 MIMD 核。
- 最大不同：多线程是GPU的基础，绝大多数矢量处理器没有。
- 寄存器
  - RV64V 寄存器文件保存整个矢量，而 GPU 将矢量分布到 SIMD 通道寄存器中；
  - RV64V 有 32 个矢量寄存器，每个包含 32 个元素（$1K$），而 GPU 有 256 个寄存器，每个包含 32 个元素（$8K$）；
  - RV64V 有 2~8 个矢量长度为 32 的通道，定时是 4~16 个周期，而 SIMD 处理器定时是 2~4 个时钟周期；
  - GPU 矢量化循环是网格
  - 所有GPU加载都是收集指令，所有GPU存储都是分散指令。

GPU 与矢量体系结构

- 四通道矢量处理器和四SIMD通道多线程SIMD处理器





GPU 与 SIMD 体系结构

- GPU 有多个 SIMD 通道
- GPU 硬件支持更多的线程
- 二者的双精度和单精度性能之比是 2: 1
- 二者都是 64 位地址，但 GPU 有更小的存储器
- SIMD 体系结构不支持收集-分散

循环级并行性

- 定义：循环承载相关性(loop carried dependence)
  - 循环不同迭代之间的数据相关性。
- 例1：非循环承载相关（相关存在一次迭代中）

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

- 例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 使用前次迭代中 S1 计算的数值（循环承载相关，强迫按序执行）
  - S2 使用前次迭代中 S2 计算的数值（循环承载相关，强迫按序执行）
  - S2 使用本次迭代中 S1 计算的数值（非循环承载相关）

循环级并行性

- 例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使用前次迭代中S2计算的数值，存在循环承载相关；
  - 但这个相关不是环状的，即：两条语句都不与自己相关；
  - 无环意味着语句之间相关性是偏序的(partial ordering)

- 两个重要观察点
  - S2 不依赖于 S1，故不成环，即：交换两条语句不影响 S2 的执行；
  - 在循环第一次迭代中，S2 依赖初始化循环时的 B[0] 值。

循环级并行性

- 例3（续）：替换前面的循环

```
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];
```

循环级并行性

- 分析从寻找所有循环承载相关性开始，这种信息是不准确的。
- 例4：

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

  - 对 A 的第二次访问不需要转换为 load 语句，因为知道上一条指令计算和存储的数值。
  - 可以优化为直接访问寄存器，但需要保证两次访问相同的存储地址，且两次访问中间没有对同一存储地址的访问。

循环级并行性

- 循环承载相关性通常是重现的(recurrence)。
- 重现：发生在使用先前迭代中该变量的取值定义该变量时
- 例5：

```
for(i=0;i<100;i=i+1){
     Y[i]=Y[i-1]+Y[i];
}
```

- 检测重现的重要性
  - 某些体系结构对执行重现语句具有特殊的支持
  - 在 ILP 层面，仍然有可能利用相当数量的并行性

发现相关性

- 相关性分析的复杂性来源于编程语言中数组和指针的使用
  - 因为标量变量访问是使用名称的
- 所有相关性分析算法均基于数组索引是仿射的前提
  - $a*i+b$ （i 是循环索引变量）
- 假如：
  - 存储到 $a\times i+b$，然后从 $c\times i+d$ 加载
  - i 从 m 执行到 n
  - 如果下列条件满足，则存在相关性：
    - 已知 j,k，使得 $m\le j\le n, m\le k\le n$
    - 存储到 $a\times j+b$，从 $c\times k+d$ 加载，且 $a*j+b=c*k+d$
- 一般情况下，编译时很难确定相关性（系数不确定）
- 稀疏矩阵访问是非仿射访问

发现相关性

- 最大公约数(greatest common divisor, GCD)测试：如果存在着循环承载相关性，则 $GCD(c,a)\ mod\ (d-b)=0$。
  - 这是不存在相关性的简单且充分的测试方法
- 例6：

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

  - $a=2, b=3, c=2$ 和 $d=0$；
  - $GCD(a,c)=2, d-b=-3$，不能整除，不存在相关性。
- 注意：非必要条件，故，能够整除不能保证存在相关性。

- 确定是否存在相关性是个 NP-完全(Non-deterministic Polynomial Complete) 问题（数学难题之一）

发现相关性

- 编译器辨别名称相关性，且在编译时通过重命名方法消除名称相关性。
- 例7：

```
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 */
}
```

  - S1 与 S3 和 S1 与 S4 因为 `Y[i]` 存在着真相关，非循环承载；
  - S1 与 S2 因为 `X[i]` 存在反相关；
  - S3 与 S4 因为 `Y[i]` 存在反相关；
  - S1 与 S4 因为 `Y[i]` 存在输出相关。

发现相关性

- 例7（续）：消除假相关

```
for(i=0;i<100;i=i+1){
    T[i]=X[i]/c; // Y renamed to T to remove output-dep.
    X1[i]=X[i]+c;// X renamed to X1 to remove anti-dep.
    Z[i]=T[i]+c; // Y renamed to T to remove anti-dep.
    Y[i]=c-T[i];
  }
```

  - 循环后，变量 X 重命名为 X1。

- 相关性分析是利用并行性的关键技术
  - 矢量计算机，SIMD 计算机或多处理器的编译器深深地依赖该技术
  - 主要缺点是只可以在有限情形下使用，如使用指针而不是索引来访问数组时，分析就困难得多。

归约

- 相关计算最重要的形式之一就是重现，如点积计算。

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

  - 由于变量 sum 具有循环承载相关性，故不能并行计算

- 将循环变换为一个完全并行和其它部分并行的

```
for(i=9999;i>=0;i=i-1)
    sum[i] = x[i] * y[i]; // scalar expansion
for(i=9999;i>=0;i=i-1)
    finalsum = finalsum + sum[i] // reduce step
```

- 归约(reduction)
  - 线性代数中常见
  - 在矢量和SIMD体系结构中可以通过特殊硬件处理

归约

- 例8：在 p 颗处理器上执行

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

  - 每颗处理器上计算 1000 个元素之和，完全并行；
  - 然后，简单标量循环完成 10 项求和。

谬误和陷阱

- GPU受制于协处理器
  - PTX存在于编译器和硬件之间，赋予GPU体系结构更大的灵活性
- 专注于矢量体系结构中的峰值性能而忽视启动开销
  - 启动开销需要一定的矢量长度才能抵消，以表现优良的性能
- 增加矢量性能，但标量性能没有相应提高
  - 这个不平衡问题存在于早期矢量处理器，甚至目前的处理器中；
  - 良好的标量性能可以减少开销代价且减少受 Amdahl 定律的影响
- 在不提供存储器带宽的情形下可以获得良好的矢量性能
  - 存储器带宽对所有 SIMD 体系结构都是非常重要的
  - 如：DAXPY 代码需要浮点数操作的1.5倍存储器访问
- 在GPU上，如果没有足够内存性能只需添加更多线程
  - 虽然 CUDA 线程可以隐藏主存延迟；但，如果内存访问在CUDA线程间分散或不连贯，则内存对访问的响应将逐渐变慢。

诚信 创新 实践