数据级并行性(1)

引言

- SIMD 体系结构利用数据级并行性(data-level parallelism)
  - 面向矩阵的科学计算
  - 面向多媒体的图像和声音处理器
  - 机器学习算法

- SIMD 比 MIMD 更节能
  - 与 MIMD 相比，SIMD 的单条指令可以同时对许多数据进行操作
  - 个人移动设备使 SIMD 更具吸引力

- 在 SIMD 架构中，程序员从代码顺序执行的角度来思考问题，却获得代码并行执行的加速比。

SIMD 并行性

- SIMD 的三种变型
  - 矢量体系结构
  - SIMD 扩展指令集
  - 图形处理器单元(GPUs)

- Intel x86 处理器
  - 预计每年每个芯片增加两个处理器核
  - SIMD 的宽度每四年翻一番
  - SIMD 的潜在加速比是 MIMD 的两倍

SIMD并行性

- 矢量体系结构
  - 比另外两种变型早 30 年提出
  - 将许多数据操作扩展为流水执行，更容易理解和编译
  - 对微处理器来说，代价昂贵
    - 需要更多的晶体管资源
    - 需要足够的 DRAM 带宽

- SIMD 扩展指令集
  - 广泛存在于支持多媒体应用的指令集体系结构中
  - x86处理器：
    - 1996年，MMX (multimedia extensions)
    - 十年间，几个版本的 SSE (streaming SIMD extensions)
    - 今天，AVX (advanced vector extensions)

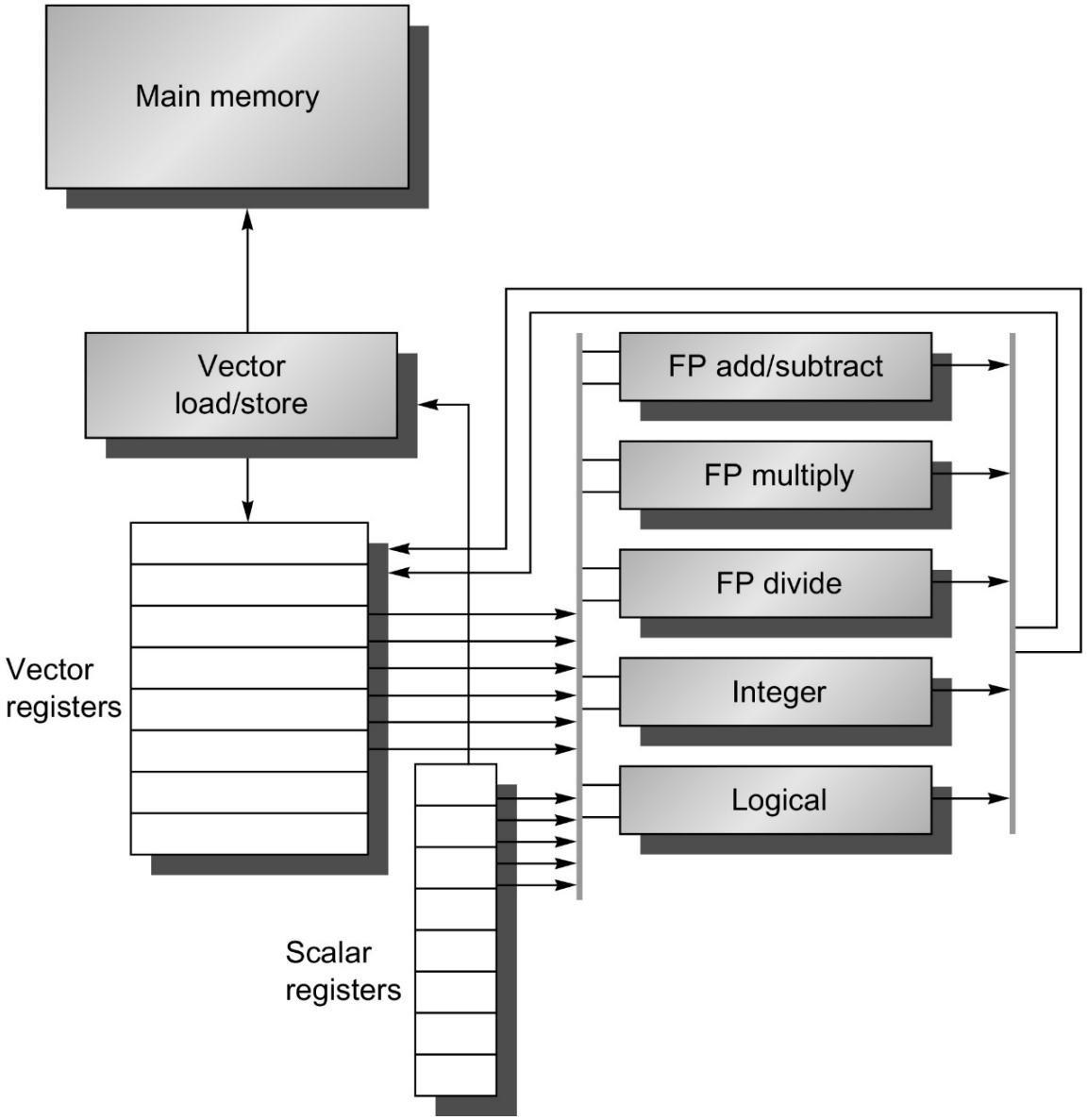
SIMD并行性

- 图形处理单元
  - 与矢量体系结构共享特征，但具有自己的特性
  - 独特的生态环境：系统处理器与系统内存 + GPU 与图形内存

矢量体系结构

- 基本思路
  - 将一组分散在内存中的数据元素读入“矢量寄存器”
  - 对这些矢量寄存器进行操作
  - 把操作结果放回内存中
- 对矢量数据执行一条指令，将获得对独立数据元素进行的多条寄存器-寄存器操作。
- 大型寄存器文件做为编译器“可控的”缓存器
  - 隐藏内存延迟：每次矢量加载或存储将导致一次内存延迟，而非每个元素一次，由此均摊了延迟时间。
  - 降低读写次数：利用内存带宽
- 矢量体系结构在提供良好性能的同时，没有付出乱序超标量处理器在能耗和设计复杂性方面的代价。

矢量体系结构 RV64V (RISC-V 64V)



RV64V 架构

- 以 40 年前的 Cray-1（第一代超级计算机）为基础
- 32 个 64 比特（8 字节）矢量寄存器
  - 寄存器文件有 16 个读端口和 8 个写端口
- 矢量功能单元
  - 全流水
  - 检测数据和控制相关
- 矢量加载-存储单元
  - 全流水
  - 在初始延迟之后每个时钟周期一个字
- 标量寄存器
  - 31 个通用寄存器
  - 32 个浮点数寄存器

指令集

Mnemonic	Name	Description
vadd	ADD	Add elements of V[rs1] and V[rs2], then put each result in V[rd]
vsub	SUBtract	Subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
vmul	MULtiply	Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
vdiv	DIVide	Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
vrem	REMainder	Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd]
vsqrt	SQuare RooT	Take square root of elements of V[rs1], then put each result in V[rd]
vsll	Shift Left	Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
vsrl	Shift Right	Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
vsra	Shift Right Arithmetic	Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd]
vxor	XOR	Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vor	OR	Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vand	AND	Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
vsgnj	SiGN source	Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd]
vsgnjn	Negative SiGN source	Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd]
vsgnjx	Xor SiGN source	Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd]

指令集（续）

vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vpeq	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpne	Compare !=	Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vplt	Compare <	Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpxor	Predicate XOR	Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpor	Predicate OR	Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpand	Predicate AND	Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
setvl	Set Vector Length	Set vl and the destination register to the smaller of mvl and the source regsiter

特色指令

- 跨步装载和存储
  - Strided Load：$V[rd] \Leftarrow Memory(R[rs1] + i \times R[rs2])$
  - Strided Store： $V[rd] \Rightarrow Memory(R[rs1] + i \times R[rs2])$
- 索引装载（收集, Gather)
  - Indexed Load：$V[rs1] \Leftarrow Memory(R[rs2] + V[rs2])$
- 索引存储（分散， Scatter)
  - Indexed Store：$V[rs1] \Rightarrow Memory(R[rs2] + V[rs2])$

- 名词解释
  - 谓词(predicate)：在计算机语言环境下，是指条件表达式的求值过程。

数据类型和长度

- 多种数据大小
  - 矢量寄存器：32个64-比特元素 ⇔ 128个16-比特元素 ⇔ 256个8-比特元素
  - 适合多媒体应用和科学计算

- 支持的数据类型和长度

Integer	8, 16, 32, and 64 bits	Floating point	16, 32, and 64 bits

动态寄存器类型

- 创新点：动态寄存器类型(dynamic register typing)
  - 数据类型与大小和每个矢量寄存器相关联，而不是与指令相关联；
  - 指令执行之前，程序配置矢量寄存器用于特定的数据类型和宽度。

- 动态寄存器类型将所有矢量内存分配给启用的(enabled)矢量寄存器做为长矢量使用。
  - 例：假设矢量内存为 1024 字节，如果启用 4 个矢量寄存器，类型是 64-比特浮点数，则处理器给每个矢量寄存器 $1024 \div 4 = 256$ 个字节，可以存储 $256 \div (64 \div 8) = 32$ 个元素。
  - 最大矢量长度(maximum vector length, mvl)：每个矢量寄存器可以容纳的元素个数。

矢量加载和存储指令

- 指令 `vld/vst`：从地址中加载或存储整个矢量
  - 一个操作数是矢量寄存器
  - 另一个操作数是通用寄存器，保存矢量在存储器中的起始地址

DAXPY 的 RISC-V 代码

- 缩写
  - SAXPY: single precision a × X plus Y
  - DAXPY: double precision a × X plus Y
- RISC-V 代码

```
        fld     f0,a              # load scalar a
        addi    x28,x5,#256       # last address to load
Loop:   fld     f1,0(x5)          # load X[i]
        fmul.d  f1,f1,f0          # a × X[i]
        fld     f2,0(x6)          # load Y[i]
        fadd.d  f2,f2,f1          # a × X[i] + Y[i]
        fsd     f2,0(x6)          # store into Y[i]
        addi    x5,x5,#8          # increment index to X
        addi    x6,x6,#8          # increment index to Y
        bne     x28,x5,Loop       # check if done
```

  - 执行指令数目： $2 + 32 \times 8 = 258$

DAXPY 的 RV64V 代码

- RV64V代码

```
vsetdcfg   4*FP64       # Enable 4 DP FP vregs
fld        f0,a         # Load scaler a
vld        v0,x5        # Load vector X
vmul       v1,v0,f0     # Vector-scalar mult
vld        v2,x6        # Load vector Y
vadd       v3,v1,v2     # Vector-vector add
vst        v3,x6        # Store the sum
vdisable                # Disable vector regs
```

  - 执行指令数目：8
  - 每条矢量指令只需在第一个元素时停顿一次，而非每个元素停顿一次。

若干概念

- 可矢量化(vectorizable)
  - 定义：为代码序列生成的矢量指令，如果其执行过程中绝大多数执行时间处于矢量运行模式，则称代码序列可矢量化。
  - 当循环程序相邻迭代之间不存在相关性时，循环代码可矢量化。
- 链接(chaining)
  - 定义：同时执行相关数据的矢量指令
  - 一旦源矢量操作数的一个元素有效时，可以立即启动矢量操作。
- 执行时间
  - 是操作数矢量长度、结构危害和数据相关的函数

若干概念

- 道(lane)：在矢量功能单元中配置的多个并行流水线。
- 引发速率(initiation rate)：矢量单元消耗操作数的速度
  - RV64V功能单元每个时钟周期消耗一个元素
  - 执行周期数近似为矢量长度（忽略处理器开销）
- 编队(convoy)：可以同时开始执行的一组矢量指令
  - 指令之间不存在结构危害和 RAW 相关（可以通过链接技术解决）
- 定时(chime，钟声，铃声)：执行一个指令编队所需要的时间
  - 包含 m 个编队的矢量指令序列需要执行 m 个定时
  - 当矢量长度为 n 时，RV64V 需要近似运行 $m \times n$ 个时钟周期

编队(convoy)

- 代码序列

```
vld       v0,x5       # Load vector X
vmul      v1,v0,f0    # Vector-scalar multiply
vld       v2,x6       # Load vector Y
vadd      v3,v1,v2    # Vector-vector add
vst       v3,x6       # Store the sum
```

- 编队

```
1     vld     vmul
2     vld     vadd
3     vst
```

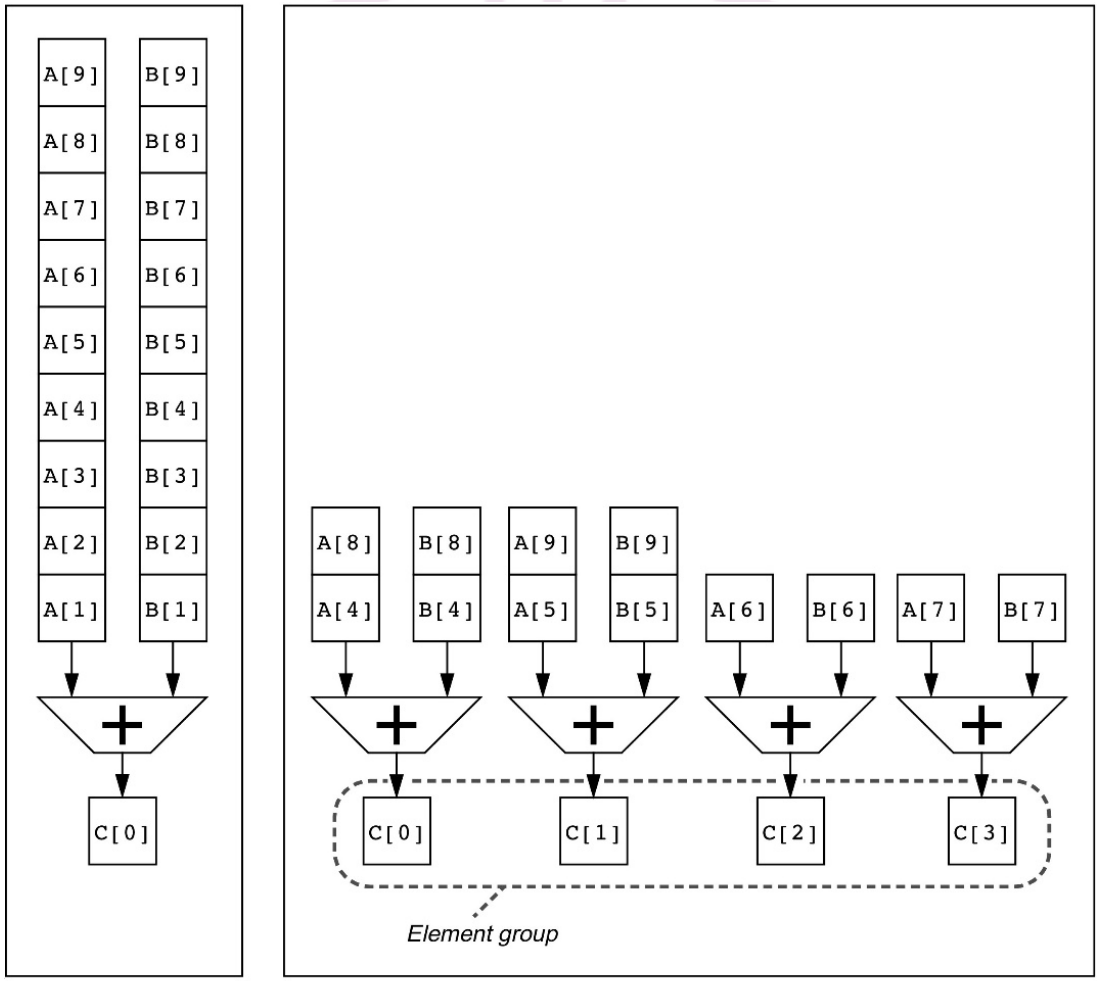
  - 三个编组需三个定时
  - 每个结果两个浮点数运算，每个浮点运算需要 $3 \div 2 = 1.5$ 个时钟周期
  - 包含 32 个分量的矢量需要 $32 \times 3 = 96$ 个时钟周期

发出限制

- 启动时间(start-up time)
  - 等待矢量功能单元满负荷运行的延迟时钟周期数
  - 所有功能单元全流水(full pipeline)
    - 浮点数加法 ⇒ 6 时钟周期
    - 浮点数乘法 ⇒ 7 时钟周期
    - 浮点数除法 ⇒ 20 时钟周期
    - 矢量加载 ⇒ 12 时钟周期
- 改进方法
  - 每个时钟周期处理超过 1 个矢量元素
  - 非 64 比特宽矢量
  - 矢量代码中的取指令声明
  - 优化存储系统来支持矢量处理器
  - 多维矩阵和稀疏矩阵
  - 编程矢量计算机

多通道概念

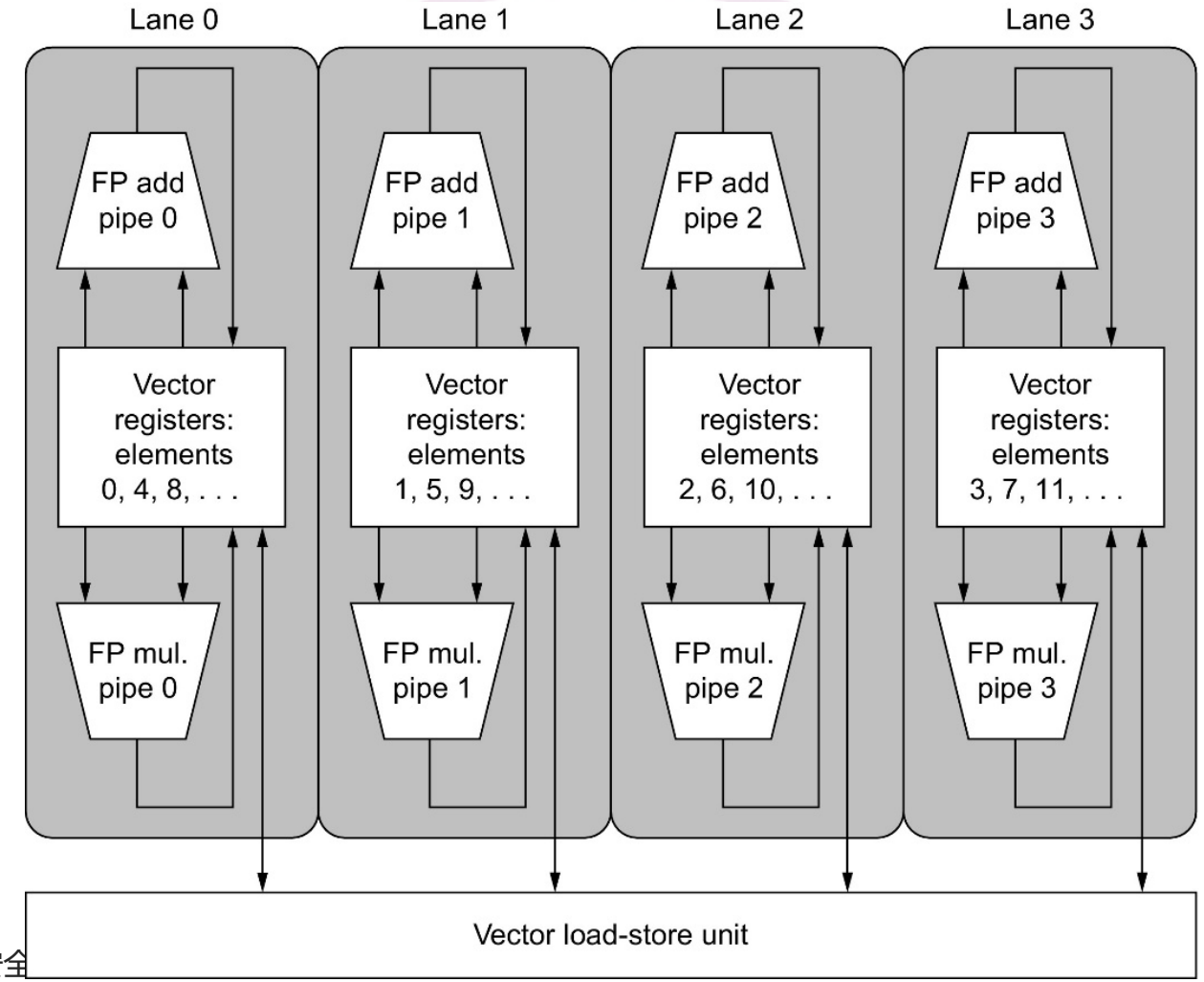
- 问题：每个时钟周期处理多个矢量元素
  - 使用多个功能单元以提高性能



(A) (B)

RV64V 多通道

- 问题：矢量算术指令只允许两个源寄存器的对应元素操作
  - 结构化为多个并行通道(lane) ⇒ 4 通道使定时周期数从 32 减到 8



多通道特点

- 为了发挥多通道的优势，具体的计算应用和体系结构必须支持长矢量。

- 优点
  - 算术流水线在一个通道上完成矢量中一个元素的操作，而不需要与其它通道进行通信，降低了写代价，减少了所需的寄存器文件端口数目。
  - 增加通道数目，不需要修改已有的机器代码。

- 缺点
  - 增加通道数目，稍微增加控制复杂度。

矢量长度寄存器

- 问题：如何处理矢量长度与最大矢量长度不同的数据
  - 在编译时未必知道矢量操作的长度

```
for(i=0;i<n;i=i+1)
  Y[i]=a*X[i]+Y[i];
```

- 矢量长度寄存器(vector-length register, `vl`)
  - 控制任何矢量操作的长度，包括矢量装入和存储操作
  - 不能大于最大矢量长度（mvl）
  - 在下一代计算机中只需要增加矢量寄存器长度而不需要修改指令集
- 条纹挖掘(strip mining)
  - 生成代码保证每个矢量操作处理的矢量长度小于等于 mvl。
- RISC-V指令集
  - 指令 `setvl` 将 $\min(mvl, n)$ 写入 `vl`，同时写标量寄存器以辅助控制循环运行。

矢量长度寄存器

- RISC-V 代码

```
        vsetdcfg 2 DP FP    # Enable 2 64b FP registers
        fld      f0,a       # Load scalar a
loop:   setvl    t0,a0      # vl = t0 = min(mvl,n)
        vld      v0,x5      # Load vector X
        slli     t1,t0,3    # t1 = vl*8(in bytes)
        add      x5,x5,t1   # Increment pointer to X by vl*8
        vmul     v0,v0,f0   # Vector-scalar mult
        vld      v1,x6      # Load vector Y
        vadd     v1,v0,v1   # Vector-vector add
        sub      a0,a0,t0   # n -= vl(t0)
        vst      v1,x6      # Store the sum into Y
        add      x6,x6,t1   # Increment pointer to Y by vl*8
        bnez     a0,loop    # Repeat if n!=0
        vdisable            # Disable vector regs
```

谓词寄存器

- 问题：代码中存在条件语句如何矢量化?
- 根据 Amdahl 定律，中低级矢量化对程序加速比提高有限
  - 循环内存在条件语句和使用稀疏矩阵是造成低级矢量化的主要原因

```
for(i=0;i<64;i=i+1)
  if(X[i]!=0)
    X[i]=X[i]-Y[i]
```

- 矢量掩码控制(vector-mask control)
  - RV64V中谓词寄存器(predicate register)提供矢量指令中对每个元素条件执行操作所必需的掩码和其它内容
  - 置位谓词寄存器 `p0` 后，所有后续的矢量指令只对谓词寄存器中对应位为 1 的矢量元素进行操作。

矢量掩码寄存器

- 使用谓词寄存器“禁用”元素

```
vsetdcfg    2*FP64      # Enable 2 64b FP vector regs
vsetpcfgi   1           # Enable 1 predicate register
vld         v0,x5       # Load vector X into v0
vld         v1,x6       # Load vector Y into v1
fmv.d.x     f0,x0       # Put floating-point zero into f0
vpne        p0,v0,f0    # Set p0(i) to 1 if v0(i)!=f0
vsub        v0,v0,v1    # Subtract under vector mask
vst         v0,x5       # Store the result in X
vdisable                # Disable vector registers
vpdisable               # Disable predicate registers
```

- 使用矢量掩码寄存器是有代价的
  - 消除分支指令和关联的控制相关，使得条件指令执行得更快，尽管可能做些无用功。

存储体

- 问题：没有足够的存储器带宽，矢量执行就是徒劳的。
- 装入指令启动时间：从内存获得第一个字放入寄存器的时间
  - 如果矢量剩余部分可以无停顿地提供，则矢量的引发速率将等于装载或存储新数据字的速度。
  - 许多处理器的装载/存储单元启动惩罚大约 100 个时钟周期
- 跨多个存储体分散访问通常可获得期望的速率
  - 独立控制存储体地址
  - 装载和存储非顺序数据字（需要独立存储体寻址）
  - 支持多矢量处理器共享同一个存储器
- 例：
  - 32个处理器中每个处理器每个时钟周期产生 4 条装载和 2 条存储指令
  - 处理器时钟周期：$2.167ns$，SRAM 周期：$15ns$
  - 需要：$32 \times (4 + 2) \times 15 \div 2.167 \approx 1330$ 个存储体

跨距

- 问题：如何处理多维矩阵
- 矢量中相邻分量在内存中的存储位置可能是不连续的

```
for(i=0;i<100;i=i+1)
  for(j=0;j<100;j=j+1){
    A[i][j]=0.0;
    for(k=0;k<100;k=k+1)
      A[i][j]=A[i][j]+B[i][k]*D[k][j]
  }
```

  - 矢量化矩阵 B 的每行与矩阵 D 的每列之间的乘法
  - 矩阵分配内存是线性的，必须按照行优先序（C语言）或列优先序（FORTRAN语言）进行。

- 跨距(stride)：收集到单个向量寄存器中元素相距的距离
  - 如果采用行优先序，矩阵 D 在内循环中的跨距是 $8 \times 100 = 800$ 字节

跨距

- 非单位跨距(non-unit strides)
  - 矢量处理器只需使用矢量装载指令 VLDS 和矢量存储指令 VSTS 来处理非单位跨距问题。
  - 在矢量寄存器中的矢量元素不存在非单位跨距问题。
- 当对同一个存储体的访问命中时间间隔小于存储体处于忙的时间时，存储体将发生冲突（停顿）
  - $LCM(stride, \#bank) \div stride < time_{\text{bank-busy}}$（$LCM$：最小公倍数）
- 例：
  - 已知：8个存储体，存储体忙时间为6个周期，总内存延迟为12个周期。
  - 问题：完成单位跨距 64 个元素矢量的装载时间？跨距为 32 时？
  - 答案：
    - 跨距为1时不会碰撞，读入时间：$12 + 64 = 76$ 个周期，每个元素 1.2 周期
    - 跨距为32时会碰撞，读入时间：$12 + 1 + 6 \times 63 = 391$ 个周期，每个元素 6.1 周期

收集-分散(Gather-Scatter)

- 问题：如何处理稀疏矩阵
- 稀疏矩阵的矢量元素以某种紧凑方式存储且需要间接访问。

```
for(i=0;i<n;i=i+1)
  A[K[i]]=A[K[i]]+C[M[i]];
```

  - 矩阵 A 和 C 的非零元素通过索引矢量 K 和 M 间接访问
  - 矩阵 A 和 C 的非零元素个数必须相同

- 支持稀疏矩阵的主要机制是使用索引矢量的收集-分散操作
  - 收集操作：使用“基地址+索引矢量偏移量”来读入矢量
  - 分散操作：使用相同地址进行分散存储

收集-分散(Gather-Scatter)

- 使用索引矢量

```
vsetdcfg    4*FP64      # Enable 4 64b FP vector registers
vld         v0,x7       # Load K[]
vldx        v1,x5,v0    # Load A[K[]]
vld         v2,x28      # Load M[]
vldx        v3,x6,v2    # Load C[M[]]
vadd        v1,v1,v3    # Add them
vstx        v1,x5,v0    # Store A[K[]]
vdisable                # Disable vector registers
```

矢量体系结构编程

- 问题：如何编程矢量计算机
- 程序员在代码编译时就可知其是否能够矢量化及原因
  - 允许其它领域专家学会如何通过改变代码来提高性能
  - 在矢量计算机中，影响程序成功的主要因素是程序结构本身
    - 循环中是否存在真正的数据相关
    - 如何重构以消除这些数据相关

矢量体系结构编程

- Perfect Club基准程序在Cray Y-MP上运行的矢量化水平

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

SIMD 指令集扩展

- 观察：媒体应用程序通常处理字长较短的数据类型
  - 图形系统使用三种颜色加上透明度共 8-bit × 4
  - 音频数据通常使用 8-bit 或 16-bit
  - 断开 256-bit 加法器的“部分”进位链，处理器可以对 32 个 8-bit 、16 个 16-bit 和 8 个 32-bit 的短矢量同时操作。

- 256-bit 操作数支持的 SIMD 多媒体

Instruction category	Operands
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Maximum/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit

与矢量指令相比的局限性

- 寄存器文件更小
  - RV64V 共有 32 个矢量寄存器，每个存储 32 个 64-bit 元素
- 在操作码中固定操作数数目
  - 没有矢量长度寄存器
  - 导致在 MMX, SSE 和 AVX 指令扩展中增加数百条指令
- 没有更加复杂的寻址方式
  - 没有跨距访问指令
  - 没有收集-分散访问指令
- 没有掩码寄存器
  - 掩码寄存器支持元素的有条件执行
- 结果
  - 编译器难于生成 SIMD 代码
  - 增加了使用 SIMD 汇编语言进行编程的难度

SIMD指令集扩展的实现

- Intel MMX (1996年)
  - 同时完成 8 个 8 比特整数操作或 4 个 16 比特整数操作
- 流 SIMD 扩展(SSE)(1999年)
  - 添加 16 个 128 比特寄存器(XMM寄存器)
  - 同时完成 16 个 8 比特操作、8 个 16 比特操作和 4 个 32 比特操作
- 高级矢量扩展(AVX) (2010年)
  - 将寄存器宽度扩展到 256 比特(YMM寄存器)
  - 加倍了所有短数据类型的同时操作数目
- AVX-512 (2017年)
  - 再次将寄存器宽度扩展到 512 比特(ZMM寄存器)，寄存器数目扩展到 32 个
  - 添加了分散 (VPSCATTER) 指令和掩码寄存器 (OPMASK)
- 操作数必须是连续存储且存储位置对齐的

SIMD 指令集扩展

- 目标：加速已有的库函数，而不是通过编译器生成代码。

- 普及的原因
  - 添加标准算术单元的代价很小且容易实现
  - 只需要与矢量体系结构相比很小的额外处理器
  - 许多计算机不具备支持矢量体系结构的存储器带宽
  - 不需要处理虚拟存储的问题
  - 二进制代码兼容性

SIMD代码示例

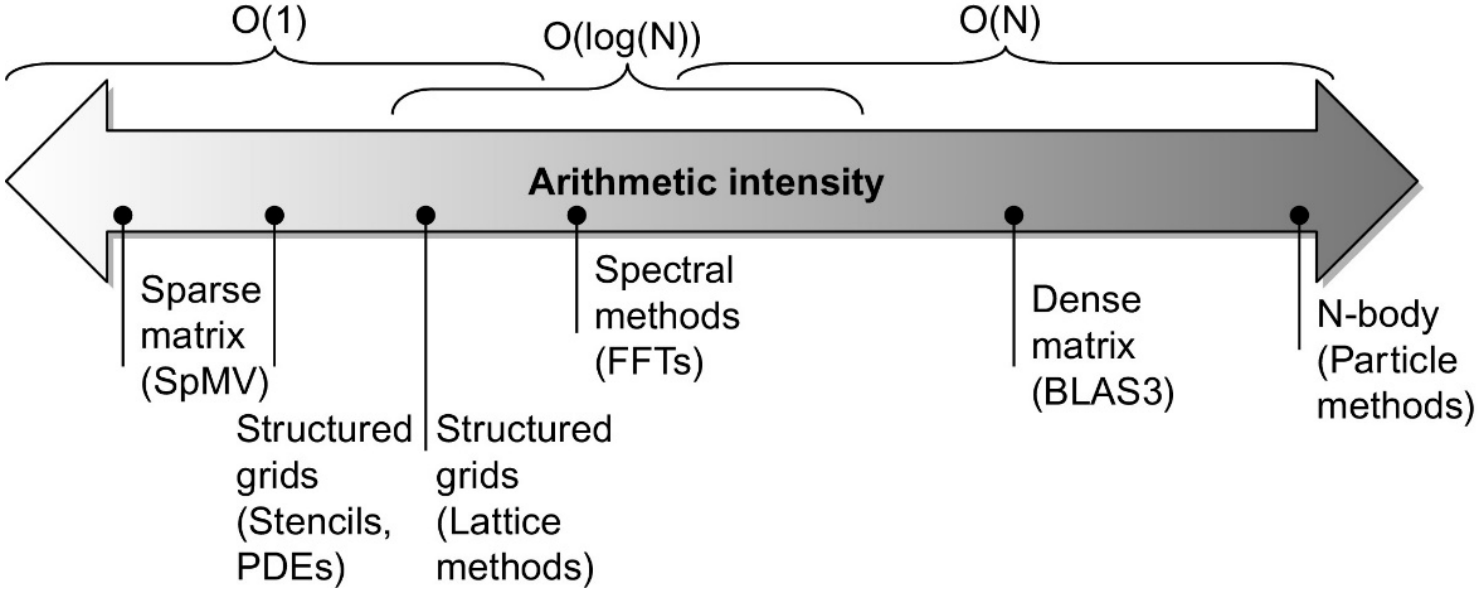
- 例：DAXPY 程序的 RISC-V SIMD代码
  - 后缀 .4D 表示一次对 4 个双精度浮点数同时进行操作

```
        fld       f0,a          # Load scalar a
        splat.4D  f0,f0         # Make 4 copies of a
        addi      x28,x5,#256   # Last address to load
Loop:   fld.4D    f1,0(x5)      # Load X[i]…X[i+3]
        fmul.4D   f1,f1,f0      # a*X[i]…a*X[i+3]
        fld.4D    f2,0(x6)      # Load Y[i]…Y[i+3]
        fadd.4D   f2,f2,f1      # a*X[i]+Y[i]…a*X[i+3]+Y[i+3]
        fsd.4D    f2,0(x6)      # Store Y[i]…Y[i+3]
        addi      x5,x5,#32     # Increment index to X
        addi      x6,x6,#32     # Increment index to Y
        bne       x28,x5,Loop   # Check if done
```

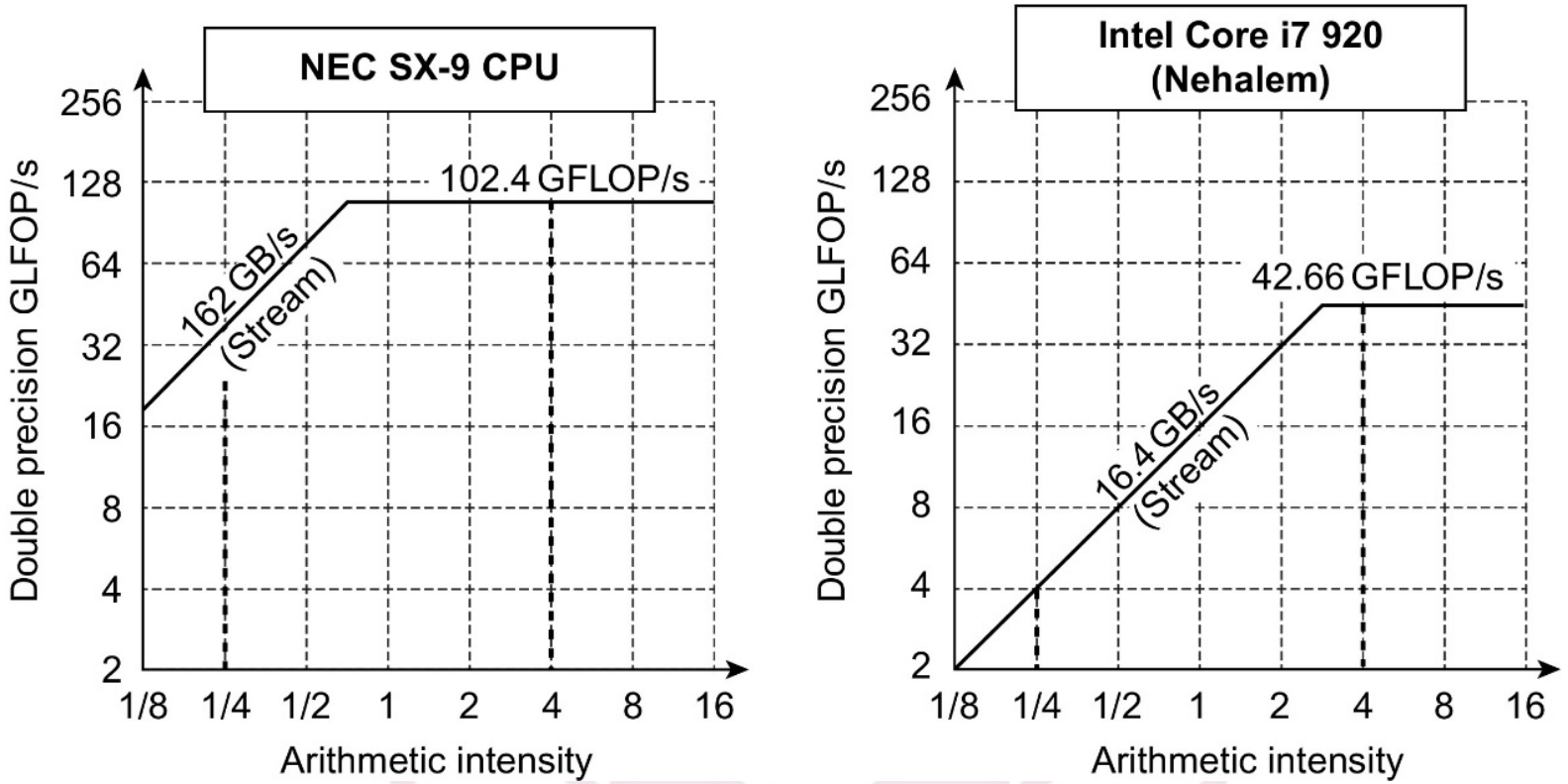
  - 获得大概 4 倍的降低：67 条指令 vs. 258 条指令

屋顶可视化性能模型

- 用途：比较各种 SIMD 体系结构的浮点数性能
- 基本思路
  - 将浮点吞吐量峰值(GFLOP/s)绘制为算术强度(FLOP/Byte)的函数
  - 将目标计算机的浮点数性能与存储性能(GB/s)相结合
- 算术强度(Arithmetic Intensity)
  - 定义：完成任务所需的浮点操作数目与内存传输数据量之比



例：屋顶可视化性能模型



- GFLOPs/秒 = min(存储器带宽峰值×算术强度，浮点数性能峰值)
- 右边虚线：击中屋顶的平坦部分，性能受到算力的限制
- 左边虚线：击中屋顶的倾斜部分，性能受到内存带宽的限制
- 脊点：斜线与水平屋顶线的交点——兴趣点

诚信 创新 实践