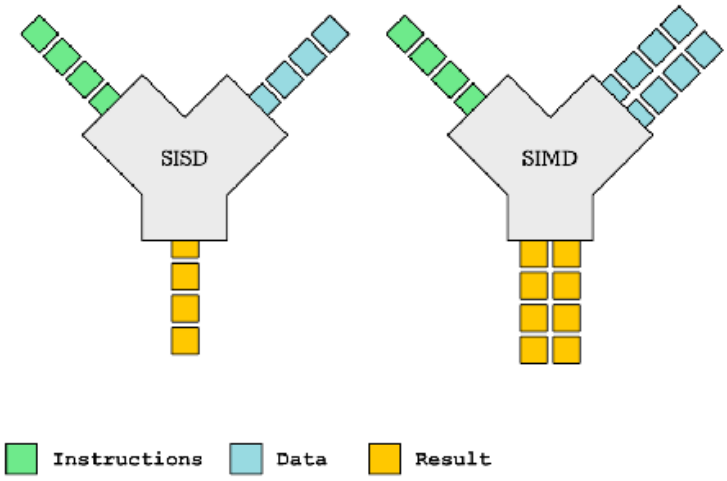
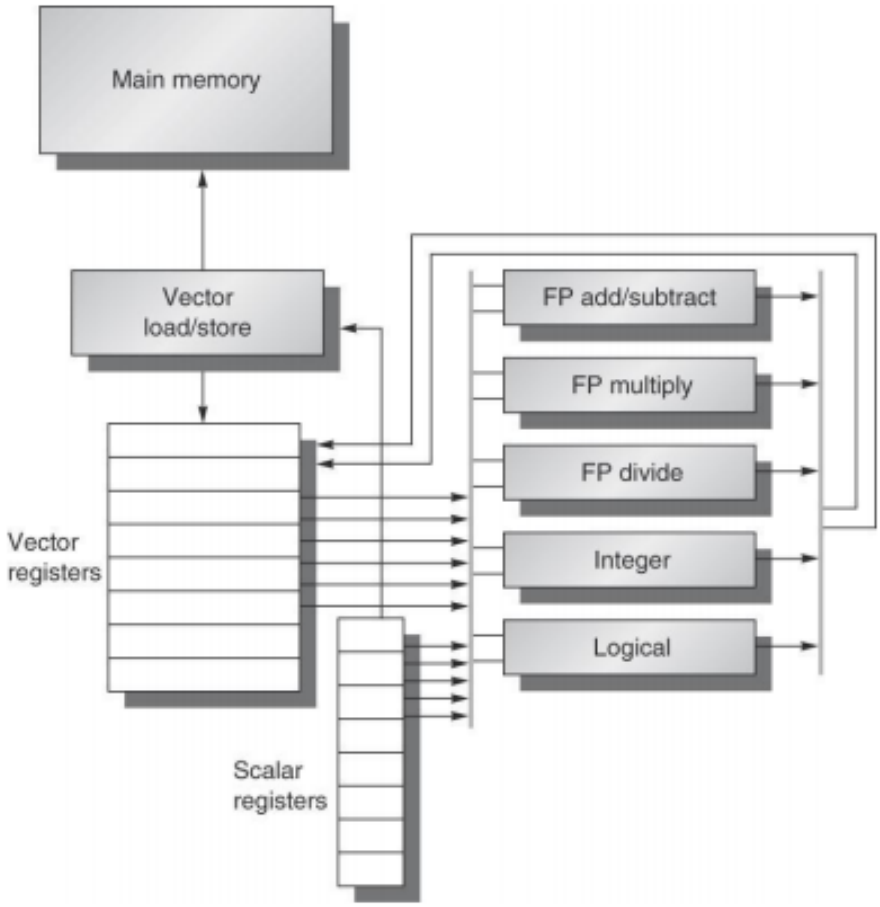
9 体系结构

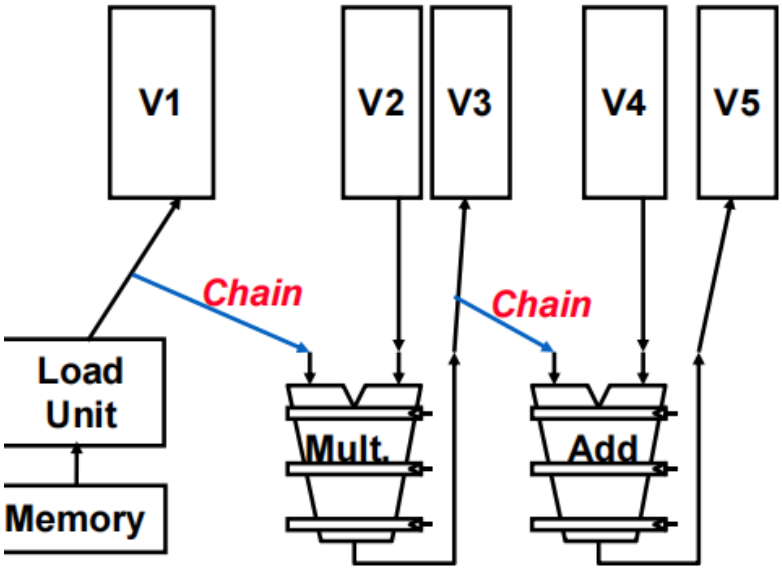
- 红色加粗字为课程要点
- **Flynn's Taxonomy**（分类）
  - SISD
  - SIMD
    - 扩展多数据操作的流水线执行，超标量
    - 大部分指令集（SSE，AVX）并行数据操作
    - SIMT：单指令多线程，用于GPU
  - MISD：没实现
  - MIMD
    - 紧耦合的MIMD（共享内存，NUMAS）OpenMP，Pthreads
    - 松耦合的MIMD（分布式系统内存，例如集群），MPI
- SIMD architecture
  - **SIMD架构的优势**
    - 可以利用数据级并行
      - 矩阵的科学计算
      - 面向媒体的图像和声音处理器
    - 比MIMD更节能
      - 多次数据操作只需要读取一条指令，而不是每个数据操作取指一次
      - 对个人移动设备具有吸引力
    - 允许程序员按顺序思考
    - 与MIMD相比：SIMD潜在加速是MIMD的两倍
      - x86 处理器 -> 预计每个芯片每年增加两个内核
      - SIMD -> 宽度每四年翻一番
  - SIMD 并行性
    - SIMD架构



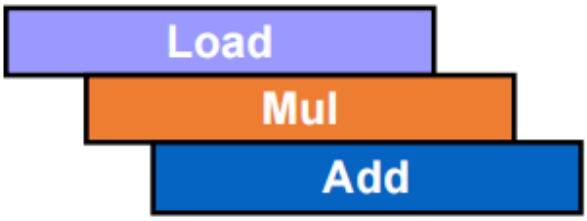
    - 向量架构，扩展多数据操作的流水线执行
    - 图形处理单元（SIMT，GPU）
- 向量架构
    - 基本操作：
        - 将数据元素集读入“向量寄存器”
        - 在这些寄存器上操作
        - 将结果打包发回内存
        - 一条指令处理数据向量，这导致在独立的数据元素上进行几十次寄存器操作。（寄存器最快）
    - 寄存器由编译器控制
        - 用于隐藏内存延迟
        - 利用内存带宽
        - 原因：向量loads and stores是深度流水线化和重叠的，程序努力为每个向量加载或存储只支付一次长内存延迟，而不是每个元素一次，从而分摊延迟
        - 总的来说，矢量程序努力保持内存繁忙。
    - **向量架构的例子**
        - 向量寄存器
        - 向量功能单元——FP加法和乘法单元
            - 完全流水线
            - 数据和控制冒险可以被检测
        - 向量加载存储单元
            - 完全流水线
            - 初始延迟后每个时钟周期一个字
        - 标量寄存器
        - RV(PPT____11)



    - RV64V 功能单元每个时钟周期消耗一个元素 -> 执行时间约为向量长度
- 执行时间——流水线向量指令
    - 流水线向量执行时间取决于
        - 操作数向量的长度
        - 结构风险：当两个或多个已经在流水线中的指令需要相同的资源时，就会发生结构风险，例如内存控制区冲突
        - 数据依赖
    - convoy护航指令组：一组可能一起执行的向量指令
        - 如何利用流水线?
            - 需要没有结构竞争，若有必须在不同的convoy中将指令序列化并启动
    - chaining（连接操作）
        - 允许一个向量操作在其向量源的各个元素可用时立即开始



- 若没有chaining，在开始有依赖的指令之前必须等待上一指令的结果的最后一个元素写入。但有chaining，可以在第一个结果元素出现时就立即启动相关指令

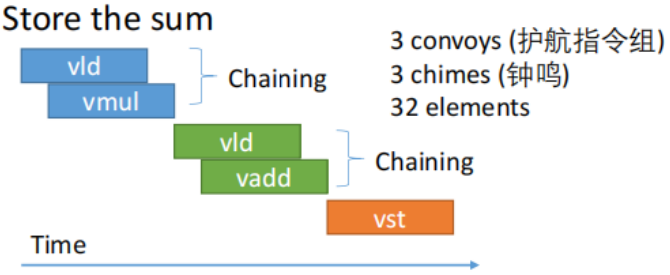


- chimes（钟鸣）
  - 执行一次护航指令组的时间单位
  - m个护航指令组需要m个钟鸣来执行
  - 对于长度为n的向量，需要$m \times n$个时钟周期——忽略了很多例如start-up-time
- 对于有read-after-write（先读后写）依赖的序列可以通过chaining来处于同一个convoy中
- 实例

```
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
```

Three convoys:

1. vld vmul
2. vld vadd
3. vst

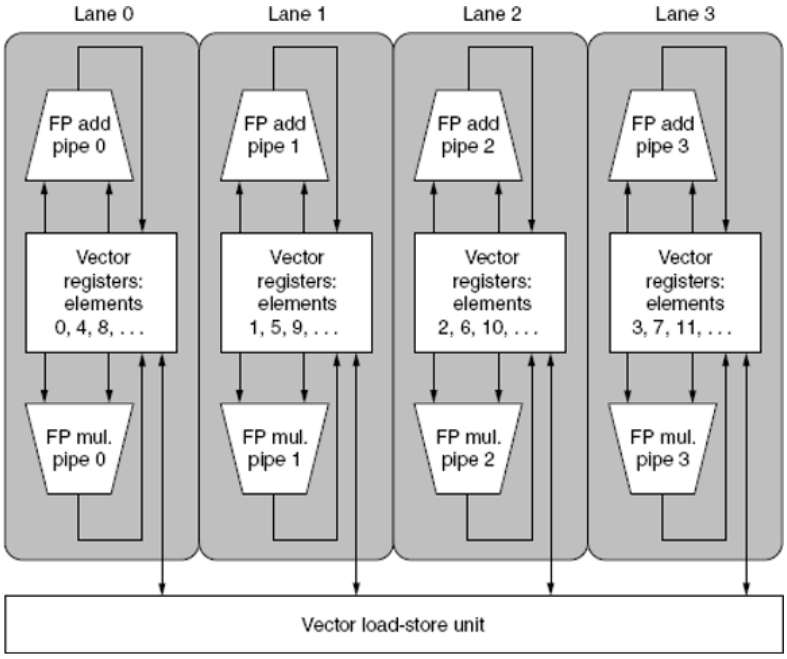


3 convoys (护航指令组)
3 chimes (钟鸣)
32 elements

3 chimes, 2 FP ops per result (element), cycles per FLOP = 1.5
For 32 element vectors, requires 32 x 3 = 96 clock cycles

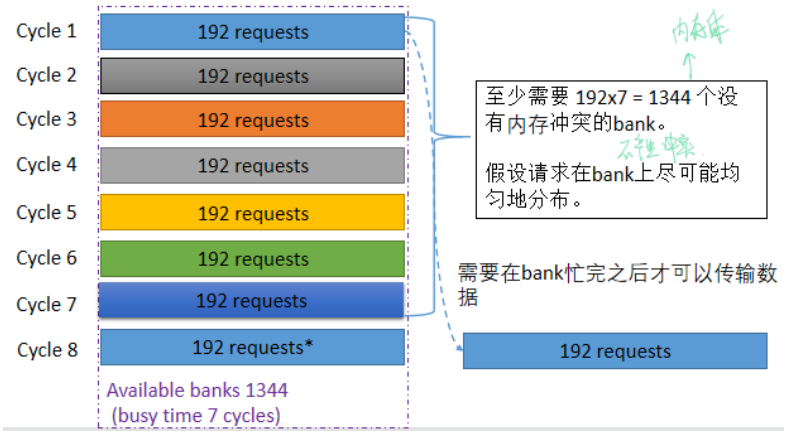
Overheads ignored: 1) vector instruction issue time, 2) vector startup time: the latency in clock cycles until the pipeline is full.

- **Vector architectures optimizations: Multiple Lanes, Vector Length Registers, Vector Mask Registers, Memory Banks, Stride, Scatter-Gather**
  - Multiple Lanes（多条车道）：每个时钟周期处理一个以上的元素
    - **四车道向量单元**

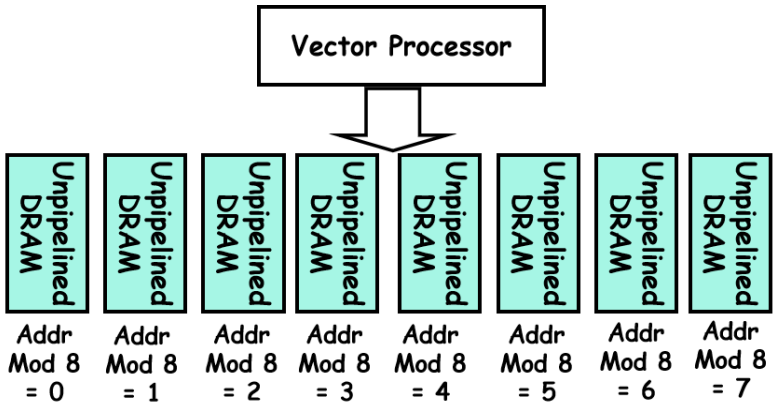


      - RV64V 指令只允许 一个向量的元素 N 参与涉及来自其他向量寄存器的元素 N 的操作，这简化了高度并行向量单元的构造
      - lane：包含向量寄存器元素的一部分和每个功能单元的一个执行管道
  - Vector Length Registers（向量长度寄存器）：处理非32宽的向量
  - Vector Mask Registers（向量遮罩寄存器）：处理 IF 语句
  - **Memory Banks**（内存组）：内存系统优化以支持向量处理器
    - 内存系统必须设计为支持向量加载和存储的高带宽
      - 内存启动时间：从内存中获取第一个字到寄存器

- 跨多个banks分散访问
  - 独立控制bank地址
  - 加载或存储非连续的字
  - 支持多向量处理器共享内存
- 这儿有个例题38页！
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, DRAM bank busy time (bank latency) is 15 ns
  - How many minimum memory banks needed to allow all processors to run at the full memory bandwidth (no bank conflicts)?
    - 32 processors x 6 =192 memory accesses,
    - 15ns DRAM cycle /2.167ns processor cycle≈7 processor cycles
    - 7 x 192 →1344!
- DRAM 的时钟频率 * 每个时钟周期传输的数据 * 内存总线（接口）带宽 * 接口数量 = 理论上的最大带宽
- fully 内存带宽：没有内存冲突；若有冲突，放不下的数据要排队等候可用bank，至少需要两轮bank的busy time才可以得到初始数据。



- 内存操作
  - 加载/存储操作在寄存器和内存之间移动数据组
  - 三种寻址方式
    - 单位步幅
      - 访问内存中连续的内存块
      - 最快：总是有可能优化
    - 非单位步幅（非连续访问，例如按列访问矩阵）
      - 更难对所有可能的步幅优化内存系统
      - 质数个数的数据banks使得在全带宽下支持不同步幅更容易
    - 索引（聚集-分散）
      - 寄存器间接地等价于向量
      - 适用于稀疏数据数组
      - 增加向量化程序数量
- 内存布局——轮询式分布



- 对单位步幅最优
  - 不同的DRAM连续的元素
  - 向量作的启动时间是单次读取的延迟
- 非单位步幅
  - 对质数数量的banks更好
  - 对2，4，8最糟

- stride（步幅）：处理多维度数组
- Scatter-Gather（分散-集中）：处理稀疏矩阵
- Programming Vector Architectures（向量体系结构编程）：影响性能的编程结构

- Programming Vector Architectures
- SIMD extensions for media apps
- **GPUs – Graphical Processing Units**
  - 如何提升GPU性能使其更广泛应用
    - 异构执行模型
    - CPU是主机，GPU是设备
    - 为GPU开发一个类似于C语言的编程语言
      - CUDA（Compute Unified programming language）
      - OpenCL
    - 将所有形式的 GPU 并行性统一为 CUDA 线程
    - 编程模型：“单指令多线程”（SIMT）
  - **thread, block, grid**
    - thread 和每个数据元素相关联
      - CUDA thread：数千个threads用于各种风格的并行化（SIMD，MIMD，多线程的）
    - thread 被组织成 block
      - 最多512个元素
      - 多线程SIMD处理器：执行整个block的硬件（每次执行32个元素）
    - block 被组织成 gird
      - block是以任意顺序独立执行的

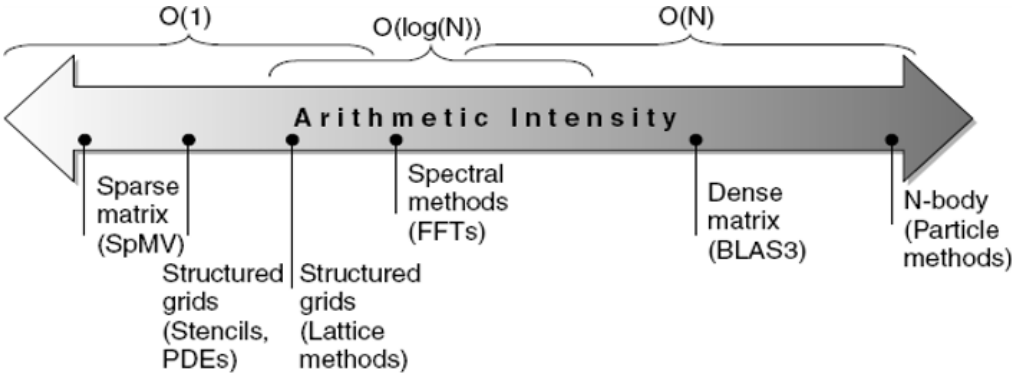
    - 不同block之间是不可以直接通信的，但可以使用**全局内存**中的原子内存进行协调
  - 由GPU硬件而非应用程序或操作系统进行线程管理
    - 一个由多线程SIMD处理器组成多处理器
    - 一个Thread Block Scheduler
- **Example:两个向量相乘**：每个block有512个线程，在host端4计算需要的block数

```
// Invoke DAXPY with 512 threads per Thread Block
__host__
int nblocks = (n+ 511) / 512;
daxpy<<<nblocks, 512 >>>(n, 2.0, A, B, C);
// DAXPY in CUDA
__global__
void daxpy (int n, double a, double *A, double *B,
            double *C)
{
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i < n)
                A[i] = B[i] * C[i];
}
```

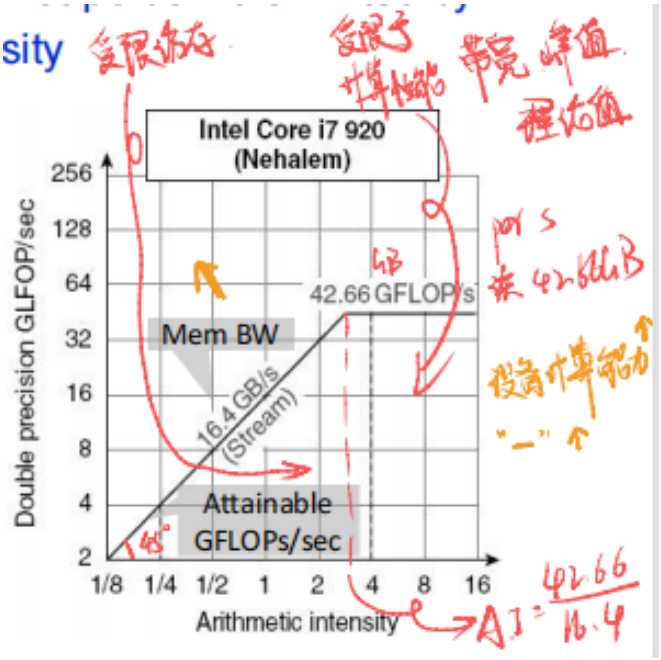
  - SIMD指令一次执行32个元素
  - SIMD线程=warp=传统意义上的线程
  - 线程块调度器（Thread block scheduler）：将线程块分配到多线程SIMD处理器上
  - 多线程处理器有16个lane，所以一个warp需要执行两次
  - 目前GPU有56个多线程SIMD处理器
- **NVIDA GPU 内存结构**
  - 每个SIMD lane有片外DRAM的私有部分
    - 私有内存，不被任何lane共享
    - 包含堆栈，寄存器和不适合寄存器的私有变量
    - 目前GPU将私有内存缓存在L1和L2caches
  - 每个多线程SIMD处理器也有片上的本地内存（shared memory）
    - 仅由块内的SIMD lanes之间共享
    - 延迟比GPU内存小100倍
    - 线程可以访问同意线程块中其他线程从全局内存中加载到本地内存中的数据
  - SIMD处理器共享的片外内存是GPU的global memory
    - 主机可以读写GPU global memory
  - GPU 内存（global memory）被所有grids共享
  - local memory （shared Memory） 被线程块内的SIMD指令的所有线程共享
  - private memory 单个CUDA thread 私有

- **术语**
  - threads of SIMD instructions（SIMD指令的线程）
    - 有自己的PC
    - 线程调度器使用分板调度
    - 线程之间没有数据依赖
    - 跟踪48个SIMD指令线程
      - 隐藏内存延迟
  - thread block scheduler(线程块调度器)
    - 将线程块调度到SIMD处理器上，每个线程块被拆分为子块（warp）并分配给在SIMD处理器上运行的SIMD线程
  - 在每个SIMD处理器中
    - 16/32个SIMD lanes
    - 与向量处理器相比：宽而浅
- **条件分支**——没怎么懂想说啥
  - GPU分支硬件使用
    - 内部掩码——遮罩寄存器
    - 分支同步栈
      - 由每个SIMD lane的掩码组成
    - 当分支分为多个执行路径时需要管理的指令标记
      - 压入分叉
    - 当路径汇聚时
      - 充当障碍
      - 弹出栈
  - 每线程lane 1 bit 谓词寄存器，由程序员指定
- Fermi architecture innovations
- Examples of loop-level parallelism
  - 在warp临界处有可能会有数据依赖
  - 寻找依赖
    - 若存的地址是：a * i + b
    - 取的地址是：c * i + d
    - i 从m到n
    - gcd(c, a)最大公约数一定可以被 (d - b)整除
- Fallacies
- **Roofline**
  - 基本思路
    - 峰值浮点吞吐作为计算密度的一个函数

    - 将浮点计算的性能和内存性能结合在一起
- 计算密度、计算访存比：每读取一个字节做多少浮点操作
- 可以达到的 每秒浮点运算 = Min（内存带宽峰值<每秒读多少> * 计算密度<读一次做多少浮点计算>， CPU的理论浮点计算峰值）
- 稠密矩阵计算密度会随着问题规模增大而增大，但稀疏矩阵不会



- 在roofline的倾斜部分，性能受到内存带宽的限制，在平坦部分，它受到计算性能的限制



    - 带宽决定斜率
    - 计算能力决定横线高度

以上内容整理于 幕布文档