计算机体系结构量化研究方法

王若琪 高性能期末总结

第四章：
PPT 内容
Page 4 Flynn's Taxonomy

Flynn's Taxonomy 分类法

- *SISD* - Single instruction stream, single data stream
  单指令流，单数据流
- *SIMD* - Single instruction stream, multiple data streams 单指令流，多数据流
  - New: *SIMT* – Single Instruction Multiple Threads (for GPUs) 单指令多线程
- *MISD* - Multiple instruction streams, single data stream 多指令单数据流
  - No commercial implementation 没有商业实现
- *MIMD* - Multiple instruction streams, multiple data streams 多指令流多数据流
  - Tightly-coupled MIMD (shared memory, NUMAs) 紧耦合
  - Loosely-coupled MIMD (distributed memory system, e.g. cluster) 松耦合

Page 5 Advantages of SIMD architectures

SIMD体系结构的优点

Advantages of SIMD architectures

1. Can exploit significant data-level parallelism for:
   1. matrix-oriented scientific computing
   2. media-oriented image and sound processors
2. More energy efficient than MIMD 比MIMD更节能
   1. Only needs to fetch one instruction per multiple data operations, rather than one instr. per data op.
   2. Makes SIMD attractive for personal mobile devices
3. Allows programmers to continue thinking sequentially
   允许程序员继续按串行顺序思考

可利用显著的数据级并行：
matrix-oriented科学计算
面向媒体的图像和声音处理器

每次多重数据操作只需要获取一条指令。

使SIMD对个人移动设备具有吸引力

- SIMD/MIMD comparison. Potential speedup for SIMD twice that from MIMD!
  - x86 processors → expect two additional cores per chip per year
  - SIMD → width to double every four years

  SIMD /MIMD比较。SIMD的潜在加速是MIMD的两倍!

Page 10 Example of vector architecture

向量结构的例子

Example of vector architecture

- RV64V → RV64G extended with vector instructions
- Vector registers
  - Each register holds a 32-element, 64 bits/element vector
  - Register file has 16 read ports and 8 write ports
- Vector functional units – FP add and multiply
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 31 general-purpose registers
  - 32 floating-point registers

向量寄存器
每个注册持有32-element, 64位/元素向量
寄存器文件有16个读港口和8写端口
向量功能单元- FP添加和繁殖
完全管道化
检测数据和控制危害
向量负载储备单位
完全管道化
初始延迟后一个词每个时钟周期
标量寄存器
31通用寄存器
32位浮点寄存器

Page 25 Optimizations

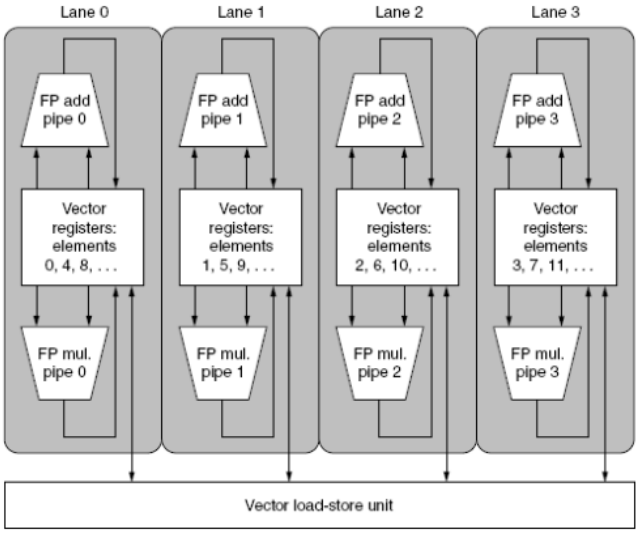
Optimizations 优化

1. *Multiple Lanes （多条车道）→* processing more than one element per clock cycle 每个时钟周期处理多个元素
2. *Vector Length Registers （向量长度寄存器）→* handling non-32 wide vectors 处理非32宽向量
3. *Vector Mask Registers （向量遮罩寄存器）→* handling IF statements in vector code 处理向量中的IF语句的代码
4. *Memory Banks （内存组）→* memory system optimizations to support vector processors 支持向量处理器的内存系统优化
5. *Stride （步幅）→* handling multi-dimensional arrays 处理多维数组
6. *Scatter-Gather （分散-集中）→* handling sparse matrices 处理稀疏矩阵
7. *Programming Vector Architectures （向量体系结构编程）→* program structures affecting performance 影响性能的程序结构

Page 26 1. A four lane vector unit

1. A four lane vector unit

- RV64V instructions only allow element N of one vector to take part in operations involving element N from other vector registers→ this simplifies the construction of a highly parallel vector unit
  - Lane → contains one portion of the vector register elements and one execution pipeline from each functional unit
  - Analog with a highway with multiple lanes!!



RV64V指令只允许一个向量中的N元素参与涉及元素N从其他向量寄存器这简化了建设一个高度并行向量单位巷→向量寄存器的元素包含一个部分和一个执行管道从每个功能单元
模拟与多个高速公路车道!

Page 31 Memory banks, 课本 page 298-299 中文 276-277

内存系统必须支持高带宽向量加载和存储
发散访问多个银行
  独立控制银行地址
  负载或存储非时序的单词
  支持多个向量处理器共享相同的内存
例子:
32处理器,每个生成4load和2store/周期
处理器周期是2.167 ns, DRAM组繁忙时间是15 ns
多少最低内存组需要允许所有处理器运行在整个内存带宽(没有银行冲突)?

4. Memory banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non-sequential words
  - Support multiple vector processors sharing the same memory

- Example:
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, DRAM bank busy time 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!

4.2.7 内存组：为向量载入/存储单元提供带宽

载入/存储向量单元的行为要比算术功能单元的行为复杂得多。载入操作的开始时间就是它从存储器向寄存器中载入第一个字的时间。如果在无停顿情况下提供向量的其他元素，那么向量初始化速率就等于提取或存储新字的速度。这一初始化速率不一定是一个时钟周期，因为存储器组的停顿可能会降低有效吞吐量，这一点不同于较简单的功能单元。

一般情况下，载入/存储单元的起始代价要高于算术单元的这一代价——在许多处理器中要多于 100 个时钟周期。对于 VMIPS，我们假定起始时间为 12 个时钟周期，与 Cray-1 相同。（最近的向量计算机使用缓存来降低向量载入与存储的延迟。）

为了保持每个时钟周期提取或存储一个字的初始化速率，存储器系统必须能够生成或接受这么多的数据。将访问对象分散在多个独立的存储器组中，通常可以保证所需速率。稍后将会看到，拥有大量存储器组可以很高效地处理那些访问多行或多列数据的向量载入或存储指令。

大多数向量处理器都使用存储器组，允许进行多个独立访问，而不是进行简单的存储器交错，其原因有以下 3 个。

(1) 许多向量计算机每个时钟周期可以进行多个载入或存储操作，存储器组的周期时间通常比处理器周期时间高几倍。为了支持多个载入或存储操作的同时访问请求，存储器系统需要有多个组，并能够独立控制对这些组的寻址。

(2) 大多数向量处理器支持载入或存储非连续数据字的功能。在此类情况下，需要进行独立的组寻址，而不是交叉寻址。

(3) 大多数向量计算机支持多个共享同一存储器系统的处理器，所以每个处理器会生成其自己的独立寻址流。

这些特征综合起来，就有了大量的独立存储器组，如下例所示。

例题 Cray T90（Cray T932）的最高配置有 32 个处理器，每个处理器每个时钟周期可以生成 4 个载入操作和 2 个存储操作。处理器时钟周期为 2.167 ns，而存储器系统所用 SRAM 的周期时间为 15 ns。计算：为使所有处理器都以完全存储器带宽运行，最少需要多少个存储器组。

解答 每个时钟周期产生的最大存储器引用数目为 192：每个处理器每个时钟周期产生 6 次引用共 32 个处理器。每个 SRAM 组的繁忙时钟周期数为 15/2.167=6.92，四舍五入为 7 个处理器时钟周期。因此，至少需要 $192 \times 7 = 1344$ 个存储器组！

Cray T932 实际上有 1024 个存储器组，所以早期型号不能让所有处理器都同时维持完全带宽。后来对存储器进行升级时，用流水化同步 SRAM 代替了 15 ns 的异步 SRAM，存储器周期时间缩短一半，从而可以提供足够的带宽。

从更高一级的角度来看，向量载入/存储单元与向量处理器中的预取单元扮演着类似的角色，它们都是通过向处理器提供数据流来尝试提供数据带宽。

77

Page 34-37

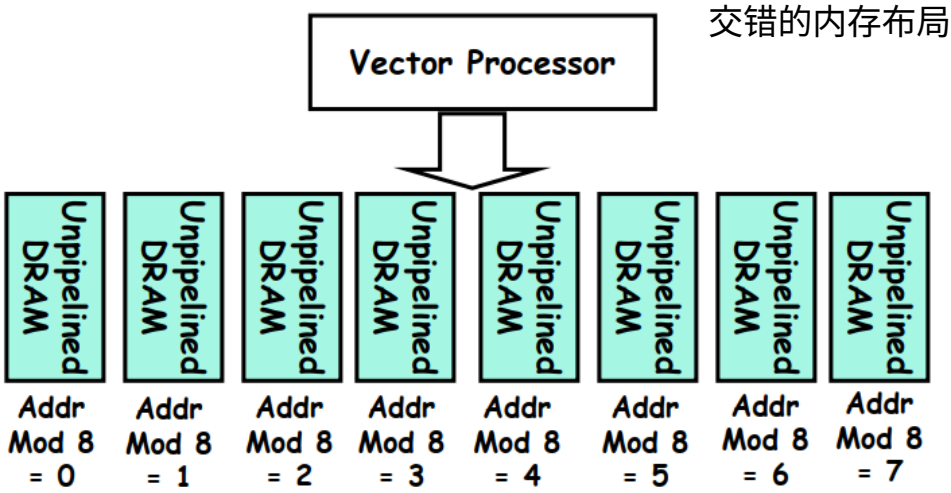
Memory operations

- Load/store operations move groups of data between registers and memory 加载/存储操作在寄存器和内存之间移动数据组
- Three types of addressing
  - Unit stride
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth
  - Indexed (gather-scatter)
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize

三种寻址方式
单位步
•内存中的连续信息块
•最快:总是可以优化这个
非单位(常量)步幅
•很难对内存系统进行所有可能的优化
•Prime number of data banks使它更容易支持不同的大步在全带宽
索引(gather-scatter)
•间接寄存器的向量等价
•适用于稀疏数组的数据
•增加向量化的程序数量

Interleaved Memory Layout

交错的内存布局



- Great for unit stride:
  - Contiguous elements in different DRAMs
  - Startup time for vector operation is latency of single read
- What about non-unit stride?
  - Above good for strides that are relatively prime to 8
  - Bad for: 2, 4, 8 (worst)
  - Better: prime number of banks…!

单位步幅great:
•不同dram中的连续元素向量操作的启动时间是单次读取的延迟
那么非单位步幅呢?
•以上对prime to 8的步幅比较好
坏处:2,4,8(最糟糕)
•更好的：素数组

5. Stride → multiple dimensional arrays

步幅-多维数组

技术来获取内存中不相邻的向量元素
步幅-元素之间的距离聚集在一个寄存器

- Technique to fetch vector elements that are not adjacent in memory
- Stride → the distance between elements to be gathered in one register.
- Example (recall that in C an array is stored in major row order!!)

在C语言中，数组是按主行顺序存储的

```
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];
    }
```

一定要向量化B的行与D的列的乘积

- Must vectorize multiplication of rows of B with columns of D
- Use *non-unit stride; D's stride is 100 double words (800 bytes); B's stride is one double word (8 bytes)*

使用non-unit步;D的步幅是100个双字(800字节);B的步幅是一个双字(8字节)

- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride, #banks) < bank busy time (in # of cycles)

看书4.2.7　4.2.8

Stride example

- Given
  - 8 memory banks
  - bank busy time of 6 cycles
  - total memory latency of 12 cycles.
  - bank busy time: the busy time is the time the bank is occupied with one request
- **Questions**: How long will it take to complete a 64-element vector load
  1. With a stride of 1?
  2. With a stride of 32?
- **Answers:**
  1. Stride of 1: number of banks is greater than the bank busy time, so it takes 12 + 64 = 76 clock cycles → 76/64 = 1.2 cycle for each vector element
  2. Stride of 32: the worst case scenario happens when the stride value is a multiple of the number of banks, which this is! Every access to memory will collide with the previous one! Thus, the total time will be:

     12 + 1 + 6 * 63 = 391 clock cycles→ 391/64 = 6.1 clock cycles per vector element!

Page 45-46 Roofline Model 课本 Page 307-310 4.3.2

基本思想:
浮点峰值吞吐量as计算强度的函数
浮点性能和内存性能联系在一起
车顶:屋顶上的倾斜的部分内存带宽性能是有限的,在平面上,部分是受限于计算强度

4.3.2 Roofline可视性能模型

285 有一种直观的可视方法可以对比各种 SIMD 体系结构变体的潜在浮点性能,那就是 Roofline 模型[Williams 等人，2009]。它将浮点性能、存储器性能和运算密度汇总在一个两维图形中。运算密度等于浮点运算数与所访问存储器字节的比值。其计算方法为：获取一个程序的总浮点运算数，然后再除以在程序执行期间向主存储器传送的总数据字节。图 4-6 给出了几种示例内核的相对运算密度。

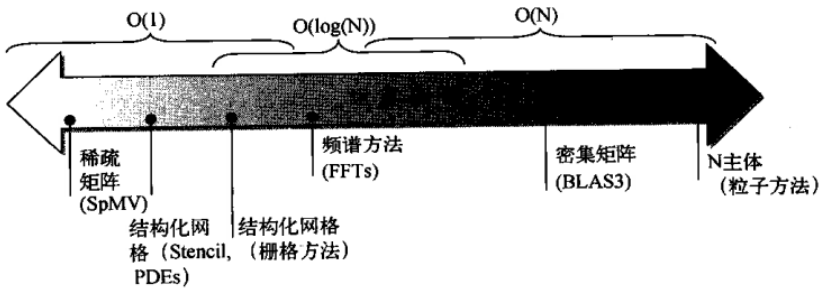


图 4-6 运算密度，定义为：运行程序时所执行的浮点运算数除以在主存储器中访问的字节数[Williams 等人，2009]。一些内核的运算密度会随问题的规模（比如密集矩阵）而缩放，但有许多核心的运算密度与问题规模无关

峰值浮点性能可以使用硬件规范求得。这一实例研究中的许多核心都不能放到片上缓存中，所以峰值性能是由缓存背后的存储器系统确定的。注意，我们需要的是可供处理器使用的峰值存储器带宽，而不只是 4.7 节表 4-13 中 DRAM 管脚处的可用宽带。要求出（所提供的）峰值存储器性能，其中一种方法是运行 Stream 基准测试。

图 4-7 在左侧给出 NEC SX-9 向量处理器的 Roofline 模型，在右侧给出 Intel Core i7 920 多核计算机的相应模型。垂直的 Y 轴是可以实现的浮点性能，为 2～256 GFLOP/s。水平的 X 轴是运算密度，在两个图中都是从 1/8 FLOP/DARM 访问字节到 16 FLOP/DARM 访问字节。注意，该图为对数-对数图尺，Roofline 对于一种计算机仅完成一次。

对于一个给定内核，我们可以根据它的运算密度在 X 轴上找到一个点。如果过该点画一条垂线，此内核在该计算机上的性能必须位于该垂线上的某一位置。我们可以绘制一个水平线，显示该计算机的浮点性能。显然，由于硬件限制，实际浮点性能不可能高于该水平线。

如何绘制峰值存储器性能呢？由于 X 轴为 FLOP/字节，Y 轴为 FLOP/s，所以字节/s 就是

图中 45 度角的对角线。因此，我们可以画出第三条线，显示该计算机的存储器系统对于给定运算密度所能支持的最大浮点性能。我们可以用公式来表示这些限制，以绘制图 4- 7 中的相应曲线： 286

可获得的 GFLOP/s=Min（峰值存储器带宽 × 运算密度，峰值浮点性能）

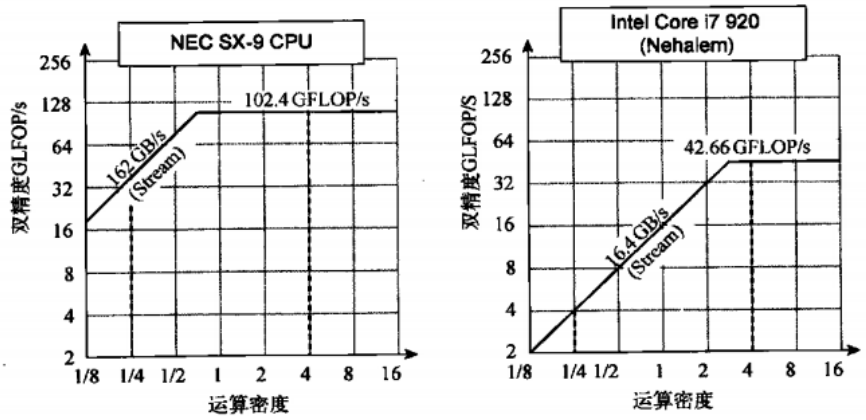


图 4-7 左图为 NEC SX-9 向量处理器上的 Roofline 模型，右图为采用 SIMD 扩展的 Intel Core i7 920 多核计算机的相应模型[Williams 等人，2009]。这个 Roofline 模型针对的单位步幅的存储器访问和双精度浮点性能。NEC SC-9 是在 2008 年发布的超级向量计算机，耗费了数百万美元。根据 Stream 基准测试，它的峰值 DP FP 性能为 102.4 GFLOP/s，峰值存储器宽度为 162GB/s。Core i7 920 的峰值 DP FP 性能为 42.66 GFLOP/s 和峰值存储器带宽为 16.4 GB/s。在运算密度为 4 FLOP/字节处的垂直虚线显示两个处理器都以峰值性能运行。在这个示例中，102.4 GFLOP/s 处的 Sx-9 要比 42.66 GFLOP/s 处的 Core i7 快 2.4 倍。在运算密度为 0.25 FLOP/字节处，SX-9 为 40.5 GFLOP/s，比 Core i7 的 4.1 GFLOP/s 快 10 倍

水平线和对角线给出了这个简单模型的名字，并指出了它的取值。Roofline 根据内核的运算密度设定了其内核的性能上限。如果我们把运算密度看作是触及房顶的柱子，它既可能触及房顶的平坦部分（表示这一性能是受计算功能限制的），也可能触及房顶的倾斜部分（表示这一性能最终受存储器带宽的限制）。在图 4-7 中，右侧的垂直虚线（运算密度为 4）是前者的示例，左侧的垂直虚线（运算密度为 1/4）是后者的示例。给定一台计算机的 Roofline 模型，就可以重复应用它，因为它是不会随内核变化的。

注意对角线与水平线交汇的“屋脊点”，通过它可以深入了解这台计算机的性能。如果它非常靠右，那么只有运算密度非常高的内核才能实现这台计算机的最大性能。如果它非常靠左，那么几乎所有内核都可能达到最高性能。后面将会看到，与其他 SIMD 处理器相比，这个向量处理器的存储器带宽要高得多，屋脊点非常靠左。

图 4-7 显示 SX-9 的峰值计算性能比 Core i7 快 2.4 倍，但存储器性能要快 10 倍。对于运算密度为 0.25 的程序，SX-9 快 10 倍（40.5 GFLOP/s 比 4.1 GFLOP/s）。更宽的存储器带宽将屋脊点从 Core i7 的 2.6 移动到 SX-9 的 0.6，这就意味着有更多的程序可以在这个向量处理器上达到峰值计算性能。

C. Graphical Processing Unit - GPU

- Given the hardware invested to do graphics well, how can it be supplemented to improve performance of a wider range of applications (General-purpose GPU)?
- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
    - Compute Unified Device Architecture (CUDA)
    - OpenCL for vendor-independent language
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model: "Single Instruction Multiple Thread" (SIMT)

考虑到已经投入了大量的硬件来做图形处理，那么如何补充这些硬件来提高更广泛的应用(通用GPU)的性能呢?
基本思想:
异构执行模型
CPU主机,GPU设备
为GPU开发c编程语言
统一计算设备架构(CUDA)
OpenCL的独立于供应商的语言
统一所有形式的GPU并行CUDA线程
编程模型:“单指令多线”(SIMT)

 看书4.4.2

Threads, blocks, and grid (CUDA)

- A ***thread*** is associated with each data element 线程与每个数据元素相关联
  - *CUDA threads* ➔ thousands of threads are utilized to various styles of parallelism: multithreading, SIMD, MIMD, ILP 数以千计的线程被用于各种类型的并行
- Threads are organized into ***blocks*** 线程被组织成块
  - *Thread Blocks*: groups of up to 512 elements 线程块:最多由512个元素组成的组
  - *Multithreaded SIMD Processor*: hardware that executes a whole thread block (32 elements executed per thread at a time) 多线程SIMD处理器:执行整个线程块的硬件(每个线程一次执行32个元素)
- Blocks are organized into a ***grid*** 块被组织成一个grid
  - Blocks are executed independently and in any order 块以任意顺序独立执行
  - Different blocks cannot communicate directly but can *coordinate* using atomic memory operations in Global Memory 不同的块不能直接通信，但可以在全局内存中使用原子内存操作进行协调
- Thread management handled by GPU hardware not by applications or OS 线程管理由GPU硬件而不是应用程序或操作系统处理
  - A multiprocessor composed of multithreaded SIMD processors
  - A Thread Block Scheduler 由多线程SIMD处理器组成的多处理器调度程序线程块

NVIDIA GPU architecture

- Similarities to vector machines: 向量机的相似之处：
  - Works well with data-level parallel problems 可以很好地处理数据级并行问题
  - Scatter-gather transfers Scatter-gather 转化
  - Mask registers mask寄存器
  - Large register 大寄存器

- Differences:
  - No scalar processor 没有标量处理器
  - Uses multithreading to hide memory latency 使用多线程来隐藏内存延迟
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor
    有许多功能单元，而不是像向量处理器那样的几个深度流水线单元

Page 52-55 Example: multiply two vectors of length 8192

Example: multiply two vectors of length 8192

Graphical Processing Units

- C code version:

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n = 8192, double a, double *A, double *B, double *C)
{
        for (int i = 0; i < n; ++i)
        A[i] = B[i] * C[i];
}
```

- CUDA version:

```
// 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];
}
```

Example: multiply two vectors of length 8192

- Grid → Code that works over all elements
- Thread block→ analogous to a strip-mined vector loop with vector length of 32. Breaks down the vector into manageable set of vector elements

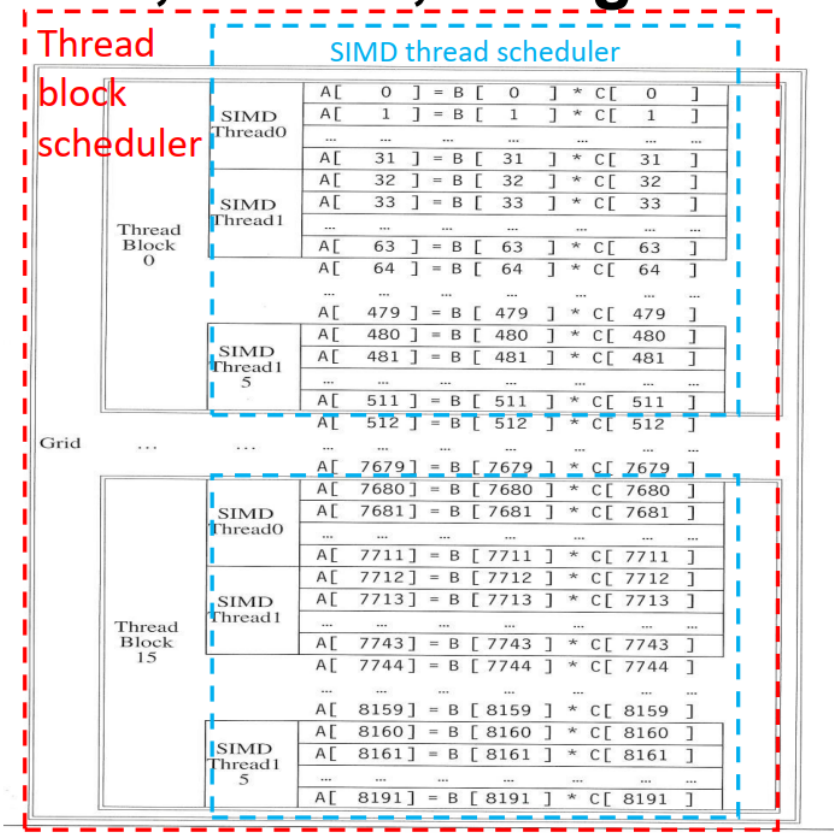
  32 elements/SIMD thread x 16 SIMD threads/block → 512 elements/block

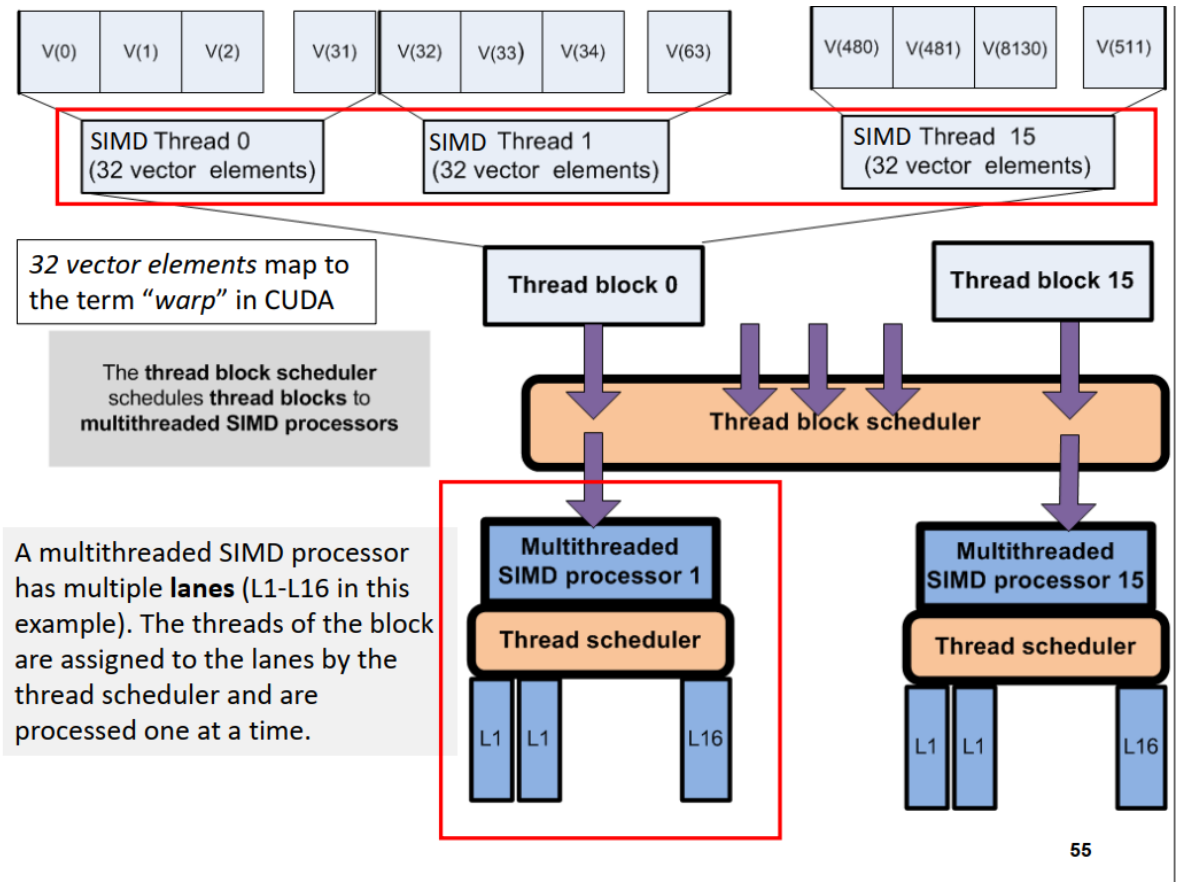
  SIMD instruction executes 32 elements at a time

  Grid size = 8192 vector elements / 512 elements/block = 16 blocks

- *Thread block scheduler* → assigns a thread block to a *multithreaded SIMD processor*
- Current-generation GPUs (Pascal P100) have 56 multithreaded SIMD processors

Threads, blocks, and grid example





Page 59-60 NVIDIA GPU memory structures 课本 Page 326-328　4.4.5

NVIDIA GPU memory structures

每个SIMD Lane都有片外DRAM的私有部分

- Each SIMD Lane has private section of *off-chip* DRAM
  - "Private memory", not shared by any other lanes　"私人记忆"，不被任何其他通道共享
  - Contains stack frame, register spilling, and private variables that don't fit in the registers. 包含堆栈框架、寄存器溢出和寄存器容纳不下的私有变量。
  - Recent GPUs cache this private memory in L1 and L2 caches 最近的gpu将这个私有内存缓存在L1和L2缓存中
- Each multithreaded SIMD processor also has local memory (Shared Memory) that is *on-chip* 每个多线程SIMD处理器还具有片上的本地内存(共享内存)
  - Shared by SIMD lanes / threads *within a block only* 仅在一个区块内由SIMD lane /线程共享
  - Latency is100x lower than GPU Memory. 时延比GPU内存低100倍。
  - Threads can access data in local memory loaded from global memory by other threads within the same thread block 线程可以访问本地内存中的数据，这些数据是由同一线程块中的其他线程从全局内存加载的
- The *off-chip* memory shared by SIMD processors is GPU memory (Global memory) SIMD处理器共享的片外内存是GPU内存(全局内存)
  - Host can read and write GPU memory　主机可对GPU内存进行读写

55

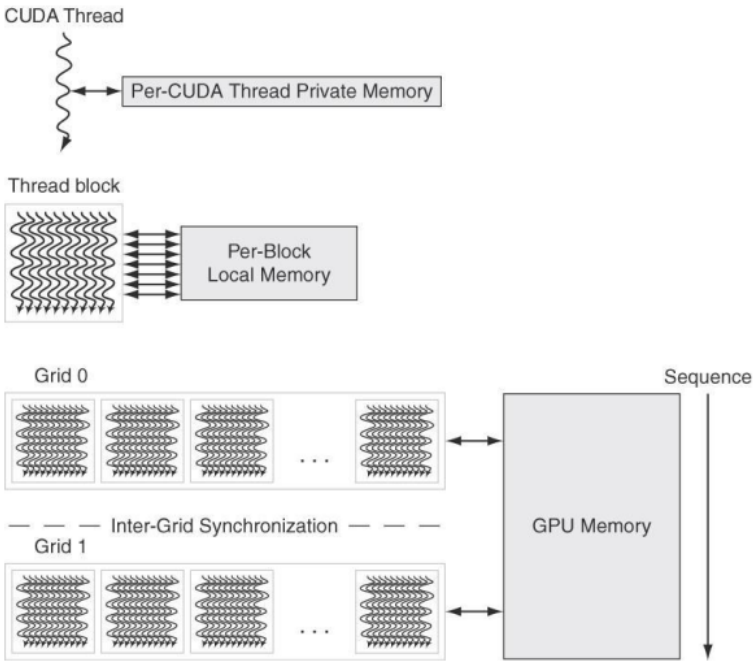


Figure 4.18 GPU Memory structures. 所有网格(向量化循环)共享GPU内存(全局内存)

- *GPU Memory* (*Global Memory*) shared by all Grids (vectorized loops),
- *Local Memory* (*Shared Memory*) shared by all threads of SIMD instructions within a thread block (body of a vectorized loop). 本地内存(共享内存)，由线程块(向量化循环体)中的所有SIMD指令线程共享。
- *Private Memory* private to a single CUDA thread.

私有内存私有到一个单独的cuda thread。

Page 61 Terminology (GPU)

术语

Terminology (GPU hardware)

- *Threads of SIMD instructions* SIMD指令的线程
  - Each has its own PC
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Keeps track of up to 48 threads of SIMD instructions
    - Hides memory latency

每个人都有自己的PC
线程调度程序使用记分板来调度
没有线程之间的数据依赖关系!
跟踪多达48个线程的SIMD指令
隐藏内存延迟

- *Thread block scheduler* 线程阻塞调度
  - Schedule thread blocks to SIMD processors, each thread block is split to sub-blocks (warp) (e.g. 32 CUDA threads / warp) and assigned to SIMD threads running on a SIMD processor.

为SIMD处理器调度线程块，每个线程块被分割成子块(扭曲)(例如32个CUDA线程/扭曲)，并分配给运行在SIMD处理器上的SIMD线程。

- Within each SIMD processor:
  - 16 or 32 SIMD lanes
  - Wide and shallow compared to vector processors

在每个SIMD处理器内:
16或32 SIMD车道
比向量处理器更宽、浅

Page 68-69 Conditional branching 课本 Page 323 - 326 4.4.4

Conditional branching

- GPU branch hardware uses: GPU分支硬件用
  - Internal masks 内部遮罩
  - Branch synchronization stack 分支同步栈
    - Entries consist of masks for each SIMD lane 条目由每个SIMD lane的遮罩组成
    - i.e. which threads commit their results (all threads execute) 哪些线程提交它们的结果(所有线程都执行)
  - Instruction markers to manage when a branch diverges into multiple execution paths 指令标记 当一个分支分叉到多个执行路径时，要管理的指令标记
    - Push on divergent branch 推动不同的分支
  - …and when paths converge 当路径收敛时
    - Act as barriers 作为障碍
    - Pops stack 弹出堆栈
- Per-thread-lane 1-bit predicate register, specified by programmer 由程序员指定的每线程1位谓词寄存器

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
```

Note: a thread has 64 vector components, each a 32 bit floating point