《计算机系统结构》课程直播

2020. 5.12

听不到声音请及时调试声音设备，可以下课后补签到

请将ZOOM名称改为“姓名”；

本节内容

- 数据级并行性
  - 回顾：Vector, SIMD Architectures
  - 计算机系统结构分类：SISD、SIMD、MIMD
  - GPU初步介绍

From : H&P Computer Architecture: A Quantitative Approach, Fifth Edition, （5th edition)

Data-Level Parallelism

- 数据级并行的研究动机
  - 传统指令级并行技术的问题
  - SIMD结构的优势

- SIMD体系结构
  - 向量处理模型
    - 起源于超级计算机的应用续签
  - 通用处理器的扩展
    - 面向多媒体应用的SIMD指令集扩展
  - 协处理单元 GPU
    - GPU简介
    - GPU的编程模型
    - GPU的存储系统

SIMD的向量处理模型

SISD 对比 SIMD

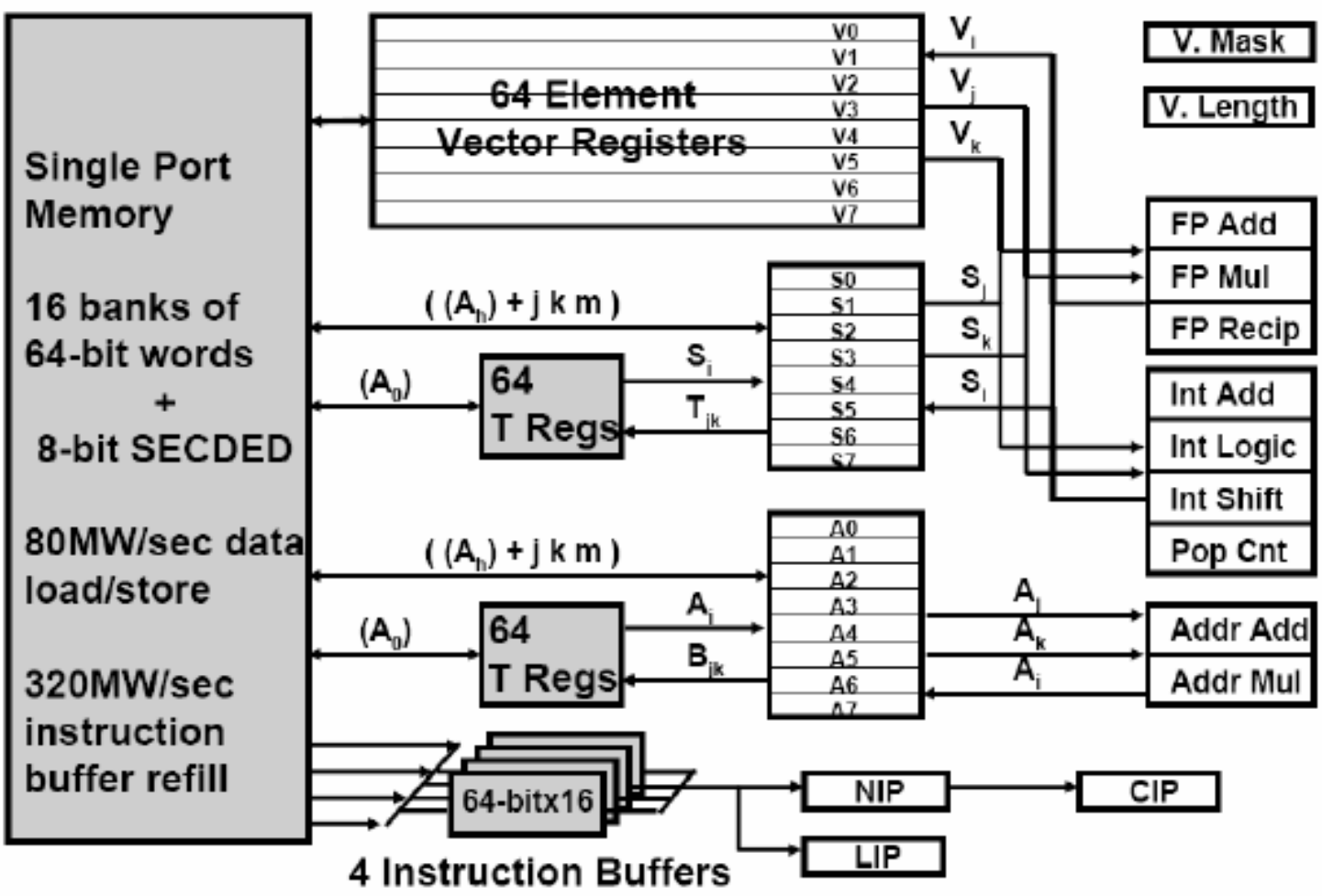
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] * B[i];
```

```
# Scalar Assembly Code
  LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  MUL.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

```
# Vector Assembly Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  MULVV.D V3, V1, V2
  SV V3, R3
```

向量处理机的起源: 超级计算机

Cray－1（1976）



内存体读写：50ns； 处理器周期：12.5ns（80MHz）

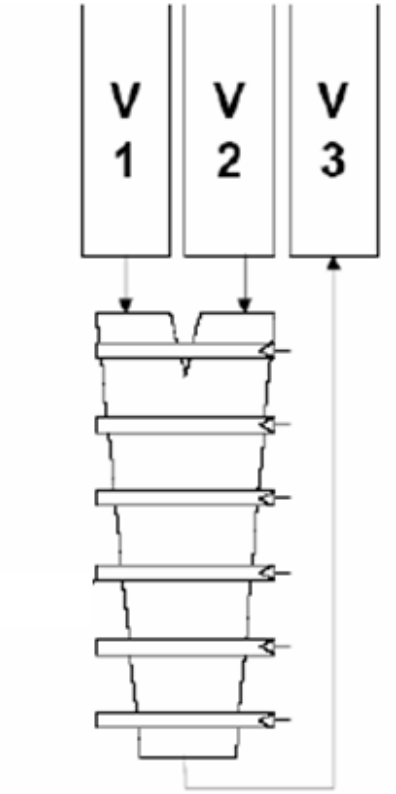
向量体系结构

- 向量处理机**基本概念**
  - 基本思想：两个向量的对应分量进行运算，产生一个结果向量
- 向量处理机**基本特征**
  - VSIW- 一条指令包含多个操作
  - 单条向量指令内所包含的操作相互独立
  - 以已知模式访问存储器-多体交叉存储系统
  - 控制相关少
- 向量处理机**基本结构**
  - **向量指令并行执行**
  - **向量运算部件的执行方式-流水线方式**
  - **向量部件结构-多“道”结构-多条运算流水线**
- 向量处理机性能优化
  - 链接技术
  - 条件执行

Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations

- Simplifies control of deep pipeline because elements in vector are independent
  - no data hazards!
  - no bypassing needed

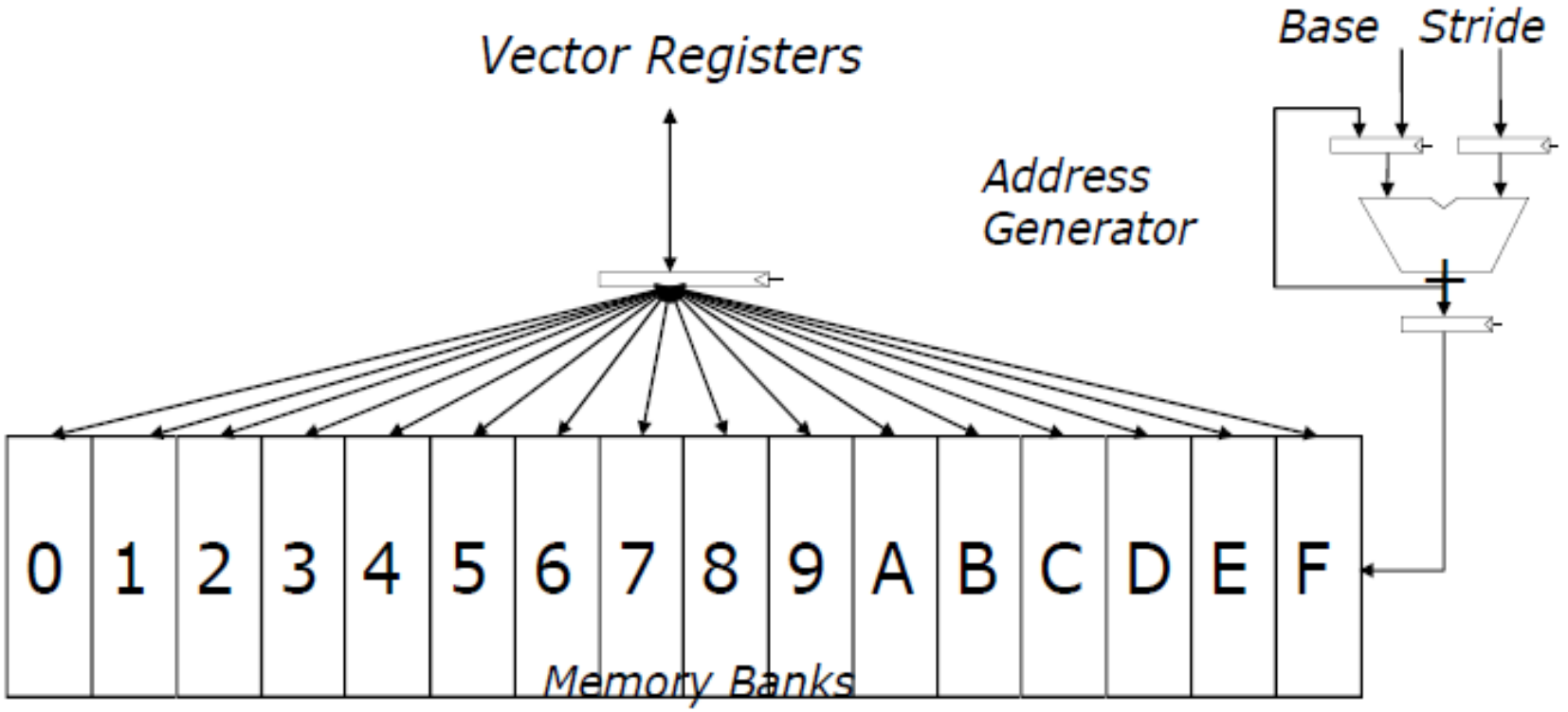


V3 <- v1 * v2

Interleaved Vector Memory System

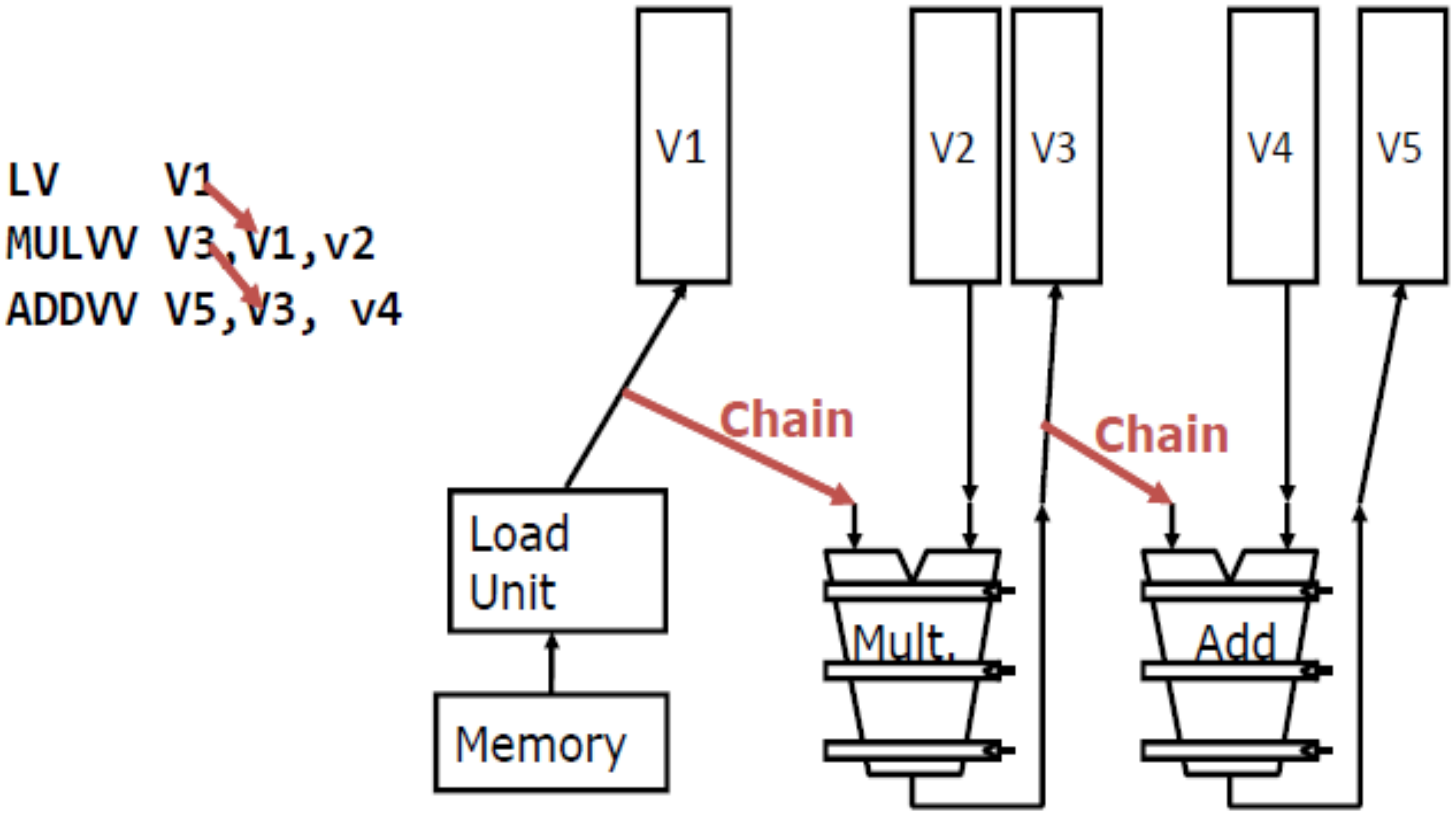
Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- *Bank busy time*: Time before bank ready to accept next request



Vector Chaining

- Vector version of register bypassing
  - introduced with Cray-1



Vector Conditional Execution

- Problem: Want to vectorize loops with conditional code:

  for (i=0; i<N; i++)

  if (A[i]>0) then A[i] = B[i];

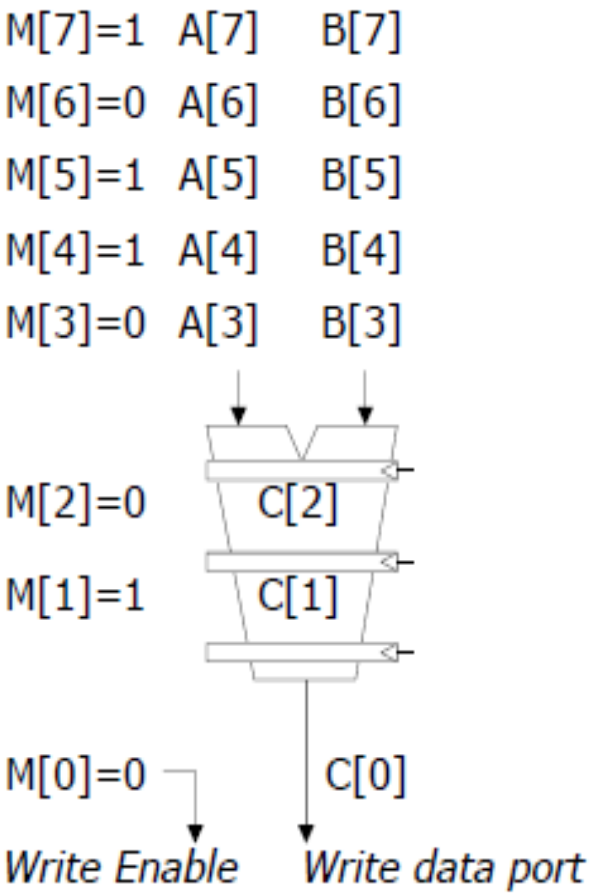
- Solution: Add vector mask (or flag) registers
- Code example:

```
CVM              # Turn on all elements
LV VA, RA        # Load entire A vector
SGTVS.D VA, F0   # Set bits in mask register where A>0
LV VA, RB        # Load B vector into A under mask
SV VA, RA        # Store A back to memory under mask
```

Masked Vector Instructions

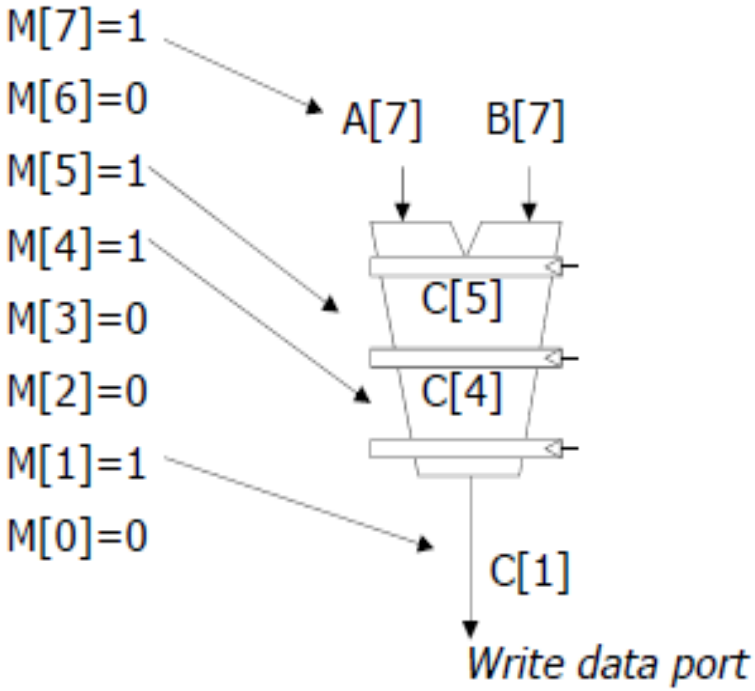
Simple Implementation

- execute all N operations, turn off result writeback according to mask



Density-Time Implementation

- scan mask vector and only execute elements with non-zero masks



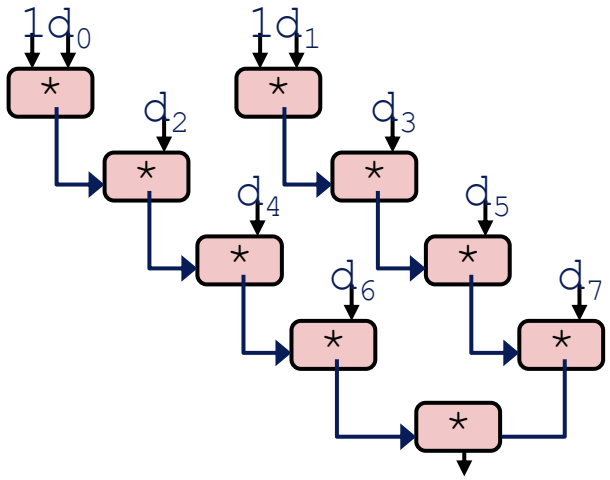
通用处理器的向量化扩展

回顾：一个例子：循环展开、 独立计算(2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
     x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

循环展开、 独立计算

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



改变之处:

- 两个独立的计算“流”

性能分析:

N 个元素, D cycles latency/op

(N/2+1)*D cycles, **CPE = D/2**

循环展开、 独立计算: Double *

- Case
  - Intel Haswell
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 0.50

FP *	Unrolling Factor L							
Accumulators K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

通用处理器的 DLP 扩展

- 如何让浮点运算部件忙起来?
- 需要足够多的数据并行性!
- 足够填充浮点运算部件的流水线

Multimedia Extensions (aka SIMD extensions)

- 在已有的ISA中添加一些向量长度很短的向量操作指令
- 将已有的 64-bit 寄存器拆分为 2x32b or 4x16b or 8x8b
- 新设计具有较宽的寄存器
  - 128b for PowerPC Altivec, Intel SSE2/3/4
  - 256b for Intel AVX (Advanced Vector Extensions)
- 单条指令可实现寄存器中所有向量元素的操作

Intel Pentium MMX Operations

- idea: 一条指令操作同时作用于不同的数据元
  - 全阵列处理
  - 用于多媒体操作

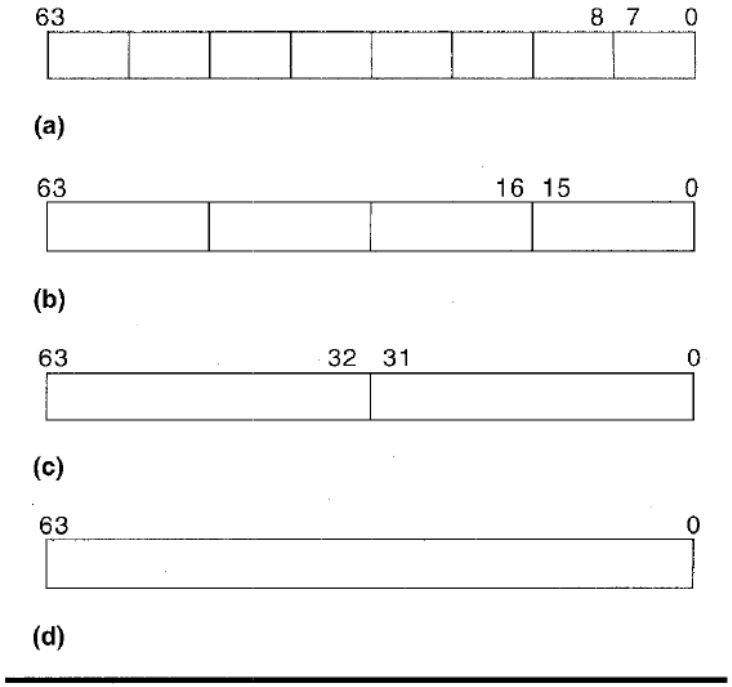
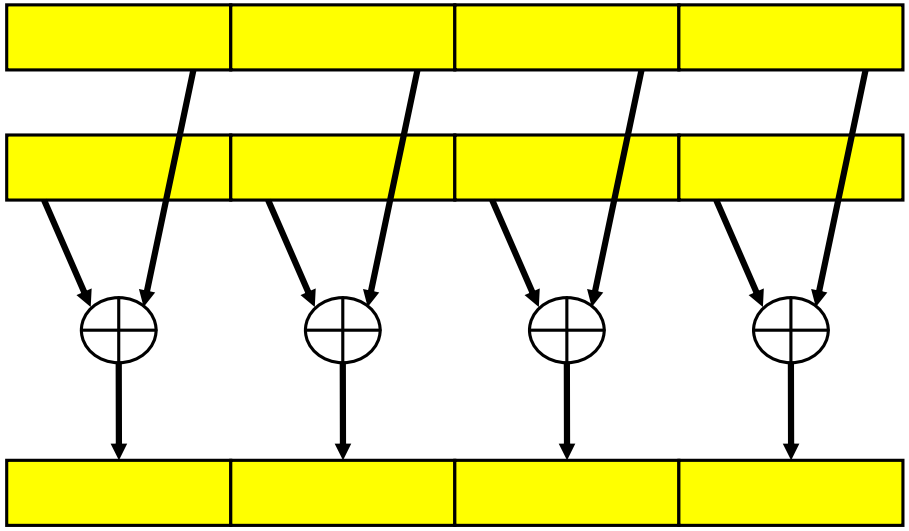


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

- No VLEN register
- Opcode determines data type:
  - 8 8-bit bytes
  - 4 16-bit words
  - 2 32-bit doublewords
  - 1 64-bit quadword
- Stride always equal to 1.

扩展的SIMD指令

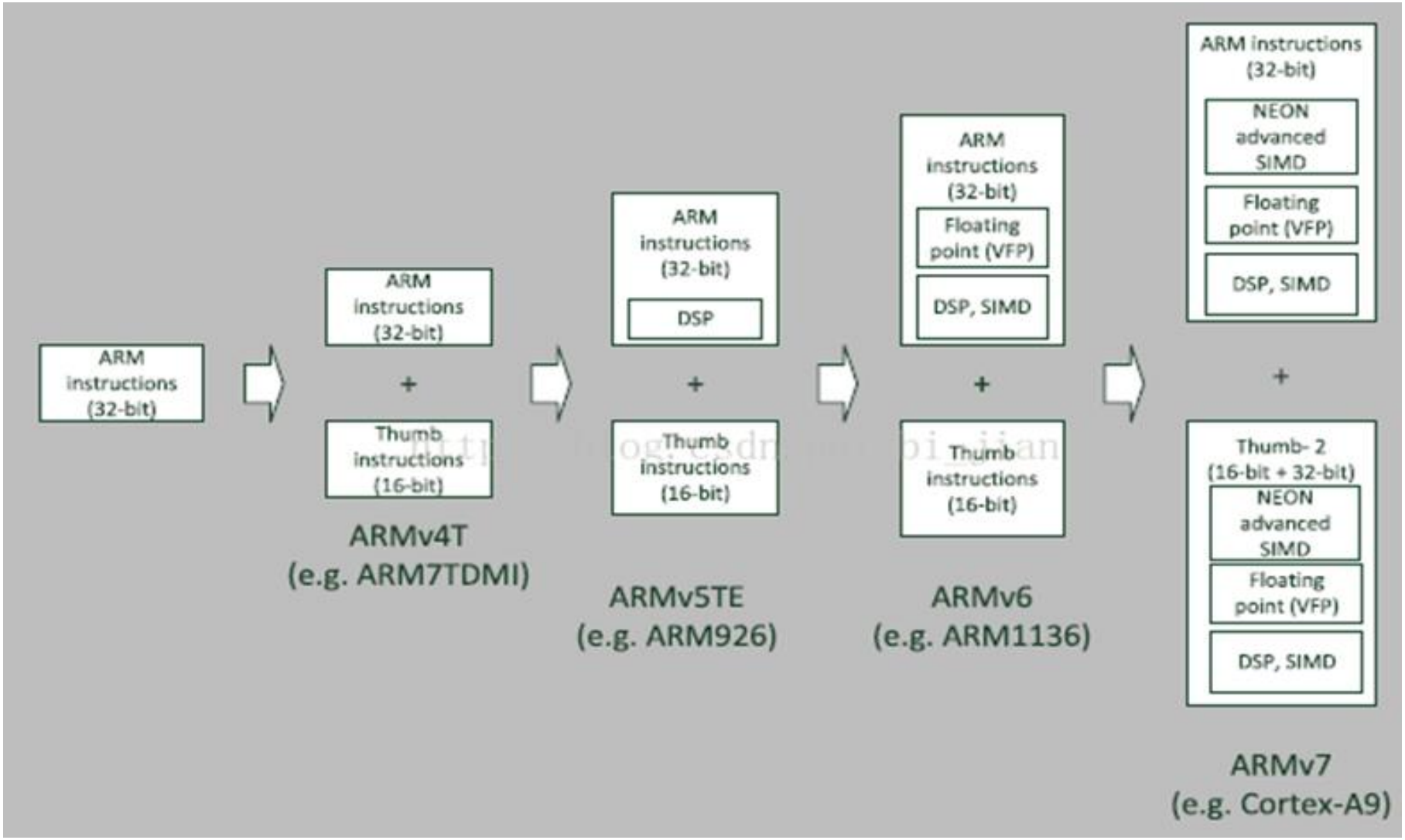
- 对原有指令集进行SIMD扩展
- 例如 x86 SSE 指令: 对128-bit 的数据并行操作:
  - 4x32 bits or 2x64 bits (4-wide float vectors)



SIMD on modern CPUs

- x86
  - Intel and AMD: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2
  - AVX instructions: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors) - 2011，Sandy Bridge架构开始支持
  - 目前: AVX 512 提供 512位长的 vectors， 2016.10 宣布
  - “It is made easier by the fact that companies like Google have open sourced their code such as Tensorflow.”
- PowerPC
  - AltiVEC/VMX: 128b
- ARM
  - NEON: 128b
  - 2016.8： Scalable Vector Extensions (SVE): SVE将作为ARMv8-A指令集的可选扩展，支持最低128-bit、最高2048-bit

ARM指令集中的SIMD(NEON)



Example SIMD Code

- Example DXPY:

```
        L.D       F0,a          ;load scalar a
        MOV       F1, F0        ;copy a into F1 for SIMD MUL
        MOV       F2, F0        ;copy a into F2 for SIMD MUL
        MOV       F3, F0        ;copy a into F3 for SIMD MUL
        DADDIU    R4,Rx,#512    ;last address to load
Loop:   L.4D F4,0[Rx]           ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D    F4,F4,F0      ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D      F8,0[Ry]      ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D    F8,F8,F4      ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D      0[Ry],F8      ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU    Rx,Rx,#32     ;increment index to X
        DADDIU    Ry,Ry,#32     ;increment index to Y
        DSUBU     R20,R4,Rx     ;compute bound
        BNEZ      R20,Loop      ;check if done
```

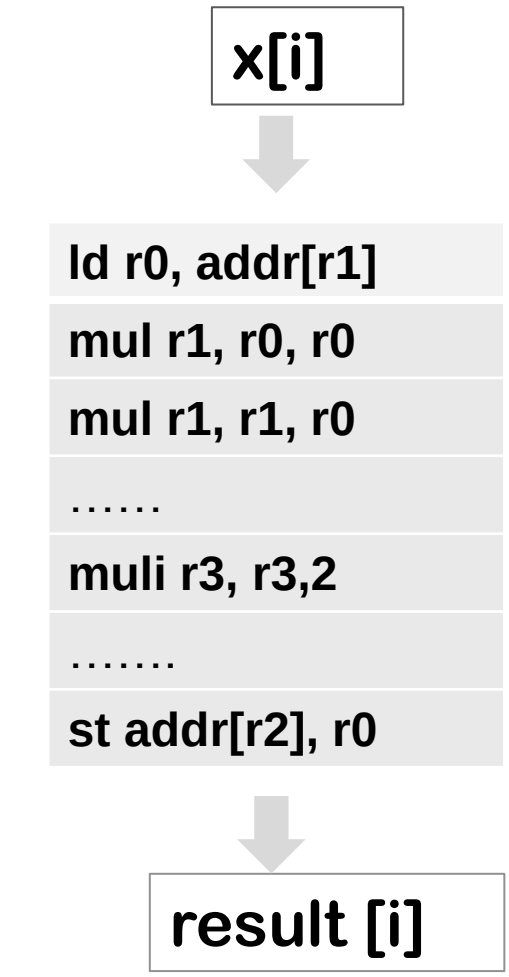
Using Vectors in Your Code

- Use “intrinsic” (固有的）functions and data types
  - For example: _mm_mul_ps() and “__m128” datatype
- Use vector data types
  - typedef double v2df __attribute__ ((vector_size (16)));
- Use a library someone else wrote
  - Let them do the hard work
  - Matrix and linear algebra packages
- Let the compiler do it (automatic vectorization, with feedback)
  - GCC’s “-ftree-vectorize” option, -ftree-vectorizer-verbose=**n**
  - Limited impact for C/C++ code (old, hard problem)

举例：Using Vectors in Your Code

例如： 泰勒级数展开计算 $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

```
void sinx(int N, int terms, float* x, float* result)
{
  for (int i=0; i<N; i++)
  {
    float value = x[i];
    float numer = x[i] * x[i] * x[i];
    int denom = 6;       // 3!
    int sign = -1;

    for (int j=1; j<=terms; j++)
    {
      value += sign * numer / denom;
      numer *= x[i] * x[i];
      denom *= (2*j+2) * (2*j+3);
      sign *= -1;
    }

    result[i] = value;
  }
}
```



“显式的 SIMD”:

由程序员显式声明数据并行性，调用 intrinsics 中的库函数

编译器产生对应的SIMD指令

```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float* sinx)
{
  float three_fact = 6; // 3!
  for (int i=0; i<N; i+=8)
  {
    __m256 origx = _mm256_load_ps(&x[i]);
    __m256 value = origx;
    __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
    __m256 denom = _mm256_broadcast_ss(&three_fact);
    int sign = -1;
   for (int j=1; j<=terms; j++)
   {    // value += sign * numer / denom
         __m256 tmp =
_mm256_div_ps(_mm256_mul_ps(_mm256_broadcast_ss(sign),numer),denom)
;
        value = _mm256_add_ps(value, tmp);
        numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
        denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) *
(2*j+3)));
         sign *= -1;
   }
  _mm256_store_ps(&sinx[i], value);
}
}
```

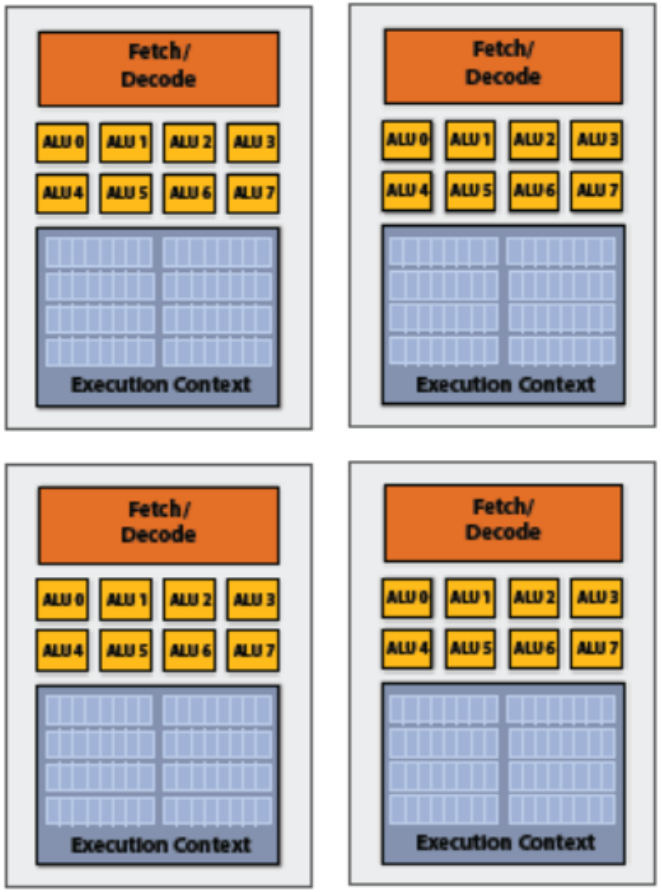
```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float* sinx)
{
  float three_fact = 6; // 3!
  for (int i=0; i<N; i+=8)
  {
    __m256 origx = _mm256_load_ps(&x[i]);
    __m256 value = origx;
    __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
    __m256 denom = _mm256_broadcast_ss(&three_fact);
    int sign = -1;
   for (int j=1; j<=terms; j++)
   {    // value += sign * numer / denom
         __m256 tmp =
_mm256_div_ps(_mm256_mul_ps(_mm256_broadcast_ss(sign),numer),denom)
;
         value = _mm256_add_ps(value, tmp);
         numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
         denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) *
(2*j+3)));
          sign *= -1;
   }
  _mm256_store_ps(&sinx[i], value);
}
}
```

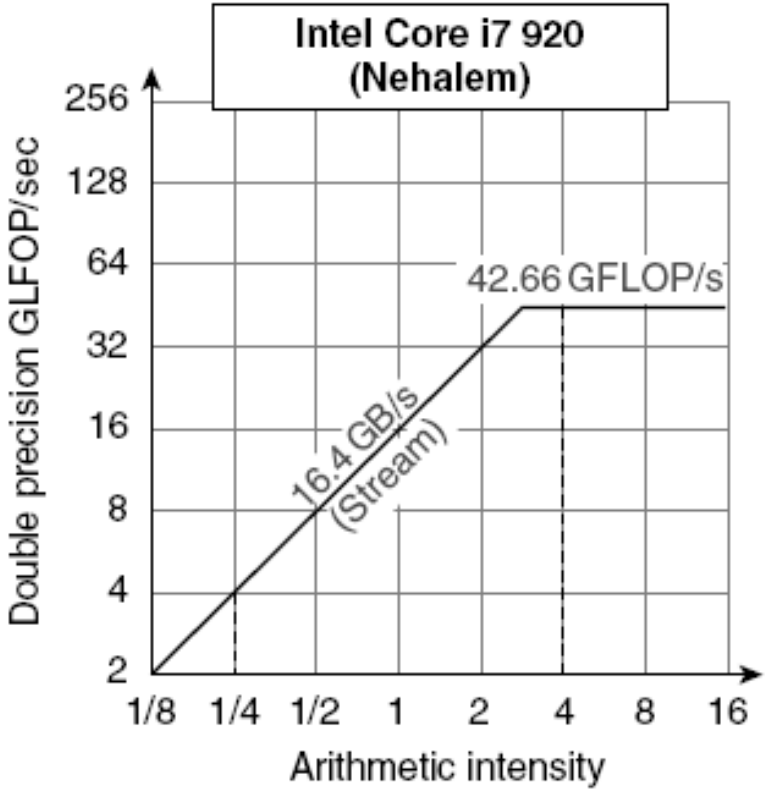
```
vloadps xmm0, addr[r1]
vmulps xmm1, xmm0, xmm0
vmulps xmm1, xmm1, xmm0
…….
vstoreps addr[xmm2], xmm0
```

编译程序: 使用一个256-位的向量寄存器，同时处理八个数据单元

SIMD CPU Example: Intel Core i7

4 cores， 8 SIMD ALUs per core （TLP， DLP 都支持）





I7 875K，单精度：86.83， 双精度：45.46 GFLops

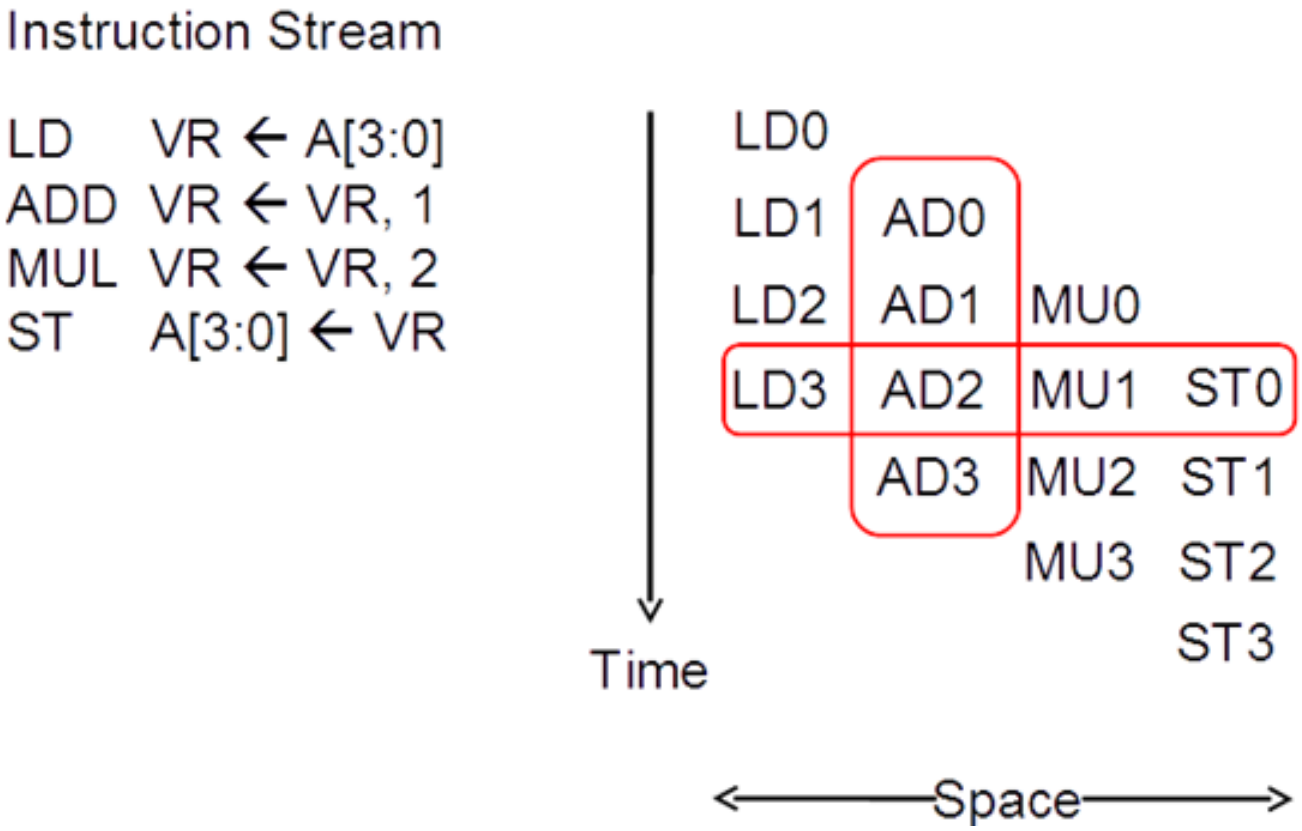
“arithmetic intensity” — ratio of math operations to data access operations

By the numbers: CPUs vs GPUs

	Intel Xeon Platinum 8168 “Skylake”	Nvidia Tesla P100	Intel Xeon Phi 7290F
frequency	2.7 GHz	1.3 GHz	1.5 GHz
cores / threads	24 / 48	56 (“3584”) / 10Ks	72 / 288
RAM	768 GB	16 GB	384 GB
DP TFLOPS	1.0	4.7	3.5
Transistors	>5B ?	15.3B	>5B ?
Price	$5,900	$6,000	$3,400

SISD vs. SIMD vs. MIMD ?

Vector/SIMD processor



对比 VLIW

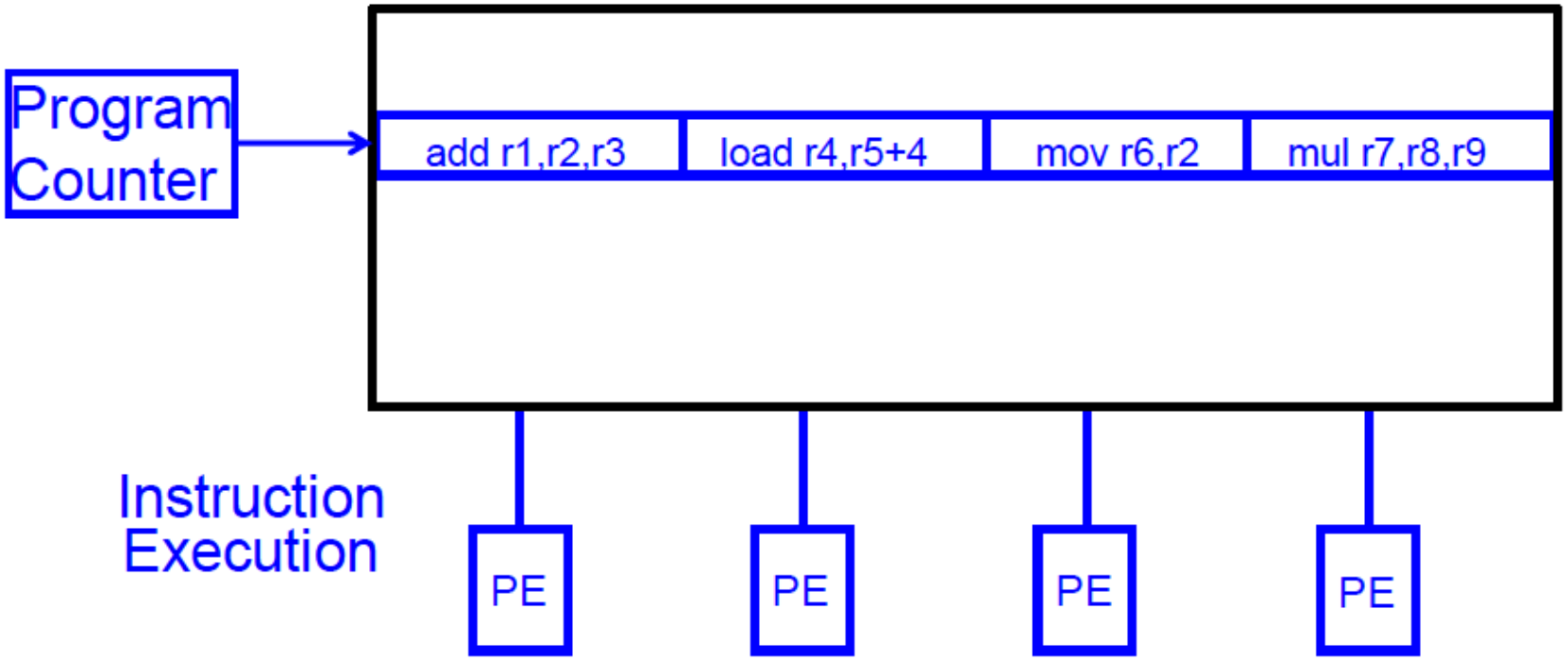
```
lp:     lw      $t0,0($s1)

        lw      $t1,-4($s1)
        lw      $t2,-8($s1)
        lw      $t3,-12($s1)
        addu    $t0,$t0,$s2
        addu    $t1,$t1,$s2
        addu    $t2,$t2,$s2
        addu    $t3,$t3,$s2
        sw      $t0,0($s1)
        sw      $t1,-4($s1)
        sw      $t2,-8($s1)
        sw      $t3,-12($s1)
        addi    $s1,$s1,-16
        bne     $s1,$0,lp
```

ALU or branch	Data transfer	CC
addi $s1,$s1,-16	lw $t0,0($s1)	1
	lw $t1,12($s1)	2
addu $t0,$t0,$s2	lw $t2,8($s1)	3
addu $t1,$t1,$s2	lw $t3,4($s1)	4
addu $t2,$t2,$s2	sw $t0,16($s1)	5
addu $t3,$t3,$s2	sw $t1,12($s1)	6
	sw $t2,8($s1)	7
bne $s1,$0,lp	sw $t3,4($s1)	8

- 14 条指令8个周期,
- CPI =0.57 (最佳情况是0.5)

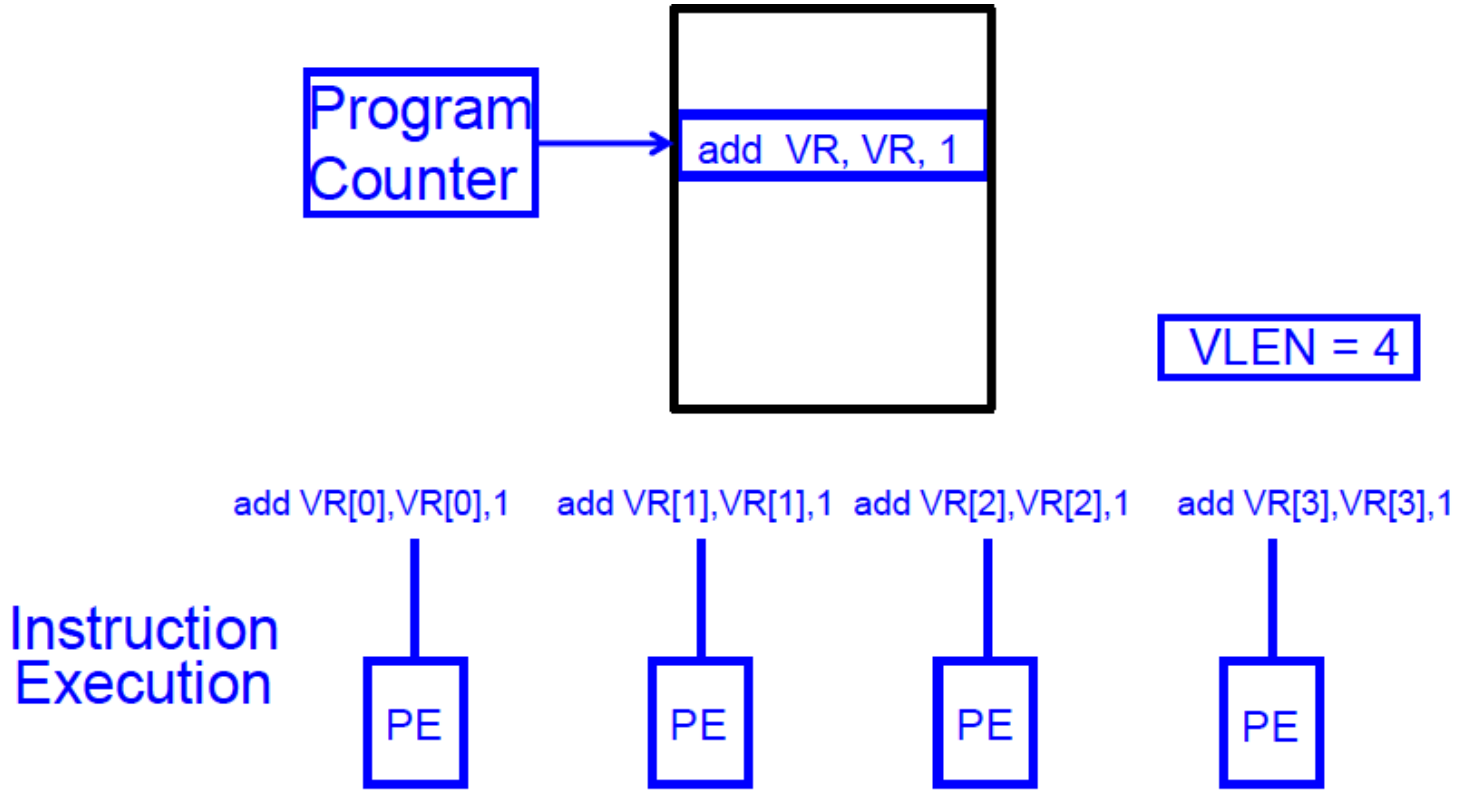
VLIW and Superscalar

- VLIW: 多个独立的操作由编译器封装在一起
- SuperScalar：多个独立的操作由硬件调度单元动态发射



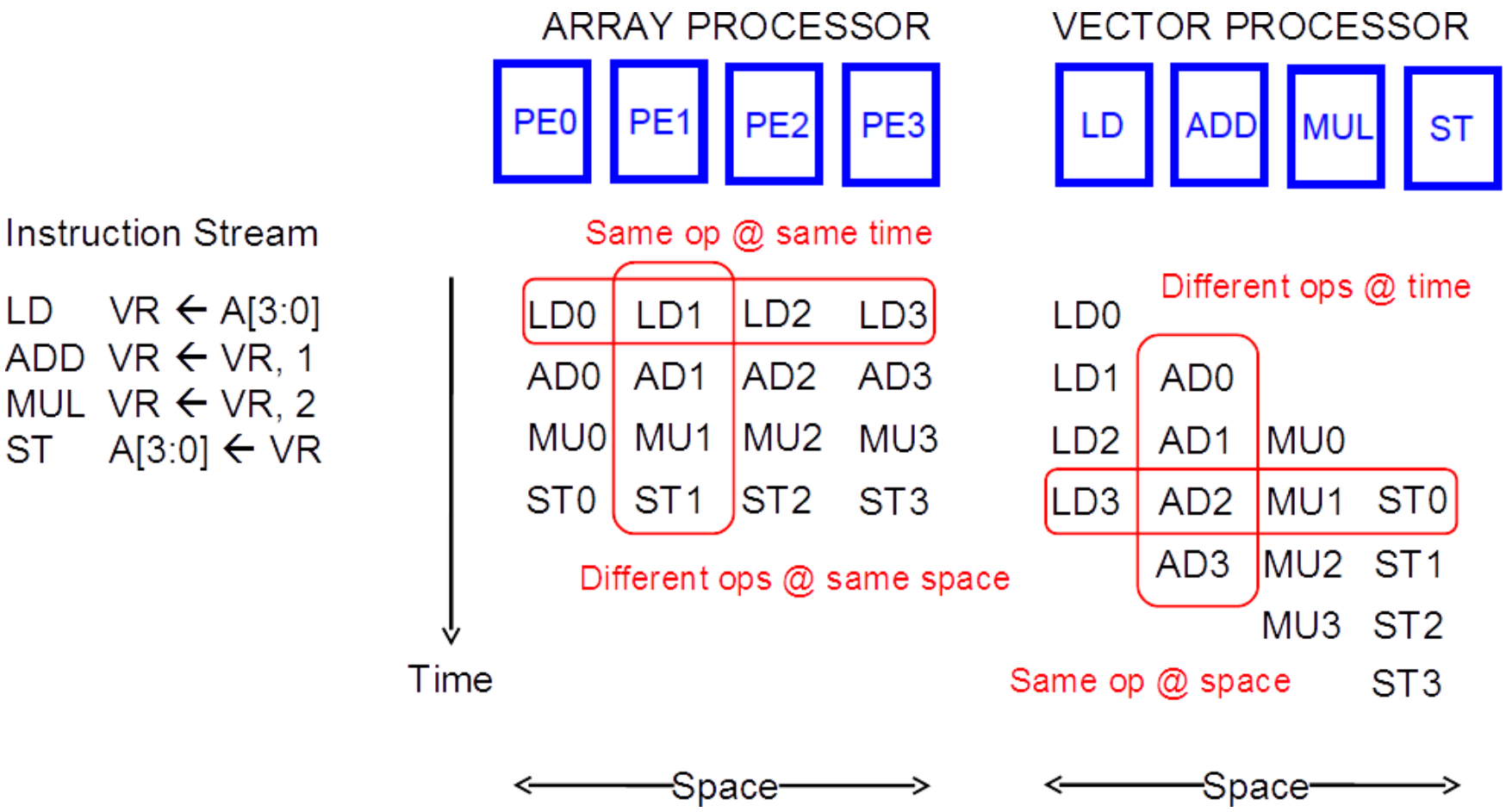
SIMD Array Processing

Array processor：又称为SIMD处理器。一个由**多个处理单元**构成的**阵列**，用**单一的控制部件**来控制多个处理单元对**各自的数据**进行**相同的运算**和操作。



- Array processor: 单个操作作用在多个不同的数据元素上

SIMD Array vs. Vector Processors



如何区分?

- Flynn计算体系结构分类方法

- 根据指令流、数据流进行分类
  - 指令流：机器执行的指令序列
  - 数据流：指由指令流调用的数据序列，包括输入数据和中间结果，但不包括输出数据。

- SISD、SIMD 、MIMD
  - 单指令流单数据流（SISD）：传统的计算机包含单个CPU，它从存储在内存中的程序那里获得指令，并作用于单一的数据流
  - 单指令流多数据流（SIMD）：单个的指令流作用于多于一个的数据流上。
  - 多指令流单数据流（MISD）：实际上很少见。冗余多用于容错系统。
  - 多指令流多数据流（MIMD）：类似于多个SISD系统。实际上，常见例子是多处理器计算机，如企业级服务器。

Graphics Processing Units

GPU

Graphics Processing Units (GPUs)

- **早期的GPU**是指带有高性能浮点运算部件、可高效生成3D图形的具有固定功能的专用设备 (mid-late 1990s)
  - 让PC机具有类似工作站的图形功能
  - **用户可以配置图形处理流水线，但不是真正的对其编程**

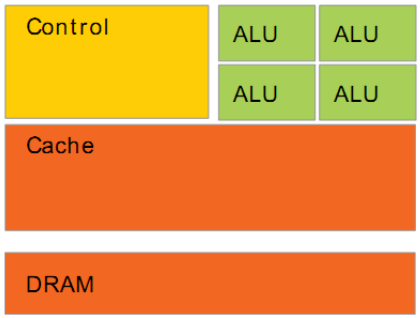
- 2001-2005，GPU加入了越来越多的可编程性
  - 例如新的语言 Cg可用来编写一些小的程序处理图形的顶点或像素，是Windows DirectX的变体
  - 大规模并行（针对每帧上百万顶点或像素）但非常受限于编程模型

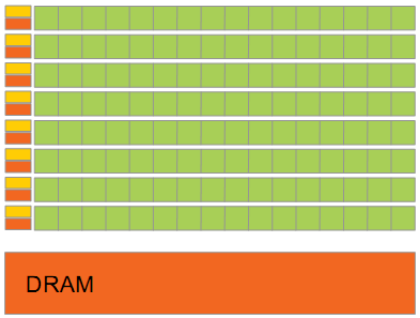
General-Purpose GPUs (GP-GPUs)

- 2006年, Nvidia 的 GeForce 8800 GPU 支持一种新的编程语言: CUDA
  - “Compute Unified Device Architecture”
  - 随后工业界推出OpenCL，与CUDA具有相同的ideas, 但独立于供应商
- Idea: 针对通用计算，发挥GPU计算的高性能和存储器的高带宽来加速一些通用计算中的核心（Kernels）
- 一种协处理器模型（GPU作为附加设备）：Host CPU发射数据并行的kernels 到GP-GPU上运行
- 我们仅讨论Nvidia CUDA样式的简化版本，仅考虑GPU的计算核部分，不涉及图形加速部分

Using CPU+GPU Architecture

- CPU+GPU异构多核系统
  - 针对每个任务选择合适的处理器和存储器
- 通用CPU 适合执行一些串行的线程
  - 串行执行快
  - 带有cache，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

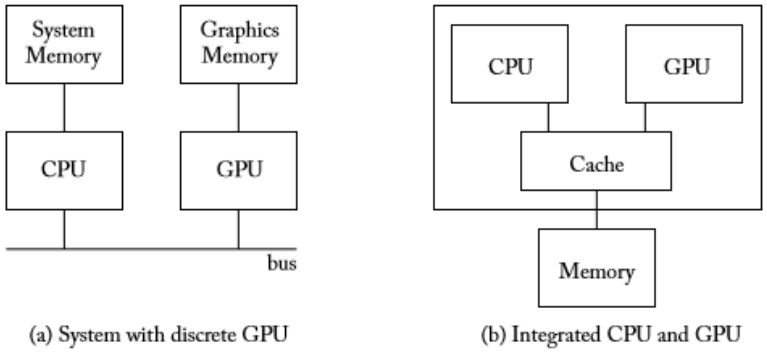
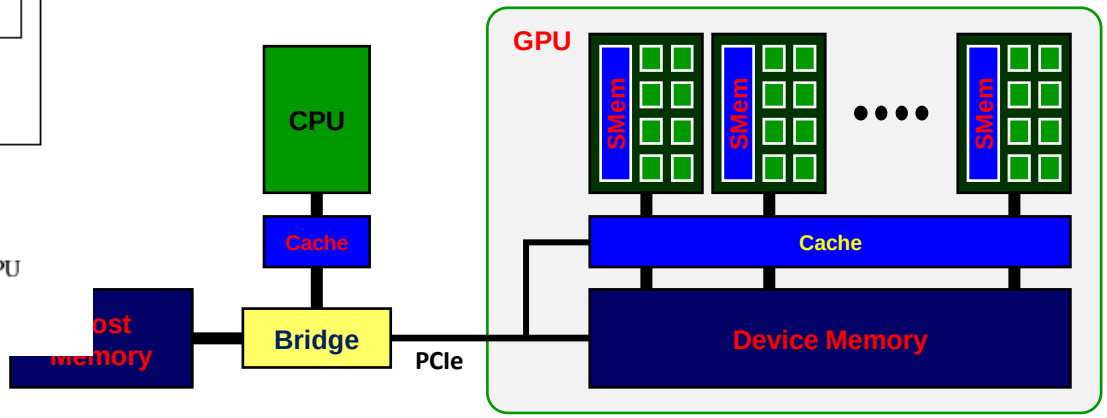


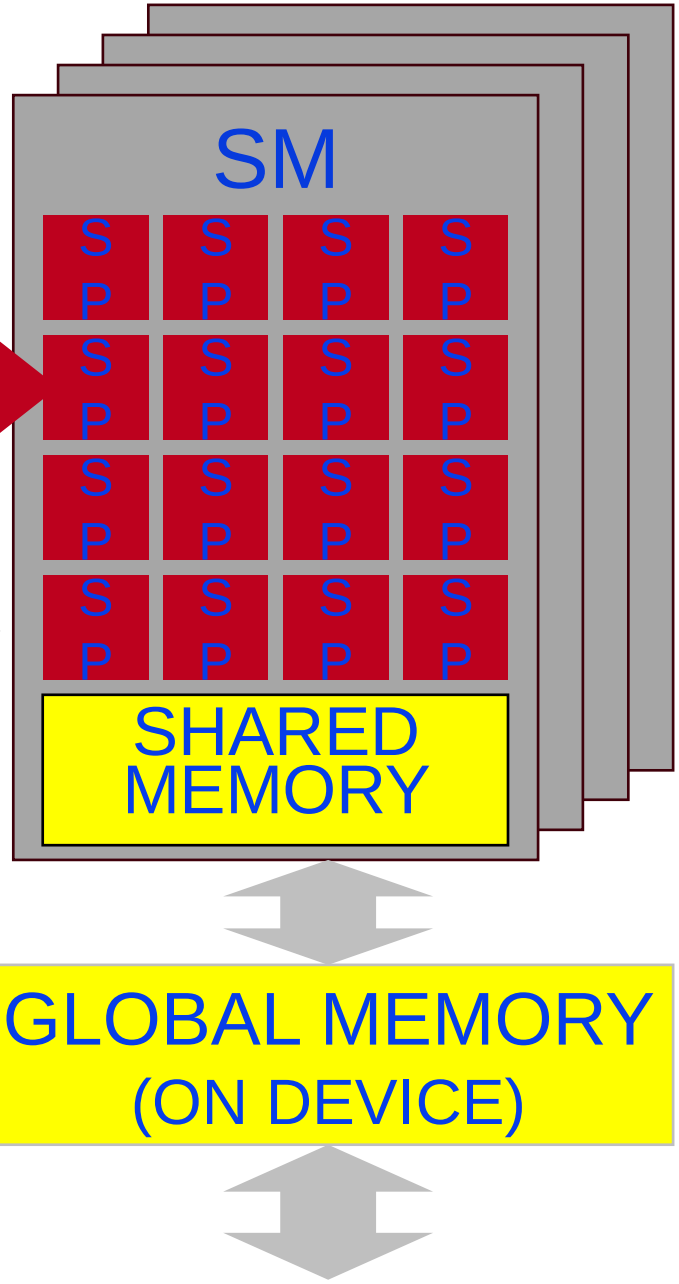
Figure 1.1: GPU computing systems include CPUs.



GPU: a multithreaded coprocessor

SP: scalar processor
'CUDA core'

Executes one thread



SM
streaming multiprocessor
32xSP (or 16, 48 or more)

Fast local '**shared memory**'
(shared between SPs)
16 KiB (or 64 KiB)

GPUs are SIMD Engines Underneath

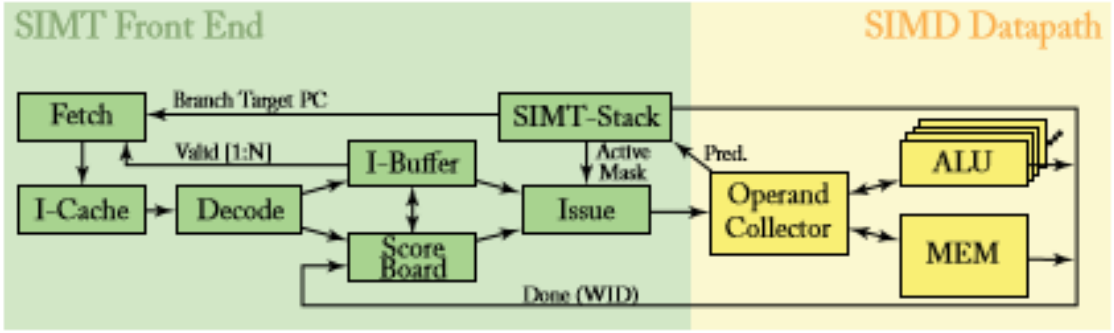


Figure 3.1: Microarchitecture of a generic GPGPU core.

- Programming Model (Software) vs Execution Model (Hardware)
- 编程模型指程序员如何描述应用（从程序员角度看到的机器模型）
  - 例如, 顺序模型 (von Neumann), 数据并行(SIMD)模型、多线程模型 (MIMD, SPMD), …
- 执行模型指硬件底层如何执行代码
  - 例如, 乱序执行、向量机、数据流处理机、多处理机、多线程处理机等
- 执行模型与编程模型可以差别很大
  - 例如., 顺序模型可以在乱序执行的处理器上执行。
  - SPMD 模型可以用SIMD处理器实现 (a GPU)

Simplified CUDA Programming Model

- 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]; }
```

SPMD

- Single procedure/program, multiple data
  - 它是一种编程模型而不是计算机组织
- 每个处理单元执行同样的过程，处理不同的数据
  - 这些过程可以在程序中的某个点上同步，例如 barriers

- 多条指令流执行相同的程序
  - 每个程序/过程
    - 操作不同的数据
    - 运行时可以执行不同的控制流路径
  - 许多科学计算应用以这种方式编程，运行在MIMD硬件结构上 (multiprocessors)
  - 现代 GPUs 以这种类似的方式编程，运行在SIMD硬件上

A GPU is a SIMD (SIMT) Machine

- GPU不是用SIMD指令编程

- 使用线程 (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

下一节

- GPU 硬件执行模型
- GPU存储模型

From : H&P Computer Architecture: A Quantitative Approach, Fifth Edition, （5th edition)

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

谢　谢！