



第3讲 SIMD编程



提纲

- SIMD概念
- SIMD并行的问题
- SSE/AVX编程



提纲

- SIMD概念
- SIMD并行的问题
- SSE/AVX编程

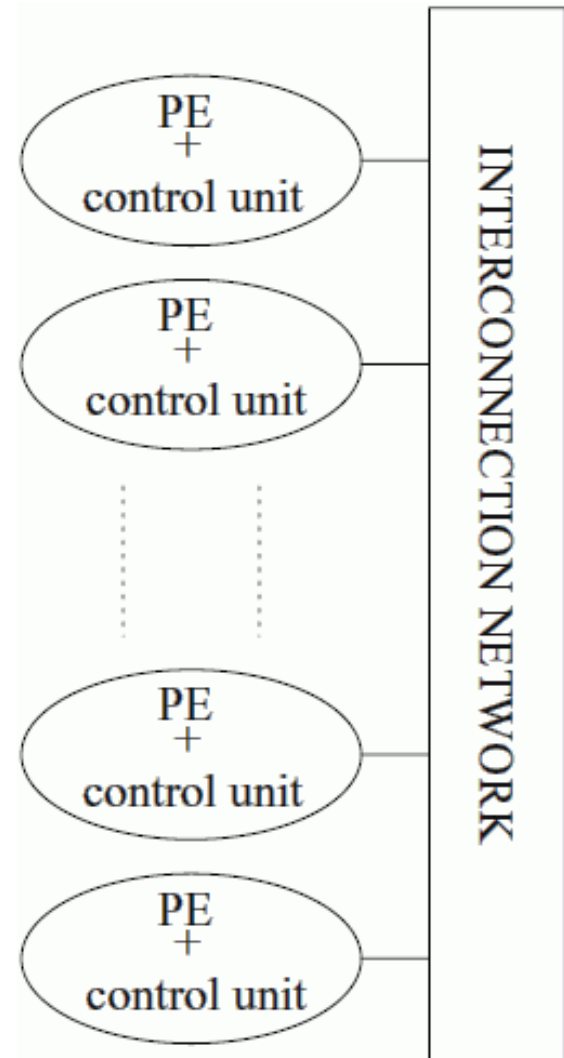
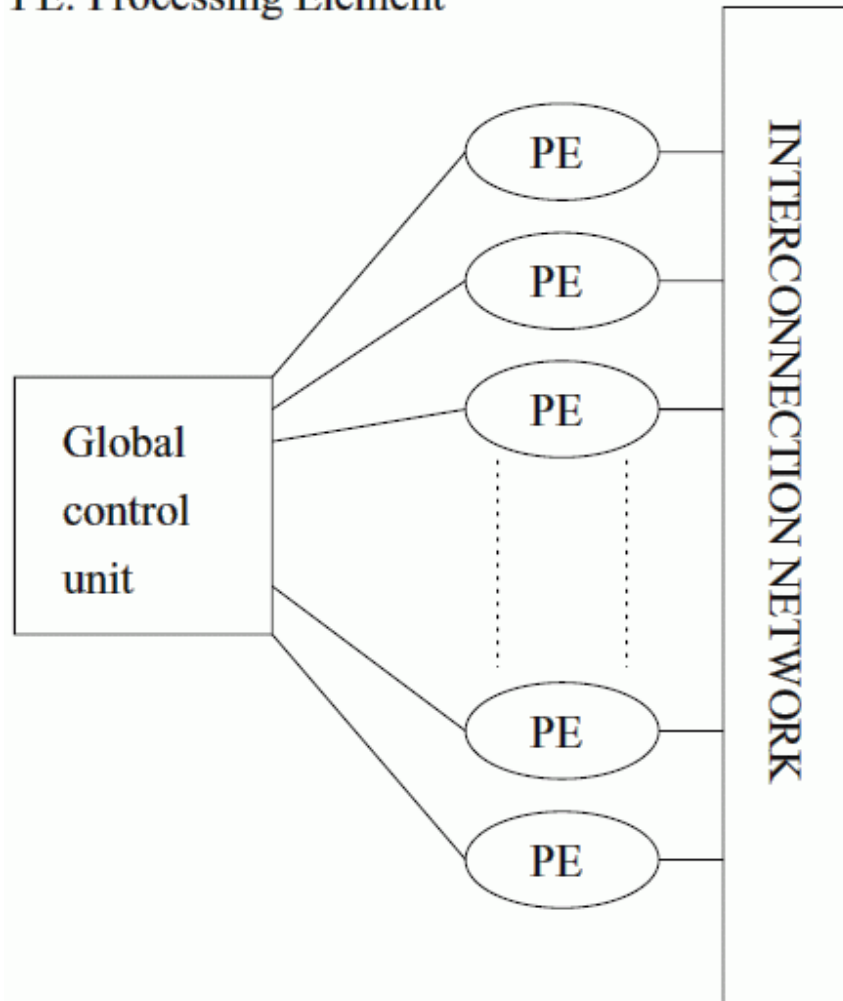


回顾：主要的并行控制机制

Name	Meaning	Examples
Single Instruction, Multiple Data (SIMD)	A single thread of control, same computation applied across "vector" elts	Array notation as in Fortran 90: $A[1:n] = A[1:n] + B[1:n]$
Multiple Instruction, Multiple Data (MIMD)	Multiple threads of control, processors periodically synch	Parallel loop: <code>forall (i=0; i<n; i++)</code>
Single Program, Multiple Data (SPMD)	Multiple threads of control, but each processor executes same code	Processor-specific code: <code>if (\$myid == 0) { }</code>

SIMD和MIMD的差别

PE: Processing Element



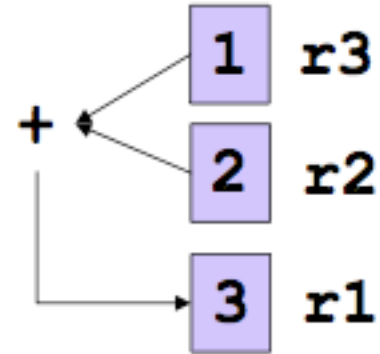


SIMD编程概述

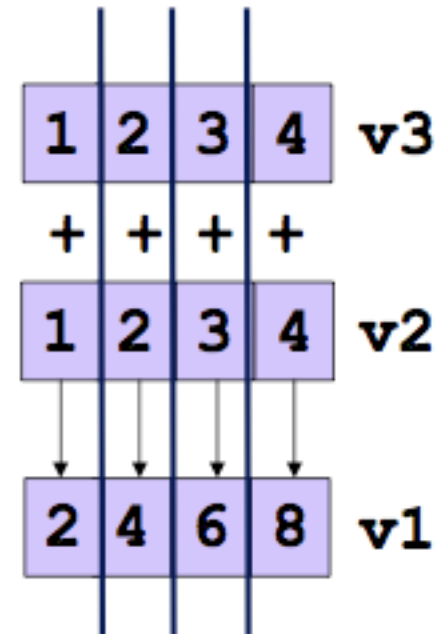
- 向量计算机
- 早期的SIMD超级计算机
- 当前的SIMD架构
 - 多媒体扩展：SSE、AVX
 - 图形和游戏处理器：CUDA
 - 协处理器：Xeon Phi
- 没有占压倒优势的SIMD编程模型
 - 向量计算机都是科学家用编程
 - 多媒体扩展指令集多是系统程序员在用
 - GPU多是游戏开发者、大数据分析人员使用

标量 vs. SIMD (多媒体扩展)

Scalar: `add r1,r2,r3`



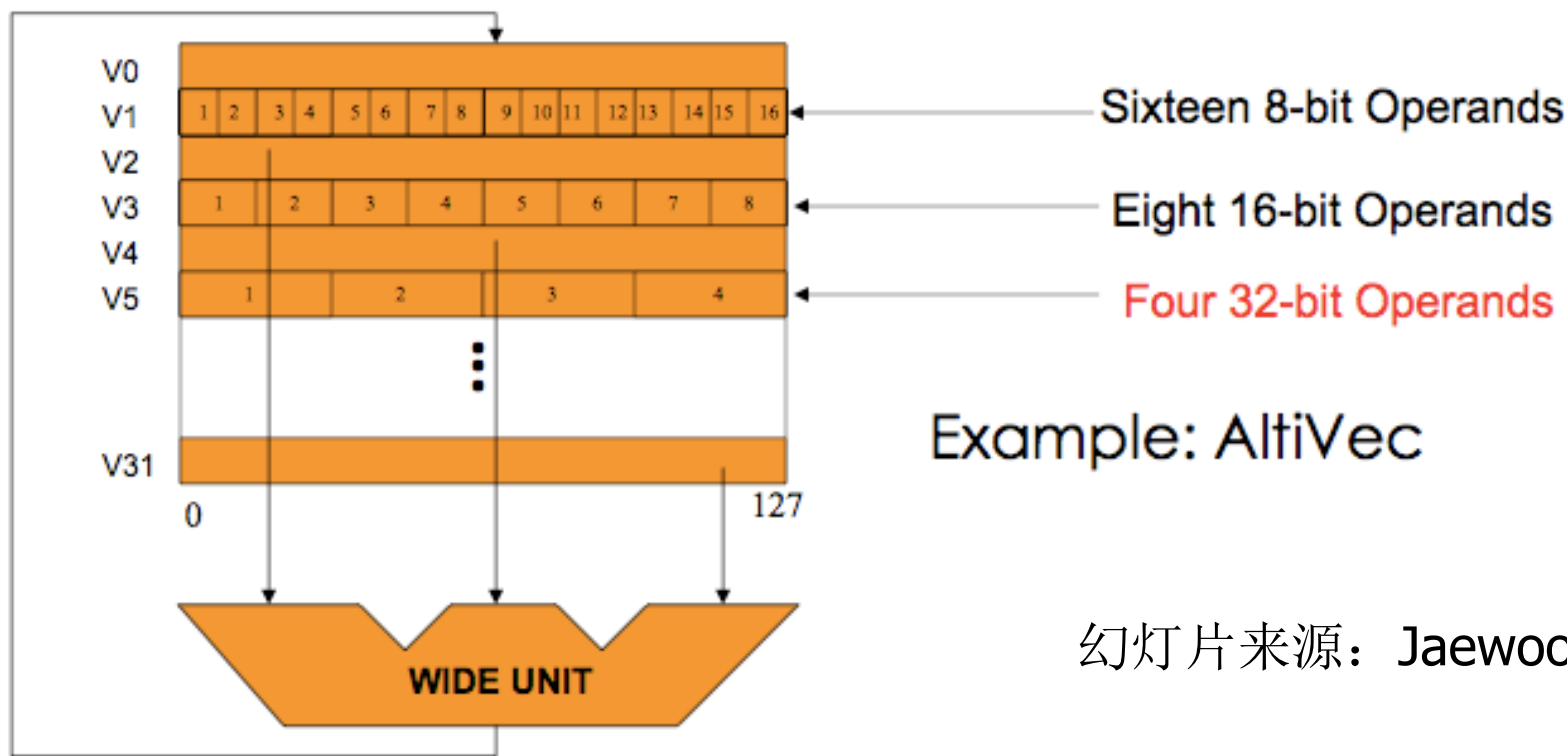
SIMD: `vadd<sws> v1,v2,v3`



幻灯片来源: Sam Larsen

SIMD架构

- 多媒体扩展架构的核心
 - SIMD并行
 - 可变大小的数据域
 - 向量长度=寄存器宽度/类型大小



幻灯片来源: Jaewook Shin



SIMD的应用

○ 图像处理

- 图形：3D游戏、电影
- 图像识别
- 视频编码/解码：JPEG、MPEG4

○ 音频

- 编码/解码：IP Phone、MP3
- 声音识别
- 数字信号处理：移动电话

○ 科学计算

- 基于数组的数据并行计算，另一级并行



适合应用的特点

- 规律的数据访问模式
 - 数据项在内存中连续存储
- 短数据类型：8、16、32位
- 流式数据处理，一系列处理阶段
 - 时间局部性，数据流重用
- 有些情况下
 - 很多常量
 - 循环迭代短
 - 算术运算饱和



为什么采用SIMD?

- +更大的并发度

- 当并发度足够时，是ILP的很好补充

- +设计简单：重复功能单元即可

- +更小的芯片尺寸

- 必须显式接触硬件

- 通过编译器，或程序员自己



多媒体扩展编程

○ 语言/指令集扩展

- 编程接口类似函数调用

- C/C++: 内置函数、intrinsics

- 大多数编译器支持多媒体扩展

- gcc: -march=corei7, -faltivec

- SSE2: `dst= _mm_add_ps(src1, src2);`

- Altivec: `dst= vec_add(src1, src2);`

- Neon: `dst = vaddq_f32(src1, src2)`

- 无统一标准

○ 很多编译器支持自动编译



提纲

- SIMD概念
- SIMD并行的问题
- SSE/AVX编程



SIMD并行

- 多个算术运算 → 一个SIMD操作
- 多个取数/存结果操作 → 一个更宽的内存操作



独立的算术运算

$$\begin{aligned}R &= R + XR * 1.08327 \\G &= G + XG * 1.89234 \\B &= B + XB * 1.29835\end{aligned}$$



$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} XR \\ XG \\ XB \end{bmatrix} * \begin{bmatrix} 1.08327 \\ 1.89234 \\ 1.29835 \end{bmatrix}$$

幻灯片来源: Sam Larsen



连续的内存访问

```
R = R + X[i+0]
G = G + X[i+1]
B = B + X[i+2]
```



R	=	R	+	X[i:i+2]
G		G		
B		B		

幻灯片来源: Sam Larsen



可向量化的循环

```
for (i=0; i<100; i+=1)  
    A[i+0] = A[i+0] + B[i+0]
```

幻灯片来源: Sam Larsen

可向量化的循环

```
for (i=0; i<100; i+=4)
    A[i+0] = A[i+0] + B[i+0]
    A[i+1] = A[i+1] + B[i+1]
    A[i+2] = A[i+2] + B[i+2]
    A[i+3] = A[i+3] + B[i+3]
```



```
for (i=0; i<100; i+=4)
    A[i:i+3] = A[i:i+3] + B[i:i+3]
```

幻灯片来源: Sam Larsen



可部分向量化的循环

```
for (i=0; i<16; i+=1)
    L = A[i+0] - B[i+0]
    D = D + abs(L)
```

可部分向量化的循环

```
for (i=0; i<16; i+=2)
    L = A[i+0] - B[i+0]
    D = D + abs(L)
    L = A[i+1] - B[i+1]
    D = D + abs(L)
```



```
for (i=0; i<16; i+=2)
    

|    |   |          |   |          |
|----|---|----------|---|----------|
| L0 | = | A[i:i+1] | - | B[i:i+1] |
| L1 |   |          |   |          |


    D = D + abs(L0)
    D = D + abs(L1)
```



SIMD编程的额外开销

- 理解多媒体指令的执行
- 理解额外开销
- 学习如何编写代码处理这种开销
- 有哪些开销？
 - ▣ 打包/解包数据的开销：重排数据使之连续
 - ▣ 对齐：调整数据访问，使之对齐
 - ▣ 控制流可能要求执行所有路径



打包/解包开销

$$\begin{aligned} C &= A + 2 \\ D &= B + 3 \end{aligned}$$



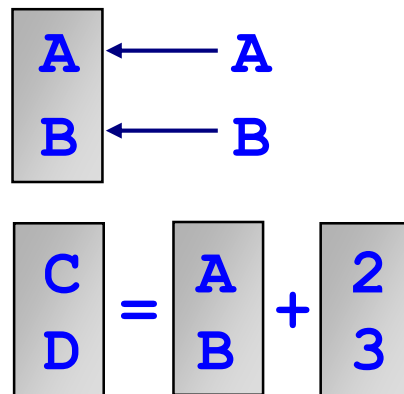
$$\begin{bmatrix} C \\ D \end{bmatrix} = \begin{bmatrix} A \\ B \end{bmatrix} + \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

幻灯片来源: Sam Larsen

打包/解包开销

- 打包源运算对象——拷贝到连续内存区域

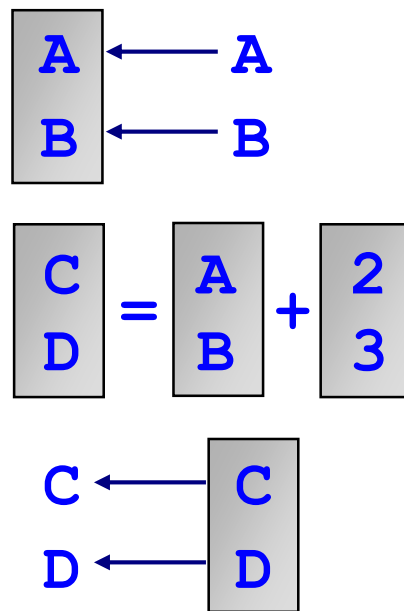
```
A = f()  
B = g()  
C = A + 2  
D = B + 3
```



打包/解包开销

- 打包源运算对象——拷贝到连续内存区域
- 解包目的运算对象——拷贝回内存

```
A = f()  
B = g()  
C = A + 2  
D = B + 3  
E = C / 5  
F = D * 7
```



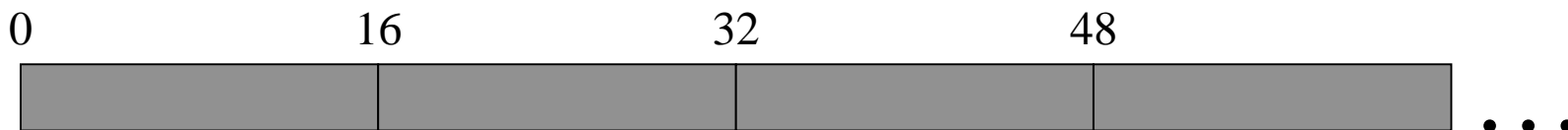
幻灯片来源: Sam Larsen

对齐开销

○ 对齐的内存访问

- 地址总是向量长度的倍数（例如16字节）

```
float a[64];  
for (i=0; i<64; i+=4)  
    Va = a[i:i+3];
```



对齐开销

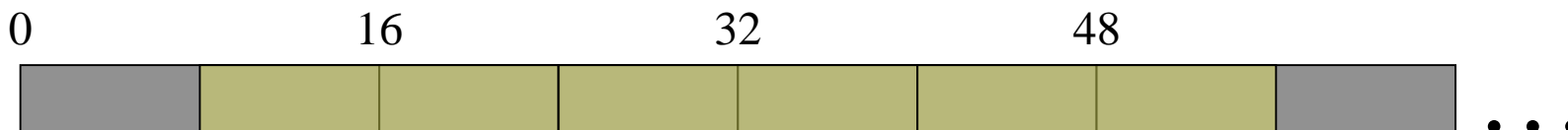
○ 未对齐的内存访问

- 地址不是16字节的整数倍
- 静态对齐：对未对齐的读操作，做两次相邻的对齐读操作，然后进行合并
- 有时硬件会帮你做，但仍然会产生多次内存操作

```
float a[64];  
for (i=0; i<60; i+=4)  
    Va = a[i+2:i+5];
```



```
float a[64];  
for (i=0; i<60; i+=4)  
    V1 = a[i:i+3];  
    V2 = a[i+4:i+7];  
    Va = merge(V1, V2, 8);
```





对齐开销

○ 静态调整循环

```
float a[64];
```

```
for (i=0; i<60; i+=4)
```

```
    Va = a[i+2:i+5];
```

```
float a[64];
```

```
Sa2 = a[2]; Sa3 = a[3];
```

```
for (i=4; i<64; i+=4)
```

```
    Va = a[i:i+3];
```

对齐开销

○ 未对齐的内存访问

- 距16字节边界的偏移是变化的或未知的
- 动态对齐：合并点在运行时计算

```
float a[64];  
start = read();  
for (i=start; i<60; i+=4)  
    Va = a[i:i+3];
```



```
float a[64];  
start = read();  
off = start % 4;  
for (i=start; i<60; i+=4)  
    V1 = a[i-off:i-off+3];  
    V2 = a[i-off+4:i-off+7];  
    Va = merge(V1, V2, off*4);
```



对齐问题小结

- 最坏情况需要计算地址，动态对齐
- 编译器（程序员）可分析确认对齐
 - 一般而言数据是从起始地址处对齐的
 - 如果在一个循环中顺序访问数据，起始位置固定，则对齐特性是不变的
- 可调整算法，先串行处理到对齐边界，然后进行SIMD计算
- 有时对齐开销会完全抵消SIMD的并行收益



控制流导致额外开销

- 如果程序中有控制流变化怎么办?
 - 所有路径都必须执行

```
for (i=0; i<16; i++)  
    if (a[i] != 0)  
        b[i]++;
```

What happens:

对所有元素计算 $a[i] \neq 0$
计算所有元素的 $b[i]++$ 存入临时寄存器t1
将原 $b[i]$ 拷贝到寄存器t2
根据 $a[i] \neq 0$ 的结果合并t1和t2

```
for (i=0; i<16; i++)  
    if (a[i] != 0)  
        b[i] = b[i] / a[i];  
    else  
        b[i]++;
```

What happens:

对所有元素计算 $a[i] \neq 0$
计算所有 $b[i] = b[i]/a[i]$ 存入t1
计算所有 $b[i]++$ 存入t2
根据 $a[i] \neq 0$ 的结果合并t1和t2

控制流例子

```
for (i=0; i<16; i++)  
    if (a[i] != 0)  
        b[i]++;
```



```
for (i=0; i<16; i+=4){  
    pred = a[i:i+3] != (0, 0, 0, 0);  
    old   = b[i:i+3];  
    new   = old + (1, 1, 1, 1);  
    b[i:i+3] = SELECT(old, new, pred);  
}
```

额外开销：永远是两个控制流路径都执行！

能否改进？

- 假定所有控制流路径执行频率都不同
- 应该针对频率最高的路径优化代码
- 其他路径按默认方式执行

```
for (i=0; i<16; i++)  
    if (a[i] != 0)  
        b[i]++;
```

若**a**多数不为0可以怎么做(特例):
对满足向量**a[i] != 0**的分支进行向量化,
否则按常规方式执行



```
for (i=0; i<16; i+=4) {  
    if (a[i:i+3] != (0, 0, 0, 0))  
        b[i:i+3] = b[i:i+3] + (1, 1, 1, 1);  
    else  
        //执行常规操作;  
}
```

若没有上述先验知识(**a**多数不为0),
一般认为这种控制流不适合向量化。



控制流开销小结

- 当前的商用编译器不太可能支持
 - 增加了编译器的复杂性
 - 可能减慢速度
 - 性能依赖于输入数据
- 一般观点，当存在控制流问题时，SIMD不是一个好的编程模型
- 但一些情况下还是可能加速的。



SIMD编程的复杂性

- 高层编程：利用编译器
 - ▣ 不总是有效
- 低层编程：使用intrinsic或汇编繁琐易错
- 数据必须对齐，在内存中连续存储
 - ▣ 未对齐的数据可能产生不正确的结果
 - ▣ 可能需要拷贝到连续区域（额外开销）
- 控制流问题引入了更多复杂性，还可能导致低效



提纲

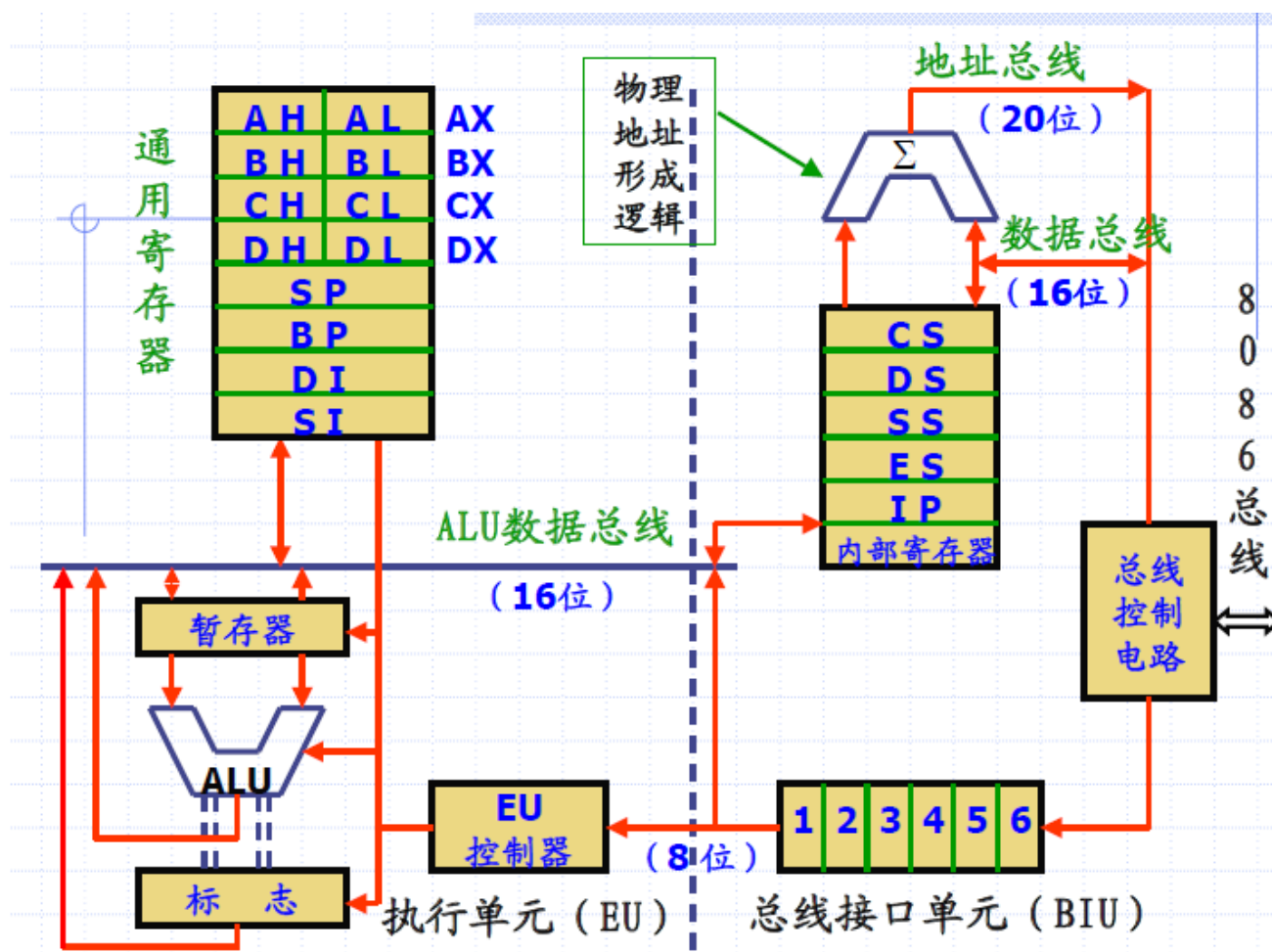
- SIMD概念
- SIMD并行的问题
- SSE/AVX编程



X86架构发展历史

- X86: Intel开发的一种微处理器体系结构
- 出现: 1978年Intel 8086 CPU中
- 发展:
 - 1971-1992年数字编号: 80X86系列
 - 1993-2005年奔腾系列: Pentium
 - 2005酷睿系列: Core

X86架构基本框架





X86架构基本框架

- 基本的执行模式
- 数据类型
- 指令集合
- 寄存器



X86架构基本框架

○ 基本的执行模式

- 真实模式（寻址1M，段地址+偏移地址）
- 保护模式（扩充寻址范围）
- 系统管理模式（系统中断进入，可执行所有I/O和合适的系统指令）



X86架构基本框架

○ 数据类型

- 基本数据类型

- 数字数据类型

 - 整型

 - 浮点

- 指针类型

- 位字段数据类型

- 字符串数据类型

- 打包的SIMD数据类型等



X86架构基本框架

○ 指令集合

- 通用指令（传送，算术，逻辑，控制等）
- X87 FPU指令（传送，算术，比较，控制等）
- MMX指令（传送，转化，打包，比较等）
- SSE指令（增加寄存器，SIMD浮点数运算）
- SSE2指令（整数指令，64-bit SIMD浮点运算）



X86架构基本框架

- 指令集合

- SSE3指令

- SSE4指令

- 后续

- SSE5指令

- AVX指令

- FMA指令



X86架构基本框架

○ 寄存器

- 4个数据寄存器(EAX、EBX、ECX和EDX)
- 2个变址和指针寄存器(ESI和EDI) 2个指针寄存器(ESP和EBP)
- 6个段寄存器(ES、CS、SS、DS、FS和GS)
- 1个指令指针寄存器(EIP) 1个标志寄存器(EFlags)



x86架构SIMD支持

- 当前AMD和Intel的x86的ISA和微架构都支持SIMD运算
- ISA支持
 - MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX, AVX2
 - 查询电脑配置：windows可以用CPU-Z等工具；Linux系统，查看/proc/cpuinfo中的标志(flags)位
 - SSE (Streaming SIMD extensions)：x86架构的SIMD扩展指令集
- 微架构支持
 - 多功能单元
 - 8个128位的向量寄存器（SSE）
16个256位的向量寄存器（AVX）



SSE指令集是什么

- SSE: Streaming SIMD Extension
- SIMD(Single Instruction Multiple Data): 单指令流多数据流
- 用一个控制器对一组数据（又称“数据向量”）中的每一个分别执行相同的操作来实现空间上的并行性
- MMX
- SSE



SSE指令集的发展历史

- SSE是MMX的超集
- MMX: Matrix Math eXtension / MultiMedia eXtension
 - 1996年Intel在奔腾处理器集成MMX指令，为应对音频、图片、视频等多媒体应用的密集的计算需求
 - 64-bit的MMX寄存器（8个，复用了浮点寄存器的尾部，与x87共用寄存器，缺少浮点指令）
 - 支持在打包的字，字节，双字整数上的SIMD操作



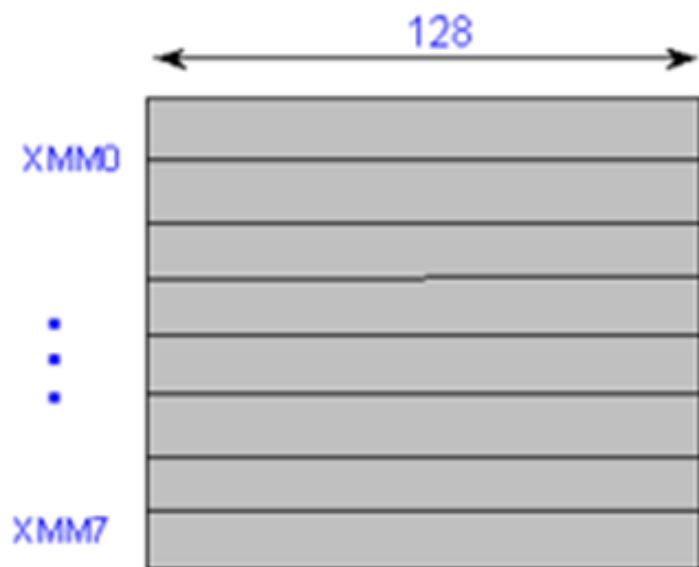
SSE指令集

- SSE指令集出现在Pentium III处理器中,1999年
 - 包括了70条指令，其中50条SIMD浮点运算指令、12条MMX 整数运算增强指令、8条内存连续数据块传输指令
 - 新增8个XMM寄存器（XMM0-XMM7）
 - 在X86_64中额外增加8个（XMM8-XMM15）
 - 一个时钟周期内同时进行多个相同性质的运算
 - 新版本加强整数和双精度浮点数的运算

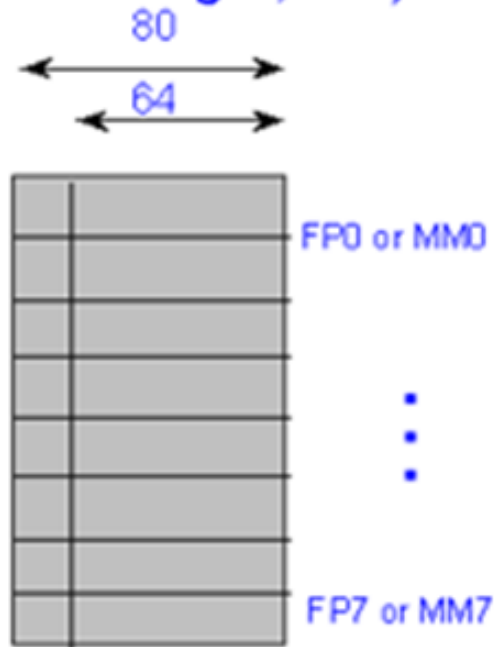
SSE指令集

○ SSE128bit寄存器与MMX寄存器的区别

Internet SSE
(Scalar/packed SIMD-SP)



MMX/x87
(64-bit Integer, x87)

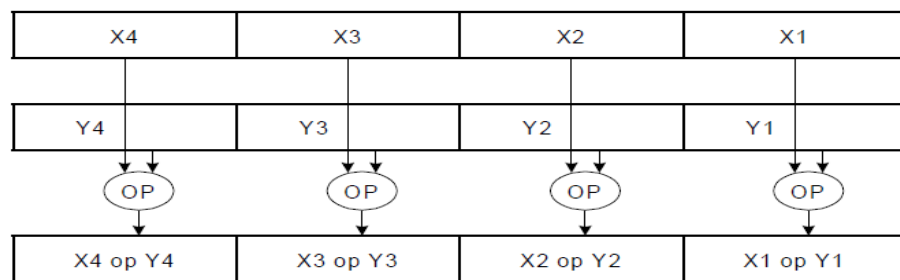


SSE指令集

○ SIMD优化

□ 内存设置

□ 数据操作



OM15148

Figure 2-15. Typical SIMD Operations

64-bit MMX Registers

MM7
MM6
MM5
MM4
MM3
MM2
MM1
MM0

128-bit XMM Registers

XMM7
XMM6
XMM5
XMM4
XMM3
XMM2
XMM1
XMM0

OM15149

Figure 2-16. SIMD Instruction Register Usage



SSE2

- SSE2指令集，2000年

- 增加到144条指令

- 加强64位双精度浮点数及字节、16位短整型、32位整型、64位长整型的处理能力

- cache预取指令

- SSE和SSE2使用8个128bits寄存器

- XMM0~XMM7

- 16个8位整数，8个16位整数，4个32位整数

- 4个单精度浮点数，2个双精度浮点数



SSE3、SSE4

○ SSE3指令集，2004年

- 增加13条指令（允许寄存器内部之间运算，浮点数到整数的转换）
- 超线程性能增强指令可以提升处理器的超线程处理能力
- 快速浮点→整数转换

○ SSE4指令集，2007年

- 新增的47条新多媒体指令集
- STTNI(String and Text New Instructions)——XML
- CRC校验



AVX

- AVX（Advanced Vector eXtensions）指令集，2011年
 - 16个256bits的寄存器YMM0~YMM15（低128bits XMM0~XMM15）
- AVX2，2013年
 - 扩展了大部分SSE到256位
 - 三对象运算：融合乘加运算
 - gather：非连续内存取数
 - 32位、64位任意重排，向量移位

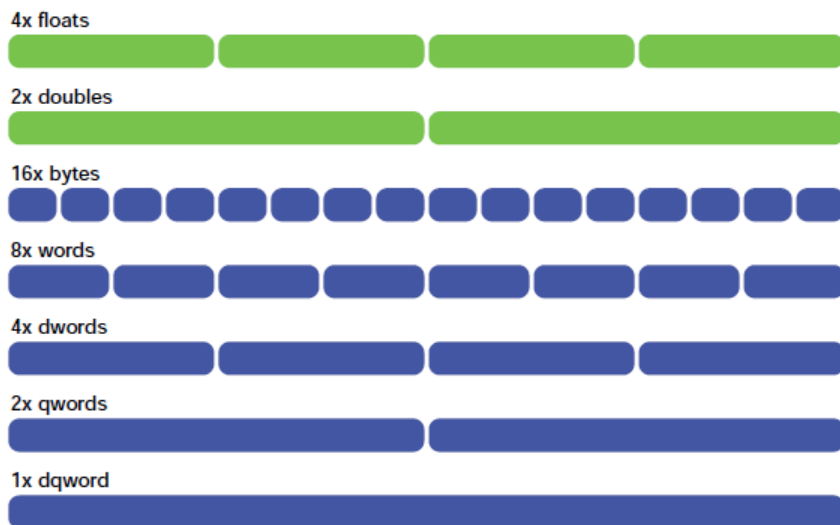
指令集的发展

指令集	条	Date	ICPU	IDate	ACPU	ADate	Memo
MMX	57	1996-10-12	Pentium MMX(P55C)	1996-10-12	K6	1997-4-1	MultiMedia eXtension
SSE	70	1999-5-1	Pentium III(Katmai)	1999-5-1	Athlon XP	2001-10-9	Streaming SIMD Extensions
SSE2	144	2000-11-1	Pentium 4(Willamette)	2000-11-1	Opteron	2003-4-22	
SSE3	13	2004-2-1	Pentium 4(Prescott)	2004-2-1	Athlon 64	2005-4-1	
SSSE3	16	2006-1-1	Core	2006-1-1	Fusion(Bobcat)	2011-1-5	最早出现在Tejas核心(功耗过高而取消)
SSE4.1	47	2006-9-27	Penryn	2007-11-1	Bulldozer	2011-9-7	
SSE4.2	7	2008-11-17	Nehalem	2008-11-17	Bulldozer	2011-9-7	
SSE4a	4	2007-11-11			K10	2007-11-11	K10还加了 POPCNT 与 LZCNT 指令
SSE5		2007-8-30					被AVX搅局。后来XOP/FAM4/CVT16
AVX		2008-3-1	Sandy Bridge	2011-1-9	Bulldozer	2011-9-7	Advanced Vector Extensions
AVX2		2011-6-13	Haswell	2013-4-1			
AES	7	2008-3-1	Westmere	2010-1-7	Bulldozer	2011-9-7	Advanced Encryption Standard
3DNow!Prefetch	2	2010-8-1			K6-2	1998-5-28	2010年8月放弃3DNow!, 仅保留2条预取
3DNow!	21	1998-1-1			K6-2	1998-5-28	
3DNow!+		1999-6-23			Athlon	1999-6-23	Enhanced 3DNow!, 共52条?
MmxExt					Athlon	1999-6-23	Extensions MMX
3DNow! Pro					Athlon XP	2001-10-9	3DNow! Professional.兼容SSE
POPCNT	1	2007-11-11			K10	2007-11-11	
ABM	1	2007-11-11			K10	2007-11-11	advanced bit manipulation. LZCNT
CLMUL	5	2008-5-1	Westmere	2010-1-7	Bulldozer	2011-9-7	PCLMULQDQ等
F16C		2009-5-1	Ivy Bridge	2012-4-1	Bulldozer	2011-9-7	CVT16
FAM4		2009-5-1			Bulldozer	2011-9-7	
XOP		2009-5-1			Bulldozer	2011-9-7	

SSE编程

○ 向量寄存器支持多种数据类型

- 整数（16字节、8 short、4 int、2 long long、1 dqword）
- 单精度浮点数（4 floats）
- 双精度浮点数（2 doubles）



Anything that fits into 16 bytes

[Klimovitski 2001]

Figure 1. SSE/SSE2 data types



SSE指令

- 数据移动指令：将数据移入/出向量寄存器
- 算术指令：多个数据（2 doubles、4 floats等）上的算术运算
- 逻辑指令：多个数据上的逻辑运算
- 比较指令：多个数据上的比较运算
- 洗牌指令：在SIMD寄存器内移动数据
- 其他
 - 类型转换：x86和SIMD寄存器之间
 - 缓存控制：向量可能污染cache
 - 状态管理



数据移动指令(汇编)

- MOV**U**PS – 从内存或SIMD寄存器移动128位数据到一个SIMD寄存器。未对齐的
- MOV**A**PS – 从内存或SIMD寄存器移动128位数据到一个SIMD寄存器。对齐的
- MOV**H**PS – 将64位数据移动到SIMD寄存器高位
- MOV**L**PS – 将64位数据移动到SIMD寄存器低位
- MOV**HL**PS – 将源寄存器高64位移动到目标寄存器低64位
- MOV**LH**PS – 将源寄存器低64位移动到目标寄存器高64位
- MOV**MSK**PS – 将4个打包的标量数据的符号位移动到一个x86整数寄存器
- MOV**SS** – 从内存或SIMD寄存器将32位数据移动到一个SIMD寄存器



SSE指令

○ 算术指令

- PD: 两个双精度, PS: 四个单精度, SS: 标量
- ADD、SUB、MUL、DIV、SQRT、MAX、MIN、RCP等
 - ADDPS: 四个单精度加法; ADDSS: 标量加法

○ 逻辑指令

- AND、OR、XOR、ANDN等
 - ANDPS – 运算对象位与
 - ANDNPS – 运算对象位与非

○ 比较指令

- CMPPS、CMPSS: 比较运算对象, 每个比较结果影响SIMD寄存器中32位——全1或全0



SSE指令

○ 洗牌指令

- SHUFPS: 从一个运算对象洗牌数据保存到另一个运算对象
- UNPCKHPS: 解包高位数据到一个SIMD寄存器
 - $\text{UNPCKHPS}[x4, x3, x2, x1][y4, y3, y2, y1] = [y4, x4, y3, x3]$
- UNPCKLPS

○ 其他指令

- 类型转换: CVTTPS2PI mm, xmm/mem64
- 缓存控制
 - MOVNTPS将浮点数据从一个SIMD寄存器保存到内存, 绕过缓存
- 状态管理: LDMXCSR读取MXCSR状态寄存器



SSE C/C++编程

○ SSE指令对应C/C++ **intrinsic**

- **intrinsic**: 编译器能识别的函数，直接映射为一个或多个汇编语言指令。Intrinsic函数本质上比调用函数更高效
- **Intrinsics**为处理器专有扩展特性提供了一个C/C++编程接口
- 主流编译器都支持，如GCC



SSE intrinsics

- 使用SSE intrinsics所需的头文件（向前兼容）
 - `#include <xmmintrin.h> //SSE`
 - `#include <emmintrin.h> //SSE2`
 - `#include <pmmmintrin.h> //SSE3`
 - `#include <tmmintrin.h> //SSSE3`
 - `#include <smmintrin.h> //SSE4.1`
 - `#include <nmmintrin.h> //SSSE4.2`
 - `#include <immintrin.h> //AVX、AVX2`
- AMD CPU对MMX/SSE/SSE2支持较好，SSE4支持较差
- 编译选项： `-march=corei7`



SSE intrinsics

- 数据类型（映射到XMM寄存器）
 - `__m128`: float
 - `__m128d`: double
 - `__m128i`: integer
- 数据移动和初始化
 - `_mm_load_ps`, `_mm_loadu_ps`, `_mm_load_pd`, `_mm_loadu_pd`, etc
 - `_mm_store_ps`, ...
 - `_mm_setzero_ps`



SSE intrinsics

- 数据类型（映射到XMM寄存器）
 - `__m128`: float
 - `__m128d`: double
 - `__m128i`: integer
- 数据移动和初始化
 - `_mm_load_ps`, `_mm_loadu_ps`, `_mm_load_pd`, `_mm_loadu_pd`, etc
 - `_mm_store_ps`, ...
 - `_mm_setzero_ps`
 - `_mm_loadl_pd`, `_mm_loadh_pd`
 - `_mm_storel_pd`, `_mm_storeh_pd`



SSE intrinsics

- 数据类型（映射到XMM寄存器）

- `__m128`: float
- `__m128d`: double
- `__m128i`: integer

- 数据移动和初始化

- `_mm_loadps`, `_mm_loadups`, `_mm_loadpd`, `_mm_loadupd`, etc
- `_mm_storeps`, ...
- `_mm_setzerops`
- `_mm_loadlpd`, `_mm_loadhpd`
- `_mm_storelpd`, `_mm_storehpd`
- `_mm_setps`, `_mm_setlss`, `_mm_setzeross`



SSE intrinsics

○ 算术intrinsics:

□ `_mm_add_ss, _mm_add_ps, ...`

□ `_mm_add_pd, _mm_mul_pd`

□ `_mm_hadd_ps:`

$[x4, x3, x2, x1][y4, y3, y2, y1] =$
 $[y3+y4, y1+y2, x3+x4, x1+x2]$

□ 逻辑: `_mm_and_ps` (andnot, or, xor)

□ 比较: `_mm_cmpeq_ps` – 结果寄存器全0/全1
`_mm_comieq_ss` – 影响EFLAGS, 返回布尔值



SSE intrinsics

○ 预取intrinsics:

- `_mm_prefetch(char const *a, int sel)`
- 预取从a开始的cache line大小的内存
- sel: PREFETCHNTA – 预取到L2, 尽量避免污染
PREFETCHNT0(1-2): P4以上预取到L2

○ 更多细节参考

- <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>
- 《Intel 64 and IA-32 Architectures Software Developer's Manual》

Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX-512
- ☐ KNC
- ☐ SVML
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare
- ☐ Convert
- ☐ Cryptography
- ☐ Elementary Math

Functions

- ☐ General Support
- ☐ Load
- ☐ Logical
- ☐ Mask
- ☐ Miscellaneous
- ☐ Move
- ☐ OS-Targeted
- ☐ Probability/Statistics
- ☐ Random
- ☐ Set
- ☐ Shift
- ☐ Special Math Functions
- ☐ Store
- ☐ String Compare
- ☐ Swizzle
- ☐ Trigonometry

<code>__m128i _mm_hadd_epi16 (__m128i a, __m128i b)</code>	phaddw
<code>__m256i _mm256_hadd_epi16 (__m256i a, __m256i b)</code>	vphaddw
<code>__m128i _mm_hadd_epi32 (__m128i a, __m128i b)</code>	phadd
<code>__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)</code>	vphadd
<code>__m128d _mm_hadd_pd (__m128d a, __m128d b)</code>	haddpd
<code>__m256d _mm256_hadd_pd (__m256d a, __m256d b)</code>	vhaddpd
<code>__m64 _mm_hadd_pi16 (__m64 a, __m64 b)</code>	phaddw
<code>__m64 _mm_hadd_pi32 (__m64 a, __m64 b)</code>	phaddw
<code>__m128 _mm_hadd_ps (__m128 a, __m128 b)</code>	haddps

Synopsis

```
__m128 _mm_hadd_ps (__m128 a, __m128 b)
#include <pmmmintrin.h>
Instruction: haddps xmm, xmm
CPUTID Flags: SSE3
```

Description

Horizontally add adjacent pairs of single-precision (32-bit) floating-point elements in `a` and `b`, and pack the results in `dst`.

Operation

```
dst[31:0] := a[63:32] + a[31:0]
dst[63:32] := a[127:96] + a[95:64]
dst[95:64] := b[63:32] + b[31:0]
dst[127:96] := b[127:96] + b[95:64]
```

Performance

Architecture	Latency	Throughput (CPI)
Skylake	6	2
Broadwell	5	2
Haswell	5	2
Ivy Bridge	5	2

<code>__m256 _mm256_hadd_ps (__m256 a, __m256 b)</code>	vhaddps
<code>__m128i _mm_hadds_epi16 (__m128i a, __m128i b)</code>	phaddsw
<code>__m256i _mm256_hadds_epi16 (__m256i a, __m256i b)</code>	vphaddsw
<code>__m64 _mm_hadds_pi16 (__m64 a, __m64 b)</code>	phaddsw



Intrinsic函数的命名

- Intrinsic函数的命名有一定的规律，一个Intrinsic通常由3部分构成：
 - 第一部分为前缀_mm，表示是SSE指令集对应的Intrinsic函数。
_mm256或_mm512是AVX,AVX-512指令集的Intrinsic函数前缀，这里只讨论SSE故略去不作说明。
 - 第二部分为对应的指令的操作，如_add，_mul，_load等，有些操作可能会有修饰符，如loadu将16位未对齐的操作数加载到寄存器中。
 - 第三部分为操作的对象名及数据类型，_ps packed操作所有的单精度浮点数；_pd packed操作所有的双精度浮点数；_pixx（xx为长度，可以是8，16，32，64）packed操作所有的xx位有符号整数，使用的寄存器长度为64位；_epixx（xx为长度）packed操作所有的xx位的有符号整数，使用的寄存器长度为128位；_epuxx packed操作所有的xx位的无符号整数；_ss操作第一个单精度浮点数。
- 将这三部分组合就是一个Intrinsic函数，如_mm_mul_epi32对参数中所有的32位有符号整数进行乘法运算。



SSE intrinsics

- 数据对齐问题

- 某些intrinsics可能要求数据对齐到16字节内存边界

- 若不对齐可能无法执行

- 编写更通用的SSE程序

- 检查内存边界

- 某些情况下，SSE可能无法带来性能收益



例：矩阵乘法

```
#include <stdio.h>
#include <pmmintrin.h>
#include <stdlib.h>
#include <algorithm>
#include <windows.h>
```

```
using namespace std;
```

```
const int maxN = 1024;      // magnitude of matrix
const int T = 64;           // tile size
```

```
int n;
float a[maxN][maxN];
float b[maxN][maxN];
float c[maxN][maxN];
```

```
long long head, tail, freq;    //timers
```



例：矩阵乘法——串行算法

```
void mul(int n, float a[][maxN], float b[][maxN], float c[][maxN]) {  
    for (int i = 0; i < n; ++i) {  
        for (int j = 0; j < n; ++j) {  
            c[i][j] = 0.0;  
            for (int k = 0; k < n; ++k) {  
                c[i][j] += a[i][k] * b[k][j];  
            }  
        }  
    }  
}
```

12.4s

在向量化之前，能否有其他优化？



例：矩阵乘法——cache优化

```
void trans_mul(int n, float a[][maxN], float b[][maxN], float
c[][maxN]){
    // transposition
    for (int i = 0; i < n; ++i) for (int j = 0; j < i; ++j) swap(b[i][j], b[j][i]);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            c[i][j] = 0.0;
            for (int k = 0; k < n; ++k) {
                c[i][j] += a[i][k] * b[j][k];
            }
        }
    }
    // transposition
    for (int i = 0; i < n; ++i) for (int j = 0; j < i; ++j) swap(b[i][j], b[j][i]);
}
```

1.12s !

例：矩阵乘法——SSE版本

```
void sse_mul(int n, float a[][maxN], float b[][maxN], float c[][maxN]){
    __m128 t1, t2, sum;
    for (int i = 0; i < n; ++i) for (int j = 0; j < i; ++j) swap(b[i][j], b[j][i]);
    for (int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j){
            c[i][j] = 0.0;
            sum = _mm_setzero_ps();
            for (int k = n - 4; k >= 0; k -= 4){ // sum every 4 elements
                t1 = _mm_loadu_ps(a[i] + k);
                t2 = _mm_loadu_ps(b[j] + k);
                t1 = _mm_mul_ps(t1, t2);
                sum = _mm_add_ps(sum, t1);
            }
            sum = _mm_hadd_ps(sum, sum);
            sum = _mm_hadd_ps(sum, sum);
            _mm_store_ss(c[i] + j, sum);
        }
    }
}
```

此时得到了什么？

_mm_hadd_ps:

操作数 a(A3, A2, A1, A0)

操作数 b(B3, B2, B1, B0)

结果是 (B3 + B2, B1 + B0,
A3 + A2, A1 + A0)。



例：矩阵乘法——SSE版本

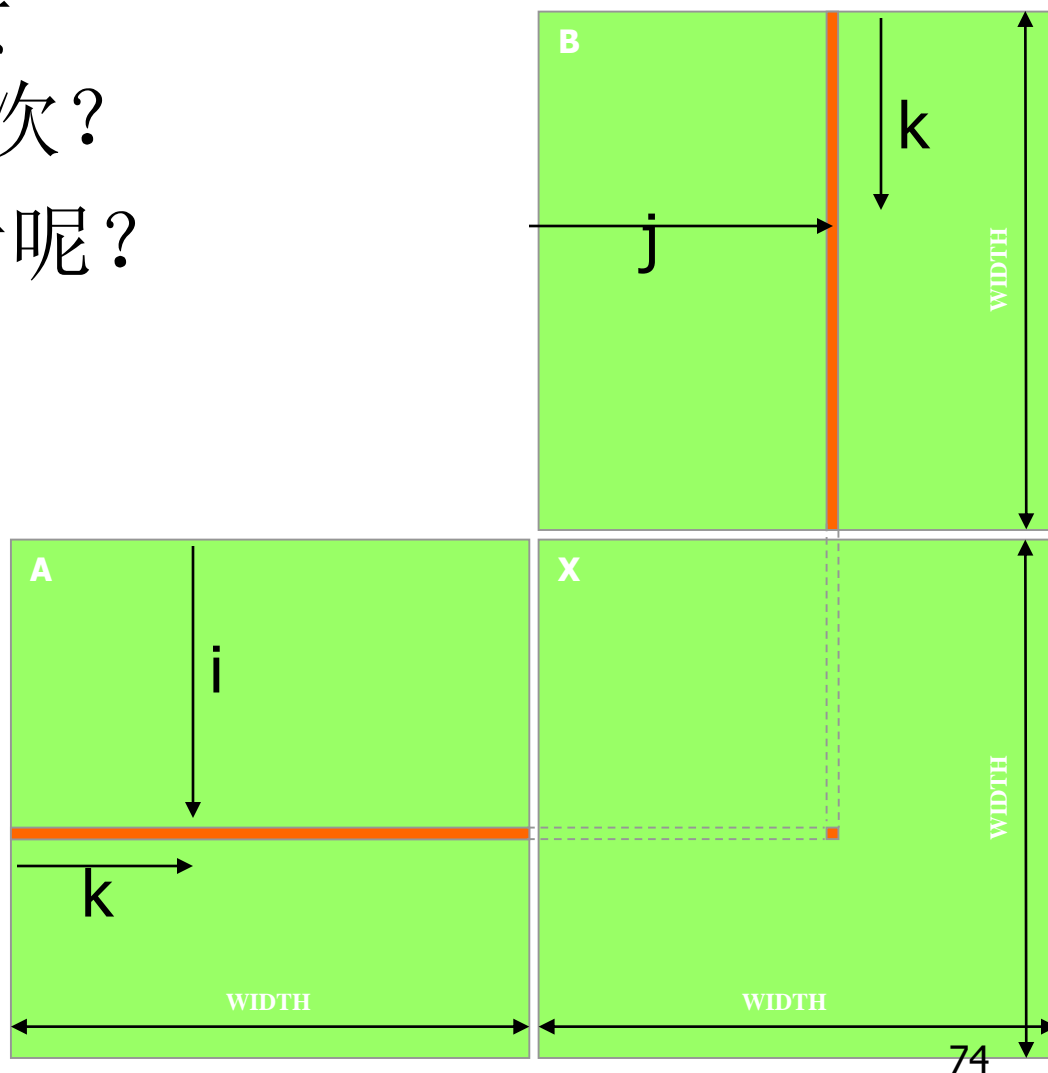
```
        for (int k = (n % 4) - 1; k >= 0; --k){    // handle the last n%4
elements
            c[i][j] += a[i][k] * b[j][k];
        }
    }
}
for (int i = 0; i < n; ++i) for (int j = 0; j < i; ++j) swap(b[i][j], b[j][i]);
}
```

0.42s

还能优化吗？

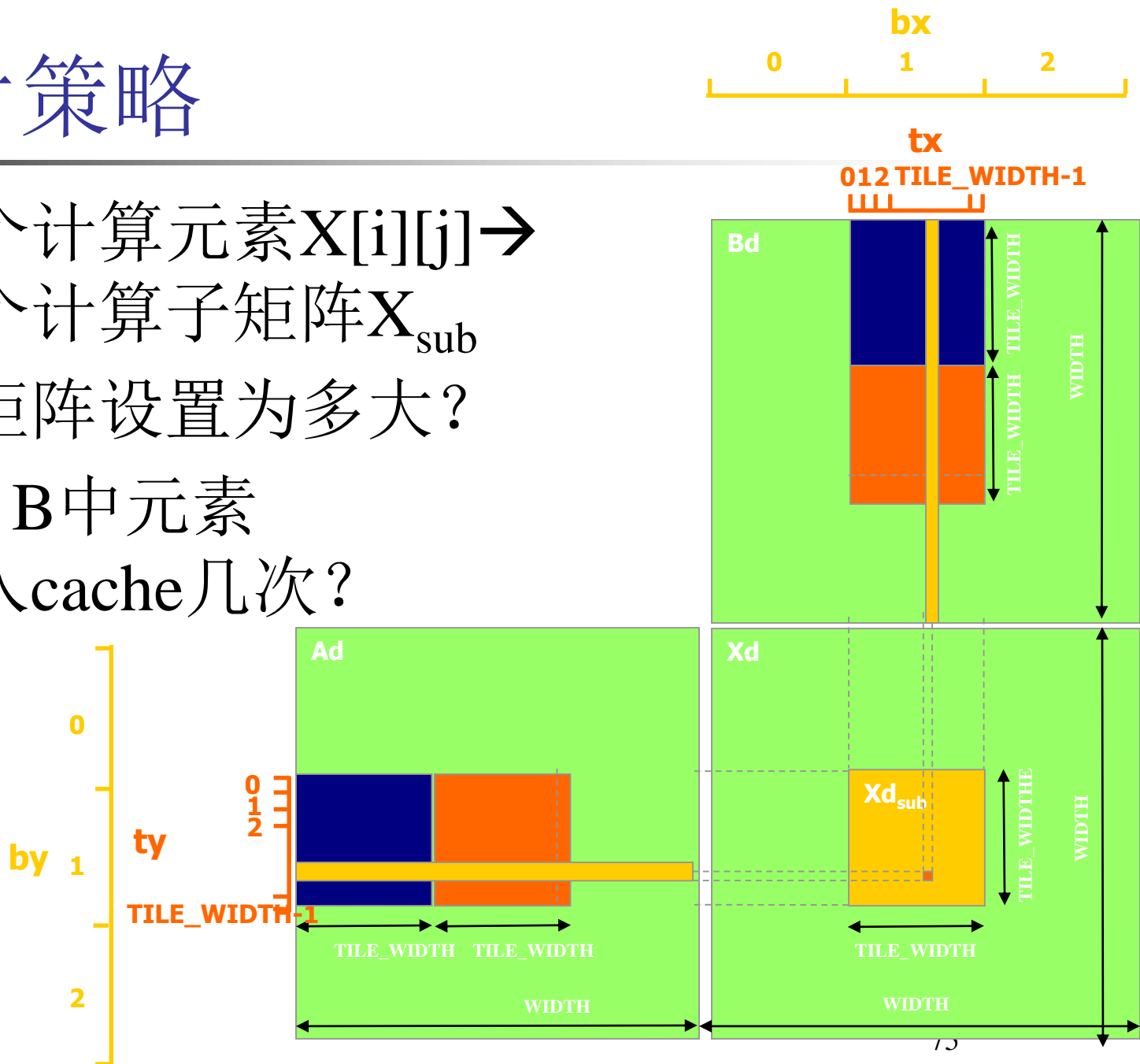
SSE版本cache效率

- A的每个元素进入cache几次?
- B的每个元素呢?



分片策略

- 逐个计算元素 $X[i][j] \rightarrow$
逐个计算子矩阵 X_{sub}
- 子矩阵设置为多大?
- A、B中元素
进入cache几次?





例：矩阵乘法——分片策略

```
void sse_tile(int n, float a[][maxN], float b[][maxN], float c[][maxN])
{
    __m128 t1, t2, sum;
    float t;

    for (int i = 0; i < n; ++i) for (int j = 0; j < i; ++j) swap(b[i][j], b[j][i]);
    for (int r = 0; r < n / T; ++r) for (int q = 0; q < n / T; ++q) {
        for (int i = 0; i < T; ++i) for (int j = 0; j < T; ++j) c[r * T + i][q * T +
j] = 0.0;

        for (int p = 0; p < n / T; ++p) {
            for (int i = 0; i < T; ++i) for (int j = 0; j < T; ++j) {
                sum = _mm_setzero_ps();
```

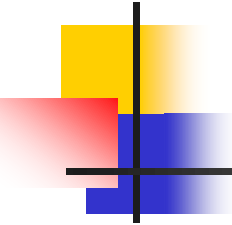


例：矩阵乘法——分片策略

```
for (int k = 0; k < T; k += 4){    //sum every 4th elements
    t1 = _mm_loadu_ps(a[r * T + i] + p * T + k);
    t2 = _mm_loadu_ps(b[q * T + j] + p * T + k);
    t1 = _mm_mul_ps(t1, t2);
    sum = _mm_add_ps(sum, t1);
}
sum = _mm_hadd_ps(sum, sum);
sum = _mm_hadd_ps(sum, sum);
_mm_store_ss(&t, sum);
c[r * T + i][q * T + j] += t;
}
}
}
```

0.3s

```
for (int i = 0; i < n; ++i) for (int j = 0; j < i; ++j) swap(b[i][j], b[j][i]);
}
```



例：矩阵乘法——AVX版本

```
void avx_mul(int n, float a[][maxN], float b[][maxN], float c[][maxN])
{
    __m256 t1, t2, sum;
    __m128 s1, s2;

    for (int i = 0; i < n; ++i) for (int j = 0; j < i; ++j) swap(b[i][j], b[j][i]);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            c[i][j] = 0.0;
            sum = _mm256_setzero_ps();
            for (int k = n - 8; k >= 0; k -= 8) { //sum every 8 elements
                t1 = _mm256_loadu_ps(a[i] + k);
                t2 = _mm256_loadu_ps(b[j] + k);
                t1 = _mm256_mul_ps(t1, t2);
                sum = _mm256_add_ps(sum, t1);
            }
        }
    }
}
```

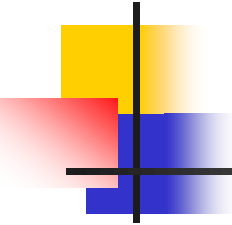
编译选项：-march=native

例：矩阵乘法——AVX版本

```
s1 = _mm256_extractf128_ps(sum, 0); // s1=[a0,a1,a2,a3]
s2 = _mm256_extractf128_ps(sum, 1); // s2=[a4,a5,a6,a7]
s1 = _mm_hadd_ps(s1, s2); // s1=[a0+a1,a2+a3,a4+a5,a6+a7]
s1 = _mm_hadd_ps(s1, s1); //
s1=[a0+a1+a2+a3,a4+a5+a6+a7,a0+a1+a2+a3,a4+a5+a6+a7]
s1 = _mm_hadd_ps(s1, s1); //
s1=[a0+a1+a2+a3+a4+a5+a6+a7,...]
_mm_store_ss(c[i] + j, s1);
for (int k = (n % 8) - 1; k >= 0; --k) { // handle the last n%8
elements
    c[i][j] += a[i][k] * b[j][k];
}
}
}
for (int i = 0; i < n; ++i) for (int j = 0; j < i; ++j) swap(b[i][j], b[j][i]);
}
```

0.32s

AVX分片：0.27s



例：矩阵乘法——Neon版本

```
#include <arm_neon.h>
void neon_mul(int n, float a[][maxN], float b[][maxN], float c[][maxN])
{
    float32x4_t t1, t2, sum;
    float32x2_t s1, s2;
    float tmp;
    for (int i = 0; i < n; ++i) for (int j = 0; j < i; ++j) {
        tmp = b[i][j]; b[i][j] = b[j][i]; b[j][i] = tmp; }
    for (int i = 0; i < n; ++i) {    for (int j = 0; j < n; ++j) {
        c[i][j] = 0.0;
        sum = vdupq_n_f32(0.0);
        for (int k = n - 4; k >= 0; k -= 4) {    //sum every 4th elements
            t1 = vld1q_f32(a[i] + k);
            t2 = vld1q_f32(b[j] + k);
            t1 = vmulq_f32(t1, t2);
            sum = vaddq_f32(sum, t1);
        }
    }
}
```




例：矩阵乘法——Neon版本(2)

```
s1 = vget_low_f32(sum);
s2 = vget_high_f32(sum);
s1 = vpadd_f32(s1, s2);
s1 = vpadd_f32(s1, s1);
vst1_lane_f32(c[i] + j, s1, 0);
for (int k = (n % 4) - 1; k >= 0; --k) {    //handle the last n%4
elements
    c[i][j] += a[i][k] * b[j][k];
}
}
}
for (int i = 0; i < n; ++i) for (int j = 0; j < i; ++j) {
    tmp = b[i][j]; b[i][j] = b[j][i]; b[j][i] = tmp;
}
}
```



例：矩阵乘法——Neon版本(3)

Seq: 18687.131000ms

Trans: 3046.320000ms

Neon: 2071.477000ms



其他SSE指令简介—shuffle

`__m128 __mm_shuffle_ps (__m128 a, __m128 b, unsigned int imm8)`

```
SELECT4(src, control){  
    CASE(control[1:0])  
    0:      tmp[31:0] := src[31:0]  
    1:      tmp[31:0] := src[63:32]  
    2:      tmp[31:0] := src[95:64]  
    3:      tmp[31:0] := src[127:96]  
    ESAC  
    RETURN tmp[31:0]  
}
```

```
dst[31:0] := SELECT4(a[127:0], imm8[1:0])  
dst[63:32] := SELECT4(a[127:0], imm8[3:2])  
dst[95:64] := SELECT4(b[127:0], imm8[5:4])  
dst[127:96] := SELECT4(b[127:0], imm8[7:6])
```



其他SSE指令简介—blend

`__m128 __mm_blend_ps (__m128 a, __m128 b, const int imm8)`

```
FOR j := 0 to 3
    i := j*32
    IF imm8[j%8]
        dst[i+31:i] := b[i+31:i]
    ELSE
        dst[i+31:i] := a[i+31:i]
    FI
ENDFOR
```



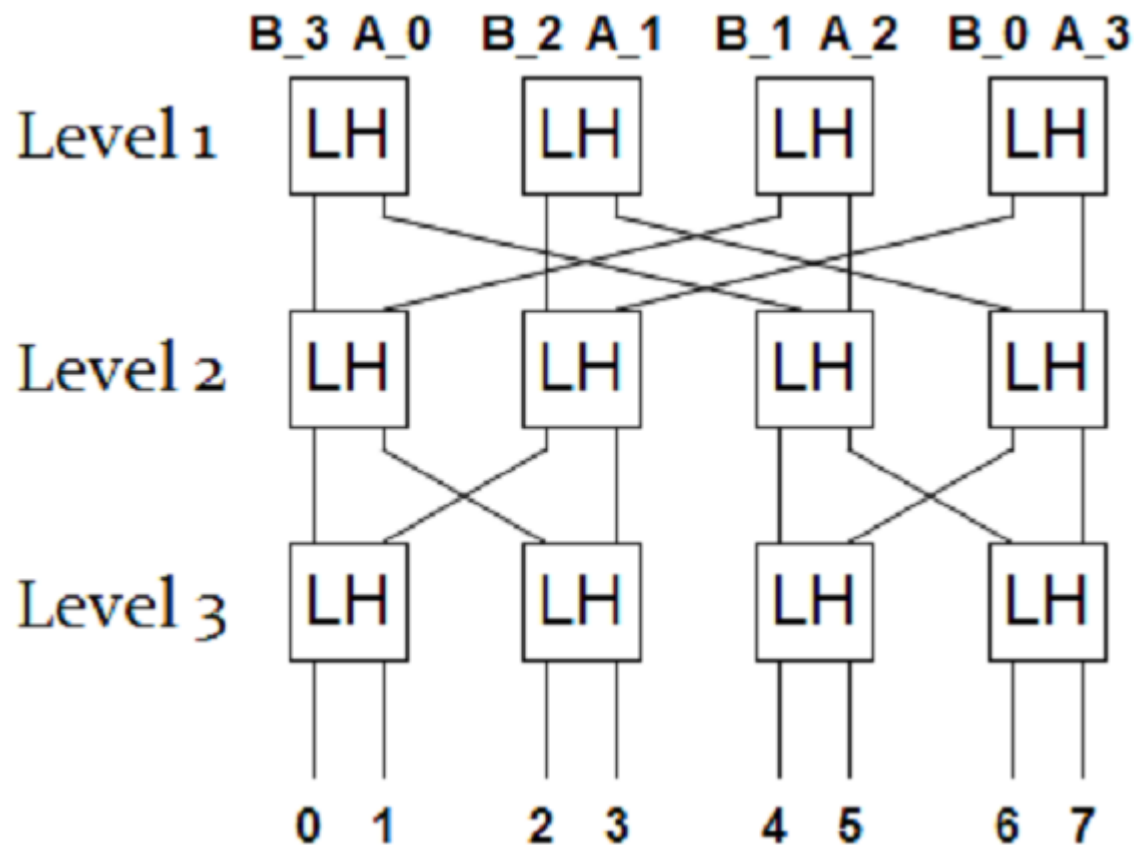
作用？

- 输入两向量A1 A2 A3 A4和B1 B2 B3 B4
- 希望比较A1 A2、 A3 A4、 B1 B2、 B3 B4
- 则向量应是A1 B2 A3 B4和A2 B1 A4 B3

1010
C = Blend (A, B, 0xA) // gives : A1 B2 A3 B4
D = Blend (B, A, 0xA) // gives : B1 A2 B3 A4
D = Shuffle (D, D, 0xB1) // gives : A2 B1 A4 B3
10 11 00 01

排序网络

○ 双调排序



寄存器内排序

```
_MM_TRANSPOSE4_PS (__m128 row0,  
__m128 row1, __m128 row2, __m128 row3)
```

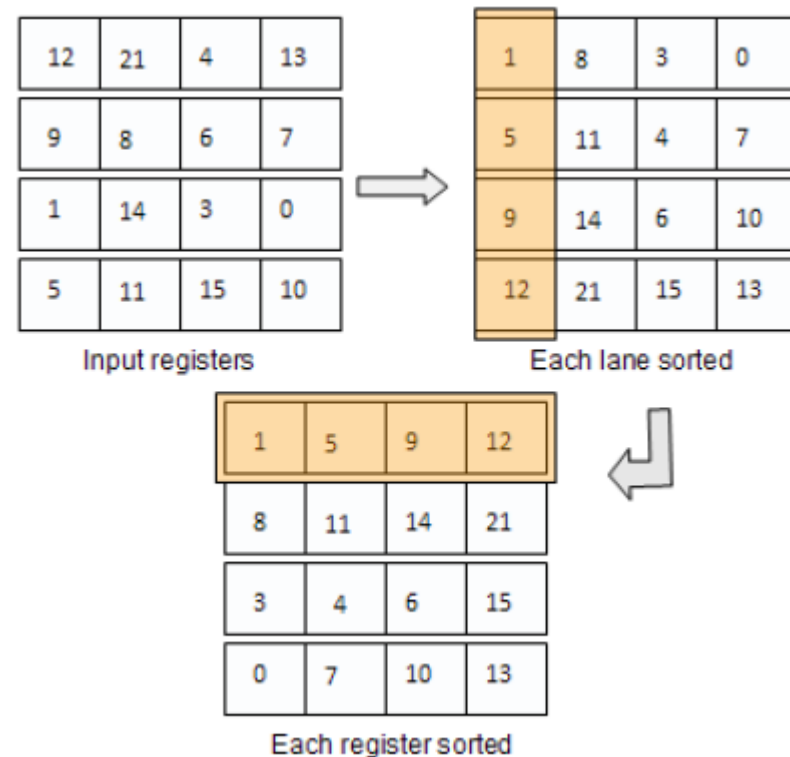
```
__m128 tmp3, tmp2, tmp1, tmp0;  
tmp0 = _mm_unpacklo_ps(row0, row1);  
tmp2 = _mm_unpacklo_ps(row2, row3);  
tmp1 = _mm_unpackhi_ps(row0, row1);  
tmp3 = _mm_unpackhi_ps(row2, row3);  
row0 = _mm_movehl_ps(tmp0, tmp2);  
row1 = _mm_movehl_ps(tmp2, tmp0);  
row2 = _mm_movehl_ps(tmp1, tmp3);  
row3 = _mm_movehl_ps(tmp3, tmp1);
```

unpacklo: r0=_A0, r1=_B0, r2=_A1, r3=_B1

unpackhi: r0=_A2, r1=_B2, r2=_A3, r3=_B3

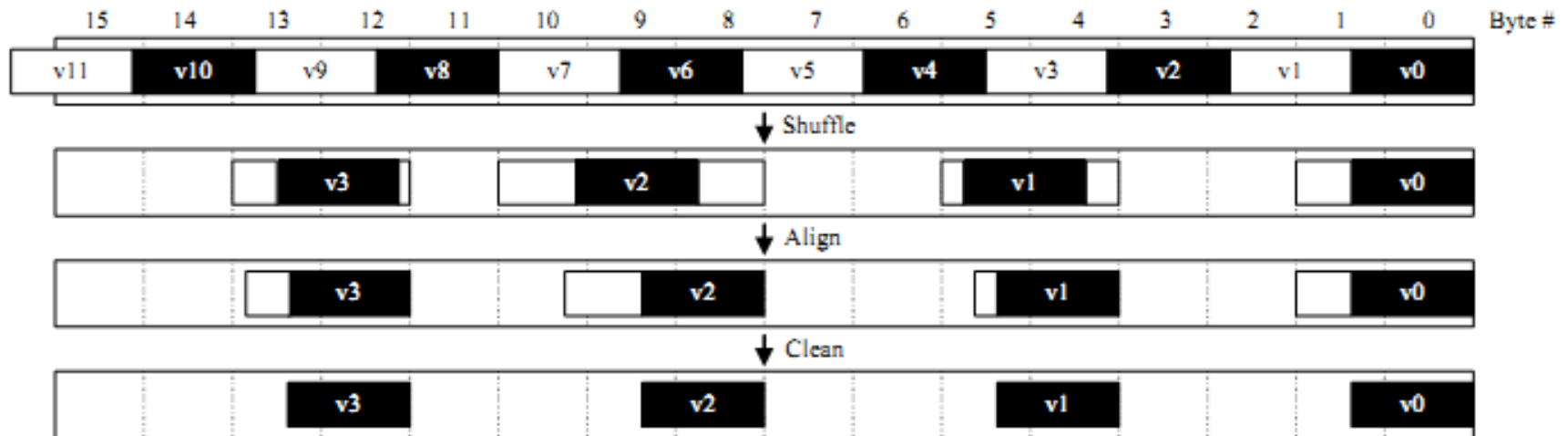
movehl: r3=_B1, r2=_B0, r1=_A1, r0=_A0

movehl: r3=_A3, r2=_A2, r1=_B3, r0=_B2



倒排索引压缩

- 每个32位整数压缩成12位存储
- 解压时用shuffle指令将4个数移动到4个32位整数的区域，在进行移位对齐完成解码





参考文献

Chhugani J, Nguyen A D, Lee V W, et al. Efficient implementation of sorting on multi-core SIMD CPU architecture[J]. Proceedings of the VLDB Endowment, 2008, 1(2): 1313-1324.

Naiyong Ao, Xiaoguang Liu, Gang Wang, Efficient Decoding of Posting Lists with SIMD Instructions. Journal of Computational Information Systems 11: 24 (2015).

张墨华，多核平台倒排索引压缩及请求处理并行算法研究，南开大学硕士学位论文，2014.