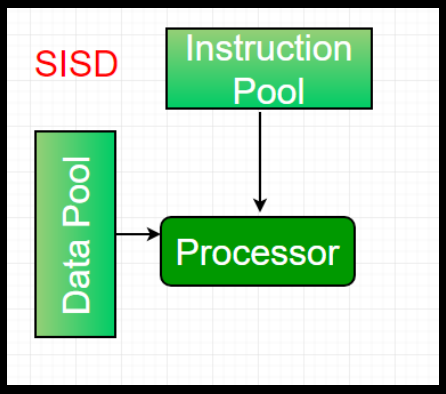
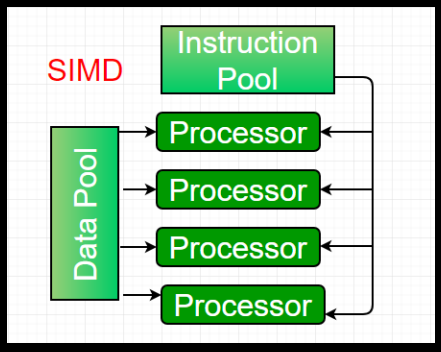


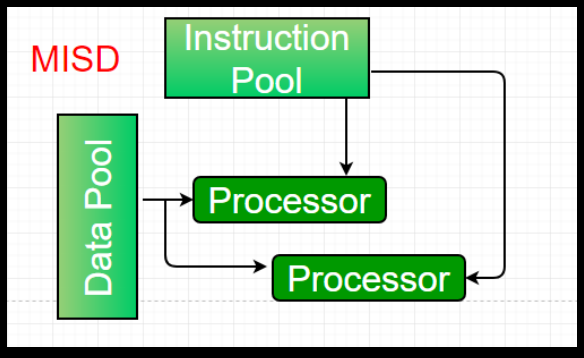
Speed limited by the rate computer transfer information internally

Instruction Streams	one	many
Data Streams: one	SISD: traditional von Neumann single CPU computer	MISD: May be pipelined Computers
Data Streams: many	SIMD: Vector processors fine grained data Parallel computers	MIMD: Multi computers Multiprocessors



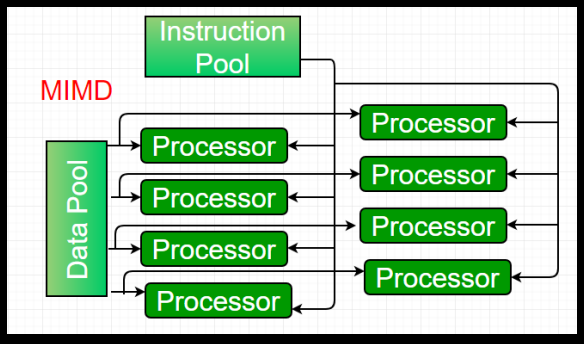
Suits for vector / matrix calculation

Single instruction multiple data



$Z = \sin(x) + \tan(x) + \cos(x)$

not used commercially



Shared / distributed memory

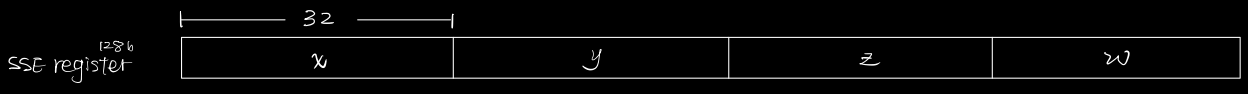
↓ all access to mem, tight couple

↓ local mems communicate through network

△ need to scale

△ no tolerance to failure

SIMD example



addps xmm0, xmm1

尽量避免浮点数运算与SIMD运算混合，否则

① 需从浮点运算寄存器将数据全部拷贝至SIMD寄存器

② 拷贝时需等待SIMD完成手头的所有工作，STALL造成时期浪费

★ 尽量将数据存在SIMD寄存器中

ex. 点积结果为标量，存储于SIMD中

可将单个浮点值存于SSE的4个位置中，表标量

```
#include <xmmintrin.h> // __m128 and SSE intrinsic functions
```

内联汇编

```
__m128 addWithAssembly(const __m128 a, const __m128 b)
{
    // a与b均已存储于xmm0与1中
    __asm addps xmm0, xmm1
}
```

xmm0负责存储结果，无需返回值

限制3 compiler 的优化空间
因直接"命令"编译器

内部函数

```
__m128 addWithIntrinsics(const __m128 a, const __m128 b)
{
    return _mm_add_ps(a,b);
}
```

如何调用函数
提供"元信息"以助优化

```
#include <xmmintrin.h>

// ……定义之前那两个函数……

void testSSE()
{
    __declspec(align(16)) float A[4];
    __declspec(align(16)) float B[4]
                          = { 8.0f, 6.0f, 4.0f, 2.0f };
    __declspec(align(16)) float C[4];
    __declspec(align(16)) float D[4];

    // 使用字面量设置 a = (1, 2, 3, 4),并且
    // 从浮点数组载入 b = (2, 4, 6, 8)
    // (只是用于演示两种方法)
    // 注意, B[] 是从后至前写入的, 因为 Intel 是小端机器
```

强制声明16字节对齐

```
    __m128 a = _mm_set_ps(1.0f, 2.0f, 3.0f, 4.0f); // 将4个float打包成
    __m128 b = _mm_load_ps(&B[0]);                 // 从内存中载入寄存器
                                                   __m128

    // 测试那两个函数
    __m128 c = addWithAssembly(a, b);
    __m128 d = addWithIntrinsics(a, b);

    // 把a和b的值存储回原来的数组,才能打印
    _mm_store_ps(&A[0], a);
    _mm_store_ps(&B[0], b);

    // 把两个结果存储至数组,以便打印
    _mm_store_ps(&C[0], c);
    _mm_store_ps(&D[0], d);

    // 检查结果(注意结果是从后往前的,因为Intel是小端机器)
    printf("a = %g %g %g %g\n", A[0], A[1], A[2], A[3]);
    printf("b = %g %g %g %g\n", B[0], B[1], B[2], B[3]);
    printf("c = %g %g %g %g\n", C[0], C[1], C[2], C[3]);
    printf("d = %g %g %g %g\n", D[0], D[1], D[2], D[3]);

    return 0;
}
```

使用SSE进行矢量&矩阵计算

4.7.4 用SSE实现矢量对矩阵相乘

让我们来看看如何用SSE实现矢量对矩阵的相乘。目的是把1×4矢量**v**和4×4矩阵**M**相乘，得出乘积矢量**r**。

$$\mathbf{r} = \mathbf{vM}$$

$$\begin{bmatrix} r_x & r_y & r_z & r_w \end{bmatrix} = \begin{bmatrix} v_x & v_y & v_z & v_w \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

$$= \begin{bmatrix} v_xM_{11} + v_yM_{21} + v_zM_{31} + v_wM_{41} \\ v_xM_{12} + v_yM_{22} + v_zM_{32} + v_wM_{42} \\ v_xM_{13} + v_yM_{23} + v_zM_{33} + v_wM_{43} \\ v_xM_{14} + v_yM_{24} + v_zM_{34} + v_wM_{44} \end{bmatrix}^T$$

此乘法涉及计算行矢量**v**和**M**矩阵列矢量的点积。若要使用SSE指令来计算，可先把**v**存储至SSE寄存器（__m128），再把**M**矩阵的每个列矢量存储至SSE寄存器。那么就可利用mulps指令，并行计算所有v_kM_{ij}：

```
__m128 mulVectorMatrixAttempt(__m128 v,
    __m128 Mcol1, __m128 Mcol2, __m128 Mcol3, __m128 Mcol4)
{
    __m128 vMcol1 = _mm_mul_ps(v, Mcol1);
    __m128 vMcol2 = _mm_mul_ps(v, Mcol2);
    __m128 vMcol3 = _mm_mul_ps(v, Mcol3);
    __m128 vMcol4 = _mm_mul_ps(v, Mcol4);
    // ……然后呢？
}
```

以上代码能求出以下这些中间结果：

$$\text{vMcol1} = [v_xM_{11} \quad v_yM_{21} \quad v_zM_{31} \quad v_wM_{41}]$$
$$\text{vMcol2} = [v_xM_{12} \quad v_yM_{22} \quad v_zM_{32} \quad v_wM_{42}]$$
$$\text{vMcol3} = [v_xM_{13} \quad v_yM_{23} \quad v_zM_{33} \quad v_wM_{43}]$$
$$\text{vMcol4} = [v_xM_{14} \quad v_yM_{24} \quad v_zM_{34} \quad v_wM_{44}]$$

但问题是这么做的话，就需要在寄存器内做加法，才能计算所需结果。例如，$r_x = v_x M_{11} + v_y M_{21} + v_z M_{31} + v_w M_{41}$，这需要把vMcol1的4个分量相加。把寄存器内的分量相加，是既困难又低效的。再者，相加后的结果将分散在4个SSE寄存器中，那么还需要把它们结合到单个结果矢量r。好在还有更好的做法。

这里的“技巧”是，使用M的行矢量相乘，而不是用列矢量。这样，就可以并行地进行加法，最终结果也会置于代表输出矢量r的单个SSE寄存器中。然而，在本技巧中不能直接用矢量v乘以M的行，而是需要用v_x乘以第1行，v_y乘以第2行，v_z乘以第3行，v_w乘以第4行。要这么做，就需要把v里的单个分量如v_x，**复制**（replicate）到其余的分量里去，生成一个$[v_x \quad v_x \quad v_x \quad v_x]$的矢量。之后就可以用已**复制**某分量的矢量，乘以M中适当的行。

幸好，有强大的SSE指令shufps（对应内部函数为_mm_shuffle_ps()）支持这种复制运算[46]。这个强大指令比较难理解，因为它是通用的指令，可把SSE寄存器的分量次序任意调乱。然而，这里只需知道以下的宏可用来复制x、y、z或w分量至整个寄存器：

```
#define SHUFFLE_PARAM(x, y, z, w) \
    ((x) | ((y) << 2) | ((z) << 4) | ((w) << 6))

#define _mm_replicate_x_ps(v) \
    _mm_shuffle_ps((v), (v), SHUFFLE_PARAM(0, 0, 0, 0))

#define _mm_replicate_y_ps(v) \
    _mm_shuffle_ps((v), (v), SHUFFLE_PARAM(1, 1, 1, 1))

#define _mm_replicate_z_ps(v) \
    _mm_shuffle_ps((v), (v), SHUFFLE_PARAM(2, 2, 2, 2))

#define _mm_replicate_w_ps(v) \
    _mm_shuffle_ps((v), (v), SHUFFLE_PARAM(3, 3, 3, 3))
```

给定这些方便的宏，就可以编写矢量矩阵乘法函数如下：

```
__m128 mulVectorMatrixAttempt2(__m128 v,
    __m128 Mrow1, __m128 Mrow2, __m128 Mrow3, __m128 Mrow4)
{
    __m128 xMrow1 = _mm_mul_ps(_mm_replicate_x_ps(v), Mrow1);
    __m128 yMrow2 = _mm_mul_ps(_mm_replicate_y_ps(v), Mrow2);
    __m128 zMrow3 = _mm_mul_ps(_mm_replicate_z_ps(v), Mrow3);
    __m128 wMrow4 = _mm_mul_ps(_mm_replicate_w_ps(v), Mrow4);
```

[46] 译注：Shuffle为洗牌、变换位置的意思。但实际上这条指令可把来源矢量的某些分量组合，复制到目标矢量的分量。

```
    __m128 result = _mm_add_ps(xMrow1, yMrow2);
    result        = _mm_add_ps(result, zMrow3);
    result        = _mm_add_ps(result, wMrow4);
}
```

这段代码产生以下的中间矢量：

$$\texttt{xMrow1} = [v_x M_{11} \quad v_x M_{12} \quad v_x M_{13} \quad v_x M_{14}]$$
$$\texttt{yMrow2} = [v_y M_{21} \quad v_y M_{22} \quad v_y M_{23} \quad v_y M_{24}]$$
$$\texttt{zMrow3} = [v_z M_{31} \quad v_z M_{32} \quad v_z M_{33} \quad v_z M_{34}]$$
$$\texttt{wMrow4} = [v_w M_{41} \quad v_w M_{42} \quad v_w M_{43} \quad v_w M_{44}]$$

把这4个中间矢量相加，就能求得结果r：

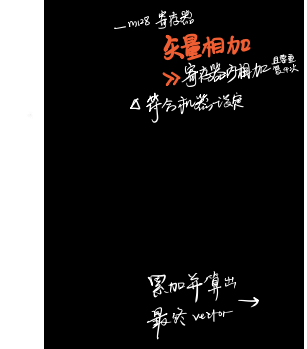
$$\mathbf{r} = \begin{bmatrix} v_x M_{11} + v_y M_{21} + v_z M_{31} + v_w M_{41} \\ v_x M_{12} + v_y M_{22} + v_z M_{32} + v_w M_{42} \\ v_x M_{13} + v_y M_{23} + v_z M_{33} + v_w M_{43} \\ v_x M_{14} + v_y M_{24} + v_z M_{34} + v_w M_{44} \end{bmatrix}^{\mathrm{T}}$$

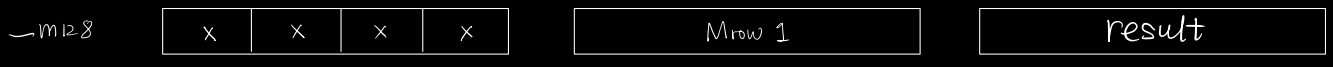
对某些CPU来说，以上代码还可以进一步优化，方法是使用相对简单的**乘并加**（multiply-and-add）指令，通常表示为madd。此指令把前两个参数相乘，再把结果和第3个参数相加。可惜SSE并不支持madd指令，但我们可以用宏代替它，效果也不错：

```
#define _mm_madd_ps(a, b, c) \
    _mm_add_ps(_mm_mul_ps((a), (b)), (c))

__m128 mulVectorMatrixFinal(__m128 v,
    __m128 Mrow1, __m128 Mrow2, __m128 Mrow3, __m128 Mrow4)
{
    __m128 result;
    result = _mm_mul_ps (_mm_replicate_x_ps(v), Mrow1);
    result = _mm_madd_ps(_mm_replicate_y_ps(v), Mrow2, result);
    result = _mm_madd_ps(_mm_replicate_z_ps(v), Mrow3, result);
    result = _mm_madd_ps(_mm_replicate_w_ps(v), Mrow4, result);
    return result;
}
```

当然，矩阵对矩阵的乘法也可以用类似方法实现。对于微软Visual Studio编译器提供的所有SSE内部函数，可参阅MSDN。






```
__m128 result;
result = _mm_mul_ps (_mm_replicate_x_ps(v), Mrow1);
result = _mm_madd_ps(_mm_replicate_y_ps(v), Mrow2, result);
result = _mm_madd_ps(_mm_replicate_z_ps(v), Mrow3, result);
result = _mm_madd_ps(_mm_replicate_w_ps(v), Mrow4, result);
return result;
```