

第 1 章

Chapter 1

X86 SSE/AVX 指令集

SSE/AVX 是 Intel 公司设计的、对其 X86 体系的 SIMD 扩展指令集，它基于 SIMD 向量化技术，提高了 X86 硬件的计算能力，增强了 X86 多核向量处理器的图像和视频处理能力。

SSE/AVX 指令支持向量化数据并行，一个指令可以同时多个数据进行操作，同时操作的数据个数由向量寄存器的长度和数据类型共同决定。例如，SSE4 向量寄存器（xmm）长度为 128 位，即 16 个字节，如果操作 float 或 int 数据，可同时操作 4 个，如果操作 char 数据，可同时操作 16 个。而 AVX 向量寄存器（ymm）长度为 256 位，即 32 字节，如果操作 char 类型数据，可同时操作 32 个，潜在地大幅度提升程序性能。

虽然 SSE4/AVX 指令向量寄存器的长度为 128/256 位，但是同样支持 64 位长度的向量操作，64 位向量映射到向量寄存器的前 64 位，这保证了兼容性。在 64 位程序下，SSE4/AVX 向量寄存器的个数是 16。

通常 SSE 指令要求访问时内存地址对齐到向量长度，主要是为了减少内存或缓存操作的次数，如果内存地址没有对齐到向量长度，则会增加访问存储器的次数。SSE4 指令要求存储器地址必须 16 字节对齐，而 AVX 指令最好也 32 字节对齐。SSE4 及以前的 SSE 指令不支持不对齐的读写操作，为了简化编程和扩大应用面，AVX 指令集支持不对齐的读写，但是性能大约会下降 20%。为了性能，在可能的情况下，还是应当使用对齐的读写操作。

SSE4/AVX 加入了 stream 的概念，意为不使用缓存的读写，因为不使用缓存则留给其他读取操作的缓存就多些，可能会提高性能。实际上这可能是一个“搬石头砸自己脚”的技术，使用不好的话，可能会对程序性能有负面的影响。笔者并不是不建议使用它，而是让读者明白，使用它可能会适得其反，因此要特别注意。

2 并行编程方法与优化实践

Intel ICC 和开源的 GCC 编译器支持的 SSE/AVX 指令的 C 接口 (intrinsic, 内置函数) 声明在 intrinsic.h 头文件中。其数据类型命名主要有 __m128/ __m256、__m128d/ __m256i, 默认为单精度 (d 表示双精度, i 表示整型)。其函数的命名可大致分为 3 个使用 “_” 隔开的部分, 3 个部分的含义如下。

- 第一个部分为 _mm 或 _mm256。_mm 表示其为 SSE 指令, 操作的向量长度为 64 位或 128 位。_mm256 表示 AVX 指令, 操作的向量长度为 256 位。本节只介绍 128 位的 SSE 指令和 256 位的 AVX 指令。
- 第二个部分为操作函数名称, 如 _add、_load、mul 等, 一些函数操作会增加修饰符, 如 loadu 表示不对齐到向量长度的存储器访问。
- 第三个部分为操作的对象名及数据类型, _ps 表示操作向量中所有的单精度数据; _pd 表示操作向量中所有的双精度数据; _pixx 表示操作向量中所有的 xx 位的有符号整型数据, 向量寄存器长度为 64 位; _epixx 表示操作向量中所有的 xx 位的有符号整型数据, 向量寄存器长度为 128 位; _epuxx 表示操作向量中所有的 xx 位的无符号整型数据, 向量寄存器长度为 128 位; _ss 表示只操作向量中第一个单精度数据; si128 表示操作向量寄存器中的第一个 128 位有符号整型。

3 个部分组合起来, 就形成了一条向量函数, 如 _mm256_add_ps 表示使用 256 位向量寄存器执行单精度浮点加法运算。

由于使用指令级数据并行, 因此其粒度非常小, 需要使用细粒度的并行算法设计。SSE/AVX 指令集对分支的处理能力非常差, 而从向量中抽取某些元素数据的代价又非常大, 因此不适合含有复杂逻辑的运算。

1.1 SSE 内置函数

本节列出 SSE4 内置函数支持的运算, 由于其命名和普通的 C 函数一致, 因此不会详细解释其含义。为了避免增加不必要的细节, 本节不包含 commit 指令和类型转换指令, 感兴趣的读者可以参考 Intel 官方手册。

1.1.1 算术运算

SSE 内置函数中支持的算术操作如表 1-1 所示。从表中可以看出, SSE 几乎支持所有常用的算术运算。故可以预知: 大多数算法只要满足向量化数据并行的特点, 就有可能能够使用 SSE 指令进行向量化。

表 1-1 SSE 算术运算

操 作	数 据	描 述
add	ss ps epi8 epi16 epi32 epi64 sd pd	加
hadd	pd ps epi16 epi32	相邻数据相加
sub	ss ps epi8 epi16 epi32 epi64 sd pd	减
hsub	pd ps epi16 epi32	相邻数据相减
addsub	ps pd	偶数索引减, 奇数索引加
mul	ss ps epi32 epu32 sd pd	乘
mulhi	epi16 epu16	取乘法结果的高位
mullo	epi16 epi32	取乘法结果的低位
div	ss ps sd pd	除
max	ss ps epi16 epu8 sd pd epi8 epi32 epu32 epu16	最大值
min	ss ps epi16 epu8 sd pd epi8 epi32 epu32 epu16	最小值
minpos	epu16	返回最小值及其索引
rsqrt	ss ps	开方的倒数
sqr	ss ps sd pd	开方
ceil	pd ps sd ss	向上取整
floor	pd ps sd ss	向下取整
abs	epi8 epi16 epi32	求绝对值
avg	epu8 epu16	求均值
dp	pd ps	依据 mask 做乘法
blend	pd ps epi16	类似 C 中的三元运算符
blendv	pd ps epi8	类似 C 中的三元运算符
sign	epi8 epi16 epi32	依据参数改变参数符号
sad	epu8	计算差的绝对值
round	pd ps sd ss	舍入
rcp	ps ss	求倒数
popcnt	u32 u64	求二进制数中为 1 的位数

表 1-1 中有一些不太常见的指令, 本节简要介绍一些。

hadd 表示将一个向量相邻的两个元素相加, 并要保持原来的向量大小。因此它具有两个参数, 如下例所示:

```
__m128 _mm_hadd_ps(__m128 a, __m128 b);
r[0] = a[0]+a[1];
r[1] = a[2]+a[3];
```

4 并行编程方法与优化实践

```
r[2] = b[0]+b[1];
r[3] = b[2]+b[3];

__m128 __mm_addsub_ps(__m128 a, __m128 b);
r[0] = a[0]-a[1];
r[1] = a[2]+a[3];
r[2] = b[0]-b[1];
r[3] = b[2]+b[3];
```

其中：r 表示结果，使用数组表示法就是表示向量中的第几个元素，如 [2] 表示其为向量寄存器中保存的第三个元素。

使用 hadd 能够比较容易地实现多个向量求内积，如代码清单 1-1 所示。

代码清单1-1 使用hadd求向量内积

```
__m128 a = _mm_mul_ps(b, c);
__m128 zero = _mm_setzero_ps();
a = _mm_hadd_ps(a, zero);
a = _mm_hadd_ps(a, zero);
```

运算完成后，a 中的第一个元素即为 b、c 两个向量的内积。

其实 addsub 应该写作 subadd，因为第一个操作是减法。hsub 和 hadd 类似，只是执行的运算是减法。

比较有意思的是 dp 操作，对于单精度浮点类型的数据，其定义如下：

```
__m128 __mm_dp_ps(__m128 a, __m128 b, const int mask);
__m128 tmp;
tmp[0] = (mask[4]==1) ? (a[0]*b[0]) : 0.0;
tmp[1] = (mask[5]==1) ? (a[1]*b[1]) : 0.0;
tmp[2] = (mask[6]==1) ? (a[2]*b[2]) : 0.0;
tmp[3] = (mask[7]==1) ? (a[3]*b[3]) : 0.0;
__mm32 tmp4
tmp4 = tmp[0] + tmp[1] + tmp[2] + tmp[3];
r[0] = (mask[0]==1) ? tmp4 : 0.0;
r[1] = (mask[1]==1) ? tmp4 : 0.0;
r[2] = (mask[2]==1) ? tmp4 : 0.0;
r[3] = (mask[3]==1) ? tmp4 : 0.0;
```

很明显，一条 dp 操作可以直接实现两个向量的内积运算，如代码清单 1-2 所示。

代码清单1-2 使用dp操作实现两个向量的内积运算

```
int mask = 1+(1<<4)+(1<<5)+(1<<6)+(1<<7);
__m128 r = _mm_dp_ps(b, c, mask);
```

blend 和 blendv 操作类似于 C/C++ 中的三目运算符 (?:)，blend 依据掩码的位选择，而 blendv 依据掩码向量中各元素的符号位选择，示例如下：

```
__m128 __mm_blend_ps(__m128 a, __m128 b, const int mask);
```

```

r[0] = (mask[0]==0) ? a[0] : b[0];
r[1] = (mask[1]==0) ? a[1] : b[1];
r[2] = (mask[2]==0) ? a[2] : b[2];
r[3] = (mask[3]==0) ? a[3] : b[3];

__m128 __mm_blendv_ps(__m128 a, __m128 b, __m128 mask);
r[0] = (mask[0]&0x80000000) ? b[0] : a[0];
r[1] = (mask[1]&0x80000000) ? b[1] : a[1];
r[2] = (mask[2]&0x80000000) ? b[2] : a[2];
r[3] = (mask[3]&0x80000000) ? b[3] : a[3];

```

很容易想到，可以使用 `blendv` 向量化一些简单的分支运算，如代码清单 1-3 所示。

代码清单 1-3 简单的可向量化的分支运算示例

```

for(int i = 0; i < n; i++){
    r[i] = a[i] > 0 ? b[i]:c[i];
}

```

这同时要求许多比较指令返回结果的符号位为 0 或 1。

1.1.2 逻辑运算

SSE 内置函数支持的逻辑运算及其数据类型如表 1-2 所示。不但支持与、或和异或，还支持移位，但是不直接支持非，这是个奇怪的地方（但可以通过 `andnot` 实现）。

表 1-2 SSE 逻辑指令

操 作	数 据	描 述
and	ps si128 pd	余
andnot	ps si128 pd	前一参数取反再与后一参数
xor	ps si128 pd	异或
or	ps si128 pd	或
sra	epi16 epi32	算术右移
srl	epi16 epi32 epi64	逻辑右移
sll	epi16 epi32 epi64	左移

以 `si128` 为例，展示 `andnot` 操作的定义如下：

```

__m128i __mm_andnot_si128(__m128i a, __m128i b);
r = (~a) & b;

```

如果 `b` 的位模式为全 1，那么 `andnot` 就实现了 `not` 运算。

1.1.3 比较

SSE 内置函数支持的比较操作及其数据类型如表 1-3 所示。不但有测试向量中元素是否

6 并行编程方法与优化实践

为全 0 或全 1 的指令，还有几乎所有可能的比较测试指令。

表 1-3 SSE 判断指令

操 作	数 据	描 述
cmp	pd ps sd ss	比较
test_all_ones	_m128i 向量	测试是否所有位都是 1
test_all_zeros	_m128i 向量	测试和掩码的位与结果是否是 0
cmpeq	ss ps epi8 epi16 epi32 sd pd epi64	比较是否相等
cmplt	ss ps sd pd	测试是否小于
cmple	ss ps sd pd	小于等于
cmpgt	ss ps epi8 epi16 epi32 sd pd epi64	大于
cmpge	ss ps sd pd	大于等于
cmpneq	ss ps sd pd	不等于
cmpnlt	ss ps sd pd	不小于
cmpngt	ss ps sd pd	不大于
cmpnge	ss ps sd pd	不大于等于
cmpnle	ss ps sd pd	不小于等于

比较指令返回的结果是掩码，即如果比较成立的话，则对应位置的值为 1，否则为 0。

下面给出通用的 cmp 指令定义，其他的 cmp* 指令的语义都可归为 cmp 的某种特殊情况。

```
_m128 _mm_cmp_ps(_m128 a, _m128 b, const int mask)
r[0] = (a[0] op b[0]) ? 0xffffffff:0;
r[1] = (a[1] op b[1]) ? 0xffffffff:0;
r[2] = (a[2] op b[2]) ? 0xffffffff:0;
r[3] = (a[3] op b[3]) ? 0xffffffff:0;
```

其中 mask 的数值表示了执行的具体比较操作 (op)，可查阅 Intel 指令集手册，本节就不详细列出了。

1.1.4 加载和存储

SSE 内置函数支持的加载和存储操作及其数据类型如表 1-4 所示。不但支持正向设置、逆向设置、清零、全部设置为某值，还支持不经过缓存的加载和存储，以及对应的存储和加载。

表 1-4 SSE 加载和存储

操 作	数 据	描 述
set	ss sd ps pd epi64 epi32 epi16 epi8	设置为某值

(续)

操 作	数 据	描 述
setl	ps pd epi64 epi32 epi16 epi8	全部设置为某值
setr	ps pd epi64 epi32 epi16 epi8	反向设置
setzero	ps pd si128	清零
stream	ps si128 si32 pi pd	不经过缓存的存储
stream_load	si128	不经过缓存的加载
store	ss sd ps pd si128	将向量存储到数组
storel	ps pd	存储第一个数据
storeh	pi pd	存储高位数据
storel	pi epi64 pd	存储低位数据
storer	ps pd	逆序存储
storeu	ps pd si128	不对齐存储
prefetch	char*	预取数据到缓存层次中
move	ss sd epi64	复制
movehl	ps	复制高位
movelh	ps	复制低位
load	ss sd ps pd si128	从数组中读取向量
loadl	ps pd	读取首个数据
loadr	ps pd	逆向读取
loadu	ps pd si128	不对齐读取
lddqu	si128	读取不对齐的整数
extract	epi8 epi16 epi32 epi64	依据标签提取
insert	epi8 epi16 epi32 epi64 ps	依据掩码决定插入元素和位置

其中 extract 操作单精度浮点数据的定义如下：

```
int _mm_extract_ps(__m128 a, const int id);  
r = a[id];
```

在 SSE 指令中，extract 是非常耗时的操作，故不建议使用。

insert 按照掩码的对应比特值决定对应位量元素乘积是否相加，并将结果写入掩码指定位置，如下所示：

```
__m128 _mm_insert_ps(__m128 a, __m128 b, const int id);  
int bid = value of bit 6-7 for id  
int bv = b[bid];
```

8 并行编程方法与优化实践

```
bid = value of bit 4-5 for id
r[0] = (bid == 0) ? bv : a[0];
r[1] = (bid == 1) ? bv : a[1];
r[2] = (bid == 2) ? bv : a[2];
r[3] = (bid == 3) ? bv : a[3];

r[0] = (0x1 & id == 1) ? 0.0 : r[0];
r[1] = (0x2 & id == 1) ? 0.0 : r[1];
r[2] = (0x4 & id == 1) ? 0.0 : r[2];
r[3] = (0x8 & id == 1) ? 0.0 : r[3];
```

掩码的第 6、7 位决定了取 b 的哪个元素取代 a 中元素，而第 4、5 位决定了 a 的哪个元素被取代。

1.2 AVX 内置函数

Intel 从 SNB (Sandy Bridge) 和 IVB (Ivy Bridge) 架构开始支持 256 位的 SIMD 浮点指令集 AVX，即可以同时执行 8 个单精度 float 运算或者 4 个双精度 double 运算。本节的 AVX 内置函数是指 Intel Haswell 及更新的处理器支持的 AVX/AVX2/FMA 指令的 C 封装。限于篇幅，笔者不罗列所有 AVX 函数，只罗列笔者认为常用的函数。

1.2.1 算术运算

AVX 内置函数中支持的算术运算如表 1-5 所示。

表 1-5 AVX 算术运算

操 作	数 据	描 述
abs	epi16 epi32 epi8	绝对值
add	epi16 epi32 epi64 epi8 ps pd	加
hadd	ps pd	相邻加
sub	ps pd	减
hsub	ps pd	相邻减
addsub	ps pd	偶数索引减，奇数加
mul	ps pd	乘
fmadd	ss sd ps pd	乘加 ($a*b+c$)
fnmadd	ss sd ps pd	$c-a*b$
fmsub	ss sd ps pd	乘减 ($a*b-c$)
fnmsub	ss sd ps pd	$-(a*b+c)$
div	ps pd	除

(续)

操 作	数 据	描 述
max	ps pd epi8 epi16 epi32	最大值
min	ps pd epi8 epi16 epi32	最小值
rsqrt	ps	开方倒数
sqrt	ps pd	开方
ceil	pd ps	向上取整
floor	pd ps	向下取整
dp	ps	依据 mask 做乘法
round	pd ps	舍入
rcp	ps	求倒数
blend	epi16 epi32 ps pd	三目运算
blendv	epi8 ps pd	三目运算
sign	epi8 epi16 epi32	符号

新增加的乘加 / 乘减系列指令, 使得优化人员更易于发挥处理器的峰值计算能力。在之前的 Intel 处理器上, 要同时发挥处理器乘法和加法的性能需要非常小心地安排指令系列才有可能。许多科学计算任务 (如矩阵运算) 的核心运算就是乘加, FMA 指令的出现使得科学计算应用比以前更易于发挥处理器的计算能力, 如代码清单 1-4 代码所示。

代码清单 1-4 fma 示例

```
for(int i = 0; i < numRows; i++){  
    for(int j = 0; j < numColumns; j++){  
        float ret = r[i*numColumns+j]*beta;  
        float sum = 0.0f;  
        for(int k = 0; k < K; k++){  
            sum += a[i*K + k]*b[k*numColumns+j];  
        }  
        ret += sum*alpha;  
        r[i*numColumns+j] = ret;  
    }  
}
```

假设数组 a、b 和 r 都可以保存到一级缓存中, 即假设读取数据不会成为瓶颈。在这个前提下, 代码清单 1-4 中的时间主要消耗在计算乘法和加法上。如果没有 FMA 指令, 那么为了获得最高性能, 代码优化人员需要依据处理器发射能力、算法和加法流水线的计算能力、乘法和加法指令的依赖关系、并行度等相关因素, 合理地安排指令调度, 这会相当复杂。而 FMA 指令的存在使得代码优化人员只需要关注一种指令即可, 这就简化了指令调度工作。

10 ◆ 并行编程方法与优化实践

1.2.2 逻辑运算

AVX 内置函数支持的逻辑运算及其数据类型如表 1-6 所示。

表 1-6 AVX 逻辑指令

操 作	数 据	描 述
and	ps pd si256	余
andnot	ps pd si256	取反再与
xor	ps pd si256	异或
or	ps pd	或
sll	epi16 epi32 epi64	逻辑左移
sra	epi16 epi32	算术右移

这些操作和 SSE 指令基本上一致，所不同的是：向量寄存器长度由 128 位提升到 256 位，因此不加以详细说明。

1.2.3 比较

AVX 内置函数支持的比较操作及其数据类型如表 1-7 所示。

表 1-7 AVX 判断指令

操 作	数 据	描 述
cmp	pd ps	比较
cmpeq	epi8 epi16 epi32 epi64	相等比较
cmpgt	epi8 epi16 epi32 epi64	大于比较

AVX 的比较指令通常生成“掩码”，AVX 支持一些掩码的条件执行，适当使用这些指令能够向量化一些具有简单判断逻辑的运算。

1.2.4 加载和存储

AVX 内置函数支持的加载和存储操作及其数据类型如表 1-8 所示。

表 1-8 AVX 加载和存储

操 作	数 据	描 述
set	ps pd epi64x epi32 epi16 epi8	设置
set1	ps pd epi64x epi32 epi16 epi8	全部设置为某值
setr	ps pd epi64x epi32 epi16 epi8	逆向设置
setzero	ps pd si256	设置为零

(续)

操 作	数 据	描 述
broadcast	ps pd ss sd	广播
shuffle	ps pd epi32 epi8	交换向量中元素位置
gather	epi32 epi64 ps pd	收集
stream	ps pd si256	不加缓存的读取
store	ps pd si256	对齐存储
storeu	ps pd si256	非对齐存储
load	ps pd si256	对齐加载
loadu	ps pd si256	非对齐加载
lddqu	si256	读取不对齐的整数
maskload	ps pd	依据掩码读取
extract	epi8 epi16 epi32 epi64	从向量中抽取元素

broadcast 指令将一个标量复制成一个向量，或将一个长度为 128 位的向量复制一份，组成一个 256 位的向量。

gather 指令从地址不连续的内存中取出多个元素，组成一个向量，使得某些以前不能被量化的应用也可以被向量化。

shuffle 指令依据参数选择从两个向量中获得元素，以组成一个新向量。

1.3 优化实例及分析

本节以计算圆周率、稀疏矩阵向量乘法和二维卷积为例，介绍如何使用 AVX 指令优化程序性能。在正式介绍之前，为了更好地说明 X86 处理器 SIMD 指令的情况，首先介绍如何测得处理器的浮点峰值性能。

1.3.1 如何测得 CPU 的浮点峰值性能

一颗 CPU 的理论浮点峰值性能反映了这颗 CPU 的最大浮点处理能力，通常也称为 CPU 的吞吐量。一般来说，CPU 浮点峰值统计的浮点计算类型只包含乘法和加法（包括减法）。这主要是因为：浮点乘法和加法构成了很多广泛使用且基础的重要算法的核心，如矩阵乘法、FFT 等。

本节以 Intel Sandy Bridge（简称 SNB）和 Ivy Bridge（简称 IVB）架构的 CPU 为例，介绍如何计算理论浮点峰值，并实测出该峰值。这里以单精度峰值为例，双精度一般为单精度的 1/2，测试方法也相同。

12 ❖ 并行编程方法与优化实践

计算理论浮点峰值实际上就是计算 CPU 每个处理核心一个周期可以处理的浮点乘法和浮点加法的次数。查看 Intel 的手册知道, 在 SNB 和 IVB 架构下, 浮点乘法的延迟是 5, 吞吐量是 1; 浮点加法的延迟是 3, 吞吐量是 1。可以得到 SNB 和 IVB 的单精度浮点峰值计算公式如下:

$$\text{单精度浮点峰值} = \text{CPU 核心数} \times \text{频率} \times (8\text{Mul} + 8\text{Add})$$

市面上的一颗普通的 Intel i3 2100 的 SNB 处理器, 双核, 频率是 3.1GHz, 那么它的单精度浮点峰值就是 $2 \times 3.1\text{GHz} \times (8+8) = 99.2\text{GFLOPS}$ 。

AVX 中的 vmulps 和 vaddps 指令就是计算 256 位单精度浮点乘法和加法的指令。要同时发挥乘法和加法性能, 就要求相邻的乘法和加法指令没有依赖关系, 而 SNB 和 IVB 架构一个时钟周期最多可以发射 4 条指令, 故处理器的指令发射能力不会成为瓶颈。了解了这些信息之后, 相应的测试程序也就容易写了, 汇编代码如代码清单 1-5 所示。

代码清单 1-5 测试 SNB CPU 峰值性能程序

```
mov $0x1000000, %rax           @ 0x1000000 是循环次数, 根据需要可以放大或改小
v xorps %ymm0, %ymm0, %ymm0
v xorps %ymm1, %ymm1, %ymm1
v xorps %ymm2, %ymm2, %ymm2
v xorps %ymm3, %ymm3, %ymm3
loop:
v mulps %ymm1, %ymm1, %ymm0
v addps %ymm3, %ymm3, %ymm2
subq $0x1, %rax
jne loop
```

代码中前 5 行是初始化寄存器状态, 设置 rax 为循环计数器, 主要在 loop 循环内不断执行两条乘法和加法指令。由于 SNB 架构对 add 和 sub 类指令也可以做微指令融合, 所以 subq 和 jne 两条指令可以与 vmulps 和 vaddps 在同一个周期发射。vmulps 和 vaddps 之间没有数据依赖, 前后两次循环之间也没有数据依赖。

其余的工作就是开启和处理器核数同样多的 C 语言线程调用, 且每个线程绑定到不同的处理器核, 然后调用这个函数, 用计时器记录多线程执行的时间。利用公式 $16 \times \text{线程数} \times \text{循环次数} / \text{时间}$, 即可得到这颗处理器的实测 GFLOPS 峰值性能。在笔者的 i3 2100 SNB 处理器上实测得到 97.87GFLOPS, 很接近理论峰值 99.2GFLOPS。

实际上, 代码清单 1-5 在 X86 处理器上能够获得接近峰值的性能有以下原因:


1) 寄存器重命名机制。前后两次循环都写入同一个寄存器, 这是一个典型的“写后写”依赖。如果没有寄存器重命名机制的帮助, 那么后一条指令就必须等待前一条指令执行完成, 形成了一条“完美”的关于寄存器的数据依赖链, 那么就不可能接近峰值。

2) 预测执行。如果没有预测执行机制, 则下次循环只能等待循环条件判断执行完成之后

才能开始执行，这样就会基于判断条件指令形成一条依赖链（控制依赖），也就不可能获得接近峰值的性能。

3) 4 发射。循环内部至少有 4 条指令，如果发射能力小于 4 的话，那么发射能力就成为了瓶颈。比如假设处理器的指令发射能力为 2，那么一次循环需要 2 个周期才能发射完，也不可能达到接近峰值的水平。

4) 4 条指令发射到不同的流水线上。如果有两个指令发射到同一条流水线上，那么这两条指令就必须相互等待（只有一条执行完成，另一条才有可能得到执行）。而若 4 条指令发射到不同的流水线上的话，那么这 4 条指令可以同时在不同的流水线上执行，即不存在结构化瓶颈。

 **注意** 这个方法在同一颗处理器上测出的单核和多核性能，不一定呈线性倍数关系，原因在于，现在 Intel 的处理器很多都有 turbo boost 技术，会根据负载给处理器自动超频，一般单核负载时超频更高，多核时较低，甚至完全不超频。所以除非关掉 turbo boost，或者在没有 turbo boost 支持的处理器上测试，才可能呈现倍数关系。

Intel 最新的 Haswell 架构处理器开始支持 256 位的浮点融合乘加指令 FMA，即将乘法和加法融合在一条指令中（即 $a \times b + c$ ）。FMA 指令的吞吐量是 2，延迟是 5。由于 Intel 设计的是 FMA3 指令，即三操作数的 FMA 指令，所以在 a、b 和 c 三个输入向量中，会有一个作为输出结果。这就导致我们在写 FMA 的峰值测试程序时，无法满足前后两个循环的数据无依赖特征这个条件，导致下一个循环需要等待上一个循环的结果计算完毕才开始执行。而 FMA 的延迟有 5 个周期，即两个循环之间的延迟需要 5 个周期，这会极大地拖慢处理速度，也造成了流水线的浪费。

为了解决这个问题，测试程序需要在一次循环内添加足够多的数据无依赖关系的 FMA 指令，将流水线填满，下次循环到同一条指令时，上个循环的该指令延迟已经结束，可以立即发射。由于一个周期可以同时发射两条 FMA 指令，且延迟是 5，那么需要在一个循环内加入 10 条 FMA 指令方能填满执行流水线，如代码清单 1-6 所示。

代码清单 1-6 测试 Haswell 处理器峰值性能

```
loop:
vfmadd231ps %ymm0, %ymm0, %ymm0    @ 周期0,5,10,15...发射
vfmadd231ps %ymm1, %ymm1, %ymm1    @ 周期0,5,10,15...发射
vfmadd231ps %ymm2, %ymm2, %ymm2    @ 周期1,6,11,16...发射
vfmadd231ps %ymm3, %ymm3, %ymm3    @ 周期1,6,11,16...发射
vfmadd231ps %ymm4, %ymm4, %ymm4    @ 周期2,7,12,17...发射
vfmadd231ps %ymm5, %ymm5, %ymm5    @ 周期2,7,12,17...发射
vfmadd231ps %ymm6, %ymm6, %ymm6    @ 周期3,8,13,18...发射
vfmadd231ps %ymm7, %ymm7, %ymm7    @ 周期3,8,13,18...发射
vfmadd231ps %ymm8, %ymm8, %ymm8    @ 周期4,9,14,19...发射
vfmadd231ps %ymm9, %ymm9, %ymm9    @ 周期4,9,14,19...发射
```

14 ❖ 并行编程方法与优化实践

```
subq $0x1, %rax  
jne loop
```

本节以这个例子作为基础，主要阐释了浮点运算的量化准则和流水线技术，并初步展示了达到高浮点吞吐量的基本方法。下面 3 节中的例子将会由简入难地展示实际问题在 Intel x86 处理器下性能优化的各种方法和实战技巧。

1.3.2 积分计算圆周率 π

这个简单的例子展示了使用指令级并行、数据级并行技术的初级方法，并且不涉及存储器和缓存的使用。

使用以下公式：

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx$$

并采用离散积分的方法求 π 值。

1) 给出一个简单的版本 1 作为基准 (baseline)，具体如代码清单 1-7 所示。

代码清单 1-7 版本 1：计算圆周率的基准版

```
double compute_pi_naive(size_t dt) // version 1  
{  
    double pi = 0.0;  
    double delta = 1.0 / dt;  
    for (size_t i = 0; i < dt; i++) {  
        double x = (double)i / dt;  
        pi += delta / (1.0 + x * x);  
    }  
    return pi * 4.0;  
}
```

dt 为 [0, 1] 区间分段的大小，理论上 dt 越大计算结果越精确，但同时计算量也越大；delta 为每个分段的长度；x 是第 i 个分段的横坐标，用它带入积分函数，并乘以 delta 表示每个分段的面积；用 pi 累加所有的分段面积，乘以 4 并返回结果。

在笔者的 Intel i3 2100 SNB 处理器上，这个版本在输入 dt=128M 时的执行时间约为 1.919 秒。为了方便比较，将这个版本的性能标准化，记为 1。

2) 下面的版本 2 将在单线程内使用 SIMD 指令数据并行技术进一步缩短执行时间。SNB 处理器支持 AVX 指令集，这里使用 AVX 指令来向量化单线程内部的计算，如代码清单 1-8 所示。

代码清单 1-8 版本 2：在线程内部使用 SIMD 数据并行技术加速线程执行

```
double compute_pi_omp_avx(size_t dt){  
    double pi = 0.0;  
    double delta = 1.0 / dt;  
    __m256d ymm0, ymm1, ymm2, ymm3, ymm4;
```



```
ymm0 = _mm256_set1_pd(1.0);
ymm1 = _mm256_set1_pd(delta);
ymm2 = _mm256_set_pd(delta * 3, delta * 2, delta, 0.0);
ymm4 = _mm256_setzero_pd();

for (int i = 0; i <= dt-4; i += 4) {
    ymm3 = _mm256_set1_pd(i * delta);
    ymm3 = _mm256_add_pd(ymm3, ymm2);
    ymm3 = _mm256_mul_pd(ymm3, ymm3);
    ymm3 = _mm256_add_pd(ymm0, ymm3);
    ymm3 = _mm256_div_pd(ymm1, ymm3);
    ymm4 = _mm256_add_pd(ymm4, ymm3);
}

double tmp[4] __attribute__((aligned(32)));
_mm256_store_pd(tmp, ymm4);
pi += tmp[0] + tmp[1] + tmp[2] + tmp[3];

return pi * 4.0;
}
```

`_mm256_set1_pd()` 语句用于将一个 `double` 型浮点值赋给一个 `_m256` 变量的所有位置；`_mm256_set_pd(e3, e2, e1, e0)` 是将 4 个参数依次放入 `_m256` 变量的各个位置，参数顺序和存储顺序相反。注意，这两个 `intrinsic` 实际上并不对应单一条指令，而是多条指令混合而来，故应尽量减少将其用于最内层循环。

程序的最内层循环的前两条语句首先构造出递增的 `x` 的向量表示，然后进行向量的自乘加 1，再被 `delta` 的向量除，加到一个累加向量 `ymm4` 中。

这样就完成了初步的向量化工作。经测试，同样在 i3 2100 的 CPU 上，相比没做向量化的程序加速比达到 2.6x 左右。256 位的向量寄存器可以同时存储计算 4 个 `double` 类型的浮点数，但由于浮点标量除法和向量除法的延迟不同，并不能达到 4x 的加速比。

3) 初步的向量化并没有充分填充向量指令的流水线，即没有掩盖浮点计算的延迟。版本 3 通过使用循环展开技术，以使用更多的寄存器，来解除数据依赖，填充浮点计算流水线（这里仅列出循环内代码），如代码清单 1-9 所示。

代码清单1-9 版本3：使用循环展开技术解除数据依赖

```
ymm5 = _mm256_setzero_pd();
for (i = 0; i <= dt - 8; i += 8){
    ymm3 = _mm256_set1_pd(i * delta);
    ymm3 = _mm256_add_pd(ymm3, ymm2);
    ymm3 = _mm256_mul_pd(ymm3, ymm3);
    ymm3 = _mm256_add_pd(ymm0, ymm3);
    ymm3 = _mm256_div_pd(ymm1, ymm3);
    ymm4 = _mm256_add_pd(ymm4, ymm3);

    ymm6 = _mm256_set1_pd((i+4) * delta);
```

16 ❖ 并行编程方法与优化实践

```
ymm6 = _mm256_add_pd(ymm6, ymm2);  
ymm6 = _mm256_mul_pd(ymm3, ymm6);  
ymm6 = _mm256_add_pd(ymm0, ymm6);  
ymm6 = _mm256_div_pd(ymm1, ymm6);  
ymm5 = _mm256_add_pd(ymm5, ymm6);  
}  
ymm4 = _mm256_add_pd(ymm4, ymm5);
```

ymm3 和 ymm4 串起两条独立的指令依赖关系，在 X86 这种乱序多发射的指令调度里可以并行执行，或者互相填充流水线，最后都加到 ymm5 上。测试表明，相对初步向量化加速 50% 左右。

4) 将前面 3 个版本综合，如表 1-9 所示。

表 1-9 积分求 π (3 个版本性能比较)

基准版本	初步向量化版本	向量流水化版本
1	2.6	3.9

从表中可以看出，SIMD 指令向量化和循环展开以优化流水线效率，都获得了比较好的性能提升。由于本章的主要目的是介绍 Intel X86 CPU 上的 SSE/AVX 指令优化技术，故并不会提到多线程优化技术。

1.3.3 稀疏矩阵向量乘法

稀疏矩阵向量乘法 (SPMV) 可在很多情况下代替稠密矩阵运算，可以大量节省内存占用，减少计算开销。矩阵向量乘法不同于矩阵和矩阵的乘法，这是完全访存密集型的计算，我们主要的优化方向是提升访存效率或减少访存开销。

稀疏矩阵一般只存储非零元的信息，非零元的存储格式决定了访存的模式，这需要根据非零元的分布模式和要做的计算类型来设计。我们假设分布模式并非对角线分布，整体分布较均匀，局部可能会有聚集，计算类型是稀疏矩阵乘以稠密向量，结果为稠密向量。

标准的稀疏矩阵存储格式主要有：COO (Coordinate Format) 和 CSR (Compressed Sparse Row) 等。COO 很简单，就是使用 3 个数组，分别存储全部非零元的行下标 (row index)、列下标 (column index) 和值 (value)；CSR 稍复杂，对行下标进行了压缩，假设矩阵行数是 m ，则压缩后的数组长度为 $m+1$ ，记作 (row ptr)，其中第 i 个元素 (0-base) 表示矩阵前 i 行的非零元个数。

图 1-1 和图 1-2 展示了 COO 和 CSR 格式存储稀疏矩阵的一个例子。

1							
			1	1			
	1	1				1	
		1					1
	1				1		
		1	1				1
1		1		1			
			1				1

稀疏矩阵示例

我们来考虑矩阵向量乘法计算 $y=Ax$ ，其中 A 是稀疏矩阵，图 1-1 一个稀疏矩阵的例子

维度是 m 和 n ，非零元个数是 k ； x 和 y 是稠密向量，维度分别是 n 和 m ， $m \times n \gg k \gg \max(m, n)$ 。做这个稀疏矩阵向量乘法就要遍历 A 的每一行，和 x 对应位置相乘，把结果累加到 y 的对应位置。这个过程对 A 的 k 个非零元全部访问了一遍，对 x 也访问了 k 个元素（重叠），对 y 访问了一遍，所以优化重点在于减少访问 A 的冗余，并提升访问 x 的效率。下面这几个优化标准稀疏矩阵存储格式的方法，可以提升访存效率，减少冗余。

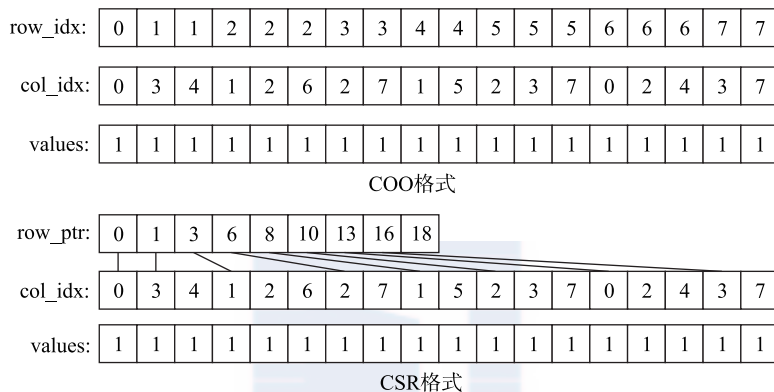


图 1-2 COO 和 CSR 格式的稀疏矩阵存储方法

（1）对矩阵 A 做行列分块处理

对 x 的访问每次总是从左到右进行稀疏的遍历，如果 n 很大（比如上百万甚至更多），则访问 x 的空间局部性较差。所以我们首先改进矩阵 A 的访问顺序，将矩阵 A 分解成多个方形的子矩阵。子矩阵的维度适应较高层 CPU 硬件 cache 的大小，这样在遍历每一个子矩阵时，对 x 的访问相对集中于一个较小的区间，这个区间内的 x 会被 cache 缓存，这样能够大大提高访问效率。分块方式如图 1-3 所示。

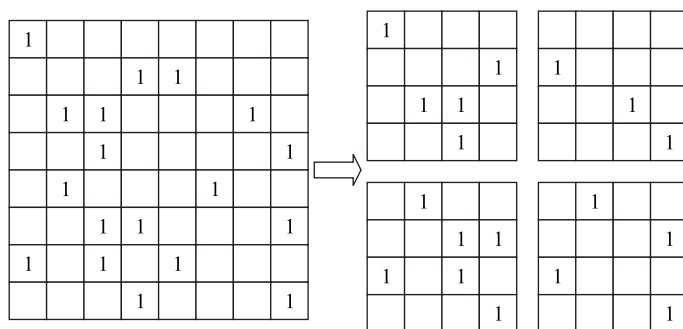


图 1-3 稀疏矩阵的分块存储

（2）自适应分块存储结构

由于稀疏矩阵的非零元分布不一定均匀，有的分块会非常稀疏，有的则会相对稠密。对

于极稀疏的分块（非零元数量远小于行数），如果用和 CSR 相似的压缩行存储策略，则会浪费空间，所以用 COO 的方式反而更能节省存储空间，提高访问效率。

对于哪些分块使用 CSR，哪些使用 COO 方式，可以通过实验的方式确定一个非零元的数量和分块大小的比值。高于该值的用 CSR 方式存储，否则用 COO 方式存储。

如图 1-4 所示，一共使用 5 个数组存储自适应分块信息的稀疏矩阵，灰色的部分是 CSR 的相关信息，白色的部分是 COO 的相关信息。col_idx 和 vals 的意义不变；types 存储分块类型，标识当前分块是 CSR 还是 COO；如果当前分块是 CSR，则 row_info 存储类似 row_ptr 的信息（第 k 个元素表示分块内第 k 行的非零元个数），否则存储 COO 的 row_idx 的信息；row_id 存储每个分块在 row_info 上的起始地址。

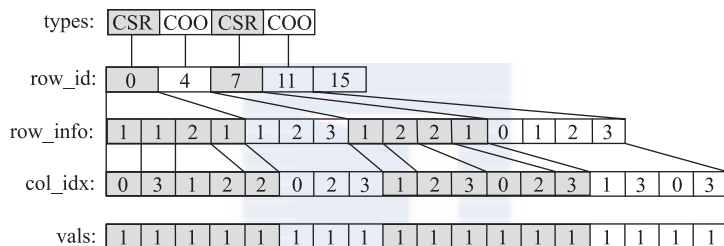


图 1-4 自适应分块的稀疏矩阵存储格式

（3）减少下标存储的冗余

矩阵分块后，分块内间址的下标并不需要 4 字节 int 型整数存储，比如分块维度在 64K 以内，可以用 2 字节的 unsigned short 来存储。这样，无论是 CSR 或 COO 的 row_idx、row_ptr，还是 col_idx，都可以减少 50% 的存储空间，并同时提升访存效率。

（4）多线程和 NUMA 特性

单处理器多核多线程并行计算稀疏矩阵向量乘的过程比较简单，只需把矩阵划分成线程数量的子矩阵。这里采用横切的方法，计算结果不用合并。

但是对于多处理器非一致内存访问（NUMA），就需要对数据在内存中的分布做特殊处理，才能最大程度地利用全部的内存带宽。

一个典型的 Intel X86 双路服务器的拓扑架构如图 1-5 所示。

Memory #0 是 CPU #0 的本地内存，Memory #1 是 CPU #1 的本地内存，它们有各自独立的内存带宽。CPU #0 访问 Memory #1 需要经过内部总线（在 Intel 的架构中叫 QPI 总线），这个总线的带宽一般小于内存带宽。另外如果要访问的数据只集中在一颗 CPU 的本地内存中，那么只能利用一个 NUMA node 的内存带宽，这就限制了系统的总体吞吐。

所以需要把稀疏矩阵的存储均匀地分配到两颗处理器各自的本地内存中。对于一个双 CPU，每颗 CPU 一共 4 核的系统，需要开 8 个线程，并把这 8 个线程分别绑定到 8 颗 CPU

核上,使线程的上下文不会在核间迁移。对于每个线程要处理的稀疏矩阵数据,也通过系统调用(在 Linux 中是 mbind),绑定到所在 CPU 核的本地内存中。这样每个核处理的数据一定是从本地内存中获得的,不会经过 QPI 总线。这就最大程度地利用了系统内存的带宽。经过实测,这个优化方法可以提升 70% 左右的内存带宽。

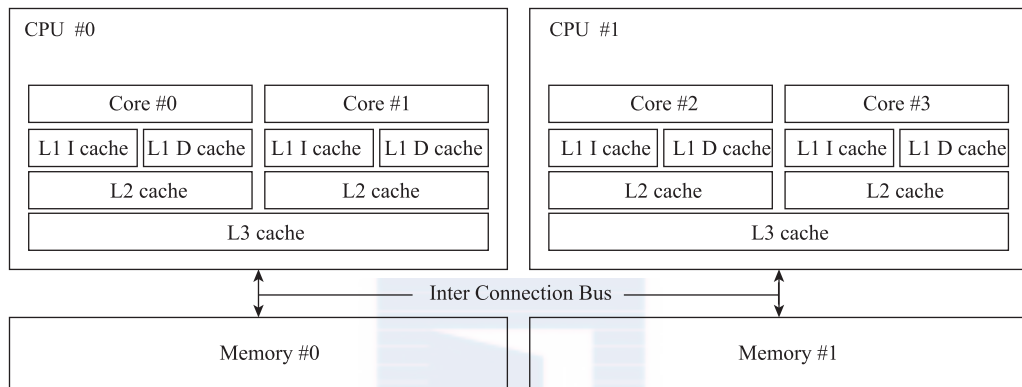


图 1-5 双路 NUMA 服务器系统互连拓扑结构

对于我们测试的一个维度大约 1M、稀疏度 0.0001 的稀疏矩阵来说,所有优化加起来,相对 Intel MKL 库中 CSR 矩阵的 SpMV API 加速了 2.5x 左右。学术界还有很多针对稀疏矩阵存储格式的讨论和研究,其中有些还利用了 SIMD 向量指令,这里介绍的稀疏矩阵乘法方法,更多是为了讨论内存和 cache 优化的一些基本原理。稀疏矩阵根据稀疏度和非零元分布的不同,需要使用不同的存储策略,所以遇到实际的稀疏矩阵问题,需要根据实际情况开发不同的存储格式,不必局限于本节描述的方法。

1.3.4 二维单通道图像离散卷积

在现在流行的深度神经网络做图像识别的应用中,有一种关键的运算是计算二维图像卷积。

$$f * h = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) h(x-m, y-n)$$

这里我们只考虑单通道和单精度浮点类型的二维离散卷积,输入是一个由浮点的二维数组保存的图像,还有一个 $m \times n$ 的模板 kernel, m 和 n 一般相等,常用的大小有 3、5、7...等。设图像大小是 $M \times N$, 则浮点计算次数是 $2 \times (M-m+1) \times (N-n+1) \times m \times n$ 。如果 m 和 n 比较大,则这是一个典型的计算访存比很高的计算过程。经过精心调优,应该可以充分利用 CPU 的浮点运算性能进行优化。

这里主要的难点是如何利用 SIMD 数据并行计算,并做好指令流水调度。对于 4×4 或者 8×8 这样的 kernel,可以直接对 kernel 做 SIMD 向量化,但这种方法限制太多,对于非 4 倍

大小的 kernel 会有冗余和截断, 指令流水长度也会受限, 所以需要换一个并行的维度。

一般情况下, 输入图像和输出图像的大小不会太小, 可以在输出图像上直接做寄存器分块。SSE 指令的向量寄存器长度是 4 个 float, AVX 或 FMA 指令是 8 个 float。前面章节讨论过, SSE 和 AVX 的乘加延迟总和都是 8 个周期, FMA 是 10 个周期。综上, 对于 SSE 指令, 可以构造 8×4 的寄存器分块; AVX 指令可以构造 8×8 的分块; FMA 指令可以构造 10×8 的分块。

下面我们就以 AVX 指令为例来说明如何计算一个寄存器分块。这里以 3×3 的 kernel 为例。

X64 指令集有 16 个向量寄存器, 使用 8 个向量寄存器保存当前计算的 8×8 个输出结果, 首先初始化为 0; 从左到右每次从 kernel 中的取出一列值, 这里一次取 3 个, 各广播到一个向量寄存器中; 从输入矩阵中每次读取一个向量, 和前面广播的 3 个 kernel 值向量做乘法, 并累加到输出矩阵当前的寄存器分块中; 再从下一行同样的偏移位置读一个向量, 做相同的操作, 直到第 $8+3-1=10$ 行为止。图 1-6 展示了计算到 kernel 第二列的值的乘累加位置, 其中输入矩阵里深灰色的元素表示 3×3 的 kernel 需要额外读入的 2 行 2 列的数据 ($3-1=2$)。

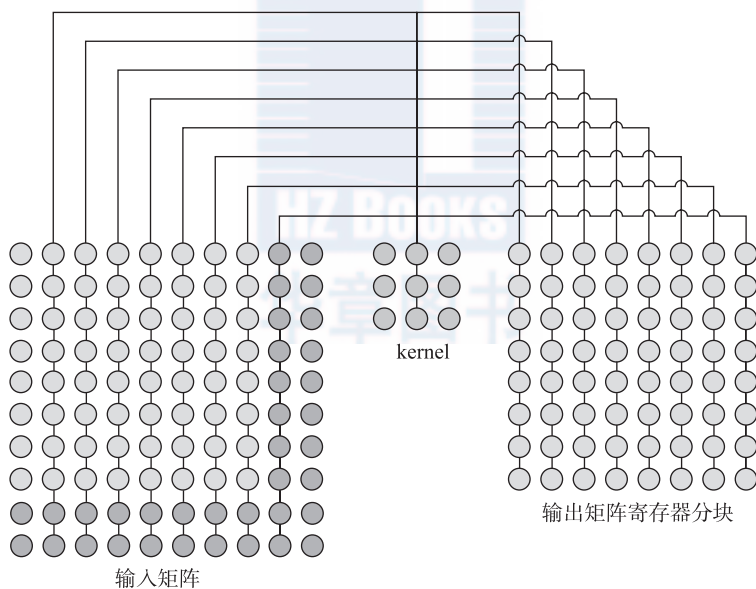


图 1-6 单通道二维卷积的向量寄存器分块算法

一个 8×8 的寄存器分块计算完毕, 就可以写回到输出矩阵, 并可以开始计算右边紧挨着的一个新的寄存器分块。不断推进这个过程, 直到输出矩阵所有位置被计算完毕。这个计算过程对 cache 的重用已经非常好了, 不需要再做专门的局部化处理。对于边界不满足 8×8 的分块, 可以用宏或者代码生成器生成小于 8×8 的寄存器分块计算过程。

下面的代码片段展示了一个 8×8 寄存器分块和 3×3 大小的 kernel 计算一列 kernel 的

过程。

```
ymm13 = _mm256_broadcast_ss(flt + flt_w * 0 + n); // 从kernel中
ymm14 = _mm256_broadcast_ss(flt + flt_w * 1 + n); // 读入一列元素
ymm15 = _mm256_broadcast_ss(flt + flt_w * 2 + n); // 广播到3个向量
ymm10 = _mm256_loadu_ps(src + src_w * 0); // 读取输入数组第0个向量
ymm11 = _mm256_mul_ps(ymm13, ymm10); // 第0个向量只跟kernel最上面元素做乘加
ymm0 = _mm256_add_ps(ymm11, ymm0);
ymm10 = _mm256_loadu_ps(src + src_w * 1); // 读取输入数组第1个向量
ymm11 = _mm256_mul_ps(ymm13, ymm10); // 第1个向量跟kernel前两个元素做乘加
ymm1 = _mm256_add_ps(ymm11, ymm1);
ymm11 = _mm256_mul_ps(ymm14, ymm10);
ymm0 = _mm256_add_ps(ymm11, ymm0);
ymm10 = _mm256_loadu_ps(src + src_w * 2); // 读取输入数组第2个向量
ymm11 = _mm256_mul_ps(ymm13, ymm10); // 从第2个向量开始跟全部3个kernel元素做乘加
ymm2 = _mm256_add_ps(ymm11, ymm2);
ymm11 = _mm256_mul_ps(ymm14, ymm10);
ymm1 = _mm256_add_ps(ymm11, ymm1);
ymm11 = _mm256_mul_ps(ymm15, ymm10);
ymm0 = _mm256_add_ps(ymm11, ymm0);
ymm10 = _mm256_loadu_ps(src + src_w * 3);
ymm11 = _mm256_mul_ps(ymm13, ymm10);
ymm3 = _mm256_add_ps(ymm11, ymm3);
ymm11 = _mm256_mul_ps(ymm14, ymm10);
ymm2 = _mm256_add_ps(ymm11, ymm2);
ymm11 = _mm256_mul_ps(ymm15, ymm10);
ymm1 = _mm256_add_ps(ymm11, ymm1);
ymm10 = _mm256_loadu_ps(src + src_w * 4);
ymm11 = _mm256_mul_ps(ymm13, ymm10);
ymm4 = _mm256_add_ps(ymm11, ymm4);
ymm11 = _mm256_mul_ps(ymm14, ymm10);
ymm3 = _mm256_add_ps(ymm11, ymm3);
ymm11 = _mm256_mul_ps(ymm15, ymm10);
ymm2 = _mm256_add_ps(ymm11, ymm2);
ymm10 = _mm256_loadu_ps(src + src_w * 5);
ymm11 = _mm256_mul_ps(ymm13, ymm10);
ymm5 = _mm256_add_ps(ymm11, ymm5);
ymm11 = _mm256_mul_ps(ymm14, ymm10);
ymm4 = _mm256_add_ps(ymm11, ymm4);
ymm11 = _mm256_mul_ps(ymm15, ymm10);
ymm3 = _mm256_add_ps(ymm11, ymm3);
ymm10 = _mm256_loadu_ps(src + src_w * 6);
ymm11 = _mm256_mul_ps(ymm13, ymm10);
ymm6 = _mm256_add_ps(ymm11, ymm6);
ymm11 = _mm256_mul_ps(ymm14, ymm10);
ymm5 = _mm256_add_ps(ymm11, ymm5);
ymm11 = _mm256_mul_ps(ymm15, ymm10);
ymm4 = _mm256_add_ps(ymm11, ymm4);
ymm10 = _mm256_loadu_ps(src + src_w * 7);
ymm11 = _mm256_mul_ps(ymm13, ymm10);
ymm7 = _mm256_add_ps(ymm11, ymm7);
```

22 ❖ 并行编程方法与优化实践

```
ymm11 = _mm256_mul_ps(ymm14, ymm10);  
ymm6 = _mm256_add_ps(ymm11, ymm6);  
ymm11 = _mm256_mul_ps(ymm15, ymm10);  
ymm5 = _mm256_add_ps(ymm11, ymm5);  
ymm10 = _mm256_loadu_ps(src + src_w * 8);  
ymm11 = _mm256_mul_ps(ymm14, ymm10); // 倒数第2个向量只跟kernel最后两个元素做乘加  
ymm7 = _mm256_add_ps(ymm11, ymm7);  
ymm11 = _mm256_mul_ps(ymm15, ymm10);  
ymm6 = _mm256_add_ps(ymm11, ymm6);  
ymm10 = _mm256_loadu_ps(src + src_w * 9); // 读取输入数组最后一个向量  
ymm11 = _mm256_mul_ps(ymm15, ymm10); // 最后一个向量只跟kernel最后一个元素做乘加  
ymm7 = _mm256_add_ps(ymm11, ymm7);
```

下面来分析一下这个算法的 cache 效率。对于一个 8×8 的分块，总的浮点计算量是 $8 \times 8 \times 3 \times 3 \times 2$ （乘加各算一次浮点操作）=1152；访问 cache 的数量（这里只以读取 float 个数计）是 $(8 \times (8+3-1)+3) \times 3=243$ ；经过测试，支持 AVX 指令的 SNB 架构 CPU 的 cache 带宽是一个周期 8 个 float，同时 SNB 架构一个周期可以发射 8 个单精度浮点乘法和单精度浮点加法，即一个周期内 cache 和浮点计算的吞吐比例是 1:2；但因为大部分 cache 访问并非对齐到向量长度，我们保守估计 cache 带宽会损失一半，即降到 1:4，但仍然高于我们前面计算过的 243:1152。综上，cache 延迟可以被浮点计算掩盖，带宽足够满足浮点计算的峰值性能。

但由于 cache 访问量的减少会导致流水线有部分依赖，这个算法仍不能达到浮点峰值性能，表 1-10 展示了不同尺寸的 kernel 实测的浮点性能（在 Haswell 架构下使用 FMA 指令），以及与浮点峰值的比较。可以看出，kernel 尺寸越大，计算访存比越高，峰值性能也越高，最高可以达到大约 66% 的浮点峰值性能。

表 1-10 二维离散单通道卷积的性能测试

Kernel 大小	3×3	5×5	7×7	9×9	11×11	13×13	15×15	浮点峰值
性能 (GFLOPS)	45.99	67.00	72.53	82.09	77.42	82.89	83.30	126.78
达到峰值百分比	36.3%	52.8%	57.2%	64.7%	61.1%	65.4%	65.7%	100%

1.4 本章小结

本章简略介绍了 X86 处理器支持的 SIMD 指令的 C 封装（内置函数），并且通过由浅入深的 4 个例子展示了 X86 体系结构的常用优化技术。