AVX指令集

一、AVX指令集

CPU依靠指令来计算和控制系统，指令集是指CPU能执行的所有指令的集合，每一类CPU都有其支持的指令集。比如说目前intel和AMD的绝大部分处理器都使用X86指令集，因为它们都源自于X86架构。

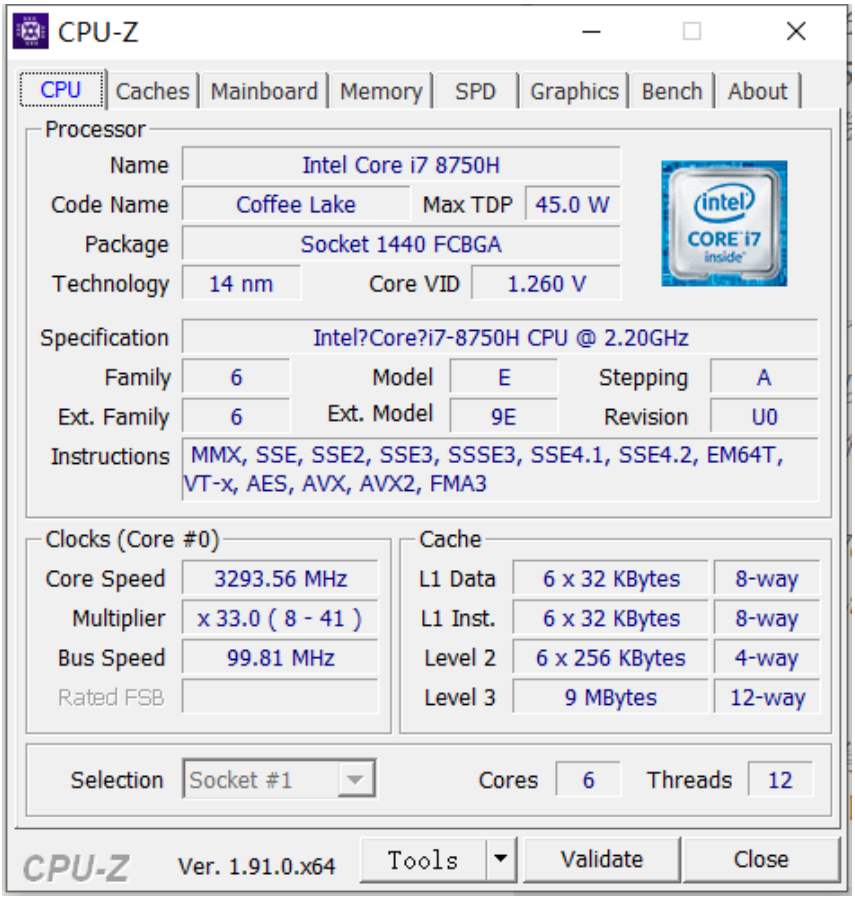
但无论CPU有多快，X86指令也只能一次处理一个数据，但这种单指令单数据（SISD）的指令集效率并不高，因此，为了提高CPU的工作效率，需要增加一些特殊的指令满足时代进步的需求，这些新增的指令就构成了扩展指令集。

一条指令操作多个数据（SIMD）．是CPU基本指令集的扩展．主要用于提供fine grain parallelism，即小碎数据的并行操作．比如说图像处理。Intel的SIMD指令集有MMX、SSE、SSE2、SSE4指令集，AVX指令集也是Intel的一类SIMD指令集。

> *AVX指令集是Sandy Bridge和Larrabee架构下的新指令集。AVX是在之前的128位扩展到和256位的单指令多数据流。而Sandy Bridge的单指令多数据流演算单元扩展到256位的同时数据传输也获得了提升，所以从理论上看CPU内核浮点运算性能提升到了2倍。*
>
> *Intel AVX指令集，在单指令多数据流计算性能增强的同时也沿用了的MMX/SSE指令集。不过和MMX/SSE的不同点在于增强的AVX指令，从指令的格式上就发生了很大的变化。x86(IA-32/Intel 64)架构的基础上增加了prefix(Prefix)，所以实现了新的命令，也使更加复杂的指令得以实现，从而提升了x86 CPU的性能。*
>
> *AVX (Advanced Vector Extensions，高级矢量扩展）指令集借鉴了一些AMD SSE5的设计思路，进行扩展和加强，形成一套新一代的完整SIMD指令集规范。*

指令集需要CPU的支持才能执行，在Linux系统下可以执行以下指令查看CPU支持的指令集，而Windows可以使用CPU-Z软件查看。

```
gcc -march=native -Q --help=target|grep march
或
cat /proc/cpuinfo
```



二、AVX编程

0、编译

.cpp文件需要加上头文件#include <immintrin.h>，在编译时需要添加后缀-mavx -mavx2 以启用AVX指令集：

```
g++ test.cpp -mavx -mavx2 && ./a.out
```

1、数据类型

数据类型	描述
__m128	包含4个float类型数字的向量
__m128d	包含2个double类型数字的向量
__m128i	包含若干个整型数字的向量

数据类型	描述
__m256	包含8个float类型数字的向量
__m256d	包含4个double类型数字的向量
__m256i	包含若干个整型数字的向量

每一种类型，由2个下划线开头，接一个m，然后是vector的位长度。

两种浮点向量被单独列出：__m128，__m256，每个数由4byte的float构成；

d后缀表示双精度浮点数，例如__m128d，__m256d，每个数由8byte的double构成。

而__m128i，__m256i是由整型构成的向量，char，short，int，long均属于整型（以及unsigned以上类型），所以例如__m256i就可以由32个char，或者16个short，或者8个int，又或者4个long构成，这些整型可以是有符号类型也可以是无符号类型。

2、函数名称

_mm<bit_width>_<name>_<data_type>

函数由_mm开头，后面接上向量长度、操作类型、参数类型：

<bit_width> 表明了向量的位长度，对于128位的向量，这个参数为空，对于256位的向量，这个参数为256。

<name>描述了内联函数的算术操作。

<data_type> 标识函数主参数的数据类型，参数含义如下：

1. ps：里面都是float，把32bits当成一个数看
2. pd：里面都是double，把64bits当成一个数看
3. epi8/epi16/epi32/epi64：向量里每个数都是整型，一个整型8bit/16bit/32bit/64bit
4. epu8/epu16/epu32/epu64：向量里每个数都是无符号整型(unsigned)，一个整型8bit/16bit/32bit/64bit
5. m128/m128i/m128d/m256/m256i/m256d：输入值与返回类型不同时会出现，例如__m256i_mm256_setr_m128i(__m128ilo,__m128ihi)，输入两个__m128i向量 ，把他们拼在一起，变成一个__m256i返回 。另外这种结尾只见于load

6. si128/si256：不care向量里到底都是些啥类型，反正128bit/256bit，例如： `__m256i _mm_broadcastsi128_si256 (__m128i a)`

3、基本函数

AVX指令的函数可以通过**Intrinsics Guide**查询，下面进行一点整理：

（1）初始化

用0初始化：

```
__m256d _mm256_setzero_pd (void)

__m256 _mm256_setzero_ps (void)

__m256i _mm256_setzero_si256 (void)
```

用一个标量初始化：

```
__m256i _mm256_set1_epi16 (short a)

__m256i _mm256_set1_epi32 (int a)

__m256i _mm256_set1_epi64x (long long a)

__m256i _mm256_set1_epi8 (char a)

__m256d _mm256_set1_pd (double a)

__m256 _mm256_set1_ps (float a)
```

用多个标量初始化：

```
__m256i _mm256_set_epi16 (short e15, short e14, short e13, short e12, short e11, short e10, short e9, short e8, short e7, short e6, short e5, short e4, short e3, short e2, short e1, short e0)

__m256i _mm256_set_epi32 (int e7, int e6, int e5, int e4, int e3, int e2, int e1, int e0)
```

```
__m256i _mm256_set_epi64x (__int64 e3, __int64 e2, __int64 e1, __int64 e0)

__m256d _mm256_set_pd (double e3, double e2, double e1, double e0)

__m256 _mm256_set_ps (float e7, float e6, float e5, float e4, float e3, float e2, float e1, float e0)

__m256i _mm256_set_epi8 (char e31, char e30, char e29, char e28, char e27, char e26, char e25, chare24, char e23, char e22, char e21, char e20, char e19, char e18, char e17, char e16, char e15, char e14, char e13, char e12, char e11, char e10, char e9, char e8, char e7, char e6, char e5, char e4, char e3, char e2, char e1, char e0)
```

注意形参的顺序，初始位置的值位于最后

用128bit向量初始化：

```
__m256 _mm256_set_m128 (__m128 hi, __m128 lo)

__m256d _mm256_set_m128d (__m128d hi, __m128d lo)

__m256i _mm256_set_m128i (__m128i hi, __m128i lo)
```

（2）数据读取

对齐：

```
__m256d _mm256_load_pd (double const * mem_addr)

__m256 _mm256_load_ps (float const * mem_addr)

__m256i _mm256_load_si256 (__m256i const * mem_addr)
```

未对齐：

```
__m256d _mm256_loadu_pd (double const * mem_addr)

__m256 _mm256_loadu_ps (float const * mem_addr)

__m256i _mm256_loadu_si256 (__m256i const * mem_addr)
```

从两个内存地址分别读取至向量的高128bit和低128bit：

```
__m256 _mm256_loadu2_m128 (float const* hiaddr, float const* loaddr)

__m256d _mm256_loadu2_m128d (double const* hiaddr, double const*
loaddr)

__m256i _mm256_loadu2_m128i (__m128i const* hiaddr, __m128i const*
loaddr)
```

load和loadu分别对应内存地址的是否对齐，一般使用loadu，否则会出现段错误；

加载整型指针时，需要先进行显式类型转换(const __m256i*)x，否则编译器会报错：

```
int *x = new i[10]{0};
__m256i v1 = _mm256_loadu_si256 ( (const __m256i*)x );
```

分别读取/写入的函数都是**未对齐**形式。

（3）数据写回

对齐：

```
void _mm256_store_pd (double * mem_addr, __m256d a)

void _mm256_store_ps (float * mem_addr, __m256 a)

void _mm256_store_si256 (__m256i * mem_addr, __m256i a)
```

未对齐：

```
void _mm256_storeu_pd (double * mem_addr, __m256d a)

void _mm256_storeu_ps (float * mem_addr, __m256 a)

void _mm256_storeu_si256 (__m256i * mem_addr, __m256i a)
```

将256bit的高128bit和低128bit分别写回两个内存地址：

```
void _mm256_storeu2_m128 (float* hiaddr, float* loaddr, __m256 a)

void _mm256_storeu2_m128d (double* hiaddr, double* loaddr, __m256d a)

void _mm256_storeu2_m128i (__m128i* hiaddr, __m128i* loaddr, __m256i a)
```

（4）算术运算

加法：

```
__m256i _mm256_add_epi16 (__m256i a, __m256i b)

__m256i _mm256_add_epi32 (__m256i a, __m256i b)

__m256i _mm256_add_epi64 (__m256i a, __m256i b)

__m256i _mm256_add_epi8 (__m256i a, __m256i b)

__m256d _mm256_add_pd (__m256d a, __m256d b)

__m256 _mm256_add_ps (__m256 a, __m256 b)
```

减法：

```
__m256i _mm256_sub_epi16 (__m256i a, __m256i b)

__m256i _mm256_sub_epi32 (__m256i a, __m256i b)

__m256i _mm256_sub_epi64 (__m256i a, __m256i b)

__m256i _mm256_sub_epi8 (__m256i a, __m256i b)

__m256d _mm256_sub_pd (__m256d a, __m256d b)

__m256 _mm256_sub_ps (__m256 a, __m256 b)
```

整型加减法时可以将函数名设置为adds或subs，这样在加减时会考虑饱和度，如果计算结果发生溢出，结果会被设置为最大值或者最小值。

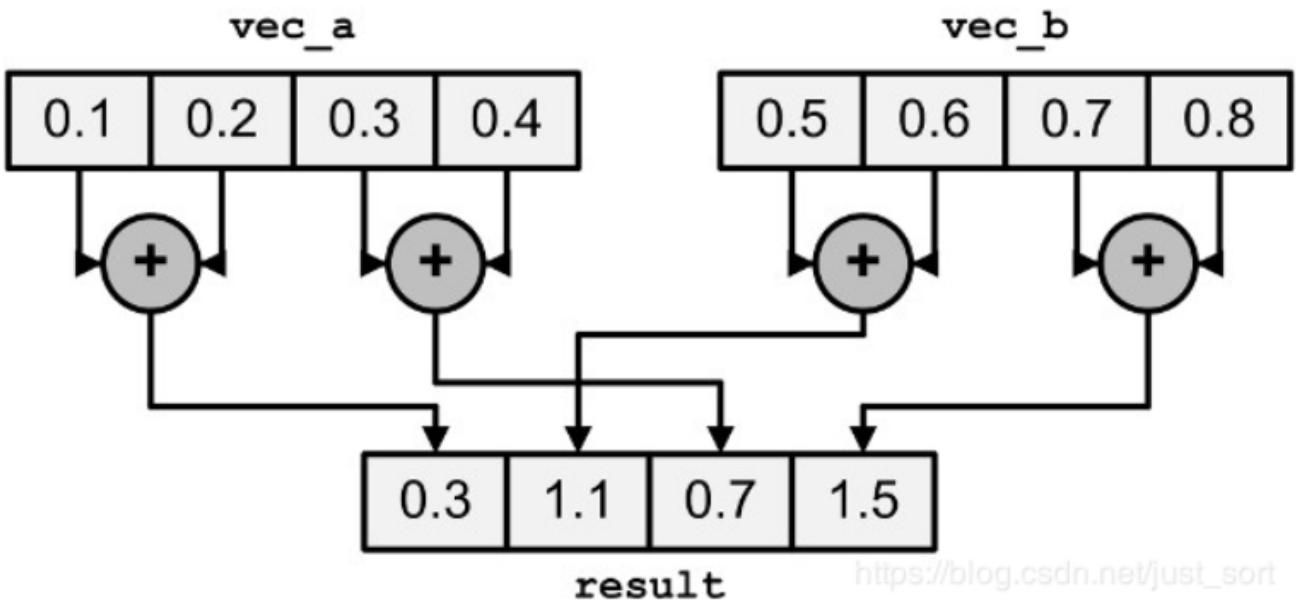
水平加法：

```
__m256i _mm256_hadd_epi16 (__m256i a, __m256i b)

__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)

__m256d _mm256_hadd_pd (__m256d a, __m256d b)

__m256 _mm256_hadd_ps (__m256 a, __m256 b)

__m256i _mm256_madd_epi16 (__m256i a, __m256i b)
```

水平减法：

```
__m256i _mm256_hsub_epi16 (__m256i a, __m256i b)

__m256i _mm256_hsub_epi32 (__m256i a, __m256i b)

__m256d _mm256_hsub_pd (__m256d a, __m256d b)

__m256 _mm256_hsub_ps (__m256 a, __m256 b)

__m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)
```

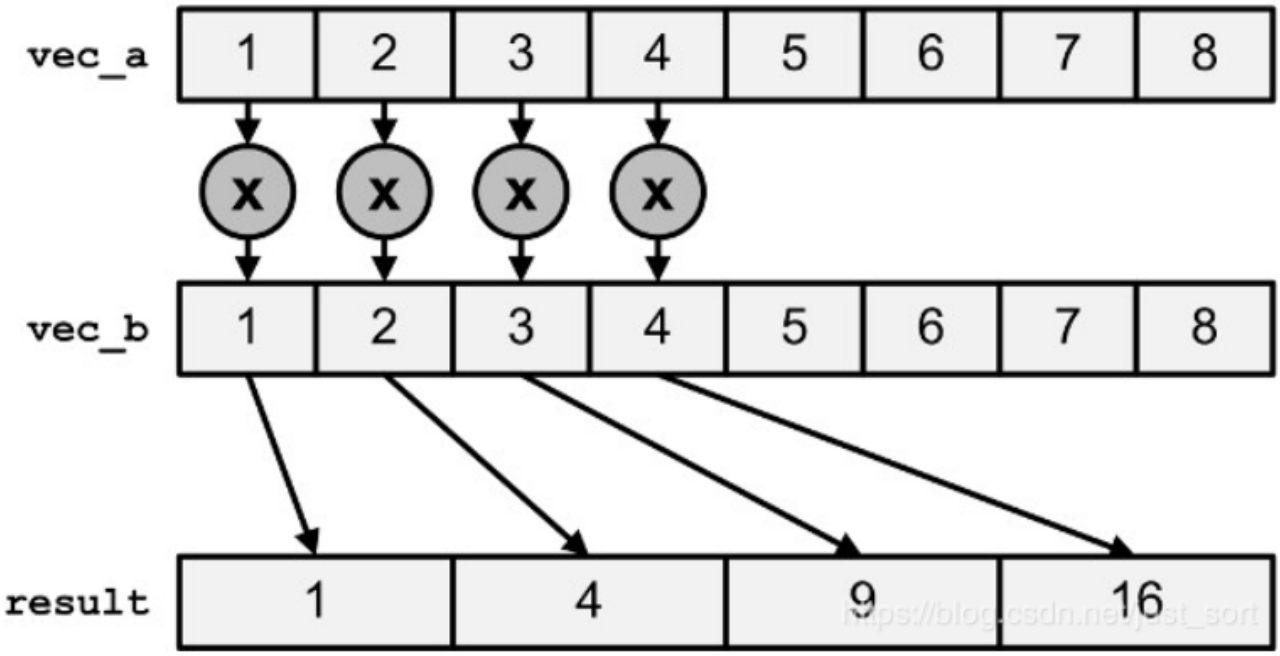
在水平方向上做加减法的意思如下：



乘法：

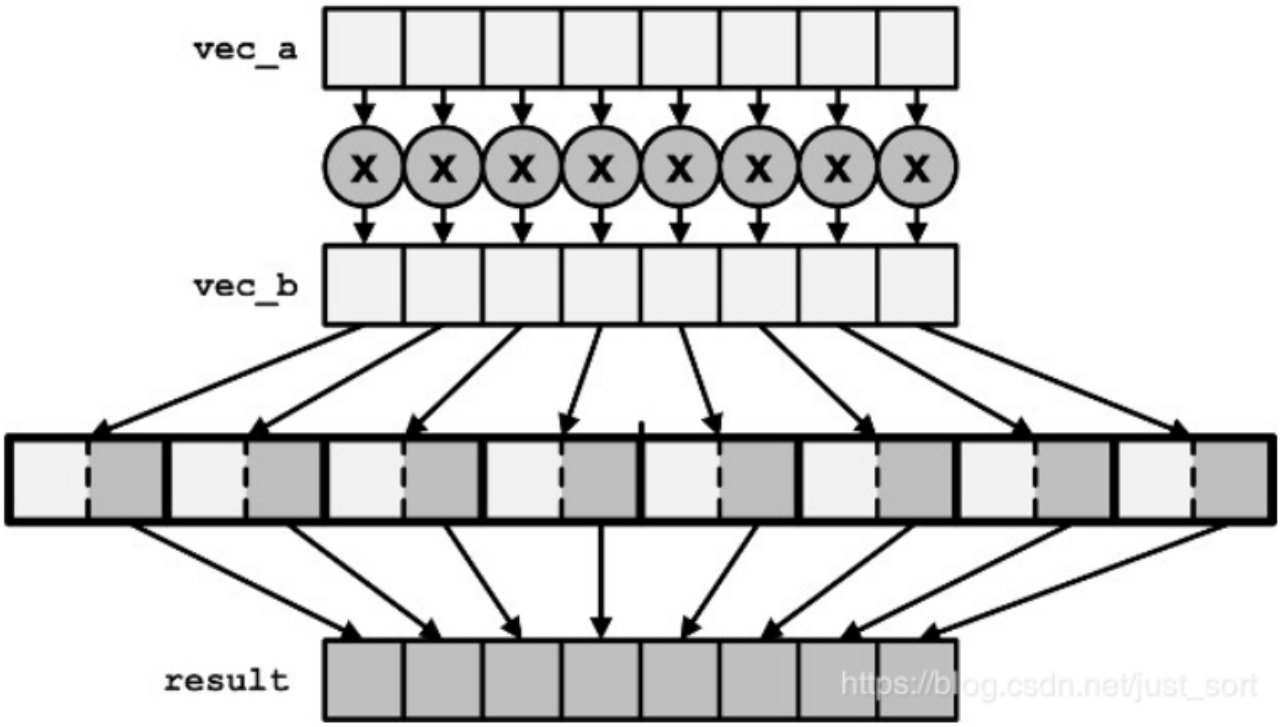
```
__m256i _mm256_mul_epi32 (__m256i a, __m256i b)

__m256i _mm256_mul_epu32 (__m256i a, __m256i b)

__m256d _mm256_mul_pd (__m256d a, __m256d b)

__m256 _mm256_mul_ps (__m256 a, __m256 b)
```

mul对epi32、epu32的运算只针对低4个元素，得到结果为64位整数，如图所示：




```
__m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)

__m256i _mm256_mulhi_epu16 (__m256i a, __m256i b)

__m256i _mm256_mullo_epi16 (__m256i a, __m256i b)

__m256i _mm256_mullo_epi32 (__m256i a, __m256i b)
```

mulhi、mullo的运算会针对所有元素，然后分别取运算结果的高/低位，如图所示：



除法：

```
__m256d _mm256_div_pd (__m256d a, __m256d b)

__m256 _mm256_div_ps (__m256 a, __m256 b)
```

只有浮点数形式。

（5）类型转换

```
__m256 _mm256_castpd_ps (__m256d a)

__m256i _mm256_castpd_si256 (__m256d a)

__m256d _mm256_castpd128_pd256 (__m128d a)

__m128d _mm256_castpd256_pd128 (__m256d a)

__m256d _mm256_castps_pd (__m256 a)

__m256i _mm256_castps_si256 (__m256 a)

__m256 _mm256_castps128_ps256 (__m128 a)

__m128 _mm256_castps256_ps128 (__m256 a)
```

```
__m256i _mm256_castsi128_si256 (__m128i a)

__m256d _mm256_castsi256_pd (__m256i a)

__m256 _mm256_castsi256_ps (__m256i a)

__m128i _mm256_castsi256_si128 (__m256i a)
```

（6）其他运算

下面只列出操作对应的名称，具体类型匹配直接到官网查询。

madd：乘加

abs：绝对值

floor/ceil/round：取整

unpack：间隔读取

insert：插入元素

三、AVX测例

为了测试AVX指令集的加速比，我们写一个简单的程序测试一下，将两个106个元素的数组x、y相加，测试有无使用AVX指令集消耗的时间：

```
#include <iostream>
#include <immintrin.h>
#include <ctime>
#include <cmath>
using namespace std;

int main(){
        int n = 10,m = 10;
        double s=0,p=0;
        int num = pow(10,6);
        int x [num] = {0};
        int y [num];
        for(int i=0;i<num;i++){
                y[i] = 2;
        }
        cout << "serials:"<<endl;
        while(m--){
```

```
                clock_t start, end;
                start = clock();
                int k = 10;
                while(k--)
                        for(int i = 0;i<num;i++){
                                x[i] += y[i];
                        }
                end = clock();
                cout << (double)(end - start) / CLOCKS_PER_SEC << endl;
                s+=(double)(end - start) / CLOCKS_PER_SEC;
        }
        cout << endl << "parallels:"<<endl;
        while(n--){
                clock_t start, end;
                start = clock();
                __m256i v1,v2;
                int k = 10;
                while(k--){
                        int i=0;
                        for(;i<num;i+=8){
                                int *x0 = x+i,*y0 = y+i;
                                v1 = _mm256_loadu_si256((const __m256i*)x0);
                                v2 = _mm256_loadu_si256((const __m256i*)y0);
                                v1 = _mm256_add_epi32(v1,v2);
                                _mm256_storeu_si256 ((__m256i*)x0, v1);
                        }
                        for(;i<num;i++){
                                x[i] += y[i];
                        }
                }
                end = clock();
                cout << (double)(end - start) / CLOCKS_PER_SEC << endl;
                p+=(double)(end - start) / CLOCKS_PER_SEC;
        }
        cout << endl <<"speedup: "<< s/p << endl;
    return 0;
}
```

执行 `g++ t.cpp -mavx -mavx2 && ./a.out` 指令进行编译：

```
max@max-MIIX-510-12ISK:~/parall$ g++ t.cpp
serials:
0.030867
0.03025
0.029938
0.030211
0.030077
0.03006
0.030189
0.030111
0.030134
0.030244

parallels:
0.012288
0.012213
0.012156
0.012124
0.012162
0.012371
0.01233
0.012245
0.013577
0.014124

speedup: 2.4053
```

可以看到，使用串行编程的耗时约为0.030，使用AVX指令集的耗时约为0.012，加速比约为2.4053。