实验： SIMD 指令

目的： 了解并掌握 Intel SIMD 指令的基本用法

Exercise 1: 熟悉 SIMD intrinsics 函数

Intel 提供了大量的 SIMD intrinsics 函数，你需要学会找到自己想要使用的那些函数。

你可以通过查看：Intel® Intrinsics Guide ，勾选某个 checkboxes 你可以看到该指令集下支持的所有操作及相关描述。

这些函数的用法，你还可以参考：

Intrinsics 函数总结 - 百度文库 (baidu.com)

如果是 ARM 处理器，可以从官网或其他来源查看相关资料：Neon - Arm®

回答问题：找出能完成以下操作的 128-位 intrinsics 函数： (one for each):

- Four floating point divisions in single precision (i.e. float)（4 个并行的单精度浮点数除法）**函数：__m128 _mm_div_ps (__m128 a, __m128 b)；**
- Sixteen max operations over unsigned 8-bit integers (i.e.char)（16 个并行求 8 位无符号整数的最大值）**函数：__m128i _mm_max_epu8 (__m128i a, __m128i b)**
- Arithmetic shift right of eight signed 16-bit integers (i.e.short) (8 个并行的 16 位带符号短整数的算术右移）
- **函数：__m128i _mm_srl_epi16 (__m128i a, __m128i count)**

Hint: Things that say “epi” or “pi” deal with integers, and say ”epu” deal with unsigned integers, and those that say “ps” or “pd” deal with single precision and double precision floats.

Exercise 2： 阅读 SIMD 代码

本练习对 SIMD intrinsics 函数是使用进行了示范。

实现双精度浮点数的矩阵乘法：

$$\begin{pmatrix} C[0] & C[2] \\ C[1] & C[3] \end{pmatrix} = \begin{pmatrix} C[0] & C[2] \\ C[1] & C[3] \end{pmatrix} + \begin{pmatrix} A[0] & A[2] \\ A[1] & A[3] \end{pmatrix} \begin{pmatrix} B[0] & B[2] \\ B[1] & B[3] \end{pmatrix}$$

这个操作会产生如下运算：

```
C[0] += A[0]*B[0] + A[2]*B[1];
C[1] += A[1]*B[0] + A[3]*B[1];
C[2] += A[0]*B[2] + A[2]*B[3];
C[3] += A[1]*B[2] + A[3]*B[3];
```

在 sseTest.c 文件中，可以看到矩阵乘法的 SIMD 实现，它使用了以下 intrinsics 函数：

__m128d _mm_loadu_pd(double *p)	returns vector (p[0], p[1])
__m128d _mm_load1_pd(double *p)	returns vector (p[0], p[0])
__m128d _mm_add_pd(__m128d a, __m128d b)	returns vector (a_0+b_0, a_1+b_1)
__m128d _mm_mul_pd(__m128d a, __m128d b)	returns vector (a_0b_0, a_1b_1)
void _mm_storeu_pd(double *p, __m128d a)	stores p[0]=a_0, p[1]=a_1

通过以下命令，编译 sseTest.c 产生 x86 汇编文件：

```
make sseTest.s
```

回答问题： sseTest.s 文件的内容 ，哪些指令是执行 SIMD 操作的？
为了执行并行load指令，.s文件中选择了多个movsd操作。而为了执行并行加法或乘法，.s文件中有专门的并行指令mulpd、addpd进行执行。

Exercise 3:书写SIMD代码
以下代码是原始版本，用于将数组a中的内容累计求和。

```
static int sum_naive(int n, int *a)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

使用以下函数：

__m128i _mm_setzero_si128( )	returns 128-bit zero vector
__m128i _mm_loadu_si128(__m128i *p)	returns 128-bit vector stored at pointer p
__m128i _mm_add_epi32(__m128i a, __m128i b)	returns vector (a_0+b_0, a_1+b_1, a_2+b_2, a_3+b_3)
void _mm_storeu_si128(__m128i *p, __m128i a)	stores 128-bit vector a at pointer p

修改 sum.c 文件中的 sum_vectorized() 函数
编译并运行你的程序：

```
make sum
./sum
```

回答问题：性能是否有改善？ 输出结果是什么？
性能有所改善。输出结果是：
naive: 24.91 microseconds；unrolled: 15.10 microseconds；
vectorized: 13.12 microseconds；vectorized unrolled: ERROR!
Exercise 4: Loop Unrolling循环展开
在sum.c中，我们提供了sum_unrolled()函数的实现：

```
static int sum_unrolled(int n, int *a)
{
    int sum = 0;

    // unrolled loop
    for (int i = 0; i < n / 4 * 4; i += 4)
    {
        sum += a[i+0];
        sum += a[i+1];
        sum += a[i+2];
```

```
            sum += a[i+3];
        }

        // tail case
        for (int i = n / 4 * 4; i < n; i++)
        {
            sum += a[i];
        }

        return sum;
    }
```

将 sum_vectorized()代码拷贝到 sum_vectorized_unrolled() 中并循环展开 4 次，编译并运行代码：

```
make sum
./sum
```

回答问题：性能是否有改善？ 输出结果是什么？将你修改后的源代码 sum.c 和实验文档打包一并提交。

性能有些许改善。一个输出结果是：

```
                naive: 16.16 microseconds
            unrolled: 15.10 microseconds
          vectorized: 12.32 microseconds
 vectorized unrolled: 11.92 microseconds
```

Exercise 5:

我们提供的 makefile 没有采用编译器-o3 优化，你可以试试采用 gcc -o3 编译未向量化的原始文件 sum.c,观察程序是否会被编译器自动向量化，程序性能是否有改善，改善情况如何？你可以加入条件分支语句，例如：if (a[i]>0) sum+= a[i]; 观察自动向量化的效果。

这篇博客介绍了向量化编译供你参考：向量化编译选项 - CSDN

我们提供的 makefile 采用的是-msse4.2 编译选项，该指令集支持的并行计算宽度是 128 位。你可以查看当前计算机所支持的指令集，例如 AVX 或 avx512f，找出相应的 256-位 或 512 位并行度的 intrinsics 函数，修改 sum.c 文件中的 sum_vectorized()函数，将 SIMD 指令的并行度提高，并选用合适的编译选项，例如 -O2 -mavx 或者 -O2 -mavx512f，观察程序性能的改善情况，分析后给出一些你的看法和结论。

编译了sum256.c之后的一次输出是：

```
                naive: 16.30 microseconds
            unrolled: 15.10 microseconds
          vectorized: 11.92 microseconds
 vectorized unrolled: ERROR!
```

和之前的输出对比发现，性能改善是较明显的（运行时间从13ms左右到了12ms左右）。我认为，通过增大并行字节数，使得程序在相同时间能够处理更多数据，这是该现象的原因。在编程中适当使用硬件并行技术能够较好提高程序性能。