Homework 4

环境配置

- CPU：AMD Ryzen 5 6600H with Radeon Graphics
- 支持的SIMD指令集：MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, AVX, AVX2, AES, SHA
- 操作系统：Windows 11
- WSL中的Linux版本：

  Distributor ID: Ubuntu
  Description: Ubuntu 24.04.1 LTS
  Release: 24.04
  Codename: noble

Exercise 1：熟悉SIMD intrinsics函数

- 8个并行的单精度浮点数除法

  `__m256 _mm256_div_ps (__m256 a, __m256 b)`
- 32 个并行求 8 位无符号整数的最大值

  `__m256i _mm256_max_epu8 (__m256i a, __m256i b)`
- 16 个并行的 16 位带符号短整数的算术右移

  `__m256i _mm256_srai_epi16 (__m256i a, int imm8)`

Exercise 2：阅读SIMD代码

- `avxTest.s`的内容

  这是用`avxTest.c`进行汇编而成的在`x86-64`架构下的汇编代码，使用了AVX指令集

  `vxorpd  xmm8, xmm8, xmm8`：清空寄存器

  `vmovapd ymm0, YMMWORD PTR .LC0[rip]`：加载对齐的双精度浮点数

  `vbroadcastsd    ymm0, QWORD PTR .LC2[rip]`：复制64位的浮点数到4个通道

  `vmulpd  ymm15, ymm15, ymm2`：逐通道乘法后赋值

  `vaddpd  ymm1, ymm0, ymm8`：逐通道相加后赋值

  `vmovupd YMMWORD PTR -192[rbp], ymm1`：将双精度浮点数写入内存

Exercise 3：编写SIMD代码

我的代码如下

```
static int sum_vectorized(int n, int *a)
{
    // WRITE YOUR VECTORIZED CODE HERE
    __m256i sum = _mm256_setzero_si256();

    for (int i = 0; i < n / 8 * 8; i += 8)
    {
        __m256i tmp = _mm256_loadu_si256((__m256i*)(a + i));
        sum = _mm256_add_epi32(sum, tmp);
    }

    int S[8] = {0};
    _mm256_storeu_si256((__m256i*)S, sum);
    S[0] = S[0] + S[1] + S[2] + S[3] + S[4] + S[5] + S[6] + S[7];

    for (int i = n / 8 * 8; i < n; i++)
    {
        S[0] += a[i];
    }

    return S[0];
}
```

它没有显著地改善性能，输出结果是做这个计算的时长

可能原因是数据量太小了，初始化时间占主导

Exercise 4：Loop Unrollin循环展开

代码如下

```
static int sum_vectorized_unrolled(int n, int *a)
{
    // UNROLL YOUR VECTORIZED CODE HERE
        __m256i sum = _mm256_setzero_si256();

    for (int i = 0; i < n / 32 * 32; i += 32)
    {
        __m256i tmp = _mm256_loadu_si256((__m256i*)(a + i));
        sum = _mm256_add_epi32(sum, tmp);
        tmp = _mm256_loadu_si256((__m256i*)(a + i + 8));
        sum = _mm256_add_epi32(sum, tmp);
        tmp = _mm256_loadu_si256((__m256i*)(a + i + 16));
        sum = _mm256_add_epi32(sum, tmp);
        tmp = _mm256_loadu_si256((__m256i*)(a + i + 24));
        sum = _mm256_add_epi32(sum, tmp);
    }

    int S[8] = {0};
    _mm256_store_si256((__m256i*)S, sum);
    for (int i = 1; i < 8; i++)
```

```
    {
        S[0] += S[i];
    }

    for (int i = n / 32 * 32; i < n; i++)
    {
        S[0] += a[i];
    }

    return S[0];
}
```

性能相比`sum_vectorized()`提升了一倍，最终输出结果如下

> naive: 1.84 microseconds
> unrolled: 1.81 microseconds
> vectorized: 1.75 microseconds
> vectorized unrolled: 1.04 microseconds

Exercise 5：了解编译器提供的向量化优化

在看了一下`Makefile`我发现，`FLAGS = -std=gnu99 -O3 -DNDEBUG -g0 -mavx2`，说明原先已经开启了`-O3`优化，在删除`-O3`后再次测试，得到的结果如下：

> naive: 24.11 microseconds
> unrolled: 13.89 microseconds
> vectorized: 16.35 microseconds
> vectorized unrolled: 16.25 microseconds

这才是没有优化的运行时间，对比可以发现`-O3`优化了很多。

在删除后的`Makefile`中，使用了`make sum.s`后产生的`sum.s`文件，前两个函数没用使用到AVX指令集的指令

在原来的`Makefile`中，同样创建`sum.s`，观察到

```
;展示部分用到了AVX指令集的代码
sum_naive:
.LFB6644:
    vpxor   xmm1, xmm1, xmm1        ;创建全0的256位寄存器
.L4:                                ;几乎全是AVX指令
    vpaddd  ymm1, ymm1, YMMWORD PTR [rax]
    add rax, 32
    cmp rax, rcx
    jne .L4
    vmovdqa xmm0, xmm1
    vextracti128    xmm1, ymm1, 0x1
    mov ecx, edx
    vpaddd  xmm0, xmm0, xmm1
    and ecx, -8
    vpsrldq xmm1, xmm0, 8
    vpaddd  xmm0, xmm0, xmm1
    vpsrldq xmm1, xmm0, 4
    vpaddd  xmm0, xmm0, xmm1
```

```
    vmovd   eax, xmm0
    test    dl, 7
    je  .L12
    vzeroupper
```

说明编译器自动产生了向量化优化，程序的性能改善见上二表格