arch hw4

郑航 520021911347

Exercise 1: 熟悉 SIMD intrinsics 函数

- __m128 _mm_div_ps (__m128 a, __m128 b)
- __m128i _mm_max_epu8 (__m128i a, __m128i b)
- __m128i _mm_srli_epi16 (__m128i a, int imm8)

Exercise 2： 阅读 SIMD 代码

SIMD指令：movapd，movsd，addpd，mulpd，unpckhpd

Exercise 3: 书写 SIMD 代码

编写代码如下：

```
static int sum_vectorized(int n, int *a)
{
    // WRITE YOUR VECTORIZED CODE HERE
    __m128i res = _mm_setzero_si128();
    for (int i = 0; i < n / 4 * 4; i += 4)
    {
        res = _mm_add_epi32(res, _mm_loadu_si128(a + i));
    }
    int RES[4];
    _mm_storeu_si128(RES, res);

    int sum = RES[0] + RES[1] + RES[2] + RES[3];

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

    return sum;
}
```

截止目前为止的运行结果如下（暂时只实现了vectorized）：

```
zh@ubuntu:~/homework/lab4$ ./sum
               naive: 5.73 microseconds
            unrolled: 4.47 microseconds
          vectorized: 3.75 microseconds
```

由运行结果可知，向量化确实可以提升程序的性能，从5.73ms提升到了3.75ms，得到了接近1/2的性能提升（仅使用一次测试计算所得的结果）

计算：(1/3.75 - 1/5.73) / (1/5.73) * 100%= 52.8%

Exercise 4: Loop Unrolling 循环展开

编写代码如下：

```
static int sum_vectorized_unrolled(int n, int *a)
{
    // unrolled loop
    __m128i sum_vec = _mm_setzero_si128();
    for (int i = 0; i < n / 16 * 16; i += 16)
    {
        __m128i tmp1 = _mm_add_epi32(_mm_loadu_si128(a + i),
_mm_loadu_si128(a + i + 4));
        __m128i tmp2 = _mm_add_epi32(_mm_loadu_si128(a + i + 8),
_mm_loadu_si128(a + i + 12));
        sum_vec = _mm_add_epi32(sum_vec, tmp1);
        sum_vec = _mm_add_epi32(sum_vec, tmp2);
    }
    int RES[4];
    _mm_storeu_si128(RES, sum_vec);
    int sum = RES[0] + RES[1] + RES[2] + RES[3];

    // tail case
    for (int i = n / 16 * 16; i < n; i++)
    {
        sum += a[i];
    }

    return sum;
}

```

运行结果如下：

```
zh@ubuntu:~/homework/lab4$ ./sum
              naive: 5.45 microseconds
           unrolled: 4.47 microseconds
         vectorized: 2.34 microseconds
vectorized unrolled: 1.54 microseconds
```

可以看到，在循环展开的方法中使用向量化运算的方式，程序性能会得到进一步的改善

Exercise 5:

- 使用-O3对原始sum.c（在此命名为sum_naive.c）进行编译，并直接运行以测试其运算时间，与未经-O3优化的程序对比如下：通过naive和unrolled两项可以看出，gcc的-O3优化确实能提升程序性能

```
zh@ubuntu:~/homework/lab4$ ./sum_naive
               naive: 2.88 microseconds
             unrolled: 3.43 microseconds
           vectorized: ERROR!
 vectorized unrolled: ERROR!
zh@ubuntu:~/homework/lab4$ ./sum
               naive: 8.91 microseconds
             unrolled: 6.29 microseconds
           vectorized: 2.92 microseconds
 vectorized unrolled: 1.70 microseconds
```

- 在sum_naive.c中的每个赋值语句前加上if语句，使其不能被自动向量化，并将文件另存为sum_naive_with_branch.c，编译运行后结果如下：

```
zh@ubuntu:~/homework/lab4$ ./sum_naive
               naive: 2.70 microseconds
             unrolled: 3.43 microseconds
           vectorized: ERROR!
 vectorized unrolled: ERROR!
zh@ubuntu:~/homework/lab4$ ./sum_naive_with_branch
               naive: 4.73 microseconds
             unrolled: 13.21 microseconds
           vectorized: ERROR!
 vectorized unrolled: ERROR!
```

- 通过对比可以看到，加上语句后就无法自动进行向量化了，程序的性能也因此得到下降，尤其是unrolled的情况，其中本来四个赋值语句都可以并行执行，如今却都只能串行执行，因此性能受到严重影响，下降到接近原来的四分之一

- 查了一下，我的机子只支持256位的AVX指令集，并不支持512位，所以下面给出的是256位并行宽度的优化

  其实256位的intrinsics函数和128位的非常类似，对整个程序的设计也差别不大，只需要修改一些函数名以及单个循环的数据处理量即可，将256位并行处理的程序另存为256_sum.c（代码较长且与上面展示代码重合度较高，故在此不加展示）

  按-O2 -mavx2 编译运行后，结果如下：

```
zh@ubuntu:~/homework/lab4$ ./256_sum
               naive: 7.94 microseconds
             unrolled: 6.57 microseconds
           vectorized: 2.53 microseconds
 vectorized unrolled: 1.96 microseconds
zh@ubuntu:~/homework/lab4$ ./sum
               naive: 8.57 microseconds
             unrolled: 7.08 microseconds
           vectorized: 3.74 microseconds
 vectorized unrolled: 2.49 microseconds
```

- 可以看到，采用256位的avx指令集，确实可以进一步提高程序的性能。理论上来说，程序性能应该提高为原来的两倍左右，但是实际比较结果可知，程序性能只提升了10%左右，推测可能是由于数据量较小，使得提高并行化程度以及尾部数据处理的开销相比整体运行时间无法忽略所致

本作业所有源代码及可执行文件详见code文件夹