Exercise1:

（1）4个并行的单精度浮点数除法：\_\_m128 \_mm\_div\_ps (\_\_m128 a, \_\_m128 b)

（2）6个并行求 8 位无符号整数的最大值：

\_\_m128i \_mm\_mask\_max\_epu8 (\_\_m128i src, \_\_mmask16 k, \_\_m128i a, \_\_m128i b) 1

（3）8个并行的 16 位带符号短整数的算术右移：

\_\_m128i \_mm\_mask\_shrdi\_epi16 (\_\_m128i src, \_\_mmask8 k, \_\_m128i a, \_\_m128i b, int imm8)

Exercise2:

观察sseTest.s函数中的各类x86汇编指令，经过查阅资料，发现Movapd、mulpd addpd、Movsd、movaps这些x86instruction为SIMD指令。

Exercise3 & Exerciese4：

（1）若采取intrinsic函数来修改Exercise3原代码，但是仅仅使用\_\_m128i的前32bit进行运算（即使用SIMD指令，但仅仅处理了单个数据），函数代码与试验结果如下：

static int sum\_vectorized(int n, int \*a) //only use first 32-bit integer in a \_\_128i

{

// WRITE YOUR VECTORIZED CODE HERE

int A[4] = {0, 0, 0, 0};

\_\_m128i sum = \_mm\_setzero\_si128();

//\_\_m128i sum = \_mm\_loadu\_si128(A);

//loop part

for (int i = 0; i < n; i += 1)

{

\_\_m128i tmp = \_mm\_loadu\_si128(a + i);

sum = \_mm\_add\_epi32(sum, tmp);

}

\_mm\_storeu\_si128(A, sum);

return A[0];

}

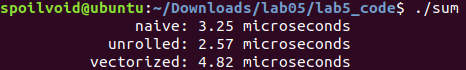


图1

可以看到，由于一个SIMD指令函数实际上要处理128bit，但我们仅仅使用了前32bit，没有并行，同样的操作处理数据增多，所以在原初naïve函数的基础上，运行时间反而增长，性能没有改善。输出结果见图1。

（2）若采取intrinsic函数来修改Exercise3原代码，且使用\_\_m128i的所有128bit进行运算（即使用SIMD指令，每次处理4个数据），函数代码与试验结果如下：

static int sum\_vectorized(int n, int \*a)

{

// WRITE YOUR VECTORIZED CODE HERE

int A[4] = {0, 0, 0, 0};

\_\_m128i sum = \_mm\_setzero\_si128();

//\_\_m128i sum = \_mm\_loadu\_si128(A);

//loop part

for (int i = 0; i < n / 4 \* 4; i += 4)

{

\_\_m128i tmp = \_mm\_loadu\_si128(a + i);

sum = \_mm\_add\_epi32(sum, tmp);

}

\_mm\_storeu\_si128(A, sum);

int res = A[0] + A[1] + A[2] + A[3];

//tail part

for (int i = n / 4 \* 4; i < n; i++)

res += a[i];

return res;

}

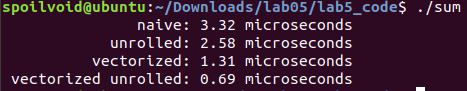


图2

我们使用SIMD指令函数将naïve函数进行了并行化，一个循环中intrinsic函数处理4个数据，将运行时间减至原来naïve函数，性能有所改善。运行结果如图2。

（3）参照Exercise4的源代码对已经SIMD代码化的sum\_vectorized函数（如2）进行4x4展开，增加并行性，试验结果即上述图2，函数代码如下：

static int sum\_vectorized\_unrolled(int n, int \*a)

{

// UNROLL YOUR VECTORIZED CODE HERE

int A[4] = {0, 0, 0, 0};

\_\_m128i sum1 = \_mm\_setzero\_si128();

\_\_m128i sum2 = \_mm\_setzero\_si128();

\_\_m128i sum3 = \_mm\_setzero\_si128();

\_\_m128i sum4 = \_mm\_setzero\_si128();

//\_\_m128i sum = \_mm\_loadu\_si128(A);

//loop part

for (int i = 0; i < n / 16 \* 16; i += 16)

{

\_\_m128i tmp1 = \_mm\_loadu\_si128(a + i);

\_\_m128i tmp2 = \_mm\_loadu\_si128(a + i + 4);

\_\_m128i tmp3= \_mm\_loadu\_si128(a + i + 8);

\_\_m128i tmp4 = \_mm\_loadu\_si128(a + i + 12);

sum1 = \_mm\_add\_epi32(sum1, tmp1);

sum2 = \_mm\_add\_epi32(sum2, tmp2);

sum3 = \_mm\_add\_epi32(sum3, tmp3);

sum4 = \_mm\_add\_epi32(sum4, tmp4);

}

sum1 = \_mm\_add\_epi32(sum1, sum2);

sum1 = \_mm\_add\_epi32(sum1, sum3);

sum1 = \_mm\_add\_epi32(sum1, sum4);

\_mm\_storeu\_si128(A, sum1);

int res = A[0] + A[1] + A[2] + A[3];

//tail part

for (int i = n / 16 \* 16; i < n; i++)

res += a[i];

return res;

}

由运行结果可知，我们对sum\_vectorized函数进行进一步展开后，相较unrolled和sum\_vectorized函数性能均有较大提升，并行化程度进一步增加，性能有所改善。运行结果见图2。