

# 浙江大学实验报告

课程名称：汇编与接口

实验项目名称：**SIMD指令的探索与实验分析**

电子邮件地址：

实验日期：2021.10.28

## 背景说明

SIMD全称Single Instruction Multiple Data，单指令多数据流，能够复制多个操作数，并它们打包在大型寄存器的一组指令集。使用SIMD指令，可以同时多个操作数上执行同一指令，进而加快运行速度。举个例子，一般SISD（Single Instruction Single Data）指令，如多个加法指令，会依次访问内存、取得操作数并相加；而如果使用SIMD指令，程序则会为这几条加法指令同时访问内存取得操作数，再同时运行加法运算，加快了运行速度。

SIMD的首次使用出现在1966年。SIMD是1980年代早期矢量超级计算机的基础。事实上，从SIMD的原理不难看出矢量化的影子。而现代SIMD计算机的第一个时代以大规模并行处理风格的超级计算机为特征。

到了当下，当前的SIMD指令被广泛用于计算机市场，而不仅限于超级计算机市场。SIMD的一大应用是游戏、音频、视频的处理，这种多媒体的处理尤其适合并行计算。1996年，英特尔在x86架构内部署MMX扩展，是为第一个广泛部署的台式机SIMD。此后，英特尔又逐步开发出了SSE、AVX等SIMD指令集。这三种指令集的简介如下：

- MMX是最早的SIMD指令集，其使用过程占用八个浮点数寄存器；数据宽度需要对齐
- SSE在MMX的基础上发展而来，英特尔为其特地设置了单独的128位XMM寄存器，使得其执行不再与浮点运算相矛盾；引入了SIMD浮点数据类型
- AVX在SSE的基础上发展而来，将128位的XMM寄存器扩展为了256位的YMM寄存器

## 探索过程

此前虽然学习过汇编，但是我并没有学习过SIMD相关的内容。考虑到不熟悉指令，以及不熟悉SIMD引入的XMM、YMM寄存器，我决定采用编写C程序并利用汇编进行探索。为了体现SIMD的优势，我决定引入对比，探索的要点如下：

1. 计时比较SIMD与SISD的速度差别，在Linux系统下进行探索可以得到更准确的计时结果
2. 编写C程序进行运算，编译后通过查看汇编代码；为了实现对比的效果，代码中最好不要有指定SSE命令的代码

首先我查看我的设备是否支持SIMD，在控制台输入 `cat /proc/cpuinfo` 命令，可以看到设备支持mmx、sse、sse2等SIMD指令集。

然后，查看gcc是否支持SIMD，在控制台输入 `gcc -march=native -c -Q --help=target` 命令，可以看到gcc同样支持指定mmmx、mavx、msse等参数。

确认以上两步后，开始编写C语言代码。这里为了更简单地得到使用SIMD指令的效果，我没有按照数学的定义去写矩阵的乘法，而是直接让两个矩阵同位置的元素相乘并加到结果上。同时，为了方便汇编查看是否使用了SIMD指令，我将矩阵乘法单独写成一个函数。被注释掉的部分是指定使用SSE命令的代码，因为不方便拿来对比，我将其去掉。总的来说：

- 代码内：随机产生矩阵，并进行“累乘”
- 代码外：改变编译的参数，对比使用SIMD命令和不使用SIMD命令的“累乘”运行时间并分析

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

const int N = 128;
int times = 2048, lower = -1e3, upper = 1e3;

void multiply(double m1[][N], double m2[][N], double m3[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            m3[i][j] += m1[i][j] * m2[i][j];
        }
    }

    // return the result to m1
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            m1[i][j] = m3[i][j];
        }
    }
}

int main()
{
    double m1[N][N], m2[N][N], m3[N][N];

    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            m1[i][j] = lower + 1.0 * (rand() % RAND_MAX / RAND_MAX * (upper -
lower));
        }
    }
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            m2[i][j] = lower + 1.0 * (rand() % RAND_MAX / RAND_MAX * (upper -
lower));
        }
    }

    clock_t start, end;
```

```

start = clock();
for (int i = 0; i < times; i++)
{
    multiply(m1, m2, m3);
}
end = clock();
printf("time used: %f\n", (double)(end - start) / CLOCKS_PER_SEC);

// float op1[4] = { 1.0, 2.0, 3.0, 4.0 };
// float op2[4] = { 5.0, 6.0, 7.0, 8.0 };
// float result[4];
// __m128 a, b, c;

// Load
// a = _mm_loadu_ps(op1);
// b = _mm_loadu_ps(op2);

// Calculate
// c = _mm_add_ps(a, b);

// Store
// _mm_storeu_ps(result, c);

return 0;
}

```

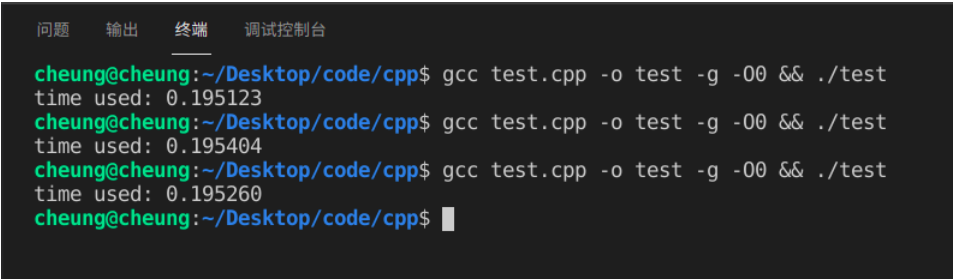
## 效果分析

在实验前，我的第一想法是对比不使用SIMD、使用MMX、使用SSE的计算时间。但是实验过程中发现简单地指定 `gcc -mmmx` 或 `gcc -msse` 似乎并不能编译出SIMD指令。在搜索解决方案无果后，我发现指定不同的优化等级可以让gcc编译出SIMD指令，优化等级为3时可以让gcc编译出SIMD指令（在这个优化等级，gcc会加入很多向量化算法）。

必须说明的是，在对比性能差异时，指定优化等级的方法不如指定SIMD指令集的方法来得准确。但是gcc参数众多，目前我未能找到让gcc仅加入SIMD优化而不做其他优化的参数配置，因此这里采用指定优化等级的方法进行分析。事实上，`gcc -O3` 和 `gcc -O2` 的主要差别就是在 `-O2` 的基础上加入向量化算法，因此我认为这样进行的分析仍然有一定的价值，能说明性能差异。

## 优化等级0

指定优化等级为 `-O0`，可以看到执行时间约为0.195秒。



```

问题  输出  终端  调试控制台
cheung@cheung:~/Desktop/code/cpp$ gcc test.cpp -o test -g -O0 && ./test
time used: 0.195123
cheung@cheung:~/Desktop/code/cpp$ gcc test.cpp -o test -g -O0 && ./test
time used: 0.195404
cheung@cheung:~/Desktop/code/cpp$ gcc test.cpp -o test -g -O0 && ./test
time used: 0.195260
cheung@cheung:~/Desktop/code/cpp$

```

在这个等级下，查看函数 `multiply` 的汇编代码如下：

```

0x000000000000011a9 <+0>:    endbr64
0x000000000000011ad <+4>:    push    %rbp
0x000000000000011ae <+5>:    mov     %rsp,%rbp

```

```

0x00000000000011b1 <+8>:      mov     %rdi,-0x18(%rbp)
0x00000000000011b5 <+12>:     mov     %rsi,-0x20(%rbp)
0x00000000000011b9 <+16>:     mov     %rdx,-0x28(%rbp)
0x00000000000011bd <+20>:     movl    $0x0,-0x10(%rbp)
0x00000000000011c4 <+27>:     cmpl    $0x7f,-0x10(%rbp)
0x00000000000011c8 <+31>:     jg      0x126d <multiply(double (*) [128], double
(*) [128], double (*) [128])+196>
0x00000000000011ce <+37>:     movl    $0x0,-0xc(%rbp)
0x00000000000011d5 <+44>:     cmpl    $0x7f,-0xc(%rbp)
0x00000000000011d9 <+48>:     jg      0x1264 <multiply(double (*) [128], double
(*) [128], double (*) [128])+187>
0x00000000000011df <+54>:     mov     -0x10(%rbp),%eax
0x00000000000011e2 <+57>:     cltq
0x00000000000011e4 <+59>:     shl     $0xa,%rax
0x00000000000011e8 <+63>:     mov     %rax,%rdx
0x00000000000011eb <+66>:     mov     -0x28(%rbp),%rax
0x00000000000011ef <+70>:     add     %rax,%rdx
0x00000000000011f2 <+73>:     mov     -0xc(%rbp),%eax
0x00000000000011f5 <+76>:     cltq
0x00000000000011f7 <+78>:     movsd   (%rdx,%rax,8),%xmm1
0x00000000000011fc <+83>:     mov     -0x10(%rbp),%eax
0x00000000000011ff <+86>:     cltq
0x0000000000001201 <+88>:     shl     $0xa,%rax
0x0000000000001205 <+92>:     mov     %rax,%rdx
0x0000000000001208 <+95>:     mov     -0x18(%rbp),%rax
0x000000000000120c <+99>:     add     %rax,%rdx
0x000000000000120f <+102>:    mov     -0xc(%rbp),%eax
0x0000000000001212 <+105>:    cltq
0x0000000000001214 <+107>:    movsd   (%rdx,%rax,8),%xmm2
0x0000000000001219 <+112>:    mov     -0x10(%rbp),%eax
0x000000000000121c <+115>:    cltq
0x000000000000121e <+117>:    shl     $0xa,%rax
0x0000000000001222 <+121>:    mov     %rax,%rdx
0x0000000000001225 <+124>:    mov     -0x20(%rbp),%rax
0x0000000000001229 <+128>:    add     %rax,%rdx
0x000000000000122c <+131>:    mov     -0xc(%rbp),%eax
0x000000000000122f <+134>:    cltq
0x0000000000001231 <+136>:    movsd   (%rdx,%rax,8),%xmm0
0x0000000000001236 <+141>:    mulsd   %xmm2,%xmm0
0x000000000000123a <+145>:    mov     -0x10(%rbp),%eax
0x000000000000123d <+148>:    cltq
0x000000000000123f <+150>:    shl     $0xa,%rax
0x0000000000001243 <+154>:    mov     %rax,%rdx
0x0000000000001246 <+157>:    mov     -0x28(%rbp),%rax
0x000000000000124a <+161>:    add     %rax,%rdx
0x000000000000124d <+164>:    addsd   %xmm1,%xmm0
0x0000000000001251 <+168>:    mov     -0xc(%rbp),%eax
0x0000000000001254 <+171>:    cltq
0x0000000000001256 <+173>:    movsd   %xmm0, (%rdx,%rax,8)
0x000000000000125b <+178>:    addl    $0x1,-0xc(%rbp)
0x000000000000125f <+182>:    jmpq    0x11d5 <multiply(double (*) [128], double
(*) [128], double (*) [128])+44>
0x0000000000001264 <+187>:    addl    $0x1,-0x10(%rbp)
0x0000000000001268 <+191>:    jmpq    0x11c4 <multiply(double (*) [128], double
(*) [128], double (*) [128])+27>
0x000000000000126d <+196>:    movl    $0x0,-0x8(%rbp)
0x0000000000001274 <+203>:    cmpl    $0x7f,-0x8(%rbp)

```

```

0x0000000000001278 <+207>:  jg      0x12ce <multiply(double (*) [128], double
(*) [128], double (*) [128])+293>
0x000000000000127a <+209>:  movl    $0x0,-0x4(%rbp)
0x0000000000001281 <+216>:  cmpl    $0x7f,-0x4(%rbp)
0x0000000000001285 <+220>:  jg      0x12c8 <multiply(double (*) [128], double
(*) [128], double (*) [128])+287>
0x0000000000001287 <+222>:  mov     -0x8(%rbp),%eax
0x000000000000128a <+225>:  cltq
0x000000000000128c <+227>:  shl     $0xa,%rax
0x0000000000001290 <+231>:  mov     %rax,%rdx
0x0000000000001293 <+234>:  mov     -0x28(%rbp),%rax
0x0000000000001297 <+238>:  lea     (%rdx,%rax,1),%rcx
0x000000000000129b <+242>:  mov     -0x8(%rbp),%eax
0x000000000000129e <+245>:  cltq
0x00000000000012a0 <+247>:  shl     $0xa,%rax
0x00000000000012a4 <+251>:  mov     %rax,%rdx
0x00000000000012a7 <+254>:  mov     -0x18(%rbp),%rax
0x00000000000012ab <+258>:  add     %rax,%rdx
0x00000000000012ae <+261>:  mov     -0x4(%rbp),%eax
0x00000000000012b1 <+264>:  cltq
0x00000000000012b3 <+266>:  movsd   (%rcx,%rax,8),%xmm0
0x00000000000012b8 <+271>:  mov     -0x4(%rbp),%eax
0x00000000000012bb <+274>:  cltq
0x00000000000012bd <+276>:  movsd   %xmm0, (%rdx,%rax,8)
0x00000000000012c2 <+281>:  addl    $0x1,-0x4(%rbp)
0x00000000000012c6 <+285>:  jmp     0x1281 <multiply(double (*) [128], double
(*) [128], double (*) [128])+216>
0x00000000000012c8 <+287>:  addl    $0x1,-0x8(%rbp)
0x00000000000012cc <+291>:  jmp     0x1274 <multiply(double (*) [128], double
(*) [128], double (*) [128])+203>
0x00000000000012ce <+293>:  nop
0x00000000000012cf <+294>:  pop     %rbp
0x00000000000012d0 <+295>:  retq

```

需要说明的是，后缀为 `sd` 的指令在我看来仍然是SISD指令。`s` 代表标量方法，只对XMM寄存器中最低的一个浮点数进行运算；`d` 则代表着浮点数的宽度。查看汇编指令可以看出，这个等级下的汇编代码没有使用SIMD指令，并且编译出的汇编指令十分冗长。

## 优化等级2

指定优化等级为 `-O2`，可以看到执行时间约为0.04秒。

```

问题  输出  终端  调试控制台
cheung@cheung:~/Desktop/code/cpp$ gcc test.cpp -o test -g -O2 && ./test
time used: 0.041147
cheung@cheung:~/Desktop/code/cpp$ gcc test.cpp -o test -g -O2 && ./test
time used: 0.040775
cheung@cheung:~/Desktop/code/cpp$ gcc test.cpp -o test -g -O2 && ./test
time used: 0.040621
cheung@cheung:~/Desktop/code/cpp$ █

```

在这个等级下，查看函数 `multiply` 的汇编代码如下：

```

0x0000000000001340 <+0>:  endbr64
0x0000000000001344 <+4>:  push    %rbx
0x0000000000001345 <+5>:  mov     %rdi,%r11

```

```

0x0000000000001348 <+8>:      mov     %rsi,%rbx
0x000000000000134b <+11>:     mov     %rdx,%r10
0x000000000000134e <+14>:     xor     %r9d,%r9d
0x0000000000001351 <+17>:     mov     %r10,%rcx
0x0000000000001354 <+20>:     lea     (%r10,%r9,1),%rdx
0x0000000000001358 <+24>:     mov     %r11,%rsi
0x000000000000135b <+27>:     xor     %eax,%eax
0x000000000000135d <+29>:     lea     (%r9,%r11,1),%r8
0x0000000000001361 <+33>:     lea     (%rbx,%r9,1),%rdi
0x0000000000001365 <+37>:     nopl    (%rax)
0x0000000000001368 <+40>:     movsd   (%r8,%rax,1),%xmm0
0x000000000000136e <+46>:     mulsd   (%rdi,%rax,1),%xmm0
0x0000000000001373 <+51>:     addsd   (%rdx,%rax,1),%xmm0
0x0000000000001378 <+56>:     movsd   %xmm0, (%rdx,%rax,1)
0x000000000000137d <+61>:     add     $0x8,%rax
0x0000000000001381 <+65>:     cmp     $0x400,%rax
0x0000000000001387 <+71>:     jne     0x1368 <multiply(double (*) [128], double
(*) [128], double (*) [128])+40>
0x0000000000001389 <+73>:     add     $0x400,%r9
0x0000000000001390 <+80>:     cmp     $0x20000,%r9
0x0000000000001397 <+87>:     jne     0x1351 <multiply(double (*) [128], double
(*) [128], double (*) [128])+17>
0x0000000000001399 <+89>:     add     $0x20000,%r10
0x00000000000013a0 <+96>:     xor     %eax,%eax
0x00000000000013a2 <+98>:     nopw    0x0(%rax,%rax,1)
0x00000000000013a8 <+104>:    movsd   (%rcx,%rax,1),%xmm0
0x00000000000013ad <+109>:    movsd   %xmm0, (%rsi,%rax,1)
0x00000000000013b2 <+114>:    add     $0x8,%rax
0x00000000000013b6 <+118>:    cmp     $0x400,%rax
0x00000000000013bc <+124>:    jne     0x13a8 <multiply(double (*) [128], double
(*) [128], double (*) [128])+104>
0x00000000000013be <+126>:    add     $0x400,%rcx
0x00000000000013c5 <+133>:    add     $0x400,%rsi
0x00000000000013cc <+140>:    cmp     %rcx,%r10
0x00000000000013cf <+143>:    jne     0x13a0 <multiply(double (*) [128], double
(*) [128], double (*) [128])+96>
0x00000000000013d1 <+145>:    pop     %rbx
0x00000000000013d2 <+146>:    retq

```

可以看出，经过优化，编译出的汇编指令精简了许多，同时执行时间大大缩短。意料之中地，这个优化等级依然没有编译出SIMD指令。

## 优化等级3

指定优化等级为 `-O3`，可以看到执行时间约为0.025秒。

```

问题  输出  终端  调试控制台
cheung@cheung:~/Desktop/code/cpp$ gcc test.cpp -o test -g -O3 && ./test
time used: 0.025097
cheung@cheung:~/Desktop/code/cpp$ gcc test.cpp -o test -g -O3 && ./test
time used: 0.026145
cheung@cheung:~/Desktop/code/cpp$ gcc test.cpp -o test -g -O3 && ./test
time used: 0.026383
cheung@cheung:~/Desktop/code/cpp$

```

在这个等级下，查看函数 `multiply` 的汇编代码如下：

```

0x000000000000013c0 <+0>:      endbr64
0x000000000000013c4 <+4>:      mov     %rdi,%r8
0x000000000000013c7 <+7>:      mov     %rdx,%rcx
0x000000000000013ca <+10>:     mov     %rsi,%rdi
0x000000000000013cd <+13>:     lea     0x10(%rdx),%r9
0x000000000000013d1 <+17>:     lea     0x20000(%rdx),%r10
0x000000000000013d8 <+24>:     mov     %r8,%rsi
0x000000000000013db <+27>:     lea     0x10(%rsi),%rax
0x000000000000013df <+31>:     cmp     %rax,%rdx
0x000000000000013e2 <+34>:     setae  %r11b
0x000000000000013e6 <+38>:     cmp     %r9,%rsi
0x000000000000013e9 <+41>:     setae  %al
0x000000000000013ec <+44>:     or      %al,%r11b
0x000000000000013ef <+47>:     je      0x14c9 <multiply(double (*) [128], double
(*) [128], double (*) [128])+265>
0x000000000000013f5 <+53>:     lea     0x10(%rdi),%rax
0x000000000000013f9 <+57>:     cmp     %rax,%rdx
0x000000000000013fc <+60>:     setae  %r11b
0x00000000000001400 <+64>:     cmp     %r9,%rdi
0x00000000000001403 <+67>:     setae  %al
0x00000000000001406 <+70>:     or      %al,%r11b
0x00000000000001409 <+73>:     je      0x14c9 <multiply(double (*) [128], double
(*) [128], double (*) [128])+265>
0x0000000000000140f <+79>:     xor     %eax,%eax
0x00000000000001411 <+81>:     nopl    0x0(%rax)
0x00000000000001418 <+88>:     movupd (%rsi,%rax,1),%xmm0
0x0000000000000141d <+93>:     movupd (%rdi,%rax,1),%xmm1
0x00000000000001422 <+98>:     movupd (%rdx,%rax,1),%xmm2
0x00000000000001427 <+103>:    mulpd  %xmm1,%xmm0
0x0000000000000142b <+107>:    addpd  %xmm2,%xmm0
0x0000000000000142f <+111>:    movups %xmm0, (%rdx,%rax,1)
0x00000000000001433 <+115>:    add     $0x10,%rax
0x00000000000001437 <+119>:    cmp     $0x400,%rax
0x0000000000000143d <+125>:    jne     0x1418 <multiply(double (*) [128], double
(*) [128], double (*) [128])+88>
0x0000000000000143f <+127>:    add     $0x400,%rdx
0x00000000000001446 <+134>:    add     $0x400,%r9
0x0000000000000144d <+141>:    add     $0x400,%rsi
0x00000000000001454 <+148>:    add     $0x400,%rdi
0x0000000000000145b <+155>:    cmp     %r10,%rdx
0x0000000000000145e <+158>:    jne     0x13db <multiply(double (*) [128], double
(*) [128], double (*) [128])+27>
0x00000000000001464 <+164>:    lea     0x10(%rcx),%rax
0x00000000000001468 <+168>:    cmp     %rax,%r8
0x0000000000000146b <+171>:    jae     0x1476 <multiply(double (*) [128], double
(*) [128], double (*) [128])+182>
0x0000000000000146d <+173>:    lea     0x10(%r8),%rax
0x00000000000001471 <+177>:    cmp     %rax,%rcx
0x00000000000001474 <+180>:    jb      0x14aa <multiply(double (*) [128], double
(*) [128], double (*) [128])+234>
0x00000000000001476 <+182>:    xor     %eax,%eax
0x00000000000001478 <+184>:    nopl    0x0(%rax,%rax,1)
0x00000000000001480 <+192>:    movupd (%rcx,%rax,1),%xmm3
0x00000000000001485 <+197>:    movups %xmm3, (%r8,%rax,1)
0x0000000000000148a <+202>:    add     $0x10,%rax
0x0000000000000148e <+206>:    cmp     $0x400,%rax

```



```

0x0000000000001494 <+212>:   jne      0x1480 <multiply(double (*) [128], double
(*) [128], double (*) [128])+192>
0x0000000000001496 <+214>:   add     $0x400,%rcx
0x000000000000149d <+221>:   add     $0x400,%r8
0x00000000000014a4 <+228>:   cmp     %r10,%rcx
0x00000000000014a7 <+231>:   jne     0x1464 <multiply(double (*) [128], double
(*) [128], double (*) [128])+164>
0x00000000000014a9 <+233>:   retq
0x00000000000014aa <+234>:   xor     %eax,%eax
0x00000000000014ac <+236>:   nopl    0x0(%rax)
0x00000000000014b0 <+240>:   movsd   (%rcx,%rax,1),%xmm0
0x00000000000014b5 <+245>:   movsd   %xmm0, (%r8,%rax,1)
0x00000000000014bb <+251>:   add     $0x8,%rax
0x00000000000014bf <+255>:   cmp     $0x400,%rax
0x00000000000014c5 <+261>:   jne     0x14b0 <multiply(double (*) [128], double
(*) [128], double (*) [128])+240>
0x00000000000014c7 <+263>:   jmp     0x1496 <multiply(double (*) [128], double
(*) [128], double (*) [128])+214>
0x00000000000014c9 <+265>:   xor     %eax,%eax
0x00000000000014cb <+267>:   nopl    0x0(%rax,%rax,1)
0x00000000000014d0 <+272>:   movsd   (%rsi,%rax,1),%xmm0
0x00000000000014d5 <+277>:   mulsd   (%rdi,%rax,1),%xmm0
0x00000000000014da <+282>:   addsd   (%rdx,%rax,1),%xmm0
0x00000000000014df <+287>:   movsd   %xmm0, (%rdx,%rax,1)
0x00000000000014e4 <+292>:   add     $0x8,%rax
0x00000000000014e8 <+296>:   cmp     $0x400,%rax
0x00000000000014ee <+302>:   jne     0x14d0 <multiply(double (*) [128], double
(*) [128], double (*) [128])+272>
0x00000000000014f0 <+304>:   jmpq    0x143f <multiply(double (*) [128], double
(*) [128], double (*) [128])+127>

```

可以看出，在这个等级下，编译出的汇编指令更多，甚至比不优化的 -o0 行数更多。但同时，这里真正出现了实现SIMD效果的SIMD指令，后缀为 ps 或者 pd 的指令即为这种指令。p 代表打包方式，对XMM寄存器中的所有浮点数都进行计算；s 应该是代表单精度浮点数；d 应该是代表双精度浮点数。

虽然汇编指令更多了，出现了多条指令可以真正实现单指令流多数据流的效果，但是计算时间得到了进一步缩短（对比 -o2 约40%的提升），这其中与SIMD的使用有很大关系。如果测试代码是在进行多媒体处理并且可以指定mmx或sse进行优化，相信结果会更加明显。

## 实验体会

实验过程中我实验了多种可能，比如矩阵乘法的具体实现和gcc的参数配置。虽然结果并不如一开始计划的完美，但是在这个过程中我又学习到了很多SIMD的相关知识和gcc的相关知识。

而在分析汇编指令时，我还发现即使在不使用SIMD指令时，汇编使用的寄存器也是XMM寄存器而不是浮点数寄存器。我推测在今日的计算机体系下，英特尔因为XMM、YMM寄存器的存在而舍弃了浮点数寄存器，节省了资源（好像看过哪个资料说过类似的话，但是现在找不到了）。而使用同一套寄存器而导致的SIMD与浮点数操作之间的矛盾，应该是通过命令中的打包方式（即后缀中包含了p）和标量方式（即后缀中包含了s）进行了处理。

## 经验教训

实验要求较宽，遇到难以解决的困难时要在合理的范围内调整原计划，以及要多学点gcc。



## 参考文献

- 【1】 [SIMD](#) BAIDU
- 【2】 [SIMD](#) WIKI
- 【3】 [MMX](#) BAIDU
- 【4】 [SSE](#) BAIDU
- 【5】 [SSE指令集](#)
- 【6】 [SSE指令集简介](#)
- 【7】 [SIMD指令学习笔记](#)