# 南 开 大 学

NanKai University


# SIMD 编程实验——以高斯消去为例

姓名：文静静

学号：1811507

年级：2018级

专业：计算机科学与技术

2021年4月13日

## 摘要

本文探究串行的高斯消去法(LU分解)在使用并行优化时性能的提升，同时探究了不同算法策略对性能的提升，如使用SSE和AVX指令、对齐与不对齐、是否完全向量化等，并使用工具VTune具体剖析了性能变化。

**关键字：**高斯消去　SSE　AVX　对齐　向量化

# 目录

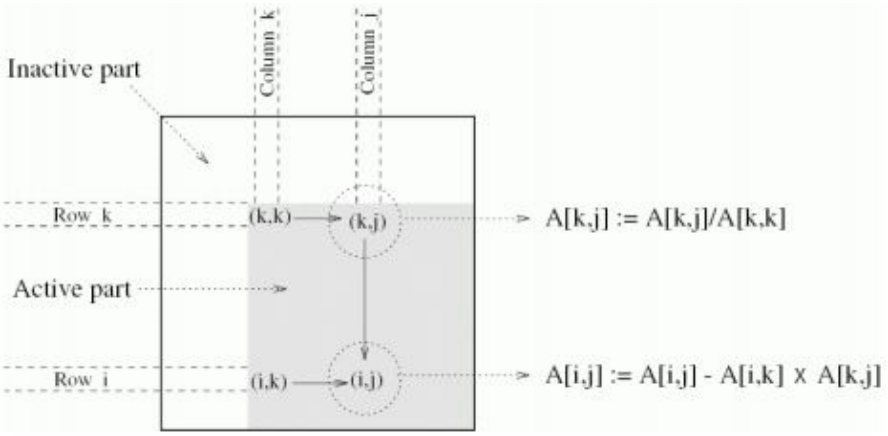
# 1 绪论

本文通过高斯消去法探究 SSE 指令和 AVX 指令对于性能的提升，理解 SIMD 实现数据并行从而提高运算效率。具体探究了使用 SSE 和 AVX 指令、对齐与不对齐、是否完全向量化的不同情况下的时间性能以及加速比，还探究了算法优化产生的误差。使用了性能剖析根据 VTune 对几种算法进行了具体的剖析，对比了 SSE/AVX 优化与一般串行，对齐与不对齐等策略最终所执行的指令数，周期数，CPI 。

# 2 实验设计

## 2.1 实验选题

高斯消去的计算模式如图所示，在第 k 步时，对第 k 行从 (k, k) 开始进行除法操作，并且将后续的 k + 1 至 N 行进行减去第 k 行的操作。




高斯消去法（LU 分解）SIMD 并行化为例，具体要求如下：

1. 设计实现 SSE（及 AVX 等）算法，加速计算过程。

2. 讨论不同算法策略对性能的影响，如 SSE 和 AVX、对齐与不对齐、4-5行的循环是否向量化、cache 等与体系结构相关的优化、arm 平台等。

## 2.2 算法设计与编程

初始化系数矩阵：在使用高斯消去法的时候，有除法操作，为了避免出现除数为0以及出现因为算法改变运算顺序导致误差的情况，将系数矩阵初始为单位矩阵以便运算和验证。（N为系数矩阵规模大小）

```
// 初始系数矩阵，为了方便将其设置成单位矩阵A
void init()
{
    A = new float *[N];
    for (int i = 0; i < N; i++)
    {
        A[i] = new float[N];
        A[i][i] = 1.0;
    }
}
```

串行的高斯消去法代码如下：

```
// 高斯消去法（串行）
void Gauss_plain()
{
    for (int k = 0; k < N; k++)
    {
        for (int j = k + 1; j < N; j++)
        {
            A[k][j] = A[k][j] / A[k][k];
        }
        A[k][k] = 1.0;
        for (int i = k + 1; i < N; i++)
        {
            for (int j = k + 1; j < N; j++)
            {
```

```
15                  A[i][j] = A[i][j] - A[i][k] * A[k][j];
16              }
17              A[i][k] = 0;
18          }
19      }
20  }
```

## 2.3 SSE的C++编程

使用SSE进行并行优化的时候，考虑对齐与不对齐、部分优化与完全优化对性能提升的影响，因此使用不同函数进行对比。

SSE指令的数据类型为 __m128，能同时处理4个float数据。其中指导书的第 4、5 行的内嵌循环和第 9、10 行的内嵌循环都可以进行向量化下面会对这两步进行优化。将分别对 4、5 行和第 9、10 行进行向量化进行对比体现不同部分优化对程序性能提升的影响。

要设计对齐与不对齐的对比，在对齐的情况下，地址总是向量长度的倍数（在 SSE 算法中是 16 字节的倍数），只有一个超级字的读写操作，使用 _mm_load_ps 和 _mm_store_ps。在未对齐的情况下，会产生对齐开销，需要使用 _mm_loadu_ps 和 _mm_storeu_ps。一般而言数据是从起始地址处对齐的，对齐时默认规模N为4的倍数测试，不对齐时可以将N–来处理，这样当串行处理完前面不够组成 4 个的数据，新的起始地址就不是 16 字节的倍数，可以测试未对齐的函数。

另外，指导书串行算法中第 4、5 行的循环中用到了除法，为了提高效率，可以使用 _mm_set1_ps(1/A[k][k])，将 128 位寄存器的值设置为 4 个 A[k][k] 的倒数，在循环中乘以它的倒数，减少除法的次数。

**SSE优化：对齐，4、5行和8、9行向量化：**

```
// 并行优化，对齐，向量化SSE
void SSE_aligned_45()
{
    for (int k = 0; k < N; k++)
    {
        __m128 t1, t2, t3, t4;
        // 不能凑够个的部分串行计算4
        int remain = (N - k - 1) % 4;
        for (int j = k + 1; j <= k + remain; j++)
        {
            A[k][j] = A[k][j] / A[k][k];
        }
        // 并行计算剩下的倍数个，减少使用除法4
        t1 = _mm_set1_ps(1 / A[k][k]);
        for (int j = k + remain + 1; j < N; j += 4)
        {
            t2 = _mm_load_ps(A[k] + j);
            t2 = _mm_mul_ps(t1, t2);
            _mm_store_ps(A[k] + j, t2);
        }
        A[k][k] = 1;
        for (int i = k + 1; i < N; i++)
        {
            // 不能凑够个的部分串行计算4
            for (int j = k + 1; j <= k + remain; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            t1 = _mm_set1_ps(A[i][k]);
            // 并行计算剩下的倍数个4float
            for (int j = k + remain + 1; j < N; j += 4)
            {
```

```
                t2 = _mm_load_ps(A[k] + j);
                t3 = _mm_mul_ps(t1, t2);
                t4 = _mm_load_ps(A[i] + j);
                t4 = _mm_sub_ps(t4, t3);
                _mm_store_ps(A[i] + j, t4);
            }
            A[i][k] = 0;
        }
    }
}
```

**SSE优化：对齐，4、5行不向量化，8、9行向量化：**

```
// 并行优化，对齐，向量化，不向量化SSE8945
void SSE_aligned_u45_89()
{
    for (int k = 0; k < N; k++)
    {
        __m128 t1, t2, t3, t4;
        for (int j = k + 1; j < N; j++)
        {
            A[k][j] = A[k][j] / A[k][k];
        }
        A[k][k] = 1.0;
        for (int i = k + 1; i < N; i++)
        {
            // 先把不能凑够个的部分串行计算4
            int remain = (N - k - 1) % 4;
            for (int j = k + 1; j <= k + remain; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            t1 = _mm_set1_ps(A[i][k]);
            // 并行计算剩下的倍数个4float
```

```
            for (int j = k + remain + 1; j < N; j += 4)
            {
                t2 = _mm_load_ps(A[k] + j);
                t3 = _mm_mul_ps(t1, t2);
                t4 = _mm_load_ps(A[i] + j);
                t4 = _mm_sub_ps(t4, t3);
                _mm_store_ps(A[i] + j, t4);
            }
            A[i][k] = 0;
        }
    }
}
```

**SSE优化：对齐，4、5行向量化，8、9行不向量化：**

```
// 并行优化，对齐，向量化，不向量化SSE4589
void SSE_aligned_45_u89()
{
    for (int k = 0; k < N; k++)
    {
        __m128 t1, t2, t3, t4;
        // 不能凑够个的部分串行计算4
        int remain = (N - k - 1) % 4;
        for (int j = k + 1; j <= k + remain; j++)
        {
            A[k][j] = A[k][j] / A[k][k];
        }
        // 并行计算剩下的倍数个，减少使用除法4
        t1 = _mm_set1_ps(1 / A[k][k]);
        for (int j = k + remain + 1; j < N; j += 4)
        {
            t2 = _mm_load_ps(A[k] + j);
            t2 = _mm_mul_ps(t1, t2);
            _mm_store_ps(A[k] + j, t2);
```

```
        }
        A[k][k] = 1;
        for (int i = k + 1; i < N; i++)
        {
            for (int j = k + 1; j < N; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

**SSE优化：不对齐，4、5行和8、9行向量化：**

```
// 并行优化，不对齐，向量化SSE4589
void SSE_unaligned_45_89()
{
    for (int k = 0; k < N; k++)
    {
        __m128 t1, t2, t3, t4;
        // 不能凑够个的部分串行计算4
        int remain = (N - k - 1) % 4;
        for (int j = k + 1; j <= k + remain; j++)
        {
            A[k][j] = A[k][j] / A[k][k];
        }
        // 并行计算剩下的倍数个，减少使用除法4
        t1 = _mm_set1_ps(1 / A[k][k]);
        for (int j = k + remain + 1; j < N; j += 4)
        {
            t2 = _mm_loadu_ps(A[k] + j);
            t2 = _mm_mul_ps(t1, t2);
            _mm_storeu_ps(A[k] + j, t2);
```

```
        }
        A[k][k] = 1.0;
        for (int i = k + 1; i < N; i++)
        {
            // 不能凑够个的部分串行计算4
            for (int j = k + 1; j <= k + remain; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            t1 = _mm_set1_ps(A[i][k]);
            // 并行计算剩下的倍数个4float
            for (int j = k + remain + 1; j < N; j += 4)
            {
                t2 = _mm_loadu_ps(A[k] + j);
                t3 = _mm_mul_ps(t1, t2);
                t4 = _mm_loadu_ps(A[i] + j);
                t4 = _mm_sub_ps(t4, t3);
                _mm_storeu_ps(A[i] + j, t4);
            }
            A[i][k] = 0;
        }
    }
}
```

**SSE优化：不对齐，4、5行不向量化，8、9行向量化：**

```
// 并行优化，不对齐，不向量化，向量化SSE4589
void SSE_unaligned_u45_89()
{
    for (int k = 0; k < N; k++)
    {
        __m128 t1, t2, t3, t4;
        for (int j = k + 1; j < N; j++)
        {
```

```
            A[k][j] = A[k][j] / A[k][k];
        }
        A[k][k] = 1.0;
        for (int i = k + 1; i < N; i++)
        {
            // 先把不能凑够个的部分串行计算4
            int remain = (N - k - 1) % 4;
            for (int j = k + 1; j <= k + remain; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            t1 = _mm_set1_ps(A[i][k]);
            // 并行计算剩下的倍数个4float
            for (int j = k + remain + 1; j < N; j += 4)
            {
                t2 = _mm_loadu_ps(A[k] + j);
                t3 = _mm_mul_ps(t1, t2);
                t4 = _mm_loadu_ps(A[i] + j);
                t4 = _mm_sub_ps(t4, t3);
                _mm_storeu_ps(A[i] + j, t4);
            }
            A[i][k] = 0;
        }
    }
}
```

**SSE优化：不对齐，4、5行向量化，8、9行不向量化：**

```
// 并行优化，不对齐，向量化，不向量化SSE4589
void SSE_unaligned_45_u89()
{
    for (int k = 0; k < N; k++)
    {
        __m128 t1, t2, t3, t4;
```

```
7           // 不能凑够个的部分串行计算4
8           int remain = (N - k - 1) % 4;
9           for (int j = k + 1; j <= k + remain; j++)
10          {
11              A[k][j] = A[k][j] / A[k][k];
12          }
13          // 并行计算剩下的倍数个，减少使用除法4
14          t1 = _mm_set1_ps(1 / A[k][k]);
15          for (int j = k + remain + 1; j < N; j += 4)
16          {
17              t2 = _mm_loadu_ps(A[k] + j);
18              t2 = _mm_mul_ps(t1, t2);
19              _mm_storeu_ps(A[k] + j, t2);
20          }
21          A[k][k] = 1.0;
22          for (int i = k + 1; i < N; i++)
23          {
24              for (int j = k + 1; j < N; j++)
25              {
26                  A[i][j] = A[i][j] - A[i][k] * A[k][j];
27              }
28              A[i][k] = 0;
29          }
30      }
31  }
```

## 2.4 AVX的C++编程

AVX的指令与SSE类似，使用__m256来存储8个float，由于SSE中已经比较了部分向量化的影响，因此AVX只测试对齐与不对齐的情况。

**AVX优化：对齐，4、5行和8、9行向量化：**

```
// 并行优化，对齐，向量化AVX
void AVX_aligned_4589()
{
    for (int k = 0; k < N; k++)
    {
        __m256 t1, t2, t3, t4;
        // 不能凑够个的部分串行计算8
        int remain = (N - k - 1) % 8;
        for (int j = k + 1; j <= k + remain; j++)
        {
            A[k][j] = A[k][j] / A[k][k];
        }
        // 并行计算剩下的倍数个，减少使用除法8
        t1 = _mm256_set1_ps(1 / A[k][k]);
        for (int j = k + remain + 1; j < N; j += 8)
        {
            t2 = _mm256_load_ps(A[k] + j);
            t2 = _mm256_mul_ps(t1, t2);
            _mm256_store_ps(A[k] + j, t2);
        }
        A[k][k] = 1.0;
        for (int i = k + 1; i < N; i++)
        {
            // 不能凑够个的部分串行计算8
            for (int j = k + 1; j <= k + remain; j++)
            {
```

```
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            t1 = _mm256_set1_ps(A[i][k]);
            // 并行计算剩下的倍数个8float
            for (int j = k + remain + 1; j < N; j += 8)
            {
                t2 = _mm256_load_ps(A[k] + j);
                t3 = _mm256_mul_ps(t1, t2);
                t4 = _mm256_load_ps(A[i] + j);
                t4 = _mm256_sub_ps(t4, t3);
                _mm256_store_ps(A[i] + j, t4);
            }
            A[i][k] = 0;
        }
    }
}
```

**AVX优化：不对齐，4、5行和8、9行向量化：**

```
// 并行优化，不对齐，向量化AVX
void AVX_unaligned_4589()
{
    for (int k = 0; k < N; k++)
    {
        __m256 t1, t2, t3, t4;
        // 不能凑够个的部分串行计算8
        int remain = (N - k - 1) % 8;
        for (int j = k + 1; j <= k + remain; j++)
        {
            A[k][j] = A[k][j] / A[k][k];
        }
        // 并行计算剩下的倍数个，减少使用除法8
        t1 = _mm256_set1_ps(1 / A[k][k]);
        for (int j = k + remain + 1; j < N; j += 8)
```

```
16          {
17              t2 = _mm256_loadu_ps(A[k] + j);
18              t2 = _mm256_mul_ps(t1, t2);
19              _mm256_storeu_ps(A[k] + j, t2);
20          }
21          A[k][k] = 1.0;
22          for (int i = k + 1; i < N; i++)
23          {
24              // 不能凑够个的部分串行计算8
25              for (int j = k + 1; j <= k + remain; j++)
26              {
27                  A[i][j] = A[i][j] - A[i][k] * A[k][j];
28              }
29              t1 = _mm256_set1_ps(A[i][k]);
30              // 并行计算剩下的倍数个8float
31              for (int j = k + remain + 1; j < N; j += 8)
32              {
33                  t2 = _mm256_loadu_ps(A[k] + j);
34                  t3 = _mm256_mul_ps(t1, t2);
35                  t4 = _mm256_loadu_ps(A[i] + j);
36                  t4 = _mm256_sub_ps(t4, t3);
37                  _mm256_storeu_ps(A[i] + j, t4);
38              }
39              A[i][k] = 0;
40          }
41      }
42  }
```

# 3 编译运行

在CodeBlocks中编译运行：通过QueryPerformance来精确计时，同时运行epoch=10次取平均值。

SSE的编译选项：-march=corei7、-march=native

AVX的编译选项：-march=corei7-avx、-march=native

串行算法时可以选择打开-O3选项，会对程序做向量化。

# 4 结果分析

## 4.1 测试规模与cache

使用CPU-Z查看本机cache的信息如下：

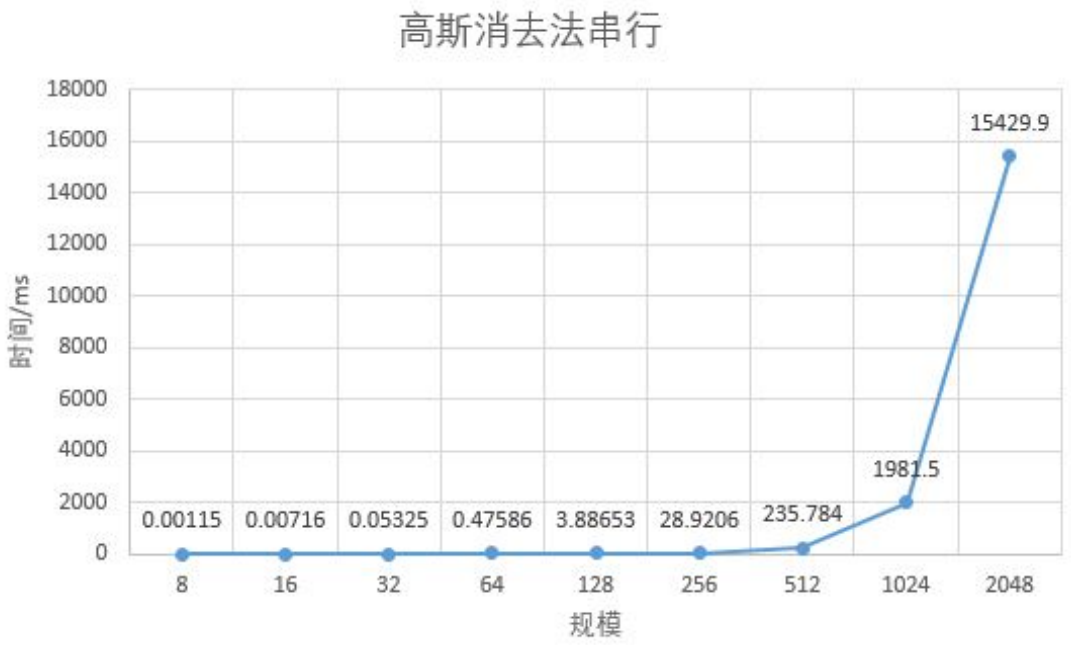

一级数据缓存大小为 32KBytes，二级缓存为 256KBytes，三级缓存为 6MBytes。一级数据缓存能放下 8192 个 float，，二级缓存

能放下 256*256 个 float，三级缓存能放下 1024*1536 个 float，所以选择矩阵大小 N=8、16、32、64、128、256、512、1024、2048 九组数据进行测试。

得到如下表的运行结果：

| 规模N | 串行 | 串行-O3优化 | SSE(对齐，45向量化，89向量化) | SSE(对齐，45不向量化，89向量化) | SSE(对齐，45向量化，89不向量化) | AVX(对齐，45向量化，89向量化) |
|---|---|---|---|---|---|---|
| 8 | 0.00115 | 0.00083 | 0.00094 | 0.00146 | 0.00186 | 0.00517 |
| 16 | 0.00716 | 0.00546 | 0.00564 | 0.00606 | 0.00766 | 0.00638 |
| 32 | 0.05325 | 0.04296 | 0.03071 | 0.03377 | 0.05383 | 0.02361 |
| 64 | 0.47586 | 0.33988 | 0.2188 | 0.22346 | 0.40211 | 0.15987 |
| 128 | 3.88653 | 2.68532 | 1.98478 | 2.11676 | 3.52207 | 0.95414 |
| 256 | 28.9206 | 21.6384 | 15.4225 | 15.6266 | 28.3865 | 6.39229 |
| 512 | 235.784 | 160.638 | 115.489 | 116.855 | 229.47 | 51.6315 |
| 1024 | 1981.5 | 1296.83 | 986.349 | 954.323 | 1866.02 | 468.935 |
| 2048 | 15429.9 | 10539.1 | 7959.8 | 7971.22 | 13313.3 | 3784.13 |

其中串行算法的时间随规模变化如下图：




可以发现当规模超过256时，时间增加明显增多，规模到1024时，时间开始急速增长，跟我们的cache存储还是比较符合的。串行算法的第二个内层循环，是逐行访问矩阵元素的，这相比于逐列访问已经很好地利用了 cache 的特点，因此算法在cache上的优化暂无法做到更好。

## 4.2 对齐与不对齐

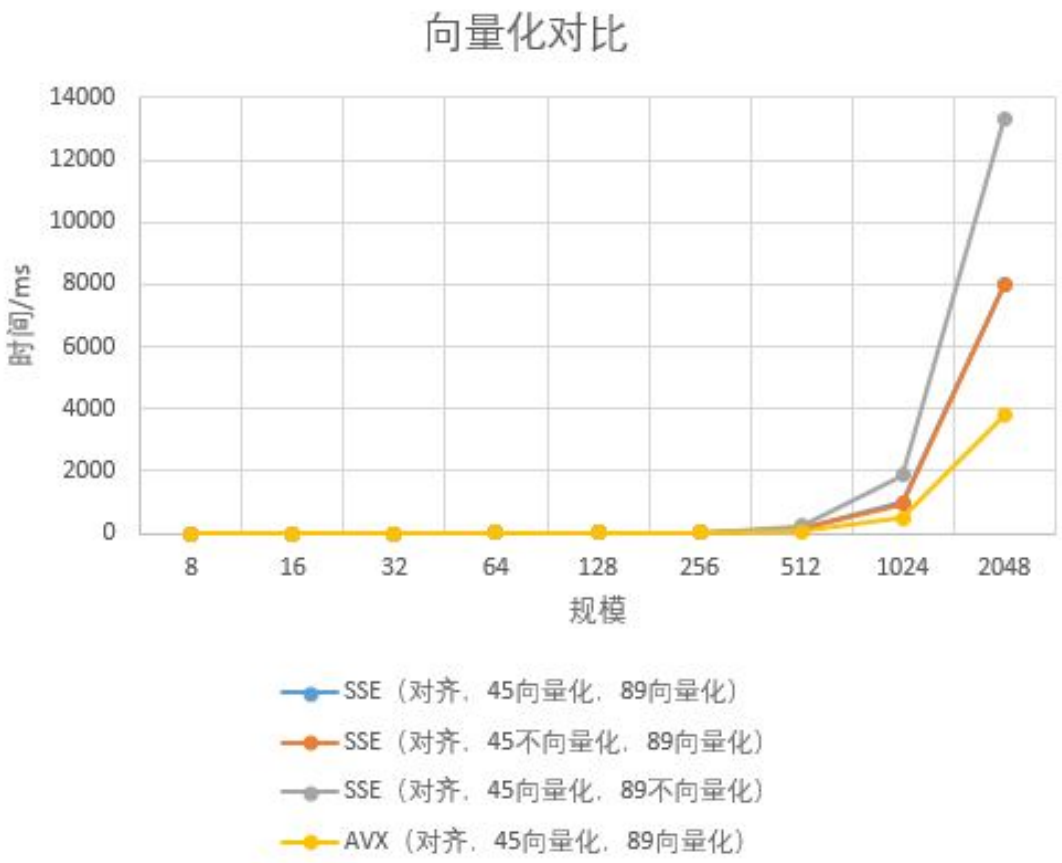
补充未对齐时的实验结果如下表：

| 规模N | 串行 | SSE(对齐，45向量化，89向量化) | SSE(不对齐，45向量化，89向量化) | AVX(对齐，45向量化，89向量化) | AVX(不对齐，45向量化，89向量化) |
|---|---|---|---|---|---|
| 8 | 0.00115 | 0.00094 | 0.0028 | 0.00517 | 0.00263 |
| 16 | 0.00716 | 0.00564 | 0.00681 | 0.00638 | 0.00791 |
| 32 | 0.05325 | 0.03071 | 0.03816 | 0.02361 | 0.03618 |
| 64 | 0.47586 | 0.2188 | 0.19836 | 0.15987 | 0.16504 |
| 128 | 3.88653 | 1.98478 | 1.51893 | 0.95414 | 0.96584 |
| 256 | 28.9206 | 15.4225 | 15.7674 | 6.39229 | 6.50446 |
| 512 | 235.784 | 115.489 | 117.9799 | 51.6315 | 50.5042 |
| 1024 | 1981.5 | 986.349 | 997.705 | 468.935 | 470.778 |
| 2048 | 15429.9 | 7959.8 | 7961.93 | 3784.13 | 3822.77 |

可以看出大部分时间上，对齐均优于不对齐的情况，由此可见数据不对齐时带来的开销对性能的影响。

## 4.3 向量化

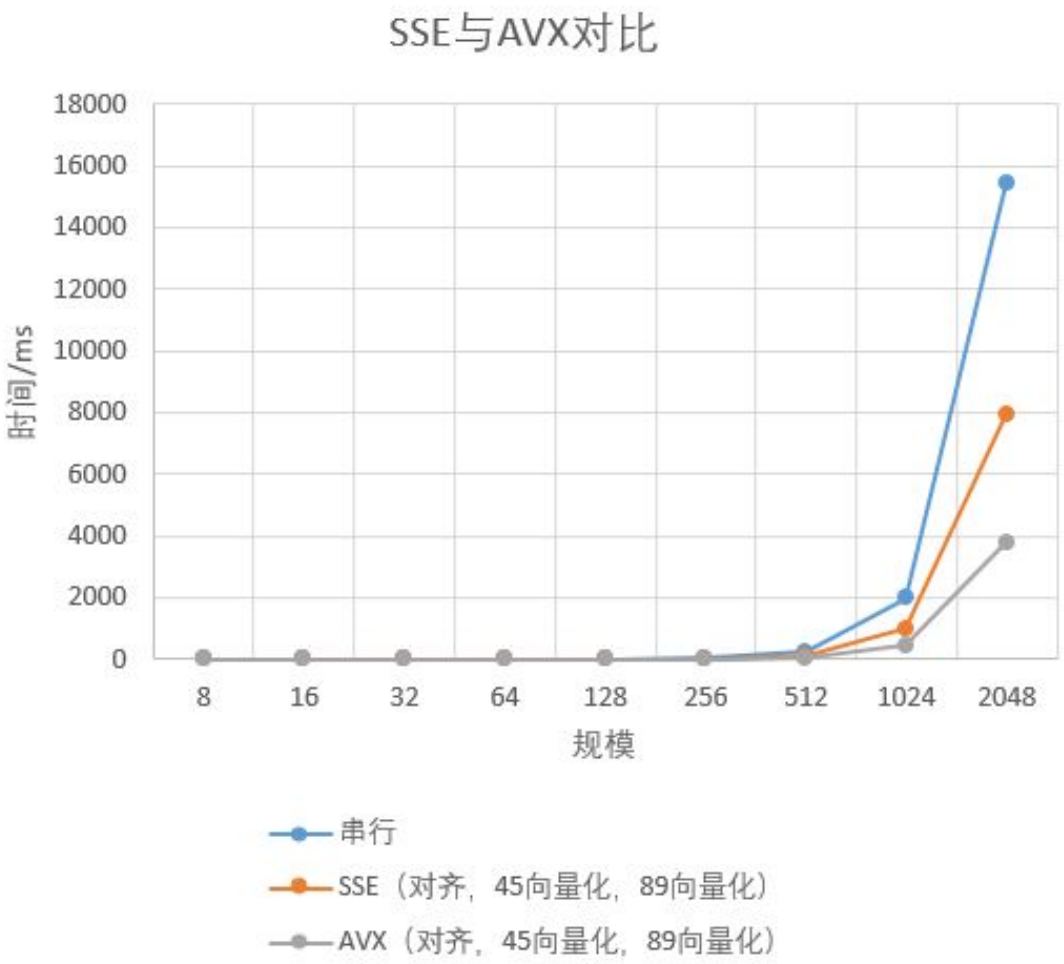
对比不同部分向量化带来的影响：




可见当串行45行不进行向量化时，时间比全部向量化略多，但是当串行89行不进行向量化时，时间远多于全部向量化的处理，主要还是89行的代码比较多，向量化能提升的性能比较多。

## 4.4 SSE/AVX

串行、SSE优化、AVX优化的时间随规模变化如下：




对于高斯消去法的串行算法的时间复杂度是 O($n^3$)。假设 SSE 寄存器的读写、计算所用时间是普通寄存器用时的 k1 倍，对于全部并行的 SSE 算法，复杂度大致为 O(k1/4*$n^2$+k1/4*$n^3$)，加速比约为 4/k1。同理，假设 AVX 寄存器的读写、计算所用时间是普通寄存器用时的 k2 倍，那么加速比约为 8/k2。

计算加速比得到表如下：

| 规模N | 串行-O3优化 | SSE(对齐，45向量化，89向量化) | SSE(不对齐，45向量化，89向量化) | AVX(对齐，45向量化，89向量化) | AVX(不对齐，45向量化，89向量化) |
|---|---|---|---|---|---|
| 8 | 1.38554 | 1.2234 | 0.410714 | 0.22243 | 0.43726 |
| 16 | 1.31136 | 1.2695 | 1.051395 | 1.122257 | 0.9052 |
| 32 | 1.23953 | 1.73396 | 1.39544 | 2.2554 | 1.47181 |
| 64 | 1.40008 | 2.17486 | 2.0991 | 2.9765 | 2.8833 |
| 128 | 1.4473 | 1.9582 | 2.0587 | 4.07333 | 4.02399 |
| 256 | 1.33654 | 1.8752 | 1.8342 | 4.52429 | 4.4463 |
| 512 | 1.4678 | 2.0416 | 1.99851 | 4.56667 | 4.6686 |
| 1024 | 1.52796 | 2.0089 | 1.9861 | 4.85662 | 4.8455 |
| 2048 | 1.4641 | 2.10384 | 1.9379 | 5.0029 | 4.95375 |

可以发现SSE的最大加速比为 2.17486，而AVX的最大加速比为 5.0029。当数据规模较小，小于 256 时，SSE 和 AVX 优化的加速比随着数据规模的增加而增加，当数据规模较大后，加速比有时不再上升，反而有一些下降，并且与上面复杂度分析中 SSE 的加速比约为 4/k1，AVX 的加速比为 8/k2 的结论并不完全相符，分析原因，可能是打包、解包、对齐的开销抵消了一部分 SIMD 并行带来的收益，数据规模较小时并行的数据量不足，收益很小；后期数据规模增加时，并行的收益在增加但是代价也在增加。

## 4.5 浮点误差

关于指令运算顺序导致的浮点数的误差的探究，在上一次实验中已经通过代码和测试得出了相应结论。此次测试通过comp()函数来对高斯消去（串行）和高斯消去（SSE并行优化）得到的矩阵进行比较，如果误差超出10e-6，将会返回false。

```
bool comp(float A[N][N],float B[N][N]){
    for (int i = 0; i < N; i++)
    {
        for (int j = i; j < N; j++)
        {
            if(abs(A[i][j] - B[i][j])>1e-6){
                return false;
            }
        }
    }
    return true;
}
```

同时将初始化进行修改，初始成上三角矩阵，确保高斯消去能运行，不会出现除数为0的情况，又能测试结果。

```
void init(float A[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = i; j < N; j++)
        {
            A[i][j] = i + j + 1.0;
        }
    }
}
```

测试N=32时，还未出现误差：

```
当N=32时：
高斯串行算法与SSE并行优化的比较：
误差未超出10e-6

Process returned 0 (0x0)   execution time : 0.263 s
Press any key to continue.
```

测试N=64时，就已经出现超出10e-6了。

```
当N=64时：
高斯串行算法与SSE并行优化的比较：
误差超出10e-6，误差为：1.90735e-006

Process returned 0 (0x0)   execution time : 0.271 s
Press any key to continue.
```

# 5 使用VTune剖析程序性能

SSE/AVX 优化与一般串行，对齐与不对齐等策略最终所执行的指令数，周期数，CPI 是不一样的。使用 VTune 分析对比以上不同策略在 N=1024（未对齐情况取 N=1023）下的执行的指令数和 CPI。

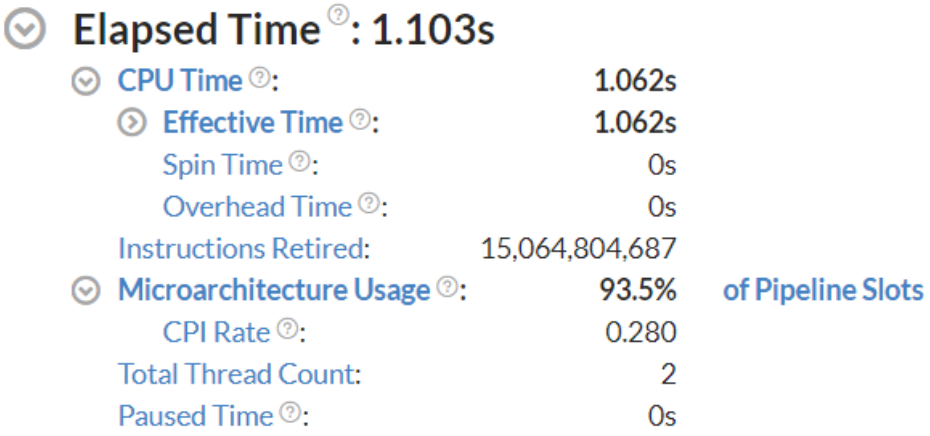



图 1: Gauss_plain


图 2: Gauss_SSE_aligned_45_89


图 3: Gauss_SSE_unaligned_45_89


图 4: Gauss_SSE_aligned_u45_89


图 5: Gauss_SSE_unaligned_u45_89


图 6: Gauss_AVX_aligned_45_89

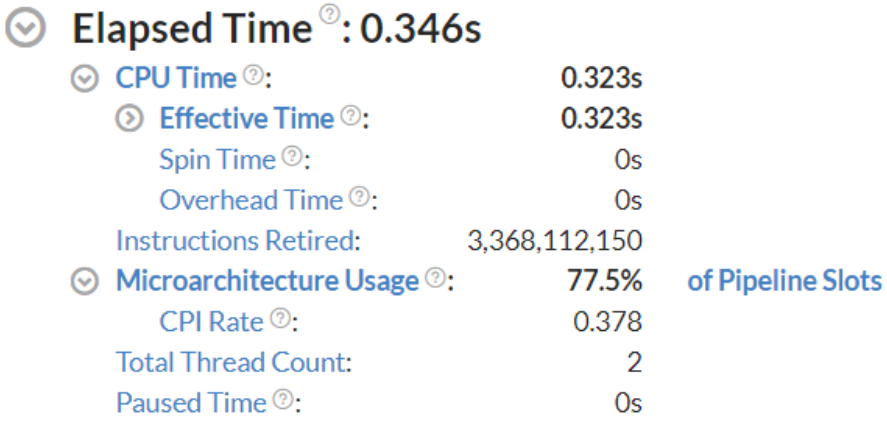



图 7: Gauss_AVX_unaligned_45_89

由此可得到时间与指令数、CPI的对比表格如下：

| 算法 | 时间 | 指令数 | CPI |
|---|---|---|---|
| 串行 | 1. 103 | 15064804687 | 0. 28 |
| SSE，对齐，全部向量化 | 0. 586 | 5518344772 | 0. 399 |
| SSE，不对齐，全部向量化 | 0. 626 | 6577287693 | 0. 352 |
| SSE，对齐，45不向量化 | 0. 585 | 5532643296 | 0. 399 |
| SSE，不对齐，45不向量化 | 0. 623 | 6593351587 | 0. 354 |
| AVX，对齐，全部向量化 | 0. 323 | 2831652936 | 0. 42 |
| AVX，不对齐，全部向量化 | 0. 346 | 3368112150 | 0. 378 |

可以发现，AVX 优化算法的指令数最少，CPI 最大，串行算法的指令数比其他优化算法多出了一个数量级，其他优化策略下的 CPI 是串行算法的2倍左右。这说明 SSE/AVX 优化可以减少指令条数，提高 CPI，从而提高程序的运行效率。而且对齐时的指令周期数明显少于不对齐时的指令周期数，CPI 大于不对齐时的 CPI。

# 6 总结

这次实验理解了 SIMD 实现数据并行从而提高运算效率。也体会到 SIMD还有很多瓶颈，如地址不对齐、运算顺序带来的误差、数据重组和控制相关的向量化等问题。