



南開大學
Nankai University

南 開 大 學

計 算 機 學 院

並行程序設計實驗報告

高斯消去法 SIMD 并行化实验报告

周辰霏 1712991

年 級：2017 級

專 業：計算機科學與技術

2020 年 3 月 24 日

摘要

针对高斯消元法 (LU 分解) 的 SIMD 并行化的 SSE 和 AVX 实验。并对每一种算法进行基于 Vtune 的微观 Profiling 验证实验结果。

关键字：LU 分解；SIMD 编程；SSE/AVX；Vtune

目录

一、 概述	1
(一) 问题描述	1
(二) 实验内容	1
(三) 实验环境	1
二、 算法设计与实现	1
(一) 串行朴素 LU 实现	1
(二) SSE 加速	2
(三) SSE 对齐加速	3
(四) SSE 向量化加速	3
(五) AVX 优化加速	4
三、 高精度计时性能结果比较	7
(一) 基于 Vtune 的程序性能剖析	11
四、 总结	12

一、 概述

(一) 问题描述

高斯消元法 (Gaussian Elimination) 是线性代数中的一个算法, 用于线性方程组求解、求矩阵的秩、求可逆方阵的逆矩阵。[1]

高斯消元法实际上就是最朴素的线性方程组消元, 其时间复杂度是 $O(n^3)$, 是一个在矩阵规模 N 较大时非常费时的算法, 且在高斯消元法的串行平凡算法中, 并非每一步都是必须的, 或者某几步是可以进行向量化的循环的。加之最近对于 SIMD 编程的学习, 所以在此通过实现 SSE 以及 AVX 及其不同策略的一系列优化达到 SIMD 并行化的目的。

(二) 实验内容

针对高斯消去法 (LU 分解) 进行以下 SIMD 并行化:

- 串行朴素 LU
- SSE 优化 LU、SSE 对齐 LU、SSE4-5 行向量化
- AVX 优化 LU、AVX 对齐 LU、AVX4-5 行向量化

并对各种并行化方法进行性能和参数上的比较包括但不限于数据测试规模、执行指令数、周期数、CPI 等的测算比对。

(三) 实验环境

- 系统环境: Windows10 家庭中文版 (64 位)
- 编译环境: gcc 8.1.0 x86_64-posix-seh, Codeblocks17.12, SSE3 指令
- 实验测试工具: Intel Vtune profiler 2020

二、 算法设计与实现

(一) 串行朴素 LU 实现

根据 Wiki 以及实验指导书, 串行朴素 LU 实际上就是将矩阵化为一个上三角矩阵, 其实现步骤为:

1. 第 k 步的时候从第 $k+1$ 个元素开始除以第 k 个元素
2. 第 k 步的时候从第 $k+1$ 个元素开始减去第 k 个元素与上一行第 $k+1$ 个元素的乘积

串行朴素

```
1 void naive_lu(float mat[][N])
2 {
3     for (int k = 0; k < N; k++)
4     {
5         for (int j = k + 1; j < N; j++) // 除法
6             mat[k][j] /= mat[k][k];
```

```

7         mat[k][k] = 1.0;
8         for (int i = k + 1; i < N; i++) // 减去第k行
9         {
10             for (int j = k + 1; j < N; j++)
11                 mat[i][j] -= mat[i][k] * mat[k][j];
12             mat[i][k] = 0;
13         }
14     }
15 }

```

(二) SSE 加速

由于高斯消去法做了较多的重复同类运算, 故可以通过合并运算进行 SSE 优化。此算法对伪码 8-9 行进行减法并行向量化。算法设计主要工作为: 中间变量引入及其数据类型更改 float→__m128、_mm_loadu_ps 加载数据到寄存器 _mm_storeu_ps 寄存器赋给变量、对循环进行 SSE 合并计算乘法 _mm_mul_ps 和减法 _mm_sub_ps。时间复杂度由原来的三次方变为 $O(n^3)/4$

SSE 优化

```

1 void sse_lu(float mat[][N])
2 {
3     __m128 t1, t2, t3; // 中间变量
4     for (int k = 0; k < N; k++)
5     {
6         for (int j = k + 1; j < N; j++)
7             mat[k][j] /= mat[k][k];
8         mat[k][k] = 1.0;
9         for (int i = k + 1; i < N; i++)
10        {
11            float temp[4] = {mat[i][k], mat[i][k], mat[i][k], mat[i][k]};
12            t1 = _mm_loadu_ps(temp);
13            int j = k + 1;
14            for (; j < N - 3; j += 4) // 合并
15            {
16                t2 = _mm_loadu_ps(mat[i] + j);
17                t3 = _mm_loadu_ps(mat[k] + j);
18                t3 = _mm_mul_ps(t1, t3);
19                t2 = _mm_sub_ps(t2, t3);
20                _mm_storeu_ps(mat[i] + j, t2); // mat[i][j]=t2
21            }
22            for (; j < N; j++) // 此时j=N-3只需计算余数部分
23                mat[i][j] -= mat[i][k] * mat[k][j];
24            mat[i][k] = 0;
25        }
26    }
27 }

```

(三) SSE 对齐加速

与 SSE 加速不同就在于数据加载的时候使用的是要求数据对齐的指令 (`_mm_load_ps` 和 `_mm_store_ps`) 以及对齐属性。由于主体部分未变复杂度仍为 $O(n^3)/4$

SSE 对齐

```

1 void sse_lu_aligned(float mat[][N])
2 {
3     __m128 t1, t2, t3; // 中间变量
4     for (int k = 0; k < N; k++)
5     {
6         for (int j = k + 1; j < N; j++)
7             mat[k][j] /= mat[k][k];
8         mat[k][k] = 1.0;
9         for (int i = k + 1; i < N; i++)
10        {
11            float temp[4] __attribute__((aligned(16)))
12            = {mat[i][k], mat[i][k], mat[i][k], mat[i][k]};
13            t1 = _mm_load_ps(temp);
14            int j = k + 1;
15            for (j; j % 4 != 0; j++) // 不对齐部分
16                mat[i][j] -= mat[i][k] * mat[k][j];
17            for (j; j < N - 3; j += 4) // 对齐
18            {
19                t2 = _mm_load_ps(mat[i] + j);
20                t3 = _mm_load_ps(mat[k] + j);
21                t3 = _mm_mul_ps(t1, t3);
22                t2 = _mm_sub_ps(t2, t3);
23                _mm_store_ps(mat[i] + j, t2); // mat[i][j]=t2
24            }
25            for (j; j < N; j++) // 余下
26                mat[i][j] -= mat[i][k] * mat[k][j];
27            mat[i][k] = 0;
28        }
29    }
30 }

```

(四) SSE 向量化加速

根据实验指导,除法计算以及减去前 k 行的循环都可以进行循环向量化。前文的 SSE 优化仅针对更复杂的第二个循环进行了优化。此处向量化是针对两个可以向量化的部分进行并行优化。此处是不对齐数据移动,处理方式与上文 SSE 优化一致。原来 $O(N^2)$ 的部分也变成 $O(N^2)/4$ 了,但总体仍为 $O(n^3)/4$

SSE 向量化

```

1 void sse_lu_vectorized(float mat[][N])
2 {
3     __m128 t1, t2, t3;

```

```

4   for (int k = 0; k < N; k++)
5   {
6       float temp1[4] = {mat[k][k], mat[k][k], mat[k][k], mat[k][k]};
7       t1 = _mm_loadu_ps(temp1);
8       int j = k + 1;
9       for (j; j < N - 3; j += 4) // 除法
10      {
11          t2 = _mm_loadu_ps(mat[k] + j);
12          t3 = _mm_div_ps(t2, t1);
13          _mm_storeu_ps(mat[k] + j, t3);
14      }
15      for (j; j < N; j++)
16          mat[k][j] /= mat[k][k];
17      mat[k][k] = 1.0;
18      for (int i = k + 1; i < N; i++) // 反减
19      {
20          float temp2[4] = {mat[i][k], mat[i][k], mat[i][k], mat[i][k]};
21          t1 = _mm_loadu_ps(temp2);
22          j = k + 1;
23          for (j; j <= N - 3; j += 4)
24          {
25              t2 = _mm_loadu_ps(mat[i] + j);
26              t3 = _mm_loadu_ps(mat[k] + j);
27              t3 = _mm_mul_ps(t1, t3);
28              t2 = _mm_sub_ps(t2, t3);
29              _mm_storeu_ps(mat[i] + j, t2);
30          }
31          for (j; j < N; j++)
32              mat[i][j] -= mat[i][k] * mat[k][j];
33          mat[i][k] = 0;
34      }
35  }
36 }

```

(五) AVX 优化加速

AVX 的算法设计思路与 SSE 一致, 唯一的区别在于指令使用不同。float→__m256, 以及其他 mm 改为 mm256 即可, 同时由于变为了 256 位, 所以对齐需要 8 个原数据。复杂度为 $O(N^3)/8$

AVX 加速

```

1   void avx_lu(float mat[][N])
2   {
3       __m256 t1, t2, t3;
4       for (int k = 0; k < N; k++)
5       {
6           for (int j = k + 1; j < N; j++)
7               mat[k][j] /= mat[k][k];
8           mat[k][k] = 1.0;

```

```

9      for (int i = k + 1; i < N; i++)
10     {
11         float temp[8] = {mat[i][k], mat[i][k], mat[i][k],
12                          mat[i][k], mat[i][k], mat[i][k], mat[i][k], mat[i][k]};
13         t1 = _mm256_loadu_ps(temp);
14         int j = k + 1;
15         for (j; j < N - 7; j += 8)
16         {
17             t2 = _mm256_loadu_ps(mat[i] + j);
18             t3 = _mm256_loadu_ps(mat[k] + j);
19             t3 = _mm256_mul_ps(t1, t3);
20             t2 = _mm256_sub_ps(t2, t3);
21             _mm256_storeu_ps(mat[i] + j, t2);
22         }
23         for (j; j < N; j++)
24             mat[i][j] -= mat[i][k] * mat[k][j];
25         mat[i][k] = 0;
26     }
27 }
28

```

同 SSE 的对齐处理一致, 只是指令上的更改。复杂度也为 $O(N^3)/8$

AVX 对齐优化

```

1 void avx_lu_aligned(float mat[][N])
2 {
3     __m256 t1, t2, t3;
4     for (int k = 0; k < N; k++)
5     {
6         for (int j = k + 1; j < N; j++)
7             mat[k][j] /= mat[k][k];
8         mat[k][k] = 1.0;
9         for (int i = k + 1; i < N; i++)
10        {
11            int j = k + 1;
12            for (j; j % 8 != 0; j++)
13                mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
14            float temp[8] = {mat[i][k], mat[i][k], mat[i][k],
15                             mat[i][k], mat[i][k], mat[i][k], mat[i][k], mat[i][k]};
16            t1 = _mm256_load_ps(temp);
17            for (j; j < N - 7; j += 8)
18            {
19                t2 = _mm256_load_ps(mat[i] + j);
20                t3 = _mm256_load_ps(mat[k] + j);
21                t3 = _mm256_mul_ps(t1, t3);
22                t2 = _mm256_sub_ps(t2, t3);
23                _mm256_store_ps(mat[i] + j, t2);
24            }
25            for (j; j < N; j++)

```

```

26         mat[i][j] -= mat[i][k] * mat[k][j];
27         mat[i][k] = 0;
28     }
29 }
30 }

```

AVX 向量化与 SSE 向量化一致,都对两部分进行了向量化。保持不对齐。复杂度也为 $O(N^3)/8$

AVX 向量化

```

1 void avx_lu_vectorized(float mat[][N])
2 {
3     __m256 t1, t2, t3;
4     for (int k = 0; k < N; k++)
5     {
6         float temp1[8] = {mat[k][k], mat[k][k], mat[k][k],
7                             mat[k][k], mat[k][k], mat[k][k], mat[k][k], mat[k][k]};
8         t1 = _mm256_loadu_ps(temp1);
9         int j = k + 1;
10        for (j; j < N - 7; j += 8)
11        {
12            t2 = _mm256_loadu_ps(mat[k] + j);
13            t3 = _mm256_div_ps(t2, t1);
14            _mm256_storeu_ps(mat[k] + j, t3);
15        }
16        for (j; j < N; j++)
17            mat[k][j] /= mat[k][k];
18        mat[k][k] = 1.0;
19        for (int i = k + 1; i < N; i++)
20        {
21            float temp2[8] = {mat[i][k], mat[i][k], mat[i][k],
22                                mat[i][k], mat[i][k], mat[i][k], mat[i][k], mat[i][k]};
23            t1 = _mm256_loadu_ps(temp2);
24            j = k + 1;
25            for (j; j < N - 7; j += 8)
26            {
27                t2 = _mm256_loadu_ps(mat[i] + j);
28                t3 = _mm256_loadu_ps(mat[k] + j);
29                t3 = _mm256_mul_ps(t1, t3);
30                t2 = _mm256_sub_ps(t2, t3);
31                _mm256_storeu_ps(mat[i] + j, t2);
32            }
33            for (j; j < N; j++)
34                mat[i][j] -= mat[i][k] * mat[k][j];
35            mat[i][k] = 0;
36        }
37    }
38 }

```


三、高精度计时性能结果比较

由于实验中可能出现种种误差,为尽量保证实验的公平性,作以下假设:

- Cache 对两种算法都是公平的,每次进行矩阵向量内积之前通过 reset 函数令初始矩阵一致,最终结果一致的情况下才进行高精度计时
- 由于 N 较小的时候测出的计算时间也较小,误差较大,故采取多次测量取均值的方法确定较合理的性能测试结果,同时保证几种算法重复次数一致,减少误差

由于运行时间与问题规模的变化趋势并不是本题的关注重点,所以本题的测试区间并不如上一次那么密集,但是仍然考虑 cache 大小的情况进行 N 的取值设定。根据全部测试数据绘制运行

N	naive	sse	sse-aligned	sse-vectorized	avx	avx-aligned	avx-vectorized
16	0.00093	0.00079	0.00089	0.0007	0.00238	0.00225	0.00146
128	0.4357	0.1709	0.2104	0.1809	0.214	0.1842	0.2061
512	30.6704	9.7099	10.002	9.2121	9.0066	9.8389	9.3193
1000	228.978	87.7108	88.0601	78.9313	77.9771	54.359	72.61
2000	1876.72	1016.18	995.169	993.608	944.825	951.351	956.05
5000	30654.6	24033.3	21380.9	26619	21326.7	22115.3	21704.9
10000	224241	134755	131855	135308	130597	124544	130546

表 1: 性能测试结果 (单位:ms)

时间以及加速比对比图,运行时间图可以看出各种方法对于朴素方法都节约了近一半的运行时间,而在 N 较小时 SSE 系列算法运行更快,而 N 较大时 (>500)AVX 系列在运行时间上优于 SSE。原因可能是 AVX 打包和解包的代价相较于 SSE 大得多,虽然加速较 SSE 显著,但是在 N 较小时没能抵消这种打包解包的影响。

加速比对比图则可以看出各个算法在加速上的趋势基本一致,最高可达 3.4,最差也比串行朴素算法快一些。对比两图可以发现 N 较小时 SSE 加速比强于 AVX,而 N 较大时 AVX 后来居上。

整体来看,对齐的效果比不对齐的效果好得多,AVX 和 SSE 的对齐算法是整体性能表现最好的,尤其是 N 较大时,这表明对其操作在并行化上助益大,取数对齐因为 cache 等是对齐的所以节省了一部分时间,但是没有拉开明显的差距可能是因为加入了更多的判断。vector 标记的是将 4-5 行除法部分也进行向量化的算法,并没有和预想中的一样得到更优解,且都在 N>2000 以后性能明显落后于未经过向量化的算法,原因可能是这一部分计算只有除法这一种,进行向量化反而增大了开销不如 8-9 行两种运算的合并并行效果好。此外,图中出现拐点的地方也与 cache 大小表现一致。

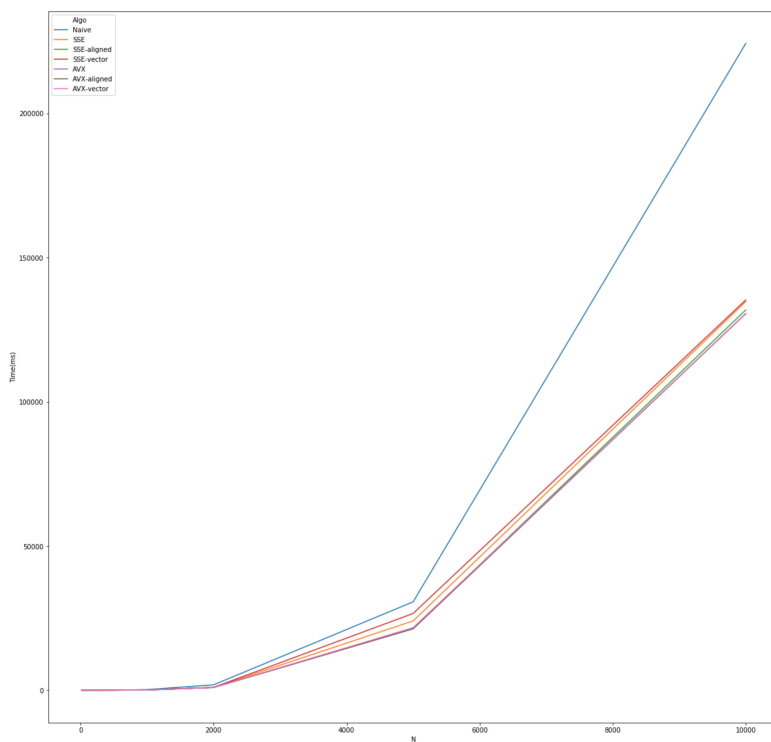


图 1: 运行时间

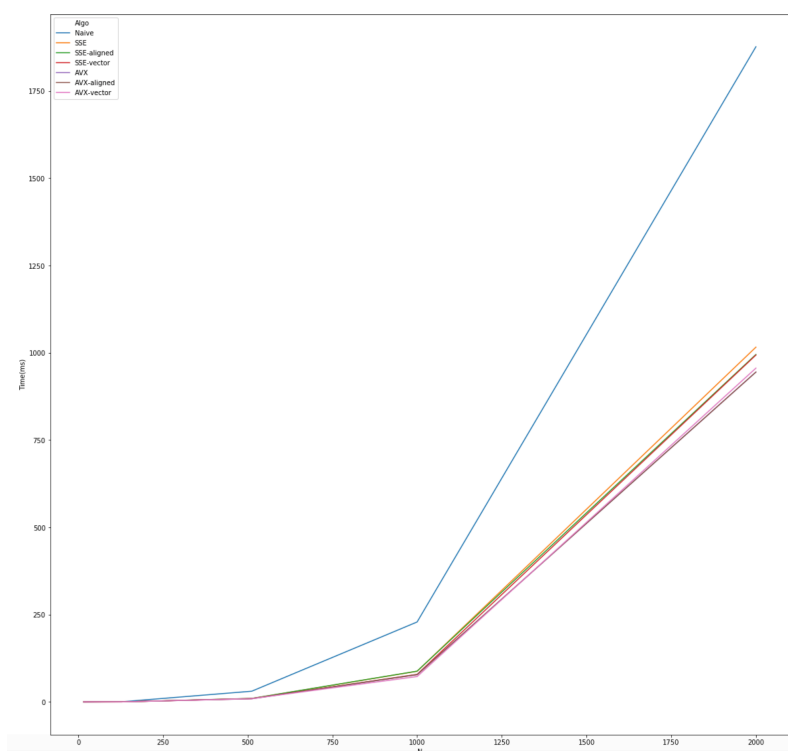


图 2: 运行时间 (1-2000)

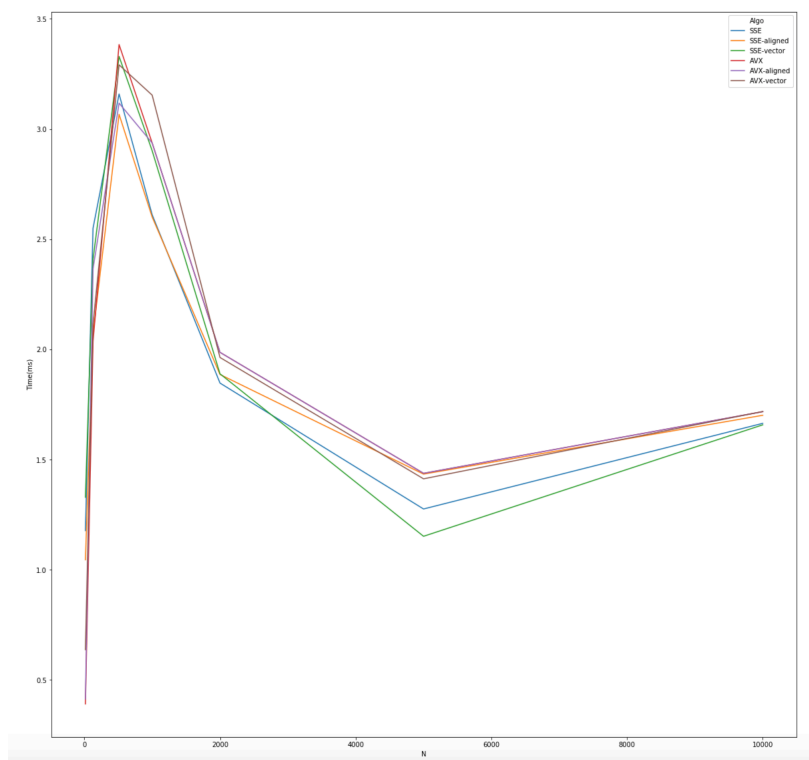


图 3: 加速比

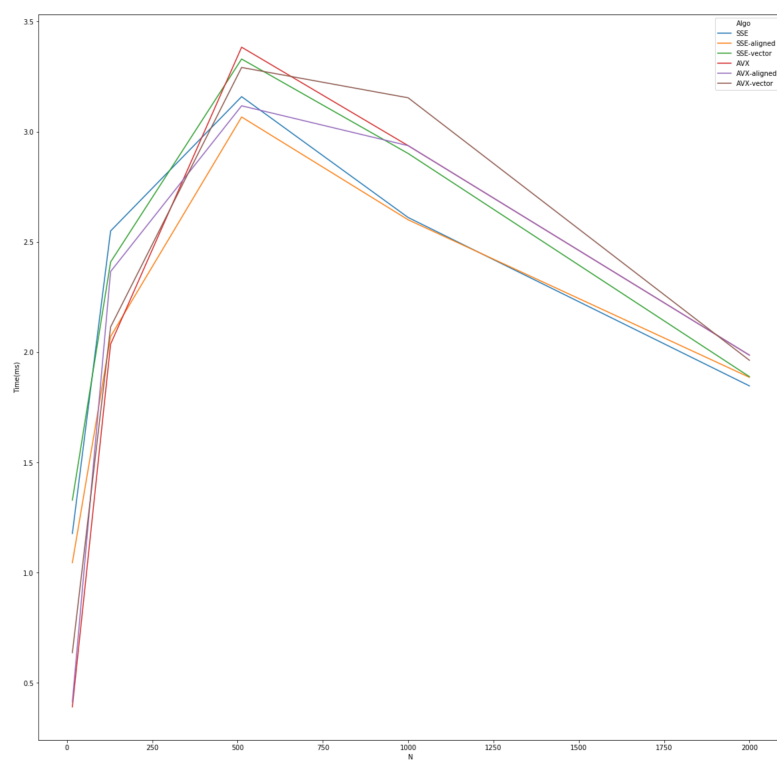
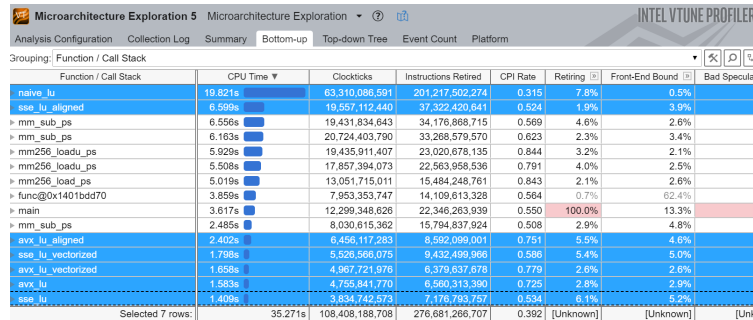


图 4: 加速比 (1-2000)

(一) 基于 Vtune 的程序性能剖析

由于这次的编程侧重点是编译后指令上的优化, 所以使用 Vtune profiling 的侧重点在微体系结构的指令数、周期数以及 CPI 等等上, 故按上次实验进行相同配置即可, 由于这次测试用例规模的变动不是重点, 所以 CPU 间隔取 0.1ms 即可。

如图为 N=2000 时的 Profiling, 其他 N 取值表现相似, 而过小和过大可能导致精确度以及结果等等表现不够好。



Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound	Bad Specul
naive_iu	19.821s	63,310,086,591	201,217,502,274	0.315	7.8%	0.5%	
sse_iu_aligned	6.599s	19,557,112,440	37,322,420,641	0.524	1.9%	3.9%	
mm_sub_ps	6.556s	19,431,834,643	34,176,868,715	0.569	4.6%	2.6%	
mm_sub_ps	6.163s	20,724,403,790	33,268,579,570	0.623	2.3%	3.4%	
mm256_loadu_ps	5.929s	19,435,911,407	23,020,678,135	0.844	3.2%	2.1%	
mm256_loadu_ps	5.508s	17,857,394,073	22,563,958,536	0.791	4.0%	2.5%	
mm256_load_ps	5.019s	13,051,715,011	15,484,248,761	0.843	2.1%	2.6%	
func@0x1401bdd70	3.859s	7,953,353,747	14,109,613,328	0.564	0.7%	62.4%	
main	3.617s	12,299,348,626	22,346,263,939	0.550	100.0%	13.3%	
mm_sub_ps	2.485s	8,030,615,362	15,794,837,924	0.508	2.9%	4.8%	
avx_iu_aligned	2.402s	6,456,117,283	8,592,099,001	0.751	5.5%	4.6%	
sse_iu_vectorized	1.798s	5,526,566,075	9,432,499,966	0.586	5.4%	5.0%	
avx_iu_vectorized	1.658s	4,967,721,976	6,379,637,678	0.779	2.6%	2.6%	
avx_iu	1.583s	4,755,841,770	6,560,313,390	0.725	2.8%	2.9%	
sse_iu	1.409s	3,834,742,573	7,176,793,757	0.534	6.1%	5.2%	
Selected 7 rows:	35.271s	108,408,188,708	276,681,266,707	0.392	[Unknown]	[Unknown]	[Unk]

图 5: N=2000

根据前面加速比对比图 N=2000 的时候, 算法加速比应该集中在 2.0 附近, 但是根据 Vtune 结果中对应函数的 CPU time, 最好的加速比可以达到 14, 最差也可以有 3, 这之中巨大的差距坑与 SSE 和 AVX 需要做的大量准备相关。而周期数和加速比呈一样的比例, 指令数也得到了相应减少, 只不过指令数并非和加速比一样的比例, 但是其减少比例与其对应周期数的减少是对应的, 确实如前文推测, 对齐所需指令数更多, AVX 所需指令数更多。从 CPI 来看反倒是朴素算法最小, 这可能与朴素算法没有复杂的指令且指令数较多有关。

四、 总结

完成这个实验首先需要对 SIMD 编程的 SSE 和 AVX 方法进行掌握, 而且最终实验结果也不能说是完全和预料相符, 这次同样要考虑到较多的方面, 但是整体上进展顺利许多, 整个实验让我较好的了解到了 SIMD 编程的本质, 以及通过显性的运行时间指令数等等数据更好的了解到 MD 是如何并行化程序并做到加速的。

参考文献

- [1] 高斯消元法 [EB/OL].<https://zh.wikipedia.org/wiki/高斯消去法>
- [2] <https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE3,AVX&expand=3931>