南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

高斯消去法的 SIMD 并行化

姓名：姚知言

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2024 年 4 月 28 日

摘要

在本实验中针对普通高斯消去和特殊高斯消去, 掌握原理并编写串行算法。在此基础上, 以 SSE，NEON 等不同方式进行 SIMD 优化，经过性能测试和 Vtune 分析后总结自己对 SIMD 并行实现的理解。

关键字：并行；高斯消去；特殊高斯消去；NEON；SSE；

目录

一、 问题描述

在本次实验将在 SIMD 编程下探究普通高斯消去算法以及特殊高斯消去算法的优化程度, 通过性能测试以及 profiling 探究 SIMD 编程的原理以及其对高斯消去算法的优化情况，并加以总结。该实验同时也是期末大作业的一个分支。

二、 普通高斯消去算法

(一) 算法设计

1. 总体思路

高斯消去法主要分为消去和回代两个部分，如图1。消去部分复杂度较高，为 $O(n^3)$，回代部分也有 $O(n^2)$ 的复杂度。

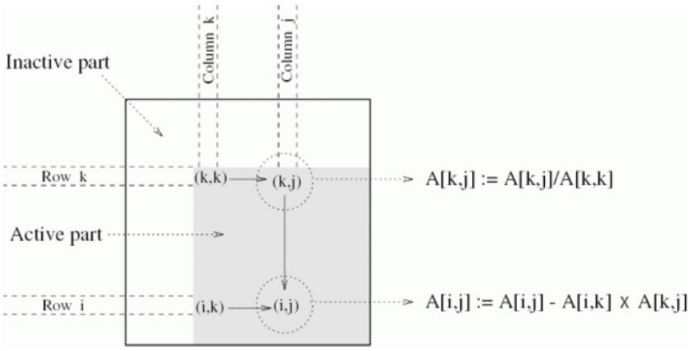


图 1: 高斯消元法示意图

在消去过程中进行第 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，全部结束后结果如图2所示，然后从最后一行向上回代。

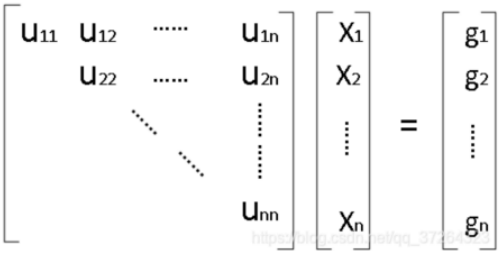


图 2: 消去过程结果示意图

回代过程从矩阵的最后一行开始向上回代，对于第 i 行，利用已知的 x_{i+1},x_{i+2},..., x_n 计算出 x_i。

在本实验中，选取 n=16,32,64,128,256,500,750,1000,1500,2000 十个矩阵规模取值进行实验。考虑到在 n 规模较小的时候运行时间较短，并进行多次实验取平均值，每个 n 的遍历次数 t 计为 5 和 $1000000/n^2$。

2. SIMD 并行化优化

针对循环嵌套部分着重进行并行化优化。主要针对 NEON 优化算法进行研究，对各部分循环语句分别进行优化比对。同时对 n^3 部分优化兼有 SSE/AVX 优化算法，对比各优化算法效率。

(二) 编程实现

1. 总体框架

为避免程序出现过于复杂的结果，并增加数值的随机性，引入 srand(time(NULL)) 函数，并对部分元素加以约束。

矩阵重置函数

```
void reset(int n){
    srand(time(NULL));
    for(int i=0;i<n;i++){
        b[i]=rand();
        for(int j=0;j<i;j++)
            A[i][j]=0;
        A[i][i]=1.0;
        for(int j=i+1;j<n;j++)
            A[i][j]=rand();
    }
    for(int k=0;k<n;k++)
        for(int i=k+1;i<n;i++)
            for(int j=0;j<n;j++)
                A[i][j]+=A[k][j];
}
```

计时函数选用 chrono 库实现。

计时和调用算法框架

```
void op(void (*method)(int),int n,int t){
    cout<<method<<" n="<<n<<" t="<<t;
    auto start=chrono::steady_clock::now();
    for(int i=0;i<t;i++)method(n);
    auto finish=chrono::steady_clock::now();
    auto duration=chrono::duration_cast<chrono::milliseconds>(finish-start);
    double pertime=duration.count()*double(1.0)/t;
    cout<<" pertime="<<pertime<<"ms\n";
}
```

2. 平凡算法

平凡算法

```
void common(int n){
    for(int k=0;k<n;k++) {
        for (int i=k+1;i<n;i++){
```

```
            float factor=A[i][k]/A[k][k];
                for (int j=k+1;j<n;j++) A[i][j]-=factor*A[k][j];
                b[i]-=factor*b[k];
        }
    }
    x[n-1]=b[n-1]/A[n-1][n-1];
    for(int i=n-2;i>=0;i--){
        float sum=b[i];
        for(int j=i+1;j<n;j++)
            sum-=A[i][j]*x[j];
        x[i]=sum/A[i][i];
    }
}
```

3. NEON 优化算法

NEON 优化算法 1 主要针对 n^3 部分，即平凡算法第 4-5 行的内容进行优化如下。(对应实验部分 NEON 组，实验结果见性能测试1、2小节)

NEON 优化算法 1

```
float factor1=A[i][k]/A[k][k];
for(int j=n-1;;j--){
    if((j-k)%4)A[i][j]-=factor1*A[k][j];
    else break;
}
float32x4_t factor=vmovq_n_f32(factor1);
for(int j=k+1;j<=n-4;j+=4){
    float32x4_t cal=vld1q_f32(A[k]+j);
    float32x4_t sum4=vmulq_f32(cal,factor);
    cal=vld1q_f32(A[i]+j);
    cal=vsubq_f32(cal,sum4);
    vst1q_f32(A[i]+j,cal);
}
```

NEON 优化算法 2 在 NEON 优化算法 1 的基础上，将平凡算法 1 的第 4 行 factor 计算进行 NEON 优化。(对应实验部分 NEON01 组，实验结果见性能测试2小节)

NEON 优化算法 2

```
for (int i=k+1;i<=n-4;i+=4){
    float32x4_t factor4=vld1q_f32(A[i]+k);
    vsetq_lane_f32(A[i+1][k],factor4,1);
    vsetq_lane_f32(A[i+2][k],factor4,2);
    vsetq_lane_f32(A[i+3][k],factor4,3);
    float32x4_t akk=vmovq_n_f32(A[k][k]);
    factor4=vdivq_f32(factor4,akk);
    for(int j=n-1;;j--){
        if((j-k)%4)A[i][j]-=vgetq_lane_f32(factor4,0)*A[k][j];
        else break;
```

```
    }
    //以下仅以第一行为例，后续三行类似，略去
    float32x4_t factor=vmovq_n_f32(vgetq_lane_f32(factor4,0));
    for(int j=k+1;j<=n-4;j+=4){
        float32x4_t cal=vld1q_f32(A[k]+j);
        float32x4_t sum4=vmulq_f32(cal,factor);
        cal=vld1q_f32(A[i]+j);
        cal=vsubq_f32(cal,sum4);
        vst1q_f32(A[i]+j,cal);
    }
    b[i]-=vgetq_lane_f32(factor4,0)*b[k];
}
//在此之后针对不能被4整除的部分进行计算，实现思路类似优化算法1，略去，可于
    github仓库中查看。
```

NEON 优化算法 3 主要针对回代过程，即平凡算法 12-13 行进行 NEON 优化如下。(对应实验部分 NEON2 组，实验结果见性能测试2小节)

NEON 优化算法 3

```
float32x4_t sum4=vmovq_n_f32(0);
for(int j=i+1;j<n;j+=4){
    float32x4_t a4=vld1q_f32(A[i]+j);
    float32x4_t x4=vld1q_f32(x+j);
    a4=vmulq_f32(a4,x4);
    sum4=vaddq_f32(sum4,a4);
}
float32x2_t suml2=vget_low_f32(sum4);
float32x2_t sumh2=vget_high_f32(sum4);
suml2=vpadd_f32(suml2,sumh2);
sum-=vpadds_f32(suml2);
for(int j=n-1;;j--){
    if((j-i)%4)sum-=A[i][j]*x[j];
    else break;
}
```

NEON 优化算法 4 为优化算法 1 和优化算法 3 的结合，对消元和回代过程分别进行优化。(对应实验部分 NEON02 组，实验结果见性能测试2小节)

4. SSE/AVX 优化算法

SSE/AVX 优化算法主要针对 n^3 部分，即平凡算法第 4-5 行的内容进行优化如下。其中，SSE 为 4 路并行算法，AVX 为 8 路并行算法。(分别对应实验部分 SSE 组、AVX 组，实验结果见性能测试1小节)

SSE 优化算法

```
float factor1=A[i][k]/A[k][k];
for(int j=n-1;;j--){
    if((j-k)%4)A[i][j]-=factor1*A[k][j];
    else break;
```

```
}
__m128 factor=_mm_set_ps(factor1,factor1,factor1,factor1);
for(int j=k+1;j<=n-4;j+=4){
    __m128 cal=_mm_loadu_ps(A[k]+j);
    __m128 sum4=_mm_mul_ps(cal,factor);
    cal=_mm_loadu_ps(A[i]+j);
    cal=_mm_sub_ps(cal,sum4);
    _mm_storeu_ps(A[i]+j,cal);
}
```

AVX 优化算法

```
float factor1=A[i][k]/A[k][k];
for(int j=n-1;;j--){
    if((j-k)%8)A[i][j]-=factor1*A[k][j];
    else break;
}
__m256 factor=_mm256_set_ps(factor1,factor1,factor1,factor1,factor1,factor1,
    factor1,factor1);
for(int j=k+1;j<=n-8;j+=8){
    __m256 cal=_mm256_loadu_ps(A[k]+j);
    __m256 sum4=_mm256_mul_ps(cal,factor);
    cal=_mm256_loadu_ps(A[i]+j);
    cal=_mm256_sub_ps(cal,sum4);
    _mm256_storeu_ps(A[i]+j,cal);
}
```

(三) 性能测试

本次实验主要依托 ARM 平台华为鲲鹏服务器，兼有与 x86 平台的对比试验。

x86 平台环境配置：x86-64 平台，Intel(R) Core(TM) i5-10200H，windows 11 系统，G++ 编译器，C++ 17 GNU，O2 优化。

ARM 平台环境配置：华为鲲鹏服务器，G++ 编译器，02 优化。

1. 针对 n^3 部分优化：x86 平台和 ARM 平台实现

在此部分，使用的参数为-march=native。其中 SSE/AVX 在 x86 平台下实验，NEON 在 ARM 平台下实现。并在各平台测试平凡算法作为对照。实验结果如表6所示。

2. NEON 优化算法对各过程优化

在此部分使用 ARM 平台实现，使用的参数为-march=armv8-a。结果如表2所示。

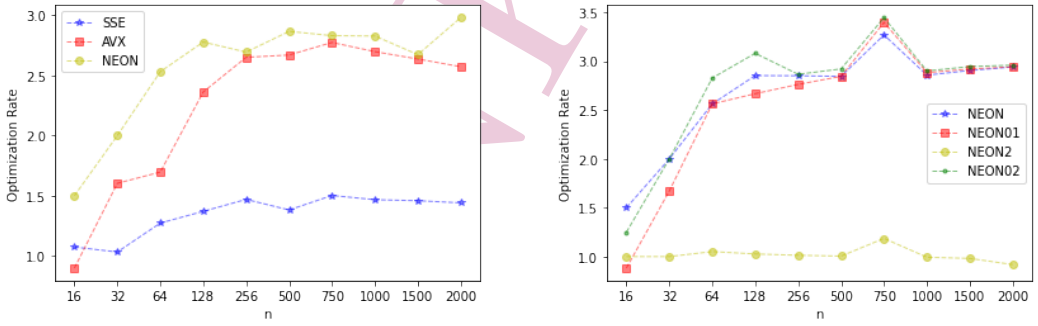
矩阵大小 n	16	32	64	128	256	500	750	1000	1500	2000
实验次数 t	3906	976	244	61	15	5	5	5	5	5
平凡 $_{x86}$	0.0044	0.0327	0.229	1.819	14.5	103.6	356.8	824	2845	6695
SSE	0.0041	0.0317	0.180	1.328	9.87	75	237.6	561.8	1951	4648
AVX	0.0049	0.0204	0.135	0.770	5.47	38.8	128.6	305.4	1079	2602
平凡 $_{ARM}$	0.0015	0.0102	0.081	0.639	5.20	51.6	184.6	411.2	1334.6	3062
NEON	0.0010	0.0051	0.032	0.230	1.93	18	65.2	145.4	499.2	1027

表 1: 各类型算法平均运行时间 (单位:ms)

矩阵大小 n	16	32	64	128	256	500	750	1000	1500	2000
实验次数 t	3906	976	244	61	15	5	5	5	5	5
平凡	0.0015	0.0102	0.082	0.656	5.33	43.8	190.2	386.8	1320.2	3017.2
NEON	0.0010	0.0051	0.032	0.230	1.87	15.4	58.2	135.4	454.6	1026.2
NEON01	0.0017	0.0061	0.032	0.246	1.93	15.4	56	134.2	452.2	1025.6
NEON2	0.0015	0.0102	0.078	0.639	5.26	43.6	160.4	389.4	1345.8	3286.8
NEON02	0.0012	0.0051	0.029	0.213	1.86	15	55.2	133.4	448.4	1019.8

表 2: 各部分优化算法平均运行时间 (单位:ms)

3. 总结



第1节各算法优化比 第2节各算法优化比

图 3: 各算法优化比（与相同环境下平凡算法比较）

(四) profiling

在 x86 环境 (环境说明同上节，-march=native)，将先前的十个测试点一起测试，检测 SSE、AVX 优化算法的效率，结果如表3所示。

(五) 结果分析

通过对1、2两节的对比分析,我们可以发现在 ARM 环境下,-march=native 和-march=armv8-a 性能差距不大，仅在矩阵大小较大的情况下-march=armv8-a 性能略好。

通过图3可以看出，在 n 较小的时候，SIMD 优化算法优势并不明显，甚至在部分情况下出

	平凡算法	SSE 优化算法	AVX 优化算法
CPU Time/s	11.495	6.304	5.160
Instructions Retired	1.52e11	3.36e10	1.78e10
CPI rate	0.303	0.728	1.083

表 3: Vtune 分析结果

现优化比小于 1（即优化算法运行时间比平凡算法更长的情况）。但随着 n 的增大，大多优化算法均有着明显的优化表现。

通过图3左图可以看出，SSE 的 4 路并行算法在 n 较大的时候，优化比稳定在约 1.3-1.4 之间，优化效果比较一般，AVX 的 8 路并行算法在 n 较大的时候，优化比稳定在约 2.5-2.7 之间，优化效果尚可，但考虑到其消耗并不如 NEON 并行算法。NEON 的 4 路并行算法在 n 较大的时候取得了接近 3.0 的优化比，优化效果非常可观。当然，这部分差异也有可能是处理器能效的差别造成的。

通过图3右图可以看出，NEON2 优化算法的优化比稳定在 1.0 左右，即使在 n 较大的时候也存在优化比小于 1 的情况。这种情况是由于优化的部分并不是核心的 n^3 循环，而调用 NEON 库的耗能过大导致的。然而这并不是说这部分不需要优化，对比 NEON02 和 NEON、NEON01 的优化比，可以看到 NEON02 的优化比能够稳定略高于另外两种算法。

NEON01 算法的优化比仅仅略高于 NEON 算法，不如 NEON02 算法。猜测这是因为外层循环要读取的数据不是连续存储的，使用向量优化效果并不显著。

通过 Vtune 对 SSE/AVX 的分析，我们可以看出 SSE 和 AVX 优化算法对于指令执行数量有非常大的优化，SSE 相比平凡算法优化了 4 倍，AVX 相比 SSE 算法再优化了接近 2 倍。然而，这却导致了 CPI rate 的快速上升，这也与上一部分所得优化效率没有那么高相对应。

三、 特殊高斯消去算法

(一) 算法设计

1. 总体思路

特殊高斯消去计算来自一个实际的密码学问题–Gröbner 基的计算。

运算均为有限域 GF(2) 上的运算，即矩阵元素的值只可能是 0 或 1。1. 其加法运算实际上为异或运算:0+0=0、0+1=1、1+0=1、1+1=0 由于异或运算的逆运算为自身，因此减法也是异或运算。乘法运算 0*0=0、0*1=0、1*0=0、1*1=0。因此，高斯消去过程中实际上只有异或运算–从一行中消去另一行的运算退化为减法。

考虑单次完成数据读取的简单情况，此特殊的高斯消去计算过程 (串行算法) 如下:

对于每个被消元行，检查其首项，如有对应消元子则将其减去 (异或) 对应消元子，重复此过程直至其变为空行 (全 0 向量) 或首项无对应消元子。

如果某个被消元行变为空行，则将其丢弃，不再参与后续消去计算；如其首项被没有对应消元子，则将它“升格”为消元子，在后续消去计算中将以消元子身份而不再以被消元行的身份参与;

重复上述过程，消元子和被消元行共同组成结果矩阵–可能存在很多空行。

2. SIMD 并行化优化

主要针对最内层循环进行并行化优化以得到较好的优化比。分别采用 SSE 优化算法以及 NEON 优化算法在 x86 和 ARM 平台中测试。

(二) 编程实现

1. 总体框架

为节约内存空间，采取用 int 整型的每一位表示矩阵的每一列的形式，根据题目的 io 要求定义输入函数及输出函数如下。

变量定义及读函数

```
const int xsize=130,xlen=xsize/32+1,maxrow=10;//数值以样例1为例，xsize设置为矩阵列数，maxrow略大于被消元行数即可
int row[maxrow][xlen];int xyz[xsize][xlen];
bool is[xsize];int nrow;
void read(){
    fstream xyzfile("消元子.txt", ios::in | ios::out);
    string line;
    while (getline(xyzfile, line)) {
        istringstream iss(line);int value;bool first=true;
        while (iss >> value) {
            if(first){nrow=value;is[value]=true;first=false;}
            (xyz[nrow][value/32])|=(1<<value%32);
        }
    }
    xyzfile.close();
    fstream bxyfile("被消元行.txt", ios::in | ios::out);nrow=0;
    while (getline(bxyfile, line)) {
        istringstream iss(line);int value;
        while (iss >> value) {(row[nrow][value/32])|=(1<<value%32);}
        nrow++;
    }
    bxyfile.close();
}
```

写函数

```
void write(){
    ofstream ansfile("ans.txt",ios::app);
    ansfile<<"_____\n";
    for(int i=xsize-1;i>0;i--){
        if(is[i]){
            for(int j=xlen-1;j>=0;j--){
                for(int u=31;u>=0;u--){
                    if((xyz[i][j])&(1<<u))
                        ansfile<<u+32*j<<'␣';
                }
            }
```

```
            ansfile<<'\n';
        }
    }
}
```

使用 chrono 库，定义封装及计时函数如下。

封装及计时函数

```
void op(void(*method)(),int t){
    double tt=0;auto start = chrono::steady_clock::now();
    for(int i=0;i<t;i++) {
        memset(row, 0, sizeof(row));
        memset(xyz, 0, sizeof(xyz));
        memset(is, 0, sizeof(is));
        read();method();write();
    }
    auto finish = chrono::steady_clock::now();
    auto duration = chrono::duration_cast<chrono::milliseconds>(finish -
        start);
    tt=(duration.count() * double(1.0))/t;cout<<tt<<"ms\n";
}
```

2. 平凡算法

根据对特殊高斯消元法的分析，得到平凡算法如下。

平凡算法

```
void common(){
    for(int i=0;i<nrow;i++){
        bool isend=false;
        for(int j=xlen-1;j>=0 && !isend;j--){
            while(true){
                if(!row[i][j])break;int f;
                for(int u=31;u>=0;u--){
                    if((row[i][j])&(1<<u)){
                        f=u;break;
                    }
                }
                f+=32*j;
                if(!is[f]){
                    for(int k=0;k<xlen;k++)
                        xyz[f][k]=row[i][k];
                    is[f]=true;
                    isend=true;
                }
                else{
                    for(int k=0;k<xlen;k++)
                        row[i][k]^=xyz[f][k];
```

```
                }
            }
        }
    }
}
```

3. SSE 优化算法

主要针对平凡算法的第 14-15 行和 20-21 行两个内层循环进行优化如下。

SSE 优化算法

```
//14-15行优化如下
int k=0;
for(;k<=xlen-4;k+=4) {
    __m128i temp = _mm_loadu_si128((__m128i *) (row[i] + k));
    _mm_storeu_si128((__m128i *) (xyz[f] + k), temp);
}
for(;k<xlen;k++)xyz[f][k]=row[i][k];
//20-21行优化如下
int k=0;
for(;k<=xlen-4;k+=4) {
    __m128i tx = _mm_loadu_si128((__m128i *) (xyz[f] + k));
    __m128i tr = _mm_loadu_si128((__m128i *) (row[i] + k));
    tr= _mm_xor_si128(tx, tr);
    _mm_storeu_si128((__m128i *) (row[i] + k), tr);
}
for(;k<xlen;k++)row[i][k]^=xyz[f][k];
```

4. NEON 优化算法

与 SSE 优化算法类似，针对平凡算法的第 14-15 行和 20-21 行两个内层循环进行优化如下。

NEON 优化算法

```
//14-15行优化如下
int k=0;
for(;k<=xlen-4;k+=4) {
    int32x4_t temp = vld1q_s32(row[i] + k);
    vst1q_s32(xyz[f] + k, temp);
}
for(;k<xlen;k++)xyz[f][k]=row[i][k];
//20-21行优化如下
int k=0;
for(;k<=xlen-4;k+=4) {
    int32x4_t tx = vld1q_s32(xyz[f] + k);
    int32x4_t tr = vld1q_s32(row[i] + k);
    tr= veorq_s32(tx, tr);
    vst1q_s32(row[i] + k, tr);
}
```

```
16 for (;k<xlen;k++)row[i][k]^=xyz[f][k];
```

(三) 性能测试

通过检查输出文件，各算法的正确性已经得到检验。

1. 样例数据集说明

样例编号	1	2	3	4	5	6	7
矩阵列数	130	254	562	1011	2362	3799	8399
非零消元子数	22	106	170	539	1226	2759	6375
被消元行数	8	53	53	263	453	1953	4535

表 4: 样例数据集说明

2. x86 平台：SSE 优化算法性能测试

x86 平台环境配置：x86-64 平台，Intel(R) Core(TM) i5-10200H，windows 11 系统，G++ 编译器，C++ 17 GNU，O2 优化，-march=native。

样例编号	1	2	3	4	5	6	7
实验次数 t	5000	1000	100	100	20	10	10
平凡 $_{x86}$	0.399	1.579	2.99	16.18	70.5	422.6	3976.7
SSE	0.386	1.502	2.77	15.58	64.55	343.7	2776.7

表 5: x86 平台算法平均运行时间 (单位:ms)

3. ARM 平台：NEON 优化算法性能测试

ARM 平台环境配置：华为鲲鹏服务器，G++ 编译器，02 优化，-march=native。

样例编号	1	2	3	4	5	6	7
实验次数 t	5000	1000	100	100	100	10	5
平凡 $_{ARM}$	0.1552	0.897	1.85	11.97	56.49	505.9	5188.4
NEON	0.1548	0.893	1.83	11.09	49.11	394.1	3453.2

表 6: ARM 平台算法平均运行时间 (单位:ms)

4. 总结

从图4中可以看出，随着矩阵列数和被消元行的增加，SSE 优化算法和 NEON 优化算法的优化比也在逐步增加，在矩阵有数千行下优化比可以达到接近 1.5。但在此情况下对于系统的空间和时间负载已经较大，故没有对更大的样例集进行试验。

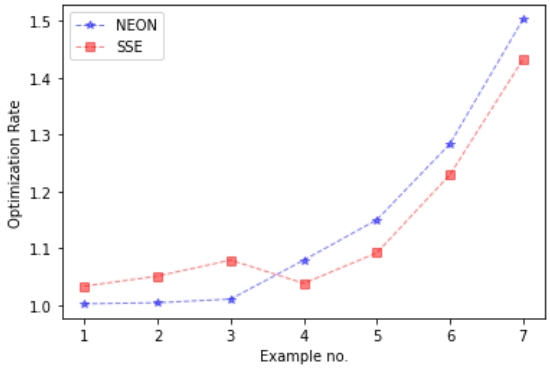


图 4: 各算法优化比

(四) profiling

在 x86 平台 (环境说明同上节)，使用 Vtune 对 SSE 优化算法进行效率分析，结果如表7所示。(选用样例数据集 7，单次循环)

	平凡算法	SSE 优化算法
CPU Time/s	4.359	2.863
Instructions Retired	3.80e10	2.29e10
CPI rate	0.409	0.447

表 7: Vtune 分析结果

(五) 结果分析

随样例规模增大，实验时间快速增大，这与算法较大的时间复杂度有关。但各优化算法的优化比也在逐步增加。然而，优化比在数据规模较大的情况下，仍不够显著。这可能是因为在规模较大的时候，IO 的时间消耗也很大，而本实验在计算过程中计算了全流程的时间，却仅仅优化了中间的计算部分。同时，我们可以看到在各组实验中前期 x86 平台效率较低，后期 ARM 平台效率较低。这可能与当时电脑环境相关，也与鲲鹏服务器负载相关。然而，对于每一组实验的平凡算法运行时间和优化算法运行时间都是在一次执行中完成的，因此该优化比的计算仍然较为可靠。

根据表7可以看出，优化算法的 CPI rate 略高于平凡算法，但 Instructions Retired 完成的指令数量远低于平凡算法，从而得到更小的 CPU Time 结果。同时，减少的指令数并没有那么多也和先前猜测的 IO 开支对应。

四、 总结

在本次实验中，对期末选题高斯消元法和特殊高斯消元法有了更深刻的认识，成功完成了对两种算法的平凡算法的编写以及优化算法的改进。分别在 x86 平台和 ARM 平台针对 SSE/AVX 优化算法和 neon 优化算法进行了设计与分析。通过对不同平台，不同优化环节，不同优化算法之间的对比，使用 Vtune 进行分析，对 SIMD 编程有了进一步的认识。这次实验也是期末实验报告中的重要实验环节。相关完整代码可在GitHub 仓库中查看。