

SIMD 编程实验

——以高斯消去为例

杜忱莹

2021 年 4 月

安祺

2022 年 3 月

目录

1 实验介绍	3
1.1 实验选题	3
1.2 实验要求	3
1.2.1 默认选题基本要求（最高获得 80% 分数）	3
1.2.2 默认选题进阶要求（最高可获得剩余 20% 分数）	3
1.2.3 自主选题	4
1.2.4 实验报告	4
2 实验设计指导	4
2.1 普通高斯消去算法	4
2.1.1 算法分析	4
2.1.2 算法设计与编程	5
2.1.3 NEON 的 C/C++ 编程	10
2.1.4 SSE/AVX 的 C/C++ 编程	12
3 程序编译及运行	13
4 使用 VTune 等工具剖析程序性能	14

1 实验介绍

1.1 实验选题

1. 默认选题：高斯消去法的 SIMD 并行化。
2. 鼓励自主选题，与期末研究报告结合：在期末研究报告大的研究课题中，选取适合的子问题（如某步关键运算）进行 SIMD 并行实验，这部分工作未来可作为期末研究报告的一部分。
3. 自选题目难度至少与默认选题相当，且适合 SIMD 并行化。期末研究报告是两人小组合作方式的话，本次作业可独立完成、也可小组合作完成。无论选择哪种方式，总工作量应是单人的两倍，且两人应完成不同工作内容——即对期末研究报告的两个子问题分别进行 SIMD 并行化、而不能是“编程、实验、撰写报告”这样的分工。选择小组合作方式的话，在实验报告中应描述清楚两人的分工。
4. 自主选题应在研究报告中首先简要描述期末研究报告的选题大方向，然后详细描述本次 SIMD 编程实验的选题，接下来才是算法设计、实现、实验和结果分析等内容。

1.2 实验要求

1.2.1 默认选题基本要求（最高获得 80% 分数）

ARM 平台上普通高斯消去计算的基础 SIMD 并行化实验，包括设计 Neon 算法、编程实现、进行实验，讨论一些基本的算法/编程策略对性能的影响，如对齐与不对齐、选择对串行算法的不同部分（4-6 行除法、8-13 行消去）进行向量化等，实验中应测试不同问题规模下串行算法/并行算法的性能、不同算法/编程策略对性能的影响等。

1.2.2 默认选题进阶要求（最高可获得剩余 20% 分数）

除普通高斯消去外，还对一种特殊的高斯消去计算进行 SIMD 并行化实验，在其中还可探讨不同平台（如 **x86**）、不同指令集（SSE、AVX、AVX-512 等）以及体系结构相关优化（如 cache 优化）等实验。

1.2.3 自主选题

自主选题视难度和工作量与默认题目对等评分。

1.2.4 实验报告

撰写研究报告（**问题描述**（特别是对自主选题，首先简要描述期末研究报告的大问题，然后具体描述本次 SIMD 编程实验涉及的子问题）、**SIMD 算法设计**（最好有复杂性分析）与实现、**实验及结果分析**），符合科技论文写作规范，附 **Git 项目链接**。

2 实验设计指导

2.1 普通高斯消去算法

2.1.1 算法分析

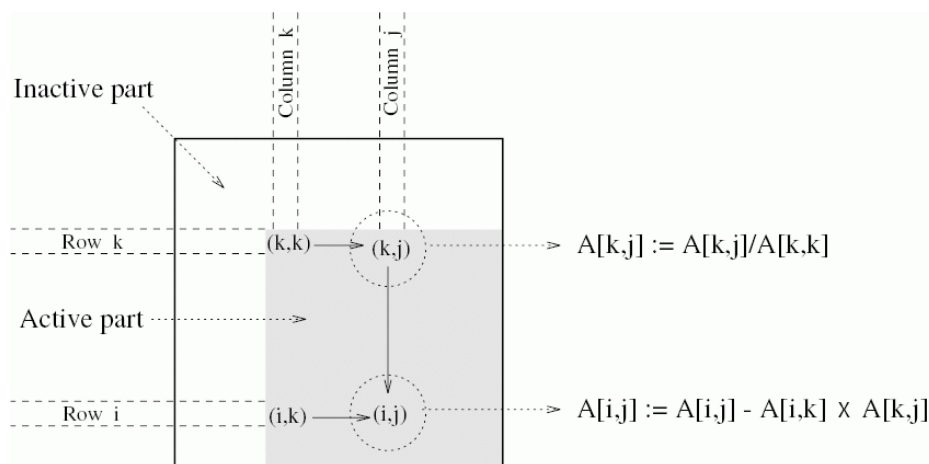


图 2.1: 高斯消去法示意图

高斯消去的计算模式如图2.1.1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，串行算法如下面伪代码所示。

```

1 procedure LU (A)
2 begin
3   for k := 1 to n do

```

```
4   for j := k+1 to n do
5       A[k, j] := A[k, j]/A[k, k];
6   endfor;
7   A[k, k] := 1.0;
8   for i := k + 1 to n do
9       for j := k + 1 to n do
10          A[i, j] := A[i, j] - A[i, k]×A[k, j];
11      endfor;
12  A[i, k] := 0;
13  endfor;
14 endfor;
15 end LU
```

观察高斯消去算法，注意到伪代码第 4, 5 行第一个内嵌循环中的 $A[k, j] := A[k, j]/A[k, k]$ 以及伪代码第 8, 9, 10 行双层 for 循环中的 $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$ 都是可以进行向量化的循环。可以通过 SIMD 扩展指令对这两步进行并行优化。

2.1.2 算法设计与编程

下面给出一个使用 SIMD Intrinsic 函数对普通高斯消元进行向量化的伪代码，见算法1，同学们基本上可以逐句翻译为 Neon/SSE/AVX(-512) 高斯消元函数，完成 ARM 和 X86 平台的基本实验（当然要补齐测试样例生成、时间测量等辅助程序）。篇幅所限，这里只给出了 4 路向量化的算法，对 AVX/AVX-512，同学们需将其改为 8 路/16 路向量化算法。此外，这里只给出了支持内存不对齐的 SIMD 访存操作。注意，在 x86 平台上还要考虑是否使用了支持内存不对齐的访存指令，使用内存对齐的访存指令需要对起始下标进行调整。

在设计算法时，我们需要注意以下要点：

1. 测试用例的确定。

对本题而言，运行时间与问题规模的变化趋势不是关注重点，但是测试规模较小时，可能会出现并行算法比串行算法还要耗时的情况。此外，类似之前的作业，我们同样可以考虑 cache 大小等系统参数来设计实验中的不同问题规模。

此外，方便起见，可以人为选取一段不会出现运算错误的测试用例，否则计算结果可能会出现 Nan 或无穷，可参考如下代码。

Listing 1: 测试用例生成

Algorithm 1: SIMD Intrinsic 版本的普通高斯消元

Data: 系数矩阵 $A[n,n]$
Result: 上三角矩阵 $A[n,n]$

```

1 for  $k = 0$  to  $n-1$  do
2    $vt \leftarrow \text{dupTo4Float}(A[k,k]);$ 
3   for  $j = k + 1; j + 4 \leq n; j += 4$  do
4      $va \leftarrow \text{load4FloatFrom}(\&A[k,j]);$  // 将四个单精度浮点数从
        内存加载到向量寄存器
5      $va \leftarrow va/vt;$  // 这里是向量对位相除
6      $\text{store4FloatTo}(\&A[k,j],va);$  // 将四个单精度浮点数从向量
        寄存器存储到内存
7   for  $j$  in 剩余所有下标 do
8      $A[k,j] = A[k,j]/A[k,k];$  // 该行结尾处有几个元素还未计算
9    $A[k,k] \leftarrow 1.0;$ 
10  for  $i \leftarrow k+1$  to  $n-1$  do
11     $vaik \leftarrow \text{dupToVector4}(A[i,k]);$ 
12    for  $j = k + 1; j + 4 \leq n; j += 4$  do
13       $vakj \leftarrow \text{load4FloatFrom}(\&A[k,j]);$ 
14       $vaij \leftarrow \text{load4FloatFrom}(\&A[i,j]);$ 
15       $vx \leftarrow vakj * vaik;$ 
16       $vaij \leftarrow vaij - vx;$ 
17       $\text{store4FloatTo}(\&A[i,j],vaij);$ 
18    for  $j$  in 剩余所有下标 do
19       $A[i,j] \leftarrow A[i,j] - A[k,j] * A[i,k];$ 
20     $A[i,k] \leftarrow 0;$ 

```

```

1 float m[N][N];
2 void m_reset()
3 {
4     for(int i=0;i<N;i++)
5     {
6         for(int j=0;j<N;j++)
7             m[i][j]=0;
8         m[i][i]=1.0;
9         for(int j=i+1;j<N;j++)
10            m[i][j]=rand();
11     }
12     for(int k=0;k<N;k++)
13         for(int i=k+1;i<N;i++)
14             for(int j=0;j<N;j++)
15                 m[i][j]+=m[k][j];
16 }

```

2. 设计对齐与不对齐算法策略时，注意到高斯消去计算过程中，第 k 步消去的起始元素 k 是变化的，从而导致距 16 字节边界的偏移是变化的。对于 ARM 平台的实验，在 AArch64 NEON 访存指令默认支持未对齐内存访问；在 NEON 汇编代码中可以指定对齐比特位数，这里可以进一步探究对齐 NEON 指令与未对齐性能差异，更多信息可参考官方手册对应内容(超链接)：

- 《NEON Programmer's Guide》中关于内存对齐访存指令语法和性能影响的说明

对于 x86 平台的实验，如果设计不对齐的算法策略，直接使用 `_mm_loadu_ps` 即可。如果设计对齐算法使用 `_mm_load_ps` 时，我们可以调整算法，先串行处理到对齐边界，然后进行 SIMD 的计算。可对比两种方法的性能。C++ 中数组的初始地址一般为 16 字节对齐，所以只要确保每次加载数据 $A[i:i+3]$ 中 i 为 4 的倍数即可，大家如果不确定地址是否对齐，可以直接将地址打印出来对比。同理当进行 AVX 算法设计时应该注意是否 32 字节对齐问题。还可查阅资料，不同平台和编译器下一般都有指定对齐方式的动态内存分配函数，可采用这种方式确保分配的内存起始地址是对齐的。

例如 c11 标准中可以使用 `aligned_alloc` 进行对齐内存分配。

```

1 //函数原型
2 void *aligned_alloc( size_t alignment, size_t size );
3
4 //例子

```

```
5 int *p2 = aligned_alloc(1024, 1024*sizeof *p2);
6 printf("1024-byte aligned addr: %p\n", (void*)p2);
7 free(p2);
```

3. 对不同部分的优化可进行对比实验。

高斯消去法中有两个部分可以进行向量化，我们可以对比一下这两个部分（一个二重循环、一个三重循环）进行 SSE 优化对程序速度的影响。

4. 并行计算结果的误差处理。

并行计算由于重排了指令执行顺序，加上计算机表示浮点数是有误差的，可能导致即使数学上看是完全等价的，但并行计算结果与串行计算结果不一致。这不是算法问题，而是计算机表示、计算浮点数的误差导致，一种策略是允许一定误差，比如 $< 10e^{-6}$ 就行；另外一种策略，可在程序中加入一些数学上的处理，在运算过程中进行调整，来减小误差。我们用以下两个程序来展示。

（1）两个数相除再相乘：

```
1 float a = 1.0;
2 float b = 3.0;
3 for (int i = 0; i < N; i++)
4 {
5     a /= b;
6 }
7 for (int i = 0; i < N; i++)
8 {
9     a *= b;
10 }
11 cout << a << endl;
```

当 N 大于一定值时输出的 a 不为 1:

```
1 ...
2 N为77结果: 1
3 N为78结果: 1
4 N为79结果: 1
5 N为80结果: 1
6 N为81结果: 1
7 N为82结果: 0.999999
8 N为83结果: 0.999997
9 N为84结果: 0.999998
10 N为85结果: 0.99998
11 N为86结果: 1.00003
12 N为87结果: 1.00018
13 N为88结果: 1.00018
```


14 | ...

(2) N 个数求和

```

1  const int NUM = 2048;
2  const int LOGN = 12;
3
4  double elem[6][NUM], sum, sum1, sum2;
5  void init(double e[][NUM], int m)
6  {
7      for (int i = 0; i < NUM; i++)
8      {
9          e[m][i] = (rand() % 10) / 7.0;
10     }
11 }
12
13 void chain(int m, int n)
14 {
15     sum = 0;
16     for (int i = 0; i < n; i++) {
17         sum += elem[m][i];
18     }
19 }
20 void tree(int m, int n)
21 {
22     int i, j;
23
24     while (n >= 8) {
25         for (i = 0, j = 0; i < n; i += 8) {
26             elem[m][j] = elem[m][i] + elem[m][i + 1];
27             elem[m][j + 1] = elem[m][i + 2] + elem[m][i + 3];
28             elem[m][j + 2] = elem[m][i + 4] + elem[m][i + 5];
29             elem[m][j + 3] = elem[m][i + 6] + elem[m][i + 7];
30             j += 4;
31         }
32         n >>= 1;
33     }
34
35     elem[m][0] += elem[m][1];
36     elem[m][2] += elem[m][3];
37     elem[m][0] += elem[m][2];
38 }

```

运行 chain 以及 tree 函数，由于这两个函数的求和顺序不一致，结果可能不一样。这里最终的一次结果为：

```

1  Tree: 1301.14285714285688300151

```

2 Chain: 1301.14285714285460926476

5. 更多探索。

同学们如有余力，可探索更多的算法策略、程序优化方法，如循环展开、cache 优化等等。自主选题的同学不要局限于例子中循环展开、打包向量化的思路，可根据选题的特点选择恰当的 SIMD 指令进行并行优化。

2.1.3 NEON 的 C/C++ 编程

这里主要围绕 Neon Intrinsic 进行说明，如果希望进行更细粒度的编程可以考虑在源程序直接内嵌汇编代码。

使用 NEON intrinsic 需要包含的头文件如下

1 `#include <arm_neon.h>`

编译选项：-march=native 或 -march=armv8-a

一些常用的指令：

```

1 //数据类型
2 float32_t, float32x4_t, float64x2_t, float32x4x2_t...
3 int8_t, int8x16_t, int16x8_t, int32x4_t, int64x2_t, int8x16x2_t...
4
5 //load: 以float为例
6 float32x2_t vld1_f32(float32_t const *ptr); //读取连续2个单精度浮点数到向量寄存器
7 float32x4_t vld1q_f32(float32_t const *ptr); //读取连续4个单精度浮点数到向量寄存器
8 float32x4x2_t vld2q_f32(float32_t const *ptr); //读取连续8个单精度浮点数到2个向量寄存器
9
10 //store:
11 void vst1q_s32(int32_t * ptr, int32x4_t val); //将向量元素保存为连续4个32位整数数
12 void vst1q_u32(uint32_t * ptr, uint32x4_t val);
13 //将向量元素保存为连续4个32位无符号整数数
14 void vst1q_f32(float32_t * ptr, float32x4_t val);
15 //将向量元素保存为连续4个单精度浮点数数
16
17 //move
18 int32x2_t vget_high_s32(int32x4_t a); //将a高位的两个元素复制到另一个向量寄存器
19 float32x2_t vget_low_f32(float32x4_t a); //将a低位的两个元素复制到另一个向量寄存器
20 float32_t vget_lane_f32(float32x2_t v, const int lane); //获得向量v第lane个通道的元素
21 float32_t vgetq_lane_f32(float32x4_t v, const int lane); //获得向量v第lane个通道的元素
22 float32x2_t vset_lane_f32(float32_t a, float32x2_t v, const int lane); //设置向量v第lane个通道的元素的值为a
23 float32x4_t vsetq_lane_f32(float32_t a, float32x4_t v, const int lane); //设置向量v第lane个通道的元素的值为a
24

```

```

25 //arithmetic
26 float32x4_t vaddq_f32(float32x4_t a, float32x4_t b); //对位加法
27 float32x4_t vmulq_f32(float32x4_t a, float32x4_t b); //对位乘法
28 float32x4_t vsubq_f32(float32x4_t a, float32x4_t b); //对位减法
29 float32x4_t vdivq_f32(float32x4_t a, float32x4_t b); //对位除法

```

一个简单的例子：

```

1 float sum(float* array,int n)
2 {
3     float sum=0;
4     for (int k = 0; k < n; k++)
5     {
6         sum+=array[k];
7     }
8
9     return sum;
10 }
11
12 float sum_neon(float* array,int n)
13 {
14     assert(n%4==0); // 假设n为4的倍数
15     // 声明一个包含4个单精度浮点数的向量变量，用0初始化
16     float32x4_t sum4=vmovq_n_f32(0);
17     for(int i=0;i<n;i+=4){
18         // 从(array+i)地址加载连续4个32位整数，保存到temp向量
19         float32x4_t temp=vld1q_f32(array+i);
20         // sum4与temp向量对位相加
21         sum4=vaddq_f32(sum4,temp);
22     }
23     // 将低位两个元素保存到suml2向量
24     float32x2_t suml2=vget_low_f32(sum4);
25     // 将高位两个元素保存到sumh2向量
26     float32x2_t sumh2=vget_high_f32(sum4);
27     // 向量进行水平加法，得到suml2中两元素的和以及sumh2中两元素的和
28     suml2=vpadd_f32(suml2,sumh2);
29     // 再次进行水平加法，得到sum4向量4个元素的和
30     float32_t sum=vpadds_f32(suml2);
31     return (float)sum;
32 }

```

详细完整的 *NEON Intrinsic* 函数说明可以查询 *ARM* 官网文档 (点击超链接): [Intrinsics – Arm Developer](#)

2.1.4 SSE/AVX 的 C/C++ 编程

SSE 指令对应了 C/C++ 的 intrinsics（编译器能识别的函数，直接映射为一个或多个汇编语言指令）。使用 SSE intrinsics 所需的头文件：

```

1 #include <xmmintrin.h> //SSE
2 #include <emmintrin.h> //SSE2
3 #include <pmmmintrin.h> //SSE3
4 #include <tmmintrin.h> //SSSE3
5 #include <smmintrin.h> //SSE4.1
6 #include <nmmmintrin.h> //SSE4.2
7 #include <immintrin.h> //AVX、AVX2

```

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

一些常用的指令：

```

1 //数据类型
2 __m128//float
3 __m128d// double
4 __m128i//integer
5 //load:
6 _mm_load_ps(float *p) //将从内存中地址p开始的4个float数据加载到寄存器中，要求p的地址是16字节对齐
7 _mm_loadu_ps(float *p) //类似_mm_load_ps但是不要求地址是16字节对齐
8 //set:
9 _mm_set_ps(float a,float b,float c,float d) //将a,b,c,d赋值给寄存器
10 //store:
11 _mm_store_ps(float *p, __m128 a) //将寄存器a的值存储到内存p中
12 //数据计算:
13 _mm_add_ps //加法
14 _mm_mul_ps //乘法
15 _mm_sub_ps //减法
16 _mm_div_ps //除法

```

AVX 的各个指令与 SSE 类似，如 `_mm_loadu_ps` 的 AVX 版本为 `_mm256_loadu_ps`。

一个简单的例子：

```

1 \\串行加法:
2 void add()
3 {
4     for (int k = 0; k < N; k++)
5     {
6         matrix[k] += 2;
7     }
8 }

```

```

9  \\SSE优化
10 void add()
11 {
12     __m128 t1, t2;
13     t1 = _mm_set1_ps(2); //t1中4个单精度浮点数设为2
14     for (int k = 0; k < N; k += 4)
15     {
16         // 从(matrix+k)读取连续的4个单精度浮点数
17         t2 = _mm_loadu_ps(matrix+k);
18         // 两个向量的4个单精度浮点数对位相加
19         t2 = _mm_add_ps(t1, t2);
20         // 将向量t2, 保存在地址(matrix+k)处
21         _mm_store_ps(matrix+k, t2);
22     }
23 }

```

可参考此例以及课程讲义中矩阵乘法的例子对 LU 中的关键循环进行向量化。更多 SSE/AVX 指令，以及 AVX 的编程大家可以参考课程讲义。

所有 SSE/AVX 指令的细节可以查询官网文档 (点击超链接): Intel® IntrinsicsGuide

3 程序编译及运行

环境搭建和工具使用参考“实验环境搭建”指导书，ARM 平台实验使用我们的华为鲲鹏集群环境完成，x86 的实验可使用个人的笔记本电脑/台式机完成，AVX-512 的实验使用 Intel DevCloud 环境完成（为保持一致，SSE/AVX/AVX-512 的实验可均在此平台完成）。另外，鼓励大家测试多组数据和多种不同的优化算法进行对比，实验指导中所有结果仅供参考。

在鲲鹏服务器上编译的例子。

```

1 // gcc 编译器
2 gcc -g -march=native add.c -o add // c语言
3 g++ -g -march=native add.cpp -o add // c++
4 // 毕升编译器 (clang)
5 clang -g -march=armv8-a add.c -o add // c语言
6 clang++ -g -march=armv8-a add.cpp -o add // c++

```

这里补充一个 linux 下高精度时间测量的例子。

```

1 #include <stdio.h>
2 #include <time.h>
3

```

```
4 struct timespec sts, ets;
5 timespec_get(&sts, TIME_UTC);
6 // to measure
7 timespec_get(&ets, TIME_UTC);
8 time_t dsec=ets.tv_sec-sts.tv_sec;
9 long dnsec=ets.tv_nsec-sts.tv_nsec;
10 printf("%llu.%09llu\n", dsec, dnsec);
```

4 使用 VTune 等工具剖析程序性能

类似之前的实验，实际上 NEON、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，我们可以使用 per、VTune 等 profiling 工具分析对比。具体使用方法参考体系结构相关及性能测试实验指导书。