SIMD 编程实验
——以高斯消去为例

杜忱莹

April 2021

目录

1 实验介绍

1.1 实验选题

1. 默认选题：高斯消去法（LU 分解）。
2. 鼓励自主选题，与期末研究报告结合：期末研究报告研究一个较大的问题，SIMD 编程实验 (包括后面其他并行编程实验) 关注其中的某部分重要计算，进行向量化，这样，这部分工作未来可纳入期末研究报告。
3. 自选题目难度至少与默认选题（共轭梯度）相当，适合 SIMD 并行化。
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

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。
3. 设计实验方案，进行实验。
4. 撰写研究报告（问题描述、SSE 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，与源码一起提交（只提交源程序文件和程文件，不要提交编译出的目标文件和可执行文件）。

2 实验设计指导

2.1 题目分析

高斯消去的计算模式如图 2.1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。

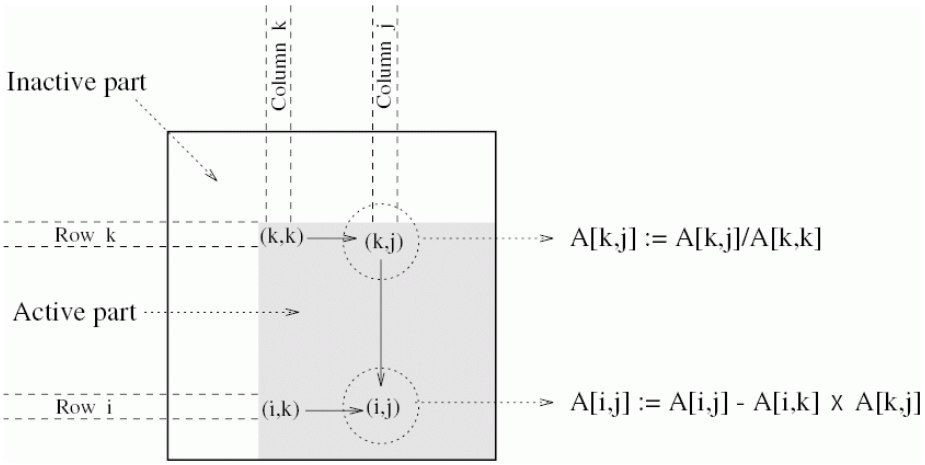

图 2.1: 高斯消去法示意图

```
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     A[k, k] := 1.0;
7     endfor;
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 SSE/AVX 对这两步进行并行优化。

2.2 算法设计与编程

在设计算法时，我们需要注意以下要点：

1. **测试用例规模的确定**。

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。

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。**

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。

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：

```
...
N为77结果：1
N为78结果：1
N为79结果：1
N为80结果：1
N为81结果：1
N为82结果：0.999999
N为83结果：0.999997
N为84结果：0.999998
N为85结果：0.99998
N为86结果：1.00003
N为87结果：1.00018
N为88结果：1.00018
...
```

（2）N 个数求和

```
const int NUM = 2048;
const int LOGN = 12;

double elem[6][NUM], sum, sum1, sum2;
void init(double e[][NUM], int m)
{
    for (int i = 0; i < NUM; i++)
    {
        e[m][i] = (rand() % 10) / 7.0;
    }

}
void chain(int m, int n)
{
    sum = 0;
    for (int i = 0; i < n; i++) {
        sum += elem[m][i];
    }
}
void tree(int m, int n)
{
    int i, j;

    while (n >= 8) {
        for (i = 0, j = 0; i < n; i += 8) {
            elem[m][j] = elem[m][i] + elem[m][i + 1];
            elem[m][j + 1] = elem[m][i + 2] + elem[m][i + 3];
            elem[m][j + 2] = elem[m][i + 4] + elem[m][i + 5];
            elem[m][j + 3] = elem[m][i + 6] + elem[m][i + 7];
```

```
            j += 4;
        }
        n >>= 1;
    }

    elem[m][0] += elem[m][1];
    elem[m][2] += elem[m][3];
    elem[m][0] += elem[m][2];
}
```

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：

```
Tree: 1301.14285714285688300151
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。

2.3　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：

```
#include <xmmintrin.h> //SSE
#include <emmintrin.h> //SSE2
#include <pmmintrin.h> //SSE3
#include <tmmintrin.h> //SSSE3
#include <smmintrin.h> //SSE4.1
#include <nmmintrin.h> //SSSE4.2
#include <immintrin.h> //AVX、AVX2
```

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

一些常用的指令：

```
    //数据类型
__m128//float
__m128d// double
__m128i//integer
//load:
_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
```

```
_mm_loadu_ps(float *p)//类似_mm_load_ps但是不要求地址是16字节对齐
//set:
_mm_set_ps(float a,float b,float c,float d) //将a,b,c,d赋值给寄存器
//store:
_mm_store_ps(float *p, __m128 a) //将寄存器a的值存储到内存p中
//数据计算:
_mm_add_ps //加法
_mm_mul_ps //乘法
_mm_sub_ps //减法
_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：

```
\\串行加法:
void add()
{
  for (int k = 0; k < N; k++)
  {
    matrix[k] += 2;
  }
}
\\SSE优化
void add()
{
    __m128 t1, t2;
    t1 = _mm_set1_ps(2);
    for (int k = 0; k < N; k += 4)
    {
      t2 = _mm_loadu_ps(matrix+k);
      t2 = _mm_add_ps(t1, t2);
      _mm_store_ps(matrix+k, t2);
    }
}
```

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。

ARM 平台 Neon 编程类似，可参考讲义。

3 程序编译及运行

参考“实验环境搭建”指导书 1.2.2-1.2.4 部分对 CodeBlocks 进行对应实验环境配置后，使用 F9 进行“Build and run”。其他环境下的编译和运行可参考“实验环境搭建”指导书的其他部分。

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。

4 使用 VTune 等工具剖析程序性能

类似之前的实验，实际上 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 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。