SIMD 编程作业

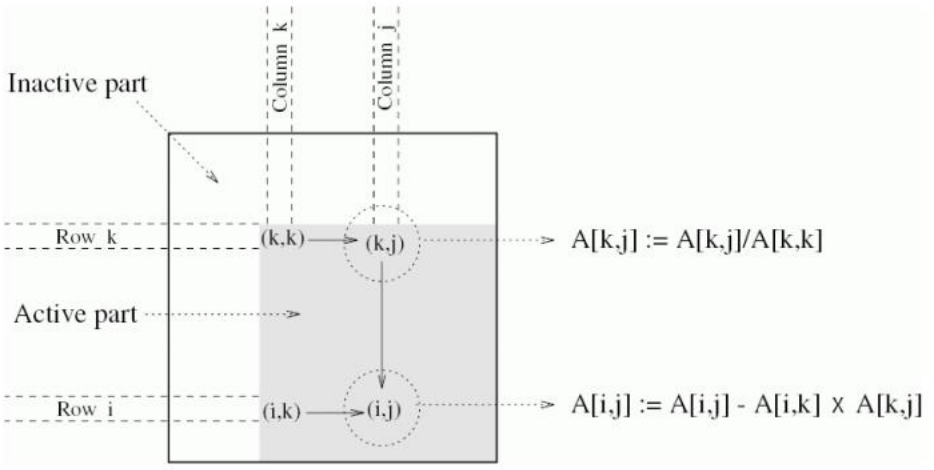
学号：2120210492　　姓名：肖飞

目录

一、问题描述

基于 X86/X64 平台完成普通高斯消去计算的基本 SIMD 并行化。

普通高斯消去法的计算模式如下图：



其中，主要涉及的操作为：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。

二、实验环境

实验平台：x64 平台

CPU：12th Gen Intel(R) Core(TM) i5-12500H　　3.10 GHz

支持的指令集如下：

操作系统：　Windows 11 系统

编译器：TDM-GCC 编译器

IDE：　Code::Blocks 集成开发环境

三、SIMD 算法设计与实现

3.1、串行算法及其并行潜力分析

串行算法的伪代码如下：

```
procedure LU (A)
begin
    for k := 1 to n do
        for j := k+1 to n do
            A[k, j ] := A[k, j]/A[k, k];
        endfor;
        A[k, k] := 1.0;
        for i := k + 1 to n do
            for j := k + 1 to n do
                A[i, j ] := A[i, j ]  −  A[i, k]×A[k, j];
            endfor;
            A[i, k] := 0;
        endfor;
    endfor;
end LU
```

可以看到，串行算法复杂度为 O(n^3)。

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]×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。

串行算法的具体实现如下：

```
//串行普通高斯消去算法
void m_gauss(int n)
{
    for(int k = 0 ; k < n ; k++)
    {
        for(int j = k+1 ; j < n ; j++)
        {
            m[k][j] = m[k][j]/m[k][k];
        }
        m[k][k] = 1.0;
        for(int i = k+1 ; i < n ; i++)
        {
            for(int j = k+1 ; j < n ; j++)
            {
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            }
            m[i][k] = 0;

        }
    }
}
```

完整代码实现请参考附件中的 GitHub 项目。

3.2、SIMD 算法并行优化设计与实现

本文主要设计采用 SSE 版本的 SIMD 并行化算法，由于 SSE 支持的向量寄存器为 128 位，即 4 个单精度浮点数，而本题中系数矩阵元素数据类型为单精度浮点数，故所设计的 SIMD 算法为 4 路向量化算法。

本文讨论对比的主要编程策略如下：

3.2.1、对齐与不对齐算法策略

注意到在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。

在 x86/64 平台上，通过_mm_loadu_ps()可支持不对齐的算法策略；如果采用对齐的策略，需动态对齐，即先串行处理到对齐边界，然后进行 SIMD 的计算。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。

3.2.2、串行算法不同部分向量化

前面看到，高斯消去法中有两个部分（除法部分和乘法部分）可以进行向量化，本文之后对比这两个 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。

同时程序主体循环均按照行主次序访问，空间局部性较好，也不需要专门事先对矩阵进行转置。

为了支持不同策略的对比，需要分别实现以下四个不同版本的向量化：

a. 除法部分向量化、不对齐

b. 乘法部分向量化、不对齐

c. 乘法和除法部分全部向量化、不对齐

d. 乘法和除法部分全部向量化、对齐

下面是使用 SIMD Intrinsic 函数采用不对齐内存访问策略对普通高斯消元进行向量化的伪代码。

```
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 ← dupTo4Float(A[k,k]);
3 for j = k + 1; j + 4 <= n; j+ = 4 do
4 va ← 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 ← va/vt ; // 这里是向量对位相除
6 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] ← 1.0;
10 for i ← k+1 to n-1 do
11 vaik ← dupToVector4(A[i,k]);
12 for j = k + 1; j + 4 <= n; j+ = 4 do
13 vakj ← load4FloatFrom(&A[k,j]);
```

14 vaij ← load4FloatFrom(&A[i,j]);
15 vx ← vakj*vaik;
16 vaij ← vaij-vx;
17 store4FloatTo(&A[i,j],vaij);
18 for j in 剩余所有下标 do
19 A[i,j] ← A[i,j] -A[k,j]*A[i,k];
20 A[i,k] ← 0;

不同版本具体代码实现如下：

1、除法部分向量化、不对齐

```
//除法部分向量化、不对齐
void m_gauss_simd_div(int n)
{
    __m128 vt, va;
    for(int k = 0; k < n; k++){
        vt = _mm_set_ps1(m[k][k]);
        int j;
        for(j = k+1; j+4 <= n; j+=4){
            va = _mm_loadu_ps(&m[k][j]);
            va = _mm_div_ps(va, vt);
            _mm_storeu_ps(&m[k][j], va);
        }
        if(j < n){
            for(;j < n; j++){
                m[k][j] = m[k][j]/m[k][k];
            }
        }
        m[k][k] = 1.0;
        for(int i = k+1 ; i < n ; i++)
        {
            for(int j = k+1 ; j < n ; j++)
            {
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            }
            m[i][k] = 0;

        }
    }
}
```

2、乘法部分向量化、不对齐

```
//乘法部分向量化、不对齐
void m_gauss_simd_mul(int n)
{
    __m128 vaik, vakj, vaij, vx;
    for(int k = 0; k < n; k++){
        for(int j = k+1 ; j < n ; j++)
        {
            m[k][j] = m[k][j]/m[k][k];
        }
        m[k][k] = 1.0;
        for(int i = k+1; i < n; i++){
            vaik = _mm_set_ps1(m[i][k]);
            int j;
            for(j = k+1; j+4 <= n; j+=4){
                vakj = _mm_loadu_ps(&m[k][j]);
                vaij = _mm_loadu_ps(&m[i][j]);
                vx = _mm_mul_ps(vakj, vaik);
                vaij = _mm_sub_ps(vaij, vx);
                _mm_storeu_ps(&m[i][j], vaij);
            }
            if(j < n){
                for(;j < n; j++){
                    m[i][j] = m[i][j] - m[k][j]*m[i][k];
                }
            }
            m[i][k] = 0;
        }
    }
}
```

3、乘法和除法部分全部向量化、不对齐

```
//乘法和除法部分全部向量化、不对齐
void m_gauss_simd(int n)
{
    __m128 vt, va, vaik, vakj, vaij, vx;
    for(int k = 0; k < n; k++){
        vt = _mm_set_ps1(m[k][k]);
        int j;
        for(j = k+1; j+4 <= n; j+=4){
            va = _mm_loadu_ps(&m[k][j]);
            va = _mm_div_ps(va, vt);
            _mm_storeu_ps(&m[k][j], va);
        }
        if(j < n){
            for(;j < n; j++){
                m[k][j] = m[k][j]/m[k][k];
            }
        }
        m[k][k] = 1.0;
        for(int i = k+1; i < n; i++){
            vaik = _mm_set_ps1(m[i][k]);
            int j;
            for(j = k+1; j+4 <= n; j+=4){
                vakj = _mm_loadu_ps(&m[k][j]);
                vaij = _mm_loadu_ps(&m[i][j]);
                vx = _mm_mul_ps(vakj, vaik);
                vaij = _mm_sub_ps(vaij, vx);
                _mm_storeu_ps(&m[i][j], vaij);
            }
            if(j < n){
                for(;j < n; j++){
                    m[i][j] = m[i][j] - m[k][j]*m[i][k];
                }
            }
            m[i][k] = 0;
        }
    }
}
```

4、乘法和除法部分全部向量化、对齐

```
//乘法和除法部分全部向量化、对齐
void m_gauss_simd_align(int n)
{
    __m128 vt, va, vaik, vakj, vaij, vx;
    for(int k = 0; k < n; k++){
        vt = _mm_set_ps1(m[k][k]);
        int j;
        int start = k-k%4+4;
        for(j = k+1; j < start && j < n; j++){
            m[k][j] = m[k][j]/m[k][k];
        }
        if(j != n){
            for(j = start; j+4 <= n; j+=4){
                va = _mm_load_ps(&m[k][j]);
                va = _mm_div_ps(va, vt);
                _mm_store_ps(&m[k][j], va);
            }
            if(j < n){
                for(;j < n; j++){
                    m[k][j] = m[k][j]/m[k][k];
                }
            }
        }
        m[k][k] = 1.0;
        for(int i = k+1; i < n; i++){
            vaik = _mm_set_ps1(m[i][k]);
            int j;
            int start = k-k%4+4;
            for(j = k+1; j < start && j < n; j++){
                m[i][j] = m[i][j] - m[k][j]*m[i][k];
            }
            if(j != n){
                for(j = start; j+4 <= n; j+=4){
                    vakj = _mm_load_ps(&m[k][j]);
                    vaij = _mm_load_ps(&m[i][j]);
                    vx = _mm_mul_ps(vakj, vaik);
                    vaij = _mm_sub_ps(vaij, vx);
                    _mm_store_ps(&m[i][j], vaij);
                }
                if(j < n){
                    for(;j < n; j++){
                        m[i][j] = m[i][j] - m[k][j]*m[i][k];
                    }
                }
            }
            m[i][k] = 0;
        }
    }
}
```

完整代码实现请参考附件中的 GitHub 项目。

四、实验及结果分析

程序不同问题规模下、不同算法/编程策略性能测试结果部分截图如下:

整理如下:

n	串行算法 (ms)	“除法部分” 不对齐(ms)	“乘法部分” 不对齐(ms)	“全部” 不对齐(ms)	“全部” 对齐(ms)
10	0.0005	0.0005	0.0005	0.0006	0.0021
20	0.0017	0.0016	0.0012	0.0015	0.0016
30	0.0041	0.0049	0.0027	0.0029	0.0032
40	0.0103	0.0105	0.0051	0.0055	0.0047
50	0.02	0.0172	0.0087	0.0086	0.0089
60	0.0293	0.0366	0.0141	0.0138	0.0105
70	0.0515	0.056	0.02	0.0199	0.0181
80	0.0873	0.0662	0.0288	0.0287	0.0207

90	0.1061	0.1194	0.0373	0.0385	0.0309
100	0.1391	0.1627	0.0547	0.0551	0.0363
200	1.036	1.0895	0.4226	0.4159	0.2562
300	3.6113	3.8907	1.3715	1.3558	0.8986
400	7.2661	9.0594	3.5762	3.2045	2.2882
500	16.1224	20.2261	6.897	6.2451	4.288
600	26.3135	35.3986	12.7556	12.3989	8.8755
700	42.9281	55.2675	20.3895	20.2452	15.139
800	68.7046	78.0836	27.9411	30.4078	23.1199
900	87.5921	113.069	40.8275	41.3449	34.4532
1000	117.627	156.978	57.2039	55.533	45.4777

通过对以上结果进行分析，可以得到以下结论：

1、 当测试规模较小时，可能出现并行算法比串行算法还要耗时的情况。

2、 上述最优的 SIMD 并行算法（对齐的向量化）加速比可以达到 2.6 左右。

3、 不同部分向量化对比："除法部分"时间复杂度为 O(n^2)，"乘法部分"为 O(n^3)，对两部分分别进行向量化，绝大部分情况下，在不对齐策略下，"除法部分"耗时反而大于串行算法，对齐开销完全抵消了 SIMD 的并行化收益，"乘法部分"加速比能达到 2.0 左右，可见对程序运行占比很小的循环体部分向量化有时反而可能增加耗时，而对占比较大的循环体部分进行向量化是有效的并行优化策略。另外，

“两部分全部向量化”与“乘法部分向量化”对比，两者耗时相当，且没有一定相对最优的选择。

4、 对齐与不对齐策略：采用对齐访问策略的算法除了问题规模较小时，执行结果都明显优于采用不对齐策略的算法，且性能优化可以达到 20%左右。

五、附件

Git 项目链接：https://github.com/XFLasdf/Guass_SIMD