南開大學
Nankai University

计算机学院

并行程序设计第 **3** 次作业

高斯消去法的 SIMD 并行化

姓名：丁屹

学号：2013280

专业：计算机科学与技术

2022 年 4 月 12 日

目录

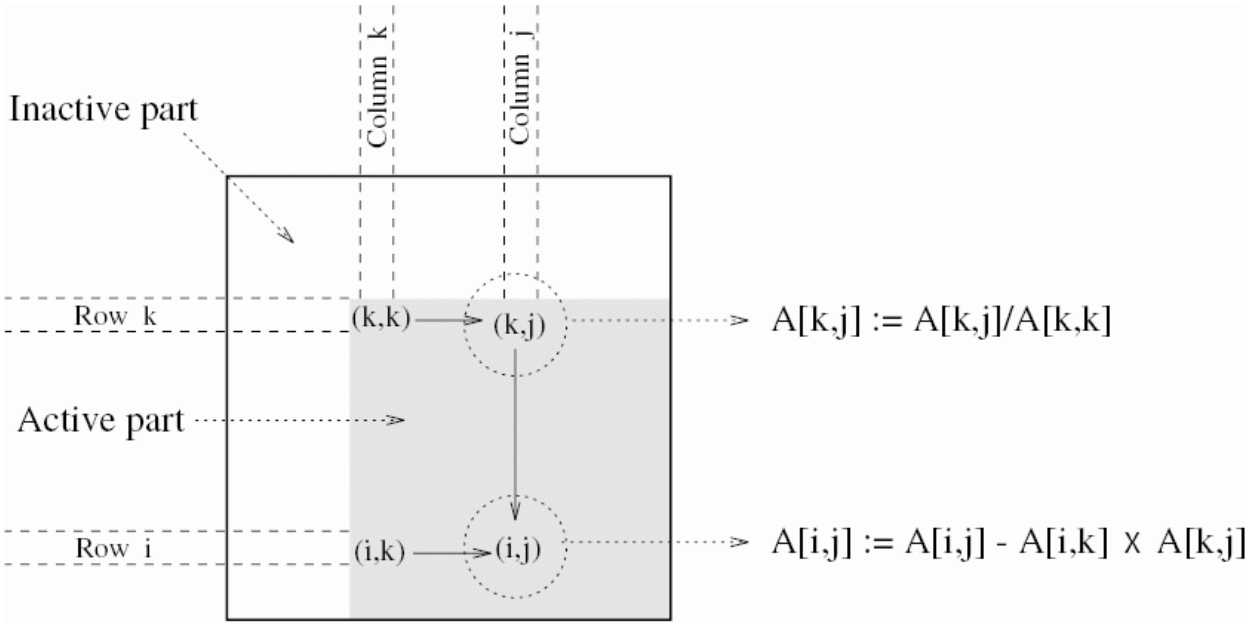

图 1.1: 高斯消去法示意图

1 问题描述

高斯消去的计算模式如图 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。

Algorithm 1 普通高斯消元算法伪代码

```
1:  function LU
2:      for k := 0 to n do
3:          for j := k + 1 to n do
4:              A[k, j] := A[k, j]/A[k, k]
5:          end for
6:          A[k, k] := 1.0
7:          for i := k + 1 to n do
8:              for j := k + 1 to n do
9:                  A[i, j] := A[i, j] − A[i, k] ∗ A[k, j]
10:             end for
11:             A[i, k] := 0
12:         end for
13:     end for
14: end function
```

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 SIMD 算法设计

2.1 测试用例的确定

由于测试数据集较大，不便于各个平台同步，所以采用固定随机数种子为 12345687 的 mt19937 随机数生成器。经过实验发现不同规模下，所有元素独立生成，限制大小在 $[0, 100]$，能够生成可以被正确消元的矩阵。

代码如下：

测试数据集生成器

```
uniform_real_distribution<float> dist(0, 100);
mt19937          mt(12345687);
int              n;
istringstream iss(argv[1]);
iss >> n;
cout << n << endl;
for (int i = 1; i <= n; ++i)
  for (int j = 1; j <= n; ++j) cout << dist(mt) << " \n"[j == n];
```

2.2 算法设计

2.2.1 默认平凡算法

编译选项：-O3；

使用一维数组模拟矩阵，避免改变矩阵大小时第二维不方便调整、必须设成最大值的问题，可以减少 cache 失效；

使用 $\#define\ matrix(i,j)\ arr[i*n+j]$ 宏，增强可读性；

平凡算法

```
void func(int& ans, float arr[], int n) {
#define matrix(i, j) arr[i * n + j]
  for (int k = 0; k < n; ++k) {
    for (int j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
    matrix(k, k) = 1.0;
    for (int i = k + 1; i < n; ++i) {
      for (int j = k + 1; j < n; ++j)
        matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
      matrix(i, k) = 0;
    }
  }
#undef matrix
}
```

2.2.2 使用 NEON 指令集并行化加速

实验在华为鲲鹏 ARM 集群平台完成；

编译选项：-march=native 或 -march=armv8-a，全部开启 -O3；

NEON 内存未对齐算法

```
void func(int& ans, float arr[], int n) {
#define matrix(i, j)  arr[i * n + j]
#define pmatrix(i, j) (arr + (i * n + j))
  for (int k = 0; k < n; ++k) {
    auto vt = vdupq_n_f32(matrix(k, k));
    int  j;
    for (j = k + 1; j + 4 <= n; j += 4) {
      auto va = vld1q_f32(pmatrix(k, j));
      va      = vdivq_f32(va, vt);
      vst1q_f32(pmatrix(k, j), va);
    }
    for (; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
    matrix(k, k) = 1.0;
    for (int i = k + 1; i < n; ++i) {
      auto vaik = vdupq_n_f32(matrix(i, k));
      for (j = k + 1; j + 4 <= n; j += 4) {
        auto vakj = vld1q_f32(pmatrix(k, j));
        auto vaij = vld1q_f32(pmatrix(i, j));
        auto vx   = vmulq_f32(vakj, vaik);
        vaij      = vsubq_f32(vaij, vx);
        vst1q_f32(pmatrix(i, j), vaij);
      }
      for (; j < n; ++j)
        matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
      matrix(i, k) = 0;
    }
  }
#undef matrix
#undef pmatrix
}
```

NEON 内存对齐算法

```
void func(int& ans, float arr[], int n) {
#define matrix(i, j)  arr[i * n + j]
#define pmatrix(i, j) (arr + (i * n + j))
  for (int k = 0; k < n; ++k) {
    auto vt = vdupq_n_f32(matrix(k, k));
    int  j  = k + 1;
    for (; j < n && j % 4; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
    for (; j + 4 <= n; j += 4) {
      auto va = vld1q_f32(pmatrix(k, j));
      va      = vdivq_f32(va, vt);
      vst1q_f32(pmatrix(k, j), va);
    }
    for (; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
```

```
    matrix(k, k) = 1.0;
    for (int i = k + 1; i < n; ++i) {
      auto vaik = vdupq_n_f32(matrix(i, k));
      for (j = k + 1; j < n && j % 4; ++j)
        matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
      for (; j + 4 <= n; j += 4) {
        auto vakj = vld1q_f32(pmatrix(k, j));
        auto vaij = vld1q_f32(pmatrix(i, j));
        auto vx   = vmulq_f32(vakj, vaik);
        vaij      = vsubq_f32(vaij, vx);
        vst1q_f32(pmatrix(i, j), vaij);
      }
      for (; j < n; ++j)
        matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
      matrix(i, k) = 0;
    }
  }
#undef matrix
#undef pmatrix
}
```

可以看到内存对齐算法就是在 $j := k+1$ 之后先平凡地计算，直到 $j\%4 = 0$，保证了偏移 $4\times4 = 16$ 字节对齐由于使用了 *aligned_alloc* 分配了 32 字节对齐的内存块，故保证了 16 字节对齐。

2.2.3 使用 SSE、AVX 指令集并行化加速

实验在本地 x86 平台完成；

编译选项：-march=native -O3；

x86 下的 SSE、AVX 指令与 NEON 指令有相似的定义，可以方便地转换。内存对齐方式也与 NEON 的内存对齐方式类似，篇幅原因不再给出代码。

SSE 内存未对齐算法

```
void func(int& ans, float arr[], int n) {
#define matrix(i, j)  arr[i * n + j]
#define pmatrix(i, j) (arr + (i * n + j))
  for (int k = 0; k < n; ++k) {
    auto vt =
      _mm_set_ps(matrix(k, k), matrix(k, k), matrix(k, k), matrix(k, k));
    int j;
    for (j = k + 1; j + 4 <= n; j += 4) {
      auto va = _mm_loadu_ps(pmatrix(k, j));
      va      = _mm_div_ps(va, vt);
      _mm_storeu_ps(pmatrix(k, j), va);
    }
    for (; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
    matrix(k, k) = 1.0;
    for (int i = k + 1; i < n; ++i) {
      auto vaik =
```

```
        _mm_set_ps(matrix(i, k), matrix(i, k), matrix(i, k), matrix(i, k));
      for (j = k + 1; j + 4 <= n; j += 4) {
        auto vakj = _mm_loadu_ps(pmatrix(k, j));
        auto vaij = _mm_loadu_ps(pmatrix(i, j));
        auto vx   = _mm_mul_ps(vakj, vaik);
        vaij      = _mm_sub_ps(vaij, vx);
        _mm_storeu_ps(pmatrix(i, j), vaij);
      }
      for (; j < n; ++j)
        matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
      matrix(i, k) = 0;
    }
  }
#undef matrix
#undef pmatrix
}
```

AVX 内存未对齐算法

```
void func(int& ans, float arr[], int n) {
#define matrix(i, j)  arr[i * n + j]
#define pmatrix(i, j) (arr + (i * n + j))
  for (int k = 0; k < n; ++k) {
    auto vt =
      _mm256_set_ps(matrix(k, k), matrix(k, k), matrix(k, k), matrix(k, k),
                    matrix(k, k), matrix(k, k), matrix(k, k), matrix(k, k));
    int j;
    for (j = k + 1; j + 8 <= n; j += 8) {
      auto va = _mm256_loadu_ps(pmatrix(k, j));
      va      = _mm256_div_ps(va, vt);
      _mm256_storeu_ps(pmatrix(k, j), va);
    }
    for (; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
    matrix(k, k) = 1.0;
    for (int i = k + 1; i < n; ++i) {
      auto vaik =
        _mm256_set_ps(matrix(i, k), matrix(i, k), matrix(i, k), matrix(i, k),
                      matrix(i, k), matrix(i, k), matrix(i, k), matrix(i, k));
      for (j = k + 1; j + 8 <= n; j += 8) {
        auto vakj = _mm256_loadu_ps(pmatrix(k, j));
        auto vaij = _mm256_loadu_ps(pmatrix(i, j));
        auto vx   = _mm256_mul_ps(vakj, vaik);
        vaij      = _mm256_sub_ps(vaij, vx);
        _mm256_storeu_ps(pmatrix(i, j), vaij);
      }
      for (; j < n; ++j)
        matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
      matrix(i, k) = 0;
    }
```

Scale	Reperat times	x86 ordinary (s)	x86 SSE unaligned (s)	x86 SSE aligned (s)	x86 AVX unaligned (s)	x86 AVX aligned (s)	arm ordinary (s)	arm NEON unaligned (s)	arm NEON aligned (s)
8 × 8	100	0.000001330460	0.000001379360	0.000001326980	0.000001479190	0.000001370300	0.000000525400	0.000000541900	0.000000484900
16 × 16	50	0.000001706920	0.000001955580	0.000001729300	0.000001857780	0.000001653840	0.000001666000	0.000001702000	0.000001401600
32 × 32	50	0.000003640080	0.000004258960	0.000003834360	0.000004063340	0.000003761560	0.000007127000	0.000008225000	0.000006609400
64 × 64	20	0.000015253300	0.000021343650	0.000015731900	0.000015885350	0.000014212650	0.000037566500	0.000051374500	0.000040106500
128 × 128	15	0.000098880800	0.000121785400	0.000096791267	0.000089601000	0.000077025000	0.000231574000	0.000349447333	0.000275770000
256 × 256	10	0.000716408500	0.000878330700	0.000622779500	0.000588631100	0.000452169800	0.001820356000	0.002725389000	0.002021820000
512 × 512	10	0.006722607300	0.007444306600	0.005199414400	0.005004266800	0.003661343100	0.014974396000	0.021199319000	0.016449107000
1024 × 1024	5	0.064893815400	0.084234089600	0.063566091600	0.052573721800	0.047696106600	0.135511226000	0.196087378000	0.137847262000
2048 × 2048	3	1.400074583333	1.534334469667	1.501020112000	1.366758369000	1.317500752000	1.101775523333	1.511987780000	1.093070066667
4096 × 4096	1	10.705585484000	11.438922085000	11.608043578000	9.813709308000	10.472416784000	13.088073440000	14.938220180000	14.540709080000

表 1: 所有平台结果对比

```
31    }
32  #undef matrix
33  #undef pmatrix
34  }
```

3 实验及结果分析

从表 1 中可以发现：

- x86 平台比 arm 平台普遍更快，可能是 CPU 单核性能差异的结果
- 单纯使用 SIMD 指令但不进行内存对齐带来的性能优化有限，甚至比 O3 优化的平凡算法慢
- 使用 SIMD 指令且进行内存对齐可以带来明显的性能提升
- SSE 的指令级并行程度不够，提升不太明显，但内存对齐的 AVX 指令性能优化明显，某些情况下对比平凡算法速度有近一倍的提升
- 在 arm 中，指令级并行加速效果不明显，甚至比 O3 优化的平凡算法慢，内存对齐的 NEON 指令在某些情况下可以超过 O3 优化的平凡算法