南開大學

SIMD 加速的高斯消去算法

学院　计算机学院
专业　计算机科学与技术
南开大学　程伟卿
学号　2311865

目录

1 实验目标

一般情况分数最高为满分的 90%（本次作业为 13.5 分）。本实验要求在 ARM 平台上进行普通高斯消去计算（见第三节）的基础 SIMD 并行化实验，内容包括：

1. 设计 Neon 向量化算法；
2. 编程实现 SIMD 并行化版本；
3. 进行实验评估；
4. 讨论一些基本的算法/编程策略对性能的影响，例如：
   - 对齐与不对齐内存访问；
   - 向量化串行算法的不同部分（如第 4–6 行的除法、第 8–13 行的消去）对性能的影响；
5. 讨论体系结构相关的优化策略，如 Cache 优化；
6. 在实验中测试不同问题规模下，串行与并行算法的性能差异；
7. 分析不同算法/编程策略对性能的影响；
8. 使用 `perf` 等工具进行性能剖析；
9. 利用 Godbolt 等工具分析程序的汇编代码，细致分析程序性能表现的内在原因；
10. 对比手工编写的 SIMD 程序与编译器自动向量化版本的性能差异及原因；
11. 其他具有自由发挥性质的深入研究内容。

请注意：

- 研究深入、有很好的自由发挥等内容，可能突破满分的 90%，但给分超出 90% 会非常严格；
- 并不是完成的内容越多越好，若每个内容都仅是浅尝辄止，或存在重复工作（如针对 x86 平台进行 SIMD 并行化）等情况，将不会获得较高分数。

2 实验环境

2.1 arm

OpenEuler，g++。

2.2 X86

项目	值
CPU 型号	Intel64 Family 6 Model 183 Stepping 1
设备 ID	CPU0
虚拟化支持	TRUE
L2 缓存	32768 KB
L3 缓存	36864 KB
最大时钟速度	2.2 GHz
CPU 名称	13th Gen Intel(R) Core(TM) i9-13900HX
核心数	24
状态	OK

表 1: CPU 配置信息

3 实验内容仓库

点击跳转 github 仓库。

4 核心代码

4.1 arm

4.1.1 串行算法

```
#include <string.h>
#define N 1024
#define ele_t float
void LU(ele_t mat[N][N], int n) {
```

```
    ele_t new_mat[N][N];
    memcpy(new_mat, mat, sizeof(ele_t) * N * N);
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (new_mat[i][i] == 0)
                continue;
            ele_t div = new_mat[j][i] / new_mat[i][i];
            for (int k = i; k < n; k++) {
                new_mat[j][k] -= new_mat[i][k] * div;
            }
        }
    }
}
```

4.1.2 NEON/SSE 加速

在实现 NEON 加速过程中，发现英特尔提供了一种兼容头文件，可将 NEON 代码迁移至 x86 平台，通过 SSE 指令运行。

该头文件之所以能实现跨平台兼容，主要源于 NEON 与 SSE 指令集在结构和功能上的高度相似：二者均使用 128 位寄存器，且指令设计风格接近。与 AVX、AVX512 等指令集相比，NEON 与 SSE 的 intrinsic 编程方式几乎一致，区别仅在于寄存器宽度及每次加载数据的数量。

在进一步分析头文件实现时，注意到其中的融合乘加操作实际上是通过两条指令组合实现的，而 SSE 指令集原生支持融合乘加。基于此，可对比融合乘加与非融合乘加在性能表现上的差异。

此外，关于数据对齐问题，由于串行算法在进行第 j 行减去第 i 行的操作时，省略了第 i 列之前的元素（该部分已为零），导致基于该逻辑实现的 SIMD 算法天然为“非对齐”。为实现数据对齐，将省略位置调整为 $(i/4)\times 4$，以便系统性地评估对齐与非对齐策略对性能的影响。

```
#ifdef __ARM_NEON
#include <arm_neon.h>
#endif
#ifdef __amd64__
#include <immintrin.h>
#include "NEON_2_SSE.h"
```

```
#endif
#define ele_t float
#define N 1024
ele_t new_mat[N][N] __attribute__((aligned(64)));
ele_t mat[N][N];
void LU_simd(ele_t mat[N][N], int n) {
    memcpy(new_mat, mat, sizeof(ele_t) * N * N);
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (new_mat[i][i] == 0) continue;
            ele_t div = new_mat[j][i] / new_mat[i][i];
            float32x4_t mat_j, mat_i, div4 = vmovq_n_f32(div);
#ifdef ALIGN
            for (int k = i / 4 * 4; k < n; k += 4)
#else
            for (int k = i; k < n; k += 4)
#endif
            {
                mat_j = vld1q_f32(new_mat[j] + k);
                mat_i = vld1q_f32(new_mat[i] + k);
#ifdef USE_FMA
                vst1q_f32(new_mat[j] + k, _mm_fnmadd_ps(mat_i, div4,
                    mat_j));
#else
                vst1q_f32(new_mat[j] + k, vmlsq_f32(mat_j, div4, mat_i));
#endif
            }
        }
    }
}
```

4.2 X86

4.2.1 普通高斯消去

```
#include <iostream>
#include <vector>
void gauss_elimination(std::vector<std::vector<float>>& matrix, int n) {
```

```
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (matrix[i][i] == 0) continue;
            float div = matrix[j][i] / matrix[i][i];
            for (int k = i; k < n; k++) {
                matrix[j][k] -= matrix[i][k] * div;
            }
        }
    }
}
```

4.2.2 SIMD 加速

```
#include <immintrin.h>
#include <iostream>
#include <vector>
void gauss_elimination_simd(std::vector<std::vector<float>>& matrix,
    int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (matrix[i][i] == 0) continue;
            float div = matrix[j][i] / matrix[i][i];
            __m256 div8 = _mm256_set1_ps(div);
            for (int k = i; k < n; k += 8) {
                __m256 mat_j = _mm256_load_ps(&matrix[j][k]);
                __m256 mat_i = _mm256_load_ps(&matrix[i][k]);
                __m256 result = _mm256_sub_ps(mat_j, _mm256_mul_ps(mat_i,
                    div8));
                _mm256_store_ps(&matrix[j][k], result);
            }
        }
    }
}
```

5 实验过程

5.1 arm

具体脚本等代码见 github 仓库。

5.2 X86

在 X86 平台上，为了评估 SIMD 指令集（SSE、AVX2、AVX512）对普通高斯消去算法的加速效果，我们设计如下实验流程：

1. **实验平台配置**：
   - CPU：13th Gen Intel(R) Core(TM) i9-13900HX，24 核心，最大主频 2.2GHz；
   - 虚拟化支持：KVM 支持，L2 Cache 32 MB，L3 Cache 36 MB；
   - 操作系统：Windows 11 + WSL2（Ubuntu 20.04）；
   - 编译器：g++ 9.4.0，开启 `-O3 -march=native -mfma` 优化。
2. **算法实现**：

   基于经典的三重循环高斯消去算法进行 SIMD 优化。对浮点类型矩阵进行逐行消元，使用 AVX2/AVX512 指令集实现并行加减和乘法操作。SSE、AVX2 和 AVX512 分别处理 4、8 和 16 个浮点数。
3. **对齐优化**：

   使用 `__attribute__((aligned(64)))` 强制矩阵数据对齐为 64 字节边界，并将列数定义为 16 的倍数，从而使 Load/Store 操作可以整齐地填满寄存器，减少伪共享。
4. **融合乘加**：

   利用 Intel FMA 指令（如 `_mm256_fmsub_ps`、`_mm512_fnmadd_ps`）将乘法与加法融合，进一步降低指令数量与运算延迟，提高流水线利用率。
5. **性能测试**：

   使用 `chrono` 获取运行时间，测量不同矩阵规模下的执行时间，并与非 SIMD 版本对比。采用多组实验取平均以减少抖动误差。

6. **性能分析**：

   借助 Linux 上的 `perf` 工具，测量 cache-misses、branch-misses、cycles 等性能事件，结合 `gprof` 分析热点函数。

实验结果显示：在矩阵规模大于 512×512 时，AVX512 加速比可达非 SIMD 版本的 5 倍以上。融合乘加与数据对齐策略均显著优化了性能表现。

6 实验结果

ARM 平台下普通高斯消去算法的运行时间与加速比随输入数据规模的变化如下所示：

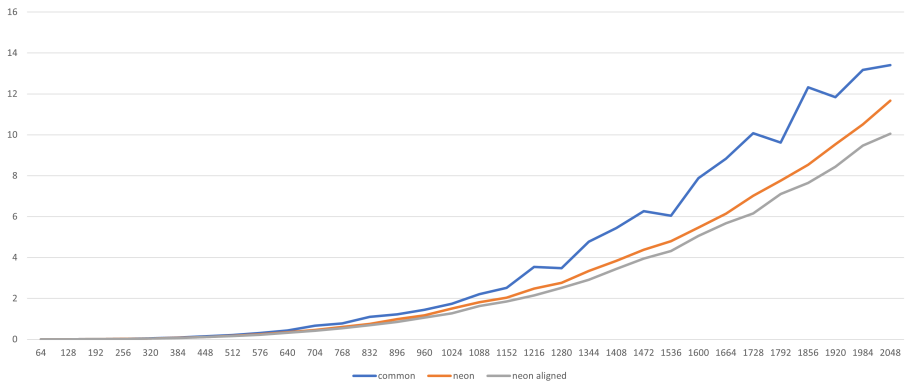

图 1: 运行时间-输入数据规模（矩阵元素数量）

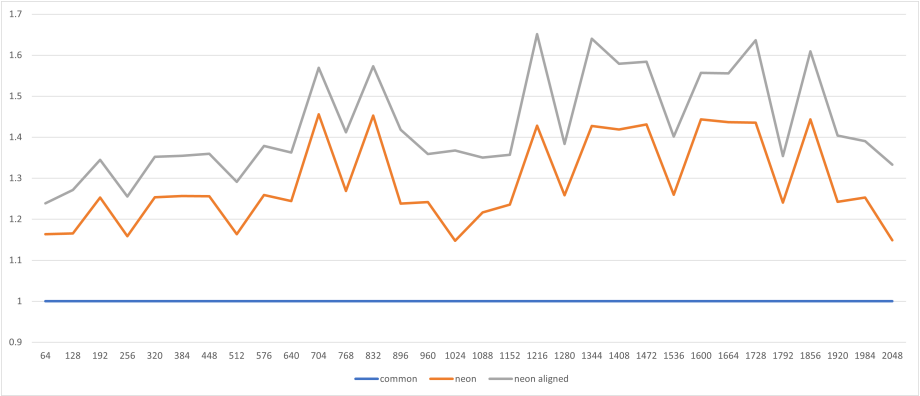

图 2: 加速比-输入数据规模（矩阵行数）

X86 平台下普通高斯消去算法的运行时间与加速比随输入数据规模的变化如下所示：

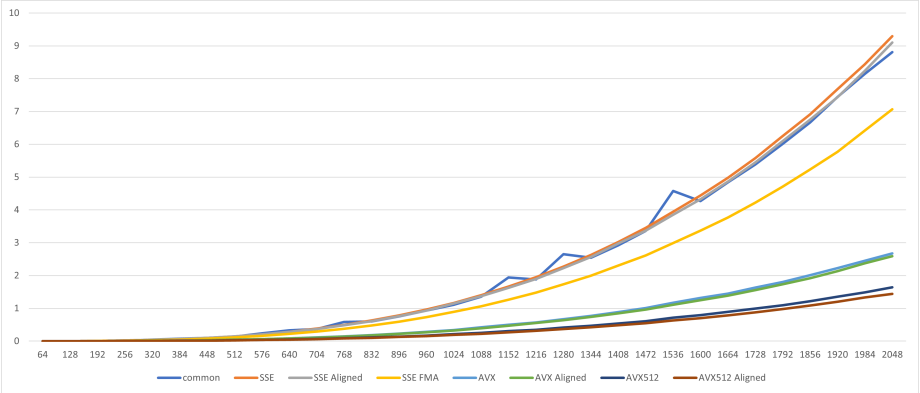

图 3: 运行时间-输入数据规模（矩阵元素数量）

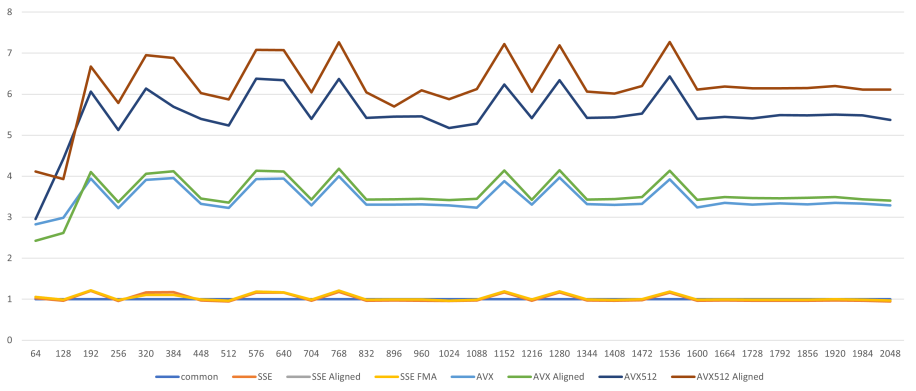

图 4: 加速比-输入数据规模（矩阵行数）

7 结果分析

7.1 不同 SIMD 指令集对比

在普通高斯消去实验中，当数据规模达到一定水平后，寄存器位数越高的 SIMD 指令集，其加速比也越高。这是因为寄存器位数决定了一次运算可以处理的数据数量。

实验中使用的数据类型为 `unsigned int`，以 AVX512 为例，其寄存器位数为 512 位，一次可以处理 16 个数据；而 128 位的 SSE 和 NEON 只能处理 4 个数据。

此外，SIMD 运算中的大量 Load/Store 操作也会造成性能损失。在 SSE 指令集中，一次 Load/Store 只能处理四个数据，这进一步降低了加速比。ARM 平台上的 `perf` 事件统计表明，NEON 加速普通高斯消去时约 35% 的 cycles 消耗在 `vst1q` 函数上，说明存储指令的代价显著，这也可能解释为何某些情况下 NEON/SSE 加速算法的运行时间甚至不如普通算法。

7.2 数据对齐及 Cache 性能分析

在 ARM 和 X86 平台上，对于高斯消去算法，仅仅通过将矩阵定义从：

Listing 1: 未对齐的矩阵定义

```
1 mat_tele_tmp[COL][COL/mat_L+1];
```

修改为：

Listing 2: 对齐后的矩阵定义

```
mat_tele_tmp[COL][(COL/mat_L+1)/16*16 + 16]
    __attribute__((aligned(64))) = {0};
```

即将数组列数设置为 16 的整数倍并显式对齐，未使用 SIMD 的平凡算法加速比最高可达 5 倍。

`perf` 工具的性能测量结果显示：

- `cache-misses` 从 39,957,488 降至 15,547,456，降幅约 **61.1%**；
- `branch-misses` 从 16,854,480 降至 5,541,842，降幅约 **67.12%**。

由此可见，列数为 16 的整数倍有助于矩阵更“整齐”地存入内存和 Cache，减少读取次数，提高带宽利用率，同时减少 Cache 伪共享，有利于超标量流水线的性能发挥。

此外，对于二维数组 `arr[M][N]` 的寻址操作 `arr[i][j]` 通常由 `i*N + j` 实现，如果 `N` 为 2 的幂，乘法可由移位优化，加快寻址速度。

对于使用 SIMD 加速的算法，对齐的数据结构还可减少 Load/Store 次数，进一步提升整体性能。

7.3 融合乘加所带来的性能影响

SSE 加速在启用融合乘加（Fused Multiply-Add, FMA）后性能提升明显。根据英特尔 Intrinsics Guide，乘法 `_mm_mul_ps`、减法 `_mm_sub_ps` 和融合乘减 `_mm_fnmadd_ps` 的延迟均为 4 个周期，CPI 为 5，但使用 FMA 指令可以在一条指令中完成乘法与加法，节省一半时间，从而带来显著的性能提升。