南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

SIMD 高斯消元

蒋薇

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2023 年 5 月 16 日

摘要

通过 SIMD 并行化实现高斯消元,分析优化性能,并将其与串行高斯消元对比,在 ARMX86 不同架构下分别 NENOSSE 测试，且初步探索 SIMD 特殊高斯消元。

关键字：Parallel,SIMD,Gaussian

目录

一、 SIMD 并行高斯消元

(一) 设计 SSE 算法

$$
\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\
 & & & & \vdots & & & & \\
a_{m1}x_1 & + & a_{m2}x_2 & + & \cdots & + & a_{mn}x_n & = & b_m
\end{array}
$$

$$
\begin{array}{ccccccc}
l_{1,1}x_1 & & & & & = & b_1 \\
l_{2,1}x_1 & + & l_{2,2}x_2 & & & = & b_2 \\
\vdots & & \vdots & & \ddots & & \vdots \\
l_{m,1}x_1 & + & l_{m,2}x_2 & + \cdots + & l_{m,m}x_m & = & b_m
\end{array}
\qquad
\begin{aligned}
x_1 &= \frac{b_1}{l_{1,1}}, \\
x_2 &= \frac{b_2 - l_{2,1}x_1}{l_{2,2}}, \\
&\vdots \\
x_m &= \frac{b_m - \sum_{i=1}^{m-1} l_{m,i}x_i}{l_{m,m}}
\end{aligned}
$$

伪代码

高斯消元实现思想

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


内嵌第一个 for 循环的作用: 把第 k 行的所有元素除以第一个非零元素，使得第一个非零元为 1；第二个内嵌 for 循环作用：

从 k+1 行开始减去第 k 行乘以这一行的第一个非零元，使得 k+1 行的第 k 列为 0;

伪代码第第一个内嵌循环中的 $A[k,j] := A[k,j]/A[k,k]$ 以及双层 for 循环中的 $A[i,j] := A[i,j]-A[i,k]\times A[k,j]$ 都可以进行向量化, 通过 SIMD 扩展指令对这两步进行并行优化。

$SIMDIntrinsics$ 函数高斯消元向量化伪代码

高斯消元向量化实现思想

```
Data: 系数矩阵 A[n,n]
Result: 上三角矩阵 A[n,n]
for k = 0 to n−1 do
vt ← dupTo4Float(A[k,k]);
    for j = k + 1; j + 4 <= n; j+ = 4 do
        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
        va ← va/vt ; // 这里是向量对位相除
```

```
        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
    for j in 剩余所有下标 do
        A[k,j]=A[k,j]/A[k,k] ; // 该行结尾处有几个元素还未计算
        A[k,k] ← 1.0;
    for i ← k+1 to n−1 do
        vaik ← dupToVector4(A[i,k]);
    for j = k + 1; j + 4 <= n; j+ = 4 do
        vakj ← load4FloatFrom(&A[k,j]);
        vaij ← load4FloatFrom(&A[i,j]);
        vx ← vakj∗vaik;
        vaij ← vaij−vx;
        store4FloatTo(&A[i,j],vaij);
    for j in 剩余所有下标 do
        A[i,j] ← A[i,j] −A[k,j]∗A[i,k];
        A[i,k] ← 0;
```

（二） 编程实现

第一个内嵌 for 循环里的 A[k, j]:= A[k, j]/A[k, k] 做除法并行, 第二个双层 for 循环里的 A[i, j] := A[i, j] - A[i, k] × A[k, j] 做减法并行;

_mm_loadu_ps 用于 packed 的加载不要求地址是 16 字节对齐，对应指令为 movups;

_mm_sub_ps(___m128$_A$, ___$m128$_B); ___$m128$ _A _B 32bit _A ,

例如 _A=(_A0,_A1,_A2,_A3), _B=(_B0,_B1,_B2,_B3), 则返回寄存器为

r=(_A0*_B0,_A1,_A2,_A3);

_mm_storeu_ps(float *_V, ___m128_A); 返回一个 ___m128 的寄存器。

1. 测试样例生成

测试规模较小时，并行算法可能比串行算法还要耗时, 改变 cache 大小等系统参数，设计不同问题规模完成实验。

为避免出现极端情况计算结果可能会出现 Naf 或无穷等问题，生成测试用例，代码如下：

测试用例生成代码

```
float m[N][N];
void m_reset() {
    for(int i=0;i<N;i++) {
    m[i][j]=0;
    m[i][i]=1.0;
    for(int j=i+1;j<N;j++)
        m[i][ j]=rand();
    }
    for(int k=0;k<N;k++)
        for(int i=k+1;i<N;i++)
            for(int j=0;j<N;j++)
                m[i][ j]+=m[k][j];
}
```

2. 高精度时间测量

为较为精确测试程序运行时间，采用的时间测量函数代码如下:

高精度时间测量

```c
#include <stdio.h>
#include <time.h>
struct timespec sts,ets;
timespec_get(&sts, TIME_UTC);
// to measure
timespec_get(&ets, TIME_UTC);
time_t dsec=ets.tv_sec-sts.tv_sec;
long dnsec=ets.tv_nsec-sts.tv_nsec;
if (dnsec<0){
    dsec--;
    dnsec+=1000000000ll;
 }
printf ("%lld.%09llds\n",dsec,dnsec);
```

3. 程序编译运行

采用 SSE 高斯消元关键代码如下:

SSE 高斯消元关键步骤

```c
void SSE_LU(int n,float a[][maxN]){
    float temp;
    __m128 div,t1,t2,sub;
    for(int i = 0;  i < n - 1;  i++){
        for(int j = i + 1; j < n; j++){
            temp = a[j][i]/a[i][i];
            div = _mm_set1_ps(temp);
            int k = n - 3;
            for(;k >= i + 1; k-= 4){
            t1 = _mm_loadu_ps(a[i] + k);
            t2 = _mm_loadu_ps(a[j] + k);
            sub = _mm_sub_ps(t2,_mm_mul_ps(t1,div);
            _mm_store_ss(a[j] + k,sub);
            }
            for(k += 3; k >= i + 1;k--){
            a[j][k] -= a[i][k] * temp;
            }
        }
    }
}
```

详细代码请见: **SSE 高斯消元** 请点击这里

截图示例如下:

```
C:\Users\HONOR\source\repos\GaussSSE\Debug\GaussSSE.exe
size: 128
串行消元时间：3.98789ms
串行回代时间：0.097318ms
并行消元时间：2.50938ms
并行回代时间：2.37967ms

size: 256
串行消元时间：28.2208ms
串行回代时间：0.333174ms
并行消元时间：14.8488ms
并行回代时间：14.8559ms

size: 384
串行消元时间：95.2211ms
串行回代时间：0.731958ms
并行消元时间：43.6867ms
并行回代时间：42.7688ms

size: 512
串行消元时间：270.151ms
串行回代时间：1.27824ms
并行消元时间：92.7096ms
并行回代时间：90.0778ms

size: 640
串行消元时间：583.282ms
串行回代时间：2.19922ms
并行消元时间：226.168ms
并行回代时间：223.026ms
```

```
size: 896
串行消元时间：1390.46ms
串行回代时间：4.51844ms
并行消元时间：613.974ms
并行回代时间：604.599ms

size: 1024
串行消元时间：2044.58ms
串行回代时间：5.74364ms
并行消元时间：938.607ms
并行回代时间：978.561ms

size: 1152
串行消元时间：2749ms
串行回代时间：6.98456ms
并行消元时间：1342.47ms
并行回代时间：1380.04ms

size: 1280
串行消元时间：4002.24ms
串行回代时间：8.52873ms
并行消元时间：1870.4ms
并行回代时间：1867.29ms


C:\Users\HONOR\source\repos\GaussSSE\Debug\GaussSSE.exe (进程
```

表

矩阵规模	10	100	512	1000	1024
串行	0.0030	1.8973	343.542	1794.62	2707.7
并行	0.8534	22.5583	156.443	1831.7	1471.66
加速比	0.003515	0.741625	0.979757	2.195956	11.839895

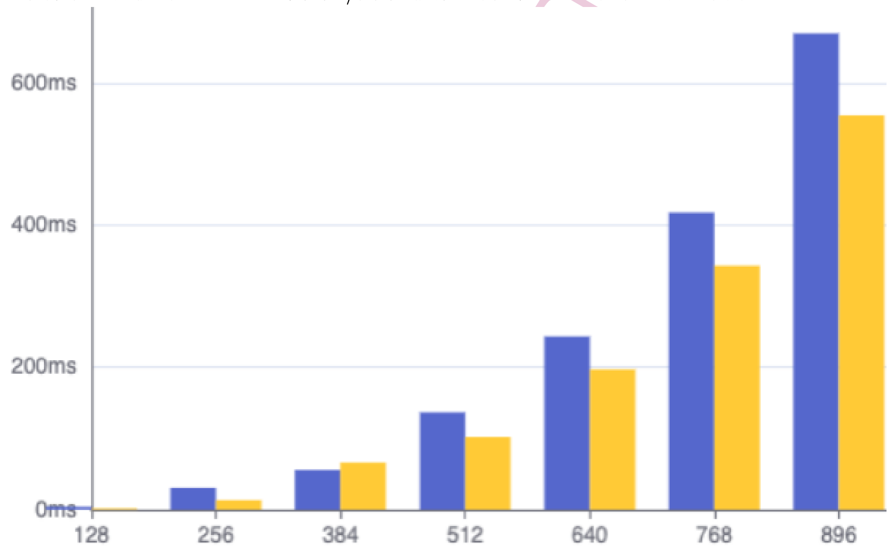
表 1: 不同矩阵规模下串行 SSE 并行运行时间对比 (单位:ms)

实验数据均为 5 次结果取平均值。

在运算的数据较少的时候，并行算法和串行算法的速率几乎没有区别，串行算法甚至比并行算法快。但是随着数据量的增长，明显并行速度要比串行的速度快很多。注意其中的数据量为 512 和 1024 的位置，要比之前快的还要明显，当数据量为 4 的整数倍的时候，SSE 并行算法发挥的用更大一些。

总结：此并行算法对串行算法进行了性能上的优化，在数据量越来越多的时候，优化的效果也越来越好，并且当数据量为 4 的整数倍的时候，优化效果明显。

不同问题规模下加速比（串行/并行执行时间）性能图形对比展示：



SSE 算法的消元总体来说是运行效率是高于串行算法的, 且随着矩阵规模增大而增大。

SSE 算法的回代竟比串行算法的回代运行效率要慢, 猜想其中可能一部分原因是因为使用的 SSE 算法回代过程中转置过程本身就比较耗费时间。

不同算法/编程策略对性能的影响:

高斯消去法中两个部分可以进行向量化，对比这两个部分 (一个二重循环、一个三重循环) 进行 SIMD 优化对程序速度的影响:

相同算法对于不同问题规模的性能提升有一定的影响，但是影响效果不大，消元过程中采用向量编程的的性能有提高，但效果不是太明显，回代过程也可以向量化，但是性能不提反降，总体来说，SSE 算法对运行效率还是有一定程度的提升, 可继续优化。

编译实现

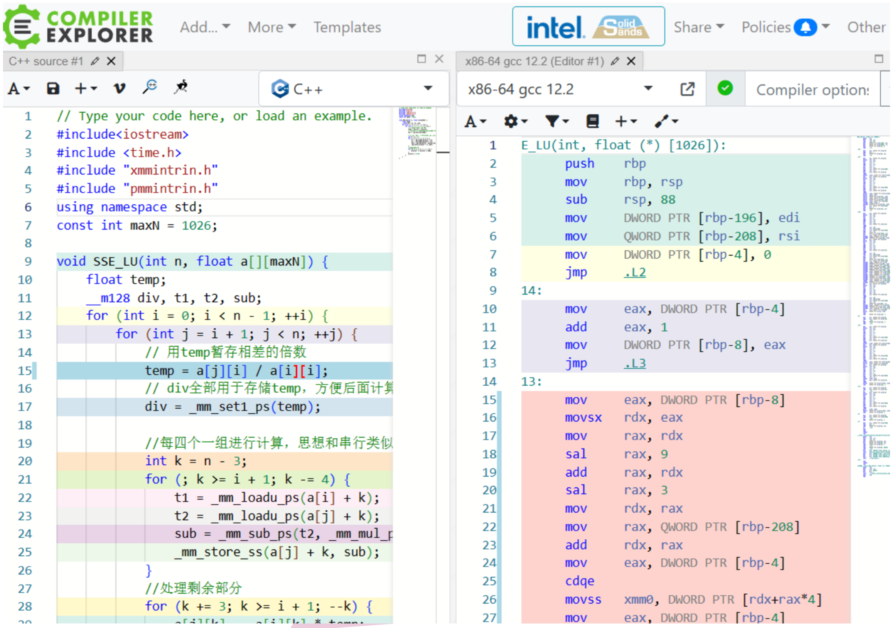
```
$ gcc column_sum.c -g -o column_sum
 # perf record -e L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores -ag
     ./row_column
```

```
# perf report
```

涉及到二维数组或三维数组的多次访问，各个维度的下标访问顺序会严重的影响运行时间（尤其当 N 足够大时），若访问时间过长，主要是因为访问顺序与存储顺序不同，因此在高速缓冲中命中的数量大大减小，导致不在高速缓冲中取数据而是去硬盘当中取，则最终影响了运行时间。

4. godbolt 汇编

godbolt 分析 SSE_LU() 函数得到的汇编代码如下：



为提高执行效率，应使用翻译成汇编语句少的代码，注意：高级语言代码简洁，汇编语句不一定少，时间复杂度不一定低。

5. 手工编写与编译器自动向量化

手工编写的 SIMD 程序可以根据具体的应用场景进行优化，通常可以实现更加紧凑和高效的代码。但是需要手动处理数据对齐、访问、加载和存储等问题，编写起来相对较为繁琐。由于不同的处理器架构和指令集有所不同，需要编写多个版本的代码，以便充分利用硬件资源。

编译器自动向量化版本可以自动将标量代码转换为 SIMD 代码，充分利用硬件资源，但是其性能通常不如手工编写的 SIMD 程序。编译器对代码的转换质量和效率有一定的影响，有时候可能会出现性能瓶颈或者意外的行为。有时编译器无法充分理解应用程序的语义，有些特定的优化可能需要手动实现。

（三） arm 平台串行 NEON 并行高斯消元

1. 编程实现

arm 平台串行高斯消元 请点击这里

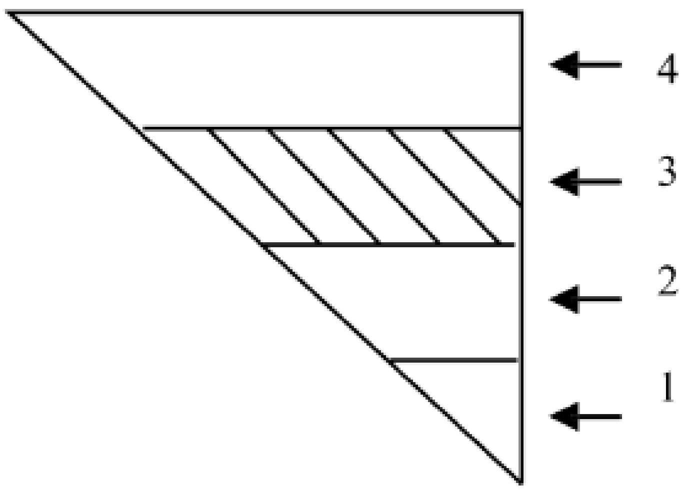
arm 平台 NENO 高斯消元 请点击这里

2. 编译执行

其某次结果如下:

```
[ss2110957@master test]$ ll
total 48
-rw-r--r--. 1 ss2110957 student  1106 Apr 14 15:05 arm_串行.cpp
-rw-r--r--. 1 ss2110957 student  1786 Apr 14 14:29 Neon.cpp
-rwxr-xr-x. 1 ss2110957 student 71704 Apr 14 14:33 test
-rwxr-xr-x. 1 ss2110957 student 71712 Apr 14 15:08 test1
[ss2110957@master test]$ ./test
 ParallelAlgorithm time: 7.98596 ms
[ss2110957@master test]$ ./test1
 SequentialAlgorithm time: 6.725 ms
[ss2110957@master test]$ 
```

3. 分析优化



Cache 分析

A 从下至上被分成了 4 块，图中的数字为各个块的的编号，每块包含的矩阵行数相等 (最后一块的行数根据分块策略可能会出现较少的情况)。

原始从矩阵 A 的最下面一行开始往上执行优化后从任务块开始从下往上执行，每个任务块内部，采取按列从右往左的方式执行，每一列按从下往上的方式执行。

假设编号为 3 的任务块包含的矩阵行数为 N，其中编号为 i 的列中含有 Y 个元素 1(行号为 noi、nui、nyi), 程序任务执行到这一列时, 异或操作代码为 pBdata[i]=pBdata[i]pBaddil(ixno、n1i、ny1i)，对于矩阵 B 中第 i 行将有 Y 次的复用; 对于任务块中的第 k 行，含有 X 个元素 1(列号为 mok、mk、···、mx-1,k), 程序中对于这一行, 异或操作代码 pBdata[k]=pBdata[k]pBadd[il(ixmok、m1.k、mx-1.k)，对于矩阵 B 中第 k 行将有 2X 次的复用。

在矩阵 A 的任务分块策略中，N 太小，数据复用不够，N 太大 cache 命中率降低，N 的取值通过 cache 缓存大小计算结合实验测试所得。利用算法中数据的重用性，大大增加了 cache 缓存命中率，减少了数据从内存由访存数据的次数，提高了数据访存性能。

对齐与不对齐

arm 架构是否对齐：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，探究对齐 NEON 指令与未对齐性能差异对

于对齐的数据访问方式，NEON 指令可以使用一些特殊的指令进行高效地处理，比如 VLD1 和 VST1 指令，这些指令可以将多个数据同时加载到寄存器中进行处理，从而提高处理效率。此外，对齐的数据访问方式还可以更好地利用硬件缓存，从而进一步提高性能。而对于未对齐的数据访问方式，NEON 指令需要使用一些额外的指令进行处理，比如 VLD1_UND 和 VST1_UND 指令，这些指令会额外增加处理开销，降低性能。此外，未对齐的数据访问方式还可能会导致一些不可预测的行为，比如数据对齐不正确可能会导致程序崩溃。

x86 架构是否对齐:

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，调整算法，先串行处理到对齐边界，然后进行 SIMD 的计算，对比两种方法的性能。

定义一个包含大量浮点数的数组，这个数组的长度可以根据需要进行调整。分别编写两个函数，一个使用 _mm_loadu_ps 进行未对齐的计算，另一个使用 _mm$_{l}oad_{p}s$

在未对齐的函数中，直接使用 _mm_loadu_ps 加载浮点数数组，然后用 SSE 指令集中的 _mm_add_ps 进行加法计算。

在对齐的函数中，先使用串行方式处理数组中的前几个元素，直到数据对齐边界，然后使用 _mm_load_ps 加载对齐的数据，再使用 _mm_add_ps 进行 SIMD 计算。

使用时间函数，比较这两个函数的执行时间

结论：使用 _mm_loadu_ps 指令进行未对齐的数据加载和计算，则可以直接处理任意地址的数据。由于需要额外的指令进行未对齐的数据访问，性能可能受到一定的影响。如果使用 _mm_load_ps 指令进行对齐的数据加载和计算，则需要保证数据的地址是 16 字节对齐的可以利用 SSE 指令集的特殊指令进行高效的 SIMD 计算，性能通常会比未对齐的方式更好。

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。

二、 特殊高斯消元 SIMD 并行

(一) 特殊高斯消元 (Grobner 基) 实现思想

高斯消元模块分两种行：消元子、被消元行（被消元多项式）

其中消元子为根据系统中存储的多项式和其倍数多项式逐步导入，保证消元子首项各异但不能保证以每一项为首项的消元子都存在。其在高斯消元模块只充当“减数”而不会充当被减数，在高斯消元模块可以认为消元子已知。

被消元行在高斯消元过程中既充当“被减数”又充当“减数”，导入高斯消元模块的被消元行通常不是首项各异的。

高斯消元时，可视为消元子和被消元行合并为一个矩阵，该矩阵经过高斯消元之后非零行首项各异，消元结束后返回所有被消元行的消元结果。

(二) 编程实现

特殊高斯消元串行伪代码

```
for i := 0 to m − 1 do
    while E[i]! = 0 do
        if R[lp(E[i])]! = NULL then
            E[i] := E[i] − R[lp(E[i])]
        else
```

```
            R[lp(E[i])] := E[i]
            break
        end if
    end while
end for
return E
```

串行算法伪代码，其中被消元行有 m 行，消元子最大首项为 t, 其中外层循环表示遍历每个被消元行。内层循环表示针对每个被消元行，如果该行未被消为 0，那么根据其首项选择消元子进行消元；当存在合适的消元子，则用该消元子进行消元；否则将该被消元行作为消元子，参与后续高斯消元过程。

特殊高斯消元并行伪代码

$\text{Todo} \leftarrow [\,[1,\ldots,K],\ [\,],\ [\,]\,]$, $\text{Done} \leftarrow [\,]$
$\text{Function} \leftarrow [\texttt{Trsm}, \texttt{Axpy}, \texttt{Gauss}]$, $\text{Pr} \leftarrow [3,2,1]$

```
/* Something to do
while Done ≠ [1,...,K] do
    /* search a task from high to low priority
    for i ← 1 to 3 do
        if Todo_Pr[i] ≠ [] then
            Lock()
            ind ← Todo_Pr[i][1]
            Todo_Pr[i] ← Todo_Pr[i] \ [ind]
            Unlock()
            /* The computation is performed
            Function[i](ind)
            Lock()
            if i ≤ 2 then
                /* Next operation must be performed on
                   this block
                Todo_Pr[i+1] ← Sort(Todo_Pr[i+1] ∪ [ind])
            else
                /* All the operations are done
                Done ← Done ∪ [ind]
            Unlock()
```

（三） armx86 架构不同指令集 (SSEAVXAVX-512Neno) 高斯消元影响

X86 平台的 SSE 指令集，支持 128 位的 SIMD 计算，可以实现对四个单精度浮点数或两个双精度浮点数的并行处理。在高斯消去的并行化实现中，利用 SSE 指令集来实现多个方程的并行计算，但是 SSE 指令集的并行性有限，只能同时处理有限个数据，可能无法满足大规模计算的需求。

AVX 指令集，支持 256 位的 SIMD 计算，可以实现对八个单精度浮点数或四个双精度浮点

数的并行处理。AVX-512 指令集，支持 512 位的 SIMD 计算，可以实现对十六个单精度浮点数或八个双精度浮点数的并行处理，ARM 平台的 NEON 指令集，它支持 64 位和 128 位的 SIMD 计算。

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。