南開大學
Nankai University

计算机学院

并行程序设计实验报告

SIMD 编程实验

姓名：黄天昊

学号：2011763

专业：计算机科学与技术

2023 年 4 月 16 日

目录

1 问题描述

问题围绕高斯消去展开：首先对所需要实现的算法进行复杂度分析；

2 分工情况

本次实验由王旭尧（2012527）和黄天昊（2011763）共同完成。我们共同设计了普通高斯消元法和特殊高斯消元。对于不同情况下的优化，我们分别设计了各自的实验，并进行了测试。

接下来实现了普通高斯消去的 SIMD 并行化实验，包括串行/并行对比，x86 平台/ARM 平台对比，对齐/不对齐对比，cache 优化的前后对比，对串行中除法/消去的优化对比，不同问题规模对比，编译器不同优化力度对比等。然后设计算法并实现，最后对实验结果进行分析；

最后实现了特殊高斯消去（消元子模式）的 SIMD 并行化实验，包括串行/并行对比，不同编程语言 (Neon,SSE,AVX) 对比，x86 平台/ARM 平台对比，对齐/不对齐对比，不同问题规模对比，编译器不同优化力度对比等。然后设计算法并实现，最后使用 Vtune 和 perf 测量 cycles，instructions 和 CPI 等性能指标对实验结果进行分析。

3 普通高斯消元

3.1 算法设计

串行算法： 首先对普通高斯消去的串行算法进行复杂度分析：(i)3-5 行代码复杂度为 O(N);(ii)7-12 行复杂度为 O(N2);(iii) 最外层循环为 n 次。综上所述，该算法的复杂度为 O(N3)。
串行算法按普通高斯消去思路即可得到，伪代码为：

Algorithm 1 普通高斯消去串行算法思路

Input: 待消去矩阵 A，矩阵行、列数均为 n
Output: 消元后矩阵 A

```
1:  function LU
2:      for k := 1 to n do
3:      for j := k + 1 to n do
4:      A[k,j] := A[k,j]/A[k,k];
5:      endfor;
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]xA[k,j];
10:     endfor;
11:     A[i,k] := 0;
12:     endfor;
13:     endfor;
14: return
15: end function
```

对 3-4 行部分我们称之为除法，对 7-9 行部分我们称之为消去，下面提到的不同部分优化基于这两部分。

并行算法 (Neon)： 首先对普通高斯消去的并行算法进行复杂度分析：(i) 原串行代码 3-5 行代码复杂度为 O(N)，并行化后为 O(N/4);(ii) 原串行代码 7-12 行复杂度为 O(N2)，并行化后为 O(N2/4);(iii) 最外层循环为 n 次。综上所述，该算法的复杂度量级为 O(N3)，具体是串行算法的 1/4。
可以对串行算法的除法和消去部分优化。将原有数据类型修改为 float32_t、float32x4_t，并将原循环中 j++ 改为 j+=4，使用 vld1q_f32 和 vmovq_n 装载入向量寄存器，使用 vdivq_f32 做向量除法，使用 vmulq_f32 做向量乘法，使用 vsubq_f32 做向量减法，使用 vst1q_f32 将向量寄存器的值载回数组。具体代码如下：

```
//除法向量化
float32x4_t vt = vmovq_n_f32(A[k][k]);
for(;j + 4 <= maxN; j += 4){
        float32x4_t va = vld1q_f32(&A[k][j]);
        va = vdivq_f32(va,vt);
        vst1q_f32(&A[k][j],va);
}
//消去向量化
float32x4_t vaik = vmovq_n_f32(A[i][k]);
for(;j + 4 <= maxN;j += 4){
        float32x4_t vakj = vld1q_f32(&A[k][j]);
        float32x4_t vaij = vld1q_f32(&A[i][j]);
        float32x4_t vx = vmulq_f32(vakj,vaik);
        vaij = vsubq_f32(vaij,vx);
        vst1q_f32(&A[i][j],vaij);
}
```

对于剩余无法做向量化的部分，使用普通串行算法解决。

测试用例： 要避免计算结果出现 Naf 或无穷的情况，且要尽量大，因为测试规模较小时可能并行比串行还耗时。本实验中我们采用 500，1000，2000，3000 作为测试样例大小，测试样例生成代码见实验指导书。

不同部分优化： 分别对除法和消去两个部分进行并行化，测量哪部分并行化的收益更大。

对齐/不对齐： 由于普通高斯消去中的 Neon 支持不对齐算法，因此我们探究对齐 Neon 指令与未对齐性能差异。对齐方法为：由于 C++ 中数组的初始地址一般为 16bytes 对齐，因此加载数据时确保第一个待取的数在数组中的相对位置是 4 的倍数即可。对齐部分代码如下，其中 k 为待处理元素所在行，maxN 为矩阵列数，j 为待处理元素所在列，$(k * maxN + j)$ 表示该元素相对数组首地址的偏移，当该偏移不是 4 的倍数时进行串行简单处理，直到成为 4 的倍数时开始并行处理。

```
while((k * maxN + j) % 4 != 0){//do the alignment
        A[k][j] = A[k][j] * 1.0 / A[k][k];
```

```
3            j++;
4    }
```

X86/ARM 平台： x86 平台选择 dev cloud，ARM 平台选择鲲鹏服务器，分别进行实验。

3.2 结果：对比及分析

3.2.1 串行/并行

串行与并行在不同数据规模下的对比如图3.1所示。可以看到，在所有问题规模下并行算法都优于串行算法，体现了并行计算的强大优化能力。

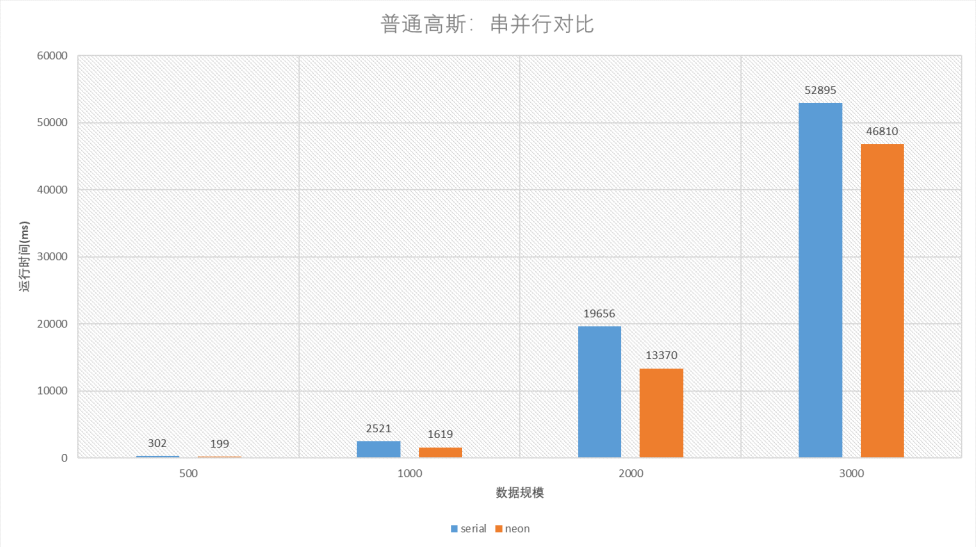

图 3.1: 串并行对比

我们使用 Vtune 和 perf 对串并行性能进行剖析，结果如图3.5所示。可以看到，并行算法的 Cycles、CPI 偏高，Instructions 偏低。这是由不同平台造成的，由前面实验可知，ARM 平台的性能低于 X86 平台。

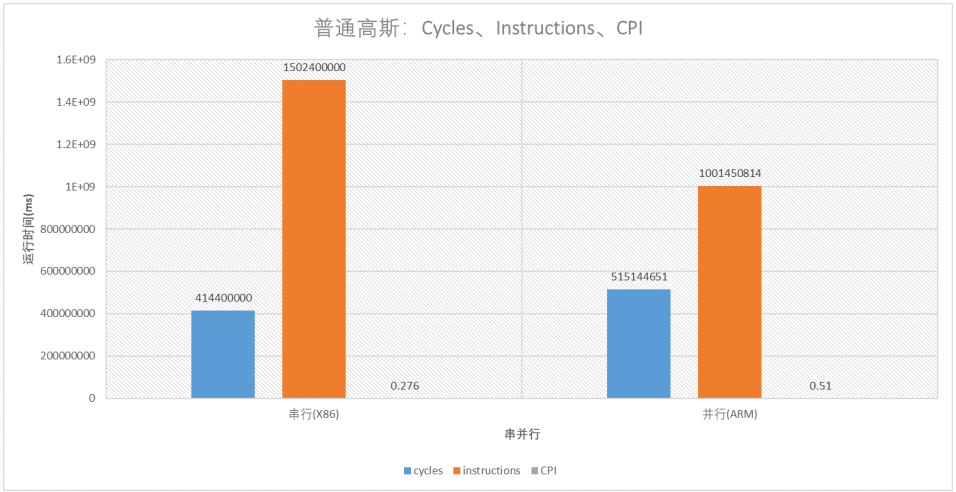

图 3.2: 串并行性能剖析

3.2.2 除法/消去优化

选择不同部分优化在不同数据规模下的对比如图3.3所示。可以看到，对消去的优化效果高于对除法的优化。我认为，原因是：(1) 除法向量化中，有 load、除法、store 三条指令，其中 load 和 store 为相对串行的额外指令，只有除法负责提升效率；而 (2) 消去向量化中，有 load、mul、sub、store 指令，其中 mul 和 sub 都负责提升效率。因此，消去部分提升效率的代码占比更多，故消去优化效果更好。

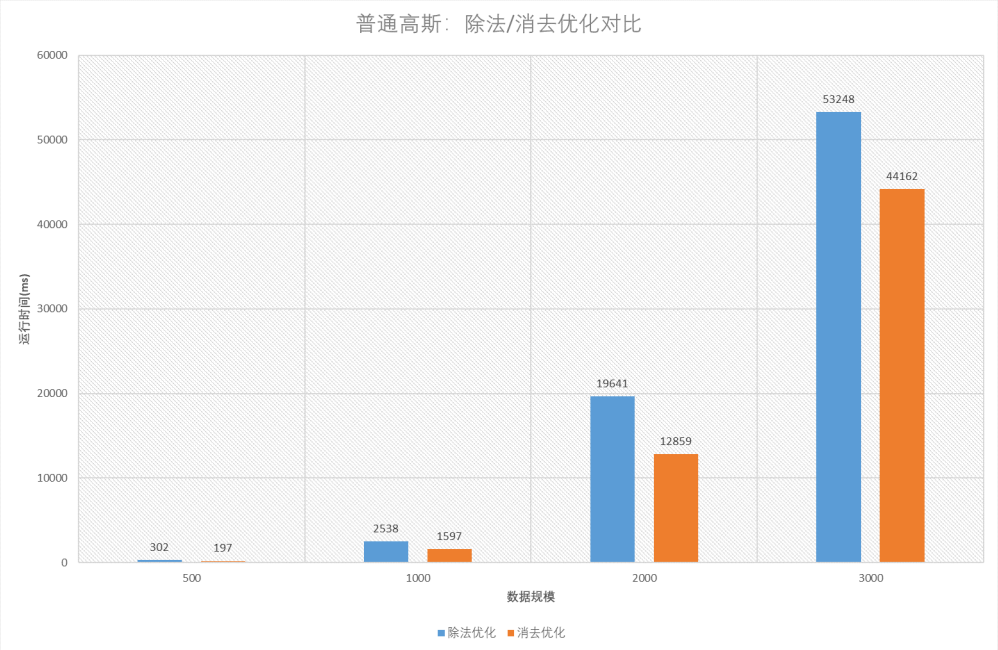

图 3.3: 除法/消去优化

3.2.3 对齐/不对齐

对普通并行高斯消去做对齐，在不同数据规模下的对比如图3.4所示。可以看到，对齐的效果较好，提高了近 10%的效率，可以在并行化的基础上进一步提高性能。

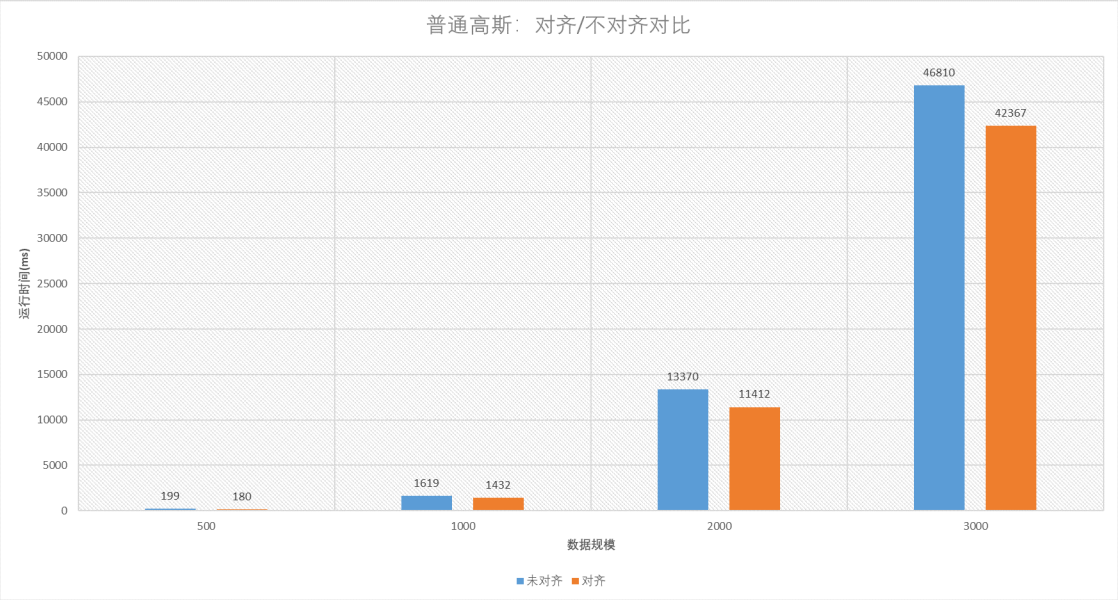

图 3.4: 对齐/不对齐

我们使用 Vtune 和 perf 进行性能分析，结果如图3.5所示。可以看到，对齐后的算法 Cycles、Instructions 和 CPI 均低于未对齐算法，性能较好。

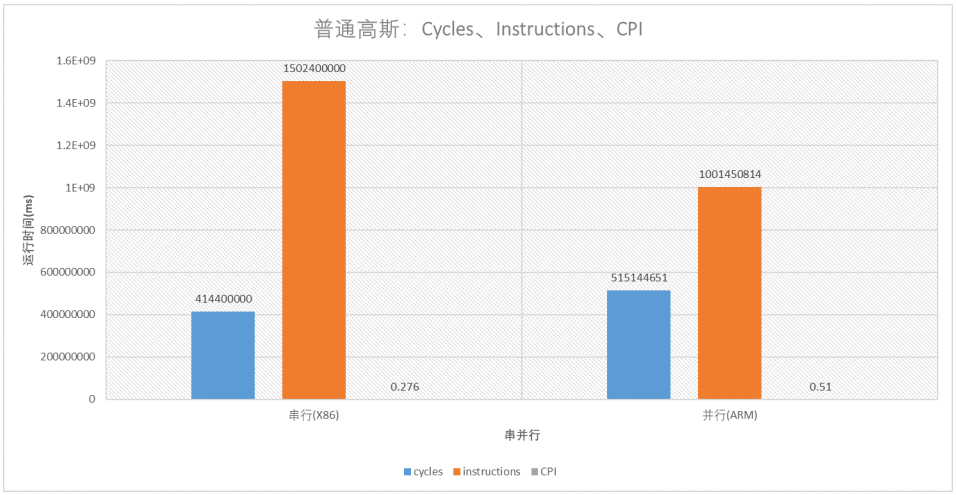

图 3.5: 对齐/不对齐的性能剖析

3.2.4 x86/ARM

在 x86 平台和 ARM 平台运行不同数据规模的算法，结果如图3.6所示。可以看到，在所有规模下 x86 平台的表现都优于 ARM 平台，节约了近 30%的时间。

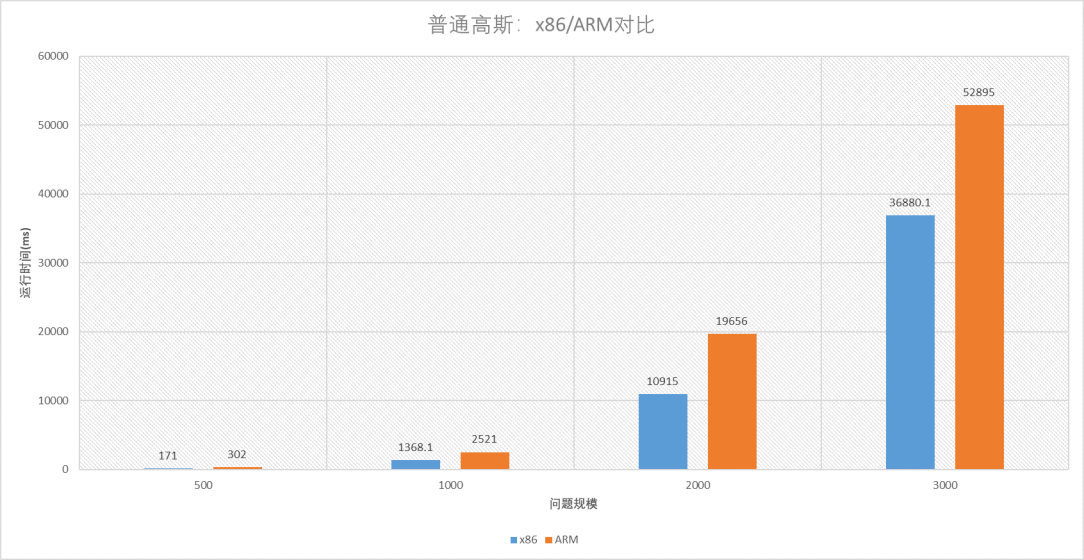

图 3.6: x86/ARM

4 特殊高斯消元

4.1 算法设计

串行算法： 串行算法的伪代码如下。首先对串行算法进行复杂度分析。(i) 第 4 行代码复杂度主要产生在判断消元子是否为空，复杂度为消元子/被消元行列数 N/32 即 O(N/32);(ii) 第 5 行代码复杂度为 O(1);(iii) 第 7 行代码复杂度为 O(N/32);(iv) 每次执行 if 语句后需要修改 lp(E[i])，复杂度为 O(N/32 + 32)=O(N);(v)while 循环整体复杂度为 O(N);(vi) 整个算法的复杂度为 O(MN)。（M 为被消元行行数，N 为列数）

注意伪代码中 E[i] 指第 i 个被消元行，R[i] 指首项为 i 的消元子（首项指某行下标最大的非零项的下标），lp(E[i]) 指被消元行第 i 行的首项。

Algorithm 2 特殊高斯消去串行算法思路

Input: 被消元矩阵 E，消元子矩阵 R

Output: 被消元后矩阵 E

```
1:  function SPECIAL GAUSS
2:      for i := 1 to m do
3:      while E[i] != 0 do
4:      if R[lp(E[i])] != NULL then
5:      E[i] := E[i] xor R[lp(E[i])]
6:      else
7:      R[lp(E[i])] := E[i]
8:      break
9:      end if
10:     end while;
11:     end for;
12: return E
13: end function
```

在将伪代码转为串行代码的过程中，我设置了几个函数：

1. Find_First 函数负责找出指定消元行的第一个 1 所在位置；
2. Init_R 负责从文件读入数据初始化消元子；
3. Init_E 负责从文件读入数据初始化被消元行；
4. Is_NULL 负责判断消元子是否为 null，对应伪代码中第 4 行 if 判断语句；
5. Set_R 负责被消元行到消元子的“升格”过程，对应伪代码第 7 行；
6. XOR 负责最基本的异或运算，对应伪代码第 5 行；
7. Serial 为主要处理函数，对应整个伪代码。

此外，我还使用了一个数组 First 存储 E 中第一个 1 所在位置，在每次执行 if 语句后更新，当值为-1 时表示该行已经全 0，使用于 while 的循环判断条件。
需要特别注意消元子和被消元行的存储方式，尤其是各 bits 究竟是如何存储的，并推广计算得到公式。计算式可在各特殊高斯的.cpp 文件开头变量声明处看见。具体代码见 github 中 Special Serial 文件夹。

并行算法 Neon： 首先对特殊高斯消元的并行 Neon 算法进行复杂度分析。和串行相比，Neon 没有在量级上改变复杂度，复杂度量级仍为 O(MN)，但将具体复杂度减少至原来的 1/4。

可以对串行算法的除法和消去部分优化。将原有数据类型修改为 float32_t、float32x4_t，并将原循环中 j++ 改为 j+=4，使用 vld1q_f32 和 vmovq_n 装载入向量寄存器，使用 vdivq_f32 做向量除法，使用 vmulq_f32 做向量乘法，使用 vsubq_f32 做向量减法，使用 vst1q_f32 将向量寄存器的值载回数组。具体代码如下：

```
//除法向量化
float32x4_t vt = vmovq_n_f32(A[k][k]);
for(;j + 4 <= maxN; j += 4){
        float32x4_t va = vld1q_f32(&A[k][j]);
        va = vdivq_f32(va,vt);
        vst1q_f32(&A[k][j],va);
}
//消去向量化
float32x4_t vaik = vmovq_n_f32(A[i][k]);
for(;j + 4 <= maxN;j += 4){
        float32x4_t vakj = vld1q_f32(&A[k][j]);
        float32x4_t vaij = vld1q_f32(&A[i][j]);
        float32x4_t vx = vmulq_f32(vakj,vaik);
        vaij = vsubq_f32(vaij,vx);
        vst1q_f32(&A[i][j],vaij);
}
```

对于剩余无法做向量化的部分，使用普通串行算法解决。Neon 的总体代码见 github 中 Special Neon 文件夹。

并行算法 SSE： SSE 的并行化思路与 Neon 类似，只是在具体代码实现上有区别，其向量声明为 _m128，装载指令为 _mm_loadu_ps（对齐的为 _mm_load_ps），异或指令为 _mm_xor_ps，存储指令为 _mm_storeu_ps（对齐的为 _mm_store_ps）。SSE 的总体代码见 github 中 SSE 文件夹。
经查阅，本算法中使用的 SSE 指令性能如图4.7所示：
将 SSE 与 AVX 指令性能对比后可知，SSE 的 loadu 指令性能高于 AVX，xor 指令低于 AVX。

Architecture	Latency	Throughput (CPI)
Skylake	6	0.5
Broadwell	1	0.5
Haswell	1	0.5
Ivy Bridge	1	1

(a) _mm_loadu_ps

Architecture	Latency	Throughput (CPI)
Broadwell	1	1
Haswell	1	1
Ivy Bridge	1	1

(b) _mm_xor_ps

Architecture	Latency	Throughput (CPI)
Skylake	5	1
Broadwell	1	0.5
Haswell	1	0.5
Ivy Bridge	1	1

(c) _mm_storeu_ps

图 4.7: SSE 各指令性能

并行算法 AVX： AVX-512 的并行化思路与前两者类似，只是在具体代码实现上有区别，其向量声明为 _m512，装载指令为 _mm512_loadu_ps(对齐的为 _mm512_load_ps)，异或指令为 _mm512_xor_ps，存储指令为 _mm512_storeu_ps（对齐的为 _mm512_store_ps）。需要提出的是，Neon 和 SSE 都采用的 4 路向量化方法，在 AVX-512 中则需要改为 16 路向量化方法，即循环时由 $j+=4$ 改为 $j+=16$。AVX-512 的总体代码见 github 中 AVX 文件夹。
经查阅，本算法中使用的 AVX 指令性能如图4.8所示：
将其于 SSE 指令性能对比后可知，AVX 的 loadu 指令性能低于 SSE，xor 指令高于 SSE。

Architecture	Latency	Throughput (CPI)
Icelake	8	0.5
Skylake	8	0.5

(a) _mm512_loadu_ps

Architecture	Latency	Throughput (CPI)
Icelake	1	0.5

(b) _mm512_xor_ps

Architecture	Latency	Throughput (CPI)
Skylake	5	1

(c) _mm512_storeu_ps

图 4.8: AVX 各指令性能

测试用例： 测试用例采用鲲鹏服务器中标准用例 2_254_106_53、7_8399_6375_4535 和 10_43577_39477_54274。

编译器不同优化力度对比 在编译时选择-O0、-O1、-O2、-O3 优化，观察编译器不同力度下的运行时间。

4.2 结果：对比及分析

加入 Print() 函数打印特殊高斯消元结果，并与样例中 3.txt 对比，经对比结果正确。若需要验证，可以在主函数里找到被注释掉的 Print 函数，调用该函数则可得到消元结果。

4.2.1 串行/并行（不同编程语言）

串并行不同编程语言在不同规模下的运行时间如图4.9所示。在较小规模如 254 和 8399 规模下，并行效率高于串行，大约提升了 50%的性能，且 Neon 优于 AVX，AVX 优于 SSE；在较大规模如 43577

规模下，串行效率反而高于并行。

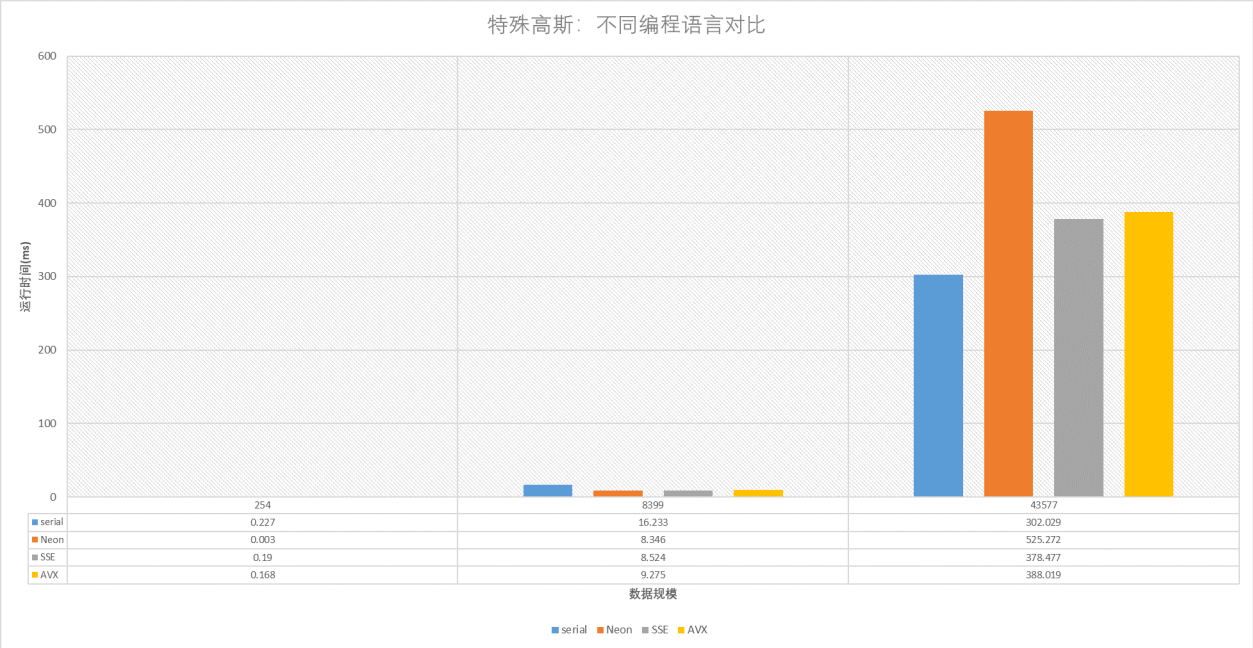

图 4.9: 不同编程语言

为进一步分析，我们使用 Vtune 和 perf 对串并行代码剖析其性能，结果如图4.10。可以看见，并行算法的 cycles、instructions 和 CPI 均高于串行算法，这也是为什么并行算法的普遍运行时间长于串行算法。

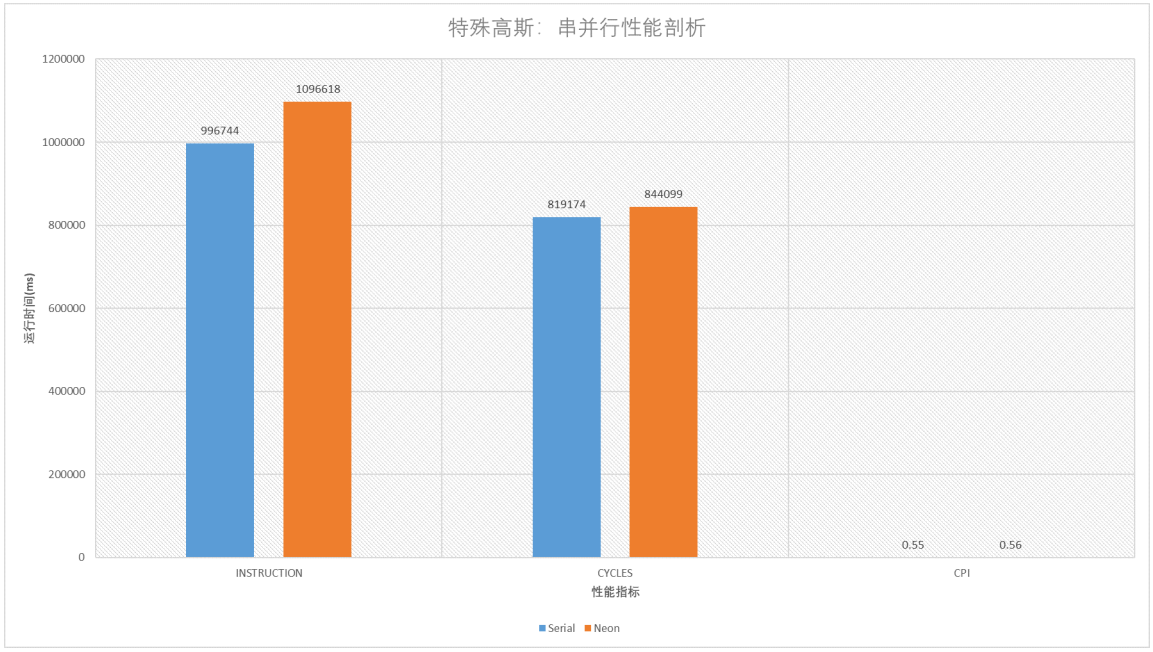

图 4.10: 不同编程语言

4.2.2 不同优化力度

不同编译器优化力度下的运行时间如图4.11所示。可以看到，O1 优化的效果最显著，将效率约提升了原来的 70%，与普通高斯相似。O2、O3 的算法优化不如 O1 显著，且在 AVX 的编译中出现 O2 优化的效率反而低于 O1 优化的情况。

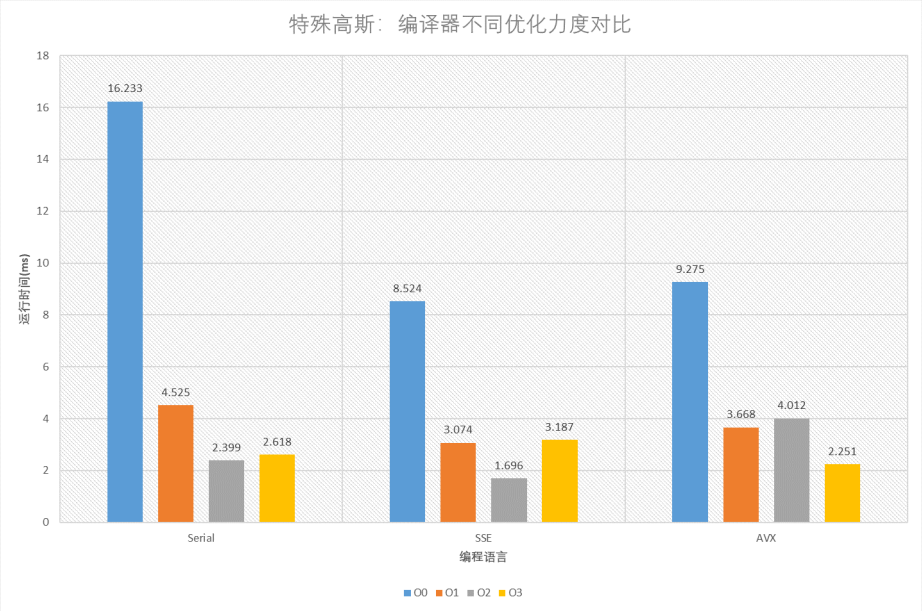

图 4.11: 不同优化力度

5 其他

5.1 总结

在本次实验中，首先对求解的问题进行了复杂度分析，然后使用串行语言、并行 Neon 语言、并行 SSE 语言、并行 AVX 语言求解了普通高斯消去和特殊高斯消去问题，并比较了 Cache 优化、不同部分优化、编译器优化、对齐优化、不同平台等因素，使用 Vtune 和 Perf 分析了性能。

5.2 代码

5.2.1 github 链接

https://github.com/XuyaoWang/nku_parallel_programming_2023spring/tree/main/SIMD

5.2.2 各文件夹说明

- 文件夹 Serial 指普通高斯的串行算法，Neon 指普通高斯的 Neon 算法；
- 文件夹 Special Serial 指特殊高斯的并行算法，Special_Neon 指特殊高斯的 Neon 算法，SSE 指特殊高斯的 SSE 算法，AVX 指特殊高斯的 AVX 算法；
- 在特殊高斯算法的几个文件夹中，EliminatedLine.txt 是被消元行的读入数据，Eliminator.txt 是消元子的读入数据。