第一次编程作业

计算机科学与技术
1511186
梁宸

February 25, 2020

1 简介

本次编程作业使用C++语言实现了基于AVX2指令集的SIMD版本的高斯消元法程序, 实验环境为intel i7, Linux Mint 18.3.

2 算法设计

2.1 高斯消元法

高斯消元法的串行版本作业文档中已经给出, 在此不多赘述. 可以将每行的操作分为两部分: 先将当前行对主元进行归一化; 再对以下各行进行消去.

2.2 AVX算法设计

本次作业的算法核心在于使用AVX的YMM寄存器进行向量化的浮点运算加速. YMM寄存器为256位, 即一次可装载8个单精度浮点数进行运算.

首先针对矩阵的存储, 选择按行映射, 即同一行的元素存在相邻的内存中[1].

其后的部分与串行的高斯消元法并无太大区别, 但对循环的顺序做了细微的改变: 把每一行的元素按8个分组, 枚举组, 做归一化, 然后对余下行的所有同组元素做消去. 这样需要先记录每一列的因子, 即第一个非零元素, 到一个临时内存区域中. 这样更改顺序的好处是, 能够针对后面元素个数不足8的组集中处理, 从而减少额外的内存访问次数[2].

[1]如果采用列主映射, 算法会有很大不同. 列主映射的归一化部分需要不连续的内存访问, 并且经过粗略计算, 两种方式的代价相差不大.

[2]在这个实现中, 针对不足8个float的组, 为了避免store操作污染, 我使用了一个临时的缓存区.

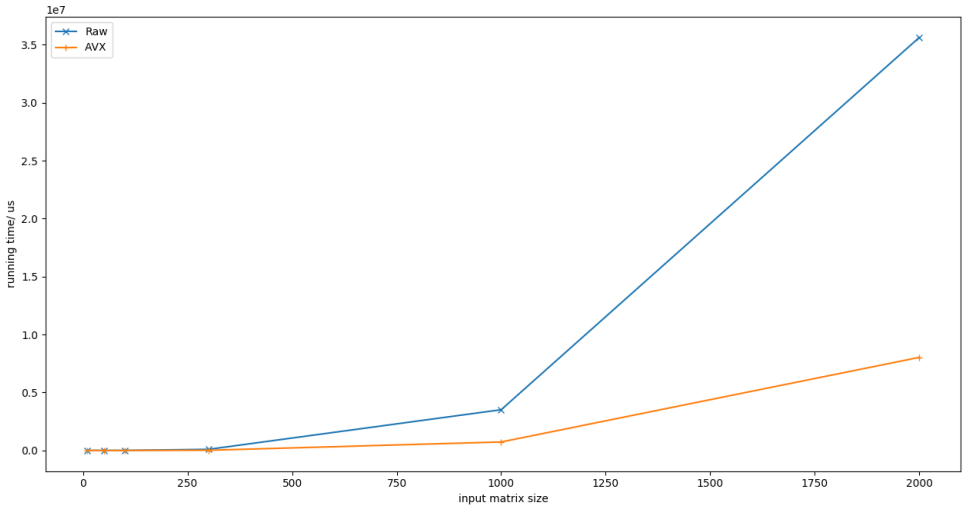
3 算法分析

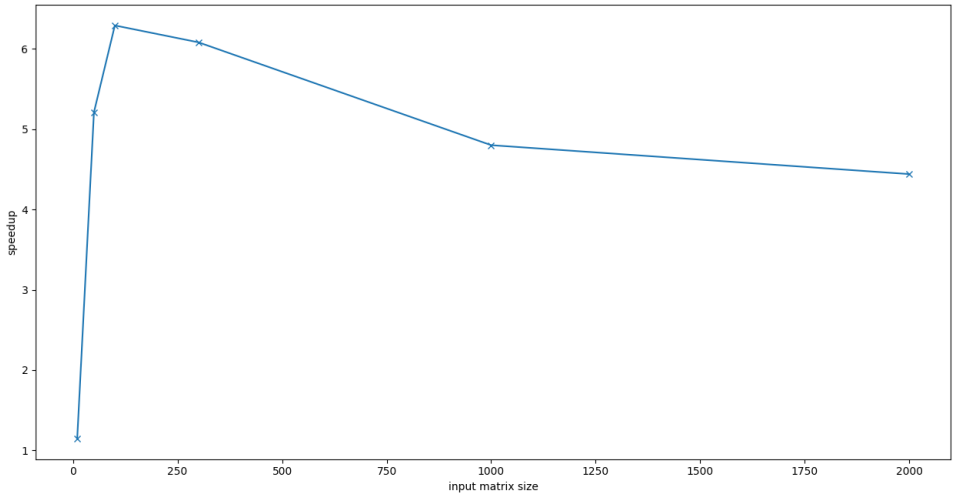
如不考虑缓存[3], 分析朴素的高斯消元法的时间复杂度, 其操作数为$\sum_{i=1}^{n}(n-i+1)+(n-i)^2$, 即有$O(n^3)$的时间复杂度. 而对于AVX版本的算法, 因其向量化特性, 可粗略地认为减少了矩阵的列数, 因此时间复杂度仍是$O(n^3)$, 但减少了常数因子, 比例取决于AVX运算指令与常规指令的指令周期之比. 更详细的分析, 对于第i行, 先需要进行一次AVX broadcast指令装载第一个非零元素. 之后对于该行的$\frac{n-i+1}{8}$个组, 每组需要进行一次AVX load指令, 一次div指令, 一次store指令来进行归一化. 此后对于余下的(n-i)行, 先需要记录每行的第一个非零元素, 每行的$\frac{n-i}{8}$个组, 需要进行一次load指令, 一次mul指令, 一次sub指令, 一次store指令, 来进行消除. 因此复杂度与上述粗略估计一致.

4 实验结果

测试矩阵规模选取10, 50, 100, 300, 1000, 2000共六组. 每组随机生成10个测试矩阵, 使用OS提供的gettimeoftoday函数获取运行时间, 取平均值. 测试结果如下:

[3]或认为整个矩阵都在缓存中.





其中, 每组串行版本的平均运行时间分别为4.5, 476.3, 3699.0, 95498.9, 3513668.4, 35632162.9微秒; AVX版本的运行时间分别为3.9, 91.5, 588.0, 15711.5, 731419.6, 8031495.0微秒. 对应的加速比为1.15, 5.21, 6.29, 6.08, 4.80, 4.44.

5 结果分析

从执行时间的绝对值来看, 基本可以认为AVX版本是全面快于串行版本的. 而从

加速比来看, 两个版本的算法加速比比呈单峰趋势, 可以分为三个阶段:

1. 在测试集规模很小时, AVX与串行版本的执行效率相近. 这是因为AVX的指令会有额外的打包和对齐等开销, 在矩阵很小时, 这些额外开销占据了执行时间的主要部分.

2. 在测试集规模适中时, SIMD算法的加速比达到最高. 这时AVX向量化指令带来的运算加速成为了主要因素, 理论上加速比能大概趋近8*(普通指令周期/AVX指令周期)[4].

3. 在测试集规模很大时, 加速比下降. 这是因为此时程序的瓶颈变成了内存而不是计算, 密集且跨度大的内存访问消耗了大部分的执行时间, 而计算部分占的部分很小, 因此AVX带来的计算加速也变得不重要了. 可以猜想当测试集规模趋近无穷时, 加速比会趋近于1.

[4]这需要有无限大的内存