基于并行编程的矩阵相乘算法实现

摘要

矩阵相乘是线性代数中一个基本数学问题，在基于SSE/AVX指令集的Intel-X86系列多核CPU/GPU的环境下（IntelSSE技术：一次实现4个float数相乘技术），就矩阵相乘并行算法对计算机性能的影响，从算法设计，实现，对比分析等角度进行研究。并行矩阵相乘算法应用十分广泛，随着深度学习，多核处理器计算机，分布式并行计算等领域的高速发展下，对计算要求便提出了很高的要求。

本文主要根据矩阵乘法从数学角度上就具有并行特点，针对矩阵乘法本身是每行与每列的对应相乘，而每个相乘的行与列没有直接关联，可知矩阵乘法具有并行设计特性，本文从矩阵相乘的并行算法角度来研究不同并行编程方法对核心处理器的性能影响，以及并行算法在不同环境（CPU:基于pthread多线程的矩阵相乘的并行算法；GPU：基于CUDA编程的并行算法C实现）并行下对程序性能测试。

关键词：矩阵相乘，并行计算，CUDA并行编程 ,多线程

第一章：引言

计算机在计算数学领域中矩阵相乘问题的常规计算方法主要从串行，并行，线程池三个角度来解决。但是随着高性能科学运算的要求提高，基于GPU的多核处理器，以及深度学习的高速发展。出现了CUDA编程，基于云计算分布式的并行计算等新技术来解决矩阵相乘问题。

就算法方面，解决矩阵相乘方法很多，从串行，并行，多线程，以及随着深度学习，CUDA编程的引入，算法从最初的基于分而治之思想的cannoon算法，到基于C++MPI编程的，以及基于GPU的CUDA编程的棋盘划分矩阵相乘算法等等。本文不多列举。

第二章：实验环境

操作系统：centos7/Windows10

物理内核数：8核心

编程语言：c/c++,GPU-CUDA语言

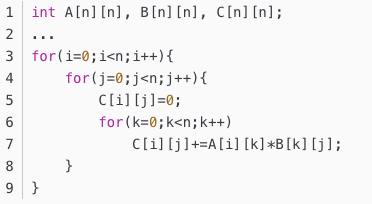
内存大小：8G

第三章：基于CPU实现矩阵相乘并行算法

1.串行，并行的并行化分析

在矩阵相乘问题中，我们可以得知其串行实现：根据线性代数的基本知识，的矩阵A，乘以一个大小为的矩阵B，将得到一个m\*n的矩阵C=A\*B，其中： 

如图1-1为常规串行实现的矩阵乘法代码：



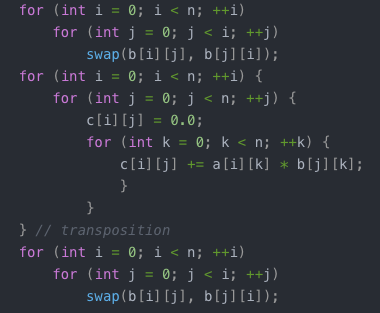
显而易见，该算法的复杂度为O（n^3）,其中n为矩阵的阶，随着n值增大，其计算效率就会受到影响。

根据矩阵乘法的定义特点可知，“结果矩阵”C的每个元素无依赖性，即每个元素均可参与独立计算，如果采用多核处理器实现并行计算，就会提高计算效率，同时提高程序性能。

根据上述代码显而易得：对于n\*n的矩阵中，处理器核心数为n,那么矩阵矩阵的每一行的计算都可通过一个处理器来处理。经计算得知：总并行计算步数为：O（n^2）.从编程的角度而言，外层循环for(i=0;i<n;i++)中分别由n个处理器并行执行，节点计算部分仅为内部两层循环，即for(k=0;k<n;k++),那么算法复杂度就变为O(logn).。因此可证矩阵相乘的并行化可行性为真。

2.并行算法的实现与优化

2.1 Cache优化

内存的本质是寄存器从内存中取值，即对内存寻址，在串行算法中可以发现B[k][j]需要先对矩阵B的行进行遍历操作。从编程角度，c++中，二维数组相邻的元素地址很近如果先做遍历操作，就会出现地址跳跃现象，即程序运行时间增长。如果将矩阵B进行转置操作，就会使得遍历矩阵实现连续寻址，此时就会节省寻址时间。从下图1-2所示在最内层循环，对于a和b的访问都是顺序的了，而这个循环中b[i][k]不变，这样就更好的利用了cpu cache。矩阵越大，这个加速效果越明显。

Cache优化的根本就是对初始算法：串行算法进行寻址方式/时间进行优化，我们可以从串行算法复杂度就可以知道，如果n值变大，矩阵就会变大，同时带来的I/O量也就会增大，当输入量增大，就会发生内存无法完全装入数据，这时候对内存刷新优化，修改矩阵循环的写法，从而改变寻址方式从随机非连续寻址变为连续寻址，即内存利用率最大。从代码看不够直观，下面我将举一个简单的例子：

我们都知道常规矩阵乘法的写法：

//A, B, C都是浮点数矩阵    
for (i=0; i<N; i++)  
for (j=0; j<N; j++)  
for (k=0; k<N; k++)  
C[i][j] += A[i][k]\*B[k][j];

这样写，本身没错，但是如果将N升维至1024，2048或者更高级数，就会产生cache conflict（高速缓存冲突）【computer system: a programmer's perspective第六章-做一个脚注】。那么如果交换k&j的位置，就会缩小寻址时间，即loop permutation（循环置换）即如图1-2的代码算法实现。实际上如果将利用分而治之的思想来解决，就会有新的写法：

for(it=0; it<N; it+=T)  
for(jt=0; jt<N; jt+=T)  
for(kt=0; kt<N; kt+=T)  
for(i=it; i<it+T; i++)  
for(j=jt; j<jt+T; j++)  
for(k=kt; k<kt+T; k++)  
C[i][j] += A[i][k]\*B[k][j];

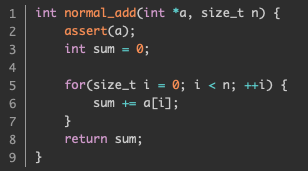
这个写法叫做loop tilling（循环分块），这种思想可以追溯到分而治之思想（关于分而治之思想的论文文献-脚注）又或是现代分布式计算的前身（mapreduce文献-脚注）。在上述代码中T=tilling size，可以整除N，这样大矩阵乘法就会转换为小矩阵的算法，那么这样的好处在哪呢？如果先计算小矩阵，cache存储的I/O缩小，内存能装下elements，根据分布式中Hadoop-mapreduce的计算思想，将数据分发至节点，呆计算完成最后将结果传入master。进而就会进一步缩小进程运行时间。

这样cache优化从内存方面解决cpu的最大化利用，进而实现矩阵乘法的算法优化。

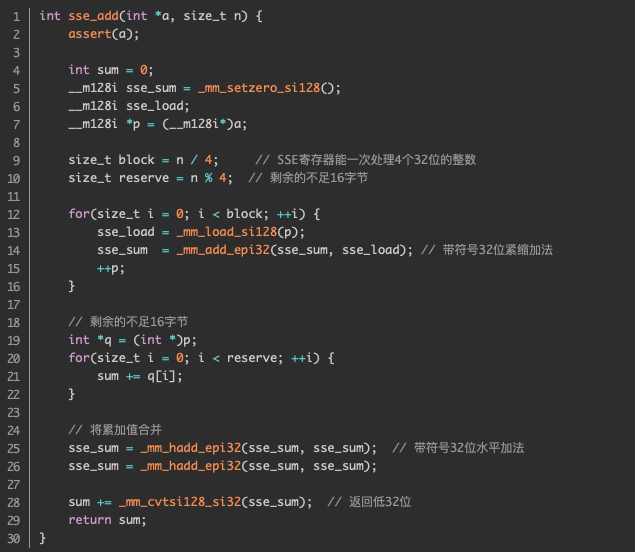
在阅读文献时，我发现如果使用SIMD编程那么就会进一步提高程序性能。下面将介绍这种算法的实现与分析。

2.2 基于SSE/AVX指令集的矩阵乘法的并行化

Intel-SSE技术，全称Sreaming SIMD Extensions,它是一组基于Intel-x86-cpu指令，使用此命令集主要是通过8个寄存器来完成：xmm0-xmm7(arm-x86架构的区别这类文献)。因为这方面知识了解不足，下面我会举个简单的例子来分析SSE指令集对矩阵乘法算法的优化：如下图1-3



这是一个简单累加求和的算法，再通过测试类来实现算法，定义一个timer类，计算normal\_add函数的时间，通过传参的方式申请内存，如图1-4中posix-memalign函数表示在堆栈申请16字节对齐（需要解释字节对齐-脚注）的内存。



备注：

\_mm\_load\_si128函数表示从内存中加载一个**128bits**值到暂存器，也就是**16字节**，**注意：**p必须是一个16字节对齐的一个变量的地址。返回可以存放在代表寄存器的变量中的值。  
\_mm\_loadu\_si128函数和\_mm\_load\_si128一样的，但不要求地址p是16字节对齐。

store系列的\_mm\_store\_si128和\_mm\_storeu\_si128函数，与上面的load系列的函数是对应的。 表示将\_\_m128i 变量a的值存储到p所指定的地址。

在上述代码中，我们使用的是posix\_memalign申请了16字节对齐的内存，因此对照-mmload\_si128函数传参应当就是16字节对齐了。

2.3 基于分片策略的矩阵乘法并行优化

分片策略包含分片键+分片算法，通过分片算法的独立性，因此可以进行分离操作，在hadoop-mapreduce中mapreduce分片分块策略，将大任务split为小任务完成。

同理在矩阵乘法中先针对cpu线程数设置分片数，将矩阵分为多个小矩阵，对每个小矩阵进行SSE指令计算。时间复杂度为O(n) = n^3，同样对矩阵B进行转置。代码如图1-5所示：

double sse\_tile(int n, float a[][maxN], float b[][maxN], float c[][maxN]){ //分片策略 O(n) = n^3

double time;

clock\_t start,end;

start = clock();

\_\_m128 t1, t2, sum;

float t;

for (int i = 0; i < n; ++i)

for (int j = 0; j < i; ++j)

swap(b[i][j], b[j][i]);

for (int r = 0; r < n / T; ++r)

for (int q = 0; q < n / T; ++q) {

for (int i = 0; i < T; ++i)

for (int j = 0; j < T; ++j)

c[r \* T + i][q \* T +j] = 0.0;

for (int p = 0; p < n / T; ++p) {

for (int i = 0; i < T; ++i)

for (int j = 0; j < T; ++j) {

sum = \_mm\_setzero\_ps();

for (int k = 0; k < T; k += 4){ //sum every 4th elements

t1 = \_mm\_loadu\_ps(a[r \* T + i] + p \* T + k);

t2 = \_mm\_loadu\_ps(b[q \* T + j] + p \* T + k);

t1 = \_mm\_mul\_ps(t1, t2);

sum = \_mm\_add\_ps(sum, t1);

}

sum = \_mm\_hadd\_ps(sum, sum);

sum = \_mm\_hadd\_ps(sum, sum);

\_mm\_store\_ss(&t, sum);

c[r \* T + i][q \* T + j] += t;

}

}

}

for (int i = 0; i < n; ++i)

for (int j = 0; j < i; ++j)

swap(b[i][j], b[j][i]);

end = clock();

time = (double)(end-start);

cout<<"分片策略耗时: "<<(time/CLOCKS\_PER\_SEC)<<"s"<<endl;

return time/CLOCKS\_PER\_SEC;

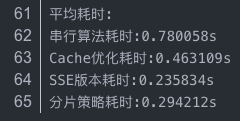
}

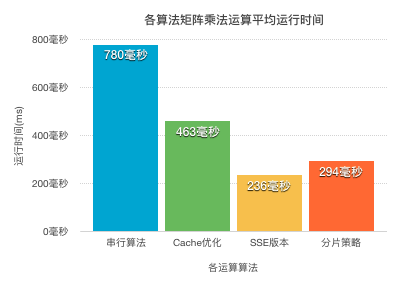
分片策略在实验中表现并不好，因为对mapreduce不是很熟悉，因此对于编程工具还是选用了c++，分片数量以及数据健壮性较差导致了分片策略优化就会低于SSE的效率

3 实验结果

如图1-5为result-logs的3次运行时间，如图1-6为平均运行数据，图1-7为通过E charts实现的数据可视化。







根据试验结果就可以发现在排除未经优化的串行算法，在cache优化对与并行实现有很明显的优化效率，大约增量40%，但是cache结合SIMD编程的SSE优化混合则表现出236毫秒的表现。在分片策略优化算法中，则略低于SSE版本，说明在应用分片算法时对分片划分出现了问题。

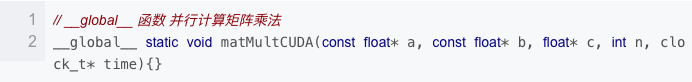
第四章：基于GPU-CUDA编程的矩阵相乘的并行算法

4.1 GPU介绍

在深度学习中数学计算问题中，应用GPU加速来解决并行计算。在并行计算中，常见划分方法为行列划分（带状划分）与棋盘划分（块状划分），在本文中我们利用基于GeForce GT640针对矩阵乘法实现并行化。

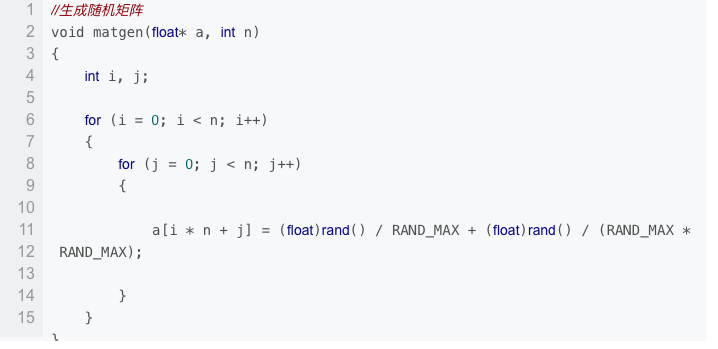
4.2 基于GPU-CUDA的并行矩阵相乘算法及其分析

首先要实现并行算法，我先从核心函数入手，从第二章就可以知道我们需要有A,B,C三个浮点数矩阵，参数为矩阵大小，计算时间的类Timer，如图1-7所示：



。

对于矩阵的随机生成，我采用的是i\*n+j的方式，参数则是矩阵尺寸，如图所示：

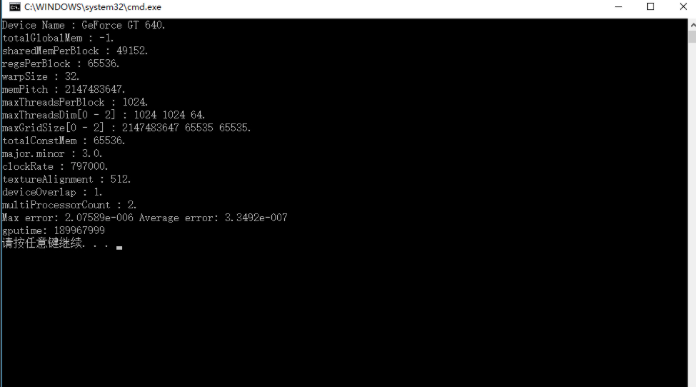


对于CUDA的初始化引入和打印设备信息，这一块我是根据英伟达CUDA开发者社区的官方文档学习的。

做好核函数，CUDA的引入，以及随机矩阵，就可以写一个main函数来实现算法了。最终写出测试代码。

4.3 试验结果

因为GPU这一块的确是不熟悉，因此只是做了一个实现，优化相对CPU常规加速比来说还是很大提升的。



用了189967999个周期，大概是0.23秒，因为cpu版本可以使用double（64bits浮点数）来并行计算，而GPU则只能用float（32bits浮点数）实际上，为了引入共享内存的概念降低GPU带宽使用，把要计算的两个矩阵A B 先分解成BLOCK\_SIZE=16大小的submatrix，每一个block结构运算一个submatrix乘法，这样在一个block中所有的线程是共享参数的，不用每次计算都从global memory中重新加载。

第五章：总结

本文从CPU和GPU两个方向实现矩阵乘法并行算法的研究，以串行，并行，并行优化算法以及基于GPU-CUDA的tiled算法实现矩阵相乘。最终发现，GPU加速效率对于计算性能的提升都是相当大的，当然基于cpu的并行计算同时也给我们研究新型分布式计算的新算法实现进而加速矩阵计算问题。在现代人工智能与云计算时代的高速发展下，GPU加速对科学计算问题而言是十分普遍的，那么cuda C 做矩阵乘法运算呢？在深度神经网络中，全连接层、卷积层、池化层，几乎我们可以想到的所有操作都离不开矩阵运算，卷积层最后其实也是转化为矩阵的乘法操作进行优化，所以本文研究矩阵乘法问题的意义非凡。

本文不足之处：

1. 本文从单机&虚拟机环境下测试代码。实际上可以通过docker微架构部署环境来做实验，亦或是利用大数据集群环境实验分片策略部分。以及增加N值，进而结合分片策略与CUDA-共享内存开发出新算法来解决矩阵相乘问题。
2. 我认为代码需要做进一步优化，因为对x86-SEE/AVX不熟悉，因此在实验过程中，很不理解。因此检索了很多文献
3. 在性能分析这一块做的不是很好，应该偏向运维角度来分析服务器性能。考虑是否可以用redis来解决内存问题。

参考文献