中国石油大学（华东）
CHINA UNIVERSITY OF PETROLEUM

《并行计算与分布式计算》课程论文

题　目：矩阵运算的 CUDA 加速

学生姓名：李 宇 科

学　　号：2009050216

专业班级：数据科学与大数据技术 20-2 班

2022 年 12 月 26 日

目录

关键词：高性能计算，GPU 计算，集群计算

一、 矩阵乘法的提出与综述

矩阵乘法是一种很耗时的运算，并且这种运算在数学领域、计算机领域和机器学习领域都非常普遍。因此，提升矩阵乘法运算的计算速率具有很高的实用价值。

本文中矩阵 A 阶数为 m*n，矩阵 B 阶数为 n*k，A，B，C 均为单精度浮点矩阵。A，B 中元素为-100 到 100 之间的随机浮点数。为方便表示，本文中 A 与 B 为阶数相同的两方阵。

二、 矩阵乘法串行算法原理

比如 m 行 n 列的矩阵 A 和 n 行 k 列的矩阵 B 相乘得到矩阵 C，则 C 为 m 行 k 列的矩阵。C 矩阵中角标为 x，y 的元素就是 A 矩阵第 x 行和 B 矩阵 y 列点积之和。

三、 矩阵乘法串行算法设计与实现

浮点数矩阵存储为一行。最常见的算法，共有三重循环，时间复杂度为 $O(n^3)$。实现方法大致如下：

```
int m = A.height;
int n = A.width;
int k = B.width;
float temp = 0.0f;
for (int i = 0; i < m; i++){
    for (int j = 0; j < k; j++){
        temp = 0.0f;
        for (int h = 0; h < n; h++)
            temp += A.elements[i * n + h] * B.elements[h * k + j];
        C.elements[i * k + j] = temp;
    }
}
```

于此同时，之后设计的各种实现矩阵乘法的程序均是根据上述单线程计算的结果判断计算结果的正误，以及计算对应程序的加速倍率，本文中未标识的时间单位均为毫秒（ms）。

四、 基于 OPENMP 的并行算法设计与实现

并行计算是指将问题分解成若干个部分，各部分均由一个独立的处理机来并行计算。通过并行计算集群完成数据的处理，再将处理的结果返回给用户。

OpenMP 是由 OpenMP Architecture Review Board 牵头提出的，并已被广泛接受，用于共享内存并行系统的多处理器程序设计的一套指导性编译处理方案。OpenMP 提供的这种对于并行描述的高层抽象降低了并行编程的难度和复杂度，这样程序员可以把更多的精力投入到并行算法本身，而非其具体实现细节。对基于数据分集的多线程程序设计，OpenMP 是一个很好的选择。同时，使用 OpenMP 也提供了更强的灵活性，可以较容易的适应不同的并行系统配置。

但是,作为高层抽象,OpenMP 并不适合需要复杂的线程间同步和互斥的场合。OpenMp 的另一个缺点是不能在非共享内存系统(如计算机集群)上使用。在这样的系统上，MPI 使用较多。

4.1 基于 OPENMP 的多核并行算法设计

此并行计算的大致流程是每个线程读取矩阵 B 的一列，并存储在线程内的一块区域中，顺序遍历矩阵 A，这一列和矩阵 A 的每一行进行向量内积和计算，将结果保存到 C 中。这种方法相对于普通方法，因为求矩阵乘法要用到的是 A 矩阵的行和 B 矩阵的列进行计算，而一维数组的顺序读取速度要快于跳越读取。因此使用此方法可以减少线程读取内存所用的时间,进而加快矩阵进行乘法计算的速度。这种方法相当于改变了单线程矩阵乘法的三层循环的顺序，使内存的利用更加高效。

关于矩阵 B 的一列：在 OMP 循环之前定义指针 float *col，并在编译指导语句中设定 col 为各线程私有变量。之后各个线程为 col 指针分配内存,存入数据，并在计算结束之后释放 col 内存空间。

4.2 基于 OpenMP 的多核并行算法实现

实现方法大致如下：

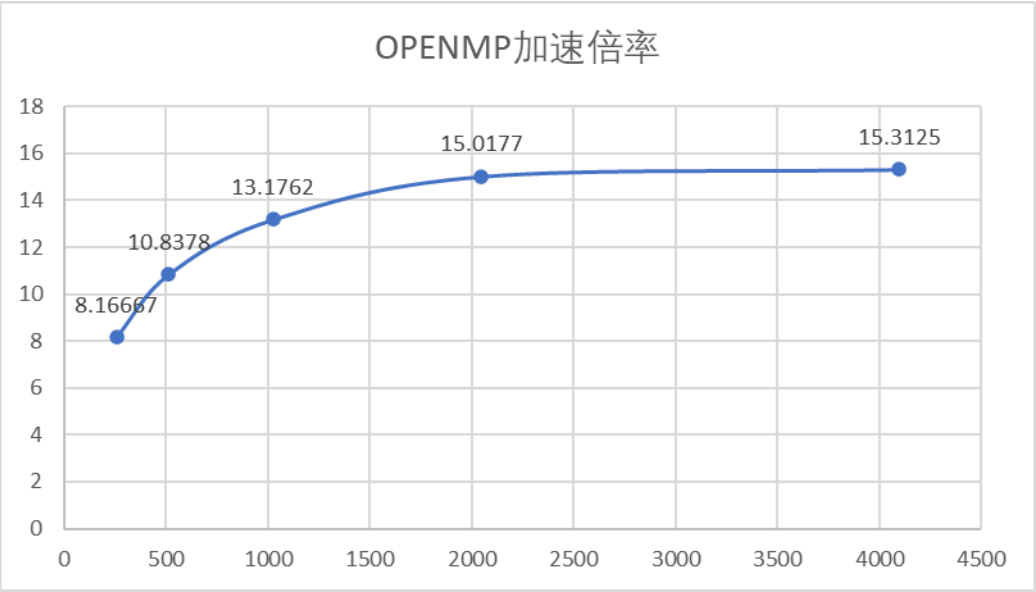
```
int m = A.height;
int n = A.width;
int k = B.width;
float* col;
#pragma omp parallel for schedule(dynamic) private(col)
for (int jj = 0; jj < k; ++jj){
    float sum = 0.0f;
    int id = omp_get_thread_num();
    col = (float*)calloc(n, sizeof(float));
    for (int temp = 0; temp < n; ++temp)
        col[temp] = B.elements[temp * k + jj];
    for (int ii = 0; ii < m; ++ii) {
        sum = 0.0f;
        for (int kk = 0; kk < n; ++kk) {
            sum += col[kk] * A.elements[ii * n + kk];
        C.elements[ii * k + jj] = sum;}
    free(col);
}
```

4.3 实验和结论

	单线程用时	OPENMP 用时	加速倍率	最大误差
256	49	6	8.16667	0
512	401	37	10.8378	0
1024	3779	287	13.1762	0
2048	34856	2321	15.0177	0
4096	288197	18821	15.3125	0

下图为平滑直线拟合数据点（矩阵阶数，OPENMP 相对于单线程加速倍率）之后的图像。可以看出当方阵阶数在 2000 以上时，加速倍率趋于稳定，大约为 15 倍数。对内存的高效利用使得加速倍率超过了所利用的核心数目。乘法是数据密集型计算任务，如何高效利用内存（显存）也是后文*“五、基于 CUDA 的并行算法设计与实现”*效率优化的关键点。

下图为 OPENMP 加速倍率关于矩阵结束的拟合曲线：



另外，研究中单线程和多线程运算的程序代码是用 C++语言编写的，GPU-CUDA 运算的程序代码是用 CUDAC++编写的。所采用的硬件型号为 CPU(AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz)和 GPU(NVIDIA GeForce RTX3060 Laptop GPU)，操作系统和运行环境为 Windows10 和 VisualStudio2022，CUDA 版本为 CUDA11.7。CPU 有八个物理核心，GPU 有 3840 个 CUDA 核心。文中 OMPENMP 和 MPI 均是调用八个 CPU 核心进行计算。

五、 基于 CUDA 的并行算法设计与实现

GPU 是处理计算机图形的专用设备，内部有数目庞大的计算核心，因此非常适合并行计算。CUDA 是一种由 NVIDIA 推出的通用并行计算架构，该架构使 GPU 能够解决复杂的计算问题。无论使用 GPU 并行计算，还是用 CPU 并行计算，都能显著提升计算效率。

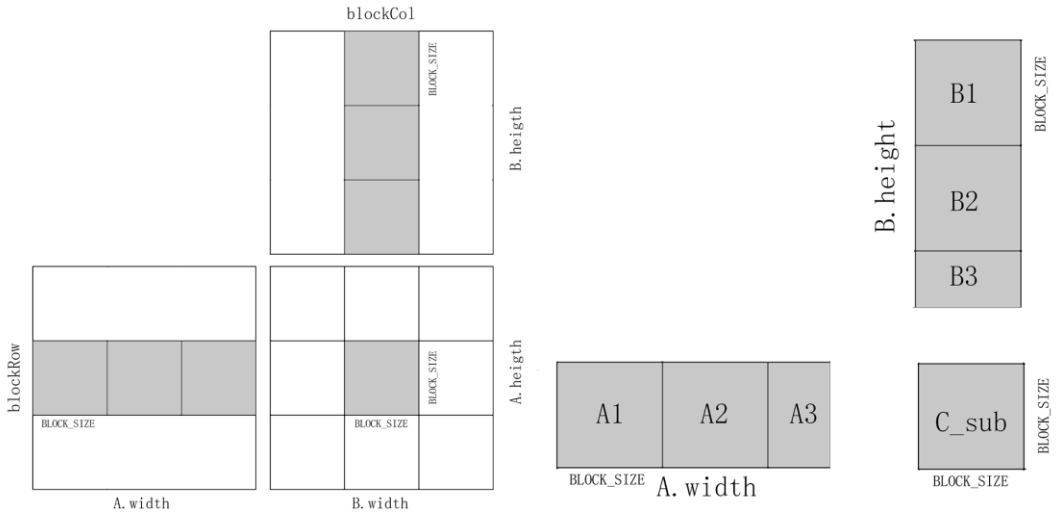
在 CUDA 的架构下，一个程序分为两个部分：Host 端和 Device 端。Host 端是指 CPU 上执行的部分，Device 端是在 GPU 上执行的部分。Device 端的程序又称为 Kernel 函数，一般把算法最耗时的部分放在这里，利用 GPU 的强大计算能力来快速完成。通常 Host 端程序会将数据准备好后，复制到显卡的内存中，再由显示芯片执行 Device 端程序，完成后再由 Host 端程序将结果从显卡的内存中取回。在 CUDA 架构下，显示芯片执行时的最小单位是线程（thread）。几个线程可以组成一个线程块（block），几个线程块可以组成一个网格（grid）。

5.1 基于 CUDA 的多核并行算法设计

CUDA 矩阵乘法算法流程图见**附件 1**。

设计 1：函数中每个线程计算 C 矩阵的一个单元，原理同上文矩阵乘法的单线程计算。

设计 2：优化使用寄存器和共享内存，以减少访存开销。设定变量 BLOCK_SIZE：每个块的大小。为了便于理解，以 N*N 阶方阵相乘为例，已知计算结果也为 N*N 阶矩阵。C 中的一块定义为 C_sub，CUDA 中每个线程块都负责一块 C_sub，而每个线程块中的线程都负责 C_sub 中的一个元素。如图所示：C_sub 等于 A 的子矩阵（宽度为 A.width，高度为 BLOCK_SIZE）和 B 的子矩阵（宽度为 BLOCK_SIZE，高度为 B.height）



之后将根据需要将两个矩形矩阵分成任意数量的维度为 BLOCK_SIZE 的正方形矩阵，并将两方阵的乘积累加到 C_sub。每一次两方阵相乘都首先将两个对应的方阵从全局内存加载到共享矩阵内存，一个线程加载方阵的一个元素，然后让每个线程计算 C_sub 的一个元素。每个线程累积的结果这些产品中的每一个都放入寄存器中，完成之后，将结果写入全局内存。计算过程如上图所示：C_sub = A1*B1+A2*B2+A3*B3

5.2 基于 CUDA 的多核并行算法实现

启动核：

```
#define BLOCK_SIZE 16
int calculateDimGrid(int a, int b) {
    return int((float(a) + (float(B) - 1)) / float(B));
}
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(calculateDimGrid(B.width, dimBlock.x),
calculateDimGrid(A.height, dimBlock.y));
//核心 v1 的启动
MatMulKernelV1 <<<dimGrid, dimBlock >>> (A, B, C);
////核心 v2 的启动
//MatMulKernelV2 <<<dimGrid, dimBlock >>> (A, B, C);
```

核心 v1：

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;
if (col < k && row < m){
    for (int i = 0; i < n; i++)
        sum += A.elements[row * n + i] * B.elements[i * k + col];
    C.elements[row * k + col] = sum;
}
```

核心 v2：

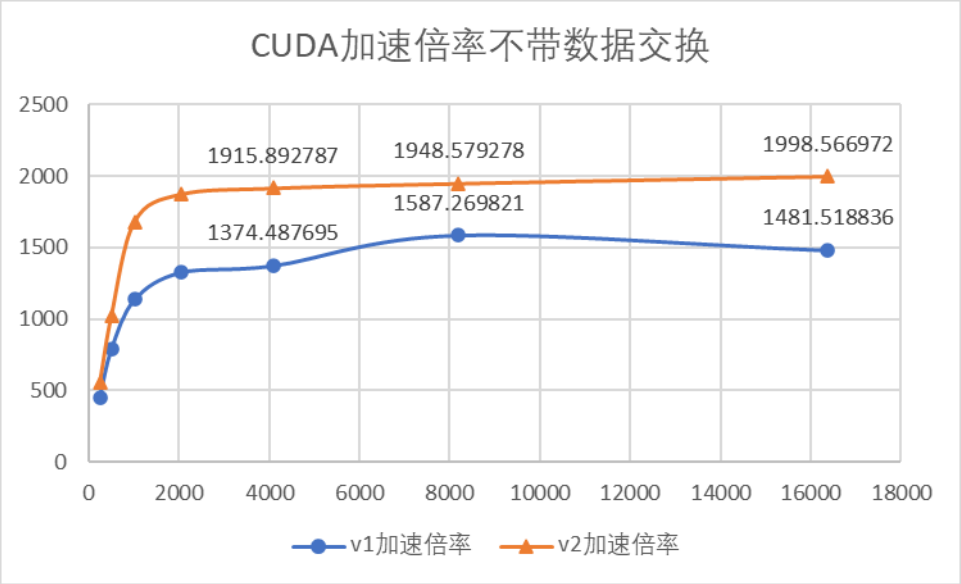
```
float Ctemp=0.0f;
__shared__float AS[BLOCK_SIZE][BLOCK_SIZE];
__shared__float BS[BLOCK_SIZE][BLOCK_SIZE];
for(…){
    // 分块矩阵加载到 shared memory
    As[ty][tx]=A[indexA];
    Bs[ty][tx]=B[indexB];
    indexA+=BLOCK_SIZE;
    indexB+=widthB*BLOCK_SIZE;
    _syncthreads（）;
    // 计算部分结果
    for（i=0；i<BLOCK_SIZE；i++）
        Ctemp+=As[ty][i]*Bs[i][tx];
    // 加载新线程块前再次同步
    _syncthreads（）;
}
C[indexC]=Ctemp;
```

5.3 实验和结论

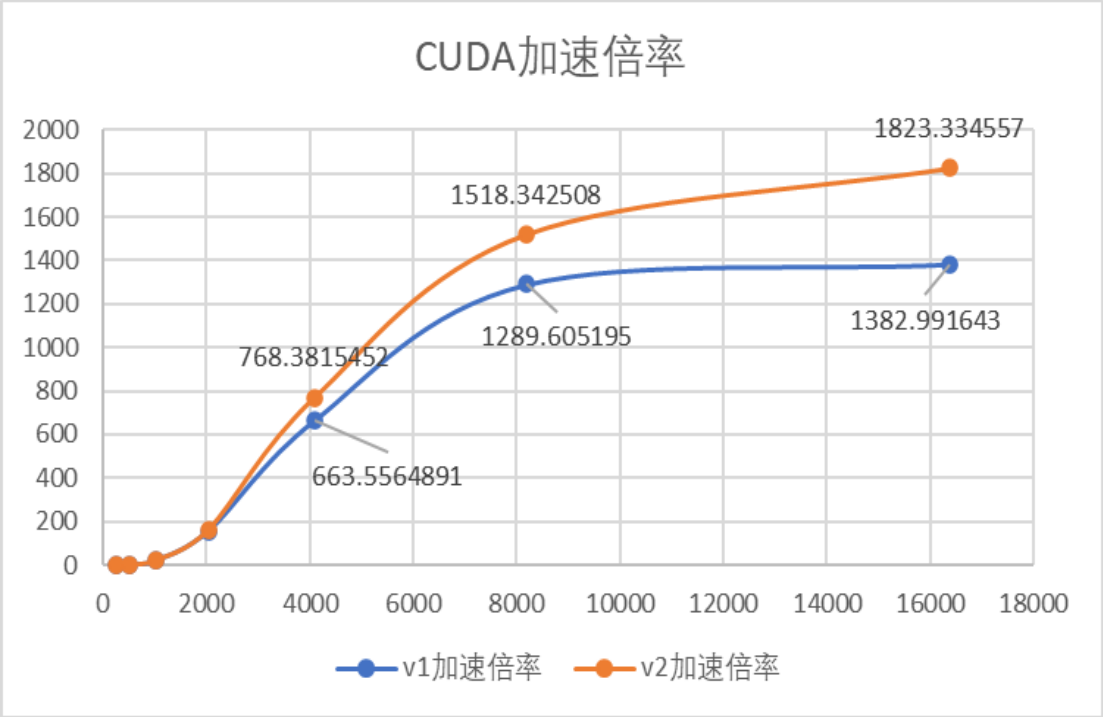
	单线程	CUDAv1	v1 倍率	CUDAv2	v2 倍率	最大误差	数据交换用时
256	50	0.11	454.545	0.09	555.556	0.03125	171
512	410	0.52	788.462	0.4	1025	0.046875	179
1024	3952	3.48	1135.63	2.35	1681.7	0.109375	182
2048	35608	26.84	1326.68	19	1874.11	0.1875	200
4096	293783	213.74	1374.49	153.34	1915.89	0.28125	229
8192	2482490	1564	1587.26	1274	1948.57	0.51225	361
16384	-	13405	-	9937	-	-	955

对于上表中的数据：

1. CUDA 运算的精度问题：CUDA 计算单精度浮点数会产生较大的误差，如上表所示，改用双精度浮点数会好很多，如 1024 阶方阵相乘最大误差为：1.74623e-10，2048 阶方阵相乘最大误差为：2.91038e-10。但一般民用显卡双精度性能受限，专业计算卡不在本文讨论范围之内。
2. CUDAv1 是上文提到的第一种 CUDA 实现矩阵相乘的方法，即直接计算方法，CUDAv2 是第二种实现方法，即共享内存方法。



3. 上述 CUDAv1 和 CUDAv2 最大误差都一样，故合并为一项。
4. 在计时方面：CUDAv1 和 CUDAv2 计算时间统计的是计算核心的计算时长，没有把显存中分配空间、矩阵数据从内存发送到显存、把计算结果从显存发送到内存的耗时，“数据交换用时”显示的就是这三项的用时之和。
5. 16384 阶方阵 CPU 单线程计算为理论时间：5.5166 小时，加速倍率图像依据此数据绘制。



由图表可知：

1. 上述几种规模的矩阵计算，对于数据交换用时影响都很小，随着矩阵阶数的增长，数据交换用时对于总计算时间的影响可以基本忽略。
2. 根据 CUDAv1 和 CUDAv2 方法的计算用时和加速度比，可知共享内存方法相对于普通的直接计算方法有着明显的性能提升。
3. 因为启动CUDA核心需要时间，所以小规模矩阵的运算在CUDA上优势不大，反倒是数据交换和启动核心花掉了较多的时间。

六、 基于 MPI 的并行算法设计与实现

MPI 是一个跨语言的通讯协议，用于编写并行计算机。支持点对点和广播。MPI 的目标是高性能，大规模性，和可移植性。MPI 在今天仍为高性能计算的主要模型。与 OpenMP 并行程序不同，MPI 是一种基于信息传递的并行编程技术。消息传递接口是一种编程接口标准，而不是一种具体的编程语言。简而言之，MPI 标准定义了一组具有可移植性的编程接口。

6.1 基于 MPI 的多核并行算法设计

MPI 矩阵乘法算法流程图见**附件 2**。在矩阵乘法的 MPI 多核并行算法的设计中，主进程将矩阵 B 整体发送给各个从进程，将矩阵 A 整除分割发送给各从进程，从进程计算分割后的矩阵 A 和 B 相乘的结果并发送给主线程，主线程收集汇总这些数据，并将矩阵 A 分割余下的行（行数不会超过所分进程数）进行计算，最终汇总成结果 C。对于矩阵乘法计算项目，MPI 有两套信息传递方式可以满足要求：点对点通信和集群通信。

6.2 基于 MPI 的多核并行算法实现

实现一：点对点通信（通过 MPI_Send 和 MPI_Recv 实现。）

主进程：

```
int line = m / numprocs;
// 1 将矩阵 B 发送给其他进程
for (i = 1; i < numprocs; ++i)
    MPI_Send(B.elements, n*k, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
// 2 将矩阵 A 的各行发送给各个从进程
for (i = 1; i < numprocs; ++i)
    MPI_Send(A.elements + (i - 1) * line * n, line * n, MPI_FLOAT,
i, 1, MPI_COMM_WORLD);
// 3 接受从进程的计算结果并存入矩阵 C
for (k = 1; k < numprocs; ++k) {
    MPI_Recv(ans.elements, line * k, MPI_FLOAT, k, 3,
MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
    copyAnsToMatrixC(ans,C);}
```

从进程：

```
//接收数据
MPI_Recv(B.elements, B.width * B.height, MPI_INT, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(buffer.elements, line * nnn, MPI_INT, 0, 1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
//计算结果
matrixMUL(buffer.elements, B.elements, ans.elements,
buffer.height, buffer.width, B.width);
// 将结果发送给主进程
MPI_Send(ans.elements, line * kkk, MPI_INT, 0, 3,
MPI_COMM_WORLD);
```

实现二：集合通信（通过 MPI_Bcast、MPI_Scatter 和 MPI_Gather 实现。）

主进程：

```
//集合通信主进程
// 1 将矩阵 B 发送给其他进程
MPI_Bcast(B.elements, B.height * B.width, MPI_FLOAT, 0,
MPI_COMM_WORLD);
// 2 将矩阵 A 的各行发送给各个从进程
MPI_Scatter(A.elements, line * A.width, MPI_FLOAT,
buffer.elements, line * nnn, MPI_FLOAT, 0, MPI_COMM_WORLD);
//计算本地结果
matrixMUL(buffer.elements, B.elements, ans.elements,
buffer.height, buffer.width, B.width);
//结果聚集
MPI_Gather(ans.elements, line * kkk, MPI_FLOAT, C.elements, line
* kkk, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

从进程：

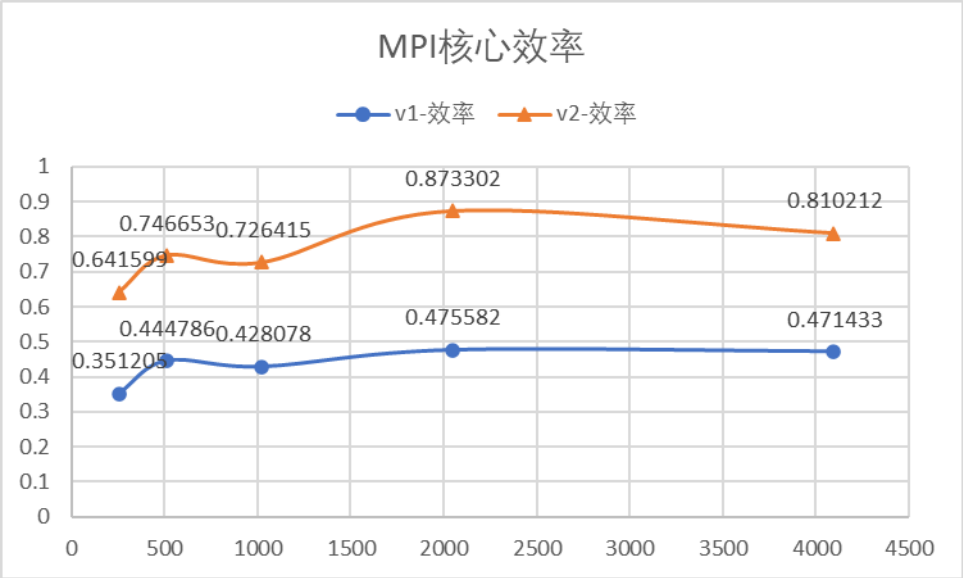
```
// 集合通信从进程
// 接收数据
MPI_Bcast(B.elements, B.width * B.height, MPI_FLOAT, 0,
MPI_COMM_WORLD);
MPI_Scatter(A.elements, line * nnn, MPI_FLOAT, buffer.elements,
line * nnn, MPI_FLOAT, 0, MPI_COMM_WORLD);
// 计算本地
matrixMUL(buffer.elements, B.elements, ans.elements,
buffer.height, buffer.width, B.width);
// 结果聚集
MPI_Gather(ans.elements, line * kkk, MPI_FLOAT, C.elements, line
* kkk, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

6.3 实验和结论

点对点通信实现称为 MPIv1，集合通信实现称为 MPIv2。数据表如下：

	单线程用时	MPIv1	v1 加速倍率	v1-效率	MPIv2	v2 加速倍率	v2-效率
256	50.25	17.88	2.80964	0.351205	9.78	5.13279	0.641599
512	405.41	113.93	3.55829	0.444786	67.59	5.97323	0.746653
1024	3714.96	1084.78	3.42462	0.428078	642.81	5.81132	0.726415
2048	35738.3	9393.3	3.80466	0.475582	5160.61	6.98642	0.873302
4096	287905	76337	3.77147	0.471433	44241	6.4817	0.810212

MPI 方法也是用 CPU 计算，与单线程计算结果一致，最大误差均为零。



上图显示的是 MPIv1 和 v2 方法随方阵阶数增长的计算效率。

由图可知，MPIv2 方法即集合通信方法在矩阵乘法中有较高的计算效率。

七、 分布式平台的并行算法设计和实现

7.1 基于 MPI+CUDA 的多核并行算法设计

MPI 并行计算的相对于 OPENMP 和 CUDA 来说，它的优点是可以放在集群上面运行，这意味着它可以不单单运行在一台计算机的多个核心上，而是可以将多台计算机的计算资源都运用起来。

因此，联合运用 MPI 和 CUDA，便可以同时利用多台计算机的 CPU 和 GPU 资源，通过堆叠计算机的数量，得到更优的加速性能。

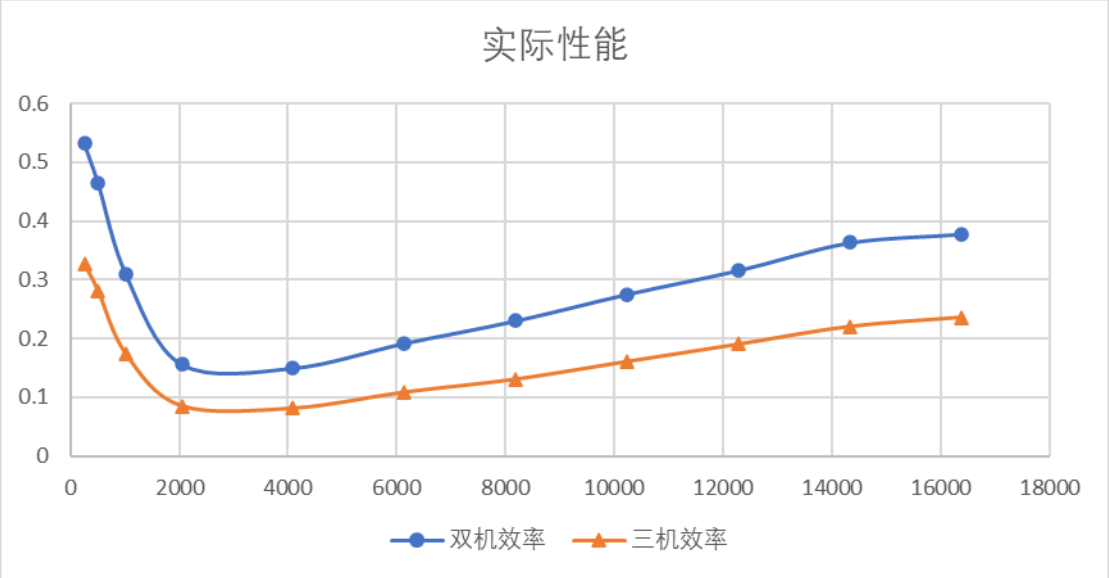
通过*“五、基于 CUDA 的并行算法设计与实现”*和*“六、基于 MPI 的并行算法设计与实现”*可知，在面对大规模矩阵乘法问题时，CUDAv2 方法（即共享内

存方法）和 MPIv2 方法（即集合通信方法）有更高的运行效率，将这两种方法合并编译，简单来说，就是将 MPIv2 方法中涉及到矩阵乘法的地方替换为 CUDAv2 方法。

7.2 实验和结论

	单机	双机	双机效率	三机	三机效率
256	123.17	115.648	0.532521	125.231	0.327847
512	124.03	133.619	0.464118	147.342	0.280594
1024	130.67	211.503	0.308908	251.331	0.173304
2048	175.14	561.456	0.155969	684.811	0.08525
4096	594	1988.93	0.149327	2431.44	0.081433
6144	1791	4670.07	0.191753	5491.96	0.108704
8192	4001	8695.27	0.230068	10198	0.130777
10240	7831	14222	0.275313	16225.5	0.160878
12288	13799	21832.8	0.316015	24157.7	0.190402
14336	22756	31328	0.363189	34389	0.220574
16384	32174	42580.6	0.377801	45419.4	0.236125

上述是 MPI+CUDA 程序在 Windows 集群上的运行结果（详细数据可见**附件 3**），三个节点情况相同：CPU 是 E5-2678v3，GPU 是 GTX1050Ti，内存 48G，系统为 Windows10，MPI 版本和之前相同，CUDA 版本为 12.0，GPU 有 768 个 CUDA 核心。



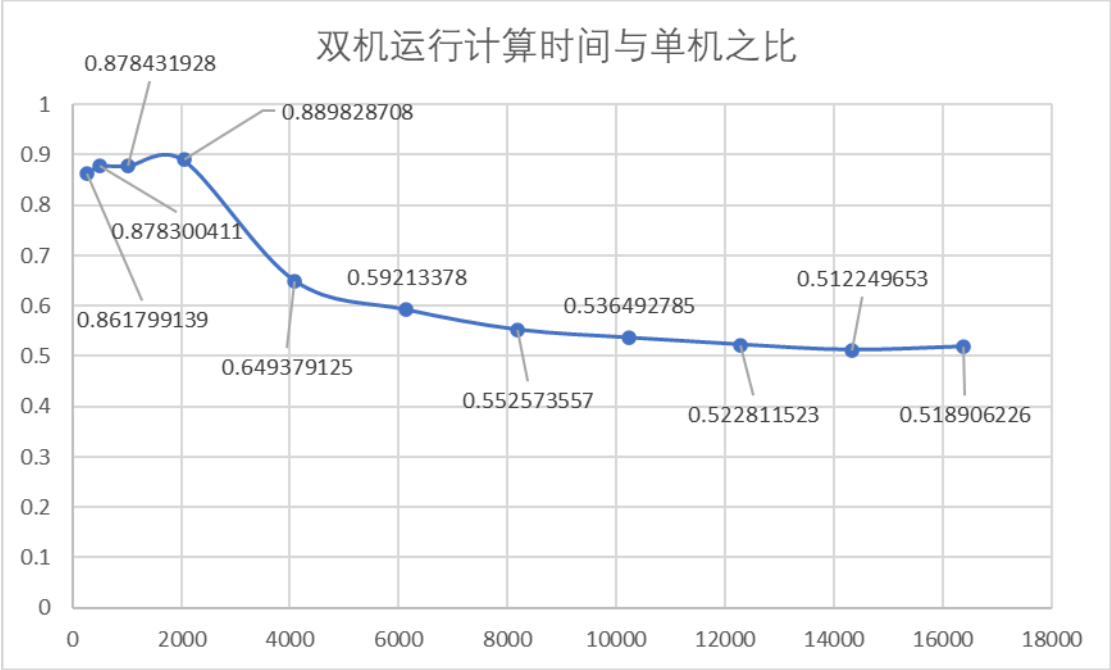
此外，单机情况是直接用 CUDAv2 方法计算的计算时间，并非是 MPI 设置进

程数为 1 的结果，目的是比较 MPI 方法相对于普通方法的性能损耗。通过上图可以看出：多机效率在方阵阶数为 3000 左右时降至最低，联系到 CUDAv2 方法对比于普通方法在方阵阶数为 3000 左右趋于高位和平缓，MPI+CUDA 多机运行效率的瓶颈可能在 MPI 进程之间的数据传输。

因此，采集了双机运行情况下 MPI+CUDA 程序花费在各个阶段的时长。

	单机用时	广播	分发	计算	聚集
256	123.17	5.8853	1.6233	106.1478	1.9054
512	124.03	13.5772	5.5348	108.9356	5.4948
1024	130.67	50.2122	24.5523	114.7847	21.8382
2048	175.14	200.463	100.508	155.8446	104.4971
4096	594	783.862	413.368	385.7312	405.8377
6144	1791	1763.1	910.909	1060.512	935.4279
8192	4001	3153.44	1655.37	2210.847	1675.493
10240	7831	5055.77	2583.19	4201.275	2581.672
12288	13799	7184.66	3696	7214.276	3737.73
14336	22756	9629.91	4991.2	11656.75	5050.059
16384	32174	12724.3	6598.34	16695.29	6562.578

通过上表绘制双机“计算”过程时间和单机运行时间的比与矩阵阶数的图像如下：



在双机情况下：随着矩阵阶数的增长，平分之后的计算任务所花费的时间是

逐渐趋近原本计算时间的一半的（由上文可知 CUDA 方法在内存和显存数据交换会花费较多时间，所以在矩阵阶数较低时与 CUDA 实际运行时间差距不大）。

由此可以确定妨碍多机运行效率提升的问题在于节点之间的数据交换。

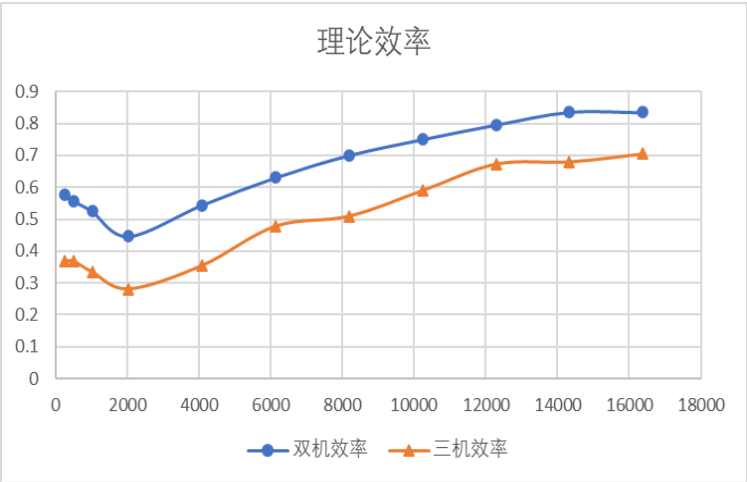
已知节点之间时通过网络通信的，通过在运行时监测记录节点的网络上下行速度，双机运行时主机上行速度最大为 700Mbps，三机运行时主机上行速度最大为 900Mbps。计算得知 8192*8192 浮点矩阵数据量为 2048Mb，理论用时 2925.71 ms，实际用时 3153.44ms。

因此，现有集群环境不能满足此计算任务的要求。现有集群节点配备的都是千兆网卡，而千兆网卡早已不是新鲜的技术。就目前而言，千兆广域网络正在普及，虽然万兆广域网络还不能实现，但在局域网内，万兆网络或更进一步的网络早已有了它的应用场景：超算中心、数据中心、企业校园、网络存储等。

为节省成本，模拟 MPI+CUDA 项目在万兆内网环境下的运行情况。将双机和三机运行数据中广播、分发、聚集所用时间除以十得到新的数据，如下表所示：

	单机用时	时间-2	加速比-2	效率-2	时间-3	加速比-3	效率-3
256	123.17	107.0892	1.150163	0.575081	111.6984	1.102701	0.367567
512	124.03	111.3963	1.113412	0.556706	112.2003	1.105433	0.368478
1024	130.67	124.445	1.050022	0.525011	129.8096	1.006628	0.335543
2048	175.14	196.3914	0.891791	0.445895	207.4395	0.844294	0.281431
4096	594	546.038	1.087836	0.543918	557.6143	1.065253	0.355084
6144	1791	1421.455	1.259976	0.629988	1247.467	1.435709	0.47857
8192	4001	2859.277	1.399305	0.699652	2615.222	1.529889	0.509963
10240	7831	5223.338	1.499233	0.749616	4421.898	1.770959	0.59032
12288	13799	8676.115	1.590458	0.795229	6829.969	2.020361	0.673454
14336	22756	13623.87	1.670304	0.835152	11152.5	2.04044	0.680147
16384	32174	19283.81	1.668446	0.834223	15205.77	2.115907	0.705302

由上表绘制出如下万兆网络环境理论效率图。由此可知，该 MPI 和 CUDA 混合编译大规模矩阵乘法项目，在万兆或以上网络情况下有良好的加速性能。



八、 关于矩阵乘法运算的讨论

当矩阵阶数较低时（如低于 300）适合用 OMP 或者 MPI 方法并行，大于此阶数的矩阵乘法适合用 CUDAv2 方法，若阶数大于 6000 且有万兆或更高内网环境可以用 MPI+CUDA 方法并行计算。

参考文献

[1] 陈华. 高性能并行计算[M].青岛:中国石油大学出版社,2018.

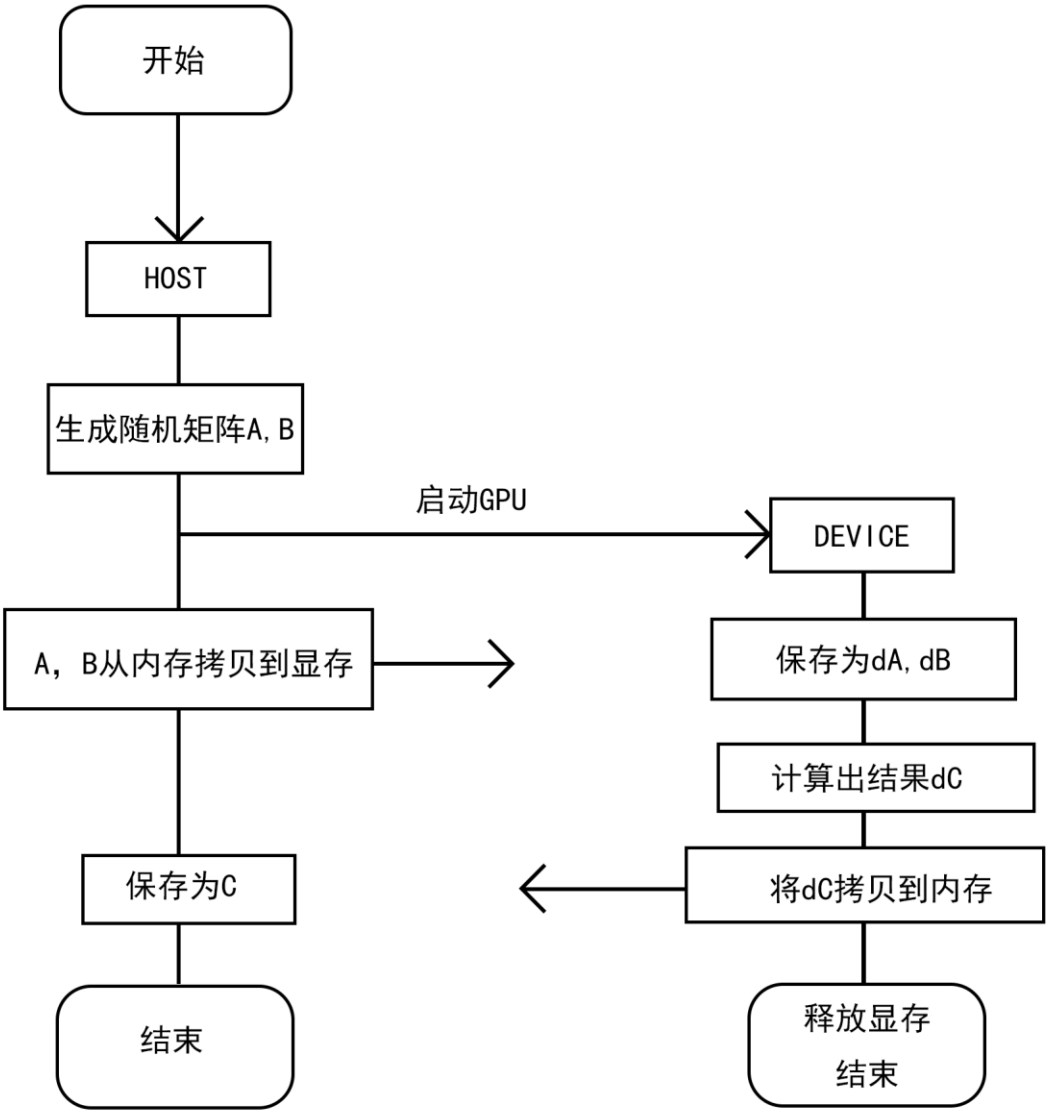
[2] 白洪涛. 基于 GPU 的高性能并行算法研究[D].吉林大学,2010.

[3] CUDA Toolkit Documentation v12.0 - landing 12.0 documentation. Nvidia. https://docs.nvidia.com/cuda/index.html. 2022

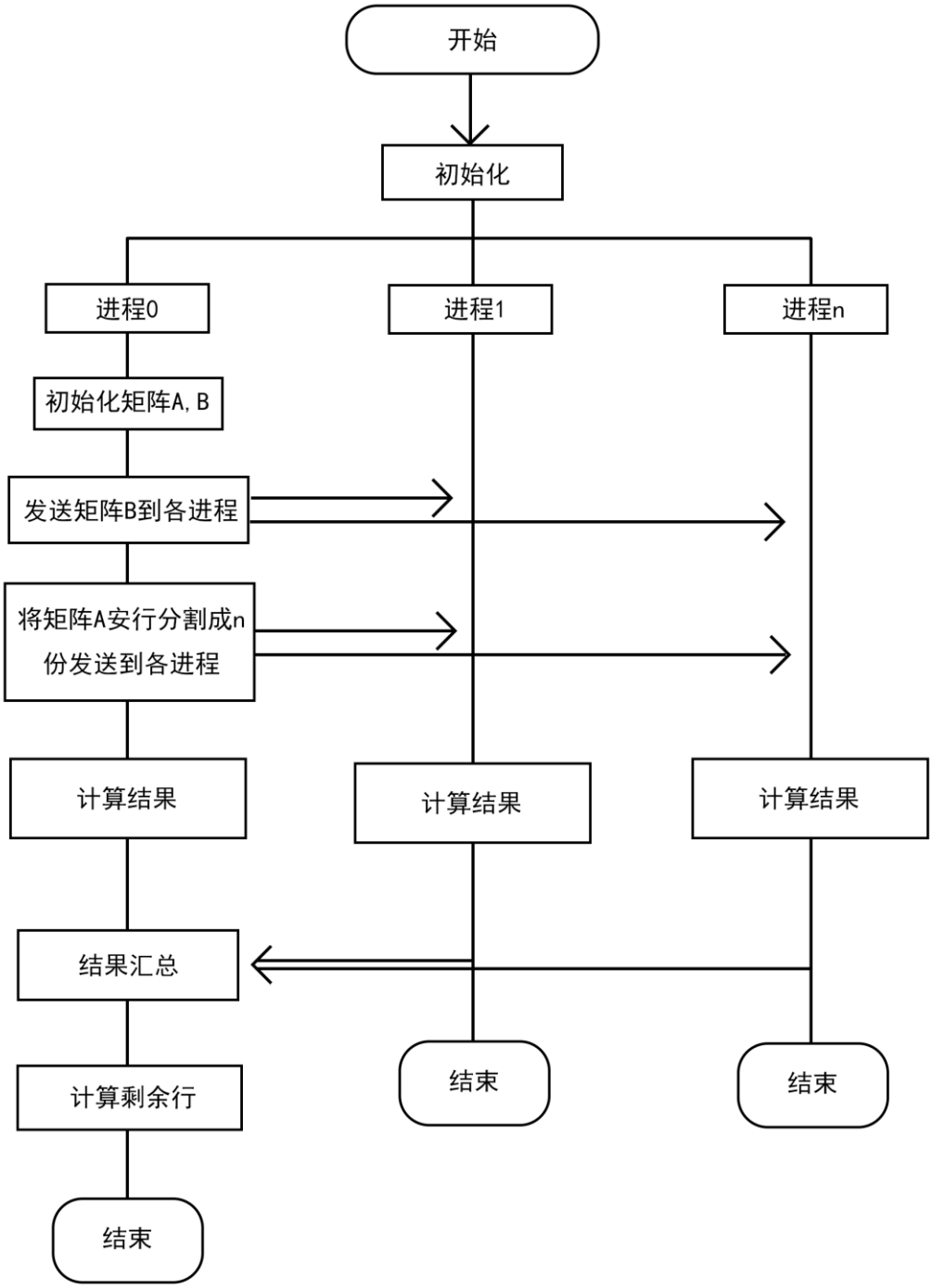
[4] 周文荣.并行计算与 MPI 研究[J].无线互联科技,2017(12):23-25.

附件

附件 1：CUDA 矩阵乘法流程图



附件 2：MPI 矩阵乘法流程图



附件 3：实验数据

实验数据.xlsx