**深 圳 大 学 实 验 报 告**

|  |  |
| --- | --- |
| **课程名称** | **：并行计算** |
| **实验项目名称** | **：矩阵乘法的CUDA并行程序** |
| **学院** | **：计算机与软件学院** |
| **专业** | **：计算机科学与技术** |
| **指导教师** | **：陆克中** |
| **报告人** | **：何泽锋** |
| **学号** | **：2022150221** |
| **实验时间** | **：2025年06月11日** |
| **实验报告提交时间** | **：2025年06月11日** |

**教务部制**

## 一、实验目的

1. 学会编写简单的CUDA程序；

2. 掌握CUDA程序的归约求和方法；

3. CUDA对并行程序进行简单的性能分析；

## 二、实验环境

1. 硬件环境：NVIDIA GeForce RTX 4090D（114个SM，每个SM有128个SP）；

2. 软件环境：Ubuntu Linux、nvcc（nvcc -o cuda\_hello cuda\_hello.cu）；

3. 远程登录：本地PowerShell中先执行ssh [bxjs@hpc.szu.edu.cn](mailto:bxjs@hpc.szu.edu.cn)，再执行ssh CUDA；

4. 传输文件：本地PowerShell中执行scp c:\a.cpp [bxjs@hpc.szu.edu.cn:/home/bxjs/](mailto:bxjs@hpc.szu.edu.cn:/home/bxjs/)或<ftp://hpc.szu.edu.cn>。

## 三、实验内容

1. 用CUDA编写两个*n*阶方阵*x*和*y*的并行相乘程序，结果存放在方阵*z*中，然后计算方阵*z*中的各个元素之和，最后输出和值及执行时间（矩阵乘法及求和的时间，不含复制矩阵数据的时间），并与CPU的计算结果相比较。优先使用树形求和。

2. 测试并行程序在不同方阵阶数下的执行时间和加速比（与串行程序的执行时间相比）。其中，*n*分别取32、64、128、256、512、1024，网格和线程块取合适的配置。为了减少误差，每项实验进行5次，取平均值作为实验结果。

## 四、代码描述

*在下面写出完整的程序代码（文本格式，不可截图），对于其中的关键代码，以注释方式给出必要的描述。*

|  |
| --- |
| #include <stdio.h>  #include <cuda.h>  #include <time.h>  #include <math.h>  #include <stdlib.h>  #define BLOCK\_SIZE 32 // 块边长  #define REDUCTION\_BLOCK 1024 // 归约线程数  // GPU矩阵乘法核函数  \_\_global\_\_ void gpuMat(double\* matA, double\* matB, double\* matC, int dimension) {  // 计算当前线程处理的矩阵行列索引  int rowIdx = blockIdx.y \* blockDim.y + threadIdx.y;  int colIdx = blockIdx.x \* blockDim.x + threadIdx.x;  // 检查索引是否在有效范围内  if (rowIdx < dimension && colIdx < dimension) {  double accumulator = 0.0f;  // 计算矩阵乘法的点积  for (int idx = 0; idx < dimension; idx++) {  accumulator += matA[rowIdx \* dimension + idx] \* matB[idx \* dimension + colIdx];  }  // 将结果写入输出矩阵  matC[rowIdx \* dimension + colIdx] = accumulator;  }  }  // 并行规约求和核函数  \_\_global\_\_ void Sum(double\* dataIn, double\* dataOut, int totalElements) {  extern \_\_shared\_\_ double sharedMem[]; // 动态共享内存  unsigned int localId = threadIdx.x; // 线程的本地ID  unsigned int globalId = blockIdx.x \* blockDim.x \* 2 + threadIdx.x; // 线程的全局ID  // 每个线程计算两个元素的和  double localSum = 0.0f;  if (globalId < totalElements) localSum += dataIn[globalId];  if (globalId + blockDim.x < totalElements) localSum += dataIn[globalId + blockDim.x];  // 将局部和存入共享内存  sharedMem[localId] = localSum;  \_\_syncthreads(); // 同步线程  // 在共享内存中进行规约求和  for (unsigned int stride = blockDim.x / 2; stride > 0; stride /= 2) {  if (localId < stride) {  sharedMem[localId] += sharedMem[localId + stride];  }  \_\_syncthreads(); // 每次规约后同步  }  // 第一个线程将最终结果写入全局内存  if (localId == 0) dataOut[blockIdx.x] = sharedMem[0];  }  // GPU计算封装函数  inline void GPU(double\* d\_x, double\* d\_y, double\* d\_z, int n, double\* sum\_result) {  // 设置块和网格维度  dim3 dimBlock(BLOCK\_SIZE, BLOCK\_SIZE);  dim3 dimGrid((n + BLOCK\_SIZE - 1) / BLOCK\_SIZE, (n + BLOCK\_SIZE - 1) / BLOCK\_SIZE);  // 调用矩阵乘法核函数  gpuMat << <dimGrid, dimBlock >> > (d\_x, d\_y, d\_z, n);  cudaDeviceSynchronize(); // 同步设备  int totalCount = n \* n;  int blockCount = ceil(totalCount + REDUCTION\_BLOCK \* 2 - 1) / (REDUCTION\_BLOCK \* 2);  double\* devTempBuffer;  cudaMalloc(&devTempBuffer, blockCount \* sizeof(double)); // 分配临时缓冲区  // 第一次规约求块内和  Sum << <blockCount, REDUCTION\_BLOCK, REDUCTION\_BLOCK \* sizeof(double) >> > (  d\_z, devTempBuffer, totalCount);  // 继续归约块间和  while (blockCount > 1) {  int nextBlockCount = (blockCount + REDUCTION\_BLOCK - 1) / (REDUCTION\_BLOCK);  Sum << <nextBlockCount, REDUCTION\_BLOCK, REDUCTION\_BLOCK \* sizeof(double) >> > (  devTempBuffer, devTempBuffer, blockCount);  blockCount = nextBlockCount;  }  // 将最终结果拷贝回主机  cudaMemcpy(sum\_result, devTempBuffer, sizeof(double), cudaMemcpyDeviceToHost);  cudaDeviceSynchronize();  cudaFree(devTempBuffer);  }  // CPU矩阵乘法实现  inline void CPU(double\* x, double\* y, double\* z, int n) {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  double sum = 0.0f;  for (int k = 0; k < n; k++) {  sum += x[i \* n + k] \* y[k \* n + j];  }  z[i \* n + j] = sum;  }  }  }  // CPU上的求和函数  inline double cpuSum(double\* z, int n) {  double sum = 0.0f;  for (int i = 0; i < n \* n; i++) {  sum += z[i];  }  return sum;  }  // GPU预热函数  inline void warmupGPU(int n) {  double\* warm\_x, \* warm\_y, \* warm\_z, \* warm\_sum\_p;  // 分配主机内存  warm\_x = (double\*)malloc(n \* n \* sizeof(double));  warm\_y = (double\*)malloc(n \* n \* sizeof(double));  warm\_z = (double\*)malloc(n \* n \* sizeof(double));  warm\_sum\_p = (double\*)malloc(sizeof(double));  // 分配GPU内存  double\* d\_warm\_x, \* d\_warm\_y, \* d\_warm\_z;  cudaMalloc(&d\_warm\_x, n \* n \* sizeof(double));  cudaMalloc(&d\_warm\_y, n \* n \* sizeof(double));  cudaMalloc(&d\_warm\_z, n \* n \* sizeof(double));  // 初始化数据为1.0  for (int i = 0; i < n \* n; i++) {  warm\_x[i] = 1.0;  warm\_y[i] = 1.0;  }  // 复制数据到GPU  cudaMemcpy(d\_warm\_x, warm\_x, n \* n \* sizeof(double), cudaMemcpyHostToDevice);  cudaMemcpy(d\_warm\_y, warm\_y, n \* n \* sizeof(double), cudaMemcpyHostToDevice);  // 执行一次GPU计算作为预热  GPU(d\_warm\_x, d\_warm\_y, d\_warm\_z, n, warm\_sum\_p);  cudaDeviceSynchronize();  cudaFree(d\_warm\_x);  cudaFree(d\_warm\_y);  cudaFree(d\_warm\_z);  free(warm\_x);  free(warm\_y);  free(warm\_z);  free(warm\_sum\_p);  printf("GPU预热完成\n");  }  int main(void) {  int n = 1024; // 矩阵维度  double\* x, \* y, \* z, \* cz; // 主机端矩阵指针  double\* sum\_p; // 用于存储GPU求和结果  double sum; // 用于存储CPU求和结果  // 分配主机内存  x = (double\*)malloc(n \* n \* sizeof(double));  y = (double\*)malloc(n \* n \* sizeof(double));  z = (double\*)malloc(n \* n \* sizeof(double));  cz = (double\*)malloc(n \* n \* sizeof(double));  sum\_p = (double\*)malloc(sizeof(double));  // 初始化矩阵数据为随机值  for (int i = 0; i < n \* n; i++) {  x[i] = rand() \* 1.0 / RAND\_MAX;  y[i] = rand() \* 1.0 / RAND\_MAX;  }  // 分配GPU内存  double\* d\_x, \* d\_y, \* d\_z;  cudaMalloc(&d\_x, n \* n \* sizeof(double));  cudaMalloc(&d\_y, n \* n \* sizeof(double));  cudaMalloc(&d\_z, n \* n \* sizeof(double));  // 将数据复制到GPU  cudaMemcpy(d\_x, x, n \* n \* sizeof(double), cudaMemcpyHostToDevice);  cudaMemcpy(d\_y, y, n \* n \* sizeof(double), cudaMemcpyHostToDevice);  clock\_t start, finish;  printf("开始GPU预热...\n");  warmupGPU(256); // 预热  // GPU计算  start = clock();  GPU(d\_x, d\_y, d\_z, n, sum\_p);  finish = clock();  cudaMemcpy(z, d\_z, n \* n \* sizeof(double), cudaMemcpyDeviceToHost);  printf("GPU执行时间: %f秒\n", (finish - start) \* 1.0 / CLOCKS\_PER\_SEC);  printf("矩阵元素之和: %f\n", \*sum\_p);  // CPU计算  start = clock();  CPU(x, y, cz, n);  sum = cpuSum(cz, n);  finish = clock();  double cpuElapsedTime = (finish - start) \* 1.0 / CLOCKS\_PER\_SEC;  printf("CPU执行时间: %f秒\n", cpuElapsedTime);  printf("矩阵元素之和: %f\n", sum);  // 验证GPU和CPU结果差值  double diff\_norm = 0;  for (int i = 0; i < n \* n; i++) {  double diff = z[i] - cz[i];  diff\_norm += diff \* diff;  }  diff\_norm = sqrt(diff\_norm);  printf("Two-norm of difference between host and device = %e\n", diff\_norm);  // 释放所有资源  free(x);  free(y);  free(z);  free(cz);  free(sum\_p);  cudaFree(d\_x);  cudaFree(d\_y);  cudaFree(d\_z);  return 0;  } |

## 五、实验结果和分析

*实验结果以及对实验结果的比较分析和综合概括。*

表1 并行程序在不同方阵阶数下的执行时间和加速比（无预热）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 方阵阶数  执行时间 | 32 | 64 | 128 | 256 | 512 | 1024 |
| 第1次 | 0.000226 | 0.000227 | 0.000248 | 0.000284 | 0.000648 | 0.002398 |
| 第2次 | 0.000223 | 0.000228 | 0.000248 | 0.000280 | 0.000708 | 0.002323 |
| 第3次 | 0.000222 | 0.000197 | 0.000209 | 0.000252 | 0.000680 | 0.002296 |
| 第4次 | 0.000192 | 0.000232 | 0.000243 | 0.000247 | 0.000658 | 0.002300 |
| 第5次 | 0.000221 | 0.000204 | 0.000251 | 0.000282 | 0.000693 | 0.002230 |
| 平均值 | 0.000217 | 0.000218 | 0.000240 | 0.000269 | 0.000677 | 0.002309 |
| CPU时间 | 0.000062 | 0.000535 | 0.004365 | 0.034487 | 0.298805 | 3.072881 |
| 加速比 | 0.285978 | 2.458640 | 18.202669 | 128.204461 | 441.105698 | 1330.597125 |

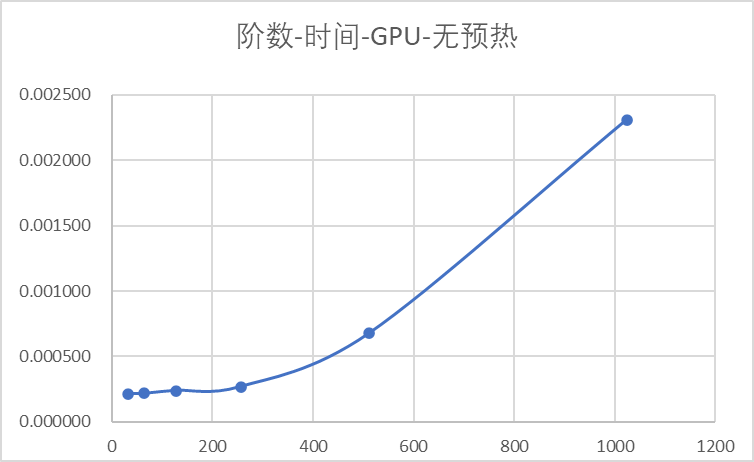
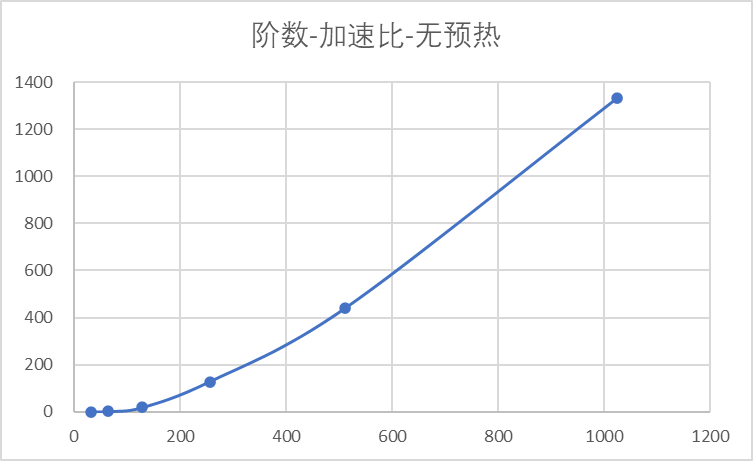
 

图1 GPU运行下的阶数-时间（无预热） 图2 GPU运行下的阶数-加速比（无预热）

如表1所示，在无预热的情况下，随着方阵阶数的上升，GPU进行矩阵乘法并求和的时间逐渐增加，但在阶数较低的情况下能看到一定的固定时常（约0.0002秒），经过分析可知，这是GPU启动所需的时间，因此在无预热的情况下所得趋势并不准确，需要进一步进行分析。而加速比能够持续上升的原因，一方面是GPU多线程的充分利用，另一方面GPU启动时间也被稀释，因此整体趋势与下方有预热的基本一致。（预热是指在进行GPU计算前先运行一次矩阵乘法，提前初始化GPU驱动与内存资源，消除首次调用时的编译优化、线程调度等随机性延迟）

表2 并行程序在不同方阵阶数下的执行时间和加速比（预热）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 方阵阶数  执行时间 | 32 | 64 | 128 | 256 | 512 | 1024 |
| 第1次 | 0.000031 | 0.000039 | 0.000052 | 0.000082 | 0.000418 | 0.002038 |
| 第2次 | 0.000034 | 0.000043 | 0.000052 | 0.000081 | 0.000403 | 0.002036 |
| 第3次 | 0.000031 | 0.000040 | 0.000053 | 0.000081 | 0.000412 | 0.002038 |
| 第4次 | 0.000032 | 0.000038 | 0.000053 | 0.000083 | 0.000411 | 0.002037 |
| 第5次 | 0.000031 | 0.000039 | 0.000052 | 0.000082 | 0.000415 | 0.002029 |
| 平均值 | 0.000032 | 0.000040 | 0.000052 | 0.000082 | 0.000412 | 0.002036 |
| CPU时间 | 0.000068 | 0.000566 | 0.004542 | 0.034228 | 0.396559 | 3.072881 |
| 加速比 | 2.138365 | 14.221106 | 86.679389 | 418.435208 | 962.989315 | 1509.570151 |

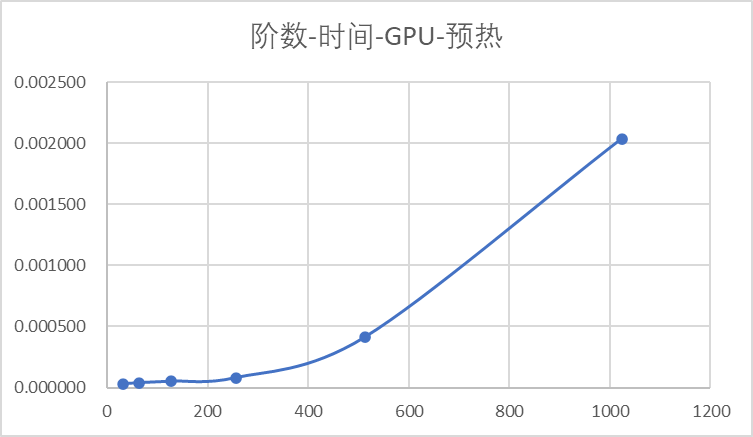
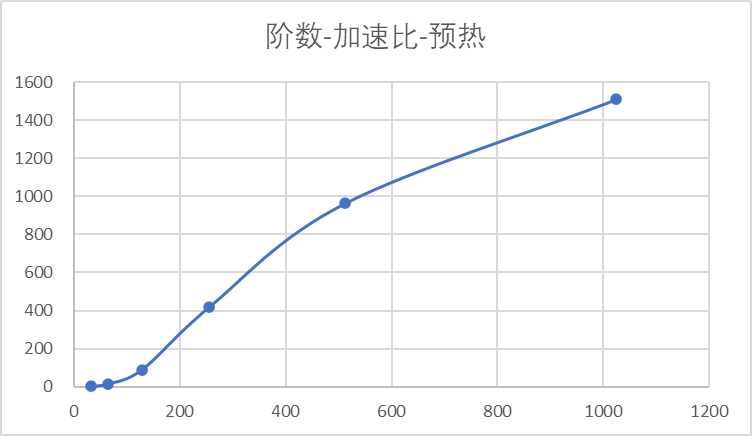
 

图3 GPU运行下的阶数-时间（预热） 图4 GPU运行下的阶数-加速比（预热）

如表2所示，在进行预热之后，在方阵阶数较小的情况下，GPU的运行时间变短，更加符合实际的计算所需时间，对比多个方阵大小在预热与非预热的差值，计算得到差值大约为0.0002，与上方分析一致。

使用预热得到的数据进行分析，可以看到随着阶数的增加，所需时间增加的趋势呈指数的方式递增，对比GPU与CPU结果，随着阶数的上升，CPU的计算时间显著提升，这是因为本次实验CPU并未采用并行的策略，而是直接采用基本矩阵乘法，并使用遍历的方式进行矩阵求和，总的时间复杂度为，对比二者的加速比，可以看到预热后的加速比更高，在各个阶数都能显著高于非预热情况，这说明预热对GPU计算非常重要。

进一步分析GPU计算的时间复杂度，分为两个阶段，分别时矩阵乘法和矩阵求和

①矩阵乘法：

|  |
| --- |
| ·每个线程计算matC的一个元素，需要*n*次乘加运算  ·总线程数 = 网络大小 × 块大小 =  ·因此计算时间为 ，同时运行p个线程，因此理论时间复杂度为 |

②矩阵求和

|  |
| --- |
| 采用树形归约  ·归约的数据量为  ·同时运行p个线程，每个线程计算量为 ，每轮归约后剩余数据量为  ·计算得到需要执行的轮数为 ，时间复杂度为 |

③GPU总复杂度

|  |
| --- |
| 结合上述两个过程可以得到时间复杂度为  ·考虑到实验过程 ，理论的时间复杂度约为 |

对比GPU和CPU的时间复杂度，可以发现理论加速比应该为 ，但实际运行的复杂度与理论值不一致，在阶数较小时加速比低于n，当阶数较大时加速比大于n，这可能与共享内存的使用有关。实验过程采用块内共享内存替代了全局内存，块内的访问速度大幅加快，在n=1024的情况下，使用全局内存的运行时间约为0.13秒，而使用共享内存的情况下运行时间约为0.002秒，提升了约50倍。因此在块内通讯部分，共享内存能大幅减少通信时间。但阶数较大的时候，每个块内的线程数都会接近饱和，能够充分利用共享内存减少时间，而但阶数较小，所需的线程少，共享内存的提升不明显。

表3 并行程序在不同方阵阶数下的误差平均值

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 方阵阶数  误差 | 32 | 64 | 128 | 256 | 512 | 1024 |
| 平均值 | 2.60E-14 | 9.91E-14 | 4.00E-13 | 1.62E-12 | 6.49E-12 | 2.59E-11 |

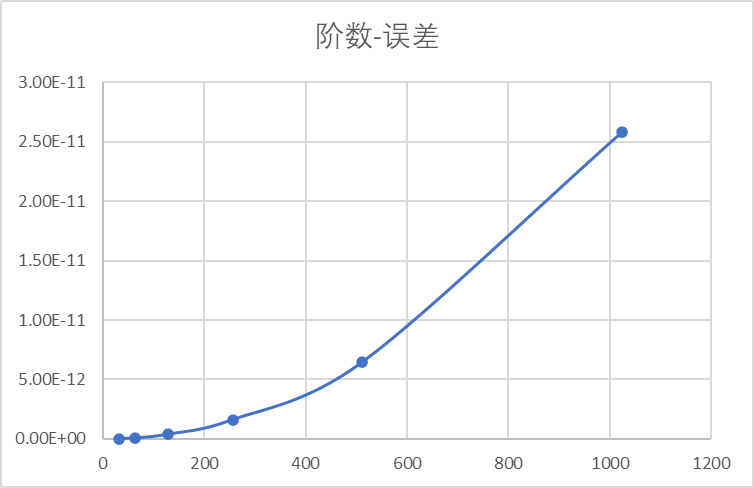


图5 阶数-误差图

分析表3可知，随着方阵阶数的上升，CPU与GPU计算结果的误差是逐渐上升的，这是因为此处计算误差的方式是比累加每个元素之间的误差（绝对值），因此随着阶数的增加，元素数量逐渐增加，元素间的误差也会被逐渐放大。

误差的数量级范围在1.0×10-14~1.0×10-11之间，说明GPU并行计算与标准的矩阵乘法得到的结果非常相似，没有因为并行导致数据丢失，答案计算正确。

表4 并行程序在不同块边长下的运行时间（固定n=1024，归约线程数=512）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 块边长  执行时间 | 1 | 2 | 4 | 8 | 16 | 32 |
| 平均时间 | 0.06208 | 0.015631 | 0.003988 | 0.002045 | 0.002030 | 0.002031 |

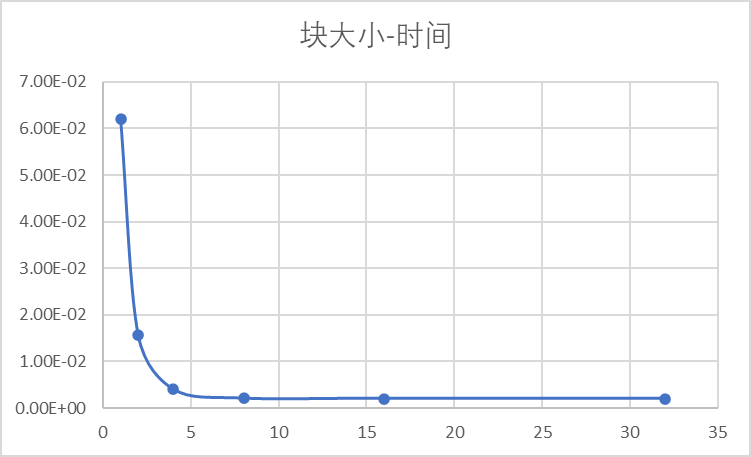


图6 块大小与运行时间的关系

实验验证了块的大小与运行时间的关系，如上表4数据所示，随着块的大小（块的边长）的增加，运行时间逐渐减少，已知CUDA块内最多为1024个线程，对应块边长为32，因此此处只分析块边长小于等于32的情况。观察数据，但块边长小于8的时候，随着块增加其运行时间显著减少，一方面是SP使用率上升，另一方面是共享内存减少了数据通信时间。当块边长大于的时候，SP逐渐趋于饱和，此时块的大小对运行效率影响较小。

表5 并行程序在不同归约线程数的执行时间（固定n=1024，块边长=32）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 归约线程数  执行时间 | 2 | 4 | 8 | 16 | 32 |
| 执行时间 | 0.002305 | 0.002122 | 0.002064 | 0.002041 | 0.00203 |
| 归约线程数  执行时间 | 64 | 128 | 256 | 512 | 1024 |
| 执行时间 | 0.002033 | 0.002031 | 0.002029 | 0.002022 | 0.002030 |

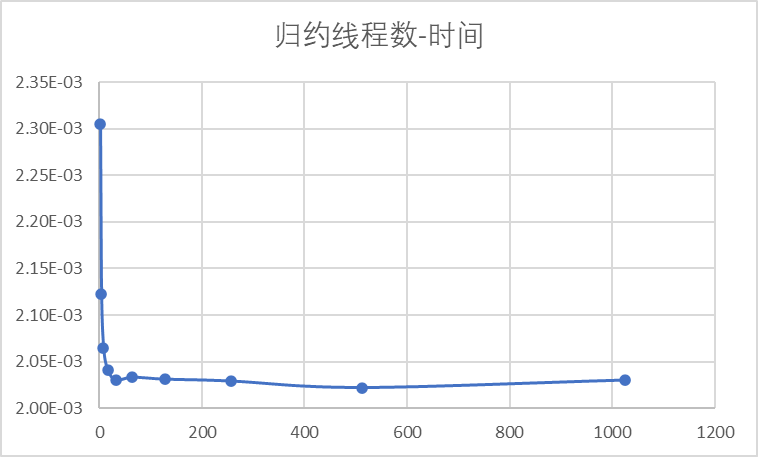


图7 归约线程数与时间的关系

实验也测试了树形归约的线程数与运行时间的关系，此处记录的时间为GPU计算的总时间，可以看到，当归约线程数小于等于8的时候其运行时间随线程数增加而显著下降，当线程数大于8时，归约时间基本没有变化，说明此时归约时间占很少的运行时间，增加线程对整体的提升很小。结合前面计算得到的归约时间复杂度可以知道，当大于8时再增加线程，复杂度减少非常小，对整体时间影响不大。

## 六、实验结论

*实验过程中遇到的问题及解决办法，运用了哪些技术方法以提高实验性能，从该实验得到的客观结论，等等。*

本次实验通过CUDA实现了矩阵乘法和求和，对比了GPU与CPU在不同矩阵规模下的性能表现。实验数据表明，预热能显著消除GPU首次调用的初始化开销。在无预热情况下，小规模矩阵（如n=32）的计算时间稳定在0.0002秒左右，这与GPU内核启动和资源调度的固定延迟有关；而预热后，相同规模的计算时间降至0.00003秒，更真实反映了实际计算耗时。预热后，n=1024时加速比达1509倍，超过理论线性加速。这是因为共享内存优化作用，在矩阵乘法阶段显著降低了实际计算时间。

理论分析指出，GPU矩阵乘法的计算复杂度为 ，其中P为GPU并行计算能力（约等于线程数）。实验时，P≈n，因此实际复杂度趋近于。

实验通过调整块边长验证共享内存的重要性：当块边长从1增至32时，运行时间从0.062秒降至0.002秒，提升约30倍。这是因为较大的块能更充分利用共享内存，减少全局内存访问延迟。在n=1024时，块边长32的配置接近硬件极限（1024线程/块），此时共享内存的通信优化达到最佳效果，进一步增大块边长对性能无显著改善。

归约求和的理论复杂度为 ，其中p为归约线程数。实验结果显示：当归约线程数p从2增至1024时，总运行时间从0.0023秒降至0.0020秒，降幅有限。这是因为在n=1024时，归约仅需2步即可完成，线程数超过8后对总时间的影响可忽略。

GPU与CPU计算结果的误差数量级稳定在1.0×10-14~1.0×10-11，且误差随矩阵规模增大而缓慢累积。这是因为浮点运算的累加误差，但误差绝对值极小，验证了GPU并行计算的数值稳定性。差异主要由并行计算顺序与CPU串行顺序不同导致，不影响实际应用。