中山大学计算机学院本科生实验报告

(2020 学年秋季学期)

课程名称: 高性能计算程序设计　　任课教师: 黄聃　　批改人:

年级 + 班级	18 计科 8 班	专业（方向）	超算
学号	18340208	姓名	张洪宾
Email	2285075600@qq.com	完成时间	2020 年 12 月 13 号

目录

1 实验目的

- 通过解决实际问题，更加理解 CUDA 的结构。
- 熟悉 CUDA 的基本编程接口。
- 感受向量处理器的加速能力。

2 实验过程及核心代码

2.1 实验环境

2.1.1 硬件

在这里我采用了超算中心的 th2k 集群上的 GPU 节点, 先用 salloc -N 1 -p gpu_v100 -J zhb 申请一个节点，用 ssh 连接上去。上面的显卡情况如下：

```
[sysu_hpcedu_302@gpu31 ~]$ nvidia-smi
Wed Dec 16 10:50:42 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:8A:00.0 Off |                    0 |
| N/A   33C    P0    36W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:8B:00.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:B3:00.0 Off |                    0 |
| N/A   30C    P0    35W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:B4:00.0 Off |                    0 |
| N/A   32C    P0    38W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
[sysu_hpcedu_302@gpu31 ~]$
```

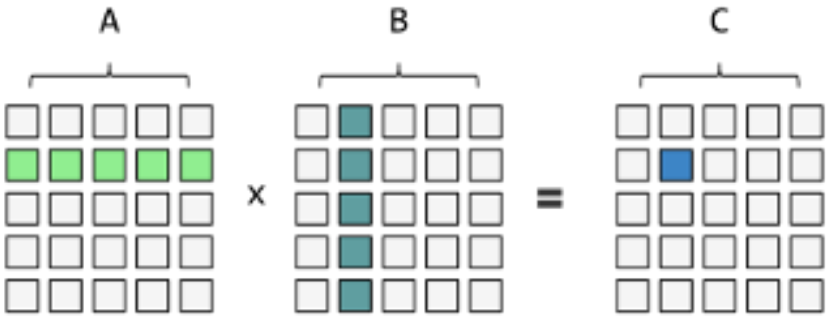
对应的 CPU 采用了 Intel(R) Xeon(R) Gold 6132 CPU @ 2.60GHz。

2.1.2 软件

- 操作系统：CentOS Linux release 7.6.1810 (Core)
- Toolkit:Cuda compilation tools, release 10.1, V10.1.243

2.2 实现 CUDA 版本的通用矩阵乘法

通用矩阵乘法如下：



在这里我参考了CUDA C++ Programming Guide中的 Thread Hierarchy 的部分，发现官网的教程是用如下的方式来让核函数取到 GPU 显存中的矩阵的数值的：

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
```

因此我们很容易想到，我们只需要将矩阵乘法的单次相加改为计算结果矩阵中对某一个元素即可。

在这里我编写的核函数如下：

```
__global__ void MatMul(double * A, double * B ,double * C,int m,int n,int k)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    double sum = 0;
    if (i < m && j < k){
        for(int x = 0;x < n;x++){
            sum += A[i * n + x] * B[x * k + j];
        }
        C[i * k + j] = sum;
    }
}
```

然后我们就要合理地给核函数分配线程来执行了。CUDA 的线程模型如下：

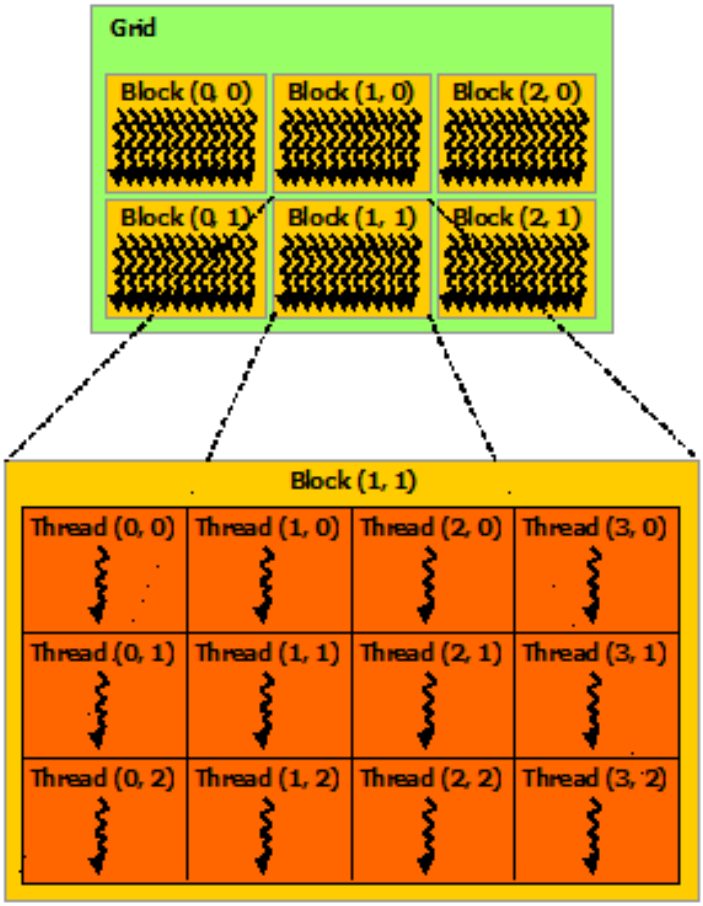


图 1: CUDA 线程模型

在该问题中，因为最终的结果矩阵 C 有 $m \times k$ 个元素，每个元素都需要用一个线程来计算，所以我们需要 $m \times k$ 个线程。

而查看 GPU 的信息，我们可以知道，我们使用的 V100 的一个 block 最多可以有 1024 个线程，而我们可以通过指定 dim3 变量来决定每个 block 要以什么样的拓扑来执行核函数。而不同的线程拓扑也会使得 grid 中 block 的数量和放置变得不同，但是总的来说，在我们的问题中，grid 的长度应该是 m 个线程，宽度应该是 k 个线程，这样就可以保证结果数组中的每一个元素都会被计算。

为了确保不会出现线程不够用的情况，我找到了查看 GPU 相关参数的程序，查看对应的参数，因为四张卡都是相同的，在这里我只展示第一张卡的结果：

```
Device0:"Tesla V100-SXM2-16GB"
Total amount of global memory                   4029153280 bytes
Number of mltiprocessors                        80
Total amount of constant memory:                65536 bytes
Total amount of shared memory per block         49152 bytes
Total number of registers available per block:  65536
Warp size                                       32
Maximum number of threada per block:            1024
Maximum sizes of each dimension of a block:     1024 x 1024 x 64
Maximum size of each dimension of a grid:       2147483647 x 65535 x 65535
Maximum memory pitch :                          2147483647 bytes
Texture alignmemt                               32 bytes
Clock rate                                      1.53 GHz
```

图 2: Tesla V100-SXM2-16GB 参数

可以看到一个 grid 中各个维度的最大值为 $2147483647 \times 65535 \times 65535$，用来解决我们这次的问题绰绰有余，因此可以放心地分配 block。

于是我们可以通过指定每个 block 的 thread 结构中前两个维度 x,y，进而计算每个 grid 的 block 结构的两个维度，在这个问题中是 $\lceil \frac{m}{x} \rceil$ 和 $\lceil \frac{k}{y} \rceil$，来指定核函数的执行方式。

这部分的核心代码如下：

```
1   timeval t1, t2;
2   int x,y;
3   cout << "Input threadsPerBlock.x:";
4   cin >> x;
5   cout << "Input threadsPerBlock.y:";
6   cin >> y;
7   dim3 threadsPerBlock(x,y);
8   int m, n, k;
9   cout << "Input problem size:";
10  cin >> m;
11  n = m;
12  k = m;
13
14  dim3 numBlocks((m % threadsPerBlock.x) ? m / threadsPerBlock.x + 1 : m /
        threadsPerBlock.x ,(k % threadsPerBlock.y) ? k / threadsPerBlock.y + 1 : k /
        threadsPerBlock.y);
15  double *A,*B,*C;
16  A = (double*)malloc(sizeof(double) * m * n);
17  B = (double*)malloc(sizeof(double) * k * n);
18  C = (double*)malloc(sizeof(double) * m * k);
19  for(int i = 0;i < m;i++){
20      for(int j = 0;j < n;j++){
```

```
            A[i * n + j] = rand() % 10;
        }
    }
    for(int i = 0;i < n;i++){
        for(int j = 0;j < k;j++){
            B[i * k + j] = rand() % 10;
        }
    }
    memset(C,0,sizeof(C));

    double * d_A,*d_B,*d_C;
    gettimeofday(&t1, NULL);
    cudaMalloc(&d_A, sizeof(double) * m * n);
    cudaMalloc(&d_B,sizeof(double) * n * k);
    cudaMalloc(&d_C,sizeof(double) * m * k);
    cudaMemcpy(d_A, A, sizeof(double) * m * n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, sizeof(double) * n * k, cudaMemcpyHostToDevice);
    MatMul<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, m, n, k);
    cudaMemcpy(C, d_C, sizeof(double) * m * k, cudaMemcpyDeviceToHost);
    gettimeofday(&t2, NULL);
    printf("GPU time is:%ld s\n",t2.tv_sec*1000000 + t2.tv_usec - t1.tv_sec*1000000 - t1
        .tv_usec);
```

为了验证结果的准确性，我编写了一个用 CPU 执行 GEMM 的函数，如下：

```
void CPU_MatMul(double * A, double * B ,double * C,int m,int n,int k){
    for(int i = 0;i < m;i++){
        for(int j = 0;j < k;j++){
            for(int x = 0;x < n;x++){
                C[i * k + j] += A[i * n + x] * B[x * k + j];
            }
        }
    }
}
```

执行完 GPU 的部分后，执行如下代码：

```
    CPU_MatMul(A,B,C1,m,n,k);
    gettimeofday(&t2, NULL);
    printf("CPU time is:%ld s\n",t2.tv_sec*1000000 + t2.tv_usec - t1.tv_sec*1000000 - t1
        .tv_usec);

    int flag = 0;
```

```
    for(int i = 0;i < m * k;i++){
        if(fabs((C[i] - C1[i])) > 1e-4){
            flag = 1;
            break;
        }
    }
    if(flag){
        cout << "Wrong result." << endl;
    }
    else {
        cout << "The results are correct. " << endl;
    }
```

即可验证 GPU 部分的正确性。在接下来的两个任务中我也将用这个函数来验证结果的正确性。

2.3 实现 OpenMP+CUDA 的多层次并行矩阵乘法

在这里 GPU 的部分还是采用任务一使用的 CPU_MatMul，在 CPU 的部分要考虑如何用 OpenMP 来优化。

我首先对任务 1 的代码进行修改，测试了不考虑 cudaMemcpy 的纯计算的时间，结果发现，在我们测试的问题规模下，除去 cudaMemcpy 几乎都只需要十几微秒，说明用 GPU 计算的瓶颈主要在 cudaMemcpy 的过程中。而我又去了解了一下，V100 使用 PCI-E 接口与 CPU 相连，而单个 PCI-E 接口的带宽是有限的，这也就意味着，如果在单 GPU 加 openMP 下，多个线程同时发送数据并不会使得总带宽增加，相反，因为 CPU 要维护 OpenMP 线程，所以总的时间不降反增。

而如果有多张 GPU 的话，就可以采用 OpenMP 做优化了，可以利用多张卡的 PCI-E 接口，增大总的传输带宽。

在这里我使用了 lspci | grep NVIDIA 来查看 NVIDIA 的设备，发现四张卡分布在四个接口上，所以我们可以用 OpenMP 来加大总带宽。

```
[sysu_hpcedu_302@gpu4 ~/asc21/zhb/hpc_lab6/Performance]$ lspci | grep NVIDIA
8a:00.0 3D controller: NVIDIA Corporation GV100GL [Tesla V100 SXM2 16GB] (rev a1)
8b:00.0 3D controller: NVIDIA Corporation GV100GL [Tesla V100 SXM2 16GB] (rev a1)
b3:00.0 3D controller: NVIDIA Corporation GV100GL [Tesla V100 SXM2 16GB] (rev a1)
b4:00.0 3D controller: NVIDIA Corporation GV100GL [Tesla V100 SXM2 16GB] (rev a1)
[sysu_hpcedu_302@gpu4 ~/asc21/zhb/hpc_lab6/Performance]$
```

图 3: lspci 中 NVIDIA 的设备

在这里我使用 cudaSetDevice 函数来指定某个 GPU 来执行。然后就是问题划分的问题了，因为在 GPU 之间的内存不是共享的，所以在这里可以采用类似于 MPI 矩阵乘法划分的方式，条带化矩

阵 A，将问题划分为矩阵 A 的若干个子矩阵与整个矩阵 B 相乘。

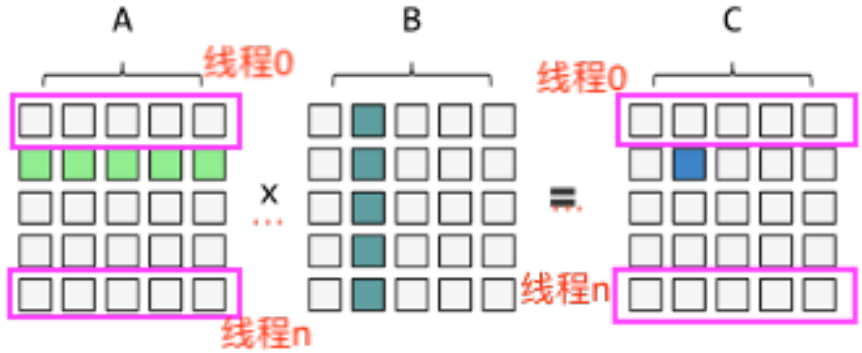


图 4: 线程划分方式：矩阵 A 分为若干块，每个包含 $\frac{m}{thread\_count}$ 行，每个线程将自己的块分发给自己对应的 GPU，将整个矩阵 B 则发给自己对应的 GPU，根据这些数据每个 GPU 就可以算出对应的 C 的块返回给 host 的线程，汇总得到了正确结果

这一部分与任务一不同的代码如下：

```
int gpu_count;
cudaGetDeviceCount(&gpu_count);
#pragma omp parallel num_threads(omp_threads)
{
    int id = omp_get_thread_num();
    int size = omp_get_num_threads();
    cudaSetDevice(id % gpu_count);
    double * d_A,*d_B,*d_C;
    cudaMalloc(&d_A, sizeof(double) * m * n / size);
    cudaMalloc(&d_B,sizeof(double) * n * k);
    cudaMalloc(&d_C,sizeof(double) * m * k / size);

    cudaMemcpy(d_A, A + id * m * n / size, sizeof(double) * m * n / size,
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, sizeof(double) * n * k, cudaMemcpyHostToDevice);
    MatMul<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, m / size, n, k);
    cudaMemcpy(C + id * m * k / size, d_C, sizeof(double) * m * k / size,
        cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

2.4 用 CUBLAS 实现矩阵乘法

阅读cuBLAS 的教程可以知道，cuBLAS 的数据的存放方式和 CPU 中的存放方式不太一样，是列优先的，而且其索引也是依据 Fortran，从 1 开始。

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

cuBLAS 中能用于运算矩阵乘法的函数有 4 个，分别是 cublasSgemm（单精度实数）、cublasDgemm（双精度实数）、cublasCgemm（单精度复数）、cublasZgemm（双精度复数），在这里因为我采用了 double 类型，所以我重点关注了 cublasDgemm，如下：

```
#define cublasDgemm cublasDgemm_v2
CUBLASAPI cublasStatus_t CUBLASWINAPI cublasDgemm_v2
(
    cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const double *alpha,
    const double *A, int lda,
    const double *B, int ldb,
    const double *beta,
    double *C, int ldc
);
```

该函数有 14 个参数，其中的 handle 与我们平时常见的句柄类似，需要用 cublasCreate() 创建和 cublasDestroy() 销毁。而 transa，transb 则是用来决定输入的矩阵是否需要转置，cublasOperation_t 的定义如下：

```
typedef enum {
    CUBLAS_OP_N = 0,//不转置
    CUBLAS_OP_T = 1,//普通转置
    CUBLAS_OP_C = 2 //共轭转置
}cublasOperation_t;
```

值得注意的是，因为 cublas 中默认的是用列优先的，所以在 C++ 中的行优先在这里是需要选择转置的，也就是说，在我们这次作业中，需要使用 CUBLAS_OP_T 来作为 transa 和 transb 的值。

而参数中的 m，n，k 就是问题中的 m，n，k。

然后是关于变量 alpha 和 beta，有如下公式：

$$C = \alpha A \cdot B + \beta C$$

所以我们要计算通用矩阵乘法的时候，只需要让 $\alpha = 1, \beta = 0$。

最后是三个矩阵指针 A,B,C 和对应的主维 lda,ldb 和 ldc，对于矩阵 A 和 B，因为选择了转置，所以主维是它们的列数。而对于矩阵 C，返回的结果一定是列优先的，所以主维是 C 的行数。

因此，核心代码如下：

```
double a = 1,b = 0;
cublasDgemm(
        handle,
        CUBLAS_OP_T, CUBLAS_OP_T,
        m, n, k,
        &a, //alpha
        d_A, n,
        d_B, k,
        &b, //beta
        d_C, m
);
```

其他部分与任务 1 大致相同，不过值得注意的是，在这里因为返回的 d_C 是列主元的，所以我们在用 CPU 计算验证矩阵的时候就应该将原来的算法改成列优先的，如下：

```
void CPU_MatMul(double * A, double * B ,double * C,int m,int n,int k){
    for(int i = 0;i < m;i++){
        for(int j = 0;j < k;j++){
            for(int x = 0;x < n;x++){
                C[IDX2C(i,j,k)] += A[i * n + x] * B[x * k + j];
            }
        }
    }
}
```

3 实验结果

3.1 准备工作

在这里我编写了 Makefile，如下：

```
CXX = nvcc
CXXFLAGS = -O3
```

```
all: CUDA_MatMul omp_CUDA_MatMul CUBLAS_MatMul

CUDA_MatMul: Q1/CUDA_MatMul.cu
	$(CXX) $(CXXFLAGS) -o $@ $<

omp_CUDA_MatMul: Q2/omp_CUDA_MatMul.cu
	$(CXX) $(CXXFLAGS) -o $@ $< -Xcompiler -fopenmp

CUBLAS_MatMul: Q3/CUBLAS_MatMul.cu
	$(CXX) $(CXXFLAGS) -o $@ $< -lcublas
```

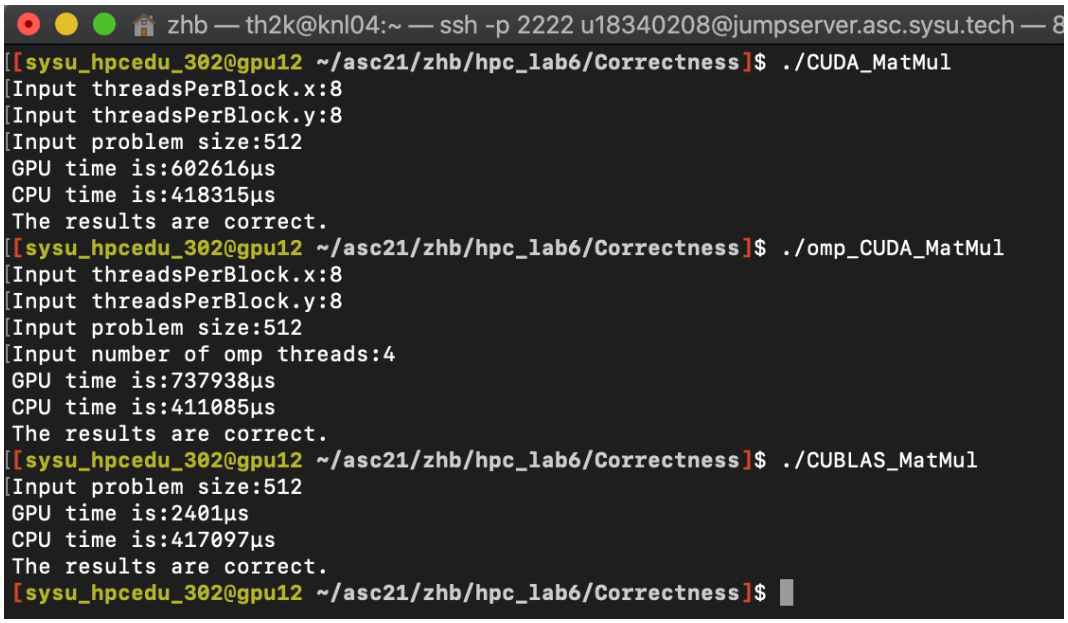
module load CUDA 后，就可以用 make 命令来编译，就可以生成可执行文件了。

在这里我有两个版本,第一个是用来验证 CUDA 程序正确性的 Correctness,其中有源码和 Makefile，源码中有验证正确性的模块 CPU_MatMul, 另外一个则是测试性能的 Performance，只是去掉了验证结果正确性的部分。

3.2 验证结果

进入 Correctness 后，输入 make，然后就可以得到可执行程序。因为如果问题规模过大的话，CPU 执行的时间会多达六七个小时甚至若干天，所以我将问题规模定为 512。

验证的结果如下：



可以看到三个任务的代码的结果都是正确的。

3.3 程序性能分析与比较

通过测试，我发现问题规模为 8192 的时候问题规模仍然是较小的，不能完整地反映实验结果，因此我将测试的范围增加为从 512 到 16384.

并且在问题 1 和问题 2 中，相同线程数量可能会有不同的组织结构，查阅相关资料，发现当 threadsPerBlock.x = 1,threadsPerBlock.y = n 的时候 Cache 重用较好, 这与数据访问的模式有关，在这里为了使得性能尽可能优越, 也为了控制变量, 我都让 threadsPerBlock.x = 1,threadsPerBlock.y = n。

3.3.1 任务一的性能测试

如图，执行 CUDA_MatMul 后，输入 threadsPerBlock.x 和 threadsPerBlock.y，再输入问题规模就可以计算了。

在不同的问题规模和 blockSize 下运行，结果如下：

问题规模 / Block size	512	1024	2048	4096	8192	16384
32	132094	389284	421993	849472	1898243	10973994
64	110241	387665	393679	786761	1528046	11204327
128	118779	234815	402230	829105	1427033	8134953
256	111208	239294	423046	813906	1980050	18369852
512	128648	366368	420556	861188	4119473	34398378

表 1: 不同问题规模在不同 Block size 下的运算时间 (μs)

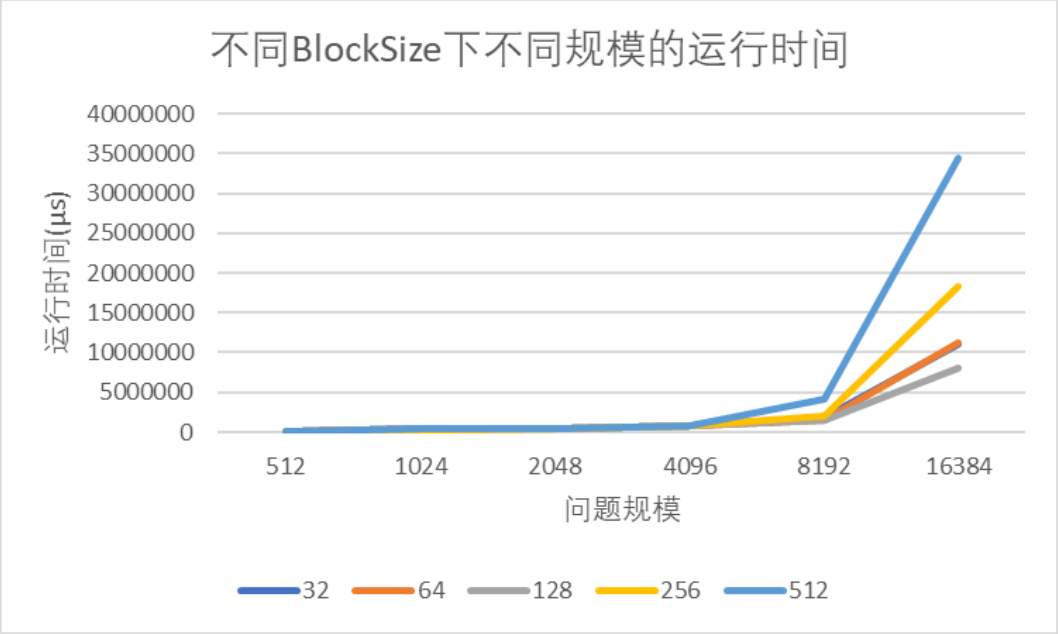
在运行的过程中，我开启了另外一个终端，用 watch -n 0.001 nvidia-smi 实时监控 GPU 状况，在运行任务的时候，如下：

```
Thu Dec 17 12:30:20 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:8A:00.0 Off |                    0 |
| N/A   43C    P0   190W / 300W |   6459MiB / 16130MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:8B:00.0 Off |                    0 |
| N/A   33C    P0    38W / 300W |     10MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:B3:00.0 Off |                    0 |
| N/A   32C    P0    37W / 300W |     10MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:B4:00.0 Off |                    0 |
| N/A   35C    P0    39W / 300W |     10MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0    321297      C   ./CUDA_MatMul                              6449MiB |
+-----------------------------------------------------------------------------+
```

可以看到运行任务的时候，某张卡被充分利用。

对数据做可视化：



从总体上来看，选择块大小为 128 的时候性能最优，并且问题规模越大的时候，优势越明显。

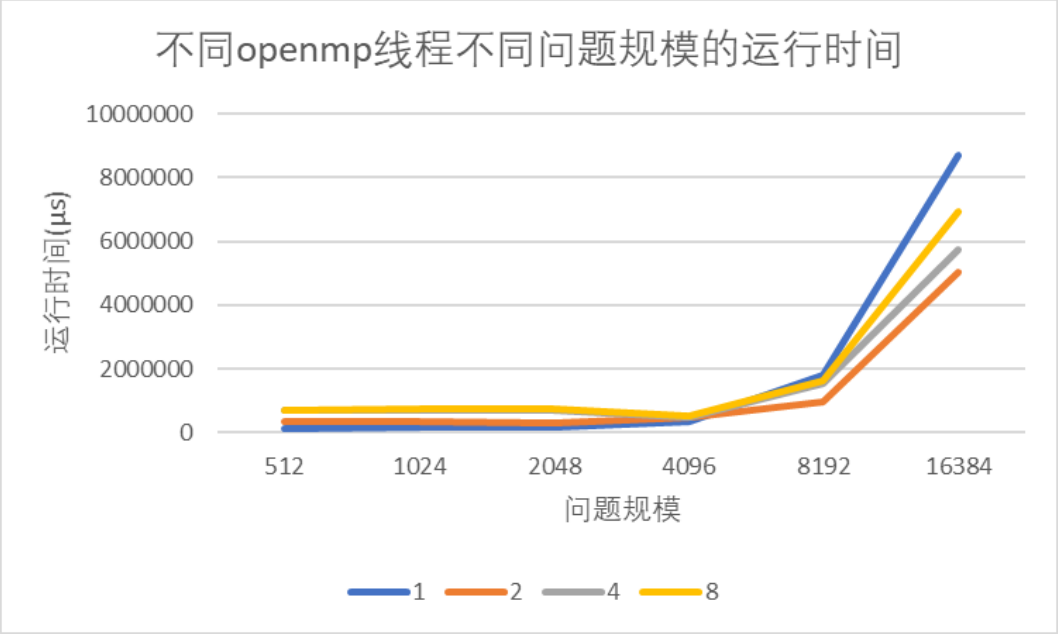
3.3.2 任务二的性能测试

在这里通过观察任务一的结果，发现当 threadsPerBlock.x = 1，threadsPerBlock.y = 128 时总体新能较好，于是在 openMP 版本的新能测试中，不同 openMP 线程下，CUDA 的块结果都指定为 threadsPerBlock.x = 1，threadsPerBlock.y = 128。运行结果如下：

线程数量 \ 问题规模	512	1024	2048	4096	8192	16384
1	136773	141603	172995	338291	1812926	8708693
2	331331	341716	278993	482778	941761	5014023
4	699120	696498	700328	475759	1546742	5739422
8	673617	715221	749895	518088	1611321	6936521

表 2: 不同线程下不同问题规模的运算时间 (μs)

对数据做可视化：



在前期问题规模比较小的时候，创建线程的开销在整个求解过程中的比重比较大，因此虽然并行加速了求解的阶段，但是无法抵消线程创建带来的开销，总的时间大于一个线程时的用时。而随着问题规模的增加，创建线程的开销不变，而计算的优化变得十分明显，因此总的时间会小于一个线程时的用时。

而性能的提升又主要是 Memcpy 上的优化，因为单个线程拷贝的数据减小了，可以用多张卡进

行计算，如下：

```
Thu Dec 17 15:39:06 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:8A:00.0 Off |                    0 |
| N/A   40C    P0   294W / 300W |   4411MiB / 16130MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:8B:00.0 Off |                    0 |
| N/A   39C    P0   291W / 300W |   4411MiB / 16130MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:B3:00.0 Off |                    0 |
| N/A   30C    P0    38W / 300W |     10MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:B4:00.0 Off |                    0 |
| N/A   30C    P0    37W / 300W |     10MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0    308920      C   ./omp_CUDA_MatMul                          4401MiB |
|    1    308920      C   ./omp_CUDA_MatMul                          4401MiB |
+-----------------------------------------------------------------------------+
```

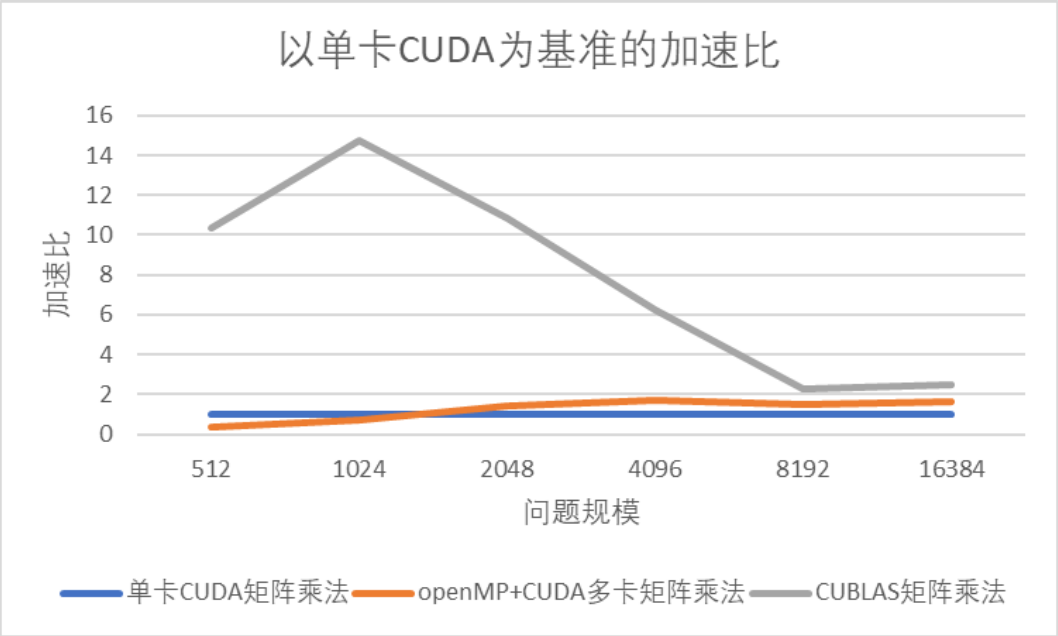
3.3.3 任务三的性能测试

直接运行./CUBLAS_MatMul, 输入问题规模即可，统计的结果如下：

问题规模	运行时间 (μs)
512	11467
1024	15948
2048	37064
4096	133160
8192	626313
16384	3244879

表 3: CUBLAS 矩阵乘法在不同规模下的运行时间 (μs)

为了更好地比较三个任务的性能，我从任务一和任务二中挑选出最优的运行参数与 CUBLAS 的矩阵乘法做比较，以单卡版本的 CUDA 矩阵乘法作为基准计算加速比，结果如下：



可以看到，在问题规模很小的时候，CUBLAS 的优势非常明显，对应的 openMP+CUDA 版本反而出现了负优化。而随着问题规模的增加，加速比上的差距逐渐缩小，但是 CUBLAS 的实现仍然有较大的优势。

3.4 改进策略

通过查阅相关的文献，我找到了一些提升速度的方法，虽然没有足够的时间去实现，但是我觉得也可以作为一个拓展的思路，等有时间的时候自己试一下。具体的方法如下：

- 在多卡版本的时候使用矩阵分块：我在实现 openMP+CUDA 的多卡版本的时候只对其中的一个矩阵进行了分块，但是事实上可以对两个矩阵都进行分块，然后用其他的方式计算出结果。通过这种方式减小了从内存发送到 GPU 的数据量，可以进一步减小 cudaMemcpy 占用的时间。
- 使用 shared memory、constant memory、device memory 或 device memory 进行优化。通过提升访存的速度来提升性能。
- 使用精度更小的 float 代替 double 进行优化。在误差允许的范围内，使用 float 会更快的计算得到结果。
- 尝试使用 float2,float3,float4 矢量类型来加速计算。

事实上还有很多优化的方法，但是目前对 CUDA 还不够熟悉，在将来的学习中会不断提升对优化的能力。

4 实验感想

第一次动手实践 CUDA，虽然很多东西做的不是很完善，但是还是学到了很多东西。特别是对 CUDA 处理向量的能力有了极大的认识，与 CPU 计算矩阵相比，GPU 对大规模矩阵的计算有碾压性的优势，并且我也了解到 GPU 的发展还未到达瓶颈，摩尔定律在 GPU 依旧适用，更加对 GPU 编程充满了兴趣。