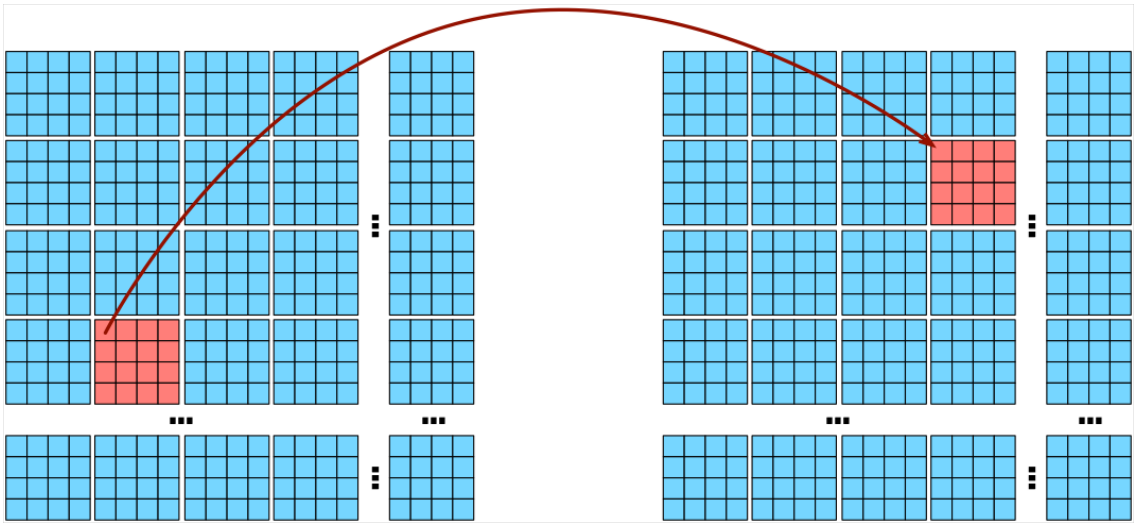
# 并行程序设计与算法实验 9

| 实验 | CUDA实现转置 | 专业 | 计算机科学与技术 |
| --- | --- | --- | --- |
| 学号 | 21311525 | 姓名 | 马梓培 |
| Email | mazp@mail2.sysu.edu.cn | 完成日期 | 2024/05/25 |

## 1. 实验要求

> 矩阵转置：
>
> - 使用CUDA完成并行矩阵转置
> - 随机生成n×n的矩阵A
> - 对其进行转置得到$A^T$
> - 分析不同线程块大小，矩阵规模，访存方式，任务/数据划分方式，对程序性能的影响
>
> 使用共享内存的CUDA矩阵转置
>
> - 将矩阵划分为数据块
> - 每个线程块处理一个数据块的转置
> - 从全局内存中读入第（i，j）个数据块至共享内存
> - 从共享内存中写出至第（j，i）个数据块



## 2. 实现代码

见附件或github

## 2.1 朴素矩阵转置

```
__global__ void transformNaiveRow(float * in,float * out,int nx,int ny)
{
    int ix=threadIdx.x+blockDim.x*blockIdx.x;
    int iy=threadIdx.y+blockDim.y*blockIdx.y;
    int idx_row=ix+iy*nx;
    int idx_col=ix*ny+iy;
    if (ix<nx && iy<ny)
    {
      out[idx_col]=in[idx_row];
    }
}
```

比较简单。总结来说，就是以行主序的方式从`in`中读取数据，再以列主序的方式写入进`out`。

然而，读取操作是合并的，但写入操作是非合并的，导致整体性能较低。

## 2.2 使用共享内存的CUDA矩阵转置

除ppt，借鉴自blog

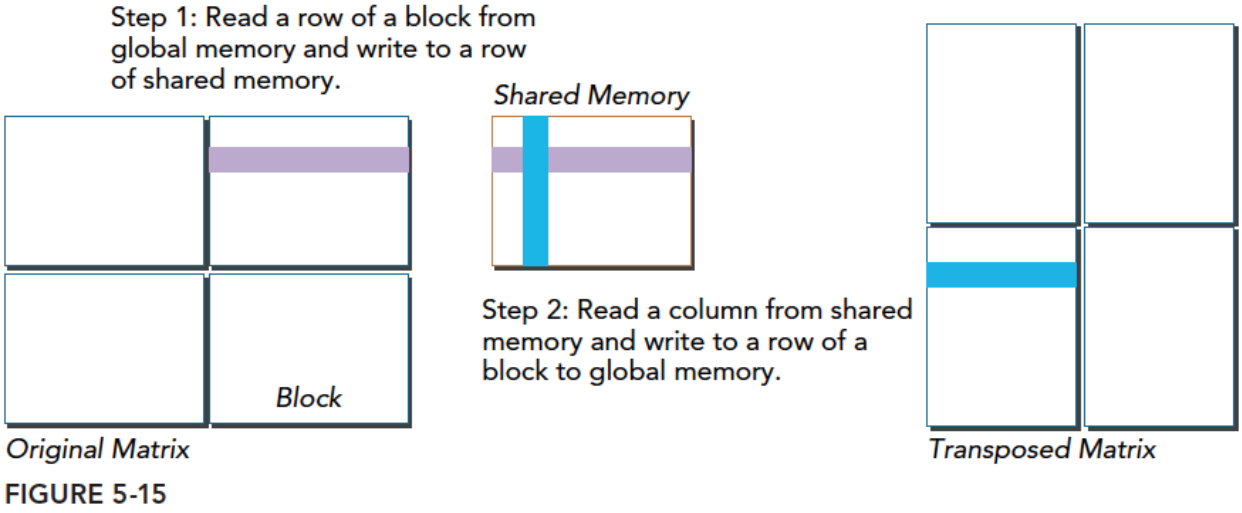


FIGURE 5-15

上面这个图就是我们的转置过程

- 从全局内存读取数据（按行）写入共享内存（按行）
- 从共享内存读取一列写入全局内存的一行

```
__global__ void transformSmem(float *in, float *out, int nx, int ny) {
    __shared__ float tile[BDIMY][BDIMX];
    unsigned int ix, iy, transform_in_idx, transform_out_idx;
    ix = threadIdx.x + blockDim.x * blockIdx.x; // 全局的，没转置的ix
    iy = threadIdx.y + blockDim.y * blockIdx.y; // 全局的，没转置的iy
    transform_in_idx = iy * nx + ix;            // 全局的 in 数组的下标

    unsigned int bidx, irow, icol;
    bidx = threadIdx.y * blockDim.x + threadIdx.x; // bidx 表示块内线程的线性索引
```

```
    irow = bidx / blockDim.y;                       // 转置后块内的行索引
    icol = bidx % blockDim.y;                       // 转置后块内的列索引

    ix = blockIdx.y * blockDim.y + icol; // 转置后全局的 ix
    iy = blockIdx.x * blockDim.x + irow; // 转置后全局的 iy

    transform_out_idx = iy * ny + ix; // 全局的 out 数组的线性索引

    if (ix < nx && iy < ny)
    {
        tile[threadIdx.y][threadIdx.x] = in[transform_in_idx];
        __syncthreads();
        out[transform_out_idx] = tile[icol][irow];
    }
}
```

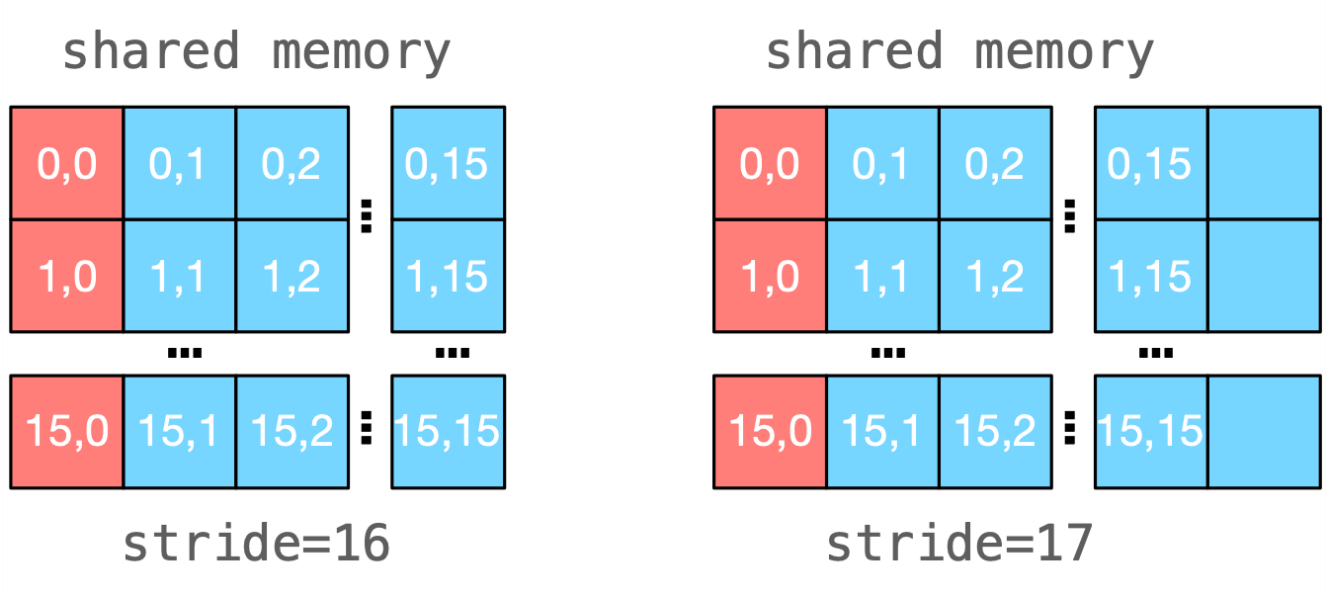
解释代码，这段代码是通用代码，并不需要矩阵为正方形（结合上面的图，过程会更清晰）：

1. 计算当前块中的线程的全局坐标（相对于整个网格），计算对应的一维线性内存的位置
2. bidx表示block idx也就是在这个快中的线程的坐标的线性位置（把块中的二维线程位置按照逐行排布的原则，转换成一维的），然后进行转置，也就是改成逐列排布的方式，计算出新的二维坐标，逐行到逐列排布的映射就是转置的映射，这只完成了很多块中的一块，而关键的是我们把这块放回到哪
3. 计算出转置后的二维全局线程的目标坐标，注意这里的转置前的行位置是计算出来的是转置后的列的位置，这就是转置的第二步。
4. 计算出转置后的二维坐标对应的全局内存的一维位置
5. 读取全局内存，写入共享内存，然后按照转置后的位置写入

## 2.3 使用填充共享内存的矩阵转置

共享内存访问：存储体冲突

- 从全局内存拷贝至共享内存过程中无冲突
- 从共享内存拷贝至全局内存有冲突！
    - 16-way conflicts：所有线程只对两个存储体进行操作
- 解决从共享内存拷贝至全局内存的存储体冲突
    - 给共享内存分配一列空白数据
    - Stride=17：无存储体冲突！




```cpp
__global__ void transformSmemPad(float *in, float *out, int nx, int ny) {
    __shared__ float tile[BDIMY][BDIMX + IPAD];
    unsigned int ix, iy, transform_in_idx, transform_out_idx;
    ix = threadIdx.x + blockDim.x * blockIdx.x;
    iy = threadIdx.y + blockDim.y * blockIdx.y;
    transform_in_idx = iy * nx + ix;

    unsigned int bidx, irow, icol;
    bidx = threadIdx.y * blockDim.x + threadIdx.x;
    irow = bidx / blockDim.y;
    icol = bidx % blockDim.y;

    ix = blockIdx.y * blockDim.y + icol;
    iy = blockIdx.x * blockDim.x + irow;

    transform_out_idx = iy * ny + ix;

    if (ix < nx && iy < ny)
    {
        tile[threadIdx.y][threadIdx.x] = in[transform_in_idx];
        __syncthreads();
        out[transform_out_idx] = tile[icol][irow];
    }
}
```

## 3. 实验结果

矩阵规模：

1. N = 2 ^ 10

- naive

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP9$ ./transform_matrix.out 0
Using device 0: NVIDIA GeForce RTX 3090
CPU Execution Time elapsed 0.005803 sec
transformNaiveRow  Time elapsed 0.000028 sec
Check result success!
```

- shared memory

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP9$ ./transform_matrix.out 1
Using device 0: NVIDIA GeForce RTX 3090
CPU Execution Time elapsed 0.005718 sec
transformSmem  Time elapsed 0.000032 sec
Check result success!
```

- padded

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP9$ ./transform_matrix.out 2
Using device 0: NVIDIA GeForce RTX 3090
CPU Execution Time elapsed 0.005768 sec
transformSmemPad  Time elapsed 0.000033 sec
Check result success!
```

2. N = 2 ^ 11

- naive

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP9$ ./transform_matrix.out 0
Using device 0: NVIDIA GeForce RTX 3090
CPU Execution Time elapsed 0.029241 sec
transformNaiveRow  Time elapsed 0.000097 sec
Check result success!
```

- shared memory

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP9$ ./transform_matrix.out 1
Using device 0: NVIDIA GeForce RTX 3090
CPU Execution Time elapsed 0.028520 sec
transformSmem  Time elapsed 0.000069 sec
Check result success!
```

- padded

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP9$
./transform_matrix.out 2
Using device 0: NVIDIA GeForce RTX 3090
CPU Execution Time elapsed 0.028300 sec
transformSmemPad  Time elapsed 0.000082 sec
Check result success!
```

3. N = 2 ^ 12

- naive

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP9$
./transform_matrix.out 0
Using device 0: NVIDIA GeForce RTX 3090
CPU Execution Time elapsed 0.160856 sec
transformNaiveRow  Time elapsed 0.000272 sec
Check result success!
```

- shared memory

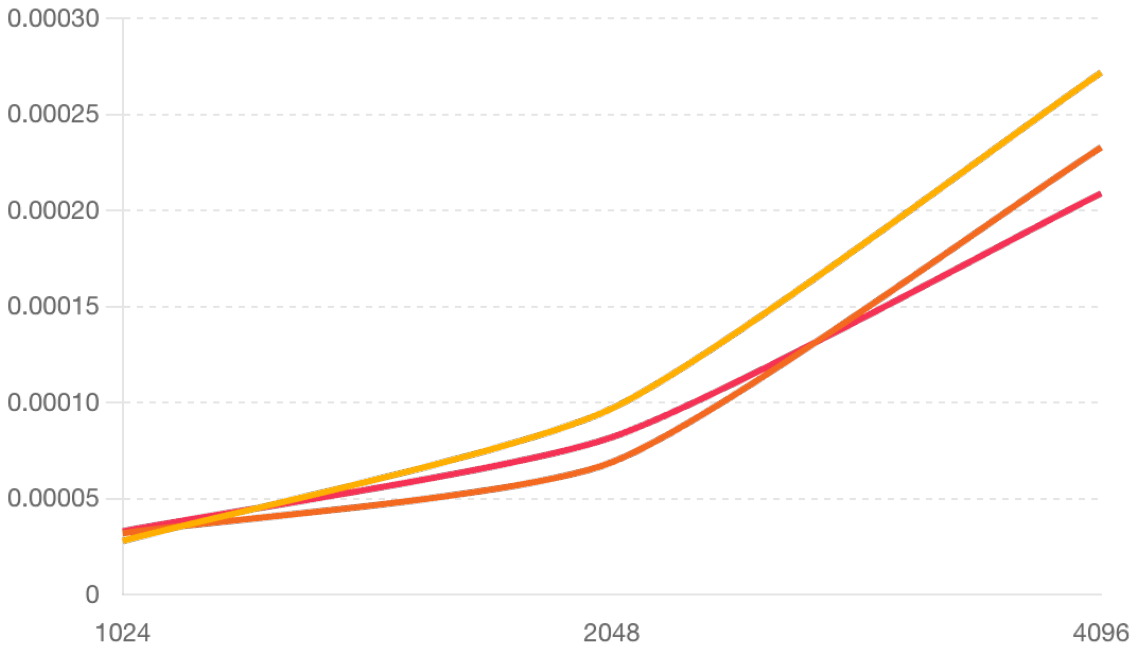
```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP9$
./transform_matrix.out 1
Using device 0: NVIDIA GeForce RTX 3090
CPU Execution Time elapsed 0.164755 sec
transformSmem  Time elapsed 0.000233 sec
Check result success!
```

- padded

```
(base) duantong@user-R8428-A12:/data/duantong/mazipei/Parallel-Programming/PP9$
./transform_matrix.out 2
Using device 0: NVIDIA GeForce RTX 3090
CPU Execution Time elapsed 0.165534 sec
transformSmemPad  Time elapsed 0.000209 sec
Check result success!
```

| **\N** | **2 ^ 10** | **2 ^ 11** | **2 ^ 12** |
|---|---|---|---|
| naive | 0.000028 | 0.000097 | 0.000272 |
| shared mem | 0.000032 | 0.000069 | 0.000233 |
| padded | 0.000033 | 0.000082 | 0.000209 |

可视化：




## 结果总结：

- 对于小矩阵（`2 ^ 10`），naive 实现略快于共享内存实现，这可能是因为共享内存的额外开销在小矩阵上不明显。
- 对于中等大小的矩阵（`2 ^ 11`），共享内存实现明显优于naive实现，因为合并的内存访问提高了效率。
- 对于大矩阵（`2 ^ 12`），使用共享内存和填充的实现（padded）性能最好，显著优于naive实现。这是因为填充减少了共享内存冲突，进一步提高了效率。

# 4. 实验感想

在本次 CUDA 实现矩阵转置的实验中，我学到了许多关于 GPU 并行计算的实际应用。通过对比朴素实现和使用共享内存及填充优化的实现方式，我深刻体会到了性能优化的重要性。初步的朴素实现由于未能充分利用 GPU 的共享内存，导致整体性能较低。而通过使用共享内存，将矩阵分块并在共享内存中进行转置，性能得到了显著提升，特别是在处理大规模矩阵（如 2^12 的矩阵）时，运行时间大大缩短。这使我认识到，在实际开发中，合理利用 GPU 资源和优化内存访问模式可以带来巨大的性能提升。

此外，通过实验，我不仅加深了对 CUDA 理论知识的理解，还学会了如何在实际编程中应用这些知识。在使用 Nsight进行性能分析时，我发现共享内存访问中的存储体冲突会严重影响性能。通过在共享内存中添加填充，使内存访问的步长为 17，从而避免了 16-way 冲突，进一步提高了程序的执行效率。这次实验不仅提高了我的编程能力，也让我体会到细致分析和优化程序性能的重要性。