# CS433 Parallel and distributed programming

# Matrix Transpose

Tianhao Li 5140309349

Chao Gao 5142029014

2016.12.22

## 1. Introduction

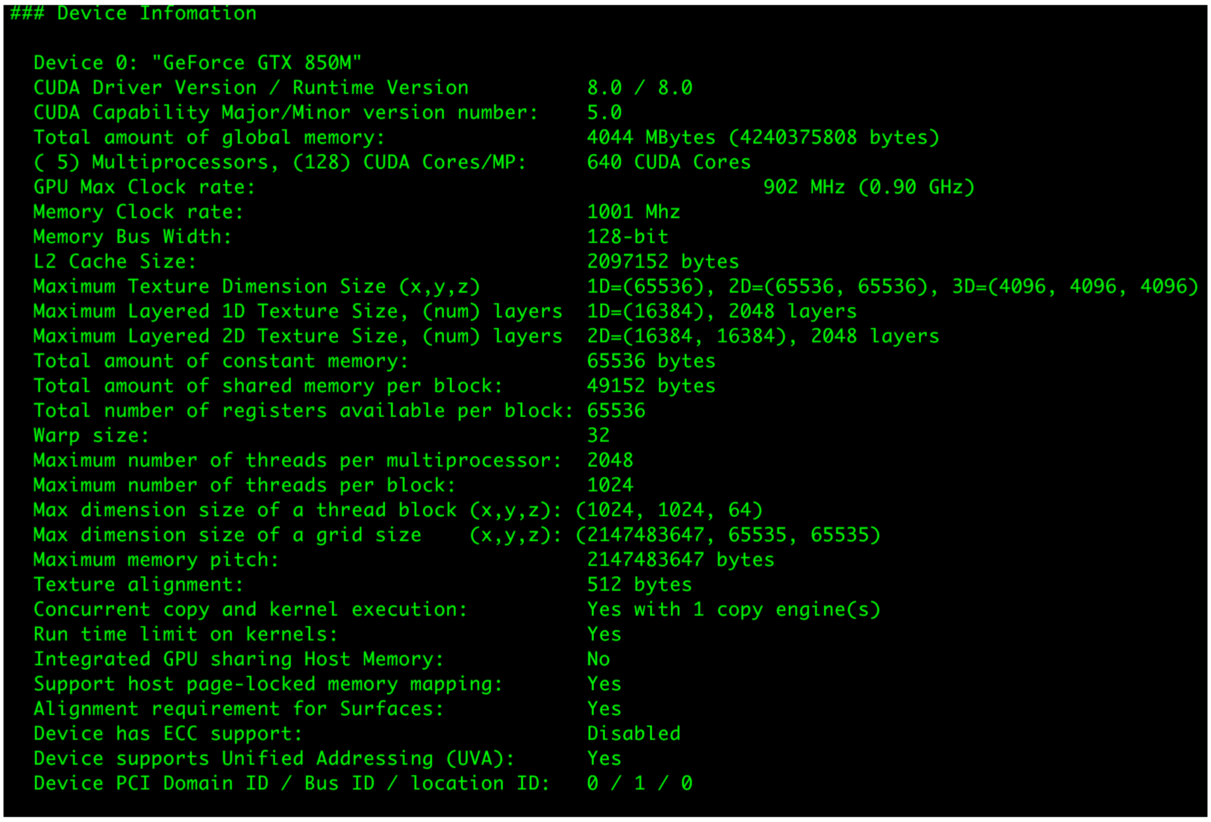
CUDA(Compute Unified Device Architecture) is the powerful computing platform published by NIVIDIA, which has become very popular nowadays. In this project, we will implement the matrix transpose problem by using CUDA. We will have 5 kinds of versions of matrix transpose programs: CPU transpose, naive GPU, shared memory GPU, no bank conflicts GPU, loop unrolled GPU. And we test our programs both on our own computer and Pi cluster.

## 2. File structure

- report.pdf ———— this file
- source code ———— five program files
- readme.md ———— hardware information and compile specification

## 3. Hardware information

Because the hardware information is a little bit long, I only show the hardware information of our own computer here, I get the information as below:



```
### Device Infomation

  Device 0: "GeForce GTX 850M"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 4044 MBytes (4240375808 bytes)
  ( 5) Multiprocessors, (128) CUDA Cores/MP:     640 CUDA Cores
  GPU Max Clock rate:                                           902 MHz (0.90 GHz)
  Memory Clock rate:                             1001 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  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 threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
```

And with CUDA scripts provided by TA, we also get the hardware information of K80. You can refer to the readme.md for details.

## 4. Implementation

### 4.1 CPU transpose

The CPU version of matrix transpose is quite easy. Just show the code.

```
void transpose(float **src, float **dst, int dim) {
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j)
            dst[i][j] = src[j][i];
    }
}
```

### 4.2 Naive GPU

This time we start GPU programming. This version is also very easy to implement. We use the CUDA keyword **__glocal__** indicates that the function runs on the device. Besides, we have to use **cudaMemcpy** to copy the data from the host to the device. At last, we add the function call from host code to device code which is called a “**kernel launch**”.

```
__global__ void kernel(float *d_src, float *d_dst, int cols, int rows) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    int index_in = i * cols + j;
    int index_out = j * rows + i;

    d_dst[index_out] = d_src[index_in];
}
```

**copy data**

```
cudaMalloc((void **) &d_src, size);
cudaMalloc((void **) &d_dst, size);
cudaMemcpy(d_src, src, size, cudaMemcpyHostToDevice);
```

**kernel launch**

```
for(int i=0;i<REP_TIME;++i)
        kernel <<< GridDim, BlockDim >>> (d_src, d_dst, cols, rows);
```

### 4.3 Shared Memory GPU

Naive GPU version has a problem of non-coalesced memory. We can use shared memory to improve the performance. In this implementation, we divide the whole matrix to 32 * 32 smaller matrices. Here we use the CUDA keyword **__share__ float mat[BLOCK_SIZE_X][BLOCK_SIZE_Y]** to declare the shared memory space.

```
__global__ void kernel(float *d_src, float *d_dst, int cols, int rows) {

    __shared__ float mat[BLOCK_SIZE_X][BLOCK_SIZE_Y];

    //block start element
    int bx=blockIdx.x*BLOCK_SIZE_X;  //block start x
    int by=blockIdx.y*BLOCK_SIZE_Y;  //block start y
    //thread element
    int i = by+ threadIdx.y;
    int j = bx+ threadIdx.x;
    //transfered element
    int ti=bx+threadIdx.y;
    int tj=by+threadIdx.x;

    //load element to corresponding block
    if(i<rows&&j<cols)
    mat[threadIdx.x][threadIdx.y]=d_src[i*cols+j];
    __syncthreads();

    if(tj<rows&&ti<cols)
    d_dst[ti*rows+tj]=mat[threadIdx.y][threadIdx.x];
}
```

## 4.4 No Bank Conflict GPU

Due to the bank conflicts of the Shared Memory GPU version, we are still not close to max bandwidth. So in this version, we start to eliminate the bank conflicts. When we access to the different elements in a bank will cause bank conflicts, we simply add an unused column to avoid bank conflict. The remaining code is the same as the previous version.

```
__global__ void kernel(float *d_src, float *d_dst, int cols, int rows) {

    __shared__ float mat[BLOCK_SIZE_X][BLOCK_SIZE_Y+1];

    //block start element
    int bx=blockIdx.x*BLOCK_SIZE_X;  //block start x
    int by=blockIdx.y*BLOCK_SIZE_Y;  //block start y
    //thread element
    int i = by+ threadIdx.y;
    int j = bx+ threadIdx.x;
    //transfered element
    int ti=bx+threadIdx.y;
    int tj=by+threadIdx.x;

    //load element to corresponding block
    if(i<rows&&j<cols)
    mat[threadIdx.y][threadIdx.x]=d_src[i*cols+j];
    __syncthreads();

    if(tj<rows&&ti<cols)
    d_dst[ti*rows+tj]=mat[threadIdx.x][threadIdx.y];
}
```

## 4.5 Loop Unrolled GPU

We already achieve very good performance now. In this version, we start to implement loop unrolled version. Actually, it is the software-level optimization(compiler optimization), not like the hardware-level optimizations before. What we need to do is to simply add the CUDA directive **#progma unroll** to the kernel execution. The code is as below.

```
__global__ void kernel(float *d_src, float *d_dst, int cols, int rows) {

    __shared__ float mat[TILE][TILE+1];

    //block start element
     int bx=blockIdx.x*TILE;  //block start x
     int by=blockIdx.y*TILE;  //block start y
     //thread element
     int i = by+ threadIdx.y;
     int j = bx+ threadIdx.x;

    #pragma unroll
      for(int k=0;k<TILE;k+=SIDE){
        if(i+k<rows&&j<cols)
          mat[threadIdx.y+k][threadIdx.x]=d_src[((i+k)*cols)+j];
      }

    __syncthreads();

    int ti=bx+threadIdx.y;
    int tj=by+threadIdx.x;
    #pragma unroll
    for(int k=0;k<TILE;k+=SIDE){
      if((ti+k)<cols&&tj<rows)
        d_dst[(ti+k)*rows+tj]=mat[threadIdx.x][threadIdx.y+k];
    }
}
```

## 5. Performance analysis

### 5.1 How to time the kernel execution of CUDA

Here we use CUDA's built-in related functions to get the time of the kernel execution accurately.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
//kernel execution
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float kernelTime;
cudaEventElapsedTime(&kernelTime, start, stop);
```

Two variables **start, stop**, after we call **cudaEventRecord(start, 0)**, the kernel execution starts, when it finishes, we call **cudaEventRecord(stop, 0)**. And here we should also call **cudaEventSynchronize(stop)** to make our timing accurate. Finally, we can compute the difference of the two events and store the result by using **cudaEventElapsedTime**(&kernelTime, start, stop).

## 5.2 The results

After our testing, we can get the performance result as below.
4096 * 4096 matrix, 100 iterations.
Our own computer:

| | Time(ms) | Bandwidth(GB/s) | step speedup | speed up vs CPU |
|---|---|---|---|---|
| CPU | 147 | 0.85 | | |
| Naïve GPU | 6.86 | 18.23 | 21.44 | 21.44 |
| Shared memmory | 5.51 | 22.67 | 1.24 | 26.67 |
| No bank conflict | 4.53 | 27.58 | 1.22 | 32.45 |
| Loop unrolled | 4.48 | 27.91 | 1.01 | 32.84 |

However, it seems that the Loop unrolled does not have a large improvement over the No bank conflict. After checking the code again, we think there is no bug in the code, so we test our program on Pi cluster(k80), and we get the below table.

| | Time(ms) | Bandwidth(GB/s) | step speedup | speed up vs CPU |
|---|---|---|---|---|
| CPU | 225.14 | 0.55 | | |
| Naïve GPU | 2.89 | 43.23 | 78.6 | 78.6 |
| Shared memmory | 2.06 | 60.58 | 1.4 | 110.15 |
| No bank conflict | 1.36 | 91.6 | 1.51 | 166.5 |
| Loop unrolled | 1.05 | 119.27 | 1.32 | 216.85 |

Actually it has the improvement. And for this problem, we think it is because in our own computer, the bandwidth is very close to the max bandwidth of GPU, so the improvement is not that obvious.

From the table we get, we can have some conclusions:

1. When we change the CPU version into the GPU version, it has a huge improvement over ten times.
2. The GPU rate takes an important role, we can see the execution time over Pi cluster is much faster than that over our own computer. If the GPU rate is too small, we can not get the large improvement.
3. For the 4 GPU versions tested on Pi cluster, every time we can almost get 1.4 times improvement. Loop unrolled version is the best one.

## 5.3 Analysis(further thinking)

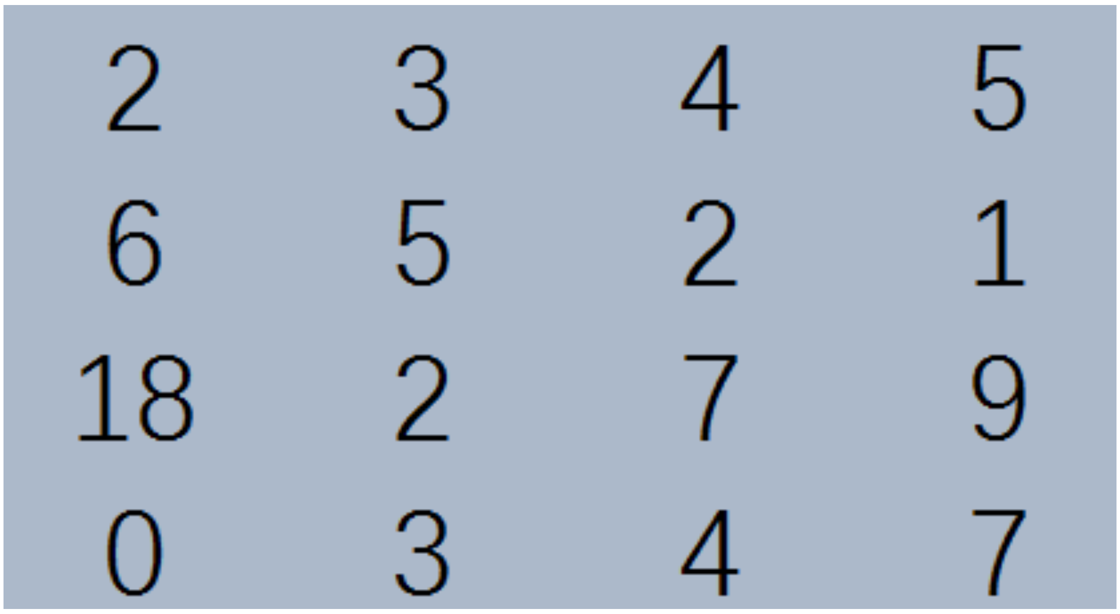
How could shared memory method give us the boost?
The first idea comes up in our mind is the high accessing-speed of shared memory for the threads in the same block compared with accessing global memory.
Is that true? In fact it's not so simple. Let's see the details in the Naive GPU method. When we want to transpose one matrix, we need first read every element once, and write them into the target matrix. In the Naive GPU method, both source matrix and target matrix(transposed matrix) are stored in GPU. That means we need to access global memory twice for each element. And for Shared Memory method, every thread load one element into the shared memory in blocks, and we write these element into the target matrix. In this situation, both source matrix and target matrix are stored in global memory. We still need access every element twice, what's more, the overload of copying element into the shared memory. Have you find that we do not decrease the cost to access element? (Even more in fact)

After finishing the whole project and reviewing this part, we find the real reason is still bank conflict.

In the Naive GPU method, we set every block is 32 * 32 dimension. Let's see how this method works. The first 32 threads are started at first(one warp), they load the first row of the block(32 float elements) and store them into the first col of the target block. Here occurs bank conflicts that all the threads tried to write same bank.

Then think about how we transpose matrix in Shared Memory method. We copy the source matrix into the shared memory( the cost here in fact is same as the cost in Naive GPU Method, just accessing the source matrix more concentrated). What's different is that for the first warp threads, they fetch the elements that should be stored in the first row of target matrix from the shared matrix, and store them into the target matrix in global memory. As we access the target matrix in row order, we eliminate the bank conflict of accessing the target matrix.

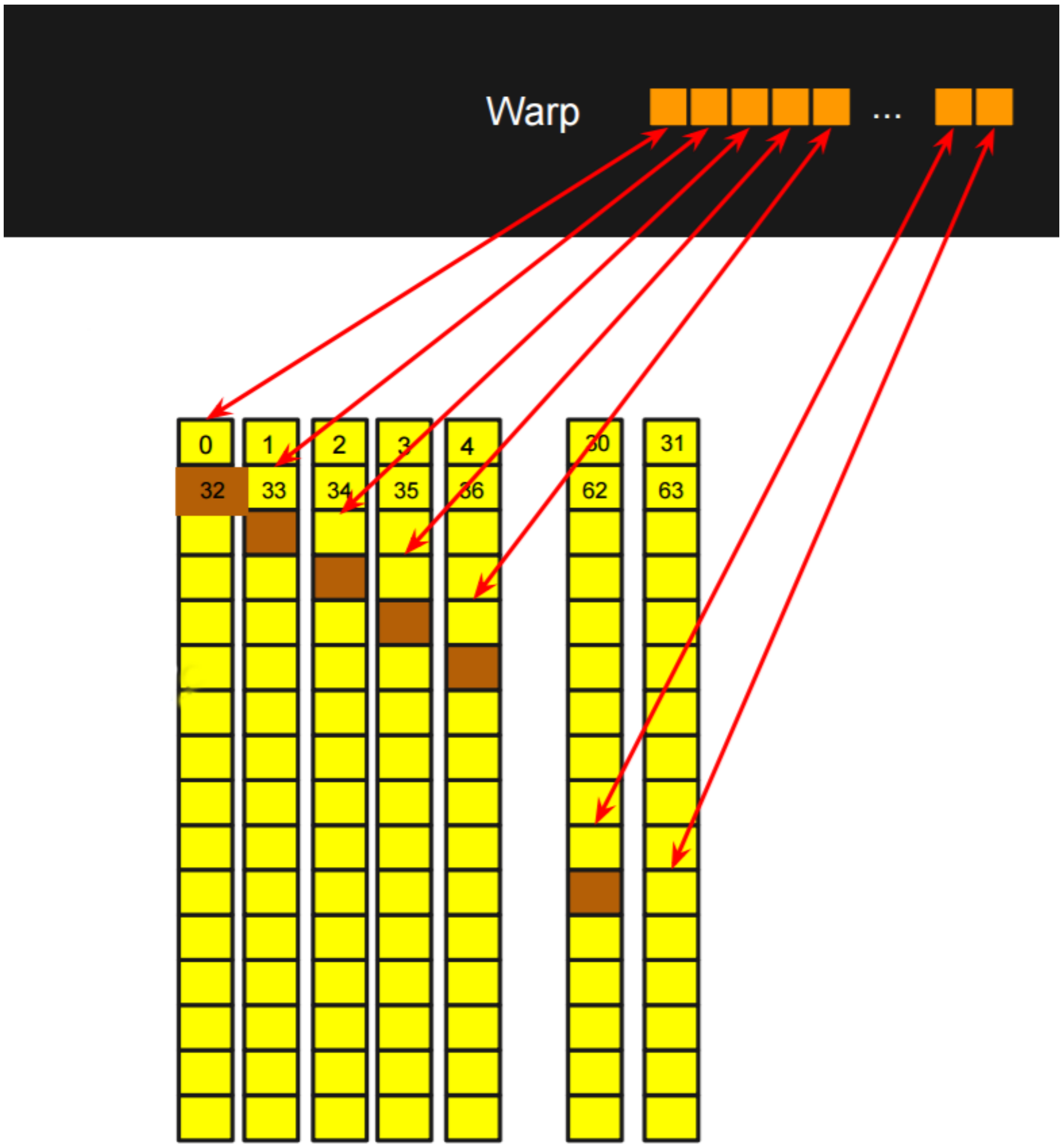
However in fact we add one new bank conflicts to the algorithm when threads try to fetch the first col of shared memory matrix to store them into the first row of target matrix. So we just transfer then bank conflicts from the procedure of accessing the target matrix to the procedure of accessing shared memory matrix. And because of the fast accessing-speed of shared memory, we can decrease the cost of bank conflicts.



| | | | |
|---|---|---|---|
| 2 | 3 | 4 | 5 |
| 6 | 5 | 2 | 1 |
| 18 | 2 | 7 | 9 |
| 0 | 3 | 4 | 7 |

How could No bank conflict method can give us the boost?
According to what we discussed before, we have know the overload of bank conflicts. Can we eliminate bank conflicts totally?  Yes, we can simply add one

more col into the shared memory matrix, which makes the accessing of same rows happens to different banks.



## 6. Summary

It is a good beginning for us to learn CUDA programming. We learn a lot about hardware-level optimization, now we have a deeper understanding about the computer architecture.

From this project, we give our first attempt to use Pi cluster. Thanks to the detailed instruction documentation, we finally get our programs compiled and run successfully on Pi cluster.
Thanks Prof. Deng for the guidance on CUDA programming and TAs for answering our questions.