Programming Assignment 3 - Matrix Transpose

Yesheng Ma
Ke Chang

December 22, 2016

Abstract

Deep learning is gaining popularity these days and matrix computation forms most of the computation in training a deep neural network. Among these matrix computation, matrix transpose is a typical one. In the project, we will use the CUDA toolkit to implement GPU-based efficient matrix transpose computation.

1 Introduction

In this project, we will implement 5 kinds of matrix transpose program: naive CPU, naive GPU, shared memory GPU, matrix transpose without bank conflict GPU, loop unrolled GPU. I will explain how I implement this 5 kinds of matrix transpose programs later.

In this project, the hardware we use is Nvidia GPU and these matrix transpose programs are compiled and executed on Pi cluster.

2 Hardware Information

To fetch the hardware information when we execute CUDA program in Pi supercomputer, the TA has already given us a CUDA script. What we need to do is compile the CUDA program to binary and submit to Pi. I queried two kinds of GPU on Pi: K40 and K80 and the GPU information is listed as follows:

```
Device 0: "Tesla K40c"
Total amount of global memory:                 11520 MBytes
(15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
GPU Max Clock rate:                            745 MHz (0.75 GHz)
Memory Clock rate:                             3004 Mhz
Memory Bus Width:                              384-bit
L2 Cache Size:                                 1572864 bytes
Maximum Texture Dimension Size (x,y,z)         (65536), (65536 * 2), (4096 * 3)
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
```

Since the information is a bit too long and I will not the the hardware information of Tesla K80. However, I looked into the difference of K40C and K80 and find that K80 has higher GPU clock rate but lower memory clock rate. Detailed result will be given later in the benchmark part.

3 Implementation of Matrix Transpose

In this part, we will discuss 5 different kinds of matrix transpose implementation, i.e. naive CPU, naive GPU, shared memory, shared memory without bank conflict and loop unrolling.

3.1 Naive CPU

The naive CPU version of matrix transpose is quite easy and is more like a hands-on practice to beginner programmer. The key idea is we use a nested loop to assign `a[i][j]` to `b[j][i]`. Since it is quite easy, I will just show how I implement it as follows:

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

Listing 1: naive CPU

3.2 Naive GPU

The naive GPU version is also not very difficult. We exploit the parallelism by simply translate the naive CPU version to GPU version. What we need to care about is that the function needs to be annotated by `__global__` so that `nvcc` can know that this function should be executed in GPU. Also, before call the matrix transpose function, we need to call cudaMemcpy to copy data in host memory to device memory.

```
__global__ void matrixTranspose(float *_a, float *_b, int cols,int rows)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y; // row
    int j = blockIdx.x * blockDim.x + threadIdx.x; // col

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

    _b[index_out] = _a[index_in];
}
```

Listing 2: Naive GPU

3.3 Shared Memory GPU

In the shared memory GPU implementation, we divide the whole matrix of size 1024×1024 to $D \times D$ smaller matrices where we make D to be 16. In the CUDA function, we declare a shared memory space `__shared__ float mat[P][P]`. Then we use CUDA primitive `_synctheads()` to wait for all threads in a particular block. Then the GPU begins to transpose the matrix in shared memory to destination submatrix.

```
__global__ void matrixTransposeShared(float *_a, float *_b, int cols,
     int rows)
{
    __shared__ float mat[P][P];
    int bx = blockIdx.x * blockDim.x;
    int by = blockIdx.y * blockDim.y;
    int i = by + threadIdx.y; int j = bx + threadIdx.x;
    int ti = bx + threadIdx.y; int tj = by + threadIdx.x;

    if (i < rows && j < cols)
        mat[threadIdx.x][threadIdx.y] = _a[i*cols + j];
    __syncthreads();
    if (tj < cols && ti < rows)
        _b[ti*rows+tj] = mat[threadIdx.y][threadIdx.x];
}
```

Listing 3: Shared Memory GPU

3.4 Shared Memory without Bank Conflicts GPU

Though eliminate bank conflict seems difficult at first thought, it can be accomplished by simply change the colon size by 1. Every other things will keep the same.

```
__global__ void matrixTransposeSharedwBC(float *_a, float *_b, int cols,
    int rows)
{
    __shared__ float mat[P][P+1];
    int bx = blockIdx.x * blockDim.x;
    int by = blockIdx.y * blockDim.y;
    int i = by + threadIdx.y; int j = bx + threadIdx.x;
    int ti = bx + threadIdx.y; int tj = by + threadIdx.x;

    if (i < rows && j < cols)
        mat[threadIdx.x][threadIdx.y] = _a[i*cols + j];
    __syncthreads();
    if (tj < cols && ti < rows)
        _b[ti*rows+tj] = mat[threadIdx.y][threadIdx.x];
}
```

Listing 4: Shared Memory without Bank Conflicts GPU

3.5 Loop Unrolling GPU

Loop unrolling is a common optimization at compile time, which can efficiently decrease the overhead of branch predication and make GPU more pipelined. CUDA provides convenient compile time macros unrolling loops. What we need to do is simply add the `#progma unroll` macro before each for loop.

```
__global__ void matrixTransposeUnrolled(float *_a, float *_b, int cols,
    int rows)
{
    __shared__ float mat[P][P+1];
    int x = blockIdx.x * P + threadIdx.x;
    int y = blockIdx.y * P + threadIdx.y;

    #pragma unroll
    for (int k = 0; k < P; k += 8) {
        if (x < rows && y+k < cols)
            mat[threadIdx.y+k][threadIdx.x] = _a[(y+k)*rows + x];
    }

    __syncthreads();

    x = blockIdx.y * P + threadIdx.x;
    y = blockIdx.x * P + threadIdx.y;
    #pragma unroll
    for (int k = 0; k < P; k += 8) {
        if (x < cols && y+k < rows)
            _b[(y+k)*cols + x] = mat[threadIdx.x][threadIdx.y+k];
    }
}
```

Listing 5: Loop Unrolling GPU

That's all for our discussion on implementation of different kinds of matrix transpose. Since this is only a introductory project on CUDA, I think it's more important for us to get a rough idea of GPU performance and let's go to discuss the performance of each implementation.

4 Performance & Analysis

In this section, we will discuss how we test these functions on Pi supercomputer, list the results we get, and analyze the result.

4.1 How to Get Running Time

To test the running time of each CUDA function, we use CUDA's built-in `Event` related functions, which can get the running time of a CUDA event accurately. The common work flow is as follows:

```
cudaEventRecord(tStart, 0);
// execute a CUDA event for several times
cudaEventRecord(tEnd, 0);
cudaEventSynchronize(tEnd);
cudaEventElapsedTime(&duration, tStart, tEnd);
```

Listing 6: Workflow of Time Measuring

In this way, we can accurately estimate the time of a single matrix transpose operation by repeating it for several times. Immediately after we call `cudaEventRecord(tStart, 0)`, we begin to execute CUDA kernel for say `N` times. When the execution is done, we call `cudaEventRecord(tEnd, 0)`. Note here that we need to call an extra `cudaEventSynchronized(tEnd)` since in CUDA events happen asynchronously and we need to call this function to make our timing accurate. Lastly, we calculate the difference of these two events can store the result to a float number by `cudaEventElapsedTime(&duration, tStart, tEnd)`.

4.2 Performance of Different Implementations

The following two tables shows the result of different implementations on different hardwares.

	Time(ms)	Bandwidth(GB/s)	Step Speedup	Speedup vs CPU
naive CPU	4.593	1.83	1.00	1.00
naive GPU	0.129	65.02	35.61	35.61
Shared Memory	0.094	89.24	1.37	48.86
Shared Mem wBC	0.067	125.20	1.40	68.56
Loop Unrolling	0.054	155.34	1.24	85.06

Table 1: Performance on K40

	Time(ms)	Bandwidth(GB/s)	Step Speedup	Speedup vs CPU
naive CPU	5.580	1.50	1.00	1.00
naive GPU	0.196	42.80	28.47	28.47
Shared Memory	0.143	58.66	1.37	39.02
Shared Mem wBC	0.094	89.24	1.52	59.36
Loop Unrolling	0.071	118.15	1.32	78.59

Table 2: Performance on K80

From the above tables, we can conclude that:

1. Though K80 has higher GPU rate, it executes slower than K40, I guess the reason may be K80 is busy at that time.
2. We gain a huge performance gain of about 30 times by moving from CPU to GPU.
3. From naive GPU to shared memory and from shared memory to shared memory without bank conflicts, we gain about 1.4 times performance by optimizing memory access.
4. We further optimize matrix transpose by loop unrolling, which takes less branch and increases parallelism.

5 Conclusion

It is really meaningful for us to learn CUDA programming in this era of deep learning since deep learning applications are matrix-computation-intensive. Most popular deep learning frameworks like Tensorflow and CNTK use CUDA as an interface of top-layer model to bottom-layer hardware.

From this project, we can get a concrete idea of how fast a GPU can be when it comes to computation-intensive jobs. The most difficult thing is actually getting our program compiled and making it run on Pi supercomputer. When these configurations are done, the CUDA programming is not that hard. And actually CUDA is a good abstraction of hardware and gives programmers much convenience.

Acknowledgement

Thanks Prof. Deng for guidance on CUDA programming and TAs for configuring cluster and kindly answering questions.