Matrix Reduction

1. naive

   vector size: 67108864

   time to execute kernel: 0.002588 secs

   Effective bandwidth: 103.72 GB/s

   `double bw = (N * sizeof(dtype)) / (t_kernel_0 * 1e9);`

   First, calculate how many bytes of data are being processed, then divide the number of bytes by time and convert it to the unit of GB/s.

2. stride

   Time to execute naive CPU reduction: 0.675595 secs

   Effective bandwidth: 164.79 GB/s

   speedup: 1.59x

3. sequential

   Effective bandwidth: 210.87 GB/s

4. first add then reduce

   Effective bandwidth: 402.45 GB/s

5. unroll

   Effective bandwidth: 589.97 GB/s

6. multiple

   Effective bandwidth: 645.28 GB/s

Transpose

naive:

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.x + threadIdx.y;
int width = N;

for (int j = 0; j < blockDim.x; j+= blockDim.y){
  if(y+j < N && x < N)
          AT[x*width + (y+j)] = A[(y+j)*width + x];

}
```

The process of the naive algorithm is as follows: each thread retrieves the corresponding element from global memory and then writes it back to the corresponding position in global memory. The threads read the elements in a contiguous manner from memory, but they write them back in a non-contiguous manner.

optimized:

```
    __shared__ dtype tile[64][64 + 1];

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.x + threadIdx.y;
    int width = N;

    for (int j = 0; j < blockDim.x; j += blockDim.y)
        if((y+j)*width + x < N * N)
            tile[threadIdx.x][threadIdx.y+ j] = A[(y+j)*width + x];

    __syncthreads();

    x = blockIdx.y * blockDim.x + threadIdx.x;  // transpose block offset
    y = blockIdx.x * blockDim.x + threadIdx.y;

    for (int j = 0; j < blockDim.x; j += blockDim.y){
        if((y+j) < N && x < N)
            AT[(y+j)*width + x] = tile[threadIdx.y + j][threadIdx.x];
    }
```

First I use shared memory to avoid the large strides through global memory. And I also pad the width in the declaration of the shared memory tile, making the tile 65 elements wide rather than 64 to avoid bank conflict. Otherwise, all the elements in the same column will be map to the same shared memory bank.

performance results:

N=16384

naive: GPU transpose: 0.063849 secs ==> 4.20422 billion elements/second

optimized: GPU transpose: 0.006769 secs ==> 39.6566 billion elements/second

Reference:

1. An Efficient Matrix Transpose in CUDA C/C++ | NVIDIA Technical Blog
2. CUDA矩阵转置优化 - 知乎 (zhihu.com)