### FINAL PROJECT MRI-Q

Chenhao Wang ID:861244527

Computation of a matrix Q， representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space. Because the algorithm for Q is embarrassingly data-parallel, it’s good to implement it on GPU. In my project, the algorithm first compute the magnitude-squared of φ at each point in the trajectory space(k-space), then computes the real and imaginary components of Q at any point in the image space.

For all the results following, I use tools/compare-output to check the accuracy and use two method to measure the running time : the first is pb\_SwitchToTimer(&timers, pb\_TimerID\_xxx) get the running time quickly and directly; the second is adding a library: LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/cuda-5.0/lib64, and using nvprof ./mri-q to specify the running time.

In the report, I use red mark to show the main difference of each solutions

#### Solution\_1. Global Memory #MRIQ00

The first solution is directly computing Q on GPU via global memory and tile in grid. There are two parts:

**the first is to precompute PhiMag:**

\_\_global\_\_ void

ComputePhiMag\_GPU(float\* phiR, float\* phiI, float\* phiMag, int numK) {

int indexK = blockIdx.x\*KERNEL\_PHI\_MAG\_THREADS\_PER\_BLOCK + threadIdx.x;

if (indexK < numK) {

float real = phiR[indexK];

float imag = phiI[indexK];

phiMag[indexK] = real\*real + imag\*imag;

}

}

void computePhiMag\_GPU(int numK, float\* phiR\_d, float\* phiI\_d, float\* phiMag\_d)

{

int phiMagBlocks = (numK-1) / KERNEL\_PHI\_MAG\_THREADS\_PER\_BLOCK+1;

dim3 DimPhiMagBlock(KERNEL\_PHI\_MAG\_THREADS\_PER\_BLOCK, 1);

dim3 DimPhiMagGrid(phiMagBlocks, 1);

ComputePhiMag\_GPU <<< DimPhiMagGrid, DimPhiMagBlock >>>

(phiR\_d, phiI\_d, phiMag\_d, numK);

}

**The second is to compute Q:**

\_\_global\_\_ void ComputeQ\_GPU(int numK, int kGlobalIndex, float\* x, float\* y, float\* z, float\* Qr, float\* Qi, kValues\* ck)

{

int xIndex = blockIdx.x\*KERNEL\_Q\_THREADS\_PER\_BLOCK + threadIdx.x;

int kIndex = 0;

for (kIndex=0; kIndex < KERNEL\_Q\_K\_ELEMS\_PER\_GRID; kIndex++)

{ float expArg = PIx2 \* (ck[kIndex].Kx \* x[xIndex] +

ck[kIndex].Ky \* y[xIndex] +

ck[kIndex].Kz \* z[xIndex]);

Qr[xIndex] += ck[kIndex].PhiMag \* cos(expArg);

Qi[xIndex] += ck[kIndex].PhiMag \* sin(expArg);

}

}

void computeQ\_GPU(int numK, int numX,

float\* x\_d, float\* y\_d, float\* z\_d,

kValues\* kVals,

float\* Qr\_d, float\* Qi\_d)

{

int QGrids = (numK-1) / KERNEL\_Q\_K\_ELEMS\_PER\_GRID+1;

int QBlocks = (numX-1) / KERNEL\_Q\_THREADS\_PER\_BLOCK+1;

dim3 DimQBlock(KERNEL\_Q\_THREADS\_PER\_BLOCK, 1);

dim3 DimQGrid(QBlocks, 1);

for (int QGrid = 0; QGrid < QGrids; QGrid++) {

int QGridBase = QGrid \* KERNEL\_Q\_K\_ELEMS\_PER\_GRID;

kValues\* kValsTile = kVals + QGridBase;

ComputeQ\_GPU <<< DimQGrid, DimQBlock >>>

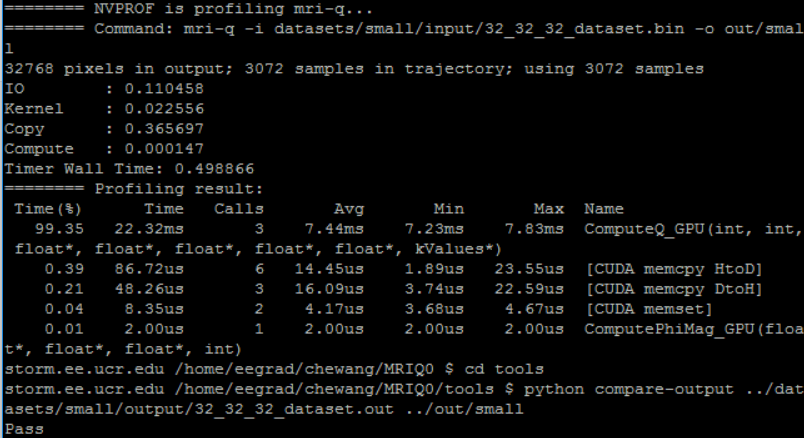
(numK, QGridBase, x\_d, y\_d, z\_d, Qr\_d, Qi\_d, kValsTile);

}

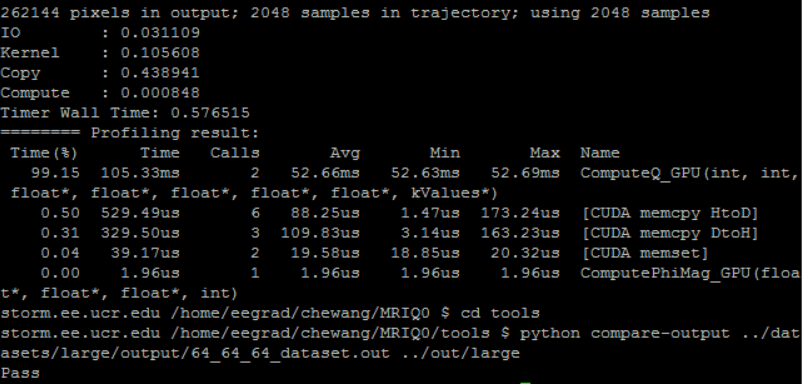
}

The result is as follows:

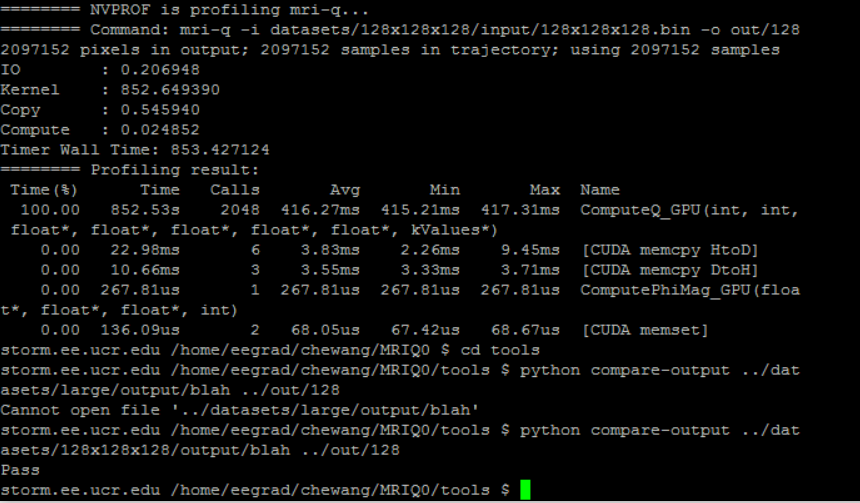
32x32x32



64x64x64



128x128x128



We can see the running time is long and it still has some space to improve.

#### Solution\_2 Global Memory+ Shared Memory #MRIQ01

I observe that in Solution\_1, when computing Q, this loop may take some extra time

for (kIndex=0; kIndex < KERNEL\_Q\_K\_ELEMS\_PER\_GRID; kIndex++)

{ float expArg = PIx2 \* (ck[kIndex].Kx \* x[xIndex] +

ck[kIndex].Ky \* y[xIndex] +

ck[kIndex].Kz \* z[xIndex]);

Qr[xIndex] += ck[kIndex].PhiMag \* cos(expArg);

Qi[xIndex] += ck[kIndex].PhiMag \* sin(expArg);

}

since in every iteration, x[xIndex], y[xIndex], z[xIndex], Qr[xIndex], Qi[xIndex] will access global memory once and once again. So I think I may be better when put these on shared memory or register. I use shared memory:

\_\_shared\_\_ float sx,sy,sz,sQr,sQi;

int xIndex = blockIdx.x\*KERNEL\_Q\_THREADS\_PER\_BLOCK + threadIdx.x;

sx = x[xIndex];

sy = y[xIndex];

sz = z[xIndex];

sQr = Qr[xIndex];

sQi = Qi[xIndex];

and then the loop becomes:

for (int kIndex=0 ; (kIndex < KERNEL\_Q\_K\_ELEMS\_PER\_GRID) && (kGlobalIndex < numK);

kIndex ++, kGlobalIndex ++) {

float expArg = PIx2 \* (ck[kIndex].Kx \* sx +

ck[kIndex].Ky \* sy +

ck[kIndex].Kz \* sz);

sQr += ck[kIndex].PhiMag \* cos(expArg);

sQi += ck[kIndex].PhiMag \* sin(expArg);

}

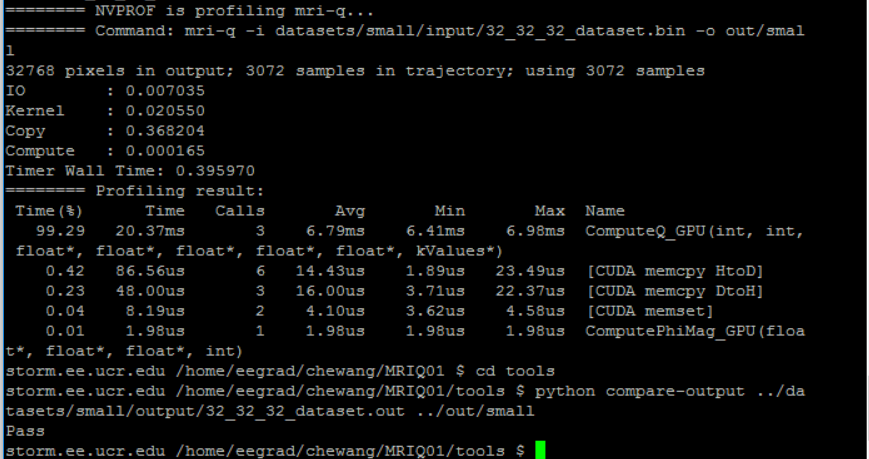
Qr[xIndex] = sQr;

Qi[xIndex] = sQi;

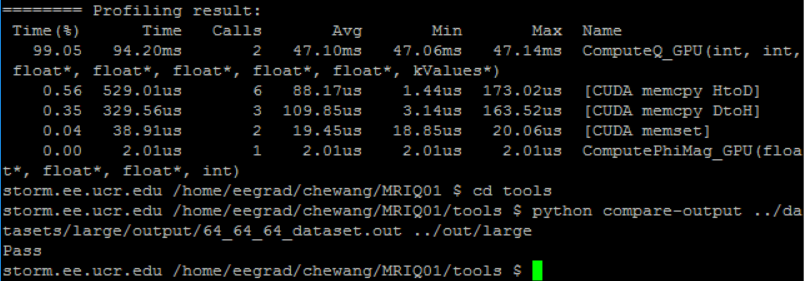
}

And get the results as follows:

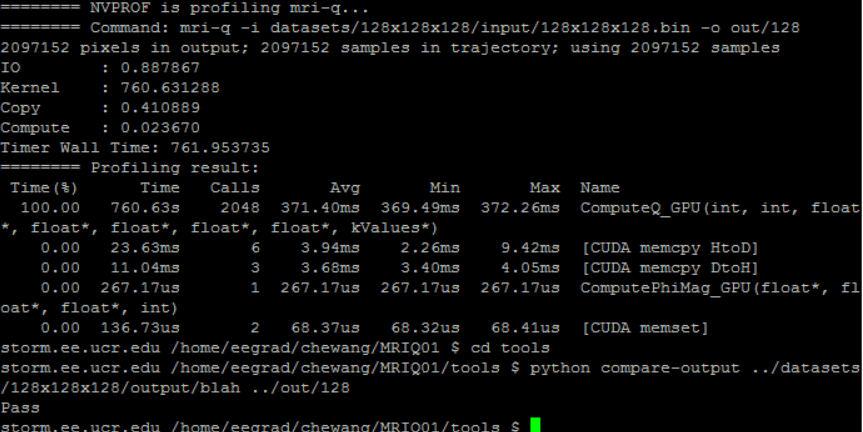
32x32x32



64x64x64



128x128x128



As I expected, it does speed up especially when the input data is large(i.e.128x128x128 saves 92s). After I checked to put in register, the saving time is nearly the same so I don’t append the details.

#### Solution3\_ Constant Memory +Shared Memory #MRIQ02

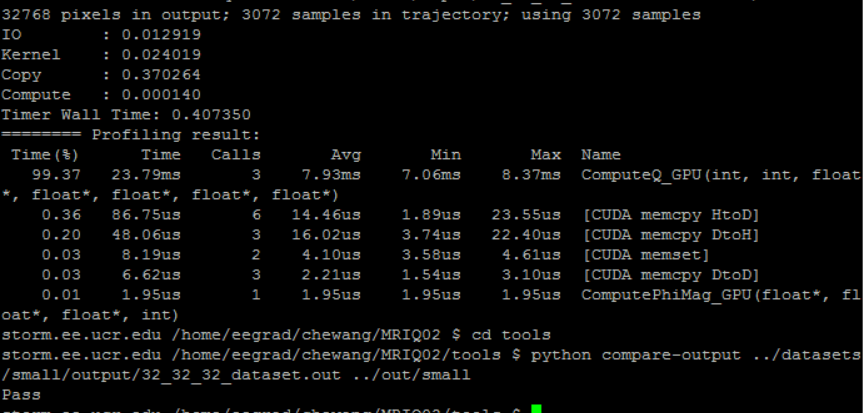
Next, also in the previous loop, I find ck[kIndex].Kx never changes in every iteration and keep the same in every grid iteration. From the reference article “Accelerating advanced MRI reconstructions on GPUs” , I learned that dividing the computation into many CUDA grids can overcome memory bottleneck. for each CUDA grid, the host CPU loads the data tile into constant memory before invoking the kernel. Each thread in the CUDA grid then computes a partial sum for a single element of Q by iterating over all the points in the data tile. This optimization increases the ratio of FP operations to global memory accesses dramatically.

So I use constant memory and load ck into constant memory

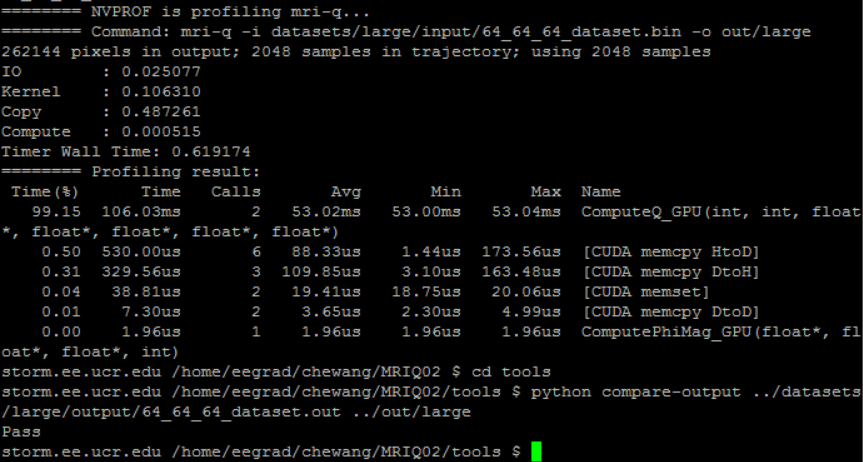
\_\_constant\_\_ \_\_device\_\_ kValues ck[KERNEL\_Q\_K\_ELEMS\_PER\_GRID];

The results are as follows:

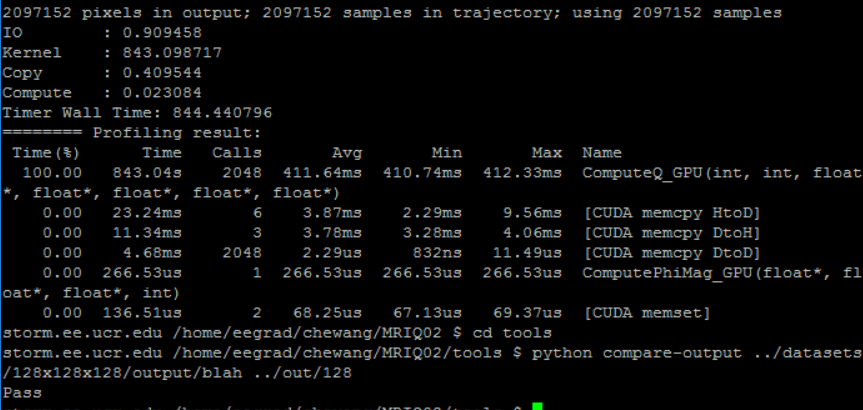
32x32x32



64x64x64



128x128x128



Disappointedly, it’s even slow than directly computing on global memory. I don’t know whether the ratio of FP operations to global memory accesses is improved or not, but the speed is too slow so I need make it faster!

#### Solution 4\_ Constant Memory + Shared Memory+ One iteration compute 2 times #MRIQ03

I find each iteration it only computes one Q, if each computes two Q, the iterations will be halved and may be faster. So I did a little trick:

First to judge numK is even or odd, fi it’s odd I initialize the first via ck[0].

int kIndex = 0;

if (numK % 2) {

float expArg = PIx2 \* (ck[0].Kx \* sx + ck[0].Ky \* sy + ck[0].Kz \* sz);

sQr += ck[0].PhiMag \* cos(expArg);

sQi += ck[0].PhiMag \* sin(expArg);

kIndex++;

kGlobalIndex++;

}

Then the rest is even and it can compute 2Q in every iteration:

for (; (kIndex < KERNEL\_Q\_K\_ELEMS\_PER\_GRID) && (kGlobalIndex < numK);

kIndex += 2, kGlobalIndex += 2) {

float expArg = PIx2 \* (ck[kIndex].Kx \* sx +

ck[kIndex].Ky \* sy +

ck[kIndex].Kz \* sz);

sQr += ck[kIndex].PhiMag \* cos(expArg);

sQi += ck[kIndex].PhiMag \* sin(expArg);

int kIndex1 = kIndex + 1;

float expArg1 = PIx2 \* (ck[kIndex1].Kx \* sx +

ck[kIndex1].Ky \* sy +

ck[kIndex1].Kz \* sz);

sQr += ck[kIndex1].PhiMag \* cos(expArg1);

sQi += ck[kIndex1].PhiMag \* sin(expArg1);

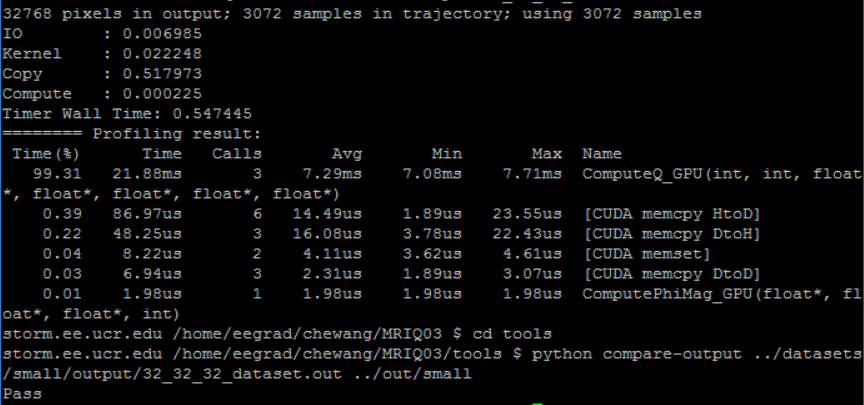
}

Qr[xIndex] = sQr;

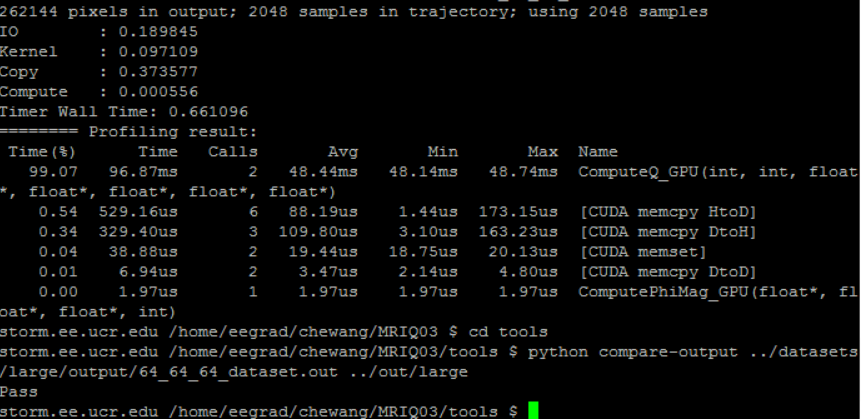
Qi[xIndex] = sQi;

}

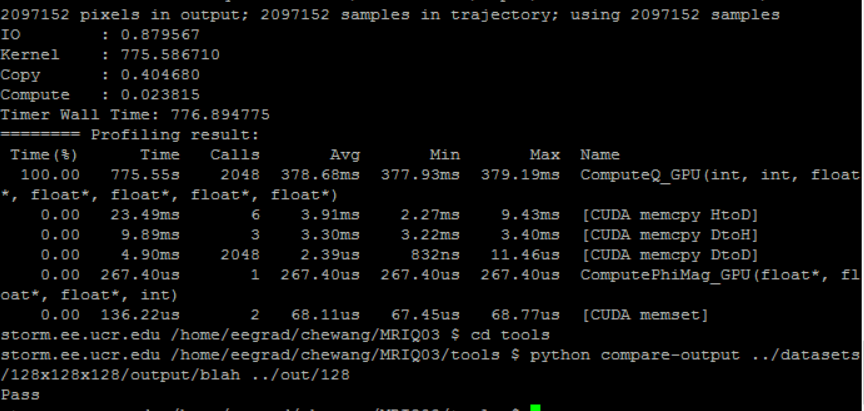
32x32x32



64x64x64



128x128x128



It really speeds up the computation but is still a little far away from my expectation.

#### Solution 5\_Constant Memory+Shared Memory +One iteration compute 4 times #MRIQ04

One iteration computing 2Q has effect, and how about computing 4Q once? So I make a little change:

First initialize: if numK%4, it will be 1,2,3, so use a small loop to handle this, then the rest is numK%4==0.

int kIndex = 0;

if (numK % 4)

{

for (int j=0;j<numK%4;j++)

{

float expArg = PIx2 \* (ck[j].Kx \* sx + ck[j].Ky \* sy + ck[j].Kz \* sz);

sQr += ck[j].PhiMag \* cos(expArg);

sQi += ck[j].PhiMag \* sin(expArg);

kIndex++;

kGlobalIndex++;

}

}

for (; (kIndex < KERNEL\_Q\_K\_ELEMS\_PER\_GRID) && (kGlobalIndex < numK);

kIndex += 4, kGlobalIndex += 4) {

float expArg = PIx2 \* (ck[kIndex].Kx \* sx +

ck[kIndex].Ky \* sy +

ck[kIndex].Kz \* sz);

sQr += ck[kIndex].PhiMag \* cos(expArg);

sQi += ck[kIndex].PhiMag \* sin(expArg);

int kIndex1 = kIndex + 1;

float expArg1 = PIx2 \* (ck[kIndex1].Kx \* sx +

ck[kIndex1].Ky \* sy +

ck[kIndex1].Kz \* sz);

sQr += ck[kIndex1].PhiMag \* cos(expArg1);

sQi += ck[kIndex1].PhiMag \* sin(expArg1);

int kIndex2 =kIndex+ 2;

float expArg2 = PIx2 \* (ck[kIndex2].Kx \* sx +

ck[kIndex2].Ky \* sy +

ck[kIndex2].Kz \* sz);

sQr += ck[kIndex2].PhiMag \* cos(expArg2);

sQi += ck[kIndex2].PhiMag \* sin(expArg2);

int kIndex3 =kIndex+ 3;

float expArg3 = PIx2 \* (ck[kIndex3].Kx \* sx +

ck[kIndex3].Ky \* sy +

ck[kIndex3].Kz \* sz);

sQr += ck[kIndex3].PhiMag \* cos(expArg3);

sQi += ck[kIndex3].PhiMag \* sin(expArg3);

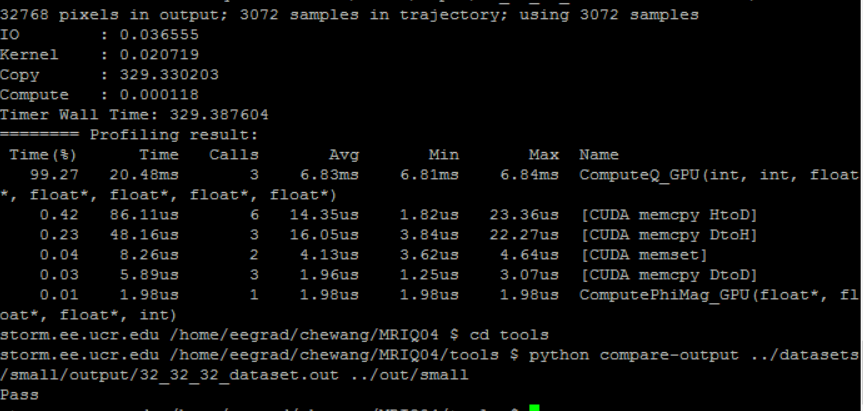
}

Qr[xIndex] = sQr;

Qi[xIndex] = sQi;

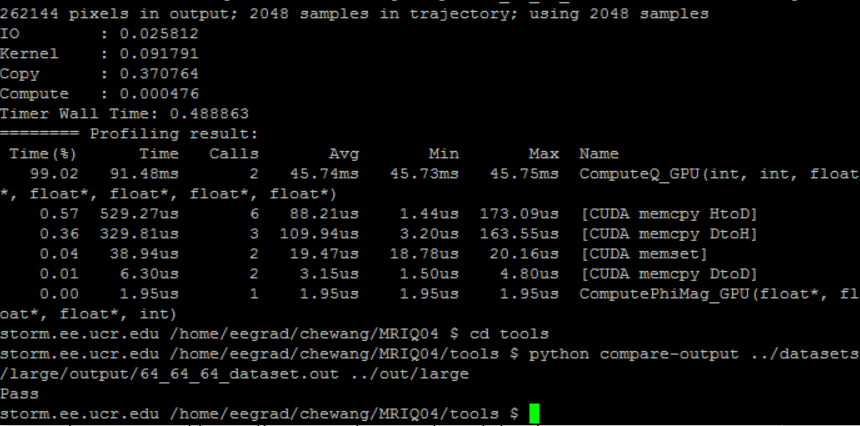
}

32x32x32

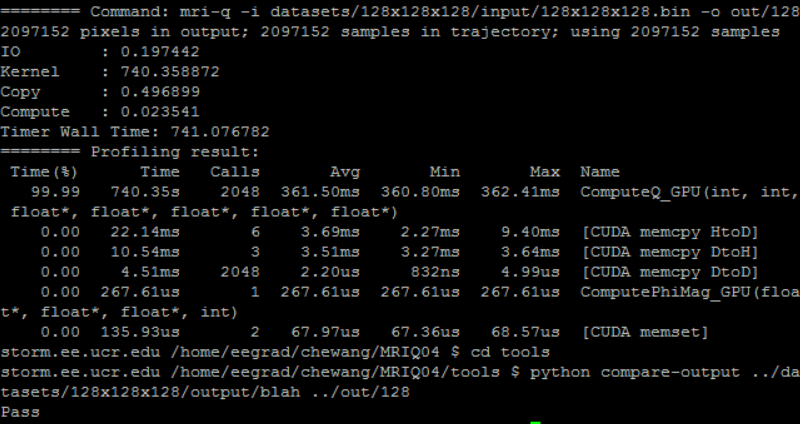


The storm may have some problem so the copying time is extremely long!

64x64x64



128x128x128



It’s 35s shorter than solution 4 in 128x128x128, so it’s a right direction to improve the algorithm. How about 8 times?

I don’t have enough time to check the further algorithm, but I think it will get a bottleneck when increase the times in each iteration because the judgement of whether numK % n and initialization has a small loop and each iteration’s time is also increased.

In my project, Solution 5 is the fastest, however when in the same optimization on global memory, it will get even faster. If only consider the speed, I find constant memory doesn’t have enough advantages than global memory. If consider the potential bottleneck posed by memory bandwidth and latency or ratio of FP operations to global memory access, constant memory is necessary. I think there are still a lot of things to do for accelerating the algorithm, I will continue to do this research in the future.