Practice 1

Name: 魏程涌潇

Student ID: 515030910598

Practical 1: Getting Started

Compare *prac1a.cu* with *prac1b.cu*

The two programs aims to do the same thing: store the tid (`threadIdx.x + blockDim.x*blockIdx.x`) in `d_x` and copy `d_x` to `h_x` and finally print all the tid. However, there are some error-checking codes in *prac1b.cu* compared to *prac1a.cu* and the program *prac1b.cu* will print the device information:

- *prac1b.cu* adds `findCudaDevice(argc, argv);` to output the device information.
- *prac1b.cu* adds `checkCudaErrors` at `cudaMalloc` , `cudaMemcpy` and `cudaFree` for error-checking sake. `checkCudaErrors` is a macro defined in header `helper_cuda.h` .

The two outputs are shown below:

prac1a.cu

```
n,  x  =  0  0.000000
n,  x  =  1  1.000000
n,  x  =  2  2.000000
n,  x  =  3  3.000000
n,  x  =  4  4.000000
n,  x  =  5  5.000000
n,  x  =  6  6.000000
n,  x  =  7  7.000000
n,  x  =  8  0.000000
n,  x  =  9  1.000000
n,  x  =  10  2.000000
n,  x  =  11  3.000000
n,  x  =  12  4.000000
n,  x  =  13  5.000000
n,  x  =  14  6.000000
n,  x  =  15  7.000000
```

prac1b.cu

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

 n,  x  =  0  0.000000
 n,  x  =  1  1.000000
```

```
 n,  x  =  2  2.000000
 n,  x  =  3  3.000000
 n,  x  =  4  4.000000
 n,  x  =  5  5.000000
 n,  x  =  6  6.000000
 n,  x  =  7  7.000000
 n,  x  =  8  0.000000
 n,  x  =  9  1.000000
 n,  x  =  10  2.000000
 n,  x  =  11  3.000000
 n,  x  =  12  4.000000
 n,  x  =  13  5.000000
 n,  x  =  14  6.000000
 n,  x  =  15  7.000000
```

Try introducing errors

- Setting `nblocks=0`
  - For *prac1a.cu*, there is no output and no error information.
  - For *prac1b.cu*, there's only an output printing the device information and an error information `prac1b.cu(52) : getLastCudaError() CUDA error : my_first_kernel execution failed : (9) invalid configuration argument.`

From my point of view, *prac1a.cu* admits the parameter with `nblocks=0`, and then the program actually does nothing. Nevertheless, *prac1b.cu* with error-checking codes, is able to check the error and output an error message.

Add a printf statement to kernel_routine

Add a statement to kernel_routine, and `my_first_kernel` becomes:

```
__global__ void my_first_kernel(float *x)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;

  x[tid] = (float) threadIdx.x;
  printf("tid = %d\n", tid);
}
```

Output:

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

tid = 0
tid = 1
tid = 2
tid = 3
tid = 4
```

```
tid = 5
tid = 6
tid = 7
tid = 8
tid = 9
tid = 10
tid = 11
tid = 12
tid = 13
tid = 14
tid = 15
 n,  x  =  0  0.000000
 n,  x  =  1  1.000000
 n,  x  =  2  2.000000
 n,  x  =  3  3.000000
 n,  x  =  4  4.000000
 n,  x  =  5  5.000000
 n,  x  =  6  6.000000
 n,  x  =  7  7.000000
 n,  x  =  8  0.000000
 n,  x  =  9  1.000000
 n,  x  =  10  2.000000
 n,  x  =  11  3.000000
 n,  x  =  12  4.000000
 n,  x  =  13  5.000000
 n,  x  =  14  6.000000
 n,  x  =  15  7.000000
```

Note that, `my_first_kernel` are called just like a function that the new output is written to screen between device information and the main results.

Modify *prac1b.cu*

Add two pairs of host and device values:

```
int *h_ix, *d_ix, *h_iy, *d_iy;
```

Allocate memory:

```
h_ix = (int *)malloc(nsize*sizeof(int));
h_iy = (int *)malloc(nsize*sizeof(int));
checkCudaErrors(cudaMalloc((void **)&d_ix, nsize*sizeof(int)));
checkCudaErrors(cudaMalloc((void **)&d_iy, nsize*sizeof(int)));
```

Initialise:

```
for (int i = 0; i < nsize; ++i) {
  h_ix[i] = i;
}

for (int i = 0; i < nsize; ++i) {
  h_iy[i] = i + 1;
}
```

Copy from host to device:

```
checkCudaErrors(cudaMemcpy(d_ix, h_ix, nsize*sizeof(int),
                cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_iy, h_iy, nsize*sizeof(int),
                cudaMemcpyHostToDevice));
```

Modify `my_first_kernel`:

```
__global__ void my_first_kernel(float *x, int *ix, int *iy)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;

  x[tid] = (float) threadIdx.x;
  printf("tid = tid, ix = %d, iy = %d\n", tid, ix[tid], iy[tid]);
}

int main(int argc, const char **argv)
{
    ...;
    my_first_kernel<<<nblocks,nthreads>>>(d_x, d_ix, d_iy);
    ...;
}
```

Output:

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

tid = 0, ix = 0, iy = 1
tid = 1, ix = 1, iy = 2
tid = 2, ix = 2, iy = 3
tid = 3, ix = 3, iy = 4
tid = 4, ix = 4, iy = 5
tid = 5, ix = 5, iy = 6
tid = 6, ix = 6, iy = 7
tid = 7, ix = 7, iy = 8
tid = 8, ix = 8, iy = 9
tid = 9, ix = 9, iy = 10
tid = 10, ix = 10, iy = 11
```

```
tid = 11, ix = 11, iy = 12
tid = 12, ix = 12, iy = 13
tid = 13, ix = 13, iy = 14
tid = 14, ix = 14, iy = 15
tid = 15, ix = 15, iy = 16
 n,  x  =  0  0.000000
 n,  x  =  1  1.000000
 n,  x  =  2  2.000000
 n,  x  =  3  3.000000
 n,  x  =  4  4.000000
 n,  x  =  5  5.000000
 n,  x  =  6  6.000000
 n,  x  =  7  7.000000
 n,  x  =  8  0.000000
 n,  x  =  9  1.000000
 n,  x  =  10  2.000000
 n,  x  =  11  3.000000
 n,  x  =  12  4.000000
 n,  x  =  13  5.000000
 n,  x  =  14  6.000000
 n,  x  =  15  7.000000
```

Unified Memory

Instead of claiming two values, one host value and one device value, program *prac1c.cu* claims just only one value `x`. This is one benefit of using Unified Memory. This is a feature introduced in CUDA 6.0 and it makes writting CUDA program easier. However, there're three basic requirements for Unified Memory:

1. **A GPU with SM architecture 3.0 or higher (Kepler class or newer)**
2. **A 64-bit host application and operating system, except on Android**
3. **Linux or Windows**

Besides, the program also calls `cudaDeviceSynchronize`, and halts execution in the CPU/host thread (that the cudaDeviceSynchronize was issued in) until the GPU has finished processing all previously requested cuda tasks (kernels, data copies, etc.)

Here the cuda tasks are kernels and data copies.

Practical 2: Monte Carlo

Output

Version 1

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

CURAND normal RNG  execution time (ms): 39.585152,  samples/sec:
4.850304e+09
Monte Carlo kernel execution time (ms): 5.505728

Average value and standard deviation of error  =    0.41732911
0.00048176
```

Version 2

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

CURAND normal RNG  execution time (ms): 40.110752,  samples/sec:
4.786746e+09
Monte Carlo kernel execution time (ms): 45.214306

Average value and standard deviation of error  =    0.41732812
0.00048176
```

It's obvious that Version 2 spends 10 times the time longer than Version 1 on Monte Carlo kernel execution.

1 block of 32 threads and 1 timestep

Version 1

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

CURAND normal RNG  execution time (ms): 21.067743,  samples/sec:
9.113458e+07
y1 = d_z[0] = 0.780974
y1 = d_z[1] = -1.801578
y1 = d_z[2] = -0.280849
y1 = d_z[3] = 0.563547
y1 = d_z[4] = -1.011333
y1 = d_z[5] = 0.414039
y1 = d_z[6] = 1.458326
y1 = d_z[7] = -0.562136
......
y1 = d_z[23] = 1.221632
y1 = d_z[24] = 0.457925
y1 = d_z[25] = -0.751256
y1 = d_z[26] = -0.774908
y1 = d_z[27] = 1.388553
y1 = d_z[28] = 0.624958
y1 = d_z[29] = -0.730621
y1 = d_z[30] = 2.029238
```

```
y1 = d_z[31] = 0.320050
y2 = rho*y1*d_z[32], d_z[32] =  -0.571309
y2 = rho*y1*d_z[33], d_z[33] =  1.160371
y2 = rho*y1*d_z[34], d_z[34] =  -0.468651
y2 = rho*y1*d_z[35], d_z[35] =  1.303609
......
y2 = rho*y1*d_z[56], d_z[56] =  2.232951
y2 = rho*y1*d_z[57], d_z[57] =  0.733853
y2 = rho*y1*d_z[58], d_z[58] =  -0.839378
y2 = rho*y1*d_z[59], d_z[59] =  -0.832520
y2 = rho*y1*d_z[60], d_z[60] =  0.021353
y2 = rho*y1*d_z[61], d_z[61] =  -0.876494
y2 = rho*y1*d_z[62], d_z[62] =  0.366337
y2 = rho*y1*d_z[63], d_z[63] =  -0.011825
Monte Carlo kernel execution time (ms): 4.437408

Average value and standard deviation of error  =    0.00001585
0.00000396
```

Version 2

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

CURAND normal RNG  execution time (ms): 20.824160,  samples/sec:
9.220060e+07
read: d_z[0] = 0.780974
read: d_z[2] = -0.280849
read: d_z[4] = -1.011333
read: d_z[6] = 1.458326
read: d_z[8] = -0.148576
read: d_z[10] = -0.004148
read: d_z[12] = 1.450197
read: d_z[14] = 0.489706
......
read: d_z[52] = -0.497151
read: d_z[54] = 1.124668
read: d_z[56] = 2.232951
read: d_z[58] = -0.839378
read: d_z[60] = 0.021353
read: d_z[62] = 0.366337
y2 = rho * t1 + alpha * d_z[1], d_z[1] = -1.801578
y2 = rho * t1 + alpha * d_z[3], d_z[3] = 0.563547
y2 = rho * t1 + alpha * d_z[5], d_z[5] = 0.414039
y2 = rho * t1 + alpha * d_z[7], d_z[7] = -0.562136
y2 = rho * t1 + alpha * d_z[9], d_z[9] = -0.395185
y2 = rho * t1 + alpha * d_z[11], d_z[11] = -0.529958
y2 = rho * t1 + alpha * d_z[13], d_z[13] = 0.404758
......
y2 = rho * t1 + alpha * d_z[51], d_z[51] = -0.377947
```

```
y2 = rho * t1 + alpha * d_z[53], d_z[53] = -0.186477
y2 = rho * t1 + alpha * d_z[55], d_z[55] = 0.749611
y2 = rho * t1 + alpha * d_z[57], d_z[57] = 0.733853
y2 = rho * t1 + alpha * d_z[59], d_z[59] = -0.832520
y2 = rho * t1 + alpha * d_z[61], d_z[61] = -0.876494
y2 = rho * t1 + alpha * d_z[63], d_z[63] = -0.011825
Monte Carlo kernel execution time (ms): 4.393216

Average value and standard deviation of error  =    0.00001486
0.00000384
```

From the results, we can see that in Version 1 the 32 threads read in a contiguous block of 32 numbers at the same time, but they don't in Version 2. Since in Version 1, `ind` is determined by `threadIdx.x` while `ind` is determined by `2 * N * threadIdx.x` in Version 2, in Version 1 the 32 threads read in a contiguous block of 32 numbers at the same time but they don't in Version 2.

Determine the effective transfer rate

From the above outputs, we can see that the code are running in device "Tesla K20m". NVIDIA Tesla K20M's memory bandwidth is 208GB/s. The Monte Carlo kernel execution time is 4.393216 ms. The effective data is sizeof(float)*2*h_N*NPATH) = 32 * 2 * 100 * 960000 / 8 / 2^30 GB = .7152557 GB. Thus the effective data transfer rate is .7152557 / 5.505728 * 10^3 GB/s = 129.911194305 < 208 GB/s. It turns out that the effective data transfer rate is much smaller than the peak capability of the hardware.

Compute average value of $az^2 + bz + c$

Changed the kernel function:

```
__global__ void average(float *d_z, float *d_v)
{

  int ind = threadIdx.x + blockIdx.x*blockDim.x;
  float z = d_z[ind];
  d_v[ind] = a * z * z + b * z + c;

}
```

The result are shown below:

```
GPU Device 0: "Tesla K20m" with compute capability 3.5

CURAND normal RNG  execution time (ms): 20.252096,  samples/sec: 6.320333e+03
average value execution time (ms): 0.232000
a = 2, b = 3, c = 8
average value =   10.10309846
```

The average value is close to (a + c).

Practical 3: Finite difference equations

Run the code

laplace output:

```
Grid dimensions: 256 x 256 x 256
GPU Device 0: "Tesla K20m" with compute capability 3.5


Copy u1 to device: 20.1 (ms)

10x GPU_laplace3d_naive: 18.2 (ms)

Copy u2 to host: 32.1 (ms)

10x Gold_laplace3d: 2328.7 (ms)

rms error = 0.000000
```

laplace_new output:

```
Grid dimensions: 256 x 256 x 256
GPU Device 0: "Tesla K20m" with compute capability 3.5


Copy u1 to device: 20.0 (ms)

10x GPU_laplace3d_naive: 14.2 (ms)

Copy u2 to host: 32.2 (ms)

10x Gold_laplace3d: 2459.1 (ms)

rms error = 0.000000
```

Changing the thread block size

laplace:

With the help of **Occupancy calculator**, I have tried many times and finally got a 100% occupancy with:

```
#define BLOCK_X 16
#define BLOCK_Y 8
```

This time, the output is:

```
Grid dimensions: 256 x 256 x 256
GPU Device 0: "Tesla K20m" with compute capability 3.5


Copy u1 to device: 20.0 (ms)

10x GPU_laplace3d_naive: 14.4 (ms)

Copy u2 to host: 32.2 (ms)

10x Gold_laplace3d: 2455.7 (ms)

rms error = 0.000000
```

Notice that the time cost in `GPU_laplace3d_naive` declined from 18.2 ms to 14.4 ms.

laplace_new:

It surprises me that the occupancy is already 100% for the origin code.

But I changed the size to:

```
#define BLOCK_X 32
#define BLOCK_Y 2
#define BLOCK_Z 4
```

Then the occupancy is still 100% while the time cost decreased from 14.2 ms to 14.0 ms:

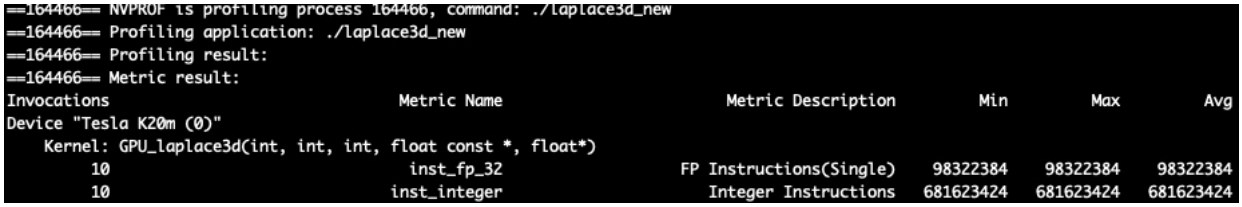
```
Grid dimensions: 256 x 256 x 256
GPU Device 0: "Tesla K20m" with compute capability 3.5


Copy u1 to device: 19.8 (ms)

10x GPU_laplace3d_naive: 14.0 (ms)

Copy u2 to host: 31.6 (ms)

10x Gold_laplace3d: 2224.3 (ms)
```

NVIDIA profiler

The output is stored in .err file.

For **laplace**:

For **laplace_new:**



Notice that the new version has the same number of float point operations with the old version while has more integer operations than the old version.

Changing the computational grid size

After changing the computational grid size to 255 x 255 x 255, I get the following results:

laplace:

```
Grid dimensions: 255 x 255 x 255
GPU Device 0: "Tesla K20m" with compute capability 3.5


Copy u1 to device: 19.5 (ms)

10x GPU_laplace3d_naive: 18.6 (ms)

Copy u2 to host: 31.4 (ms)

10x Gold_laplace3d: 2385.0 (ms)

rms error = 0.000000
```

laplace_new:

```
Grid dimensions: 255 x 255 x 255
GPU Device 0: "Tesla K20m" with compute capability 3.5


Copy u1 to device: 19.5 (ms)

10x GPU_laplace3d_naive: 14.5 (ms)

Copy u2 to host: 31.5 (ms)

10x Gold_laplace3d: 2279.6 (ms)

rms error = 0.000000
```

Compared these results to the previous, we can conclude that the performance does be worse.

However, after setting `JOFF=256` and `KOFF=255*JOFF` as the guide says, make the files and run the programs, I got errors in the .err file:

```
CUDA error at laplace3d.cu:124 code=77(cudaErrorIllegalAddress) "cudaMemcpy(h_u2, d_u1, sizeof(float)*NX*NY*NZ, cudaMemcpyDeviceToHost)"
CUDA error at laplace3d_new.cu:126 code=77(cudaErrorIllegalAddress) "cudaMemcpy(h_u2, d_u1, sizeof(float)*NX*NY*NZ, cudaMemcpyDeviceToHost)"
```

Then I tried to change (NX, NY, NZ) to (256, 255, 255), and got proper results:

laplace:

```
Grid dimensions: 256 x 255 x 255
GPU Device 0: "Tesla K20m" with compute capability 3.5


Copy u1 to device: 19.8 (ms)

10x GPU_laplace3d_naive: 14.1 (ms)

Copy u2 to host: 32.0 (ms)

10x Gold_laplace3d: 2438.9 (ms)

rms error = 0.000000
```

laplace_new:

```
Grid dimensions: 256 x 255 x 255
GPU Device 0: "Tesla K20m" with compute capability 3.5


Copy u1 to device: 20.0 (ms)

10x GPU_laplace3d_naive: 14.3 (ms)

Copy u2 to host: 32.1 (ms)

10x Gold_laplace3d: 2427.6 (ms)

rms error = 0.000000
```

It turns out that I did get a better performance.

Device memory bandwidth

The effective data is 2 * sizeof(float)*NX*NY*NZ = 2 * 32 * 256^3 / 8 / 2^30 GB = 0.125 GB.

laplace:

The average execution time is 14.1 ms / 10 = 1.41 ms. Thus the transfer rate is : 0.125 GB / 1.41 ms = 88.65248227 GB/s.

laplace_new:

The average execution time is 14.3 ms / 10 = 1.43 ms. Thus the transfer rate is : 0.125 GB / 1.43 ms = 87.412587413 GB/s.

The device memory bandwidth is 42.6% of the peak bandwidth capability of the hardware (208 GB/s).

Practical 4: Reduction Operation

Run the code

Output:

```
GPU Device 0: "Tesla K40c" with compute capability 3.5

reduction error = 0.000000
```

As we can see, the reduction error is 0, which means the result is correct.

Handle cases which number of threads is an arbitrary number

Change the kernel code to:

```
_global__ void reduction(float *g_odata, float *g_idata)
{
    // dynamically allocated shared memory

    extern  __shared__  float temp[];

    int tid = threadIdx.x;

    // first, each thread loads data into shared memory

    temp[tid] = g_idata[tid];

    // next, we perform binary tree reduction
    unsigned int floowPow2 = blockDim.x;
    if (floowPow2 & (floowPow2 - 1))
    {
        while(floowPow2 & (floowPow2 - 1))
        {
            floowPow2 &= (floowPow2 - 1);
        }
        if (tid >= floowPow2)
        {
            temp[tid - floowPow2] += temp[tid];
        }
        __syncthreads();
    }

    for (int d = floowPow2>>1; d > 0; d >>= 1) {
      __syncthreads();  // ensure previous step completed
      if (tid<d)  temp[tid] += temp[tid+d];
    }

    // finally, first thread puts result into global memory

    if (tid==0) g_odata[0] = temp[0];
}
```

Then tried the new program with threads number 513 and the result is:

```
GPU Device 0: "Tesla K40c" with compute capability 3.5

reduction error = 0.000000
```

It turns out that the result is still correct.

Try multiple blocks

Change the kernel code using the first method:

```
__global__ void reduction(float *g_odata, float *g_idata)
{
    // dynamically allocated shared memory

    extern  __shared__  float temp[];

    int tid = threadIdx.x;
    int idx = blockIdx.x * blockDim.x + tid;

    // first, each thread loads data into shared memory

    temp[tid] = g_idata[idx];

    // next, we perform binary tree reduction
    unsigned int floowPow2 = blockDim.x;
    if (floowPow2 & (floowPow2 - 1))
    {
        while(floowPow2 & (floowPow2 - 1))
        {
            floowPow2 &= (floowPow2 - 1);
        }
        if (tid >= floowPow2)
        {
            temp[tid - floowPow2] += temp[tid];
        }
        __syncthreads();
    }

    for (int d = floowPow2>>1; d > 0; d >>= 1) {
      __syncthreads();  // ensure previous step completed
      if (tid<d)  temp[tid] += temp[tid+d];
    }

    // finally, first thread puts result into global memory

    if (tid==0) g_odata[blockIdx.x] = temp[0];
}
```

Set the number of blocks with 4, and then the output shows that the result is still correct.

Use shuffle instructions

Change the kernel code to:

```
__global__ void reduction(float *g_odata, float *g_idata, int len)
{
    float sum = float(0);
```

```
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < len; i +=
blockDim.x * gridDim.x) {
        sum += g_idata[i];
    }

    for (int d = warpSize >> 1; d > 0; d >>= 1) {
        sum += __shfl_down(sum, d);
    }

    if ((threadIdx.x & (warpSize - 1)) == 0) {
        atomicAdd(g_odata, sum);
    }
}
```

Output:

```
GPU Device 0: "Tesla K80" with compute capability 3.7

reduction error = 0.000000
```

It turns out that the result is still correct.