TD : Histogram

ROUSSEAUX Thibaut 11/01/2021 – AGH Uni
Faculty of computer science

You can find the code HERE : https://github.com/Swyfee/Histogram_CU

Or on the lhcbgpu2 server if you can access student's directories here : /home/thirou_cuda/Histo

It can be executed using a command like :

./histo -displayData=1 -size=256

DisplayData being the parameter to display all the data (I used a parameter for it as printing a 100000 test would just kill everything) and size being the dataSize.

1 : Answers to the questions :

-What should be considered the most important part for the speed up of the algorithm ?

Look for optimum occupency : There are multiple speed up factors in my opinions, which are :

-When grinding on size, that the number of threads must be a multiple of 32 (warp size is 32). We have to try to have a good ratio of active warps / SM.

-Blocks must have a balanced workload, and all execute for the same amount of time.

-Two few blocks launched

-To not overload threads (1024 thread for 1 block) or blocks (32 threads on 32 blocks) but to have a nice « in between », following CUDA recommandations. Now for the number of blocks I didnt really know how to set it. Here's another point where the block number should be a multiple of the SM's number (30 in our case). I so set the maximum number of blocks to a multiple of 30

After looking for feedback and run some test I've had really good results with a minimum amount of thread/block following the dataSize (if DS = 32 then there is 32 threads) and maximum 256 threads/blocks (multiple of 32). For the number of blocks, as for the number of threads I follow the datasize, adding one block / set of 256 values, and a maximum number of block of 18000 (multiple of 30). This is how I've the best results with lower datasizes as well as huge ones.

This subject is really hard to handle perfectly (but super interesting !) so I stopped my researches for optimization at some point.

TD : Histogram

ROUSSEAUX Thibaut 11/01/2021 – AGH Uni
Faculty of computer science

I tried increasing the number of threads for high dataSizes but it didn't change much the results so I prefered to keep 256 for low dataSizes.

Example :

```
[thirou_cuda@lhcbgpu2 Histo]$ ./histo -size=1000000
GPU Device 0: "GeForce RTX 2060" with compute capability 7.5

Data Size is: 1000000
nb thread: 256
nbblock: 3906
Generation of data sets randomly.
Done
End of the kernel, fetching the results :
All good ! Histograms match
256 threads :
Cuda processing time = 0.195ms,
 Perf = 5.134 Gflops
1 thread :
Cuda processing time = 75.488ms,
 Perf = 0.013 Gflops
```

Example with low data size :

```
[thirou_cuda@lhcbgpu2 Histo]$ ./histo -size=128
GPU Device 0: "GeForce RTX 2060" with compute capability 7.5

Data Size is: 128
nb thread: 128
nbblock: 1
Generation of data sets randomly.
Done
End of the kernel, fetching the results :
All good ! Histograms match
128 threads :
Cuda processing time = 0.104ms,
 Perf = 0.001 Gflops
1 thread :
Cuda processing time = 0.264ms,
 Perf = 0.000 Gflops
```

I so ended up with these values :

```
    //Defining the number of threads to follow the need (ds) with max value 256 (multiple of 32) and < 1024
    int nbThread = min((int)ds, 256);
    printf("nb thread: %d \n", nbThread);
    //Defining the number of blocks to follow the need (if ds = 500 only 2 blocks) with max value a multiple of
30
    int nbBlock =  min(((int)ds/256),18000);
    if (nbBlock == 0) nbBlock = 1;
```

TD : Histogram

ROUSSEAUX Thibaut 11/01/2021 – AGH Uni
Faculty of computer science

-What is the essential step when using shared memory?

Shared memory is a memory, shared between each threads of **a** block. However each threads go on at their own pace. In order to use multiple threads to do a specific task, they need to be synchronized with the use of `__syncthreads();`. Each call to this function is a synchrozination point between each threads of a block, meaning that each threads wait for each other threads of the block to get to this sync.point before going further. This allows to have total control over the program and threads and keep coherent values.

-Describe the memory movement(access and writes into the global memory, take into account both kernels!)

I'm using 3 arrays to complete this task, one with the generated data, and two with the results of respectfully the multithreaded function call and single threaded one.

The data is initialized in the wrapper :

```
// Allocating memory
checkCudaErrors(cudaMalloc((void **)&d_histo, sizeof(unsigned int) * binSize));
checkCudaErrors(cudaMalloc((void **)&d_data, sizeof(unsigned int) * dataSize));
```

Then the generated data is copied to the d_data variable and sent to the device :

```
// Copy the data to the device
checkCudaErrors(cudaMemcpy(d_data, data, sizeof(unsigned int) * dataSize, cudaMemcpyHostToDevice));
```

After this the first kernel is launched with multiple threads and the multi_threaded histogram will be built. We then need to fetch the results from the device and put it in a (local) host variable set to « histo » :

```
// Fetch the result from device to host into histo
printf("End of the kernel, fetching the results :\n");
checkCudaErrors(cudaMemcpy(histo, d_histo, sizeof(unsigned int) * binSize, cudaMemcpyDeviceToHost));
```

After cleaning the « d_histo » I lanch again the kernel on a single thread and fetch the result of this last kernel once more onto the host on another variable « histo_single ».

```
// Launch the kernel on a single thread
histogram<<<1, 1,sizeof(unsigned int) * binSize>>>(d_data, d_histo, dataSize, binSize);
cudaDeviceSynchronize();

// Fetch the result of the last kernel onto the host
checkCudaErrors(cudaMemcpy(histo_single, d_histo, sizeof(unsigned int) * binSize, cudaMemcpyDeviceToHost));
```

TD : Histogram

ROUSSEAUX Thibaut 11/01/2021 – AGH Uni
Faculty of computer science

After using the results of the two histogram by printing them and comparing them, I free the memory used for variables.

```
checkCudaErrors(cudaFree(d_data));
checkCudaErrors(cudaFree(d_histo));
free(histo);
free(histo_single);
free(data);
```

2 : Comment regarding the Shared memory :

The two void functions « staticReverse » and « dynamicReverse » do the same thing which is reversing the generated array (main) of n (64) values but the allocation of memory is different : staticReverse uses « __shared__ int » while dynamicReverse uses « extern __shared__ int ».

The static uses the same method as the one I used in the histogram program, copying from the host to the device, changing the variable and copying back from device to host (for retrieving the values).

In the dynamic method the size is not defined « extern __shared__ int s[] ; » which would mean that we don't know it at the compilation. In order to define its size, we add a 3rd argument to the kernel's call : `dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);`

Which is n * the size of an int. The size of the extern shared variable will so take the size of n*sizeof(int).