1. Abstract

This project realizes filtering an image with a median filter in both CPU and GPU.

The whole idea of this program can be roughly divided into 4 procedures:

1. Load in the image and store it in a data buffer in host memory.
2. Make a copy of the image data in the device memory, launch the kernel to do the filtering, and copy the filter output back to the host memory.
3. Realize the same median filtering process with host CPU, and compare the host filter output with the GPU filter output.
4. Free both host and device memory, write GPU filter output to file, and report timing statistics.
5. About the device

The GPU card I used for this assignment is Nvidia Tesla K80. By running the executable called deviceQuery in ~/NVIDIA\_CUDA-9.1\_Samples/1\_Utilities/deviceQuery/ I can get a lot of device information. The information related to this project is listed below.

1. This device has 13 multiprocessors, with 192 CUDA cores each, so 2496 CUDA cores in total.
2. Compute capability of this device is 3.7, where 3 and 7 are the major/ minor version numbers, respectively. Technical specifications of the compute capability is shown in Table 13 and Table 14 of this webpage: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>
3. Maximum number of resident blocks per multiprocessor is 16, and maximum number of resident threads per multiprocessor is 2048. It means if we have 128 threads per block, we could fit 16 blocks in a multiprocessor before hitting the 2048 thread limit. If we have 256 threads per block, we can only fit 8, but we're still using all of the available threads and will still have full occupancy. However, using 64 threads per block will only use 1024 threads when the 16 block limit is hit, so only 50% occupancy.
4. Warp size is 32. So 32 threads execute the same instruction at the same time (the SIMD fashion). Thread block size should always be a multiple of 32. Otherwise, say, if we have a block size of 50 threads, the GPU will still issue commands to 64 threads and we are just wasting the rest 14 threads.
5. Maximum number of threads per block is 1024. That means if the thread block size equals to , we have reached its maximum; if the thread block size equals to , there should be an error in launching the kernel.
6. Max dimension size of a thread block . The maximum thread block size used in this project is , safe.
7. Max dimension size of a grid size . The maximum grid size used in this project is , safe.
8. Image loading

The image loading uses sdkLoadPGM or sdkLoadPPM4 which are declared in /usr/local/cuda-9.1/samples/common/inc/helper\_image.h.

To make the type in function call consistent, it is better that the data buffer is in the type unsigned char.

1. Copy-compute-copy process

Before copying the image data to the device, we have to allocate memory with the function cudaMalloc, By calling cudaMalloc, we actually dynamically allocate global memory, which resides in device DRAM, for transfers between the host and device as well as for the data input to and output from kernels. The name global here refers to scope, as it can be accessed and modified from both the host and the device.

Use the function cudaMemcpy to transfer data between the host and the device, by setting the parameter cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost.

Before launching the kernel, we need to specify the size of blocks and grids. To make the problem simple, we use square blocks, that is, we have the same number of threads in vertical and horizontal directions. Since maximum number of threads per block is 1024, we can have block size up to . Known that the image size is , so in this case the grid size is . Besides this configuration, I also tried block size = , grid size = , and block size = , grid size = .

The kernel is executed by the device. First, we need to find the index of current thread among all the threads by referencing to blockDim, blockIdx, and threadIdx, which are provided by CUDA.

1. **int** x = blockDim.x \* blockIdx.x + threadIdx.x;
2. **int** y = blockDim.y \* blockIdx.y + threadIdx.y;

We do nothing on the pixels on edges and corners, so they are directly copied to the corresponding positions in the output image. For the pixels that we can apply a square window with given filter size, we perform a median filter. Specifically, take the pixel at position as the center, traverse the all its neighbors in a square window, store the neighborhood pixels in an array, sort the array with random quicksort algorithm, and put the middle value of the sorted array at position in the output image. So each thread takes care of each pixel, but not all the threads will perform a median filter on its pixel.

After the median filter is done, the output image data in the device needs to be copied back to the host by calling cudaMemcpy.

We need error checking code to monitor possible failure in each step of memory allocation, memory copy, and the compute process.

1. Golden standard comparison and timing

A similar median filtered output image is realized in CPU as the golden standard. After the output image from GPU is copied back to the host, it will be compared with the golden standard pixel by pixel, to see if the GPU code was correctly implemented.

In timing the copy-compute-copy process, 100 times iteration is executed and the total cost time is averaged.

1. Timing statistics and analysis



Figure 1 The copy-compute-copy time vs. filter size in various grid/block configurations. I don’t have timing for block size and when filter size = 15, since I got “invalid memory access” error on both cases.

From the black dot line in Figure 1, we can see that the copy-compute-copy time increases as filter size gets larger from 3 up to 15 when the grid size is and block size is . This result makes sense. First, the time spend in copying data from the host to the device and the reverse direction is the same for each case. Second, the compute time increases as the filter size gets larger, since we need more memory to hold the neighborhood data and longer time to sort the data.

Comparing different grid/block configurations, we can see that for a fixed filter size, as thread block size goes down, the copy-compute-copy time goes down as well. It make sense as well. The analysis is shown below.

Since maximum number of resident blocks per multiprocessor is 16, and there are 13 multiprocessors in total, the maximum thread blocks in simultaneous execution is . Hence, in this project, there are thread blocks waiting for execution when calling the kernel. Whenever an SM executes a thread block, all the threads inside the thread block are executed at the same time. As soon as one of its thread block has completed execution, it takes up the serially next thread block. Hence to free a memory of a thread block inside the SM, it is critical that the entire set of threads in the block have concluded execution. Therefore, we could make the argument that having smaller blocks may complete faster since a particularly slow block will hog fewer resources.

1. The filtered images

  

(a) (b) (c)

 

(d) (e)

Figure 2 comparison of the images. (a) original image;

median filtered image with filter size equals to (b) 3; (c) 7; (d) 11; and (e) 15.

From Figure 2(a) and Figure 2(b), we cannot see obvious difference. But it is obvious that as filter size gets larger, the filtered image gets more and more blur. This result is in my expectation. With a larger filter size, a pixel value will be influenced by a larger neighborhood, and the effect of smoothing up the image can be more obvious.

1. Lessons learnt from this project

I learnt a lot from this project, more than programming on GPU. First, this is my first time usign the Google Cloud Platform, and I tried some others services beyond using a VM. Second, I have to build everything up on a total clean VM, installing cmake, gcc, CUDA, and many more. I failed twice in installing the packages, and it turned out to be a version incompatible issue. Third, I have a more thorough understanding of the architecture of GPU, like the concepts of thread, warp, block, SM, GPU core, etc.

Something beyond this project is to try using shared memory, and cuda-gdb.