Report: Project GPGPU

Nguyen Viet Sang, Le Tan Nhat Linh
{viet-sang.nguyen, tan-nhat-linh.le}@etu.unilim.fr

Master 1 Cryptis, Faculty of Science and Technique, University of Limoges

1 Introduction

In this report, we propose 4 versions of implementation to apply an filter to the provided image, including one sequential version, one version with OpenMP and two versions with CUDA. The main steps of this work are listed in section 2.

In general, after reading an image by the provided function `readPPM`, we get an 1D array of its pixels. In this whole report, for the currently processed image, we denote that `y` and `x` are corresponding the number of rows and the number of columns. In other words, we say that `x=image->x` and `y=image->y`. More specifically, the image of Limoges-Bénédictins train station has `x = 500` and `y = 1000`.

For each version, we also measure the average of computing time after running the filtering functions 10000 times. We then compare the results and review on them in section 3. Figure 1 shows the images collected in the outputs.

2 Different versions of implementation

2.1 Sequential processing

We sequentially process pixel by pixel in the 1D array in a manner which applies the target filter to the three values of R, G, B at each pixel. We implement the proposed algorithm in the description of requirement with this sequential processing.

We use some conditions to eliminate the work of computation for four rows and four columns at the borders of the image when calculating with 5×5 filters. The following code in listing 1.1 represents this condition.

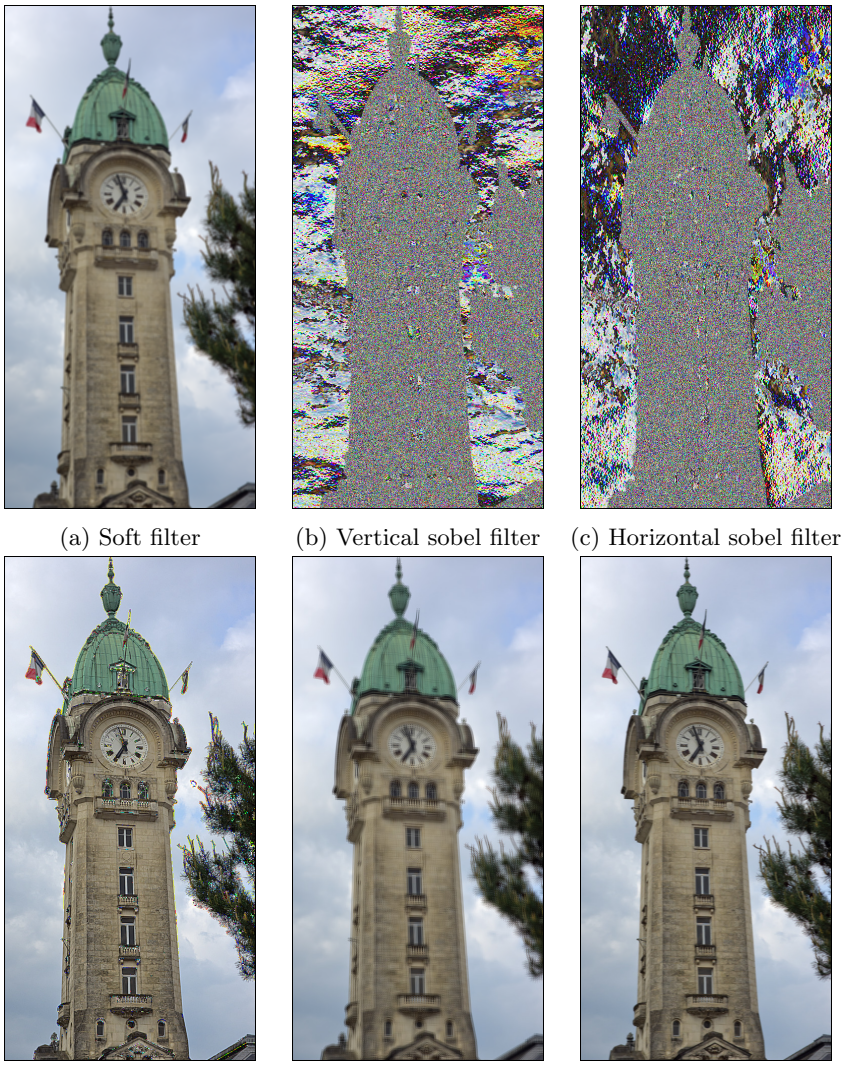
```
for (int i = 0; i <= x * y; i++){
    if ((i / x >= 2)         //do not process the first two rows
    && (i / x < (y - 2))     //do not process the last two rows
    && (i % x != 0)          //do not process the first column
    && (i % x != 1)          //do not process the second column
    && (i % x != (x - 1))    //do not process the last column
    && (i % x != (x - 2))){ //do not process the previous column of the last
    //...
    }
}
```

Listing 1.1: Conditions of eliminating borders

The above conditions are also used in the other versions of implementation which we mention in the next sections. For the results, our filtering function processes the image of the Limoges train station in 81.675 ms on average after measuring 10000 times.

2.2 OpenMP version

In this section, we propose a version of implementation with OpenMP using `pragma parallel` and `pragma for` [2]. Regarding to the `pragma parallel` version, we make a request of 4 (`N_OMP_THREADS = 2`) threads relying on the configuration of the used CPU. We also define the number of threads and the shared variables in the same line with `pragma` as shown in listing 1.2.



(a) Soft filter (b) Vertical sobel filter (c) Horizontal sobel filter

(d) Sharpen filter (e) Shatter filter (f) Blur filter

Fig. 1: Images after applying filters

For the `pragma for`, the executor equally divides the number of iterations in the loop among `N_OMP_THREADS` threads. We choose the `dynamic` mode for `schedule` without `chunk-size` which specifies that the for loop has the dynamic scheduling type and OpenMP divides the iterations into chunks of size 1 (by default).

```
#pragma omp parallel num_threads(N_OMP_THREADS) shared(img_in, img_out,
    m_filter, divide_factor, x, y)
{
    #pragma omp for schedule(dynamic)
    for (int i = 0; i < x * y; i++){
        //...process pixel img_in->data[i]
    }
}
```

Listing 1.2: OpenMP version with `pragma for`

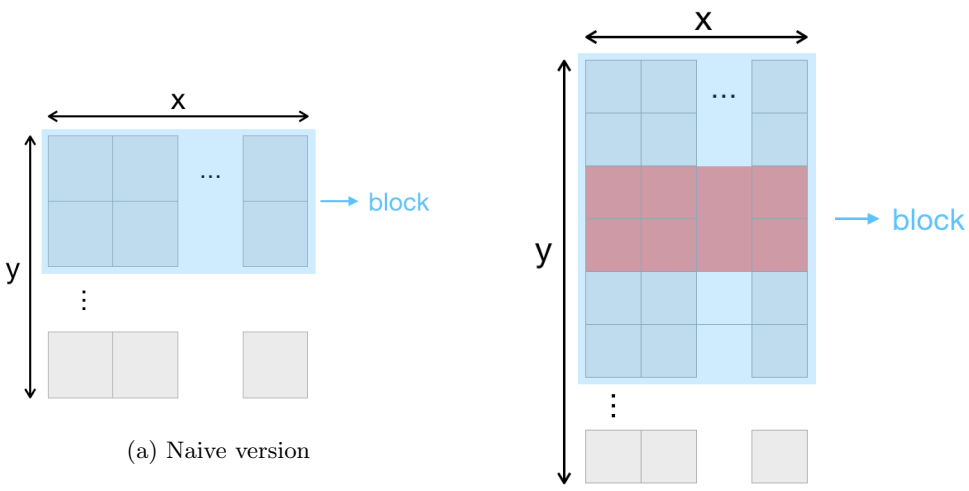
For the results, this program runs the filtering task in 85.671ms on average with 10000 testing times.

2.3 Parallel versions with CUDA

In the section of CUDA programming, we propose two versions with different configuration in the manner of organizing threads and blocks as well as using shared memory. These versions are improved through time during the period that we join this project. The first one is a naive version in which a block contains `blockDim.x = 1024/x * x` threads and the grid consists of `gridDim.x = (x*y + blockDim.x - 1) / blockDim.x` blocks. For the image of Limoges-Bénédictins train station, `x = 500` and `y = 1000`, we have `blockDim.x = 1000` and `gridDim.x = 500`. By this way, the number of threads per block equivalents to the number of pixels of two rows as show in figure 2a and listing 1.3. This naive approach achieves the target image in 1.506ms.

```
1 //...
2 dim3 block_dim = (1024 / x * x);
3 dim3 grid_dim = ((x * y + block_dim.x - 1) / block_dim.x);
4 apply_filter<<<grid_dim, block_dim>>>(dev_img, dev_img_out, dev_filter, x, y,
      divide_factor);
5 //...
```

Listing 1.3: Naive CUDA version



(a) Naive version

(b) Shared memory version

Fig. 2: Parallel versions with CUDA

In the second version, we aim to improve the performance by using shared memory as shown in listing 1.4. The configuration of blocks and threads is the same as the previous version. Recalling that each block contains `2*x` threads which process `2*x` pixels in each two rows of the image (`n_row_per_block = 2`). We make use of shared memory to store `n_row_per_block + (N_FILTER-1)` rows of pixels per block (`N_FILTER = 5`). The center two rows are the current ones whose pixels are processed, and the others are used to support for the calculation with the filter as shown in figure 2b. For general case, we allocate shared memory by dynamic method to adapt any size of the input image. We consider the line 10 and the loop in listing 1.4, only the first `x` pixels in block are used to copy data from global memory to shared memory `s_data`. This kernel takes 0.905 ms on average to finish the task.

```
1 __global__ void apply_filter(PPMPixel *img, PPMPixel *out, int *m_filter, int
      x, int y, int divide_factor){
2     int tid = blockIdx.x * blockDim.x + threadIdx.x;
3     int t = threadIdx.x;
```

```
    int n_col_per_block = blockDim.x / x;

    extern __shared__ PPMPixel s_data[];

    if (tid >= 2 * x
    && tid <= x * y - 2 * x - 1
    && t < x){
        for(int i = 0; i < (n_col_per_block + N_FILTER - 1); i++){
            s_data[i * x + t].red   = img[tid + (i - 2) * x].red;
            s_data[i * x + t].blue  = img[tid + (i - 2) * x].blue;
            s_data[i * x + t].green = img[tid + (i - 2) * x].green;
        }
    }
    __syncthreads();
    //... process  pixel  img[tid]
}

int main(void){
    //...
    dim3 block_dim = (1024 / x * x);
    dim3 grid_dim = ((x * y + block_dim.x - 1) / block_dim.x);
    int n_col_per_block = 1024 / x;
    size_t size_shared_data = (n_col_per_block + (N_FILTER - 1))*x*sizeof(
    PPMPixel);
    //...
    apply_filter<<<grid_dim, block_dim, size_shared_data>>>(dev_img,
    dev_img_out, dev_filter, x, y, divide_factor);
    //...
}
```

Listing 1.4: CUDA shared memory version

3 Results, comparison and discussion

For the hardware, we write a file of code to retrieve the information of the device that we use to work on [1]. The configuration of the device in TP room which we use to practice is as the listing 1.5. To make it fair to compare the different versions of implementation, we use the same device to run as well as the same input image and filter.

```
Detected 1 CUDA Capable device(s)
Device 0: "Quadro K620"
CUDA Driver Version / Runtime Version          10.1 / 9.1
CUDA Capability Major/Minor version number:    5.0
Total amount of global memory:                 1.95 GBytes (2090991616 bytes)
GPU Clock rate:                                1124 MHz (1.12 GHz)
Memory Clock rate:                             900 Mhz
Memory Bus Width:                              128-bit
L2 Cache Size:                                 2097152 bytes
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 65536
Warp size:                                     32
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Maximum sizes of each dimension of a block:    1024 x 1024 x 64
Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
Maximum memory pitch:                          2147483647 bytes
```

Listing 1.5: CUDA Information

Table 1 shows the elapsed time for device to compute with the above versions. We also compare the result of each version to the others to point out whether it is faster or not. As we can see, the sequential version takes 81.675 ms to finish the task while the naive CUDA version only takes 1.506 ms, which means 54.23 times faster. The most impressing result is that the shared memory version just executes in 0.905 ms which achieves 90.25 times faster than the sequential implementation.

A strange point is OpenMP version takes more time than the sequential version (115.346 ms comparing to 81.675 ms). This situation usually happens when we measure the execution time. The reason could be the amount of work to do is not large enough to see the effective impact of OpenMP mechanism. Hence, the OpenMP needs a lot of time to establish the parallel environment rather than do the computation with the filter.

Versions	Sequential	OpenMP `for`	Naive CUDA	Shared memory
Elapsed time (ms)	81.675	115.346	1.506	0.905
Sequential	-	0.71x	54.23x	90.25x
OpenMP `for`	-	-	81.52x	127.45x
Naive CUDA	-	-	-	1.66x

Table 1: Comparison the speed of execution (Run 10000 times)

When comparing the others to OpenMP version, we could see that naive CUDA version and shared memory version are 81.52 times and 127.45 times faster than it respectively. Finally, as our expectation, filtering the image with shared memory approach is 1.66 times faster than executing with the naive approach.

Extra experiments. Besides the main above results, we also measure the average of execution time by running 1000 times. The table 2 shows our collected results. We learn from these results that there is no significant difference between these ones and the previous ones.

Versions	Sequential	OpenMP `for`	Naive CUDA	Shared memory
Elapsed time (ms)	81.158	104.169	1.415	0.916
Sequential	-	0.78x	57.35x	88.60x
OpenMP `for`	-	-	73.62x	113.72x
Naive CUDA	-	-	-	1.54x

Table 2: Comparison the speed of execution (Run 1000 times)

However, we also use Google Colab to get a free CUDA environment to deal with this project. Our implementation could be found by this link [1]. From the table 3 of results with the columns of CUDA versions, we notice that in the many first times of running, the device takes more time to compute than the later ones. Here, we could make a prediction about the scheduler of the Operating System. After running the same instructions, the OS may "know" and schedule the data with cache in a convenient way to execute the next instructions.

Versions	Sequential	OpenMP `for`	Naive CUDA	Shared memory
10000 times	121.522	276.158	0.745	0.542
1000 times	120.759	278.247	0.903	0.702

Table 3: Comparison the speed of execution in Google Colab

Moreover, we improve our naive CUDA version based on the knowledge about the concept of warp. From the hardware point of view, threads in a block are divided into warps of 32 threads which

[1] https://colab.research.google.com/drive/1g_Q0Mbh_lwALGGYfm5k5NONuO_9qIU8-

always execute the same instruction at a certain moment. To avoid the problem of divergence, we set the number of threads in a block to 1024 which is the multiple of 32. This method yields a little improvement (about $20 - 50$ ns) as shown in table 4 (run in Google Colab). This also happens when using device in TP room.

Naive CUDA	10000 times	1000 times
1000 threads/block	0.745 ms	0.903 ms
1024 threads/block	0.721 ms	0.886 ms

Table 4: Comparison the speed of execution with 1024 threads per block

Future work. For the improvement in terms of warp, we have already applied this idea to the naive CUDA version. We also try our best to apply for the share memory version, but the calculation relating to the relationship between indexes of the original array (the image) and the thread identifiers as well as the indexes of elements in shared memory are more complex. If we apply this idea well, it could reduce the execution time.

Besides, in the shared memory version, our approach successfully adapts to the given image. Nevertheless, this kind of implementation is not flexible. If the number of pixels in a row of the given image is greater than the maximum number of threads in a block (`image->x > 1024`), we need to change some lines of code in the current approach to face with new situation.

4 Conclusion

The work shown in this report almost responses to all requirements of the project. We clearly introduce our four version of implementation to apply a 5×5 filter to a certain image. We then show the results collected from our experiments. We also give a comparison and point out the improvement in our results. We could conclude that computation on CUDA is extremely faster than on normal CPU, and working with shared memory CUDA could give the program a better performance in terms of execution time.

References

1. Professional CUDA C Programming. Wrox Press Ltd., Birmingham, UK, UK, 1st edn. (2014)
2. Kenjiro Taura: Openmp (2018), https://www.eidos.ic.i.u-tokyo.ac.jp/~tau/lecture/parallel_distributed/2018/slides/pdf/openmp2.pdf