Contrast Enhancement Using Thrust Library

Bilgin Aksoy

MMI713-Applied Parallel Programming

Bilgin Aksoy
MMI
2252286
01 January 2018

1 Problem Definition

During the assignment, the contrast enhancement algorithm were developed using the THRUST library.

- Finding the minimum valued pixel. ($nMin$)
- Finding the maximum valued pixel. ($nMax$)
- Applying the following operation(Equ-1) to all pixels (i, j) of the source ($pSrc$) and write to the destination ($pDst$).

$$pDst(i,j) = \frac{pSrc(i,j) - nMin}{nMax - nMin} \times 255 \tag{1}$$

2 Algorithm Desciption

I used 4 Thrust function to be able to find the expected outcome:

2.0.1 Finding Minimum-Maximum

Thrust library provides reduction algorithm. I used the reduce function to find the minimum and the maximum.(Listing-1) I realized that the reduction module is also run the kernel two times, when I used the NSight Performance Analysis tool. In the previous implementation (for Assignment-II) I also used the each kernel two times.

Listing 1: Functor

```
int nMin = thrust::reduce(pDst_Dev.begin(), pDst_Dev.end(),257, thrust
    ::minimum<int>());
int nMax = thrust::reduce(pDst_Dev.begin(), pDst_Dev.end(), 0, thrust
    ::maximum<int>());
```

2.0.2 Subtract The Minimum

I used the Thrust iterator algorithm to subtract the minimum value from the input.(Listing-2)

Listing 2: Functor

```
thrust::for_each(pDst_Dev.begin(), pDst_Dev.end(), thrust::
    placeholders::_1 -= nMin);
```

2.0.3 Multiplication-Division

I used the Thrust transform module and send each pixel to a functor(Listing-3). This functor calculates the multiplication and division operations.

Listing 3: Functor

```
struct muldiv_functor
{
        unsigned int a;

        muldiv_functor(unsigned int nConstant, unsigned int
            nNormalizer) {
```

```
                a = round(nConstant / nNormalizer);
        }

        __host__ __device__
        Npp8u operator()(const Npp8u& x) const
        {
        return a*x ;
        }
};
```

3 Benchmarking

The algorithm implemented on CPU is still faster (Table-1) than both the algorithm using NPP Library, GPU implementation(for Assignment-II), and new implementation (for Assignment-III) several times. One of the main bottleneck in parallel programming is memory copies between host and device. Table-2 shows the durations for copying the data. I compared the results of previous implementation(for Assignment-II), and new implementation. The input and results can be seen on Figures 1a to 1c. The output is visually good.

Table 1: The Time-Consuming Of The Three Algorithms

Algorithm	Minimum (μs)	Maximum (μs)	Average Time (μs)
CPU	15,3902	23,0447	19,01675
GPU	654,743	779,464	703,4921
NPP	784,412	888,618	833,2274
Thrust	1110,455	1121,249	6893,76

Table 2: Time Consuming-Memory Copy

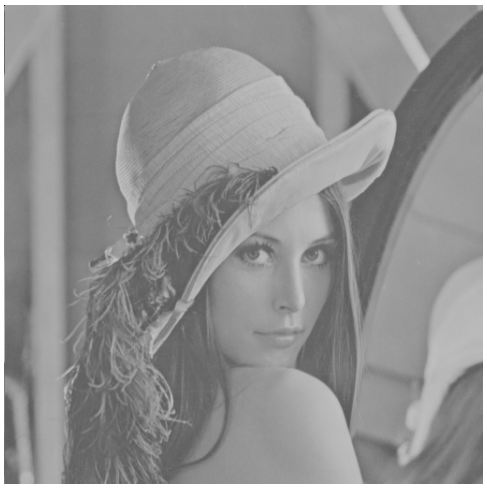
Source	Destination	Duration (μs)	Size (bytes)
HostUnpinned	Device	21123	262144
Device	HostUnpinned	800	4
Device	HostUnpinned	832	4
Device	HostUnpinned	20483	262144

4 Pros-Cons of Solution

Using Thrust library solves many problems about CUDA programming. One doesn't manually manage memory allocations, copy operations, kernel execution parameters etc., and can easily implement an parallel algorithm.

5 Discussion

The one of the main reasons for CPU algorithm is faster than GPU is copy operation from host to device. Another reason is the input size is very small, thinking the device capabilities. My CUDA



(a) Before Enhancement

(b) GPU Result

(c) Thrust Result

Figure 1: Input 1a, GPU Result 1b, and Thrust Result 1c

device has 1920 cores, many of which didn't used at all. Using Thrust library is very easy. But I couldn't be so comfortable when thinking Thrust library like a black-box.

6 Environment

Table 3: Environment

Properties	Specifications
GPU Name	GeForce GTX 1070
Driver Type	WDDM
PCI Bandwidth (GB/s)	15,754
Frame Buffer Physical Size (MiB)	8192
Frame Buffer Bus Width (bits)	256
RAM Type	GDDR5
Frame Buffer Bandwidth (GB/s)	256,256
Graphics Clock (MHz)	1746,5
Processor Clock (MHz)	1746,5
Memory Clock (MHz)	4004
SM Count	15
CUDA Cores	1920