Why Opencv GPU code is slower than CPU?

Asked 12 years, 4 months ago Modified 9 years, 4 months ago Viewed 28k times

14

I'm using opencv242 + VS2010 by a notebook.
I tried to do some simple test of the GPU block in OpenCV, but it showed the GPU is 100 times slower than CPU codes. In this code, I just turn the color image to grayscale image, use the function of **cvtColor**

Here is my code, PART1 is CPU code(test cpu RGB2GRAY), PART2 is upload image to GPU, PART3 is GPU RGB2GRAY, PART4 is CPU RGB2GRAY again. There are 3 things makes me so wondering:

1 In my code, part1 is 0.3ms, while part4 (which is exactly same with part1) is 40ms!!!
2 The part2 which upload image to GPU is 6000ms!!!
3 Part3(GPU codes) is 11ms, it is so slow for this simple image!

```
#include "StdAfx.h"
#include <iostream>
#include "opencv2/opencv.hpp"
#include "opencv2/gpu/gpu.hpp"
#include "opencv2/gpu/gpumat.hpp"
#include "opencv2/core/core.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <ctime>
#include <windows.h>

using namespace std;
using namespace cv;
using namespace cv::gpu;

int main()
{
    LARGE_INTEGER freq;
    LONGLONG QPart1,QPart6;
    double dfMinus, dfFreq, dfTim;
    QueryPerformanceFrequency(&freq);
    dfFreq = (double)freq.QuadPart;

    cout<<getCudaEnabledDeviceCount()<<endl;
    Mat img_src = imread("d:\\CUDA\\train.png", 1);

    // PART1 CPU code~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // From color image to grayscale image.
    QueryPerformanceCounter(&freq);
    QPart1 = freq.QuadPart;
    Mat img_gray;
    cvtColor(img_src,img_gray,CV_BGR2GRAY);
    QueryPerformanceCounter(&freq);
    QPart6 = freq.QuadPart;
    dfMinus = (double)(QPart6 - QPart1);
```

```
        dfTim = 1000 * dfMinus / dfFreq;
        printf("CPU RGB2GRAY running time is %.2f ms\n\n",dfTim);

        // PART2 GPU upload image~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        GpuMat gimg_src;
        QueryPerformanceCounter(&freq);
        QPart1 = freq.QuadPart;
        gimg_src.upload(img_src);
        QueryPerformanceCounter(&freq);
        QPart6 = freq.QuadPart;
        dfMinus = (double)(QPart6 - QPart1);
        dfTim = 1000 * dfMinus / dfFreq;
        printf("Read image running time is %.2f ms\n\n",dfTim);

        GpuMat dst1;
        QueryPerformanceCounter(&freq);
        QPart1 = freq.QuadPart;

        /*dst.upload(src_host);*/
        dst1.upload(imread("d:\\CUDA\\train.png", 1));

        QueryPerformanceCounter(&freq);
        QPart6 = freq.QuadPart;
        dfMinus = (double)(QPart6 - QPart1);
        dfTim = 1000 * dfMinus / dfFreq;
        printf("Read image running time 2 is %.2f ms\n\n",dfTim);

        // PART3~ GPU code~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        // gpuimage From color image to grayscale image.
        QueryPerformanceCounter(&freq);
        QPart1 = freq.QuadPart;

        GpuMat gimg_gray;
        gpu::cvtColor(gimg_src,gimg_gray,CV_BGR2GRAY);

        QueryPerformanceCounter(&freq);
        QPart6 = freq.QuadPart;
        dfMinus = (double)(QPart6 - QPart1);
        dfTim = 1000 * dfMinus / dfFreq;
        printf("GPU RGB2GRAY running time is %.2f ms\n\n",dfTim);

        // PART4~CPU code(again)~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        // gpuimage From color image to grayscale image.
        QueryPerformanceCounter(&freq);
        QPart1 = freq.QuadPart;
        Mat img_gray2;
        cvtColor(img_src,img_gray2,CV_BGR2GRAY);
        BOOL i_test=QueryPerformanceCounter(&freq);
        printf("%d \n",i_test);
        QPart6 = freq.QuadPart;
        dfMinus = (double)(QPart6 - QPart1);
        dfTim = 1000 * dfMinus / dfFreq;
        printf("CPU RGB2GRAY running time is %.2f ms\n\n",dfTim);

        cvWaitKey();
        getchar();
        return 0;
    }
```

c++ opencv gpu

Share

Improve this question

Follow

edited May 1, 2015 at 19:52

boatcoder

18.1k ● 21 ● 124 ● 184

asked Aug 22, 2012 at 13:29

David Ding

680 ● 3 ● 9 ● 19

12 It's not that the GPU is generally "slow". However, memory transfer between host and device is *extremely* slow. GPU computation only makes sense if you can offload a very large, highly parallel computation to the device. – Kerrek SB Aug 22, 2012 at 13:42

Should also check answers.opencv.org/question/1670/... – Sam Aug 22, 2012 at 14:10

Then you pass not allocated GpuMat you have GPU memory allocation inside GPU-optimized functions. To avoid it you should preallocate your memory with proper size before function usage. – geek Aug 22, 2012 at 19:33

5 Answers

Sorted by: Highest score (default)

32

Most answers above are actually wrong. The reason why it is slow by a factor 20.000 is of course not because of 'CPU clockspeed is faster' and 'it has to copy it to the GPU' (accepted answers). These are factors, but by saying that you omit the fact that you have vastly more computing power for a problem that is disgustingly parallel. Saying 20.000x performance difference is because of the latter is just so plain ridiculous. The author here knew something was wrong that's not straight forward. Solution:

Your problem is that CUDA needs to initialize! It will always initialize for the first image and generally takes between 1-10 seconds, depending on the alignment of Jupiter and Mars. Now try this. Do the computation **twice** and then time them both. You will probably see in this case that the speeds are within the same order of magnutide, not 20.000x, that's ridiculous. Can you do something about this initialization? Nope, not that I know of. It's a snag.

edit: I just re-read the post. You say you're running on a notebook. Those often have shabby GPU's, and CPU's with a fair turbo.

Share

Improve this answer

Follow

edited Jul 30, 2015 at 13:04

answered Jun 7, 2015 at 12:40

TimZaman

2,699 ● 2 ● 28 ● 36

3 This should be the accepted answer as it gives the actual reason of the observed performance issue. @Tae already pointed it out 3 years before, but was not as explicit in the analysis. – Ale Nov 2, 2018 at 17:55

25

cvtColor isn't doing very much work, to make grey all you have to is average three numbers.

The cvColor code on the CPU is using SSE2 instructions to process upto 8 pixels at once and if you have TBB it's using all the cores/hyperthreads, the CPU is running at 10x the clock speed of the GPU and finally you don't have to copy data onto the GPU and back.

Share Improve this answer Follow

answered Aug 22, 2012 at 13:55

Martin Beckett
96k ● 28 ● 195 ● 268

Thanks for answering!! In this code, I used CPU for RGB2GRAY for twice(same code, one before GPU code, while the other is after GPU). It shows that second time is much slower than the first one. But they are same codes! Could you please give me some prompts? – David Ding Aug 23, 2012 at 6:42

2 This may also come from the fact that creating a CUDA context takes time. I don't know how you evaluate the timing of your code. – Amir Zadeh Apr 17, 2013 at 19:00

9

try to run more than once....

----------excerpt from http://opencv.willowgarage.com/wiki/OpenCV%20GPU%20FAQ
Perfomance

Why first function call is slow?

That is because of initialization overheads. On first GPU function call Cuda Runtime API is initialized implicitly. Also some GPU code is compiled (Just In Time compilation) for your video card on the first usage. So for performance measure, it is necessary to do dummy function call and only then perform time tests.

If it is critical for an application to run GPU code only once, it is possible to use a compilation cache which is persistent over multiple runs. Please read nvcc documentation for details (CUDA_DEVCODE_CACHE environment variable).

Share Improve this answer Follow

answered Sep 24, 2012 at 5:36

Tae
111 ● 4

1

cvtColour is a small operation, and any performance boost you get from doing it on the GPU is vastly outweighed by memory transfer times between host (CPU) and device (GPU). Minimizing the latency of this memory transfer is a primary challenge of any GPU computing.

Share Improve this answer Follow

answered Jun 12, 2013 at 3:17

1"
27.1k ● 32 ● 148 ● 202

0

What GPU do you have?

Check compute compability, maybe it's the reason.

https://developer.nvidia.com/cuda-gpus

> This means that for devices with CC 1.3 and 2.0 binary images are ready to run. For all newer platforms, the PTX code for 1.3 is JIT'ed to a binary image. For devices with CC 1.1 and 1.2, the PTX for 1.1 is JIT'ed. For devices with CC 1.0, no code is available and the functions throw Exception. For platforms where JIT compilation is performed first, the run is slow.

http://docs.opencv.org/modules/gpu/doc/introduction.html

Share Improve this answer Follow

answered Apr 16, 2013 at 13:22

mrgloom
21.5k ● 41 ● 189 ● 318