Image resize with OpenCV 3.x and CUDA

Posted on December 22, 2015 by claudiu

This prototype implements and tests different downsampling algorithms of grayscale and color images to any size using C++, CUDA, OpenCV 3.X.

In image analysis downsampling is a fundamental transformation to significantly decrease the processing time with little or no errors introduced into the system.

Several algorithms exist, from simple skipping pixels which results in aliased images to more sophisticated reduction kernels with decrease in speed. Popular image processing libraries such OpenCV implement it as a simple geometric transformation like pixel area relation where areas are shrunk independently.

Three different methods are being implemented and compared to each other in the prototype :

- OpenCV 3.x CPU based method cv::rezize() from the imgproc module.
- OpenCV 3.x GPU based method cuda::rezize() from the cudawarping module.
- own method downscaleCuda() implemented in Cuda to run in parallel on the GPU. This method simply merges frames of pixels into a single points in the output image. For instance, a downscale of 4x on both x and y scale will merge frames of 4×4 pixels of the original image in a single pixel in the output image.

The code is build and compiled on a MSI laptop with Geforce GTX 970M, 3GB GDDR5 and 13 streaming multi processors.

OpenCV CPU implementation

In this first test, the image is loaded from the disk and and downscaled 4 times in both x and y scale using cv::resize.

Execution of OpenCV method

```
void processUsingOpenCvCpu(std::string input_file, std::string output_file) {
 //Read input image from the disk
 Mat input = imread(input_file, CV_LOAD_IMAGE_COLOR);
 if(input.empty())
 {
	std::cout<<"Image Not Found: "<< input_file << std::endl;
  return;
 }

 //Create output image
 Mat output;

 GpuTimer timer;
 timer.Start();

 resize(input, output, Size(), .25, 0.25, CV_INTER_AREA); // downscale 4x on both x and y

 timer.Stop();
 printf("OpenCv Cpu code ran in: %f msecs.\n", timer.Elapsed());

 imwrite(output_file, output);
}
```

OpenCV GPU implementation

In this second test, the image is loaded from the disk and and downscaled 4 times in both x and y scale using cuda::resize.

Execution of OpenCV method

```
void processUsingOpenCvGpu(std::string input_file, std::string output_file) {
 //Read input image from the disk
 Mat inputCpu = imread(input_file,CV_LOAD_IMAGE_COLOR);
 cuda::GpuMat input (inputCpu);
 if(input.empty())
 {
  std::cout<<"Image Not Found: "<< input_file << std::endl;
  return;
 }

 //Create output image
 cuda::GpuMat output;

 GpuTimer timer;
 timer.Start();

 cuda::resize(input, output, Size(), .25, 0.25, CV_INTER_AREA); // downscale 4x on both x and y

 timer.Stop();
 printf("OpenCv Gpu code ran in: %f msecs.\n", timer.Elapsed());

 Mat outputCpu;
 output.download(outputCpu);
 imwrite(output_file, outputCpu);

 input.release();
 output.release();
}
```

Own CUDA implementation

In this third test, the image is loaded from the disk and and downscaled 4 times in both x and y scale using own implemented cuda method.

Execution of OpenCV method

```
void processUsingCuda(std::string input_file, std::string output_file) {
 //Read input image from the disk
 cv::Mat input = cv::imread(input_file,CV_LOAD_IMAGE_UNCHANGED);
 if(input.empty())
 {
  std::cout<<"Image Not Found: "<< input_file << std::endl;
  return;
 }

 //Create output image
 Size newSize( input.size().width / 4, input.size().height / 4 );
 Mat output (newSize, input.type());

 downscaleCuda(input, output);

 imwrite(output_file, output);
}
```

Execution of

```
void downscaleCuda(const cv::Mat& input, cv::Mat& output)
{
 //Calculate total number of bytes of input and output image
 const int inputBytes = input.step * input.rows;
 const int outputBytes = output.step * output.rows;

 unsigned char *d_input, *d_output;

 //Allocate device memory
 SAFE_CALL(cudaMalloc<unsigned char>(&d_input,inputBytes),"CUDA Malloc Failed");
 SAFE_CALL(cudaMalloc<unsigned char>(&d_output,outputBytes),"CUDA Malloc Failed");

 //Copy data from OpenCV input image to device memory
 SAFE_CALL(cudaMemcpy(d_input,input.ptr(),inputBytes,cudaMemcpyHostToDevice),"CUDA Memcpy Host To Device Failed");

 GpuTimer timer;
 timer.Start();

 //Specify a reasonable block size
 const dim3 block(16,16);

 //Calculate grid size to cover the whole image
 const dim3 grid((output.cols + block.x - 1)/block.x, (output.rows + block.y - 1)/block.y);

 // Calculate how many pixels in the input image will be merged into one pixel in the output image
 const float pixelGroupSizeY = float(input.rows) / float(output.rows);
 const float pixelGroupSizeX = float(input.cols) / float(output.cols);

 //Launch the size conversion kernel
 resizeCudaKernel<<<grid,block>>>(d_input,d_output,output.cols,output.rows,input.step,output.step, pixelGroupSizeX, p

 timer.Stop();
 printf("Own Cuda code ran in: %f msecs.\n", timer.Elapsed());

 //Synchronize to check for any kernel launch errors
 SAFE_CALL(cudaDeviceSynchronize(),"Kernel Launch Failed");

 //Copy back data from destination device meory to OpenCV output image
 SAFE_CALL(cudaMemcpy(output.ptr(),d_output,outputBytes,cudaMemcpyDeviceToHost),"CUDA Memcpy Host To Device Failed");

 //Free the device memory
 SAFE_CALL(cudaFree(d_input),"CUDA Free Failed");
 SAFE_CALL(cudaFree(d_output),"CUDA Free Failed");
 //SAFE_CALL(cudaDeviceReset(),"CUDA Device Reset Failed");
}
```

Definition of

```
__global__ void resizeCudaKernel( unsigned char* input,
  unsigned char* output,
  const int outputWidth,
  const int outputHeight,
  const int inputWidthStep,
  const int outputWidthStep,
  const float pixelGroupSizeX
  const float pixelGroupSizeX,
  const float pixelGroupSizeY,
  const int inputChannels)
{
 //2D Index of current thread
 const int outputXIndex = blockIdx.x * blockDim.x + threadIdx.x;
 const int outputYIndex = blockIdx.y * blockDim.y + threadIdx.y;

 //Only valid threads perform memory I/O
 if((outputXIndex<outputWidth) && (outputYIndex<outputHeight))
 {
  // Starting location of current pixel in output
  int output_tid  = outputYIndex * outputWidthStep + (outputXIndex * inputChannels);

  // Compute the size of the area of pixels to be resized to a single pixel
  const float pixelGroupArea = pixelGroupSizeX * pixelGroupSizeY;

  // Compute the pixel group area in the input image
  const int intputXIndexStart = int(outputXIndex * pixelGroupSizeX);
  const int intputXIndexEnd = int(intputXIndexStart + pixelGroupSizeX);
  const float intputYIndexStart = int(outputYIndex * pixelGroupSizeY);
  const float intputYIndexEnd = int(intputYIndexStart + pixelGroupSizeY);

  if(inputChannels==1) { // grayscale image
   float channelSum = 0;
   for(int intputYIndex=intputYIndexStart; intputYIndex<intputYIndexEnd; ++intputYIndex) {
    for(int intputXIndex=intputXIndexStart; intputXIndex<intputXIndexEnd; ++intputXIndex) {
     int input_tid = intputYIndex * inputWidthStep + intputXIndex;
     channelSum += input[input_tid];
    }
   }
   output[output_tid] = static_cast<unsigned char>(channelSum / pixelGroupArea);
  } else if(inputChannels==3) { // RGB image
   float channel1stSum = 0;
   float channel2stSum = 0;
   float channel3stSum = 0;
   for(int intputYIndex=intputYIndexStart; intputYIndex<intputYIndexEnd; ++intputYIndex) {
    for(int intputXIndex=intputXIndexStart; intputXIndex<intputXIndexEnd; ++intputXIndex) {
     // Starting location of current pixel in input
     int input_tid = intputYIndex * inputWidthStep + intputXIndex * inputChannels;
     channel1stSum += input[input_tid];
     channel2stSum += input[input_tid+1];
     channel3stSum += input[input_tid+2];
    }
   }
   output[output_tid] = static_cast<unsigned char>(channel1stSum / pixelGroupArea);
   output[output_tid+1] = static_cast<unsigned char>(channel2stSum / pixelGroupArea);
   output[output_tid+2] = static_cast<unsigned char>(channel3stSum / pixelGroupArea);
  } else {
  }
 }
}
```

Execution times of the 3 implementations on simple_room-wallpaper-4096×3072.jpg, 2,3 MB, 4096×3072 pixels to a 4x downscale

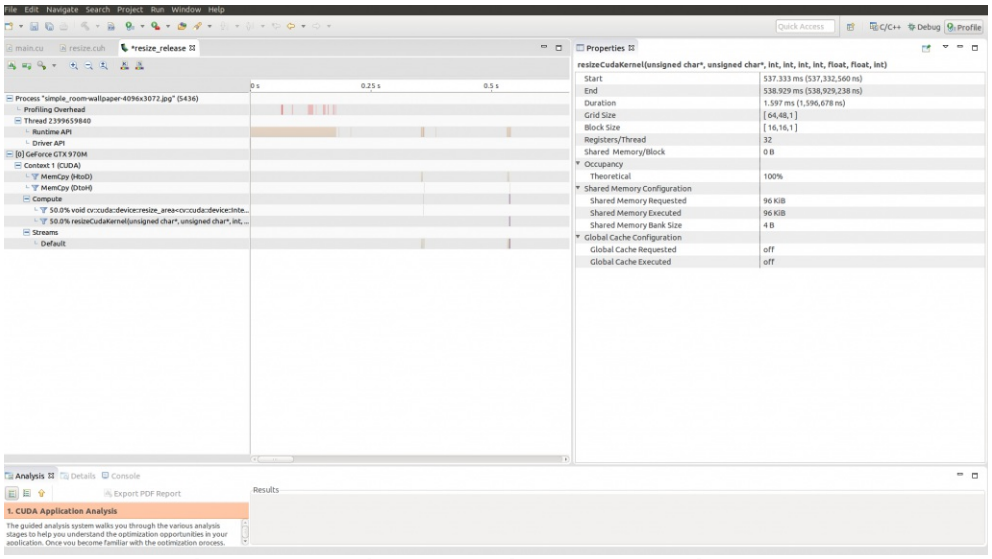
The 3 implementations were run 10 times in a row. By excluding the peaks from the output, we can see that the OpenCV CPU implementation took in average ~6.5 ms, the OpenCV GPU implementation took in average ~1.65 ms and the CUDA implementation took in average ~1.60 ms.

```
OpenCv Cpu code ran in: 5.895104 msecs.
OpenCv Gpu code ran in: 2.064192 msecs.
Own Cuda code ran in: 1.587136 msecs.
OpenCv Cpu code ran in: 6.800448 msecs.
OpenCv Gpu code ran in: 2.160992 msecs.
Own Cuda code ran in: 1.792928 msecs.
OpenCv Cpu code ran in: 6.707936 msecs.
OpenCv Gpu code ran in: 1.602560 msecs.
Own Cuda code ran in: 1.611424 msecs.
OpenCv Cpu code ran in: 6.764896 msecs.
OpenCv Gpu code ran in: 1.604352 msecs.
Own Cuda code ran in: 1.598176 msecs.
OpenCv Cpu code ran in: 6.612064 msecs.
OpenCv Gpu code ran in: 1.606272 msecs.
Own Cuda code ran in: 1.605056 msecs.
OpenCv Cpu code ran in: 5.397920 msecs.
OpenCv Gpu code ran in: 1.609248 msecs.
Own Cuda code ran in: 1.602912 msecs.
OpenCv Cpu code ran in: 6.442816 msecs.
OpenCv Gpu code ran in: 1.909632 msecs.
Own Cuda code ran in: 1.596384 msecs.
OpenCv Cpu code ran in: 6.464128 msecs.
OpenCv Gpu code ran in: 1.605632 msecs.
Own Cuda code ran in: 1.576896 msecs.
OpenCv Cpu code ran in: 6.534048 msecs.
OpenCv Gpu code ran in: 1.605664 msecs.
Own Cuda code ran in: 1.597440 msecs.
OpenCv Cpu code ran in: 6.459168 msecs.
OpenCv Gpu code ran in: 1.613088 msecs.
Own Cuda code ran in: 1.589792 msecs.
```

All these times does not include the time spent on loading the image into CPU or GPU memory, but only the time spent in processing the downscale. In a normal usage an image is loaded once and many transformations are aplied on it, so the loading time is irrelevant to the total processing time.

Profiling of the OpenCV GPU and own CUDA implementation using NSight profiler

Also in profiling can be seen the time spent on computing is about 1.5 ms for both methods.



```
Process "simple_room-wallpaper-4096x3072.jpg" (5436)
  Profiling Overhead
  Thread 2399659840
    Runtime API
    Driver API
[0] GeForce GTX 970M
  Context 1 (CUDA)
    MemCpy (HtoD)
    MemCpy (DtoH)
    Compute
      50.0% void cv::cuda::device::resize_area<cv::cuda::device::Inte...
      50.0% resizeCudaKernel(unsigned char*, unsigned char*, int, ...
    Streams
      Default
```

Properties

resizeCudaKernel(unsigned char*, unsigned char*, int, int, int, int, float, float, int)

Start	537.333 ms (537,332,560 ns)
End	538.929 ms (538,929,238 ns)
Duration	1.597 ms (1,596,678 ns)
Grid Size	[64,48,1]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Occupancy	
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

1. CUDA Application Analysis

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process,

Original and generated rescaled images with OpenCV GPU and own method

Original image

Rescaled image using OpenCV GPU

Rescaled image using CUDA method

Conclusion

Both GPU versions tend to have a similar speed. While the OpenCV implementation is straightforward and easy to use, the CUDA implementation gives more flexibility like sharing of the GPU memory between CUDA kernels without the need to copy back and forth between the CPU and GPU. Speedups can also been achieved using features of the CUDA runtime such streams, shared memory or pinned memory.

Sources

https://bitbucket.org/coldvisionio/coldvision-library/src/6f9843646d1a4443d860df42e642a4a1151d6c10/samples/2_imaging/resize/

Resources

https://en.wikipedia.org/wiki/Decimation_(signal_processing)

http://stackoverflow.com/questions/6133957/image-downsampling-algorithms

http://stackoverflow.com/questions/31761203/the-most-efficient-way-to-resize-cvmat

http://docs.opencv.org/master/da/d54/group__imgproc__transform.html#ga47a974309e9102f5f08231edc7e7529d

http://docs.opencv.org/master/db/d29/group__cudawarping.html#ga4f5fa0770d1c9efbadb9be1b92a6452a

http://docs.opencv.org/master/d3/d63/classcv_1_analysis1Mat.html#gsc.tab=0

http://docs.opencv.org/master/d0/d60/classcv_1_1cuda_1_1GpuMat.html#ab44144c4debe262119be744715138977

https://github.com/Itseez/opencv/blob/master/modules/cudawarping/src/resize.cpp

https://github.com/Itseez/opencv/blob/master/samples/gpu/hog.cpp

http://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

http://docs.opencv.org/2.4/doc/tutorials/introduction/linux_eclipse/linux_eclipse.html

This entry was tagged *c++*, *cuda*, *downsampling*, *downscaling*, *image resize*, *opencv*, *tegra tk1*. Bookmark the *permalink*.

← RGB to grayscale conversion in CUDA and OpenCV

Bilateral Filtering with CUDA and OpenCV 3.x →

Leave a Reply

Your email address will not be published.

Comment

Name

Email

Website

POST COMMENT

Search …

RECENT POSTS

- Model Predictive Control for cars in C++
- Robot Localization with Particle Filter and C++
- Object tracking with LIDAR, Radar, sensor fusion and Extended Kalman Filter
- Vehicle Detection using OpenCV and SVM classifier
- Advanced Lane Finding using OpenCV

ARCHIVES

- August 2017
- June 2017
- April 2017
- March 2017
- February 2017
- July 2016
- June 2016

- May 2016
- April 2016
- March 2016
- February 2016
- January 2016
- December 2015
- November 2015

TAGS

algorithm autonomous driving bilateral filter c++ classification cuda deep learning downsampling downscaling driving simulator edge detection feature descriptor feature detection feature point gaussian blur gpu grayscale histogram histogram calculation image gradient image histogram image processing image resize image segmentation jetson tk1 knn linux motion model noise removal nvidia object detection object tracking opencv python rgb segmentation self-driving sketch smoothing sobel derivatives sobel operator surf tegra tk1 video processing watershed