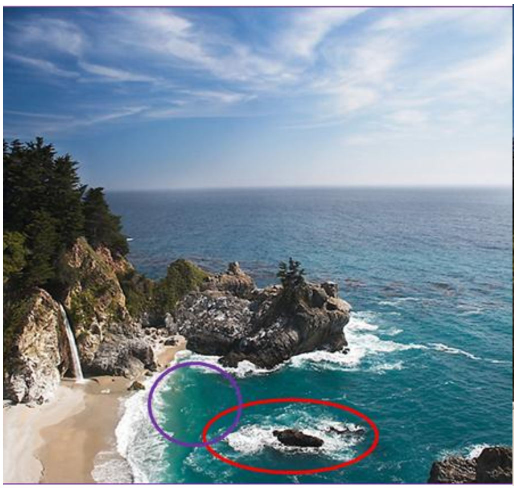
A Content-Aware image resizer on CUDA model

--Haojian's Final report of High Performance Computing

Introduction

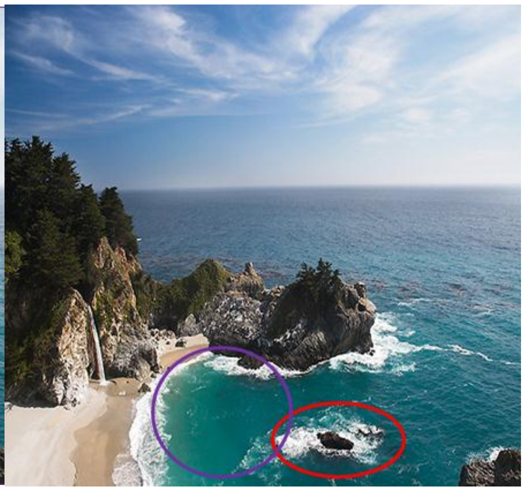
Content Aware Image Resizing is a way to re-target an image size without modifying its content ratio, in other words: non-linear image resizing. The algorithm was first explained by Shai Avidan and Ariel Shamir and published in 2007 ("Seam Carving for Content-Aware Image Resizing.") The algorithm described in this paper shows a rather easy way one can do non-linear image resizing. This allows the image to be resized while changing its aspect ratio and keeping important features untouched. Essentially, it removes (or adds) parts to an image that would be least noticed. One of the samples is like the following figures.

Original pic @ 700 * 466



Resize to 500 * 466 with CAIR

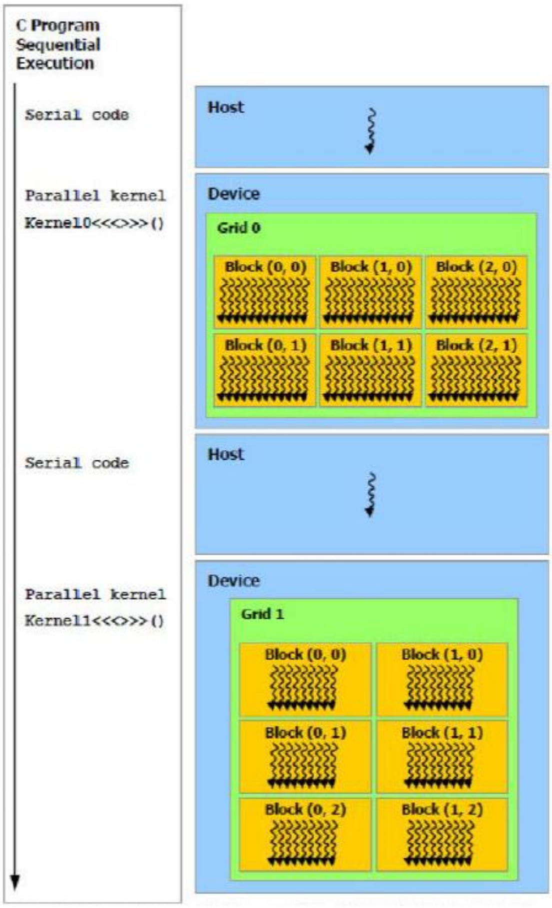
Keep important details; remove some unnecessary details, like sea or sky.



Resize to 500 * 466 directly

Keep important details; remove some Unnecessary details, like sea or sky.

And ***CUDA™*** (Compute Unified Device Architecture) is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). Namely, CUDA is the computing engine in Nvidia graphics processing units (GPUs) that is accessible to software developers through variants of industry standard programming languages. Programmers use 'C for CUDA' (C with Nvidia extensions and certain restrictions), compiled through a PathScale Open64 C compiler, to code algorithms for execution on the GPU.



In CUDA platform, Computing is evolving from "central processing" on the CPU to "co-processing" on the CPU and GPU. In most cases, GPU-accelerated applications run the sequential part of their workload on the CPU – which is optimized for single-threaded performance – while accelerating parallel processing on the GPU. This is part of "GPU computing".

Proposal

However, the problem is that ***Content Aware Image Resizing (CAIR)*** is usually a computational expensive task for most personal laptop. Even with a parallel CAIR version, the time cost is still nearly 6 seconds for a picture at resolution 700 * 466. What's worse, the time cost increased rapidly to 25 seconds when we increase the size to 1024 * 768.

In my expectation, I also wish this interesting technique could apply to video target, which a stricter requirement for performance. So I am trying to improve the performance with my laptop device, which is a topic in Desktop High Performance Computing.

The test environment is on my laptop, which has following components:

Intel® Core™2 Duo Processor T7700 (4M Cache, 2.40 GHz, 800 MHz FSB)

NVIDIA Quadro NVS 140M, Memory Size (128MB), CUDA cores (16)

The expectation is that my program can process a photo with common resolution in 0.5 second.

Program Design

Algorithm for Image Processing:

1. Load an image and convert it to array[pixel]
2. A grayscale version of the image has to be calculated
3. The edges of the image (Sobel convolution is used in our case) and its energy matrix is computed.

4. The seam of least energy (1 pixel vertical line from the bottom to the top of the energy matrix) is detected
5. Then the pixel of the detected seam is removed from the original image and the result is re-injected as a source image to step 3.

Which part can be parallelized?

1. Grayscale_Image

   Multi-threaded with each thread getting a strip across the image.
2. Edge_Detect (process vertically & horizontal separately)

   Multi-threaded with each thread getting a strip across the image.
3. Add/Remove operations

   Split the task to calculate the minimum of energy

Pseudocode for Structure:

1. CPU sequential program Init:

   cudaMalloc & cudaMemcpy, split a task to blocks
2. Call '__global__ void' function [GrayImage], and run the GPU parts.

   cudaThreadSynchronize();
3. Call '__global__ void' function [EdgeDetect], and run the GPU parts.

   cudaThreadSynchronize();
4. Repeated to more tasks

Some sample codes are as follows:

CPU code:

```
CML_element **dv; //device array
cudaMalloc((void **)&dv, imageMemSize);
cudaMemcpy(dv, Source, imageMemSize, cudaMemcpyHostToDevice);
dim3 dimGrid(1,1);
dim3 dimBlock(height, 1, 1); //divided vertical.
energy_Cal_CUDA<<<dimGrid, dimBlock, vsize >>>(dv,n);
cudaThreadSynchronize();
cudaMemcpy(Source, dv, imageMemSize, cudaMemcpyDeviceToHost);
```

CPU calls GPU code:

```
__global__ void energy_Cal_CUDA(CML_element **dv, int n)
{
    extern __shared__ int sv[];
    int myIndex = threadIdx.x;
    int i=0;
    for(i=0; i<n;i++)
        sv[myIndex] = basic_energy_cal_CUDA(dv[i], dv);
}
```

GPU computation code:

```
__device__ int basic_energy_cal_CUDA(CML_element *element,CML_element **_dv)
```

GPU computation part is as same as the one on CPU, and the code is quite long, so I haven't put it here.

Performance Evaluation

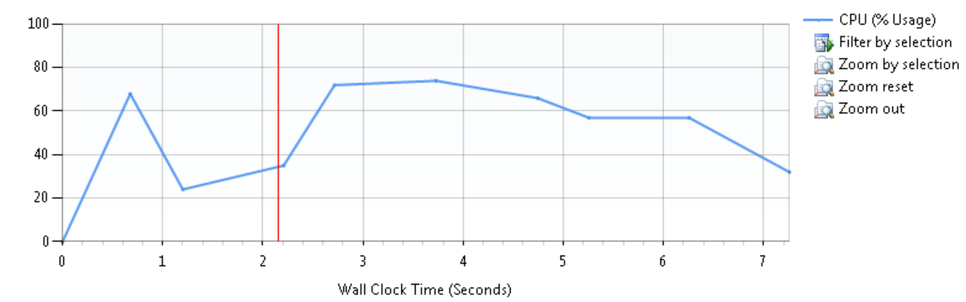
Then, I collected some data from my code and some sample image under the windows default photo direction.

Category	700 * 466 ⇒500 * 466 [#(thread) = 4]	1024 * 768 ⇒500 * 500 [#(thread) = 4]	1024 * 768 => 500 * 500 [#(thread) = 8]	1024 * 768 => 500 * 500 [#(thread) = 16]
Sequential Application	5.824 sec	25.237 sec		
Parallel Application based on pthread	4.922 sec	25.451 sec	22.04 sec	22.57 sec
GPU-Accelerated Application (without improvement on energy computing)	4.571 sec	22.226 sec		

*All the time is the average value of 3 times.

In fact, there is no big improvement with GPU or Parallel solution. That's wired.
Later, I used the Performance Profiling tools in Microsoft Visual Studio package to assist my work, that's also the most important reason for me to work on visual studio & windows.
Here is the profiling usage result:

Sample Profiling Report
1,097 total samples collected



Hot Path

The most expensive call path based on sample counts

Function Name	Inclusive Samples %	Exclusive Samples %
CAIR(class CML_Matrix<struct CML_RGBA> *,class CML_Matrix<int> *,int,int,enum CAIR...	80.04	0.00
CAIR_Remove(class CML_Matrix<struct CML_element *> *,int,enum CAIR_convolutio...	77.30	0.00
Energy_Path(class CML_Matrix<struct CML_element *> *,int *,enum CAIR_energy,...	77.30	0.00
Energy_Map(class CML_Matrix<struct CML_element *> *,enum CAIR_energy,int...	75.75	7.84
CML_Matrix<struct CML_element *>::operator()(int,int)	**56.79**	**56.79**

Related Views: Call Tree Functions

Function Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %
__tmainCRTStartup	931	0	84.87	0.00
_main	931	0	84.87	0.00
_mainCRTStartup	931	0	84.87	0.00
CAIR(class CML_Matrix<struct CML_RGBA	878	0	80.04	0.00
CAIR_Remove(class CML_Matrix<struct C	848	0	77.30	0.00
Energy_Path(class CML_Matrix<struct CM	848	0	77.30	0.00
Energy_Map(class CML_Matrix<struct CM	831	86	75.75	7.84
CML_Matrix<struct CML_element *>::ope	742	742	67.64	67.64
[MSVCRT.dll]	166	0	15.13	0.00
Unknown Frame(s)	166	0	15.13	0.00
[pthreadVC2.dll]	161	0	14.68	0.00
min_of_three(int,int,int)	109	109	9.94	9.94
Remove_Quadrant(void *)	104	6	9.48	0.55
Convolve_Pixel(class CML_Matrix<struct (	103	24	9.39	2.19
Edge_Quadrant(void *)	44	1	4.01	0.09
CML_to_BMP(class CML_Matrix<struct CM	25	1	2.28	0.09
CML_Matrix<struct CML_RGBA>::operato	22	22	2.01	2.01
Init_CML_Image(class CML_Matrix<struct	19	0	1.73	0.00
BMP::operator()(int,int)	18	18	1.64	1.64
BMP_to_CML(class BMP *,class CML_Mat	17	1	1.55	0.09
[CAIRPTHREADNOV26B.exe]	15	15	1.37	1.37
Generate_Path(class CML_Matrix<struct C	14	1	1.28	0.09
Get_Max(class CML_Matrix<struct CML_el	13	6	1.19	0.55
Gray_Quadrant(void *)	13	2	1.19	0.18
CML_Matrix<struct CML_element *>::Shif	10	3	0.91	0.27
Extract_CML_Image(class CML_Matrix<str	9	1	0.82	0.09
CML_Matrix<struct CML_element>::opera	8	8	0.73	0.73
_memmove	7	7	0.64	0.64
Grayscale_Pixel(struct CML_RGBA *)	7	4	0.64	0.36
__nh_malloc_dbg	5	0	0.46	0.00

From both figures, we can find this program spent most resource on the energy computation, which I didn't do any paralyzed revision.

Then I am working on further improvement.

Further Improvement

Here's the most computational expensive code:

```
for(int y = 1; y <= height; y++)
    for(int x = (min_x + boundry_min_x); x <= (max_x - boundry_max_x); x++)
    {
        (*Source)(x,y)->energy = min_of_three((*Source)(x-1,y-1)->energy,
            (*Source)(x,y-1)->energy,
            (*Source)(x+1,y-1)->energy)
            + (*Source)(x,y)->edge + (*Source)(x,y)->weight;
    }|
```

Since the communication in GPU is very expensive, the first thing for me is working on removing the dependencies.

Each line's energy is based on the (y-1)'s energy distribution and edge data.

The way to parallelize is:

1. Compute the first horizontal line's energy;

2. Split the image size to several strips, which have a width at Max(Max_width/threadNum, 50);
3. Every block would compute for one strip, and one thread per block.

P.S.: I had already pated some of the code in the program design section, so I just write the logic of parallelism here.

Here is the final result:

Category	700 * 466 => 500 * 466	1024 * 768 => 500 * 500	1024 * 768 => 500 * 500 (For #(thread) = 8)	1024 * 768 => 500 * 500 (For #(thread) = 16)
Sequential Application	5.812 sec	25.202 sec		
Parallel Application based on pthread [#(thread) = 4]	4.913 sec	24.241 sec		
GPU-Accelerated Application (without improvement on energy computing)	4.571 sec	22.221 sec	22.047 sec	22.575 sec
GPU-Accelerated Application (with improvement on energy computing)	2.262 sec	7.224sec		

GPU acceleration works! But it's not as perfect as my expectation.
At first I thought the time cost could be decreased to less than 1 sec since my GPU has 16 cores. I think it's because of the share memory usage in my program, which is not encouraged in GPU computing.

Future work

The future expectation could be concluded in following aspects:

- Further improve the performance of energy computing function. (Like communication method, share memory, and other refinement).
- Develop sample techniques to reduce computing cost.
- GPU invoke sometimes failed (Currently, the initialization of CUDA often not succeeds).
- Ambition: this technique could be applied to real-time video retargeting.
  - 24FPS, every photo should be processed in 0.0416s.
  - A reasonable resolution for Video should be 640*480.
  - Decoding the video format.

Lessons & Conclusion

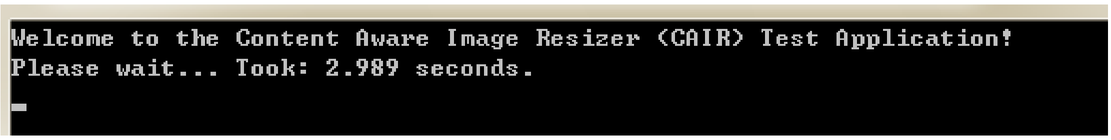
Here is my experience and achievement from this term project:

- Share Memory Problem is very important for image processing program on GPU.
    - Need to hold several memory for the original image, and temporary image processing result.
    - Communication between CPU and GPU is expensive.
- Parallelizing the sequential programs should begin with the profiling analysis.
    - Started from working on an existing code in Pthread, revised it to a CUDA program.
    - Found the performance problem until profiling time.
    - Rewrote the key function to CUDA code.

Code setup reference

My test environment is on WINDOWS 7 + VISUAL STUDIO + CUDA SDK for Windows.
The biggest problem of setup should be setup the CUDA development environment.
I followed a script here:
http://coitweb.uncc.edu/~abw/SIGCSE2011Workshop/ConfiguringVSforCUDA.pdf

Another problem may be occurred is that I also used pthread in my code. So you also need to setup a pthread environment.
Then just open the project file, and compile it. Finally it works.



Reference

1. Seam Carving for Content-Aware Image Resizing
http://www.win.tue.nl/~wstahw/2IV00/seamcarving.pdf