Implementation of parallel image processing using nvidia gpu framework computer s...

Technology, Computer

We introduced a real time Image Processing technique using modern programmable Graphic Processing Units in this paper. GPU is a SIMD (Single Instruction, Multiple Data) device that is inherently data-parallel. By utilizing NVIDIA’s new GPU Programming framework, “ Compute Unified Device Architecture” (CUDA) as a computational resource, we realize significant acceleration in the computations of different Image processing Algorithms. Here we present an efficient implementation of algorithms on the NVIDIA GPU. Specifically, we demonstrate the efficiency of our approach by a parallelization and optimization of the algorithm. In result we show time comparison between CPU and GPU implementations.

Most powerful CPUs having multi-core processing power are not capable to attain Real-time image processing. Increasing resolution of video captures devices and increased requirement for accuracy make it is harder to realize real-time performance. Recently, graphic processing units have evolved into an extremely powerful computational resource. For example, The NVIDIA GeForce GTX 280 is built on a 65nm process, with 240 processing cores running at 602 MHz, and 1GB of GDDR3 memory at 1. 1GHz running through a 512-bit memory bus. Its Peak processing power is 933 GFLOPS [1], billions of floating-point operations per second, in other words. As a comparison, the quad-core 3GHz Intel Xeon CPU operates roughly 96 GFLOPS [2]. The annual computation growth rate of GPUs is approximately up to 2. 3x. In contrast to this, that of CPUs is 1. 4x [2]. At the same time, GPU is becoming cheaper and cheaper.

As a result, there is strong desire to use GPUs as alternative computational platforms for acceleration of computational intensive tasks beyond the domain of graphics applications. To support this trend of GPGPU (General-Purpose Computing on GPUs) computation [3], graphics card vendors have provided programmable GPUs and high-level languages to allow developers to generate GPU-based applications.

In this paper we demonstrate a GPU-based implementation of pyramidal blending algorithm implemented on NVIDIA’s CUDA (Compute Unified Device Architecture). In Section 2, we describe the recent advances in GPU hardware and programming framework, we also discuss previous efforts on application acceleration using CUDA framework, and the use of GPUs in computer vision applications. In Section 3, we detail the implementation of the pyramidal blending algorithm. In Section 4, we made various design and optimization choices for GPU-based Implementation of the algorithm, then we demonstrate the efficiency of our approach by applying it to CUDA framework.

Background

The NVIDIA CUDA Programming Framework

Traditionally, general-purpose GPU programming was accomplished by using a shader-based framework [4]. The shader-based framework has several disadvantages. This framework has a steep learning curve that requires in-depth knowledge of specific rendering pipelines and graphics programming. Algorithms have to be mapped into vertex transformations or pixel

illuminations. Data have to be cast into texture maps and operated on like they are texture data. Because shader-based programming was originally intended for graphics processing, there is little programming support for control over data flow; and, unlike a CPU program, a shader-based program cannot have random memory access for writing data. There are limitations on the number of branches and loops a program can have. All of these limitations hindered the use of the GPU for general-purpose computing. NVIDIA released CUDA, a new GPU programming model, to assist developers in general-purpose computing in 2007 [3]. In the CUDA programming framework, the GPU is viewed as a compute device that is a co-processor to the CPU. The GPU has its own DRAM, referred to as device memory, and execute a very high number of threads in parallel. More precisely, data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads.

In order to organize threads running in parallel on the GPU, CUDA organizes them into logical blocks. Each block is mapped onto a multiprocessor in the GPU. All the threads in one block can be synchronized together and communicate with each other. Because there are a limited number of threads that a block can contain, these blocks are further organized into grids allowing for a larger number of threads to run concurrently as illustrated in Figure 1. Threads in different blocks cannot be synchronized, nor can they communicate even if they are in the same grid. All the threads in the same grid run the same GPU code.

Fig1. Thread and Block Structure of CUDA.

CUDA has several advantages over the shader-based model. Because CUDA is an extension of C, there is no longer a need to understand shader-based graphics APIs. This reduces the learning curve for most of C/C++ programmers. CUDA also supports the use of memory pointers, which enables random memory-read and write-access ability. In addition, the CUDA framework provides a controllable memory hierarchy which allows the program to access the cache (shared memory) between GPU processing cores a GPU global memory. As an example, the architecture of the GeForce 8 Series, the eighth generation of NVIDIA’s graphics cards, based on CUDA is shown in Fig 2.

Fig 2. GeForce 8 series GPU architecture

The GeForce 8 GPU is a collection of multiprocessors, each of which has 16 SIMD (Single Instruction, Multiple Data) processing cores. The SIMD processor architecture allows each processor in a multiprocessor to run the same instruction on different data, making it ideal for data-parallel computing. Each multiprocessor has a set of 32-bit registers per processors, 16KB of shared memory, 8KB of read-only constant cache, and 8KB of read-only texture cache. As depicted in Figure 2, shared memory and cache memory are on-chip. The global memory and texture memory that can be read from or written to by the CPU are also in the regions of device memory. The global and texture memory spaces are persistent across all the multiprocessors.

GPU Computation in Image Processing

Graphics Processing Units (GPUs) are high-performance many-core processors that can be used to accelerate a wide range of applications. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. In a personal computer, a GPU can be present on a video card, or it can be on the motherboard. More than 90% of new desktop and notebook computers have integrated GPUs, which are usually far less powerful than those on a dedicated video card. [1]

Most computer vision and image processing tasks perform the same computations on a number of pixels, which is a typical data-parallel operation. Thus, they can take advantage of SIMD architectures and be parallelized effectively on GPU. Several applications of GPU technology for vision have already been reported in the literature. De Neve et al. [5] implemented the inverse YCoCg-R colour transform by making use of pixel shader. To track a finger with a geometric template, Ohmer et al. constructed gradient vector field computation and canny edge extraction on a shader-based GPU which is capable of 30 frames per second performance. Sinha et al. [6] constructed a GPU-based Kanade-Lucas-Tomasi feature tracker maintaining 1000 tracked features on 800? 600 pixel images about 40 ms on NVIDIA GPUs. Although all these applications show real-time performance at intensive image processing calculations, they do not scale well on newer generation of graphics hardware including NVIDIA’ CUDA.

Pyramidal Blending

In Image Stitching application, once all of the input images are registered (align) with respect to each other, we need to decide how to produce the final stitched (mosaic) image. This involves selecting a final compositing surface (flat, cylindrical, spherical, etc.) and view (reference image). It also involves selecting which pixels contribute to the final composite and how to optimally blend these pixels to minimize visible seams, blur, and ghosting.

In this section we describe an attractive solution to this problem was developed by Burt and Adelson [7]. Instead of using a single transition width, a frequency adaptive width is used by creating a band-pass (Laplacian) pyramid and making the transition widths a function of the pyramid level. First, each warped image is converted into a band-pass (Laplacian) pyramid. Next, the masks associated with each source image are converted into a low pass (Gaussian) pyramid and used to perform a per-level feathered blend of the band-pass images. Finally, the composite image is reconstructed by interpolating and summing all of the pyramid levels (band-pass images).

3. 1 Basic Pyramid Operations

Gaussian Pyramid: A sequence of low-pass filtered images G0, G1 , . . ., GN can be obtained by repeatedly convolving a small weighting function with an image [7, 8]. With this technique, image sample density is also decreased with each iteration so that the bandwidth is reduced in uniform one-octave steps. Both sample density and resolution are decreased from level to level of the pyramid. For this reason, we shall call the local averaging process

which generates each pyramid level from its predecessor a REDUCE operation. Again, let G0 be the original image. Then for 0 < l < N:

G l = REDUCE [G l-1], which we mean

5

G l (I, j) = l? l? W (m, n) G l-1 (2i+m, 2j+n)

m , n= 1

Laplacian Pyramid: The Gaussian pyramid is a set of low-pass filtered images. In order to obtain the band-pass images required for the multi resolution blend we subtract each level of the pyramid from the next lowest level. Because these arrays differ in sample density, it is necessary to interpolate new samples between those of a given array before it is subtracted from the next lowest array. Interpolation can be achieved by reversing the REDUCE process. We shall call this an EXPAND operation. Let G l, k be the image obtained by expanding G l, k times. Then

G l, 0 = G l, which we mean,

2

G l, k (I, j) = 4 l? l? G l, k – 1 (2i+m/2, 2j+n/2)

m , n=-2

Here, only terms for which (2i + m)/2 and (2j + n)/2 are integers contribute to the sum. Note that Gl, 1 is the same size as Gl-1, and that Gl, l is the same

size as the original image. We now define a sequence of band-pass images L0, L1. . . LN. For 0 < l < N, L l = Gl – EXPAND [Gl+1] = Gl – G l+l, l. Because there is no higher level array to subtract from GN, we define LN = GN. Just as the value of each node in the Gaussian pyramid could have been obtained directly by convolving the weighting function W l with the image, each node of Ll can be obtained directly by convolving W l – Wl+1 with the image. This difference of Gaussian-like functions resembles the Laplacian operators commonly used in the image processing, so we refer to the sequence L0, L1aˆ¦. LN as the Laplacian pyramid.

Algorithm

Step 1: Build Laplacian pyramids LA and LB from images A and B.

Step 2: Build a Gaussian pyramid GR from selected region R. Step 3: Form a combined pyramid LS from LA and LB using nodes of GR as Weights. LS (i , j) = GR(i , j)*LA(i , j) + (1-GR(i , j))*LB(i , j)

Step 4: Collapse (by expanding and summing) the LS pyramid to get the final blended Image.

Proposed Implementation Details

In this section we describe various implementation strategies of the algorithm. We need to find possible parallelization in different functions of the algorithm. Pyramidal blending requires construction of Gaussian and Laplacian pyramid which are following the SIMD paradigm.

We set the execution configuration depending on size of shared memory of CUDA Memory hierarchy as it is the essential to execute Threads parallel. Number of blocks each multiprocessor can process depends on how many registers per thread and how much shared memory per block is required for a given kernel. Since shared memory is not used in the implementation with texture memory, we only need to be concerned about the number of registers used and we can maximize the size of block and grid as much as possible.

We set each thread process P data, P is the pixel value which required n = 4B, if image is in RGBA format. Ti represents any thread in a block, where i is the thread index. THREAD_N is the total number of threads in each block, BLOCK_N is the block number of each grid, N is the total size of the input data, n 16KB is the size of shared memory of the NVIDIA G80 series cards, so the execution configuration can be set below:

a) Ti processes P data; (THREAD_N *P)B <16KB;

b) BLOCK_N = N / (n*P).

It is desirable not to occupy the whole shared memory; some place should be remained to put some special variables. We describe various design strategies for various operations in pyramidal blending algorithm below

4. 1. Construction of Gaussian Pyramid

A sequence of low-pass filtered images G0, G1 , . . ., GN can be obtained by repeatedly convolving a small weighting function with an image. The

convolution operation is following SIMD paradigm. We apply following two functions in NVIDIA’s CUDA. We define proposed strategy for implementation.

CUDA Gaussian Blur

The first step is applying 5? 5 Gaussian blur filters. We take Gaussian constant equal to 1 In all cases of implementation, the kernel configuration is of 16A-16 threads of each block and 32 of blocks on 512? 512 pixel image. This kernel configuration is applied to each grid and there are total 32 grids of image size. The convolution is parallelized across the available computational threads where each thread computes the convolution result of its assigned pixels sequentially. Pixels are distributed evenly across the threads. All threads read data from share memory but due to limitation in shared memory data should be moved from global memory to shared memory. Synchronization of the threads can be done by CUDA Synchronized function Blocks. Which will do thread synchronization per block automatically to maintain results.

CUDA Reduce Operation

In this operation a sequence of low-pass filtered images G0, G1. . . GN can be obtained by repeatedly convolving a small weighting function with an image, which can be worked in grids. With this technique, image sample density is also decreased with each iteration so that the bandwidth is reduced in uniform one-octave steps we first need to reduce the image size by half at each level of pyramid. This implementation can be done in texture memory.

The texture memory is used to implement the function using OpenGL graphics library. Standard API will call to execute it in CUDA. Intermediate results of each level images will copied from shared memory to Global memory to implement REDUCE operation as defined in the previous section.

4. 2 Construction of Laplacian Pyramid

Expand Operation

Expand operation can be achieved by reversing the REDUCE process. This implementation can be done in texture memory. The texture memory is used to implement the function using OpenGL graphics library. Standard API will call to execute it in CUDA. Intermediate results of each level images will copied from shared memory to Global memory to implement EXPAND operation as defined in the previous section.

Laplacian of Gaussian

In order to obtain the band-pass images required for the pyramidal blend we subtract each level of the pyramid from the next lowest level. Because these arrays differ in sample density, it is necessary to interpolate new samples between those of a given array before it is subtracted from the next lowest array. Interpolation can be achieved by reversing the REDUCE process called EXPAND defined above. To implement Laplacian of Gaussian we follow SIMD paradigm. we will use the same thread configuration as we described before. Each thread need the result of Expand operation as described above for each pyramid level so we can get it from Global memory. Intermediate results can be copied from shred memory to Global Memory.

Results

In result we have shown pyramidal blending of two images With resolution of 1147 A- 608. figure 3a, 3b shows left image and right image respectively, figure 3c sows final blended panorama and figure 3d shows time comparison between CPU and GPU implementation.

(a)

(b)

(c)

Fig. 3. Pyramidal Blending (a) left image (b) right image

(c) Blended panorama

Pyramidal

Belding

CPU time(s)

GPU time(s)

Speed up

Combine operation

7. 18(s)

2. 30(s)

3. 13

Table 1. Time comparison

Conclusion

For parallel computing by CUDA, we should pay attention to two points. Allocating data for each thread is important. So if better allocation algorithms of the input data are found, the efficiency of the image algorithms would be improved. In addition, the memory bandwidth of host device is the bottleneck of the whole speed, so the quick read of input data is also very important and we should attach importance to it. Obviously, CUDA provides us with a novel massively data-parallel general computing method, and is cheaper in hardware implementation.