Parallel Image Processing Based on CUDA

Zhiyi Yang, Yating Zhu, Yong Pu
Dept. Computer of Northwestern Polytechnical University
Xi'an, Shaanxi, China
zhuyating02@163.com

Abstract—**CUDA (Compute Unified Device Architecture) is a novel technology of general-purpose computing on the GPU, which makes users develop general GPU (Graphics Processing Unit) programs easily. This paper analyzes the distinct features of CUDA GPU, summarizes the general program mode of CUDA. Furthermore, we implement several classical image processing algorithms by CUDA, such as histogram equalization, removing clouds, edge detection and DCT encode and decode etc., especially introduce the first two algorithms. If we don't take the data transfer time in experiment between host memory and device memory into account, as the image size increase, histogram computation can get a more than 40x speedup, removing clouds can get an about 79x speedup, DCT can gain around 8x and edge detection more than 200x.**

Keywords-CUDA; GPU Computing; Parallel Computing; Image Processing

I. INTRODUCTION

In recent years, the computation speed of graphics processing unit (GPU) has increased rapidly. We only take the float-point operation as an example, and GPU's computation speed is several times faster than CPU's. The Flops of NVIDIA Ge80 series has gotten 520G in late 2006 [1], whereas Intel 64-bit dual-core CPU has only 32 GFlops. Moreover, now the mainstream GPU's scale has exceeded significantly the CPU's. The transistor number of NVIDIA Quadro FX 5600 has been more than 0.7 billion. From the above, we can see the powerful computational capability of the GPU. Moreover, as the programmability and parallel processing emerge [3], GPU begins being used in some non-graphics applications, which is general-purpose computing on the GPU (GPGPU). However, the traditional GPGPU development is based on graphics function library, for example OpenGL and Direct 3D, which makes the GPU used only by the professional people familiar with graphics API, and brings many inconveniences to the common users.

The emergence of CUDA (Compute Unified Device Architecture) technology can meet the demand of GPGPU in some degree. CUDA brings the C-like development environment to programmers for the first time, which uses a C compiler to compile programs, and replaces the shader languages with C language and some CUDA extended libraries. Users needn't map programs into graphics APIs any more, so GPGPU program development becomes more flexible and efficient. More than one hundred processors resided in CUDA graphics card schedules hundreds of threads to run concurrently, resolving complex computing problems.

II. THE FEATURES OF CUDA GPU

Over the past few years, the performance of GPU has been improving at a much faster rate than the performance of CPUs. In 2007, NVIDIA's most advanced GPU provided six times the peak performance of Intel's most advanced CPU [4]. GPU has evolved from special-purpose processor to programmable processor, and meanwhile the programmability has been the most important feature. Compared to the previous GPU, CUDA GPU has the following advantages:

- General programming environment: CUDA uses C programming tools and C compiler, which make programs have better compatibility and portability.
- More powerful parallel computing capability: CUDA graphics card applied more transistors to computing, not to data cache or flow control [1]. GeForce 8800 has 128 1.35GHz stream processors, 512 bit DDR3, and 768M device memory, far bigger than the L1 cache of CPU.
- Higher bandwidth: Take the GeForce 8800 as an example, its bandwidth gets to 86.4GB/s between GPU and device memory, and 4GB/s between host memory and device memory via PCI-E x16 bus.
- Instruction operation: CUDA GPU supports integer and bit operation.

III. CUDA INTRODUCTION

The hardware architecture can be seen from Fig. 1. CUDA cards contain many SIMD stream multi-processors (SM), and each SM has also several stream processors. Each SM has four type memories, constant memory, texture memory and global memory can communicate with host memory except on-chip shared memory. These cards use on-chip memories and cache to accelerate memory access.

The hardware architecture has three novel features [5]:

- General write/read global memory: GPU can gather data from any location or the global memory, and also scatter data to any location, almost as flexible as CPU.
- On-chip shared memory: It can make threads in the same multi-processor get data quickly, avoiding accessing global memory frequently. The access speed of shared memory is as quick as registers, fetching a data from shared memory just costs 4 clock cycles, whereas from global memory need 400~600 clock cycles.

- Thread synchronization: Threads in a thread group can synchronize, so they can communicate and cooperate to resolve complex problems.

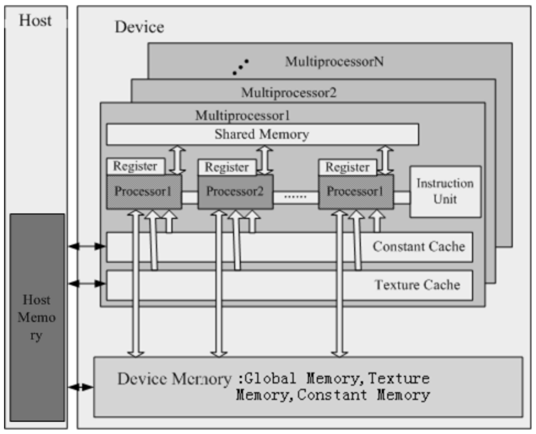


Figure 1. CUDA hardware model

In CUDA program, any call to a GPU function from within a CPU function must include an execution configuration [4], which determines the block count, thread count in a thread-block, and the account of shared memory. Each block can only run on a multi-processor, but each multi-processor can be allocated multiple blocks. Whether many blocks can run concurrently on a multi-processor depends on the occupancy of the registers and shared memory of each SM. At a time, a batch of blocks ready to run on a multi-processor are in active state, the threads of these blocks are divided into SIMD groups (called warps) by the order of thread index. Each multi-processor slices time to execute warps, each thread fetch different data to execute kernel according with its index.

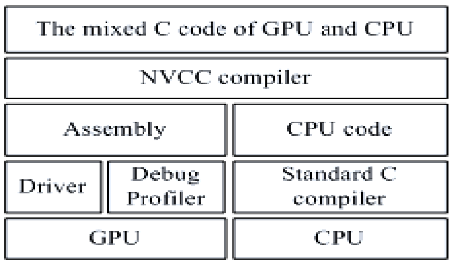


Figure 2. CUDA compiler procedure

For software, CUDA mainly extended the driver program and function library. The software stack consists of driver, runtime library, some APIs and NVCC compiler developed by NVIDIA. In Fig. 2, the integrated GPU and CPU program is compiled by NVCC, and then the GPU code and CPU code are separated.

IV. The General Programming Mode of CUDA

CUDA hides the low-level hardware details. For data-parallel computing, we can use the same mode to program with C language. The main steps of the general mode describes below:

- Copy data from host memory to device memory: Because of the limit of bandwidth, data transfer between host memory and device memory is the bottleneck that restricts the whole speed. So an effective method is to bind data to a texture, operating data by texture functions.
- CPU schedules the kernel to execute. This stage mainly contains the following three steps.
  a) Set the kernel execution configuration. Decompose the input data, and allocate data blocks to each thread block.
  b) Read data from global memory to shared memory [6]. This step is not essential but very effective. Because of the quick speed of shared memory, we should make the best use of shared memory. Here, if we adopt some optimization strategy, the read speed and memory bandwidth can be improved significantly. For example, access the global memory in a coalesced way and don't incur memory access conflicts when accessing the shared memory.
  c) Launch the kernel computation.
- Write the result back to host memory, do the post processing.

V. The Applications of CUDA in Image Processing

We usually process substantive pixel data in image processing, especially for airplane or satellite pictures. These images have large resolution and size, and the traditional processing methods can't satisfy the high real-time requirement. CUDA can provide highly data-parallel processing, so it is one of the ideal solutions of processing those big images.

A. *Histogram equalization*

For some image, such as remote sensing image, because the gray distribution is in a relatively narrow range, resulting in image details are not clear enough, and have lower contrast. Therefore, we need to do histogram equalization processing for these images. The process of histogram equalization contains three steps: 1) computing the original gray distribution; 2) computing the new distribution via cumulative distribution function; 3) do histogram transformation for the original distribution.

We designed the implementation process described detailedly as following.

1) Compute the original and new histogram distribution.

Firstly, set the execution configuration. We set each thread process L data, L is the gray level (like L=64,256), Ti represents any thread in a block, THREAD_N is the thread number of each block, BLOCK_N is the block number of each grid, N is the total size of the input data, 16KB is the size of shared memory of the NVIDIA G80 series cards, so the execution configuration can be set below:

a) Ti processes L data;

(THREAD_N *L)B<16KB;

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

If L=256, THREAD_N can only be 32.Because we can not occupy the whole shared memory, some place should be remained to put some special variables. We can find these variables by disassembling command *cubin* [8], which are block index, grid index and so on.

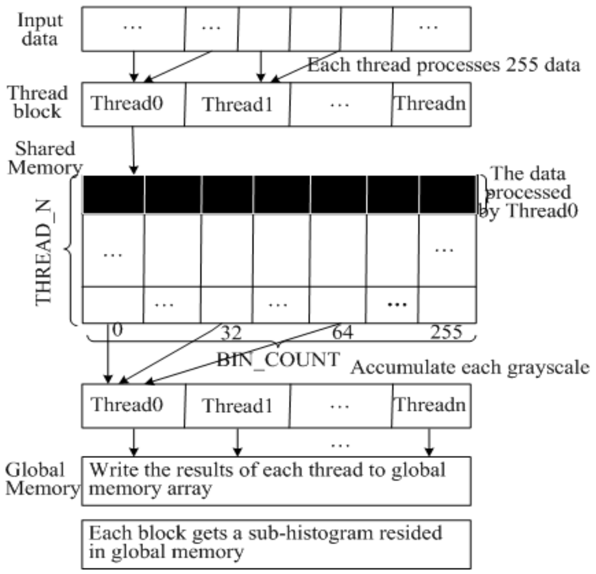

Input data
...
Each thread processes 255 data
Thread block
Thread0
Thread1
Threadn
Shared Memory
The data processed by Thread0
THREAD_N
0
32
64
255
BIN_COUNT
Accumulate each grayscale
Global Memory
Write the results of each thread to global memory array
Each block gets a sub-histogram resided in global memory

Figure 3. The computation of each thread block. BIN_COUNT represents gray level, equal to L. We set L=256, the thread number of each block is 32.When each block reduces thread sub-histograms, THREAD_N is smaller than L, so each thread accumulates more than one gray level, as the following order:Thread0: 0, 32, …,24 gray level; Thread1:1,33, … ,225 gray level ; ……Thread31:31, 63,…, 255 gray level.

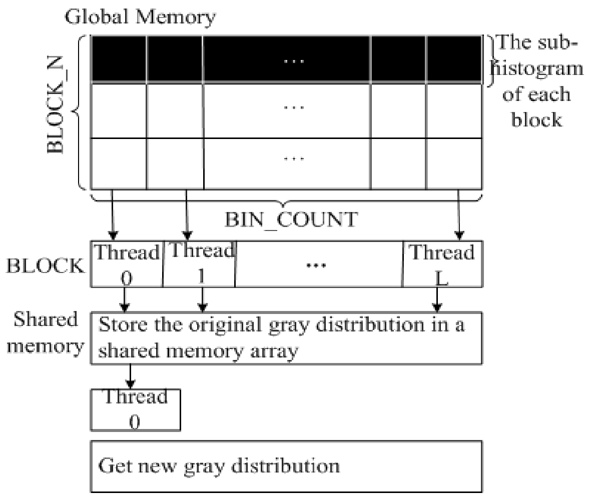

Global Memory
BLOCK_N
The sub-histogram of each block
BIN_COUNT
BLOCK
Thread 0
Thread 1
Thread L
Shared memory
Store the original gray distribution in a shared memory array
Thread 0
Get new gray distribution

Figure 4. Reduce sub-histograms of each thread blocks

Secondly, start computation of each thread block. Each thread process L data, and put the result into shared memory .Because of the size limit of shared memory, we can’t first copy data from global memory to shared memory, and each thread must read data from global memory. After threads finish computing, each thread has a sub-histogram that size is L. And then, each block need to reduce its sub-histograms to a bigger sub-histogram. This process is shown as Fig. 3.

Thirdly, reduce each block sub-histogram. At this step, we should reconfigure the kernel to set more threads in each block. Seen from Fig. 4, each block has L threads, we use the first block to reduce the all blocks in a grid, and each thread accumulates one gray level. Finally, use the first thread in the block computes the new gray distribution according with the theory of cumulative distribution function.

2) Histogram equalization. We set the thread number of each block is L, and each thread processes one gray level. At this step, the old gray value should be replaced by the new value.

B. More CUDA applications in image algorithms

We also do experiments on removing clouds, DCT encode and decode and edge detection algorithms.

In removing clouds algorithm, modulo 2 and logarithm operations could be parallelized. We set many threads so that each thread processes one pixel data. For Fourier Transform, CUDA provide a library of CUFFT which contains many highly optimized function interfaces, we can call these APIs simply. However a fly in the ointment, CUDA only supports the single-precision, resulting in the image contrast is not enough. We can see the result in Fig. 6.

In DCT encode and decode algorithm, we first divide the image data into N x N blocks, and then do the DCT transform for each data block. Because we adopt the quick DCT and IDCT based on the butterfly algorithm, FFT transform is needed. Before using the CUFFT library, users just need to create a Plan of FFT transform, and then call the APIs. For FFT transform, device memory is needed to be allocated when creating the Plan, and the device memory will not vary in succeeding computations. Therefore, image size needs to be smaller than device memory. For example, when device memory is 1.5GB and the image size is bigger than 6144 x 6144 pixels, an error will be occurred with the reason of deficient device memory.

C. Experiment results and analysis

For each image algorithm, we design both CPU serial code and GPU parallel code, and then compare the executed time. When calculating the executed time, we don’t consider the data transfer time from host memory to device memory.

The speed comparison of histogram is shown in TABLE I and Fig. 1 below. As the image size increases, GPU can improve the computing speed significantly. We can gain a more than 40x speedup.

An around 80x speedup can be gained in removing clouds algorithm showed in TABLE II. The processed images of removing clouds are shown in Fig. 6. We can seen two kind of different results, the CPU’s result has a higher contrast and the details of GPU’s is more clear. The reason of this difference is that GPU can only support the single-precision, whereas CPU supports 64-bit double float point type.

For DCT, we can gain an 8x speedup by GPU, and the

TABLE I. THE TIME COMPARISON OF 256 GRAY LEVEL HISTOGRAM EQUALIZATION

Input image	CPU(ms)	GPU(ms)
5120x5120	1746.40	37.29
4096x4096	1126.38	24.12
3072x2304	479.11	10.57
2048x2048	280.5	6.5
1600x1200	128.51	3.25
1280x1024	91.44	2.46
1024x816	54.6	1.79
512x512	17.53	0.93
320x408	8.83	0.97

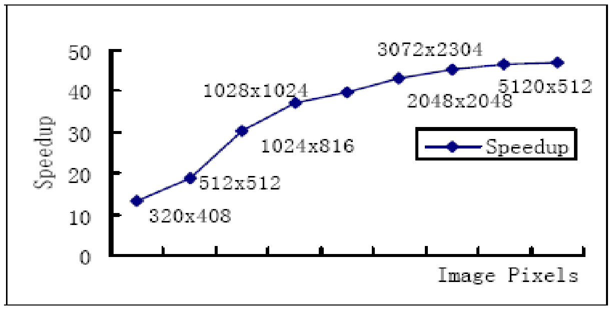
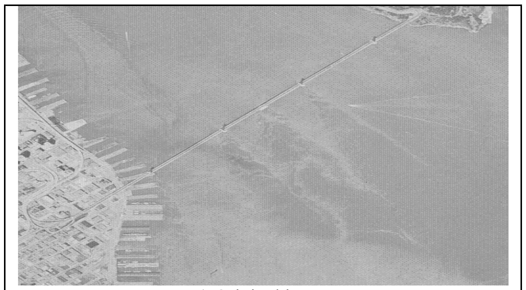
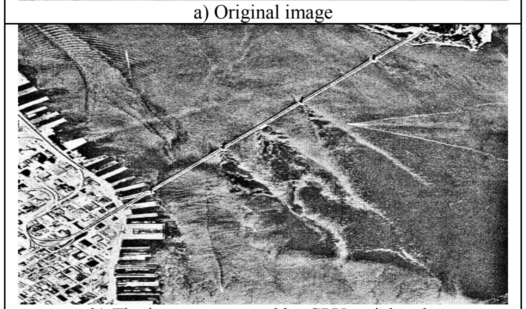


Figure 5. The speedup of histogram equalization by GPU. The values of the horizontal axis are marked by the side of the speedup graph. As the image size increases, the speedup is more obvious. When the image is 4096 by 4096, we can gain a 46x speedup.

TABLE II. THE TIME COMPARISON OF REMOVING CLOUDS

Input image	CPU(ms)	GPU(ms)	Speedup
4096x4096	31925.99	402.01	79.4
2048x2048	7644.09	97.09	78.7
1024x1024	1778.28	22.53	78.9
700x525	1745.02	21.98	79.4


a) Original image


b) The image processed by CPU serial code

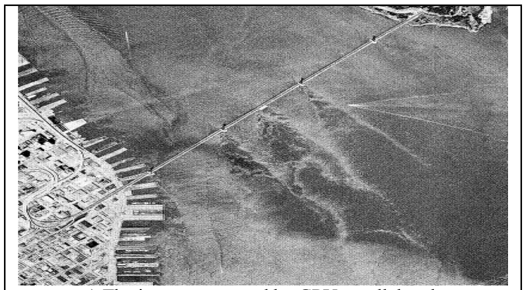

c) The image processed by GPU parallel code

Figure 6. The results of removing clouds

speedup doesn't increase with the image size. Especially for edge detection, we gain an over 200x speedup. For the speedup of edge detection, on the one hand, the reason is because we use the texture when reading data from host memory to device memory, which indicates that the way of reading data from host memory is very important for the whole speed. On the other hand, the whole computing process of edge detection is parallelized.

VI. CONCLUSIONS

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. In future, we will do more work on some optimization strategies and how to make the best of the memory spaces.

REFERENCES

[1] NVIDIA Corporation, CUDA Programming Guide 1.0, http://www.nvidia.com,2007.

[2] Tom R. Halfhil, "Parallel Processing With CUDA", Microprocessor Report, Scottsdale, Arizona, Jan 28, 2008.

[3] WU En Hua , "State of the Art and Future Challenge on General Purpose Computation by Graphics Processing Unit", Journal of Software, vol. 15, no. 10, 2004,pp.1493~1504.

[4] Michael Boyer, Kevin Skadron, Westley Weimer. "Automated Dynamic Analysis of CUDA Programs", STMCS 2008, Boston, Massachusetts, Apr 06, 2008.

[5] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, "Scan Primitives for GPU Computing", Graphics Hardware 2007, San Diego, California, August 04 - 05, 2007.

[6] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, et al.. "StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems", HPDC'08, June 23–27, 2008, Boston, Massachusetts, USA.

[7] John D. Owens, Mike Houston, David Luebke, et al., "GPU Computing", Proceedings of the IEEE, vol. 96, no. 5, May 2008 ,pp.879-897.

[8] NVIDIA CUDA. http://forums.nvidia.com,2007.