Numerical Parallel Processing Based on GPU with CUDA Architecture

Chengming Zou,Chunfen Xia,Guanghui Zhao
College of Computer Science and Technology
Wuhan University of Technology
Wuhan, China
E-mail: zoucm@hotmail.com

Abstract: **The characteristics of modern graphics processing unit (GPU) is programmable, high price / performance ratio and high speed . It has a strong ability to adapt the parallel calculation, Based on this, the article study the general method of GPU calculating and use compute unified device architecture (CUDA) to design new parallel algorithm to accelerate the matrix inversion and Binarization algorithm. The results show that with the increase of matrix dimension, GPU performs much better than CPU in increase multiple.**

Key Words: GPU, parallel algorithm, CUDA, matrix inversion, Binarization algorithm

I. INTRODUCTION

Parallel computing is the development trend of high-performance computing. The development of future processors will continue to focus on increasing the number of core rather than improving the dominant frequency. Highly parallel graphics processing unit (GPU)[1] is a kind of chip which has a good performance in the acceleration of 3D graphics. It is designed to handle single-instruction multiple data stream (SIMD) tasks and mainly used to accelerate the graphic scanning, image processing and 3D rendering. So it is applicable to the highly parallel task which has a demand for huge computing. In an early stage, GPU was designed for high-performance 3D graphics applications with strong computing power and high memory bandwidth, which are essential for high-performance 3D graphics applications. GPU has simple architecture, allowing high degree optimization design and large-scale floating point numbers parallel computing. Now a new type of GPU (e.g.: GeForce 8800) allows a certain degree user programmability, which makes the GPU can be suitable for more general calculation[2].

II. GPU GENERAL PURPOSE COMPUTING

A. GPU architecture

GPU is a highly parallel, multi-threaded and multi-core processor with high computing power and high memory bandwidth. Core 2 Extreme QX9650 CPU in Intel's Yorkfield has four cores, 8.2 million transistors; while C1060 GPU in NVIDIA's Tesla has 240 processing cores, 1.4 billion transistors. As the graphics rendering requires high-density, parallel computing, GPU, being not the same with CPU, will put the majority of the transistors into the data processing instead of the data catching and flow control (Fig.1), and this coincides with a number of scientific computing. Besides, GPU is particularly suitable for high-density parallel computing because its multi-stream processor could operate independently and concurrently at high speed when the processor solves problem, reducing the requirement of complex flow control and the computing complexity significantly. The memory bandwidth of G80 Ultra GPU reaches 103.7GB / s, being close to 10 times as broad as the CPU's bandwidth. Through driving multicore processor, broad memory bandwidth could provide strong computing power for high-performance computing. Multicore processor could carry out many data elements and have high computing density, therefore it could ignore the defer of memory access approximately and pave the way for the GPU's implementation in scientific computation[3]. Fig.2 and Fig.3 show the current comparison between GPU and CPU in bandwidth and floating point processing capability. High floating point computing power and high bandwidth make GPU, compared with CPU, have incomparable natural advantages in high-performance computing.

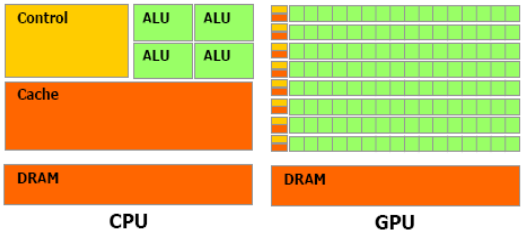

Figure 1. GPU has more transistors for data processing

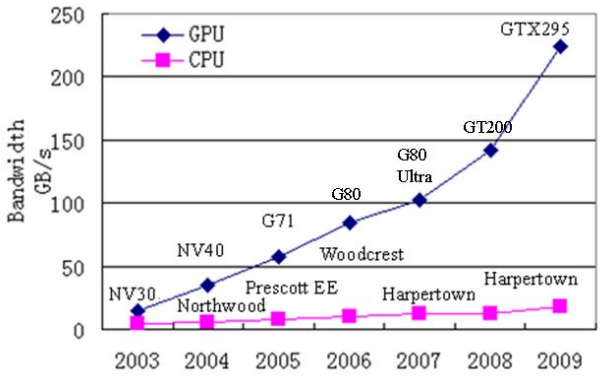

Figure 2: The Comparision of Memory Bandwidth between CPU and GPU

978-0-7695-3901-0/09 $26.00 © 2009 IEEE
DOI 10.1109/WNIS.2009.46

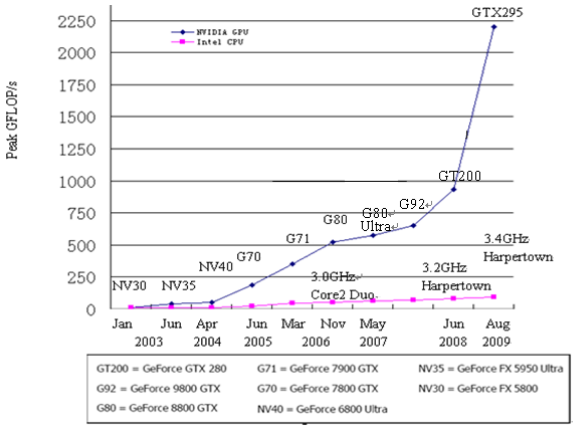

Figure 3: The Comparision of Floating Point Processing Performance between CPU and GPU

Parallel computing architecture inside GPU is designed based on two basic concept. First, the data of program can be divided into many parts, and a large number of cores can process these data parallel. The second assumption about framework is that data could not match with high-speed cache. For example, in the data processing of oil&gas data processing or graphical computing, the data size is likely to amount to megabytes or even terabytes. It is almost impractical to use high-speed cache to accommodate such a huge amount of data. Taking these two ideas into account, GPU is designed to use thousands of threads and all of these threads could execute parallel and could access local memory with huge volume[4].

GPU's high performance in general-purpose computing could not only save a lot of space, but also have higher performance per watt and then the cost of system operation and maintenance would be reduced greatly.

B. GPU computing applications

Oil / Gas / seismic data processing, financial risk modeling, medical imaging, finite element analysis, biological sequence matching, Monte Carlo simulation and molecular dynamics simulations and other applications require large-scale parallel computing power. GPU has 128 or even 240 processor cores, so it is very suitable for these areas.

III. COMPUTE UNIFIED DEVICE ARCHITECTURE

Compute Unified Device Architecture (CUDA) [5] is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without need of mapping them to a graphics API. When programmed through CUDA, the GPU is viewed as a compute device of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU, or host.

The CUDA software stack is composed of several layers as illustrated in Fig.4, a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries of common usage, Fast Fourier Transform 1D, 2D and 3D transforms of complex-valued signal data (CUFFT) and an implementation of basic linear algebra (CUBLAS). The hardware has been designed to support lightweight driver and runtime layers, resulting in high performance[6].

The batch of threads that executes a kernel is organized as a grid of thread blocks in a batch of threads that can cooperate together sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses efficiently[7]. Each thread is identified by its thread ID, which is the thread number within the block and it specify a block as a 2 or 3 dimensional array of arbitrary size. For a 2 dimensional block size (Dx;Dy) the thread ID of a thread of index (x; y) is (x + yDx) and for a 3 dimensional block size (Dx;Dy;Dz) the thread ID of a thread of index (x; y; z) is (x + yDx + zDxDy) (Fig. 5).

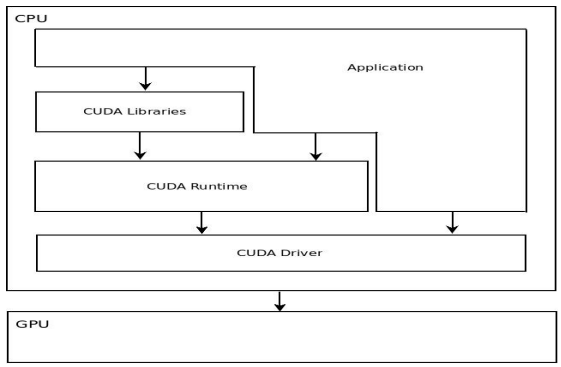

Figure 4. CUDA Stack

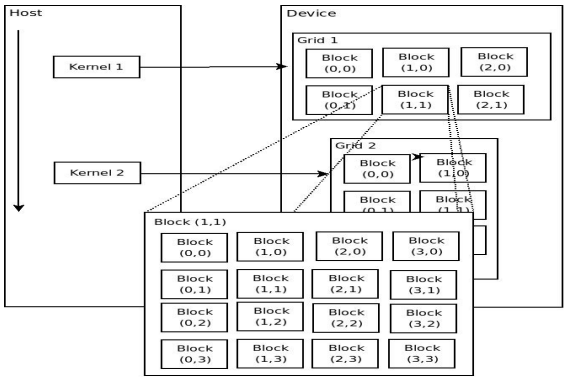

Figure 5. Thread Batching

A grid of thread blocks is executed on device by executing one or more block on each multiprocessor using time slicing. Each block is split into SIMD groups of threads called warps, each of these warps contains the same number of threads, called the warp size, and is executed by the multiprocessor in a SIMD fashion[8]; a thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources[9].

The CUDA API comprises an extension to the C programming and for the compilation of the CUDA source files it is used the NVCC, a compiler driver that simplifies the process of compiling. It provides simple and familiar command line options and executes them by invoking the collection of tools that implement the different compilation stages. NVCC's basic workflow consists in separating the device code from the host code and compiling the device code into a binary form or cubin[10]. The generated host code is output either as C code that is left to be compiled using another tool or as object code directly by invoking the host compiler during the last compilation stage[11].

IV. EXPERIMENTS

A. *matrix inverse algorithm*

1) Theorem:If $A = (a_{ij})_{n\times n}$, The algebraic cofactor of elements a_{ij} is A_{ij} , Adjoint matrix :

$$A^* = \begin{bmatrix} A_{11} & A_{21} & \cdots & A_{n1} \\ A_{12} & A_{22} & \cdots & A_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ A_{1n} & A_{2n} & \cdots & A_{nn} \end{bmatrix} = \mathrm{adj}A$$

,

when A be inverse, $A^{-1} = \frac{1}{|A|}A^*$ 。

2) The description of the algorithm

The method is presented in the article, the key steps of the calculation focus on the cofactor of elements A (mn) in k × k matrix. The process involves a large number of parallel computing, so the part of the parallel computing is given to GPU to deal with. Some of the code is as follows:

```
cudaMalloc((void**)&gpu_buffer,row*row*sizeof(float));
cudaMalloc((void**)&gpu_p,row*row*sizeof(float));
cudaMalloc((void**)&p_creat,(row-1)*(row-
1)*sizeof(float));
cudaMemcpy(gpu_buffer,buffer,row*row*sizeof(float),cuda
MemcpyHostToDevice);
cudaMalloc((void**)&gpu_row,sizeof(int));
cudaMemcpy(gpu_row,&rowTemp,sizeof(int),cudaMemcpy
HostToDevice);
Creat_gpu_M<<<1,row*row>>>(gpu_buffer,gpu_p,p_creat,p
_mid,gpu_row);
p=buffer + row * row;
cudaMemcpy(p,gpu_p,row*row*sizeof(float),cudaMe
mcpyDeviceToHost);
```

The function would configure the memory stored with matrix in the graphics memory and copy matrix data from main memory to the memory of graphics card[12].

Setting the kernel is as follows:

```
m=threadIdx.x;n = threadIdx.y;
    for(i=0;i<*k;i++)
            for(j=0;j<*k;j++)
            {
                    if(i!=m && j!=n)
                *p_mid++ = *(buffer+i*(*k)+j);
            }
```

B. *Binarization Algorithm for Image Processing*

1) Binarization Algorithm

Binary image processing means that the point of the gray image is set to be 0 or 255, which would make the whole image show evident black and white effect. Selecting appropriate threshold for 256 levels of gray-scale image intensity could acquire binary image which also could reflect the global and local features of the image. In digital image processing, binary image plays an important part. Especially in practical image processing, there are many systems based upon the binary image processing. This is helpful if we want to deal with the image further because the collection nature of image is only related with the location of the point whose pixel value is 0 or 255. In this way, the processing will be simpler and the amount of data which need to be processed and compressed will be smaller.

2) The description of the algorithm

Each pixel has its own thread and all the pixels compute parallel, without disturbing each other. Kernel code works in GPU while host code works in CPU. When kernel code is working, the data and result of original image will be stored in the memory of graphics card. Host code is in charge of the assignment of memory and copy of data.
Some of the code is as follows:

```
__global__ static void ChangeValue(BYTE* lpPoi)
{
    if(lpPoi[threadIdx.x]> (BYTE)threshold)
            lpPoi[threadIdx.x] = 255;
    else
            lpPoi[threadIdx.x] = 0;
}
```

copy data from main memory to the memory of graphics card:

```
cudaMalloc((void**)&lpCudaPoints,sizeof(BYTE)*nWidth*
Height);
cudaMemcpy(lpCudaPoints,lpPoints,sizeof(BYTE)*nWidth*
nHeight,cudaMemcpyHostToDevice);
ChangeValue<<<1,nWidth*nHeight,0>>>(lpCudaPoints);
cudaMemcpy(lpPoints,lpCudaPoints,sizeof(BYTE)*nWidth*
nHeight,cudaMemcpyDeviceToHost);
```

C. *The result*

The procedure experimental platform: Processor CPU for Pentium (R) Dual-Core 2.00GHz, 4GB RAM, Windows XP operating system. GPU is NVIDIA's GeForce GTX 260+, the core frequency 576MHz, memory frequency 1998MHz, Graphics memory width is 448bit. 216 stream processing units.

1) matrix inverse algorithm

The matrix randomly generated by the computer is used to test. Compare traditional algorithms efficiency of CPU and GPU , The results shown in Table Ⅰ:

TABLE I. CPU AND GPU ALGORITHM EFFICIENCY

Matrix Dimension	Data	CPU Time(ms)	GPU Time(ms)	Multiple
100	10000	7.734	0.516	15
110	12100	14.422	0.609	24
120	14400	20.422	0.672	31

2) Binarization Algorithm for Image Processing

The bitmap 5000*3750 after processing, as shown in the fig 6:

(1) (2)

Figure 6.(1)Original Image (2)Image after Processing

Compare traditional algorithms efficiency of CPU and GPU , The results shown in Table Ⅱ:

TABLE II. CPU AND GPU ALGORITHM EFFICIENCY

Image Size	CPU Time(ms)	GPU Time(ms)	Multiple
5000*3750	1.938	0.123	16
10000*7500	8.297	0.281	30

V. CONCLUSIONS

Based on GPU parallel computing, the article uses CUDA SDK to implement matrix inversion algorithm and binarization algorithm about image processing. The key of Algorithms is to mapped the major part of the parallel computing tasks to the GPU. It was found that the use of GPU for parallel computation algorithm could implement the algorithm with the help of graphics processing units' acceleration. Compared to CPU, it can increase the speed significantly.

REFERENCES

[1] Pawan Harish and P.J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. S. Aluru et al. (Eds.): HiPC 2007, LNCS 4873, pp. 197–208, 2007.

[2] Bader, D.A., Madduri, K.: Parallel algorithms for evaluating centrality indices in real-world networks. In: ICPP 2006. Proceedings of the 2006 International Conference on Parallel Processing,

[3] Hubert Nguyen. GPU Gems,vol.3. Addison Wesley Professional, 2007

[4] NVIDIA Corporation. CUDA Programming Guide Version 2.0. 2008

[5] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. NVIDIA, 2007.

[6] G. Poli, A. M. L. Levada, J. F. Mari, and J. H. Saito. Voice command recognition with dynamic time warping (dtw) using graphics processing units (gpu) with compute unified device architecture (cuda). SBAC-PAD International Symposium on Computer Architecture and High Performance Computing, pages 19–27, 2007.

[7] L. J. Ribeiro. Paralelizac¸ ˜ao da rede neural neocognitron em cluster smps. Dissertac¸ ˜ao de Mestrado da Universidade Federal de S˜ao Carlos (UFSCar)., 2002.

[8] T. Sim and M. Bsat. The cmu pose, illumination, and expression database. IEEE Transaction on Pattern Analysis and and Machine Intelligence, pages 1615–1618, 2003. 88

[9] J. Owens, "GPU architecture overview,."in SIGGRAPH '07: ACM SIGGRAPH 2007 courses, (New York,NY,USA),p.2,ACM, 2007.

[10] Hsi-Yu Schive,Chia-Hung Chiena,Shing-Kwong Wonga et ac.Graphic-Card Cluster for Astrophysics (GraCCA) ---Performance Tests.New Astronomy.2008,13(6):418-435

[11] Weiguo Liu, Bertil Schmidt, Gerrit Voss. GPU-ClustalW Using Graphics Hardware to Accelerate Multiple Sequence Alignment. HiPC 2006, LNCS 4297:363–374

[12] D. Luebke, M. Harris, N.Govindaraju, A..Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, "GPGPU: general-purpose computation on graphics hardware," in SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, (New York, NY, USA), p. 208, ACM, 2006.