TELKOMNIKA, Vol. 11, No. 8, August 2013, pp. 4379~4384
e-ISSN: 2087-278X

A Novel Architecture of Multi-GPU Computing Card

Sen Guo*[1,2], Sanfeng Chen[1,2], YongSheng Liang[1,2]
[1]ShenZhen Institute of Information Technology ShenZhen, China
[2]Shenzhen Key Laboratory of Visual Media Processing and Transmission, Shenzhen, Guangdong, 518172, China
Corresponding author, e-mail: ybbsss1210@126.com, ChenSf@sziit.com.cn, LiangYS@sziit.com.cn

Abstract
The data transmission between GPUS in the existing multi_GPU computing card is often through PCIE which is in relative low speed, so the PCIE has become bottleneck of Overall performance. A novel architecture of multi_GPU computing card have been proposed in this paper: A multi-channel memory which have multiple interfaces is added, including one common interface shared by different GPUs, which is connected with a FPGA arbitration circuit and several other interfaces connected with dedicated GPUs frame buffer independently, and this multi-channel memory is called "global shared memory". The result of a simulation of accelerating computer tomography algebraic reconstruction on multi-GPU demonstrates effectiveness of this approach.

Keywords: *multi-GPU, tomography algebraic reconstruction, GPGPU, CUDA*

1. Introduction

Computer graphics hardware has been widely used for general purpose computing in various applications, beyond the original target of computer graphics and gaming industry. The using computer graphics hardware to accelerate common computation can be tracked back to machines like the Ikonas [1], the Pixel Machine [2] and Pixel-Planes [3, 4]. In 1999, NVIDIA Corporation introduced Geforce 256, which was the first consumer-level card on the market with hardware-accelerated T&L (Transform & Lighting). After that, programable graphics pipeline was introduced and shading languages were become popular for GPU general purpose computing. In 2006, NVIDIA Corporation introduced Compute Unified Device (CUDA), as a flag of the arrival of modern GPGPU. Comparing with traditional GPGPU techniques, CUDA has several advantages, such as scattered reads, shared memory, faster downloads and readbacks to or from the GPU, and fully support for integer and bitwise operations. OpenCL, which is very similar with CUDA, was introduced in 2008 as the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices [5].

Recent years, a new term "personal supercomputer" emerges in the parallel computing industry. The personal supercomputers usually have one or more hardware accelerators (mostly GPUs). Having the advantages like portable, cost-effective and energy-efficient, the personal supercomputers are obtaining the favorite of engineers, scientists and computer experts. Furthermore, more and more professional industry software companies provide GPU acceleration for their products. With the power of GPU, these software usually can get 2x to more than 10x acceleration on the personal supercomputers, comparing to their corresponding CPU-only versions.

Multi-GPU graphics card like NVIDIA GTX 690 or AMD Radeon 7970, which has more than one GPU chips in a single graphics board, is also getting used in personal supercomputers. But these multi-GPU graphics cards have a disadvantage that the GPUs in the same board are not intra-connected. The intra-connection doesn't mean the electrical connection, but the data connection. Each GPU of this single board has different memory storage, and they cannot access other GPUs memory units directly. This issue will be detailed in next section.

In this paper, we propose a solution to bridge the different GPUs in the same board. And we analyze the performance improvement expectation if this solution is adopted by the

Received February 27, 2013; Revised May 12, 2013; Accepted May 22, 2013

graphics cards manufacturers. Because CUDA code is clearer to understand and easier to use than OpenCL, we use CUDA for the solution description and for the experiment implementation.

2. The Disadvantage of Current Multi-GPU graphics card design

As described above, in current design, the GPUs in one single multi-GPU board cannot communicate to each other directly. But in lots of cases, the programmers usually want to share some data between different GPUs [6]. Since the GPUs cannot communicate directly, to copy a buffer from GPU 1 to GPU 2, the program should copy the GPU 1's buffer to system memory through PCIE and then copy the buffer to GPU 2 through PCIE. Figure 1 gives a diagram of memory copy between different GPUs.

The PCIE bandwidth is relative much less than GPU access to its memory units [7]. According GPU bandwidth test utility provided by NVIDIA, the host-device download/upload usually has 2GB/s to 3GB/s bandwidth, while the GPU has more than 100GB/s bandwidth to their on-board memory storage. So, the data transmission through PCIE usually becomes a bottleneck for lots of applications running on multi-GPU systems.

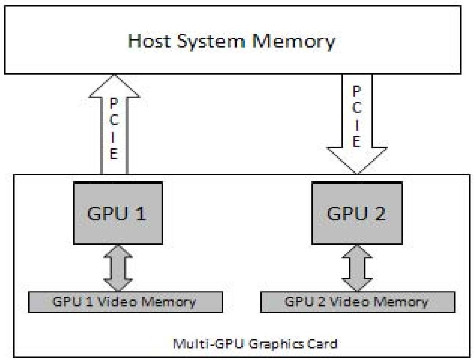

Figure 1. Data Flow of Memory Copy from GPU1 to GPU2

3. An Example: Computer Tomography Algebraic Reconstruction

Medical imaging is one of the domains where GPU parallel computing are wildly used. Generally the CT scanner generates large x-ray projection data, and then these projection data are reconstructed to a 3D volume, in which each voxel represents the density of the inspected object. The process of reconstructing the 3D data is called tomographic reconstruction.

There are several methods for CT reconstruction. Usually these methods can be divided into two categories: the analytic reconstruction methods (such like Comparing to the commonly used Filtered Back-Projection (FBP) algorithm [8]) and the algebraic reconstruction methods [9]. The algebraic reconstruction methods usually apply iterative reconstruction techniques, such as Simultaneous Algebraic Reconstruction Technique(SART) [10].

Comparing to more commonly used FBP method, SART usually performs better, especially when the set of available projections is sparsely or non-uniformly distributed in angles [11]. However, SART are rarely applied in most of medical CT systems due to their high complexity and high computational costs. For example, the SART requires a sequence of alternating volume projections and corrective back-projections until the reconstructed volume fits all projection images. This process is very time consuming and difficult to converge to a result instantaneously. In this paper, we will introduce the multi-GPU implementation of SART, and analyze the performance bottleneck of it, and how may the global shared memory improve it.

3.1. SART Algorithm

Given a $N = n^3$volume V , M projection images are obtained by the X-ray detector in M different angles. Let P_φ denotes the projection image in angle φ, and P denotes a vector storing all pixels of P_φ, $\varphi = 0, 1, ..., M$. Then the algebraic tomography reconstruction can be described as a linear algebra problem:

$$W V = P \tag{1}$$

Although the volume has three dimensions, it is flattened as a vector. Similarly although we should get M project images, and each projection image P_φ has two dimensions, the M projected images are also flattened as a vector. W is a Rn^3 weight matrix, in which w_{ij} denotes the influence factor that voxel $v_j \in V$ contributes its value to pixel $pi \in P$.

An iterative method is used to solve the equation. At the angle φ, a projection image P'_φ is computed. Then, each voxel is corrected by an accumulated correction according to all pixels in P'_φ. In the back-projection stage, voxels $v_j \in V$ are corrected by the following equation:

$$v_j^{k+1} = v_j^k + \lambda \frac{\sum_{i \in P_\phi} \frac{p_i - p_{i'}}{\sum_{n=1}^{N} w_{in}} w_{ij}}{\sum_{i \in P_\Phi} w_{ij}} \tag{2}$$

Where p_i is the projection pixel value, p_i is the integral acroses the ray, λ is a relaxation factor and $\sum_{n=1}^{N} w_{in}$ is the length of the line. This equation can be divided into following two equations:

$$\partial_i = \frac{p_i - p_i^{'}}{\sum_{n=1}^{N} w_{in}} \tag{3}$$

And:

$$v_j^{k+1} = v_j^k + \lambda \frac{\sum_{i \in P_\Phi} w_{ij} \partial_i}{\sum_{i \in P_\phi} w_{ij}} \tag{4}$$

From Equation 3 and Equation 4, we can find that each iteration in the SART algorithm can be divided into four main stages: projection, correction, back-projection and update.

1. Projection stage: Compute line integrals p'_l for all rays of $P_{\varphi.}$
2. Correction stage: Subtract the calculated line integral from projection p p_i in the projection image, and normalize it.
3. Back-projection stage: Distribute corrections onto voxels.
4. Update stage: Update the volume.

Equation 3 is computed in projection and correction steps, and Equation 4 is computed in back-projection and update steps.

Clearly that the most computational intensive processes of this algorithm are projection and back-projection. Projection and back-projection require integral computation acroses a ray or inside a voxel. In the other side, the correction and update processes are just matrix add/ substract operations, which are very fast. So the GPU acceleration should focus on the projection and back-projection processes.

3.2. SART on single GPU

Several papers have proposed GPU-based SART parallelization. [11] gives a detail CUDA implementation of SART with NVIDIA graphics cards. In this implementation, the user proposed two techniques: ray-driven projection and voxel-driven back-projection. Ray-driven, means each ray in the projection stage will be assigned a GPU thread to do the integral acroses this ray; voxel-driven, means each voxel in the back-projection will be assigned a GPU thread to handle its correction calculation. The correction step is computed in the ray-driven threads and the update step is computed in the voxel-driven threads.

These two techniques can be mapped onto GPU programming models trivially. In projection step, each ray is assigned a thread, and this thread just reads the data from the voxels it penetrates and calculates the integral. The computation of one ray doesn't rely on, and will not affect other rays' computation result. The voxel-driven back-projection step has the same situation, each voxel can be calculated and updated independently.

So the implementation is trial, just copy the data from host memory to the device memory, then setup and launch the kernel threads. We will not discuss the performance optimization on hardware. For detail, please see [11].

3.3. SART on Multi-GPU Graphics

To utilize more than one GPUs to implement SART. We should divide the entire thread grid to several parts and map them to different GPUs. Because the penetration of the rays of the corresponding thread parts with the inspect object volume is not cuboid, and is varying due to different angles, we should sync-up the whole volume data between different GPUs after the projection steps. For the same reason, We should sync-up the projection plane data (pi) after the back-projection steps.

Figure 2 shows how to map SART onto more than one GPUs. The problem comes from the data update in each iteration, especially the data volume update. Because the GPU thread kernels are very fast, the data transmission between two GPUs contributes a big part to the overall processing time.

In our experiments on a platform with a NVIDIA GTX690 and Intel i7 3770K, we use a 256×256×256 volume (each voxel is 16-bit) projected to a 512×512 plane for simulation. In each iteration, the projection kernel takes about 0.8 ms, the back-projection kernel takes about 18.5 ms, but the memory copies even takes near 20.0ms. In other way, each iteration takes about 40 ms.

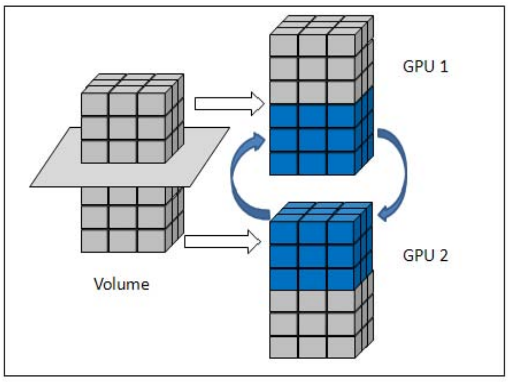

Figure 2. Map SART onto more than One GPUs

4. The Global Shared Memory Solution

To reduce the communication cost between GPUs on a single board, we have to establish a inner data connection between the GPUs. A simple solution is let each GPU can access other GPUs' memory, but this will raise two serious problems: cache consistency problem and read-after-write consistency problem. Cache consistency problem is caused by each GPU in the same board has an individual L1 cache, and the read-after-write problem is caused by that the order of memory access is not guaranteed, especially considering the fact that the latency of access video memory from GPU memory controller is very high (usually hundreds GPU clocks).

Here we propose a design idea to address this problem. We call this an idea because we haven't really built a multi-GPU board according to this design. We can only present this idea's concept and analyze it theoretically. The design is to add a multi-channel memory to the multi-GPU board, and this memory is only for transferring data between different GPUs. This multi-channel memory should have multiple interfaces, including one common interface shared by different GPUs, which is connected with a FPGA arbitration circuit and several other interfaces connected with dedicated GPUs frame buffer independently. We can call it "Shared Memory", but to distinguish the CUDA Shared Memory of a stream multiprocessor (SM), we call this memory Global Shared Memory. The "Global" means it is not only shared by he processors in a SM, but shared by all the GPUs. Figure 3 gives a diagram of this design:

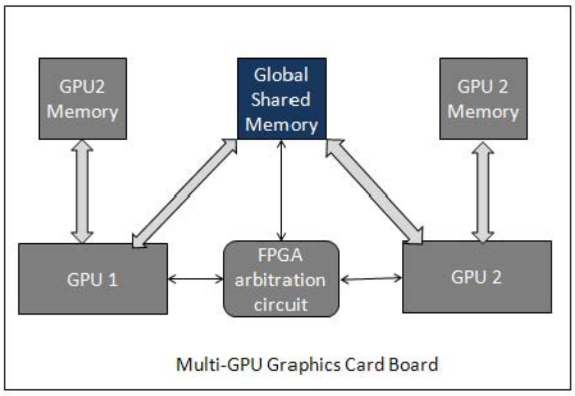

Figure 3. Diagram of Global Shared Memory Design

The FPGA arbitration circuit controls the access of this Global Shared Memory (GSM), and make sure only one GPU can access the GSM in one moment. It realizes the lock () and unlock() functionality for the GSM. The access control to GSM should be very fast for low-latency GSM read/write access response. This is why we use on-board FPGA, not CPU to control GSM, because CPU control signals should go through PCIE to each GPU and the same for the GPU's responses, which will result high latency.

With this GSM, a memory copy from GPU 1 to GPU 2 in a dual-GPU system can be described as follow:

1. GPU1 requests to lock the global shared memory, wait until success.
2. GPU1 reads its local device memory and write to global shared memory.
3. GPU1 unlocks the global shared memory if it's full or the copy is finished.
4. GPU2 requests to lock the global shared memory, wait until success.
5. GPU2 reads the global shared memory and write to its device memory.
6. GPU2 unlocks the global shared memory if all in global shared memory is read to local device memory or the copy is finished.

5. Experiments and Conclusion

Because we don't have a real graphics card with this design, we only evaluate the performance improvement brought by this design in a common multi-GPU graphics cards. In our experiments, we used a dual-GPU NVIDIA Geforce GTX 690. We allocate a 256MB buffer in both GPU1 memory and GPU2 memory to simulate a 256MB Global Shared Memory. And to simulate the real operation of copy the data from GPU1 to GPU2 through GSM, we first let GPU1 copy the data in GPU1's video memory to the 256MB buffer in GPU1, after GPU1's copy is finished, let GPU2 copy the data with same size from the 256MB in GPU2 to the destination buffer of GPU2. Note that this process doesn't perform a real copy from GPU1 to GPU2,it just simulates the performance of this copy. So the output will be incorrect. However this will not affect our evaluation for the performance.

The real memory copy from GPU1 to GPU2 with GSM should have some performance difference with our experiments, but the difference will be not big. Theoretically analyzed, we can find that the speed of memory copy with GSM is determined by the memory clock, memory interface width and the latency of FPGA controller. Because FPGA controller and the two GPUs are directly inter-connected on the board, we can image that the latency will be very low. The FPGA controller latency should contribute little to the overall memory copy time when the data to be copied have relative big size.

Again in our experiment on a platform with a NVIDIA GTX690 and Intel i7 3770K, with a 256×256×256 volume (each voxel is 16-bit) projected to a 512×512 plane for simulation. In each iteration, while the kernels' used time remains the same, the time used by data transmission has been reduced from 20.0 ms to nearly 1.0 ms. And the overall used time is reduced from nearly 40.0 ms to nearly 20.0 ms. Almost half time can be saved. Wecan find that the memory copy between GPUs will no longer be the bottleneck for overall performance.

Acknowledgements

This research was financially supported by ShenZhen International cooperation project of science and technology research (NO: ZYA201007070116A), and Science and Technology Planning Project of ShenZhen (NO.JC200903180648A).

References

[1] D Luebke, G Humphreys. How GPUs work?. *Computer*. 2007; 40(2): 96-100.
[2] Pat Hanrahan. *Why are Graphics Systems so Fast?.* Proceeding of 18th International Conference on Parallel Architectures and Compilation Techniques. NewYork. 2009; 9: 34-36.
[3] William Mark. Future Graphics Architectures. ACM Queue. 2008; 23(3): 56-64.
[4] J Nickolls, WJ Dally. The gpu computing era. *IEEE Transactions on Microsoft*. 2010; 30(2): 56-69.
[5] Kayvon Fatahalian, Mike Houston. A Closer Look at GPUs. *Communications of the ACM*. 2008; 51(10): 50-57
[6] Benjamin Block, Peter Virnau, Tobias Preis. Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising model. *Computer Physics Communications*. 2010; 181(4): 1549-1556.
[7] Enos JJ, Guochun Shi, Showerman. *GPU clusters for high-performance computing*. Proceeding of Cluster Computing and Workshops. Beijing. 2009; 4: 1-8.
[8] Gordon R, Bender R, Herman GT. Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and x-ray photography. 1970; 29(3): 471-481.
[9] AH Andersen, AC Kak. Simultaneous Algebraic Reconstruction Technique (SART): A superior implementation of the ART algorithm. *Ultrasonic Imaging*. 1984; 6(1): 81-94.
[10] Avinash C Kak, Malcolm Slaney. Principles of computerized tomographic imaging. *Classics in Applied Mathematics*. 2001; 33(1): 329-332.
[11] Yuqiang Lu. *Accelerating Algebraic Reconstruction Using CUDA-Enabled GPU*. Proceeding of Computer Graphics, Imaging and Visualization. HongKong. 2009; 11: 480-485.