International Journal of Hybrid Information Technology
Vol.8, No.7 (2015), pp.253-266
http://dx.doi.org/10.14257/ijhit.2015.8.7.24

Multi-GPU Parallel Computing and Task Scheduling under Virtualization

Yujie Zhang, Jiabin Yuan, Xiangwen Lu and Xingfang Zhao

College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China
shuangyujie@qq.com, jbyuan@nuaa.edu.cn

Abstract

General Purpose Graphics Units (GPGPUS) have seen a tremendous rise in scientific computing application. To fully utilize the powerful parallel computing ability of GPU, and combine the isolation characteristic of virtualization, a GPU virtualization method that supports dynamic scheduling and multi-user concurrency is proposed. For multi-task of GPU general computing programs in virtualization environment, the existing GPU scheduling algorithms have been improved for achieving a more fine-grained and more accurate load evaluation. For large-scale computing programs, we present a method for multi-GPU collaborative computing in virtualization environment, which can effectively deals with accelerating the large-scale program on multi-GPU within a single node. In the experiments, we make verifications by using the representative scientific computing examples, such as classical matrix calculation and discrete Fourier transformation. The experimental results prove that with the increasing of the calculation scale, the speedup can go up and finally close to the numbers of GPU.

Keywords: *general-purpose computation on GPU, virtualization, CUDA, load balancing, OpenMP*

1. Introduction

Virtualization, as a technology that enables easy and effective resource sharing with a low cost and energy footprint, is becoming increasingly popular not only in enterprises but also in high performance computing. Applications with stringent performance often need to make use of graphics processors for accelerating their computations. It can effectively solve the adaptation problem of general-purpose computation under the virtualization platform by using interior features of virtualization platform and general computing API virtualization technologies. This provides effective technical supports for the applications in high-performance computing and cloud computing by using GPU and virtualization technology [1].

For the current academic research, GPU virtualization in general-purpose computation field has raised a number of possible options. However, the functions in most scenarios are too simplistic. The key point of these researches is fixed on the user transparency and data transmission optimization between virtual machines [2]. At the same time, most solutions are designed specifically to process data on single-GPU environment. As a result, they cannot take advantage of multi-GPU to accelerate large-scale computing program in single-node multi-GPU environment.

In order to support parallel processing of multi-task by using GPU general computing in virtualization environment better, in this paper a GPU scheduling algorithm based on the existing option is optimized. In the proposed algorithm, we firstly add a GPU registration center between the client and server, which is responsible for task scheduling, so as to achieve the purpose of concurrent execution of multi-tasks. Additionally, in order to reduce the turnaround time for multi-tasks scheduling, we involve the loaded evaluation results for

ISSN: 1738-9968 IJHIT
Copyright ⓒ 2015 SERSC

scheduling. For the collaborative computing, many academic researchers focus only on the CPU and GPU collaborative computing [3]. There is seldom involved in collaborative computing for multi-GPU environment. This article introduces OpenMP [4] into GPU, and design multi-GPU collaborative computing method in the virtualization environment. We use multi-GPU to accelerate large-scale computing programs.

The rest of this paper is organized as follows. Section 2 briefly describes the related technology. Section 3 discusses the improvements of GPU scheduling algorithm for multi-tasking in detail. The multi-GPU collaborative computing methods for single-task are given in Section 4. Section 5 then gives the experimental platform configuration and comparative analysis of the experimental results, followed by our conclusions in Section 6.

2. Related Technology

2.1. GPU and GPU Virtualization

GPU (Graphics Processing Unit) is one of the important modern computer equipments, which is originally designed for image processing and 3D rendering [5]. GPU for areas outside of the graphics rendering is called GPGPU [6]. Nowadays, as the complexity of diversity of I/O peripherals, I/O virtualization has always been a major bottleneck in VMM (Virtual Machine Monitor) [7]. GPU belongs to special equipment in I/O devices, thus academic circles one after another are devoted to the development of GPU virtualization technology.

In general computing, GPU virtualization is mainly based on Compute Unified Device Architecture (CUDA) solutions, such as vCUDA [8], Gvim [9], gVirtuS [10], rCUDA [11], and so on. vCUDA makes it possible to use hardware GPU to run CUDA programs in virtual machines. Gvim uses Xen-specific mechanisms that allow virtual machines to access GPU device via a communication channel between the virtual machine and the privileged domain, thus providing the support for virtual machine to access GPU. gVirtuS utilize a similar mechanism as vCUDA and Gvim. All of them use the front end/back end communication solution. Virtual machines are front end and privileged domain is back end. rCUDA is designed to communicate a GPU-accelerated process running in a computer not having a GPU with a remote host providing GPGPU services. The communication between the server and the client uses the socket rather than Remote Procedure Call Protocol (RPC). As the result, its efficiency has relatively improved compared with vCUDA.

2.2. Parallel Computing Framework OpenMP

OpenMP (Open Multi-Processing) is a technology that belongs to the shared memory programming model. Programmers add compiler-guided instruction (#pragma) to indicate the properties of concurrent programs in the source code. When you choose to ignore these #pragma, or the compiler does not support OpenMP, the program can be reduced to a serial program. Most of the codes can still be normal and correct except that it cannot use multi-threading to speed up program. Since OpenMP is based on guidance, it is simple, portable, and highly scalable and it supports incremental parallelization development. All above features make it as the parallel programming standard of shared storage system [12]. OpenMP supports C, C++, Fortran, and other mainstream programming languages. The commercial compiler which supports OpenMP covers Microsoft Visual Studio and Intel Compiler, etc. [13].

3. Scheduling Algorithm

In this section, a new method of scheduling algorithm oriented GPU for multi-task has

Copyright ⓒ 2015 SERSC

been proposed.

Literature [14] has proposed a scheduling algorithm based on GPU features. It introduces the ability and the global memory of GPU into the load evaluation. At the same time the scale of the task is also considered. But its influence factor of GPU computing power and global memory are both set to a fixed 0.5, which is not consistent with the requirements of the tasks in computing power and global memory in the actual environment. In addition, the type of the task has not been considered, which results in the same scale but different computational complexity. Therefore this scheduling algorithm is simple, and there is many differences with in the actual environment.

In this paper a more fine-grained load evaluation method based on this scheduling algorithm has been proposed. By introducing calculating complexity and time complexity of the task into load evaluation, we make feedback adjustment on the influence factor of the computing capability and global memory of GPU, so as to achieve a more fine-grained load evaluation. The scheduling algorithm based on the GPU resources is finished by the GPU registration center which suggested in this paper. It uses client and server components based on gVirtuS, and submits client tasks to the GPU registration center, and then schedules by the GPU registration center according to the workload. The following is the detailed description of the scheduling algorithm.

3.1. Overall Framework

Considering the system is based on software architecture, the basic objects of the system should have the correctness, usability, efficiency. Correctness means the software should attach the expected function degree. Usability means the basic structure of the software and implementation to attain the extent of the users. Efficiency generally describes the extent to which time, effort or cost is well used for the intended task or purpose. The principles of the system are as follows: adopt right development paradigm, use good design method, and provide high quality engineering support. The software architecture is shown in Figure 1.

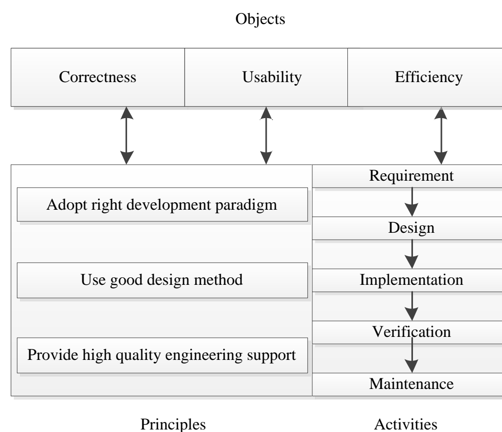

Figure 1. Software Architecture.

The overall framework of GPU virtualization uses gVirtuS as a reference. It is composed of a server middleware and a client middleware. The GPU registration center middleware has been added on this basis. The overall framework is shown in Figure 2. The whole system can be divided into modules in horizontal way and vertical way. The horizontal can be divided into the application layer and the system layer, and the vertical can be divided into the physical machine and the virtual machine. The system layer in physical machine has several GPUs. These GPUs will provide their own resource capacity information to the GPU registration center. The GPU registration center responses to the

Copyright ⓒ 2015 SERSC

client requests and return the lightest workload of GPU information to the client.

In the framework of the whole system, the CUDA client component is in a virtual machine application layer. The server component is located in the application layer of the physical machine. The GPU registration center can be located in any physical machine or virtual machine that can communicate with the client and the server. This article will set it in another virtual machine which is shared in the same VMM with the client.

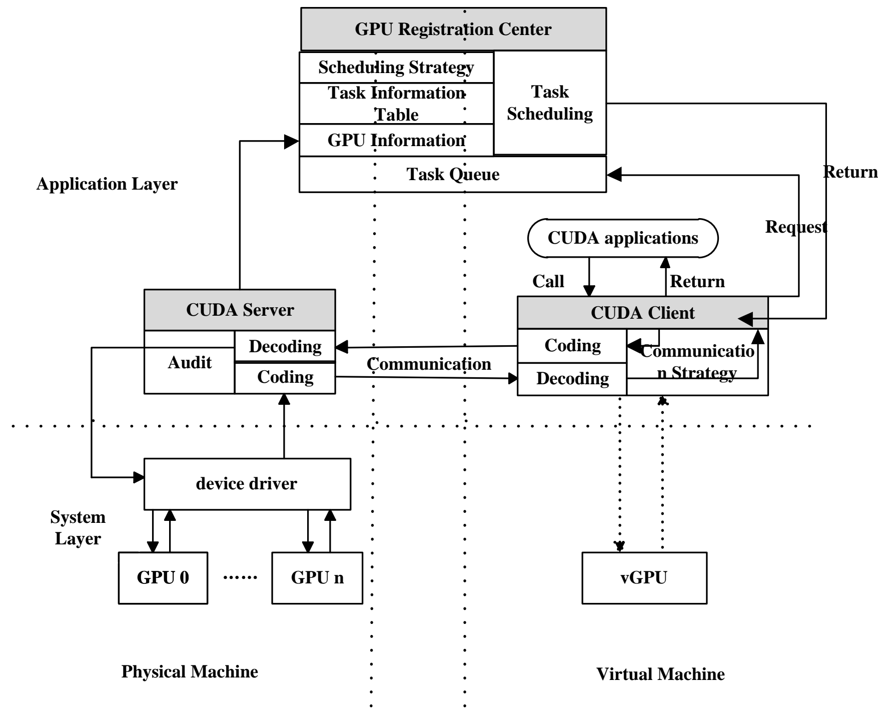

Figure 2. System Framework

CUDA client is oriented with CUDA applications. It accomplishes the main functions as follows. 1) Intercept all the CUDA function calls made by all applications. 2) Request for GPU resources from GPU registration center and obtain the IP address and device number of the physical machine which contains the GPUs. 3) Choose the communication strategy according to the positional relationship between itself with CUDA service side. 4) Pack, encapsulate and code the interface and the parameter of CUDA API. 5) Decode the returned data from CUDA server and return it to the caller.

CUDA server component is located in the application layer -with host physical machine. Host physical machine can interact directly with the hardware, so the server-side components can accomplish the general computing tasks by using physical GPUs. CUDA server is oriented to physical GPUs. It accomplishes the main functions as follows. 1) Submit GPU information and GPU load information to the GPU registration center. 2) Receive datagrams from the CUDA client and resolve the parameters and calls. 3) Audit the parameters and calls. 4) Calculate the program by using local CUDA that calls local GPUs. 5) Encode the results and return to the CUDA client. 6) Manage the GPUs that support CUDA in the system.

GPU registration center can be located anywhere in the physical machine or the virtual machine that can communicate with the client and the server. It will isolate and manage the GPU resources on a logical level. It makes the unified management for GPU resources and accomplishes the main functions as follows. 1) Respond to the registration request in the GPU client and maintain all the registered GPU load informations.2) Respond the GPU informations from the server side and update the GPU load evaluation value.3)

 Copyright ⓒ 2015 SERSC

Respond the GPU resource requests from the client side and return the IP and device number of the physical machine with the smallest load evaluation value to the client. 4) Regulate the task type by using feedback algorithm and find the optimal configuration of GPU computing power and the impact factors of global memory.

3.2. Basic Elements

The GPU scheduling algorithm in this paper is completed by the GPU registration center. According to the literature [10], the speed of GPU computing depends on the number of Streaming Multiprocessor(SM) instead of the block numbers or thread numbers. Operational efficiency is roughly proportional to the product of the number of cores and the clock frequency of GPU. Besides, the size of GPU global memory also has a great impact on computing efficiency. If the calculation scale is greater than the size of GPU global memory, the calculation cannot be completed in one time by GPU, thus it will induce additional communication overhead. As the result, the registration information should primarily contain the number of processing cores, clock frequencies as well as global memory. Besides, the registration center takes other elements into consideration including the scale of the task, computational and time complexity. It is possible to schedule the GPU resource according to these elements.

In this paper, we design a comprehensive evaluation index I to evaluate the workload of load for all GPUs in the server: $I = \sum_{i=1}^{N} \frac{Scale_i * cmplx_i}{\alpha_i * \mathrm{P} * \mathrm{R} + \beta_i * \mathrm{G}}$.

In the formulation, the scale of task $Scale_i$ is provided by the CUDA applications as the interface parameter. The GPU registration center maintains a job information table. Different types of tasks are corresponding to different complexities as well as the corresponding value α_i and β_i. $cmplx_i$ is decided by the task types provided by the client. We set $cmplx_i$ as product of the time complexity of the task and the reciprocal of the task numbers that one processor completes in a clock cycle. For example, the task types are defined as 1 in single-precision floating-point add, multiply and multiply-add. The number of operations is 8 times in per clock cycle and the corresponding complexity is 1/8. The task type, which is of single precision floating-point is defined as 2, and the number of operations done per clock cycle is 2 with the complexity of 1/2 [15, 16]. P, R, G (P represents the number of GPU kernels, R represents GPU clock frequency, G represents the GPU global memory) are provided by the registration information of the server and maintained by the GPU registration center.

3.3. Algorithm Workflow

The GPU registration center returns the smallest value of I based on the scheduling principles. The smaller value of *I* means the current load of GPU is light. Besides, another basic principle of task scheduling is locally priority processing. In this way it can greatly reduce the performance overhead caused by the transmission. When the CUDA server submits information to the GPU registration center, it not only needs to send its own IP addresses and device numbers, but also needs to give the information of GPU, such as the frequency, number of cores, global memory, etc. When the CUDA client requests service from the registration center, the registration center first updates the comprehensive load evaluation of the service side. Then the local server load evaluation is multiplied by a weight factor. Next add it into the items and sort them. Then get the GPU device with the lowest workload lowest load and assign it to the CUDA client. The size of the weight coefficient can control the local service side priority. The smaller of the weight coefficient, the larger possibility of services provide by the local server. The weight coefficient can be considered as the ratio that the priority of local service side and its own load.

Figure 3 shows the dispatched flow chart of the registration center.

Copyright ⓒ 2015 SERSC

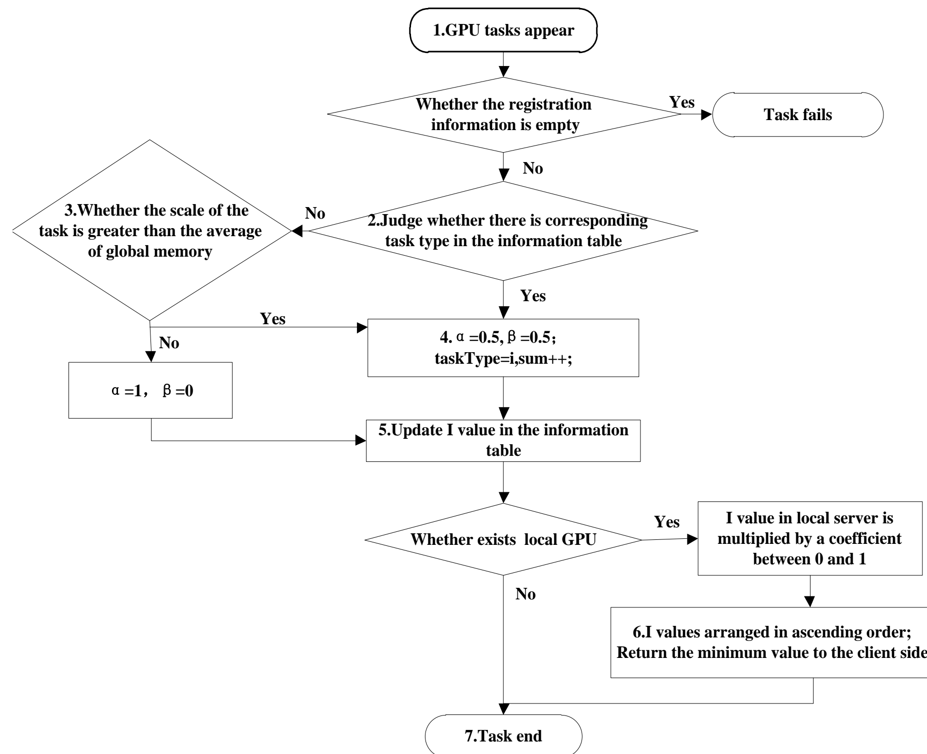

Figure 3. Scheduling Process in GPU Registration Center

1. GPU tasks appear. Determine whether the registration information is empty. If it is empty, the GPU resource is not currently available, then the task fails, otherwise turn 2.
2. Judge whether there is corresponding task type in the information table. If there is that means there is the corresponding value α and β , then enter 4, otherwise enter 3.
3. If the scale of the task is greater than the average of global memory, then enter 4, otherwise set α =1, β =0, and enter 5.
4. Both α and β are set 0.5. Record task types. Plus the global variable sum, pend feedback regulation, enter 5.
5. Update the value of I in the information table. If there is a local GPU, then it is multiplied by a coefficient between 0 and 1, enter 6. Otherwise, step 7.
6. Arrange the value of I in ascending order. Return the minimum value to the client side.
7. End.

4. Multi-GPU Coordination Computing

This section motivates multi-GPU coordination computing method for single-task oriented. At present CUDA does not provide APIs for multi-GPU environment. Academic researchers against multi-GPU coordination computing just focus on physical machines. There is little research on multi-GPU collaborative computing method in the virtualization environment. This paper first introduces the workflow of multi-GPU coordination computing in the virtualization environment. In addition, data decomposition (i.e., the division of tasks), data calculation and data consolidation in multi-GPU collaborative computing will be elaborated in detail. The main factors affecting the performance of multi-GPU collaborative computing will be got through theoretical analysis and experimental results.

 Copyright ⓒ 2015 SERSC

4.1. Data Decomposition

Data decomposition is a process that decomposes a large-scale computer program into multiple small tasks, and delivers them to multiple GPUs to execute in parallel. For the GPU devices with the same computing power, it is the best and most efficient allocation method that equally allocates the data to each GPU. Take a single node which has four GPU devices for example. In this section, four threads will be opened in the host on the use of OpenMP. Each thread controls one GPU device. It can maintain computing synchronization in OpenMP compiler-guided region and controls four GPU collaborative computing. Finally, experimental verification of multi-GPU collaborative computing technology will be made in virtualization environment with high computational complexity computing such as discrete Fourier transformation.

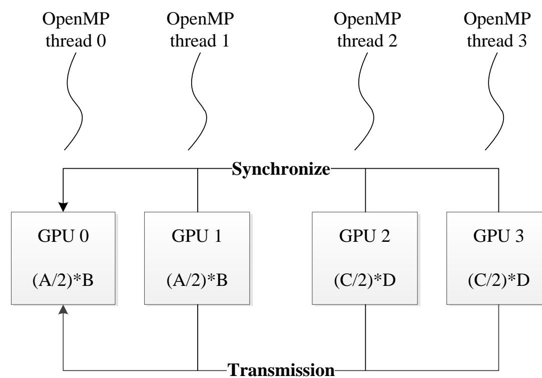

Figure 4. Sample of Data Decomposition of Matrix Composite Operation

Take the composite matrix operation A * B + C * D for example. Figure 4 is a schematic view of data decomposition of matrix. Seen from the figure, GPU 0 completes the multiplication between half of the matrix A and B, GPU 1 completes the multiplication between the other half of the matrix A and B; GPU 2 completes the multiplication between the half of the matrix C and D, and GPU 3 completes the other half of the matrix A multiplied by D. To ensure the correctness of the calculation results, thread synchronization is kept in the process of doing matrix multiplication. In other words, matrix multiplication complete on each GPU while waiting for multiplication completed on other GPUs. Finally, we copy the completed data of each GPU to GPU 0. The complexity and parallelism degree of matrix addition is relatively low, so it can be completed by one GPU. Finally, the result will be delivered to the host by copying function and will be validated the correctness on the host.

4.2. Data Calculation and Consolidation

In this paper, the speedup of large-scale computing program by using multi-GPU collaborative computing compared with single GPU should be discussed, and at the same time the effectiveness and efficiency of multi-GPU collaborative computing should be verified. Therefore, we did not use the forefront CUBLAS library to accelerate matrix operations in a single GPU. Instead each thread calculates one element of the result matrix. As the key of numerical computing problems, matrix multiplication is one of the most common and important scientific computing.

To simplify the description, this section illustrates the data calculation and data consolidation by doing matrix multiplication with four GPUs. In the specific implementation, the realization of the core algorithm is to open four threads via OpenMP on the host side. The areas which need multi-GPU collaborative computing are set as parallel regions by compiling guidance statements. The thread id number in host side

Copyright ⓒ 2015 SERSC

corresponds with the number of GPU device, so that a thread controls a GPU. Finally, each GPU will copy the completed data from its own side of the device pointer to the area of the host pointer, so that to achieve data consolidation purpose.

5. Experimental Results and Discussions

5.1. Experiment Setup

Based on the analysis from the previous section, we estimate the performance over the common and typical applications. The performance of the system will be test and analyzed by using the matrix multiplication. The target system consists of the CUDA client which is consisted of 10 virtual machines, the CUDA server which is consisted of 2 servers and the GPU registration center located at the virtual machine on one of the servers. The software and hardware environments of the system are shown in Table 1. The CUDA server is deployed by two servers. One of them is equipped with Intel Xeon E7-4830, 48 GB system memory and 4 NVIDIA Tesla C2050 GPUs, and the other is equipped with Intel Core i5-2300, 16 GB system memory and 24 NVIDIA Tesla C2050 GPUs. The CUDA client is deployed with 10 virtual machines. The virtual machine is turned on through the workstation on the physical machine. The specific configuration of the virtual machine contains two vCPUs, 1G system memory and 20G external memory. The GPU registration center and the CUDA client are at the same level of virtual machine and their configuration are the same.

Table 1. The Configuration of System Environment

Node Names	Number	Description of the Hardware Environment	Description of the Software Environment
CUDA Client	10	vCPU*2, 1GB memory	CentOS 6.0
CUDA Server	1	Intel Xeon E7-4830, Tesla C2050*4, 48G memory	CentOS 6.0
CUDA Server	1	Intel Core i5-2300, Tesla C2050*2, 16G memory	CentOS 6.0
GPU Register Center	1	vCPU*1、1GB memory	CentOS 6.0

5.2. Factor Obtained

In this paper, the value of α_i and β_i are got via plus or minus 0.02 to step lengths. Based on the feedback of the turnaround time of a given task and when the turnaround time is the shortest, α_i 、 β_i corresponds to the value on the corresponding task type value. The data structure is defined for each α_i and β_i in the scheduling algorithms: *struct get_alpha_and_beta {float alpha; float beta; float cycling_time;} state*. We validate the possible values for each α_i and β_i and sort the results by *state->cycling_time*. The minimum value corresponding with α_i and β_i is returned. If the size is same it returns the average.

 Copyright ⓒ 2015 SERSC

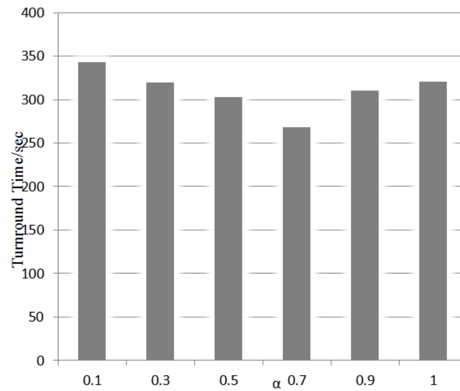

Figure 5. Corresponding Turnaround Time with Different Configurations of α_i and β_i ($\beta_i = 1 - \alpha_i$)

In this paper, we take matrix multiplication for example. First we send the tasks to different GPUs by setting different α_i and β_i value. It will generate a number of tasks to simulate a real environment load in the service side. At last, we evaluate the best combination about α_i and β_i via the feedback of turnaround time and written into registration table. In different configurations of α_i and β_i corresponding turnaround time is shown in Figure5.

5.3. Basic Performance

For the scheduling algorithm in this paper, it is mainly validated the feasibility and efficiency from two aspects. One is within a certain time, the completion of average number of tasks which are sent by the client side and done by the server, and the other is the average turnaround time of the system. The experimental environment is configured in table I. In this section the experiment of matrixMul program which is coded by ourselves will be tested. It will randomly generate matrixMul tasks within a certain time on each virtual machine, and record a total number of completed tasks. Matrix parameters of matrixMul in the experiment are set as 4096 * 4096.

The comparison of the scheduling algorithm between the native environment of this paper and the literature [9] GV-GS task scheduling are shown in Figure 6. GV-GS in the picture is GPU-based task scheduling which is applied from the literature [14]. Local mode does not apply to multi-machine environment, and both servers cannot be used at the same time. The digital shown in the picture is the average under the native mode of the two servers. Seen from the figure, the numbers of tasks which are completed are significantly better than GV-GS after the optimization. In Figure 7, it shows the situation of the whole system of the average turnaround time after the task scheduling through the GPU registration center.

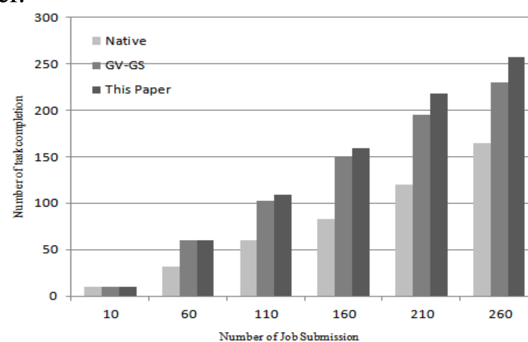

Figure 6. Average Generation GPU Virtualization Task Completion Case

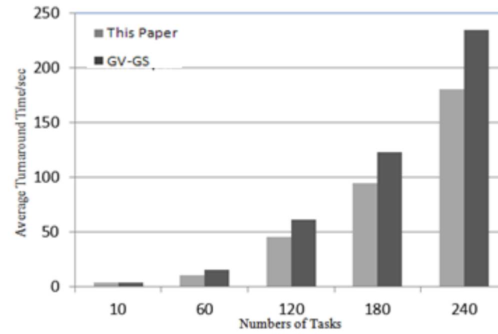

Figure 7. Average Turnaround Time

As can be seen from Figure 7, when the scale of the task becomes larger, the improved scheduling algorithm can significantly reduce the turnaround time. This is because in reality, the task type on each GPU card is not identical. Moreover, its time complexity of the calculation and the completed numbers of operation in per instruction cycle of multiprocessor are not the same, which leads to a big difference of their real load with the same tasks on each card. The improved scheduling algorithm for the evaluation of GPU load is not only closer to the real situation, and more able to reduce the turnaround time for the task under the premise of GPU load balancing.

5.4. Matrix Operations and DFT Sample

1) Matrix Operations

After the description of data decomposition, data computing and data consolidation in the computational process under the multi-GPU collaborative computing, this section analyses the basic performance in the multi-GPU environment with the experiment of matrix composite computing and discrete Fourier transform. The statistical characteristics of these two programs include data scale, kernel execution times (iterations), memory capacity used, experimental variance, and data traffic between memory and display memory of their own. Figure 8 shows the computing time of A * B + C * D in single GPU and multi-GPU. Figure 9 depicts the speedup.

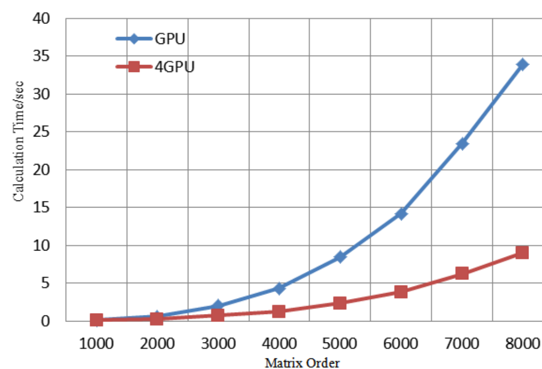

Figure 8. The Calculation Time of Matrix Composite Computing In Multi-GPU and Single GPU

Copyright ⓒ 2015 SERSC

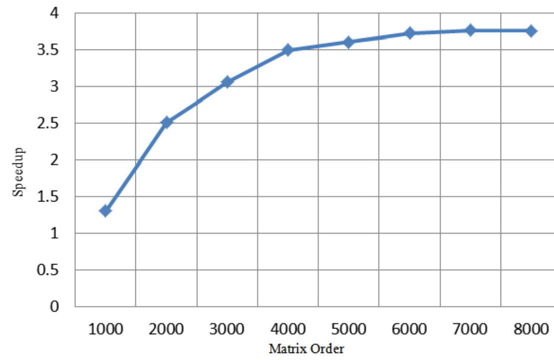

Figure 9. The Speedup of Matrix Composite Computing Between Multi-GPU and Single GPU

As is shown in Figure 8, with the matrix size becoming larger, the time of computing on four GPU shows a slow growth. While the growth rate of the single-GPU computing time is relatively fast.

As is shown in Figure 9, the ratio is low when the scale of matrix is small. Ratio here means the calculation time of matrix relative to overall time (asynchronous data copy and data calculation). In that case, the acceleration ratio is relatively low. As the size of matrix increasing, the proportion of matrix calculation time relative to overall time gradually increases. So the speedup grows linearly before the stable trend, and increasingly close to the number of GPUs. The accelerated growth rate becomes very slow when the acceleration ratio reaches 3.76. Although four GPUs work concurrently in matrix operations, when they carry out the two copies of the data is asynchronous because the use of PCI-E bus is exclusive. With the data size increasing, the time of asynchronous transmission is also growing. Therefore, the growth rate of the proportion of data calculation time relative to overall time becomes slower and slower, which leads to the slow of the acceleration growth rate.

2) Discrete Fourier Transform

Discrete Fourier transform (short by DFT) [17], is the extension of continuous Fourier transform. DFT is discrete form in the time domain and frequency domain and convert the time-domain signal sample to the frequency domain samples in a discrete-time Fourier transform. The DFT is widely used in the areas of signal processing, image processing and so on. Therefore, there is some theoretical and practical significance if we select DFT as the verification object. In this paper, we choose the higher complexity of computing of DFT to verify the speedup which is multi-GPU with respect to of a single GPU in virtualization environment. Next we describe the ideas and processes of how to accelerate DFT by using multi-GPU.

DFT for converting: The sequence of N complex numbers $x_0, x_1, \ldots, x_{N-1}$ is transformed into an N-periodic sequence of complex numbers.

$$\hat{x}[k] = \sum_{n=0}^{N-1} e^{-i\frac{2\pi}{N}nk} x[n] \quad k=0,1,\ldots,N-1.$$

Among them, e is the base of natural logarithm, i is the imaginary unit. Let $\omega = 2\pi kn / N$, and replace $e^{-i\frac{2\pi}{N}nk}$ with $\cos(\omega) - i\sin(\omega)$ by using Euler function. Figure 10 is performed by the DFT calculation time in single GPU and four GPUs respectively. Figure 11 shows the acceleration of four GPUs with respect to single GPU.

Because the size of the input points of DFT is an integer multiple of 10000, we don't use 2^n . As shown in Figure 10, with the increasing of DFT input points, the calculation time of 4 GPUs shows a slow growth trend, while the growth rate of the single-GPU computing time is relatively fast. As can be seen from Figure 11, when the input

sequences of points are small, the proportion of calculation time relative to overall time (asynchronous data copy and data calculation) is low, so the speedup is relatively low. As the input sequence of points increases, the speedup is closer and closer to the GPU numbers. It is basically stable when the acceleration ratio reaches 3.784. Here speedup variation is the same as Figure 9. In addition, the speedup of DFT is higher relative to matrix operations because the scale size of DFT is smaller than matrix operations. The main computation in GPU is trigonometric operations. It belongs to computationally intensive algorithm because the instruction cycle for the completion of one calculation is very large and calculates fetch relatively high. So the overhead which is due to asynchronous copies is relatively slow and the speedup the relatively high.

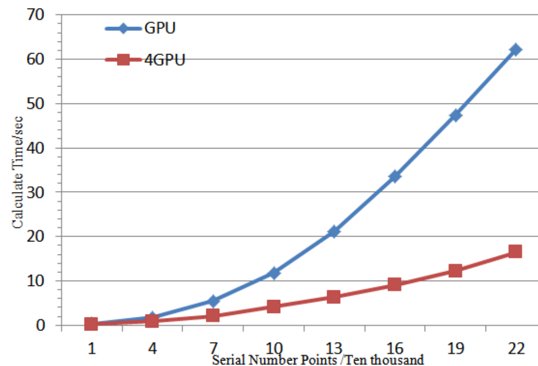

Figure 10. DFT's Computing Time with the Input Series in 4GPU and Single GPU

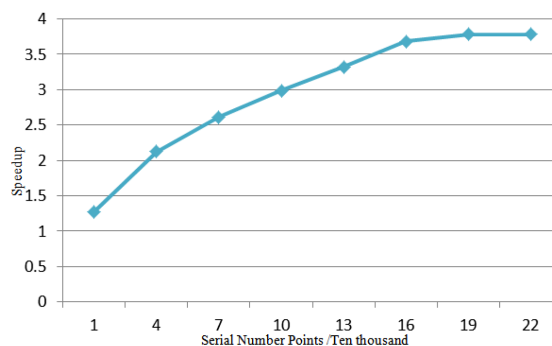

Figure 11. Speedup OF Dft

6. Conclusion

We first introduce the scheduling algorithm to virtualization environment and multi-GPU collaborative computing. We make a detailed description respectively in three components: CUDA client, CUDA server and GPU registration center, and then describe the entire process and its sub-GPU scheduling algorithm in detail. By simulating the fixed number of tasks in the client, we test the completed tasks within a certain period of time and the average turnaround time for the entire systems. Also, we verify the feasibility and efficiency of the proposed scheduling algorithm by experiments. The realization of computing the composite matrix and discrete Fourier transform on multi-GPU not only prove the herein collaborative computing solutions support for virtualization platforms, but also evaluate the basic performance of the proposal in a single GPU and multi-GPU environment comprehensively.

 Copyright ⓒ 2015 SERSC

Acknowledgements

This work was supported in part by the National High Technology Research and Development Program ("863"Program) of China (No.2009AA044601), the National Natural Science Foundation of China (GrantNo.61139002), Fundamental Research Special Research Projects of Nanjing University of Aeronautics and Astronautics (No.NP2013308).

References

[1] E. Overby, “Process Virtualization Theory and the Impact of Information Technology. Organization Science archive”, Organization Science, vol. 19, no. 2, **(2008)**, pp. 277-291.
[2] M. Dowty, J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture", ACM SIGOPS Operating Systems Review, Vol. 43, no. 3, **(2009)**, pp. 73-82.
[3] W. Shen, L. Sun, D. Wei, et al, “Load-Prediction Scheduling for Computer Simulation of Electrocardiogram on a CPU-GPU PC”, 2013 IEEE 16th International Conference on Computational Science and Engineering (CSE) , **(2013)**, pp. 213-218.
[4] B. J. N. Wylie, W. Frings, “Scalasca support for MPI+OpenMP parallel applications on large-scale HPC systems based on Intel Xeon Phi”, in Extreme Science and Engineering Discovery Environment, **(2013)** ,pp. 1-8.
[5] J. N. Imamura, “Imogen: a parallel 3D fluid and MHD code for GPUs” Proceedings of the 27th international ACM conference on International conference on supercomputing (ICS '13), **(2013)**, pp. 479-480.
[6] O. Maitre, N. Lachiche, P. Clauss, et al, “Efficient Parallel Implementation of Evolutionary Algorithms on GPGPU Cards”, Lecture Notes in Computer Science, vol. 5704, **(2009)**, pp. 974-985.
[7] D. Bruneo, S. Distefano,F. Longo, et al, “Workload-Based Software Rejuvenation in Cloud Systems”, IEEE Transactions on Computers, **(2013)**, pp. 1072-1085.
[8] Shi L, Chen H, Sun J, “vCUDA:GPU Accelerated High Performance Computing in Virtual Machines”, Proceedings of International Parallel & Distributed Processing Symposium, Rome,**(2009)**, pp.1-11.
[9] V. Gupta, A. Gavrilovska, K. Schwan, et al, “GViM:GPU-accelerated virtual machines”, ACM Workshop on System-level Virtualization for High Performance Computing, **(2009)**, pp. 17- 24.
[10] G. Giunta, R. Montella,Agrillo ,et al, “A GPGPU Transparent Virtualization Component for High Performance Computing Clouds”, EuroPar conference on Parallel Processing, **(2010)**, pp. 379-391.
[11] J. Duato, A. Pena, F. Silla, et al, “rCUDA:Reducing the number of GPU-based accelerators in high performance clusters”, International Conference on High Performance Computing and Simulation, **(2010)**, pp.224-231.
[12] M. Hongyi, R. S... Diersen, W. Liqiang, et al, “Symbolic Analysis of Concurrency Errors in OpenMP Programs”, International Conference on Parallel Processing, **(2013)**, pp. 510-516.
[13] L. Xiaoxian, Z. Rongcai, H. Lin, L. Peng, “An Automatic Parallel-Stage Decoupled Software Pipelining Parallelization Algorithm Based on OpenMP”, Security and Privacy in Computing and Communications (TrustCom): 2013 IEEE International Conference on Trust, **(2013)**, pp. 1825-1831.
[14] M. Ye, “The Research of Virtualization on GPU General-Purpose Computation in Cloud Computing”, The Graduate School, Nanjing University of Aeronautics and Astronautics, **(2012)**.
[15] C. Shuai, M. Jiayuan, et al, “A performance study of general-purpose applications on graphics processors using CUDA. Parallel and Distributed Computing”, Vol. 68, no. 10, pp. 1370-1380, **(2008)**.
[16] NVIDIA CUDA: NVIDIA CUDA C Programming Guide. NVIDIA Corp., (2013), version 5.5.
[17] DFT. http://en.wikipedia.org/wiki/Discrete_Fourier_transform.

Authors

Yujie Zhang was born on March 3, 1991 in JiangSu. She is a graduate student with College of Computer Science and Technology, Nanjing University of Aeronautics & Astronautics. Her research interests include virtualization technology and high performance computing.

Copyright ⓒ 2015 SERSC

Jiabin Yuan was born on January, 1968 in JiangSu. He is a professor and Ph. D. supervisor of Nanjing University of Aeronautics & Astronautics. His research interests include information security, high performance computing, and quantum cryptography.

Xiangwen Lu is pursuing ph.D degree in College of Computer Science and Technology, Nanjing University of Aeronautics & Astronautics. His research interests include high performance computing, cloud computing, quantum simulation.

Xingfang Zhao was born in Sichuan Province, China, in 1991. He is currently a master degree candidate at Nanjing University of Aeronautics and Astronautics, Jiangsu, China. His research interests include virtualization technology and high performance computing.

 Copyright ⓒ 2015 SERSC