Sharing Computing Resources with Virtual Machines by Transparent Data Access

Takuma Nakajima, Masato Yoshimi, Hidetsugu Irie, and Tsutomu Yoshinaga
Graduate School of Information Systems
The University of Electro-Communications
tnakajima@comp.is.uec.ac.jp, {yoshimi,irie,yosinaga}@is.uec.ac.jp

***Abstract*—Cloud computing has rapid growth in enterprise and academic areas. Computing platform makes up the transition from physical servers to virtual machines (VMs) in the cloud. Instead of many advantages, VMs remain several problems to employ effective utilization of physical computing resources, especially many-core accelerators. Even though GPGPU is a hopeful solution for high-load applications, existing methods to utilize GPUs from VMs are subjected to various restraints. In order to solve this problem, we propose a flexible method to share external computing resources by providing transparent access for data in the VMs. By committing commands to a computing host which processes the jobs as substitution, VMs can process high load jobs as necessary even if the VM has a tiny configuration. The computing host mounts the working directories in the VMs and enqueues jobs committed by the VMs. Experimental results show that the overhead of our implementation is sufficiently small in the low I/O load processes.**

I. Introduction

Since cloud computing was introduced around 2006[1], various services have been appeared in the form of cloud. Companies and universities gradually shift their research and development environments to the cloud. Researchers can create and deploy new fresh testing VMs onto their private clouds in a few seconds with no maintaining costs for local computers.

Although cloud has drastically reduced the costs for building, operating and managing, several problems are still remaining to effective utilization of physical computing resources. Since the VMs are usually crammed into the physical machines, computing resources of each VM are insufficient to compute high-load applications such as image processing and video encoding. Since hardware of VMs is full- or para-virtualized, their performance is not as powerful as a VM host because of its overheads.

Even though utilizing many-core accelerators such as GPU is a promising solution for high-load applications, there are several challenges to use limited number of accelerators from VMs. Typical works are rCUDA[2], V-OpenCL[3], and NVIDIA GRID[4]. They use API interception or PCI pass-through to access GPUs from the VMs. These methods are effective for applications written with CUDA or OpenCL, but these are restricted for hardware limitations. Since applications are not always written with these languages, more flexible mechanism to use remote computing resources for various types of application is required. In addition, these technologies might incur considerable performance degradation when communicating between CPU and GPU, which is connected via network instead of a high-speed bus. Live migration of VMs is also restricted because of strong relation between the VMs and the computing host. Utilizing intra communication between CPU and GPU on the host and its optimization may solve these problems.

Meanwhile, there is a concept of Resource-as-a-Service (RaaS) cloud[5]. The main purpose of RaaS is minimizing cost and energy consumption of computing resources, such as CPU, memory, GPU, and other accelerators, by on-demand attachment of them to VMs. Since computational load of a VM is not always high, the configurations of VMs should be minimized for more efficient use of the cloud environment. On-demand attaching and detaching of the high performance computing resources will increase their efficient usage.

To share the computing resources with maintaining advantages of the cloud computing, we propose a method to share computing resources by providing transparent data access. In this method, a remote computing host mounts the working directories of VMs via the network and executes requested jobs as substitutions. It broadly divides operations into computing ones which are processed on the computing host, and I/O which is performed on the client VM. Unlike the API interception and PCI pass-through, this method makes best use of performance of the computing host by no memory transmission and sequential uploading and downloading. In addition, since the VMs and the computing host are associated with the network level, VM live migration need not stop nor suspend the execution process.

To implement and evaluate the resource sharing system, we configure private cloud and execute several image processing as benchmarks. As results of the evaluation, the image processing job is completed in about 6 seconds with offloading to a computing host, which takes up to 30 seconds in VM locally. These executions required about 0.3 seconds overheads for providing transparent access to the working directory. Through the evaluations, we confirmed that proposed methodology permits sharing computing resources such as GPU which has not been able to utilize easily, maintaining admissibility overhead.

II. Related Work

A. Using Remote Computing Resources from VMs

1) PCI Pass-through Model: PCI Pass-through [6] is a common method to use a physical device directly in VMs. A hypervisor passes through the physically attached PCI devices to the VM. Because of the direct attachment, PCI pass-through devices achieve almost same performance as using the device in a local host environment[7]. Although we can use many PCI devices in the VMs by the PCI pass-through technology, these VMs cannot migrate from their original host to others. Moreover, the attached PCI devices usually cannot be shared with multiple VMs.

NVIDIA GRID[4] is a technology that shares a single GPU with multiple VMs. NVIDIA GRID GPUs equip multiple GPUs on a board and each GPU can be logically divided into multiple virtual GPUs. Since divided virtual GPUs are recognized as PCI attached devices, users can pass-through them to VMs. However, the attached virtual GPUs become exclusive use with their attached VMs because of the limitation of PCI pass-through. Hence users cannot move the VMs from an original NVIDIA GRID host to the others. Moreover, GPU series which support this technology is limited to expensive ones.

2) API Interception Model: API interception is another method to use a GPU device with VMs. It intercepts CUDA or OpenCL API calls and rewrites or resends them to the other hosts by using wrapper libraries. vCUDA[8] and V-OpenCL[3] use this model. When the VM host intercepts the API calls through these libraries, they call the real API of the GPU driver instead of emulating these calls by the CPU. Although VMs can share a single GPU on a host, they cannot migrate to the other VM hosts because of its strong relationship. VGRIS[9] and gVirtuS[10] also use the API interception model and they call the real GPU driver of the VM host.

rCUDA[2] also utilizes a similar API interception to share a remote GPU from VMs. With this method, when the VM host gets API calls of VMs, it sends trapped API calls to a remote GPU host through the network socket. The remote GPU host executes the received API calls through the real GPU driver. After executing the call, the remote GPU host returns a result of the call to the VM host through the network socket. Since VMs can use remote GPUs, VM hosts are not necessary to equip GPUs. Although rCUDA executes CUDA API calls on a single GPU node, there is a method to execute these API calls with distributed GPU cluster[11]. Since rCUDA requires communication between the VMs and the computing host, rCUDA enabled systems incur relatively large overhead[7].

3) Resource-as-a-Service (RaaS) Cloud: RaaS cloud is a concept for selling individual resources, such as CPU, memory, and I/O devices, on-demand[5]. Most of providers in current market do business by selling units of time to run the VMs. This is not efficient one for users because they don't always use computing resources fully. RaaS type is more efficient way to provide tiny VMs for users by dynamically attaching and detaching required resources. Additionally, more number of VMs can be run on a single VM host because the VMs become small.

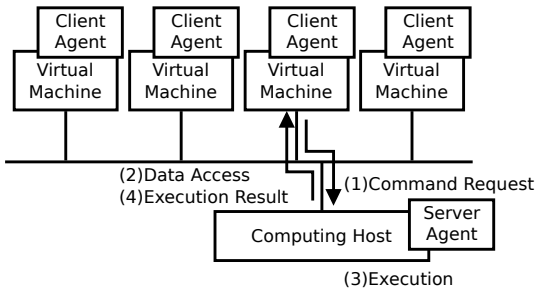

Fig. 1. Design of our resource sharing method

Although there exist alternative technology to attach computing resources to the VMs on-demand, such as CPU overcommittng and memory ballooning, few services enable these features because of their complex settings.

III. Design properties and operation

A. Setting up situation

To propose a method for sharing computing resources, suppose a environment illustrated in Fig.1. VMs are typically provided for users as a tiny configuration, small processor power, memory size and disk capacity to saturate a physical machine for energy efficient utilization. Users regularly compute their own tasks in their VMs, such as some web services, processing or other operations.

We suppose that each VM is usually in low-load, but sometimes required to operate high-load jobs. When a high-load job is committed, the VM throws the job to the computing host, which is configured comparatively large such as multi-core physical machines with huge memory and many-core accelerators. Computing hosts are shared by multiple VMs connected with a network, and operate requests in order of arrival.

The architecture mentioned above introduces cloud environment higher computing density and users lower usage fee, since physical machines can be stored more number of VMs and users need not a VM with large configuration to prepare beforehand rush time.

B. Sequence to share computing resources

Each VM and computing hosts run a client agent and a server agent, respectively as shown in Fig.1. A client agent issues control signal and data to server agent to utilize features in the computing host with following sequences which corresponds to the numbered operations in Fig.1.

1) The client agent tries to connect and requests the command to the computing host,
2) the computing host accesses data in the VM to execute the command after acceptance of the request,
3) the requested command is executed, and
4) the result of the command is returned to the client agent.

Although such a part of sequences mentioned above is already realized by rsh or ssh commands, the exclusive feature of proposed methodology is that the server agent mounts the local directory through client agent in sequence (2) besides traditional client-server model. This feature brings cloud following advantages;

1) Reducing networking overhead caused by excluding frequent communication between the client and the server in the process.
2) Since the server agent plays a proactive role in execution in sequence (3), requests from the client agents can be controlled ordering, and an exclusion control for limited hardware such as GPUs can be implemented easily.
3) The medium between client and server is made by loose network to introduce flexible operation including live migration.

C. Advantages in proposed methodology

The primary objectives in proposed methodology are summarized that (1) sharing computing resources from multiple VMs, and (2) improving performance efficiency in the sight of whole cloud (3) maintaining advantages of utilizing cloud environment.

Live migration is a key feature of cloud computing that VMs migrate from a physical machine in which the VM is running to another without turning off. Attaching and detaching the computing devices directly to the VMs like PCI Pass-through have made strong connections between physical device and VMs, which restrict the live migration.

With flexible communication, maximizing computational performance is also important. This means that some artifice to minimize the overhead by introducing proposed methodology. Since frequent communication between client and server is caused by overhead accumulation such as API interception, the client agent exports the working directory on which executes the commands by the server agent. Adopting network file system, which transfers data file by file, introduces the reduction of the number of data transfer through the sequence.

Exporting directories by client agent is also regarded as that the VMs provide transparent data access to the computing host. All computing operations are executed in the computing host and the computing resources are shared through the network, VMs can migrate one host to another and also computational performance are maximized.

From the perspective of the user of the VM, the command is executed on and returned the results from on the VM. The detailed implementation is described in the next section.

IV. Implementation of Resource Sharing Environment

A. Resource Sharing Architecture

To evaluate our method, we constructed a private cloud, and implemented the server and client agents. The server and client agents provide a function to share computing resources on the computing host among VMs. The physical architecture of the private cloud is shown in Fig.2. In the figure, physical machines are shaded. A computing host consists of CPU, GPU and memory which are shared resources. The computing host executes the server agent, while each VM executes the client agent. These agents run as normal processes, they do not have to recognize their VMs' live migration. There exists a cloud gateway server in order to access to the cloud network. The communications between the VMs and the computing host are done through the gateway. The cloud management server executes cloud computing software to manage VMs and the VM hosts.

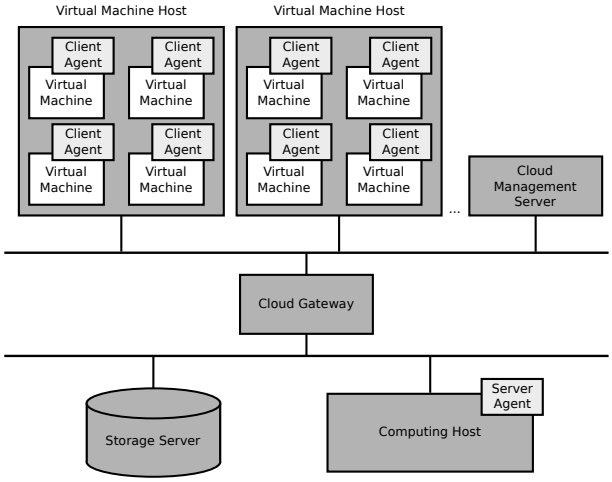

Fig. 2. Physical architecture of the private cloud

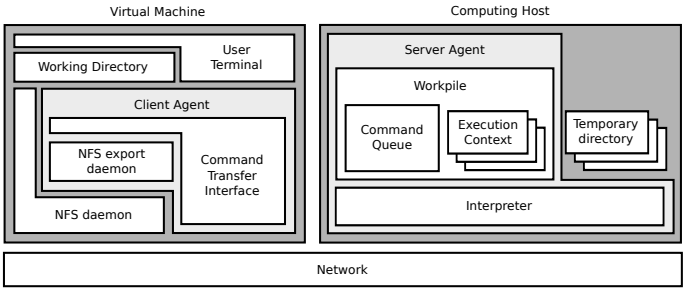

Fig. 3. Software architecture of the resource sharing interface

B. Server Agent for Computing Hosts

Fig.3 shows a software architecture of the proposed resource sharing interface. The server agent consists of an interpreter and a workpile component. The workpile component is consisted of a command queue and execution contexts. Execution contexts are created by the interpreter and registered to the command queue with the requested command. Temporary directory is also created dynamically by the interpreter for mounting the working directory of the VM. We describe detailed roles of the interpreter and workpile in the following paragraphs.

Interpreter component accepts the messaging commands from the client agent and it communicate with workpile. In our implementation, the interpreter understands five messaging

commands; checkin, mount, exec, umount, and checkout. The client agent sends these messaging commands usually in this order. Checkin and checkout messaging commands are used for initialization and finalization. Mount and umount messaging commands mount and unmount the working directories of the VM. Exec messaging command is used perform a job.

Workpile is a command execution component with a command queue and execution contexts. The command queue stores the requested commands. The execution context contains a path of the temporary directory which is mounted a working directory of the VM, and each execution context is associated with the connection to the VM. Workpile is implemented as an event-driven model. Events include registering a command to the command queue and finishing the execution of the command. These events trigger the command execution as the event handler, and a command which is dequeued from the command queue is set to a callback function. In the event handler, it executes the requested command as the callback function. Results as well as errors are sent back to the standard output of the VM. In our current implementation, workpile does not check whether a command uses any special devices or not, and therefore only a single command is executed at a time. For this reason, the server agent currently do not hold a device list used by commands.

C. Client Agent for Virtual Machines

The client agent is composed of a NFS export daemon and a command transfer interface. The NFS daemon in the VM is provided by the operating system.

The Client agent provides the resource sharing interface in the command line. Although the working directory is not a component of the client agent, it is an important component to provide the transparent data access to the computing host.

NFS export daemon is used to export the working directory with NFS. When this daemon receives the path of the working directory, it exports the directory to the network. Since user id and group id are different in each platform, this daemon sets proper permissions. The computing host will read and write files with the permission of the user of the VM.

Command Transfer Interface is a command line application to send a terminal command to the computing host. The user of this interface issues terminal command with its arguments to execute on the computing host. When the interface received the terminal command and its arguments, it requests to export the working directory to the NFS export daemon. Then it commits the command to the computing host. After requesting execution of the command to the computing host, the computing host responses the command outputs as well as errors to the VM. The command transfer interface displays the results of the commands. After the command execution, the command transfer interface requests the NFS export daemon to finish exporting of the working directory. Table I shows the format of the Command Transfer Interface and its arguments. This interface try to read specific configuration file by default when no configuration file specified, these options are not always needed.

V. Evaluation

A. Setup Experiments

Table II shows the specification of VM hosts. A computing host is organized with the same specification of VM hosts as shown in Table III except for the GPU device. Four types of VMs as shown in Table IV are configured for experiments. Two types of full duplex network connections are also configured. Each host is connected each other via gigabit Ethernet. VMs are deployed by CloudStack which is one of the typical cloud computing software.

Two types of image processing jobs, *blur* and *resize*, with three image sizes are evaluated for offloading processes to the computing host. The matrix multiplication program is utilized from AMD APP SDK. Both image processing and matrix multiplication are OpenCL enabled, however, only the latter uses the GPU device.

Details of the test image is described in Table V. The types of image processing are blurring and resizing a 10,000x10,000 image. Blurring processes calculate the average of 5x5, 10x10, and 20x20 adjacent areas for each pixel, and resizing processes reduce the image geometrically to 5%, 10%, and 20%, respectively.

B. Evaluation of Single VM

Computational time including overhead is measured when a single VM offloads the image processing jobs to the computing host to evaluate the efficiency of the job offloading system. To obtain the computational time is as follows;

1) The VM offloads the job and measures time 12 times,
2) omits the longest and the shortest times,
3) and derives the average value from the rest of results.

Computational time is measured on each VM without communication time between the VM and users since the communication time varies depending on the network types and their status.

TABLE I
COMMAND LINE FORMAT AND ARGUMENTS

```
usage:
$ offld [options..] [--] exec_command [exec_command_args..]
```

Option	Description
-c, --config=<filename>	load configuration file from <filename>
-p, --profile	show execution profile
-H, --host=<host>[:<port>]	connect to host:port
-n, --no-mount	do not mount current directory
-o, --output=<filename>	output stdout and stderr to <filename>
-Q, --no-logging	do not create logfile for stdout and stderr
-q, --quiet	do not output stdout
-d, --debug	show debug message
-h, --help	show help message

TABLE II
SPECIFICATIONS OF THE VM HOSTS

CPU	Intel Core i7 3770 (3.40GHz)
RAM	32GB
NIC	1 Gigabit Ethernet

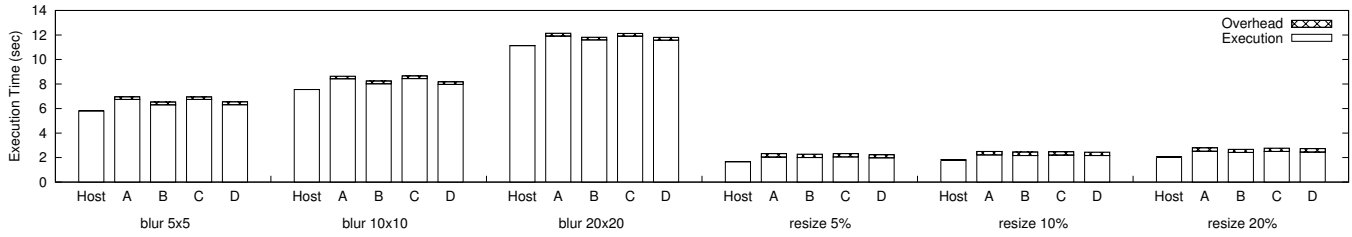

Fig. 4. Execution time of image processing jobs for a 100Mpx image

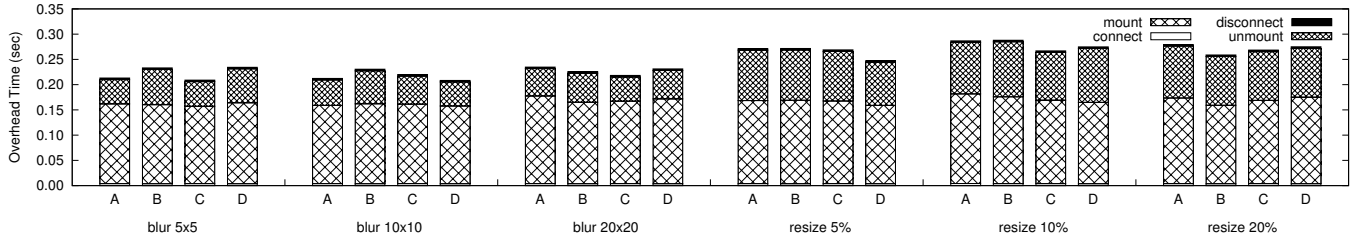

Fig. 5. Overhead time of image processing jobs for a 100Mpx image

Fig.4 and Fig.5 shows the computational time of each jobs and their breakdown of overhead, respectively. The overhead includes mount, connect, disconnect, and unmount introduced by the offload jobs. VMs maintains offloading jobs properly even when live migration occurs. Env.A, B, C, and D in these figures correspond to the specification of VMs in Table IV. The labels named Host in Fig.4 represent the computational time when the job is executed on the computing host. Table VI shows the execution time on VMs with 1GHz CPU and 1GB memory without offloading.

Fig.4, Fig.5 and Table VI show that the computational time on the computing host is much faster than on the VM. The result is quite natural because the computing resources of VMs is restricted compared to the computing host. the overhead is almost constant time between 0.2 and 0.3 seconds regardless of the network throughput and the configuration of VMs. It is sufficiently small values for high load applications which usually take more than 5 seconds long. Fig.4 also shows the execution times are comparable when the computing host executes the job directly and when the VM offloads the job. The computational time is affected by the network throughput since the computing host mounts the working directory in the VMs. Fig.5 explains that primary overhead is introduced by mounting and unmounting the working directory in the VMs. The overhead can be reduced by utilizing more efficient file system.

C. Evaluation of Multiple VMs

To evaluate the performance of the resource sharing system, the experiment that multiple VMs offload the jobs to a single computing host simultaneously is observed. Since the frequency that multiple VMs offload the jobs at the same time is not so high, the situation can be regarded as the worst case. Configuration D in Table IV is used for VM. Fig.6 shows the computational time per each job measured in the computing host. These results do not include the waiting time for the other jobs' execution. Computational time for a single VM directly incurs the offloading overhead, including connection, disconnection, mount, and unmount. With the number of VMs increases, the overhead is gradually hidden by the computation because the jobs are processed in a pipelined fashion. As a result, the computational time per job is inversely reduced as the number of VMs is increased.

Fig.7 shows the computational time per job measured in the VM. The computational time in Fig.7 contains the queuing time in the computing host. It is obvious that a single job consumes around 0.5 seconds. The computational time gradually

TABLE III
SPECIFICATIONS OF THE COMPUTING HOST

CPU	Intel Core i7 3770 (3.40GHz)
RAM	32GB
NIC	1 Gigabit Ethernet
GPU	AMD Radeon HD 6970

TABLE IV
CONFIGURATIONS OF EXPERIMENTAL ENVIRONMENT

Environment	VM CPU clock	VM Memory	Network throughput
A	500MHz	1GB	100Mbps
B	500MHz	1GB	1000Mbps
C	1GHz	1GB	100Mbps
D	1GHz	1GB	1000Mbps

TABLE V
DETAILS OF THE TEST IMAGE

Type	JPEG
geometry size	10,000x10,000
file size	8MB

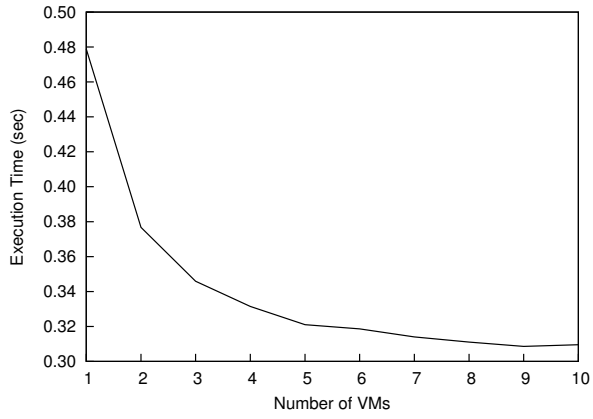

Fig. 6. Transaction time on a computing host for matrix multiplication

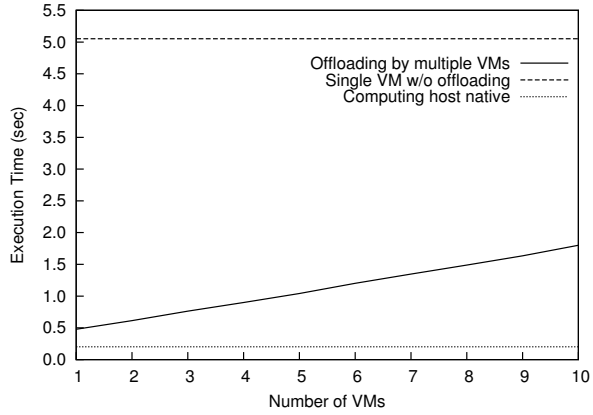

Fig. 7. Computational time for each VM including queuing time

increases caused by the waiting time according to the number of VMs. However, the overhead is quite small for the sake of light-weight job switching.

VI. Discussion

The main purpose of this research is to share computing resources with multiple VMs and maximize its computational performance without harming the advantages. We confirmed the overhead of the job is sufficiently smaller than the benefit of acceleration by offloading to the computing host, which marks higher performance.

TABLE VI
Computational time on a VM without offloading (Environment D)

Job Type	Option	Computational Time (sec)
blur	5x5	778.1
blur	10x10	810.3
blur	20x20	806.4
resize	5%	28.6
resize	10%	26.1
resize	20%	37.7

Due to the design, any computational resources can be shared with multiple VMs flexibly, without complex modifications of applications. Note that the commands used to the evaluations have low I/O load, therefore the high I/O load processes might have the I/O bottleneck, but VMs should execute such high I/O load processes on each VM without offloading. As a consequence of these results, our method makes it possible to execute high computational load jobs on VMs which have tiny configurations.

In current implementation, the computing host is required to prepare the command which is requested by VMs because the commands are executed completely within the computing host. Although GUI applications cannot be executed with current implementation neither, when we use SSH X11Forwarding, GUI applications which use our method as back-end can be executed. Applications which need real-time rendering support still will be difficult to reach the sufficient frame rate because the command output of our method is limited to files and console-based output. Technologies of PCI pass-through model and API interception model may allow us to use the visual application, although they restrict VM live migration.

From a perspective of security, current implementation includes a security risk to share the working directory with other nodes. The files in shared directories may be deleted or modified without confirmation of VMs by the computing host. To solve this problem, the connection is established from the VMs to the computing host with encrypted data.

VII. Conclusion

Cloud computing need more efficient use of computing resources. In this research, we proposed a method to share computing resources with VMs by providing access to the working directories of the VMs to the computing host. Although existing methods to share computing resources restrict the live migration of the computational performance, we confirmed that our method does not harm them. Our method has sufficiently small constant overhead compared with the execution time.With these characteristics, our method is quite useful for tiny VMs in cloud environment. Meanwhile we have to enable secure data transmission instead of sharing the working directories for the security risk. We can be resolve the security risk by changing the data transfer method to one which makes a data connection from VMs to the computing host with encryption of communications.

References

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology (NIST), Gaithersburg, MD, Tech. Rep. 800-145, September 2011.

[2] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, 2010, pp. 224–231.

[3] C. Wang, T. Jiang, and R. Hou, "V-OpenCL: a method to use remote GPGPU," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013.

[4] H. Marinez, "Top Enterprise Technology Companies Embrace NVIDIA GRID," http://nvidianews.nvidia.com/Releases/Top-Enterprise-Technology-Companies-Embrace-NVIDIA-GRID-953.aspx.

[5] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "The resource-as-a-service (RaaS) cloud," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, ser. HotCloud'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 12–12.

[6] C.-T. Yang, H.-Y. Wang, W.-S. Ou, Y.-T. Liu, and C.-H. Hsu, "On implementation of GPU virtualization using PCI pass-through," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, 2012, pp. 711–716.

[7] M. Vinaya, N. Vydyanathan, and M. Gajjar, "An evaluation of CUDA-enabled virtualization solutions," in *Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on*, 2012, pp. 621–626.

[8] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines," *Computers, IEEE Transactions on*, vol. 61, no. 6, pp. 804–816, 2012.

[9] M. Yu, C. Zhang, Z. Qi, J. Yao, Y. Wang, and H. Guan, "VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming," in *ACM Symposium on High-Performance Parallel and Distributed Computing 2013*, 2013.

[10] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A gpgpu transparent virtualization component for high performance computing clouds," in *Euro-Par 2010 - Parallel Processing*, ser. Lecture Notes in Computer Science, P. D ' Ambra, M. Guarracino, and D. Talia, Eds. Springer Berlin Heidelberg, 2010, vol. 6271, pp. 379–391.

[11] A. Kawai, K. Yasuoka, K. Yoshikawa, and T. Narumi, "Distributed-shared cuda: Virtualization of large-scale gpu systems for programmability and reliability," in *FUTURE COMPUTING 2012, The Fourth International Conference on Future Computational Technologies and Applications*, 2012, pp. 7–12.