Dynamic Sharing of GPUs in Cloud Systems

Khaled M. Diab†, M. Mustafa Rafique†‡⋆ , Mohamed Hefeeda†
†Qatar Computing Research Institute (QCRI), Qatar Foundation; ‡IBM Research, Ireland
Email: kdiab@qf.org.qa; mustafa.rafique@ie.ibm.com; mhefeeda@qf.org.qa

***Abstract*—The use of computational accelerators, specifically programmable GPUs, is becoming popular in cloud computing environments. Cloud vendors currently provide GPUs as dedicated resources to cloud users, which may result in under-utilization of the expensive GPU resources. In this work, we propose gCloud, a framework to provide GPUs as on-demand computing resources to cloud users. gCloud allows on-demand access to local and remote GPUs to cloud users only when the target GPU kernel is ready for execution. In order to improve the utilization of GPUs, gCloud efficiently shares the GPU resources among concurrent applications from different cloud users. Moreover, it reduces the inter-application interference of concurrent kernels for GPU resources by considering the local and global memory, number of threads, and the number of thread blocks of each kernel. It schedules concurrent kernels on available GPUs such that the overall inter-application interference across the cluster is minimal. We implemented gCloud as an independent module, and integrated it with the OpenStack cloud computing platform. Evaluation of gCloud using representative applications shows that it improves the utilization of GPU resources by 56.3% on average compared to the current state-of-the-art systems that serialize GPU kernel executions. Moreover, gCloud significantly reduces the completion time of GPU applications, e.g., in our experiments of running a mix of 8 to 28 GPU applications on 4 NVIDIA Tesla GPUs, gCloud achieves up to 430% reduction in the total completion time.**

I. Introduction

Cloud computing has become a paradigm of choice for enterprises to host large-scale applications on scalable infrastructure, and for scientific communities to expedite their discoveries. With the availability of commodity, off-the-shelf, computational accelerators, such as programmable Graphics Processing Units (GPUs), their use in cloud computing is becoming popular, and many cloud providers [1], are now offering GPUs as a computational resource in their cloud computing solutions. These solutions offer GPUs using the Infrastructure as a Service (IaaS) model, where each user gets a dedicated access instead of on-demand access to the GPU resources. This results in low utilization of GPU resources, and requires cloud providers to install more GPUs in their cloud setups to service the requests of their users.

⋆This work was done as a postdoctoral researcher at QCRI.

Traditionally, GPUs were only able to execute a compute kernel[1] from one application at a time. However, the latest GPUs [2] from NVIDIA can execute multiple kernels that are launched from different threads. The caveat is that these threads should be part of the same process. This limitation makes GPU sharing between multiple applications challenging for cloud computing setups. Unlike the current cloud computing solutions, it is desirable to share the under-utilized GPUs between multiple cloud users [3], [4], and offer them as on-demand resources for cloud users. This means that cloud users should have access to GPUs only when a compute kernel is ready to execute on a GPU device.

In this paper, we focus on the problem of enabling GPU sharing between multiple concurrent applications in a cloud computing environment. Current commercial cloud computing solutions do not support on-demand GPU computing and sharing. Moreover, previous research approaches [3]–[5] have several limitations and they cannot be used directly to enable GPU sharing in a typical cloud computing framework, such as OpenStack [6] cloud computing platform. We design, develop, and evaluate a framework that enables transparent sharing of GPUs between concurrent applications in a cloud computing environment. Our framework, named gCloud, is designed for "GPU-enabled Clouds". gCloud registers the specifications of GPU kernels from different applications, and executes these kernels when required by the applications on the available GPUs. For executing the kernels, gCloud consolidates multiple kernels and runs them concurrently at the target GPUs. It provides on-demand access to the distributed GPUs to cloud users only when a GPU kernel is ready for execution. With gCloud, we propose a scheduling mechanism that decides the number and the specific applications that should be consolidated on the same GPU for concurrent execution. Specifically, this paper makes the following contributions:

- We present a framework, gCloud, that enables on-demand use of GPUs in cloud computing environ-

[1]"Compute kernel", "GPU kernel", and simply "kernel" refer to the function that is offloaded to a GPU device for computation by an application.

ments. gCloud also supports efficient and dynamic sharing of GPUs among multiple concurrent applications.
- We propose a mechanism to consolidate multiple kernels for concurrent execution based on their global and local memory, number of threads and thread blocks requirements. The proposed consolidation mechanism minimizes the interference between concurrent applications running on the same GPU.
- We implement and integrate gCloud, along with the proposed consolidation scheme, in the OpenStack [6] cloud computing platform.
- We compare the performance of our interference-aware GPU scheduling and consolidation scheme against a round-robin consolidation and scheduling scheme.

II. Related Work

Resource management for GPU-based heterogeneous clusters is an active area of research, and several recent efforts [7] have proposed solutions to enable GPU sharing in time. Several solutions [8] have been proposed to virtualize GPUs by intercepting and redirecting GPU calls. However, none of them simultaneously supports enabling on-demand GPU access, sharing of GPU resources for concurrently executing multiple applications, and scheduling distributed GPUs between multiple applications. In the following, we discuss and compare the existing state-of-the-art that is closely related to gCloud.

GVirtuS [3] is a GPU virtualization service that executes CUDA kernels from the guest virtual machine (VM) by assuming that a CUDA-enabled GPU is available locally. rCuda [9] is a similar framework that supports executing CUDA calls on a remote GPU. Although both GVirtuS and rCuda can be used in cloud setups, neither of them provides support for GPU kernel consolidation for concurrent kernels execution. Furthermore, they do not provide a solution for managing multiple GPUs in cloud environments.

A recent effort [5] proposed a context funneling technique for sharing a GPU context between multiple threads. This work supports GPU kernel launches from multiple threads of the same application, but does not support concurrently executing kernels from different applications. Furthermore, it assumes that the GPU device and applications are located at the same host and that there is only one GPU device per host. These assumptions may not hold in GPU-enabled cloud computing environments.

The work in [4] consolidates GPU kernels in a virtualized environment by extending the GVirtuS [3] back end. It enables applications running within different virtual machines to share one or more GPUs concurrently. It assumes that the GPU kernels are flexible in terms of the number of thread blocks and the number of threads per block, and it modifies these numbers for the consolidated kernels. For finding if two kernels are compatible with each other for consolidation, it assigns affinity scores to each kernel based on their global memory usage, and uses these affinity scores to execute multiple GPU kernels on the same GPU. Since this approach treats each GPU kernel launch independent from each other, it does not support sequence of kernel launches from the same application to reuse the residual data of the previous kernels on the same GPU device. Moreover, it does not manage multiple GPUs in a distributed setting for cloud computing environments.

The use of computational accelerators, specifically GPUs, has been explored for accelerating the compute kernels in HPC environments [10], [11], for improving the performance of computationally intensive OS kernels [12], and for general-purpose scientific and enterprise applications [13]. These efforts explore its use in hybrid CPU/GPU environments to accelerate the compute intensive part of the applications. In contrast to the existing efforts, gCloud provides a holistic approach of using and sharing GPUs for cloud computing setups as a true on-demand computing resource. We demonstrate the effectiveness of gCloud by implementing and integrating it in the open-source OpenStack [6] cloud computing platform in a distributed setting.

III. Proposed GPU Sharing Framework

A. System Overview

Figure 1 shows the high-level system architecture of gCloud, and the key software components that run on the manager and the GPU-based cloud nodes. The cloud computing setup used in our system consists of conventional high-end servers with a mix of GPU and non-GPU enabled nodes. In addition to the software components of gCloud, the manager and cloud nodes in our setup run the default OpenStack processes. Specifically, the manager runs the OpenStack Controller and the cloud nodes run the OpenStack Compute [6]. We assume that the user and VM placement policies are managed by the OpenStack, and it is independent of gCloud. Note that gCloud does not require all cloud nodes to have physical GPUs installed on them, and it allows VMs and applications requiring GPUs to be hosted on any available server. It assumes that a global file system, such as the Network File System (NFS) [14], is available in the setup to host shared objects, which is accessible from all nodes in the cloud setup. Such a file system with global accessibility is common in cloud setups to guarantee the availability of the input data to cloud nodes. gCloud does not require all physical GPUs in the

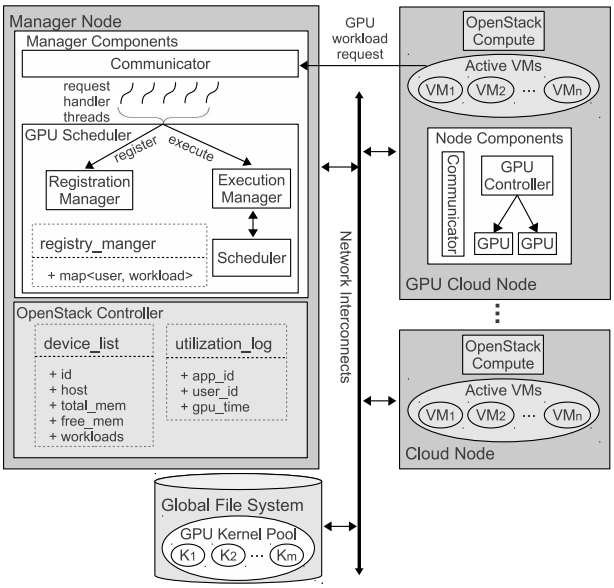


Fig. 1: High-level system architecture and software components of gCloud to enable GPU sharing in cloud environment.

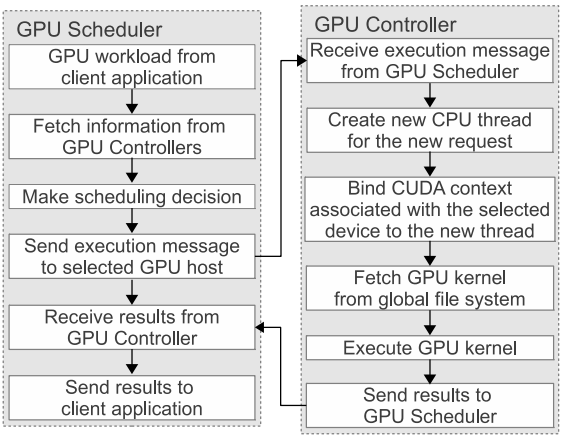


Fig. 2: Execution flow for processing a GPU kernel request at GPU Scheduler and GPU Controller.

system to be capable of executing concurrent kernels. If a GPU does not support concurrent kernel execution, then the kernels are executed on it serially, which is the same as the current state-of-the-art.

At high-level, gCloud works as follows. The manager initializes the system and initiates GPU Scheduler at the manager node and GPU Controller at each GPU-enabled cloud node. The manager then waits for requests from user applications. It registers new kernel requests from users with the system and waits for their requests to execute the registered kernels on distributed GPUs. When a request to execute a kernel is received by the manager, it finds a suitable GPU for the given kernel, and executes it on the selected GPU. From the users' perspective, instead of sending a request for executing a kernel to the GPU, a request is sent to the manager, which manages all internal details of executing the kernel on the available GPU.

GPU Controller initializes the GPU context for each local GPU and uses the same context to run multiple kernels from different applications concurrently. Since multiple kernels launched from a process can be executed concurrent on a GPU, the kernel requests from applications are passed on to the GPU Controller which loads them dynamically at runtime and run them concurrently on a GPU using the same context. Moreover, the input/output of the GPU kernels are handled by the user in the shared object, and remains oblivious to gCloud.

In the following, we describe the operations of gCloud on the Manager and Cloud nodes.

1) Manager Node Operations: At the start of the system, the manager loads information about GPU devices installed in each server along with the amount of physical memory and the compute capabilities of each GPU device. GPU kernels are registered with the manager, which assigns a unique internal identifier to each kernel, and copies the shared object that contains the kernel to the global file system. The information about the GPU kernel and the user is updated in the OpenStack database to keep track of the resource requirements and the utilization of the source VM. Upon the termination of a VM, all shared objects and kernels associated with it are removed from the global file system. Moreover, our system provides a set of APIs to VM, as described in Section III-B, to explicitly unregister a GPU kernel, and to subsequently remove the associated shared objects from the global file system.

One of the key operations of the manager is to schedule and execute GPU kernels on an available GPU device. Figure 2 shows the execution flow for processing a request from the user to run the given GPU kernel on a GPU device. Upon receiving the request from a VM, the manager verifies if it has the latest information about the status of GPUs in the cluster from the local data structures. If required, the manager communicates with GPU Controllers to get the up-to-date information about the state, number of executing kernels, available memory, and global memory usage at each GPU and cloud node in the cluster. The GPU Scheduler uses this information to make scheduling decisions for the requested GPU kernel. After selecting the suitable GPU, the manager sends a message to the corresponding GPU Controller to run the given kernel. When kernel execution is completed, the manager receives a completion message from the GPU Controller and notifies the application about the completion of the given kernel.

2) Cloud Node Operations: gCloud runs the GPU Controller and Communicator components at each cloud

node. The GPU Controller runs as a daemon process on the GPU-enabled cloud node, and is responsible for managing local GPUs. It uses the CUDA driver APIs to initialize and mange GPU devices. At the initialization of gCloud, GPU Controller fetches the information, e.g., device memory, driver version, and the CUDA device computing capability version, about the GPU devices available to the node, and sends this information to the manager node. The GPU Controller also creates one CUDA context for each physical GPU at the initialization of the system, and uses this context to execute multiple kernels on the particular GPU device.

After initializing local GPU devices, the GPU Controller waits for the message from the GPU Scheduler to execute a given kernel (Figure 2). The execution is carried out in a separate thread, which is provided with the pre-initialized device context by the GPU Controller to run the given kernel. The execution message contains the unique identifier of the GPU kernel, the globally accessible path of the shared object that contains the kernel, the location of the data that may be provided as input to the kernel, and the id of the GPU device on which this kernel should be executed. The execution thread uses this information to dynamically load the required kernel from the shared object and execute it with the device context of the specified GPU. The execution thread terminates once the kernel execution completes, and a completion message along with the kernel execution profile is sent to the GPU Scheduler. It is important to note that the maximum number of concurrent kernels on a GPU device in our system does not exceed the maximum number of concurrent kernels supported on that GPU device, which is 16 for the NVIDIA Fermi GPUs [15].

3) Communication between Cluster Nodes: The Communicator component provides the communication mechanisms between the manager and the cloud nodes, and is a common component in these nodes. This communication is internal to gCloud, and it remains transparent to the hosted VM. We have used client-server communication primitives using the TCP/IP protocol [16] to implement a light-weight communication mechanism between the cluster nodes. We use pre-established TCP connections in our implementation to reduce the communication latencies and overhead.

B. gCloud APIs

From the application perspective, gCloud requires minimal modifications and provides three APIs to register, execute and unregister a compute kernel. gCloud generates a unique identifier for each kernel using the OpenStack id of the VM, the shared object and the kernel names, and returns it after successfully registering the kernel to the application. The application uses this identifier to execute the kernel on GPUs. Note that gCloud supports registering and executing multiple kernels from the same application at the same time. For unregistering a kernel, the user provides gCloud with the identifier of the kernel, and gCloud unregisters and removes the target shared object.

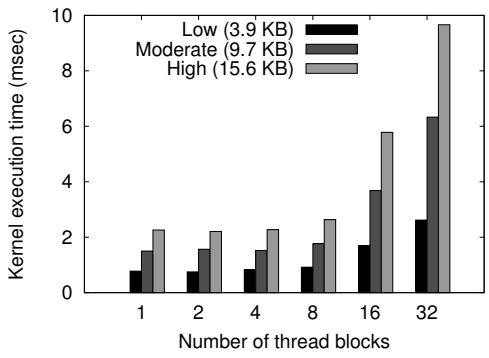
For executing multiple GPU kernels concurrently with each other, gCloud assumes the target GPU kernels to be compiled as self-contained shared objects. All operations related to data manipulations, such as reading input data, moving data between host and GPU memory, and handling results, are performed inside the self-contained shared object. The user operations to manage the input and output data associated with the GPU kernel are enclosed in the shared object and are managed by the user. This maintains the integrity and security of the user data by not exposing it to the outside world.

gCloud does not impose any restrictions on GPU applications. All NVIDIA CUDA idioms are supported, since gCloud provides application with a context in which any CUDA idioms can be executed. However, gCloud assumes that the shared objects do not use any device specific operations, e.g., `cudaSetDevice`, `cudaDeviceReset`, and `cudaChooseDevice`, to select a specific device for executing the GPU kernel. GPU kernels should be launched from the shared objects assuming that the GPU device has already been initialized and is ready for executing the given kernel.

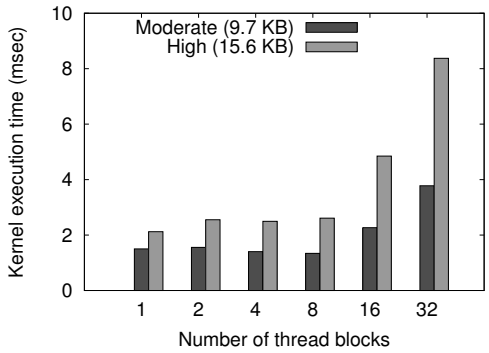
IV. Scheduling Concurrent Kernels

This section presents the proposed scheduling scheme to run concurrent GPU kernels on distributed GPU devices.

We conduct several experiments to measure the impact of the local memory size, number of threads and number of thread blocks on the GPU kernel execution time for executing concurrent kernels on the same GPU device using NVIDIA Tesla C2075 that has 448 CUDA cores and 6 GB device memory with the CUDA computing capability version 2.0 [2]. We use a custom GPU kernel such that the local memory usage can be altered without impacting any other factor, such as the semantics or the complexity, of the GPU kernel. The kernel allocates an array in local memory, initializes it, and then computes and returns the sum of its elements. We use three instances (*low*, *moderate*, and *high*) of the used kernel, where each instance has (3.9 KB, 9.7 KB, and 15.6 KB) of local memory size, respectively. We run these kernels with dedicated GPU access to see the impact of the number of threads and the number of thread blocks on the execution time. We also run two of these kernels concurrently with each other on the same GPU to study the impact of concurrent execution on their execution



(a) 1024 threads/block



(b) Moderate/High local memory requirements

Fig. 3: Impact on the execution time in case of standalone and concurrent execution when varying the number of thread blocks on kernels with different local memory requirements.

time. We observe that increasing the number of threads and the thread blocks for the *low*, *moderate*, and *high* local memory usage leads to an increase in the execution time for these kernels as shown in Figure 3(a).

We conducted experiments to determine the impact of the local memory size of a GPU kernel on the execution time when GPU is shared between multiple applications using the same kernels described before. For concurrently executing two kernels, we executed the combinations of *low*, *moderate*, and *high* kernels on the same GPU device. We observe that execution time with increasing local memory size is high when running these kernels concurrently with each other, especially for the scenario when local memory, number of threads and the number of thread blocks requirements of the concurrent kernels are high as shown in Figure 3(b).

We use these observations to schedule and share GPUs between multiple applications for concurrent execution. Our scheduling algorithm first tries to find an idle GPU device to balance the kernel load across all GPU devices. This ensures minimal interference for that kernel since it is scheduled on a dedicated device. If there is no idle device available, then devices that meet the minimum memory requirements of the input kernel are selected as potential candidates to execute the kernel. The GPU device from the list of candidate devices that gives the minimum interference if the input kernel is executed on it is selected.

Application	Input Size	GPU Utilization (%)	
		Memory	Processing
Binomial Options (BO)	512 MB	1.33	8
Eigenvalue (EV)	4 M points	0.96	99
Black Scholes (BS)	8 M stocks	2.18	67
Sorting Network (SN)	1 M points	0.93	1

TABLE I: Benchmark applications with corresponding input sizes and GPU utilization.

V. Evaluation

We evaluate gCloud using representative GPU applications. We first show the effectiveness of gCloud in concurrently executing multiple applications on the same GPU, and then we evaluate its performance in a distributed setting with the OpenStack cloud computing platform. Finally, we compare our interference-aware scheduling scheme with a round-robin scheduling scheme for executing concurrent kernels on distributed GPUs.

A. Experimental Setup

1) Cloud Setup: We evaluate gCloud on a three-node cluster where one of the cluster node is a dedicated front end node for managing the cloud computing setup and running the cloud controller. The remaining two nodes are GPU-enabled cloud nodes and they host virtual machines. Each cloud node has two high-end NVIDIA Tesla C2075 GPUs [2] with 6 GB of dedicated memory installed on them. Each node in our cluster has two Intel Xeon E5645 [17] processors, 12 GB of main memory, and 1000 GB of local storage. All nodes are connected using a 10 G Ethernet network. We use NFS [14] in our setup to share the library objects that encapsulate GPU kernels and their corresponding input data.

2) GPU Benchmark Applications: We use 4 benchmark applications, shown in Table I, to evaluate gCloud. We have selected these applications because they are diverse and represent workloads [4], [18] commonly offloaded to GPUs from scientific and enterprise applications. We have modified these applications from the NVIDIA SDK package to use gCloud, and each application required about 4 lines of code modification to use gCloud. We run these applications in isolation of each other using dedicated GPU access, which is the current method used in commercial cloud solutions. We also run these benchmark applications concurrently on GPUs using our framework. Moreover, when a large number of applications are required in an experiment,

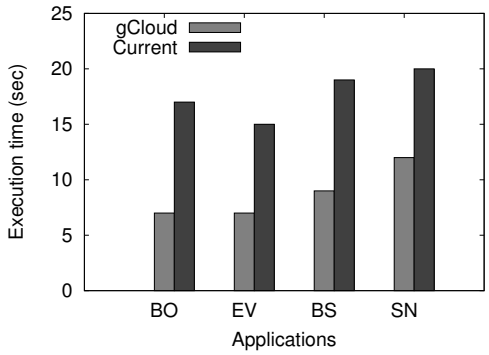


Fig. 4: Performance comparison of *current* approach and gCloud for concurrently executing two instances of same application.

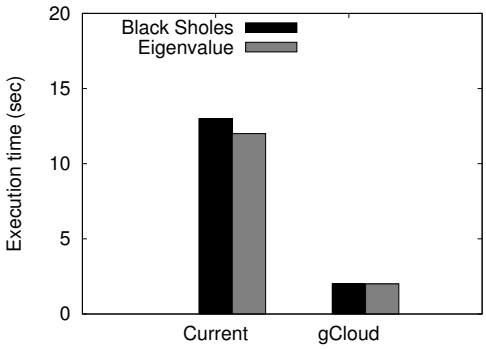


Fig. 5: Black Scholes - Eigenvalue

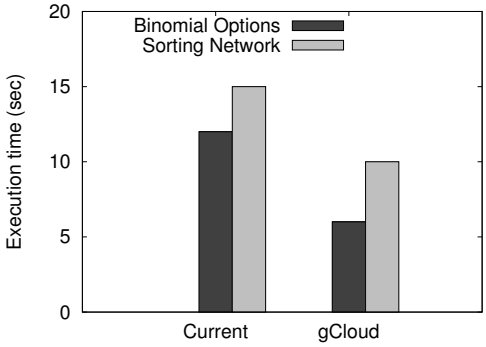


Fig. 6: Binomial Options - Sorting Network.

we replicate these applications to create unique instances that request GPU resources independent of each other. We have used `nvidia-smi` [19] for finding the GPU utilization by each application. The low GPU utilization for the studied benchmark applications indicates the need for GPU sharing to fully utilize expensive GPU resources.

3) Performance Metrics: We use the overall application execution time as the performance metric to compare gCloud with other approaches. We measure the application execution time from the time when the application starts execution to the time when it completes execution. For concurrently executing multiple applications, we measure the time required to complete the execution of all applications as a batch job. To account for experimental errors, the presented numbers in our evaluation represent averages over three runs. Since we use a dedicated cloud setup for our evaluation, we do not observe high variance in the numbers between multiple runs of the same experiment.

4) Systems Compared Against: We compare the performance of gCloud in executing concurrent kernels with the "current" approach, which serializes the executions of given kernels. When multiple kernels are executed using *current* approach, each kernel gets the dedicated access to the GPU, i.e., multiple kernels are executed one after the other. For studying the importance of interference-aware scheduling, we compare gCloud with *round-robin* approach, which uses round-robin strategy to select a GPU to execute the given kernel concurrently with other kernels.

B. Performance Gain from GPU Sharing

In this set of experiments, we study the performance gain of sharing a GPU among concurrent applications.

1) Concurrently Executing Two Instances of the Same Application: Figure 4 shows the result of running two instances of the same application concurrently on a GPU using gCloud, and its comparison with the *current* approach. The *current* approach serializes the execution of concurrent application, and each application gets the dedicated GPU access. However, each application execution in *current* approach incurs additional overhead of initializing the GPU context. We observe the performance improvement in the overall execution time using gCloud since it allows concurrent execution and alleviates the application from waiting in the queue to get the dedicated GPU access. On average, gCloud achieves 59.8% improvement across all applications compared to the *current* approach.

2) Concurrently Executing Two Instances of Different Applications: We study the gain on the application execution time for running two different applications concurrently on a single GPU with gCloud and compare its performance with the *current* approach. We use all possible combinations of running two different applications from our benchmark applications and execute them concurrently on one of the GPUs in our cloud setup. We consider jobs from multiple applications as a single batch job, and measure the time required to execute the batch job using gCloud and *current* approaches. Figure 5 and Figure 6 summarize the results of this experiment. gCloud consistently reduces the application execution

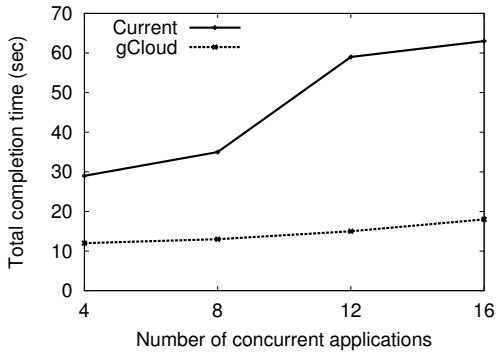


Fig. 7: Impact of increasing number of applications on application completion time.

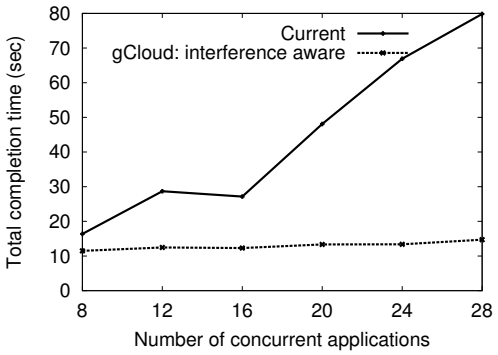


Fig. 8: Performance comparison of gCloud with the *current* state-of-the-art.

time for each combination for two different applications. We observe the maximum speedup of $5.66\times$ in the execution time using gCloud for running Black Scholes and Eigenvalue concurrently. The speedup occurs because the applications use a GPU context that is pre-initialized by gCloud. Similarly, the minimum speedup of $1.62\times$ is achieved using gCloud for running Binomial Options and Sorting Network concurrently. This small speedup in the case of Binomial Options and Sorting Network is because of the contention for accessing the global memory.

3) Sharing Scalability: We study the impact of sharing a GPU on the application performance with the increasing number of applications using gCloud and compare it with the *current* execution scheme. Figure 7 shows the results of this experiment. For executing more than 4 applications concurrently, we replicate our benchmark applications, e.g., for executing 8 concurrent applications we launch two instances of each benchmark application. Since the maximum number of concurrent kernels that can be executed in the NVIDIA Tesla GPU is 16 [15], we increase the number of applications to up to 16 in this experiment. We find a linear increase in the application execution time using gCloud showing that it scales linearly with increasing the number of concurrent applications executing on the same GPU. In the case of *current* execution scheme for running multiple applications, the application execution time increases rapidly with the increasing number of applications because the default NVIDIA driver serializes the application execution and therefore each application has to wait longer for its share of the GPU. We achieve $3.93\times$ maximum performance gain using gCloud for concurrently executing 12 applications compared to the *current* execution scheme. On average, gCloud achieves $3.13\times$ performance gains across all runs compared to the *current* execution scheme.

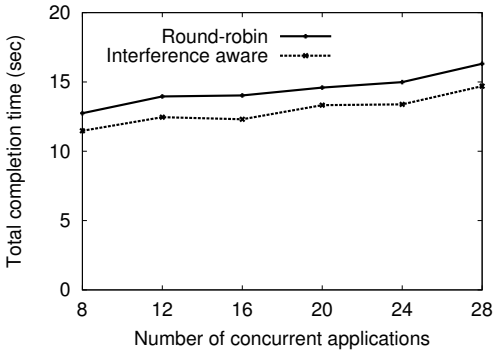


Fig. 9: Performance comparison of gCloud with round-robin scheduling schemes.

C. Distributed GPU Sharing/Scheduling

We compare the performance of our scheduler with the *current* state-of-the-art scheduling that serializes the execution of concurrent GPU kernels in the distributed setting using 4 GPU devices in our setup. Figure 8 shows the performance comparison of our scheduling scheme with the *current* scheme with increasing number of concurrent applications. We observe a performance gain of up to $5.43\times$ using our scheduling framework compared to the *current* scheme. gCloud outperforms the *current* scheme by 70.9% on average across all data points because the *current* scheduler serializes the GPU kernel executions without exploiting the concurrent execution features of the GPUs.

D. Importance of Interference-Aware Scheduling

We compare our scheduling scheme with a sharing-enabled round-robin scheduling scheme that incurs no scheduling overhead. The *round-robin* scheme shares GPUs between multiple applications for concurrent execution using round-robin scheduling scheme to select a GPU for the next request. The difference between our interference-aware scheduling scheme and the

sharing-enabled round-robin scheduling scheme is that our scheme incorporates the information about the inter-application interference for GPU resources in making scheduling decisions. Whereas in *round-robin* scheme, the scheduler is agnostic to the inter-application interference for GPU resources, and selects a GPU for the next kernel in a round-robin fashion.

Figure 9 shows the performance comparison of our scheduling scheme with the *round-robin* scheme with increasing number of concurrent applications in the cloud setup. On average our scheduling scheme performs 10.3% better than the *round-robin* scheme because gCloud incorporates the information about the inter-application interference for the concurrent GPU kernels. Using round-robin scheduling scheme, we have observed that the completion time for two kernels that are incompatible with each other is almost similar to the completion time for these kernels when executed serially with each other.

VI. Conclusions

We presented a framework, called gCloud, that enables on-demand GPU access and sharing among multiple applications in cloud computing environments. It incorporates the local and global memory, number of threads, and number of thread blocks requirements of each GPU kernel in making the consolidation decisions. such that the inter-application interference across the cluster is minimal. gCloud registers and executes GPU kernels from multiple applications concurrently on distributed GPUs. We implemented and integrated gCloud in the OpenStack cloud computing environment and showed that it improves the utilization of GPU resources by 56.3% on average compared to the current state-of-the-art scheme. We showed a speedup of up to $5.66\times$ in the application completion time using gCloud compared to the current scheme. The dynamic sharing of GPUs using gCloud, and the proposed consolidation policy reduce the overall application time by 10.3% on average, compared to the round-robin based consolidation policy.

References

[1] Amazon, "Amazon Elastic Compute Cloud (Amazon EC2)," http://www.amazon.com/b?ie=UTF8&node=201590011.

[2] NVIDIA Corporation, "NVIDIA TESLA C2075 COMPANION PROCESSOR," 2011, http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf.

[3] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, ser. EuroPar'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 379–391.

[4] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *Proceedings of the 20th international symposium on High performance distributed computing*, ser. HPDC'11. New York, NY, USA: ACM, 2011, pp. 217–228.

[5] L. Wang, M. Huang, and T. El-Ghazawi, "Exploiting concurrent kernel execution on graphic processing units," in *Proceedings of International Conference on High Performance Computing and Simulation (HPCS '11)*. Istanbul, turkey: IEEE, July 2011, pp. 24 –32.

[6] Rackspace Cloud Computing, "OpenStack Compute: An Overview," 2012, http://openstack.org/downloads/openstack-compute-datasheet.pdf.

[7] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar, "Resource Sharing in GPU-Accelerated Windowing Systems," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, april 2011, pp. 191–200.

[8] T. Li, V. Narayana, E. El-Araby, and T. El-Ghazawi, "GPU Resource Sharing and Virtualization on High Performance Computing Systems," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, Sept. 2011, pp. 733 –742.

[9] J. Duato, A. J. Pena, F. Silla, J. C. Fernandez, R. Mayo, and E. S. Quintana-Orti, "Enabling CUDA acceleration within virtual machines using rCUDA," in *Proceedings of the International Conference on High-Performance Computing*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 1–10.

[10] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, ser. SC '04. Washington, DC, USA: IEEE Computer Society, 2004.

[11] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos, "A capabilities-aware framework for using computational accelerators in data-intensive computing," *J. Parallel Distrib. Comput.*, vol. 71, no. 2, pp. 185–197, February 2011.

[12] M. Curry, A. Skjellum, H. Ward, and R. Brightwell, "Arbitrary dimension Reed-Solomon coding and decoding for extended RAID on GPUs," in *Proceedings of the 3rd Petascale Data Storage Workshop(PDSW '08)*, Nov. 2008.

[13] M. Fatica, "Accelerating LINPACK with CUDA on heterogenous clusters," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 46–51.

[14] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun network file system," in *Proc. Summer USENIX*, Portland, OR, June 1985, pp. 119–130.

[15] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[16] W. R. Stevens, *TCP/IP illustrated (vol. 1): the protocols*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993.

[17] Intel Corporation, "Intel Xeon Processor E5645," 2012, http://ark.intel.com/products/48768/.

[18] M. M. Rafique, S. Cadambi, K. Rao, A. R. Butt, and S. Chakradhar, "Symphony: A Scheduler for Client-Server Applications on Coprocessor-Based Heterogeneous Clusters," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, ser. CLUSTER '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 353–362.

[19] NVIDIA Corporation, "Tools for Managing Clusters of NVIDIA GPUs," 2010, http://www.nvidia.com/content/GTC-2010/pdfs/2225_GTC2010.pdf.