GScheduler: Optimizing Resource Provision by using GPU Usage Pattern Extraction in Cloud Environment

Zhuqing Xu
Southeast University

Fang Dong
Southeast University, fdong@seu.edu.cn

Jiahui Jin
Southeast University

Junzhou Luo
Southeast University

Jun Shen
University of Wollongong, jshen@uow.edu.au

GScheduler: Optimizing Resource Provision by using GPU Usage Pattern Extraction in Cloud Environment

Abstract

GPU-based clusters are widely chosen for accelerating a variety of scientific applications in high-end cloud environments. With their growing popularity, there is a necessity for improving the system throughput and decreasing the turnaround time for co-executing applications on the same GPU device. However, resource contention among multiple applications on a multi-tasked GPU leads to the performance degradation of applications. Previous works are not accurate enough to learn the characteristics of GPU application before execution, or cannot get such information timely, which may lead to misleading scheduling decisions. In this paper, we present GScheduler, a framework to detect and reduce interference for co-executing applications on the GPU-based cloud. The most important feature of GScheduler is to utilize GPU usage pattern extractor for detecting interference between applications. It is composed of key function-call graph extractor and key GPU resource usage vector extractor, the former is used to detect the similarity of GPU usage mode between applications, while the latter is used to calculate the similarity of GPU resource requirements in-between. In addition, an interference aware scheduler is proposed to minimize the interference. We evaluated our framework with 26 diverse, realworld CUDA applications. When compared with state-of the-art interference-oblivious schedulers, our framework improves system throughput by 36% on average, and achieves a 30.5% reduction of turnaround time on average.

Keywords

pattern, extraction, gscheduler:, optimizing, resource, provision, cloud, gpu, environment, usage

Disciplines

Engineering | Science and Technology Studies

Publication Details

Xu, Z., Dong, F., Jin, J., Luo, J. & Shen, J. (2017). GScheduler: Optimizing Resource Provision by using GPU Usage Pattern Extraction in Cloud Environment. IEEE International Conference on Systems, Man and Cybernetics (pp. 3225-3230). IEEE.

GScheduler: Optimizing Resource Provision by Using GPU Usage Pattern Extraction in Cloud Environments

Zhuqing Xu[1], Fang Dong[1], Jiahui Jin[1], Junzhou Luo[1], Jun Shen[2]

[1]School of Computer Science and Engineering, Southeast University, Nanjing, P.R.China
[2]School of Computing and Information Technology, University of Wollongong, NSW, Australia
Email: {zqxu, fdong, jjin, jluo}@seu.edu.cn, jshen@uow.edu.au

***Abstract*—GPU-based clusters are widely chosen for accelerating a variety of scientific applications in high-end cloud environments. With their growing popularity, there is a necessity for improving the system throughput and decreasing the turnaround time for co-executing applications on the same GPU device. However, resource contention among multiple applications on a multi-tasked GPU leads to the performance degradation of applications. Previous works are not accurate enough to learn the characteristics of GPU application before execution, or cannot get such information timely, which may lead to misleading scheduling decisions. In this paper, we present *GScheduler*, a framework to detect and reduce interference for co-executing applications on the GPU-based cloud. The most important feature of *GScheduler* is to utilize *GPU usage pattern* extractor for detecting interference between applications. It is composed of key function-call graph extractor and key GPU resource usage vector extractor, the former is used to detect the similarity of GPU usage mode between applications, while the latter is used to calculate the similarity of GPU resource requirements in-between. In addition, an interference aware scheduler is proposed to minimize the interference. We evaluated our framework with 26 diverse, real-world CUDA applications. When compared with state-of the-art interference-oblivious schedulers, our framework improves system throughput by 36% on average, and achieves a 30.5% reduction of turnaround time on average.**

Keywords—GPU; cloud computing; CUDA; task scheduling

I. INTRODUCTION

Using GPU to accelerate the computationally intensive workloads, such as scientific computing [1], image processing [2], data mining [3] and searching [4], has become more and more popular. Many of the large-scale cloud providers, such as Amazon EC2 [5], Nimbix [6], Peer1 Hosting [7] and Penguin Computing [8], now offer GPU services.

However, the usage effectiveness of GPUs in such cloud environments suffers from long turnaround time and low system throughput, since it is limited by static provisioning of GPU resources [9, 10]. Applications are typically assigned to GPUs in a static mode, which implies that, during the execution of applications, the dedicated access (i.e. GPU pass-through) to the GPU is offered. This dedicated access will result in over provisioning of GPU resources. When multiple applications contend for the GPU resources, the turnaround time of applications will be increased and the overall system throughput will be decreased.

In order to optimize resource provisioning, so as to decrease turnaround time and improve system throughput, one approach is to schedule multiple applications on multiple GPU compute nodes. Nevertheless, the scenarios that multiple applications run on the same multi-tasked GPU device may lead to performance degradation for one or more applications [9, 11]. The performance degradation is caused by GPU resource contention among co-executing applications. Therefore, to optimize resource provisioning, it is crucial to:

1) Obtain the characteristics of GPU applications before the actual execution. The characteristics of GPU applications refer to the GPU usage model, GPU resource demand of applications, such as invoking GPU memory copy functions, GPU memory allocation functions, executing GPU kernel functions, blocks, threads, shared memory, and registers, etc. The purpose to discover the characteristics of GPU applications is to guide the scheduler to assign applications to the appropriate GPU compute node. It is considered to be inefficient and intolerable that the characteristics of GPU applications are captured during its execution rather than before running, because the application is likely to be dynamically suspended or reassigned to the other GPU compute nodes according to the current system status.

2) Explore some scheduling strategies. The aim of scheduling is to assign the incoming application to an appropriate GPU device, and reduce the resource contention among co-executing applications on the same GPU device. Scheduling strategy is closely related to the acquisition of GPU application characteristics.

The effort to acquire the characteristics of GPU applications has been advanced over past few years. One of the existing approaches is to use a profiling tool, such as CUPTI [12], PAPI [13], Tau [14], Vampir [15]. The other existing approach, such as Mystic [9], is to predict the characteristics of incoming GPU applications according to a priori empirical application or previously executed application corpus.

However, when an application arrives, we need to decide which GPU the application should be scheduled to run on. The above methods either cannot acquire the accurate characteristics of GPU applications before execution, or introduce too much additional overhead and increase the turnaround time. Therefore, they cannot meet our requirements.

After all, it might be difficult to obtain the characteristics of GPU applications before the actual execution, and to schedule multiple applications on multiple GPU devices.

In this paper, we use *GPU usage pattern* to represent the characteristics of applications. *GPU usage pattern* is represented by 1) a key function-call graph (directed graph) and 2) a key GPU resource usage vector (7-tuple). Each vertex in key function-call graph indicates a pivotal CUDA activity (introduced in section III), such as GPU kernel function execution, GPU memory allocation, Host-to-Device memory copy and Device-to-Host memory copy, etc. The key GPU resource usage vector expresses GPU resource requirements of an application, which includes the usage information of blocks, threads, global memory, registers, shared memory, constant memory, and local memory.

GPU usage pattern describes how the key functions (illustrated in Section III) are invoked and how many GPU resources are needed for an application. It accurately characterizes the GPU usage information of an application. By using *GPU usage pattern* information, it is possible to reduce resource contention among co-executing applications.

We present *GScheduler*, a framework to detect and reduce interference for co-executing applications on the GPU-based cloud. It's designed as a 3-stage control layer, which is mainly hosted in the head node. First of all, the key function-call graph extractor is presented in Stage 1. It is used to extract key function-call graph by an intermediate code file. Some codes should be inserted into the original intermediate code file to overwrite the key functions. With the execution of updated intermediate code file, the Graph Constructor in Stage 1 will generate the key function-call graph. Secondly, the key GPU resource usage extractor is designed in Stage 2. It can extract the PTX (illustrated in Section II) information from the executable file. We parse the GPU resource usage of GPU kernel functions by using *ptxas* command. Lastly, the interference aware scheduler is launched in Stage 3. The similarity score of key function-call graph between applications can be calculated, and the similarity score of key GPU resource usage vector between them can also be obtained. Depending on these data, the interference between two applications can be measured. Then, the scheduler can assign an application to the GPU compute node with the lowest interference. In a word, when an application arrives at the head node, *GPU usage pattern* can be obtained with the above method, and the scheduling algorithm is used to assign applications to the appropriate GPU compute node.

To summarize, the original contributions of this work include the following:

1) Algorithm for extracting *GPU usage pattern* is proposed, which can be used separately to capture the characteristics of GPU applications before execution.

2) Algorithm for measuring interference between two applications is proposed, which can be used to indicate the degree of GPU resource contention.

3) *GScheduler* framework, which is composed of *GPU usage pattern* extractor and interference aware scheduler, is implemented. It is readily deployable within current data centers without hardware modification. Extensive evaluations on the specific test bed are conducted (in Section IV).

The rest of the paper is organized as follows. Section II describes the background and basic techniques to implement *GScheduler*. Section III illustrates the system architecture and the detailed designs of each module. Section IV describes the evaluation methodology and experimental results. Section V describes related work. Section VI briefly concludes the paper.

II. Background

A. CUDA

CUDA (Compute Unified Device Architecture) is a GPU parallel computing platform and programming model which is launched by the graphics vendor NVIDIA. CUDA proposes a concept of thread grid. A grid is a collection of thread blocks which is capable of executing a kernel function. Each grid is composed of multiple thread blocks, and each block consists of multiple threads. All threads in a thread block are executed concurrently. All threads in one block are executed on one SM. The number of blocks that can run on the same SM depends on the resource requirements of each block, such as the number of registers and shared memory.

The number of blocks per grid and the number of threads per block are specified by the programmer. In addition, programmers also need to implement GPU kernel function, which is used to appoint the way of parallel execution and collaboration among threads.

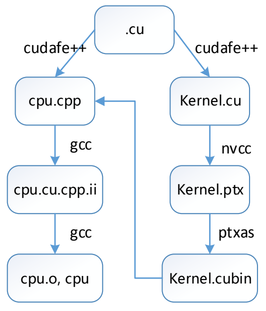

Fig. 1. CUDA Program Compilation Trajectory

B. Intermediate Code

At present, CUDA supports Java, Python and other high-level programming languages. There are more and more GPU applications being written by them. These interpreted languages have a feature that the source code is translated to bytecode instead of machine code. The bytecode is an intermediate code which can be edited and executed. Therefore, we can modify the intermediate code file by overwriting some of GPU key functions (see section III), execute the updated intermediate code file to acquire the *GPU usage pattern*, calculate interference between two applications, and conduct scheduling. For a GPU application written by CUDA C/C++, if the intermediate code file with suffix "*.cu.cpp.ii*" is given, the *GPU usage pattern* and interference score between two applications can be acquired in the same way. Taking GPU applications written by CUDA C/C++ as an example, we will illustrate the design and implementation of *GScheduler*. The idea of our framework is also suitable for CUDA Java/Python applications.

C. *CUDA Program Compilation Trajectory*

The CUDA compilation trajectory is represented in Fig. 1. The source code is located in "*.cu*" file. The NVIDIA compiler is responsible for compiling the GPU kernel functions into the corresponding GPU binary code. The GPU kernel function can be compiled to the PTX code with *nvcc* command, and the PTX code can be compiled to CUDA binary file with *ptxas* command. On the other hand, GPU binary code is loaded into the C/C++ code, and finally compiled into executable files by using *gcc*.

As shown in Fig. 1, the intermediate code file ("*.cu.cpp.ii*") that we used, have not included the source codes of GPU kernel functions. Moreover, as a matter of experiences, the core of the GPU application is the GPU kernel functions, host codes just perform some basic work for GPU device. Therefore, even if the user provides the intermediate code file, it will not bring the risk of code leakage.

D. *The Rationality of GPU Usage Pattern Extraction*

It is advisable to obtain the characteristics of the incoming GPU application before scheduling for it. We define the characteristics of GPU application as *GPU usage pattern*. *GPU usage pattern* is acquired by extracting the key function-call graph and key GPU resource usage of an application.

By observing and running CUDA applications, we can find the key feature of GPU applications: For a general GPU application, CPU occupancy time is far less than GPU occupancy time. On the other hand, the PTX code can be extracted by the *cuobjdump* command, and the GPU resources usage information about the function can be required through the *ptxas* command.

As a result, the basic technical steps include: 1) modify the intermediate code file by overwriting GPU key function (in section III), 2) compile and execute the updated intermediate code file to obtain *GPU usage pattern*.

III. GSCHEDULER ARCHITECTURE

A. *System Overview*

GScheduler is designed as a 3-stage control layer, mainly deployed in the head node of a GPU based cloud environment. The framework is capable of calculating the interferences between the incoming application and the currently running applications in the cluster. *GScheduler* schedules an incoming application to the compute node with the lowest interference. Fig. 2 shows the *GScheduler* architecture.

B. *Stage 1: Key function-call Graph Extractor*

The key functions refer to the functions which will impact GPU resource allocation and usage. In general, the usage of GPU resources mainly includes computing (by invoking GPU kernel functions) and GPU memory operations, such as GPU memory allocation, GPU memory copy. In addition, synchronous operations has a great impact on the key function-call graph, so that these functions are also of concern. Herein, we select these functions which are listed in Table I as the key functions.

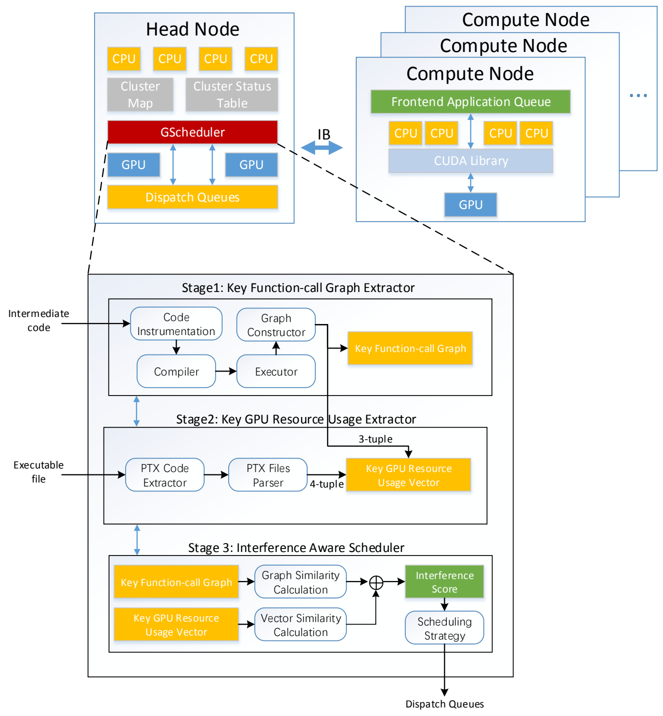

Fig. 2: System Architecture

Code instrumentation. We use the intermediate code file with the format of "*.cu.cpp.ii*" as the input of key function-call graph extractor. It has hidden the source code of GPU kernel functions. The main behavior of code instrumentation is to 1) replace the key functions as standard output functions and 2) insert some specific control code into the intermediate code file.

In 1), for each key function, function name and parameter(s) are outputted instead of executing it. The aim of 1) is to prevent the key functions from running, thus drastically reducing the time of extracting the application's characteristics.

In 2), an additional exit condition is inserted into each cycle. Whether the exit condition is satisfied is associated with the content of a file. The Graph Constructor (see below) will fill the file according to its analysis in real time. The purpose of 2) is to handle the case of dead cycles.

Table I: GPU *key functions* used in the extractor

Type	Function name
Memory allocation*	cudaMalloc
Memory operations*	cudaMemcpy, cudaMemcpyAsync, cudaMemcpyToSymbol, cudaMemcpyToSymbolAsync, cudaMemset, cudaMemsetAsync
Computing	cudaConfigureCall
Synchronous operations	cudaDeviceSynchronize, cudaThreadSynchronize, cudaStreamSynchronize

* For the types of memory allocation and memory operations, only partial functions are supported at present.

Compiler. When updated intermediate code file is generated, we need to compile it. The "*.cu.cpp.ii*" file can be compiled to the "*.obj/.o*" file by using "*nvcc –c*" command. After that, we can use the *nvcc* command to generate an executable file further.

Graph Constructor. The Graph Constructor is designed to construct a key function-call graph according to the invoked key functions in the Table I.

Firstly, we illustrate the algorithm of constructing the key function-call graph. The key to construct the key-function-call graph is to identify the relationships between two adjacent functions, such as father-son relationship, sibling relationship, and aggregate relationship. It is described in algorithm 1.

Algorithm 1 Construct the key-function-call graph

```
1:  Input: Function Line A, Function Line B (B is the next line of A)
2:  If isSynchronize(B)=true then aggregate relationship
3:  Else if streamParameter(B)=null && streamParameter(A)=null
        father-son relationship
4:  Else if streamParameter(B)!=null && streamParameter(A)=null
        add branch point A; father-son relationship
5:  Else if streamParameter(B)=null && streamParameter(A)!=null
        aggregate relationship; remove corresponding branch point
6:  Else if streamParameter(B)!=null && streamParameter(A)!=null
        If stream(A)=stream(B) then father-son relationship
        Else sibling relationship
```

Next, we discuss the specific case where the application may sink into dead cycle. The structured programming is composed of sequence, selection and iteration [16]. Hence, dead cycle must occur in iteration pattern where the iteration condition value depends on the execution result of GPU kernel functions. It is inevitable that a sequence of GPU kernel functions are invoked periodically in the dead cycle. Hence, we can identify and avoid the dead cycle according to the parameters passed to these functions.

C. Stage 2: Key GPU Resource Usage Extractor

The aim of key GPU resource usage extractor is to obtain the GPU resource demand information for an application, which includes blocks, threads, global memory, registers, shared memory, constant memory and local memory.

As shown in Fig. 2, the PTX Code Extractor will extract the PTX code from the executable file. In other words, it extracts all of the GPU kernel functions invoked in the application and presents in the format of PTX code. The core of the extractor is to use the *cuobjdump* command in NVIDIA CUDA Toolkit [17]. It can dump PTX files for all listed device functions.

The PTX Files Parser can parse PTX files and obtain the properties including registers, shared memory, constant memory and local memory. The key of the parser is to use *ptxas* command in NVIDIA CUDA Toolkit. The parser first processes each PTX input file with *ptxas* command. Then, it captures each resource usage information according to the output format of the command. Accumulating corresponding resource usage data is required if there are more than one PTX file.

In addition, the remaining three GPU resource usage information: blocks, threads, and global memory are captured in the stage 1. Blocks number and threads number can be obtained in the *cudaConfigureCall* function. The global memory usage information can be acquired from the *cudaMalloc* function.

D. Stage 3: Interference Aware Scheduler

The Graph similarity metric. The similarity calculation procedure between two graphs is as follows: First of all, we use the node similarity update function suggested by Zager et al. in [18] to calculate the node similarity:

$$x_k \leftarrow (A \otimes B + A^T \otimes B^T + D_{A_S} \otimes D_{B_S} + D_{A_T} \otimes D_{B_T})x_{k-2}$$

where x_k denotes the node similarity matrix after *k* iterations. A and B are the standard node-node adjacency matrices of G_A and G_B, respectively. D_{A_S} (D_{A_T}) is a diagonal matrix for graph A that satisfies the following conditions: the diagonal element in the *i*-th diagonal entry is the out-degree (in-degree) of node *i*. D_{B_S} and D_{B_T} may be deduced by analogy. The $\otimes$ notation denotes the Kronecker product of matrices. The $\leftarrow$ notation indicates the normalization by the Frobenius norm at each stage.

In the case of key function-call graph matching, nodes can be divided into different types. Each node represents a function call such as *cudaMalloc*, *cudaMemcpy*, etc. The nodes that invoke the same function share the same type, and they can match each other, vice versa. Once the node similarity matrix between G_A and G_B is computed, the element values between different types of nodes are assigned a negative number.

Next, we need to find the optimal matching between the respective elements of two graphs by using the node similarity matrix. Let's treat two matching graphs as a partition of bipartite graph, respectively. The two partitions of bipartite graph are connected by the edges. The weights of the edges can be found in the corresponding elements of the node similarity matrix. Accordingly, graph matching problem is converted into the maximum weight matching problem. Thus, optimal matching can be obtained by using Kuhn-Munkres (KM) algorithm [19].

Finally, a similarity score is introduced by the result of graph matching. We define similarity score between G_A and G_B as follows:

$$\text{gscore} = \sum_{i=1}^{n} M_i^2$$

where M denotes the node similarity matrix.

GPU resource usage similarity metric. As described earlier in this section, the key GPU resource usage is a vector consisting of 7 elements. We use cosine similarity to measure the similarity of GPU resource requirements between two applications. For convenience, we use *vscore* to represent $\text{sim}(\vec{V_i}, \vec{V_j})$.

Interference aware scheduling strategy. Graph similarity score (*gscore*) represents the similarity of using GPU between two applications, and vector similarity score (*vscore*) indicates the similarity of GPU resource requirements between them. Once these metrics are calculated, the interference score can be obtained by adding *gscore* and *vscore* proportionally. According to the experiments, we set the proportion to 0.5. The higher the interference score value is, the higher interference probability between two applications will be.

Next, the interference aware scheduling strategy is made: The incoming application is assigned priority to the idle GPU compute node. Otherwise, the scheduler will assign application to the compute node with minimal interference score.

IV. Experimental Evaluation

A. *Experimental Setup*

In order to evaluate the performance and effectiveness of the *GScheduler* framework, we have deployed it on a private GPU cluster of high performance computing center at Southeast University. The cluster system consists of a head node and 4 compute nodes, and each of them has the same configuration.

The *GScheduler* framework is mainly deployed on the head node of the cluster, and it will extract *GPU usage pattern* of each incoming application by analyzing its intermediate code file, schedule the application to the compute node with the lowest interference score. A daemon is deployed on each compute node to monitor the system running state, such as GPU utilization, the running time of each application, etc. The head node periodically communicates with each compute node to obtain the status of each node.

B. *Experimental Benchmarks*

We select 26 typical applications as the workloads of the experiments. They were from the classic benchmark suites of NVIDIA SDK (11 apps.) [17], Rodinia (15 apps.) [20]. These 26 applications arrive at the cluster, which has up to 26! arrival sequences. It's impractical to schedule all of these sequences in a tolerable time. Hence, we generate 50 random arrival sequences and schedule them to evaluate our scheduler.

C. *Evaluation Metrics*

We use Average Normalized Turnaround Time (ANTT) and System throughput (STP) metrics presented by Eyerman et al. [21] for quantifying system performances, and use Jain's fairness index [22] to evaluate the equality of performance degradation experienced by co-execution applications.

D. *Evaluation Results*

System Performance Using GScheduler. The system performance of the *GScheduler* is compared to the Round Robin (RR) and Least Loaded (LL) schedulers. Fig. 3 shows ANTT and STP for each of 50 arrival sequences. ANTT measures the time interval between submission and completion of the application. It's a lower-is-better metric whose value is larger than or equal to 1. As is shown in Fig. 3(a), *GScheduler* achieves the lowest ANTT among all of the schedulers. The average ANTT for LL scheduler is 1.85, which means an average performance degradation of 1.85x per application across 50 arrival sequences, and the average ANTT for RR scheduler is 1.4. In comparison, *GScheduler* obtains an average ANTT of 1.1, which is 40% lower than the average ANTT of LL and 21% lower than the average ANTT of RR.

STP measures the number of tasks that a system can handle per unit time. It's a higher-is-better metric. From Fig. 3(b) we can observe that *GScheduler* achieves an average STP of 24.5, which is 52% higher than LL scheduler, and 21% higher than the RR scheduler.

Scheduling Fairness Using GScheduler. Fairness is an important performance criterion for evaluating multi-program computer systems. Fairness represents the similarity of per-application interference in multiple arrival sequences. It's a value that ranges from 0 to 1. 0 indicates no fairness and 1 indicates perfect fairness (the application experiences equal slowdown in different arrival sequences). The fairness index is calculated for each of the 26 applications across 50 arrival sequences. According to the Fig. 4, *GScheduler* achieves an average fairness promotion of 5% and 9% over RR scheduler and LL scheduler, respectively

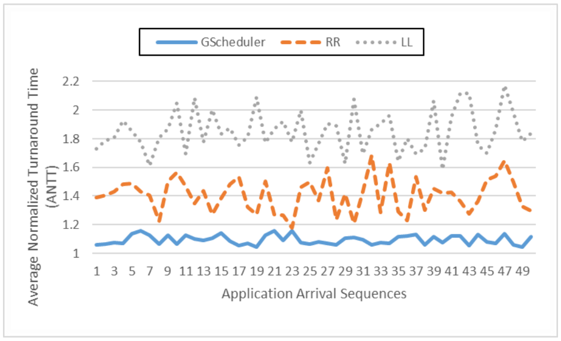

Fig. 3(a): ANTT metrics of GScheduler

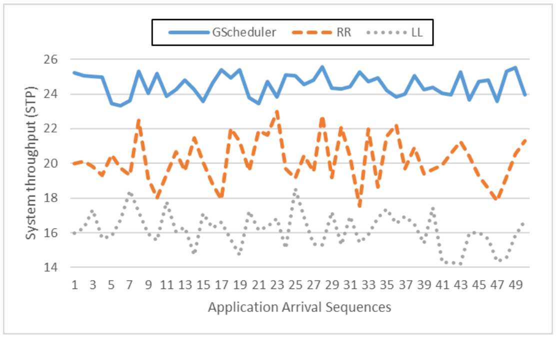

Fig. 3(b): STP metrics of GScheduler

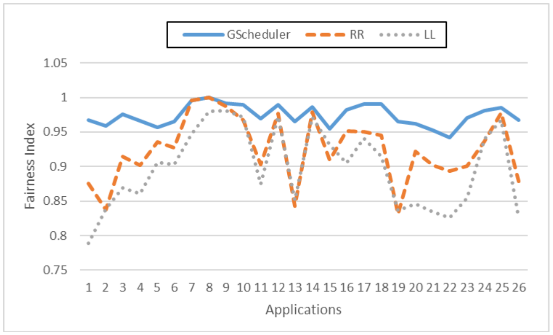

Figure 4: Scheduling fairness of *GScheduler*

E. *Discussion*

The LL scheduler will schedule the application to the compute node with the least load. Thus, many applications have a great possibility to be scheduled to the same compute node when these applications are small and short-running. This may lead to a serious interference and low STP in such a compute node.

Alternatively, the RR scheduler assigns the applications to different compute nodes in turn in the polling mode. It also does not take into consideration that GPU resource contention among multiple co-execution applications. Therefore it is inevitable that the system performance decreased significantly.

However, by mining the characteristics of GPU resource usage of applications, *GScheduler* is aware of the GPU resource usage interference among them, thus effectively improving system performance and fairness.

V. RELATED WORK

To acquire the characteristics of GPU applications, one of the existing approaches is to use a profiling tool called the CUDA Performance Tool Interface (CUPTI) [12]. Users can register callback functions by using CUPTI and these functions will be invoked when specific CUDA runtime events happen. None of the CUDA activity calls will be recorded completely until the application finishes execution. In addition, some third-party profilers, such as PAPI [13], Tau [14], Vampir [15], are also able to collect the characteristics of applications. PAPI specifies a standard API for accessing hardware performance counters. Tau is able to gather performance data through instrumentation of functions, methods and statements, etc. Vampir enables developers to obtain the behavior of an application and rapidly convert the performance data into different performance views. The other existing approach is to predict characteristics of incoming applications according to a priori empirical application or previously executed application corpus. Mystic [9] is an example, which would sample an incoming application for 5 seconds, while predicting its execution behavior according to an offline trained application corpus. To sum up, we maintain that previous works are not accurate enough to learn the characteristics of GPU applications before execution, or cannot get such information timely, which may lead to misleading scheduling decisions.

VI. CONCLUSION AND FUTURE WORK

We have presented *GScheduler*, a framework to detect and reduce interference for co-executing applications on the GPU-based cloud. *GScheduler* is composed of *GPU usage pattern* extractor and interference aware scheduler. The former is used to capture key function-call graph and key GPU resource usage before applications execution, while the latter is used to conduct scheduling for minimizing the interference among co-executing applications. *GScheduler* has been deployed it on a private GPU cluster. We evaluated it with 26 diverse, real-world CUDA applications. In future work, we plan to extend the key function library in our framework to acquire more accurate more characteristics of applications.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China under Grants No.61572129, No.61320106007, No. 61632008, No.61602112, No.61502097, No.61370207, International S&T Cooperation Program of China No. 2015DFA10490, Jiangsu Provincial Key Laboratory of Network and Information Security under Grants No.BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grants No.93K-9, and partially supported by Collaborative Innovation Center of Novel Software Technology and Industrialization and Collaborative Innovation Center of Wireless Communications Technology.

REFERENCES

[1] Luebke, David, "CUDA: Scalable parallel programming for high-performance scientific computing," IEEE International Symposium on Biomedical Imaging: From Nano To Macro IEEE Xplore, 2008:836-838.

[2] Anders, Eklund, et al, "Medical image processing on the GPU - past, present and future," Medical Image Analysis, 2013:1073-1094.

[3] Y. Zhang, F. Mueller, X. Cui and T. Potok, "GPU-accelerated text mining," In Prof. of EPHAM, 2009.

[4] C. Kim et al., "FAST: Fast architecture sensitive tree search on modern CPUs and GPUs," In Prof. of SIGMOD, 2010:339-350.

[5] Amazon, "Amazon Elastic Compute Cloud," http://aws.amazon.com/ec2/, 2017.

[6] Nimbix, "Nimbix Cloud Services," https://www.nimbix.net/, 2017.

[7] Cogeco Peer 1, "Peer1 hosting," https://www.cogecopeer1.com/, 2017.

[8] Penguin computing, "Accelerated Computing Platforms," http://www.penguincomputing.com/, 2017.

[9] Ukidave, Yash, X. Li, and D. Kaeli, "Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning," IEEE International Parallel and Distributed Processing Symposium IEEE, 2016:353-362.

[10] Sengupta, Dipanjan, et al, "Scheduling multi-tenant cloud workloads on accelerator-based systems," High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for. IEEE, 2014:513-524.

[11] Phull, Rajat, et al, "Interference-driven resource management for GPU-based heterogeneous clusters," International Symposium on High-Performance Parallel and Distributed Computing ACM, 2012:109-120.

[12] NVIDIA, "CUDA Profiling Tools Interface," https://developer.nvidia.com/, 2017.

[13] S. Browne, J. Dongarra, et al, "A Portable Programming Interface for Performance Evaluation on Modern Processors," International Journal of High Performance Computing Applications, 2000:189-204.

[14] S. S. Shende, A. D. Malony, "The Tau Parallel Performance System," International Journal of High Performance Computing Applications, 2006:287-311.

[15] Knüpfer, Andreas, et al, "The Vampir Performance Analysis Tool-Set," Proceedings of the International Workshop on Parallel TOOLS for High PERFORMANCE Computing, 2008:139-155.

[16] Edsger W. Dijkstra, "Notes on structured programming," In Structured programming, Academic Press Ltd., London, UK, 1972:1-82.

[17] NVIDIA, "Nvidia cuda toolkit 8.0," https://developer.nvidia.com/, 2017.

[18] Laura A. Zager, and George C. Verghese, "Graph similarity scoring and matching," Applied mathematics letters, 2008:86-94.

[19] James Munkres, "Algorithms for the assignment and transportation problems," Journal of the society for industrial and applied mathematics, 1957:32-38.

[20] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), 2010:1-11.

[21] Eyerman, Stijn, and Lieven Eeckhout, "System-level performance metrics for multiprogram workloads," IEEE micro, 2008:42–53.

[22] R. Jain, D-M. W. Chiu and W. R. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," DEC Research Report TR-301, 1984.