Multi-tenancy on GPGPU-based Servers

Dipanjan Sengupta
Georgia Institute of Technology
dsengupta6@gatech.edu

Raghavendra Belapure
Georgia Institute of Technology
rbelapure3@gatech.edu

Karsten Schwan
Georgia Institute of Technology
karsten.schwan@cc.gatech.edu

ABSTRACT

While GPUs have become prominent both in high performance computing and in online or cloud services, they still appear as explicitly selected 'devices' rather than as first class schedulable entities that can be efficiently shared by diverse server applications. To combat the consequent likely under-utilization of GPUs when used in modern server or cloud settings, we propose 'Rain', a system level abstraction for GPU 'hyperthreading' that makes it possible to efficiently utilize GPUs without compromising fairness among multiple tenant applications. Rain uses a multi-level GPU scheduler that decomposes the scheduling problem into a combination of load balancing and per-device scheduling. Implemented by overriding applications' standard GPU selection calls, Rain operates without the need for application modification, making possible GPU scheduling methods that include prioritizing certain jobs, guaranteeing fair shares of GPU resources, and/or favoring jobs with least attained GPU services. GPU multi-tenancy via Rain is evaluated with server workloads using a wide variety of CUDA SDK and Rodinia suite benchmarks, on a multi-GPU, multi-core machine typifying future high end server machines. Averaged over ten applications, GPU multi-tenancy on a smaller scale server platform results in application speedups of up to 1.73x compared to their traditional implementation with NVIDIA's CUDA runtime. Averaged over 25 pairs of short and long running applications, on an emulated larger scale server machine, multi-tenancy results in system throughput improvements of up to 6.71x, and in 43% and 29.3% improvements in fairness compared to using the CUDA runtime and a naïve fair-share scheduler.

Categories and Subject Descriptors

D.4.1.e [**Operating Systems**]: scheduling

General Terms

Design, Experimentation, Performance, Measurement

Keywords

GPU hyperthreading, multi-tenancy, hierarchical scheduling.

1. INTRODUCTION

A clear trend over the past few years is the increased deployment of GPUs in HPC clusters and supercomputers e.g., the Tianhe supercomputer. An interesting second trend is the consistent gain for GPU usage seen in enterprise server systems, for running computationally intensive image processing algorithm like video transcoding (e.g., for online video services) [20], other multimedia services (e.g., Adobe's Photoshop.com) [19], online gaming (e.g. NVIDIA cloud gaming) [21], and financial algorithms [17].

A challenge for efficient GPU usage in server and cloud computing systems is that GPUs are not yet first class schedulable entities, in lieu of current programming models that treat them as target 'devices' selected by applications running on attached host machines. One consequence of using this model is likely GPU under-utilization, with some devices well-utilized while others remain idle, in part because there is considerable diversity in the fraction of CPU vs. GPU processing in applications. Reasons also include difficulties in application parallelization, limits on GPU use due to necessary data movements between CPU hosts and GPUs, as well as diversity in the mix of applications currently mapped to server hardware comprised of CPU-based hosts with attached GPUs.

This paper argues for GPU multi-tenancy – which we term 'GPU hyperthreading' -- to increase GPU utilization for server clusters. The purpose of such multi-tenancy is to service requests from multiple tenants of a cluster, with specific goals that include fairness to the different applications being run, meeting real-time deadlines, and high throughput. Specifically, given the diverse goals of multi-tenancy – or GPU hyperthreading – articulated above, we propose and develop flexible software, rather than hardware, methods for GPU sharing. These methods, realized with our *Rain* scheduling infrastructure, decompose the problem of managing a server's GPU resources into two levels where (1) per GPU scheduling is responsible for efficiently multiplexing multiple requests found in its queue onto a single GPU and (2) higher level load balancing is responsible for appropriately mapping applications (and their requests) to the GPUs available on the underlying server hardware.

Rain builds on earlier work by our group like GVim [2] and Pegasus [10]. GVim describes the need for cluster level solutions of mapping jobs to potentially heterogeneous host nodes. Pegasus explores how to coordinate the scheduling of CPU and GPU resources on each cluster node. Rain extends their scheduling functionality (1) by considering multiple GPUs as potential targets for each application requesting GPU resources and (2) running at application level rather than in the virtualized settings in which GVim and Pegasus operate. For ongoing work in our group – termed GPGPU assemblies [5] – Rain can act as the scheduling framework needed to run across the logical multi-node CPU/GPU platforms provided by assembly software.

Focusing on single-node scheduling, in this paper, Rain schedules applications' GPU requests across the multiple GPUs present on high end server machines, which we emulate with a platform with two machines connected via a high end network interconnect, where each machine has two GPUs and 12 CPUs, resulting in a powerful platform with 4 GPUs and 24 CPUs total.

For such high-end servers, Rain operates as follows. First, breaking the common GPU programing model, which allows

VTDC'13, June 18, 2013, New York, NY, USA.

applications to explicitly select their target GPUs, we override their device selection calls [5], e.g., their cudaSetdevice() calls, and redirect them to a workload balancer. The balancer selects the target GPU for the application, based on available system information like current device loads, device capability or power, application characteristics, etc. Once a GPU has been assigned, individual requests are dispatched by lower (device) level scheduling. This results in the following technical contributions:

- The Rain, a GPU aggregation and two-level hierarchical scheduling framework, that decouples the GPU and CPU components of applications in order to (1) load balance workload across the multiple GPUs on high end servers and then, (2) performs device-level scheduling to handle multiple applications sharing a single GPU – GPU multi-tenancy or 'hyperthreading'.
- Dynamic feedback from device-level scheduling to load balancing is shown to improve the latter's decision making, in part because the device-level scheduler has detailed data about current device and application behavior not available to the load balancer.
- Least-attained-service (LAS) in terms of GPU request servicing is a novel metric used by a new scheduling policy shown capable of increasing overall system throughput. The policy operates by monitoring the services attained by each application running on a GPU in some given period of time and then prioritizing those applications that have attained lesser degrees of service.
- Three workload balancers are evaluated in detail, along with two device level schedulers, as well as two feedback policies, with the goals to achieve high application performance as well as high levels of system throughput and fairness.

The remainder of the paper is organized as follows. Sections 2 and 3 motivate and elaborate on the Rain multi-level scheduling approach, its software architecture and its operation. Section 4 describes the implemented scheduling policies and algorithms. Section 5 experimentally evaluates Rain's basic costs and its diverse scheduling policies. Section 6 describes related work followed by conclusions and future work in Section 7.

2. MOTIVATION

Need for GPU scheduling: Current GPU applications are written to choose some GPU device before commencing to use it. For multi-GPU servers, there are several problems with such a programmer-defined approach to selecting GPUs. First, when multiple applications are consolidated onto a single multi-GPU node, their individual target GPU selection calls, e.g., NVIDIA's cudaSetDevice() calls, being agnostic of each other can compete for the same GPUs, e.g., the one with id 0, leaving the other GPUs in the node idle. This serializes each application's GPU request which otherwise could have been served in parallel. We term such conflicts *static collisions* for applications' GPU requests. Second, applications are diverse in their GPU characteristics, with some heavily using the GPU while others use it less frequently. Since different applications will not be aware of each other's usage characteristics, they cannot assess whether they can usefully, i.e., without unnecessarily degrading their performance, share a single GPU or whether they require assignment to different GPU devices. We term this a *character collision* of applications' GPU requests. Third, static and character collisions become even more important when nodes have heterogeneous GPUs, each with different capabilities in terms of their compute and memory levels

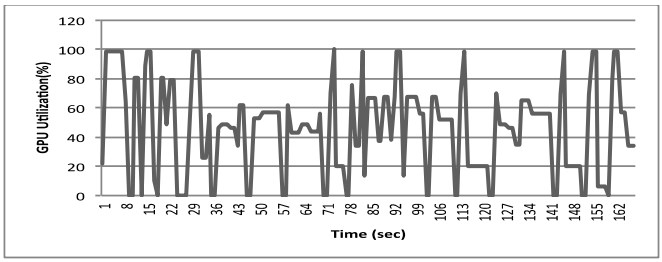

Figure 1. GPU utilization with a stream of BlackScholes requests following negative exponential distribution of inter-arrival time

of GPU utilization, particularly for web applications driven capacities, and memory bandwidths. Fourth and perhaps most importantly, single applications have difficulties achieving high by end user requests. An illustration of this fact appears in Figure 1, which shows the GPU utilization of a single application server receiving a stream of BlackScholes requests following a negative exponential distribution of inter-arrival times: there are substantial GPU idle times.

Given these facts, this paper argues for system-level support for GPU multi-tenancy: 1. to properly load balance application requests across the multiple GPUs present on high end server machines, to avoid static collisions and unnecessary GPU idleness, and 2. for each GPU, to schedule requests to obtain high application performance in lieu of potential character collisions. Such GPU load balancing and scheduling must take into account per application behavior like request frequencies and completion times, application throughput/performance, as well as desired system-level quantities like fairness, resource utilization, and others. We next describe Rain, the two-level scheduling approach to GPU multi-tenancy developed in our work.

3. SYSTEM OVERVIEW

Figure 4 presents a high level overview of a high-end multi-GPU node, which we emulate with a two-node, tightly networked cluster of GPGPU-based servers. In our test bed, each such server has six CPU cores and two GPUs attached, creating a composite machine with 4 GPUs and 12 CPUs.

3.1 Frontend and Backend Software

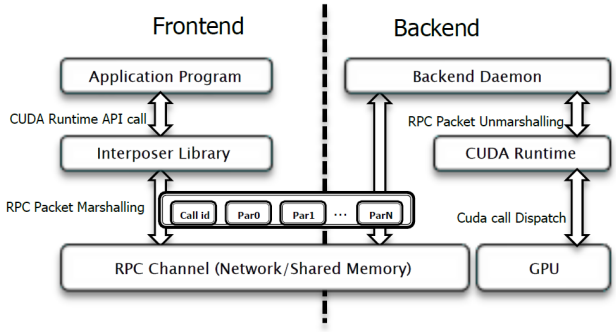

Figure 2. Architecture of GPU Remoting

Common to previous solutions for GPU scheduling, like GVim [2], Pegasus [10], rCuda [3], vCuda [7], and gVirtuS [6], is the separation of the GPU from the CPU components of an application, so that both can be managed or scheduled separately. Since NVIDIA's CUDA [4] library and driver code are 'closed', i.e. there is no low-level standard interface for accessing GPUs, separation happens at the level of the runtime API, as shown in

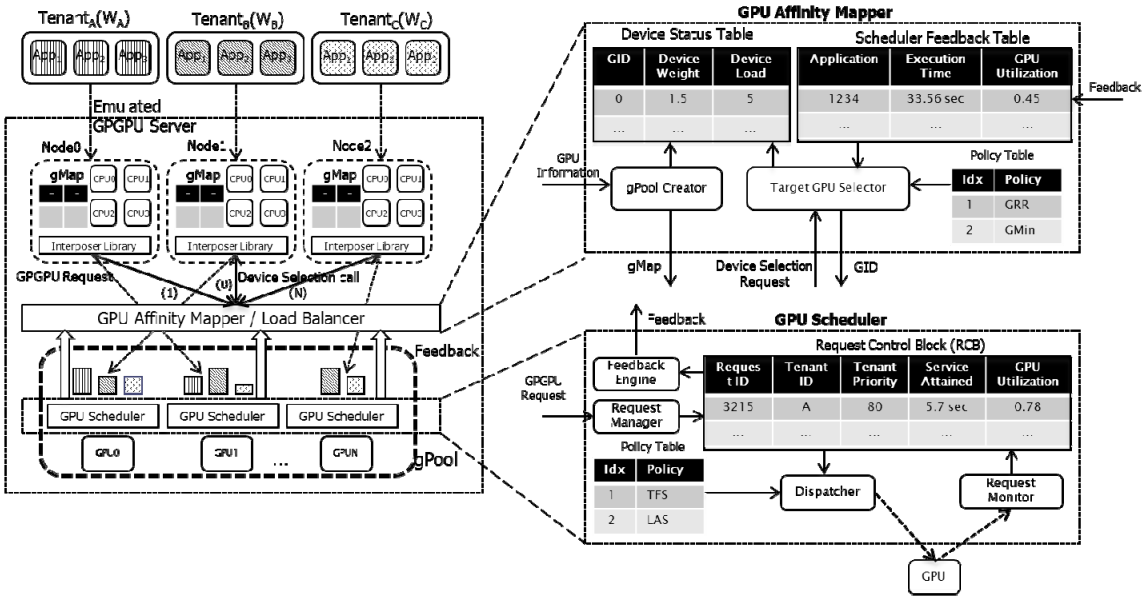

Figure 3. Architecture of Rain

Figure 2, where the *frontend* is a CUDA runtime interposer library that dynamically links with the CUDA based application, intercepting or overriding the CUDA runtime API calls.

The *backend* is a daemon running on every node that has GPUs attached to it and interacts with the frontend, e.g., to receive forwarded calls. It is this backend that is responsible for dispatching or making the actual CUDA runtime library calls to the GPU and returning error codes to the frontend. The interaction between frontend and backend involve the interposer library intercepting CUDA calls, marshalling their call information and parameters into an RPC packet, and forwarding those to the backend. The backend maintains request queues, makes the appropriate CUDA calls with the unpacked parameters, and forwards error codes to the frontend. The outcome is the decoupling of CPU/GPU associations for GPU-based application. A useful side effect of the approach is that it also makes possible 'remote' GPU accesses – GPU remoting – so that calls can be run on nodes other than where they are generated. This paper uses this fact to create the emulated higher end server machine used to evaluate Rain's load balancing and scheduling. In associated work, we further develop this functionality to enable cluster-level GPU sharing [5].

3.2 GPU Pool Creation

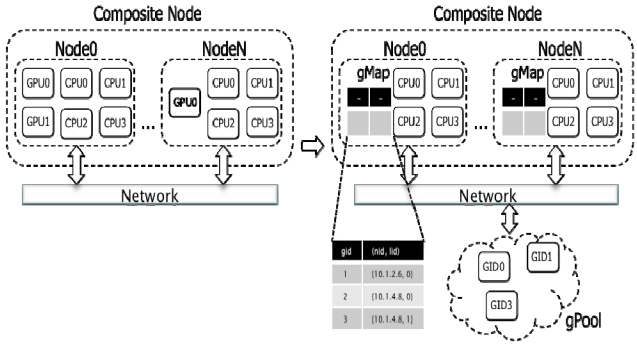

Figure 4. Logical transformations of GPU cluster after gPool creation

The Rain framework implements the following abstractions to realize scheduled (vs. uncontrolled) application access to GPUs, using the frontend/backend approach described above. First, as shown in Figure 4, all GPUs used for shared access are collected into a GPU pool, termed *gPool.* A simple example is a gPool containing the two GPUs on the small scale server used in our evaluation. A more interesting example is a gPool containing the GPUs resident on multiple nodes in a GPU cluster.

gPools are formed when the GPU virtualization runtime is started and backend daemon processes are spawned. Pool creation is supported by a *GPU Affinity Mapper* cognizant of all participating nodes and GPUs, which once information about all cluster GPUs is available, determines and broadcasts a global ordering of GPUs in the pool to all participating nodes. Each node, therefore, has a unique and globally known mapping from global device id to <node_id (IP address), local device_id> pair, which we term *gMap.* The cluster-level implementation of our work [5] permits any cluster node in the gMap to participate, by forwarding requests to GPUs on remote nodes via the network. This paper focuses on single machines, for the aforementioned small scale and larger scale multi-GPU server system. For generality, a three node system is shown in Figure 3.

3.3 Software Architecture

As shown in Figure 3, the Rain load balancer and scheduler are comprised of several components, described next.

3.3.1 GPU Affinity Mapper – Workload Balancing

This software layer has two responsibilities: (1) server-wide aggregation of GPUs through gPool creation and (2) selection of target GPUs for particular client applications, i.e., to map each application's GPU component to an appropriate GPU. Such load balancing requires static information, like the total number of GPUs in the server and their respective capabilities, and dynamic information, like current device loads, the characteristics of the application currently bound to each GPU and so on. While static information is obtained during gPool creation, dynamic information is available from the affinity mapper's own decision history and/or from feedback given by per GPU scheduling, which is discussed below in details.

The current implementation of the mapper ignores/overrides program-provided GPU selections, replacing them with its own decisions, taking the following steps. First, the interposer intercepts the cudaSetDevice() call made by an application and contacts the Affinity Mapper. The mapper selects an appropriate target GPU using the gPool and, the static and dynamic information available to it, and returns the global id (GID) of the selected GPU. Using the returned GID, the interposer inspects the gMap to determine the actual GPU device selected and finally, forwards the call to appropriate backend process and the GPU. The binding is removed when the application exits or makes a cudaThreadExit call. Interesting elements of this mapping process, as shown in Figure 3, include the following. First, detailed device information is obtained during gPool creation. This includes creating the unique GID and the computation of relative weights describing innate GPU capabilities, maintained in a data structure termed the *Device Status Tabl*e (DST). Dynamic parameters in the DST, e.g., current device load, are updated at runtime by the *Target GPU Selector (TGS)*, which is explained below. The *Scheduler Feedback Table* (SFT) is a history-based table storing information about application characteristics, like execution time, GPU utilization, data transfer times, etc., provided by the lower-level GPU scheduler. TGS is the decision engine responsible for computing the target GID on behalf of the application, using the DST, SFT, and the currently selected scheduling policy from the *Policy Table*. Affinity mapper policies evaluated in this paper include those based only on information from the DST (workload balancing policies) or from both the SFT and DST (feedback policies).

3.3.2 GPU Scheduler

GPU scheduling decisions, based on metrics that include system throughput, fairness, real-time deadlines, etc., are made for all requests directed at some specific GPU, which are then run in some scheduler-determined order. It also involves feedback to workload balancer based on the requests being executed. These tasks are carried out by the following components shown in Figure 3. The Request Manager (RM) registers-unregisters the application by creating-destroying an entry in the *Request Control Block* (RCB), on cudaSetDevice() and cudaThreadExit() calls, respectively. The RCB stores tenant properties like the tenant id, tenant priority, and the application's runtime attributes, like GPU service attained in a given interval, total execution time, and total GPU time as computed by the *Request monitor* (RMO). The *Dispatcher* is the entity that orders and controls the dispatching of GPGPU requests to the device. Using the RCB and the currently selected policy from *Policy Table*, the Dispatcher signals the backend threads to dispatch GPU requests to the device, by calling the CUDA runtime APIs. The *Feedback Engine* (FE) computes and communicates feedback information to the workload balancer, by retrieving this information from the RCB and feeding it to the SFT module of the workload balancer whenever an application request completes. Specifically, when the interposer forwards a cudaThreadExit() call to the GPU scheduler, the FE marshalls the feedback information along with the return value of the CUDA call and sends it back to the interposer, which then forwards the same to the workload balancer.

3.3.3 Multi-layer Interactions

Load balancer and GPU schedulers have different degrees of visibility of the server machine. While the load balancer has a global knowledge of all the GPUs participating in resource sharing, GPU schedulers, which are running per device, have local information about the behavior and characteristics of the applications mapped to the GPU and local GPU states. Both are complimentary to each other and thus two layers can interact with one another to help each other's scheduling. As discussed in the previous section, load balancer lays the ground for GPU scheduler by intelligently mapping GPGPU application requests to different GPUs in the server. More interesting is the feedback from GPU scheduler to the load balancer about both application behavior and device utilization that improves latter's future scheduling decisions. Overheads associated with such actions are further reduced by providing feedback at somewhat coarser grain, controllable 'scheduling epochs'.

4. SCHEDULING POLICIES

4.1 Workload Balancing Policies

We implement five different workload balancing policies, described below.

Global Round Robin (GRR): simply performs the round robin assignment of applications to the GPUs in the gPool.

GMin: to take into account differences in application completion times, GMin enhances GRR by maintaining a record of the number of applications mapped to a particular device in the *device load* field. When a new application arrives, TGS finds the GPU with the minimum device load value, increments the value by 1, and returns the corresponding GID. When an application exits or makes a cudaThreadExit() call, TGS decrements the *device load* value of the mapped GPU by 1. When serving multiple server machines, because accessing a remote GPU can be expensive, GMin gives preference to local GPUs when finding the GPU with minimum load.

*Weighted-GMin***:** to take into account differences in innate GPU capabilities, the weighted-GMin (GwtMin) policy extends GMin by assigning relative weights to different GPUs, with weights calculated during initialization.

Two additional workload balancing policies, based on feedback, are described in Section 4.3 below.

4.2 GPU Scheduling Policies

GPU scheduling seeks to achieve high system throughput without compromising fairness across multiple GPU tenants with different individual credits or priorities. Two scheduling policies evaluated in our work include Least Attained Service (LAS), a throughput driven policy, and True Fair-Share (TFS), a fairness aware policy.

4.2.1 Least Attained Service (LAS)

The objective of the LAS policy is to reduce the 'stall time' of the CPU component of a GPU-based application. Such stalls or waits occur when the application calls cudaThreadSynchronise() because it cannot progress further until GPU results are available. LAS, therefore, seeks to maximize system throughput by prioritizing the jobs with the least attained service times [13]. The policy operates by increasing the priority levels of applications that have shorter GPU episodes, i.e., have attained less GPU service time in a given time quantum. This approach helps in finishing applications with smaller GPU requests, thereby minimizing the overall CPU stall time and maximizing system throughput.

4.2.2 True Fair-share Scheduler

To ensure fairness and isolation across multiple tenants sharing the same GPU, a Real-time (RT) signal-based fair-share (FS) GPU scheduler performs proportionate GPU resource allocation

on a per-tenant basis according to their assigned weights. The invariant maintained by the scheduler is that at any point of time, at most one application backend thread is awake or is using the GPU and that it remains awake only for a time period that is proportional to their tenant weight. A signal-based implementation (instead of a queue-based one) is used in order to maintain complete isolation between different GPU applications. This involves using a separate backend process for each GPU application so that even if one of the backend threads crashes, it does not affect the backend threads of other applications.

True Fair-Share (TFS): fair share scheduling of CPU threads can be easily done via signaling, but on GPUs, long running vs. short running jobs can cause unfairness in GPU access across multiple tenants. The *true fair-share (TFS)* policy realized in our work reacts to this fact by taking into account the actual GPU service attained in every epoch. That is, if any application overshoots its allocated time slice, the dispatcher penalizes it in subsequent time slices. TFS, therefore in long run, attempts to share GPU resources so that all tenants receive their weighted fair shares when the system is heavily loaded and overall GPU utilization is high. It is also work conserving, because when some tenant is not fully utilizing its share, that share is distributed to others according to their respective weights.

4.3 Feedback-based Load Balancing Policies

Runtime feedback (RTF): requires that GPU schedulers monitor the execution time of requests scheduled on the GPU and provides such data as feedback to workload balancer. This helps the workload balancer make better decision in selecting the GPU with minimum load in future.

GPU utilization feedback (GUF): this policy informs workload balancer how efficiently some application is using a GPU, by computing the application's ratio of total GPU time to its total execution time. It draws on work [1] on NUMA aware thread placement, which shows that it is unwise to co-locate memory intensive threads on the same socket, since such co-location can lead to contention on the memory system, thereby hurting overall performance. GUF can provide feedback to the load balancer to inform it about the nature of GPU requests from different applications. Specifically, the RMO component monitors the GPU time and total execution time of each application mapped to a certain GPU. When an application exits, RMO calculates the application's GPU utilization by taking the ratio of the cumulative sum of GPU time and the total virtual runtime of the application (system+user time). Provided such data as feedback, the workload balancer then avoids collocating applications with high GPU utilization on the same GPU. The load balancing decisions improve over time as it learns more about different applications' GPU characteristics.

5. EXPERIMENTAL EVALUATION

5.1 Evaluation Metrics

We measure system throughput and fairness using the *weighted-speedup* [14] and *Jain's fairness* [16] metrics, respectively. Weighted-speedup measures the average of speedups experienced by each application in a workload sharing the GPU resource compared to when it is run alone. *Jain's fairness* measures the percentage of fairness achieved when running multiple applications with different GPU shares. In the fairness equation shown below, w_i is the GPU share allocated to an application and T_i is its total execution time.

$$Weighted\ Speedup = \frac{1}{N} * \sum_{i=1}^{N} \frac{T_i^{alone}}{T_i^{shared}}$$

$$Jain's\ Fairness = \frac{\left(\sum_{i=1}^{N} \frac{T_i}{W_i}\right)^2}{N * \sum_{i=1}^{N} \left(\frac{T_i}{W_i}\right)^2}$$

5.2 Experimental Setup

Experiments are performed on a small-scale (two GPUs) server and on a higher end server (four GPUs) emulated by two dedicated two GPU nodes (call it NodeA and NodeB) connected via gigabit Ethernet. Each node has two Intel Xeon X5660 processors for a total of 12 cores running at 2.8 GHz and 12 GB of main memory, and it has two attached NVIDIA FERMI GPUs. NodeA has a Quadro 2000 and Tesla C2050 GPU, while NodeB has a Quadro 4000 and Tesla C2070 GPU. The result is a heterogeneous higher end server where all GPUs differ in terms of their compute and memory bandwidth capacities. The GPU driver version is 295.41. Our GPGPU application service model is based on the SPECpower_ssj2008 [11] benchmark, which models a server application with large number of users. Requests coming from multiple users follow a negative exponential distribution, and are served by a finite number of server threads. The exponential distribution models the intermittent bursts of load, when application requests queue up while other requests are being processed. For a particular random stream of requests, the inter-arrival time between two consecutive requests can be calculated using the formula:

$T = - \lambda * \ln(X)$

where λ is the mean inter-arrival time between consecutive requests in a stream, and X is a random number in the range [0.0, 1.0].

NodeA and NodeB are servers processing GPU application requests. In our single node two GPU server experiments, we feed NodeA with a stream of requests following a negative exponential distribution with λ proportional to the application's runtime. In the four GPU emulated server experiments emulating a higher end GPU server, we feed each of NodeA and NodeB with independent random streams of requests.

5.3 Benchmarks

The applications from the CUDA SDK and Rodinia [18] benchmark suites used in our evaluation are shown in Table 1. They are chosen to offer a pairwise mix of short running (execution time between 5sec to 17sec) and long running jobs (execution time between 35sec to 105sec). 25 such workloads are used, labeled from A to Y, where A is BS-L BS-S pair, B is BS-L MonteCarlo pair and so on following the order in Table 1.

5.4 Results

5.4.1 Benefits of Workload Balancing

We first present the performance benefits of scheduling of GPU requests in a single node with two GPUs in comparison with the bare CUDA runtime. In this set of experiments, the node receives a stream of requests from a particular application following a negative exponential distribution. We measure the average completion times of all requests in the stream for both the CUDA runtime and different load balancing policies.

Table 1. Benchmark Applications

Program	Description	Kernel calls #
Long-running Jobs		
BlackScholes-Large (BS-L)	Fair price evaluation for 4M European options	8192
DXTC	High Quality DXT Compression of image with 512 × 512 pixels	262144
Histogram	Use of NPP for equalization for 512 MB image data.	4096
Eigenvalues	Bisection algorithm to find eigenvalues of a tridiagonal symmetric matrix with 8K × 8K elements	3
Matrix multiplication	Multiplication of 3 matrices of size 480 × 480 each	8192
Short-running Jobs		
BlackScholes-Small (BS-S)	Fair price evaluation for 4M European options	1024
MonteCarlo	Fair price evaluation for 2K European options	10000
Transpose	Transpose of a 800 × 800size square matrix	10000
BFS	Traverse a tree with 1M elements	12
Gaussian	Gaussian blur using Deriche's recursive method for matrix of size 50 × 50	8192

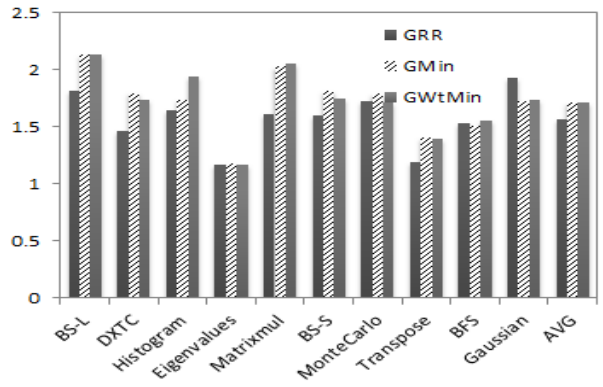

Figure 5. Performance benefit of workload balancer policies vs. CUDA runtime in a single node with 2 GPUs

Figure 5 shows the speedup achieved by three such policies compared to the CUDA runtime on ten benchmark applications. As expected, the load balancer's dynamic distribution of GPU requests across all of the GPUs in a node increases overall GPU utilization and application performance. Averaged over all applications, GRR, GMin, and GWtMin, load balancing provides weighted speedup of 1.57x, 1.72x, and 1.73x, respectively.

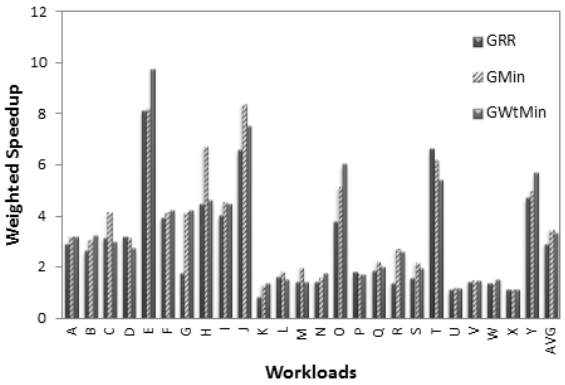

Figure 6. Performance benefit of GPU sharing in a 4 GPU server

Results with GWtMin mostly outperform those of GMin, but there are cases in which the GPU weights set do not mirror relative differences in application performance. This is one motivation for using the feedback policies evaluated below. The maximum and minimum performance improvement across all load balancing policies is 2.13x and 1.17x, respectively.

5.4.2 Benefits of GPU Sharing

This section describes results obtained with the emulated (two node) larger-scale server machine, with four GPUs total. In these experiments, one node receives a stream of short running requests and the other receives a stream of long running requests, from two different applications. Using a gPool containing all GPUs, the load balancer dynamically distributes GPU requests across all four GPUs. Choosing a single node GRR policy as the baseline, Figure 6 shows the effects of sharing GPUs between two application streams. Averaged over 25 workloads with short and long running jobs, the speedup achieved by GRR, GMin, and GWtMin policies are 2.89x, 3.48x, and 3.33x, respectively. This substantial improvement due to GPU sharing is because the peaks in GPU request volumes from the two statistically independent streams are not aligned, providing opportunities for load balancing to reap benefits from the distribution of GPU requests across all four GPUs. We also observe that of all the application pairs, maximum speedups are achieved for workloads (E, J, O, T, Y), when one of the applications is Gaussian, the shortest running job of all the benchmarks. This is because as Gaussian uses the GPU for a very short period of time, those GPUs have ample time to run the other applications.

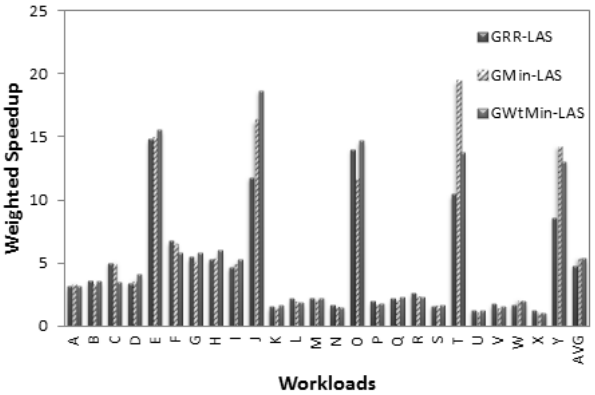

Figure 7. Performance benefit of LAS scheduling policy

5.4.3 Benefits of GPU Scheduling Policies

Two useful GPU scheduling policies are LAS and TFS.

LAS: As shown in Figure 7, we evaluate this throughput oriented scheduling policy in combination with all three load balancers. The baseline for this set of experiments is the single node GRR policy. The average weighted speedup achieved with GRR-LAS, GMin-LAS, and GWtMin-LAS is 4.75x, 5.4x and 5.37x,

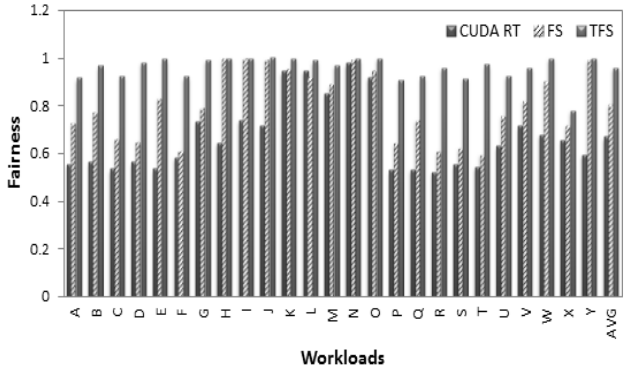

Figure 8. Fairness achieved by GPU schedulers vs. CUDA runtime among applications with equal GPU share

respectively. The high speedup achieved by LAS is due to the fact that it prioritizes the GPU requests that have attained lesser GPU time. This speeds up completion of the short running GPU jobs, thus increasing overall system throughput.

Fairshare Scheduler: we evaluate two fair-share GPU scheduling policies, FS and TFS, and compare them with the scheduler provided by the CUDA runtime. Figure 8 shows the fairness achieved in GPU resource allocation when both applications in a workload, sharing a single GPU, are given equal GPU share. We observe that TFS outperforms both the CUDA runtime and FS scheduling policy. Average fairness achieved by TFS is 96%, which is 43% and 19% better than CUDA runtime and FS respectively. TFS also achieves a near to ideal maximum fairness (99.99%). Figure 9 shows the fairness achieved when random GPU shares are assigned to both of the applications in a workload. Comparing FS and TFS, with random weight assignment, TFS performs 29.3% better than FS. This is because TFS maintains a history of GPU time attained by individual applications and penalizes any application in subsequent epochs that has used the GPU for more than its allocated share.

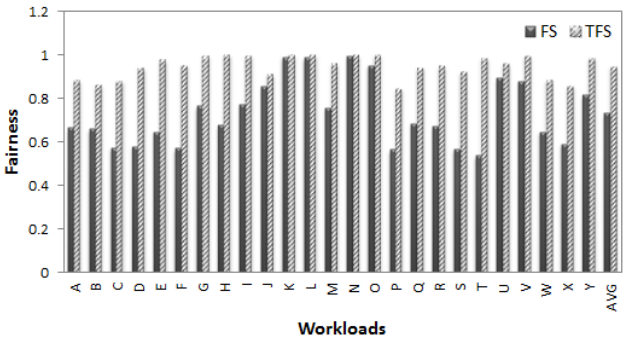

Figure 9. Fairness achieved by TFS vs. FS among applications with random assigned GPU share

5.4.4 Benefits of Feedback-based Load Balancing

The benefits of workload balancing are measured relative to the naïve single node GRR policy. Feedback-based workload balancing is automatically activated when the workload balancer receives the required feedback information from the GPU scheduler. Figure 10 shows the weighted speedup achieved by two feedback policies over the baseline. Average speedups are

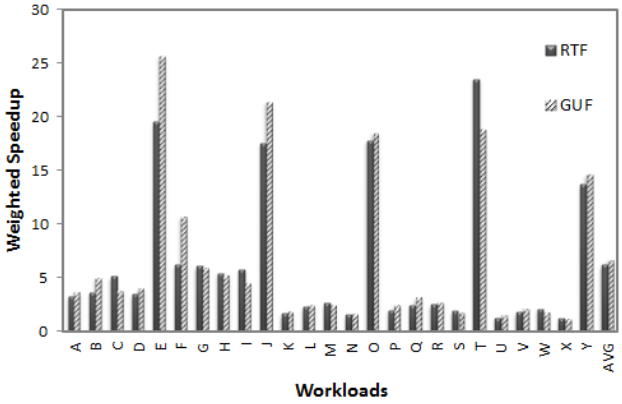

Figure 10. Performance benefit of feedback-based load balancing in 4 GPU server.

6.16x and 6.71x for RTF and GUF, respectively. Compared to the highest speedup achieved by non-feedback based balancers, combined with LAS GPU scheduling (GMin-LAS), RTF and GUF achieve 14.8% and 25% improvements, respectively. This is because these policies make use of more detailed information about application characteristics compared to GWtMin, which relies only on the static information of assigned GPU weights.

6. RELATED WORK

Previous work on GPU resource management [12] uses interference-driven job scheduling, where multiple jobs share the same GPU respecting GPU memory constraint, and the job's CPU and GPU components are scheduled on the same node. We separate these two components of every job and handle them independently; a more important difference to our work is that [12] uses static profiling, while we support online learning of an application's GPU characteristics.

Previous work on GPU virtualization GVim [2], Pegasus [10], vCuda [7], rCuda [3], gVirtuS [6] make GPUs visible from within the virtual machine. Rain extends such work by scheduling across multiple GPUs and potentially, multiple cluster nodes. vCuda and rCuda leverage the multiplexing mechanism of the bare CUDA runtime for sharing. Ravi et al [9] share GPUs by consolidation of the kernels invoked by multiple applications, but explore only a round robin policy. Their time and space sharing techniques for kernel consolidation are orthogonal to what is done in our work. Another interesting complement to our work is recent research [8] on managing GPU memory pressure arising from the consolidation of multiple applications on a single GPU.

7. CONCLUSIONS AND FUTURE WORK

This paper explores GPU multi-tenancy for GPU-based servers used in datacenter or cloud computing systems, implemented with the 'Rain' infrastructure. Rain i) dynamically constructs shared GPU pools, ii) uses multi-level scheduling by decomposing the problem into a combination of load balancing (across multiple

GPUs) and device-level scheduling, iii) evaluates a throughput oriented device-level scheduler favoring jobs with least-attained GPU service, iv) can employ feedback information from device-level scheduling to adjust load balancing, and v) implements a history-based fair-share scheduler with improved fairness.

The opportunities and costs of GPU multi-tenancy realized with Rain are evaluated across a wide variety of workloads and system configurations, and these evaluations demonstrate average speedup of 1.73x for a smaller scale server with two heterogeneous GPUs compared to using the CUDA runtime. It further achieves an average weighted speedup of 6.71x on a two node system emulating a high end server compared with applications confined to the resources of a single node. It also achieves nearly ideal levels of fairness (up to 99.99%) across applications from multiple tenants.

Future work should address memory issues with GPU multi-tenancy, and in addition, we are exploring the use of runtime binary translation (of GPU kernels) [15] to further broaden the potential targets a load balancer can choose for running GPU requests (e.g., by running requests on otherwise idle CPUs).

8. REFERENCES

[1] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. 2010. A case for NUMA-aware contention management on multicore systems. In *Proc. of PACT '10* ACM, New York, NY, USA, 557-558.

[2] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2009. GViM: GPU-accelerated virtual machines. In *Proc. of HPCVirt '09* ACM, New York, NY, USA, 17-24.

[3] J. Duato et al. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. of HPCS'10, 224-231.*

[4] NVIDIA CORP. NVIDIA CUDA Compute Unified Device Architecture. http://tinyurl.com/cx3tl3.

[5] Alexander M. Merritt, Vishakha Gupta, Abhishek Verma, Ada Gavrilovska, and Karsten Schwan. 2011. Shadowfax: scaling in heterogeneous cluster systems via GPGPU assemblies. In *Proc. of VTDC '11*. ACM, New York, NY, USA, 3-10.

[6] gVirtuS: http://osl.uniparthenope.it/projects/gvirtus.

[7] L. Shi, H. Chen, and J. Sun. 2009. vCUDA: GPU accelerated high performance computing in virtual machines. In *Proc. of IPDPS '09,* Washington, DC, USA, pp. 1-11.

[8] Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. 2012. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *Proc. of HPDC '12*. ACM, New York, NY, USA, 97-108.

[9] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. 2011. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proc. of HPDC '11.* ACM, New York, NY, USA, pp. 217-228.

[10] V. Gupta, K. Schwan, N. Tolia, et al. Pegasus: Coordi- nated scheduling for virtualized accelerator-based systems. In *USENIX ATC,* Portland, USA, 2011.

[11] Design document SSJ workload, SPECpower_ssj2008: http://www.spec.org/power/docs/SPECpower_ssj2008-Design_ssj.pdf.

[12] Rajat Phull, Cheng-Hong Li, Kunal Rao, Hari Cadambi, and Srimat Chakradhar. 2012. Interference-driven resource management for GPU-based heterogeneous clusters. In *Proc. of HPDC '12*. ACM, New York, NY, USA, 109-120.

[13] R. Righter and J. Shanthikumar. Scheduling multiclass single server queueing systems to stochastically maximize the number of successful departures. *Probability in the Engineering and Information Sciences,* 3:967–978, 1989.

[14] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS-IX,* 2000.

[15] G. Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of PACT '10.* ACM, New York, NY, USA, 353-364.

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

[17] Zillians: http://www.zillians.com/solutions/.

[18] Rodinia: http://tinyurl.com/bkqzaou.

[19] Adobe Photoshop.com: http://www.photoshop.com/.

[20] Elemental Technologies: http://tinyurl.com/bcj3jet.

[21] NVIDIA cloud gaming: http://www.nvidia.com/object/cloud-gaming.html.