Dynamic Kernel/Device Mapping Strategies for GPU-assisted HPC Systems

Jiadong Wu, Weiming Shi, and Bo Hong

School of Electric and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
jwu65, weimingshi, bohong@gatech.edu

Abstract. With their high computation throughput and outstanding performance-per-watt figures, the graphics processing units (GPU) are becoming increasingly important for high-performance computing (HPC) systems. Existing GPU execution environment restricts the GPU usage to local host node. This is suitable for standalone computer nodes, but becomes inefficient for HPC systems that consist of a large number of GPU-assisted nodes. In this paper, a novel framework is proposed to support dynamic GPU kernel/device mapping strategies for HPC systems. Adaptive mapping policies are designed to mitigate the impact of network transfer overhead. The performance of the framework is studied through extensive simulations. The results show that compared with existing local-only static mapping method, the proposed framework is capable of improving the system-wide GPU utilization rate and computation throughput, especially when the concurrent workloads exhibit different GPU usage intensities.

1 Introduction

The last two decades witnessed the evolution of graphics processing units (GPUs) from the graphics accelerators to the coprocessors that are becoming increasingly important for high-performance computing (HPC) systems. Thanks to the rapid advancement in GPU programming frameworks such as CUDA[11] and OpenCL[7], GPU computing has been successfully deployed for a wide range of applications in both desktop and HPC settings [12, 10]. These applications cover a wide distribution of GPU usage intensities.

Existing GPU-assisted HPC systems often have a cluster structure where multiple GPU-assisted compute nodes are interconnected with high speed networks such as the InfiniBand. For such emerging GPU clusters, their resource management systems often adopt existing scheduling systems such as PBS [13]. These scheduling systems were originally designed for CPU-only systems, and are augmented to treat GPUs as one more type of resource on the compute nodes. When these job scheduling systems allocate user processes to the compute nodes, the execution of each process is controlled by the GPU execution environment

of its host node. The user process is subsequently restricted to utilize the local GPU devices on the host node.

The per-node static kernel/device mapping method, while working well for standalone computer nodes, has significant shortcomings in HPC settings. Inefficiency would arise when the physical node configuration mismatches the workload pattern of the user processes:

1. *GPU underutilization* would be observed as GPU cards may be idle between kernels, especially when GPUs are used sporadically. Additionally, algorithmic requirements may restrict a host process to utilize only a subset of the locally available GPUs, thus wasting other GPUs. For example, an HPC application may be designed to use 2 GPUs on each node but is wastefully deployed to a 4-GPU-per-node system.
2. *GPU oversubscription* would be observed as the user processes of computation intensive applications may launch kernels faster than what the local GPUs can process. This is especially the case when the application consists of a large number GPU intensive tasks. With the static kernel/device mapping, the host processes may be starving for GPUs to process the tasks.

The overall system performance may therefore suffer from great performance degradation if applications with different GPU utilization run concurrently in the system, which will cause some GPUs to be underutilized while others oversubscribed.

Such static mapping method also affects programmability. With the existing method, a GPU-accelerated application may be (painfully) hand-optimized for a particular HPC deployment. But such optimization relies on the static kernel/device mapping and is therefore customized to the hardware configuration of that HPC system. When porting to a new/upgraded system with different configurations, those optimizations will become impaired and the code will underperform in the new system.

In this paper, we argue that these flaws of the current GPU-assisted HPC clusters can be alleviated and that the overall system performance and utilization can be improved when running unbalanced mixed workloads if a dynamic mapping strategies could be established between the GPU devices and the GPU kernels of the user applications. We present a novel idea of GPU resource management module (GREMM) that incorporates with existing remote GPU kernel execution technique and allows dynamic GPU kernel/device mapping. In particular, our study focuses on the dynamic kernel/device mapping policies that can proactively assign GPU kernels to remote GPUs that would otherwise be underutilized if the communication overhead is lower than the local waiting time. The main objective of the dynamic mapping framework is to refine the granularity of resource management and to explore both CPUs and GPUs to bridge the mismatch between the fixed physical node configurations and the varied workload requirement.

We demonstrate the efficiency of the proposed strategies by comparing against native systems (with static local kernel/device mapping). The results show that the dynamic kernel/device mapping outperforms the existing static execution

environment in terms of the GPU utilization ratio and the computation throughput, especially for unbalanced mixed workloads. We expect the proposed framework to improve the efficiency of GPU-assisted HPC systems.

The rest of the paper is organized as follows. In Section 2, we provide the background information on our work and survey the related works. In Section 3, we introduce the dynamic kernel/device mapping framework and categorize its overheads, which lays down the foundation for our design of mapping strategies. In Section 4, we present the design of three mapping policies. In Section 5, we develop discrete event simulation to evaluate the performance. Some concluding remarks and future work are given in Section 6.

2 Background and Related Works

Existing GPU execution environments such as the Nvidia CUDA framework [11] assume the user processes to be bound to the local GPUs. A GPU kernel request is handled by the local GPU driver, which then loads the kernel on a local GPU device, executes it, and returns the results to the requesting process. As a way to resolve the scarcity of GPUs in many computer systems, GPU sharing has attracted intensive research attention. The existing techniques employed in GPU sharing is briefly summarized as follows.

PBS[13] and Slurm[14] are two widely adopted resource management systems for HPC systems. They were originally designed for CPU-based systems and have been upgraded to support GPU-assisted nodes. PBS and Slurm track user requests and system status, and map user processes to the compute nodes. In current PBS and Slurm systems, the process/kernel mapping is static, and the execution of the processes, once mapped, is governed by the compute nodes. For GPU-accelerated applications, the execution of each process is therefore subject to existing GPU execution environment on the compute nodes, which restricts user processes to utilize local GPU devices.

rCUDA [4] is proposed to enable the compute nodes not equipped with local GPUs to access the remote GPUs hosted on remote compute nodes. It employs API remoting technique to reroute the GPU calls to a remote GPU-assisted compute node. With rCUDA, the remote GPUs are statically specified in a configuration file on the requesting node. rCUDA works between a pair of designated nodes, and is particularly useful in a cluster environment where only a few nodes are equipped with GPUs. In such settings, rCUDA allows other non-GPU nodes to execute their GPU kernels on the GPU-assisted nodes, but the kernels in rCUDA-based system remains statically bound to devices, since the users have to hard-code the remote rCUDA server IP into their application. rCUDA is not designed to manage GPUs in an GPU-assisted HPC system.

GViM[6] is an API level solution to virtualize GPU systems. GViM is not designed to access remote GPUs since it can only virtualize GPUs on a standalone computer. Shadowfax[9] is proposed to address the access limit and to support unmodified applications in multiple virtual machines in order to share both local and remote GPUs. Similar to the static designation of remote GPUs

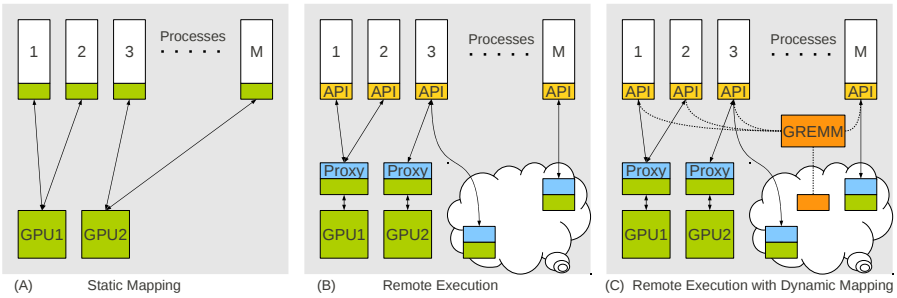

Fig. 1. Illustration of GPU kernel/device mapping models.

in rCUDA, all virtual GPUs in Shadowfax need to be manually mapped to a physical GPU, which is unsuitable for managing GPUs in HPC systems where user application requests are not known as a priori.

The capability of remote GPU kernel execution is also explored in several other projects such as SnuCL[8], MGP[1], and gVirtuS [5]. Both SnuCL and MGP target to improve the programmability of GPU-assisted applications on a GPU cluster by providing a single system image. gVirtus focuses on providing a virtualization service which supports the remote GPU sharing. However, to the best of our knowledge, little research has been done on how to efficiently schedule the remote GPU device accesses in HPC systems.

3 Dynamic Kernel-Device Mapping

A novel HPC system framework is presented to facilitate dynamic kernel/device mapping strategies, and thereby improving the system-wide GPU resource utilization. The framework is essentially a combination of the existing remote kernel execution infrastructure and the decision maker of the kernel/device mapping.

3.1 The Framework

As we noted before, the prevailing GPU invocation method is restricted to access the GPUs on the board of the local compute node only - the kernel calls are routinely directed to the binding local GPUs as illustrated in Figure 1(A). With the development of the middleware such as rCUDA[4] and gVirtus[5], the scope of invocable GPUs is expanded to all the GPUs across the cluster system, as is illustrated in Figure 1(B). In these existing remote execution infrastructures, the mapping between the GPU kernel and device is still statically bound. But the restriction can be removed by extending such existing infrastructures.

While our work mainly focuses on the dynamic kernel-device mapping policy, it is informative to have an idea on how the framework would be constructed. To

put it into perspective, such a framework can be decomposed into three components: (1) the front-end library of user API, (2) the GPU Resource Management Module (GREMM), and (3) the GPU execution proxy.

1. The front-end API library should provide equivalent interface as existing GPU programming environment such as CUDA or OpenCL, and implement functionalities that communicate to GREMM. By linking to this library, programmers can write conventional GPU codes for their applications without considering how the GPU calls are handled. Once the executable is linked with this library instead of the stock one, GPU related functions will be automatically wrapped into task messages and dynamically forwarded to proper proxy based on GREMM's decision.
2. The GPU resource management module is the middle layer of our framework that connects GPU API calls and the execution proxies. As the heart of the system, the GREMM is responsible for making the kernel mapping decisions. A variety of policies can be included in our design. Based on a specific policy, the modules will work either independently or cooperatively to assign GPU kernels.
3. The GPU execution proxy is the bottom layer of the dynamic mapping framework responsible for the host/device memory copying, kernel launching, and other device control functions. Each proxy controls one local GPU device and communicates with the local and remote API callers. Guided by the GREMM, every GPU task message will eventually be served at a execution proxy.

The described framework constitutes a direct and easy extension of the existing remote kernel execution infrastructure. More importantly, among all the design choices, we contend that the decision maker can be put into a separate module, referred to as GREMM and also illustrated in Figure 1(C), so that not too much change would be made on the side of remote kernel execution infrastructure to install various mapping policies.

3.2 Categorization of Overheads

Applications running on existing GPU-assisted HPC systems are subjected to the overhead of queuing for the statically mapped GPU devices. At the system level, this overhead is expected to be lowered through balanced allocation of GPU devices in dynamic mapping framework.

However, remote kernel execution does introduce a new type of overhead: the network overhead. Network overhead will not incur for the traditional kernel/device mapping method since it only uses local GPUs, but will incur when the GPU kernel needs to be executed on a remote node. The amount of this overhead is directly related to the volume of transfered data and network performance. Data intensive workloads will lead to negative performance gains. But the performance degradation could be avoided if an appropriate policy is available to track workload data/computation ratio and decide when to activate remote execution and when to fall back to the local-only method.

We will study the impact of network overhead and also the benefits of reduced queuing overhead in the following sections.

4 Dynamic Kernel/Device Mapping Policies

In this section, an abstraction of GPU-assisted HPC clusters is presented, followed by the design and evaluation of three mapping policies for the dynamic kernel/device mapping framework.

4.1 System Abstraction

We consider the following abstraction of GPU-assisted HPC clusters. There are N homogeneous compute nodes in the system, each configured with M processor cores per node and K GPU devices per node. And we assume $M \geq K$. A user application consists of multiple processes. Processes from all the applications are mapped to the compute nodes by a job scheduling system that employs the following rules: (1) a compute node will not be split among multiple applications, (2) processes do not migrate once mapped, (3) each compute node receives less than M processes, and (4) compute nodes receive balanced workload for each application. Such a job scheduling policy represents the typical practice of many popular scheduling systems (e.g. PBS).

We assume that each process executes a program code consisting of multiple iterations, where each iteration consists of a CPU code segment followed by a GPU code segment – the GPU kernel.

We further assume that the programmer will explicitly copy back any useful data from GPU after a kernel is finished, so the GPU context associated with certain process becomes volatile when its new kernel is not launched on the same GPU device as before. Discussions on this limitation will be presented in the final section.

4.2 Global Reservation Policy

In Global Reservation (GR) Policy, a FIFO queue is set up for the GPU cluster. GPU tasks launched by any process will be registered in this queue, which will later be served by a total number of $N \times K$ GPU devices. The actual data transfer occurs directly between the requesting process and the serving GPU, and is not transferred via the queue. Theoretically, if an infinite fast network interconnection is given, the global reservation policy is expected to achieve the best system-wide GPU utilization.

However, because the GPU device needs to be reserved while data/kernel is being transferred from a remote node, the efficiency of this policy is highly sensitive to the network overhead. For the proposed dynamic kernel/device mapping to perform well under environments of varied workload, adaptive policies are then explored.

4.3 Adaptive Greedy Policy

Adaptive Greedy (AG) Policy aims to map the kernel call to the GPU device that requires the least total waiting time every time a new kernel call is initiated.

Denote all the GPUs in the system be $\mathcal{G}$, the set of local GPUs be $\mathcal{L}$. The number of all GPUs in the system is $|\mathcal{G}| = NK$.

AG examines every GPU device g in the system, estimates the total waiting time τ_g if the kernel call is mapped to that GPU. The total waiting time τ_g is composed of the queueing delay τ_g^q and the data transfer delay τ_g^d.

$$\tau_g = \tau_g^q + \tau_g^d \tag{1}$$

The queueing delay τ_g^q is estimated by the number of queued kernel calls on that GPU device N_g and the average execution time of last k kernel calls on that GPU device τ_g^k.

$$\tau_g^q = N_g \cdot \tau_g^k \tag{2}$$

The data transfer delay τ_g^d is zero if g is a local GPU device and is estimated by the amount of data transferred from the host node to the remote node D_{out}, the amount of data transferred back from the remote node to the host node D_{in} and the outbound (resp. inbound) bandwidth B_g^{out}(resp. B_g^{in}) if g is a remote device.

$$\tau_g^d = \begin{cases} 0 & \text{if } g \in \mathcal{L} \quad (3) \\ \dfrac{D_{out}}{B_g^{out}} + \dfrac{D_{in}}{B_g^{in}} & \text{if } g \notin \mathcal{L} \quad (4) \end{cases}$$

B_g^{out} is estimated by the nominated inter-node bandwidth BW and the number of out-bound kernel calls on the host node, namely O_l and the number of in-bound kernel calls on g, namely I_g when the kernel call is to be assigned.

$$B_g^{out} = \frac{BW}{\max_{g \in \mathcal{G}-l}(O_l, I_g) + 1} \tag{5}$$

B_g^{in} is estimated by the nominated inter-node bandwidth BW and the system-wide average number of queued kernel calls per node. Notice that B_g^{in} is different from B_g^{out} as the bandwidth may change with the progress of the kernel execution.

$$B_g^{in} = \frac{BW}{\sum_{g \in \mathcal{G}-l} \max(O_g, I_l)/|\mathcal{G}|} \tag{6}$$

AG chooses the node g^* with the least total waiting time as the candidate node that the kernel call is to be assigned. The computational complexity of AG is $O(|\mathcal{G}|) = O(NK)$.

$$g^* = \arg\min_{g \in \mathcal{G}} \tau_g \tag{7}$$

4.4 Adaptive Random Policy

Adaptive Random (AR) Policy is a randomized policy. It tries to construct and maintain a table which records the probability that a particular GPU device should be chosen to serve the kernel call. It assigns the kernel call based on the probabilities in the maintained table. It resorts to the GPU driver to handle the contention for the GPU device on a particular node if there is any.

The probability of being chosen is calculated based on a weight table that is associated with the system-wide GPU availability. In this table, each GPU device is assigned a weight indicating the relative probability of being chosen.

Denote the nominated inter-node bandwidth be B, the weight of a remote idle (resp. busy) GPU device be w_{ri} (resp. w_{rb}), the weight of a local idle (resp. busy) GPU device be w_{li} (resp. w_{lb}).

Assume that the inertia towards choosing the busy GPU devices over the idle ones is characterized by a 'penalty' factor $\alpha(< 1)$, and that the preference towards choosing the local GPU devices over the remote ones is characterized by a 'bonus' factor $\beta(> 1)$. Hence we have

$$\alpha = \frac{w_{lb}}{w_{li}} = \frac{w_{rb}}{w_{ri}}, \tag{8}$$

$$\beta = \frac{w_{lb}}{w_{rb}} = \frac{w_{li}}{w_{ri}}. \tag{9}$$

Without loss of generality, if we set the $w_{ri} = 1$, then $w_{li} = \beta$, $w_{rb} = \alpha$, $w_{lb} = \alpha\beta$.

The 'penalty' factor α can be quantified by the average execution time of received kernel calls τ_g^k and the node configuration of the host node.

$$\alpha = \frac{M}{K} \cdot \frac{1}{\tau_g^k} \tag{10}$$

The design philosophy of α is that the relative probability ratio of choosing a busy node over choosing a idle node should be proportional to the relative ratio of the time ticks that a node is idle, and that the larger the ratio of the number of GPUs versus the number of CPUs on a host node, the less chance the kernel calls should be assigned to remote nodes.

The 'bonus' factor β can be quantified by the amount of transferred data D, the average execution time of received kernel calls τ_g^k and the number of nodes in the system N. The design philosophy of β is that the higher the ratio of the communication time to the computation time, or the higher the data consumption rate, or the more nodes in the system, the more chance the local nodes are favored over the remote nodes.

$$\beta = \left(\frac{D/B}{\tau_g^k}\right) \cdot \left(\frac{D}{\tau_g^k}\right) \cdot N = \left(\frac{D}{\tau_g^k}\right)^2 \cdot \frac{N}{B} \tag{11}$$

The computational complexity of AR is also $O(|\mathcal{G}|) = O(NK)$ while the information it needs to keep is less than AG. When a GPU task arrives, GREMM makes the randomized assignment decisions based on the weights in this table.

More sophisticated mapping policies can be designed, for example, to explore execution history of the system and to accept users' hint about the pattern of their jobs. We leave the exploration of the advanced mapping policies for our future work. In this paper, we focus on the benefits of dynamic GPU kernel/device mapping and its effectiveness under different workload and system conditions. Greater performance improvement is expected when more advanced mapping policies are adopted.

5 Performance Evaluation

In this section we develop a discrete event simulator to simulate the runtime behavior of large-scale GPU-assisted clusters. The performance of dynamic kernel/device mapping strategies is then verified through extensive simulations.

It is desirable to evaluate the dynamic kernel/device mapping framework in a large-scale production GPU-assisted HPC using real benchmark workloads. But because GPU-based HPC computing is an emerging field, there does not exist well-established workload traces for this type of systems. Available GPU benchmarks (e.g. RODINIA[2] and SHOC[3]) are designed to stress micro-architectural features of GPUs, which are unsuitable to describe multiple concurrent workloads at the system level for our study. To address this issue, we synthesized our workload traces, which are designed to be representative of GPU-HPC workloads.

5.1 Experimental Setup

The four GPU mapping policies tested are: 1) ST, the static kernel/device mapping policy; 2) GR, the global reservation policy; 3) AR, the adaptive random mapping policy with $k = 10$; and 4) AG, the adaptive greedy mapping policy with $k = 10$. The ST policy, as our baseline, is the conventional policy in GPU execution environment which shows the GPU utilization of the native system without remote execution or dynamic mapping. GR, AR, and AG are dynamic kernel/device mapping policies.

The two major performance metrics evaluated are GPU Utilization Rate and Mean Waiting Time. GPU Utilization Rate is the ratio of the GPU busy time to the total GPU time available. This rate directly reflects the utilization efficiency of the entire GPU cluster. The Mean Waiting Time measures the average time that a GPU task spends on data transfer and queuing for GPU devices. It reflects the average overhead for each kernel execution.

5.2 GPU-assisted Cluster Simulator

The simulated cluster consists of N computing nodes. Each node consists of M CPU cores, K GPU devices, and a full-duplex network interface card (NIC) with max bandwidth B. The CPU cores are characterized by the processing capabilities. GPU devices are also characterized by their processing capabilities. The

latency of remote execution API functions is modeled based-on data observed in previous researches[4, 6, 9].

The NIC on each node has two independent ports: the inbound port and the outbound port. A max bandwidth B is enforced on each port. Once any data is to be transmitted from one node to another, a connection will be established from the outbound NIC port of the source to the inbound NIC port of the sink. Concurrent connections on a single port share the port's bandwidth evenly. However, the effective bandwidth of a connection is limited by the busier one of the two participating ports. So, if any one of the concurrent connections fails to fully utilize its share, the remaining bandwidth will be utilized by other concurrent connections. According to this scheme, the system-wide bandwidth allocation changes when a new connection is established or a current connection is completed. We adopt this simplified network model as our focus is on the impact of network transfer overhead, rather than on how the overheads are generated. Therefore, the detail characteristics of a typical network such as the topology and the routing are not taken into account in this work.

Unless explicitly noted later, the cluster in following simulations is configured as $N = 24$, $M = 12$, $K = 3$. The bandwidth is set to $B = 100KB/ms$ for GbE and $B = 1000KB/ms$ for IB. The simulated time span is $10,000s$.

5.3 Generation of Workload Traces

The input to the simulator is the workload trace, which is organized as groups of consecutive *tasks*. Each group is associated with one software process. We characterize a CPU task by the amount of time T_c delayed on the CPU core, and a GPU task by three parameters: the amount of data D_u uploaded to the GPU device; the amount of execution time T_g on GPU device; and the amount of data D_d downloaded from the GPU device to host process.

The workload used for the evaluation of the dynamic mapping framework is generated based on following assumptions:

- Each process executes CPU and GPU tasks alternatively.
- The execution time of CPU and GPU task is random variable of exponential distribution with parameter $\lambda = 1/T_c$, $\mu = 1/T_g$ respectively.
- The size of the input and output data sets of a GPU kernel is proportional to the kernel execution time.

For example, a process P generated with parameters $T_g = 2250ms$, $D_u = 10 \times T_g$, $D_d = 0.5 \times D_u$, and $T_c = 750ms$ will have the following characteristics: the average length of GPU kernel is $2250ms$; the average data uploaded to GPU each time is $22500KB$; the average data downloaded from GPU each time is $11250KB$; and the average time spent on CPU before next GPU kernel launch is $750ms$.

Since the policies are designed to address unbalanced GPU utilizations of concurrent GPU workloads in an HPC system. The traces we used are mixed combinations of a heavy-GPU application and a light-GPU application. We assume that the system runs these two applications with full capacity: there are n_i

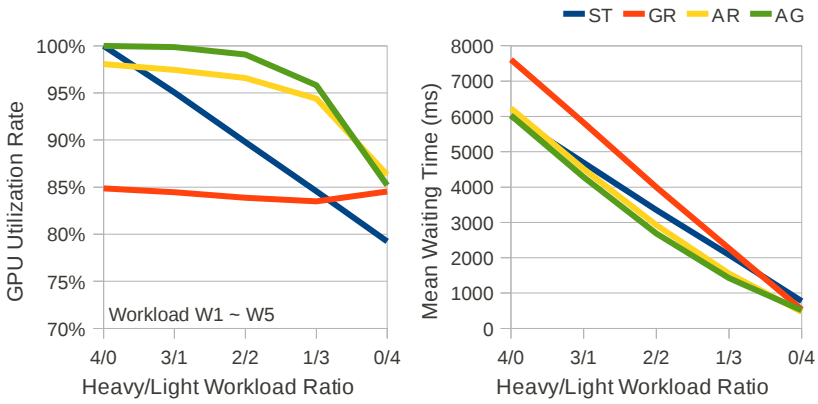

Fig. 2. Impact of workload mix.

(assuming n_i is a multiple of $\frac{M}{K}$) processes in Workload i, and $n_1 + n_2 = N \cdot M$. In such case, the GPUs in the system are subject to the different computation intensity. The benefit of routing a kernel from a stressed node to an idle remote node can potentially overweigh the extra overhead of network transfer. Our analysis can be extended to scenarios with more applications.

5.4 Workload Mix

Our first set of experiments examines a set of mixed workloads. Traces W_1 to W_5 are synthesized from two client applications submitted to the cluster. Client H's application consists of tasks with heavy GPU usage (with $T_c = 750ms$, $T_g = 2250ms$, $D_u = 10 \times T_g$, $D_d = 0.5 \times D_u$) and client L's application consists of tasks with light GPU usage (with $T_c = 2250ms$, $T_g = 750ms$, $D_u = 10 \times T_g$, $D_d = 0.5 \times D_u$). The five traces are synthesized to represent the mix of two workloads with different GPU demands. The process population ratio of H/L is 24/0 in W_1, 18/6 in W_2, 12/12 in W_3, 6/18 in W_4, and 0/24 in W_5.

The system is simulated with the network bandwidth set to $100KB/ms$ (GbE). As shown in the left subplot of Figure 2, the system-wide GPU utilization rate can be improved by dynamic mapping policies in most of the cases. Since there are underutilized GPU devices on the nodes, transferring GPU tasks from heavily occupied local devices to remote idle devices is beneficial. It is worth noting that significant improvement can be observed for the adaptive policies even for such low bandwidth network. This indicates that the dynamic kernel/device mapping is particularly useful for mixed workloads that have different GPU demands. Meanwhile, the mean waiting time is also improved as is shown in the right subplot of Figure 2.

Figure 3 shows the number of completed GPU kernels under different policies on Traces W_1 to W_5. Taking W_3 as an example, in the simulated time span, the

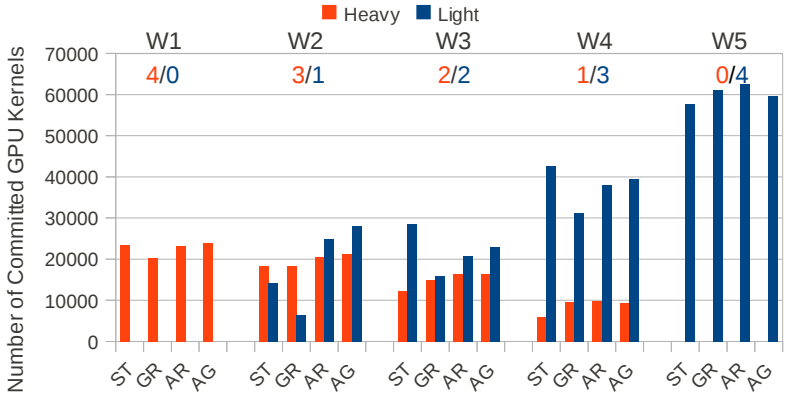

Fig. 3. Number of completed GPU kernels with different policies.

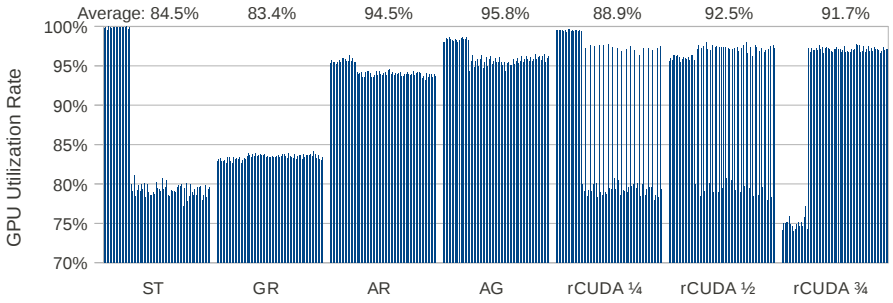

Fig. 4. Detailed GPU Utilization of the Cluster with static and dynamic policies.

conventional ST policy finishes about $12K$ kernels for client H and about $28K$ kernels for client L. When the dynamic policies are applied, the overall system-wide GPU utilization rate is improved. It is also interesting to note that client L is affected by the other policies, i.e. client L completes less kernels if remote GPU mapping is allowed. This is because the GPUs previously dedicated to client L are now executing client H's kernels too. This set of experiment suggests that if certain client's application is mission-critical, it is desirable to exclude other applications from utilizing its GPU devices, even though this will reduce the GPU utilization rate of the system. We plan to investigate the prioritized policy in our future study.

5.5 Load Balance

Figure 4 lists the detailed GPU utilization of the cluster with different policies on the mixed workload trace W_4, since W_4 is a very good example to demon-

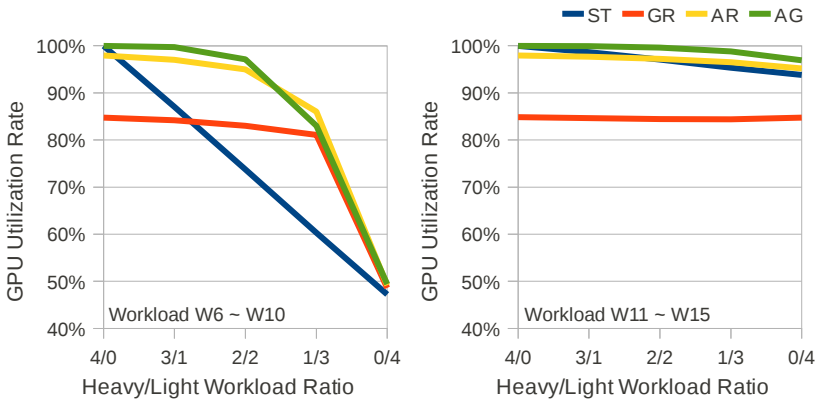

Fig. 5. Impact of GPU utilization intensity.

strate the performance improvement of the dynamic kernel/device mapping. The bandwidth is set to $100KB/ms$ in this and the following experiments as well. It shows that the utilization with ST is negatively affected by the unbalanced node utilizations. The GR policy is capable of balancing GPU utilization. The AR and AG policy outperforms the other policies for this set of experiments.

As mentioned in the background section, techniques such as rCUDA allow a process to send all its GPU kernels to a statically designated remote node, but they do not support run-time kernel/device mapping. For fair comparison, we tested three static schedulers for rCUDA on workload W_4: 1/4, 1/2, and 3/4 of the client H's GPU kernels were directed to client L's GPUs. The results show that when 1/2 of client H's processes can use rCUDA, the system achieves GPU utilization rate of 92.5%, which is still worse than the performance of the dynamic mapping policies. Nevertheless, the results also demonstrate the difficulty in optimizing the performance by the static scheduler of rCUDA: ratios 1/4 and 3/4 are less efficient, finding the better ratio of 1/2 is non-trivial. Furthermore, since the rCUDA mapping decision needs to be made before launching user applications, it is infeasible to use rCUDA for actual HPC applications since there does not exist a single static mapping policy that will be suitable for all kinds of workloads.

5.6 Workload Intensity

The impact on the GPU utilization intensity is demonstrated in Figure 5. Two new groups of workloads (W_6-W_{10} and W_{11}-W_{15}) are used in the experiment. The generating parameters of these workloads are the same as that of W_1 to W_5, except that the (T_c, T_g) of light workload is set to $(2625, 325)$ in W_6-W_{10} and $(1815, 1125)$ in W_{11}-W_{15}. As the results show, the dynamic policies are significantly effective only if enough underutilized GPUs exist. In the lighter

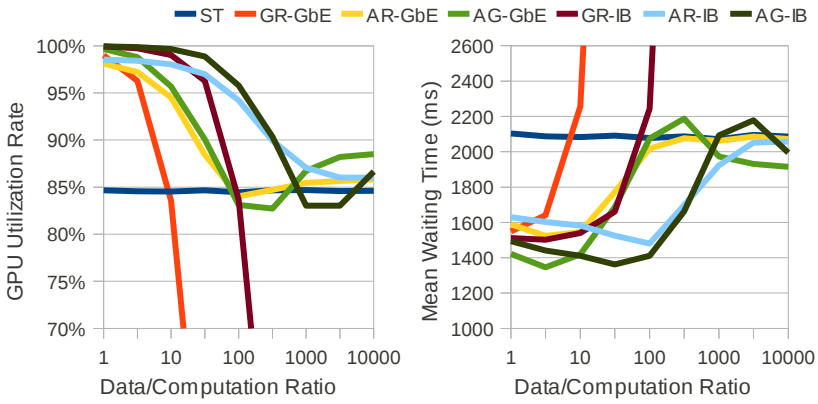

Fig. 6. Impact of network bandwidth.

group (W_6-W_{10}), up to 26% improvement can be observed, but in the heavier group (W_{11}-W_{15}) the improvement is limited by the existence of over-utilized GPUs.

5.7 Network Overhead and Efficacy of Adaptation

In this experiment, we examine the sensitivity of the policies to the network bandwidth and the data/computation ratio of the GPU kernels, which are two key factors that affect the network transfer overheads introduced by the remote execution of GPU kernels.

Figure 6 shows the system-wide GPU utilization of different policies and the underlying interconnect with varied D_u/T_g (data/computation ratio). Here D_u of the workload W_4 is sampled exponentially from $D_u = 1 \times T_g$ to $D_u = 10000 \times T_g$. As D_u/T_g increases, the overhead of remote-execution increases, which negatively affects the performance of the dynamic mapping policies (and especially of the GR policy). This indicates that the amount of transferred data or the network bandwidth plays an important role in making dynamic mapping policies effective and efficient. However, thanks to the adaptation mechanism, the performance of AR and AG can still be as good as ST when the ratio is extremely high.

The benefit of the adaptation mechanism can be clearly demonstrated with Figure 7. In this experiment, we explicitly assign several fixed values to α and β, and compare these fixed-weight random policies to AR. The result reveals that the fixed-weight may favor either the low data/computation ratio workload or the high data/computation workload. Only the adaptive-weight in AR can track the best performance over the entire range of data/computation ratio.

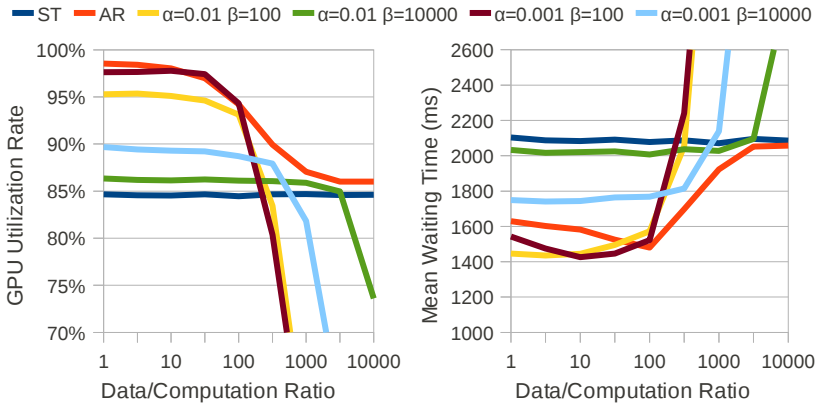

Fig. 7. Benefit of the adaptation mechanism.

5.8 Scalability

The scalability of the dynamic mapping policies is evaluated in the following two experiments: scalability with respect to the number of GPUs per node, and with respect to the number of nodes. Trace W_4 is used for the first set of experiments. For the second set of experiments, the four tested traces are half-sized, normal-sized, double-sized, and quadruple-sized versions of W_4. The network bandwidth is set to $100KB/ms$. The GR policy is excluded in this experiment due to its poor performance over lower-bandwidth network.

In the experiments, we observe higher possibility of underutilization by static mapping when more GPUs are installed in the cluster. In such cases, the necessity of an efficient GPU resource management policy becomes more significant.

The values reported in Figure 8 are the GPU utilization rate margin of the dynamic mapping policies over the ST policy. According to the results, both AR and AG exhibit good scalability over the number of GPUs per node. The AR policy also exhibits good scalability over the number of nodes. However, the AG policy doesn't scale well with the number of nodes. The key reason is that estimating the delay times in a larger-scale system becomes harder and less accurate. The larger amount of collaborative communication incurred during AG's decision making process also impairs its scalability over the number of nodes.

Since both AG and AR rely on certain amount of global information to make scheduling decisions, their performance could be significantly compromised if the system scales up to thousands of nodes. To accommodate such large systems, one effective way is to group the nodes into subsets and schedule remote GPU accesses within each subset. In future research, an alternative policy relying on distributed information and local estimation will be studied.

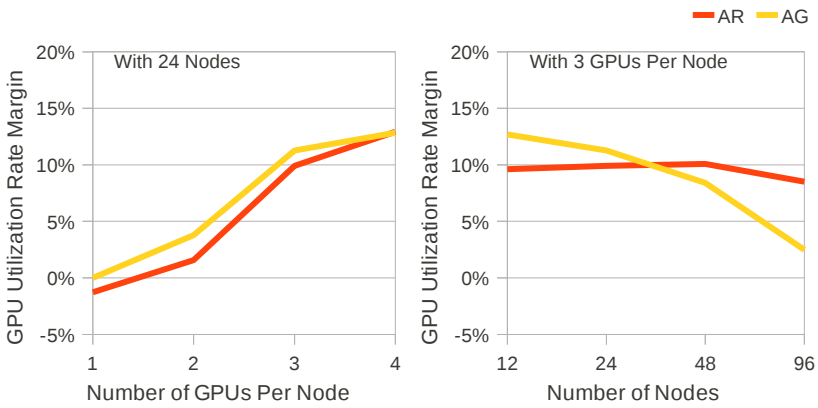

Fig. 8. Scalability of dynamic mapping policies over static mapping

5.9 Design Choices for the AR Policy

This set of experiments evaluates the performance of AR policy over certain design choices. As demonstrated in the previous experiments, AR is a balanced policy with several distinct advantages. One important choice in the implementation of this policy is how to maintain the distributed table about the GPU status. Real-time update is less desirable since it may incur extra network overhead. On the other hand, if the table is updated less frequently, outdated information may be used for GPU kernel/device mapping. We define a case to be false positive if an idle GPU is identified as busy, and false negative if a busy GPU is identified as idle. We used traces W_1 to W_5 to evaluate the performance of AR policy over different false positive ratio/false negative ratio. The values reported in Figure 9 are the GPU utilization rate margin of AR over ST. As shown in the figure, the performance is more sensitive to the false negative ratio than the false positive ratio. This implies that the status should be updated as soon as possible when a certain GPU becomes busy and the update is less urgent when a GPU becomes idle if the performance of the AR policy is valued.

6 Conclusion and Future Work

To address the performance degradation of GPU-assisted HPC system due to the mismatch between the physical node configuration and the GPU utilization of mixed workloads, we present the idea of dynamic kernel/device mapping, which relaxes the static binding between GPU kernels and local GPUs as in existing systems, and provide a sample design with the functionalities of remote kernel execution and GPU resource management, based on which the dynamic GPU allocation policies are further designed to balance the utilization of GPUs.

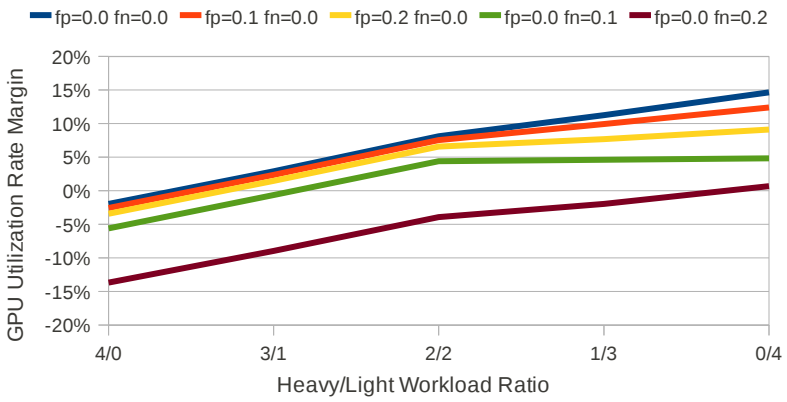

Fig. 9. The impact of false positive and false negative ratio on AR.

The benefit and efficiency of the strategies is demonstrated through simulation-based studies, which show that the dynamic mapping strategies outperforms the existing static kernel/device binding in terms of the GPU utilization and the mean waiting time for processes to acquire GPUs.

As we noted, communication intensive workload does pose challenges for the dynamic kernel/device mapping. However, if an advisable adaptive policy is adopted such as the proposed AR and AG policies, the dynamic mapping strategies will outperform existing methods for suitable workloads, and (effectively) fall back to the existing method for unsuitable workloads (e.g. communication intensive or very short kernels, both of which are untypical for GPU-assisted HPC applications). The dynamic mapping strategies provide the mechanism to improve GPU utilization for HPC systems when possible.

Additionally, existing GPU supports the concept of context where all the kernels launched from a user process are able to reuse the data that reside in GPU's global memory. Consequently, utilizing the same device for multiple kernels can save considerable amount of time for data movement. We plan to explore such context-based locality and design policies to re-utilize a remote GPU device for consecutive kernel calls from a process in order to reduce the cost of network transfer. We also plan to study the impact of process synchronization (e.g. MPI barriers) on the dynamic mapping kernel/device policy.

7 Acknowledgment

This work is supported by the US National Science Foundation under award number CNS-0845583.

References

1. A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for opencl based heterogeneous computing on clusters with many gpu devices. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1 –7, sept. 2010.
2. S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.
3. A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.
4. J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 224–231. IEEE, 2010.
5. G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A gpgpu transparent virtualization component for high performance computing clouds. *Euro-Par 2010-Parallel Processing*, pages 379–391, 2010.
6. V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009.
7. Khronos-Group. Opencl - the open standard for parallel programming of heterogeneous systems, 2011.
8. J. Kim, H. Kim, J. Lee, and J. Lee. Achieving a single compute device image in opencl for multiple gpus. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 277–288. ACM, 2011.
9. A. Merritt, V. Gupta, A. Verma, A. Gavrilovska, and K. Schwan. Shadowfax: scaling in heterogeneous cluster systems via gpgpu assemblies. In *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*, pages 3–10. ACM, 2011.
10. J. Nickolls and W. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56 –69, march-april 2010.
11. Nvidia. Gpu computing sdk, 2011.
12. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899, may 2008.
13. PBS-Works. Scheduling jobs onto nvidia tesla gpu computing processors using pbs professional. 2011.
14. S. Trofinoff. Scheduling gpus with slurm. 2011.