Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies

Alexander M. Merritt
Georgia Institute of Technology
merritt.alex@gatech.edu

Vishakha Gupta
Georgia Institute of Technology
vishakha@cc.gatech.edu

Abhishek Verma
Georgia Institute of Technology
abhishek.verma@gatech.edu

Ada Gavrilovska
Georgia Institute of Technology
ada@cc.gatech.edu

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

ABSTRACT

Systems with specialized processors such as those used for accelerating computations (like NVIDIA's graphics processors or IBM's Cell) have proven their utility in terms of higher performance and lower power consumption. They have also been shown to outperform general purpose processors in case of graphics intensive or high performance applications and for enterprise applications like modern financial codes or web hosts that require scalable image processing. These facts are causing tremendous growth in accelerator-based platforms in the high performance domain with systems like Keeneland, supercomputers like Tianhe-1, RoadRunner and even in data center systems like Amazon's EC2.

The physical hardware in these systems, once purchased and assembled, is not reconfigurable and is expensive to modify or upgrade. This can eventually limit applications' performance and scalability unless they are rewritten to match specific versions of hardware and compositions of components, both for single nodes and for clusters of machines. To address this problem and to support increased flexibility in usage models for CUDA-based GPGPU applications, our research proposes *GPGPU assemblies*, where each assembly combines a desired number of CPUs and CUDA-supported GPGPUs to form a 'virtual execution platform' for an application. System-level software, then, creates and manages assemblies, including mapping them seamlessly to the actual cluster- and node-level hardware resources present in the system. Experimental evaluations of the initial implementation of GPGPU assemblies demonstrates their feasibility and advantages derived from their use.

Categories and Subject Descriptors

C.1.3 [**Other Architectural Styles**]: Heterogeneous (hybrid) systems

General Terms

Performance, Experimentation, Design

VTDC'11, June 8, 2011, San Jose, California, USA.

Keywords

application scalability, heterogeneous clusters, CUDA, GPGPU virtualization, remote procedure call

1. INTRODUCTION

Limits on processor frequency scaling have pushed hardware developers to explore new avenues for increasing application performance. Consequently, general purpose graphics processing units (GPGPUs) have seen increased adoption across multiple areas of research in both the enterprise and high performance computing (HPC) domains and are gradually becoming core components of modern supercomputers. In the enterprise domain, GPGPUs offer increased application performance while enabling greater hardware consolidation. In the HPC domain, GPGPUs have become critical for improving compute performance. Example systems include the Amazon EC2 cloud infrastructure [1] as well as the Tianhe-1A and Nebulae supercomputers[1].

Past experience in HPC indicates that efficient exploitation of GPGPU systems can require extensive application tuning and/or hardware configuration to match application requirements. This implies application re-tuning for new hardware acquisitions or when application requirements change. Furthermore, GPGPUs continue to be treated as devices assigned to specific applications, thereby constraining flexibility in matching application requirements to hardware configurations. Consider, for example, a compute node configured with a high-end 8-core CPU coupled with a GPGPU device. Applications executing on such nodes must be structured so as to make use of all 8 CPU cores such that the attached GPGPU is efficiently exploited. At a larger scale, batch schedulers on high performance clusters must provide parallelized jobs with reservation-based access to sets of compute nodes configured with GPGPU devices; not all applications, however, will have the ability to run well on compute nodes given any specific hardware configuration. Finally, in enterprise environments, even if servers are shared across virtual machines, GPGPUs themselves are provided on an exclusive basis for use in hardware virtual machines [1, 12].

This paper proposes the concept of *GPGPU assemblies* which allow users to run applications on virtual platforms that can be created from a cluster of CPUs and GPGPUs based on application demands. An assembly management module maps these virtual platforms to underlying hardware, using methods that allow sharing and consolidation. The implementation of GPGPU assemblies

[1] http://top500.org

leverages the increasing presence of virtualization technologies, by defining *virtual GPGPUs* or *vGPUs* as representations of physical GPGPUs exposed to virtual machines. These vGPUs collectively form a virtual platform—an assembly— and are multiplexed on available local as well as remote GPGPUs without disrupting an application's execution within a virtual machine. The idea is to better match the needs of an application with respect to GPGPU capabilities. We demonstrate the feasibility and utility of this research through our prototype implementation called *Shadowfax*.

Shadowfax extends and enhances our previous single-node research platform called *Pegasus* [8]. Pegasus proposes an alternative approach to GPGPU computing, in which GPGPUs are treated like CPUs, as first class schedulable and shareable entities. The approach exploits the fact that shared access to GPGPUs can help obtain higher levels of GPGPU utilization and/or improved application performance. A limitation, however, is that applications are still limited to only using locally available GPGPUs, constraining them in how they can use combined CPU/GPGPU resources at a cluster level. This may prevent them from utilizing more GPGPUs than those physically associated with a compute node, thereby impeding application scalability and limiting throughput. Shadowfax addresses these limitations and is used to both identify and demonstrate the principle components constituting efficient assembly management for large scale GPGPU applications.

Summarizing, we make the following contributions:

- Improved application scalability: by permitting an application's hardware requirements to be fulfilled across physical machine boundaries, GPGPU assemblies enable continued scalability of applications.
- Increased application throughput: load balancing contending workloads and provisioning resources across a cluster can avoid oversubscribing GPGPUs on a single compute node to enable greater aggregate throughput
- Reduced application development effort: the use of standard programming models for writing such applications ensures a more stable abstraction irrespective of the scale. Programmers' time can be better spent towards improving compute logic for problem solving, rather than towards work partitioning across the cluster.

Related efforts already demonstrate the use of a remote invocation framework [5] using XMLRPC in the CUDA runtime API layer. The novel contributions of GPGPU assemblies are the flexibility they offer in establishing virtual platforms composed of potentially heterogeneous local and remote resources. Underlying hardware resources are dynamically shared among applications, using diverse and flexible resource management methods. Furthermore, by exploiting system virtualization technology, we decouple realization of a virtual platform from specific operating systems and application runtime requirements visible across an entire server platform. This is particularly important in enterprise and cloud settings, with recent work also showing benefits in the HPC domain [9, 13].

Experimental results from our prototype indicate that a dynamic matching algorithm will need to be well-informed about workload characterization: effects of virtualization, GPGPU access latencies and bandwidth, as well as both CPU and GPGPU contention on application performance and its ability to scale significantly depend on an application's runtime behaviors, such as being CPU- or GPGPU-bound, and expectations of the underlying hardware. We provide evidence and discussion of these metrics in Section 4. The remainder of the paper is organized as follows. Section 2 elaborates on our concept of a GPGPU assembly as well as which preliminary components Shadowfax fulfills. Section 3 provides implementation detail, followed by results from the initial evaluation gathered on our current prototype in Section 4. Related work and discussion of our future research directions appear at the end.

2. GPGPU ASSEMBLIES

The bottom half of Figure 1 shows the hardware platform targeted by our research: cluster systems consisting of host nodes with one or more attached GPUs. Examples of such systems include Amazon's EC2 cloud platform or ORNL's Keeneland machine, where cluster nodes are typically homogeneous, with possibly identical types and numbers of GPGPUs. In contrast, with GPGPU assemblies, system software can configure diverse platforms with heterogeneous node configurations for different applications based on their requirement. The inter-node differences in such heterogeneous platforms can be intentional, or a result of partial hardware upgrades occuring over time. Our solution does not make assumptions about the presence of particular GPGPU families or types on each node, but instead, can recognize their capability differences and make allocations accordingly.

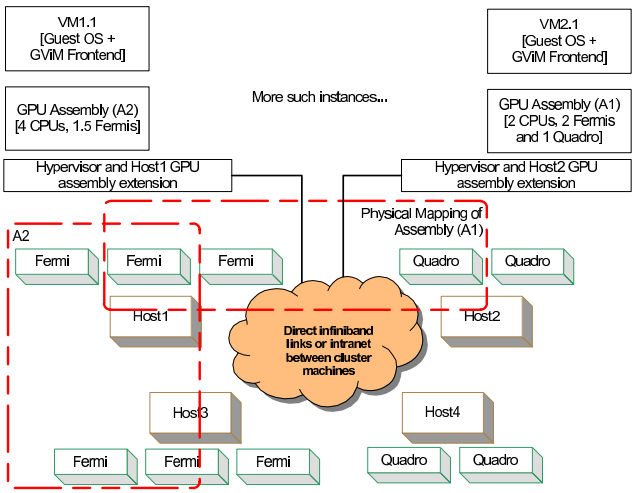


Figure 1: GPGPU assemblies spanning multiple GPGPU and CPU (implicit) nodes across a cluster for the example VMs. Partial overlap indicates partial utilization

For these platforms, specific answers sought by our work include the following:

1. *Platform capacity:* how to improve overall platform capacity and sustain improved application throughput?
2. *Platform scalability:* how to provide scalability in the presence of increasing application resource needs, without requiring upgrades to newer, more powerful hardware?
3. *Suitable resource allocation for cost of reservation:* should an application, despite being heavily GPGPU-bound, pay to reserve more cluster nodes instead of paying just the cost of the accelerators and perhaps marginal cost for CPU usage because accelerators still need host cores? The unneeded CPU, storage and I/O resources that come with typical allocations increase the cost if the resources are unwanted.

To address these issues, we propose the use of *GPGPU assemblies*: virtual platforms providing applications with required CPU and GPGPU resources. This construct realizes such resources by assembling a subset of all components from potentially different physical nodes in a cluster. GPGPU assemblies then allow applications to access these resources as if they were local, albeit with, for

example, increased access latencies whenever application code executes on a remote GPGPU. Regarding 1., system throughput is improved by allowing sub-node resource allocations and thereby supporting larger numbers of applications. By permitting the shared use of all physical resources, including GPGPUs, we ensure improved resource utilization, resulting in improved platform capacity. Regarding 2., GPGPU assemblies allow applications to execute across individual node boundaries, thereby scaling beyond the limits imposed by single physical nodes. This is possible because even on a single node, applications' GPGPU components—CUDA kernels— have well-defined code and data boundaries, making it easier to partition applications across machines without shared memory support. Finally, regarding 3., by allocating to applications matching portions of the underlying physical resources, our approach achieves a better matching between the application resource needs and the allocations made by the system management software. This can improve the overall system capacity and potentially reduce the costs incurred by applications. Note that programming models for GPGPUs, such as CUDA and OpenCL, require CPU resource allocations. This necessitates an inclusion of these resources in establishing an assembly; the utilization of which can be kept to a lower portion, as needed by the application

There is flexibility in how GPGPU assemblies may be built. If end users insist on low GPGPU call latencies, for instance, assemblies may be limited to those that only use locally accessible GPGPUs, avoiding the use of cluster-level interconnect. Conversely, substantial gains may be made for codes able to robustly scale despite higher per-call latencies and/or moderate data movements between CPUs and GPGPUs. Our work demonstrates that the use of GPGPU assemblies can improve the aggregrate GPGPU performance available to high performance codes. Or, stated differently, assuming applications do not constrain their own ability to scale, benefits of accessing any number of non-local GPGPUs may offset the time taken to reach them, surmounting any individual cluster node's ability to scale an application. This includes migrating a VM itself to a node where GPGPU load may be lower, as applications are still limited to accessing only locally-attached devices. Faster networks, for example InfiniBand, connecting cluster nodes can further lower per-call latencies.

Figure 1 illustrates this design at a high level. We assume the use of virtualization as shown in the figure due to the benefits it offers. As mentioned earlier, we have extended and enhanced the Pegasus system [8] to support GPGPU assemblies. All GPGPUs participating in an assembly may be programmed as a local device from the perspective of an application within a guest VM. To facilitate this, assemblies, i.e., virtual execution platforms for applications, are instantiated alongside their respective VMs and remain consistent throughout their lifetime. GPGPU assemblies are consistent with the CUDA programming model, thus reducing the effort for programming, and moving management to lower layers without additional burden on programmers. The Pegasus frontend installed in VMs forwards CUDA calls to the management domain on the system as explained further in Section 3. However, from the perspective of the application running inside a VM, it has access to the number of GPGPUs that were requested for the virtual platform created for the VM. This principal component in an assembly is a *virtual GPGPU instance* or *vGPU*. It is the intermediate representation between an execution stream originating from the application and the physical GPGPU on which the stream is executed, either co-located with the application or present on another networked machine. An assembly contains a number of vGPUs equal to the number of physical GPGPU participants irrespective of their location. While a vGPU gets associated with one physical GPGPU, a physical GPGPU could host multiple vGPUs depending on the resource management scheme on a node. Thus, multiple applications can end up sharing physical GPGPUs so long as the performance is acceptable.

3. SYSTEM IMPLEMENTATION

This section presents a prototype implementation of GPGPU assemblies, termed *Shadowfax*, targeting GPGPU-based clusters like Keeneland [17]. Figure 2 depicts its primary implementation modules. The system expands on our previous research effort, Pegasus [8], which provides GPGPU virtualization at the CUDA API layer within virtual machines on the Xen [2] hypervisor, allowing unmodified applications to execute. The various components leading to our implementation of assemblies are discussed next.

Virtualization of GPGPUs on a single node: A kernel module within the guest establishes shared memory regions with the management domain—ring buffers for marshalled functions plus space for application data—and event channel mechanisms for communication and coordination of requests it receives from a user-level interposer library linked with CUDA applications (implementation details discussed in [7]). Requests received in the management domain are queued and assigned per-guest poller threads that can be scheduled to execute requests on physical GPGPUs on behalf of the guest. This scheduling can be based on certain policies suitable for the system. Each poller thread executes queued requests from the ring buffer on behalf of the associated guest application and returns results along the same path. NVIDIA's CUDA runtime and driver, present in dom0, manage state and protection contexts at host CPU thread granularity (e.g. threads created by pthread).

Implementing vGPUs: In order to enable seamless execution of CUDA applications while keeping them agnostic of the physical GPGPU locations, two specific realizations of vGPUs exist in our implementation, one representing a local GPGPU and the other representing a non-local GPGPU resident on another physical machine as depicted in Figure 3. For each kind of vGPU, a polling thread is created and attached to a call buffer supplied by the VM. Together, a call buffer and polling thread directed to use the locally available NVIDIA runtime API constitute a *local* vGPU. A *remote* vGPU also consists of a polling thread attached to a shared-memory call buffer, but does not participate in scheduling schemes implemented within the management domain. Instead, it connects to the designated remote machine specified in the GPGPU assembly using a common port to establish a GPGPU link on the other side. The receiving machine has an admission thread which listens for these incoming link requests, and once approved, spawns a *remote domain*, or *server*, thread. This thread has multiple purposes. From the client machine's perspective it manages the receiving end of RPC communication and is responsible for unmarshalling requests, executing them and returning their results. From the server machine's perspective, the locally-executing remote domain thread acts as a fake guest VM, essentially emulating the booting up of a real guest VM. It triggers the creation of a polling thread the same way the creation of a VM does but because it is not a real VM it does not have a separate and isolated address space or methods for using event channels for notifications. Instead, it allocates space for a call buffer and application data and attaches this to the polling thread, allowing it to believe it is of type *local* vGPU when it is in fact the tail end of a remote vGPU. As on the client machine, the polling thread on the server machine participates in scheduling schemes when enabled. In contrast, the remote domain or server thread is not scheduled, allowing it to continue processing requests off of the network, even while the associated polling thread has been scheduled out after the expiration of its timeslice.

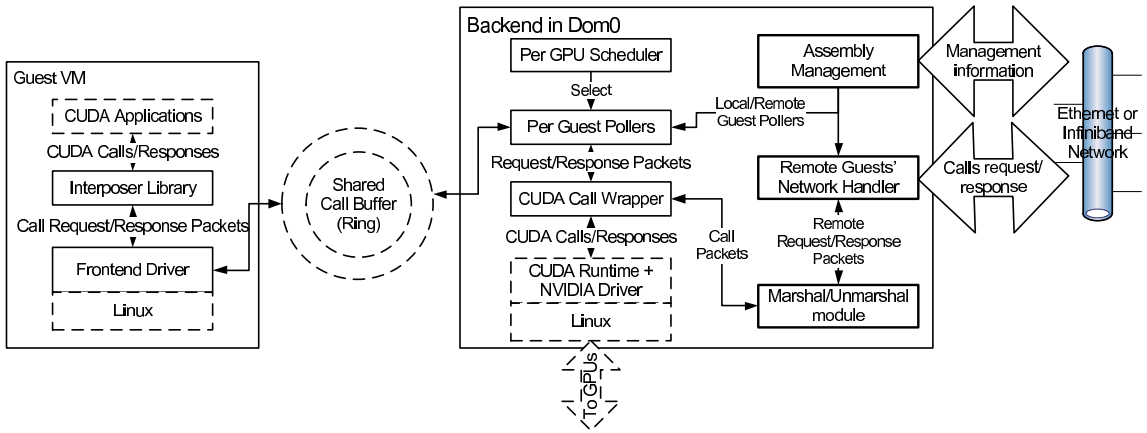


Figure 2: Implementing GPGPU Assemblies

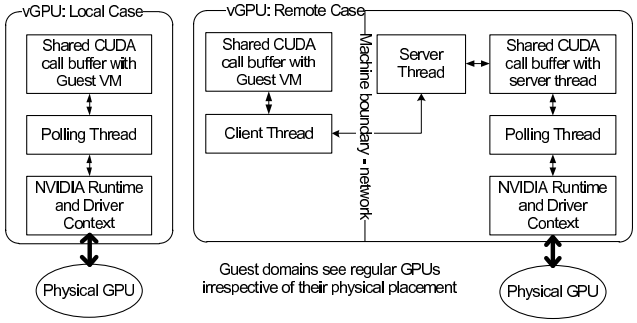


Figure 3: Implementation perspective of virtual GPUs in Shadowfax. CPU allocations are adjusted using credits assigned to VMs requesting the assembly

Thus, a virtual GPGPU instance does not constitute a single structure in our implementation but is in fact a composition of elements. To reduce runtime overheads, all call paths are instantiated before an application is executed.

Optimizations for remote execution: Anticipating additional runtime overhead from the use of remote GPGPUs, we enable support for *batching* calls and their data. The CUDA API has a mixture of synchronous and asynchronous function calls. We can send a set of asynchronous calls followed by a delimiting synchronous call as a batch to reduce network overhead. We queue calls labeled as asynchronous by the CUDA API [11] as they are picked by the scheduled domain's polling thread. Functions which carry input parameters (i.e. pointers to data, such as `cudaMemcpy-toDevice`) have their data copied to the batch directly from the VM-shared memory locations. A batch is defined to be full under either of the following situations: 1. memory allocated for the batch has reached its limit or cannot contain the input parameter for the next function call; 2. a synchronous function with or without output parameters or critical return values has been issued by the application. The latter is a hard requirement, as we cannot assume the application does not immediately depend on data produced by such a call.

Assemblies: In Section 2, we defined a GPGPU assembly to represent a virtual execution platform composed of vGPUs, each of which *may* be mapped onto a different physical GPGPU within the cluster of networked machines. Next we discuss our proposed implementation for realizing these assemblies. Applications interface with GPGPUs within an assembly through the CUDA runtime API and are led to believe all are local devices, allowing for transparent management and creation of vGPUs in the management domain. Application requests to bind to specific GPGPUs can be overridden, and queries for information pertinent to GPGPUs within its assembly can be injected to return values to function calls captured on the shared-memory call buffer. For example, information returned for calls to `getDeviceProperties()` or `getDeviceCount()` from the guest will not be given the exact information returned from a direct call to the proprietary driver in the management domain. Instead, information returned will be dynamically generated to reflect those GPGPUs participating in the application's assembly.

When a guest VM is created, it can be assigned a GPGPU assembly created by the *Assembly Management Module* residing in the backend within Dom0 (as shown in Figure 2). Therefore, the creation of an assembly for a guest VM can be viewed as an extension to the way guest VMs request certain number of VCPUs, devices and disks through a configuration file with the exception that the Shadowfax backend handles the creation instead of the systems layer.

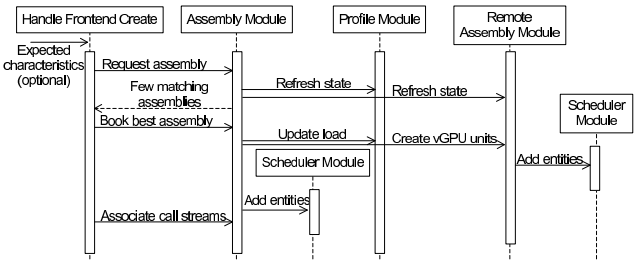


Figure 4: Sequence of actions and modules involved in creating GPGPU Assemblies

Figure 4 shows the sequence of actions required by the backend to create an assembly whenever a VM is booted. This activity diagram shows the modules or functions from the implementation that are involved and the primary messages that need to be exchanged between them on the same machine as well as across machines. Each time a new VM boots, the backend running in Dom0 is triggered, invoking the *handle domain create* method. This method is responsible for interacting with the assembly module, requesting for the creation of an assembly conforming to the corresponding guest VM's requirements. The assembly module in turn may request for up-to-date information for the various resources on its

local as well as remote machines. The local information can be maintained as part of a profile module that tracks the static resource properties as well as dynamic load characteristics (e.g. available memory on GPGPUs). For remote GPGPU information, requests can be sent to the assembly modules running on other cluster nodes. Once the local assembly module fetches sufficient data, it can return a set of equivalent assemblies matching the request. It is possible to implement policies on top of the assembly module, for picking the right kind of assembly, or for requesting that a certain assembly be booked. The module can then create vGPU components as shown in Figure 3, excluding call buffers, as these are instantiated by the VM frontend for inserting requests, shown in Figure 2. The assembly module can also take care of inserting polling threads into the scheduler on the local machine. Ring buffers carrying CUDA calls can be dynamically attached to vGPU polling threads within the assembly.

For the experiments outlined in this paper, an algorithm dynamically matching application threads to GPGPUs was not employed. As the experiments were strictly controlled, we manually determined each assembly avoiding challenges associated with concurrent requests for similar resources (each code path was instructed to follow a predetermined course). This paper aims to present the potential for a framework to support such dynamic behavior along with example scenarios targeting specifically configured workloads; dynamically determining these configurations and appropriate assemblies is a main focus of our on-going efforts.

4. EVALUATION

Experimental System. Our experimental system consists of two GPGPU cluster nodes directly connected via 1 Gbps Ethernet fabric. Each node consists of one 64-bit 3 GHz quad-core Intel Xeon X5365 CPU, 4 GiB of DDR2 main memory, an NVIDIA 8800GTX GPGPU on the "client" and a 9800GTS on the "remote" node, interfaced over the PCI-Express bus. Xen 3.2.1 hosts the management and guest domains, which run 2.6.18 Linux and the 169.09 NVIDIA driver (a patch for only this version of the driver has been made available to support GPGPU access from within the management domain in Xen).

Category	Source	Benchmarks
Financial	SDK	Binomial(BOp), BlackScholes(BS)
Media processing	SDK or parboil	MatrixMultiply (MM), MRI-FHD
Scientific	parboil	CP

Table 1: Summary of Benchmarks

Benchmarks. The benchmarks used for our evaluation address two principal uses for future GPGPU systems: 1. computationally expensive enterprise applications including financial codes or web applications requiring substantial processing resources due to their manipulation of media for large numbers of end users, and 2. parallel codes that aim to speed up computations critical to their performance. We use multiple benchmarks from both the parboil benchmark suite [14] and the CUDA SDK 1.1, as shown in Table 1. Critical to our performance studies is to conduct benchmark runs so as to represent a mix of applications which vary in 1. dataset sizes and data transfer frequency that determine network sensitivity, 2. iteration complexity which is a good measure for estimating 'kernel' size, and 3. frequency of kernel launches or rate of CUDA call execution which relates directly to the degree of coupling between CPUs orchestrating accelerator use and the

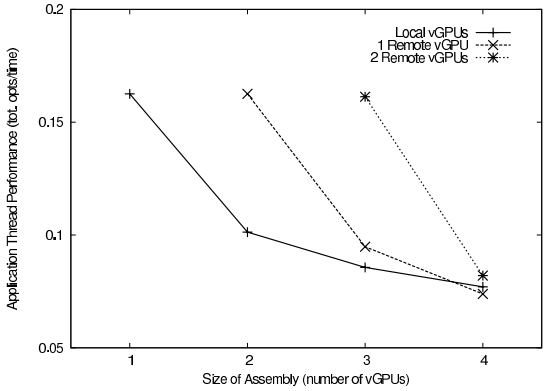


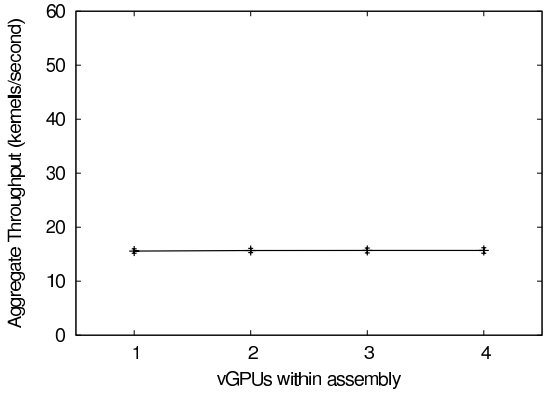
Figure 5: Effects of GPGPU contention on application thread performance running BinomialOptions, varying the assembly size and composition. All local vGPUs attach to the same accelerator.

GPGPUs executing these kernels. The reason for doing this is to be able to identify workload characteristics when establishing the most appropriate assembly for an application. Using statically configured workloads in our preliminary analysis enables us to make observations about the influence of decisions made by the assembly module when executing a dynamic matching algorithm on application performance. In our evaluation, some applications identify as 1. throughput-sensitive (e.g. BOp, MM), 2. latency-sensitive (all scientific), or 3. both (BS, CP). A benchmark is considered throughput-sensitive when its performance can be measured by the number of operations or computations performed per second, and latency-sensitive when it frequently issues CUDA calls demonstrating a sensitivity to virtualization and/or accelerator scheduling and reachability delays observable through execution time.

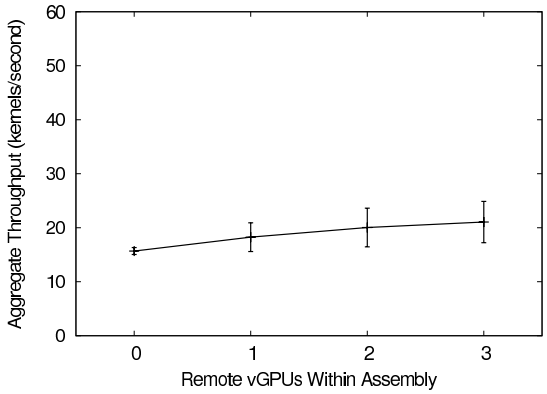
Methodology. Within multi-GPGPU systems, applications scale by spawning threads and associating each with a particular device. Given available CPU resources this allows an application to drive concurrent execution streams to devices. In our prototype we model this behavior by designating a single application within a virtual machine in Xen to act as one of these independent call streams and all VMs within a system as the collective "application". In our experiments each VM is configured identically, with 256 MiB of memory, one VCPU pinned to a physical core and all VMs containing the same single-threaded benchmark application for an experiment. VM VCPUs are pinned, alternating among two of the four cores available on a cluster node. Two Dom0 VCPUs are assigned the remaining two cores, with the polling threads attached to respective VMs similarly pinned.

A key consideration with this work is to examine and adapt to new limitations imposed on the system software by the hardware and software components within the environment as assemblies and their applications scale outward. Indeed, we make no claims that enabling applications to use non-local hardware will improve their performance. Our focus lies with the ability of the underlying system software to enable applications to scale as a whole, possibly to a larger workload, given the various system component configurations such as the availability of CPU cores, memory and bandwidth, in addition to the observed workloads executing in the system.

Systems' static configurations constrain applications in the number of GPGPUs they may use. Contention on individual components within a system increases with heavier workloads. Figure 5 illustrates the impact on an application thread's performance with increasing contention placed on a local GPGPU. Each vGPU car-



a) All vGPUs are local, attaching to one common GPGPU.



b) An assembly of four vGPUs with increasing number of remote vGPUs.

Figure 6: Scaling a GPGPU workload (BlackScholes) using an assembly. Performance aggregated across all assembly vGPUs.

ries the workload of the BinomialOptions benchmark, saturating the GPGPU with a single vGPU. The solid curve represents the performance of this thread as more vGPUs are instantiated and attached to the same device. The application thread experiences longer wait times while issuing requests, forced to share the device. Reattaching vGPUs to remote GPGPUs reduces the contention (moving vertically between data points). This suggests 1. an increased aggregate performance across all vGPUs within the assembly is possible, and 2. additional remote vGPUs, for this application characteristic, do not impact the local vGPU's ability to service its application thread.

Figure 6 juxtaposes two experiments conducted with Shadowfax, illustrating the throughput of an assembly aggregated across all vGPUs themselves. Figure 6.a shows the kernel launch rate of an assembly as more local vGPUs are instantiated and attached to application threads, each with idential workloads. A horizontal curve confirms that while more work is available for the GPGPU, it cannot sustain rates greater than it was designed for.[2] As more application threads are spawned, each will have opportunity to submit less work, but as data is aggregated across all vGPUs, throughput remains constant. Migrating portions of the total work to available devices, even if non-local, would allow the assembly, and as a result the application, to scale. Figure 6.b illustrates just this.

[2]Additional contributors to the "saturation" of a GPGPU include lack of kernel interruption and concurrent kernel execution (prior to Fermi), leading to more unstable environments when shared.

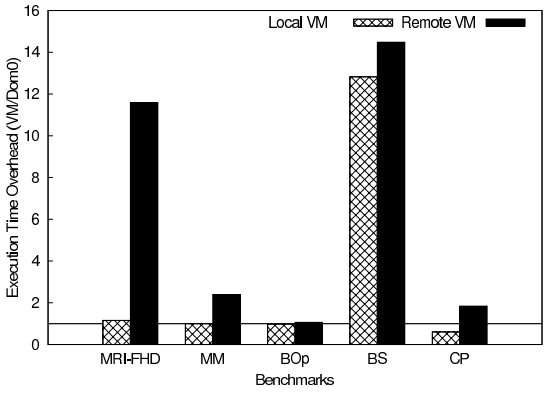


Figure 7: Application sensitivity to virtualization and network fabric overheads (Ethernet), normalized to direct execution time measurements within Dom0

Aggregate throughput increases as more work is offloaded using remote vGPUs. While only slight, it does provide a foundation for further improvements. We attribute the shallow rate of increase to both the high-latency interconnect and our unavoidable oversaturation of CPU resources. Future advancements include upgrading our testbed to use an Infiniband fabric, host greater-capability CPUs as well as adding additional cluster nodes.

We argue that the layer introduced by a GPGPU assembly presents a feasible platform solution for enabling a more effective use of the GPGPU hardware while scaling workloads. To avoid contention on a device, an application can either select an available or less loaded GPGPU, or wait for an upgrade to the cluster node's internal hardware, adding more or newer generations of GPGPUs—a less probable option. Electing to send execution streams through remote vGPUs, the application enables itself to complete work faster than if it were multiplexed among other competing workloads on the device. Figure 7 shows this is possible for workloads which exhibit characteristics similar to the BinomialOptions (BOp) benchmark. Each kernel has a relatively long execution time and is launched at a slower rate[3] than in the BlackScholes (BS) benchmark; the latter exhibits a higher issue rate of 'smaller' CUDA kernels. The higher issue rate of CUDA requests in BOp exacerbates overheads introduced by virtualization, but with the observed batching value, shows negligible performance differences between the use of a local and remote vGPU. Workloads similar to MRI-FHD indicate locally attached vGPUs remain the most viable option, with negligible virtualization overhead.

In enabling applications to scale outward using remote vGPUs, multiple opportunities exist for increasing the efficiency observed through the entire data path. In varying the characteristics of workloads on our prototype system, we observed multiple factors contributing significantly to the efficiency of an assembly's ability to scale an application. One example presented here examines the combination of highly asynchronous applications and an Ethernet interconnect. Ethernet is optimized to efficiently handle bulk data transfers for sustaining high bandwidth rates, albeit with relaxed guarantees on latencies for individual packets. Application execution streams exhibiting a high degree of asynchrony benefit from batching. Figure 8 illustrates this for the BlackScholes workload. In this specific configuration, a batch size greater than 64 CUDA requests shows negligible improvement, as the performance is almost

[3]The original BinomialOptions issues one kernel; we modified it to issue 25 launches throughout its execution.

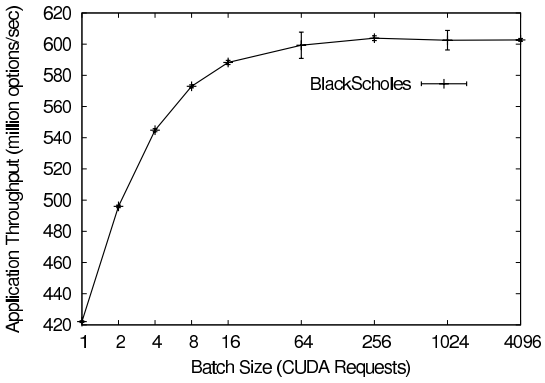


Figure 8: Effects of request batching on application throughput with one vGPU attached to a remote (Ethernet) GPGPU

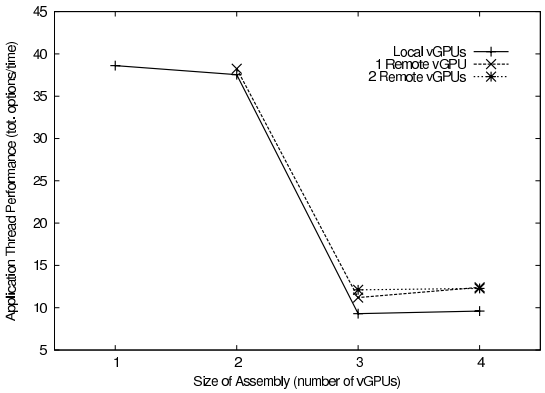


Figure 9: Effects of CPU contention on application thread performance running BlackScholes, varying the assembly size and composition. All local vGPUs attach to the same accelerator.

that of using a local vGPU (it compares with the top-left data point in Figure 9). This batching value has been used in our other experiments when assigning a remote vGPU. We note, however, that other values may show more benefit for different workload characterisitcs or degrees of asynchrony.

Figure 9 emphasizes an additional bottleneck where high kernel issue rates have a detrimental effect on overall application performance with limited CPU resources. This graph illustrates the performance as measured from an application thread. A negligible difference is noticed after instantiating a second local vGPU, and can be explained from the thread-CPU pinning configuration. The third application thread shares the same CPU as the one measured, competing directly for this resource. Both have a reduction in opportunity to issue work on the GPGPU, dramatically reducing per-thread throughput. Reattaching to a remote GPGPU shows almost no gains as the CPU is still required for moving work to its destination. Our data shows that neither local nor remote vGPUs can remove this impediment.

A final examination was performed on individual CUDA function calls themselves. Using profiling, we are able to observe the contribution of each layer within the remote vGPU call path, including overheads introduced by an Ethernet interconnect. Network overheads contribute the most, leading us to conclude that with the use of a lower-latency interconnect, this overhead may be mitigated.

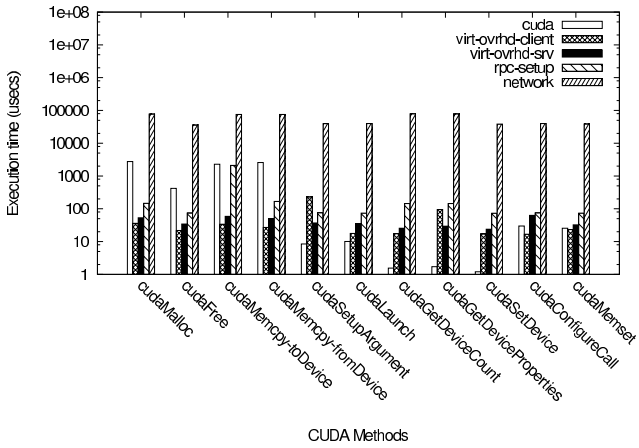


Figure 10: Profiling individual CUDA calls; breakdown of remote vGPU path (using 8 KiB data sizes for memory copies)

Label	Description
cuda	Native CUDA call latency on server
virt-ovrhd-client	Time spent on the client only
virt-ovrhd-serv	Time spent at the server, minus 'cuda'
rpc-setup	Time spent performing RPC marshalling
network	Ethernet latency

Table 2: Legend for Figure 10

5. RELATED WORK

There have been some efforts to virtualize GPGPU systems, typically with NVIDIA GPGPUs connected to general purpose CPUs, like GViM [7] and vCUDA [15]. Both are CUDA API level virtualization solutions that follow a similar approach. Neither of the systems, however, support remote access to GPGPUs. We have used technologies from GViM and added support for remote access along with adding support for creating GPGPU assemblies.

The rCUDA framework [5] enables the concurrent use of non-local CUDA-compatible devices. However, applications in this framework are forced to execute on remote nodes. There is no support for resource sharing on nodes even if local GPGPUs are available. Furthermore, host and device code require separate compilations. Shadowfax enables local as well as remote GPGPU execution, whichever is the best choice at a given time, without requiring separate compilations. It supports unmodified applications in multiple virtual machines to seamlessly share local as well as remote GPGPUs. It can scale to any number of GPGPUs with local as well as remote access, forming assemblies such that multi-GPGPU capable applications are constrained only by cluster limits. Our *GPGPU assembly* abstraction makes it easier for virtual machines to request any desired number of GPGPUs while booting, deferring to the underlying Shadowfax runtime the responsibility of multiplexing requests across these resources.

Microsoft's RemoteFX [10] exposes a WDDM driver with the virtual desktop allowing multiple virtual desktops to share a single GPGPU on a Hyper-V server. Xenserver's multi-GPGPU passthrough [4] solution now allows for the sharing of GPGPU-based host workstations across multiple users, one per GPGPU card. Our work, however, focuses on the general purpose computing aspects of GPGPUs and allows for scaling computations across GPGPUs connected to nodes in a cluster. Another difference is the placement of computation. Shadowfax sends both computation and data to remote nodes whenever a VM is scheduled to a remote vGPU while