A Systems Perspective on GPU Computing: A Tribute to Karsten Schwan

Naila Farooqui
Georgia Institute of Technology
Atlanta, Georgia
naila@cc.gatech.edu

ABSTRACT

Over a distinguished career, Regents Professor Karsten Schwan has made significant contributions across a diverse array of topics in computer systems, including operating systems for multi-core platforms, virtualization technologies, enterprise middleware, and high-performance computing. In this paper, we summarize his legacy of key research contributions in general-purpose GPU computing. His vision encompassed the conceptualization, implementation, and demonstration of systems abstractions and runtime methods to elevate GPUs into first-class citizens in today's and future heterogeneous computing environments. To this end, his contributions include novel scheduling and resource management abstractions, runtime specialization, and novel data management techniques to support scalable, distributed GPU frameworks.

CCS Concepts

•**Computer systems organization** → **Single instruction, multiple data;** *Heterogeneous (hybrid) systems;*

Keywords

GPU; High-Performance Computing; Scheduling; Virtualization; Resource Management; Dynamic Instrumentation

1. INTRODUCTION

GPUs have become pervasive in today's computing systems, due to their ability to provide significant gains in both performance and energy use for a wide variety of applications, coupled with recent improvements in their programmability. General-purpose GPUs can enable high throughput in several application domains, including data-intensive scientific applications [5], physical simulations [26], financial applications [29], and big-data applications [38, 31]. This has given rise to extensive work in high-level programming frameworks [9, 31, 7, 30, 22, 6] that insulate the programmer from the GPU's low-level architectural complexities. Such frameworks have improved programmer productivity and enabled rapid adoption of GPU-based systems to a large number of application domains. However, there remain a number of challenges in realizing the performance and productivity potential of GPU-based systems.

In spite of the tremendous performance increases provided by GPUs, operating systems and virtualization platforms have traditionally lagged behind in adopting GPUs as first-class schedulable entities. This has been in part due to closed-source, vendor-specific driver support in GPUs, which hide low-level details from operating systems and hypervisors. As a result, GPUs have generally been treated as secondary devices, with restricted facilities for scheduling and resource management. A driver-based execution model not only inhibits coordinated use of heterogeneous compute units often desired by today's enterprise workloads, but also fails to provide strong fairness and isolation guarantees in multi-tenant environments.

Manifesting performance improvements for individual applications on GPUs also remains effort intensive due to the combination of GPU's specialized architectural features, such as memory hierarchies, caches, and thread geometries, and today's increasingly complex applications. It is well-known that GPU hardware is designed to explicitly take advantage of regularity, characterized by workloads with minimal synchronization, high compute intensity, and predictable data-access and control-flow patterns. However, such pronounced regularity is not the common case for many of today's applications, which fundamentally rely on unstructured and irregular data and control-flow access patterns. GPU acceleration has proven profitable for irregular applications [8, 24, 35, 37], but typically requires the programmer to implement and compare multiple code versions that exercise different combinations of the GPU's architectural features. Moreover, the efficacy of such optimizations are highly data-dependent for irregular applications.

Furthermore, since GPUs are bandwidth limited and may have limited memory capacities (as in the case of discrete GPUs), their efficacy in distributed environments depends on novel runtime support for data transfer and management in the presence of potentially dynamic and irregular access patterns. Other useful systems abstractions for accelerator-based environments, such as providing checkpointing mechanisms in high-performance computing (HPC) systems, also depend on reducing this bandwidth impact.

Professor Schwan, one of the most prolific researchers of our time, has made numerous contributions in computer systems, in topics ranging from operating systems for many-

GPGPU-9, March 12-16, 2016, Barcelona, Spain

 ISBN 978-1-4503-4195-0/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2884045.2884057

core platforms, to enterprise middleware, to high-performance and parallel computing. Some of his recent work on GPUs addressed many of the challenges articulated above. Specifically, his vision advocated novel systems abstractions and runtime methods to elevate GPUs to first-class citizens in today's computing environments. Through his students and colleagues, he influenced the landscape of GPU computing with his articulation of key systems challenges, deep insights into performance bottlenecks, and perspectives on creative solution approaches. In this paper, we summarize some of Professor Schwan's main contributions in GPU computing, as a tribute to his research legacy.

His key contributions in GPU computing include:

- Virtualization and resource management support to fairly and efficiently share GPUs in multi-tenant cloud and HPC environments.
- Runtime specialization, using instrumentation-driven and profile-guided methods, to achieve performance scaling of data-intensive applications across emerging heterogeneous GPU processors.
- Novel data transfer and management techniques to achieve high system throughput and fairness in distributed GPU environments, as well as enable novel systems software abstractions, such as accelerator-based checkpointing in HPC systems.

The rest of the paper is organized as follows. Section 2 summarizes virtualization and resource management support proposed for GPU-based systems to enable fairness and efficiency. Section 3 surveys runtime specialization methods for heterogeneous GPU platforms, followed with Section 4, which provides an overview of novel data management techniques to support distributed GPU runtimes. Section 5 discusses future directions, and Section 6 provides concluding remarks.

2. VIRTUALIZATION AND RESOURCE MANAGEMENT

While GPUs have made their mark in both high-performance and cloud environments, the inability to manage GPUs directly has relegated them to second-class citizens in most computing environments. GPU vendors shield operating systems and runtimes from directly impacting GPU scheduling and resource management policies by hiding such details behind drivers that only expose higher-level application APIs [28, 36]. In light of these limitations, the general approach for impacting GPU scheduling and resource management has been to provide a middle runtime layer that sits between the applications and operating systems, which can interpose the application APIs to provide underlying scheduling and resource management abstractions. While such methods are unable to directly modify or influence the GPU's hardware-based, thread-level scheduling, they are still able to provide important systems guarantees for cloud and HPC environments, such as high throughput, fairness, and/or utilization.

Much of Professor Schwan's research in this space uses interposition as a basis for providing important systems guarantees for GPU-based computing environments. Key contributions of his research, therefore, include novel scheduling and resource management methods that achieve important systems goals for both cloud and HPC environments, by elevating GPUs to first-class schedulable entities, in spite of the limitations of current GPU execution models.

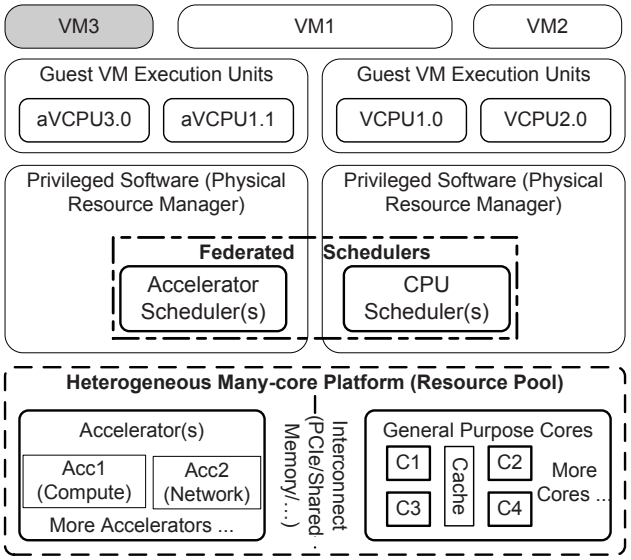

Figure 1: Logical view of the Pegasus system

2.1 Coordinated Scheduling for Virtualized GPU Systems

As noted earlier, operating systems and hypervisors have lagged behind in adopting GPUs as first-class citizens in heterogeneous computing environments. The Pegasus [16] system is the first of its kind to advocate uniform resource usage models for all cores on heterogeneous chip multiprocessors, including accelerators such as GPUs. It smartly manages GPUs by leveraging virtualization technology in cloud computing and high performance infrastructures. Specifically, the Pegasus hypervisor extensions make accelerators into first-class schedulable entities, which can be shared by multiple tasks. Task mappings to processors are also dynamic, within the constraints imposed by the accelerator software stacks. Second, Pegasus exposes heterogeneity, in terms of different GPU and CPU capabilities, to the applications and guest virtual machines that are capable of exploiting such heterogeneity. Finally, Pegasus advocates coordination as the basis for resource management, providing novel scheduling methods to align accelerator resource usage with platform-level management. In order to achieve this, Pegasus schedulers operate above the underlying native schedulers so as to influence the actions of the underlying schedulers rather than to replace them. In this way, the Pegasus system allows for sophisticated and diverse scheduling methods that underlying resources may require, while enabling the preservation of virtual platform properties, such as fair-sharing and prioritization.

Figure 1 presents the logical view of the Pegasus architecture. In this view, general-purpose and accelerator tasks are schedulable entities mapped to virtual CPUS (VCPUs) characterized as general purpose or as 'accelerator.' Since both sets of processors can be scheduled independently, platform-wide scheduling requires Pegasus to federate the platform's general purpose and accelerator schedulers. Federation is

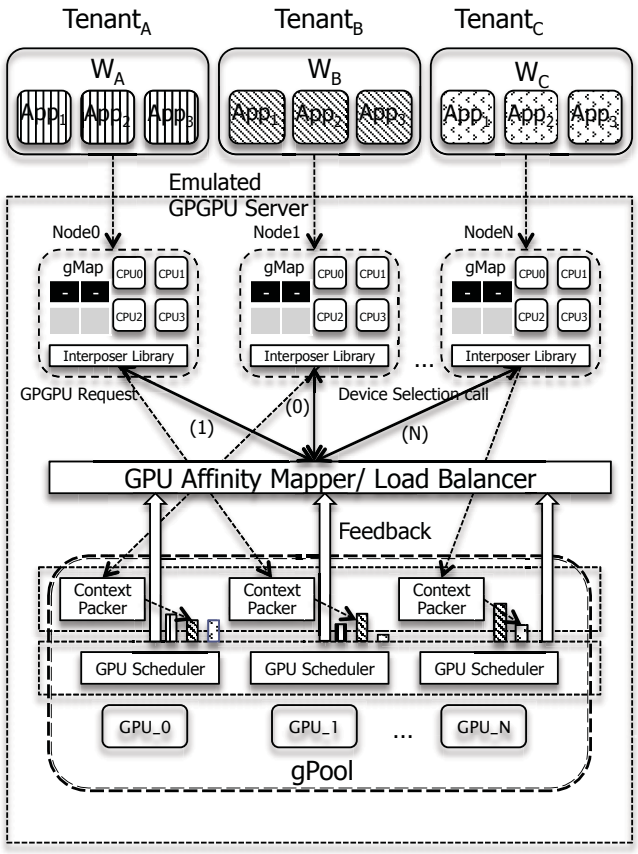

Figure 2: Strings Architecture

implemented by coordination methods that provide serviced virtual machines with shares of physical processors based on one of the diverse set of scheduling policies provided by the Pegasus system.

Schedulers in the Pegasus system use coordination to provide fairness in resource sharing, such as augmented credit-based and feedback-based proportional fair-share schemes. The performance benefits from the Pegasus system are impressive: with minimal virtualization and scheduling overheads, it achieves from 18-140% performance improvements over base GPU driver scheduling while respecting fairness, when the GPUs are shared.

2.2 Multi-Tenant Scheduling for GPU Cloud Workloads

Cloud and service infrastructures, such as Amazon ECC [1], Nimbix [2], and Peerl Hosting [3], use GPUs routinely to service computationally intensive client workloads for a variety of application domains, ranging from online gaming and multimedia services, to data-mining and search. However, the current model of static GPU provisioning in cloud infrastructures, where applications explicitly select GPU devices to run on, limits the efficiency of GPUs in such environments. In particular, static GPU assignments inhibit concurrency for varying workloads: certain services heavily utilize their GPUs during peak demands while other services' GPUs are idle or underutilized. The Strings [33] scheduler improves system throughput and fairness for such GPU-based cloud applications by adopting a multi-tenant model in which GPUs are treated as first-class schedulable entities. This is achieved by interposing device selection calls made by applications and transferring control to a two-level Strings scheduler. At the higher level, the Strings scheduler balances workloads across multiple GPUs on each platform. At the second-level, the scheduler reduces core idling via multi-tenancy, using CUDA streams support to pack different application contexts into a single protection domain and thereby enabling cross-application space-shared use of GPU resources, as well as judicious overlap of GPU execution with host-GPU data movements.

The Strings architecture is presented in Figure 2. The interposed device selection calls are forwarded to the top-level scheduler, GPU Affinity Mapper, which in turn makes a GPU selection for the application based on a combination of static (device capabilities) and dynamic (GPU load, application type, feedback from lower scheduling layer) information. The GPU Affinity Mapper is also responsible for the cluster-wide aggregation of GPUs. After workload balancing, the Context Packer packs multiple applications' GPU components that share a GPU, on the fly, into a single GPU context. Finally, the lower-layer scheduler, GPU Scheduler, addresses inter-application interference arising from multi-tenancy. It prioritizes and dispatches GPU requests to physical GPUs in order to meet system-wide policies, such as throughput and fairness. It also monitors applications' device usage and sends feedback to the GPU Affinity Mapper. The Strings GPU Scheduler implements a rich set of scheduling policies, achieving two cloud-centric goals: fairness for multiple tenants, coupled with high overall system throughput. With its scheduling methods, Strings achieves improvements in system throughput and fairness of up to 8.7× and 13%, respectively, compared to the baseline NVIDIA CUDA runtime.

2.3 Resource Management for GPU-based HPC Systems

High-performance computing systems are becoming increasingly heterogeneous, incorporating combinations of many-core processors, non-uniform memory and accelerators like GPUs. Further, variations in heterogeneity among HPC systems are significant, consisting of different configurations of general-purpose and accelerator cores, interconnect networks, and memory systems. Such diversity in heterogeneity makes the management of current and future HPC systems complex, for both application frameworks and systems software. Not only must applications incorporate a variety of distinct programming models, but also address data partitioning and movement challenges based on the physical configuration of the underlying HPC machine. The Slices [25] runtime posits that heterogeneous cluster hardware should be presented to applications as dynamic 'slices' based on the application's needs and runtime characteristics. Slices provides an application with exactly the resources it needs via a portion of the cluster's resources (general- purpose cores, accelerator cores, and memory) – called a 'GPU Assembly' – where slice allocations are made at sub-node granularity instead of static application-node assignments. In this way, Slices can provide distinct resource-to-application mappings, based on the application's runtime properties. For example, Figure 3 illustrates three alternative mappings for an assembly configured with two virtual GPUs and two general-purpose cores. The first mapping uses local GPUs for a

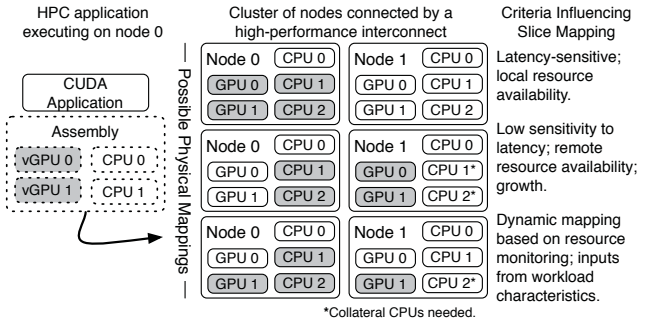

Figure 3: Example assembly mappings to slices provided by the cluster Slices runtime, considering various constraints.

latency-sensitive application; the second uses remote GPUs to accommodate throughput-oriented applications in the situation where local resources are unavailable; and a third mapping uses resources from more than one node to improve overall system utilization.

Cluster hardware slicing is realized with a distributed runtime that includes a distributed collection of stateful, persistent daemon processes with each node hosting one instance. The daemon processes maintain monitoring information and service assembly requests form applications, with a master process designated for responding to such requests. Similar to the Pegasus and Strings model, Slices also interposes application APIs to enable transparent access and use of local or remote assembly GPUs. To achieve intelligent mappings, Slices runtime leverages application profiles, obtained offline, which describe an application's GPU usage patterns and characterizes its sensitivities to changes in GPU locality and host-GPU interactions, including data movements and computation-communication dependencies.

3. RUNTIME SPECIALIZATION

While heterogeneous GPU platforms can improve performance over traditional multi-core CPU platforms for a variety of data-intensive applications, efficiently leveraging the distinct compute capabilities of the heterogeneous resources presents important challenges. First, the distinct execution models inherent in the heterogeneous devices present on such platforms drives the need to dynamically match workload characteristics to the underlying resources. Second, the complex architecture and programming models of heterogeneous GPU systems require substantial application knowledge and effort-intensive program tuning to achieve high performance. These challenges make a case for **runtime specialization**. In this context, runtime specialization is defined as using *online* methods to match *dynamic* workload characteristics to underlying resources, in order to drive greater efficiency.

Professor Schwan's research posits that runtime specialization can achieve high performance and platform throughput without the need for manual intervention for application profiling and/or tuning, for diverse data-intensive applications on heterogeneous CPU-GPU platforms. In this direction, his research contributes a dynamic instrumentation framework that provides real-time insights into application behavior seamlessly, transparently, and efficiently for GPU-based platforms. Online instrumentation and profiling methods are subsequently used to (a) drive better resource management, and (b) perform profile-guided optimizations to improve both platform throughput and individual application performance.

3.1 Dynamic GPU Instrumentation

Lynx [12] is a dynamic instrumentation engine for data-parallel applications on GPU-based architectures. Lynx provides the necessary real-time introspection capabilities into an application's runtime behavior to support *dynamic, online* methods for resource management and optimizations. Specifically, it provides an extensible set of C-based language constructs to build customizable program analysis tools that target the data-parallel programming paradigm used in GPUs. Furthermore, it uses a just-in-time (JIT) compiler to translate, insert and optimize instrumentation code at the intermediate representation (IR) layer. In an nutshell, Lynx provides the capability to write instrumentation routines that are (1) *selective*, instrumenting only what is needed, (2) *transparent*, without changes to the applications' source code, (3) *customizable*, and (4) *efficient*.

Lynx also provides portability by enabling support across several processor back-ends, including various GPU vendors (e.g. NVIDIA, AMD [14], Intel) as well as across discrete and integrated GPU platforms. Lynx's highly modular design makes it amicable to extending support to different intermediate representations (IRs) and GPU runtimes (e.g. OpenCL and CUDA). In its current implementation, Lynx includes runtime support for both OpenCL and CUDA, and instrumentation support for NVIDIA's Parallel Thread eXecution (PTX) [27], AMD's Intermediate Language (IL) [4], and LLVM [20] intermediate representations (IRs).

Figure 4 illustrates Lynx's execution/run-time flow in the context of CUDA applications. CUDA applications compiled by `nvcc` are converted into C++ programs, with PTX kernels embedded as string literals. When such a program links with our framework, the CUDA Runtime API function,
`cudaRegisterFatBinary`, parses these PTX kernels into an internal representation. The original PTX kernel is provided as input to the IR-IR Transformation Pass Manager, together with the instrumentation PTX generated from the C code specification via the COD JIT Compiler and the C-to-PTX Translator. The Pass Manager applies a sequence of PTX kernel transformations to the original PTX kernel. A detailed discussion of the Pass Manager and PTX transformation passes can be found in the following work [11].

A specific pass, C-to-PTX Instrumentation Pass, is implemented as part of the Lynx framework to insert the generated PTX into the original PTX kernel, according to Lynx's language specification. The final output, the instrumented kernel, is prepared for native execution on the selected device by the PTX Translator/Code Generator.

Since GPU Lynx implements the CUDA Runtime API as well, it enables the insertion of hooks into the runtime system for managing resources and data structures needed to support instrumentation. The Lynx framework utilizes this capability via the Instrumentor component. Its general approach for managing instrumentation-related data for discrete GPUs is to allocate memory on the device, populate the instrumentation-related data structures during kernel execution, and then move the data back to the host, freeing up allocated resources on the device. For integrated GPUs, memory is allocated across shared CPU-GPU buffers, eliminating the need for costly transfers for instrumentation-related data structures.

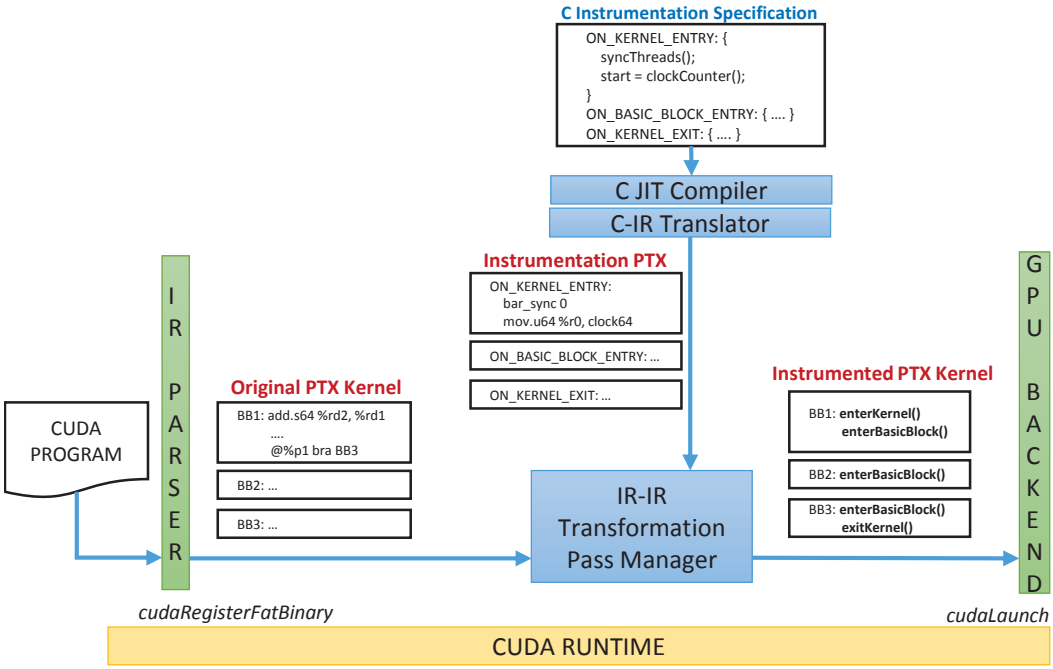

Figure 4: Lynx Dynamic Instrumentation System

3.2 Profile-Driven Dynamic GPU Optimization Framework

Leo [13] is a profile-driven, dynamic optimization framework for GPU applications, which leverages Lynx to drive GPU-specific code optimizations, specifically data layout optimizations. While GPUs enable order-of-magnitude performance increases in many data-parallel application domains, writing efficient codes that can actually manifest those increases is a non-trivial endeavor, typically requiring developers to exercise specialized architectural features exposed directly in the programming model. Achieving good performance on GPUs involves effort-intensive tuning. Leo aims to automate much of this effort using dynamic instrumentation to inform dynamic, profile-driven optimizations.

As a dynamic optimization framework, Leo orchestrates the identification and selection of the optimal code and data layout transformation during the application's execution. It consists of the following two main components: a compilation engine that generates GPU kernel code and data layout on-the-fly from higher-level language source code, and a JIT-based profiling engine that leverages Lynx to enable dynamic instrumentation and profiling of GPU code at runtime.

From an application's perspective, Leo leverages Dandelion [31] to run LINQ applications on GPU. The Dandelion system enables the execution of Language-Integrated Query (LINQ) on GPUs. LINQ introduces a set of declarative operators, which perform transformations on .NET data collections. LINQ applications are computations formed by composing these operators. Most LINQ operators are common relational algebra operators, including projection (Select), filters (Where), grouping (GroupBy), aggregation (Aggregate) and join (Join). The Dandelion compiler automatically compiles a LINQ query into a data-flow graph and any user-defined .NET code into GPU kernels. The Dandelion runtime automatically manages the execution of the data-flow graph on GPUs and the data transfer between CPU and GPU.

The Leo runtime orchestrates the identification and selection of the optimal code transformations and data layouts for GPU kernels. The computation model supported is based on streaming, i.e., the input is divided into chunks and chunks are transferred to GPU concurrently with the GPU execution. This model enables Leo to make optimization decisions based on the execution of preceding chunks. Leo runs the Lynx instrumented code for the first chunk to determine possible candidate kernels for optimization. This allows Leo to generate the optimized version of the code with the necessary code and data layout transformations. The system then runs the second and third chunks with and without the optimizations respectively, and compares the total elapsed running times to determine which version of the code to use for the subsequent chunks. This profiling is repeated at continuous intervals to detect time-varying runtime behaviors and relevant application phase changes.

Figure 5 presents a high-level overview of the design of the Leo framework, depicting the general steps the runtime takes in order to apply profile-driven optimizations to LINQ applications.

3.3 Affinity-Aware Work-Stealing for Integrated CPU-GPU Processors

Recent hardware advances in integrated CPU-GPU processors have made possible more effective, finer-grain models of combined CPU-GPU computation. Specifically, today's integrated processors, such as Intel's Broadwell and Skylake, and AMD's Kaveri and Carrizo systems, offer hardware CPU-GPU shared virtual memory (SVM), memory coherency, and atomic operations. Such hardware support is an effective basis for realizing GPU-capable fine-grain work-stealing schedulers operating across both sets of cores.

While work-stealing has been extensively optimized on multi-core systems [15, 17], little work has been done on

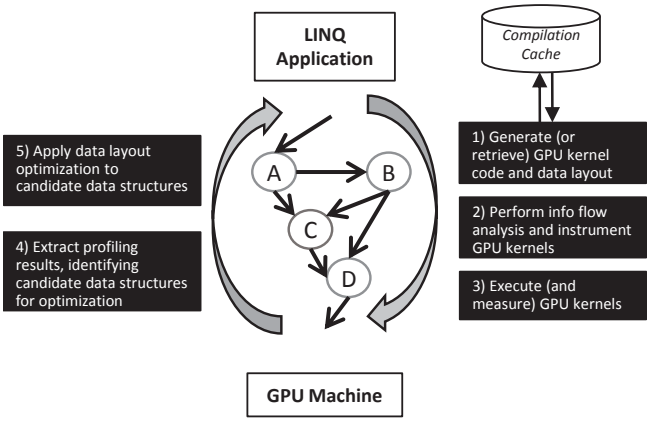

Figure 5: High-Level overview of the Leo framework

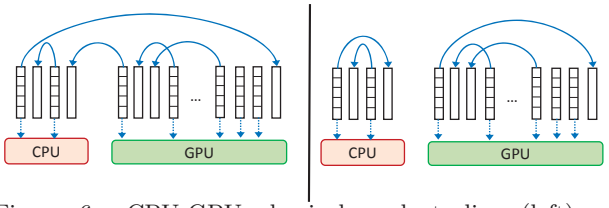

Figure 6: CPU-GPU classical work-stealing (left) and Libra's affinity-aware work-stealing that uses hierarchical stealing(right)

integrated CPU-GPU processors. Efficient work-stealing on integrated CPU-GPU processors is challenging: CPUs and GPUs typically operate at different clock frequencies and have different core configurations and memory hierarchies, making their performance differ by an order of magnitude or more. These differences also result in a huge disparity in their stealing costs, leading to workload imbalance when both devices' worker threads contend to obtain work. As a result, while classical work-stealing enables seamless and dynamic work distribution across compute units in the presence of load imbalance, it does not always work well with such a large gap in performance and stealing costs.

Libra [10] is the first implementation of a fully-integrated, affinity-aware CPU-GPU work-stealing scheduler for integrated processors with hardware SVM support, which advocates runtime specialization methods, such as lightweight online profiling for workload characterization, to improve upon classical work-stealing algorithms. Specifically, Libra's online profiling determines an application's bias to a particular device and optimizes initial work placement. However, on biased workloads, worker threads on the unbiased device may still steal too much work from the other device, due to the significant stealing cost disparity between the two devices, coupled with the application-agnostic nature of stealing. To address this, Libra introduces *hierarchical stealing*: worker threads on each device first steal only from deques on the same device. Only when all deques on its own device are empty is a worker thread allowed to steal from the other device's deques. In other words, hierarchical stealing supports an application's affinity to a particular device. Figure 6 depicts how affinity-aware work-stealing differs from classical work-stealing, when worker threads contend for chunks of work.

4. DATA TRANSFER AND MANAGEMENT

Many cloud and HPC environments employ discrete GPUs due to the unparalleled performance offered by such devices. While vendors like Intel and AMD have made substantial improvements to their integrated GPU counter-parts, discrete GPUs still have a significant edge on performance, not only because they are computationally more powerful but also because they have their own dedicated high-speed memory, unlike integrated GPUs, which share both the system memory and the data-bus with the CPU. In spite of these advantages, discrete GPU environments present data transfer and management challenges, in particular for today's increasingly complex and large-scale applications, which can benefit from a mix of CPU and GPU computation. Since data must be resident in GPU memory for computation, additional data transfers are needed between the CPU and GPU to leverage the GPU's computational power. Discrete GPUs also have limited memory capacities, as compared to the system host memory, which inhibits large-scale data processing. Finally, providing useful systems abstractions, such as checkpointing in HPC-based environments, are inhibited due to bandwidth concerns arising from large data transfers.

Professor Schwan's research advocates novel data transfer and management techniques to address these concerns. These techniques include overlapping computation with communication using both asynchronous data transfer support and advanced hardware features in today's NVIDIA GPUs, as well as mechanisms to identify and eliminate redundant data copying altogether.

4.1 Large-Scale Graph Processing on Discrete GPU Systems

Due to the massive parallelism offered by GPUs, they are being used heavily for real-world graph analytics. However, efficiently processing large-scale graphs on discrete GPU systems is challenging due to the inherent irregularity of graph algorithms and limitations in discrete GPU-resident memory for storing large graphs. Previous work on graph processing has sought out scale-out approaches, by distributing graph data across different computational nodes. The GraphReduce [34] framework recognizes the low computation to communication ratios of typical graph algorithms, and instead advocates a 'scale up' approach in which large graphs processed by memory-limited, discrete GPUs can take advantage of potentially significantly larger memory capacities of their host machines. This is achieved by novel data management and transfer techniques. Large-scale graphs are run efficiently by partitioning graphs into fixed-sized chunks, called shards, which are moved asynchronously between the GPU and the host. Additionally, GraphReduce overlaps GPU computation with data transfer via NVIDIA CUDA streams support, and uses 'spray' operations to divide shards to obtain fine-grain parallelism that exploits the Hyper-Q feature of Kepler GPUs. Spray operations are used to further divide each shard into multiple sub-buffers, which can be transferred over dynamically created CUDA streams.

The architecture of the GraphReduce framework, presented in Figure 7, has three main components: Partition Engine, Data Movement Engine, and Compute Engine. The Partition Engine is responsible for load-balanced shard creation, and providing graph partitioning logics and associated orderings of vertices/edges. The Data Movement Engine accelerates data movement via asynchronous memory-copy op-

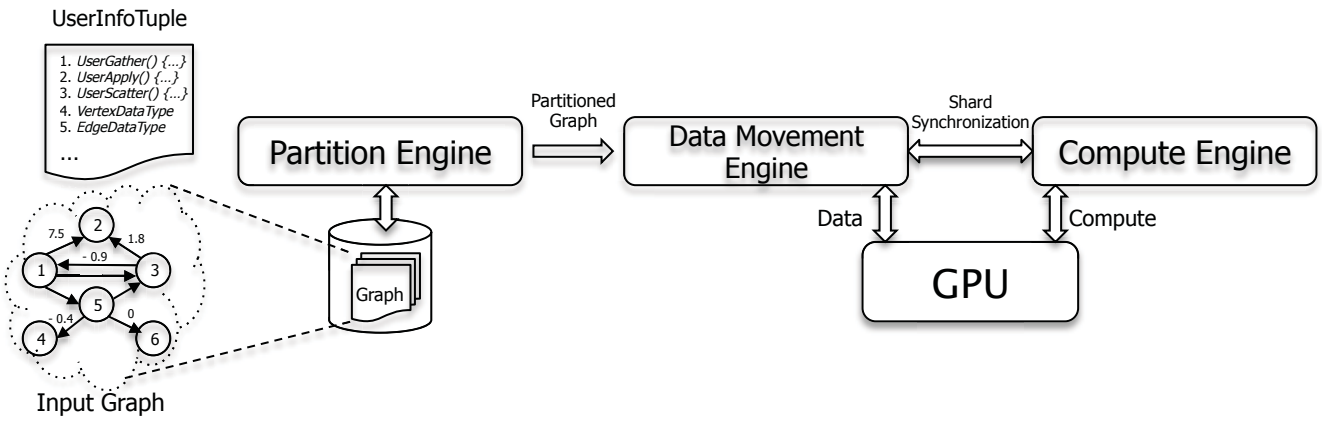

Figure 7: GraphReduce Architecture

erations for concurrent GPU kernel execution, and the Compute Engine is responsible for in-memory computation, which includes parallelizing computation with data movement, as well as sending feedback to the Data Movement Engine about the computation frontier used in subsequent iterations. Experimental evaluation demonstrate significant speedups, up to 79× and 21×, and an average of 13.4× and 5× over competing CPU-based methods implemented in GraphChi [19] and X-Stream [32], respectively. GraphReduce performance is also comparable to existing in-memory GPU frameworks, like MapGraph and CuSha, for smaller input graphs (i.e. those that fit in GPU-resident memory).

4.2 Efficient Checkpointing for GPU-based HPC Systems

As noted earlier, current HPC systems increasingly employ discrete GPUs. While it is well-known that GPUs provide substantial speedups for HPC applications, their failure rates are also at least 10× higher than CPUs on HPC machines. This makes it increasingly important for GPUs to have robust restart and checkpointing mechanisms. However, a key issue with providing checkpointing mechanisms for discrete GPUs is the lack of direct I/O access and bandwidth PCIe bandwidth limitations. Without direct I/O access, GPU-resident data has to first be moved to CPU-accessible DRAM and finally to a nonvolatile (NV) storage. With increasing device memory capacities and application footprints, data movement overheads can be substantial, with potential bottlenecks arising at both the GDRAM-DRAM-PCI interface and the interface to NV storage. Both of these scenarios are depicted in Figure 8.

To address limited data transfer bandwidth concerns, HeteroCheckpoint [18] provides efficient checkpointing methods to improve fault tolerance for GPU-based HPC machines. Specifically, HeteroCheckpoint contributes chunk-level data pre-copy and techniques to eliminate redundant data using data chunk prediction and checksums. With data pre-copy, chunks that are not always modified across kernels are identified and are pre-copied before a synchronous checkpoint is started, in parallel with computation. This reduces the total data movement needed at the time of a checkpoint, reducing the impact on bandwidth limits. Employing checksums to identify and avoid read-only chunks further avoids unnecessary copying. However, incremental data modifications are not easy to capture on GPUs due to absence of

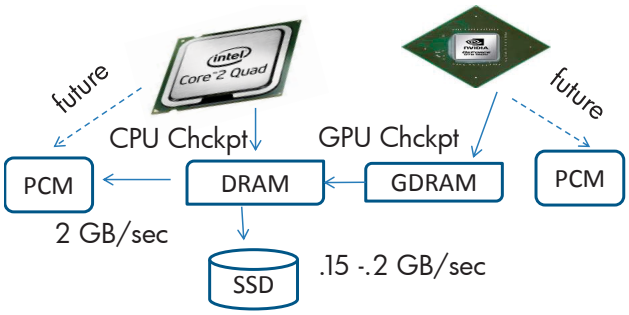

Figure 8: GPU-based checkpoint design

virtual memory page protection or page level dirty tracking techniques. HeteroCheckpoint, therefore, delegates checksum calculation to the CPU in its current implementation, but advocates a dynamic instrumentation-driven approach to maintain information about dirty variables and eliminate the need for checksums altogether.

5. FUTURE DIRECTIONS

An important future direction in Professor Karsten Schwan's research is in leveraging the power of runtime specialization to provide finer-grained sharing and resource management for GPU-based environments. For example, the Lynx dynamic instrumentation framework provides significant opportunities for incorporating a variety of GPU systems abstractions seamlessly and transparently. While GPUs have become primary processing engines in server and cloud computing environments, cloud providers are still not able to provide fine-grained GPU sharing to end-users, even though such sharing is possible for CPUs. Previous work, including Pegasus, has attempted to address shared GPU compute usage in different ways [16, 23], but has not been as effective in enforcing fine-grained service-level objectives (SLOs) due to limited control over closed-vendor, hardware thread scheduling in today's GPUs. GPU dynamic instrumentation can enable software mechanisms for even finer-grained GPU time-sharing, by incorporating yielding mechanisms at smaller execution granularities (such as thread-block level, versus kernel-level). Similarly, software-based check-pointing mechanisms for GPU-based platforms can also benefit from dynamic instrumentation to provide transparency (i.e. not re-

quiring source code modifications and therefore can work with application binaries) and portability (i.e. can target multiple GPU back-ends). Finally, in applications where precision and correctness is a necessity, dynamic instrumentation can be utilized to provide additional software abstractions for reliable operation [21].

Additionally, energy efficiency is now a top design goal for all GPU-based systems, from fitness trackers and tablets, where it affects battery life, to cloud computing centers, where it directly impacts operational cost, maintainability, and environmental impact. Much of Professor Schwan's current work on GPUs has focused on improving platform throughput and application performance, and therefore a natural extension to his work in this space is to explore energy-aware heuristics.

Finally, while the key ideas discussed in this paper focus on GPU-based systems, they can be applied to parallel multi-core and asymmetric architectures in general. The explicit parallel model of execution advocated by parallel architectures provide complimentary computational characteristics to the implicitly parallel CPU-based execution model. To enable today's increasingly complex applications to run efficiently and leverage the power of all of the computational resources in today's heterogeneous environments, intelligent systems support and runtime specialization are critical components. Future cloud computing infrastructures, which will consist of increasingly heterogeneous resources, will certainly benefit from purposefully including mixes of different platforms specialized for different classes of applications, and providing systems abstractions for managing all such resources intelligently.

6. CONCLUSION

Professor Schwan's research with respect to GPU computing advocates novel resource management, systems and runtime support for both cloud and HPC infrastructures. His research demonstrates that runtime specialization and novel data management schemes can drive greater efficiency for today's increasingly complex and irregular data-intensive applications, while novel scheduling methods can provide important systems guarantees such as fairness and throughput.

It is important to note that Professor Schwan's contributions in this space are only a small constituent of his extremely illustrious research portfolio. Over his career, he has conducted research in real-time and distributed systems, high-performance, parallel and heterogeneous computing, virtualization technologies, enterprise middleware for cloud and data-center systems, and edge cloud/mobile-cloud systems. Professor Schwan, who had joined Georgia Tech's College of Computing in 1988, has left behind more than 70 active Ph.D. students, nine active research projects, 26 software systems, and a legacy of 276 published writings in books, journals, and conference proceedings.

In addition to being an active and leading figure in computing, Professor Karsten Schwan was also an extremely compassionate advisor. He not only ensured that his students achieve success during their time at Georgia Tech, but actively supported them throughout their academic and industry careers. To his students, Professor Schwan was more than an advisor and mentor; he was a family member, who went through the ups and downs of their lives with them. He will be deeply missed by his students, peers, collaborators, and the larger computing community.

Acknowledgments

We would like to thank Professor Sudhakar Yalamanchili, Ada Gavrilovska, Vishakha Gupta, Sudarsun Kannan, Alexander Merritt, and Dipanjan Sengupta for their feedback and assistance with the paper.

7. REFERENCES

[1] Amazon ec2 gpu cluster. http://aws.amazon.com/about-aws/whats-new/2010/11/15/announcing-cluster-gpu-instances-for-amazon-ec2/.

[2] Nimbix. http://www.nimbix.net/cloud-supercomputing/.

[3] Peerl hosting. http://www.peer1hosting.co.uk/hosting/gpu-servers.

[4] AMD. *AMD Intermediate Language (IL)*, 2.4 ed. AMD, October 2011.

[5] Anderson, J. A., Lorenz, C. D., and Travesset, A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys. 227*, 10 (May 2008), 5342–5359.

[6] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 66:1–66:11.

[7] Brown, K., Sujeeth, A., Lee, H., Rompf, T., Chafi, H., and OLUKOTUN, K. A heterogeneous parallel framework for domain-specific languages. In *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2011).

[8] Burtscher, M., Nasre, R., and Pingali, K. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on* (2012), pp. 141–151.

[9] Catanzaro, B., Garland, M., and Keutzer, K. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2011), PPoPP '11, ACM, pp. 47–56.

[10] Farooqui, N., Barik, R., Lewis, B. T., Shpeisman, T., and Schwan, K. Affinity-aware work-stealing for integrated cpu-gpu processors. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2016), PPoPP 2016, ACM.

[11] Farooqui, N., Kerr, A., Diamos, G., Yalamanchili, S., and Schwan, K. A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In *Proceedings of the 4th Workshop on General-Purpose Computation on Graphics Processing Units* (Newport Beach, CA, USA, March 2011), ACM.

[12] Farooqui, N., Kerr, A., Eisenhauer, G., Schwan, K., and Yalamanchili, S. Lynx: A dynamic instrumentation system for data-parallel applications

on gpgpu architectures. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on* (April 2012), pp. 58 –67.

[13] Farooqui, N., Rossbach, C., Yu, Y., and Schwan, K. Leo: A profile-driven, dynamic optimization framework for gpu applications. In *Proceedings of the second USENIX conference on Timely Results in Operating Systems* (2014), TRIOS '14.

[14] Farooqui, N., Schwan, K., and Yalamanchili, S. Efficient instrumentation of gpgpu applications using information flow analysis and symbolic execution. In *Proceedings of Workshop on General Purpose Processing Using GPUs* (New York, NY, USA, 2014), GPGPU-7, ACM, pp. 19:19–19:27.

[15] Guo, Y., Zhao, J., Cave, V., and Sarkar, V. Slaw: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 341–342.

[16] Gupta, V., Schwan, K., Tolia, N., Talwar, V., and Ranganathan, P. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 3–3.

[17] jai Min, S., Iancu, C., and Yelick, K. Hierarchical work stealing on manycore clusters. In *In Fifth Conference on Partitioned Global Address Space Programming Models* (2011).

[18] Kannan, S., Farooqui, N., Gavrilovska, A., and Schwan, K. Heterocheckpoint: Efficient checkpointing for accelerator-based systems. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014* (2014), pp. 738–743.

[19] Kyrola, A., Blelloch, G., and Guestrin, C. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 31–46.

[20] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[21] Li, S., Farooqui, N., , and Yalamanchili, S. Software reliability enhancements for gpu applications. In *Sixth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2013)* (Jan 2013).

[22] Linderman, M. D., Collins, J. D., Wang, H., and Meng, T. H. Merge: A programming model for heterogeneous multi-core systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 287–296.

[23] Menychtas, K., Shen, K., and Scott, M. L. Disengaged scheduling for fair, protected access to fast computational accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 301–316.

[24] Merrill, D., Garland, M., and Grimshaw, A. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 117–128.

[25] Merritt, A., Farooqui, N., Slawinska, M., Gavrilovska, A., Schwan, K., and Gupta, V. Slices: Provisioning heterogeneous hpc systems. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment* (New York, NY, USA, 2014), XSEDE '14, ACM, pp. 46:1–46:8.

[26] Mosegaard, J., and Sørensen, T. Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu. In *Proceedings of Eurographics Workshop on Virtual Environments* (2005), vol. 11, pp. 105–111.

[27] NVIDIA. *NVIDIA Compute PTX: Parallel Thread Execution*, 1.3 ed. NVIDIA Corporation, Santa Clara, California, October 2008.

[28] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*, 2.1 ed. NVIDIA Corporation, Santa Clara, California, October 2008.

[29] Podlozhnyuk, V. Black-scholes option pricing. *Part of CUDA SDK documentation* (2007).

[30] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, WA, June 2013).

[31] Rossbach, C. J., Yu, Y., Currey, J., Martin, J.-P., and Fetterly, D. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (NY, USA, 2013), SOSP '13, ACM, pp. 49–68.

[32] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[33] Sengupta, D., Goswami, A., Schwan, K., and Pallavi, K. Scheduling multi-tenant cloud workloads on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Piscataway, NJ, USA, 2014), SC '14, IEEE Press, pp. 513–524.

[34] Sengupta, D., Song, S. L., Agarwal, K., and Schwan, K. Graphreduce: Processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 28:1–28:12.

[35] Silberstein, M., Ford, B., Keidar, I., and Witchel, E. Gpufs: integrating file systems with gpus. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13, ACM.

[36] Stone, J. E., Gohara, D., and Shi, G. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test 12*, 3 (May 2010), 66–73.

[37] Sujeeth, A. K., Lee, H., Brown, K. J., Rompf, T., Chafi, H., Wu, M., Atreya, A. R., Odersky, M., and Olukotun, K. Optiml: An implicitly parallel domain-specific language for machine learning. In *ICML* (2011), L. Getoor and T. Scheffer, Eds., Omnipress, pp. 609–616.

[38] Wu, H., Diamos, G., Cadambi, S., and Yalamanchili, S. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), MICRO-45 âĂŹ12.