2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Generic System Calls for GPUs

Ján Veselý
Rutgers University
jan.vesely@cs.rutgers.edu

Arkaprava Basu*
Indian Institute of Science
arkapravab@iisc.ac.in

Abhishek Bhattacharjee
Rutgers University
abhib@cs.rutgers.edu

Gabriel H. Loh
Advanced Micro Devices, Inc.
gabriel.loh@amd.com

Mark Oskin
University of Washington, Advanced Micro Devices, Inc.
mark.oskin@amd.com

Steven K. Reinhardt*
Microsoft
stever@microsoft.com

***Abstract*—GPUs are becoming first-class compute citizens and increasingly support programmability-enhancing features such as shared virtual memory and hardware cache coherence. This enables them to run a wider variety of programs. However, a key aspect of general-purpose programming where GPUs still have room for improvement is the ability to invoke system calls.**

We explore how to directly invoke system calls from GPUs. We examine how system calls can be integrated with GPGPU programming models, where thousands of threads are organized in a hierarchy of execution groups. To answer questions on GPU system call usage and efficiency, we implement GENESYS, a generic GPU system call interface for Linux. Numerous architectural and OS issues are considered and subtle changes to Linux are necessary, as the existing kernel assumes that only CPUs invoke system calls. We assess the performance of GENESYS using micro-benchmarks and applications that exercise system calls for signals, memory management, filesystems, and networking.

***Keywords*-accelerators and domain-specific architecture, graphics oriented architecture, multicore and parallel architectures, virtualization and OS, GPUs**

I. INTRODUCTION

GPUs have evolved from fixed function 3D accelerators to fully programmable units [1–3] and are now widely used for high-performance computing (HPC), machine learning, and data-analytics. Increasing deployments of general-purpose GPUs (GPGPUs) have been partly enabled by programmability enhancing features like virtual memory [4, 4–9] and cache coherence [10–14].

GPU programming models have evolved with these hardware changes. Early GPGPU programmers [15–19] adapted graphics-oriented programming models such as OpenGL [20] and DirectX [21] for general-purpose usage. Since then, GPGPU programming has become increasingly accessible to traditional CPU programmers, with graphics APIs giving way to computation-oriented languages with a familiar C/C++ heritage such as OpenCL [22], C++AMP [23], and CUDA [24]. However, access to privileged OS services via system calls is an important aspect of CPU programming that remains out of reach for GPU programs.

*Author contributed to this work while working at AMD Research

Designers have begun exploring ways to fill this research void. Studies on filesystem I/O (GPUfs [25]), networking I/O (GPUnet [26]), and GPU-to-CPU callbacks [27] established that direct GPU invocation of some specific system calls can improve GPU performance and programmability. These studies have two themes. First, they focus on specific system calls (i.e., for filesytems and networking). Second, they replace the traditional POSIX-based APIs of these system calls with custom APIs to drive performance. In this study, we ask – can we design an interface for invoking *any* system call from GPUs, and can we do so with standard POSIX semantics to enable seamless and wider adoption? To answer these questions, we design the first framework for **generic system call invocation** on GPUs, or **GENESYS**. GENESYS offers several concrete benefits.

First, GENESYS can support implementation of most of Linux's 300+ system calls. As a proof of concept, we go beyond the specific set of system calls from prior work [25–27] and implement not only filesystem and networking system calls, but also those for asynchronous signals, memory management, resource querying, and device control.

Second, GENESYS's use of POSIX allows programmers to reap the benefits of standard APIs developed over decades of real-world usage. Recent work on SPIN [28] takes a step in this direction by considering how to modify the specific system calls in GPUfs to match traditional POSIX semantics. But GENESYS's generality in supporting all system calls means that it goes further, enabling, among other things, backwards compatibility – GENESYS makes it possible to deploy on GPUs the vast body of legacy software written to invoke OS-managed services.

Third, GENESYS's generality enables GPU acceleration of programs that were previously considered a poor match for GPUs. For example, it allows applications to directly manage their memory, query the system for resource information, employ signals, interface with the terminal, etc., in a manner that lowers programming effort for GPU deployment. These examples underscore GENESYS's ability to support new programming strategies and even legacy applications (e.g., using terminal/signals).

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00075

Finally, GENESYS can leverage the benefits of support for important OS features that prior work cannot. For example, GPUnet's use of custom APIs precludes the use traffic shaping and firewalls that are already built into the OS.

When designing GENESYS, we ran into several design questions. For example, what system calls make sense for GPUs? System calls such as *pread/pwrite* to file(s) or *send/recv* to and from the network stack are useful because they underpin many I/O activities required to complete a task. But system calls such as *fork* and *execv* do not, for now, seem necessary for GPU threads. In the middle are many system calls that need adaptation for GPU execution and are heavily dependent on architectural features that could be supported by future GPUs. For example, *getrusage* can be adapted to return information about GPU resource usage. We summarize the conclusions from this qualitative study.

We then perform a detailed design space study on the performance benefits of GENESYS. Key questions are:

How does the GPU's hardware execution model impact system call invocation strategies? To manage parallelism, the GPU's underlying hardware architecture decomposes work into a hierarchy of execution groups. The granularity of these groups ranges from work-items (or GPU threads) to work-groups (composed of hundreds of work-items) to kernels (composed of hundreds of work-groups)[1]. This naturally presents the following research question – at which of these granularities should GPU system calls be invoked?

How should GPU system calls be ordered? CPU system calls are implemented such that instructions prior to the system call have completed execution, while code following the system call remains unexecuted. This model is a good fit for CPUs, which generally target single-threaded execution. But such "strong ordering" semantics may be overly conservative for GPUs. It acts as implicit synchronization barriers across thousands of work-items, compromising performance. Similar questions arise as to whether GPU system calls should be "blocking" or "non-blocking."

Where and how should GPU system calls be processed? Like all prior work, we assume that system calls invoked by GPU programs need to ultimately be serviced by the CPU. This makes efficient GPU-CPU communication and CPU-side processing fundamental to GPU system calls. We find that careful use of modern GPU features like shared virtual addressing [4] and page fault support [8, 9], coupled with traditional interrupt mechanisms, can enable efficient CPU-GPU communication of system call requests and responses.

To explore these questions, we study GENESYS with microbenchmarks and end-to-end applications. Overall, our contributions are:

①: We take a step toward realizing truly heterogeneous programming by enabling GPUs to directly invoke OS-managed services, just like CPUs. This builds on the promise of recent work [25–28] but goes further by enabling direct invocation of *any* system call through standard POSIX APIs. This permits GPUs to use the entire ecosystem of OS-managed system services developed over decades of research.

② As a proof-of-concept, we use GENESYS to realize system calls previously unavailable on GPUs to directly invoke OS services for memory management, signals, and specialized file-system use. Additionally, we continue supporting all the system services made available by prior work (i.e., GPUs, GPUnet, SPIN), but do so with standard POSIX APIs.

③ We shed light on several novel OS and architectural design issues in supporting GPU system calls. We also offer the first set of design guidelines for practitioners on how to directly invoke system calls in a manner that meshes with the execution hierarchy of GPUs to maximize performance. While we use Linux as a testbed to evaluate our concepts, our design choices are applicable more generally across OSes.

④ We publicly release GENESYS under the Radeon Open Compute stack [29–33], offering its benefits broadly.

II. MOTIVATION

A reasonable question to ponder is, why equip GPUs with system call invocation capabilities at all? Conceptually, OSes have traditionally provided a standardized abstract machine in the form of a process to user programs executing on the CPU. Parts of this process abstraction, such as memory layout, the location of program arguments, and ISA, have benefited GPU programs. Other aspects, however, such as standardized and direct protected access to the filesystem, network, and memory allocation, are extremely important for processes but are yet lacking for GPU code. Allowing GPU code to invoke system calls is a further step to providing a more complete process abstraction to GPU code.

Unfortunately, GPU programs can currently only invoke system calls indirectly, and thereby suffer from performance challenges. Consider the diagram on the left in Figure 1. Programmers are currently forced to delay system call requests until the end of the GPU kernel invocation. This is not ideal because developers have to take what was a single conceptual GPU kernel and partition it into two – one before the system call and one after it. This model, which is akin to continuations, is notoriously difficult to program [34]. Compiler technologies can assist the process [35], but the effect of ending the GPU kernel, and restarting another is the same as a barrier synchronization across all GPU threads and adds unnecessary round-trips between the CPU and the GPU, both of which incur significant overhead.

[1] Without loss of generality, we use AMD's terminology of work-items, work-groups, kernels, and compute unit (CU), although our work applies equally to the NVIDIA threads, thread-blocks, kernels, and streaming multiprocessors (SMs), respectively.

Table I
GENESYS ENABLES NEW CLASSES OF APPLICATIONS AND SUPPORTS ALL PRIOR WORK.

	Type	Application	Syscalls	Description
Previously Unrealizable	Memory Management	miniAMR	madvise, getrusage	Uses madvise to return unused memory to the OS (Sec VIII-A).
	Signals	signal-search	rt_sigqueueinfo	Uses signals to notify the host about partial work completion (Sec VIII-B).
	Filesystem	grep	read, open, close	Work-item invocations not supported by prior work, prints to terminal (Sec VIII-C).
	Device Control (ioctl)	bmp-display	ioctl, mmap	Kernel granularity invocation to query and setup framebuffer properties (Sec VIII-E)
Previously Realizable	Filesystem	wordsearch	pread, read	Supports the same workloads as prior work (GPUfs) (Sec VIII-C).
	Network	memcached	sendto, recfrom	Possible with GPUnet but we do not need RDMA for performance (Sec VIII-D).

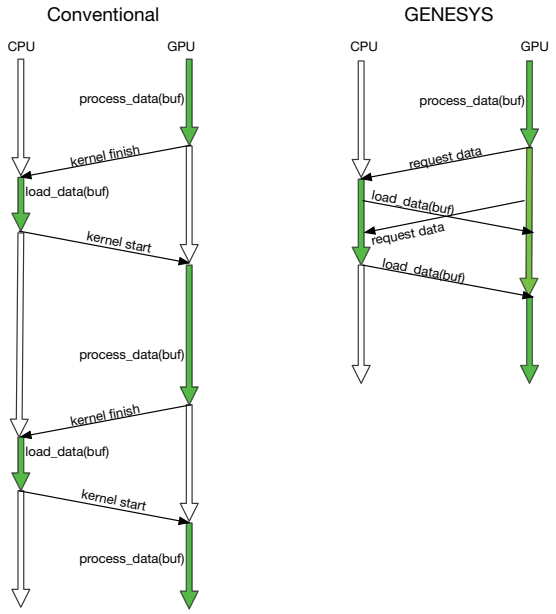

Figure 1. (Left) Timeline of events when the GPU has to rely on a CPU to handle system services; and (right) when the GPU has system call invocation capabilities.

In response, recent studies invoke OS services from GPUs [25–28], as shown on the right in Figure 1. This approach eliminates the need for repeated kernel launches, enabling better GPU efficiency. System calls (e.g., *request_data*) still require CPU processing, as they often require access to hardware resources that only the CPU interacts with. However, CPUs only need to schedule tasks in response to GPU system calls as and when needed. CPU system call processing also overlaps with the execution of other GPU threads. Studies on GPUfs, GPUnet, GPU-to-CPU callbacks, and SPIN are seminal in demonstrating such direct invocation of OS-managed services but suffer from key drawbacks:

Lack of generality: They target specific OS-managed services, and therefore, realize only specific APIs for filesystem or networking services. These interfaces are not readily extensible to other system calls/OS services.

Lack of flexibility: They focus on specific system call invocation strategies. Consequently, there has been no design space exploration on the general GPU system call interface. Questions such as the best invocation granularity (i.e., whether system calls should be invoked per work-item, work-group, or kernel) or ordering remain unexplored, and as we show, can affect performance in subtle and important ways.

Reliance on non-standard APIs: Their use of custom APIs precludes the use of many OS-managed services (e.g., memory management, signals, process/thread management, scheduler). Further, custom APIs do not readily take advantage of existing OS code-paths that enable a richer set of system features. Recent work on SPIN points this out for filesystems, where using custom APIs causes issues with page caches, filesystem consistency, and incompatibility with virtual block devices such as software RAID.

While these past efforts broke new ground and demonstrated the value of OS services for GPU programs, they did not explore the question we pose – why not simply provide generic access from the GPU to all POSIX system calls?

III. HIGH-LEVEL DESIGN

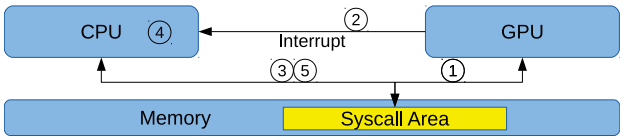

Figure 2. High-level overview of how GPU system calls are invoked and processed on CPUs.

Figure 2 outlines the steps used by GENESYS. When the GPU invokes a system call, it has to rely on the CPU to process system calls on its behalf. Therefore, in step ①, the GPU places system call arguments and information in a portion of system memory also visible to the CPU. We designate a portion of memory as the *syscall area* to store this information. In step ②, the GPU interrupts the CPU, conveying the need for system call processing. Within the

interrupt message, the GPU also sends the ID number of the wavefront issuing the system call. This triggers the execution of an interrupt handler on the CPU. The CPU uses the wavefront ID to read the system call arguments from the *syscall area* in step ③. Subsequently, in step ④, the CPU processes the interrupt and writes the results back into the *syscall area*. Finally, in step ⑤, the CPU notifies the GPU wavefront that its system call has been processed.

We rely on the ability of the GPU to interrupt the CPU and use readily-available hardware [36–38] for this. However, this is not a fundamental design requirement; in fact, prior work [25, 27] uses a CPU polling thread to service a limited set of GPU system service requests instead. Further, while increasingly widespread features such as shared virtual memory and CPU-GPU cache coherence [4, 8, 9, 13] are beneficial to our design, they are not necessary. CPU-GPU communication can also be achieved via atomic reads/writes in system memory or GPU device memory [39].

IV. Analyzing System Calls

While designing GENESYS, we classified all of Linux's over 300 system calls and assessed which ones to support. Some of the classifications were subjective and were debated even among ourselves. Many of the classification issues relate to the unique nature of the GPU's execution model.

Recall that GPUs use SIMD execution on thousands of concurrent threads. To keep such massive parallelism tractable, GPGPU programming languages like OpenCL [22] and CUDA [24] expose hierarchical groups of concurrently executing threads to programmers. The smallest granularity of execution is the GPU work-item (akin to a CPU thread). Several work-items (e.g., 32-64) operate in lockstep in the unit of wavefronts, the smallest hardware-scheduled unit of execution. Many wavefronts (e.g., 16) constitute programmer visible work-groups and execute on a single GPU compute unit (CU). Work-items in a work-group can communicate among themselves using local CU caches and/or scratchpads. Hundreds of work-groups comprise a GPU kernel. The CPU dispatches work to a GPU at the granularity of a kernel. Each work-group in a kernel can execute independently. Further, it is possible to synchronize just the work-items within a single work-group [12, 22]. This avoids the cost of globally synchronizing across thousands of work-items in a kernel, which is often unnecessary in a GPU program and might not be possible under all circumstances[2].

The bottom-line is that GPUs rely on far greater forms of parallelism than CPUs. This implies the following OS/architectural considerations in designing system calls:

Level of OS visibility into the GPU: When a CPU thread invokes the OS, that thread has a *representation* within the kernel. The vast majority of modern OS kernels maintain a data-structure for each thread for several common tasks (e.g., kernel resource use, permission checks, auditing). GPU tasks, however, have traditionally not been represented in the OS kernel. *We believe this should not change.* As discussed above, GPU threads are numerous and short lived. Creating and managing a kernel structure for thousands of individual GPU threads would vastly slow down the system. These structures are also only useful for GPU threads that invoke the OS and represent wasted overhead for the rest. Hence, we process system calls in OS worker threads and switch CPU contexts if necessary (see Section VI). As more GPUs support system calls, this is an area that will require careful consideration by kernel developers.

Evolution of GPU hardware: Many system calls are heavily dependent on architectural features that could be supported by future GPUs. For example, consider that modern GPUs do not expose their thread scheduler to software. This means that system calls to manage thread affinity (e.g., *sched_setaffinity*) are not implementable on GPUs today. However, a wealth of research has shown the benefits of GPU warp scheduling [41–45], so should GPUs require more sophisticated thread scheduling support appropriate for implementation in software, such system calls may ultimately become valuable.

With these design themes in mind, we discuss our classification of Linux's system calls.

① **Readily-implementable:** Examples include *pread, pwrite, mmap, munmap*, etc. This group is also the largest subset, comprising nearly 79% of Linux's system calls. In GENESYS, we implemented 14 such system calls for filesystems (*read, write, pread, pwrite, open, close, lseek*), networking (*sendto, recvfrom*), memory management (*mmap, munmap, madvise*), system calls to query resource usage (*getrusage*), and signal invocation (*rt_sigqueueinfo*). Furthermore, we also implement device control *ioctls*. Some of these system calls, like *read, write, lseek*, are stateful. Thus, GPU programmers must use them carefully; the current value of the file pointer determines what value is read or written by the *read* or *write* system call. This can be arbitrary if invoked at work-item or work-group granularity for the same file descriptor because many work-items/work-groups can execute concurrently.

An important design issue is that GENESYS's support for standard POSIX APIs allows GPUs to *read*, *write*, and *mmap* any file descriptor Linux provides. This is particularly beneficial because of Linux's "everything is a file" philosophy – GENESYS readily supports features like terminal for user I/O, pipes (including redirection of *stdin*, *stdout*, and *stderr*), files in */proc* to query process environments, files in */sys* to query and manipulate kernel parameters, etc. Although our studies focus on Linux, the broader domains of OS-services represented by the specific system calls generalize to other

[2] Although there is no single formally-specified barrier to synchronize across work-groups today, recent work shows how to achieve the same effect by extending existing non-portable GPU inter-work-group barriers to use OpenCL 2.0 atomic operations [40].

Table II
EXAMPLES OF SYSTEM CALLS THAT REQUIRE HARDWARE CHANGES TO BE IMPLEMENTABLE ON GPUS. IN TOTAL, THIS GROUP CONSISTS OF 13% OF ALL LINUX SYSTEM CALLS. IN CONTRAST, WE BELIEVE THAT 79% OF LINUX SYSTEM CALLS ARE READILY-IMPLEMENTABLE.

Type	Examples	Reason that it is not currently implementable
capabilities	capget, capset	Needs GPU thread representation in the kernel
namespace	setns	Needs GPU thread representation in the kernel
policies	set_mempolicy	Needs GPU thread representation in the kernel
thread scheduling	sched_yield, set_cpu_affinity	Needs better control over GPU scheduler
signals	sigaction suspend sigreturn sigprocmask	Signals require the target thread to be paused and then resumed after signal action has been completed. GPU threads cannot be targeted. It is currently not possible to independently set program counters of individual threads. Executing signal actions in newly spawned threads might require freeing of GPU resources.
architecture specific	ioperm	Not accessible from GPU

OSes like FreeBSD and Solaris.

At the application level, implementing this array of system calls opens new domains of OS managed services for GPUs. In Table I, we summarize previously unimplementable system calls realized and studied in this paper. These include applications that use *madvise* for memory management and *rt_sigqueueinfo* for signals. We also go beyond prior work on GPUfs by supporting filesystem services that require more flexible APIs with work-item invocation capabilities for good performance. Finally, we continue to support previously implementable system calls.

② Useful but implementable only with changes to GPU hardware: Several system calls (13% of the total) seem useful for GPU code, but are not easily implementable because of Linux's existing design. Consider *sigsuspend/sigaction* – there is no kernel representation of a GPU work-item to manage and dispatch a signal to. Additionally, there is no lightweight method to alter the GPU program counter of a work-item from the CPU kernel. One approach is for signal masks to be associated with the GPU context and for signals to be delivered as additional work-items. This works around the absence of GPU work-item representation in the kernel. However, POSIX requires threads that process signals to pause execution and resume only after the signal has been processed. Targeting the entire GPU context would mean that all GPU execution needs to halt while the work-item processing the signal executes, which goes against the parallel nature of GPU execution. Recent work has, however, shown the benefits of hardware support for dynamic kernel launch that allows on-demand spawning of kernels on the GPU without any CPU intervention [46]. Should such approaches be extended to support thread recombination assembling multiple signal handlers into a single warp (akin to prior approaches on control flow divergence [42]), *sigsuspend* or *sigaction* may become implementable. Table II presents more examples of currently not implementable system calls (in their original semantics).

③ Requires extensive modification to be supported: This group (8% of the total) contains perhaps the most controversial set of system calls. At this time, we do not believe that it is worth the implementation effort to support these system calls. For example, *fork* necessitates cloning a copy of the executing caller's GPU state. Technically, this can be done (e.g., it is how GPGPU code is context switched with the graphics shaders) but it seems unnecessary at this time.

V. DESIGN SPACE EXPLORATION

A. GPU-Side Design Considerations

Invocation granularity: In assessing how best to use GPU system calls, several questions arise. The first and most important question is – how should system calls be aligned with the hierarchy of GPU execution groups? Should a GPU system call be invoked separately for each work-item, once for every work-group, or once for the entire GPU kernel?

Consider a GPU program that writes sorted integers to a single output file. One might, at first blush, invoke the *write* system call at each work-item. This can present correctness issues, however, because *write* is position-relative and requires access to a global file pointer. Without synchronizing the execution of *write* across work-items, the output file will be written in a non-deterministic unsorted order.

Using different system call invocation granularities can fix this issue. One could, for example, use a memory location to temporarily buffer the sorted output. Once all work-items have updated this buffer, a single *write* system call at the end of the kernel can be used to write the contents of the buffer to the output file. This approach loses some benefits of the GPU's parallel resources, because the entire system call latency is exposed on the program's critical path and might not be overlapped with the execution of other work-items. Alternatively, one could use *pwrite* system calls instead of *write* system calls. Because *pwrite* allows programmers to specify the absolute file position where the output is to be written, per-work-item invocations present no correctness issue. However, per-work-item *pwrites* result in a flood of system calls, potentially harming performance.

Overly coarse kernel-grained invocations also restrict performance by reducing the possibility of overlapping system call processing with GPU execution. A compromise may be to invoke a *pwrite* system call per work-group, buffering

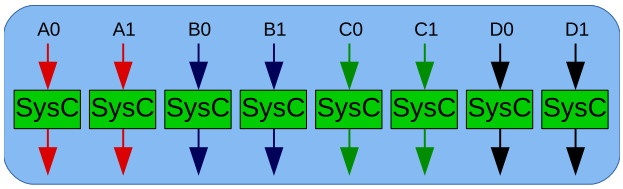

Figure 3. Work-items in a work-group (shown as a blue box) execute strongly ordered system calls.

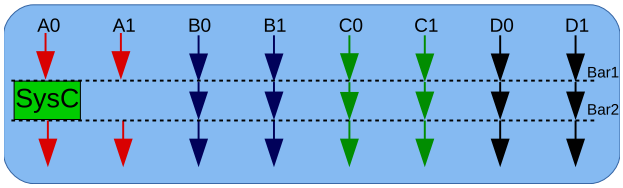

Figure 4. Work-group invocations can be relax-ordered by removing one of the two barriers.

the sorted output of the work-items until the per-work-group buffers are fully populated. Section VII demonstrates that these decisions can lead to a 1.75× performance difference.

System call ordering semantics: When programmers invoke system calls on CPUs, they expect that all program instructions before the system call will complete execution before the system call executes. They also expect that instructions after the system call will only commence once the system call returns. We call this "strong ordering." For GPUs however, we introduce "relaxed ordering" semantics. The notion of relaxed ordering is tied to the hierarchical execution scopes of the GPU and is needed for both correctness and performance.

Figure 3 shows a programmer-visible work-group (in blue), consisting of four wavefronts A, B, C and D. Each wavefront has two work-items (e.g., A0 and A1). If system calls (SysCs) are invoked per work-item, they are strongly ordered. Another approach is depicted in Figure 4, where one work-item, A0, invokes a system call, *on behalf of* the entire work-group. Strong ordering is achieved by placing work-group barriers (Bar1, Bar2) before and after system call execution. One can remove these barriers with relaxed ordering, allowing threads in wavefronts B, C, and D to execute overlapping with the CPU's processing of A's system call.

For correctness, we need to allow programmers to use relaxed ordering when system calls are invoked at kernel granularity (across work-groups). This is because kernels can consist of more work-items than can concurrently execute on the GPU. For strongly ordered system calls at the kernel-level, all kernel work-items must finish executing pre-invocation instructions prior to invoking the system call. But all work-items cannot execute concurrently because GPU runtimes do not preemptively context switch work-items of the same kernel. It is not always possible for all work-items to execute all instructions prior to the system call. Strong ordering at kernel granularity risks deadlocking the GPU.

At the work-group invocation granularity, relaxed ordering can improve performance by avoiding synchronization overheads and overlapping CPU-side system call processing with the execution of other work-items. The key is to remove the barriers Bar1/Bar2 in Figure 4. To do this, consider that from the application point of view, system calls are usually producers or consumers of data. Take a consumer call like *write*, invoked at the work-group level. Real-world GPU programs may use multiple work-items to generate the data for the write, but instructions after the *write* call typically do not depend on the *write* outcome. Therefore, other work-items in the group need not wait for the completion of the system call, meaning that we can remove Bar2, improving performance. Similarly, producer system calls like *read* typically require system calls to return before other work-items can start executing program instructions post-*read*, but do not necessarily require other work-items in the work-group to finish executing all instructions before the *read* invocation. Bar1 in Figure 4 becomes unnecessary in these cases. The same observations apply to per-kernel system call invocations that need relaxed ordering for correctness anyway.

In summary, work-item invocations imply strong ordering. Programmers balance performance/programmability for work-group invocations by choosing strong or relaxed ordering. Finally, programmers must use relaxed ordering for kernel invocations so that the GPU does not deadlock.

Blocking versus non-blocking approaches: Most traditional CPU system calls – barring those for asynchronous I/O (e.g., *aio_read*, *aio_write*) – return only after the system call completes execution. Blocking system calls have a disproportionate impact on performance because GPUs use SIMD execution. In particular, if even a single work-item is blocked on a system call, no other work-item in the wavefront can make progress either. We find that GPUs can often benefit from *non-blocking* system calls that can immediately return before system call processing completes. Non-blocking system calls can therefore overlap system call processing on the CPU with useful GPU work, improving performance by as much as 30% in some of our studies (see Section VII).

The concepts of blocking/non-blocking and strong/relaxed ordering are related but orthogonal. The strictness of ordering refers to the question of when a system call can be invoked with respect to the progress of work-items within its granularity of invocation. System call blocking refers to how the return from a system call relates to the completion of its processing. Relaxed ordering and non-blocking can be combined in several useful ways. For example, consider a case where a GPU program writes to a file at work-group granularity. The execution may not depend upon the output of the write, but the programmer may want to ensure that the write successfully completes. In such a scenario, blocking *writes* may be invoked with weak ordering. Weak ordering permits all but one wavefront in the work-group to proceed without waiting for completion of the *write* (see Section VI).

Blocking invocation, however, ensures that one wavefront waits for the *write* to complete and can raise an alarm if the write fails. Consider another scenario, where a programmer wishes to prefetch data using *read* system calls but may not use the results immediately. Here, weak ordering with non-blocking invocation is likely to provide the best performance without breaking the program's semantics. In short, different combinations of blocking and ordering enable programmers to fine-tune performance and programmability tradeoffs.

B. CPU Hardware

GPUs rely on extreme parallelism for performance. This means there may be bursts of system calls that CPUs need to process. System call coalescing is one way to increase the throughput of system call processing. The idea is to collect several GPU system call requests and batch their processing on the CPU. The benefit is that CPUs can manage multiple system calls as a single unit, scheduling a single software task to process them. This reduction in task management, scheduling, and processing overheads can often boost performance (see Section VII). Coalescing must be performed judiciously as it improves system call processing throughput at the expense of latency. It also implicitly serializes the processing of system calls within a coalesced bundle.

To allow the GPGPU programmer to balance the benefits of coalescing with its potential challenges, GENESYS accepts two parameters – a time window length within which the CPU coalesces system calls, and the maximum number of system call invocations that can be coalesced within the time window. Section VII shows that system call coalescing can improve performance by as much as 10-15%.

C. CPU-GPU Communication Hardware

Prior work implemented system calls using polling, where GPU wavefronts monitored predesignated memory locations populated by CPUs upon system call completion. But recent advances in GPU hardware enable alternate modes of CPU-GPU communication. For example, AMD GPUs now allow wavefronts to relay interrupts to CPUs and then halt execution, relinquishing SIMD hardware resources [36]. CPUs can in turn message the GPU to wake up halted wavefronts.

We have implemented polling and halt-resume approaches in GENESYS. With polling, if the number of memory locations that needs to be polled by the GPU exceeds its cache size, frequent cache misses lower performance. On the other hand, halt-resume has its own overheads, namely the latency to resume a halted wavefront.

We have found that polling yields better performance when system calls are invoked at coarser work-group granularities since fewer memory locations are needed to convey information at the work-group versus work-item level. Consequently, the memory locations easily fit in the GPU's L2 data cache. When system calls are invoked per work-item however, the sheer number of such memory locations becomes so high that cache thrashing becomes an issue. In these situations, halt-resume approaches outperform polling. We quantify the impact of this in the following sections.

Table III
SYSTEM CONFIGURATION USED FOR OUR STUDIES.

SoC	Mobile AMD FX-9800P™
CPU	4× 2.7GHz AMD Family 21h Model 101h
Integrated-GPU	758 MHz AMD GCN 3 GPU
Memory	16 GB Dual-Channel DDR4-1066MHz
Operating system	Fedora 26 using ROCm stack 1.6 (based on Linux 4.11)
Compiler	HCC-0.10.17166 + LLVM 5.0 C++AMP with HC extensions

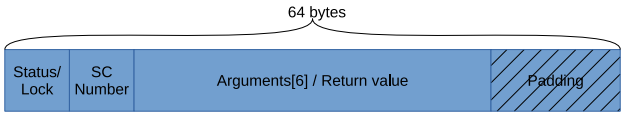

Figure 5. Content of each slot in syscall area.

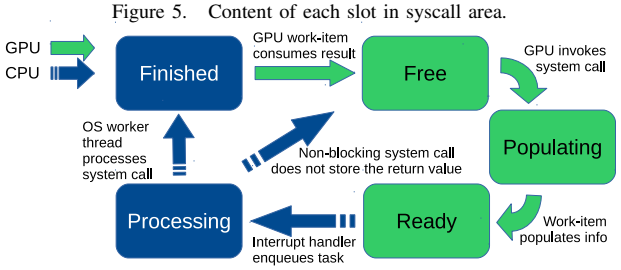

Figure 6. State transition diagram for a slot in syscall area. Green shows GPU state/actions, blue shows that of the CPU.

VI. IMPLEMENTING GENESYS

We implemented GENESYS on the system in Table III. We used an AMD FX-9800P processor with an integrated GPU and ran the open-source Radeon Open Compute (ROCm) software stack [47]. Although we use a system with integrated GPU, GENESYS is not specific to integrated GPUs, and generalizes to discrete GPUs. We modified the GPU driver and Linux kernel to enable GPU system calls. We also modified the HCC compiler to permit GPU system call invocations in C++AMP.

Invoking GPU system calls: GENESYS permits work-item, work-group, and kernel-level invocation. At work-group or kernel-level invocations, a single work-item is designated as the caller. For strongly ordered work-group invocations, we use work-group scope barriers before and after system call invocations. For relaxed ordering, a barrier is placed either before (for consumer system calls) or after (for producer calls) system call invocation.

GPU to CPU communication: GENESYS facilitates efficient GPU to CPU communication of system call requests.

GENESYS uses a preallocated shared memory *syscall area* to allow the GPU to convey parameters to the CPU (see Section III). The *syscall area* maintains one *slot* for each active GPU work-item. The OS and driver code can identify the desired slot by using the hardware ID of the active work-item, which is available to GPU runtimes. This hardware ID is distinct from the programmer-visible work-item ID. Each of the work-items has a unique work-item ID visible to the application programmer. At any one point in time, however, only a subset of these work-items executes (as permitted by the GPU's CU count, supported work-groups per CU, and SIMD width). The hardware ID distinguishes among these active work-items. Overall, our system uses 64 bytes per slot, totaling 1.25 MBs of *syscall area*.

Figure 5 shows the contents in a slot – the requested system call number, the request state, system call arguments (as many as 6, in Linux), and padding to avoid false sharing. The field for arguments is also re-purposed for the return value of the system call. When a GPU program's work-item invokes a system call, it atomically updates the state of the corresponding slot from *free* to *populating* (Figure 6). If the slot is not *free*, system call invocation is delayed. Once the state is *populating*, the invoking work-item populates the slot with system call information and changes the state to *ready*. The work-item also adds one bit of information about whether the invocation is blocking or non-blocking. The work-item then interrupts the CPU using a *scalar wavefront GPU instruction* [3] (*s_sendmsg* on AMD GPUs). For blocking invocation, the work-item either waits and polls the state of the slot or suspends itself using halt-resume.

CPU-side system call processing: Once the GPU interrupts the CPU, system call processing commences. The interrupt handler creates a new kernel task and adds it to Linux's work-queue. This task is also passed the hardware ID of the requesting wavefront. At an expedient future point in time an OS worker thread executes this task. The task scans the 64 *syscall slots* of the given hardware ID and atomically switches any *ready* system call requests to the *processing* state. The task then carries out the system call work.

A challenge is that Linux's traditional system call routines implicitly assume that they are to be executed within the context of the original process invoking the system call. Consequently, they use context specific variables (e.g., the *current* variable used to identify the process context). This presents a challenge for GPU system calls, which are instead serviced purely in the context of the OS' worker thread. GENESYS overcomes this challenge in two ways – it either switches to the context of the original CPU program that invoked the GPU kernel, or it provides additional context information in the code performing system call processing. The exact strategy is determined on a case-by-case basis.

GENESYS implements coalescing by waiting for a predetermined amount of time in the interrupt handler before enqueueing a task to process a system call. If multiple requests are made to the CPU during this time window, they are coalesced with such that they can be handled as a single unit of work by the OS worker-thread. GENESYS uses Linux's *sysfs* interface to communicate coalescing parameters.

Communicating results from the CPU to the GPU: Once the CPU worker-thread finishes processing the system call, the results are put in the field for arguments in the slot for blocking requests. Further, the thread also changes the state of the slot to *finished* for blocking system calls. For non-blocking invocations, the state is changed to *free*. The invoking GPU work-item is then re-scheduled (on hardware that supports wavefront suspension) or automatically restarted because it was polling on this state. The work-item can consume the result and continue execution.

Other architectural design considerations: GENESYS requires two key data structures to be exchanged between GPUs and CPUs – the *syscall area* and for some system calls, *syscall buffers* that maintain data required for system call processing. We discovered that carefully leveraging architectural support for CPU-GPU cache coherence in the context of these data structures was vital to overall performance.

Consider, for example, the *syscall area*. As described in Section III, every GPU work-item invoking a system call is allocated space in the *syscall area*. Like many GPUs, the one used as our experimental platform supports L2 data caches that are coherent with CPU caches/memory but integrates non-coherent L1 data caches. At first blush, one may decide to manually invalidate L1 data cache lines. However, we sidestepped this issue by restricting per-work-item slots in the *syscall area* to individual cache lines. This design permits us to use atomic instructions to access memory – these atomic instructions, by design, force lookups of the L2 data cache and guarantee visibility of the entire cacheline, sidestepping the coherence challenges of L1 GPU data caches. We quantify the measured overheads of the atomic operations we use for GENESYS in Table IV, comparing them to the latency of a standard load operation. Through experimentation, we achieved good performance using *cmp-swap* atomics to claim a slot in the *syscall area* when GPU work-items invoked system calls, *atomic-swaps* to change state, and *atomic-loads* to poll for completion.

Unfortunately, the same approach of using atomics does not yield good performance for accesses to *syscall buffers*. The key issue is that *syscall buffers* can be large and span multiple cache lines. Using atomics here meant that we suffered the latency of several L2 data cache accesses to *syscall buffers*. We found that a better approach was to eschew atomics in favor of manual software L1 data cache coherence. This

[3] Scalar wavefront instructions are part of the Graphics Core Next ISA and are executed once for the entire wavefront, rather than for each active work-item. See Chapter 4.1 in the manual [36].

Table IV
PROFILED PERFORMANCE OF GPU ATOMIC OPERATIONS.

Op	cmp-swap	swap	atomic-load	load
Time(us)	1.352	0.782	0.360	0.243

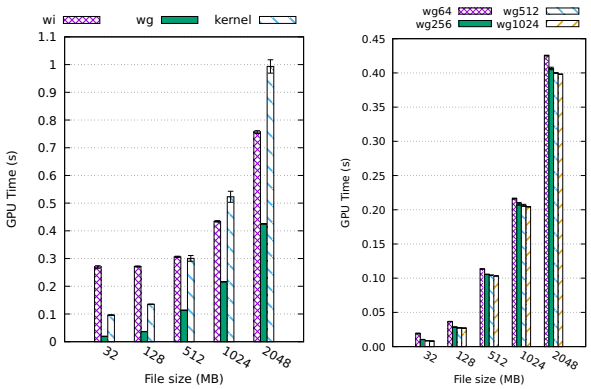

Figure 7. Impact of system call invocation granularity. The graphs show average and standard deviation of 10 runs.

meant, for example, that we preceded *sys_write* system calls with L1 data cache flush.

VII. MICROBENCHMARK EVALUATIONS

Invocation granularity: Figure 7 quantifies performance for a microbenchmark that uses *pread* on files in *tmpfs*[4]. The x-axis plots the file size, and y-axis shows read time, with lower values being better. Within each cluster, we separate runtimes for different *pread* invocation granularities.

Figure 7(left) shows that work-item invocation granularities tend to perform worst. This is not surprising as it is the finest granularity of system call invocation and leads to a flood of individual system calls that overwhelm the CPU. On the other end of the granularity spectrum, kernel-level invocation is also challenging as it generates a single system call and fails to leverage any potential parallelism in processing of system call requests. This is particularly pronounced at large file sizes (e.g., 2GB). A good compromise is to use work-group invocation granularities. It does not overwhelm the CPU with system call requests while still being able to exploit parallelism in servicing system calls.

When using work-group invocation, an important question is how many work-items should constitute a work-group. While Figure 7(left) uses 64 work-items in a work-group, Figure 7(right) quantifies the performance impact of *pread* system calls as we vary work-group sizes from 64 (wg64) to 1024 (wg1024) work-items. In general, larger work-group sizes enable better performance, as there are fewer unique system calls necessary to handle the same amount of work.

Blocking and ordering strategies: To quantify the impact of blocking/non-blocking with strong/relaxed ordering, we

[4] *Tmpfs* is a filesystem without permanent backing storage. In other words, all structures and data are memory-resident.

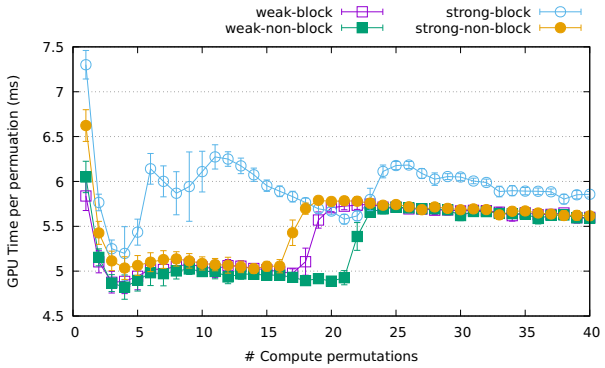

Figure 8. Performance implications of system call blocking and ordering semantics. The graph shows average and standard deviation of 80 runs.

designed a GPU microbenchmark that performs block permutation on an array, similar to the permutation steps performed in DES encryption. The input data array is preloaded with random values and divided into 8KB blocks. Work-groups each execute 1024 work-items independently permute blocks. The results are written to a file using *pwrite* at work-group granularity. The *pwrite* system calls for one block of data are overlapped with permutations on other blocks of data. To vary the amount of computation per system call, we permute multiple times before writing the result.

Figure 8 quantifies the impact of using blocking versus non-blocking system calls with strong and weak ordering. The x-axis plots the number of permutation iterations performed on each block by each work-group before writing the results. The y-axis plots execution time for one permutation (lower is better). Figure 8 shows that strongly ordered blocking invocations (*strong-block*) hurt performance. This is expected as they require work-group scoped barriers to be placed before and after *pwrite* invocations. The GPU's hardware resources for work-groups are stalled waiting for the *pwrite* to complete. Not only does the inability to overlap GPU parallelism hurt strongly ordered blocking performance, it also means that GPU performance is heavily influenced by CPU-side performance vagaries like synchronization overheads, servicing other processes, etc. This is particularly true at iteration counts where system call overheads dominate GPU-side computation – below 15 compute iterations. Even when the application becomes GPU-compute bound, performance remains non-ideal.

Figure 8 shows that when *pwrite* is invoked in a non-blocking manner (with strong ordering), performance improves. This is because non-blocking *pwrites* permit the GPU to end the invoking work-group, freeing GPU resources it was occupying. CPU-side *pwrite* processing can overlap with the execution of another work-group permuting on a different block of data. Figure 8 shows that latencies generally drop by 30% compared to blocking calls at low iteration counts. At higher iteration counts (beyond 16), these benefits diminish because the latency to perform repeated

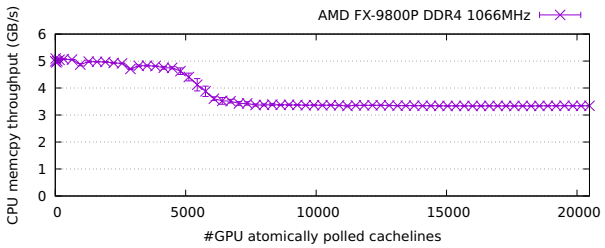

Figure 9. Impact of polling on memory contention.

permutations dominates any system call processing times.

For relaxed ordering with blocking (*weak-block*), the post-system-call work-group-scope barrier is eliminated. One out of every 16 wavefronts[5] in the work-group waits for the blocking system call, while others exit, freeing GPU resources. The GPU can use these freed resources to run other work-items to hide CPU-side system call latency. Consequently, the performance trends follow those of *strong-non-block*, with minor differences in performance arising from differences GPU scheduling of the work-groups for execution. Finally, Figure 8 shows system call latency is best hidden using relaxed and non-blocking approaches (*weak-non-block*).

Polling/halt-resume and memory contention: As previously discussed, polling at the work-item invocation granularity leads to memory reads of several thousands of per-work-item memory locations. We quantify the resulting memory contention in Figure 9, which showcases how the throughput of CPU accesses decreases as the number of polled GPU cache lines increases. Once the number of polled memory locations outstrips the GPU's L2 cache capacity (roughly 4096 cache lines in our platform), the GPU polls locations spilled to the DRAM. This contention on the memory controllers shared between GPUs and CPUs. We advocate using GENESYS with halt-resume approaches at such granularities of system call invocation.

Interrupt coalescing: Figure 10 shows the performance impact of coalescing system calls. We use a microbenchmark that invokes *pread*. We read data from files of different sizes using a constant number of work-items. More data is read per *pread* system call from the larger files. The x-axis shows the amounts of data read and quantifies the latency per requested byte. We present two bars for each point on the x-axis, illustrating the average time needed to read one byte with the system call in the absence of coalescing and when up to eight system calls are coalesced. Coalescing is most beneficial when small amounts of data are read. Reading more data takes longer; the overhead reduction is negligible compared to the significantly longer time to process the system call.

[5]Each wavefront has 64 work-items. Thus, a 1024 work-item work-group has 16 wavefronts.

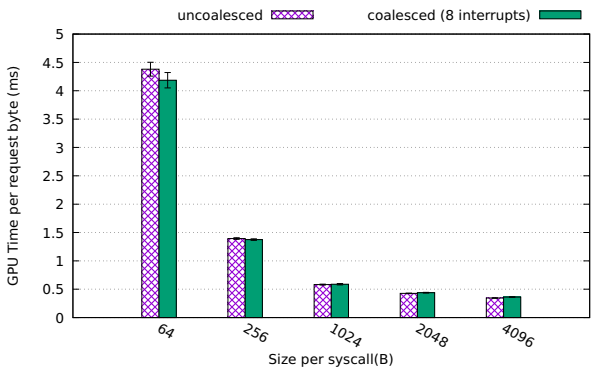

Figure 10. Implications of system call coalescing. The graph shows average and standard deviation of 20 runs.

VIII. Case Studies

A. Memory Workload

We now assess the end-to-end performance of workloads that use GENESYS. Our first application requires memory management system calls. We studied miniAMR [48] and used the *madvise* system call directly from the GPU to better manage memory. MiniAMR performs 3D stencil calculations using adaptive mesh refinement and is a candidate for memory management because it varies its memory needs in a data-model-dependent manner. For example, when simulating regions experiencing turbulence, miniAMR needs to model with higher resolution. However, if lower resolution is possible without compromising simulation accuracy, miniAMR reduces memory and computation usage, making it possible to free memory. While relinquishing excess memory in this way is not possible in traditional GPU programs without explicitly dividing the offload into multiple kernels interspersed with CPU code (see Figure 1), GENESYS permits this with direct GPU *madvise* invocations. We invoke *madvise* using work-group granularities with non-blocking and weak ordering. We leverage GENESYS' generality by also using *getrusage* to read the application resident set size (RSS). When the RSS exceeds a watermark, *madvise* relinquishes memory.

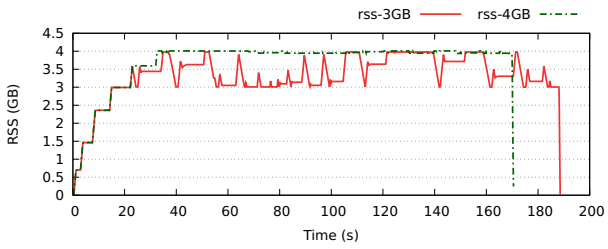

Figure 11. Memory footprint of miniAMR using *getrusage* and *madvise* to hint at unused memory.

We execute miniAMR with a dataset of 4.1GB – just beyond the hard limit we put on physical memory available to GPU. Without using *madvise*, memory swapping increases

so dramatically that it triggers GPU timeouts, causing the existing GPU device driver to terminate the application. Because of this, there is no baseline to compare to as the baseline simply does not complete.

Figure 11 shows two results: one for a 3GB RSS watermark, and one for 4GB. Not only does GENESYS enable miniAMR to complete, it also permits the programmer to balance memory usage and performance. While *rss-3GB* lowers memory utilization, it also worsens runtime compared to *rss-4GB*. This performance gap is expected; the more memory is released, the greater the likelihood that the GPU program suffers from page faults when the memory is touched again in the future, and the more frequent the *madvise* system call is invoked. Overall, Figure 11 illustrates that GENESYS allows programmers to perform memory allocation to trade memory usage and performance on GPUs analogous to CPUs.

B. Workload Using Signals

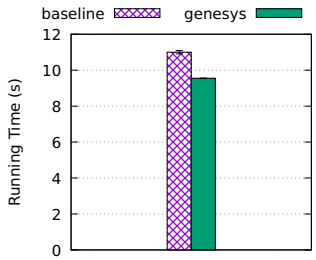

Figure 12. Runtime of CPU- GPU map reduce workload.

GENESYS also enables system calls that permit the GPU to send asynchronous notifications to the CPU. This is useful in many scenarios. We study one such scenario and implement a map-reduce application called signal-search. The application runs in two phases. The first phase performs parallel lookup in a data array, while the second phase processes blocks of retrieved data and computes sha512 checksums on them. The first phase is a good fit for GPUs since the highly parallel lookup fits its execution model, while the second phase is more appropriate for CPUs, many of which support performance acceleration of sha checksums via dedicated instructions.

Without support for signal invocation on GPUs, programmers would launch a kernel with the entire parallel lookup on the GPU and wait for it to conclude before proceeding with the sha512 checksums. GENESYS overcomes this restriction and permits a heterogeneous version of this code, where GPU work-groups can emit signals using *rt_sigqueueinfo* to the CPU, indicating per-block completions of the parallel search. As a result, the CPU can begin processing these blocks immediately, permitting overlap with GPU execution.

Operationally, *rt_sigqueueinfo* is a generic signal system call that allows the caller to fill a *siginfo* structure that is passed along with the signal. In our workload, we find that work-group level invocations perform best, so the GPU passes the identifier of this work-group through the *siginfo* structure. Figure 12 shows that using work-group invocation granularity and non-blocking invocation results in roughly 14% performance speedup over the baseline.

C. Storage Workloads

We have also studied how to use GENESYS to support storage workloads. In some cases, GENESYS permits the implementation of workloads supported by GPUfs, but in more efficient ways.

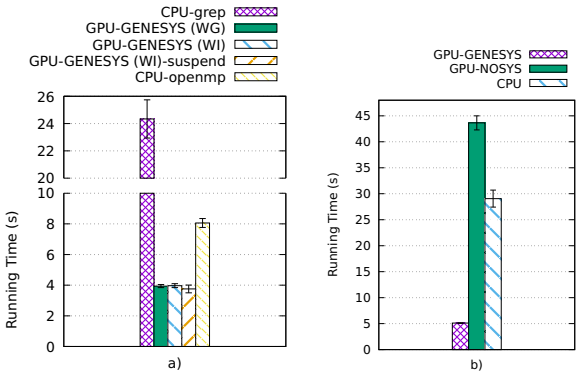

Figure 13. a) Standard *grep*, OpenMP *grep*, versus work-item/work-group invocations with GENESYS. b) Comparing the performance of CPU, GPU with no system call, and GENESYS implementations of wordcount. Graphs show average and standard deviation of 10 runs.

Supporting storage workloads more efficiently than GPUfs: We implement a workload that takes as input a list of words and a list of files. It then performs *grep -F -l* on the GPU, identifying which files contain any of the words on the list. As soon as these files are found, they are printed to the console. This workload cannot be supported by GPUfs without significant code refactoring because of its use of custom APIs. Instead, since GENESYS naturally adheres to the UNIX "everything is a file" philosophy, porting *grep* to GPUs requires only a few hours of programming.

Figure 13 shows the results of our GPU *grep* experiments. We compare a standard CPU implementation, a parallelized OpenMP CPU implementation, and two GPU implementations with GENESYS, with non-blocking system calls invoked at work-item (WI) and work-group (WG) granularities. Furthermore, since work-item invocations can sometimes achieve better performance using halt-resume (versus work-group and kernel invocations, which always achieve better performance with polling), we separate results for WI-polling and WI-halt-resume. GENESYS enables our GPU *grep* implementation to print output lines to console or files using standard OS output. GENESYS achieves 2.0-2.3× speedups over an OpenMP version of *grep*.

Figure 13 also shows that GENESYS' flexibility with invocation granularity, blocking/non-blocking, and strong/relaxed ordering can boost performance (see Section V). For our *grep* example, because a file only needs to be searched until the first instance of a matching word, a work-item can immediately invoke a *write* of the filename, rather than waiting for all matching files. We find that WI-halt-resume outperforms both WG and WI-polling by roughly 3-4%.

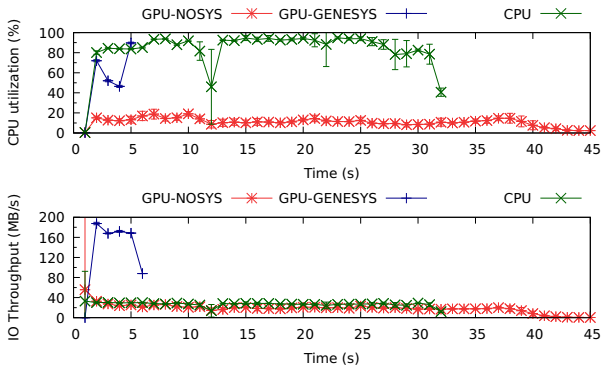

Figure 14. Wordcount I/O and CPU utilization reading from SSD. Graphs show average and standard deviation of 10 runs.

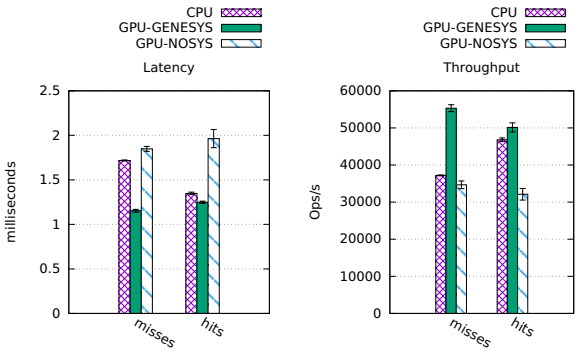

Figure 15. Latency and throughput of memcached. Graphs show average and standard deviation of 20 runs.

Workload from prior work: GENESYS also supports a version of the same workload that is evaluated in the original GPUfs work; ~~**i.e.,**~~ traditional word count where using *open*, *read*, and *close* system calls. Figure 13 shows our results. We compare the performance of a parallelized CPU version of the workload with OpenMP, a GPU version of the workload with no system calls, and a GPU version with GENESYS. Both CPU and GPU workloads are configured to search for occurrences of 64 strings. We found that with GENESYS, these system calls were best invoked at work-group granularity with blocking and weak-ordering semantics. All results are collected on a system with an SSD.

We found that GENESYS achieves nearly 6× performance improvement over the CPU version. Without system call support, the GPU version is far worse than the CPU version. Figure 14 sheds light on these benefits. We plot traces for CPU and GPU utilization and disk I/O throughput. GENESYS extracts much higher throughput from the underlying storage device (up to 170MB/s compared to the CPU's 30MB/s). Offloading search string processing to the GPU frees up the CPU to process system calls effectively. The change in CPU utilization between the GPU workload and CPU workload reveals this trend. In addition, we found that the GPU's ability to launch more concurrent I/O requests enabled the I/O scheduler to make better scheduling decisions.

D. Network Workloads

We have studied the benefits of GENESYS for network I/O in the context of *memcached*. While this can technically be implemented using GPUnet, the performance implications are unclear because the original GPUnet paper used APIs that extracted performance using dedicated RDMA hardware. We make no such assumptions and focus on the two core commands – *SET* and *GET*. *SET* stores a key-value pair, and *GET* retrieves a value associated with a key if it is present. Our implementation supports a binary UDP *memcached* version with a fixed-size hash table as a back-end storage. The hash table is shared between CPU and GPU, and its bucket size and count are configurable. Further, this *memcached* implementation enables concurrent operations. CPUs can handle *SETs* and *GETs*, while the GPU supports only *GETs*. Our GPU implementation parallelizes the hash computation, bucket lookup, and data copy. We use *sendto* and *recvfrom* system calls for UDP network access. These system calls are invoked at work-group granularity with blocking and weak ordering as this performs best.

Figure 15 compares the performance of a CPU version of this workload with GPU versions using GENESYS. GPUs accelerate *memcached* by parallelizing lookups on buckets with more elements. For example, Figure 15 shows speedups when there are 1024 elements per bucket (with 1KB data size). Without system calls, GPU performance lags behind CPU performance. However, GENESYS achieves 30-40% latency and throughput benefits over not just CPU versions, but also GPU versions without direct system calls.

E. Device Control

Finally, we also used GENESYS to implement *ioctl* system calls. As an example, we used *ioctl* to query and control framebuffer settings. The implementation is straightforward; the GPU opens */dev/fb0*, and issues a series of *ioctl* commands to query and set settings of the active frame buffer. It then proceeds to *mmap* the framebuffer memory and fill it with data from a previously *mmaped* raster image. This results in the image displayed on computer screen. While not a critical GPGPU application, this *ioctl* example demonstrates the generality and flexibility of OS interfaces implemented by GENESYS.

IX. DISCUSSION

Asynchronous system call handling: GENESYS enqueues the GPU system call's kernel task and processes it outside of the interrupt context. We do this because Linux is designed such that few operations can be processed in an interrupt handler. A potential concern with this design, however, is it defers the system call processing to potentially *past* the end of the life-time of the GPU thread and potentially the process that created the GPU thread itself! It is an example of a more general problem with asynchronous system calls [49].

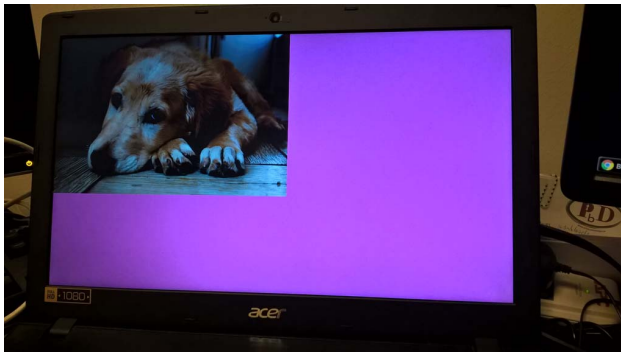

Figure 16. Raster image copied to the framebuffer by the GPU.

Our solution is to provide a new function call, invoked by the CPU, that ensures all GPU system calls have completed before the termination of the process.

Related work: Beyond work already discussed [25, 26, 28], the latest generation of C++AMP [23] and CUDA [24] provide access to the memory allocator. These efforts use a user-mode service thread on the CPU to proxy requests from the GPU [27]. Like GENESYS, system call requests are placed in a shared queue by the GPU. From there, however, the designs are different. Their user-mode thread polls this shared queue and "thunks" the request to the libc runtime or OS. This incurs added overhead for entering and leaving the OS kernel.

Some studies provide network access to GPU code [26, 50–52]. NVidia provides GPUDirect [52], used by several MPI libraries [53–55], that allows the NIC to bypass main memory and communicate directly to memory on the GPU itself. GPUDirect does not provide a programming interface for GPU-side code. The CPU must initiate communication with the NIC. Oden exposed the memory-mapped control interface of the NIC to the GPU and thereby allowed the GPU to directly communicate with the NIC [51]. This low-level interface, however, lacks the benefits of a traditional OS interface (e.g., protection, sockets, TCP).

X. CONCLUSIONS

We shed light on research questions fundamental to the idea of accessing OS services from accelerators by realizing an interface for generic POSIX system call support on GPUs. Enabling such support requires subtle changes of existing kernels. In particular, traditional OSes assume that system call processing occurs in the same context as the invoking thread, and this needs to change for accelerators. We have released GENESYS to make these benefits accessible for broader research on GPUs.

XI. ACKNOWLEDGMENTS

We thank the National Science Foundation, which partially supported this work through grants 1253700 and 1337147. We thank Guilherme Cox and Mark Silberstein for their feedback. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. PCIe is a registered trademark of PCI-SIG Corporation. ARM is a registered trademark of ARM Limited. OpenCL™ is a trademark of Apple Inc. used by permission by Khronos. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] S. Mittal and J. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," in *ACM Computing Surveys*, 2015.

[2] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," in *Computer Graphics Forum, Vol. 26, Num. 1, pp 80-113*, 2007.

[3] D. Tarditi, S. Puri, and J. Oglesby, "Acelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[4] J. Vesely, A. Basu, M. Oskin, G. Loh, and A. Bhattacharjee, "Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.

[5] J. Obert, J. V. Waveran, and G. Sellers, "Virtual Texturing in Software and Hardware," in *SIGGRAPH Courses*, 2012.

[6] G. Cox and A. Bhattacharjee, "Efficient Address Translation with Multiple Page Sizes," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[7] L. Olson, J. Power, M. Hill, and D. Wood, "Border Control: Sandboxing Accelerators," in *International Symposium on Microarchitecture (MICRO)*, 2007.

[8] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[9] J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[10] N. Agarwal, D. Nellans, E. Ebrahimi, T. Wenisch, J. Danskin, and S. Keckler, "Selective GPU Caches to Eliminate CPU-GPU HW Cache Coherence," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[11] H. Foundation, "HSA Programmer's Reference Manual [Online]," 2015. http://www.hsafoundation.com/?ddownload=4945.

[12] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review," 2012.

[13] J. Power, A. Basu, J. Gu, S. Puthoor, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *International Symposium on Microarchitecture (MICRO)*, 2013.

[14] S. Sahar, S. Bergman, and M. Silberstein, "Active Pointers: A Case for Software Address Translation on GPUs," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[15] C. J. Thompson, S. Hahn, and M. Oskin, "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis," in *International Symposium on Microarchitecture (MICRO)*, 2002.

[16] J. Owens, U. Kapasi, P. Mattson, B. Towles, B. Serebrin, S. Rixner, and W. Dally, "Media Processing Applications on the Imagine Stream Processor," in *International Conference on Computer Design (ICCD)*, 2002.

[17] J. Owens, B. Khailany, B. Towles, and W. Dally, "Comparing Reyes and OpenGL on a Stream Architecture," in *SIGGRAPH, EUROGRAPHICS Conference on Graphics Hardware*, 2002.

[18] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. Owens, "A Bandwidth-Efficient Architecture for

Media Processing," in *International Symposium on Microarchitecture (MICRO)*, 1998.
[19] B. Serebrin, J. Owens, B. Khailany, P. Mattson, U. Kapasi, C. Chen, J. Namkoong, S. Crago, S. Rixner, and W. Dally, "A Stream Processor Development Platform," in *International Conference on Computer Design (ICCD)*, 2002.
[20] K. Group, "OpenGL," 1992. https://www.khronos.org/opengl/.
[21] Microsoft, "Programming Guide for DirectX [Online]," 2016. https://msdn.microsoft.com/en-us/library/windows/desktop/ff476455(v=vs.85).aspx.
[22] K. Group, "OpenCL," 2009. https://www.khronos.org/opencl/.
[23] Microsoft, "C++ AMP (C++ Accelerated Massive Parallelism) [Online]," 2016. https://msdn.microsoft.com/en-us/library/hh265137.aspx.
[24] NVidia, "CUDA Toolkit Documentation," 2016. "http://docs.nvidia.com/cuda".
[25] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: integrating file systems with GPUs," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
[26] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
[27] J. Owens, J. Stuart, and M. Cox, "GPU-to-CPU Callbacks," in *Workshop on Unconventional High Performance Computing*, 2010.
[28] S. Bergman, T. Brokhman, T. Cohen, and M. Silberstein, "SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs," in *Usenix Technical Conference (ATC)*, 2017.
[29] AMD, "Radeon Open Compute [Online]," 2017. https://rocm.github.io/.
[30] AMD, "ROCK_syscall," 2017. https://github.com/RadeonOpenCompute/ROCK_syscall.
[31] AMD, "ROCT_syscall," 2017. https://github.com/RadeonOpenCompute/ROCT_syscall.
[32] AMD, "HCC_syscall," 2017. https://github.com/RadeonOpenCompute/HCC_syscall.
[33] AMD, "Genesys_syscall_test," 2017. https://github.com/RadeonOpenCompute/Genesys_syscall_test.
[34] R. Draves, B. Bershad, R. Rashid, and R. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," in *Symposium on Operating Systems Principles (SOSP)*, 1991.
[35] O. Shivers and M. Might, "Continuations and Transducer Composition," in *International Symposium on Programming Languages Design and Implementation (PLDI)*, 2006.
[36] AMD, "Graphics Core Next 3.0 ISA [Online]," 2013. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/07/AMD_GCN3_Instruction_Set_Architecture.pdf.
[37] Intel, "Intel Open Source HD Graphics Programmers Reference Manual [Online]," 2014. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-bdw-vol02b-commandreference-instructions_0_0.pdf.
[38] NVidia, "NVIDIA Compute PTX: Parallel Thread Execution," 2009. https://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf.
[39] PCI-SIG, "Atomic Operations [Online]," 2008. https://pcisig.com/sites/default/files/specification_documents/ECN_Atomic_Ops_080417.pdf.
[40] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamarić, "Portable inter-workgroup barrier synchronisation for gpus," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), pp. 39–58, ACM, 2016.
[41] T. Rogers, M. O'Connor, and T. Aamodt, "Cache-Conscious Wavefront Scheduling," in *International Symposium on Microarchitecture (MICRO)*, 2012.
[42] T. Rogers, M. O'Connor, and T. Aamodt, "Divergence-Aware Warp Scheduling," in *International Symposium on Microarchitecture (MICRO)*, 2013.
[43] A. ElTantawy and T. Aamodt, "Warp Scheduling for Fine-Grained Synchronization," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.
[44] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and Fair Multi-programming in GPUs via Pattern-Based Effective Bandwidth Management," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.
[45] O. Kayiran, A. Jog, M. Kandemir, and C. Das, "Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.
[46] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das, "Controlled Kernel Launch for Dynamic Parallelism in GPUs," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.
[47] AMD, "ROCm: a New Era in Open GPU Computing [Online]," 2016. https://radeonopencompute.github.io/index.html.
[48] A. Sasidharan and M. Snir, "MiniAMR - A miniapp for Adaptive Mesh Refinement," 2014. http://hdl.handle.net/2142/91046.
[49] Z. Brown, "Asynchronous System Calls," in *Ottawa Linux Symposium*, 2007.
[50] J. A. Stuart, P. Balaji, and J. D. Owens, "Extending mpi to accelerators," *PACT 2011 Workshop Series: Architectures and Systems for Big Data*, Oct. 2011.
[51] L. Oden, "Direct Communication Methods for Distributed GPUs," in *Dissertation Thesis, Ruprecht-Karls-Universitat Heidelberg*, 2015.
[52] NVidia, "NVIDIA GPUDirect [Online]," 2013. https://developer.nvidia.com/gpudirect.
[53] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 2595–2605, Oct 2014.
[54] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.
[55] IBM, "IBM Spectrum MPI," 2017. http://www-03.ibm.com/systems/spectrum-computing/products/mpi/.