Augmenting Operating Systems With the GPU

Weibin Sun Robert Ricci

University of Utah, School of Computing
{wbsun, ricci}@cs.utah.edu http://www.cs.utah.edu/flux/

Abstract

The most popular heterogeneous many-core platform, the CPU+GPU combination, has received relatively little attention in operating systems research. This platform is already widely deployed: GPUs can be found, in some form, in most desktop and laptop PCs. Used for more than just graphics processing, modern GPUs have proved themselves versatile enough to be adapted to other applications as well. Though GPUs have strengths that can be exploited in systems software, this remains a largely untapped resource. We argue that augmenting the OS kernel with GPU computing power opens the door to a number of new opportunities. GPUs can be used to speed up some kernel functions, make other scale better, and make it feasible to bring some computation-heavy functionality into the kernel. We present our framework for using the GPU as a co-processor from an OS kernel, and demonstrate a prototype in Linux.

1 Introduction

Modern GPUs can be used for more than just graphics processing; through frameworks like CUDA [1], they can run general-purpose programs. While not well-suited to *all* types of programs, they excel on code that can make use of their high degree of parallelism. Most uses of so-called "General Purpose GPU" (GPGPU) computation have been outside the realm of systems software. However, recent work on software routers [10] and encrypted network connections [14] has given examples of how GPGPUs can be applied to tasks more traditionally within the realm of operating systems. We claim that these uses are only scratching the surface. In Section 2, we give more examples of how GPU computing resources can be used to improve performance and bring new functionality into OS kernels.[1] These include tasks that have applications on the desktop, on the server, and in the datacenter.

Consumer GPUs currently contain up to 512 cores [2], and are fairly inexpensive: at the time of writing, a current-generation GPU with 336 cores can be purchased for as little as $160, or about 50 cents per core. GPUs are improving at a rapid pace: the theoretical performance of NVIDIA's consumer GPUs improved from 500 gigaFLOPS in 2007 (GeForce 8800) to over 1.3 teraFLOPS in 2009 (GTX 480) [18]. Furthermore, the development of APUs, which contain a CPU and a GPU on the same chip, is likely to drive even wider adoption. This represents a large amount of computing power, and we argue that systems software should not overlook it.

Some recent OS designs have tried to embrace processor heterogeneity. Helios [17] provides a single OS image across multiple heterogeneous cores so as to simplify program development. Barrelfish [5] treats a multicore system as a distributed system, with independent OS kernels on each core and communication via message-passing. Both, however, are targeted at CPUs that have support for traditional OS requirements, such as virtual memory, interrupts, preemption, controllable context switching, and the ability to interact directly with I/O devices. GPUs lack these features, and are thus simply not suited to designs that treat them as peers to traditional CPUs. Instead, they are better suited for use as co-processors.

Because of this, we argue that GPUs can be and should be used to augment OS kernels, but that a heterogeneous OS cannot simply treat the GPU as a fully functional CPU with different ISA. The OS kernel needs a new framework if it is to take advantage of the opportunities presented by GPUs. To demonstrate the feasibility of this idea, we designed and prototyped KGPU, a framework for calling GPU code from the Linux kernel. We describe this framework and the challenges we faced in designing it in Section 3

2 Applications

We have three motivations for offloading OS kernel tasks to the GPU:

- To reduce the *latency* for tasks that run more quickly on the GPU than on the CPU
- To exploit the GPU's parallelism to increase the *throughput* for some types of operations, such as in-

[1] In GPU terminology, a program running on the GPU is called a "kernel." To avoid confusion, we use the term "OS kernel" or "GPU kernel" when the meaning could be ambiguous.

creasing the number of clients a server can handle
- To make feasible incorporation of *new functionality* into the OS kernel that runs too slowly on the CPU

These open the door for new avenues of research, with the potential for gains in security, efficiency, functionality, and performance of the OS. In this section, we describe a set of tasks that have been shown to perform well on the CPU, and discuss how they show promise for augmenting the operating system.

Network Packet Processing: Recently, the GPU has been demonstrated to show impressive performance enhancements for software routing and packet processing. PacketShader [10] is capable of fast routing table lookups, achieving a rate of close to 40Gbps for both IPv4 and IPv6 forwarding and at most 4x speedup over the CPU-only mode using two NVIDIA GTX 480 GPUs. For IPSec, PacketShader gets a 3.5x speedup over the CPU. Additionally, a GPU-accelerated SSL implementation, SSLShader [14] runs four times faster than an equivalent CPU version.

While PacketShader shows the feasibility of moving part of the network stack onto GPUs and delivers excellent throughput, it suffers from a higher round trip latency for each packet when compared to the CPU-only approach. This exposes the weakness of the GPU in a latency-oriented computing model: the overhead caused by copying data and code into GPU memory and then copying results back affects the overall response time of a GPU computing task severely. To implement GPU offloading support, OS kernel designers must deal with this latency problem. Our KGPU prototype decreases the latency of GPU computing tasks with the techniques discussed in section 3.

Though there are specialized programmable network interfaces which can be used for packet processing, the CPU+GPU combination offers a compelling alternative: the high level of interest in GPUs, and the fact that they are sold as consumer devices drives wide deployment, low cost, and substantial investment in improving them.

In-Kernel Cryptography: Cryptography operations accelerated by GPUs have been shown to be feasible and to get significant speedup over CPU versions [12, 14]. OS functionality making heavy use of cryptography includes IPSec [10] , encrypted filesystems, and content-based data redundancy reduction of filesystem blocks [21] and memory pages [9]. Another potential application of the GPU-accelerated cryptography is trusted computing based on the Trusted Platform Module (TPM). A TPM is traditionally hardware, but recent software implementations of the TPM specification, such as vTPM [6], are developed for hypervisors to provide trusted computing in virtualized environments where virtual machines cannot access the host TPM directly. Because TPM operations are cryptography-heavy (such as secure hashing of executables and memory regions), they can also potentially be accelerated with GPUs.

The Linux kernel contains a general-purpose cryptography library used by many of its subsystems. This library can easily be extended to offload to the GPU. Our KGPU prototype implements AES on the GPU for the Linux kernel, and we present a microbenchmark in Section 3.3 showing that it can outperform the CPU by as much as 6x for sufficiently large block sizes. Due to the parallel nature of the GPU, blocks of data can represent either large blocks of a single task or a number of smaller blocks of different tasks. Thus, the GPU can not only speed up bulk data encryption but also scale up the number of simultaneous users of the cryptography subsystem, such as SSL or IPSec sessions with different clients.

Pattern Matching Based Tasks: The GPU can accelerate regular expression matching, with speedups of up to 48x reported over CPU implementations [22]. A network intrusion detection system (NIDS) with GPU-accelerated regular expression matching [22] demonstrated a 60% increase in overall packet processing throughput on fairly old GPU hardware. Other tasks such as information flow control inside the OS [16], virus detection [3] (with two orders of magnitude speedup), rule-based firewalls, and content-based search in filesystems can potentially benefit from GPU-accelerated pattern matching.

In-Kernel Program Analysis: Program analysis is gaining traction as a way to enhance the security and robustness of programs and operating systems. For example, the Singularity OS [13] relies on safe code for process isolation rather than traditional memory protection. Recent work on EigenCFA has shown that some types of program analysis can be dramatically sped up using a GPU [20]. By re-casting the Control Flow Analysis problem (specifically, 0CFA) in terms of matrix operations, which GPUs excel at, EigenCFA is able to see a speed up of 72x, nearly two orders of magnitude. The authors of EigenCFA are working to extend it to pointer analysis as well. With speedups like this, analysis that was previously too expensive to do at load time or execution time becomes more feasible; it is conceivable that some program analysis could be done as code is loaded into the kernel, or executed in some other trusted context.

Basic Algorithms: A number of basic algorithms, which are used in many system-level tasks, have been shown to achieve varying levels of speedup on GPUs. These include sort, search [19] and graph analysis [11]. GPU-accelerated sort and search fit the functionality of filesystems very well. An interesting potential use of GPU-accelerated graph analysis is for in-kernel garbage collection (GC). GC is usually considered to be time-consuming because of its graph traversal operation, but a recent patent application [15] shows it is possible to do the GC on GPUs, and that it may have better performance

than on CPUs. Besides GC for memory objects, filesystems also use GC-like operations to reorganize blocks, find dead links, and check unreferenced blocks for consistency. Another example of graph analysis in the kernel is the Featherstitch [7] system, which exposes the dependencies among writes in a reliable filesystem. One of the most expensive parts of Featherstich is analysis of dependencies in its *patch graph*, a task we believe could be done efficiently on the GPU.

GPGPU computing is a relatively new field, with the earliest frameworks appearing in 2006. Many of the applications described in this section are, therefore, early results, and may see further improvements and broader applicability. With more and more attention being paid to this realm, we expect more valuable and interesting GPU-accelerated in-kernel applications to present themselves in the future.

3 GPU Computing For The Linux Kernel

Because of the functional limitations discussed in Section 1, it is impractical to run a fully functional OS kernel on a GPU. Instead, our KGPU framework runs a traditional OS kernel on the CPU, and treats the GPU as a coprocessor. We have implemented a prototype of KGPU in the Linux kernel, using NVIDIA's CUDA framework to run code on the GPU.

3.1 Challenges

KGPU must deal with two key challenges to efficiently use the GPU from the OS kernel: the overhead of copying data back and forth, and latency-sensitive launching of tasks on the GPU.

Data Copy Overhead: A major overhead in GPGPU computing is caused by the fact that the GPU has its own memory, separate from the main memory used by the CPU. Transfer between the two is done via DMA over the PCIe bus. Applications using the GPU must introduce two copies: one to move the input to GPU memory, and another to return the result. The overhead of these copies is proportional to the size of the data.

There are two kinds of main memory the CUDA driver can use: one is general memory (called pageable memory in CUDA), allocated by `malloc()`. The other is *pinned* memory, allocated by the CUDA driver and `mmap`-ed into the GPU device. Pinned memory is much faster than the pageable memory when doing DMA.

In KGPU, we use pinned memory for all buffers because of its superior performance. The downside of pinned memory is that it is locked to specific physical pages, and cannot be paged out to disk; hence, we must be careful about managing our pinned buffers. This management is described in Subsection 3.2.

GPU Kernel Launch Overhead: Another overhead is caused by the GPU kernel launch, which introduces DMA transfers of the GPU kernel code, driver set-up for kernel execution and other device-related operations. This sets a lower bound on the time the OS kernel must wait for the GPU code to complete, so the lower we can make this overhead, the more code can potentially benefit from GPU acceleration. This overhead is not high when the GPU kernel execution time or the data copy overhead dominates the total execution time, as is the case for most GPGPU computing, which is throughput-oriented [8].

OS kernel workloads, on the other hand, are likely to be dominated by a large number of smaller tasks, and latency of each operation is of greater importance. Though larger tasks can be created by batching many small requests, doing so increases the latency for each request. CUDA has a "stream" [18] technology that allows kernel execution to proceed concurrently with GPU kernel execution and data copy. By itself, this helps to improve throughput, not latency, but we make use of it to communicate between code running on the GPU and CPU.

Instead of launching a new GPU kernel every time the OS wants to call a GPU code, we have designed a new GPU kernel execution model, which we call the Non-Stop Kernel (NSK). The NSK is small, is launched only once, and does not terminate. To communicate with the NSK, we have implemented a new CPU-GPU message-based communication method. It allows messages to be passed between the GPU and main memory while a GPU kernel is still running. This is impossible in traditional CUDA programming, in which the CPU has to explicitly wait for synchronization with the GPU.We use pinned memory to pass these messages, and NVIDIA's streaming features to asynchronously trigger transfers of the message buffer back and forth between CPU and GPU memory. Requests are sent from the CPU to the NSK as messages. The NSK executes the requested service, which it has pre-loaded into the GPU memory. Similarly, the CPU receives completion notifications from the NSK using these messages.

We measured the time to launch an empty GPU kernel, transfer a small amout of input data to it (4KB), and wait for it to return. Though most CUDA benchmarks measure only the execution time on the GPU, we measured time on the CPU to capture the entire delay the OS kernel will observe. NSK outperforms the traditional launch method by a factor of 1.3x, reducing the base GPU kernel launch time to $16.7\mu s$ for a kernel with 512 threads, $17.3\mu s$ for 1024 threads, and $18.3\mu s$ for 2048 threads. While this is much larger than the overhead of calling a function on the CPU, as we will show in Section 3.3, the speedup in execution time can be well worth the cost.

Because of a limitation in CUDA that does not allow a running GPU kernel to change its number of threads

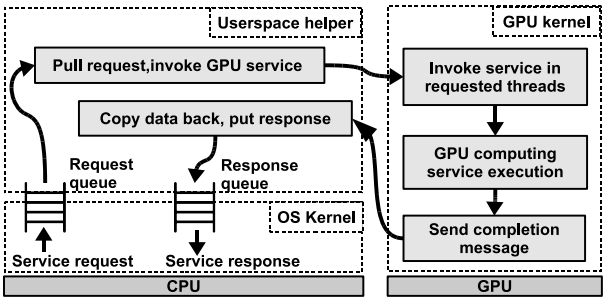

Figure 1: KGPU framework architecture

dynamically, NSK switches to a traditional CUDA kernel launch model when a service requires more threads on the GPU. This switch will not be necessary in future when the vendors provide the functionality of dynamically creating new GPU threads.

3.2 KGPU Architecture

Our framework for calling the GPU is shown in Figure 1. It is divided into three parts: a module in the OS kernel, a user-space helper process, and NSK running on the GPU. The user-space helper is necessitated by the closed-source nature of NVIDIA's drivers and CUDA runtime, which prevent the use of CUDA directly from inside the kernel.

To call a function on the GPU, the OS kernel follows the following steps:

- It requests one of the pinned-memory buffers, and fills it with the input. If necessary, it also requests a buffer for the result.
- It builds a service request. Services are CUDA programs that have been pre-loaded into NSK to minimize launch time. The service request can optionally include a completion callback.
- It places the service request into request queue.
- It waits for the request to complete, either by blocking until the completion callback is called or busy-waiting on the response queue.

The user-space helper for KGPU watches the request queue, which is in memory shared with the OS kernel. Upon receipt of a new service request, the helper DMAs the input data buffer to the GPU using the CUDA APIs. This can proceed concurrently with another service running on the GPU. When the DMA is complete, the helper sends a service request message to NSK using the message-passing mechanism described in Section 3.1. When the NSK receives the message, it calls the service function, passing it pointers to the input buffer and output buffer. When the function completes, the NSK sends a completion message to the CPU side, and resumes polling for new request messages. The user-level helper relays the result back to the OS kernel through their shared response queue.

To avoid a copy between the kernel module and the user-space helper, the pinned data buffers allocated by the CUDA driver are shared between the two. Also, because NSK allows the user-space helper to work asynchronously via messages, service execution on the GPU and data buffer copies between main memory and GPU memory can run concurrently. As a result, the data buffers locked in physical memory are managed carefully to cope with the complex uses. On the CPU side, buffers can be used for four different purposes:

1. Preparing for a future service call by accepting data from a caller in the OS kernel
2. To DMA input data from main memory to the GPU for the next service call
3. To DMA results from the last service call from GPU memory to main memory
4. Finishing a previous service call by returning data to the caller in the OS kernel

Each of these tasks can be performed concurrently, so, along with the service currently running on the GPU, the total depth of the service call pipeline is five stages. In the current KGPU prototype, we statically allocate four buffers, and each changes its purpose over time. For example, after a buffer is prepared with data from the caller, it becomes the host to GPU DMA buffer.

On the GPU, we use three buffers: at the same time that one is used by the active service, a second may receive input for the next service from main memory via DMA, and a third may be copying the output of the previous service to main memory.

3.3 Example: A GPU AES Implementation

To demonstrate the feasibility of KGPU, we implemented the AES encryption algorithm as a service on the GPU for the Linux crypto subsystem. Our implementation is based on an existing CUDA AES implementation [4], and uses the ECB cipher mode for maximum parallelism. We did a microbenchmark to compare its performance with the original CPU version in the Linux kernel, which is itself optimized by using special SSE instructions in the CPU. We used a 480-core NVIDIA GTX 480 GPU, a quad-core Intel Core i7-930 2.8 GHz CPU and 6GB of DDR3 PC1600 memory. The OS is Ubuntu 10.04 with Linux kernel 2.6.35.3.

We get a performance increase of up to 6x, as shown in Figure 2. The results show that the GPU AES-ECB outperforms the CPU implementation when the size of the data is 8KB or larger, which is two memory pages when using typical page sizes. So, kernel tasks that depend on per-page encryption/decryption, such as encrypted filesystems, can be accelerated on the GPU.

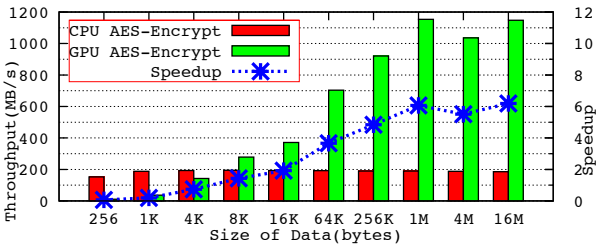

Figure 2: Encryption performance of KGPU AES. Decryption, not shown, has similar performance.

4 Discussion

The GPU-augmented OS kernel opens new opportunities for systems software, with the potential to bring performance improvements, new functionality, and security enhancements into the OS.

We will continue to develop and improve KGPU and to implement more GPU functions in our framework. One such improvement will be dynamically dispatching tasks to the CPU or GPU depending on their size. As seen in Figure 2, the overheads associated with calling the GPU mean that small tasks may run faster on the CPU. Since the crossover point will depend on the task and the machine's specific hardware, a good approach may be to calibrate it using microbenchmarks at boot time. Another improvement will be to allow other kernel subsystems to specifically request allocation of memory in the GPU pinned region. In our current implementation, GPU inputs must be copied into these regions and the results copied out, because the pinned memory is used only for communication with the GPU. By dynamically allocating pinned buffers, and allowing users of the framework to request memory in this region, they can manage structures such as filesystem blocks directly in pinned memory, and save an extra copy. This would also allow multiple calls to be in the preparing and post-service callback stages at once.

We expect that future developments in GPUs will alleviate some of the current limitations of KGPU. While the closed nature of current GPUs necessitates interacting with them from user-space, the trend seems to be towards openness; AMD has recently opened their high-end 3D GPU drivers and indicated that drivers for their upcoming APU platform will also be open-source. Furthermore, by combining a GPU and CPU on the same die, APUs, e.g. Intel SandyBridge and AMD Fusion, are likely to remove the memory copy overhead with shared cache between CPU cores and GPU cores; lower copy overhead will mean that the minimum-sized task that can benefit from GPU offloading will drop significantly.

References

[1] CUDA. `http://www.nvidia.com/object/cuda_home.html`.

[2] GTX580 GPU. `http://www.nvidia.com/object/product-geforce-gtx-580-us.html`.

[3] Kaspersky Lab. `http://www.kaspersky.com/news?id=207575979`.

[4] OpenSSL CUDA AES Engine. `http://code.google.com/p/engine-cuda`.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP 2009. ACM.

[6] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. USENIX Security 2006.

[7] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. SOSP 2007. ACM.

[8] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Comm. ACM*, 53:58–66, 2010.

[9] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. OSDI 2008.

[10] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. SIGCOMM 2010.

[11] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. HiPC 2007.

[12] O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. USENIX Security 2008.

[13] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS OSR*, 41:37–49, 2007.

[14] K. Jang, S. Han, S. Han, S. Moon, and K. Park. Accelerating SSL with GPUs. SIGCOMM 2010. ACM.

[15] A. S. Jiva and G. R. Frost. GPU assisted garbage collection. `www.faqs.org/patents/app/20100082930`.

[16] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. SOSP 2007. ACM.

[17] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. SOSP 2009. ACM.

[18] NVIDIA. CUDA C Programming Guide 3.2.

[19] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[20] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. Eigen-CFA: Accelerating flow analysis with GPUs. PoPL 2011.

[21] K. Tangwongsan, H. Pucha, D. G. Andersen, and M. Kaminsky. Efficient similarity estimation for systems exploiting data redundancy. INFOCOM 2010.

[22] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular expression matching on graphics hardware for intrusion detection. RAID 2009.