Accelerating The Cloud with Heterogeneous Computing

Sahil Suneja, Elliott Baron, Eyal de Lara, Ryan Johnson
University of Toronto

Abstract

Heterogeneous multiprocessors that combine multiple CPUs and GPUs on a single die are posed to become commonplace in the market. As seen recently from the high performance computing community, leveraging a GPU can yield performance increases of several orders of magnitude. We propose using GPU acceleration to greatly speed up cloud management tasks in VMMs. This is only becoming possible now that the GPU is moving on-chip, since the latency across the PCIe bus was too great to make fast, informed decisions about the state of a system at any given point. We explore various examples of cloud management tasks that can greatly benefit from GPU acceleration. We also tackle tough questions of how to manage this hardware in a multi-tenant system. Finally, we present a case study that explores a common cloud operation, memory deduplication, and show that GPU acceleration can improve the performance of its hashing component by a factor of over 80.

1 Introduction

Over the last decade, Graphical Processing Units (GPU) have been widely used to speed up the performance of applications from a wide range of domains beyond image processing, including bioinformatics, fluid dynamics, computational finance, weather and ocean modeling, data mining, analytics and databases, among others.

We argue that GPUs could be used to greatly accelerate common systems and management tasks in cloud environments, such as page table manipulation during domain creation and migration, memory zeroing, memory deduplication, and guest domain virus scanning. This class of tasks has so far not been amenable to GPU acceleration due to the need to perform DMA transfers over the PCIe bus associated with traditional discrete GPUs. We argue that the rollout of heterogeneous architectures, such as the AMD Fusion and Intel Sandy Bridge, which include a GPU on-socket with direct access to main memory (as shown in Figure 1), is a game changer that motivates a re-evaluation of how system-level tasks are implemented in cloud environments.

In this paper, we give examples of system level operations in a virtualization-based cloud that can benefit from GPU acceleration, and we discuss the challenges associated with efficiently sharing and managing the

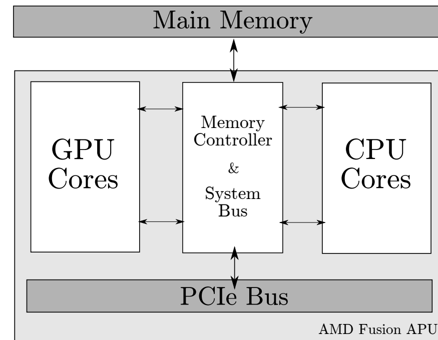

Figure 1: AMD Fusion architecture [1].

GPU in a cloud environment. To illustrate the potential benefits of GPU acceleration of common cloud management tasks, we conduct a case study which uses GPU acceleration to speed up the hashing component of a memory deduplication task. Experiments with two off-the-shelf discrete GPU cards (a heterogeneous chip with an on-socket GPU, such as the AMD Fusion, was not commercially available at the time we ran our experiments) show that when memory transfer time is not included, it is possible to achieve speedups of more than 80 times over sequential CPU hashing. While not definitive, these results give a good indication of the potential performance that an on-socket GPU with direct access to main memory could achieve. Even with the current memory transfer limitations, we observed a 6 fold speedup.

2 Accelerating The Cloud

This section details several examples of data parallel cloud management tasks that a VMM routinely performs, which can benefit from GPU acceleration.

Memory Cleanup: When a VMM destroys a VM, it is expected that none of the potentially sensitive information residing in the VM's memory is leaked when it is reused. Thus, at some point before the memory is to be allocated to a new VM, it should be cleared out. Using a GPU with direct access to the memory in question, a large number of memory pages could be zeroed in parallel. This would free the CPU to perform other hypervisor management tasks or service guest VMs, and allow these pages to be reused much sooner.

Batch Page Table Updates: These updates involve remapping a guest OS' pseudo-physical page frame numbers to the actual machine page frame numbers as

the guest VM is transported to a different physical host, such as in VM migration and cloning, or is suspended and resumed at a later point in time either on the same or a different host machine. This mapping process is also required upon the creation of a fresh VM. These page table updates are all independent of each other, and can be accelerated via the SIMD (Single Instruction, Multiple Data) processing capabilities of a GPU.

Memory Hashing: Offloading compute-intensive hashing to a GPU provides a double benefit. Not only does it lower CPU overhead, but the GPU can process orders of magnitude more data per unit time; this acceleration would prove advantageous to at least two aspects of the cloud infrastructure: effective memory deduplication through page sharing, and improved flexibility in VM migration and cloning.

Page sharing [2, 5, 9] allows over-provisioning of physical memory in a virtualized environment by exploiting homogeneity in cloud workloads, i.e. multiple co-located VMs running the same operating system and similar application stacks. Page sharing eliminates all but one copy of each duplicated page, modifying guest address translations to point to that single master copy. On popular technique scans the system periodically and builds a hash table of all memory pages [2, 9]; pages which hash to the same hash table index are candidates for page sharing. Faster hashing allows more frequent scans thereby discovering more opportunities for page sharing.

Similarly, accelerated page hashing (digest generation) enables an efficient realization of content addressable storage (CAS) which can be used to accelerate VM migration and cloning. Through the high speed page matching offered by CAS, it may not be necessary to request the network transfer of all memory pages during the process of rebuiding the working memory set of a cloned or migrated VM. These might be locally available on the host system, and a rapid local retrieval may thus satisfy the page requirement.

Memory Compression: Another technique to observe memory savings is compressing infrequently accessed pages in memory. The data parallel nature of this operation at the memory page level makes this yet another candidate for GPU acceleration, allowing extra memory to be available faster.

Virus Signature Scanning: Searching for virus signatures in physical memory or in incoming network data packets is another service that a hypervisor may provide to all its guests. Scanning memory or packet data for pattern matching of known signatures can be accelerated considerably by GPU processing, leading to more aggressive security monitoring [7, 8].

3 Managing a Virtualized GPU

This section outlines various hardware management challenges that arise in heterogeneous processors. We show that a VMM can address these challenges elegantly on a multi-tenant system. While on-chip graphics are the present, we also look at what other kinds of hardware accelerators may be useful in the future.

Exploiting processor heterogeneity for hypervisor level management tasks will usually require escalated privileges. Thus, an important challenge lies with sharing hardware resources effectively between privileged systems code (hypervisor and control-domain functions) and a non-privileged guest's applications. Further, any such sharing must maintain security and performance isolation across multiple guest VMs. In particular, a rogue or buggy process must not interfere with the confidentiality, integrity, or availability of another which happens to share the same GPU.

An easy, but suboptimal solution is to restrict access of the accelerator just to the hypervisor or a privileged control-domain, such as Xen's Dom0. Security would not be a concern as the GPU hardware is not even exposed to unprivileged guests, but these resources will likely remain underutilized.

A better alternative would expose the GPU to the guests in time slices. The VMM can use traditional CPU scheduling techniques to manage execution on the GPU, with fine-grained control over the amount of time it allocates for itself and its guests. Furthermore, tasks cannot interfere with each other as the device is allocated to only one task at a time. Again, the GPU may go underutilized if that task does not make full use of the resources it was granted. Although only one task runs at a time, it is also worth noting that the VMM will still need to impose memory protection on this task. For instance, the VMM must prevent tasks from accessing data left behind by previously completed or preempted tasks.

A superior policy might introduce space multiplexing, partitioning GPU resources concurrently among tasks from several guests. Space-based multiplexing could be implemented in several ways. If hardware support for virtualization were to become available on the GPU, then individual GPU slices could be mapped directly onto guests. For this to be possible, the GPU would have to police memory accesses itself. The VMM would then be in charge of managing slice allocation.

Given the current lack of hardware support for virtualization on existing GPUs, an alternative approach would implement a software-based approach where the VMM, or rather its privileged domain, arbitrates access to the GPU by virtualizing the General Purpose GPU (GPGPU) API. However, the VMM would also have to provide mechanisms to support simultaneous execution of multiple compute kernels. CUDA 3.0 [6], for exam-

ple, enables simultaneous execution of tasks, but provides no memory protection to isolate them from each other.

In a software-based implementation, the guest VM would access the GPU by sending the code of the kernel to run and any associated parameters to the VMM, which would then determine which of the kernels that have been sent to it can execute simultaneously. To perform this evaluation, the VMM must be able to determine which regions of memory the kernel attempts to access. This will likely require support from the kernel compiler, and may also demand a more restrictive API to the guest (e.g. disallowing arbitrary memory access via pointer arithmetic). Once the VMM is certain of the memory the kernel will access, it can make an appropriate mapping of kernels to the GPU's execution units that is both safe and that will give high occupancy. Finally, the VMM must then enqueue the kernels using the GPGPU API. If this API supports concurrent kernel execution, then the VMM will simply enqueue the kernels using the API. If the API does not support concurrent execution, a possible workaround could be for VMM to recompile the kernels into a single "meta-kernel." The meta-kernel uses thread indexes to execute each kernel's code on a subset of the hardware.

An advantage of using the VMM to control access to the GPU is that the VMM has a global view of the system. As such, the VMM can make informed decisions about scheduling time and space on the GPU to achieve better system performance. In contrast, in the case of an entirely hardware-managed virtualized GPU, where each guest is assigned a slice, these slices may go unused if the guest does not have work to perform at that time.

4 Case Study: VM Page Sharing

To illustrate the benefits and challenges of accelerating hypervisor functions, we evaluated the use of GPUs to optimize the MD5 hashing component of a page sharing task. Since a heterogeneous chip with an on-socket GPU, e.g., AMD Fusion, was not commercially available at the time of writing, we conducted our case study using two off-the-shelf discrete GPU cards. While this experimental setup is not ideal, we believe that by isolating the memory access and computation components of the task we can get a rough estimate of the potential performance of an on-socket GPU with direct access to main memory.

4.1 General Purpose GPU Computing

We use OpenCL, a vendor-neutral heterogeneous computing framework for GPGPU computing. In OpenCL, a compute kernel defines the data parallel operation we wish to apply to the input data. Kernels are written using C functions, but when invoked are executed across a range of GPU threads or *work-items* grouped together into *work-groups*. All work-items within a work-group can access a faster shared on-chip *local memory*, see Figure 2. A kernel can be thought of as the body of a nested loop, where the outer loop corresponds to work-groups and the inner loop corresponds to work-items.

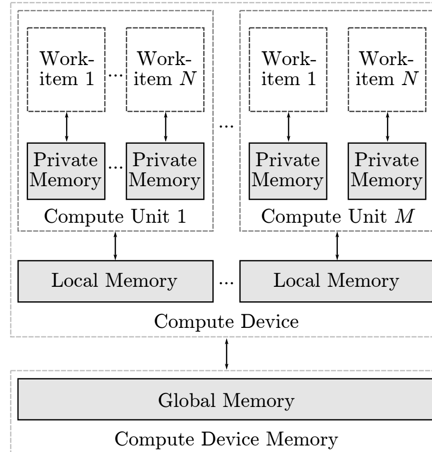

Figure 2: The OpenCL memory hierarchy. Work-groups are scheduled to compute units, each with their own low-latency shared local memory. Global memory is off-chip and has much higher latency.

4.2 GPGPU Page Sharing

We ported the MD5 implementation by Juric [4] from CUDA to OpenCL. The MD5 algorithm performs a sequence of 64 operations on each 512-bit (64-byte) chunk of the input. The resulting hash of one chunk is then fed as an input to the algorithm as it processes the next chunk. Each 4K memory page can be viewed as 64 chunks of 64 bytes each.

The need to transfer the hash output of one chunk to the next severely constraints the granularity of parallelism, limiting the assignment of a single work-item per memory page. To improve parallelism, we adopted optimizations proposed by Hu et al. [3] for hierarchical hashing. The optimized algorithm has a much finer granularity of one 64 byte chunk per work-item, enabling 64 work-items to operate in parallel on one memory page. While the value of the hash generated for a page is different from the standard MD5 hash, the encryption strength is maintained.

Figure 3 demonstrates hashing a page with 4 chunks per work-item in 2 kernel rounds. Assigning 1 chunk per work-item leads to three rounds of hashing: 4kB is reduced to 64 16-byte hashes (1024 bytes), which in turn reduced to 256- and finally 64-byte hashes. The number of successive kernel (hashing) calls can be reduced by using multiple chunks per work-item.

We present two versions of the kernel as it is the GPU

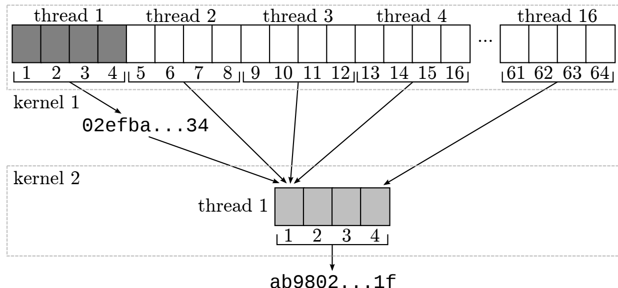

Figure 3: Our parallel hashing algorithm operating on a single 4K page. Kernel 1 hashes 64-byte chunks into intermediate values. Kernel 2 hashes the intermediate values into the final value.

	GTX 280	HD 5870
Processing cores	240	1600
Global memory size	1 GB	1 GB
Max work-group size	512	256
Wavefront size	32	64
Local memory size	16KB	32KB

Table 1: Testbed specifications.

architecture that drives the optimizations to extract maximum parallelization benefits, and no single implementation performed best on both the GPUs we used. The kernel versions are: *glmem*, which accesses the global device memory directly; and *shmem*, which makes use of faster per-work-group local (shared) memory for optimizations.

4.3 Hypervisor Integration

Integration of the accelerated hashing scheme with a virtualization environment is left for future work. We anticipate that this task would be best realized outside the hypervisor, as a user-space program in the privileged domain. This would avoid expanding the hypervisor's trusted code base and having hypervisor depend on any GPGPU framework. The user-space process would be responsible for managing hash tables and driving GPU computation and communication, while the actual sharing of pages via pseudo physical address to machine address remapping, and unsharing using copy-on-write semantics would still be handled by the hypervisor.

5 Experiments

We conducted our experiments on two GPUs: NVIDIA GeForce GTX 280 (GT200) and ATI Radeon HD 5870. The system hosting the NVIDIA GPU contains a 2.83GHz Intel Core 2 Quad CPU (Q9550). The ATI GPU is hosted on a system with a 2.80GHz Intel Core i5-760 CPU. Table 1 presents a few relevant specifications of the two GPUs we used.

The benefit of offloading hash computation to the GPU is two-fold. First, faster hashing allows for a greater memory scan frequency, thereby exploiting more opportunities for page sharing. Secondly, the computation offload results in a reduced load on the CPU allowing it to better service the guest VMs.

5.1 Speedup

We measure the time it takes to hash 100 MB of randomly generated data consisting of 25,600 4KB pages of memory. For comparison, our baseline sequential MD5 implementation takes 346 ms and 314 ms to complete when running on the main CPUs of the systems hosting the NVIDIA and ATI cards, respectively.

Tables 2 and 3 show the speedups we obtained over the sequential CPU hashing implementation for the NVIDIA and ATI cards respectively, when the data to be hashed is pre-copied to the GPUs global memory, i.e., the memory transfer time is not included.

Implementation	Work-group size	Chunks per work-item	Speedup
glmem	512	1	19.5x
shmem	240	1	40x
glmem	512	4	38.5x

Table 2: Results on NVIDIA GeForce GTX 280.

Implementation	Work-group size	Chunks per work-item	Speedup
glmem	256	1	63x
shmem	64	1	54x
glmem	256	2	87x

Table 3: Results on ATI Radeon HD 5870.

On the NVIDIA GPU, the best speedup achieved was 40x over sequential hashing using the *shmem* kernel implementation, while a *glmem* kernel yielded the highest speedup of 87x on the ATI GPU. The difference in performance between the two GPUs is a result of architecture-specific bottlenecks. Perhaps not surprisingly, each different hardware requires custom code optimizations for deriving maximum performance.

When the memory transfer times are included into our total running times, the speedups observed are about 6x for the NVIDIA GPU, and 2x for the ATI GPU. The reduced speedup is due to the overhead of data transfer from the host (CPU) memory to the device (GPU) memory, before the GPU kernel initiates computation. Processors like AMD Fusion, which house a CPU and GPU on a single die, will enable the GPU to access host memory directly. On the other hand, we expect the integrated GPUs to be less powerful that their discrete off-chip counterparts, at least for the initial architectures. Thus,

we expect the on-chip performance to lie somewhere between these lower (2x-6x) and upper bounds (40x-87x).

5.2 Hashing Overhead

Table 4 shows the overhead of the CPU and GPU versions of the hashing process when they execute concurrently with a computationally intensive process, *cpu-heavy*, on the machine hosting the ATI GPU. The process *cpu-heavy* performs 400 million floating point multiplications. Concurrently, in the background we hash 1.5 GB of memory. The experiment *gpu-hash-i* first copies the input data to the GPU, and only then does *cpu-heavy* begin its execution. Thus, *gpu-hash-i* does not include memory transfer overhead. The other experiment, *gpu-hash-ii*, begins executing *cpu-heavy* immediately and thus includes the CPU-to-GPU memory transfer overhead. We expect the performance of a Fusion-like architecture to lie somewhere between what is reflected in experiments *gpu-hash-i*, and *gpu-hash-ii* which represents the current state of the art.

Experiment	CPU version	GPU version	Relative difference
gpu-hash-i	50.0%	10.7%	78.5%
gpu-hash-ii	50.3%	24.8%	50.8%

Table 4: Runtime overhead on the *cpu-heavy* process.

As the results indicate, offloading the background process of hash computation to the GPU greatly reduces the load on the CPU. The lower overhead frees the main CPUs to better service the guest VMs, while the GPU facilitates a much more aggressive page sharing and thus efficient memory deduplication in the cloud.

6 Conclusion

With new heterogeneous multiprocessors on the horizon, we argue their potential to benefit various aspects of virtualization driven clouds. To summarize, incorporating GPU processing to the cloud infrastructure benefits the cloud service providers by accelerating hypervisor level management tasks, increasing the flexibility of the cloud infrastructure by facilitating improved VM migration and cloning, and providing better resource utilization. This enables higher server consolidation ratios and cost control by shutting down of idle servers and a possible decrease in the size of the data centres. The users experience improved VM performance and less usage cost due to better resource consolidation in the cloud hosts (pay-per-use cloud service model).

Our case study explores in detail the benefits of using heterogeneous hardware for memory deduplication in cloud via virtual machine page sharing. While we experiment with discrete graphics processors, we expect comparable results for on-chip graphics processing without the latency issues of a discrete processor.

We are working towards integrating our GPGPU driven memory hashing implementation as a scanning-based page sharing module on top of Satori's copy-on-write disk sharing implementation in Xen. In the future, we plan to validate our findings on a heterogeneous architecture like Fusion, as well as focus on the various other candidates for specialized hardware acceleration inside the cloud infrastructure. Finally, we will research mechanisms for managing heterogeneous hardware.

References

[1] Advanced Micro Devices. AMD Fusion Family of APUs Technology Overview. `http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf`, 2010.

[2] 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. In *8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[3] G. Hu, J. Ma, and B. Huang. High Throughput Implementation of MD5 Algorithm on GPU. In *International Conference on Ubiquitous Information Technologies Applications, ICUT*, 2009.

[4] M. Juric. Notes: CUDA MD5 Hashing Experiments. `http://majuric.org/software/cudamd5/`.

[5] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened Page Sharing. In *USENIX Annual Technical Conference*, 2009.

[6] NVIDIA. CUDA Toolkit 3.2. `http://developer.nvidia.com/object/cuda_3_2_downloads.html`.

[7] E. Seamans and E. Alexander. Fast Virus Signature Matching on the GPU. In *GPU Gems 3*, chapter 35. 2008.

[8] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating gpus for network packet signature matching. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[9] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *5th Symposium on Operating Systems Design and Implementation*, 2002.