Enabling Multi-GPU Support in gem5

May 30, 2020 • Bobbi W. Yogatama, Matthew D. Sinclair, Michael M. Swift

Introduction

In the past decade, GPUs have become an important resource for compute-intensive, general-purpose GPU applications such as machine learning, big data analysis, and large-scale simulations. In the future, with the explosion of machine learning and big data, application demands will keep increasing, resulting in more data and computation being pushed to GPUs. However, due to the slowing of Moore's Law and rising manufacturing costs, it is becoming more and more challenging to add compute resources into a single GPU device to improve its throughput. As a result, spreading work across multiple GPUs is popular in data-centric and scientific applications. For example, Facebook uses 8 GPUs per server in their recent machine learning platform.

However, research infrastructure has not kept pace with this trend: most GPU hardware simulators, including gem5, only support a single GPU. Thus, it is hard to study interference between GPUs, communication between GPUs, or work scheduling across GPUs. Our research group has been working to address this shortcoming by adding multi-GPU support to gem5. In this blog post, we discuss the changes that were needed, which included updating the emulated driver, GPU components, and coherence protocol.

gem5 AMD APU

The recent gem5 AMD APU model extends gem5 with an accurate, high fidelity GPU timing model that executes on top of ROCm (Radeon Open Compute Platform), AMD's framework for GPU-accelerated computing. Figure 1 shows the simulation flow when gem5 simulates a GPU. The application source is compiled by the HCC compiler, generating an application binary which comprises of both x86 code for the CPU and gcn3 code for the GPU that is loaded into the simulated memory. The compiled program invokes the ROCr runtime library which calls the ROCt user-space driver. This driver makes ioctl() system calls to the kernel fusion driver (ROCk), which is simulated in gem5 since the current GPU support uses syscall emulation (SE) mode.

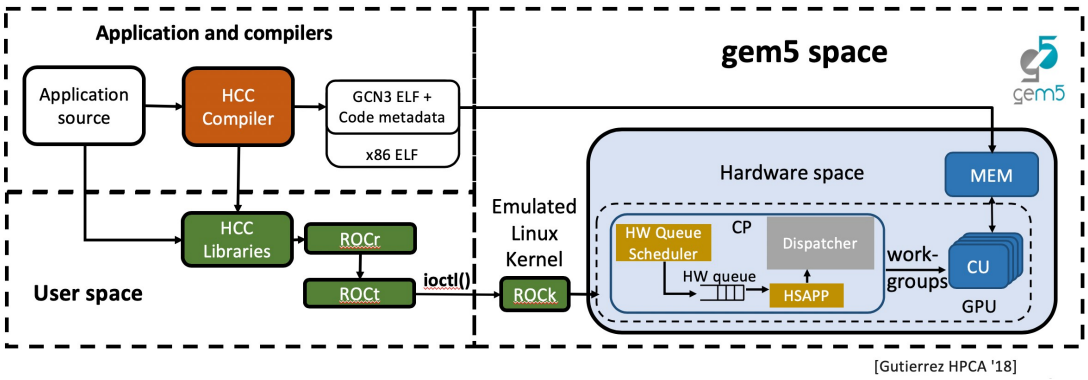

Figure 1. gem5 Simulation Flow

Multi-GPU Support in gem5

gem5
About
Publications
Contributing
Governance

Docs
Documentation
Old Documentation
Source

Help
Search
Mailing Lists
Website Source

The first step of supporting multi-GPU is to replicate GPU hardware components for each simulated GPU. Figure 2 shows the GPU components that we replicate. We opted to use a single driver across all of the GPUs since all of the GPUs share a single node. Accordingly, we added a loop in the configuration script (apu_se.py) and topology script (hsaTopology.py) to instantiate multiple GPU nodes that share a single emulated driver (ROCk). Each GPU has its own command processor, hardware scheduler, packet processor, dispatcher, and compute units. To ensure that each of the GPU components is distinguishable, we also added a unique GPU ID parameter to each component's object class. For components that operate on a specific address range (e.g., packet processor), we also assigned them unique address ranges to avoid overlapping. Finally, we replicated the cache and TLB hierarchy by changing the cache (GPU_VIPER.py) and TLB (GPUTLBConfig.py) configuration scripts.

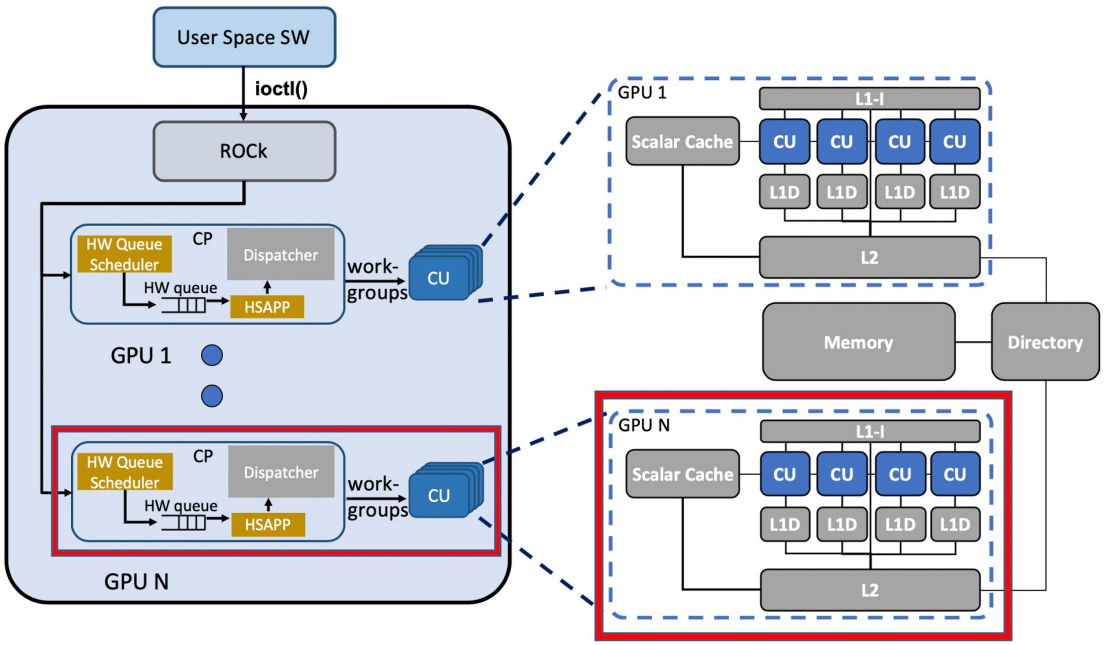

Figure 2. Replicated GPU Components

Adding Emulated Driver (ROCk) Support for Multi-GPU

Since the emulated ROCk in gem5 only supports a single GPU, we needed to add multi-GPU support to ROCk. This required two significant changes:

1. Managing software queues and sending work to multiple GPUs.
2. Mapping doorbells to multiple GPUs.

Managing Software Queues and Sending Work to Multiple GPUs.

Applications typically communicate with the GPU through one or more software queues. These software queues contain the application kernels that will be assigned to a GPU. Figure 3 shows how multiple applications interact with multiple GPUs through software queues: queues are private to an application, but an application may have more than one queue. Since gem5 emulates the ROCk kernel-space driver from Linux, this emulated driver creates and manages the software queues. To create a queue, the user space code makes an ioctl() system call to ROCk to request that a queue be created. ROCk obtains the GPU ID from the ioctl() arguments and assigns the kernel to the appropriate GPU as specified by the user. While the GPU ID is a parameter when creating a queue, it was not passed to routines that manipulate the queue. Thus, to further manage the queue when multiple GPUs are used, ROCk must know which GPU a queue serves. For example, when a program destroys a queue, ROCk must delete the shared state in the GPU that points to the queue. To resolve this problem, we added a hash table to ROCk to maintain a queue-to-GPU mapping. Thus, it is now simple and efficient for the emulated ROCk driver to identify the GPU associated with each queue.

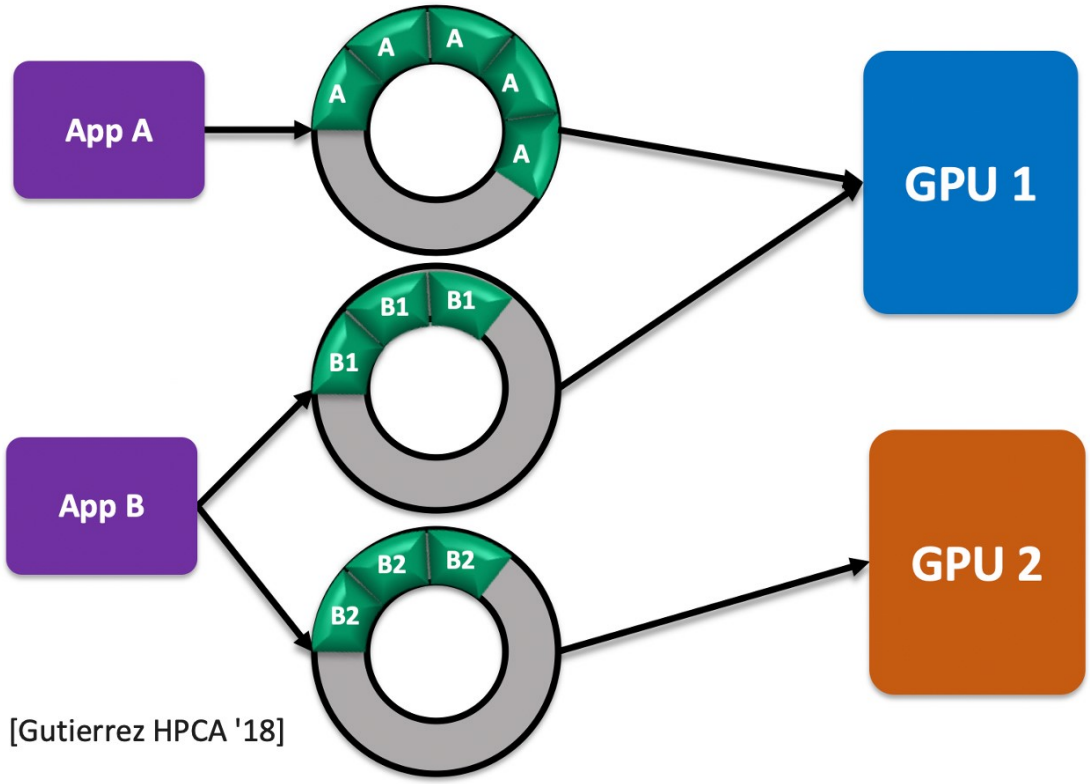

Figure 3. Applications to GPU Communication

Mapping Doorbell to Multiple GPUs.

GPUs use doorbells as a mechanism for user-space software to notify the GPU that there is work to be done. The software places data in mutually agreed-upon memory locations, and "rings the doorbell" by writing to a doorbell region. This act of "ringing the doorbell" notifies the GPU that there is some work ready to be processed. Since we are using multiple GPUs, we also need multiple doorbell regions, one for each GPU. Software accesses the doorbell regions by mapping the region into virtual memory with the mmap() system call.

However, the GPU identity was not visible to the mmap() system call, so the emulated ROCk driver did not know which GPU to map to a doorbell region to. We address this issue by encoding the GPU ID into the *offset* parameter passed to mmap(). The encoded offset is returned to user space from the create queue ioctl() call. Thus, the mmap() system call can decode the GPU ID from the offset parameter, and identify the associated GPU doorbell region used for mapping. This mechanism is illustrated in Figure 4.

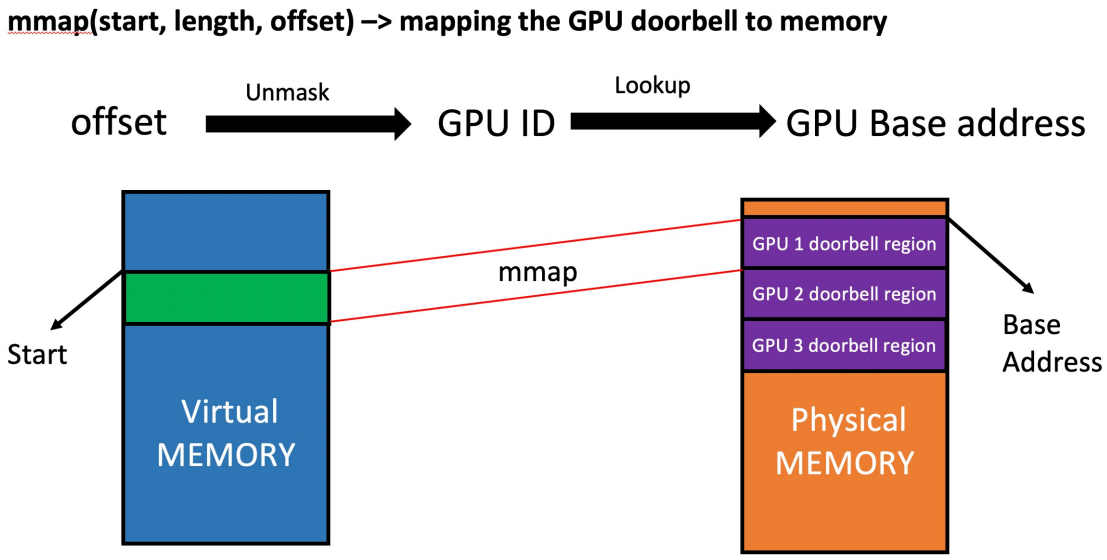

Figure 4. Mapping Doorbell Region for Multiple GPUs

Enabling Writeback Support in gem5 Coherence Protocol

Currently, the gem5 GPU coherence protocol uses a write-through (WT) approach for both L1 and L2 caches. Although this is a valid implementation, in multi-GPU systems it leads to significant bandwidth pressure on the directory and main memory. Moreover, modern AMD and NVIDIA GPUs generally have the GPU's last-level cache be writeback (WB) caches. Therefore, we decided to change the GPU's L2 cache in gem5 to be WB instead of WT. Although partial support already existed in gem5 for WB GPU L2 caches, it did not work correctly because the flush operations necessary for correct ordering were not being sent to the L2.

Figure 5 shows how our added GPU WB support compares to the current WT approach. In the WT version, all written data is propagated through the cache hierarchy and therefore is visible to the other cores shortly after the write occurs. However, our WB support holds dirty data in the GPU's L2 without notifying the directory (step 3). In this situation, the dirty data is not visible nor accessible to the other cores and directory. This is safe because the GPU memory consistency model assumes data race freedom. However, to ensure this dirty data is made visible to other cores by the next synchronization point (e.g., a store release or the end of the kernel), we modified the coherence implementation to forward the flush instructions, generated on store releases and the end of kernels, to the L2 cache (steps 4 and 5), which flushes all dirty L2 entries (step 6). This ensures that all dirty data is visible to the other cores and the directory at the end of every kernel, while also providing additional reuse opportunities and reducing pressure on the directory.

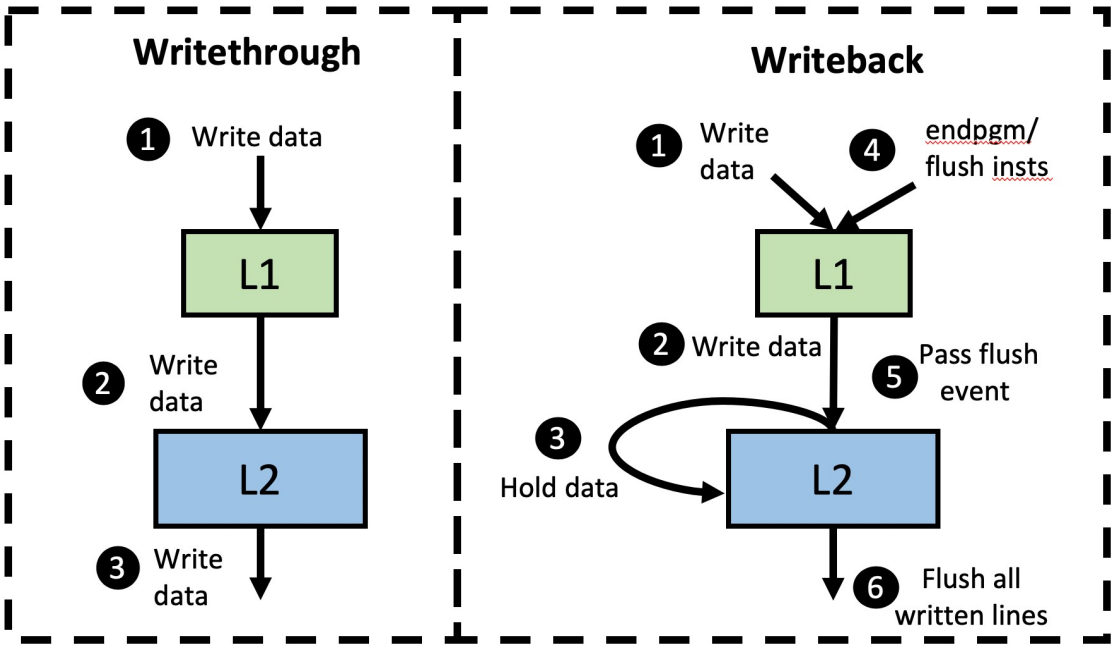

Figure 5. Writethrough vs Writeback Implementation

Conclusion

In the recent few years, multi-GPU systems have become increasingly common as more data and computation being pushed into GPU. However, up until now, gem5 only simulates a single GPU, which makes it difficult to study issues arising in multi-GPU system. Our research group attempts to address this issue by extending multi-GPU support in gem5 by (1) Replicating the GPU components, (2) Modifying the emulated driver, and (3) Enabling writeback support in the coherence protocol.

Workshop Presentation

For additional details about our changes and some experimental results, check out our workshop presentation!


Enabling Multi-GPU Support in gem5