Modeling Modern GPU Applications in gem5

May 27, 2020 • Kyle Roarty and Matthew D. Sinclair

In 2018, AMD added support for an updated gem5 GPU model based on their GCN3 architecture. Having a high-fidelity GPU model allows for more accurate research into optimizing modern GPU applications. However, the complexity of getting the necessary libraries and drivers, needed for this model to run GPU applications in gem5, made it difficult to use. This post describes the work we have done with increasing the usability of the GPU model by simplifying the setup process, extending the types of applications that can be run, and optimizing parts of the software stack used by the GPU model.

Running the GPU model

To provide accurate, high fidelity simulation, the AMD GPU model directly interfaces with the Radeon Open compute platform (ROCm) driver. Although gem5 can simulate the entire system (full system mode, or FS mode), including devices and an operating system, currently the AMD GPU model uses the syscall emulation (SE) mode. SE mode only simulates user-space execution and provides system services (e.g., malloc) in the simulator instead of executing kernel-space code. As a result, the only portion of the ROCm software stack that must be emulated is the KFD (Kernel Fusion Driver). Thus, in order to use the AMD GPU model, the user must first install ROCm on their machine.

This presents a challenge, because gem5's GPU model supports a specific version of ROCm (version 1.6) and getting the drivers installed and interacting properly with gem5 is difficult. Moreover, to run modern applications such as machine learning (ML), also referred to as machine intelligence (MI), applications, additional libraries (e.g., MIOpen, MIOpenGEMM, rocBLAS, and hipBLAS) need to be installed. However, the versions of those libraries must be compatible with ROCm version 1.6. Overall, figuring out the exact software versions and installing them is time consuming, error-prone, and creates a barrier to entry that discourages users from using the GPU model.

To help address this issue, we have created and validated a Docker image that contains the proper software and libraries needed to run the GPU model in gem5. With this container, users can run the gem5 GPU model, as well as build the ROCm applications that they want to run in the GPU model. This Docker container has been integrated in the public gem5 repository, and we intend to use the image for continuous integration on the GPU model. Furthermore, since the AMD GPU model currently models a tightly-coupled CPU-GPU system with a unified address space and coherent caches, this Docker also includes the changes necessary to HIP and MIOpen to remove discrete GPU copies in these libraries wherever possible.

Using the Docker image

The Dockerfile and an associated README are located at `util/dockerfiles/gcn-gpu`. This documentation can also be found at the GCN3 page of the gem5 website. Finally, we have also created a video demonstration of using the Docker in our gem5 workshop presentation. Next, we briefly summarize how to use the docker image.

Building the image

```
cd util/dockerfiles/gcn-gpu
docker build -t <image_name> .
```

To build gem5 in a container, the following command could be used: (Assuming the image is built as gem5-gcn)

```
docker run --rm -v /path/to/gem5:/gem5 -w /gem5 gem5-gcn scons -sQ -j$(nproc) build/GCN3_X86/gem5.opt
```

Optimizing the software stack for MI workloads

Creating the Docker image makes it easy to run HIP applications in gem5. However, running modern applications such as MI applications is more complex and required additional changes. Largely, these issues stemmed from the MI libraries utilizing features that were not designed with simulation in mind.

MIOpen is an open-source MI library designed to execute on AMD GPUs. MIOpen has HIP and OpenCL backends and implements optimized assembly kernels for many common DNN algorithms. It chooses which of these backends to use at compile time. Then, at runtime, MIOpen will use the appropriate backend to execute a given MI application on an AMD GPU. Although this support works well for real GPUs, simulating which backend to use, which GPU kernel to run, and the configuration of the data it's looking to operate on is time consuming and not part of the region of interest for simulation.

For example, MIOpen calls the backend to search for an appropriate kernel which is optimized for the given parameters. On real hardware, this process runs multiple different kernel options, then picking the fastest one and compiling it using clang-ocl. As part of this process, MIOpen caches the kernel binary locally, for subsequent uses of the same kernel. Since online compilation is computationally intensive and currently unsupported in gem5, we bypass online kernel compilation in gem5 by running the applications on a real GPU beforehand to obtain MIOpen's cached kernel binaries. Alternatively, if an AMD GPU is not available, it is also possible to compile the necessary kernels on the command line with clang-ocl.

Moreover, GEMM kernels are extremely common in MI applications. For these kernels, MIOpen uses MIOpenGEMM to identify and create the best kernel for the parameters of the inputted matrices. Unfortunately, MIOpenGEMM does this by dynamically creating a database of possible GEMM kernels and then selecting the kernel that best matches the application's matrices. Since this happens dynamically, every time a program is run, it is difficult to bypass this process. Thus, to avoid the overhead of simulating this process, we backported support from newer versions of ROCm that allowed MIOpen to use rocBLAS instead of MIOpenGEMM. Using rocBLAS instead of MIOpenGEMM removes the repeated, dynamic database creation from the critical path in simulation, since rocBLAS generates the database of optimal solutions on installation.

Overall, these changes avoid simulating work that is not part of the application's region of interest and enabled us to simulate a number of native MI applications in gem5.

What's next?

Our work has increased the usability of the gem5 GPU model, and shown how to run a variety of GPU applications, including native MI applications, in gem5. As mentioned above, we are currently in the process of integrating the Docker into the develop branch of gem5, to enable continuous integration testing on future GPU commits. Moving forward, we hope that this work can serve as a springboard to running high-level frameworks such as Caffe, TensorFlow, and PyTorch in the simulator. However, since high-level frameworks have large models and significant runtimes, to make simulating those easier to use, we plan on extending checkpointing support to include the GPU model, allowing us to focus on simulating potential regions of interest.

Workshop Presentation


Modeling Modern GPU Applications in gem5