15-418 Project Proposal: GSim

Group: Prithvi Cherabuddi (pcherabu) & Shashwat Gupta (shashwag)

Project Webpage: https://pcherabu.github.io/gsim.github.io/

Summary:

We are going to implement an execution driven, cycle-approximate GPU simulator modeled loosely after Nvidia's 9xx GPU series. The simulator aims to be accurate down to the micro-architectural level; however, by no means correlated. Furthermore, the simulator will have support for architectural and micro-architectural level statistics; these statistics can also have mapping/histogram versions to gauge different phases of program behavior.

Background:

A simulator provides the opportunity to test out performance proposals in a cheap, quick manner before going to RTL or testing them out in a limited scope (soft cores on FPGA's, cycle-accurate emulations, etcetera). However, simulators also provide another benefit: analytical support. A simulator, especially one that models u-Arch flows, provides statistical analysis of the entire simulation environment (the workload and the hardware that runs this workload). While it won't exactly give a workload characterization, it helps dynamically analyze the workload by observing simulated hardware statistics collected during the run.

Thus, we do not aim to utilize our simulator in observing different GPU performance optimizations but rather help simulate workloads and characterize their bottlenecks by utilizing our simulator's analytical capabilities. Our simulator is execution driven and not trace-driven, so ideally it will be possible to run on compiled CUDA code on it and in turn receive an accurate output. The statistics will be dumped in a separate statistics file, all statistics for whose histogram capability is turned on (for example, count the number of times we went to GPU Device Memory every 100 cycles/100 graduated instructions) will save graphs (using C++ gnu plots), and all statistics whose mapping capability (to measure populations, queue healths, etcetera) will be saved in a separate graph sheet.

By utilizing these statistics (which will also record latencies, memory subsystem effective bandwidth, execution statistics per core, caching, etc.), the user will be able to not only compute the overall Arithmetic Intensity of the workload but also understand the reason why it is not as high as it is expected to be. This allows optimization of the workload itself (especially since the user will be able to identify workload bottlenecks) and when helps develop performing CUDA code. Furthermore, if the project were to be extended in the future and proper hardware correlation is done, GSim can be used as a model to develop GPU performance proposals as well (or test out new scheduling algorithms, memory subsystem organizations, etcetera).

The Challenge: Developing any simulator in a limited time-constraint may end up limiting the scope of the project itself. Nevertheless, the simulator structure is straightforward (at least the execution side of things, simulating the memory subsystem would cause problems in correlation and might just end up as adding fixed determined latencies). However, parsing compiled CUDA code and developing the front end of the simulator will be the highest bottleneck in the project development.

Resources:

The simulator will be built from scratch; nevertheless, if we transform our simulator into a trace-driven functional simulator it will leverage existing instrumentation tools (such as nvbit) to still include the proposed analytical cabalities.

Goals and Deliverables

Plan to Achieve:

- ➔ Develop the simulator's front end that will be able to parse/decode compiled CUDA code.
- ➔ From there, build up the simulator's front end (models the GPU pipeline until the scheduler)
- ➔ Develop the back-end cores and their L1 caches
- ➔ Develop shared high levels of cache and Device Memory

Hope to Achieve:

- ➔ Accurate L2 and Device Memory models (probably won't happen and we'll end up having fixed latencies associated with instructions).
- ➔ Develop Memory images to accurately model Device Memory
- ➔ Model CPU-GPU communications

Plausible Demos:

- ➔ Ask any student to come up with example CUDA code and simple compile and run on our simulator. Our simulator will then spit out statistics and histograms that will pinpoint the reasons for the code's Arithmetic Intensity.

Platform Choice: Linux environments, source code in C++, will have python code to interface with C++ source modules for user input/parsing support/ etc.

Rough Schedule:

Week 1: Develop Front-end parser that can take in compiled CUDA code

Week 2: Built up a basic front-end

Week 3: Develop Scheduler and start on the execution model of a single-core

Week 4: Develop L1 caches and refill/miss/hit flows (this requires going back to the front-end)

Week 5: Draft a rough L2/GPU Device Memory model that takes a lot of characteristics of our L1 cache model to provide an inaccurate yet complete memory subsystem.