Enabling Reproducible and Agile Full-System Simulation

Bobby R. Bruce*, Ayaz Akram*, Hoa Nguyen*, Kyle Roarty†, Mahyar Samani*, Marjan Friborz*, Trivikram Reddy*, Matthew D. Sinclair†‡, and Jason Lowe-Power*

*University of California, Davis
Davis, California
{bbruce, yazakram, hoanguyen, msamani, mfariborz, tvreddy, jlowepower}@ucdavis.edu

†University of Wisconsin, Madison
Madison, Wisconsin
kroarty@wisc.edu, sinclair@cs.wisc.edu

‡AMD Research

***Abstract*—Running experiments in modern computer architecture simulators can be a difficult and error-prone endeavor. Users must track many configurations, components and outputs between simulation runs. The gem5 simulator is no exception to this, requiring researchers to gather, organize, and create a significant number of components for a single simulation.**

In this paper, we present the GEM5ART framework, a tool to aid gem5 users in better structuring and running architecture simulations, and GEM5 RESOURCES, a suite of resources with known compatibility with the simulator. These new additions to the gem5 project make full system simulation easier, allowing researchers to concentrate more so on their architectural innovations over setting up the simulation framework. The GEM5ART framework carefully logs the resources used in a gem5 simulation and places the results obtained within a database, thus enabling simple reproduction of experiments. The pre-built resources allow researchers to jump straight into running simulations rather than having to spend valuable time creating them. GEM5ART has been released with a permissive, open source license allowing the broader computer architecture community to contribute as workloads and workflows evolve.

An archive of the data, an related materials, presented in this paper can be found at https://doi.org/10.6084/m9.figshare.14176802.

***Index Terms*—computer architecture, simulation**

I. Introduction

As Moore's law comes to its end, there is a growing understanding that future gains in computer performance will come from new architectural designs. For this to materialize, more sophisticated tooling is required, particularly in the domain of architecture simulators. However, the complexity of these tools is growing alongside their sophistication. At present, this complexity already burdens end users, leading to frustration and occasional errors in experimental results. We need a solution to help reign in this complexity to enable future architectural innovations.

The burden of complexity is particularly noticeable in *full-system* simulators, such as gem5 [1], [2].[1] Full-system simulators have enough fidelity to boot (mostly) unmodified operating systems, emulate I/O devices, and execute unmodified applications. Today's full system simulators are unlike prior simulators (e.g., SimpleScalar [3]) which required relatively little setup where the only inputs were the simulator binary and a statically compiled executable. Today's full system simulators, like gem5, require many more inputs, such as an OS kernel, benchmarks (which themselves depend on a compiler, runtimes, and more), static and dynamic parameters, a disk image, etc. Figure 1 shows an example of this complex workflow.

It is important to use up-to-date versions of all items utilized in any experiment as runtimes, compilers, kernels, and other components are changing frequently. It is also important to record the exact versions used and, preferably, compare how new versions of these components impact performance. This is particularly important as computer architecture research increasingly requires cross-stack studies that investigate how changes in any and all levels, from hardware, through the kernel, wider OS, and up to the level of applications, impact the computer systems. These studies require methods to easily track and update each of the resources required for simulation.

In this paper, we focus on providing support for the gem5 architectural simulator; one of the most popular simulators currently used in academia and industry. While some other simulators come pre-packaged with default designs and configurations, gem5 focuses on flexibility. This makes it powerful, but puts the onus on the user to provide and configure components themselves. It is a hurdle to initiating architecture simulations, and the iterative nature of research means keeping

[1] http://www.gem5.org

track of each gem5 run's unique setup; a non-trivial task. Tracking all information (e.g., the gem5 version, the hardware configuration, the boot configuration, the benchmark, etc.) used to produce each output is error prone, and ripe for automation. It is for this reason we developed GEM5ART.

The gem5 **A**rtifact, **R**eproducibility, and **T**esting framework, which we refer to as "GEM5ART", is a set of Python libraries which can be used to run gem5 experiments in a clear and structured manner, and facilitates the storing of results and configuration information in a database for future reference and reproduction. Our overarching goal with GEM5ART is to *improve the reproducibility and agility of full-system simulation*, which should ultimately increase end-user productivity. The GEM5ART framework is open source and distributed via the Python Packaging Index (PyPI)[2]. This enables simple installation with `pip` the Python package installer. GEM5ART is an open source, freely available project[3].

The GEM5ART framework is only one component for performing full system simulation. Users must obtain all of the resources required for their experiments. For instance, a benchmark suite, such as PARSEC [4], must be downloaded, compiled, and then stored in an appropriate disk image before being loaded in a gem5 configuration. In an effort to reduce the complexity of obtaining such components, we have developed GEM5 RESOURCES, a repository containing commonly used gem5 simulation components such as kernels, tests, and benchmarks. While neither is dependent on another, we believe GEM5ART and GEM5 RESOURCES function best when working in tandem, with GEM5 RESOURCES providing commonly used components and GEM5ART recording which of these components are used, and the results obtained from gem5 experiments.

GEM5ART enforces a well-documented protocol for running gem5 experiments. This documented protocol allows the results of gem5 experiments to be portable and reproducible. By storing all inputs and results in a database, these artifacts may be made as public as the end-user desires, and freely available tools may be used to process this data to create rich data visualizations.

The main contribution of this paper is the GEM5ART tool which simplifies running gem5 experiments and GEM5 RESOURCES which contains 16 benchmark suites, tested OS kernels, applications, as well as a selection of gem5 specific tests. Both of these contributions are open source and under active development. Similar to a new benchmark suite, this work *enables new computer architecture research*.

To demonstrate the efficacy of GEM5ART, we present three use cases. (1) We show the importance of using up-to-date operating systems by comparing the results from the PARSEC benchmark suite on two different long-term service releases of Ubuntu. (2) We show the flexibility and ease of use of GEM5ART by testing an extensive cross product of system configurations and Linux kernels. The results of this test can help the gem5 developers concentrate their effort on bugs in gem5 which affect only a specific configuration or workload. (3) We show how the performance of different GPU applications is affected by the use of different register allocators.

[2]https://pypi.org/project/pip

[3]Distributed within the gem5 source repository: https://gem5.googlesource.com/public/gem5.

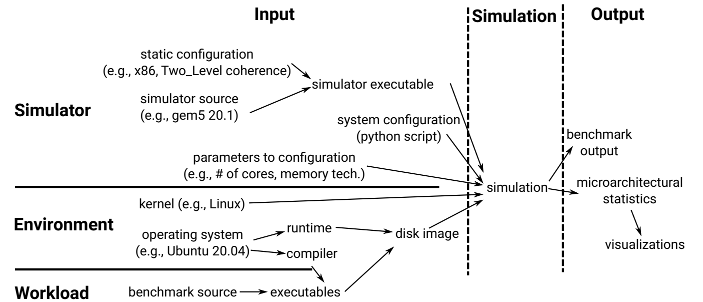

Fig. 1: An example workflow for full-system simulation with gem5.

We outline how gem5 works and is used by researchers in Section II. We discuss any related work in Section III, then the technical design of GEM5ART and how it addresses the pain-points present in the usual workflow in Section IV. In Section V we describe GEM5 RESOURCES, and, in Section VI, we demonstrate GEM5ART and GEM5 RESOURCES on three use-cases. In Section VII we conclude the paper and discuss the potential impact of our work.

II. BACKGROUND

To understand why GEM5ART is necessary, we should understand the current workflow of gem5 and the various components and resources needed to run a simulation.

A typical gem5 full-system workflow is shown in Figure 1. The user compiles the *gem5 source* with a *static configuration* (e.g., targeting the x86 ISA with a two level cache hierarchy) which then generates the *gem5 simulator executable*. From this the user compiles a *kernel binary* given the *Linux kernel source code* and a *kernel configuration file*. An *operating system* is then installed on a *disk image*, and a *benchmark* is compiled from a source on the disk image. The simulation can then be run with a given *system configuration* in the form of a Python script. A simulation will produce the *benchmark output* and a variety of *microarchitectual statistics* for evaluation of the configuration on the benchmark.

As is shown in this example, there are many different components required or created for a single run of gem5. The gem5 executable, for example, is compiled from the gem5 source via the project's git repository[4]. This executable will vary between versions of gem5, which are released roughly three times a year. Furthermore, gem5 may be compiled with different static configurations to simulate different ISAs and evaluate different cache hierarchy setups. It is therefore important to keep track of these components, and correctly document then link them to a particular simulation run. It is for this reason GEM5ART was created.

[4]https://gem5.googlesource.com

There is also the question of where a user is to obtain additional resources, such as benchmarks. We provide these in an "out-of-the-box" manner via GEM5 RESOURCES; a set of freely-available resources with known compatibility with gem5. GEM5 RESOURCES is discussed in more detail in Section V.

III. Related Work

Few of the existing frameworks available to perform experiments in a systematic way are compatible with complex simulators such as gem5. FEX [5], Occam [6], and Collective knowledge [7], for example, are far too general to incorporate into the gem5 simulation workflow. FEX [5] is a Docker container based evaluation framework for general software systems which allows reproducible experiments and customized workflows. Occam [6] is an active curation platform, which allows sharing of artifacts and workflows. It allows to artifacts to be complex tools/software like architectural simulators. Collective knowledge [7] is a cross-platform framework to automate repetitive tasks, but mainly focuses on machine learning frameworks. In comparison to these tools, GEM5ART is a customized framework to run experiments with gem5 and as a result requires less effort by the user to achieve the same goals in comparison to other tools. Moreover, GEM5ART is focused more on documenting the experiments to enable reproducibility and less on automation in contrast to the previously mentioned tools. Another important and novel contribution is GEM5 RESOURCES which provides a wide set of benchmarks, tests and other required dependencies ready to be used with gem5.

There also exist some gem5 specific tools or methodologies [8], [9], which do not have the same goals as GEM5ART, but regulate gem5 usage in some way. Walker et al. [8] proposed a methodology for finding sources of error in CPU performance models and suggested a validation methodology based on clustering and correlation analysis. DiagSim, proposed by Jo et al. [9], is a tool to diagnose hidden details (which can have major effect on simulation results) of different simulators including gem5. GEM5ART is not a tool to find inaccuracies in gem5 but provides necessary infrastructure to bring structured approach to gem5 validation experiments which otherwise can be very hard to manage (evident by our own experience with such studies).

IV. GEM5ART

The high-level goal of GEM5ART is to provide a structured and documented protocol for conducting computer architecture experiments. The GEM5ART framework requires researchers to document every input required for a particular experiment, which increases the reproducibility and understandability of these experiments.

A. Components of the Framework

GEM5ART is composed of three interrelated Python packages.

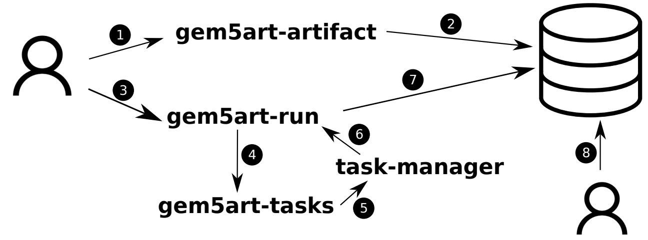

Fig. 2: User interaction with GEM5ART

a) Artifacts: Artifacts are the objects and/or components used in a gem5 run, or produced via a gem5 execution. Examples include the gem5 binary, the gem5 configuration files, a GEM5 RESOURCES resource utilized, the statistics output, and anything else which may vary between executions.

b) Runs: Within the GEM5ART library, a *run* object is a special artifact which contains all the information, and references all the main artifacts, required to execute a single gem5 experiment.

c) Tasks: A task is a gem5 job. Tasks are generated from the run objects and execute using an external job scheduler (e.g. `Celery` [10]) or a process management library (e.g. `Python multiprocessing` library).

Overall, GEM5ART interacts with the following external components: `Celery` [10], a job scheduler; `Python multiprocessing` library, a simpler task management library; `MongoDB` [11], a NoSQL database we use to store gem5 run information and their corresponding artifacts; and `Packer` [12], a tool to automate the process of disk creation.

Figure 2 depicts how a user would interact with GEM5ART to run their experiment. The user first registers all artifacts using GEM5ART (❶), any associated files are stored in the database as well (❷). The GEM5ART run objects are then created using GEM5ART run library (❸), passed to GEM5ART tasks library (❹) and executed using an external task manager (❺). Finally, results are stored in the database as well (❻ and ❼). The user can query the database at any time to access artifacts and results (❽).

B. Artifacts

The first stage of a user interacting with GEM5ART is via the *artifact* module. Through this module, the user registers the created artifacts of all the components needed to run an experiment. Since, some artifacts serve as input to other artifacts (for example, Linux kernel source repository artifact is an input to the Linux kernel binary artifact), the *artifact* module is also used to declare the dependencies that exist among different artifacts. Figure 3 shows an example of how a user can register a gem5 binary as an artifact. In this example six attributes are specified:

- **command**: The command which must be executed to create the resource. In this case, the gem5 binary. Note the checkout at a specific revision. This ensures anyone else using this artifact will obtain the the correct version of gem5,

```
gem5_binary = Artifact.registerArtifact(
    command = '''cd gem5;
    git checkout 440f0bc579fb8b10da7181;
    scons build/X86/gem5.opt -j8
    ''',
    typ = 'gem5 binary',
    name = 'gem5',
    cwd = 'gem5/'
    path = 'gem5/build/X86/gem5.opt',
    inputs = [gem5_repo,],
    documentation = 'gem5 binary...')
```

Fig. 3: Example registration of an Artifact

- **typ**: The artifact type. In this case, a gem5 binary.
- **name**: The name of the resource.
- **cwd**: The directory in which the **command** should be run.
- **path**: The path of the artifact (gem5 binary).
- **inputs**: Other resources dependency. In this example, the gem5 repository artifact must already be present before the gem5 binary artifact may be obtained.
- **documentation**: The artifact's documentation to store any user specified useful information about the artifact.

These attributes are mainly used for documentation purposes and should provide enough information to reproduce the experiment at a later time. Via the GEM5ART artifacts library, this information is uploaded to the database, along with the following which are automatically generated by GEM5ART:

- **hash**: An MD5 hash of the resources or the git revision hash if the artifact is a repository.
- **id**: A unique ID (UUID) generated for each artifact.
- **git**: A dictionary containing two keys: `git url` and `hash`, which values specify the git repo and the revision hash of the artifact. If the target attribute is not a git repository, the dictionary is left blank. As most of the artifacts used in any gem5 simulation are source code repositories (version controlled by `git`), we leverage `git url` and `hash` to indirectly store useful information about the artifact version. This, in turn, also allows us to easily communicate the status of an artifact used in an experiment (via `git hash`) to others who do not have access to the user's database.

These generated attributes are used by the GEM5ART artifact library to maintain a uniqueness of the resource in the database. The artifact is assumed to be present at the user-specified **path**. If there is any file associated with the artifact, that is stored in the database as well unless it already exists there. The **hash** attribute is used as a safety net to ensure a resource has not been altered between runs. If this changes, even if all other attributes remain the same, a new artifact is generated. Duplicate artifacts are not permitted in the database. This allows a user to see clearly which objects were used in that gem5 run.

```
def createFSRun(cls,
    gem5_binary: str,
    run_script: str,
    output: str,
    gem5_artifact: Artifact,
    gem5_git_artifact: Artifact,
    run_script_git_artifact: Artifact,
    linux_binary: str,
    disk_image: str,
    linux_binary_artifact: Artifact,
    dist_image_artifact: Artifact,
    *params: str,
    timeout: int = 60*15) -> 'gem5Run':
```

Fig. 4: GEM5ART run method for full-system simulation

C. Runs

Once the artifacts have been specified, the user then creates the run objects using the GEM5ART *run* library. The "GEM5ART run" object is a special artifact which stores all the information about a run and a pointer to its results. The "GEM5ART run" also references the GEM5ART artifacts used in a gem5 run.

Figure 4 shows an example of a function, provided by the run library to create a gem5 full-system run object. To run a full-system simulation the user must provide a gem5 binary, a kernel, a disk image, and a gem5 run script. These are all provided as parameters to the `createFSRun` method, so that the eventual gem5 run command can be constructed and executed by GEM5ART. The previously registered artifacts needed for the gem5 run are passed to this function as well. For instance, `gem5_artifact` for the gem5 binary, `gem5_git_artifact` for its repository, `run_script_git_artifact` for the gem5 run script, `linux_binary_artifact` for the Linux kernel binary, and `disk_image_artifact` for the disk image as shown in Figure 4. Any other arguments needed for the gem5 run script are passed as parameters to the function in Figure 4, and a timeout is provided (after which the gem5 job is terminated by GEM5ART if not already finished). `gem5_binary`, `run_script`, `linux_binary`, and `disk_image` in Figure 4 specify the location of each artifact in the host system, and `output` specifies the output directory. It should be noted that all of this information fed to the `createFSRun` method specifies one unique experiment (a single data point).

The results of an experiment are archived as an artifact inside the database instance. GEM5ART also stores a summary of useful information (like run status and execution time) in the database. The database can then be queried to access this information, and generate plots to visualize results for further analysis. Additionally, by tracking all of the artifacts used for each gem5 execution, any resources (disk images, kernels, results, etc.) related to a particular gem5 run can be recovered for reproduction purposes.

D. Executing Tasks

The run object, such as that created via the `createFSRun` function in Figure 4, is then passed to the GEM5ART

```
gem5_git_repo = registerArtifact('gem5/')
gem5_binary = registerArtifact(
    gem5_git_repo,
    'build/X86/gem5.opt')
linux_git_repo = registerArtifact(
    'linux-stable/')
vmlinux_binary = registerArtifact(
    linux_git_repo,
    'linux-stable/vmlinux')
parsec_repo = registerArtifact(
    'parsec/')
disk_image = registerArtifact(
    parsec_repo,
    'disks/parsec.img')

function main():
    cpus = ['kvm', 'simple']
    benchmarks = ['blackscholes', ...]
    ...
    for each combination P\
        in [cpus, benchmarks, ...]:
        run_object = gem5Run.createFSRun(
            artifacts, P)
        apply_async(run_object)
```

Fig. 5: A typical GEM5ART full-system experiment involves gem5 artifacts, Linux kernel artifacts, disk image artifacts, and the experiment artifacts. After declaring the artifacts, the main function asynchronously launches the cross product of all parameters.

tasks library. This library uses `Celery`, or `python multiprocessing` library, or no job scheduler at all, to run a gem5 job. There is no limit to how many tasks may be passed to `Celery` or the `python multiprocessing library`. They will run the tasks in accordance to the job objects given, and schedule them as the host system allows. `Celery` is more complicated (but feature rich) job manager in comparison to the `python multiprocessing` library and may be used to manage tasks over multiple machines if needed. The GEM5ART task package can be extended to other job schedulers and distributed computing environments (e.g., Condor) in the future.

E. An end-to-end example

The entire user interaction with GEM5ART takes place via Python scripts called `launch scripts`. Figure 5 provides an example of such a script.

This example `launch script` is responsible for running PARSEC benchmarks using different gem5 configurations. The first portion of this script registers all the artifacts needed to run PARSEC experiments in this example `gem5_git_repo`, `gem5_binary`, `linux_git_repo`, `vmlinux_binary`, `parsec_repo`, and the `disk_image`. In the second portion of this script (inside the `main()` function), the GEM5ART run objects are created for each combination of a PARSEC benchmark and a gem5 configuration (for example different CPU models). These runs are setup to be executed asynchronously in the last 3 lines of this script.

Through this one Python script, the entire experiment and the details required to run the experiment are documented in one place. This script, in addition to the database, can be used to communicate to others (e.g., in a reproducibility report) all necessary inputs, how they were obtained, and how they were run for a particular experiment.

V. GEM5 RESOURCES

A key contribution of this work is providing a set of known-good resources for simulating workloads. While GEM5ART will keep track of artifacts for a particular run of gem5, there is the question of where someone using gem5 obtains these artifacts. For example, to evaluate the performance of a new and novel architectural design, a researcher will likely use a benchmark suite for evaluation. Previously it was the responsibility of the researcher to obtain for themselves this benchmark suite (and whatever else they required).

GEM5 RESOURCES[5] contains components which are not strictly needed to build and run gem5 but may be utilized in the running of a gem5 simulation. At present, GEM5 RESOURCES contains disk images pre-loaded with commonly used benchmark suites, scripts to run these benchmarks, kernels, and tests. GEM5 RESOURCES is under continual development and expansion, and will remain compatible with the latest gem5 changes. Table I contains a list of the resources presently available in GEM5 RESOURCES. We demonstrate using a selection of these resources in Section VI.

Each resources in GEM5 RESOURCES provides the original source code so researchers can understand how each was constructed and reproduce the pre-build resource if required. In keeping with gem5's free and open-source ethos, these resources may be modified to meet the needs of experimenters. We also encourage contributions of new benchmarks, useful tests, etc. to help expand the set of gem5 compatible materials which may be useful to others.

When providing a disk image GEM5 RESOURCES utilizes Packer[6]. Packer is an open-source disk image building tool available on most modern desktop operating systems to build disk images for various Linux distributions. Packer is capable of automating the disk image building process given necessary inputs. For each provided disk image GEM5 RESOURCES provides such inputs: the corresponding Packer script, a Ubuntu preseed configuration, a benchmark installation script and other resources required for building the desired benchmarks. The Packer scripts do not only provide necessary documentation to reproduce the disk image, but they also serve as a simple template to facilitate the contributions of new full system benchmarks.

For proprietary benchmarks, we don't distribute the disk images, but we do distribute all of the scripts needed to build the disk images. For instance, if the user has a SPEC license with a disk image (.iso) file they can execute the scripts provided by GEM5 RESOURCES and the disk image will be created.

[5]https://gem5.googlesource.com/public/gem5-resources
[6]https://www.packer.io

TABLE I: The GEM5 RESOURCES

Name	Type	Description
boot-exit	Benchmark / Test	A collection of gem5 scripts and binaries capable of completing and exiting the booting process of a Linux kernel with a Ubuntu 18.04 Server user-land with gem5 full system mode (FS mode). The documentation details the creation process of the Linux kernel and the disk image, as well as providing pre-made binaries. This resource serves as a test suite for gem5 FS mode.
gapbs	Benchmark	A collection of gem5 scripts, binaries, and documentation that are capable of running GAP Benchmark Suite (GABPS) with a Linux kernel and a Ubuntu 18.04 Server user-land with gem5 full system mode. This resource also details the creation process of the Linux kernel and the disk image, as well as providing pre-made binaries.
hack-back	Benchmark	A collection of gem5 scripts and binaries capable of creating a checkpoint after the booting process and then executing a script provided by the host in a gem5 full system simulation. The documentation details the creation process of the Linux kernel and the disk image containing a Ubuntu 18.04 Server user-land, as well as providing pre-made binaries.
linux-kernel	Kernel	A set of Linux kernel configurations and documentation of compiling a Linux kernel.
npb	Benchmark	A collection of gem5 scripts, binaries, and documentation that are capable of running NAS Parallel Benchmark (NPB) with a Linux kernel and a Ubuntu 18.04 Server user-land with gem5 full system mode. The documentation details the creation process of the Linux kernel and the disk image, as well as providing pre-made binaries.
parsec	Benchmark	A collection of gem5 scripts, binaries, and documentation that are capable of running Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suite with a Linux kernel and a Ubuntu 18.04 Server user-land with gem5 full system mode. The documentation details the creation process of the Linux kernel and the disk image, as well as providing pre-made binaries.
riscv-fs	Test	A collection of gem5 scripts, and documentation to build riscv bbl (berkeley boot loader) with linux kernel payload and a disk image to run full system simulation for a riscv target.
spec-2006	Benchmark	A collection of gem5 scripts, binaries, and documentation that are capable of running SPEC CPU 2006 benchmark suite with a Linux kernel and a Ubuntu 18.04 Server user-land with gem5 full system mode. The documentation details the creation process of the Linux kernel and the disk image. Licensing forbids us from providing pre-made disk images.
spec-2017	Benchmark	A collection of gem5 scripts, binaries, and documentation that are capable of running SPEC CPU 2017 benchmark suite with a Linux kernel and a Ubuntu 18.04 Server user-land with gem5 full system mode. The documentation details the creation process of the Linux kernel and the disk image. Licensing forbids us from providing pre-made disk images.
GCN-docker	Environment	A docker image with ROCm 1.6 and GCC 5.4 installed to simulate GPU applications on AMD GCN3 simulated GPUs. These applications include workloads such as DNNMark, HACC, HIP sample applications, HeteroSync, LULESH, and PENNANT. This docker image is used to build and run the GCN3_X86 gem5 variant.
HeteroSync [13]	Benchmark	A benchmark suite used to test the performance of various types of fine-grained synchronizations on tightly-coupled GPUs. This resource works with the GCN3_X86 gem5 variant.
DNNMark [14]	Benchmark	A benchmark framework used to characterize the performance of primitive deep neutral network workloads. This resource works with the GCN3_X86 gem5 variant.
halo-finder [15]	Application	Part of the HACC code base, a DoE application designed to simulate the evolution of the universe. The halo-finder code can be GPU accelerated. This resource works with the GCN3_X86 gem5 variant.
Pennant [16]	Application	A GPU application designed for advanced architecture research. This resource works with the GCN3_X86 gem5 variant.
LULESH [17], [18]	Application	A DOE proxy application that is used as an example of hydrodynamics modeling. This resource works with the GCN3_X86 gem5 variant.
hip-samples	Application	A set of applications that introduce various GPU programming concepts usable in ROCm HIP. This resource works with the GCN3_X86 gem5 variant.
gem5 tests	Test	**asmtest:** a collection of RISC-V tests for instructions and syscalls. **insttest:** tests for SPARC instructions. **riscv-tests:** RISC-V processor unit tests. **simple:** tests for m5ops and ARM semi-hosting. **square:** test for squaring a vector of floats on AMD GPU.

In an ongoing effort to improve the gem5 framework, we provide a working status of the GEM5 RESOURCES on gem5 releases at http://resources.gem5.org.

A. Environment Resources

As an example of how GEM5 RESOURCES can be used to ease the burden end-users face when setting up simulations, we look at simulating GPUs. One of the gem5 GPU models is based on AMD's GCN3 architecture [19], [20]. In order for users to compile and run GPU applications on this GPU model, they must have the proper ROCm stack (version 1.6) installed [21]. Moreover, the libraries are also needed to interface with the kernel-space driver, which is emulated within gem5. Installing these libraries correctly is difficult: even with existing documents explaining what should be installed, there are many posts on the gem5 forum from frustrated users unable to get things installed properly.

Due to this complexity, we created a Docker image as part of GEM5 RESOURCES which automatically sets up the correct environment to build and run GPU applications on the simulated AMD GPU [22]. This Docker image completely removes the frustration of trying to correctly setup the environment on a host machine. Instead, users who want to run their GPU application in gem5 can simply pull the image and run it with no setup required. In GEM5 RESOURCES we also provide a dockerfile that can serve as step-by-step instructions for users who want to install the libraries on their machine to avoid any

docker overheads, or as a starting point for users who want to modify the libraries as part of their GPU experiments. Thus, thanks to GEM5 RESOURCES, users can easily compile GPU applications and simulate them within gem5.

Additionally, in GEM5 RESOURCES we provide a wide range of GPU applications, from HIP sample applications [23] which showcase various GPU features and primitives, to HeteroSync [13], an open-source GPU synchronization primitives library, to DNNMark [14], a benchmark suite with various DNN layers, as well as DOE Proxy Applications such as HACC [15], LULESH [18], [17], and PENNANT [16].

VI. USE CASES

To demonstrate the usefulness of GEM5ART and GEM5 RESOURCES we demonstrate three resources. First, we show the impact of using up-to-date resources with the PARSEC benchmark suite. We show that when used with Ubuntu 18.04 (released in April of 2018) and Ubuntu 20.04 (released two years later in April 2020) there are differences in the simulated results. Second, we demonstrate that GEM5ART enables large cross-product studies by running "Linux boot tests" for a variety of systems and Linux kernel versions. Finally, we show how register allocation scheme affects the performance of a GPU applications, and identify future GPU model contribution opportunities.

To run these use cases and other work for this paper, we used GEM5ART and stored all results in a centralized database. We provide a complete archive of the experiment data and scripts at https://doi.org/10.6084/m9.figshare.14176802.

A. Use-Case 1: PARSEC

The PARSEC [4] is a popular benchmark suite composed of multi-threaded applications. The suite contains 13 applications, each of which focus on a particular problem domain such as image processing or option pricing. In this use-case we envisioned a scenario where a researcher wishes to observe the performance of the PARSEC benchmark suite across different LTS Ubuntu OS releases. We settled on evaluating the two latest LTS released, Ubuntu 18.04 and 20.4. Table II details the system under simulation. Due to runtime issues in the `x264`, `facesim` and `canneal` applications, they have been removed from our analysis. We ran these problematic workloads in QEMU instead of gem5 and found they experienced similar errors as when they ran in gem5. Thus, we conclude that there are bugs with the benchmarks themselves. By enabling the simple reproduction and use of these workloads in an open-source manner, we hope that others who use these workloads will contribute fixes which can be used by the rest of the community.

To run the 10 remaining applications in the PARSEC benchmark suite, we executed a series of full system simulations. To do so we used the GEM5ART `createFSRun` method, as outlined in Figure 4. The `gem5_git_artifact` was set to the gem5 git source repository[7], version 20.1.0.4, with the

[7]https://gem5.googlesource.com/public/gem5.

TABLE II: Configuration Parameters for Use-Case 1

Component	Options
CPU	TimingSimpleCPU
Number of CPUs	1, 2, 8
Memory	1 channel, DDR3_1600_8x8
OS	Ubuntu 20.04 (kernel version: 5.4.51), Ubuntu 18.04 (kernel version: 4.15.18)
Workloads	Blackscholes, Bodytrack, Dedup, Ferret, Fluidanimate, Freqmine, Raytrace, Streamcluster, Swaptions, Vips
Input sizes	simmedium

`gem5_artifact` as a gem5 binary compiled from the source using the GCC 7.5 compiler. The `dist_image_artifact` varies between an image of the PARSEC benchmark suite running atop Ubuntu 18.04, and another with PARSEC benchmark suite running atop Ubuntu 20.04; both available as part of GEM5 RESOURCES [24]. All the artifacts used in this series of runs are available from GEM5 RESOURCES.

We used the Linux v4.15.18 kernel for Ubuntu 18.04 and the v5.4.51 kernel for Ubuntu 20.04, as the `linux_binary_artifact`. These, again, are available from GEM5 RESOURCES [25]. Finally, we use the run script from the PARSEC benchmark suite resource [24]. This script takes in the disk image, the kernel, the CPU type, the number of CPUs, the PARSEC application to run, and the application input as parameters. In our experiments we run using the `TimingSimpleCPU`, with 10 parsec applications, utilizing the `SimMedium` inputs, using both a single CPU, 2 CPUs, and 8 CPUs.

Though this setup, at a surface level, seems like a basic experiment, there are a lot of runs and components to keep track of. Using two OS's (each with a different kernel), 10 applications, run on a single CPU, 2 CPUs, and again on 8 CPU simulation, gives a cross product of 60 gem5 runs. Though, using GEM5ART, we only needed to outline the artifacts then execute the run functions. The job scheduler then runs gem5, with all the results and artifacts stored carefully in the database.

To improve the extracting of data from our MongoDB instance, we created a Jupyter Notebook instance [26], to analyze data and automatically created graphs using Python's Matplotlib library [27].

Figure 6 was generated from the database. The graph shows the absolute execution time difference of each PARSEC benchmark suite application in 20.04, compared to Ubuntu 18.04, for 1, 2, and 8 cores. The applications typically take longer to execute in Ubuntu 18.04, though the difference becomes less so as more CPU cores are utilized. We found upon further analysis that PARSEC running in Ubuntu 20.04 was executing significantly more instructions, but at a higher CPU utilization rate. We suspect the reason for this is Ubuntu 20.04 coming bundled with a different, newer version of GCC (version 9.3 in contract to Ubuntu 18.04's 7.4). Differences arising form using different Linux kernels in Ubuntu 20.04 18.04 could also be playing a role.

Figure 7 shows the rate of speed-up of using 8 CPUS for the

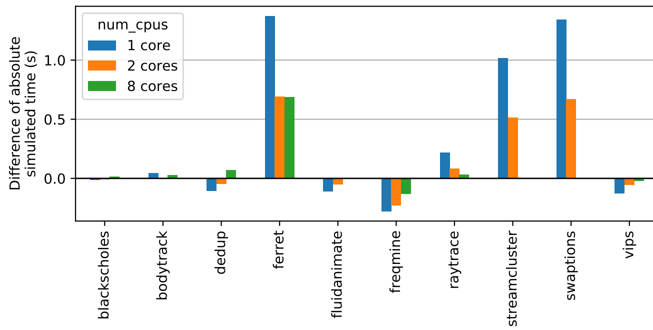

Fig. 6: The absolute execution time difference of the PARSEC benchmark suite in Ubuntu 18.04, compared to Ubuntu 20.04.

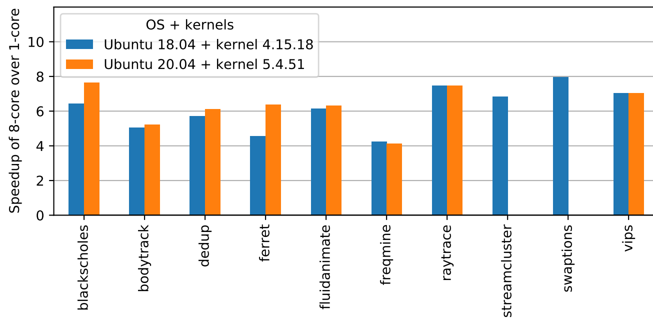

Fig. 7: The PARSEC suite execution time speedup between 1 and 8 cores across Ubuntu 18.04 and Ubuntu 20.04.

PARSEC workloads compared to using 1 CPU as a baseline, on both Ubuntu 18.04 and 20.04. The data shows that the rate of speedup is relatively consistent between the two OSs, though, on average Ubuntu 20.04 achieves a greater speedup particularly in the case of `blackscholes` and `ferret`. This suggests the Ubuntu 20.04 disk image is achieving greater CPU utilization.

Using GEM5ART and GEM5 RESOURCES, this experiment was trivial to setup, ran automatically, and stored the results in a database to query and visualize later. If required, this experiment may be run again, or with slightly altered parameters, quickly and easily. Furthermore, we are free to make this database publicly available for others to analyze and reproduce our results as they see fit.

B. Use-Case 2: Linux Boot Tests

As our second use-case study, we carry out Linux boot checks. Checking the Linux kernel boots for a particular architecture design is a standard procedure. Historically, the state of support of the latest Linux kernel versions on gem5 has remained hard to discover, and has led researchers to use archaic Linux kernel versions for their evaluations. The test inevitably validates many components due to the complexity of modern OSes, and is normally a "must have" check for many designs. In this use-case, we outline a scenario of testing a simple configuration cross five different variables. In essence, we are testing gem5's viability at booting Linux under certain configurations. The variables are CPU count, CPU type, memory system, Linux kernel, and boot type (more details in Figure 8). The `boot-exit` disk image resource, comes with a simple gem5 run script [28] which we utilize for this work.

Testing the complex cross product of all these options, under most circumstances, would be a daunting prospect. There are 480 different runs of gem5 to obtain and keep track of. As can be imagined, the room for mistakes and confusion is large. A single misconfigured run, exposing a "bug" which does not exist, could result in considerable misplaced engineering effort. It is for this reason GEM5ART was developed.

As with any other usage of GEM5ART, we simply needed to specify all the artifacts needed for these experiments in a GEM5ART launch script. In this case, the gem5 binary (v20.1.0.4) and repository, the boot-exit disk image, the linux kernels, and the gem5 run script are the artifacts. The appropriate run methods are then called in the GEM5ART launch script, with the correct artifacts, and the parameters to be passed to the gem5 run script (the Memory System, CPU Type, Number of CPUs, and the Boot Type). Though a considerable workload, each gem5 run can function independently, meaning parallelization is possible. The external job scheduler interacts with GEM5ART and completely automates this task.

Figure 8 shows the results of these gem5 runs using GEM5ART. The out-of-order CPU, and the TimingSimpleCPU cannot handle more than one core when running on the Classic memory system, and the AtomicTimingCPU cannot function on the Ruby memory system as of gem5 v20.1.0.4. For the remainder of the data, kvmCPU works in all cases. AtomicSimpleCPU works in all supported cases i.e., with Classic memory system. TimingSimpleCPU also works for all supported cases i.e., except more than 1 CPU for Classic memory system.

As shown in Figure 8, O3CPU runs have mixed results with approximately 40% of them running successfully. For O3CPU, there are 27 cases where the kernel went into panic during simulation and 31 cases where gem5 failed to simulate Linux boot because of other reasons. Out of these 31 cases, gem5 crashes because of a segmentation fault in 11 cases (the issue has been recorded on the gem5 bug tracking tool[8]). gem5 crashes in 4 cases because of a "possible deadlock detected" error (all in `MI_example` runs). For the rest of the O3CPU runs, gem5 fails to finish successfully in a reasonable amount of time (24 hours, most successful runs finish in a 12 hour timeout) without any explicit errors. These are all likely bugs within gem5; however, GEM5ART will give the gem5 developers a useful tool to help effectively direct their debugging efforts.

C. Use-Case 3: GPUs

As a third use case, we illustrate using gem55 and GEM5 RESOURCES to investigate GPU architecture design-space analysis on a variety of workloads. For modern GPUs, register

[8]https://gem5.atlassian.net/browse/GEM5-782.

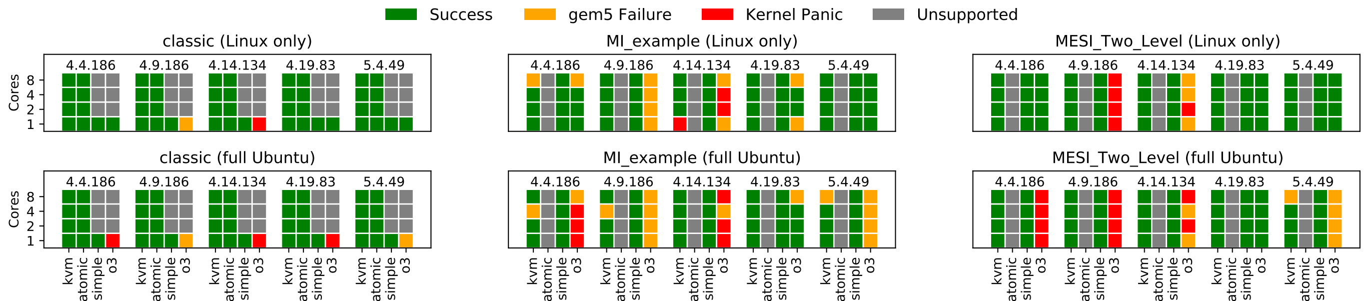

Fig. 8: Results from testing a cross product of kernels, CPU models, memory systems, and CPU cores. The top plots show the results when booting only the Linux kernel. The bottom plots show results when booting to runlevel 5 (multi-user) in Ubuntu. **CPU Type:** kvmCPU (simulates code using hosts' hardware), AtomicSimpleCPU (uses atomic memory accesses and no timing simulation), TimingSimpleCPU (uses timing simulation only for memory accesses), O3CPU (an out-of-order CPU, uses timing for both CPU and memory). **Memory System:** Classic (fast but lacks coherence fidelity), Ruby (slower but models detailed memory with cache coherence flexibility; `MI_Example` and `MESI_Two_Level` are used in this experiment). **Linux Kernel:** five different LTS (long term support) Linux kernels.

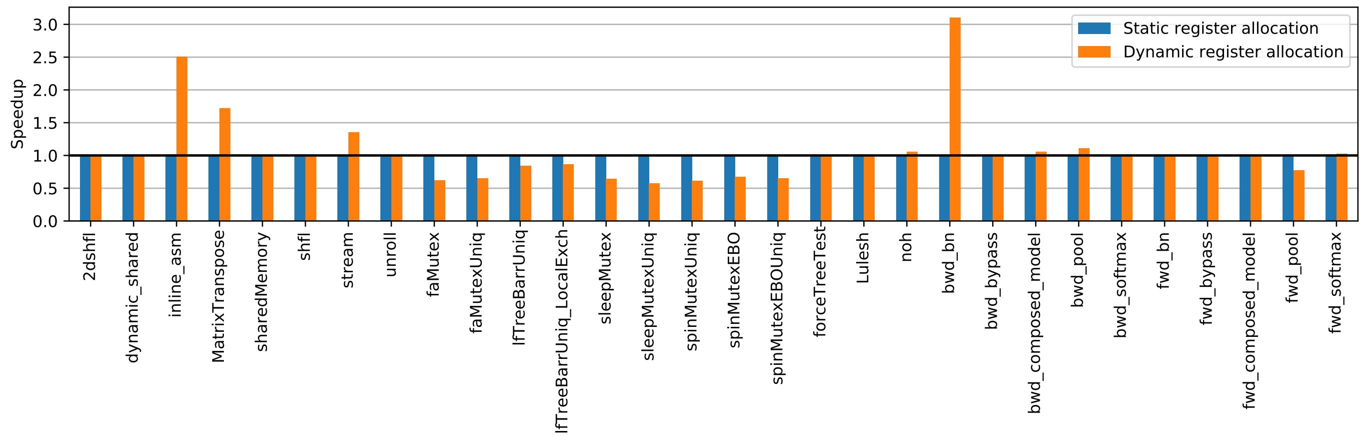

Fig. 9: The speedup of the gem5 GCN3 GPU model using simple and dynamic register allocators, normalized to the simple register allocator.

TABLE III: Key Configuration Parameters for Use-Case 3

Component	Value
Number of CUs	4
SIMD16s (vector ALUs)	4 per CU
GPU Frequency	1 GHz
Max Wavefronts	10 per SIMD16 (40 per CU)
Vector Registers	8K per CU
Scalar Registers	8K per CU
LDS	64 KB per CU
L1 instruction cache	32 KB shared between every 4 CUs
L1 data caches (1 per CU)	16 KB per CU
Unified L2 cache	256 KB
Main Memory	1 channel, DDR3_1600_8x8

usage is often a critical component that must be tuned properly to enable high performance [35], [36]. Thus, examining ways to tune GPU register allocation can significantly affect an application's performance [37], [38]. To examine how gem55 and GEM5 RESOURCES enables rapid study of GPU configurations, we study how GPU performance varies for the two available GPU register allocation schemes: a *simple* allocation scheme that allocates 1 wavefront per SIMD16 [19] in a compute unit (CU) at a time to limit stalls and a *dynamic* allocation scheme that always allows up to the max wavefronts per CU at a time by monitoring per wavefront register requirements compared to the number of available registers per CU. In theory, the dynamic scheme should outperform the simple one when there are more work groups (WGs) than can be scheduled initially,[9] because it enables this additional work to be overlapped. However, GPUs have very simple pipelines with limited mechanisms for tracking dependencies [20]; thus this additional parallelism may cause additional stalls that hurt performance. Table III details the other key system parameters for our simulated system, which models a tightly coupled CPU-GPU system with coherent caches and a unified address space.

We evaluate the simple and dynamic register allocation policies across a number of GPU workloads, all of which are

[9]WGs contain one or more wavefronts, each of which has up to 64 threads.

TABLE IV: Benchmarks & Input Sizes for Use-Case 3 (WG = Work Groups, CU = Compute Units, CS = critical section).

Application	Input Size
2dshfl [23], [29]	4x4
dynamic_shared [23], [29]	16x16
inline_asm [23], [29]	1024x1024
MatrixTranspose [23], [29]	1024x1024
sharedMemory [23], [29]	64x64
shfl [23], [29]	4x4
stream [23], [29]	32x32
unroll [23], [29]	4x4
SpinMutexEBO [13], [30] FAMutex [13], [30] SleepMutex [13], [30] SpinMutexEBOUniq [13], [30] FAMutexUniq [13], [30] SleepMutexUniq [13], [30]	10 Ld/St/thr/CS, 8 WGs/CU, 2 iters
LFTreeBarrUniq [13], [30] LFTreeBarrUniqLocalExch [13], [30]	10 Ld/St/thr/barrier, 8 WGs/CU, 2 iters
bwd_bypass [14], [31]	$NCHW = 100, 1000, 1, 1$
bwd_bn [14], [31]	$NCHW = 100, 1000, 1, 1$
bwd_composed_model [14], [31]	$NCHW = 32, 32, 3, 1$
bwd_pool [14], [31]	$NCHW = 100, 3, 256, 256$
bwd_softmax [14], [31]	$NCHW = 100, 1000, 1, 1$
fwd_bypass [14], [31]	$NCHW = 100, 1000, 1, 1$
fwd_bn [14], [31]	$NCHW = 100, 1000, 1, 1$
fwd_composed_model [14], [31]	$NCHW = 32, 32, 3, 1$
fwd_pool [14], [31]	$NCHW = 100, 3, 256, 256$
fwd_softmax [14], [31]	$NCHW = 100, 1000, 1, 1$
HACC [15], [32] (forceTreeTest)	0.5 0.1 64 0.1 100 N 12 rcb
LULESH [18], [17], [33]	1 iteration
PENNANT [16], [34]	noh

available in GEM5 RESOURCES. For the DNNMark applications and PENNANT GEM5 RESOURCES includes input files. Table IV summarizes these applications and their input sizes. The applications represent a number of different use cases and application sizes, and thus provide a wide set of data points to evaluate the efficacy of the register allocators. All applications run on AMD ROCm 1.6, and use the corresponding HIP, MIOpen, and rocBLAS library versions. Moreover, for all applications we utilize our docker support that automatically builds the correct ROCm stack (Section V-A). Thus, to conduct similar experiments, a researcher would need to checkout the appropriate version of gem5 (gem5 v21.0) and the corresponding GEM5 RESOURCES, compile gem5 with the GCN3_X86 configuration, compile GEM5 RESOURCES or download them, and then run each application using our docker, the same system configuration (Table III), and the same inputs (Table IV).

Figure 9 shows how the GPU execution time (in shader ticks) varies for the simple and dynamic register allocation policies. Surprisingly, overall the dynamic register allocator is actually outperformed by the simple register allocator: on average the simple register allocator improves GPU performance by 8% compared to the dynamic register allocator. The largest contributor to this surprising result is the overly simplistic dependence tracking information in the publicly available GPU model. The HeteroSync applications, bwd_pool, and fwd_pool in particular suffer (e.g., the dynamic register allocator is 61% and 22% worse for FAMutex and fwd_pool, respectively). This highlights how optimizing the register allocator in isolation is insufficient, and how future contributions to gem5 that improve the dependence tracking could pay significant dividends. By enabling a broad variety of workloads, we were able quickly and easily find the modeling inaccuracies in gem55.

Other applications with small kernels (e.g., 2dshfl, dynamic_shared) or limited additional work to schedule (e.g., HACC and LULESH) are less affected by running more wavefronts per CU. Thus, they show little or no difference between the register allocators. However, the remaining applications (inline_asm, MatrixTransponse, PENNANT, stream, and some of the DNNMark ML layers) having a dynamic register allocator significantly improves performance. Unlike the other applications, these applications have more work than can be scheduled initially, fully utilize the GPU, or are very compute intensive. Thus, the dynamic register allocator allows them to overlap additional computation and helps hide the latency of accessing memory. More broadly, this work shows how GPU researchers can easily configure and utilize our work to examine other interesting research questions.

VII. Conclusions and Potential Impact

GEM5ART and GEM5 RESOURCES will enable novel computer architecture research. With these tools, computer architecture researchers will be able to concentrate on novel computer architecture designs instead of wasting time getting the simulation framework up and running.

We will be releasing both GEM5ART and GEM5 RESOURCES as open source community-driven projects and encourage the entire computer architecture community to contribute new resources and we will host them in a centralized repository for the rest of the community to use. As the gem5 project continues to improve and adapt to meet the demands of researchers, so will our framework. The new framework here, GEM5ART is he latest addition to the project, and we hope, will bring great benefits to the community.

Using the GEM5ART framework, we could potentially even host *simulation results* from the broader computer architecture community in a centralized repository. With a consistent schema for representing both inputs and output of simulations, this repository would significantly improve the reproducibility of computer architecture research.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[2] J. Lowe-Power *et al.*, "The gem5 Simulator: Version 20.0+," 2020.

[3] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[4] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[5] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, and C. Fetzer, "Fex: A software systems evaluator," in *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks — DSN '17*. IEEE, 2017, pp. 543–550.

[6] L. Oliveira, D. Wilkinson, D. Mossé, and B. R. Childers, "Occam: Software environment for creating reproducible research," in *Proccedings of the IEEE 14th International Conference on e-Science — e-Science '18*. IEEE, 2018, pp. 394–395.

[7] "Collective Knowledge," Accessed 2020-10-30. [Online]. Available: https://cknowledge.org/

[8] M. Walker, S. Bischoff, S. Diestelhorst, G. Merrett, and B. Al-Hashimi, "Hardware-Validated CPU Performance and Energy Modelling," in *Proceedings of the 2018 IEEE International Symposium on Performance Analysis of Systems and Software — ISPASS '18*. IEEE, 2018, pp. 44–53.

[9] J.-E. Jo, G.-H. Lee, H. Jang, J. Lee, M. Ajdari, and J. Kim, "DiagSim: Systematically diagnosing simulators for healthy simulations," *ACM Transactions on Architecture and Code Optimization — TACO '18*, vol. 15, no. 1, pp. 1–27, 2018.

[10] "The celery project," Accessed 2020-10-26. [Online]. Available: https://docs.celeryproject.org

[11] "MongoDB," Accessed 2020-10-26. [Online]. Available: https://www.mongodb.com

[12] "Packer," Accessed 2020-10-26. [Online]. Available: https://www.packer.io

[13] M. D. Sinclair, J. Alsop, and S. V. Adve, "HeteroSync: A benchmark suite for fine-grained synchronization on tightly coupled gpus," in *Proccedings of the IEEE International Symposium on Workload Characterization — IISWC '17*. IEEE, 2017, pp. 239–249.

[14] S. Dong and D. Kaeli, "DNNMark: A deep neural network benchmark suite for gpus," in *Proceedings of the General Purpose GPUs*, 2017, pp. 63–72.

[15] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka *et al.*, "HACC: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49–65, 2016.

[16] C. R. Ferenbaugh, "PENNANT: An unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4555–4572, 2015.

[17] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong, "LULESH Programming Model and Performance Ports Overview," Lawrence Livermore National Labs, Tech. Rep. LLNL-TR-608824, December 2012.

[18] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[19] "Graphics Core Next Architecture, Generation 3," Accessed 2020-10-30. [Online]. Available: http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf

[20] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers, "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level," in *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture — HPCA '21*, ser. HPCA, 2018, pp. 608–619.

[21] "ROCm Device Libraries," Accessed 2020-10-30. [Online]. Available: https://github.com/RadeonOpenCompute/ROCm-Device-Libs

[22] "gem5 Resources. Resource: GNC3 GPU Docker," git repository at revision '2a4357b'. [Online]. Available: https://gem5.googlesource.com/public/gem5/+/2a4357bfd0c688a19cfd6b1c600bb2d2d6fa6151/util/dockerfiles/gcn-gpu

[23] AMD, "HIP Sample Apps," 2020. [Online]. Available: https://github.com/ROCm-Developer-Tools/HIP/tree/master/samples/2_Cookbook

[24] "gem5 Resources. Resource: PARSEC," git repository at revision '31924b6'. [Online]. Available: https://gem5.googlesource.com/public/gem5-resources/+/c5f5c70d0291e105444f534cf538ea40e4ddcb96/src/parsec

[25] "gem5 Resources. Resource: linux-kernel," git repository at revision 'c5f5c70'. [Online]. Available: https://gem5.googlesource.com/public/gem5-resources/+/c5f5c70d0291e105444f534cf538ea40e4ddcb96/src/linux-kernel

[26] "Project Jupyter," Accessed 2020-10-19. [Online]. Available: https://jupyter.org/

[27] "Matplotlib," Accessed 2020-10-28. [Online]. Available: https://matplotlib.org

[28] "gem5 Resources. Resource: boot-exit run script," git repository at revision 'c5f5c70'. [Online]. Available: https://gem5.googlesource.com/public/gem5-resources/+/c5f5c70d0291e105444f534cf538ea40e4ddcb96/src/boot-exit/configs

[29] "gem5 Resources. Resource: HIP Samples," git repository at revision '8a8193a'. [Online]. Available: https://gem5.googlesource.com/public/gem5-resources/+/8a8193a69075b7baa1db438a41ef56a6bf2d4d5b/src/hip-samples

[30] "gem5 Resources. Resource: HeteroSync," git repository at revision '8a8193a'. [Online]. Available: https://gem5.googlesource.com/public/gem5-resources/+/8a8193a69075b7baa1db438a41ef56a6bf2d4d5b/src/heterosync

[31] "gem5 Resources. Resource: DNNMark," git repository at revision '8a8193a'. [Online]. Available: https://gem5.googlesource.com/public/gem5-resources/+/8a8193a69075b7baa1db438a41ef56a6bf2d4d5b/src/DNNMark

[32] "gem5 Resources. Resource: Halo-Finder," git repository at revision '8a8193a'. [Online]. Available: https://gem5.googlesource.com/public/gem5-resources/+/8a8193a69075b7baa1db438a41ef56a6bf2d4d5b/src/halo-finder

[33] "gem5 Resources. Resource: LULESH," git repository at revision '8a8193a'. [Online]. Available: https://gem5.googlesource.com/public/gem5-resources/+/8a8193a69075b7baa1db438a41ef56a6bf2d4d5b/src/lulesh

[34] "gem5 Resources. Resource: PENNANT," git repository at revision '9dda943'. [Online]. Available: https://gem5.googlesource.com/public/gem5-resources/+/9dda943b120666fd44b53674a142a747c1e86892/src/pennant

[35] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture — MICRO '12*, 2012, pp. 96–106.

[36] B. Pourghassemi, C. Zhang, J. H. Lee, and A. Chandramowlishwaran, "On the Limits of Parallelizing Convolutional Neural Networks on GPUs," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA, 2020, p. 567–569. [Online]. Available: https://doi.org/10.1145/3350755.3400266

[37] D. Sampaio, "GPU Divergence: Analysis and Register Allocation," Tech. Rep., 2017.

[38] Y. You and S. Chen, "Vector-aware Register Allocation for GPU Shader Processors," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems — CASES '15*, 2015, pp. 99–108.