Gem5-X + ALPINE
Full System Manual

[*]EMBEDDED SYSTEMS LABORATORY,
SWISS FEDERAL INSTITUTE OF TECHNOLOGY, LAUSANNE (EPFL)

[‡]SCHOOL OF ENGINEERING AND MANAGEMENT VAUD (HEIG-VD),
UNIVERSITY OF APPLIED SCIENCES WESTERN SWITZERLAND (HES-SO)

[**]DEPARTMENT OF COMPUTER ARCHITECTURE,
COMPLUTENSE UNIVERSITY OF MADRID

V2.2-ALPINE BY JOSHUA KLEIN[*], MARINA ZAPATER[*‡], AND GIOVANNI ANSALONI[*]

BASED ON V2.0 OF GEM5-X TECHNICAL MANUAL BY YASIR QURESHI[*], WILLIAM SIMON[*], MARINA ZAPATER[*‡], KATZALIN OLCOZ[**], AND DAVID ATIENZA[*]

December 2022

Contents

1 Executive Summary

1.1 Abstract

The gem5 architectural simulator is well established and widely used in both the industry and academia. Based on gem5, we present we present gem5-X (*"a gem5-based full-system simulator with architectural eXtensions*"), a simulation framework that enables fast profiling and architectural exploration and optimization for system level architectural innovations. Gem5-X provides out-of-the-box simulation of ARM based systems with full Linux stack, along with several architectural extensions like ISA extensions, clustering, heterogeneous many-core simulation and HBM2 memory model. Several enhanced features have also been added, like advanced check-pointing, workload automation (WA) and gperf profiler support.

This version of the gem5-X repository, named gem5-X-ALPINE, further provides support for Analog In-Memory Computing (AIMC) accelerators, developed in the context of the Wiplash Horizon 2020 project (https://www.wiplash.eu).

In this technical manual, we first provide guidelines on how to use various architecture features and support enhancements of gem5-X. More information on downloading and source code for gem5-X can be found at https://esl.epfl.ch/gem5-x.

Then, in Section 8, we describe the features specific to gem5-X-ALPINE.

1.2 Release Information

Version	Authors	Date	Changes
v2.2-ALPINE	Joshua Klein, Giovanni Ansaloni, and Marina Zapater	December 2022	Forked gem5-X manual for ALPINE extension.
v2.2	Joshua Klein, Rafael Medina, Alireza Amirshahi, Marina Zapater, and Giovanni Ansaloni	October 2022	Reformatting and setting up for new extensions.
v2.1	Joshua Klein and Darong Huang	February 2022	Updated version information for gem5-X dependencies on Ubuntu 20.04 LTS, expanded contact info for current maintainers.
v2.0	Yasir Qureshi, William Simon, Marina Zapater, Katzalin Olcoz, and David Atienza	August 2021	Core clustering, heterogeneous cores and SPM support added in Gem5-X.

1.3 Collaboration and Contact Information

The maintainers of this project can be contacted via email at {joshua.klein, rafael medina, giovanni.ansaloni, david.atienza}@epfl.ch, marina.zapater@heig-vd.ch.

Because the scope of this project is very large, we are always interested in potential collaboration efforts to develop new features and keep gem5-X updated to gem5 master. For inquiries, source code, and additional information, please contact one of the aforementioned emails.

2 Running gem5-X Full System (FS) Mode with ARMv8 and Linux

In this chapter we describe how to configure and run our ARMv8 64-bit FS simulation in gem5-X .

2.1 Necessary Files

Because our model is run in FS mode with a full Linux environment, we need several major system components. This includes,

- A bootloader
- A kernel binary, e.g., vmlinux
- A disk image
- A device tree binary

All of the aforementioned components must be compatible with the ARMv8.

2.1.1 Full System Files

Once you register for gem5-X at https://esl.epfl.ch/gem5-x, you will receive an email with a link to all the system files, except for the device tree. The file downloaded is named ***full_system_images.tar.gz***. This contains the disk image, bootloader and kernel binary. Follow the instructions below to set it up

```
1 tar -zxvf full_system_images.tar.gz
```

The files are as follows:

- Bootloader is under **[path_to_full_system_images]/binaries/**
- Kernels (vmlimux and vmlinux_wa) are at **[path_to_full_system_images]/binaries/**
- Disk image (gem5_ubuntu16.img) can be found at **[path_to_full_system_images]/disks/**

We now need to setup the path to *full_system_images*, so that the files under it can be used and recognized by gem5-X during FS simulation.

```
1 cd <path_to_gem5-X>
2 ./apply-patch.sh <PATH_TO_FULL_SYSTEM_IMAGES>
```

Alternatively, you can also do

```
1 export M5_PATH=<PATH_TO_FULL_SYSTEM_IMAGES>
```

The full system files are now setup and ready to be used in FS mode.

2.1.2 Device Tree

The device tree files are under

```
<path_to_gem5-X>/system/arm/dt
```

If running on an Ubuntu-based host system , the following prerequisites need to be installed before generating the device tree binaries.

```
sudo apt-get install  gcc-arm-linux-gnueabihf gcc-aarch64-linux-gnu
sudo apt-get install device-tree-compiler
```

To generate the device tree binary files,

```
cd <path_to_gem5-X>
make -C system/arm/dt
```

2.2 Quick-Start Guide

In this brief start-up guide, we will guide you through the basic steps to running your first full system (FS) simualtion with gem5-X . This guide assumes you have already the bootloader, device tree, kernel file, and disk image setup as described in the previous sections.

2.2.1 Prerequisites

You will need to set up the gem5-X environment in order to compile and run the gem5-X binary using the SCons (SConstruct) builder. If running on an Ubuntu-based host system, you can use the following command to get all the required libraries. However, there are some known dependency problems on the latest Ubuntu image, i.e., 20.04. If you are running these host systems, we recommend you follow **Section 2.3** to build a docker image to run gem5-X inside.

```
    sudo apt install build-essential git m4 scons zlib1g \
    zlib1g-dev libprotobuf-dev protobuf-compiler libprotoc-dev \
    libgoogle-perftools-dev python-dev python-six python \
    libboost-all-dev swig
```

2.2.2 Building the gem5 Binary

Once the above is done, you will need to build a ARM gem5 binary. You can create multiple builds including .fast, .opt, and .debug. If you are only concerned about running experiments, it is recommended to only create gem5.fast. However, if you need to debug anything or want to generate traces, you will need to build gem5.opt or gem5.debug. Do this with the following:

```
    cd <path_to_gem5-X>/
    scons build/ARM/gem5.{fast, opt, debug}
```

Additionally, if you would like to speed up the compilation process, you can use the option ”-jN” on the scons build line where N is the number of threads you want to assign for compilation.

2.2.3 Running Your FS Simulation

Once the build process is complete you can launch your simulation in the following way

```
cd <path_to_gem5-X>/

./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<full_path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_1cpu.dtb \
-n 1 \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--mem-type=DDR4_2400_4x16 \
--mem-ranks=4 \
--mem-size=4GB \
--sys-clock=1600MHz \
```

At this point you should be able to connect to your running gem5 instance in another terminal with,

```
telnet localhost 3456
```

Alternatively, you can also build the terminal program provided with gem5-X and use it

```
cd <path_to_gem5-X>/util/term/
make
m5term 127.0.0.1 3456
```

Upon connecting to your gem5 instance, you should be able to the kernel dmesg, followed finally by a login and a terminal in the gem5-X FS mode

2.3 Hot-fixes for running gem5-X on Ubuntu 20.04 using Docker

Because of updates to gem5-X dependencies in Ubuntu 20.04, specific dependency versions must be installed. In particular, python 2.7.5 and SCons 3.0.0 (build version py27h8a56064_0) must be used. These packages can be easily configured via a virtual environment. Here we provided a Dockerfile to create the docker image, containing all of the necessary dependencies to run gem5-X.

```
FROM ubuntu:20.04
SHELL ["/bin/bash", "-c"]
ENV DEBIAN_FRONTEND=noninteractive
```

```
4  RUN echo "deb http://dk.archive.ubuntu.com/ubuntu/ xenial main" \
5    >> /etc/apt/sources.list \
6    && echo \
7    "deb http://dk.archive.ubuntu.com/ubuntu/ xenial universe" \
8    >> /etc/apt/sources.list \
9    && apt -y update \
10   && apt install -y wget \
11   && wget \
12   https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh\
13   && bash Miniconda3-latest-Linux-x86_64.sh -b -p /opt/miniconda3
14 RUN source /opt/miniconda3/bin/activate \
15   && conda init \
16   && conda create --name py275 -c free python=2.7.5 -y \
17   && conda activate py275 \
18   && conda install scons=3.0.0=py27h8a56064_0 -y
19
20 RUN apt-get -y install build-essential gcc-arm-linux-gnueabihf \
21 gcc-aarch64-linux-gnu \
22   device-tree-compiler make git m4 zlib1g \
23   zlib1g-dev libprotobuf-dev protobuf-compiler libprotoc-dev \
24   libgoogle-perftools-dev python-dev \
25   libboost-all-dev swig=3.0.8-0ubuntu3 \
26   && apt-get -y install diod \
27   && apt-get -y install qemu qemu-user qemu-system \
28   qemu-user-static
```

The above docker file should be put inside a newly created folder, e.g. gem5x-docker/Dockerfile. Then you can build the docker image in the terminal:

```
1    cd <path to gem5x-docker/Dockerfile>
2    docker build -t gem5x .
```

Wait until the image is successfully created, then you can run the image by using the following command:

```
1    (sudo) docker run -it gem5x
```

Before you go to Section 2.2.2 to build the gem5, you need to first enable the conda environment:

```
1    source /opt/miniconda3/bin/activate
2    conda activate py275
```

Besides, two additional hotfixes are required in gem5-X's SConstruct file (**gem5-X/SConstruct**): First, due to deprecated features in gcc 9.3.0+, lines 365 - 370 should be commented out. Second, the GNU assembler version in the Sconstruct file needs to be updated, so change [-1] to [3] in line 435.

Now you can go back to **Section 2.2.2** to build the gem5 binary. Please contact the maintainers if issues continue to arise.

3 Support Enhancements of Gem5-X

In this chapter we will look into the following support enhancements we have added in gem5-X:

- Enhanced checkpointing
- gperf profiler
- File sharing between gem5-X and host system using 9P over Virtio
- Modifying disk image using QEMU

3.1 Enhanced Checkpointing

The boot process during the FS simulation in gem5-X automatically takes a checkpoint when the boot and login is complete and we get the terminal. Since the boot is in SimpleAtomic CPU model, the timing information is not there in the simulation. We can now switch to an accurate in-order or our-of-order (OoO) CPU model with all the timing information.

If your simulation is still running after the boot, you can exit it using the following command in the connected terminal,

```
m5 exit
```

Now we can use the checkpoint, that was automatically taken after boot and login, and switch to an accurate CPU model.

```
./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<full_path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_1cpu.dtb \
-n 1 \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--mem-type=DDR4_2400_4x16 \
--mem-ranks=4 \
--mem-size=4GB \
--sys-clock=1600MHz \
-r 1 \
--cpu-type={MinorCPU, DerivO3CPU}
```

The number after *–r* is the checkpoint number. In this case we are resuming from the first checkpoint. The CPU type can be *MinorCPU* for in-order core or *DerivO3CPU* for OoO cores.

Sometimes it is feasible to take a checkpoint using SimpleAtomic CPU model just before your region-of-interest (ROI) and then switch to an accurate in-order or OoO CPU. You can do this in either the command prompt or a script using the following command:

```
m5 checkpoint
```

If you are in a C/C++ program, you can use the following call within the program,

```
system("m5 checkpoint");
```

3.2 Gperf Profiler

Profiling capabilities within FS, by installing the gperf profiler on the disk image. The gperf statistical profiler developed by Google provides profiling capabilities on gem5-X itself with minimal overhead, enabling the identification of application bottlenecks and exploration of the effectiveness of architectural modifications and extensions.

To enable profiling using gperf when running a program, follow the instructions below;

```
LD_PRELOAD=/usr/lib/libprofiler.so.0 CPUPROFILE=<FILE_TO_SAVE_PROFILING>
CPUFREQUENCY=1000 <program>
```

This will launch the program to be profiled with profiling data being saved to file mentioned in *CPUPROFILE* parameter.

To view the data, we first convert it to .pdf file and then write it to the host machine as follows;

```
1 google-pprof --pdf <FILE_WITH_PROFILING_DATA> > <FILENAME>.pdf
2 m5 writefile <FILENAME>.pdf
```

The file *FILENAME.pdf* will now be available to be viewed in the host system under path passed to –d parameter when launching gem5-X simulation

```
-d /path/to/your/output/directory
```

3.3 9P over Virtio

We utilize the 9P protocol developed by Bell Labsover a virtio device driver to allow fast modification of files without modifying the root file system in gem5-X. While this feature is available in vanilla gem5, it is not enabled by default and has no kernel support. Both of these features are provided in gem5-X. Once Linux is booted, a folder on the host machine can be mounted within gem5 to access files on the host system. Without 9P mounting, every time a program is modified, we need to reload the disk image required for FS simulation and reboot Linux. In gem5, this process can take up to 20-30 minutes, a bottleneck that gem5-X eliminates. To use 9p over Virtio, follow the instructions below:

- First we need to install DIOD

  ```
  sudo apt-get install diod
  ```

- After installation, check where DIOD is installed by typing "which diod". This path should be updated in the file src/dev/virtio/VirtIO9P.py at line 62. Then re-compile gem5-X using scons command as usual.

```
cd <path_to_gem5-X>/
scons build/ARM/gem5.{fast, opt, debug}
```

- Use kernel "vmlinux_wa", during the gem5 simulation. This file is provided with gem5-X under full_system_images/binaries
- Use the following additional parameter when launching the simulation

  ```
  workload-automation-vio=<FULL_PATH_TO_SHARED_FOLDER_ON_HOST_SYSTEM>
  ```

- Once the system is booted, run the following in gem5 terminal

  ```
  mount.sh <FULL_PATH_TO_SHARED_FOLDER_ON_HOST_SYSTEM>
  ```

- Now any file under the "SHARED_FOLDER_ON_HOST_SYSTEM" appears in the /mnt directory in gem5 simulation.

3.4 Modifying disk image using QEMU

To run experiments an application and benchmarks in gem5-X, they need to be on the disk image. To to this we need to update and modify the disk image with the applications.

QEMU is used to modify the disk image. If running on an Ubuntu-based host system , the following prerequisites need to be installed.

```
sudo apt-get install qemu qemu-user qemu-system qemu-user-static
```

To mount the image

```
cd <PATH_TO_FULL_SYSTEM_IMAGES>/disks/
mkdir local_mnt
sudo mount -o loop,offset=$((2048*512)) gem5_ubuntu16.img local_mnt
sudo mount -o bind /proc local_mnt/proc
sudo mount -o bind /dev local_mnt/dev
sudo mount -o bind /dev/pts local_mnt/dev/pts
sudo mount -o bind /sys local_mnt/sys
```

Now we *chroot* into the image emulating using QEMU

```
cd local_mnt/
sudo chroot ./
```

At this point we are in the ARMv8 disk image and can now compile or download applications within the image. Since it is a Ubuntu 16.04 image, you can run the following first, before installing any new packages on it,

```
apt-get update
```

When the disk image has been update with the applications or benchmarks, we can exit it and unmount the image.

```
exit
cd ..
sudo umount local_mnt/proc
sudo umount local_mnt/dev/pts
```

```
5 sudo umount local_mnt/dev
6 sudo umount local_mnt/sys
7 sudo umount local_mnt
```

The modified image is now ready to be used for gem5-X simulation with new applications or benchmarks

4 High Bandwidth Memory v2 (HBM2)

High Bandwidth Memeory (HBM) is based on 3D stacked DRAM banks made possible due to Through Silicon Vias (TSVs) achieving a high bandwidth of up to 307.2 GB/s. To implement the functional behavior of the HBM2 memory model in gem5-X, we extend the DRAM controller model of gem5 according to the architectural details of HBM2. To have 8-channels with memory interleaving, we initialized 8 DRAM controllers, each 128 bits wide. We connect all 8 DRAM controllers to a 1024-bit wide system bus, that connects to the cache hierarchy.

To use 8-channel HBM2 in gem5-X full system simulation, with appropriate bus widths throughout the system all the way to the caches, use the following command:

```
cd <path_to_gem5-X>/

./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<full_path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_1cpu.dtb \
-n 1 \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--l2bus-width=128 \
--membus-width=128 \
--mem-type=HBM2_2000_4H_1x128 \
--mem-ranks=1 \
--mem-channels=8 \
--mem-size=4GB \
--sys-clock=1600MHz \
```

No separate software support is required to use HBM2 in FS mode, and hence we are able to boot the Ubuntu Linux distribution using HBM2.

The HBM2 memory model can be found in the following file

```
<full_path_to_gem5-X>/src/mem/DRAMCtrl.py
```

5 Core Clustering

Core clustering enables group of compute cores to have their own shared cache, which can be last level cache (LLC), separate from other cores in the system. This reduces the shared resources between different compute clusters in the system to just cross bar interconnect and memory. In addition, clustering is also used when different type of cores are used in system. Same core types are clustered together with their own LLCs. This enables to have a heterogeneous system.

Cluster in now supported in gem5-X. To have different core clusters in gem5-X, use the following command:

```
./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_<NUM_CORES>cpu.dtb \
-n <NUM_OF_CORES>  \
--disk-image=gem5_ubuntu16.img  \
--caches  \
--l2cache  \
--l1i_size=32kB \
--l1d_size=32kB  \
--l2_size=1MB  \
--l2_assoc=2  \
--l2_cluster_size=<NUM_OF_CORE_PER_CLUSTER> \
--mem-type=DDR4_2400_4x16 \
--mem-channels=4 \
--mem-ranks=4 \
--mem-size=4GB \
--sys-clock=1600MHz
```

This command will simulate a system with core clusters. Each cluster will have number of cores defined in *–l2_cluster_size* parameter. The number of cores defined by *-n* parameter should be divisible by the *–l2_cluster_size*. Dividing *n* by *l2_cluster_size*, gives the number of clusters in the system. Each cluster will have its own L2 (LLC) cache.

6 Heterogeneous Cores

Heterogeneity enables different workloads with vartying performance and energy constraints to be allocated to different core types in the system. Gem5-X supports both in-order and OoO cores in the same system. Different core types are distributed into different clusters.

To use heterogeneity in gem5-X, first the system is launched to boot up the linux to reach the region-of-interest (ROI), with the following command:

```
./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_<NUM_CORES>cpu.dtb \
-n <NUM_OF_CORES> \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--l2_cluster_size=<NUM_OF_CORE_PER_CLUSTER> \
--cluster_size_1=4 \
--mem-type=DDR4_2400_4x16 \
--mem-channels=4 \
--mem-ranks=4 \
--mem-size=4GB \
--sys-clock=1600MHz
```

This command will simulate a system with core clusters. The parameter *–cluster_size_1* defines the size of the 1st cluster of type 1. This should be the same as *–l2_cluster_size*. All the cores in the remaining clusters will be of type 2. For instance, if number of cores is defined to be 16, and both *–l2_cluster_size* and *–cluster_size_1* are set to 4, this implies to have 4 clusters in the system, each with 4 cores. The first cluster will have cores of type 1 and the remaining three clusters will have cores of types 2.

Once the ROI is reached, take a checkpoint using ”m5 checkpoint” command. Then one can resume from the checkpoint with the desired core types for each cluster. For the above code type 1 cores are set to be in-order and type 2 to be OoO, as in the following command:

```
./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_<NUM_CORES>cpu.dtb \
```

```
 9 -n <NUM_OF_CORES> \
10 --disk-image=gem5_ubuntu16.img  \
11 --caches  \
12 --l2cache  \
13 --l1i_size=32kB \
14 --l1d_size=32kB  \
15 --l2_size=1MB  \
16 --l2_assoc=2  \
17 --l2_cluster_size=<NUM_OF_CORE_PER_CLUSTER> \
18 --cluster_size_1=4 \
19 --mem-type=DDR4_2400_4x16 \
20 --mem-channels=4 \
21 --mem-ranks=4 \
22 --mem-size=4GB \
23 --sys-clock=1600MHz \
24 -r 1 \
25 --cpu-type=MinorCPU \
26 --cpu-type_2=DerivO3CPU \
```

7 Scratchpad Memory (SPM)

Scratchpad Memories (SPMs) are software programmable memories at the same level as L1 cache, but controlled by the user. Gem5-X supports SPMs, which are both local and shared between two consecutive cores.

To use SPMs gem5-X, the following command can be used:

```
./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_<NUM_CORES>cpu.dtb \
-n <NUM_OF_CORES> \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--mem-type=DDR4_2400_4x16 \
--mem-ranks=4 \
--mem-size=4GB \
--sys-clock=1600MHz \
--spm \
--d_spm_size=128kB
```

The *–spm* commands enabes SPM in gem5-Xand *–d_spm_size* defines the SPM size, which is set to 128KB in the above example. The SPMs can be accessed by two consecutive cores. For instance, SPM0 is accessible by core0 and core1, SPM1 by core1 and core2, SPM2 by core2 and core3 and so on.

Since this is a FS mode of gem5-X, to use SPM, they need to be mapped using mmapp, as in the following code:

```
void * spm_mem_alloc (uint64_t mem_size, uint64_t mem_address)
{

        uint64_t alloc_mem_size, page_mask, page_size;
        void *mem_pointer;
        void *virt_addr;

        page_size = sysconf(_SC_PAGESIZE);
        alloc_mem_size = (((mem_size / page_size) + 1) * page_size);
        page_mask = (page_size - 1);

        int mem_dev = open("/dev/mem", O_RDWR | O_SYNC);
        if(mem_dev == -1)
```

```
14          {
15                  perror("Cannot open /dev/mem \n");
16                  //return -1;
17
18          }
19
20          mem_pointer = mmap(NULL,
21                          alloc_mem_size,
22                          PROT_READ | PROT_WRITE,
23                          MAP_SHARED,
24                          mem_dev,
25                          (mem_address & ~page_mask)
26          );
27
28          if(mem_pointer == MAP_FAILED)
29          {
30              perror("Cannot MAP \n");
31              //return -1;
32          }
33
34          printf("Memory Mapped \n");
35          virt_addr = (mem_pointer + (mem_address & page_mask));
36
37
38      return virt_addr;
39  }
```

The above core snippet returns a virtual pointer in SPM in FS mode. The parameter *uint64_t mem_size* is used to define the size of memory allocated within SPM. The parameter *uint64_t mem_address* defines the memory address of the SPM in physical memory space. So for SPM0 this should be at an offset after the main memory and I/O devices in gem5-X. So for instance of the main memory size is 4GB, the offset for SPM0 should be 4GB+2GB(I/O devices memory space), i.e. 6GB=6442450944. SPM1 should be at an offset defined by main-memory size + I/O devices + SPM0_size.

8 ALPINE: Analog In-Memory Computing Tile Model and Interfaces

In this chapter, we describe how to retrieve, utilize, and program applications for the ALPINE Analog In-Memory Computing extension of gem5-X. This extension was the basis for the exploration presented in "ALPINE: Analog In-Memory Acceleration with Tight Processor Integration for Deep Learning", by Klein et al., published in IEEE Transactions on computer, 2023.[1]

8.1 Retrieving and running gem5-X Full System Mode with ARMv8, Linux, and ALPINE

The guide for setting up, running, and utilizing the ALPINE extension in gem5-X follows that described in Section 2. Table 1 reports the compatibility of gem5-X-ALPINE with respect to other gem5 extensions in gem5-X. No guarantees of compatibility with any present or future gem5-X version should be assumed beyond the ones provided in this table,

Table 1: ALPINE/gem5-X Compatibility Chart

Extension	Section	Compatible with ALPINE?	Notes
Support Enhancements	3	**Yes**	
HBM2	4	**Yes**	
Core Clustering	5	**Untested**	
Heterogeneous Cores	6	**Untested**	
SPM	7	**Yes**	

ALPINE can be cloned from the associated repository via

```
1 git clone https://github.com/gem5-X/ALPINE.git
```

From this point, the gem5 binary can be compiled and run with all of the materials and steps described in Section 2 after changing directory from master to *gem5-X-ALPINE*.

8.2 Configuration Parameters

some parameters/configuration options lack scripting support, and should therefore be done before the gem5 binary is compiled with the scons script, as reported in Section 2. They are described in the rest of this section.

8.2.1 Operation Latency of Custom Instructions

In gem5, the time an instruction takes to execute is determined by its operation latency within the CPU model's functional unit multiplied with the CPU core frequency (when not accounting for any sort of blocking operations). The operation latency for the custom instructions used by ALPINE is defined in src/cpu/minor/MinorCPU.py, lines 142, 147, 152, 157, and 162:

```
1 class MinorDefaultCusProcessFU(MinorFU):
2     opClasses = minorMakeOpClassSet(['CusAluProcess'])
3     timings = [MinorFUTiming(description="CusProcess",
4         srcRegsRelativeLats=[2])]
```

[1] Klein, Joshua et al., https://arxiv.org/abs/2205.10042 accessed October 31, 2022.

```
    opLat = 200
class MinorDefaultCusQueueFU(MinorFU):
    opClasses = minorMakeOpClassSet(['CusAluQueue'])
    timings = [MinorFUTiming(description="CusQueue",
        srcRegsRelativeLats=[2])]
    opLat = 2
class MinorDefaultCusDequeueFU(MinorFU):
    opClasses = minorMakeOpClassSet(['CusAluDequeue'])
    timings = [MinorFUTiming(description="CusDequeue",
        srcRegsRelativeLats=[2])]
    opLat = 2
class MinorDefaultCusParamReadFU(MinorFU):
    opClasses = minorMakeOpClassSet(['CusAluParamRead'])
    timings = [MinorFUTiming(description="CusParamRead",
        srcRegsRelativeLats=[2])]
    opLat = 2
class MinorDefaultCusParamWriteFU(MinorFU):
    opClasses = minorMakeOpClassSet(['CusAluParamWrite'])
    timings = [MinorFUTiming(description="CusParamWrite",
        srcRegsRelativeLats=[2])]
    opLat = 2
```

In the example below, assuming a CPU clock frequency of 2GHz, processing on the AIMC would therefore take 100ns, since opLat equals 200 and 200 / 2GHz = 100ns.

8.2.2 Configuring Tile Generation

In the default ALPINE configuration, AIMC tiles are generated using the following code in src/dev/arm/aimc_cluster.cc (line 35):

```
// Constructor.
AIMCCluster::AIMCCluster(
        const AIMCClusterParams * p) :
        BasicPioDevice(p, p->pio_size),
        system(dynamic_cast<ArmSystem *>(p->system))
{
        warn("AIMC tile instantiated.");

    this->pioAddr = p->pio_addr;
    this->pioSize = p->pio_size;

        for (auto cpu : p->cpus) {
        cpus.push_back(cpu);
        tiles.push_back(new AIMCTile());
    }
}
```

The for loop in the code above creates one new AIMC tile object for every CPU on the system. These are then placed in a vector, and accessed by custom instructions, as defined below in this

Section. Custom instructions access the vector of AIMCs using the CPU number as an index, as specified in src/arch/arm/isa/insts/. The AIMC instantiation (src/dev/arm/aimc_cluster.cc) and the instruction set extensions definition (src/arch/arm/isa/insts/data64.isa) should therefore be jointly modified to modify the integration strategy of AIMC tiles, for example by only providing few cores with an AIMC accelerator, or instantiating multiple tiles for each core.

8.3 ALPINE AIMC Tile Model Implementation

The AIMC tile model in gem5-X and ALPINE is comprised of four main components: custom instruction definitions, the custom instruction to PIO device interface, the AIMC tile wrapper, and finally, the AIMC tile model itself.

8.3.1 ALPINE ISA Extension

Five custom instructions are defined for the tightly-coupled interface, shown in Table 2 note that "X" refers to a "don't care" value). All instruction functionality is implemented in src/arch/arm/isa/insts/data64.isa.

Table 2: ALPINE Custom Instruction Definitions

gem5 Mnemonic	OpCode	R_m	R/W	R_a	R_n	R_d
cmprocess	0x00C	X	0x0	X	X	X
cmqueue	0x10C	Packed Input	0x1	X	X	X
cmdequeue	0x10C	X	0x0	X	X	Packed Output
cmparamread	0x20C	Tile Column	0x1	X	Tile Row	Parameter
cmparamwrite	0x20C	Tile Column	0x0	Parameter	Tile Row	Success

All instructions are 32-bit three-register R-type instructions with 11-bit op-codes, 5-bit indexing for registers, and 1-bit R/W. While most of these instructions don't actually leverage all three registers for operations, the decision to keep all instructions as R-type is mostly for extensability purposes. A detailed description of each of the instructions' behaviour is described below (note that the specific AIMC tile selected is the single AIMC Tile associated with each CPU thread):

- *cmparamread*: : Write the 8-bit parameter held in register R_a in the AIMC tile weight at index R_n, R_m.
- *cmparamwrite*: : Read and return the 8-bit parameter into register R_d from the tile weight at index R_n, R_m.
- *cmqueue*: Write 32 bits (corresponding to four 8-bits values) to the input register of the AIMC. No explicit index in the input register is required, as this is automatically auto-incremented when queuing values
- *cmdequeue*: Read in register R_d 32-bits from the AIMC output register, corresponding to four 8-bits values. Again, the index in the output register is auto-incremented upon a dequeue command.
- *cmprocess*: Performs the Matrix-Vector Multiplication operation by multiplying-and-accumulating each crossbar matrix column with the contents of the input memory and storing the truncated

8-bit output in the AIMC tile output memory. This operation also refreshes (sets to 0) the AIMC tile's input memory contents.

All instructions are implemented as wrappers to methods implemented in the AIMC Tile Wrapper Object. Therefore, the general implementation of each instruction simply gets a pointer to the AIMC Tile Wrapper, formats the instruction arguments, and then performs the associated AIMC Tile Wrapper method. We use the thread ID to access an AIMC tile, as we assume one private to each CPU core. As mentioned above, other integration schemes can be realized by modifying the tile generation mechanism

Finally, the timings model for the AIMC tiles is implemented as the operation latency for each instruction. For example, a system with in-order CPU models running at a frequency of 2GHz that has an AIMC tile MVM latency of 100ns would need to set the *cmprocess* instruction operation latency to 200 cycles. In ALPINE, this is done in src/cpu/minor/MinorCPU.py, as shown below:

```
class MinorDefaultCusProcessFU(MinorFU):
opClasses = minorMakeOpClassSet(['CusAluProcess'])
timings = [MinorFUTiming(description="CusProcess",
    srcRegsRelativeLats=[2])]
opLat = 200
```

Note that the gem5-X/ALPINE binary needs to be recompiled whenever the opLat value changes in order for the custom instruction latency changes to be realized during simulation.

8.3.2 ISA-to-Device Interface Connection

In order to provide the tightly-coupled interface to the AIMC Tile Wrapper Object, the gem5 system object (src/arch/arm/system.hh) is connected to both the wrapper object and ISA templates directly. The system object is included in the ISA templates by including the aforementioned system.hh file as well as /src/dev/arm/aimc_cluster.hh to /src/arch/arm/isa/includes.isa. The system object then includes a pointer to the AIMC wrapper object that allows the custom instructions to query the wrapper object.

8.3.3 AIMC Tile Wrapper Object and PIO Device

To configure, generate, and access the individual AIMC tile models, the AIMC Tile Wrapper object, otherwise referred to as the AIMC cluster (src/dev/arm/aimc_cluster.hh), is responsible for the placement of and delegation of tasks to the AIMC tiles. It is configured as a gem5 peripheral input/output (PIO) device and sits atop the ARM Realview Platform (src/dev/arm/Realview.py):

```
class AIMCCluster(BasicPioDevice):
    type = 'AIMCCluster'
    cxx_header = "dev/arm/aimc_cluster.hh"
    pio_addr = Param.Addr(0x10020000, "Address for AIMC core access.")
    pio_size = Param.Int32(0x1000, "Size of AIMC memory-mapped address
        range.")
    cpus = VectorParam.BaseCPU("CPUs/hardware threads attached to this
        device.")
```

As required by PIO devices, the AIMC cluster is placed in the address range of 0x10020000 : 0x10021000. The "cpus" parameter is used to generate AIMC tiles; the implementation of ALPINE

generates one AIMC Tile per CPU on the simulated MPSoc. This can be reconfigured by modifying the source code in src/dev/arm/aimc_cluster.cc.

8.3.4 AIMC Tile Model

The individual AIMC tiles generated by the AIMC cluster are implemented as structs in src/dev/arm/aimc_cluster.hh:

```
1  struct AIMCTile {
2      // Constructor.
3      AIMCTile() :
4      crossbarHeight(2000),
5      crossbarWidth(2000),
6      crossbar(new int8_t[crossbarWidth*crossbarHeight]),
7      inputMemory(new int8_t[crossbarHeight]),
8      outputMemory(new int8_t[crossbarWidth]),
9      inputMemoryCounter(0),
10     outputMemoryCounter(0),
11     vectorization(4)
12     {
13         for (int i = 0; i < crossbarHeight; i++) {
14             inputMemory[i] = 0;
15             for (int j = 0; j < crossbarWidth; j++) {
16                 crossbar[(i * crossbarWidth) + j] = 0;
17             }
18         }
19
20         for (int i = 0; i < crossbarWidth; i++) {
21             outputMemory[i] = 0;
22         }
23     }
24
25     const int crossbarHeight;    // Height of input memory.
26     const int crossbarWidth;     // Width of output memory.
27     int8_t * crossbar;           // Parameters crossbar.
28     int8_t * inputMemory;        // Input memory (pre-DAC).
29     int8_t * outputMemory;       // Output memory (post-ADC).
30     int inputMemoryCounter;      // Index into input memory.
31     int outputMemoryCounter;     // Index into output memory.
32     const int vectorization;     // How many values do we queue/dequeue?
33 };
```

The struct holds the parameter corresponding to the emulated tile physical dimensions (height and width i.e., number of rows and columns). Furthermore, it stores the state of the AIMC accelerator: the values of its stored weights and that of the input and output registers. It is also in charge of auto-incrementing counters in the input and output registers upon a queue or dequeue operation.

8.4 AIMClib

AIMClib, the header-only C/C++ software library used to easily interface the AIMC cluster, is held in a folder separate from ALPINE in master/aimclib. AIMClib encapsulates the intrinsics necessary to perform all of the custom instructions as well as basic methods for queuing and dequeuing larger data structures such as arrays and vectors. Furthermore, it includes a C++ "checker" module that is used to debug AIMClib implementations by emulating gem5/ALPINE software behaviour.

To use AIMClib in a C program, simply include aimc.hh in the source code of the desired program and the function prototypes provided by AIMClib are available. To use the checker module specifically, the option "-DUSE_CHECKER" should be included in the gcc/g++ compilation. The most relevant function prototypes are included below:

```
1  // Write to AIMC tile crossbar.
2  inline void mapMatrix(int aimc_x, int aimc_y, int height, int width,
3      int8_t ** m);
4  inline void mapMatrix(int aimc_x, int aimc_y, int height, int width,
5      int8_t * m);
6
7  // Queue/dequeue to/from AIMC tile input/output memories.
8  inline void queueVector(int size, int8_t * v);
9  inline void dequeueVector(int size, int8_t * v);
10
11 // Perform MVM.
12 inline void aimcProcess();
```

In addition to the base prototypes listed above, these methods are also templated in case casting from a higher-precision data type to *int8_t* is required.

8.5 ALPINE Sample Application

We provide the application code to perform inference on a 1024x1024 Perceptron. It is written in C++ code using AIMCLib calls to interface the accelerator module. This application can be compiled and run on a gem5-X/Alpine instance as specified in Section 2.2. The application code is available in the gem5-X-ALPINE repository as aimclib/example.cc.

```
1  #include "aimc.hh"
2
3  int main(int argc, char * argv[])
4  {
5      // Test bench parameters.
6      int n_x         = 1024; // MLP input/output dimensions.
7      int T_x         = 10;   // Number of inferences.
8
9      // Set up and initialize vectors/matrices.
10     int8_t ** input     = new int8_t*[T_x];
11     int8_t * W1         = new int8_t[n_x*n_x];
12     int8_t ** output    = new int8_t*[T_x];
13     for (int i = 0; i < T_x; i++) {
14         input[i] = new int8_t[n_x];
```

```
15          output[i] = new int8_t[n_x];
16          for (int j = 0; j < n_x; j++) {
17              input[i][j] = (int8_t)rand();
18              output[i][j] = (int8_t)rand();
19          }
20      }
21      for (int i = 0; i < n_x*n_x; i++)
22          W1[i] = (int8_t)rand();
23
24      // Map weights to AIMC tile.
25      mapMatrix(0, 0, n_x, n_x, W1);
26
27      // Do inference.
28      for (int i = 0; i < T_x; i++)
29      {
30          // Queue input for next inference in first layer.
31          queueVector(n_x, input[i]);
32
33          // Do MVM.
34          aimcProcess();
35
36          // Dequeue output from AIMC tile MVM.
37          dequeueVector(n_x, output[i]);
38      }
39
40      // Cleanup and return.
41      for (int i = 0; i < T_x; i++) {
42          delete[] input[i];
43          delete[] output[i];
44      }
45      delete[] input;
46      delete[] output;
47      delete[] W1;
48
49      return 0;
50  }
```

A dedicated compilation script (aimclib/build_example.sh) is also provided. The application can be compiled with or without the -DUSE_CHECKER option. Using the option, compiled binaries will use stand-in behavioral models for performing AIMClib calls. Otherwise, binaries will target the gem5-X + ALPINE system simulation.