Sep 26 - Oct 03

- Working to get workstation set up

Tasks:

- Finalize workstation setup
- Identify related papers to GPU architecture/programming
    - Modern GPUs with tensor cores
    - Old GPUs (early nvidia papers?)
    - Early papers on GPU programming/architecture
    - Main point: Need to be able to understand and explain the GPU architecture and programming model

Papers & Notes

Brook:

Buck, I., Foley, T., Horn, D., & Hanrahan, P. (2004). **Brook for GPUs: Stream Computing on Graphics Hardware**. In *ACM Transactions on Graphics (TOG)* (Vol. 23, No. 3, pp. 777-786). ACM Press.

Stream Programming

Example Code:

```
// Kernel function for SAXPY (Single-Precision A * X Plus Y)
kernel void saxpy(float a, float4 x<>, float4 y<>, out float4 result<>) {
   // Input Stream: x<>, y<> (both x and y are input streams)
   // Perform the SAXPY operation: result = a * x + y
   result = a * x + y;
}
```

```
void main(void) {
   float a;                    // Scalar value
   float4 X[100], Y[100], Result[100]; // Arrays in memory (CPU-side)
   float4 x<100>, y<100>, result<100>; // Streams in GPU memory (Stream variables)

   // Initialize variables: a, X, Y
   ... initialize a, X, Y ...

   // Input Stream: Read data from memory (X, Y) into input streams (x<>, y<>)
   streamRead(x, X); // Copy data from CPU memory X to GPU stream x
   streamRead(y, Y); // Copy data from CPU memory Y to GPU stream y

   // Execute the SAXPY kernel on all elements in parallel
   saxpy(a, x, y, result); // Process each element: result = a * x + y

   // Output Stream: Write the result back from the stream (result<>) to memory (Result)
   streamWrite(result, Result); // Copy data from GPU stream result to CPU memory Result
}
```

1. What is `float2`, `float3`, and `float4`?

- **`float2`, `float3`, and `float4`** represent vectors with 2, 3, or 4 floating-point components, respectively.
- These are commonly used in **GPU programming** to store data like **positions, directions, colors**, or **texture coordinates**.
- **Examples**:
    - `float2`: Used for 2D positions or texture coordinates.
    - `float3`: Used for 3D vectors like positions or normals.
    - `float4`: Used for homogeneous coordinates or RGBA colors.

2. Kernel Execution in GPU Programming:

- A **kernel** in GPU programming is a function executed by multiple threads in parallel. Each thread typically processes one element of a **stream**.
- For example, if you have an **input stream** of 100 elements and 100 threads, each thread processes a different element.

3. Static and Global variables are not allowed in Kernel.

- Operations that may introduce side-effects between stream elements, such as writing static or global variables, are not allowed in kernels. They could cause **data race**.
- Static variables
    - Declared outside the function: Visible for all functions in that file, and retained.

- Declared inside the function: retained in that function

4. Output resize: repeating and striding:

- If a kernel is called with input and output streams of differing shape, Brook implicitly resizes each input stream to match the shape of the output. This is done by either repeating (123 to 111222333) or striding (123456789 to 13579) elements in each dimension.

5. Gather Streams vs. Input Streams:

- **Input streams (<>)** are sequentially processed by each thread in parallel. Each thread accesses its own element without arbitrary indexing. Example:

```
kernel void addOne(float input<>, out float result<>) {
    result = input + 1.0f;
}

void main(void) {
    float X[100], Result[100];
    ... initialize X ...
    streamRead(input, X);
    addOne(input, result);  // Process each element in X by adding 1
    streamWrite(result, Result);  // Write the result back to memory
}
```

- **Gather streams ([ ])** allow arbitrary access using an index (like an array). All threads can fetch elements from **any location** in memory. Example:

```
kernel void gatherExample(float gatherArray[], out float result<>) {
    float value = gatherArray[5];  // Access the element at index 5
    result = value * 2.0f;
}

void main(void) {
    float A[100], Result[100];
    ... initialize A ...
    gatherExample(A, result);  // Use arbitrary indexing
    streamWrite(result, Result);  // Write the result back to memory
}
```

6. Behavior of Gather Streams:

- When using **gather streams**, threads can fetch the **same element** from the stream but perform **different operations** depending on conditions such as **thread IDs**, and store results into different locations.
- Even though the kernel code is the same for all threads, differences in input data (like thread-specific data or indices) can result in different outcomes for each thread.

7. Thread and Stream Distribution:

- If the number of threads is **less than** the number of elements in the input stream, threads will process **multiple elements** sequentially.
- For example, with 40 threads and 100 elements, some threads process 3 elements while others process 2.

8. Distinction Between Stream and Vector Programming:

- **Vector programming** uses **simple arithmetic operations** on vectors, often requiring intermediate results to be stored in memory (high memory bandwidth).
- **Stream programming with kernels** allows for more **complex operations** and stores intermediate values in **local registers**, reducing memory traffic and increasing computational efficiency.

9. Reduction operations

- Reduction operation to be **associative**: (a ◦ b) ◦ c = a ◦ (b ◦ c). Allows the system to evaluate the reduction in whichever order is best suited for the underlying architecture.
- Both reading and writing to the reduce parameter are allowed when computing the reduction of the two values. Example:

```
reduce float sum = 0;  // Initialize the reduce parameter
kernel void sumReduction(float input<>, reduce float sum) {
    sum = sum + input;  // Add each element of the input stream to the reduce parameter
}
```

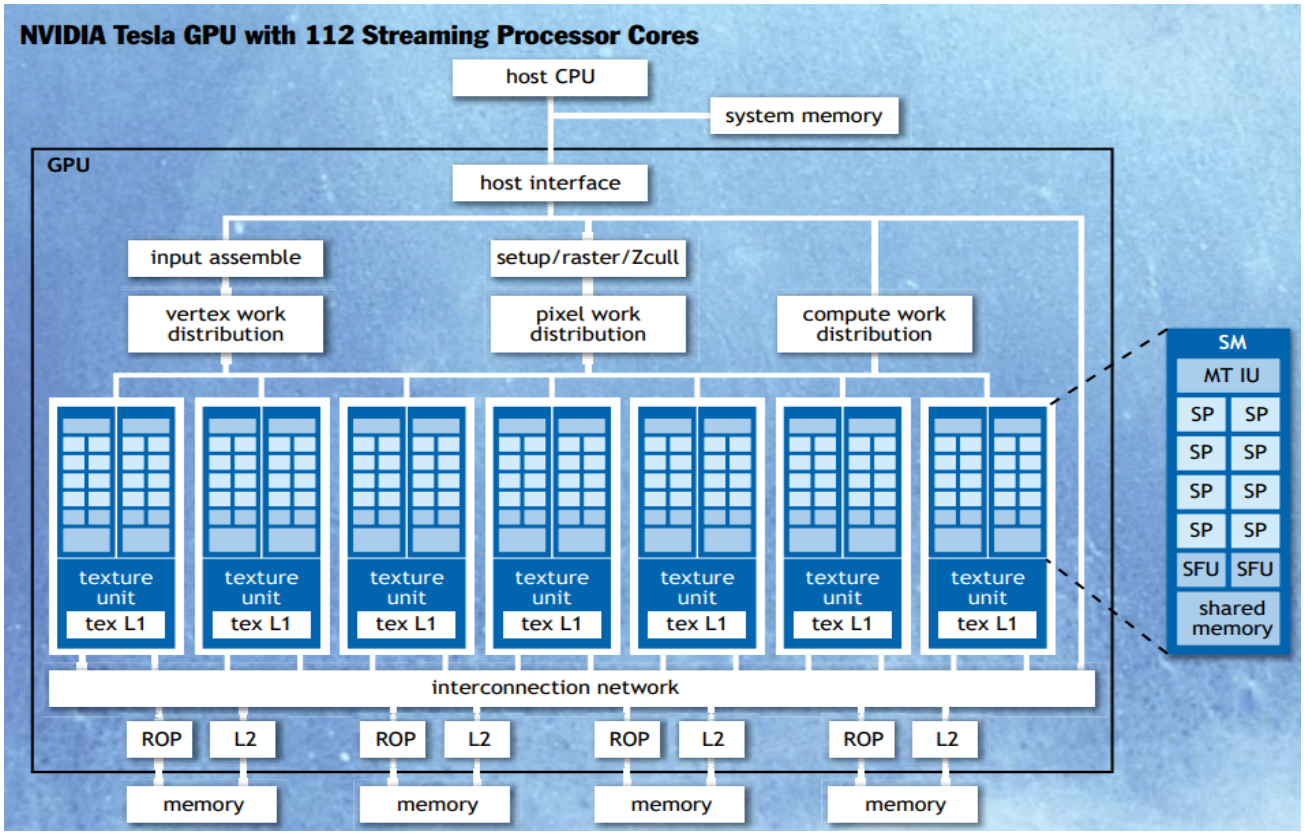
- If the output argument to a reduction is a single element, it will receive the reduced value of all of the input stream's elements. Example:

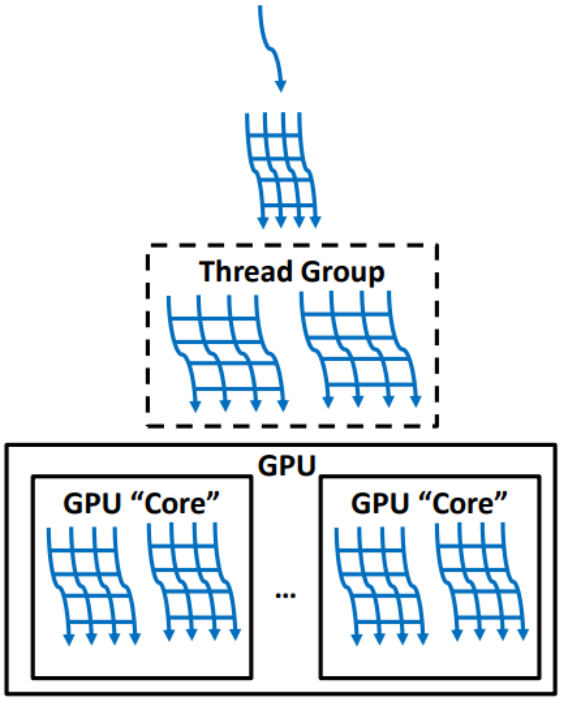
```
// Kernel to perform element-wise multiplication
kernel void mul (float a<>, float b<>, out float c<>) {
    c = a * b;
}

// Reduction kernel to sum the elements
reduce void sum (float a<>, reduce float r<>) {
    r += a;
}
```

```
// Declarations of streams and arrays
float A<50, 50>;   // Matrix A with dimensions 50x50
float x<1, 50>;    // Vector x with dimensions 1x50 (row vector)
float T<50, 50>;   // Temporary matrix T with dimensions 50x50
float y<50, 1>;    // Result vector y with dimensions 50x1 (column vector)

// Example usage of kernels
mul(A, x, T);      // Perform element-wise multiplication between A and x, store result in T
sum(T, y);         // Sum the elements of each row in T and store the result in y
```

CUDA:

Buck, I., Foley, T., Horn, D., & Hanrahan, P. (2004). **Brook for GPUs: Stream Computing on Graphics Hardware**. In *ACM Transactions on Graphics (TOG)* (Vol. 23, No. 3, pp. 777-786). ACM Press.





CUDA/HIP	OpenCL
Thread	Work-item
Warp	Wavefront
Thread Block/CTA	Workgroup
Grid (Kernel)	NDRange (Kernel)

1. CUDA Programming Model Hierarchy:

- **Threads**: The basic unit of execution in CUDA. Each thread runs a specific part of the kernel function.
- **Warps**: Threads are grouped into **warps**, and each warp consists of **32 threads**. Warps are scheduled and executed together in SIMT (Single Instruction, Multiple Threads) fashion. All 32 threads in a warp execute the same instruction at the same time but on different data.
- **Thread Blocks**: A **thread block** is a group of warps (and thus a group of threads). A block can have up to 512 threads (1024 threads from GPT) (depending on hardware), and all threads within a block can communicate with each other using **shared memory**. Independent with each other, cloud schedule in any order, serious or parallel.
- **Grid**: A **grid** is a collection of thread blocks. When a kernel is launched, the grid defines the overall execution, with thread blocks distributed across the GPU's hardware. **Independent.**

2. CUDA program is launched:

- The **CWD unit** distributes the **thread blocks(blockIdx)** of a **kernel grid** to available **SMs**.
- All the threads(threadIdx) within a thread block execute **concurrently** on the same SM.

- When a block finishes execution, the **CWD unit** assigns a new block to the SM that just completed its task, keeping the GPU busy.
- An SM consists of (depends on hardware) eight scalar **SP** cores, two **SFUs** (special function units) for transcendentals, an **MT IU** (multithreaded instruction unit), and on-chip shared memory. The SM creates, manages, and executes up to 768 concurrent threads in hardware with zero scheduling overhead.
    a. SFU for **transcendental functions** such as **sin**, **cos**, **exp**, **log**, etc.
- When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks making up the grid.
- **Grid**: The grid is the collection of thread blocks, distributed across multiple SMs.
- **Thread Blocks**: Thread blocks are mapped to SMs. Each SM can run several thread blocks at a time, depending on the number of threads and warps it can handle.
- **Warps**: The SM schedules **warps** for execution. Each warp contains 32 threads, and multiple warps can execute concurrently on an SM.
    a. **Warp Divergence, e.g. Conditional Branch:** The warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.
- **Question**: If there are only 32 SPs in a SM, how can multiple warps execute concurrently on an SM? As a warp contains 32 threads, one warp has used up all SP resources.

  GPT Answer: Some sort of like multi threads in OS, multiple warps can execute **concurrently** on an SM, but only **one warp's threads are physically executing at a time** on the SP cores. The SM schedules and switches between warps to keep the hardware fully utilized.

- **Threads**: Each thread in a warp is executed by an **SP** (Streaming Processor). If an SM has 8 SPs, it can execute 8 threads from a warp in parallel, and then continue executing the rest of the warp in subsequent cycles.

3. CUDA Programming (C)

- **Thread blocks** and **grids** may have one, two, or three dimensions, accessed via .x, .y, and .z index fields.
    - Example for Thread Blocks: We could map threads in a 2D block to the pixels in an image, where each thread handles one pixel. If the screen has resolution of 1024*1024, each thread block has 16*16 threads, then the grid has the dimension of 64*64. This code below should execute on all threads, where each thread corresponds to a pixel.
    - Code:

```
__global__ void processImage(float* image) {
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  // Process pixel at (x, y)
```

}

- **barrier** by calling the **__syncthreads()** intrinsic.
- **`__shared__`** memory is fast and located **on-chip** in the SM. It is used for **intra-block communication** and is private to each thread block. It's well-suited for storing small data that needs to be shared between threads in the same block.
- **`__device__`** memory (global memory) is located **off-chip** in the GPU's DRAM and is slower to access. It is used for storing large datasets that need to be accessed by all threads across blocks.
- Implicit barrier introduced for dependent grids.
- Each thread has a private local memory. CUDA uses this memory for thread-private variables that do not fit in the thread's registers, as well as for stack frames and register spilling.

Oct 10

- Resolved link errors
- Square workload running!
    - Changes: docker run --volume $(pwd):$(pwd) -w $(pwd) ghcr.io/gem5/gcn-gpu:v24-0 gem5/build/VEGA_X86/gem5.opt gem5/configs/example/apu_se.py -n 3 --dgpu --gfx gfx900 -c gem5-resources/src/gpu/square/bin/square
- Workstation:
    - Setting up with IT, sudo permissions and enough space
- Learning GPUs:
    - Focus on learning pytorch
        - How are pytorch models compiled? (especially ROCm)
- Read:
  Simulation Support for Fast and Accurate Large-Scale GPGPU & Accelerator Workloads. Vishnu Ramadas, Matthew Poremba, Bradford Beckmann and Matthew D. Sinclair. In 3rd Open-Source Computer Architecture Research (OSCAR), June 2024.
- Running square workload in fs mode?

Oct 17

ROCm == Radeon Open Compute
ROCm stack

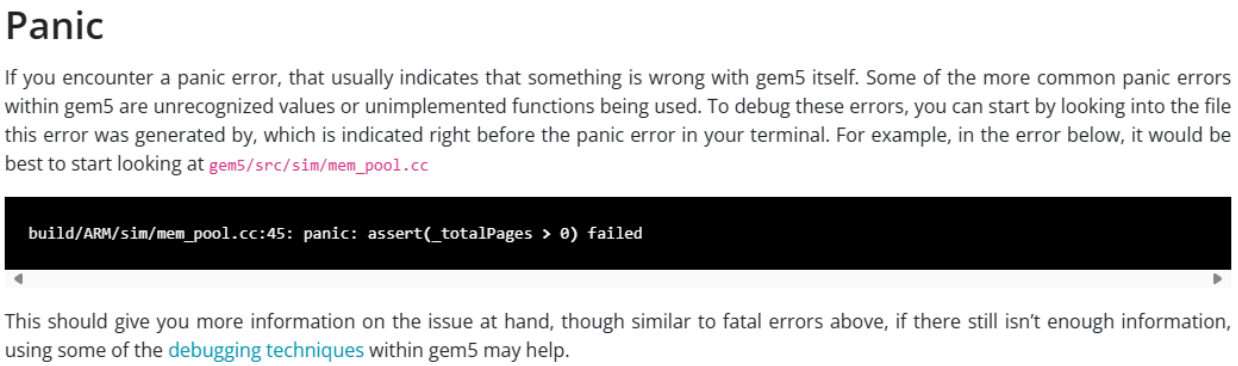
- Runtime layer – ROCr
- Thunk (user-space driver) – ROCt
- Kernel fusion driver (KFD) – ROCk (in Linux)
- MIOpen – machine intelligence (ML) library
- rocBLAS – BLAS (e.g., GEMMs) library
- HIP – GPU programming language (roughly: LLVM backend, clang front-end)

- …

- In SE mode, gem5 simulates all of these except ROCk, which it emulates through docker
    - docker pull ghcr.io/gem5/gcn-gpu:v24-0
    - cd gem5-resources/src/gpu/square
    - docker run --rm -v ${PWD}:${PWD} -w ${PWD} ghcr.io/gem5/gcn-gpu:v24-0 make
    - cd gem5
    - docker run --volume $(pwd):$(pwd) -w $(pwd) ghcr.io/gem5/gcn-gpu:v24-0 scons build/VEGA_X86/gem5.opt -j 14
    - cd ..
    - docker run --volume $(pwd):$(pwd) -w $(pwd) ghcr.io/gem5/gcn-gpu:v24-0 gem5/build/VEGA_X86/gem5.opt gem5/configs/example/apu_se.py -n 3 --dgpu --gfx gfx900 -c gem5-resources/src/gpu/square/bin/square
- In FS mode, the disk image contains the entire ROCm stack and gem5 simulates it
- FS:
    - cd ./meng/2024
    - wget https://storage.googleapis.com/dist.gem5.org/dist/v24-0/gpu-fs/kernel/vmlinux-gpu-ml.gz
    - wget https://storage.googleapis.com/dist.gem5.org/dist/v24-0/gpu-fs/diskimage/x86-ubuntu-gpu-ml.gz
    - cd /2024/gem5/util/m5
    - docker run --volume /root/meng/2024/materials/02-Using-gem5:/root/meng/2024/materials/02-Using-gem5 \
      --volume /root/meng/2024/gem5:/root/meng/2024/gem5 \
      -w /root/meng/2024/gem5/util/m5 \
      ghcr.io/gem5/gcn-gpu:v24-0 scons build/x86/out/m5 -j 14 \
      CXXFLAGS="-I/root/meng/2024/materials/02-Using-gem5/05-cache-hierarchies/ruby-example/workloads/include"
    - cd gem5-resources/src/gpu/square/
    - cp ~/meng/2024/materials/04-GPU-model/Makefile ./
    - docker run --rm -v /root/meng/2024:/root/meng/2024 -w ${PWD} ghcr.io/gem5/gpu-fs:latest make
    - /usr/local/bin/gem5-vega ./gem5/configs/example/gpufs/mi200.py --app ./gem5-resources/src/gpu/square/bin/square --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --no-kvm-per
- MFMA-FS:
    - cp -r materials/04-GPU-model/mfma_fp32/ gem5-resources/src/gpu
    - cd gem5-resources/src/gpu/mfma_fp32
    - docker run --rm -v /root/meng/2024:/root/meng/2024 -w ${PWD} ghcr.io/gem5/gpu-fs:latest make

    - /usr/local/bin/gem5-vega -d mfma-out1 gem5/configs/example/gpufs/mi200.py --kernel ./vmlinux-gpu-ml --disk-image ./x86-ubuntu-gpu-ml --app ./gem5-resources/src/gpu/mfma_fp32/mfma_fp32_32x32x2fp32 --no-kvm-perf
    - MFMA simple register allocation:
        - /usr/local/bin/gem5-vega -d mfma-out1 gem5/configs/example/gpufs/mi200.py --reg-alloc-policy=simple --kernel ./vmlinux-gpu-ml --disk-image ./x86-ubuntu-gpu-ml --app ./gem5-resources/src/gpu/mfma_fp32/mfma_fp32_32x32x2fp32 --no-kvm-perf
        - system.cpu1.shaderActiveTicks 63174997 # Total ticks that any CU attached to this shader is active (Unspecified)
    - MFMA dynamic register allocation:
        - /usr/local/bin/gem5-vega -d mfma-out1 gem5/configs/example/gpufs/mi200.py --reg-alloc-policy=dynamic --kernel ./vmlinux-gpu-ml --disk-image ./x86-ubuntu-gpu-ml --app ./gem5-resources/src/gpu/mfma_fp32/mfma_fp32_32x32x2fp32 --no-kvm-perf
        - system.cpu1.shaderActiveTicks 35069997 # Total ticks that any CU attached to this shader is active (Unspecified)
    - **Register allocation** refers to the process of assigning a program's variables (or intermediate values) to hardware registers in the CPU or GPU during execution.
        - **Static Allocation**: In a simple or static register allocation scheme, variables are assigned to specific registers throughout their lifetime.Once a register is assigned, it holds the same variable for the duration of its usage.
        - **Dynamic or Advanced Allocation**: The assignment of variables to registers can change throughout the program’s execution. Registers may be "spilled" to memory if there aren’t enough available, and data is moved in and out of registers as needed.
        - The **dynamic register allocation** appears to be **better** because it led to a significantly lower number of active ticks (35 million vs. 63 million).
- Check point:
    - cd /workspaces/2024
    - cp materials/04-GPU-model/square-cpt/square.cpp gem5-resources/src/gpu/square/
    - cp materials/04-GPU-model/mi200.py gem5/configs/example/gpufs/
    - cd gem5-resources/src/gpu/square
    - docker run --rm -v /root/meng/2024:/root/meng/2024 -w ${PWD} ghcr.io/gem5/gpu-fs:latest make clean
    - docker run --rm -v /root/meng/2024:/root/meng/2024 -w ${PWD} ghcr.io/gem5/gpu-fs:latest make (I modify the *GEM5_PATH* based on my directory)

- /usr/local/bin/gem5-vega gem5/configs/example/gpufs/mi200.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --app ./gem5-resources/src/gpu/square/bin/square --checkpoint-dir=gpuckpt/
- Unable to generate the checkpoint. “src/cpu/kvm/perfevent.cc:191: panic: PerfKvmCounter::attach failed (2)” Based on GEM5 documents, it says panic indicates that something is wrong with gem5 itself. Still Trying to debug.



Panic

If you encounter a panic error, that usually indicates that something is wrong with gem5 itself. Some of the more common panic errors within gem5 are unrecognized values or unimplemented functions being used. To debug these errors, you can start by looking into the file this error was generated by, which is indicated right before the panic error in your terminal. For example, in the error below, it would be best to start looking at `gem5/src/sim/mem_pool.cc`

```
build/ARM/sim/mem_pool.cc:45: panic: assert(_totalPages > 0) failed
```

This should give you more information on the issue at hand, though similar to fatal errors above, if there still isn't enough information, using some of the debugging techniques within gem5 may help.

- PyTorch:
  - git clone https://github.com/abmerop/gem5-pytorch
  - FS
    - /usr/local/bin/gem5-vega -d pytorch-out gem5/configs/example/gpufs/mi200.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --no-kvm-perf --app gem5-pytorch/pytorch_test.py
    - /usr/local/bin/gem5-vega -d mnist-out2 gem5/configs/example/gpufs/mi200.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --no-kvm-perf --app gem5-pytorch/MNIST/train_1batch/pytorch_qs_mnist.py

Derrick: ROCm stack is complicated, learn piece-by-piece. ROCm contains drivers, libraries, implementations of programming models, etc., it is similar to cuda. You ran SE mode so far, which does all except ROCk. Programming model info was very useful.

- Next turn attention to ROCm stack, starting with ROCr, present the same way as the programming model.

Workstation: laptop works fine to make progress, still working with IT

For milestone 1:

- Setting up and running Gem5-GPU
  - Laptop works fine for making progress today, also will have a server soon, once IT is finished

    - As stated in milestone 1 goals, setup is completed
    - As stated in milestone 1 goals, able to run the square workload without any issues
- Learning GPU architecture and programming
    - GPU programming basics learned
    - GPU architecture basics learned
    - Next: learning the ROCm stack (important because it's used in GEM5-GPU, as well as to understand GPU programming in general)
- Also: learning techniques form https://pages.cs.wisc.edu/~sinclair/papers/vramadas-oscar24-gem5GPUFS.pdf

Oct 24

ROCr

ROCm/ROCR-Runtime: ROCm Platform Runtime: ROCr a HPC market enhanced HSA based runtime

Infrastructure: HSA runtime

- **low level hardware access: user mode queues** provides programmers with a low-latency kernel dispatch interface, allowing them to develop customized dispatch algorithms specific to their application.

- **The HSA Architected Queuing Language** is an open standard, defined by the HSA Foundation, specifying the packet syntax used to control supported AMD/ATI Radeon (c) graphics devices.

- A **dispatch packet** is a specific type of packet that is used to **launch a compute kernel** or task on a GPU (or other processing agent).

- HSA runtime exposes various virtual address ranges that can be accessed by one or more of the system's graphics devices, and possibly the host. The exposed virtual address ranges either support a **fine grained** or a **coarse grained access**. Updates to memory in a fine grained region are immediately visible to all devices that can access it, but only one device can have access to a coarse grained allocation at a time. Ownership of a coarse grained region can be changed using the HSA runtime memory APIs, but this transfer of ownership must be explicitly done by the host application.

Possible issue

- Each HSA process creates an internal DMA queue, but there is a system-wide limit of four DMA queues. When the limit is reached HSA processes will use **internal kernels** for copies.
- **Internal kernels** refer to **compute kernels** that are launched on the GPU (or other accelerators) to manually handle data transfer between memory regions. This is essentially a fallback mechanism when **DMA queues** are not available.

Tasks:

- Building a test setup:
  - Update the v24-0gpu-fs disk image with llama 3.1-1B files
    - https://huggingface.co/meta-llama/Llama-3.2-1B (under files and versions tab)
    - Figure out what we need to do to put the model into the gem5-gpu base disk image
  - I'm not sure if this model is too large
  - Info: https://www.gem5.org/documentation/general_docs/fullsystem/disks
- Learning: continue learning ROCm
  - Give overview of what ROCr does

Oct 31

Run Llama on local machine (Ask for authorization of the development group **first!**):

- apt install python3.12-venv
- python3 -m venv ~/myenv
- source ~/myenv/bin/activate
- pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
- pip install --upgrade transformers

```
import torch
from transformers import pipeline
from huggingface_hub import login

login(token = 'your_token')

model_id = "meta-llama/Llama-3.2-1B"
```

```
pipe = pipeline(
    "text-generation",
    model=model_id,
    torch_dtype=torch.bfloat16,
    device_map="auto"
)

# Set pad_token_id to eos_token_id to avoid padding warning
pipe.model.config.pad_token_id = pipe.model.config.eos_token_id

# Generate text with max_new_tokens to control length
output = pipe("What is AI? ", max_new_tokens=50)

# Print output
print(output)
```

- pip install "accelerate>=0.26.0"

RUN: (myenv) root@chenxinROG:~/meng/2024/Llama-3.2-1B# python3 test.py
Setting `pad_token_id` to `eos_token_id`:None for open-end generation.
/root/myenv/lib/python3.12/site-packages/transformers/generation/utils.py:1375: UserWarning: Using the model-agnostic default `max_length` (=20) to control the generation length. We recommend setting `max_new_tokens` to control the maximum length of the generation.
  warnings.warn(
[{'generated_text': 'what is AI? AI, Artificial Intelligence, is the study of computers that can learn and think'}]

Run nanoGPT on GEM5

- cd /2024
- mkdir mnt
- mount -o loop,offset=$((2048*512)) ./x86-ubuntu-gpu-ml mnt
- cp -r gem5-pytorch/nanoGPT/nanoGPT-ff/ mnt/root/
- gem5-vega -d tutorial_nanogpt --debug-flags=GPUCommandProc gem5/configs/example/gpufs/mi200.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --app gem5-pytorch/nanoGPT/train-ff.sh --skip-until-gpu-kernel=8 --exit-after-gpu-kernel=9 --no-kvm-perf

Try Llama on GEM5

- sudo umount /root/meng/2024/mnt

- truncate -s +50G ./x86-ubuntu-gpu-ml
- sudo losetup -o $((2048*512)) /dev/loop0 ./x86-ubuntu-gpu-ml
- sudo e2fsck -f /dev/loop0
- sudo resize2fs /dev/loop0
- sudo losetup -d /dev/loop0
- sudo mount -o loop,offset=$((2048*512)) ./x86-ubuntu-gpu-ml /root/meng/2024/mnt
- python3 -m venv ./gem5_env
- source ./gem5_env/bin/activate
- pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/rocm6.2
- pip install --upgrade transformers
- pip install 'accelerate>=0.26.0'
- copy the model from local .cache to the disk image's .cache
- Modify the run script based on the directory of the disk image

/usr/local/bin/gem5-vega -d Llama-out gem5/configs/example/gpufs/mi200.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --no-kvm-perf --app Llama-3.2-1B/run.sh

Fast forwarding:
/usr/local/bin/gem5-vega -d Llama-out gem5/configs/example/gpufs/mi200.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --app Llama-3.2-1B/run.sh --skip-until-gpu-kernel=8 --exit-after-gpu-kernel=9 --no-kvm-perf

ROCr

Install required environment, skip if have installed:

- sudo apt update
- sudo apt install -y cmake
- sudo apt install -y libelf-dev
- sudo apt install -y g++
- sudo apt install -y libdrm-dev

Install rocm-core: ROCm/rocm-core

- git clone ROCm/rocm-core
- cd rocm-core; mkdir -p build ; cd build
- cmake \
  -DCMAKE_CURRENT_BINARY_DIR=$PWD \
  -DCMAKE_CURRENT_SOURCE_DIR=$PWD/../ \
  -DCMAKE_VERBOSE_MAKEFILE=1 \
  -DCMAKE_INSTALL_PREFIX=./ \
  -DCPACK_GENERATOR=DEB \

```
-DCPACK_DEBIAN_PACKAGE_RELEASE="local.9999~20.04" \
-DCPACK_RPM_PACKAGE_RELEASE="local.9999" \
-DROCM_VERSION="5.5.0" \
..
```

- make
- make install
- make package
- Check using:
    - dpkg -I rocm-core_5.5.0.50500-local_amd64.deb
    - dpkg -c rocm-core_5.5.0.50500-local_amd64.deb
    - readelf -d ./lib/librocm-core.so.1.0.50500
- sudo dpkg -i rocm-core_5.5.0.50500-local_amd64.deb

ROCr

- git clone https://github.com/ROCm/ROCR-Runtime
- cd ROCR-Runtime
- mkdir build && cd build
- sudo apt install -y pkg-config (in case of haven't installed)
- sudo apt install -y libnuma-dev (in case of haven't installed)
- sudo apt install -y clang (in case of haven't installed)
    - sudo apt install -y libclang-dev
- cmake -DCMAKE_INSTALL_PREFIX=/opt/rocm/llvm ..
-

Derrick:

- Familiarize with transformers:
    - https://www.youtube.com/watch?v=wjZofJX0v4M
    - https://www.youtube.com/watch?v=eMlx5fFNoYc

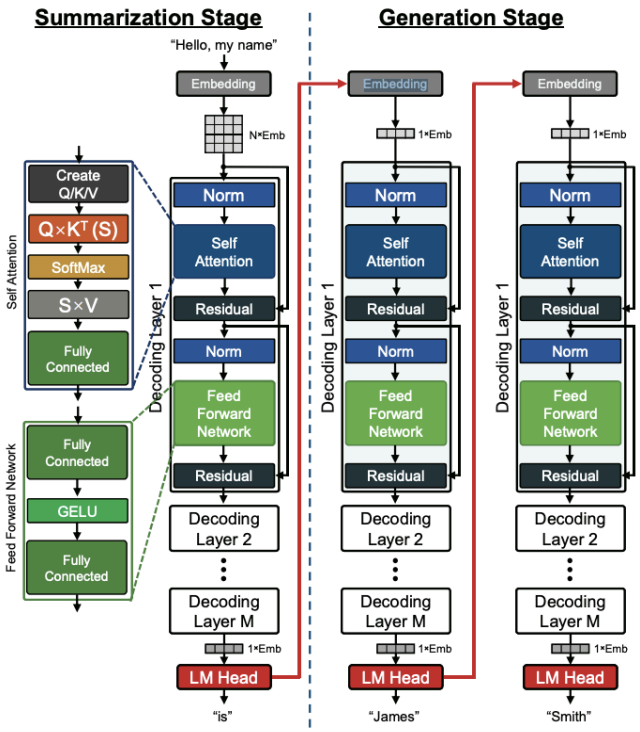


Fig. 1: GPT-3 architecture and inference process.

- Goal: identify the kernels associated a single pass through the transformer stack.
  - Run all kernels with KVM, find the first kernel that runs after a token is generated
    - See whether or not we can count the number of kernels without simulating them in Gem5
  - This way, we can measure the architectural stats of generating a single token with an LLM
  - For now, ignore summarization stage, and just capture kernels for one pass in the generation stage
  - Currently: finding the kernels associated with generation of a single token
  - Next: adding retrieval

Nov 7

Interactive:

- cd gem5/util/term/
- make
- ./gem5/util/term/m5term 3456
- ./gem5/util/term/m5term 7000

Fix mounted directory:

[1.008032] Kernel Offset: disabled
[1.008032] ---[end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(8,1)]---

PS D:\Tools\kernel_imgs> qemu-img info x86-ubuntu-gpu-ml
image: x86-ubuntu-gpu-ml
file format: raw
virtual size: 105 GiB (112407347200 bytes)
disk size: 105 GiB
Child node '/file':
filename: x86-ubuntu-gpu-ml
protocol type: file
file length: 105 GiB (112407347200 bytes)
disk size: 105 GiB

Install library:

- Python virtual environment does not work.
- Convert image to vdi:
  - convert -f raw -O vdi x86-ubuntu-gpu-ml x86-ubuntu-gpu-ml.vdi
- Run on virtual box:
  -

Dec 5:

- To move to the full RAG pipeline, I (Derrick) am providing a RAG setup that we can put into gem5. This requires a new index, which is not yet ready (early next week)
  - Corpus: un-segmented MSmarco: https://huggingface.co/datasets/microsoft/ms_marco

    - Using a 64-dimension embedding model
    - https://microsoft.github.io/msmarco/
        - This includes a set of questions with answers, as well as a set of passages (8.8 million), making it a good choice for evaluating RAG pipelines
    - This is similar to the wikipedia dataset (35 million passages), but smaller (when not segmented), and more popular
- Summary of RAG systems:
    - Retrieval phase first, then generation
    - Retrieval allows for quality and performance to be controlled, these are dependent on each other
    - Generation also allows for this
- Reading (learning about RAG pipelines):
    - ASPLOS accelerating RAG paper (ask me)
    - https://arxiv.org/abs/1611.09268
    - https://arxiv.org/abs/2205.01230

Jan 24

- Steps:
    - Evaluate Gem5 GPU model accuracy
        - Can we compare with another simulator? (Derrick needs to find other options)
        - Also compare against the hardware
        - Switch script to vLLM without needing to download
            - Derrick: write script for this
        - Also, get familiar with retrieval
        - Main focus right now is to isolate a layer in the LLM
    - Evaluating RAG
        - Evaluate retrieval phase
            - Retrieval quality, throughput (queries/second), batch size, latency, etc.
        - Evaluate generation phase
            - Throughput (queries/second), batch size, latency
        - If possible, entire system
            - Need to measure accuracy as well
- Tentatively meet twice per week
    - Move friday to earlier, monday later in the day
    - Iteratively refine the report
- https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index

Jan 31

- Able to run the bench_enns workload
- Challenges with running llama in gem5 due to dependencies
    - First option: bash script that does pip install and everything (no internet access unfortunately)
    - Second option: mount image. This doesn't work because the paths are different (can't set up)
    - Third option: download x86-ubuntu-gpu-ml image file
    - Options:
        - Try to create new disk image Creating disk images for full system mode
        - Continue working on venv setup
- Try bench_enns.py in se mode
- Summarize the findings on nanogpt on overleaf

Feb 3

Install new libraries:

- Mount as before
    - sudo mount -o loop,offset=$((2048*512)) ./x86-ubuntu-gpu-ml /root/meng/2024/mnt
- sudo echo -e "127.0.0.1 localhost\n127.0.1.1 chenxinROG" > /etc/hosts
    - If in chroot, remove sudo
    - "chenxinROG" is my host name, change it depending on your own
- Exit chroot:
    - Exit
- sudo mount -o bind /dev/pts mnt/dev/pts
- If missing mnt/dev/mem:
    - sudo mknod -m 660 mnt/dev/mem c 1 1
    - sudo chown root:kmem mnt/dev/mem
- sudo /usr/sbin/chroot mnt /bin/bash

Run RAG on gem5: msmarco_nnomic64

- pip install faiss-cpu
- /usr/local/bin/gem5-vega -d gem5_rag gem5/configs/example/gpufs/mi200.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --no-kvm-perf --app msmarco_nomic64/run.sh

- ./gem5/util/term/m5term 3456
- Run script:

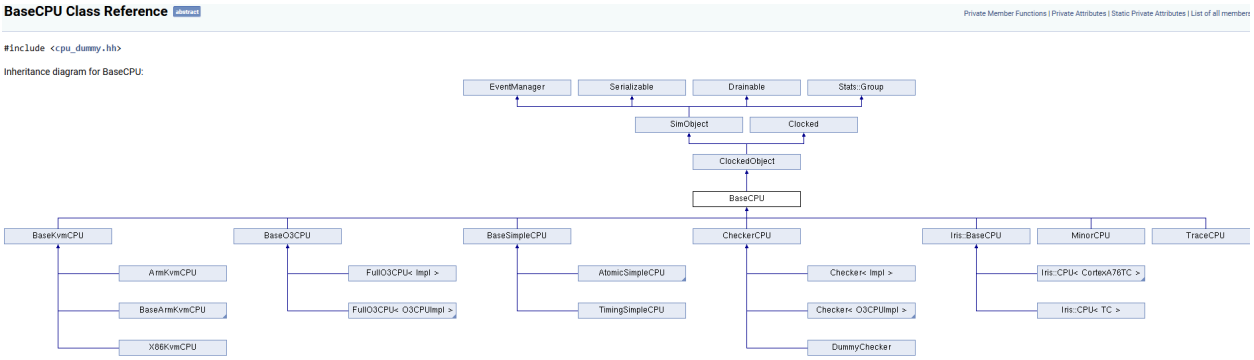
```
cd msmarco_nomic64/
pwd
python3 lib_nns/bench_enns.py --num_queries 2 -b 2
```

- Results(gem5 profiling is stored in gem5_rag folder):

```
src/dev/amdgpu/pm4_packet_processor.cc:346: warn: PM4 packet opcode 0x58 not supported.
src/dev/amdgpu/sdma_engine.cc:901: warn: SDMA poll reg is not implemented. If this is required for c
orrectness, an SRBM model needs to be implemented.
Exiting @ tick 39644313658500 because m5_exit instruction encountered
src/cpu/kvm/base.cc:591: hack: Pretending totalOps is equivalent to totalInsts()
(gem5_env) root@chenxinROG:~/meng/2024#
```

```
Running msmarco_nomic64/run.sh
/root/msmarco_nomic64
100%|                                                  | 1/1 [00:00<00:00,  3.01it/s]
{'avg_time': 0.2760171890258789, 'p50': 0.2760171890258789, 'p99': 0.2760171890258789, 'times_calle
': 1, 'total_time': 0.2760171890258789, 'avg_rate': 3.6229627710114882}
root@chenxinROG:~/meng/2024#
```

- Config the gem5 modeling to different cpu settings.
    - Different number of cores, memory channels, memory frequency?



- With gem5 GPU, only KVM and atomicsimplecpu are supported [gem5: Full System AMD GPU model]
    - Means that you can't accurately simulate CPU-GPU systems with shared memory, such as integrated GPUs or Grace superchips from NVIDIA
    - For shared memory:
        - Need GPU-CPU cache coherence
        - Needs complex datapath for GPU to access CPU memory
        - One potential contribution: propose a standardized design for modeling systems like this. The main challenge is to balance complexity
    - Can't simulate retrieval + generation end-to-end
- Retrieval: use CPU gem5 only, start with SE mode
    - Use DerivO3 CPU
    - Follow basic parameters otherwise: https://www.gem5.org/documentation/gem5-stdlib/hello-world-tutorial

        - Doing soon: configure with parameters(memory channels, freq, caches, etc) for ARM server, x86 server, compare with server results (Derrick will run these)
    - Test flat index for now, then HNSW and IVF
- GPU:
    - Validation: we can report and discuss results, even if it's hard to directly compare (since we only look at one layer right now)
    - Current task is: identifying the kernels for one "layer" of computations in llama 1B model, then use gem5's cycle numbers to figure out the runtime for a layer
        - One layer is the smallest "atomic" component of transformer
        - This allows us to extrapolate
        - We are switching to vllm which is the state-of-the-art
            - Contribution here: provide guide on how to run SOTA LLM engines in gem5 GPU

Feb 13

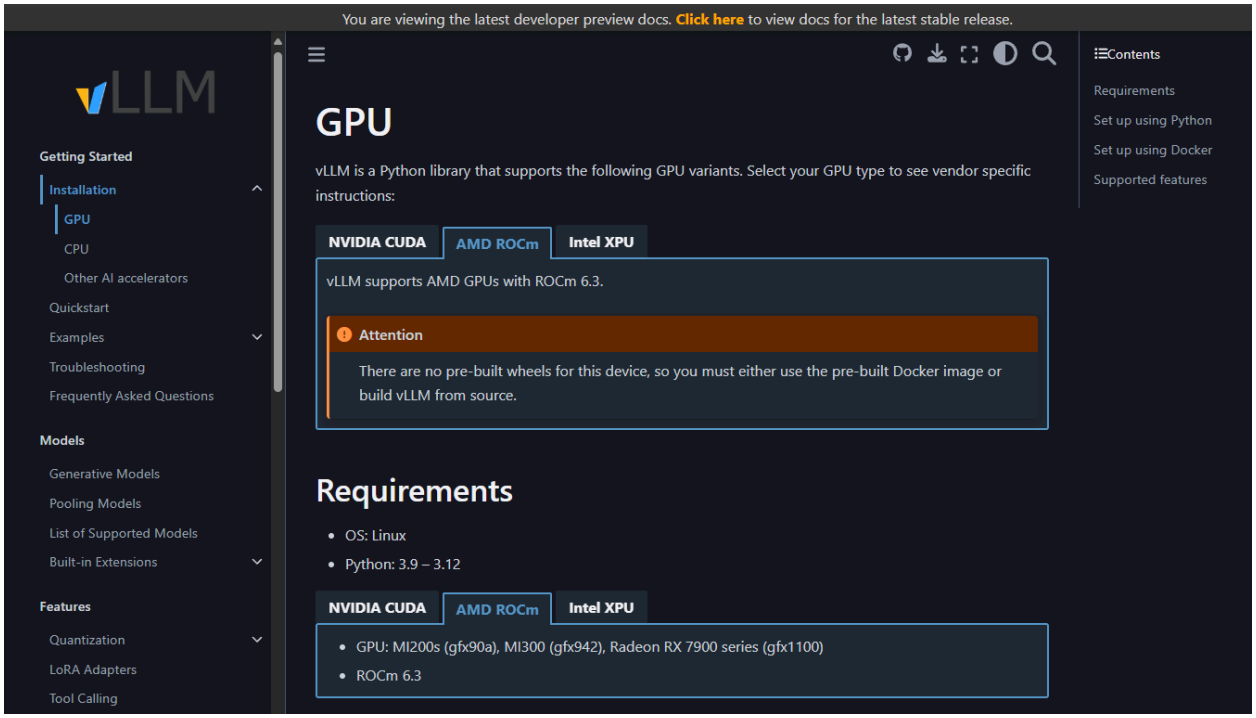
Run Llama on gem5:

- apt update
- apt install python3-venv
- GIT_LFS_SKIP_SMUDGE=1 git clone https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct
- cd Llama-3.2-1B-Instruct
- git lfs pull
- In main.py change to "`vllm_model = LLM(model="./Llama-3.2-1B-Instruct", max_model_len=100000)`"
- (myenv) root@chenxinROG:~/meng/2024# /usr/local/bin/gem5-vega -d vllm_out gem5/configs/example/gpufs/mi200.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --no-kvm-perf --app vllm_gem5-main/run.sh
- Remove all cuda related pip install from the requirement.txt
    - pip uninstall -y nvidia-cublas-cu12 nvidia-cuda-cupti-cu12 nvidia-cuda-nvrtc-cu12 \
    - nvidia-cuda-runtime-cu12 nvidia-cudnn-cu12 nvidia-cufft-cu12 \
    - nvidia-curand-cu12 nvidia-cusolver-cu12 nvidia-cusparse-cu12 \
    - nvidia-ml-py nvidia-nccl-cu12 nvidia-nvjitlink-cu12 nvidia-nvtx-cu12 \
    - triton xformers
- In gem5 root, get gem5 pip list, combine with old requirement.txt to get a new version
- Uninstall CUDA version pytorch:
    - pip uninstall torch torchvision torchaudio -y
- pip install --index-url https://download.pytorch.org/whl/rocm6.1/ torch torchvision torchaudio

- pip freeze | xargs pip uninstall -y
- Install transformer
    - pip install transformers accelerate>=0.26.0
- Pytorch-ROCm installation
    - pip3 install --upgrade pip wheel
    - wget https://repo.radeon.com/rocm/manylinux/rocm-rel-6.2.3/torch-2.3.0%2Brocm6.2.3-cp310-cp310-linux_x86_64.whl
    - wget https://repo.radeon.com/rocm/manylinux/rocm-rel-6.2.3/torchvision-0.18.0%2Brocm6.2.3-cp310-cp310-linux_x86_64.whl
    - wget https://repo.radeon.com/rocm/manylinux/rocm-rel-6.2.3/pytorch_triton_rocm-2.3.0%2Brocm6.2.3.5a02332983-cp310-cp310-linux_x86_64.whl
    - pip3 uninstall torch torchvision pytorch-triton-rocm
    - pip3 install torch-2.3.0+rocm6.2.3-cp310-cp310-linux_x86_64.whl torchvision-0.18.0+rocm6.2.3-cp310-cp310-linux_x86_64.whl pytorch_triton_rocm-2.3.0+rocm6.2.3.5a02332983-cp310-cp310-linux_x86_64.whl
- Install vllm
    - pip install vllm==0.6.6.post1

GPU — vLLM

Releases · ROCm/ROCm



Back to old way

- Create venv
- wget https://repo.radeon.com/rocm/manylinux/rocm-rel-6.2.3/pytorch_triton_rocm-2.3.0%2Brocm6.2.3.5a02332983-cp310-cp310-linux_x86_64.whl
- pip3 uninstall torch torchvision pytorch-triton-rocm
- pip3 install torch-2.3.0+rocm6.2.3-cp310-cp310-linux_x86_64.whl torchvision-0.18.0+rocm6.2.3-cp310-cp310-linux_x86_64.whl pytorch_triton_rocm-2.3.0+rocm6.2.3.5a02332983-cp310-cp310-linux_x86_64.whl
- pip install transformers accelerate
- To avoid the requirement of internet connection during gem5 simulation, you can first run the model on local machine, which will download the model into your local .cache, then copy the model from local .cache to the gem5 disk image's .cache
    -

Llama slice:

- /usr/local/bin/gem5-vega -d Llama-out gem5/configs/example/gpufs/mi300.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --no-kvm-perf --app Llama-3.2-1B/run.sh

nanogpt:

- /usr/local/bin/gem5-vega -d nanogpt-out gem5/configs/example/gpufs/mi300.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --app gem5-pytorch/nanoGPT/train-ff.sh --no-kvm-perf

Gpt2:

- /usr/local/bin/gem5-vega -d gpt2-out gem5/configs/example/gpufs/mi300.py --disk-image ./x86-ubuntu-gpu-ml --kernel ./vmlinux-gpu-ml --no-kvm-perf --app gpt2/run.sh --skip-until-gpu-kernel=8 --exit-after-gpu-kernel=9