Heterogeneous NMP Simulator Documentation

Runyang Tian

1. Overview

This project is a Python-based, event-driven simulator designed to model and analyze the performance and energy consumption of heterogeneous memory systems. It specializes in simulating workloads on an architecture combining 3D DRAM and 3D RRAM (or self-defined memory), each with its own associated compute unit on their logic dies.

The simulator is **JSON-driven**, meaning the model architecture (tensors and operations) is defined in an external JSON file.

The primary goal is to provide detailed statistics on **total latency (cycles)** and **total energy (Joules)**, with comprehensive breakdowns by operation type and hardware component.

2. Core Features

- **Heterogeneous Memory:** Simulates both a 3D DRAM and a 3D RRAM device, each with configurable capacity, bandwidth, latency, and energy parameters.
- **Heterogeneous Compute:** Models distinct Compute Units (CUs) for DRAM and RRAM (`dram_cu`, `rram_cu`), each with its own MAC and Special Function Engine (SFE) performance.
- **Memory Model:** Includes a model for Through-Silicon Vias (TSVs), calculating latency for vertical data transfer between different layers of the 3D memory stack.
- **JSON-Driven:** Models are defined using a simple JSON format, specifying tensors, their properties (shape, bits, device, layer), and the computational graph of operations.
- **Modular & Extensible:** The code is separated into logical modules (hardware, model, loader, compiler, simulator), making it easy to add new operations or modify hardware parameters.
- **Detailed Statistics:** Reports total cycles, energy, and MACs, along with breakdowns by:
    - **Operation Type** (e.g., `matmul_tile`, `geluop`, `ucieop`)
    - **Hardware Component** (e.g., `DRAM Read`, `RRAM Compute`)
- **Supported Operations:** Includes key DL operations like `MatMul` (with tiling), `Conv2D`, `AvgPool2D`, `LayerNorm`, and various element-wise ops (`GeluOp`, `AddOp`). It also supports `ParallelOp` for simulating concurrent execution. It can also support other self-defined operations.
- **Interconnect Modeling:** Features a basic model for UCIe (Universal Chiplet Interconnect Express) for off-chip communication.

3. Project Structure

The simulator is composed of several key Python files:

- `main.py`: The main entry point. This file is responsible for:
  - Defining the hardware parameters (DRAM, RRAM, CUs).
  - Parsing command-line arguments (e.g., the path to the JSON model).
  - Loading the model using `JSONModelLoader`.
  - Running the `SimpleCompiler` to create a schedule.
  - Initializing and running the `Simulator`.
  - Printing the final statistics.
- `simulator.py`: The core simulation engine. The `Simulator` class iterates through the compiled schedule, calculates the latency and energy for each operation using `_calculate_item_cost`, and aggregates the results in the `Stats` object.
- `compiler.py`: Contains the `SimpleCompiler`. This module handles two main tasks:

  1. **Placement:** Allocating tensors to either RRAM or DRAM based on the JSON specification and available capacity.
  2. **Scheduling:** Converting high-level operations (like `MatMul`) into a schedule of concrete, tile-based tasks (like `matmul_tile`). It also unwraps `ParallelOp` branches.

- `loader.py`: Contains the `JSONModelLoader`, which reads a JSON file and constructs an in-memory `Model` object by adding all tensors and operations.
- `model.py`: Defines the `Model` class, a simple container that holds dictionaries of `Tensors` and a list of `Ops`.
- `hardware_models.py`: Defines the data structures for all hardware components and statistics:
  - `MemoryDevice`: Models 3D RRAM/DRAM, including TSV latency logic.
  - `ComputeUnit`: Models the PIM CUs (MACs, SFEs).
  - `Tensor` / `TensorShape`: Data structures for tensors.
  - `Stats`: A data class for collecting all simulation results.
- `operations.py`: Defines the classes for all supported operations (e.g., `MatMul`, `Conv2D`, `AddOp`, `ParallelOp`). Each op class knows how to calculate its own FLOPs.
- `*.json`: (e.g., `decoding_stage_ffn.json`) Model definition files. These specify the tensors and operations for a given workload.

4. How to Run the Simulator

The simulator is run from the command line using `main.py` and requires a path to a JSON model file.

Command:

Bash
```
python main.py --json <path_to_your_model>.json
```

Example:

Bash
```
python main.py --json decoding_stage_ffn.json
```

5. How to Define a Model (JSON Format)

The JSON model specification has two main sections: `"tensors"` and `"ops"`.

a) Tensors

The `"tensors"` section is a list of all tensors used in the model (weights, activations, etc.). Each tensor is an object with the following properties:

- `"name"`: A unique string identifier for the tensor.
- `"shape"`: A list of integers defining the tensor's dimensions (e.g., `[1, 2048]`, `[2048, 8192]`).
- `"bits"`: The bit precision of each element (e.g., `16` for FP16, `4` for INT4).
- `"device"`: The preferred memory device: `"dram"` or `"rram"`.
- `"layer"`: The physical layer in the 3D stack where the tensor resides (e.g., `0`, `1`, `5`). This is used for TSV cost calculation.

Example `tensors` entry:

JSON
```
{
  "name": "W1_0",
  "shape": [2048, 8192],
  "bits": 4,
  "device": "rram",
  "layer": 1
}
```

b) Operations

The `"ops"` section is a list of operations that define the computational graph. The operations are executed in the order they appear.

- `"type"`: The type of operation. This must match a type handled by `JSONModelLoader` (e.g., `"MatMul"`, `"GeluOp"`, `"AddOp"`, `"UCIeOp"`, `"ParallelOps"`).
- `"A"`, `"B"`: The `"name"` of the input tensor(s).
- `"C"`: The `"name"` of the output tensor.

Example Ops:

JSON
```
"ops": [
    {
      "type": "MatMul",
      "A": "attn_out",
      "B": "W1_0",
      "C": "ff1"
    },
    {
      "type": "GeluOp",
      "A": "ff1",
      "C": "ff1"
    },
    {
      "type": "UCIeOp",
      "size_bits": 32768
    },
    {
      "type": "ParallelOps",
      "branches": [
        {"type": "MatMul", "A": "token_pe_cur", "B": "W_q_0_h1", "C":
"Q_t_h1"},
        {"type": "MatMul", "A": "token_pe_cur", "B": "W_k_0_h1", "C":
"K_t_h1"}
      ]
    }
  ]
```

6. Hardware Configuration

All hardware parameters are defined directly in `main.py` within the `main()` function. To modify the architecture, you can edit the instantiation of these objects:

- `dram`: An instance of `MemoryDevice` for 3D DRAM.
- `rram`: An instance of `MemoryDevice` for 3D RRAM.
- `dram_cu`: An instance of `ComputeUnit` for the DRAM CU.
- `rram_cu`: An instance of `ComputeUnit` for the RRAM CU.

Key parameters to modify in `MemoryDevice`:

- `capacity_bits`: Total Capacity. How much total data (in bits) this memory device (DRAM or RRAM) can store.
- `read_bw_bits_per_cycle`: Read Bandwidth. How many bits of data can be read from memory in a single clock cycle.
- `write_bw_bits_per_cycle`: Write Bandwidth. How many bits of data can be written to memory in a single clock cycle.
- `read_energy_per_bit`: Read Energy. How much energy (in nJ, nanojoules) is consumed to read 1 bit of data.
- `write_energy_per_bit`: Write Energy. How much energy (in nJ, nanojoules) is consumed to write 1 bit of data.
- `read_latency_cycles`: Read Latency. The number of clock cycles you must wait from issuing a "read" command until the data actually starts to transfer.

- `write_latency_cycles`: Write Latency. The number of clock cycles you must wait from issuing a "write" command until the data actually starts to be written.
- TSV parameters (Through-Silicon Via, for 3D stacking):
  - `tsv_bw_bits_per_cycle`: TSV Bandwidth. How many bits can be transferred vertically between layers of the 3D chip per cycle.
  - `tsv_base_latency_cycles`: TSV Base Latency. The fixed, minimum latency (in cycles) you pay any time you use TSV for a cross-layer transfer.
  - `tsv_fixed_latency_per_hop`: TSV Per-Hop Latency. The additional latency (in cycles) incurred for each layer the data has to cross.

Key parameters to modify in `ComputeUnit`:

- `macs_per_cycle`: MAC Compute Power. How many Multiply-Accumulate (MAC) operations this PIM compute unit can perform per clock cycle.
- `energy_per_mac_nj`: MAC Energy. How much energy (in nJ, nanojoules) is consumed for every single MAC operation.
- `sfe_ops_per_cycle`: SFE Compute Power. How many operations the SFE (Special Function Engine, for ops like Softmax, Gelu) can perform per cycle.
- `sfe_energy_per_op_nj`: SFE Energy. How much energy (in nJ, nanojoules) is consumed for every single SFE operation.

Key parameters in `simulator.py`:

- `ucie_bandwidth`: UCIe Bandwidth. The bandwidth of the chip-to-chip interconnect (e.g., for communication between the PIM chip and a CPU), measured in bits per cycle.
- `ucie_energy_per_bit`: UCIe Energy. How much energy (in pJ, picojoules) is consumed to transfer 1 bit of data over the UCIe interface.

Others:

- `freq_ghz`: Clock frequency.
- `tile_K, tile_M, tile_N:`

7. Understanding the Output

The simulator prints a detailed report to the console.

- **Total cycles**: The total simulation time in clock cycles. This is the primary latency metric.
- **Total MACs**: The total number of Multiply-Accumulate operations performed.
- **Estimated wall time @1GHz**: The total cycles converted to seconds, assuming a 1GHz clock.
- **Total energy (J)**: The total energy consumed by all compute and memory operations.

- **Energy Breakdown (nJ)**: Energy consumed by each *operation type* (e.g., `matmul_tile`, `geluop`).
- **MAC Breakdown**: Total MACs performed by each *operation type*.
- **Cycle Breakdown**: Total cycles spent on each *operation type*.
- **Detailed Cycle Breakdown**: Total cycles broken down by *hardware action* (e.g., DRAM Read, RRAM Compute).
- **Detailed Energy Breakdown (nJ)**: Total energy broken down by *hardware action*.

8. Extending the Simulator

a) Adding a New Operation

1. **`operations.py`**: Define a new `Op` subclass (e.g., `class my_new_op(UnaryOp)`). Implement its `flops` method.
2. **`loader.py`**: Add an `elif` block in `JSONModelLoader.build` to parse your new op from the JSON (e.g., `elif tpe == 'my_new_op'`).
3. **`compiler.py`**: In `_compile_op`, add logic to handle the new op. For most simple ops, just appending it to the schedule is sufficient (see the `else` block).
4. **`simulator.py`**: In `_calculate_item_cost`, add an `elif` block for your op's type (e.g., `elif ttype == 'my_new_op'`). Implement its specific cost logic (memory reads, compute, memory writes) and be sure to accumulate stats using `_acc_rw` and `_acc_comp`.

b) Adding a New Hardware Parameter

1. **`hardware_models.py`**: Add the new parameter to the relevant data class (e.g., add `my_param: int` to `MemoryDevice`).
2. **`main.py`**: Initialize this new parameter when you create the `MemoryDevice` objects.
3. **`simulator.py`**: Use this parameter in your cost calculations (e.g., in `_mem_read_cost`).

9. Cost Calculation Methodology

a) Latency (Cycles) Calculation

The total latency is the **sum of the latencies of each sequential operation** in the schedule. The latency for a single operation is calculated as follows:

- **For standard operations (like `matmul_tile`)**: The simulator assumes that memory access (Read), computation (Compute), and memory write-back (Write) can be overlapped (pipelined). Therefore, the latency for one operation is the **maximum** of these three stages.
    - `Latency = max(Input_Read_Cycles, Compute_Cycles, Output_Write_Cycles)`

- **For `ParallelOp` operations**: The simulator calculates the total sequential latency for *each* branch inside the parallel block. The final latency for the *entire* block is the latency of the **slowest branch**.
    - `Latency = max(Latency_Branch_1, Latency_Branch_2, ...)`

Here is how each component is calculated:

1. **Memory Cycles (`_mem_read_cost`, `_mem_write_cost`)**:
    - **Memory Access Latency: (`read_latency_cycles, write_latency_cycles`)**
      `bw_cycles = math.ceil(size_bits / dev.read_bw_bits_per_cycle)`
      `cycles = dev.read_latency_cycles + bw_cycles`
    - **Serialization (`ser`)**: The data (`size_bits`) is serialized based on the TSV's bandwidth (`tsv_bw_bits_per_cycle`).
      `ser = (size_bits + self.tsv_bw_bits_per_cycle - 1) // self.tsv_bw_bits_per_cycle`
    - **TSV Transfer Latency (`ser*(...)`)**: The total time is this number of serial transfers multiplied by the latency per transfer. This latency is
      `ser *(tsv_base_latency_cycles + hops * tsv_fixed_latency_per_hop)`
2. **Compute Cycles (`_compute_cost_engine`)**:
    - This is the total number of operations divided by the hardware's throughput.
      `MAC_Cycles = Total_MACs / macs_per_cycle`
      `SFE_Cycles = Total_SFE_Ops / sfe_ops_per_cycle` (for non-MAC ops)
3. **UCIe Cycles (`ucieop`)**:
    - This is the total bits to be transferred divided by the `ucie_bandwidth`.

2. Energy (nJ) Calculation

Energy is simpler: it is the **sum of the energy of all components**. There is no "max" or overlap; all consumed energy is added up.

- **For standard operations (like `matmul_tile`)**: The simulator **sums** the energy for all parts of the operation.
    - `Energy = Read_Energy_A + Read_Energy_B + Compute_Energy + Write_Energy_C`
- **For `ParallelOp` operations**: The simulator calculates the total energy for *each* branch and then **sums the energy of all branches** together.
    - `Energy = Energy_Branch_1 + Energy_Branch_2 + ...`

Here is how each component is calculated:

1. **Memory Energy (`_mem_read_cost` / `_mem_write_cost`)**:
    - This is a simple calculation: `Total_Bits * energy_per_bit`.
    - `Read_Energy = Total_Bits_Read * read_energy_per_bit`
    - `Write_Energy = Total_Bits_Written * write_energy_per_bit`
    - (Note: The current model in `simulator.py` does not add extra energy for TSV transfers, only latency).
2. **Compute Energy (`_compute_cost_engine`)**:

    - This is the total number of operations multiplied by the energy cost of each.
    - `MAC_Energy = Total_MACs * energy_per_mac_nj`
    - `SFE_Energy = Total_SFE_Ops * sfe_energy_per_op_nj`
3. **UCIe Energy (`ucieop`)**:
    - `UCIe_Energy = Total_Bits * ucie_energy_per_bit`
    - (The code also converts this from pJ to nJ).

Summary: Aggregation

The `Simulator.run()` method iterates through the schedule.

- `self.stats.cycles` is **incremented by the latency** of each operation (which is the `max` of its sub-parts or the `max` of its parallel branches).
- `self.stats.energy_nj` is **incremented by the energy** of each operation (which is the `sum` of its sub-parts or the `sum` of its parallel branches).