PLENA Testbench Technical Introduction

Focus: `developer_compiler.py` and `plena_program.py`

1. Scope

This document explains the architecture and execution model of two core files in the PLENA simulator testbench:

- `PLENA_Simulator/behavioral_simulator/testbench/plena_program.py`
- `PLENA_Simulator/behavioral_simulator/testbench/developer_compiler.py`

The emphasis is on:

- Sub-matrix decomposition and projection workflows.
- Memory management across HBM, VRAM, MRAM, and FPRAM.

A third file is essential for both topics and is referenced throughout:

- `PLENA_Simulator/behavioral_simulator/testbench/sub_matrix_manager.py`

---

2. High-Level Architecture

At a high level, the stack is split into two layers:

- **Front-end API layer (`PLENAProgram`)**
  - User-facing Python DSL and object model.
  - Eager execution: API calls immediately emit ISA through the compiler.
  - Main entry point: `PLENAProgram` in `plena_program.py:510`.
- **Back-end compiler layer (`DeveloperCompiler`)**
  - Owns symbol table, register allocator, and sub-matrix manager.
  - Converts high-level operations into ISA fragments.
  - Main entry point: `DeveloperCompiler` in `developer_compiler.py:121`.

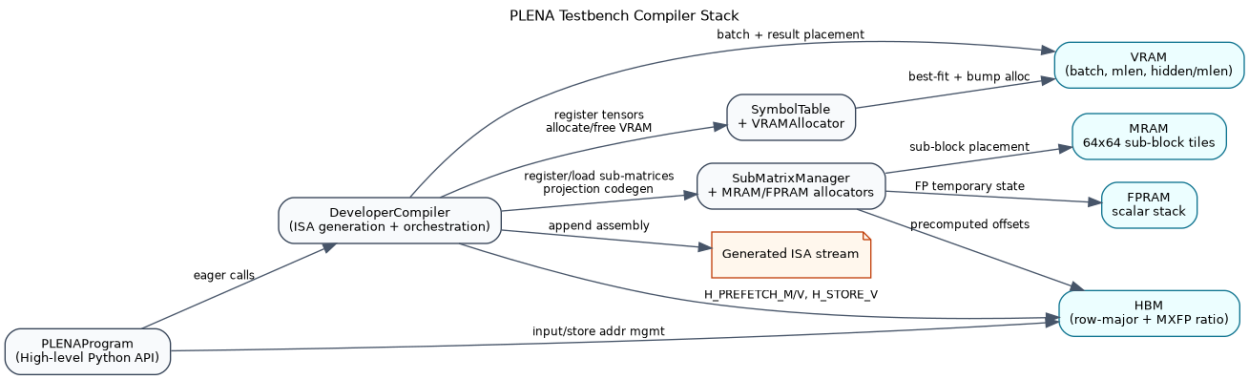

Figure 1: Architecture Overview

Key glue logic:

- `PLENAProgram` delegates all codegen to `self._compiler` (`plena_program.py:525`).
- `DeveloperCompiler` integrates `SubMatrixManager` (`developer_compiler.py:285`).

---

3. `plena_program.py`: API and Program Model

3.1 Responsibilities

`PLENAProgram` is a thin but important orchestration API:

- Input declaration and automatic HBM address assignment (`plena_program.py:579-603`).
- HBM to VRAM load (`plena_program.py:609-644`).
- VRAM to HBM store (`plena_program.py:665-705`).
- VRAM allocation/free (`plena_program.py:711-753`).
- Sub-matrix registration (`plena_program.py:1191-1224`).
- VRAM-sub-matrix registration (`plena_program.py:1227-1253`).

3.2 Naming and Scope

A dual naming model is used:

- Display name: user-facing.
- Internal name: compiler/symbol-table identity.

Scoped internal names are generated via `_scoped_name` (`plena_program.py:1567`) and function-call stack prefixes from `@prog.function` (`plena_program.py:1410-1481`).

This design prevents collisions in repeated/nested function calls.

3.3 Operator Dispatch Status

- Generic `@` for arbitrary tensor matmul is explicitly disabled (`plena_program.py:1518-1536`).
- VRAM-submatrix `@` dispatch remains available through `_dispatch_vram_sub_matmul` (`plena_program.py:1539-1560`) and deferred execution objects.

Interpretation:

- The file is transitioning toward explicit `*_to(...)` APIs for deterministic placement and control, especially in blockwise kernels.

---

4. `developer_compiler.py`: ISA-Oriented Core

4.1 Responsibilities

`DeveloperCompiler` owns low-level concerns:

- Register allocation (`RegisterAllocator`, `developer_compiler.py:22`).
- Symbol table and address ownership (`developer_compiler.py:280-286`).
- Emission of ISA code snippets into `generated_code`.
- Sub-matrix operation scheduling and address realization.

4.2 Sub-matrix Entry Points

Core methods:

- `register_sub_matrix` (`developer_compiler.py:1641`)
- `load_sub_matrix_row` (`developer_compiler.py:1701`)
- `load_sub_matrix_col` (`developer_compiler.py:1758`)
- `sub_projection` (`developer_compiler.py:1817`)
- `sub_projection_T` (`developer_compiler.py:1897`)
- `register_vram_sub_matrix` (`developer_compiler.py:1979`)
- `vram_sub_projection_to` (`developer_compiler.py:2114`)
- `vram_sub_projection_T_to` (`developer_compiler.py:2175`)

Design pattern:

1. Validate symbols and tensor kinds.
2. Ensure destination allocation (or compute explicit destination block address).
3. Allocate temporary registers.
4. Delegate loop-level ISA generation to `SubMatrixManager`.
5. Free registers and append generated text to compiler output.

---

5. Sub-Matrix System (Core Focus)

5.1 Block Model

In `sub_matrix_manager.py`, each large matrix is represented as a grid of `mlen x mlen` tiles:

- `SubMatrixInfo` (`sub_matrix_manager.py:247`)
- `MatrixBlockLayout` (`sub_matrix_manager.py:267`)
- `VRAMMatrixBlockLayout` (`sub_matrix_manager.py:351`)

Default parameters:

- `mlen = 64`, `blen = 4`.

5.2 Address Precomputation

HBM tile offset is precomputed at layout build time:

- `hbm_offset = r * block_size * cols + c * block_size`
- Implemented in `MatrixBlockLayout.__post_init__` (`sub_matrix_manager.py:290-311`).

VRAM tile base follows column-block-major layout:

- `col_block_base = vram_base + c * batch * block_size`
- `row_offset = r * block_size * block_size`
- `vram_addr = col_block_base + row_offset`
- Implemented in `VRAMMatrixBlockLayout.__post_init__` (`sub_matrix_manager.py:377-405`).

5.3 Load Paths

`load_sub_matrix_asm` / row / col loaders (`sub_matrix_manager.py:783-915`) generate `H_PREFETCH_M` sequences:

- Set scale/stride registers based on full matrix dimensions.
- Use precomputed HBM offset per tile.
- Record loaded tile and MRAM address (`mram_addr`) for downstream projection.

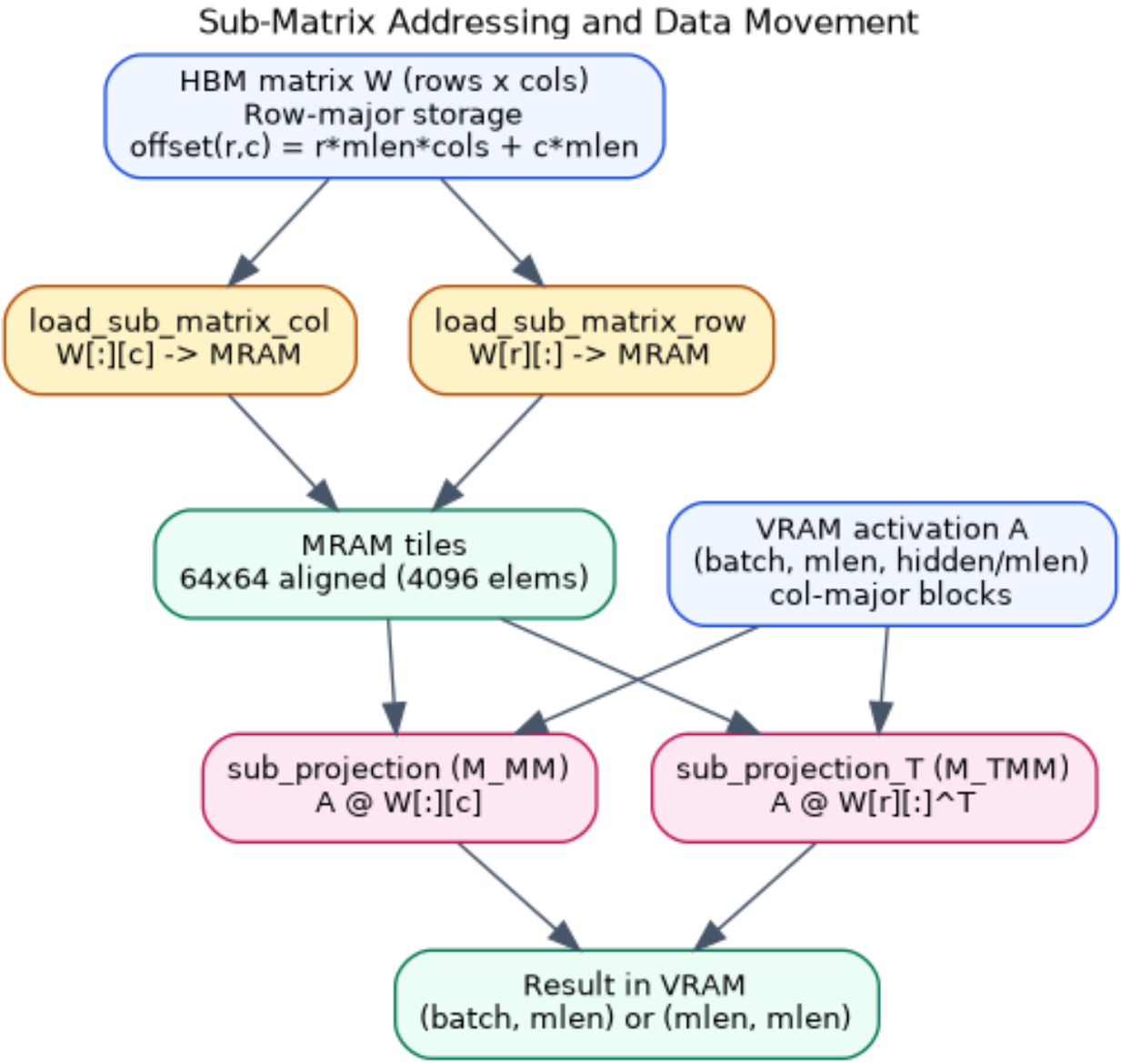

Figure 2: Sub-Matrix Addressing

5.4 Projection Paths

Two primary projection kernels:

- `sub_projection_asm` (`sub_matrix_manager.py:995`)
  - Computes `A @ W[:][col_idx]`.
  - Uses `M_MM` accumulation loops over hidden blocks.
- `sub_projection_T_asm` (`sub_matrix_manager.py:1116`)
  - Computes `A @ W[row_idx][:]^T`.
  - Uses `M_TMM` and transposed tile offseting.

VRAM-submatrix variants:

- `vram_sub_projection_asm` (`sub_matrix_manager.py:1248`)
- `vram_sub_projection_T_asm` (`sub_matrix_manager.py:1391`)

These enable block-level kernels for Flash Attention style flows (`Q_sub.row(i) @ K_sub.row(j).T`).

---

6. Memory Management (Core Focus)

6.1 Unified Virtual Memory Strategy

The core allocator is `VirtualMemoryManager` (`sub_matrix_manager.py:43`):

- Tracks `used_stack` and `free_stack`.
- `allocate(name, size)`:
  - best-fit reuse from `free_stack` first,
  - fallback to aligned bump pointer (`next_bump`).
- `free(name)` moves blocks back to reusable pool.

Implementation references:

- Allocation logic: `sub_matrix_manager.py:91-162`.
- Free logic: `sub_matrix_manager.py:164-186`.

6.2 VRAM Allocation

`SymbolTable` owns a `VRAMAllocator` (`symbol_table.py:131-138`) that wraps `VirtualMemoryManager` (`symbol_table.py:31-128`).

Properties:

- Alignment defaults to `MLEN` (64).
- Requires allocation by `name`, enabling explicit free and reuse.
- Used by `add_batch` (`symbol_table.py:139-185`) and by compiler-level VRAM allocations.

Runtime API linkage:

- `PLENAProgram.alloc(...)` -> compiler allocation (`plena_program.py:711-732`).
- `PLENAProgram.free_tensor(...)` -> `vram_allocator.free(name, strict=False)` (`plena_program.py:734-753`).

6.3 MRAM Allocation

`MRAMAllocator` (`sub_matrix_manager.py:430-505`) is also backed by `VirtualMemoryManager` but with tile-size alignment:

- Alignment = `MLEN * MLEN = 4096` elements (`sub_matrix_manager.py:450-454`).
- Default capacity = `MLEN * MLEN * 4` (four 64x64 tiles) (`sub_matrix_manager.py:444-448`).

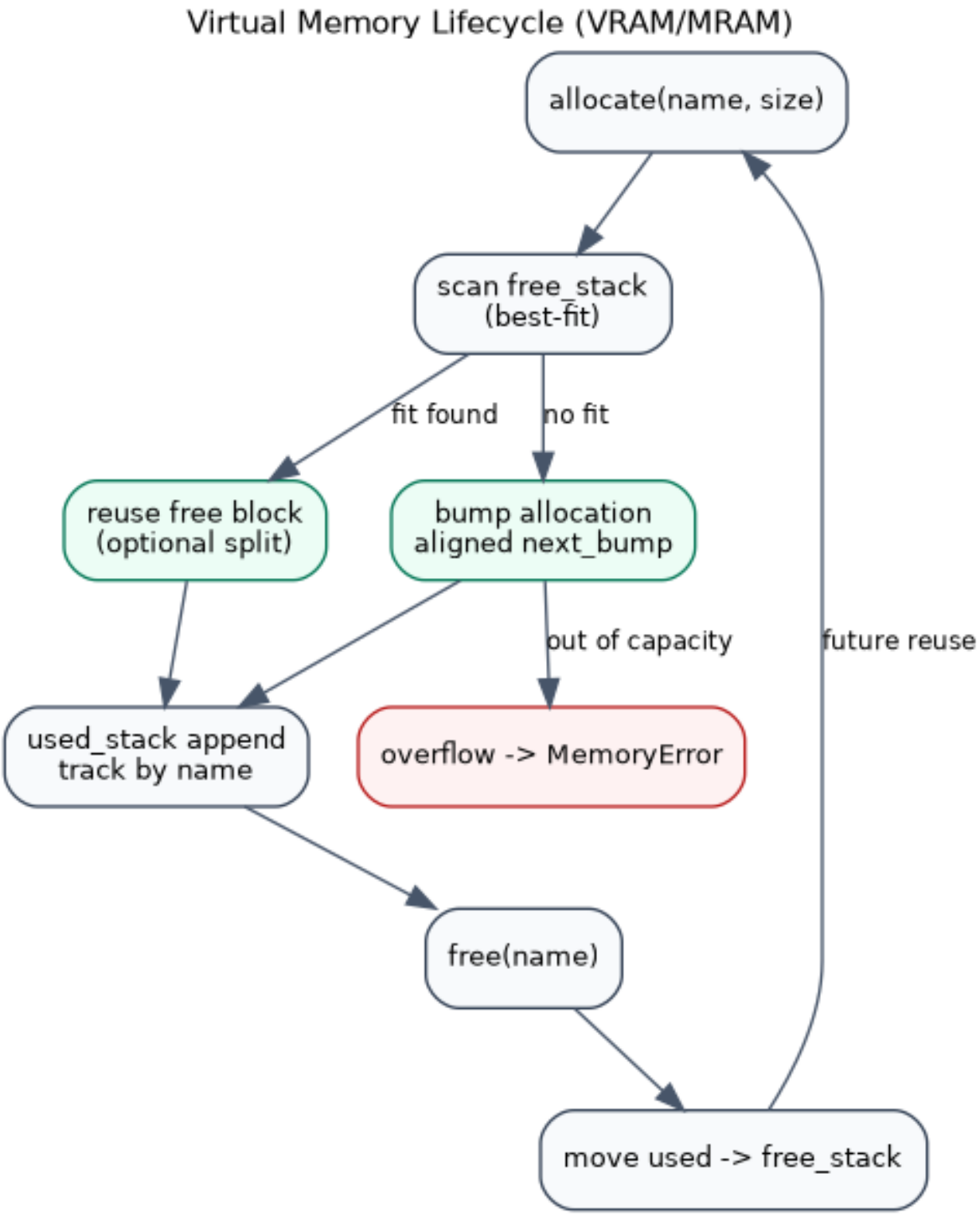

Figure 3: Memory Lifecycle

Usage in compiler:

- Row/col tile loads auto-allocate MRAM ranges when caller does not provide explicit start address (`developer_compiler.py:1722-1727`, `1781-1786`).

6.4 FPRAM Allocation

`FPRAMAllocator` (`sub_matrix_manager.py:507-550`) is stack-oriented:

- Bump allocation only.
- Snapshot/restore semantics via `save_state` / `restore_state`.
- Suited for scoped scalar temporaries in softmax and reduction pipelines.

API surface in `PLENAProgram`:

- `fp_var`, `save_fpram_state`, `restore_fpram_state` (`plena_program.py:759-797`).

6.5 HBM Address Lifecycle

In `PLENAProgram`, HBM addresses are optionally auto-assigned and 64-aligned:

- Input declaration: `input(...)` (`plena_program.py:579-603`).
- Store outputs: `store(...)` (`plena_program.py:665-705`).

The data-size model includes `real_data_ratio` (default 1.125), reflecting packed MXFP-like storage assumptions.

---

7. End-to-End Execution Example (Conceptual)

For a blockwise projection `Y = A @ W[:, c]`:

1. Declare and load activation into VRAM:
   - `input(...)` then `load_batch(...)`.
2. Register matrix `W` for sub-matrix indexing:
   - `register_sub_matrix(...)`.
3. Load tile column `W[:][c]` from HBM to MRAM:
   - `load_sub_matrix_col(...)`.
4. Emit projection loop:
   - `sub_projection(...)` delegates to `sub_projection_asm(...)`.
5. Result tile/column is materialized in VRAM; optional `store(...)` writes back to HBM.

For Flash Attention-style `S[i][j] = Q_sub.row(i) @ K_sub.row(j).T`, use:

- `register_vram_sub_matrix(Q)`
- `load_sub_matrix_row(K, j)`
- `vram_sub_projection_T_to(...)`

---

8. Key Engineering Observations

1. **The architecture is tile-first, not GEMM-first.**
   - Most semantics reduce to explicit 64x64 block movement + block compute.
2. **Addresses are computed early and reused often.**
   - HBM/VRAM tile mapping is deterministic and cached in layout objects.
3. **Memory reuse is intentional and explicit.**
   - VRAM/MRAM use a reusable free-list + bump fallback strategy.

4. **FPRAM is optimized for scoped scalar workflows.**
    - Snapshot/restore pattern cleanly supports softmax-like temporaries.
5. **The API is moving toward explicit destination placement.**
    - `*_to` methods expose destination tile coordinates and reduce ambiguity.

---

9. Referenced Source Files

- `PLENA_Simulator/behavioral_simulator/testbench/plena_program.py`
- `PLENA_Simulator/behavioral_simulator/testbench/developer_compiler.py`
- `PLENA_Simulator/behavioral_simulator/testbench/sub_matrix_manager.py`
- `PLENA_Simulator/behavioral_simulator/testbench/symbol_table.py`