SAMSUNG

SONIC MACHINE LANG

Daniyal Khan
Malte Deling
Hyunchul Park
Dong Hyuk Woo

Challenges with Current Approaches:

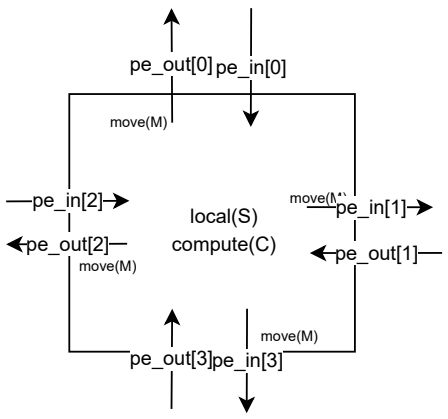
Existing ML graph compilers primarily represent the operational graph but lack detailed awareness of machine topology and data distribution.

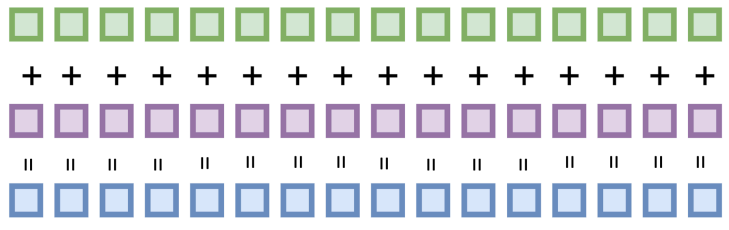
This limitation leads to inefficient distribution of compute and data workloads, causing unnecessary communication overhead and poor system performance.

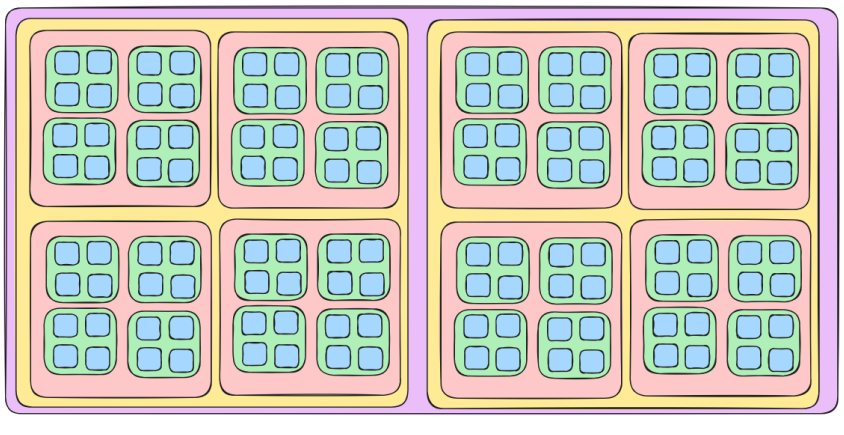
How SonicMachine Models Hardware Efficiently

- Leverages MLIR's shape system to efficiently represent hierarchical hardware components.
- Models data movement explicitly using operations such as slice, concat and broadcast
- These operations define interconnections between compute and storage units.
- Supports custom topology tiers to:
  - Specify communication pathways.
  - Define performance attributes tied to those pathways.
- Ensures accurate representation of system constraints through detailed modeling.
- Provides builder functions and templates to:
  - Enhance expressiveness.
  - Enable compact descriptions of complex hardware architectures.

```
sml.func @create_pe<P>(%pe_in: !sml.wire<Px4>) -> !sml.wire<Px4> {
  // %local_in: defined later

  // Instantiate a storage unit.
  %local_out = sml.storage<Px1> local %local_in : !sml.wire<Px1> -> !sml.wire<Px1>

  // Instantiate a compute unit.
  %compute_out = sml.compute<Px1> "compute" %local_out : !sml.wire<Px1> -> !sml.wire<Px1> {
    %10 = sml.pipeline "fu" %local_out : !sml.wire<Px1> -> !sml.wire<Px1>
    sml.yield %10 : !sml.wire<Px1>
  }

  // Instantiate move units.
  %move_in = sml.broadcast %local_out : !sml.wire<Px1> -> !sml.wire<Px4>
  %move_out = sml.move<Px4> "mesh_move" %move_in : !sml.wire<Px4> -> !sml.wire<Px4>

  // Muxing two inputs into the local input
  %local_in_concat = sml.concat %pe_in, %compute_out
      : !sml.wire<Px4>, !sml.wire<Px1> -> !sml.wire<Px5>

  %local_in = sml.mux %local_in_concat : !sml.wire<Px5> -> !sml.wire<Px1>

  sml.return %move_out : !sml.wire<Px4>
}
```







Distribution Example

Consider the blue square as the Processing Element (PE) described previously. These PEs can logically be grouped into 2x2 modules. Such modules can represent entities like a node containing 4 GPUs or a die comprising 4 PEs.

Attributes can be assigned to each PE, such as latency for computational operations, memory operations, and bandwidth. Similarly, attributes regarding latency and bandwidth can be defined for communication among PEs within the same module.

We can further group these smaller modules into larger units, adding comparable attributes at this higher level as well.

Using this structured intermediate representation (IR), Sonic Compute can effectively analyze and decide how to distribute data and computation.

Example:

In a vector addition task involving 8 GPUs, the operation might still execute entirely on a single GPU. This decision could arise because the inter-GPU bandwidth might be insufficient for efficient parallel computation across multiple GPUs.

The above machine is 2x1x2x2x2x2x2x2;

with different colours representing different hierarchy