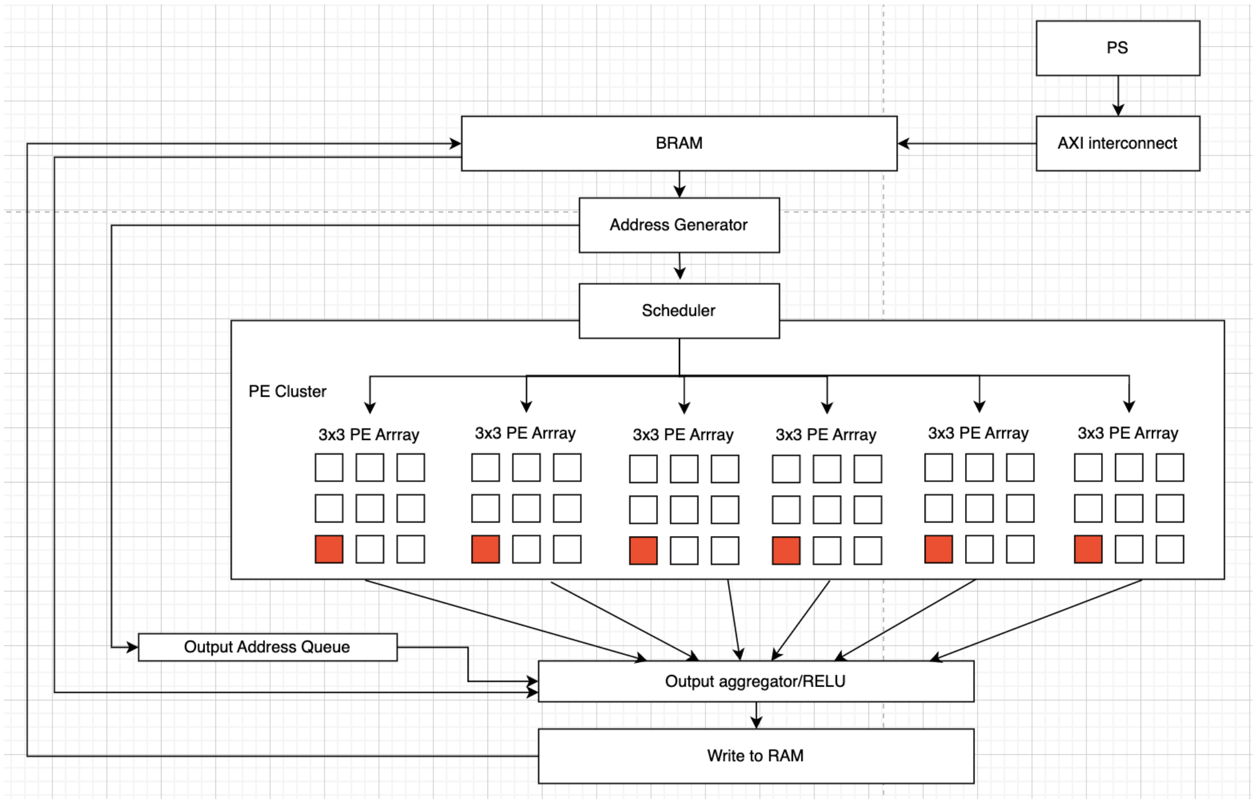
Results:
Total utilization: 82.06%
Total MACS: 14058616
Total Active MACS: 11535975
Total Active PE Array MACS: 13271040 * .81 = 10749543
Redundant MAC usage (calculating wasted values): 81% of the PE array

Analysis: As the 6th PE array only is ⅓ efficient we waste quite a few macs. For improvements we would need to be able to use partial PE arrays.

Architecture:
This implementation is of the 2nd convolution (the 3rd layer)



Component Summary:

- **Address Generator:** Goes through the data flow loop to generate the correct addresses for accessing input data, filters and also outputs.
- **Scheduler:** Takes data from address generator and distributes it correctly to each PE array with the correct timing.
- **PE Cluster:** Contains a cluster of 6 PE arrays, able to process one output row at a time
- **PE Array:** Contains 3x3 array of PEs, able to process a 5x3 tile of input to produce a 3x3 tile of output
- **PE:** Handles one portion of the row and filter, using a sliding window approach to multiply a 5 tile row for 3 outputs (per one row of filter).

- **Output address queue:** Accumulates addresses for output aggregator.
- **Output aggregator/RELU:** Brings together the outputs from the separate PE arrays and adds it to previous partial results. It also performs RELU on last output layer.
- **Write to RAM:** Pipelines the output aggregator/RELU module, storing the final result in memory

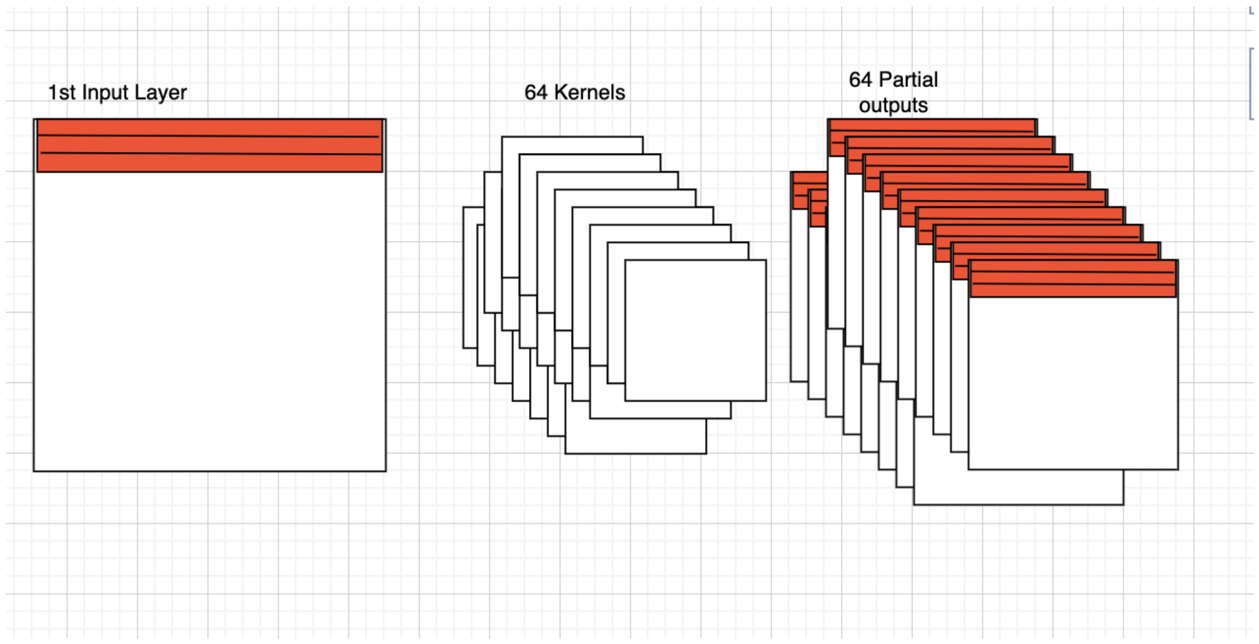
This architecture is based on Eyeriss V1; an energy efficient CNN focused on data reuse. We utilize the same row stationary format.

We have a total number of 286 MACS. Within each of the 6 PE arrays, a single PE contains 5 MACS, resulting in 270 MACS across the PE arrays; the output aggregator must also read memory and aggregate stored results with just computed results (part of the accumulate cycle), resulting in an extra 16 macs. We utilize tiling (5x5) to efficiently support reuse at the PE Array level.

Data flow (for-loop manner)

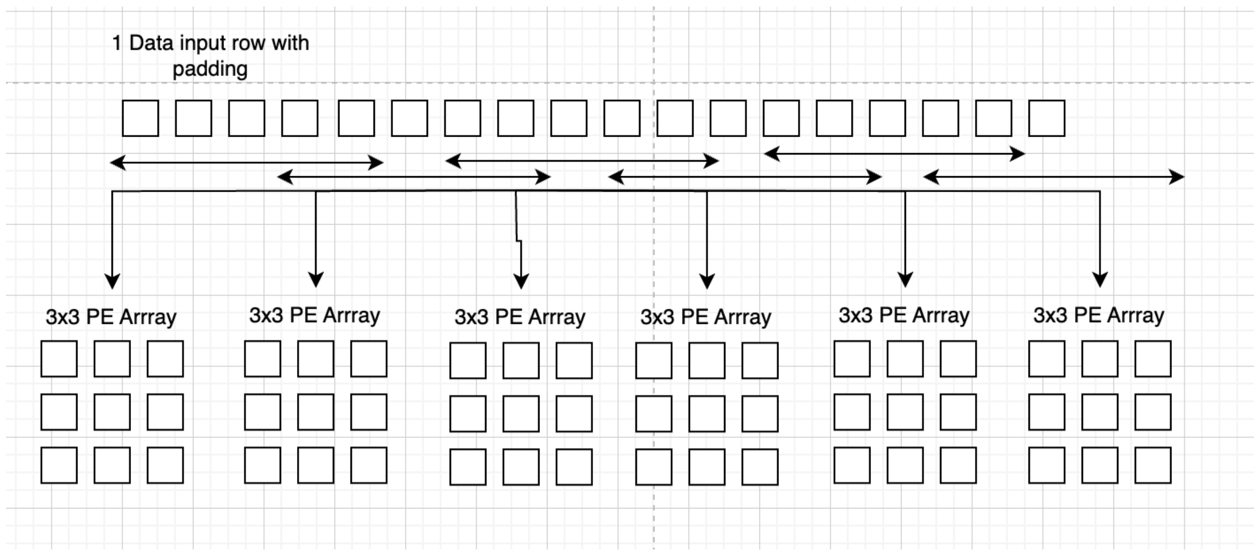
```
FOR each input_channel from 0 to C-1:
  FOR each tile_height from 0 to input_height, skip 3 rows at a time:
     // Process 3 rows at a time
    FOR each output_channel from 0 to F-1:
        FOR each 3 output pixels (width) from 0 to tile_width:
          // Split the row into 6 portions, processing 3 output pixels per PE array
          psum = 0  # Initialize partial sum
          FOR each filter_row from 0 to K-1:
            sliding_window = []  # Initialize sliding window
            FOR each filter_width from 0 to K-1:
                # Extract patch into sliding window
                sliding_window.append(
                  IFM[fmout_height + filter_row][fmout_widthx][input_channel]
                )
            # Perform matrix multiplication with filter
            psum += matrix_multiply(
              sliding_window, Filter[filter_row][filter_width][output_channel]
            )
          OFM[fmout_height][fmout_width][output_channel] = psum
```

Visual representation:

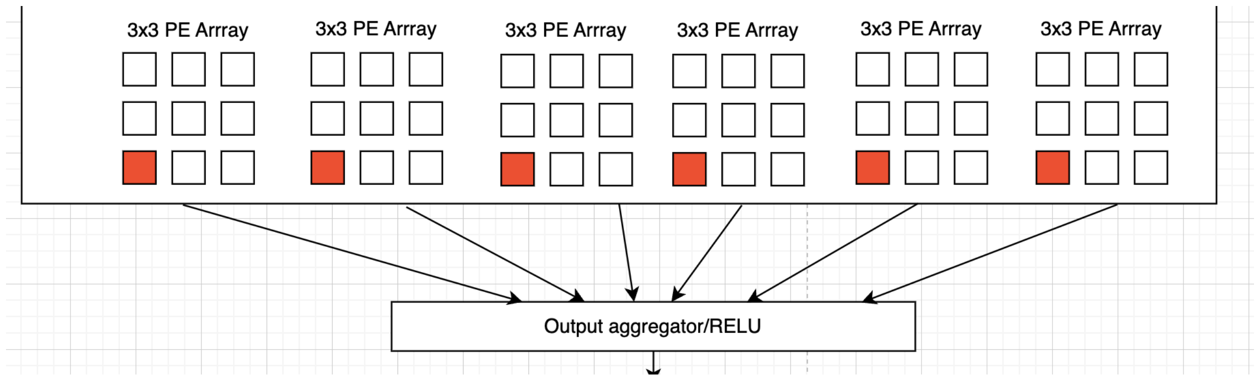


1) Within our dataflow, we are processing 3 rows at a time. For a given held constant input, we apply each of the kernels to get a partial result for the 3 rows
2) This continues on for all rows to get a partial result for all rows.
3) Finally we move on to the next layer, fetching the previous result from BRAM and adding onto it.

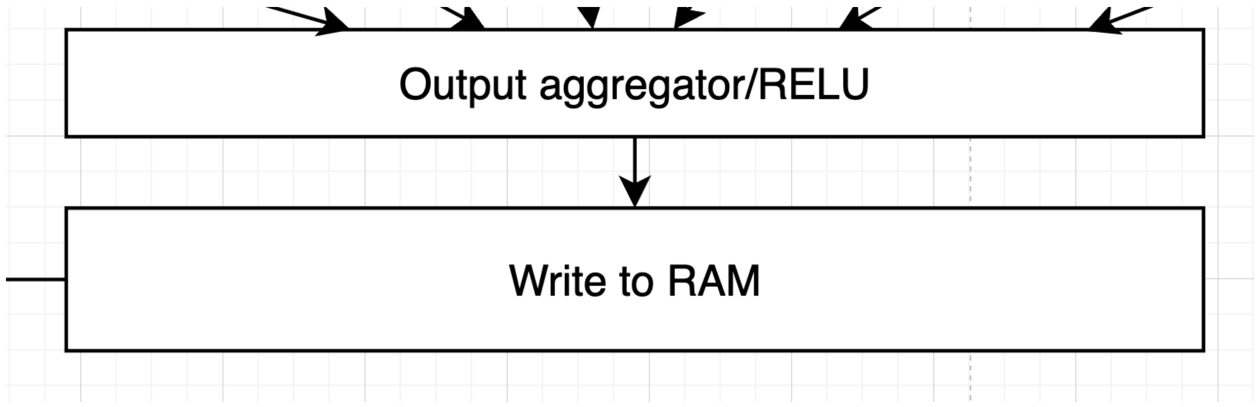
Data flow at the Cluster level:



** Split each row into 5 row tiles, splitting across our 6 PE clusters*

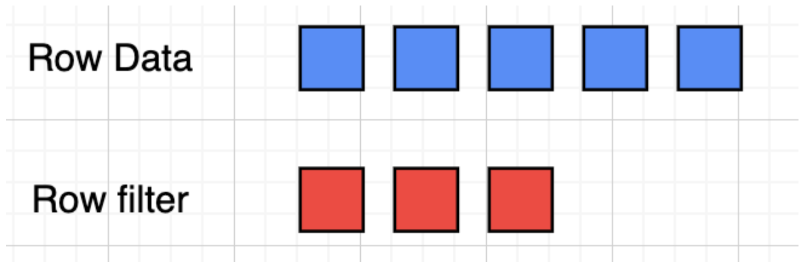


** Once a PE array finished processing a row, it gets aggregated in output aggregator. This aggregator fetches any partial results from BRAM for this output layer and adds it.*

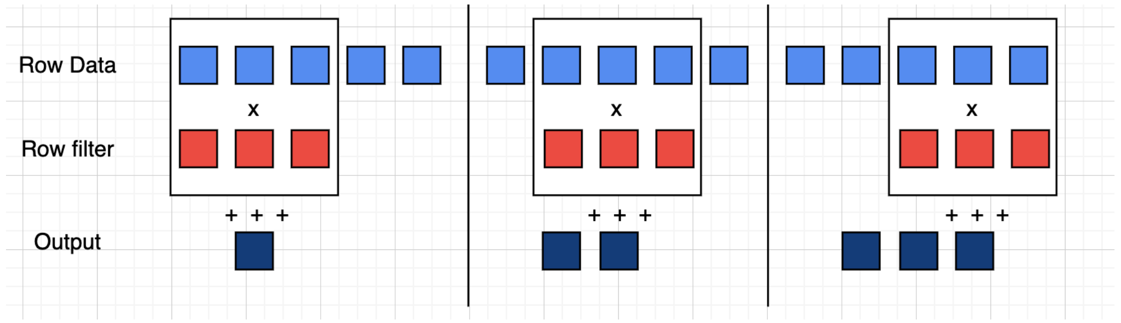


**Finally we store these results in memory*

Data flow at the PE level:



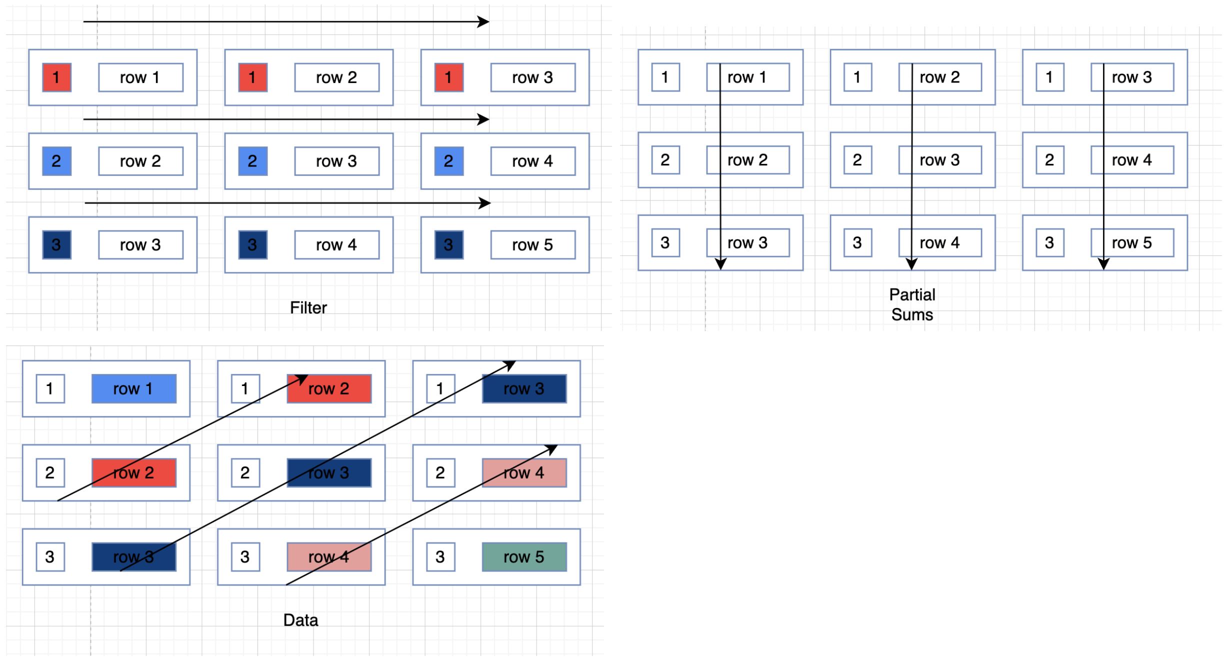
* *Receives 5 integers from a row and a row of a tile*



* *From the 5 input integers, we produce 3 output results using a sliding window approach*

A whole cycle of the row data results in 4 clock cycles, 3 for multiplying and 1 for aggregation between PEs.

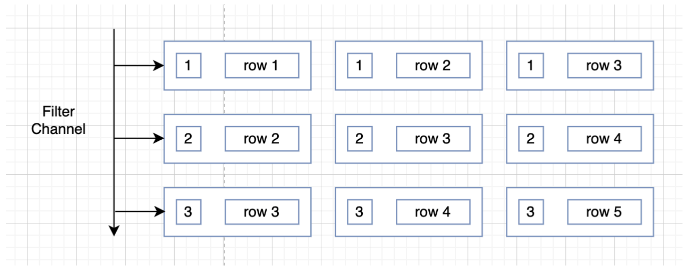
Data flow at the PE Array layer:



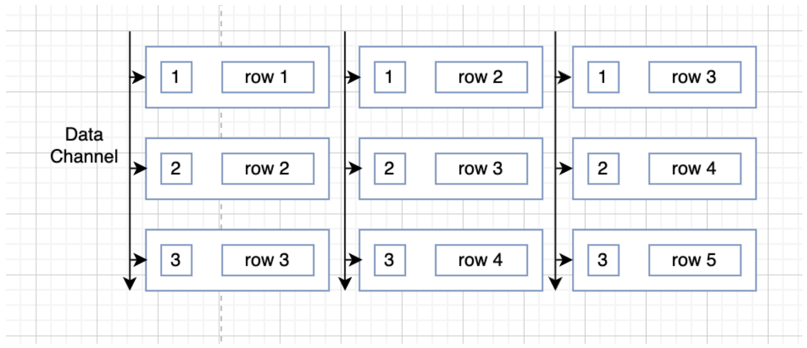
Filters: Propagated column by column across the PE, one clock cycles at a time.
Partial results: Propagated row by row across the PE, one clock cycle at a time after finished processing multiplication.

Data: Is broadcasted across the PE, each sharing data diagonally.

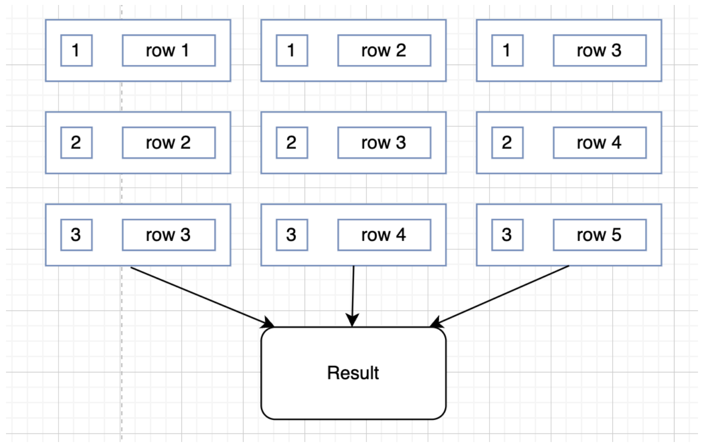
Filter Channels: Each PE array have filters broadcasted across the left column, where the PE takes the values when needed:



Data Channels: Data is broadcasted across all PEs within a cluster, the PE takes the values on need basis.

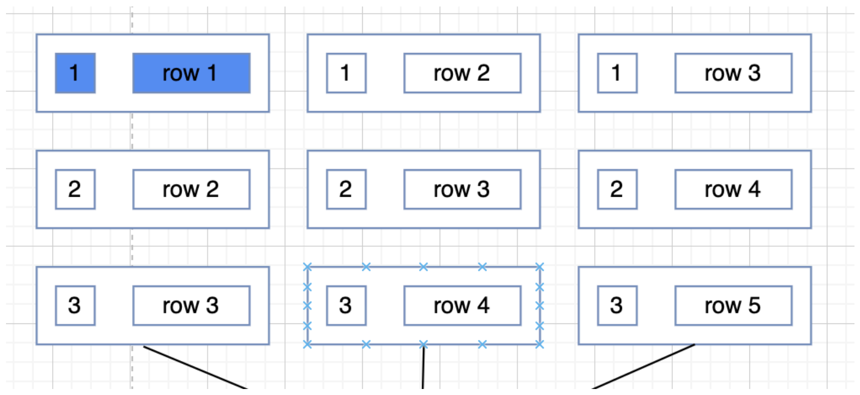


Output Channel: Results are multiplexed from the bottom PE of each column, once done processing.

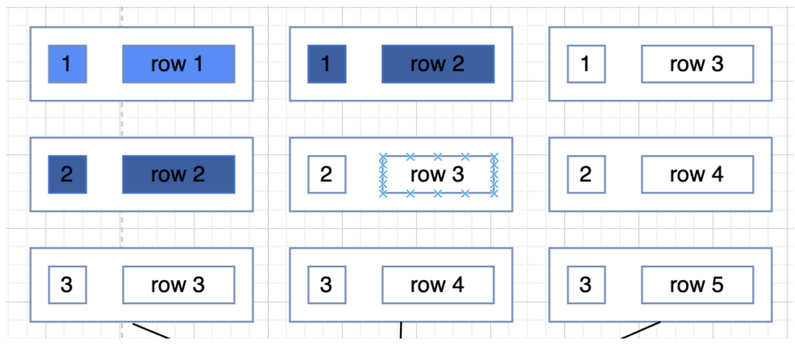


Visual Example of Data Flow

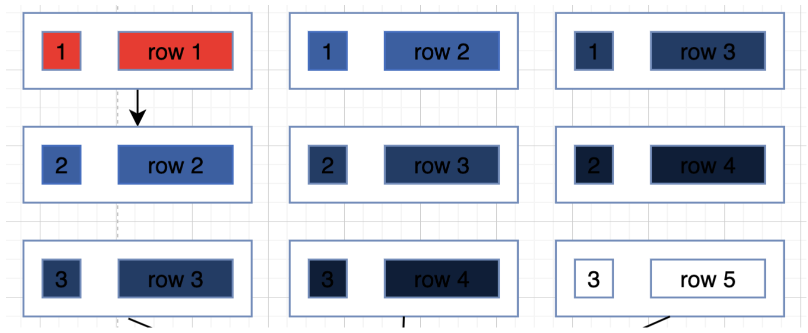
1) First row of the filter is propagated along the filter channel with the broadcast of the first row of the input data.



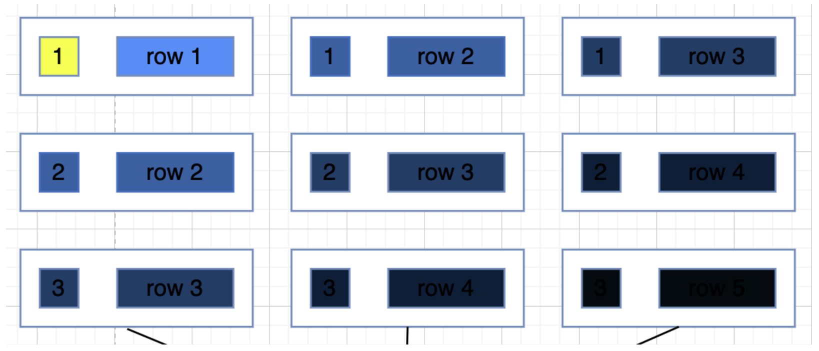
2) Second row of the filter is propagated along the filter channel with the broadcast of the second row of input data. Filter from column one, row one is propagated horizontally.



3) On the fourth cycle, the first PE is finished multiplying and accumulates the previous result (None) and propagates it to the below PE.



4) On the fifth cycle, all columns have started on the first filter. The first PE now is pipelined to start on the second output layer, holding its input data constant. Although this PE now receives a new filter for the new output layer.



Storage Structure for efficient scheduling:

Input map (stored sequentially in memory in row-major format):

Input1 ***Fmap:*** flattened [row0, row1, row2…]

Input2 ***Fmap:*** flattened [row0, row1, row2…]...

-128-bit Addressing

Weight Structure:

1st ***Kernel:*** flattened [row0, row1, row2…]

2nd ***Kernel:*** flattened [row0, row1, row2…]

3rd ***Kernel:*** flattened [row0, row1, row2…]

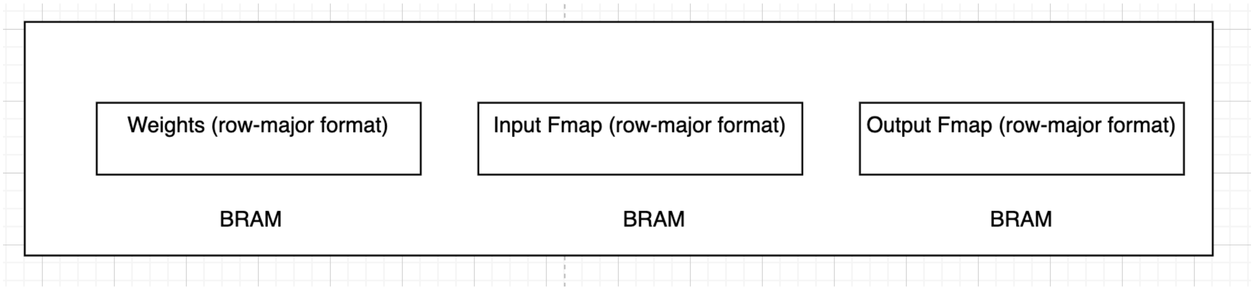
-32-bit Addressing

Output structure (re-used):

Same as input map

** Cannot reuse input map area as data fetching may still happen during writing output*

-128-bit Addressing


Weights (row-major format)
Input Fmap (row-major format)
Output Fmap (row-major format)
BRAM
BRAM
BRAM