ECE 284 Report — Little MAC

Authors: Daniel Tran, Ryan Lee, Arian Torshizi, Mitali Agrawal, Anjana Manoj, John Hsu

Abstract

We propose several variations on tensor processors based on the 2D-systolic architecture. These variations aim to improve the power consumption, flexibility, and performance of the machine.

Part 1: Base Implementation

We implemented the base 2D systolic array, pipelining L0 reads of activation, MAC array calculations, writing to OFIFO, and writing to PSUM SRAM to allow for nij values greater than 64. The PSUMs are then passed to the SFP to do the final accumulation over kij values and RELU activation. The MAC array loads weights before execution, accepting at each cycle an input activation, multiplying it by the stored weight, adding it to the PSUM passed in from above, and passing the result down. These PSUMs are all stored and accumulated once all 9 kijs have been calculated.

FPGA Mapping	***Cyclone IV GX***
OPs	128
Frequency	127.36 MHz
Dynamic Power	33.36 mW
TOPs/s	0.0163
TOPs/W	0.489
Logic Elements	16,859

Part 2: 2-bit SIMD/4-bit Reconfigurable

In this part, we built on our previous implementation, adding another bit register to help load two weights at a time per mac_tile and adding a configurable activation_mode bit input that lets the array (and therefore each tile) know what mode it is in (i.e., whether or not to shift one of the PSUMs before adding in the mac_tile). This allows us to either simulate 4-bit by 4-bit multiplication (with shifting) or 2-bit by 4-bit multiplication twice (by simply adding without shifting). We also implement column tiling by simply running the flow twice (once for each tile) and storing the two independent output channel tiles to different sections of the PSUM SRAM.

Part 3: OS/WS Reconfigurable

In order to provide potential performance improvements when weight reuse is low, we were tasked with modifying the base systolic array to be able to run in an output-stationary configuration. To do so, an IFIFO was used to stream weights down to the PEs from their northern inputs. To reduce the complexity of the design, the weight-stationary configuration no longer streamed in weights from the L0 FIFO, but rather reused the IFIFO's weight streaming.

Activations and weights were loaded into the FIFOs as described in lectures. To configure PEs to use output-stationary execution, a separate set of instructions designed specifically for OS execution were implemented.

- OS execute: The "ready-to-load-weight" state from WS is repurposed to initialize the accumulator in OS. If this bit is set high and the PE sees the "OS execute" instruction, it resets its accumulator to 0 and stores the northern input (weight) and western input (activation) by the next cycle. From then on, it performs local accumulation on the stored activations and weights, and retrieves new values for them each cycle from the north and west. It exposes its stored weight southward, and its activation eastward
- Flush: The PE exposes its accumulated value southward. It fetches the northern input into its accumulator on the next cycle (expecting it to be the above PE's accumulated value).

Notably, when a flush is issued, it fills every instruction entry in the instruction shift register, thus being issued to every row in the first cycle it is called. Flushes do not fill every column at once; they still propagate one-by-one column-wise.

To stream the correct pattern of activations for convolution (as described in lecture), the simplest solution was to store that exact pattern of activations in SRAM. This resulted in duplicate activations appearing in SRAM, increasing memory consumption. Performance-wise, WS execution completed the testbench in 1144500 cycles, whereas OS took 409500 cycles. This >50% reduction in cycles is likely due to the fact that the output-stationary testbench only computed the results for half of the output channels.

Alphas

Clock Gating

We implemented clock gating by gating off the clock signals sent to each SRAM, turning off toggling when we aren't using that memory block. This resulted in a drop from 33 mW to 28 mW in our Quartus simulation, which is a non-negligible drop-off. In the future, we would probably expand this to clock gating on the mac array, L0, SFP, and OFIFO blocks as well.

Multi-Core

In the case that power and area are not critical factors and throughput is to be maximized (and we have 16 output channels to use), we introduce multicore to double

throughput by enabling the processing of twice as many outputs at once. Unlike tiling, this does not require us to run the “execution” phase twice. Since we implement this by instantiating weight and partial sum SRAMs twice alongside two instantiations of corelet (the activations are the same for each core, so no duplication was needed), we can simply pass the same control signals to both SRAM-corelet sets, as they will work in parallel. Other than these alterations, the software is identical to Part 2, as is the input file setup. We observed that the execution time for calculating all 16 output channels using multi-core was comparable (no significant increase) compared to calculating 8 output channels in the Part 2 implementation, indicating a 2x improvement in throughput (since Part 2 must calculate 8 output channels twice).

Activation Configurability

To support activation functions other than ReLU, we incorporated hardware support for LeakyReLU as an option for choice of activation function. In order to prevent area from expanding too much via the incorporation of a full divider, we instead made the decision to limit the space of possible scaling factors to select powers of 2 (0.5, 0.25, and 0.125, to be specific). The choice of activation function is decided by reconfigurable inputs to the architecture (namely, relu_en and lrelu_en). The scaling factor can be contained in a simple two-bit input to the architecture, and is ignored unless lrelu_en is 1. To mimic this implementation of LeakyReLU, our software also only uses the aforementioned scaling factors, and uses the “floor” function to implement the clipping such that we only consider the integer portion of the output. Other than these changes, the software to generate our input files is identical to part 1. Our Quartus simulation showed that we used 17K logical elements for this optimization compared to the vanilla 16.8K, indicating minimal increase in area. We observed largely similar execution times compared to the ReLU-only implementation, suggesting that the extra hardware did not add significant overhead and that such flexibility did not cost us significantly in timing, power, or area.

Row and Column Parameterizability

We want to be able to support other configurations of the mac array besides the baseline 8x8. To support this, we extend support for the `row` and `col` parameters of the mac array by removing all hard coded values that rely on the assumption of an 8x8 array to use these parameters instead. The following are examples of this reliance on `row` and `col` in the mac_array.v:

Ex1: assign valid = valid_temp[row*col-1:row*col-col];

Ex2:
```
always @ (posedge clk) begin
    inst_w_temp[1:0]   <= inst_w;
    inst_w_temp[2*row-1:2]   <= inst_w_temp[2*row-1-2:0];
end
```

This was first tested to ensure that the testbench worked properly with 8x8 and we saw that nothing was changed with the logic after these changes and it worked as expected. Similarly, we repeated the file generation process from the vanilla version but instead generated files corresponding to 16x16, 16x8, and 8x16 channels. Accordingly, we configured the instantiated mac array with these parameters and ran the testbench for each of these configurations

separately. Ultimately, all configurations successfully passed the testbench with no errors showing and exhibited the expected behavior. This means that we are able to handle and support operations for a greater variety of layers beyond what the vanilla model requires with the use of these parameters. Similar to the multicore setup, using 16 columns at a time proved to be twice as fast as calculating 8 columns at a time. Similarly, since we did not have to break the execution flow to load in an extra set of weights (like row tiling requires), we also saw a 2x increase in throughput for the 16 rows by 8 columns case. While we did not actually implement tiling for a full 16 input channels and 16 output channels with 4-bit activations, we reasonably conclude that the 16x16 array achieves 4x throughput at the cost of 4x as much area and power costs.

Tiling

To support convolution layers whose input channel dimensions exceed the native SIMD x 2D systolic array capacity of the accelerator, a two tile input channel tiling mechanism was implemented. The design enables the core to process feature maps in multiple passes while preserving the correctness through in place partial sum accumulation. Tiling was integrated into both supported dataflows, weight-stationary (WS) and output-stationary (OS).

Weight-Stationary (WS) Tiling:

- The corelet holds weights constant in the PE array while input activations stream across it
- When the full channel depth does not fit in one pass, the layer is split into Tile 0 and Tile1.
- For Tile0, the array computes the initial partial sums which are stored in on-ship SRAM
- For Tile1, the system retrieves the Tile0 saved partial sums, feeds them back into the systolic array, and continues accumulating on them while processing the next set of input channels.
- Once all tiles have completed, the accumulated results correspond exactly to a full-channel convolution

Output-Stationary (OS) Tiling:

- In OS, the partial sums are kept inside the PEs instead of being fed back
- Each tile's activation and weights must be fully streamed to the array such that the MAC grid can produce a complete partial-sum block for its assigned mapping.
- Tile0 and Tile1 activations are pre-converted into output-stationary serialization format matching Part 3 and concatenated to form activation_os.txt.
- Since OS naturally accumulates partial sums inside the array, the software merges the results externally once both tiles have completed.

Observed Results: For both tiling modes, outputs matched the provided full-channel ground truths located in P16x8_Files/out_no_relu.txt and out_relu.txt.

In Place Accumulation

In-place accumulation was instituted in order to simplify the accumulation process. Previously, the sums for all tuples of (nij, kij) were generated for each output channel, then summed. To

perform tiling on input channels without exceeding SRAM memory limits, in-place accumulation would need to be performed for each input tile. A key observation was that accumulation over all kij for a particular choice of nij was possible using in-place accumulation as well. Implementing this simplified the programming required to perform the convolution. It also cut out inner product computations for pairs of (kij, nij) that were not used in the final output channel partial sums. Finally, this optimization reduced memory consumption by up to 16x relative to the base version, both because it did not have to store separate partial sums for each kij, and because it avoided storing the unnecessary partial sums. In weight-stationary, the computation was performed as follows:

- For each value of nij, allocate one address of psum SRAM. Each row stores 8 output channels for that nij, as described in lectures.
- For each allocated address of nij, set the address' values to zeros.
- For each value of (in_tile, kij)
    - Load kernel into PEs as normal
    - *Read current values from psum SRAM into IFIFO starting from nij=0*. During weight-stationary execution, the top row of PEs will perform accumulation on top of those previously accumulated values fed into the IFIFO.
    - Simultaneously reading values from activation sram into IFIFO and proceed with the WS computation as normal
    - As results become available in the OFIFO, load the resulting psums back into the *original* locations in the psum SRAM (nij=0 will become available first and that is assigned address=0, nij=1 should be popped next and written back to address=1, etc.).

As the output-stationary computation avoids breaking up the input channels into tiles (instead requiring the sum to happen all at once), the in-place accumulation technique described for weight-stationary does not apply.

Formal Verification

To ensure that our vanilla design (and, to a lesser extent, our part 2 implementation) are robust to the extreme input values, we generated artificial weights and activations in Python to test our design on values that may cause overflow. Specifically, we decided to set all of the weights to both the most negative and most positive values and the activations to the maximum possible magnitude (15). This would ensure the most negative and most positive (respectively) partial sums and outputs, so if no overflow error was encountered by our architecture, we could be certain that for any set of 4-bit activations and weights, our design would be robust against extreme inputs. The PyTorch code that was used to generate these weights are listed in our GitHub (the "FileGen" files). A similar approach was taken for Part 2, using 2-bit activations instead of 4-bit. Finally, we generated some completely random values for both part 1 and part 2, just to ensure that our design was not just operating well on some pattern, and could work on any arbitrary input values. Once we had verified that our designs worked as expected on all of these specially-designed inputs, we felt confident that our design was reasonably well-tested and should be able to handle any input values thrown at it.