TABLA Documentation

TABLA Overview

The TABLA platform for non-DNN algorithms can be customized to perform training and inference for a variety of machine learning algorithms, including linear regression, logistic regression, support vector machines, recommender systems, and backpropagation.

Architecture

As shown in Figure 1, the overall template-based TABLA architecture consists of multiple levels of hierarchy. An array of Processing Units (PUs) constitute the first level. The PUs are connected via two busing mechanisms – the "neighbor bus" and the "global bus." All PUs are connected to the global bus, and the neighbor bus connects adjacent PUs.

Bus arbiter: The bus arbiter (Figure 2) consists of a master controller per PU and a slave controller for each PE. The master controller determines which PE gets control of the bus in a given cycle. The slave controller has a write buffer and a set of read buffers. In each cycle, data is popped from the write buffer of the source PE and is written to the read buffer of the destination PE. The bus can be configured to either use mux or tri state logic depending on the availability of tri state cells in the technology libraries.

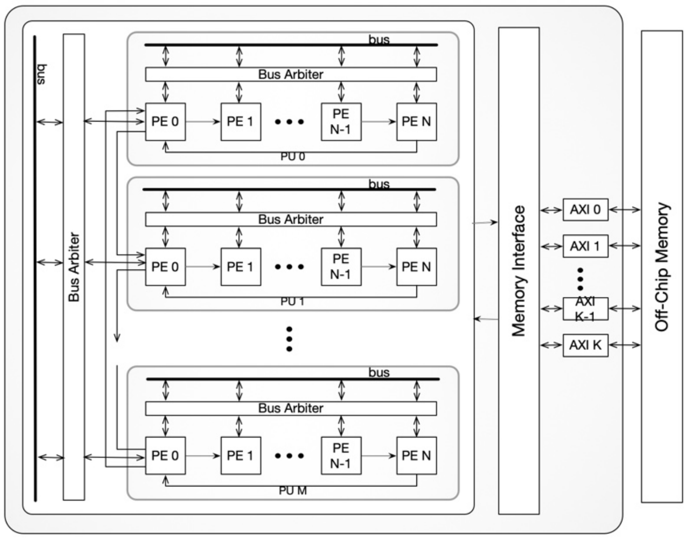

Figure 1: An overview of the template-based architecture for the TABLA non-DNN accelerator.

Memory interface: The memory interface (Figure 3) serves as a connector between main memory where data resides and accelerator memory where computation is carried out. The figure below describes the memory interface component in detail. The memory interface is designed to be parameterized in varying degrees of batch size, data size, and the number of AXI interfaces. A group of PEs are connected to Memory Lane (henceforth referred to as Lane) which connects the data buffer region from the AXI interfaces to PEs. Each Lane is connected to R PEs, and there is a total of N Lanes.

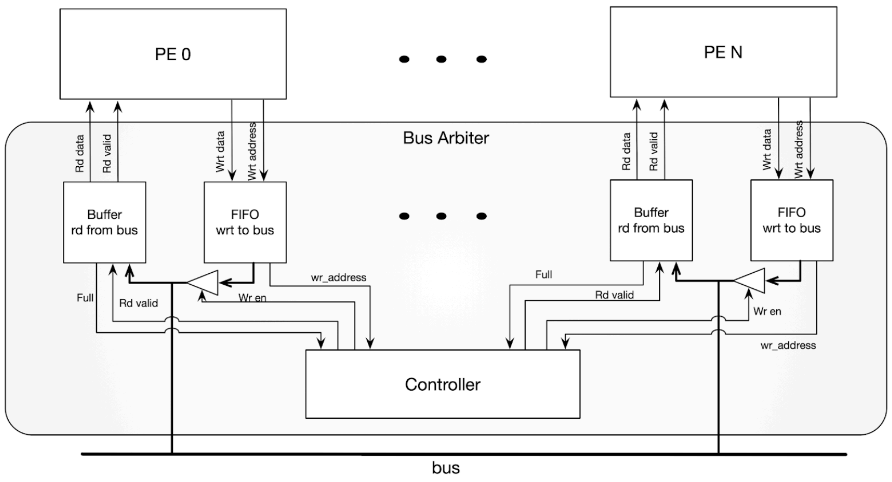

Figure 2: The bus arbiter in the TABLA non-DNN accelerator.

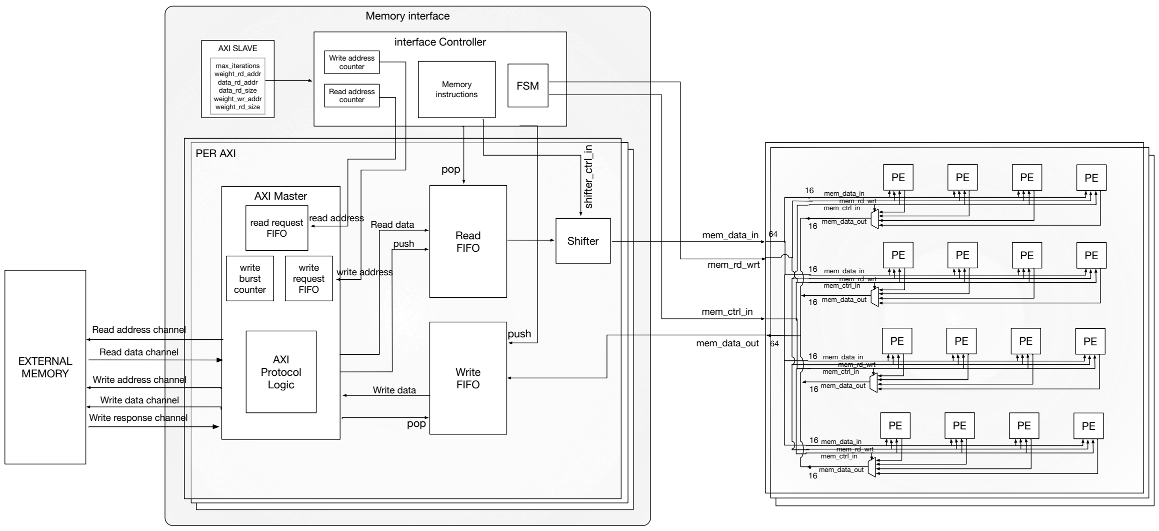

Figure 3: An overview of the memory interface for the TABLA non-DNN accelerator.

Currently this memory interface is based on the AXI protocol. The configuration which has been used extensively is the one with 4-AXI interfaces. However, the memory instruction generator has been fixed so that it can also generate memory instructions for a different AXI configuration, e.g. 1-AXI interface.

Lanes							**Opcode**	**Flag**
Lane 15		**Lane 14**		**. . . . .**	**Lane 0**			
relpe	**valid**	**relpe**	**valid**	**. . . . .**	**relpe**	**valid**		
2 bits	***1 bit***	***2 bits***	***1 bit***	**. . . . .**	***2 bits***	***1 bit***	***2 bits***	***4 bits***
0	0	0	0	. . . . .	0	0	read	
				. . . . .			shift	
0	0	0	0	. . . . .	0	0	wfi	0
0	0	0	0	. . . . .	0	0	loop	0

Table 1: Instruction set architecture for the memory interface.

ISA: Table 1 illustrates the instruction set architecture format for the memory interface. As shown in the table, there are four instructions supported in the memory interface ISA:

- The read instruction initiates a read from main memory. The flag bits indicate which AXI to read the data from. Each AXI reads in four input data at a time. In the example above, there are four AXIs, each of which reads in four input data, which means batches of 16 will be read in at a time.
- Once a batch of input data is read in, a number of shift instructions are executed on the input data to align the input data batches to corresponding memory lanes where its destination PE resides. A valid flag is also used to determine whether the corresponding lane will read in the shifted data or not. If a shift is enabled, (e.g., valid bit = 1), then the AXI data is read-in by the corresponding memory lane. The shift amount is encoded in the least significant 4 bits of the instruction, shown in the Flag column in the table above. The instruction also specifies the relative PE position (indicated by the "relpe" bits) for each memory lane. In the example in Table 1, there are four PEs assigned to each memory lane, each of which corresponds to a relative PE position value.
- The WFI instruction indicates the end of data read instructions and start of computation.
- The LOOP instruction polls for end of computation and starts executing from the first instruction again, i.e,, it reads new data and computes weights. This looping is performed for a programmable number of iterations.

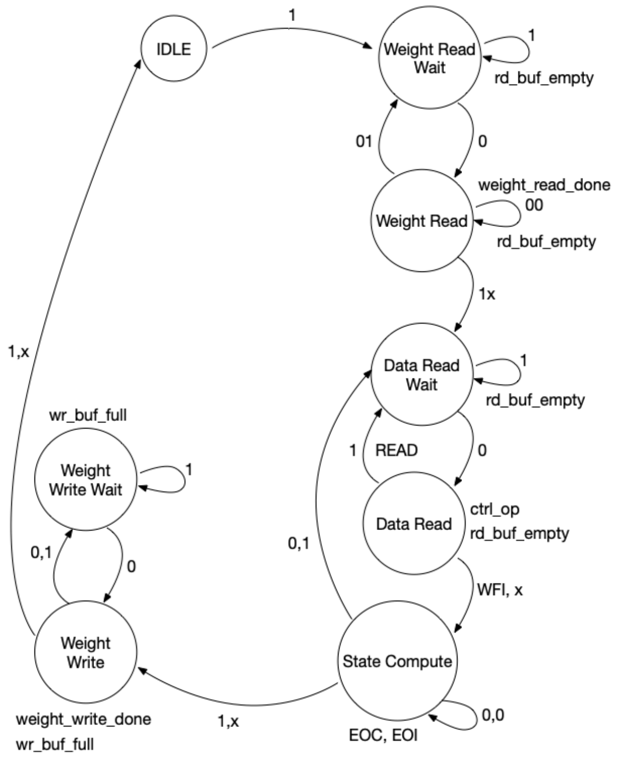

Figure 4: Finite State Machine (FSM) of the memory interface.

The finite state machine (FSM) for the memory interface is illustrated in Figure 4. The FSM is in IDLE state on reset and on receiving the start signal, it starts reading the weights. Once all the weights are read, it goes to data read phase. After the data is done, it moves to compute state. After each iteration it again goes to data read phase and this repeats for specified number of iterations. Then the weights are written back and it moves to IDLE state.

The memory instruction generator produces memory instructions according to the memory ISA described above. Using weights and training data as input, it generates synthesizable Verilog files that include the binary memory instructions using the ISA shown in Table 1.

Software simulator: The software simulator emulates the behavior of the accelerator. It simulates the architectural features describe above, such as PE global and neighbor bus communication and bus arbitration. The accelerator architecture the simulator targets is designed to be configurable, similar to how the template-based architecture is designed to be configurable. This allows experimentation using different set of configuration values such as number of PEs and number of PEs per PU. The simulator emits simulation statistics, including total number of cycles it took to complete the execution and resource utilization percentage of each architectural component. Finally, it comes with a feature to run the simulation in user-interactive mode, which allows user to run the program cycle-by-cycle and inspect the state of each component (e.g. values stored in PE namespaces after each cycle), similar to software debugger programs. This provides user with a full debugging capability of the program in the target accelerator architecture without having to rely on Verilog synthesis.

TABLA Performance Optimizations

A number of TABLA architecture improvements have been made to improve both performance and accuracy across non-DNN benchmarks. For performance, execution time was reduced by allowing instructions to begin execution as soon as the necessary data for execution has been received, instead of waiting for all data to transfer. In addition, timing of execution was improved by introducing pipeline registers to long routing paths. Furthermore, the aforementioned compiler optimization which allows validation of software computed benchmark results has been automated in an RTL testbench. Lastly, the memory ISA instruction generator has been enhanced to handle different AXI number configurations and a fix has been made to enforce correct ordering of shift instructions.

TABLA Implementation

The TABLA accelerator has been successfully synthesized and implemented both through an ASIC flow using the GF12nm technology with Arm libararies, and on a Xilinx KCU 1500 FPGA board. For this board, we have prepared the necessary execution environment to run other non-DNN ML algorithms in an end-to-end manner for TABLA on the FPGA platform.