Axiline Documentation

Axiline Overview

Axiline is a non-template-based framework for synthesizing small ML algorithms, creating efficient Verilog-based implementations by directly synthesizing the *mg*-DFG. Axiline supports linear regression, logistic regression, SVM and recommender systems.

Using the *mg*-DFG as well as its intuitive compiler pass infrastructure, we have developed an automatic Verilog generator using the input *mg*-DFG. The target algorithms for this Verilog generator are smaller algorithms which can be used to possibly compose larger programs or for generation of extremely efficient programs which could be run on the edge or elsewhere.

This pass consists of the following steps:

1. Lower the *mg*-DFG to consist of only scalar operations.
2. Traverse the lowered *mg*-DFG, and generate the Verilog module using *mg*-DFG input nodes as input signals, determine the required bit widths at each node, and create Verilog statements using the arithmetic operation at each node and its input edges.

The current programs that have been generated are working successfully, and have been tested and modeled. In addition, we are currently working on embedding values in place of variable names in the compiler pass. This is beneficial for small algorithms which are running inference, and do not have a need for updating the model/weight values, which can then be used as constants. Further, this allows optimizations to the algorithm itself, such as identifying constant values in multiplication operations which are a power of two, and can therefore be changed to bit-shift operations which are much more efficient.

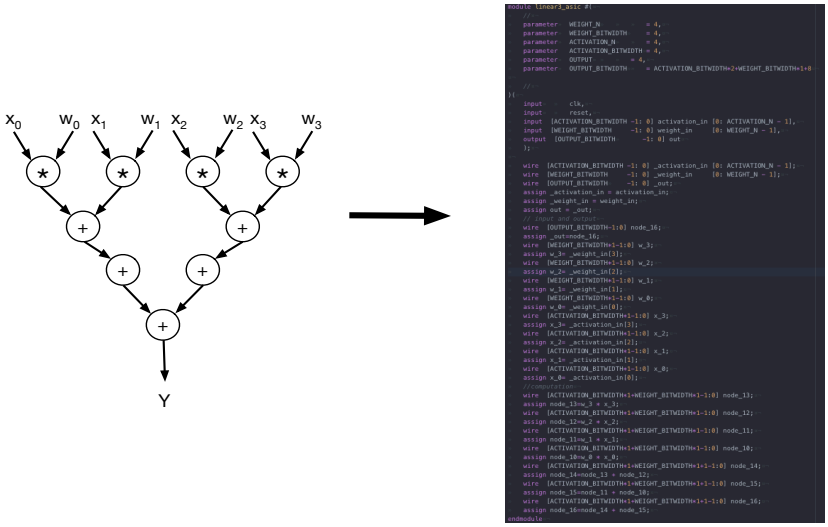

Figure 1: The *mg*-DFG for the simple regression function $Y = \sum_{i=0}^{3} x_i w_i$, and the corresponding Verilog code.

As an example of this Verilog generation for small algorithms, below is a visualization of a simple linear regression, $Y = \sum_{i=0}^{3} x_i w_i$, where x_i's constitute the set of input values that are multiplied

by a set of trained weight values, w_i, in order to perform inference. The translation of this *mg*-DFG to Verilog code is illustrated in Figure 1. A similar approach can be used for other small algorithms that can then be synthesized using non-template-based techniques to create optimal implementations. A snippet of the Verilog code generated for a recommender system is shown in Figure 2.

```
// asic_tempalte without weight and bias embedding and receiving them as input signals
//

module reco_sys #(
  //
  parameter WEIGHT_N      = 12,
  parameter WEIGHT_BITWIDTH     = 4,
  parameter ACTIVATION_N    = 4,
  parameter ACTIVATION_BITWIDTH = 4,
  parameter BIAS_N        = 6,
  parameter BIAS_BITWIDTH   = 4,
  parameter RATE_N        = 6,
  parameter RATE_BITWIDTH   = 4,
  parameter OUTPUT_N      = 12,
  parameter OUTPUT_BITWIDTH   = ACTIVATION_BITWIDTH*2+WEIGHT_BITWIDTH*1+BIAS_BITWIDTH*1+7
  //
)(
  input  [ACTIVATION_BITWIDTH -1: 0] x1_0_,
  input  [ACTIVATION_BITWIDTH -1: 0] x1_1_,
  input  [ACTIVATION_BITWIDTH -1: 0] x2_0_,
  input  [ACTIVATION_BITWIDTH -1: 0] x2_1_,

  input  [WEIGHT_BITWIDTH     -1: 0] w1_0_0,
  input  [WEIGHT_BITWIDTH     -1: 0] w1_0_1,
  input  [WEIGHT_BITWIDTH     -1: 0] w1_1_0,
  input  [WEIGHT_BITWIDTH     -1: 0] w1_1_1,
  input  [WEIGHT_BITWIDTH     -1: 0] w1_2_0,
  input  [WEIGHT_BITWIDTH     -1: 0] w1_2_1,
  input  [WEIGHT_BITWIDTH     -1: 0] w2_0_0,
  input  [WEIGHT_BITWIDTH     -1: 0] w2_0_1,
  input  [WEIGHT_BITWIDTH     -1: 0] w2_1_0,
  input  [WEIGHT_BITWIDTH     -1: 0] w2_1_1,
  input  [WEIGHT_BITWIDTH     -1: 0] w2_2_0,
  input  [WEIGHT_BITWIDTH     -1: 0] w2_2_1,

  input  [BIAS_BITWIDTH       -1: 0] y1_0_,
  input  [BIAS_BITWIDTH       -1: 0] y1_1_,
```

Figure 2: Snippet of generated Verilog code for a recommender system.

We develop the PPA models using regression models to help us predict the area and power for a specific delay value. We divide the DFG into some blocks, extract features for each block from DFG such as input size, bitwidth, add delay as a feature to train/predict the area and power. There is a trade-off when we divide the DFG into some blocks: the smaller each block is, the easier it is to extract its features, and the higher its regression accuracy. However, this also implies that the system has a larger number of blocks, which could potentially lead to larger errors in combining them. Figure 3 shows the result of using polynomial regression to predict the total power for an inner product. For a polynomial regression model with 95 training/test samples, the average coefficient of determination for cross validation is 0.9677. This result is better than the case where smaller blocks (multipliers and adders) are characterized and combined.

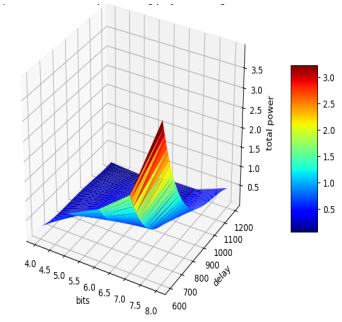

Figure 3: Polynomial regression model for the PPA of an inner product.

Pipelined Axiline Architectures: To enable the applicability of Axiline to larger designs, the Verilog generator employs a pipelined architecture. The use of pipelining allows Axiline to tackle large benchmarks, reducing the demand for memory bandwidth is included with parameterized pipeline stages. From the generated RTL, PPA curves were generated using a Synopsys DC + Innovus flow.

To develop a pipeline architecture for a set of non-DNN benchmarks, we map the DFG into 3 pipeline stages, shown in Figure 4. Stage 1 is for the inner product and Stage 2 for a combinational logic depending on benchmarks. For example, for linear regression, the combinational logic in block 2 would be a multiplier, and for logistic regression benchmark, it should be a sigmoid function and a multiplier. Block 3 is for stochastic gradient descent, consisting of two multipliers and one adder. The inner product size in Stage 1 is parameterized. Therefore, the input bandwidth can be parameterized for different FPGAs.

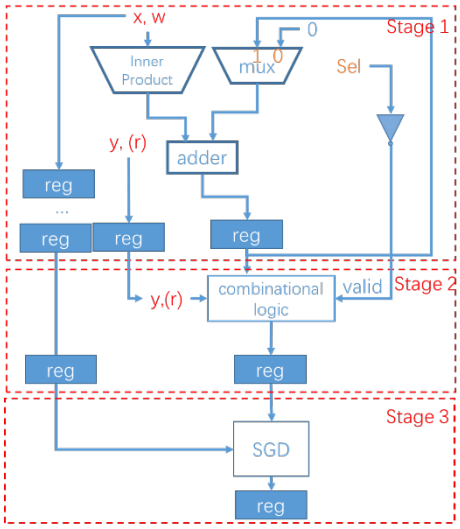

Figure 4: Pipeline implementation for Axiline benchmarks

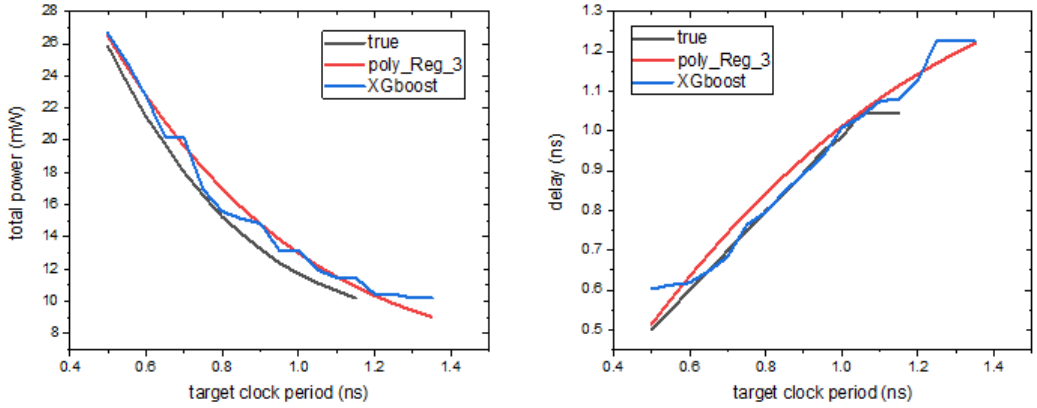

Figure 5: Axiline SVM54 PPA curve (pre-P&R) and predictions

Using such a pipelined implementation, we use different configurations to synthesize the RTL and generate PPA curves. Given the PPA curves for some configuration as training data (including

bitwidth, inner product size, target clock frequency and benchmarks as features), we use an ML model to predict the PPA for other configurations. Figure 5 shows the PPA prediction for SVM 54, we use ML models to predict the PPA for pipeline architecture with bitwidth = 8, inner product size = 6. Some PPA curves with other configurations are used as training data. Polynomial regression and XGBoost are used to predict the PPA. These results indicate that both polynomial regression and XGBoost can predict the total power and delay well.