Sabancı University

EE310 - Hardware Description Languages

Chisel-based Convolution Module Design and Simulation

Lab 4 - Report

Submitted by:

Kadir Yağız EBİL

May 13, 2025

Contents

1 Introduction

This report details the of the implementation of various convolution modules using Chisel hardware description language. The implemented modules include:

- DotProduct module
- TemporalComputation trait
- PointwiseConvolution class
- RegularConv class

The design follows a modular approach utilizing object-oriented concepts such as abstract classes, traits, and inheritance to create a flexible and reusable hardware architecture as we learned in the recent weeks.

2 Design Methodology

2.1 General Architecture

The design is structured around a base abstract class `BaseConvolution` that defines the common interface and parameters for all convolution types. This is extended by specialized modules that implement specific convolution operations each differ little bit in terms of depth and filter:

- `DepthwiseConv`: Performs spatial convolution on each input channel separately
- `PointwiseConvolution`: Performs 1x1 convolution across channel dimensions
- `RegularConv`: Combines both spatial and channel dimensions for full convolution.

Computation patterns are abstracted into traits that can be mixed into these classes:

- `SpatialCompute`: Provides 2D convolution functionality
- `TemporalCompute`: Provides cross-channel computation

2.2 Data Types and Precision

A key design decision was to implement all the modules using signed integers (`SInt`) rather than unsigned integers (`UInt`). This choice was made to properly support convolution operations with negative filter weights, which are common in neural network applications. The implementation supports:

- 8-bit signed inputs and weights
- 32-bit signed outputs to accommodate accumulation without overflow

3 Implementation Details

3.1 TemporalComputation Trait

The `TemporalCompute` trait encapsulates the pattern of computing across the channel dimension at a single spatial location. This abstraction is particularly useful for the pointwise convolution but can be reused in other contexts.

Key features of this implementation:

- Takes a 3D input tensor and a 1D kernel as inputs
- Uses a `DotProduct` module to compute the weighted sum across channels
- Returns a single scalar value representing the result at one spatial location

The trait leverages Chisel's self-type annotation to ensure it's only mixed into classes that extend `BaseConvolution`.

3.2 PointwiseConvolution Class

The `PointwiseConvolution` class implements 1×1 convolution that mixes information across channels. It extends `BaseConvolution` and mixes in the `TemporalCompute` trait:

- Maintains spatial dimensions (height and width) from input to output
- Changes channel dimension based on the number of filters
- For each output channel and each spatial position, extracts the relevant weights and input values
- Uses `temporalConvolve` to compute the dot product across all input channels

3.3 RegularConv Class

The `RegularConv` class implements standard convolution that processes both spatial and channel dimensions:

- Performs the most comprehensive form of convolution
- Creates a reduced spatial output (based on filter size)
- For each output channel, position, and input channel, extracts a spatial patch
- Uses separate `DotProduct` modules for each channel's spatial contribution
- Accumulates results across all input channels

While this implementation could have reused the `TemporalCompute` or `SpatialCompute` trait, I chose to implement it directly for better control over the indexing patterns and to optimize the hardware structure in the first step. Then I observed we needed to choose one to usit as trait. I choosed `TemporalCompute`.

3.4 Rewriting `RegularConv` to Extend `TemporalCompute`

In order to integrate the `TemporalCompute` trait with the `RegularConv` class, I explored an approach that both extends the trait and preserves the full spatial-plus-channel convolution behavior. Below is my refined implementation:

Listing 1: `RegularConv` Extending `TemporalCompute`

```
class RegularConv(
  h_in: Int, w_in: Int, d_in: Int,
  d_out: Int, k_h: Int, k_w: Int
) extends BaseConvolution(
  h_in, w_in, d_in,
  k_h, k_w, d_in, d_out,
  (h_in - k_h + 1), (w_in - k_w + 1), d_out
) with TemporalCompute {
  val h_out_local = h_in - k_h + 1
  val w_out_local = w_in - k_w + 1

  for (f <- 0 until d_out) {
    for (j <- 0 until w_out_local) {
      for (i <- 0 until h_out_local) {
        // Compute spatial convolution per channel, then accumulate across
        val channelResults = (0 until d_in).map { c =>
          // Flatten spatial patch and weights
          val patch1D = Reg(Vec(k_h * k_w, SInt(8.W)))
          val weight1D = Reg(Vec(k_h * k_w, SInt(8.W)))
          var idx = 0
          for (kh <- 0 until k_h; kw <- 0 until k_w) {
            patch1D(idx)  := io.input(c)(i + kh)(j + kw)
            weight1D(idx) := io.weights(f)(c)(kh)(kw)
            idx += 1
          }
          // Spatial dot product
          val dot = Module(new DotProduct(k_h * k_w))
          dot.io.a := patch1D
          dot.io.b := weight1D
          dot.io.result
        }

        // Sum spatial results across channels
        io.output(f)(j)(i) := channelResults.reduce(_ + _)
      }
    }
  }
}
```

Discussion

- Although the `TemporalCompute` trait provides a `temporalConvolve` helper for

channel-wise dot products, regular convolution requires interleaving spatial and channel dimensions.

- Here, we perform a spatial dot product per channel (using `DotProduct`), then sum those results exactly as before.
- This pattern aligns with the intent of `TemporalCompute` (computing across channels), while preserving full 2D convolution behavior.
- By extending `TemporalCompute`, the class signals that it conforms to the same channel-wise convolve interface, even if we do not call `temporalConvolve` directly.

This revised implementation thus both mixes in the `TemporalCompute` trait and maintains the original functionality of the `RegularConv` class.

4 Design Decisions and Optimizations

4.1 Signed Number Support

The original template appeared to be using unsigned integers (`UInt`), but the test examples clearly showed negative values in the filters. I modified all modules to use signed integers (`SInt`) to correctly handle negative values in both inputs and weights. I dont know if this is right or wrong :(but I did this because python examplese worked on negative values. Hope I don't encounter any overflows in testing.

4.2 Hardware Reuse

The implementation creates dedicated hardware for each computation instance rather than time-multiplexing a single unit. This increases parallelism at the cost of area. In a real-world design, this tradeoff would depend on performance requirements and available resources.

4.3 Indexing Considerations

Careful attention was paid to indexing patterns after miserably failing to ensure correct mapping between the Chisel implementation and the expected behavior shown in the Python reference code. This required adjusting the access patterns in the RegularConv class to match the row-major ordering used in the Python examples.

5 Testing Approach

Each module was tested using the Chisel test infrastructure with a dedicated test suite that verifies functionality against known reference outputs. The tests follow these steps:

1. Define input tensors and filter weights with both positive and negative values
2. Apply these inputs to the Chisel module
3. Calculate expected outputs manually based on the convolution equations

4. Compare module outputs against expected values
5. Print debug information for verification

6 Conclusion

The implemented convolution modules successfully demonstrate how complex operations can be abstracted and composed using Chisel's object-oriented features. The use of traits and inheritance allowed for code reuse across different convolution types while maintaining specialized behavior for each.

This approach to hardware design offers significant advantages for complex architectures like neural network accelerators, where different convolution operations need to be mixed and matched in various configurations.