REUSABLE BLOCKS 14/01

- Investigating sequential + parallel loops -> Difficulty in understanding what part of my function can be done sequencially
- I would like to look at this code together. The aim is to divide the number of times you multiply by the reuse factor, isn't it? This means you do a block of multiplications at the same time, and then start the next one using the same resources. How can I apply this concept to the layers?

```
ReuseLoop:
    for (int ir = 0; ir < rufactor; ir++) {
        #pragma HLS PIPELINE II=1 rewind

        int w_index = ir;
        int in_index = ir;
        int out_index = 0;
        int acc_step = 0;

    MultLoop:
        for (int im = 0; im < block_factor; im++) {
            #pragma HLS UNROLL

            acc[out_index] += static_cast<typename CONFIG_T::accum_t>(
                CONFIG_T::template product<data_T, typename CONFIG_T::weight_t>::product(data[in_index], weights[w_index]));

            // Increment w_index
            w_index += rufactor;
            // Increment in_index
            in_index += rufactor;
            if (in_index >= nin) {
                in_index = ir;
            }
            // Increment out_index
            if (acc_step + 1 >= multscale) {
                acc_step = 0;
                out_index++;
            } else {
                acc_step++;
            }
        }
    }

// Cast to "res_t" type
Result:
    for (int ires = 0; ires < CONFIG_T::n_out; ires++) {
        #pragma HLS UNROLL
        res[ires] = cast<data_T, res_T, CONFIG_T>(acc[ires]);
    }
```

```
template<int in_size, int out_size>
void dense_layer(
    hls::stream<data_t>& input_stream,
    hls::stream<data_t>& output_stream,
    data_t weights[out_size][in_size],
    data_t bias[out_size]
) {
    data_t input_buffer[in_size];

    for (int i = 0; i < in_size; i++) {
        input_buffer[i] = input_stream.read();
    }

    data_t output_buffer[out_size];
    for (int i = 0; i < out_size; i++) {
        data_t acc = 0;
        for (int j = 0; j < in_size; j++) {
            acc += input_buffer[j] * weights[i][j];
        }
        output_buffer[i] = acc + bias[i];
    }

    for (int i = 0; i < out_size; i++) {
        output_stream.write(output_buffer[i]);
    }
}
```

REUSABLE BLOCKS 14/01

- Other attempts

```
dense_layer<INPUT_SIZE, OUTPUT_SIZE>(input_stream, output_stream1, weights, bias);

dense_wrapper(output_stream1, input_stream2, weights1, bias1, weights_resized, bias_resized);

dense_layer<INPUT_SIZE, OUTPUT_SIZE>(input_stream2, output_stream, weights_resized, bias_resized);
```

Modules & Loops	Issue Type	Violation Type	Latency(ns)	Interval	Pipelined	BRAM	DSP	FF	LUT
top	II Violation		4.518E4	4519	no	3	128	7121	4886

```
#pragma HLS ALLOCATION function instances=dense_layer<INPUT_SIZE, OUTPUT_SIZE> limit=1

dense_layer<INPUT_SIZE, OUTPUT_SIZE>(input_stream, output_stream1, weights, bias);

dense_wrapper(output_stream1, input_stream2, weights1, bias1, weights_resized, bias_resized);

dense_layer<INPUT_SIZE, OUTPUT_SIZE>(input_stream2, output_stream, weights_resized, bias_resized);
```

Modules & Loops	Issue Type	Violation Type	Latency(ns)	Interval	Pipelined	BRAM	DSP	FF	LUT
top			4.449E4	4450	no	2	64	3708	2932
dense_layer_64_32_s	II Violation		1.197E4	1197	no	1	64	3415	2079
top_Pipeline_VITIS_LOOP_41_1			680.000	68	no	1	0	29	128
top_Pipeline_VITIS_LOOP_50_2_VITIS_LOOP_51_3			2.050E4	2050	no	0	0	40	229
top_Pipeline_VITIS_LOOP_60_4			340.000	34	no	0	0	16	90

```
dense_layer<INPUT_SIZE, OUTPUT_SIZE>(input_stream, output_stream1, weights, bias);

dense_layer<INPUT_SIZE1, OUTPUT_SIZE1>(output_stream1, output_stream, weights1, bias1);
```

Modules & Loops	Issue Type	Violation Type	Latency(ns)	Interval	Pipelined	BRAM	DSP	FF	LUT
top	II Violation		1.548E4	1549	no	1	96	5306	3333

REUSABLE BLOCKS 14/01

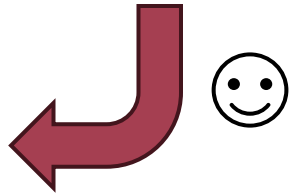
- Other attempts with UNROLL in dense_layer

```
dense_layer<INPUT_SIZE, OUTPUT_SIZE>(input_stream, output_stream1, weights, bias);

dense_wrapper(output_stream1, input_stream2, weights1, bias1, weights_resized, bias_resized);

dense_layer<INPUT_SIZE, OUTPUT_SIZE>(input_stream2, output_stream, weights_resized, bias_resized);
```

```
#pragma HLS ALLOCATION function instances=dense_layer<INPUT_SIZE, OUTPUT_SIZE> limit=1

dense_layer<INPUT_SIZE, OUTPUT_SIZE>(input_stream, output_stream1, weights, bias);

dense_wrapper(output_stream1, input_stream2, weights1, bias1, weights_resized, bias_resized);

dense_layer<INPUT_SIZE, OUTPUT_SIZE>(input_stream2, output_stream, weights_resized, bias_resized);
```

Modules & Loops	Issue Type	Violation Type	Latency(ns)	Interval	Pipelined	BRAM	DSP	FF	LUT
top			4.449E4	4450	no	2	64	3708	2932
dense_layer_64_32_s	II Violation		1.197E4	1197	no	1	64	3415	2079
top_Pipeline_VITIS_LOOP_41_1			680.000	68	no	1	0	29	128
top_Pipeline_VITIS_LOOP_50_2_VITIS_LOOP_51_3			2.050E4	2050	no	0	0	40	229
top_Pipeline_VITIS_LOOP_60_4			340.000	34	no	0	0	16	90

Same with or without pragma allocation



```
dense_layer<INPUT_SIZE, OUTPUT_SIZE>(input_stream, output_stream1, weights, bias);

dense_layer<INPUT_SIZE1, OUTPUT_SIZE1>(output_stream1, output_stream, weights1, bias1);
```

Modules & Loops	Issue Type	Violation Type	Latency(ns)	Interval	Pipelined	BRAM	DSP	FF	LUT
top			1.549E4	1550	no	1	96	5311	3415
dense_layer_64_32_s	II Violation		1.197E4	1197	no	1	64	3415	2079
dense_layer_32_16_s	II Violation		3.490E3	349	no	0	32	1791	1207

REUSABLE BLOCKS 14/01

- Next steps:

i. Keep looking at the idea of sequential and parallel loops;

ii. Keep following the other path, but using more dense_layer calls. Keep 2 or 3 main templates, as suggested by Nicolo.

    - Things I have to remember:

        1. Add the index to ensure that the right number of multiplications is performed;
        2. Add the section about increasing dimensions;
        3. Try to add all the pragmas that hls4ml usually generates.