

Evaluating Existing Verilog Designs and Designing a GEMM

Thomas Wöhrle

Practical Course – Creation of Deep Learning Methods

Technische Universität München

March 2024

Acknowledgements

This project was conducted as part of the “Creation of Deep Learning Methods” Practical Course at the Technical University Munich. All external sources are cited where relevant using [<reference number>].

Special thanks to Vladimir Golkov and Dirk Stober who provided guidance and supervision throughout the project. The hardware was kindly set up by Xuanshu Luo.

The code snippets were created using Snappify [14].

References



https://bit.ly/woehrle_bpc_references

Background

High-Level Goal of the Topic Area

Implementing Machine Learning architectures directly on hardware

- Goal is implementation from the ground up
- Pre-implemented components should be used only where necessary
- Therefore, an HDL needs to be employed
- Gain deeper understanding of the inner workings of Machine Learning architectures
- Get experience working with HDLs



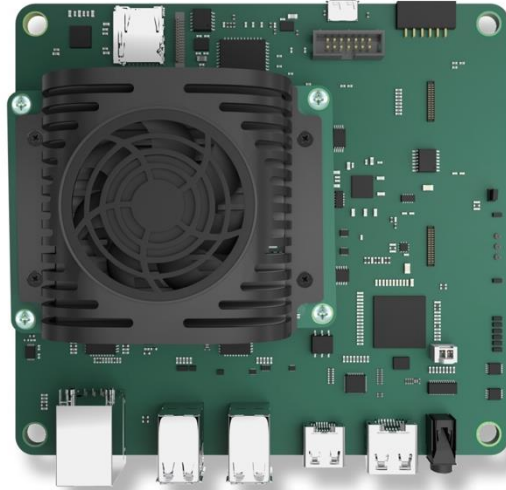
Hardware Description Languages (HDLs)

A Hardware Description Language (HDL) is a specialized computer language used to model, design, and simulate electronic systems [2]. In this project Verilog is used [3].

Verilog:

- Efficient digital circuit modeling and simulation
- Uses a familiar C-like syntax
- Widely adopted in ASIC and FPGA design
- More popular in the US and Asia compared to VHDL
- Standardized under IEEE 1364

Hardware



Kria KV260 Vision AI Starter Kit [1]

(only used sparingly up to this point of the project; most of the work happened in simulation)

Prior Work

A fundament of work has been done by Šimon Růžička and Phillip Wondra in the WS 23/24 [4]

Their Work includes:

- Floating Point Arithmetic Operations
- Basic Vector Operations
- Basic Matrix Operations
- Activation Functions
- A rudimentary neural layer
- A first attempt at Backpropagation
- A first attempt to deploy the project on hardware

Enhancing And Verifying The Existing Codebase [18]

Challenges in the existing codebase

Technical Challenges:

- Only basic verification of designs
- Inherent shortcoming of matrix multiplication design



Development Practice Challenges:

- Formatting
- Unwanted include directives

As a first step, the goal was to increase test coverage and introduce new tooling f.e. a linter

(More regarding the shortcoming of the matrix multiplication design later)

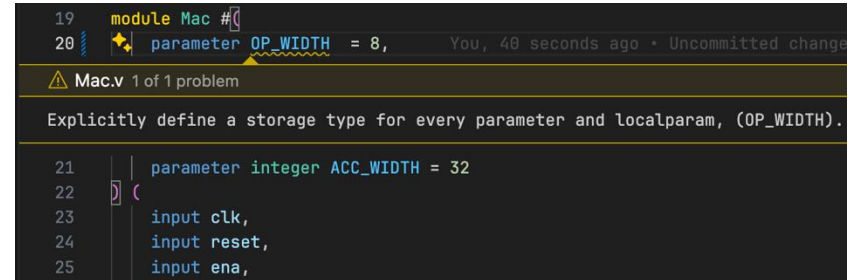
Verible [5]

"Suite of SystemVerilog developer tools":

- Linter
- Formatter
- Language Server

Usage:

- Installable via VScode market place
- Configurable via verible.fileist file
- No syntax highlighting -> separate extension for that
- See the GoogleDoc for more information regarding the toolchain [6]



```

19 module Mac #(
20     parameter OP_WIDTH = 8,
21     parameter integer ACC_WIDTH = 32
22 ) (
23     input clk,
24     input reset,
25     input ena,

```

Mac.v 1 of 1 problem

Explicitly define a storage type for every parameter and localparam, (OP_WIDTH).

Design Verification

Using SystemVerilog is the most common way to verify Verilog designs

Verification entails DUT instantiation, clock and reset generation, driving input values, time control and results monitoring.

And while SystemVerilog is widely used, it has downsides like a steeper learning curve, toolchain complexity, somewhat difficult testbench reusability among others.

```

23     VectorAdditionFlex #(LBUF(LBUF)) VA1 (.A(A), .B(B), .clk(clk), .l(length), .result(result), .done(done));
24
25     DisplayFloat display_result1 (.num(result[0 +: 32]), .id("000"), .format(1'b1));
26     DisplayFloat display_result2 (.num(result[32 +: 32]), .id("001"), .format(1'b1));
27     DisplayFloat display_result3 (.num(result[64 +: 32]), .id("002"), .format(1'b1));
28     DisplayFloat display_result4 (.num(result[96 +: 32]), .id("003"), .format(1'b1));
29     DisplayFloat display_result5 (.num(result[128 +: 32]), .id("004"), .format(1'b1));
30     DisplayFloat display_result6 (.num(result[156 +: 32]), .id("005"), .format(1'b1));
31
32     initial
33     begin
34         length[0 +: 32] = 32'd3;
35         #1
36         A[0 +: 32] = 32'b0_10000000_10011001100110011001100; // 3.2
37         B[0 +: 32] = 32'b0_10000001_00001100110011001100110; // 4.2
38
39         A[32 +: 32] = 32'b0_01111110_01010001111010111000010; // 0.66
40         B[32 +: 32] = 32'b0_01111110_00000101000111101011100; // 0.51
41
42         A[64 +: 32] = 32'b1_01111110_00000000000000000000000; // -0.5
43         B[64 +: 32] = 32'b1_10000001_10011001100110011001100; // -6.4
44
45         A[96 +: 32] = 32'b1_01111110_00000000000000000000000; // -0.5
46         B[96 +: 32] = 32'b0_10000001_10011001100110011001100; // 6.4
47
48         A[128 +: 32] = 32'b1_01111110_00000000000000000000000; // -0.5
49         B[128 +: 32] = 32'b1_10000001_10011001100110011001100; // -6.4
50
51         A[160 +: 32] = 32'b1_01111110_00000000000000000000000; // -0.5
52         B[160 +: 32] = 32'b0_10000001_10011001100110011001100; // 6.4
53
54         #100;
55     end

```

Cocotb [7]

The mentioned issues of SystemVerilog inspired the creation of frameworks in higher-level programming languages.

Cocotb is such a framework in the realm of Python. Its adoption in industry right now seems to be low, but there is growing interest in it. [8]

(The reddit post in [8] mentions Fintech as a heavy user of cocotb and I made a similar anecdotal finding while interacting with the community on GitHub)

Advantages:

- Offers familiar Python API
- Open-Source (if you consider that an advantage)
- Good documentation
- Active community answering questions [9, 10]

The first point is specifically important, because it enables access to all python libraries, easy randomness in the verification and very flexible and reusable tests (at least in theory).

Cocotb

On the right side a very simple case of a cocotb testbench can be seen, showcasing the basic intuition of how it works.

Cocotb is coroutine-based [11]. The documentation is very good and should be referred to when getting started with the topic.

In the following, I will present the way I created tests for the different kinds of existing modules using cocotb. A separate explanation of the tests I created for the modules I designed myself can be found later in this presentation

```
module adder (  
    input in1,  
    input in2,  
    output [1:0] out  
);  
  
    assign out = in1 + in2;  
  
endmodule
```

```
@cocotb.test()  
async def test_adder_basic(dut):  
    dut.in1.value = 1  
    dut.in2.value = 1  
  
    await Timer(1, units='sec')  
  
    assert dut.out.value == 2
```

Verification: Floating Point Arithmetic

Verification of Floating Point Arithmetic is rather simple because the existing design is fully combinational.

The example on the right side shows how Python's functionality enables high reusability and thus efficient testing.

A base class is created basically representing the inputs and outputs that flow into a Verilog module

```
class FloatingPointBaseTest(BaseTest):  
    def __init__(self, dut, A, B):  
        self.dut = dut  
        self.A = A  
        self.B = B  
        self.result = None  
  
    def assign_input(self):  
        self.dut.A.value = utils.float_to_ieee754(self.A)  
        self.dut.B.value = utils.float_to_ieee754(self.B)  
  
    def assign_output(self):  
        self.result = utils.ieee754_to_float(self.dut.result.value)
```

Verification: Floating Point Arithmetic

This FloatingPointBaseTest is then subclassed to create specific test classes for each DUT – for example FloatingAddition, FloatingMultiplication, etc.

Through this practice, bugs in the FloatingCompare and FloatingMultiplication modules could be found

```
class FloatingAdditionTest(test_types.FloatingPointBaseTest):
    def assert_result(self):
        assertions.assert_addition(self.A, self.B, self.result)
```

```
def assert_addition(a, b, result):
    expected = a + b
    tolerance = max(utils.get_tolerance(a), utils.get_tolerance(b))

    assert abs(result - expected) < tolerance, \
        f"Mismatch: {a} + {b} = {result} (expected {expected})"
```

```
@cocotb.test()
async def test_floating_addition_random_full_range(dut):
    test = FloatingAdditionTest(dut, None, None)
    max_val = utils.IEEE754_MAX_VAL / 2
    for _ in range(1000):
        a, b = utils.sample_a_and_b(-max_val, max_val)
        test.A = a
        test.B = b
        await test.exec_test()
```


Verification: Vector Arithmetic

Same approach: Separate class and subclass for the tests

```
class VectorAdditionTest(test_types.VectorBaseTest):
    def assert_result(self):
        for i in range(len(self.result)):
            assertions.assert_addition(self.A[i], self.B[i], self.result[i])
```

```
@cocotb.test()
async def test_vector_addition_flex_random_full_range(dut):
    test = VectorAdditionFlexTest(dut, [], [])
    max_val = utils.IEEE754_MAX_VAL / 2

    for i in range(1000):
        # see comment above regarding why 128
        length = random.randint(1, 128)
        test.A = utils.sample_array(-max_val, max_val, length)
        test.B = utils.sample_array(-max_val, max_val, length)
        await test.exec_test()
```

[robhughey.com](https://github.com/robhughey/cocotb)

```
class VectorAdditionFlexTest(test_VectorAddition.VectorAdditionTest):
    def __init__(self, dut, A, B):
        super().__init__(dut, A, B)
        self.clk = Clock(self.dut.clk, 1, units="ns")
        assert len(A) == len(B)
        self.l = len(A)

    def assign_input(self):
        # clk is already assigned
        # Assigns A and B
        super().assign_input()
        self.dut.l.value = self.l

    async def exec_test(self):
        self.assign_input()

        await Timer(1, "ns")
        await cocotb.start(self.clk.start())

        await Timer(self.l + 1, "ns")

        assert (self.dut.done.value, 1)

        # x bits should be resolved to 0, since the Flex module creates vectors b
        # export COCOTB_RESOLVE_X=ZEROS
        self.assign_output(width=self.l)
        self.assert_result()
```

Verification: Matrix Multiplication

Again, the same approach

```
class MatrixMultiplicationFlexTest(test_types.BaseTest):
    def __init__(self, dut, A: np.ndarray, B: np.ndarray):
        self.dut = dut
        self.A = A
        self.B_T = B.transpose()
        self.l = None
        self.m = None
        self.n = None
        self.clk = Clock(self.dut.clk, 1, "ns")
        self.result = None

    def assign_input(self):
        assert len(self.A.shape) == len(self.B_T.shape) == 2, f"len(self.A.shape)={self.A.shape}, len(self.B_T.shape)={self.B_T.shape}"
        assert self.A.shape[1] == self.B_T.shape[1]
        self.dut.A.value = utils.array_to_packed_integer(self.A.flatten())
        self.dut.B_T.value = utils.array_to_packed_integer(self.B_T.flatten())
        self.l = self.A.shape[0]
        self.m = self.A.shape[1]
        self.n = self.B_T.shape[0] # B_T is transposed
        self.dut.l.value = self.l
        self.dut.m.value = self.m
        self.dut.n.value = self.n

    def assign_output(self):
        temp = utils.packed_integer_to_array(
            self.dut.result.value, self.l * self.n
        )
        self.result = np.array(temp).reshape(self.l, self.n)

    def assert_result(self):
        try:
            for i in range(self.result.shape[0]):
                for j in range(self.result.shape[1]):
                    assertions.assert_vector_multiplication(
                        self.A[i], self.B_T[j], self.result[i][j])
        except AssertionError as e:
            print("Tried to calculate:")
            print(f"{self.A} @ \n{self.B_T.transpose()}")
            raise e

    async def exec_test(self):
        self.assign_input()

        await Timer(1)
        await cocotb.start(self.clk.start())

        while not self.dut.done.value:
            await Timer(1, "ns")

        self.assign_output()
        self.assert_result()
```

An Issue in The Existing Codebase

The existing Matrix Multiplication

- In the existing codebase, there were 3 modules for matrix multiplication: MatrixMultiplicationPar, MatrixMultiplicationSeq, MatrixMultiplicationFlex
- Par can be dismissed, because it is completely parallel, which obviously does not scale well
- Seq and Flex have a sequential nature, but they have some shortcomings:
 - They are based on multiple VectorMultiplications and Additions, which requires more and also more complex operations compared to systolic arrays (introduced later)
 - Long clock cycles
 - Require one operand to be transposed
 - Use of non-synthesizable elements (like “initial”)
 - ...
- Other minor shortcomings, like no reset signal, no clearly defined states

Fixing the Shortcomings

For these reasons and after talking with Dirk, we decided that it is a good idea to design a new General Matrix Multiply (GEMM) module.

This new GEMM should be based on systolic arrays, which are more efficient, more closely aligned with FPGA resources, inherently pipelined among other things. They are introduced in the next chapter of this presentation.

Since I was new to FPGA design, before starting to design the GEMM, I had to get deeper knowledge about the hardware design process. The resources I used in this process are documented in the GoogleDoc of the practical course.

Implementing a New GEMM [17]

High-Level Inspiration: Google TPU [12]

- Speeds up neural network computations with efficient matrix multiplications
- Designed for datacenter ML workloads
- Uses a custom systolic array architecture tailored for deep learning. This happens in the Matrix Multiply Unit pointed to on the right
- The TPU is only meant for inference and uses 8-bit quantization for that, boosting efficiency while maintaining acceptable accuracy

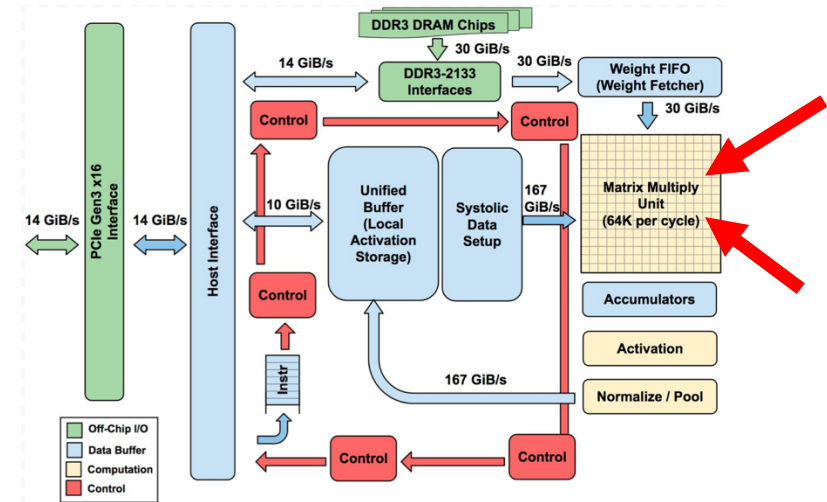


Figure 1. TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit in the upper right hand corner. Its inputs are the blue Weight FIFO and the blue Unified Buffer (UB) and its output is the blue Accumulators (Acc). The yellow Activation Unit performs the nonlinear functions on the Acc, which go to the UB.

Goal: Create a GEMM, that does similar 8-bit matrix multiplication using systolic arrays

Systolic Arrays

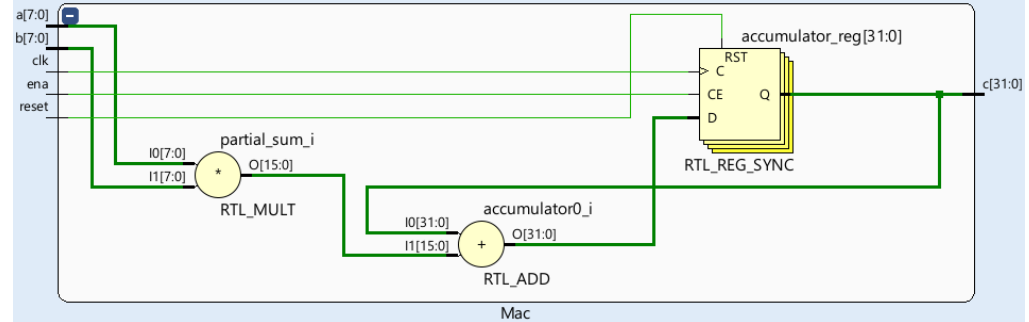
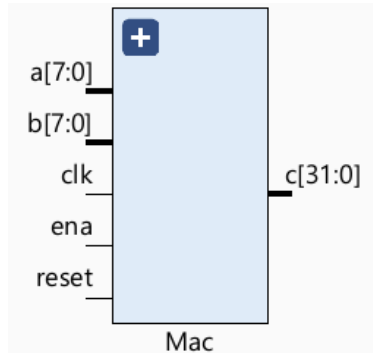
- Use a regular grid of processing elements for multiply-accumulate operations
- Data flows between elements, aligning with matrix multiplication steps
- Their modular, pipelined design maps efficiently onto FPGA resources
- They offer high parallelism and scalability, making them ideal for matrix multiplication on FPGAs



[13]

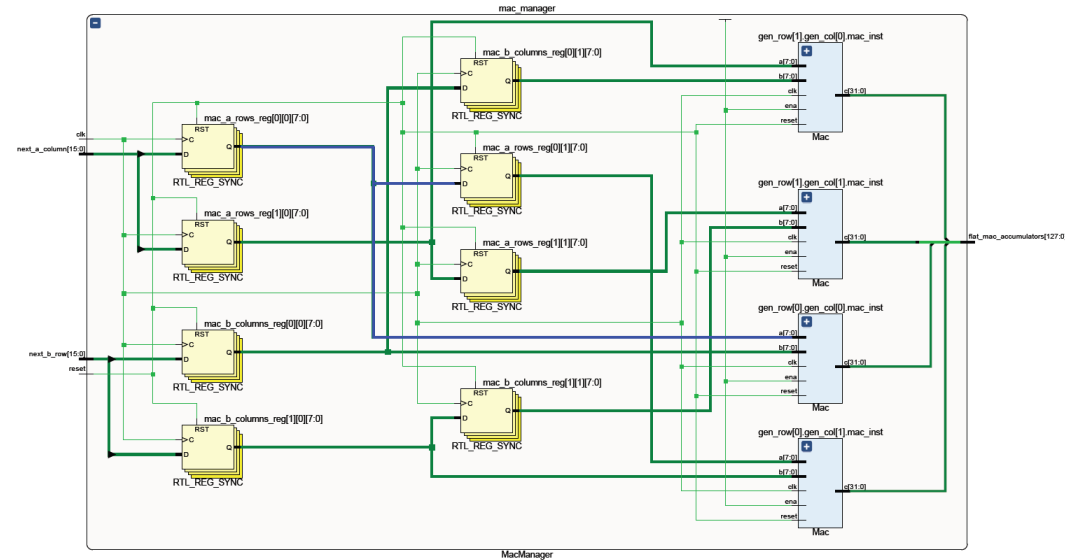
MAC

The MAC (Multiply and Accumulate) module is the most basic building block of the GEMM. It takes 2 inputs, multiplies these 2 inputs, then adds them to the existing accumulation.



MacManager

- The MacManager instantiates and manages the individual Macs that are used.
- At any point in time, it contains the input for each of the $N \times N$ Macs as well as their output, ie the accumulations.
- Its input is the respective new column (!) of a values and row (!) of b values
- Its main purpose is to implement a systolic-array-like flow behavior of these inputs through the different Macs



MacManager for the simple case of 2x2 matrices

SystolicMatrixMultiplier

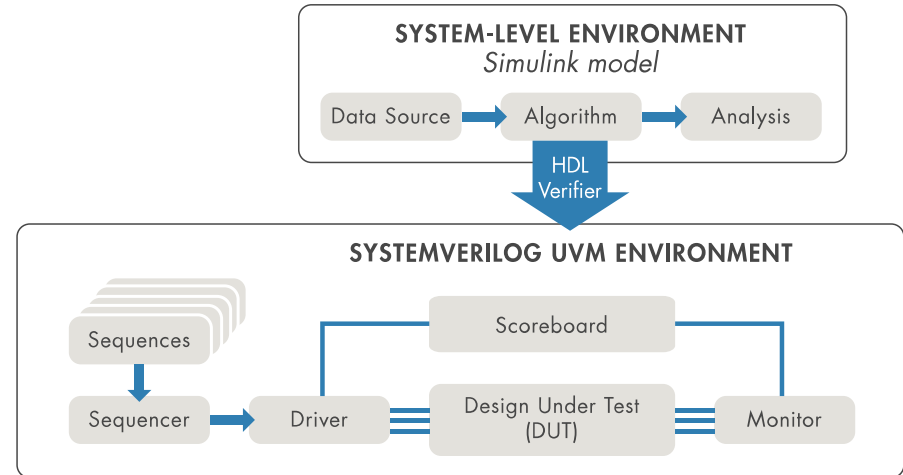
- Top-Level module implementing the matrix multiplication
- Currently receives the input matrices in bulk -> not a big issue because of small OP_WIDTH (8 bit) used
- Instantiates MacManager
- Feeds the input step-by-step to the MacManager
- Resource usage in the case of 16x16 Matrix Multiplication with 8-bit OP_WIDTH:
 - 30298 LUTs (26% of Kria KV260)
 - 12340 FFs (5% of Kria KV260)

```
module SystolicMatrixMultiplier #(
    parameter integer N = 2,
    parameter integer OP_WIDTH = 8,
    // 32 is big enough for most things. Can be finetuned
    parameter integer ACC_WIDTH = 32
) (
    input clk,
    input reset,
    input [N*N*OP_WIDTH-1:0] a,
    input [N*N*OP_WIDTH-1:0] b
);
```

Verifying The New GEMM

High-Level Idea: UVM-inspired Verification Structure

- UVM: Universal Verification Methodology [15]
- Provides a standardized approach for hardware design verification
- Extends SystemVerilog language features
- Components:
 - Factory
 - Sequencer
 - Driver
 - Monitor
 - Scoreboard
 - Agent
- My approach adapts these components as described in the following slides



UVM-Like Verification in Python

- Four components inspired by UVM
 - Input Generator
 - Input Driver
 - Scoreboard
 - Monitor
- For each DUT, a superclass is subclassed, and the respective methods are implemented

```
class TestBase[ParametersT, DutSnapshotT, InputT](ABC):  
  
    [...]  
  
    def start_soon(self):  
        """Invokes cocotb.start_soon on the different components  
        """  
        cocotb.start_soon(self.start_input_generator())  
        cocotb.start_soon(self.start_input_driver())  
        cocotb.start_soon(self.start_scoreboard())  
        cocotb.start_soon(self.start_monitor())
```

Input Generator

- Repeatedly generates input
- Puts this input into a queue, which can be consumed from the input driver

```
# Input Generator function of the SystolicMatrixMultiplier testbench

async def generate_input(self, i: int) → Input:
    # val_max is exclusive
    val_max = 2 ** self.params.OP_WIDTH
    if i % 2 == 0:
        # a and b do not matter here, because reset=True
        return Input(
            reset=True,
            a=np.random.randint(
                0, val_max, (self.params.N, self.params.N)),
            b=np.random.randint(0, val_max, (self.params.N, self.params.N))
        )
    else:
        # a and b randomized
        return Input(
            reset=False,
            a=np.random.randint(
                0, val_max, (self.params.N, self.params.N)),
            b=np.random.randint(0, val_max, (self.params.N, self.params.N))
        )
```

www.mpi-inf.mpg.de

Input Driver

- Consumes an input from the input queue
- Applies this input to the actual DUT, using cocotb's API

```
# Input Driver of the SystolicMatrixMultiplier testbench

async def drive_input(self, next_input: Input):
    self.dut.reset.value = int(next_input.reset)
    self.dut.a.value = pack_matrix_to_int(
        next_input.a, self.params.OP_WIDTH)
    self.dut.b.value = pack_matrix_to_int(
        next_input.b, self.params.OP_WIDTH)

# same waiting rule as in wait_between_transactions
if (next_input.reset):
    await ClockCycles(self.dut.clk, 1)
else:
    await ClockCycles(self.dut.clk, STATES_PER_N * self.params.N)
```


Monitor

- Repeatedly creates snapshots, both before a certain input is being processed and after the input has been sequentially processed by the DUT
- The resulting transaction is stored in a transaction queue
- The Scoreboard then consumes this transaction queue

```

async def start_monitor(self):
    """Starts the monitor

    Awaits a RisingEdge first, then repeatedly:
    - gets a snapshot
    - calls wait_between_snapshots
    - gets another snapshot
    - puts the resulting transaction in the transaction_queue
    """
    await RisingEdge(self.dut.clk)
    while True:
        snapshot_0 = self.get_dut_snapshot()

        await self.wait_between_snapshots(snapshot_0)

        snapshot_1 = self.get_dut_snapshot()

        # put transaction in queue
        t = (snapshot_0, snapshot_1)
        await self.transaction_queue.put(t)

```

github.com

```

# wait_between_snapshots() of the SystolicMatrixMultiplier testbench

async def wait_between_snapshots(self, first_snapshot: DutSnapshot):
    # if the first_snapshot was resetting, then we only have to wait for the next edge
    if first_snapshot.reset:
        await ClockCycles(self.dut.clk, 1)
    # otherwise we have to wait for the operation to finish
    else:
        await ClockCycles(self.dut.clk, STATES_PER_N * self.params.N)

```

Scoreboard

- Repeatedly takes transactions from the transaction queue which was filled by the Monitor
- Scores these transactions, ie compares the behavior of the DUT to the expected behavior

```
# Scoreboard of the SystolicMatrixMultiplier testbench

async def score_transaction(self, t: tuple[DutSnapshot, DutSnapshot]):
    if t[0].reset:
        cocotb.log.info("Asserting reset")
        assert np.all(t[1].c == 0)
    # this case should not happen, indicates that a or b are "x" or "z"
    elif t[0].a is None or t[0].b is None:
        assert False, "a or b are None"
    else:
        expected = t[0].a @ t[0].b
        actual = t[1].c

        assert np.all(expected == actual), (
            f"""
            Calculating: \n
            {t[0].a} \n\n
            @ \n\n
            {t[0].b} \n\n

            Expected: \n
            {expected} \n
            Actual: \n
            {actual}
            """)
```

Verifying via Simulation

- The actual verification happens via a simulator, like icarus Verilog
- The simulation is started through cocotb's API using a custom-built test_runner.py, which allows to easily specify the source path, DUT name and potential parameters

```
$ python test_runner.py ../src SystolicMatrixMultiplier
```

Run without parameters (defaults are used)

```
$ python test_runner.py ../src SystolicMatrixMultiplier N=16 OP_WIDTH=4
```

Run with parameters

```
9800.10ns INFO cocotb.regression test_systolic_matrix_multiplier passed
9800.10ns INFO cocotb.regression
*****
** TEST                                     STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **
*****
** SystolicMatrixMultiplier_tb.test_systolic_matrix_multiplier PASS      9800.10      2.44      4017.88 **
*****
** TESTS=1 PASS=1 FAIL=0 SKIP=0          9800.10      2.51      3911.70 **
*****
```

Cocotb results output

Meaningful Next Steps

Where we are and where we want to be (in my opinion)

The result of my work is grounded in the realization that the existing architecture was not scalable and can probably not be used for any meaningful task. In the following I created a foundational GEMM and a corresponding verification framework.

Both, the GEMM and the verification framework should be future-proof in the sense that they can be built upon and extended by future students in the practical course.

It is my understanding (mainly through the Google TPU paper) that it is not a meaningful goal to pursue backpropagation on the hardware – mainly because specialized hardware (GPUs) will always be miles superior. Instead, it could be a goal to have a design capable of doing fast inference using quantization.

To this end, the following slides explain 3 meaningful next steps to explore.

Next Step 1: Deploying on HW

The existing GEMM is fully synthesizable on the Kria KV260. However, it is isolated in the sense that it can not interact with the Processing System (PS) of the board. I tried some things in this direction, but ultimately ran out of time. This means that it can not be used from outside the Programmable Logic (PL) rendering it unusable

In my opinion, a path forward related to that would be:

1. The important first step is to establish a connection between PS and PL, namely using the AXI protocol (this is also the opinion of Dirk Stober). Doing this correctly would likely include adding some control logic into the GEMM so that the PS can steer its functionality. If done correctly, after this step, the PS can trigger execution of certain matrix multiplications and receive the result.
2. As a next step, it would be nice to make the PL functionality available via a Python interface. To this end, a small Python library could be created which offers a clearly defined API to use the PL. F.e. for the multiplication of NumPy arrays. The PYNQ framework might be useful in this process [16]

Next Step 2: Quantization

The Google TPU mentioned as inspiration above uses integers with a small width during inference instead of expensive floating-point numbers. This means a reduction of complexity and thus faster inference, while reducing accuracy. This reduced accuracy is “usually good enough for inference”, as the Google paper notes.

The current GEMM operates with integers, where the width can be manually set via the `OP_WIDTH` parameter and is set to 8 bit by default. This default is in line with the paper describing the Google TPU.

However, if the GEMM will be used with real arrays after Next Step 1 is completed, Quantization has to be taken care of. I expect this to be not a huge amount of effort, but haven't thoroughly researched the steps that would need to be taken.

Next Step 3: Scaling the GEMM

Next Steps 1 and 2 have as main focus the improvement of the usability of the current GEMM's functionality. This would include adjusting the current Verilog code in the form of the AXI implementation and control logic to execute Matrix Multiplications as desired by the PS. Additionally, Steps 1 and 2 might show other flaws the current GEMM has that need to be taken care of.

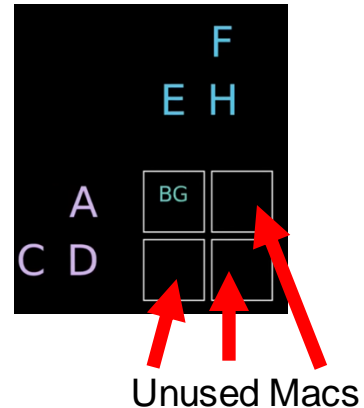
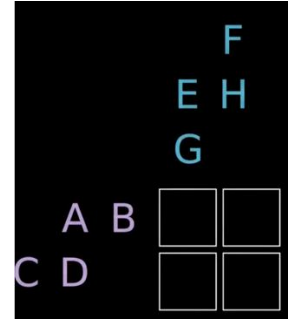
More fundamentally, for the GEMM to be used for inference in a meaningful way – that means to be useful for common tasks, instead of just being a gimmick – the GEMM would need to be able to operate multiple matrix multiplications at the same time. The reason for this lies in the way systolic arrays work with MACs, as a single matrix multiplication operation never occupies all MACs.

Next Step 3: Scaling the GEMM

The two images on the right show the issue of unused MACs when two matrices are multiplied. Not all MACs are being used. This happens at all stages of the matrix multiplication process, not just at the beginning as shown here.

To use all available MACs at any point in time, multiple matrix multiplications need to happen at the same time. Making this work, requires a lot of adjustments to the current design. Namely, there need to be multiple parallel input and outputs, consumed and produced accordingly by the GEMM.

I imagine this to be by far the most complicated next step. I don't have enough experience to judge, but it might be outside what is possible in the scope of this practical course.



Summary of the next steps

To summarize, these are the next steps as seen by me (a non-expert in the field):

1. Deploying on HW
2. Quantization
3. Scaling the GEMM

I believe that the GEMM and the verification framework created by me can serve as a capable basis to build towards these next steps.

After completing steps 1 and 2, the GEMM can be used for actual tasks. After Step 3, its efficiency and throughput would be greatly increased. At this point it also wouldn't just be a GEMM, but our own TPU.

Thomas Wöhrle
thomas.woehrle@tum.de