POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

Hardware-Software Codesign of an Accelerator for Quantized Neural Networks in a Low-Power SoC

Advisors
Prof. Mario Roberto Casu
Dr. Edward Manca
Dr. Luca Urbinati

Candidate
Andrea Redoglia

Academic Year 2023-2024

Summary

The recent advancements in the artificial intelligence field, and the explosion of applications based on Neural Network (NN) deployed in the real world, pose new challenges in optimizing their execution. Edge computing is meant to answer this challenge. Characterized by embedded, low-power System-on-Chips (SoCs) devices, it is a straightforward choice for the NN deployment requirements. However, these devices usually lack specialized hardware to efficiently execute the target workloads. Moreover, the main operations required by NNs are Multiply-And-Accumulate (MAC) operations, with operands at high precision data types and this could lead to a massive number of operations not suitable for SoCs based on CPUs. In this context, two optimizations are possible. On the one hand, NNs can be trained using low precision operands, such as 8-bit integer arithmetic. This technique leads to Quantized NNs (QNNs). On the other hand, SoCs can be integrated with dedicated accelerators to handle the computational patterns of specific NN layers. Starting from QNNs obtained using the first point, this thesis focuses on the second one, and provides a path for the codesign of a custom hardware tightly related with a software implementation that employs it to enable efficient NNs computation. The accelerator presented uses multipliers based on the precision scalable principle. A precision scalable multiplier is a multiplier capable of increasing the number of operations performed in parallel when operands have a reduced precision. One approach to achieve this are the Sum-Together (ST) multipliers, which, if operands are at reduced precision, operate more than one multiplication and sum each independent result before returning them. As an example, a 16-bit multiplier based on this approach can compute one 16-bit, two 8-bit or four 4-bit multiplications in parallel. In the latter two cases the results are summed together, achieving the goal of computing more MACs at the same time at reduced precision. This leads to lower latency when decreasing the precision of the operands. I integrated a ST MAC unit in an accelerator, composed of memory and control logic necessary for the processing. To integrate the final architecture in a SoC for verification purposes I leveraged the Embedded Scalable Platform (ESP), a tool for design automation developed by Columbia University. The accelerator has been described in C++ and synthesized with Mentor Catapult HLS. This final SoC is composed of four tiles: the first is a memory interface that connects the Network-on-Chip (NoC) to

the external DDR; the second is the low-power 32-bit RISC-V Ibex core, developed by ETH Zurich and University of Bologna; the third is my accelerator; and the fourth is an I/O tile. After the definition of the structure, I generated the SoC RTL description with ESP automated flow of integration. Then I developed the baremetal software baseline implementing a memory tiling algorithm to divide the memory to fit the data in the Private Local Memory (PLM) of the accelerator. The hardware, along with the software, has been tested and simulated in the QuestaSim environment. Finally, the custom software has been integrated in TensorFlow Lite for Microcontrollers (TFLM), an open-source ML inference framework.

Acknowledgements

I am really grateful to be surrounded by wonderful people. I cannot even express how lucky I feel when I think at my loved ones. I have to be grateful for the people that I have around, nothing is taken for granted and never will be.
Thanks to my parents to the wholehearted support that have always expressed to me. I hope that I've returned at least a fraction of the love you gave me.
Thanks to my grandparents that have loved me, and showed me that all efforts really pay off.
Thanks my close friends, that have seen me growing and were there since the beginning, really hoping that continues like this for a long time. Your support has been remarkable, hope mine is at least the same.
Thanks to my advisors that have led me through this journey mentoring me with patience and devotion. The help that you provided was indescribable, I'm grateful for that.
Thanks to all that have provided words or made actions to support me, I hope to return them at any chances. Your help has been really appreciated.

"Giovane uomo coltiva il tuo talento, concimalo con il sangue, annaffialo con le lacrime, il sudore e dai tempo al tempo"

Fabio Rizzo

Contents

1 Neural Networks 1
1.1 Fully Connected Layer 2
1.2 Quantization and Data Types 3
1.3 Benchmark 4

2 TensorFlow Lite Micro 7
2.1 Framework Structure 8
2.2 Tensor Memory Storage 10
2.3 Data Types 11
2.4 Fully Connected Computational Kernel 12
2.5 Fully Connected Data Quantization and Biasing 13

3 Accelerator Architecture 17
3.1 Memory Interfaces 18
3.2 Computational Unit 19
3.3 Private Local Memory 21

4 Fully Connected Accelerator Architecture 23
4.1 Memory Interfaces 28
4.2 Computational Unit 30
4.3 Private Local Memory 31

5 Fully Connected Accelerator Implementation 35
5.1 Memory Interfaces 35
5.2 Computational Unit 39
5.3 Private Local Memory 42

6 Embedded Scalable Platform 45
6.1 Hardware Accelerator Design Flow 47
6.2 Fully Connected Accelerator Design Flow 49
6.3 SoC Design Flow 52
6.4 Fully Connected SoC Design Flow 54

7 Accelerated Kernel 59

8 Conclusions 69

List of Figures 73

Bibliography 75

Chapter 1

Neural Networks

It is quite easy to say that Artificial Intelligence and Neural Networks have widely taken most of the research and industry interest, in electronic field at least. The possibilities and scenarios of complex problem solving unlocked by this kind of technology have limits not fully defined nowadays.
However, due to this particular attention caught by the topic, I am not the right person to further explain the root and motives under this. As a matter of fact, much charming and more expert authors have developed marvelous works with a far higher quality explanations compared to the ones I would have been able to provide. Therefore please, refer to [1] for a general introduction of AI world and for extending knowledge on Neural Networks to [2], if there is a deeper interest in the general subject.
In this work introductory chapter, will be presented only the strict necessary background to let a reader, unfamiliar with the topic, understand what the basis and the causes of some choices for the design proposed.
Neural Networks computation relies, for the most, on operations, usually additions and multiplications, which correlate each input with the outputs based on a value that is called weight. The way inputs interact with the weights is the specific layer definition.
In this work, a specific layer is taken under exam to try to exploit the possible optimization and speculations for tailoring a hardware structure that efficiently execute rapidly these operations. The layer chosen is the fully connected, although the architecture proposed fully supports also GEMM operations and can be extended to that usage which is more general, due to the high affinity between one and the other.

1.1 Fully Connected Layer

This kind of layer is the oldest and most widely used one. The layer accepts as input a flattened tensor which is entirely processed so that every element has a weighted connection with every output.

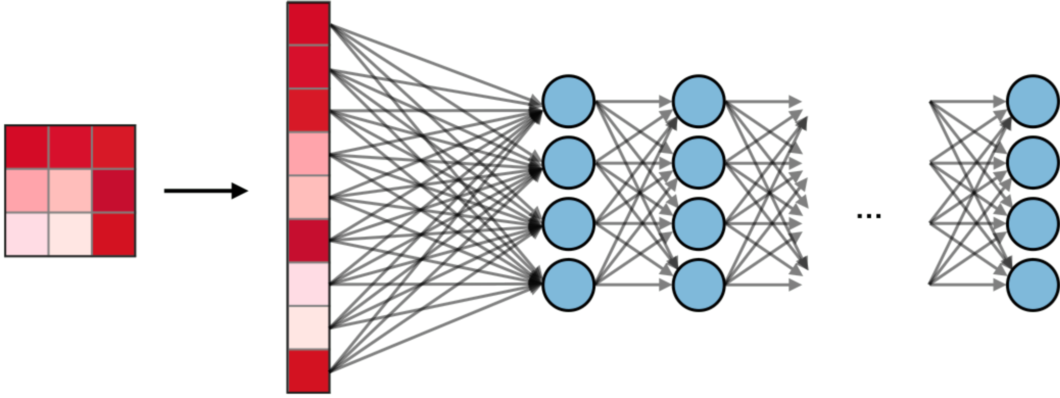

Figure 1.1: Fully Connected Layer [3]

Going deeper to the things useful for the accelerator, the mathematical function that covers this operation is the following:

$$out_j = b_j + \sum_{l=1}^{n_i} w_{j,l}\ i_l$$

where out_j is the jth output, b_j is the jth output bias, $w_{j,l}$ is the weight associated to the jth output and l^{th} input, i_l is the lth input, n_i is the input dimension and for convenience m_o will be output dimension. From this expression, it is possible to extract the number of activations present, which will be the sum of input and output activations:

$$\#activations = \#input + \#output = n_i + m_o$$

On the other hand, it is also possible to evaluate the number of parameters needed to accomplish an execution of this layer as:

$$\#parameters = \#weight + \#bias = n_i \times m_o + m_o$$

Based on the previous considerations, it is possible to evaluate the memory footprint of this specific layer M_{layer}. This kind of analysis must be executed to optimize the inferences of neural network in a resource-limited environment as an embedded system.

$$M_{layer}[B] = \#parameters \times data_size_{param}[B] + \#activations \times data_size_{act}[B]$$

Although when a whole application is under analysis, there is a memory sharing possibility for sequential NNs: multiple layers activations will not be present in memory at the same time, therefore the overall footprint on the memory of the activation is just the larger size of activations among all the layers. On the other hand, weights for all layers must be continuously stored in memory: that implies that the footprint of the weights is the sum of the sizes needed for every layer. To summarize the overall footprint M_{app} will be:

$$M_{app}[B] = max\{\#activations_{l=1}, \cdots, \#activations_{l=l_{max}}\} + \\ + \sum_{l=1}^{l_{max}} \#parameters_l \times data_size_{param|l}[B]$$

The last important parameter of the layer to compute to characterize the resources need is the number of operations. It is necessary that every input has a weighted connection with every output and the latter must be also biased. Therefore, the number of operations needed for a single layer are:

$$\#MUL = \#input \times \#output = n_i \times m_o$$
$$\#ADD = \#input \times \#output = n_i \times m_o$$

1.2 Quantization and Data Types

As showed in the section before, this type of layers can be very costly in terms of resources. Modern computing is shifting toward edge where storage and processing elements are limited and not flexible to implement complex data handling. NNs where originally intended to be structure based on floating-point representation, which is not suitable for low-end CPUs due to the need of dedicated processing elements or latency introduced by mocked floating point instruction.
Training is executed using high precision data aiming at accuracy of the model. Among these types usually appear, with increasing precision:

- **FP16**: floating-point 16-bit data. It has 1 bit for sign, 5 bits for the exponent, which sets the power of two to be multiplied to the rest of number, and 10 bits for the mantissa that is an integer with magnitude lower than the unit.
- **BF16**: it's also a floating-point 16-bit data, developed by Google Brain research team [4], trades off precision of mantissa for a boost of exponent dynamic. The format is composed with the usual 1 bit for sign, 8 bits for exponent and only 7 bits for mantissa. This trade-off enables the higher dynamic of the FP32 with the size of FP16, reaching up to 50% of training speedup in mixed precision training.
- **FP32**: floating-point 32-bit data. It has 1-bit for sign, 8-bit for exponent and 23-bit for the mantissa.

The accuracy of the model is strictly bonded to the precision of the data types used[5].
However, these data types handling for inference involves too much hardware resources and the usage for inference it is prohibitive. Quantization provides a feasible solution to reduce the hardware requirements to run inference on low-power devices: the technique consists into a conversion of floating-point data into integer data providing a method to allow reversing of the operation. The integer data, coming from this process, will be decomposed in a value, the effective quantized data, a zero-point, which represent the symmetrical point of dynamics and a scale, which represents the order of magnitude. The formula for decomposition of a certain data, d, will be:

$$d_{FP} \simeq S_d(d_{quant} + Z_d)$$

Therefore, it is possible to use integer data types for NNs applications. The most widely used are:

- **int4**: integer 4-bit data
- **int8**: integer 8-bit data
- **int16**: integer 16-bit data

1.3 Benchmark

It is necessary to present a valid model benchmark to evaluate the actual design and have a metric on the performances. The chosen benchmark is anomaly detection[6][7], a neural network designed to early detect from sound input an anomaly of different mechanical component, useful for predictive maintenance purposes. The network is composed by only fully connected layers:

- **Input Layer**: 640-unit FC layer, $n_i = 640$ and $m_o = 128$, #parameter = 82048, #activation = 768 .
- **Intermediate Layer 2**: 128-unit FC layer, $n_i = 128$ and $m_o = 128$, #parameter = 16512, #activation = 256.
- **Intermediate Layer 3**: 128-unit FC layer, $n_i = 128$ and $m_o = 128$, #parameter = 16512, #activation = 256.
- **Intermediate Layer 4**: 128-unit FC layer, $n_i = 128$ and $m_o = 128$, #parameter = 16512, #activation = 256.
- **Intermediate Layer 5**: 128-unit FC layer, $n_i = 128$ and $m_o = 128$, #parameter = 16512, #activation = 256.

- **Bottleneck Layer**: 8-unit FC layer, $n_i = 128$ and $m_o = 8$, #parameter = 1032, #activation = 136.
- **Intermediate Layer 7**: 128-unit FC layer, $n_i = 8$ and $m_o = 128$, #parameter = 1152, #activation = 136.
- **Intermediate Layer 8**: 128-unit FC layer, $n_i = 128$ and $m_o = 128$, #parameter = 16512, #activation = 256.
- **Intermediate Layer 9**: 128-unit FC layer, $n_i = 128$ and $m_o = 128$, #parameter = 16512, #activation = 256.
- **Intermediate Layer 10**: 128-unit FC layer, $n_i = 128$ and $m_o = 128$, #parameter = 16512, #activation = 256.
- **Output Layer**: 640-unit FC layer, $n_i = 128$ and $m_o = 640$, #parameter = 82560, #activation = 768.

Therefore from the info available, it is possible to compute the number of parameters of the network, as showed in Chapter 1.1, as:

$$\#parameter_{AD} = \sum_{l=1}^{l_{max}} \#parameters_l = 282376$$

Then, it is necessary to identify the layer that has the maximum number of activations, hypothesizing a homogeneous precision for all the network, as:

$$\#activation_{AD} = max\{\#activations_{l=1}, \cdots, \#activations_{l=l_{max}}\} =$$
$$= \#activations_{l=1} = \#activations_{l=11} = 768$$

Hence, another possible useful metric to extract is the number of operations, as:

$$\#MUL = \sum_{l=1}^{l_{max}} n_{i|l} \times m_{o|l} = 280576$$
$$\#ADD = \#output = \sum_{l=1}^{l_{max}} n_{i|l} \times m_{o|l} = 280576$$

Chapter 2

TensorFlow Lite Micro

With the rapid widespread of edge AI, ML models have come a long way through embedded systems and mobile applications having its maximum expression as TinyML, the intersection of machine learning and embedded world. Although there are many issues related to portability of ML algorithms on resource-limited devices: first of all, it is really challenging to port and deploy model on multiple embedded hardware, because every optimization must be performed on a specific architecture, plus lack of productivity tools that connect training to deployment and finally the incompleteness of the support for compression, quantization, model invocations and execution altogether. On the other hand, many advancements in this field were achieved condensing NN models into a few hundred of kilobytes to accommodate the most memory-limited applications and reducing the impact on battery-based systems. Therefore, the main challenges are oriented on a unified environment which directly implements portability, flexibility and minimize hardware-specific dependencies in embedded systems context. For these multiple reasons, the lack of a dedicated framework has been an overwhelming issue in the integration of small NNs and related applications into a limited-resource device.
TensorFlow Lite Micro [8] satisfies this urge : it is an end-to-end open-source platform that focuses on low resources requirements and minimal runtime performance overhead and leverage various techniques, as quantization and weight pruning, to optimize the memory footprint and latency of the applications.
It is used, as a port of the higher-end TensorFlow, to run a limited sub-set of functions on DSPs, microcontrollers, and other memory limited devices. This application stands as an inference optimization environment for pre-trained models. There is an extensive support for a pool of AI development frameworks, as Keras or PyTorch, that unlocks high-accuracy NNs portability seamlessly. The transition is immediate: convert the desired model into a FlatBuffer [9], a standard format that allows a direct access to serialized data without parsing or unpacking, and the application is ready to use.

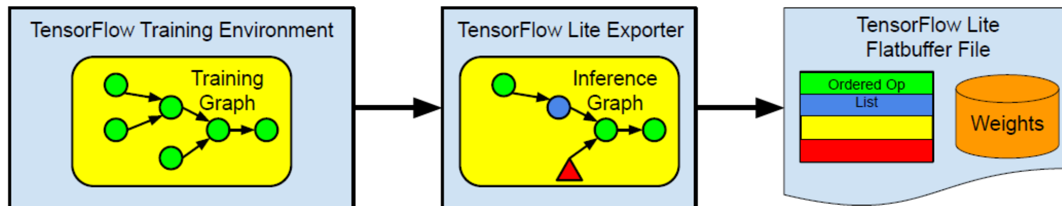

Figure 2.1: TFLM Model Import [8]

2.1 Framework Structure

It may be useful, for explaining following chapters design choices, to spend few words on how TFLM works and how it is implemented.
TFLM implements an interpret-based flexible and general structure for embedded hardware to access a general optimization and control environment for NN inferences [8]. The whole code base is written in C/C++ and is open-source available on the web [10].
TFLM is the results of some software component, which can communicate from different levels of abstraction, through the framework. A brief overview on the few key units is provided below:

- **Interpreter**: handler of runtime model data, the operator to execute and model parameters. The interpreter does not affect the overall inference time itself because the kernel time absorbs its overhead completely.
- **Model Loader**: header only library containing a portable data schema to sort and load the model to the interpreter
- **Memory Planner**: static memory allocator that plans its usage based on lifetime and size of each buffer
- **Operator Resolver**: linker of the operation to be executed to the corresponding kernel/binary. Improves the flexibility of the solution linking the specific kernel based on a directive at compilation time.

The aforementioned components, during software operating condition, interact in the framework environment to execute in five steps a thread-safe fast inference:

1. **Create the neural network**: in this phase it is allocated a memory region for the neural network. The OpResolver manages the linking of the operator to the final binary.
2. **Supply of a contiguous area of memory**: because of the impossibility to assume that the target hardware can allocate dynamic memory, TFLM

relies on a contiguous region of memory used to hold intermediate results and variables. Therefore, it is allocated this memory space called arena.

3. **Create an interpreter instance**: in this phase the model, the opresolver and arena are passed to the interpreter as function arguments. There is an initial phase where all the required memory is allocated by the interpreter so that no heap fragmentation occurs. This security policy grants to safely execute long-running applications. The operations preparation functions are also called in this phase to warn the interpreter about the amount of memory needed by their execution.

4. **Execution**: the pointers to input and output memory region are passed through an invoke to the interpreter that handles the model calculation.

5. **Control returning to application**: after the execution completion, the interpreter gives back the control to the application through a blocking call.

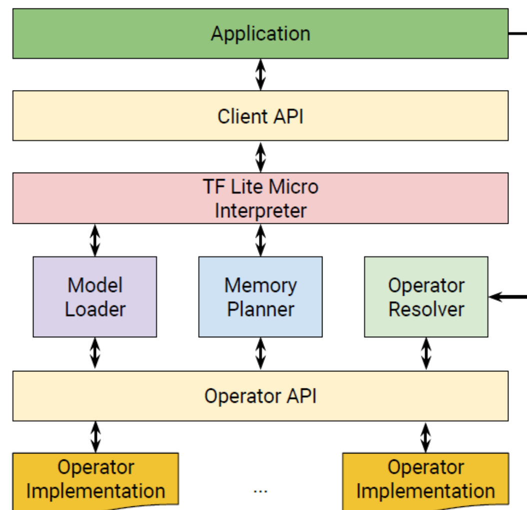

Figure 2.2: TFLM Structure [8]

TFLM developers have not assumed almost nothing on the hardware underlying their application: that makes this framework widely portable and flexible. There is also the possibility to implement multi-tenancy, which is the inference of multiple models sequentially: it is useful to decompose a complex operation into multiple small models that execute micro-tasks. The interpreter is the same and the arena is reused where possible.

2.2 Tensor Memory Storage

It may be also important, in a view of understanding the framework to design custom compatible software and hardware components, to focus on how TFLM manages the memory and how the tensor are stored. The main reason is that, taking into account the way the tensors are stored in memory, helps to access the external memory sequentially: a lot of memories topologies support the burst mode where the latency accessing sequentially is an order of magnitude faster than a random fetch of tensor entries needed. Moreover, in case of sequential access, the custom hardware needs only the pointer for the start of the tensor and the length of the data to process, and that further improve latency and accelerator local memory requirements.
Given the operation needed by a fully connected layer, explained at Chapter 1.1, TFLM implements an efficient way to represent the data that minimize the random access to elements in memory. The representations for the tensor chosen, related to this specific kernel, are:

- **Input Tensor**: it is a flattened tensor, so only one dimension supported, therefore its memory order is contiguous data storing, with a number of elements equal to the given dimension of input, and the name of the tensor is a pointer to the head of the array. If a multi-batch inference is needed, at the end of the array with input dimension elements it's stored the next batch with same elements number accessible with the name of the tensor and a memory offset equal to the input dimension.

- **Output Tensor**: it is a flattened tensor too, so its ordering is managed in the same way as the input except for the element number which is equal to the output dimension.

- **Weight Tensor**: it is a tensor, with shape input dimension output dimensions, stored as contiguous output dimension blocks of input dimension elements accessible through the name of the weight tensor providing a memory offset for each block. The weights remain the same for every batch, therefore there is no more entries needed for multi-batch inference.

This chosen approach optimizes memory accesses for general purpose computation unit, as CPU, and constraints the interface of special purpose hardware that must be able to manage this data ordering, possibly in an efficient way. However, in extreme cases where the hardware supports only sequential accesses of memory, an available, but surely inefficient option, is to reorder the memory before feeding it to a specific purpose unit which is an operation that adds a massive overhead on computation of the desired kernel.

2.3 Data Types

TFLM is designed to be used for embedded applications. It is a subsystem, deriving from TensorFlow, that handles much higher-level computation with complex data types, which supports a subset of types useful for resource-limited device. The demand of resources for certain data treatment, for example floating-point units, or instruction added to compensate the lack of this special hardware, like soft-float compile instructions, is remarkably high and impacting on the final inference energy efficiency or latency. Data types also impact on memory footprint: increasing the number of bits, independently from fixed or floating point, the overall RAM required to store the intermediate activations and weights rise massively. This was one of the hardware dependencies of the solution mentioned in Chapter 2 that TFLM aims to eliminate. On the other hand, cutting out the support for high-precision data types excludes application where accuracy figure of merit is more valuable than inference time. Taking into account the previous considerations, the policy for memory optimization deployed on provided solution, the memory planner component, mitigates the footprint size but the resource usage and latency increase are utterly present. Therefore, to avoid this performance-limiting situations, TFLM introduced a limited data type support, based on the widespread benchmarks adoptions, and use cases.
The data types actually fully supported are [11]:

- **int8**: fixed point signed 8-bit representation with available range or representation is [-128,127].
- **uint8**: fixed point unsigned 8-bit representation with available range of representation is [0,255].
- **float32**: floating point signed 32-bit representation with available finite range from [- $3.4028236 \cdot 10^{38}$,$3.4028235 \cdot 10^{38}$].

TFLM also offers a limited or mocked support for more data types to improve the flexibility of the solution making easily adaptable to many ML develop environment. The data types mentioned are:

- **int16**: fixed point signed representation on 16-bit partially supported to enhance the accuracy, available only for activations in application where quantization highly impacts on performances. In this case mixed operations int16 with int8 are supported.
- **int4**: fixed point signed representation on 4-bit for which TFLM provides a mocked support. The framework extends the operands with this data type to int8, performs kernel operations, and then quantize back to the required precision.

The approach chosen by TFLM enhance the flexibility of the solution implementing some data type for resource saving and others for accuracy metric. It is also implemented the mixed precision case: at developing stage the needed quantization is chosen so that different layer use different data types leveraging bigger and more accurate types for the layers that mostly impact the overall accuracy and more reduced sizes for the others so that a tradeoff between memory usage and accuracy can be implemented.

2.4 Fully Connected Computational Kernel

TFLM provides a stock definition for most of the operations that can be freely used across multiple hardware architectures. Operations on tensors require multiple loops and data structures that ease the sequential access of main memory to reduce latency.
The presented kernel is the fully connected, software implementation of the mathematical function described in Chapter 1.1, which can be found integrally at tflite-micro github [10] and the most important snippet is shown below along with an explanation.
Keeping in mind that the indexing of the tensors processed is the one described in Chapter 2.2, the structure of the code of the kernel is based on three nested loops, presented below ordered from outer to inner:

- **Loop on batches**: implements multi-batch inference, every cycle produces a complete fully connected layer results. The current batch number, *b*, is used to index the current batch of input to use and output to generate.
- **Loop on Output dimension**: implements an output stationary approach where a complete entry of the output tensor is generated at each cycle of the loop. After the looping on input, the result, *acc*, is biased, quantized, with *MultiplyByQuantizedMultiplier* function, and written in the output tensor. The current cycle number, *out_c*, is used to index the weight entry necessary for computation and the output entry that is currently under generation.
- **Loop on Input dimension**: implements the input-weight with the respective offsets product and the accumulation along all the input tensor. The current cycle, *d*, is used to index the input tensor and corresponding weight. It is declared by the framework designers that it is necessary to grant at least an accumulation of 2^{16} multiplications for each output of every kernel using a MAC operation.

Source Code 2.1: Fully Connected Stock Kernel Loops[10]

```
template <typename InputType, typename WeightType, typename OutputType,
          typename BiasType>
void FullyConnectedPerChannel(
    const FullyConnectedParams& params, const int32_t* output_multiplier,
    const int* output_shift, const RuntimeShape& input_shape,
    const InputType* input_data, const RuntimeShape& filter_shape,
    const WeightType* filter_data, const RuntimeShape& bias_shape,
    const BiasType* bias_data, const RuntimeShape& output_shape,
    OutputType* output_data) {
  const int32_t input_offset = params.input_offset;
  const int32_t output_offset = params.output_offset;
  const int32_t output_activation_min = params.quantized_activation_min;
  const int32_t output_activation_max = params.quantized_activation_max;
  TFLITE_DCHECK_GE(filter_shape.DimensionsCount(), 2);
  TFLITE_DCHECK_EQ(output_shape.DimensionsCount(), 2);

  TFLITE_DCHECK_LE(output_activation_min, output_activation_max);
  const int filter_dim_count = filter_shape.DimensionsCount();
  const int batches = output_shape.Dims(0);
  const int output_depth = output_shape.Dims(1);
  TFLITE_DCHECK_LE(output_depth, filter_shape.Dims(filter_dim_count - 2));
  const int accum_depth = filter_shape.Dims(filter_dim_count - 1);
  for (int b = 0; b < batches; ++b) {
    for (int out_c = 0; out_c < output_depth; ++out_c) {
      BiasType acc = 0;
      for (int d = 0; d < accum_depth; ++d) {
        int32_t input_val = input_data[b * accum_depth + d];
        int32_t filter_val = filter_data[out_c * accum_depth + d];
        acc += filter_val * (input_val + input_offset);
      }
      if (bias_data) {
        acc += bias_data[out_c];
      }
      int32_t acc_scaled = MultiplyByQuantizedMultiplier(
          acc, output_multiplier[out_c], output_shift[out_c]);
      acc_scaled += output_offset;
      acc_scaled = std::max(acc_scaled, output_activation_min);
      acc_scaled = std::min(acc_scaled, output_activation_max);
      output_data[out_c + output_depth * b] =
          static_cast<OutputType>(acc_scaled);
    }
  }
}
```

2.5 Fully Connected Data Quantization and Biasing

Data quantization is a widely used strategy to trade-off precision of an inference with memory footprint. Focusing firstly on fully connected layer mathematical expression, as presented in Chapter 1.1:

$$out_j = b_j + \sum_{l=1}^{n_i} w_{j,l}\ i_l$$

where out_j is j-th output, $w_{j,l}$ is the weight corresponding to l-th input and j-th output, i_l is the l-th input, n_i is the input dimension and b_j is the bias of the j-th output.

Quantized data, as explained in Chapter 1.2, add further complexity related to the scale and zero-point arithmetic. That should be integrated in the formula above as follows:

$$S_{out}(out_{quant,j} - Z_{out}) = S_b(b_{quant,j} - Z_b) + \sum_{l=1}^{n_i} S_w(w_{quant,j,l} - Z_w)\ \ S_i(i_{quant,l} - Z_i)$$

where S_{out} is output scale, S_b is bias scale, S_w is weight scale, S_i is input scale, Z_{out} is output zero-point, Z_b is bias zero-point, Z_w is weight zero-point and Z_i is input zero-point.
That can be refactored as:

$$out_{quant,j} = \frac{S_b}{S_{out}}(b_{quant,j} - Z_b) + \frac{S_w S_i}{S_{out}} \sum_{l=1}^{n_i} (w_{quant,j,l} - Z_w) \quad (i_{quant,l} - Z_i)$$

TFLM handles this issue, as presented in their paper [12], making few assumptions:

- $Z_b = 0$
- $S_b = S_w S_i$
- $\mathrm{M} = \frac{S_w S_i}{S_{out}}$

where M is a fused scale factor which is a non-integer multiplier. Therefore based on that, the framework refactors aiming to transform it in integer operators as:

$$M = 2^{-n} M_0$$

Essentially, it has been manually converted from a floating-point representation into an integer one, with n which is the shift number and M_0 which is the integer multiplier.
That allows to a final refactoring, the one implemented in TFLM code, of the fully connected quantized layer:

$$out_{quant,j} = Z_{out} + 2^{-n} M_0 [b_{quant,j} + \sum_{l=1}^{n_i} (w_{quant,j,l} - Z_w) \quad (i_{quant,l} - Z_i)]$$

Finally, to obtain the output quantized it is enough to multiply it with an integer, M_0, and shift the result by a fixed amount, n.
TFLM wraps this output quantization with a function:

Source Code 2.2: TFLM Multiplication Integer Quantization[10]

```
int32_t MultiplyByQuantizedMultiplier(int64_t x, int32_t quantized_multiplier,
                                      int shift) {
  // Inputs:
  // - quantized_multiplier has fixed point at bit 31
  // - shift is -31 to +7 (negative for right shift)
  //
  // Assumptions: The following input ranges are assumed
  // - quantize_scale>=0  (the usual range is (1<<30) to (1>>31)-1)
  // - scaling is chosen so final scaled result fits in int32_t
  // - input x is in the range -(1<<47) <= x < (1<<47)
  TFLITE_DCHECK(quantized_multiplier >= 0);
  TFLITE_DCHECK(shift >= -31 && shift < 8);
  TFLITE_DCHECK(x >= -(static_cast<int64_t>(1) << 47) &&
                x < (static_cast<int64_t>(1) << 47));

  const int32_t reduced_multiplier =
      (quantized_multiplier < 0x7FFF0000)
          ? ((quantized_multiplier + (1 << 15)) >> 16)
          : 0x7FFF;
```

```
  const int64_t total_shift = 15 - shift;
  const int64_t round = static_cast<int64_t>(1) << (total_shift - 1);
  int64_t result = x * static_cast<int64_t>(reduced_multiplier) + round;
  result = result >> total_shift;

  TFLITE_DCHECK(result >= std::numeric_limits<int32_t>::min() &&
                result <= std::numeric_limits<int32_t>::max());
  return static_cast<int32_t>(result);
}
```

This quantization function is used along with a clamping to the maximum or minimum values for each data representation format. Below is showed a snippet of fully connected code where this clamping is performed.

Source Code 2.3: TFLM Clamping Integer Quantization[10]

```
        MultiplyByQuantizedMultiplier(acc, output_multiplier, output_shift);
    acc_scaled += output_offset;
    acc_scaled = std::max(acc_scaled, output_activation_min);
    acc_scaled = std::min(acc_scaled, output_activation_max);
    output_data[out_c + output_depth * b] =
        static_cast<OutputType>(acc_scaled);
```

Chapter 3

Accelerator Architecture

In past years, the industry and research on computer architecture was led by two fundamental principles, based on technology development, to improve figure of merit of the developed hardware. These laws are about scaling of transistors:

- **Moore scaling**: transistors number that fits in a chip can be doubled every two years
- **Dennard scaling**: in new technology nodes, the transistor dimension become smaller compared to the previous of a factor $\frac{1}{k}$ along with the voltage. That implies that the electric field remains the same along power density and area, dissipated power and intrinsic latency of the transistor are reduced.

Using these two laws, the improvements on technology development become predictable and fixed. Unfortunately, Moore law significantly slowed down, due to the extreme small feature size reached for transistors, and Dennard scaling cannot be fully applied anymore. On this premises, the focus of computer architecture shifted toward application specific hardware to further improve performances, area, and power efficiency of the chips. This type of special purpose hardware is called accelerator.
Taxonomy of this structures is based on two main parameters: coupling and granularity [13].

Coupling

Defines how the accelerator interfaces the system and the general-purpose computing unit. On one hand, placing hierarchically close to the CPU increases interface complexity, may include modifications on CPU, to achieve a invocation latency, these are called tightly coupled accelerators. On the other hand, placing hierarchically far from the CPU ease the integration flow at the cost of invocation latency, these are called loosely coupled accelerators.

There are four main category of hierarchical placement of an accelerator in a SoC [13]:

- **Pipeline**: the accelerator is a part of the pipeline of the processor, in execution stage, that can be called for specific operation. An example of that can be an FPU.
- **Cache-level**: the accelerator is attached to the cache memories and can have a tight data coherence with the host processor. A special handle of data coherence is needed in accelerator structure.
- **Memory bus**: the accelerator is attached to the memory bus of the SoC, it is the last level of on-chip acceleration. Usually, this type of accelerator can access freely the main memory without handling coherence or address translation. An example may be an integrated on-chip video-decoder.
- **I/O bus**: the accelerator is attached to the I/O of the SoC, so the acceleration takes place off-chip. An example can be an external GPU.

Granularity

Defines the type of operations offloaded to the accelerator. Fine-grain operations allows more flexibility for software usage at price of energy efficiency and more hardware calls. Besides, coarse-grain operations are much more constrained in the usage but can be implemented more efficiently and the execution can be sped up much more. The granularity can be categorized in three main groups [13]:

- **Instruction-level**: the task offloaded is a primitive task as arithmetic function.
- **Kernel-level**: the task offloaded is a group of instruction or a key part of the running application for example a matrix multiplication.
- **Application-level**: the task offloaded is a full specific application as an NN inference.

3.1 Memory Interfaces

A RAM fetch on a modern processor requires hundreds of cycles, making external memory access a very energy-hungry and slow operation that may limit the performance of a well-designed hardware. On-chip caches partially hides this latency making available frequently used data faster but may be an issue when the amount of this data is higher than their capacity, as may happen for ML applications.
In particular for ML processing, an ulterior approach to memory interfaces must

be chosen at design time.
There are few possibilities, based on mathematical expression of the layer under analysis that in this case is the one described in Chapter 1.1, to access memory:

- **Input Stationary**: accelerator handles first all the computation that involves a particular chunk of input, loading all the weight necessary and storing partial outputs, and then moves on to the next chunk. That allows to efficiently save memory accesses when the inputs must be reused multiple times in the computation.
- **Weight Stationary**: accelerator handles first all the computation that involves a particular chunk of weights, loading all the input necessary and storing partial outputs, and then moves on to the next chunk. That allows to efficiently save memory accesses when the weights must be reused multiple times in the computation.
- **Output Stationary**: accelerator handles first the computation related to the generation of a chunk of outputs, loading all the input and weights necessary, storing at the end the final chunk of output before moving on the next. That allows to efficiently save memory accesses when the computation for generating a single output chunk is massive.

These approaches leverage different reuse of the local memory based on a specific application for both cache and external memory accesses. The effective advantage of the different approach is layer based.
On top of the previous considerations, the accelerator interface must take into account an efficient method to handle the application data ordering, as an example the one mentioned in Chapter 2.2. Granting sequential access of the external memory can boost the performances of the read/write interface because most of memory topology supports burst sequential mode that is order of magnitude faster than random mode access. Moreover, the energy required for sequential access is slightly lower than random access mode. In the designing process of a specific application compliant hardware, take it into account can sensibly maximize the overall performances and energy efficiency.

3.2 Computational Unit

Hardware accelerators speedup relies not only on the offloading of the CPU by a specific task, which is in its way a parallelization technique, but have the major advantage of implementing a tailored solution for a particular application leveraging various sources to enhance the computation performances, save energy or reduce the overall area. Modern techniques can be applied for improve different figure of merit, between power performance or area, based on the specifications of the target design. Below the most widely used are presented:

- **Topology**: considering that the task to be executed is usually very specific, one of the most used sources of improvements is the topology of the processing element used inside the accelerator. The specific operation can be decomposed, refactored or simplified, based on speculations or data correlations, to decrease sensibly the latency respect to a general purpose ALU. A CPU processing element is constrained to be as general to execute multiple task using the same hardware, as single multiplications or adds or shifts, while a knowledge on the specific task can be leveraged to fuse operations together, as an example a MAC unit, or accelerate most recurrent cases at most while slowing down others much more infrequent, as example the cache data replacement policy, or save area on unnecessary hardware. Choosing the right topology for the application can bring to massive and almost free improvement on PPA.

- **Parallelization or Unrolling**: specific hardware can implement multiple instances of the same processing element to execute more operations related to the same task in the same time slice. This principle can be exploited to bring an higher throughput if are true two main conditions: the DFG of the task doesn't show dependencies for the operations to be parallelized, otherwise the result will be incorrect, and the maximum effective utilization, it's necessary to have enough operations to run in parallel to fill every processing or the most otherwise static energy and area will be wasted for useless elements that are not bringing any speedup. Another possibility is to use this technique for low-power applications: the key idea is to replicate the architecture by a factor and reduce the operating frequency or voltage based on the same factor. This approach leverages multiple datapaths for a relaxation on the critical path maintaining constant or almost constant the throughput. In conclusion, tailoring the right parallelization is another essential design specification for application specific hardware that can lead to huge benefits in terms of PPA.

- **Resource sharing or Rolling**: this technique, counterpart of the one presented before, aims to reduce the overall circuit area sharing the available resources, processing elements, based on the DFG of the application to be executed. The operations are scheduled based on the type of functional unit, rolling factor and dependency to achieve a more serialized circuit. That leads to a save of instances of functional unit, an increase of number of multiplexers for data routing and a slight advantage on power saving at the cost of more clock cycles to have the final result.

- **Pipelining**: on one hand, in a design performance-driven, this technique implies the insertion of registers to break the critical path in more section for which the latency is less. That allows to increase the operating frequency and achieve much more throughput than the original, unpipelined, datapath but

implies an increase of needed area. On the other hand, in a design power-driven, this technique implies breaking the critical path in more section with latency equal as before but supply voltage much less than before. That saves both dynamic and static power that are proportional to the supply voltage. Although, this technique cannot be replicate indefinitely because in both design cases will lead to a saturation of performances or power savings.

- **Arithmetic Intensity vs Memory Bandwidth**: the computational unit must adhere to the memory interface specifications. That implies not only to be able to fetch data from the memory with a shared protocol but also, implies that if the data requirements of the processing elements is much more than the amount of data that can be fetched, some processing elements or the overall computational unit can stay idle for the major part of cycles wasting area and energy. This behavior can be mathematically modeled as follows:

$$Th = min\{BW_{max} \cdot I_{app}, Th_{max}\}$$

where BW_{max} is the maximum bandwidth of the memory interface, I_{app} is the arithmetic intensity of the application to be executed and Th_{max} is the maximum throughput achievable by the computational unit. Leveraging the knowledge of the application at design time, it is possible to fully maximize the usage of both memory interface and processing elements staying close to the corner of the characteristic. This methodology can be used to maximize the throughput, reduce the unnecessary area or find a trade-off. This approach is helpful only if the reading from the memory is concurrent to the execution of the computations, otherwise it will be impossible to hide processing elements idle time.

3.3 Private Local Memory

Accelerator may, if designed in the right way, exploit full parallel computation to execute an application kernel. Unfortunately, the structure commonly used in computer architecture cannot satisfy the high-throughput of this kind of processing elements: caches cannot implement efficiently a high number of ports, fixed blocks cannot represent well the variable data width needing of an accelerator and high associativity can be counter-productive in terms of energy saving.
As stated from previous research work on accelerator private local memories [14], first of all PLMs are the key to performances and energy efficiency, therefore tailoring a memory subsystem for an accelerator special needs seems essential. This small storage must provide the exact number of ports, banks and width needed to fulfill the requirements of the computational unit so that a high throughput can be achieved. Secondly PLMs can occupy mostly of the silicon area reserved to an accelerator and that implies that the energy and area budget highly depends on a

well-designed memory subsystem. Another related design consideration about the size of memory is the effective utilization during an application execution: a poor design of the PLMs subsystem can lead to a fractional utilization of the resources available that, without strategy of energy saving like power gating of unused memory banks, implies a waste of static power.
These are the main considerations that must be taken into account to reach a target PPA figure of merit for an accelerator.

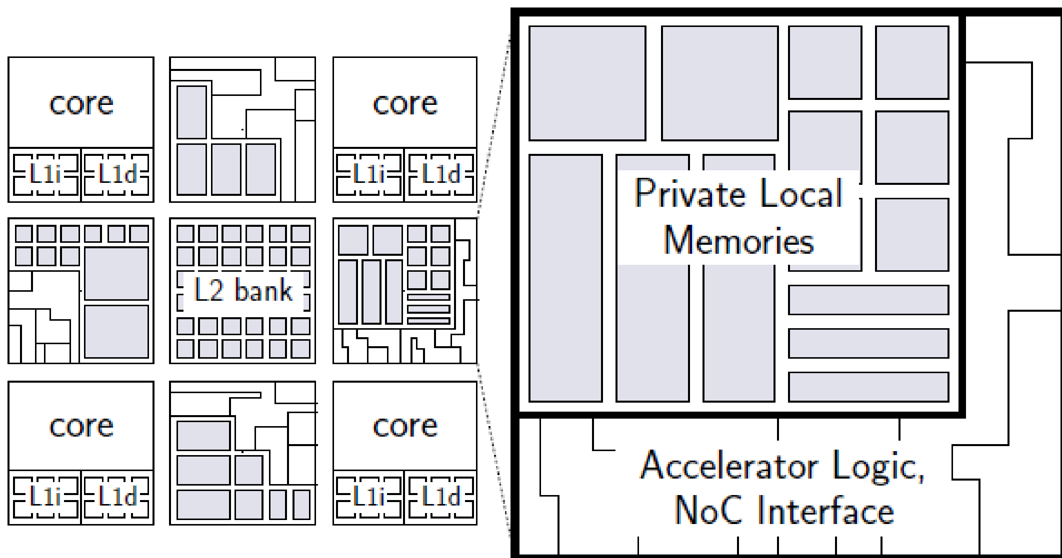

Figure 3.1: Accelerator PLM [14]

Chapter 4

Fully Connected Accelerator Architecture

The target for this work is to accelerate a fully connected layer inside the TFLM environment, targeted for a low-power resource limited deployment, and this chapter will present the architectural design choices that led the implementation.
The idea is to offload the multiplications between the input and weights to an external hardware, with more computational efforts compared a general purpose computational unit, and reduce the overall latency for the applications. Referring to the formula of the layer also presented, and utterly explained, in Chapter 2.5:

$$out_{quant,j} = Z_{out} + 2^{-n} M_0 [b_{quant,j} + \sum_{l=1}^{n_i} (w_{quant,j,l} - Z_w) \quad (i_{quant,l} - Z_i)]$$

where $out_{quant,j}$ is the jth output quantized, Z_{out} is the output offset, n is the scale shift-amount, M_0 is the scale mantissa, n_i is the input dimension, $w_{quant,j,l}$ is the quantized weight associated to the lth input and j^{th} output, Z_w is the weight offset, $i_{quant,l}$ is the quantized l^{th} input and Z_i is the input offset.

The target part of the application to be accelerated is the term:

$$\sum_{l=1}^{n_i} w_{quant,j,l} \quad i_{quant,l}$$

That is the most critical and computational intense part of the whole application. It was excluded by the acceleration scope the biasing and the quantization process because it was left to future work to decide whether quantization method to accelerate between the one proposed by TFLM and the one with which are trained the models, Keras one.

Based on this specific applications design considerations about granularity and coupling needed for this task can be done.

Coupling

The choice of the accelerator coupling comes from the trade-off between the number of hardware calls, that are costly for loosely-coupled taxonomies, the amount of data processed in parallel, that it is limited in tightly-coupled architectures, and the type of granularity of the acceleration, the bigger it is the loosely it must be coupled to implement efficiently the application [13]. On top of that, the design effort for interfacing of tightly coupled accelerator is much bigger.
For this work the **memory bus coupling** was chosen ,although below are listed all the taxonomies presented in Chapter 3 with the advantages and disadvantages, to better understand the choice, based on this specific application.

- **Pipeline**: due to the massive amount of data required to accelerate efficiently the application, this taxonomy was excluded. It would have been a massive trade-off between area and performance: to achieve a parallel acceleration, probably, it would have been necessary to adapt the processor load and store unit to accommodate a multi-data fetch from memory maybe with a modified set of instructions compared to the one already implemented. Another way, to still exploit parallel computation, could be to still use the stock CPU load unit to fetch the required data, store them in a PLM and execute the processing in a single clock cycle : unless the processor has a multi-issue stage, the offloading of the CPU is null and the application may stall till the fully completion of accelerator operation. In this work the target acceleration was aiming a kernel-level so this solution may not be suitable. Anyway, it could be feasible, as showed in STAR MAC Unit paper [15], where the acceleration focuses only on the reduced data precision, the overhead for data fetching introduced by the general purpose processor is still present and instruction set was still modified. Anyway, the area occupied and the speedup still broadly validates the alternative.

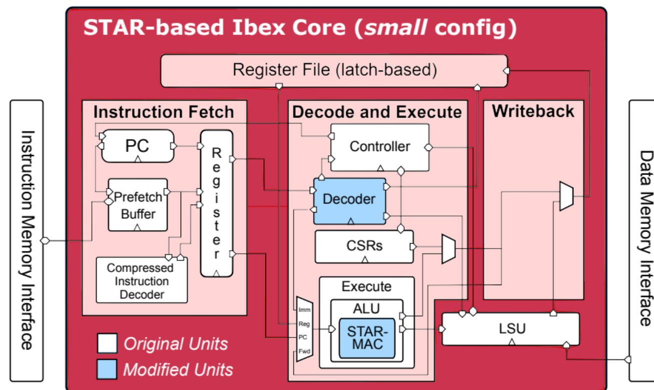

Figure 4.1: Pipeline-Level Accelerator example : STAR-based Ibex core [15]

- **Cache-level**: this taxonomy was excluded, even though a much more parallel computation can be achieved due to the size needed of a cache to efficiently handle all the data required. If the cache is sized to store the whole tensor may be too big and the read latency may critical increase, otherwise, if it does not, the number of misses may vanish all the acceleration provided by the hardware [14]. Moreover, it is required additional logic to handle coherency and data pollution. It was chosen to implement efficient PLM to store the data required so that is avoided data flush and prefetch logic inside the accelerator. Although it is left the possibility to adapt it to a socket that implements all the logic required to enable the feature and scale the coupling to a cache-level based.

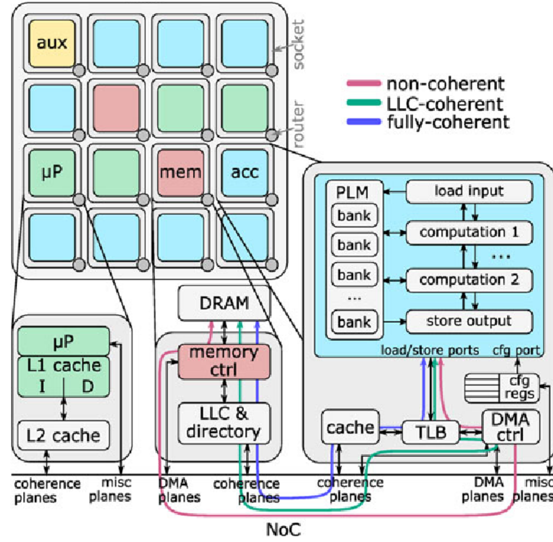

Figure 4.2: Cache-level coherency example structure[16]

- **Memory bus**: this is the approach chosen for the coupling of the accelerator. The mathematical expression proposed shows that data dependencies are very low, therefore the computation can be highly parallelized to the point that the whole kernel can be run simultaneously. Of course, this kind of brute force approach doesn't repay for the price of energy and area invested, considering a low-power application as this is. But these considerations may be used to hide the costly hardware calls, deriving from this choice of coupling, in long multi-data handling executions. Moreover, the integration effort in a SoC is straightforward, the memory interface will be quite simple, and all the memory required for processing can tailored and directly implemented inside of the accelerator to improve energy efficiency and processing unit utilization. This choice obviously implies more area invested in performances.

- **I/O bus**: this topology was excluded too for the additional complexity required by interfacing. The application complexity does not justify the area and power that would have been devoted to implement an I/O subsystem and configuration to build a standalone SoC. On top of that, the design effort would have been greater than an on-chip solution. The choice of this topology would have been reasonable for the complete execution of the network, therefore with a high workload and coarse granularity, where the communication with the host processor would have been limited to few configuration parameters. In this case, it would may be even better to implement more layers options, activations functions and tiling algorithm directly in hardware building a full TPU, as for example the one presented in Google paper [17].

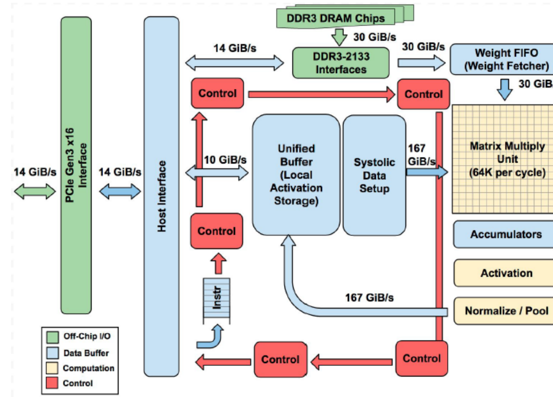

Figure 4.3: I/O-Level Accelerator example : Google TPU [17]

Granularity

The granularity of the accelerator is a trade-off between the complexity and flexibility of the hardware that will be implemented and the overall PPA of the system, based on specific operation. Moreover, the design effort increases as the coarsing of granularity.
On this premises, it is chosen the **kernel-level granularity** and presented below the advantages and disadvantages of the different taxonomies based on the specific in scope application:

- **Instruction-level**: this approach cannot efficiently represent the application under exam because the instruction to be executed are both multiplications and additions. Implementing only one of the two instructions would not be efficient and the CPU must handle the remaining operation. Therefore this granularity was discarded.

- **Kernel-level**: this is the chosen granularity because it fully represents the requirements on type of operations. A fuse operation of multiplication and

add, a MAC, is the optimal solution to handle neural network processing based on the equation presented before in the chapter. That allows a scalable and flexible design to meet area and power consumption constraints of a low-power application.

- **Application-level**: would imply a complication on hardware architecture, a lot of area and power dissipation to utterly represent the hardware for the neural network. It is not feasible, for power consumption and area, to add extra logic to handle activation functions and tensor tiling, in a low-power view that would have been an overkill. However, this remains an interesting option left for future work development.

In the following sections will be presented the architectural choices and the detailed description of the components of the accelerator developed. Below it is available the block diagram as reference.

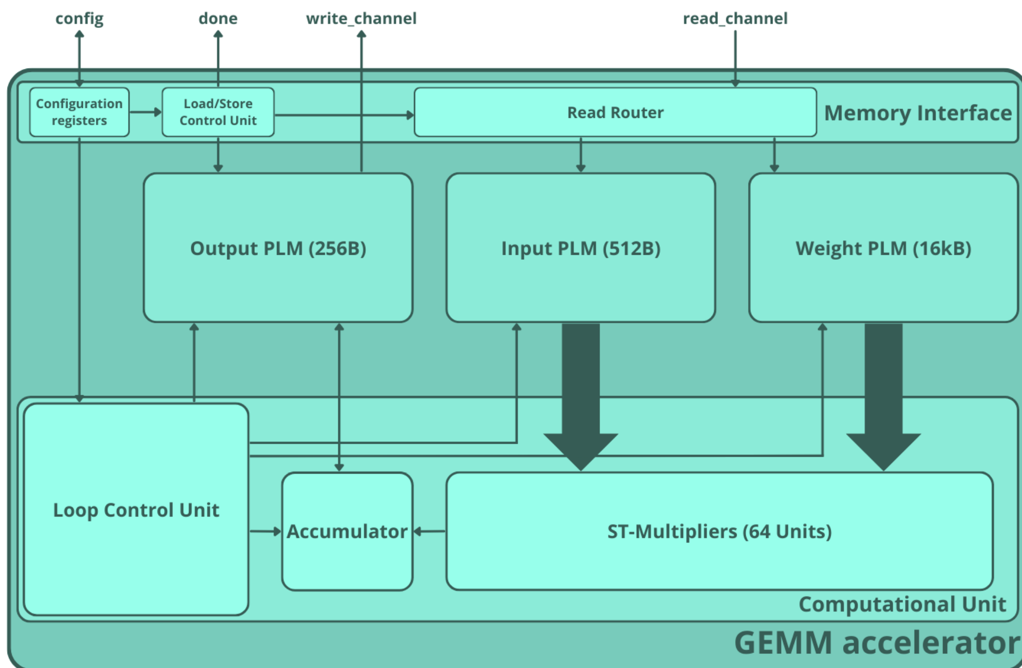

Figure 4.4: GEMM Accelerator Block Diagram

4.1 Memory Interfaces

Memory interfaces have a heavy impact on how high the outcome performances of the accelerator deployed in the specific environment executing its specific application. As aforementioned in Chapter 3.1, the trade-offs to take into account are multiple and special care should be applied to exactly match the requirements of the framework and environment requirements.

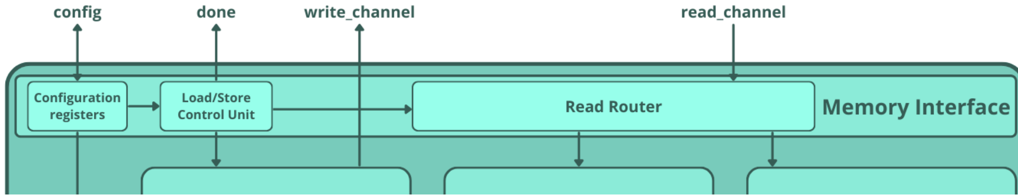

Figure 4.5: GEMM Accelerator Memory Interface Block Diagram

As a matter of fact, the coupling of the accelerator, memory bus coupled, imposes that it exposes as memory mapped some configurations registers to set the operating mode. These registers are listed below with a brief explanation of this specific application use.

- *#input*: sets the number of inputs to use
- *#output*: sets the number of output to generate
- *data_precision*: sets the data precision for the required operations
- *input_address*: hold the address of the input tensor
- *weight_address*: holds the address of the weight tensor
- *output_address*: holds the address of the first output element to write
- *flags*: holds the required flags to further customize the accelerator run.

These registers can be written at run-time by the CPU to customize the operation mode and adapt the accelerator to every need. Another parameter to consider is the parallelism of memory interface. In this specific application, it will be used a low-power processor with 32-bit parallelism, which imposes the bit-width of the chosen interface to leverage all the available NoC of the SoC in which the accelerator is deployed to exchange data.
Another key parameter to take into account in this design is the tensor ordering of the framework chosen. Referring to the one that TFLM proposes showed in Chapter 2.2, the memory interface should implement a logic that execute two different loading loops, to collect both the input activations and the weights, and one for writing the output in the order presented below:

- **Input Activations Loading Loop**: the input tensor has only one dimension, a flatten operation always occurs before the fully connected kernel if necessary, therefore it can be implemented as a sequential load of the chosen input based on an address written in the exposed configuration register of the accelerator. It is necessary to compute the number of 32-bit words to read based on the configuration registers of input number and data precision. The read size can be computed as follows:

$$\#words_{input|read} = \#input \cdot (\frac{data_precision}{32})$$

where $\#words_{read}$ is the number of 32-bit words to read, $\#input$ is the input activations number and $data_precision$ is the precision of data to be computed.

- **Weights Loading Loop**: weights tensor reading, based on the ordering, must be treated in two different nested loops: the outer one reads the block of weights associated to a specific output and the inner one reads the weight entry for the specific input. There is necessary, as before, a normalization to compute the effective number of 32-bit words to read, that can be computed as follows:

$$\#words_{weight|read} = \#words_{input|read} \cdot \#output$$

- **Output Storing Loop** : output tensor has only a single dimension therefore its storing is straightforward to offload the internal PLM for the output to the external memory in a sequential order. Referring to the minimum accumulation depth of TFLM exposed in 2.4, the chosen parallelism of output data must be at least:

$$\#bit_{output} = 2 \cdot data_precision_{bit|max} + 16 = 48$$

It has been chosen to extend compatibility for much more intensive operation to set it to 64-bit, that means a maximum accumulation of 2^{32} multiplications. Therefore the number of 32-bit words to write back will be as follows:

$$\#words_{output|write} = 2 \cdot \#output$$

This reading strategies proposed maximize the sequential reading from external memory avoiding software data reordering and granting the fastest data loading possible for this application. The loading phase will not be concurrent to the computational stage, to avoid speculation on the timing of the socket interfaced to the accelerator, decoupling it from memory-bandwidth constraints.
This chosen approach for loading the tensors needed for the computations oriented the approach to an **output stationary** because the reuse of the output values, for accumulation, is massive in this kind of applications. The software developed to use the accelerator will have to implement the specific approach.

4.2 Computational Unit

Referring to the general hardware architecture section of computational unit at Chapter 3.2, some of the techniques presented were actually leveraged to improve overall performance of the accelerator. The design choices that led to the final architecture are presented below:

- **Topology**: this is a key source of improvement for performances. It was implemented a topology, published previously in [18], that leverage the reducing data precision to parallelize operations. This structure is called ST-multiplier and is a reconfigurable MAC unit that performs the operations at the requested precision and accumulate the results outputting a 32-bit result. The 16-bit multiplier used is based on this approach and can compute one 16-bit, two 8-bit or four 4-bit multiplications in parallel. In the latter two cases the results are summed together, achieving the goal of computing more MACs at the same time at reduced precision. This reconfigurability is handled with a 3-bit signal that is coded in the following way:

CONFIG	**ST Output**
16x16 (000b)	$P_{[31:0]} = A_{[15:0]} \times B_{[15:0]}$
16x8 (100b)	$P_{[31:0]} = A_{[15:0]} \times B_{[7:0]}$
8x8 (010b)	$P_{[31:0]} = A_{[15:8]} \times B_{[7:0]} + A_{[7:0]} \times B_{[15:8]}$
8x4 (011b)	$P_{[31:0]} = A_{[15:8]} \times B_{[3:0]} + A_{[7:0]} \times B_{[11:8]}$
4x4 (001b)	$P_{[31:0]} = A_{[15:12]} \times B_{[3:0]} + A_{[11:8]} \times B_{[7:4]} + A_{[7:4]} \times B_{[11:8]} + A_{[3:0]} \times B_{[15:12]}$

Table 4.1: ST-Multiplier Configuration

  Regarding the asymmetric configurations, it's unnecessary to use them for existing benchmark and in the application running TFLM will not be called. Although, it has been chosen to leave the support for future usages. To summarize, this processing element increases the parallelization available with the decreasing of the required precision, speeding up low bit-width use cases.

- **Parallelization or Unrolling**: it was chosen, based on the design explorations info provided by [19], to implement a factor 64 parallelization of the ST-multipliers with an adder plane to further accumulate the results of the processing elements. This approach leverages the possibility of instancing multiple processing elements to achieve a lower latency in exchange of higher area

occupation and energy consumption.

- **Arithmetic Intensity vs Memory Bandwidth**: these parameters are taken into account with a multi-port interface with the PLMs. These memories, along with external memory interface loading-storing logic, allows to decouple the execution stage from data fetching from external memory. That improve the flexibility of the deployment of the accelerator in a SoC: there are no memory bandwidth requirements because there is no concurrency between execution and loading-storing stages.

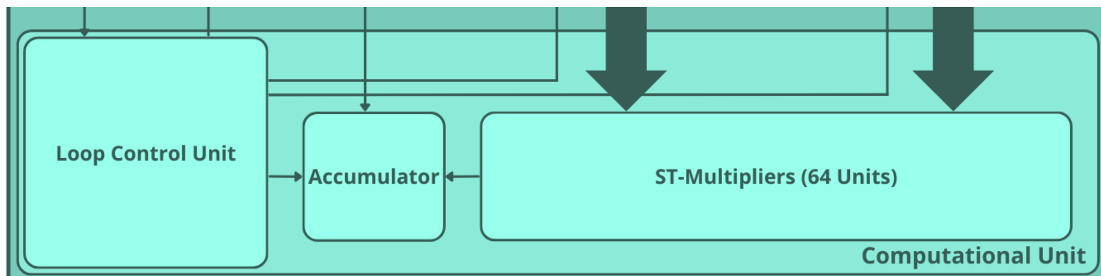

Figure 4.6: GEMM Accelerator Computational Unit Block Diagram

4.3 Private Local Memory

As mentioned before in Chapter 3.3, the tailoring of an efficient memory subsystem is essential for reaching area-energy budget. This storage must take into account the parallelism of data, in this specific application is 16-bit therefore each memory line will have this width. It has been used to better design a memory subsystem the design space exploration presented in Luca Urbinati PhD thesis, that provides insights of PPA based on accelerator similar to the one designed in this work [19]. Three different PLM are implemented that, in this specific application, have the following usages:

- **Input PLM**: stores the input activations. It has been chosen to support a maximum number of activations equal to 256, therefore the actual size is **512B**.
- **Output PLM**: stores the output activations. It has been chosen to support the generation of 32 output. Therefore the size will be **256B**.
- **Weight PLM**: stores the weights. To allow the generation of 32 output from maximum 256 input activations simultaneously, the memory size must be 256x32x16. The actual size will be **16kB**.

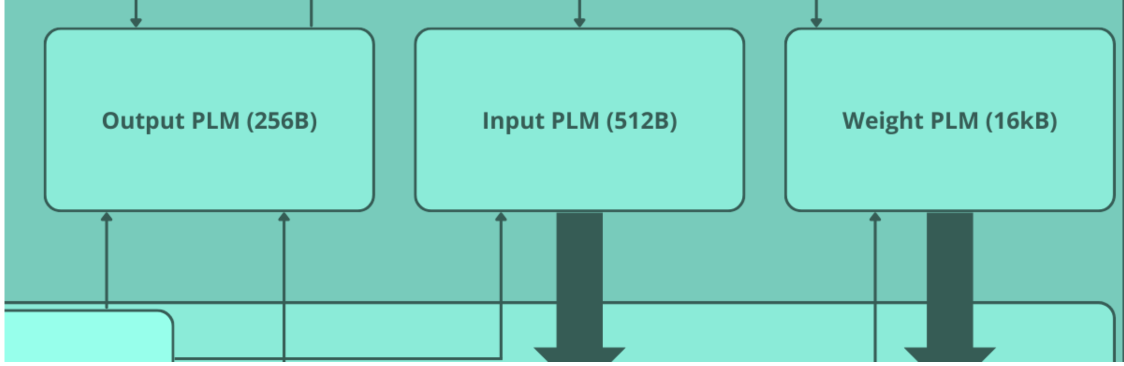

Figure 4.7: GEMM Accelerator PLM Block Diagram

After defining the size of memory needed, it is necessary to define memories ports. The data flow direction is unidirectional therefore the ports toward the memory interface will be read-only or write-only based on the data to be stored and toward computational unit the other way around. The ports are listed below.

- **Ports toward memory interface**: To correctly interface the external memory, it's necessary to tailor the number of ports based on the different parallelism. The PLMs ports are listed below.
  - **Input Write Ports**: this ports write the input activations into the corresponding PLM. The maximum parallelism supported by the computational unit is 16-bit. Therefore, to better leverage the 32-bit parallelism chosen for the memory interface, it has been chosen to instance a dual-port 16-bit write operation which can be simultaneously activated with one writing on even addresses and one on odds, simplifying a full dual port implementation.
  - **Weight Write Ports**: this ports write the weight into the corresponding PLM. Likewise the input PLM, it has been chosen to implement a dual-port 16-bit write operation.
  - **Output Read Port**: it's necessary to write back the output to the external memory, so this ports read the PLM and report the values to the memory interface. The parallelism of accelerator output data is 64-bit due to accumulation support, therefore the write back operation of a single 64-bit data must be broken into two 32-bit reads interleaved to correctly adapt to the memory interface chosen.
- **Ports toward computational unit**: similarly as before, the interface to the computational units must be sized to correctly feed the processing elements based on the data required by computation and the number of output to sample at each cycle. The actual PLMs ports implemented are listed below.

- **Input Read Ports**: the input activations must be directly routed to the ST-multipliers therefore it is necessary to implement 64 ports 16-bit wide to correctly feed all the processing elements. In this way the bandwidth of the PLM is tailored exactly to support the computational unit execution, therefore no bottlenecks are introduced.
- **Weight Read Ports** : the weights must follow the same path of the input activations therefore it is likewise the input PLM which have 64 ports 16-bit wide.
- **Output Read/Write Port** : the accelerator core read the value of the PLM and accumulates one output at each computational cycle, therefore it is necessary just a single 64-bit wide read/write port.

Chapter 5

Fully Connected Accelerator Implementation

This chapter describe the actual implementation of the hardware accelerator. As deducible from the accelerator architecture where timings and synchronization signals were left to be defined, the language chosen to describe the structure is Catapult HLS a version of C++. This approach has numerous benefits for exploring the right description for the chosen implementations, tuning different parameters to achieve better PPA, change parallelization factors or PLM sizes. On top of that, the design efforts to signoff the RTL is massively lower than the classical approach. Apart from this considerations, the actual implementation was the one described in Chapter 4. The overall description method offloads timing of structures to the synthesis tool, that automatically allocates resources and schedules operations, based on feasible transformations, to preserve the the C++ behavior optimizing the outcoming RTL. On one hand, the code in C++ describes the algorithms useful to complete the task and the parallelism of data to be processed. On the other hand, directives included in a TCL file describe the way Catapult must interpret the design, orchestrate the synthesis of the RTL, maps variables to known characterized library components, and set the design toward a specific goal. With this dual approach, it is possible to rapidly describe a specific architecture, leveraging the complex optimizations unlocked by EDAs, and decouple the algorithmic from strictly architectural description. It must be specified that there is a strict rule set for writing a synthesizable C++ and the linting of code has an impact on the hardware generation quality directly.

5.1 Memory Interfaces

External memory interfaces are implemented, following the architectural description of Chapter 4.1, to interface a memory bus that has a dedicated controls and

data ports for the accelerator. These interfaces can be seen by C++ code as the main function, core of the accelerator, parameters to be passed at runtime. The equivalent description in HLS is shown below.

Source Code 5.1: Accelerator Memory Interface

```
#ifdef __CUSTOM_SIM__
void fc_cxx_catapult(
#else
void CCS_BLOCK(fc_cxx_catapult)(
#endif
    ac_channel<conf_info_t> &conf_info,
    ac_channel<dma_info_t> &dma_read_ctrl,
    ac_channel<dma_info_t> &dma_write_ctrl,
    ac_channel<dma_data_t> &dma_read_chnl,
    ac_channel<dma_data_t> &dma_write_chnl,
    ac_sync &acc_done) {
```

These parameters are freely used as read-write variable in the HLS. The effective specialization of the parameters to be mapped as I/O comes from the directive setting included in the TCL file. These ports are mapped as a component, taken from a library, that implements the specific functionality intended. Below are presented the directive related to the ports.

Source Code 5.2: Accelerator Memory Interface Directives

```
    directive set /$ACCELERATOR/conf_info:rsc -MAP_TO_MODULE ccs_ioport.ccs_in_wait
    directive set /$ACCELERATOR/dma_read_ctrl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
    directive set /$ACCELERATOR/dma_write_ctrl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
    directive set /$ACCELERATOR/dma_read_chnl:rsc -MAP_TO_MODULE ccs_ioport.ccs_in_wait
    directive set /$ACCELERATOR/dma_write_chnl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
    directive set /$ACCELERATOR/acc_done:rsc -MAP_TO_MODULE ccs_ioport.ccs_sync_out_vld
```

The ccs_ioport.ccs_in_wait module implements synchronization between the input data at the interface using a latency-insensitive protocol, ccs_ioport.ccs_out_wait module that grants output data synchronization and ccs_ioport.ccs_sync_out_vld is a stand-alone handshake signal needed for explicit synchronization. The name of ports has been chosen with prefix *dma* because it is logical to integrate the accelerator to be CPU-independent for data fetching.
As deducible to the C++ code above, the description also implements an interface for the configuration registers. The code of configuration is shown in the snippet below.

Source Code 5.3: Accelerator Memory Mapped Custom Registers

```
struct conf_info_t {
    uint32_t acc;
    uint32_t options;
    uint32_t offset_PE;
    uint32_t offset_q_data;
    uint32_t N;
    uint32_t M;
    uint32_t in_add;
    uint32_t w_add;
    uint32_t out_add;
    uint32_t flags;
};
```

The usage of this registers is further specified along the code where this variables are used for specific tasks of customization setting. It is a gateway for the CPU to directly program the accelerator. The functionalities of the registers are described in the table below.

Register	Functionality
options	The first byte sets the precision of operation. The rest of the register is reserved for further expansions
offset_PE	Stores the memory offset to access scattered memory with the DMA of accelerator, left for future usage and not implemented in this work.
offset_q_data	Stores the address for quantization and biasing data. It is reserved to future support the functionality that is not implemented in this work yet
N	Sets the number of input
M	Sets the number of output
in_add	Stores the start address of the input tensor
w_add	Store the start address of the weight tensor
out_add	Stores the start address of the
flags	Additional flags for customizing accelerator operation on quantization and activation function. Kept to future supporting of the functionality.

Table 5.1: Accelerator Configuration Registers

As well as configuration, the memory interfaces must take also care of the loading of input values into the corresponding PLM. First of all, it is necessary to compute the effective number of 32-bit words necessary for collecting all the inputs. Than the interface stalls till the read channel is free and ready to accept a request. Then it sets a fetch request starting from *in_add* with the length computed before, and polls to probe the input values feeding the input PLM breaking a 32-bit word in two 16-bit half-words as necessary for their line width. The code that describes this behaviour is presented in the snippet below.

Source Code 5.4: Accelerator Input Loading Phase

```
do { dma_read_ctrl_done = dma_read_ctrl.nb_write(dma_read_info); } while (!dma_read_ctrl_done);
// Force serialization between DMA control and DATA data transfer
if (dma_read_ctrl_done) {
    #ifndef __SYNTHESIS__
        ESP_REPORT_INFO(VON, "Entering LOOP_LOAD_INPUTS ...");
    #endif
    LOOP_LOAD_INPUTS:
            for (uint16_t ni = 0; ni <  MAX_INPUT_ACTIVATIONS; ni += 2){
        RDIN rdata;
                #ifndef __SYNTHESIS__
                        while (!dma_read_chnl.available(1)) {}; // Hardware stalls until data ready for
                             ↪ CSIM
                #endif
        rdata = dma_read_chnl.read().template slc<DMA_WIDTH>(0);
                    plm_in.data[ni] = rdata.template slc<DATA_WIDTH>(0);
                    plm_in.data[ni+1] = rdata.template slc<DATA_WIDTH>(16);

        //Debug prints
        #ifndef __SYNTHESIS__
            ESP_REPORT_INFO(VON, "--- rdata[%u] = %d", ni/2, rdata.to_int());
            ESP_REPORT_INFO(VON, "--- plm_in[%u] = %d", ni, plm_in.data[ni].to_int());
            ESP_REPORT_INFO(VON, "--- plm_in[%u + 1] = %d", ni, plm_in.data[ni+1].to_int());
        #endif
```

```
                if ((ni >> 1) == dma_read_in_data_length - 1) break;
            }
        #ifndef __SYNTHESIS__
            ESP_REPORT_INFO(VON, "Exiting LOOP_LOAD_INPUTS ...");
        #endif
        }
```

After that the interface must implement the loading of the weights in the corresponding PLM. The procedure is the same as before, but there is an additional loop that is used to load all the blocks of weights, with dimension equal to input number normalized to 32-bit, for generating the requested number of output. The code that describes this behaviour is presented in the snippet below.

Source Code 5.5: Accelerator Weight Loading Phase

```
LOOP_LOAD_WEIGHTS:
for (uint16_t mi = 0; mi < MAX_OUTPUT_NEURONS; mi++){
    uint32_t offset_weight_M = w_add + mi*dma_read_in_data_length;
    dma_read_info = {offset_weight_M, dma_read_in_data_length, DMA_SIZE};
            bool dma_read_ctrl_done2 = false;
    LOAD_CTRL_LOOP2:
    do { dma_read_ctrl_done2 = dma_read_ctrl.nb_write(dma_read_info); } while (!dma_read_ctrl_done2);

    if (dma_read_ctrl_done2) {
        weight_load_for:
        for (uint16_t ni = 0; ni <  MAX_INPUT_ACTIVATIONS; ni += 2){
            RDIN rdata;
            uint16_t index = 2 * mi * dma_read_in_data_length + ni; //2 for 32b to 16b memory parallelism
            #ifndef __SYNTHESIS__
                while (!dma_read_chnl.available(1)) {}; // Hardware stalls until data ready for CSIM
            #endif
            rdata = dma_read_chnl.read().template slc<DMA_WIDTH>(0);
                        plm_f.data[index] = rdata.template slc<DATA_WIDTH>(0);
                        plm_f.data[index + 1] = rdata.template slc<DATA_WIDTH>(16);

            //Debug prints
            #ifndef __SYNTHESIS__
                ESP_REPORT_INFO(VON, "--- rdata[%u] = %d", ((ni/2) + mi * dma_read_in_data_length),
                    ↪ rdata.to_int());
                ESP_REPORT_INFO(VON, "--- plm_f[%u] = %d", index, plm_f.data[index].to_int());
                ESP_REPORT_INFO(VON, "--- plm_f[%u + 1] = %d", index, plm_f.data[index+1].to_int());
            #endif

                        if ((ni >> 1) == dma_read_in_data_length - 1) break;
                }
    }
    if (mi == M - 1) break;
}
```

The last task demanded to the memory interface is to write back the output to external memory. This component stalls till the core execution loads the results into the output PLM. At this point the interface stalls till the write channel is free and ready to sample the output and then the outputs PLM are offloaded into the communication channel splitting the 64-bit line width into two separate 32-bit data. The code that describes this behaviour is presented in the snippet below.

Source Code 5.6: Accelerator Output Storing Phase

```
    for (uint16_t mi = 0; mi < MAX_OUTPUT_NEURONS; mi++){
        DOUT data = plm_out.data[mi];
        assert(DMA_WIDTH == 32 && "DMA_WIDTH set to 32 (Ibex compatible build)");
        ac_int<DMA_WIDTH, false> data_ac0;
        ac_int<DMA_WIDTH, false> data_ac1;
        data_ac0.set_slc(0, data.template slc<DMA_WIDTH>(0));
        dma_write_chnl.write(data_ac0);
        data_ac1.set_slc(0, data.template slc<DMA_WIDTH>(32));
        dma_write_chnl.write(data_ac1);
        #ifndef __SYNTHESIS__
            ESP_REPORT_INFO(VON, "--- plm_out[%u] = %d", ESP_TO_UINT32(mi), data.to_int());
```

```
        #endif
        if (mi == dma_write_data_length - 1) break;
    }
    #ifndef __SYNTHESIS__
```

5.2 Computational Unit

The computational unit is the key component of the accelerator, where the data is actually processed. The way the code is written highly impacts the overall design implementation: all the variable declared are treated as data to be represented in RTL, all the loops will be transformed into FSMs and all arithmetic operations will be represented as components. However, the synthesis tool automatically detects unused structures, variables or loops and remove them based on dependency graphs of operations, which helps improving the design efficiency. Anyway, special care should be applied when writing the C++ code to obtain a clean and optimized design.
The description of the sources of improvement of the designed architecture are listed below.

- **Topology**: the topology chosen is a precision scalable multiplier named ST-multiplier. The architecture, presented in Chapter 4.2, can be described at high level with branches to choose the operation, selected by a configuration signal, and the bit-wise description of every operation. In this case it is presented also mixed precision operations, not used in this work, but implemented to allows a fast future integration of the feature.

Source Code 5.7: Accelerator Multiplier Topology Description

```
void multiplier_hls_noioreg(uint3 CTRL,
                            int16 A,
                            int16 B,
                            int32 &P)
{

    int32 output;

    if (CTRL == 4) { // 16x8
        output = A * B.slc<8>(0);
    } else if (CTRL == 1) { // 4x4
        output = A.slc<4>(12) * B.slc<4>(0) + A.slc<4>(8) * B.slc<4>(4) + A.slc<4>(4) * B.slc<4>(8) +
            ↪ A.slc<4>(0) * B.slc<4>(12);
    } else if (CTRL == 2) { // 8x8
        output = A.slc<8>(8) * B.slc<8>(0) + A.slc<8>(0) * B.slc<8>(8);
    } else if (CTRL == 3) { // 8x4
        output = A.slc<8>(8) * B.slc<4>(0) + A.slc<8>(0) * B.slc<4>(8);
    } else { // 16x16
        output = A * B;
    }

    P = output;

}
```

- **Fully Connected function**: the core function to represent, as mentioned in

Chapter 4, is the following:

$$out_{acc|quant|j} = \sum_{l=1}^{n_i} w_{quant,j,l} \quad i_{quant,l}$$

where $out_{acc|quant|j}$ is the jth quantized output of the accelerator; n_i is the input dimension; $w_{quant,j,l}$ is the quantized weight relative to the jth output and l^{th} input; $i_{quant,l}$ is the lth quantized input.
The summation can be translated into an accumulation loop: the multiplication is performed and accumulated to a partial result for the utter input dimension, which generates a single output of the accelerator. This will appear as the inner loop with label c_for.
The generation of all the outputs can also be translated into a loop, the outer one k_for, that iterate c_for for the utter output dimension defined. The whole function representing the core, GEMM_core_reconfbitwidth, is reported below.

Source Code 5.8: Accelerator Core Description

```
void GEMM_core_reconfbitwidth (
                                        plm_inputs_t     &IN,
                                        uint10           INPUT_LENGTH,
                                        plm_filters_t    &FIL,
                                        uint6            OUTPUT_LENGTH,
                                        plm_outputs_t    &OUT,
                                        uint1            RST_OUT_ACC,
                                        uint1            EN_QUANTIZATION,
                                        uint3            CONFIG1)
{


    int16 a1, a2, b1, b2;
    int idx_in;
    int idx_fil;

    ac_int<OUT_BITWIDTH_UNQUANT, true> output_acc[MAX_OUTPUT_NEURONS];

    init_for:
    for (uint6 k = 0; k < MAX_OUTPUT_NEURONS; k++) {
        #ifndef __SYNTHESIS__
            ESP_REPORT_INFO(VOFF, "acc_flag value is: %u)", RST_OUT_ACC.to_uint() ); //Debug print for
                ↪ reset of acc value
        #endif
        if (RST_OUT_ACC == 0) {
            output_acc[k] = 0;
        }
        if (k == OUTPUT_LENGTH - 1) {
            break;
        }
    }


    k_for:
    for (uint6 k = 0; k < MAX_OUTPUT_NEURONS; k++) {

        c_for:
        for (uint10 c = 0; c < MAX_INPUT_ACTIVATIONS; c++) {  //2 times because inside is used at 16b
             ↪ parallelism
            int32 product1;
            int64_t sum_of_products1;

            idx_fil = (k * 2 * INPUT_LENGTH + c).to_int(); //2 times because inside is used at 16b
                 ↪ parallelism
            idx_in = (c).to_int();

            a1 = (int16) FIL.data[idx_fil].slc<16>(0);
            b1 = (int16) IN.data[idx_in].slc<16>(0);
```

```
#ifndef __SYNTHESIS__
//Debug print for correctness data check
    ESP_REPORT_INFO(VON, "--- Previous output_acc[%u] = %d", k , output_acc[k].to_int());
    ESP_REPORT_INFO(VON, "--- IN = %d", b1.to_int());
    ESP_REPORT_INFO(VON, "--- W = %d", a1.to_int());

    if (CONFIG1 == 2) {//8b
        ESP_REPORT_INFO(VON, "---   IN(7 downto 0) = %d", b1.slc<8>(0).to_int());
        ESP_REPORT_INFO(VON, "---   W(15 downto 8) = %d", a1.slc<8>(8).to_int());
        ESP_REPORT_INFO(VON, "---   IN(15 downto 8) = %d", b1.slc<8>(8).to_int());
        ESP_REPORT_INFO(VON, "---   W(7 downto 0) = %d", a1.slc<8>(0).to_int());

    }
    else if (CONFIG1 == 1) {//4b
        ESP_REPORT_INFO(VON, "---   IN(3 downto 0) = %d", b1.slc<4>(0).to_int());
        ESP_REPORT_INFO(VON, "---   W(15 downto 12) = %d", a1.slc<4>(12).to_int());
        ESP_REPORT_INFO(VON, "---   IN(7 downto 4) = %d", b1.slc<4>(4).to_int());
        ESP_REPORT_INFO(VON, "---   W(11 downto 8) = %d", a1.slc<4>(8).to_int());
        ESP_REPORT_INFO(VON, "---   IN(11 downto 8) = %d", b1.slc<4>(8).to_int());
        ESP_REPORT_INFO(VON, "---   W(7 downto 4) = %d", a1.slc<4>(4).to_int());
        ESP_REPORT_INFO(VON, "---   IN(12 downto 15) = %d", b1.slc<4>(12).to_int());
        ESP_REPORT_INFO(VON, "---   W(3 downto 0) = %d", a1.slc<4>(0).to_int());
    }
    else {}
#endif

MUL1:
multiplier_hls_noioreg(CONFIG1, a1, b1, product1);

if (RST_OUT_ACC != 0) {
    sum_of_products1 = product1 + OUT.data[k];
} else{
    sum_of_products1 = product1;
}
//ACC:
output_acc[k] += sum_of_products1;
#ifndef __SYNTHESIS__
//Debug print for output update
    ESP_REPORT_INFO(VON, "--- Updated output_acc[%u] = %d", k , output_acc[k].to_int());
#endif


if (c == 2 * INPUT_LENGTH - 1) { //2 times because inside is used at 16b parallelism
    break;
}

} // c
if (k == OUTPUT_LENGTH - 1) {
    break;
}
} // k

//
    ↪ --------------------------------------------Quantization----------------------------------------------
if (EN_QUANTIZATION == 0) {

    wb_for:
    for (uint6 k = 0; k < MAX_OUTPUT_NEURONS; k++) {
        OUT.data[k] = output_acc[k];
        if (k == OUTPUT_LENGTH - 1) { break;
        }
    } // k

} else { // if (EN_QUANTIZATION == 1) kept for future quantization support

} // if quantization

}
```

- **Resource Sharing**: as aforementioned in Chapter 3.2, it's a technique that involves the usage of the same component for two operations or data storage if the data flow scheduling find it possible. Catapult automatically analyze the DFG and allocates the resource based on the timeline usage, allocates on the same components all the operation possible to save area for the overall design when possible.

- **Memory clock gating**: Clock gating is a low-power technique that stops the clock switching for unused components based on scheduling of resources. This design method decreases dynamic power consumption acting on switching activity of unnecessary signals. Catapult HLS automatically schedules operations and resources to decide the effective components to use in a specific time slice: if a synchronous component is found to be idle in a specific time slice then the tool adds a signal to inhibit it to switch it's signals for the rising edge of the clock with and *AND* gate. This impacts slightly negatively on the overall area but saves a lot of dynamic power.

5.3 Private Local Memory

The private memory subsystem is one of the key element to achieve the desired performances of the accelerator.
Referring to the architecture described in Chapter 4.3, the effective implementation can be describe istantiating an array of specified data type. This parameter of each line, based on architectural specifications, is a signed integer with 16-bit parallelism.

Source Code 5.9: Accelerator PLM Data Type

```
// Accelerator type constants
#define READ_IN_BITWIDTH            32
#define IN_BITWIDTH                 16
#define OUT_BITWIDTH_UNQUANT        64      // Output must take into account the accumlation over the
    ↪ channels

// Data types
typedef ac_int <READ_IN_BITWIDTH, true> RDIN;
typedef ac_int <IN_BITWIDTH, true> DIN;
typedef ac_int <OUT_BITWIDTH_UNQUANT, true> DOUT;
```

Then it's necessary to define the number of lines of the PLMs. That can be done defining a special data type that has line width equal to the data type defined before, DIN, and depth parameter to the desired one : $MAX_INPUT_ACTIVATIONS = 256$ for the input PLM; $MAX_OUTPUT_NEURONS = 32$ for the output PLM; $MAX_WEIGHTS = 8192$ for the weight PLM.

Source Code 5.10: Accelerator PLM sizes

```
// PLM Costraints
#define MAX_INPUT_ACTIVATIONS     256 //256 changed to mantain same max_register value
#define MAX_OUTPUT_NEURONS        32
#define MAX_WEIGHTS       MAX_INPUT_ACTIVATIONS * MAX_OUTPUT_NEURONS

// Private Local Memory
// Encapsulate the PLM array in a templated struct
template <class T, unsigned S>
struct plm_t {
public:
   T data[S];
};


// PLM typedefs
typedef plm_t<DIN, MAX_INPUT_ACTIVATIONS> plm_inputs_t; //16b memory access parallelism type for input
typedef plm_t<DIN, MAX_WEIGHTS> plm_filters_t; //16b memory access parallelism type for filter
typedef plm_t<DOUT, MAX_OUTPUT_NEURONS> plm_outputs_t;
```

The description in HLS defines the routing of the memory ports. The architecture of the accelerator contemplate the following port, where the routing is shown in code below.

- **Ports toward memory interface** : To correctly interface the external memory, it's necessary to tailor the number of ports based on the different parallelism.
  - **Input Write Ports** : a single 32-bit data from the memory interface must be split into two 16-bit data for two different location in parallel for the PLM.

Source Code 5.11: Accelerator Input PLM Ports toward Memory Interface

```
RDIN rdata;
        #ifndef __SYNTHESIS__
                while (!dma_read_chnl.available(1)) {}; // Hardware stalls until data
                     ↪ ready for CSIM
        #endif
rdata = dma_read_chnl.read().template slc<DMA_WIDTH>(0);
            plm_in.data[ni] = rdata.template slc<DATA_WIDTH>(0);
            plm_in.data[ni+1] = rdata.template slc<DATA_WIDTH>(16);
```

  - **Weight Write Ports** : a single 32-bit data from the memory interface must be split into two 16-bit data for two different location in parallel for the PLM.

Source Code 5.12: Accelerator Weight PLM Ports toward Memory Interface

```
RDIN rdata;
uint16_t index = 2 * mi * dma_read_in_data_length + ni; //2 for 32b to 16b memory
     ↪ parallelism
#ifndef __SYNTHESIS__
    while (!dma_read_chnl.available(1)) {}; // Hardware stalls until data ready
         ↪ for CSIM
#endif
rdata = dma_read_chnl.read().template slc<DMA_WIDTH>(0);
            plm_f.data[index] = rdata.template slc<DATA_WIDTH>(0);
            plm_f.data[index + 1] = rdata.template slc<DATA_WIDTH>(16);
```

  - **Output Read Port** : single 32-bit data port with interleaved reading to pass 64-bit data from PLM

Source Code 5.13: Accelerator Output PLM Port toward Memory Interface

```
DOUT data = plm_out.data[mi];
assert(DMA_WIDTH == 32 && "DMA_WIDTH set to 32 (Ibex compatible build)");
ac_int<DMA_WIDTH, false> data_ac0;
ac_int<DMA_WIDTH, false> data_ac1;
data_ac0.set_slc(0, data.template slc<DMA_WIDTH>(0));
dma_write_chnl.write(data_ac0);
data_ac1.set_slc(0, data.template slc<DMA_WIDTH>(32));
```

- **Ports toward computational unit** : similarly as before, the interface to the computational units must be sized to correctly feed the processing elements based on the data required by computation and the number of output to sample at each cycle. The snippets of the code has been included in the core description in Chapter 5.2.

- **Input Read Ports** : 64 parallel read of 16-bit data must occur to feed processing elements. The parallelization is not explicit in this description and demanded to directives.
- **Weight Read Ports** : 64 parallel read of 16-bit data must occur to feed processing elements. The parallelization is not explicit in this description and demanded to directives.
- **Output Read/Write Port** : the accelerator must read a selected location of PLM to execute result accumulation, then the result must be written back from a single 64-bit port.

Then the next step is to define the directive to map directly the variables associated to the memories to dual-port, one for reading and one for writing, RAM components. That can be done setting the directives -*MAP_TO_MODULE* addressing the desired type of memory in library. The last step for defining the memory topologies is to add the -*INTERLEAVED 64* directive to array partition the inputs and weights PLM that unlocks the possibility to truly unroll the computational core algorithm. The command steers Catapult HLS to implement 64 separated memories, that overall have the same capacity as the PLMs defined, to define multi-ports for a selected memory, when effectively used in parallel, otherwise it will be a reunify logic for the selected memory address space to interface it as a single port.

Source Code 5.14: Accelerator PLM directives

```
directive set /$ACCELERATOR/core/plm_in.data:rsc -MAP_TO_MODULE Xilinx_RAMS.BLOCK_1R1W_RBW
directive set /$ACCELERATOR/core/plm_out.data:rsc -MAP_TO_MODULE Xilinx_RAMS.BLOCK_1R1W_RBW
directive set /$ACCELERATOR/core/plm_f.data:rsc -MAP_TO_MODULE Xilinx_RAMS.BLOCK_1R1W_RBW
directive set /$ACCELERATOR/core/plm_in.data:rsc -INTERLEAVE 64
directive set /$ACCELERATOR/core/plm_f.data:rsc -INTERLEAVE 64
```

The description in C++ defines, along with additional directives on algorithm execution, effectively if the interleaving is necessary or not, so that the compiler can speculate on rejoin unnecessary multi-ports or keep this separation.

Chapter 6

Embedded Scalable Platform

The new demanding tasks related to the intensive computing led the industry and the research to shift toward integrating more than one core inside the same chip instead of designing a single bigger and more parallelized core. That is due to the slowing down of the scaling of transistors, which led the increasing of performances for a long golden age. Thus, as the field switched from uniprocessor to multiprocessor, now the focus is further reduction of energy per operation with application specific hardware [20].On this premises, Columbia University invested resources to release an open-source platform that enables a fast and automated heterogeneous SoC integration: ESP [21].

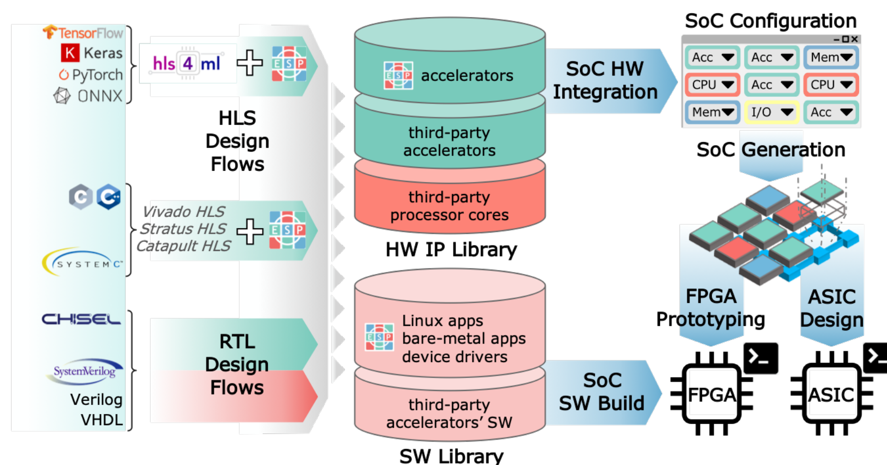

Figure 6.1: ESP Design Flow [21]

The methodology of design proposed by the Department of Computer Science

of Columbia University is a system-level approach that enables an agile flow for hardware description and the related software development. Its main focus is to ease heterogeneous integration. As a matter of fact, ESP SoC design approach proposes a layout divided in a matrix where it is possible to defines the entries, tiles, configure different clock and power domains, data coherence levels and DVFS controller. The type of tiles supported by the platform are:

- **Memory Tile**: is an interface toward an off-chip memory. Its main functionality is to translate the requests of data from all the elements attached to the NoC to the main memory through an APB-DDR converter. In multi-instance, can unlock high memory bandwidth to feed the most data-hungry designs. It can fit with a coherence model, can include an LLC caching for better performances and can handle DMA or IRQ requests from other tiles or peripherals from dedicated NoC line. Each of this kind of tiles has its own part of the global address space defined at design time, completely software transparent, and a data dispatch logic that is embedded on the SoC structure definition time.

- **Shared-Local Memory Tile**: provides a size limited on-chip memory that can be addressed inside the global address space by all the tiles in the SoC through a NoC transaction. It allows to map small applications all on the on-chip memory saving time and energy that would have been spent in accesses on the off-hip DRAM. That can be useful to store activations of a NNs in between the inner layers improving inference speed and efficiency.

- **I/O Tile**: collects all the auxiliary peripheral that the SoC may need to communicate with the off-chip environment and enable requests coming from the NoC to be forwarded to off-chip through different protocols. It supports three main ports per tile: an Ethernet connection, accessible from SSH for debugging purposes; a UART interface to collect the logs of the running application and communicate with the Soc; a monitor JTAG interface for the global Soc.

- **Processor Tile**: provides the instantiation of a CPU core for the SoC. It is possible to choose amongst the currently supported architecture in the ESP portfolio: a SPARC 32-bit LEON3 core, from Cobham Gaisler; the RISC-V 64-bit Ariane core, from the PULP Platform of ETH of Zurich; and the RISC-V 32-bit Ibex core, from lowRISC. All the processors can be equipped with a dedicated cache hierarchy on their private NoC line defined at SoC tiling definition time that can reach the full address space. Moreover, all of the three options support Linux application execution using the dedicated ESP build environment.

- **Accelerator Tile**: provides the instantiation of a pool of user-defined hardware that execute on-demand coarse grain tasks. The accelerator operation

can be controlled through its globally memory mapped configuration registers, some of them can be user-defined, to proper set parameters and operations state. On running state, this tile can directly access to memory, the shared local or the off-chip global, using a DMA interface with a span of configurable data coherence modes. Optionally, it is provided a TLB with address translation, configurable at design time, so that the accelerator accesses to memory are virtually on contiguous chunks of memory. After finalization of the task, the accelerator can autonomously notify the core and be ready to be called again.

- **Empty Tile**: its only purpose is to provides the regularity for NoC generation and manufacturing of the SoC. There are no special feature introduced by this tile, as the name suggests.

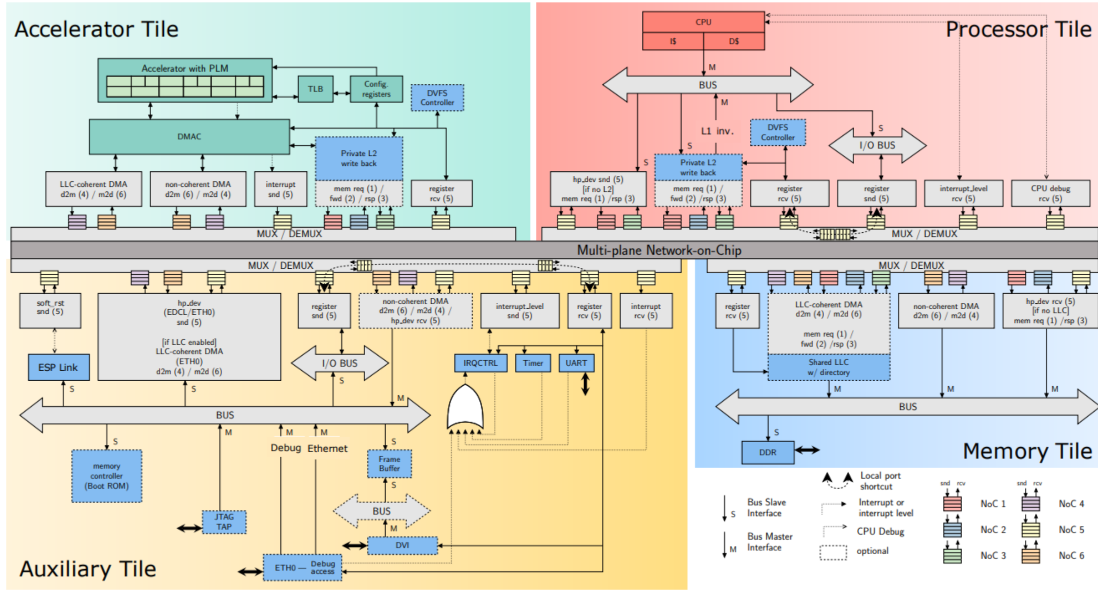

Figure 6.2: ESP Tiles interaction with NoC [21]

ESP unlocks a dedicated development suite, integrating multiple CADs, for application-specific hardware design. This flow can be customized based on the choice of the designer and on different abstraction level description languages.

6.1 Hardware Accelerator Design Flow

ESP flexible methodology embraces of a variety of languages to develop custom hardware. The designer has a wide choice from different abstraction level languages from a cycle-accurate description in HDL, like Verilog or VHDL, to untimed

C/SystemC/C++ and native languages for ML processing as HSL4ML. The user is provided with a full-aided flow for the higher abstraction level languages along with functional verification instruments that assure the quality of the design. On one hand, for RTL description the flow is straightforward to comply with the interface wrappers specification provided by Department of Computer Science of Columbia University and implement the inner modules without constraints. The HLS design on the other hand, independently from the language chosen, has a much more eased flow. The main steps are:

- **Generation of the skeleton**: in this phase the designer automatically generates the accelerator interface, required to comply with ESP framework, customizing the configuration registers, the ID and details on input and output of the accelerator. In this preliminary phase, the design must choose the description language desired so that the platform can automatically generate a template for the testbench and firmware drivers for Linux.
- **Customization**: taking as entry point the definition of previous steps, the designer defines the computation part describing functional units with the tools provided by the HLS. In this phase, the timing is marginally taken into account so that the designer can focus only on algorithms that describes the functionality of the accelerator. Finally, it is necessary to take care of the input generation or fetching and output storing, using the DMA interface provided by ESP with the functions native of the language chosen, for which a loosely timed description must be implemented.
- **HLS Validation**: This is an additional linting step that offers a further validation of the description in the same language used for previous step. The designer can implement a testbench that feeds the accelerator with the DMA channels and configuration registers, emulates its behaviour with a reference function, probes the output of the DMA channels and then validate the overall results. This testbench must be written anyway to also test the RTL.
- **RTL Generation**: in this phase the loosely-coupled/untimed HLS is translated into a cycle-accurate description of the hardware: the loops are mapped into control FSM and counters, variable into registers, functions into components and if conditions into muxes. The designer is expected to set the design constraints onto target frequency, area ad power consumption. This step is highly customizable with numerous directives that the designer can provide to the HLS tool to target the outcoming RTL toward one or another figure of merit, to set the effort to reach the specification and components to map variables or specific functions. Thanks to this, it is possible to explore the design space for the desired algorithm and decide the best implementation for the application required. In the end, the output of this step is a Verilog/VHDL that fully describes the accelerator.

- **RTL Validation**: in this last final step, the design is put under test to check its full functionality. The testbench previously described in HLS is translated to SystemVerilog, and the design can be validated in an RTL test environment, as Modelsim or QuestaSim, with the possibility to probe all the signals and ensure the full desired operation. This is a key step to debug the major issues deriving HLS to RTL translation ambiguities.

Although the described flow seems heavy, HLS is one of the fastest way to design a custom hardware. This description methods unlocks for designers a lot of benefits[22]: directly port software code directly into hardware, allowing restructuring the algorithm changes in the structure in the native language without passing from RTL; ignore the control side of the hardware that is usually demanded to the HLS tool; design space exploration becomes very fast and easy optimizing the HLS source code and structuring a synthesis directive set; HLS tools can provide a reasonably accurate insight on resources and performances before the gate-level synthesis step, further improving design space exploration time; verification time is much less because of the possibility to write and simulate the testbenches directly written in HLS.
The real reasons why this kind of methodology has not overtaken all the others are a few: it is still needed a linted code base to generate quality hardware much more constraint that the a simple software code for compilation; recursion is a software technique widely used, when compiled code is necessary, that cannot be translated efficiently into hardware along with all the concurrency statements because timing is offloaded from designer's duties; the output RTL isn't human readable and trying to manually intervene on can be very harsh. If this compromises are manageable for a certain project, this methodology can be very powerful. Moreover inside the ESP framework, the limitations aforementioned are utterly compensated by the suite itself that provides a strict quality gated automated flow.

6.2 Fully Connected Accelerator Design Flow

Amongst the wide offer that ESP presents, it has been chosen to use a C++ HLS language, supported by Mentor Catapult HLS, for describing a hardware accelerator that maps the algorithm of a fully connected layer. Following the path laid out before, the choices for each step are presented below:

- **Generation of the skeleton**: it was defined a set of custom configuration register needed to pass to the accelerator some parameters needed for computation and physical addresses of inputs, weights and output. Then it was generated a ESP compliant DMA interface, as described in Chapter 5.1:

  - *conf_info* is a sequence the memory-mapped registers that encapsulate

the accelerator configuration, comprehensive of default and the aforementioned custom ones, settable through the CPU

- *dma_read_ctrl* and *dma_write_ctrl* ports to interface the SoC DMA controller
- *dma_read_chnl* and *dma_write_chnl* interface the main memory, making the accelerator master of the communication, loading inputs and storing output of the accelerator
- *acc_done* is an IRQ signal that is used to notify the CPU task completion

- **Customization**: first of all, a DMA read function must be implemented to load the necessary data into the accelerator PLM. The DMA NoC width supported with the integration of the Ibex core is 32-bit, the data width of PLMs is 16-bit, as describe in previous chapter, therefore two reads at DMA cycle must be performed to leverage all interconnection parallelism and speed up at most data transfers. It was also implemented a logic that computes the number of 32-bit words necessary to read all the inputs and weights based on N parameter and precision of operation. The core description of the proposed accelerator can be taken from Chapter 5 , so it is omitted at this stage where the focus is ESP integration. Using the same logic of the memory read interface, two transactions of 32-bit width are used to write a single 64-bit output of the accelerator.

- **HLS Validation**: a simple C++ testbench was implemented to test the correctness of the algorithm defined in HLS accelerator core. The codeimplements few functions to accomplish the task described below.

  – **GenerateInputsWeights**: Generate random values to fill the input and weight tensor to test the application. Generates a number of values based on the input and output dimension give, rescaled from the precision specified to the number of 32-bit words needed, to stimulate the HLS of accelerator and the simulated behaviour that is used as reference.

  – **FullyConnected_tb**: Implements the function described in the HLS for the fully connected based on the precision-wise operations and accumulation parallelism. Outputs a tensor of 64-bit elements as the accelerator.

  – **Validation**: Takes both the output generated by the mocked behaviour and real HLS and confront them. Reports the results for every entry with a print and at the end the number of errors spotted in the validation process.

  – **CCS_MAIN**: Implements the main of the testbench. Coordinates the execution as follows: calls GenerateInputsWeights function; writes the configuration in the accelerator; inject in DMA channels the input-weight

data required by the accelerator for computation; runs the body of the accelerator HLS; polls for end of execution of the accelerator; probes the DMA output channels for filling the output data tensor of the accelerator; generates the reference results with FullyConnected_tb; calls the validation functions to check for errors; loops to change the precision of operation and terminate when the iterations have reached the set value; returns.

Then the testbench can be execute testing the HLS of the accelerator. Below is reported the frames of validation log that shows the three tested precision results.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS 2
INFO:   Validation():   - 18 - OUT: 227 | GOLD: 227
INFO:   Validation():   - 19 - OUT: 156 | GOLD: 156
INFO:   Validation():   - 20 - OUT: 23 | GOLD: 23
INFO:   Validation():   - 21 - OUT: 173 | GOLD: 173
INFO:   Validation():   - 22 - OUT: -71 | GOLD: -71
INFO:   Validation():   - 23 - OUT: 71 | GOLD: 71
INFO:   Validation():   - 24 - OUT: 99 | GOLD: 99
INFO:   Validation():   - 25 - OUT: 376 | GOLD: 376
INFO:   Validation():   - 26 - OUT: 41 | GOLD: 41
INFO:   Validation():   - 27 - OUT: 340 | GOLD: 340
INFO:   Validation():   - 28 - OUT: 246 | GOLD: 246
INFO:   Validation():   - 29 - OUT: 77 | GOLD: 77
INFO:   Validation():   - 30 - OUT: 13 | GOLD: 13
INFO:   Validation():   - 31 - OUT: -67 | GOLD: -67
INFO:   Validation(): -----------------
INFO:   Validation(): -#Errors : 0 ---
INFO:   Validation(): -----------------
INFO:   Validation(): -----------------
INFO:   Validation(): VALIDATION : PASS
INFO:   Validation(): -----------------
```

Figure 6.3: Results of High-Level Simulation of the GEMM Accelerator

- **RTL Generation**: after the completion of the validation of the algorithm is possible to generate the RTL description of the accelerator with CatapultHLS tool. The tool needs some directive about the I/O ports,target PPA of the overall design, parallelization factors, pipelining, blocks mapping and clocks specs. These constraints can be set through a configuration TCL file. It basically describes the design choices presented in Chapter 4 .
- **RTL Validation**: finally the validation of the RTL is mandatory to check the correctness of description. This fundamental step ensures the correctness of interpretation of the HLS description by CatapultHLS. ESP builds a verification environment with QuestaSim and reuses the testbench written in C. That guarantee an exact match between the behaviour intended and the one obtained. The waveform of key signals captured from the tool are reported below:

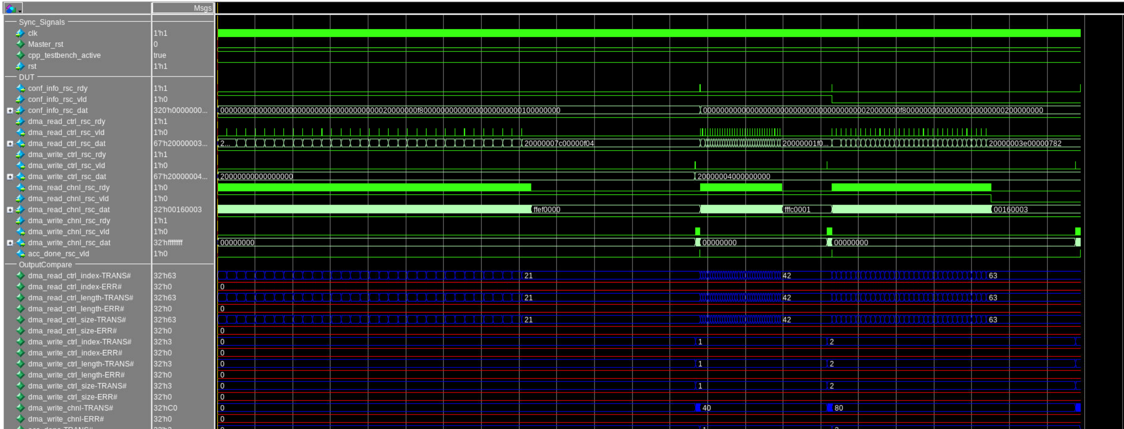

Figure 6.4: RTL Simulation of the GEMM Accelerator Standalone

As shown by the waveform above the accelerator correctly behaves in its configuration loading, input-weight loading, compute and write back stages. Moreover, the goodness of the description is further ensured by the log acquired from the simulation that match exactly the one taken from HLS validation.

6.3 SoC Design Flow

Usually, an integration of a complex SoC requires a lot of effort and design time to select the interconnection protocols, shared memory coherence levels, off-chips interfaces and other general system requirements: ESP offers an higher abstraction view focused on the system, instead on the single subsystem, providing a regular standardized NoC multi-plane and a coarse-grain data coherence level reconfigurability that masks to the final user most of the integration issues. The SoC configuration step is accessible through a GUI that allows to customize some parameters:

- **NoC Tiling**: sets the tile matrix height and width of the NoC. All the tiles are characterized from now on by their relative location (X,Y) that allows a point-to-point data exchange between tiles.
- **Tiles order**: specifies the type of each chosen tile. It is compulsory to have at least one memory tile or one local-shared memory tile and one I/O tile. Some tile types have further customizable options, for example processors and accelerator can be integrated with an L2 cache, a DVFS controller or a different clock domain. Moreover for each accelerator tile, it is possible to choose the accelerator type and implementation.
- **Processor Architecture**: defines the CPU tile architecture among the one

provided by ESP. There is also the possibility to extend ESP built-in pool of architectures with a dedicated flow.

- **Cache Hierarchy**: defines the specific implementation of the processor's L2 cache and memory tile's LLC. There are available some different choices based on the requirements of the design
- **Shared-Local Memory size**: sets the size of all the shared-local memory tiles. Based on the application that runs on the SoC that parameter may significantly impact on the performances of the system.
- **Data Transfer Addressing mode**: defines the accelerator's data transfer addressing mode, the two option available are a big-physical addressing space, where all addresses are interpreted as absolute, and a scatter-gather mode, where the addresses are relative to a page table and a TLB so that separated portions of memory can be accessed sequentially by the accelerator.
- **Hardware monitors**: sets some local debug JTAG-like interfaces to help monitor memory accesses, accelerators status, caches misses and DDR bandwidth.

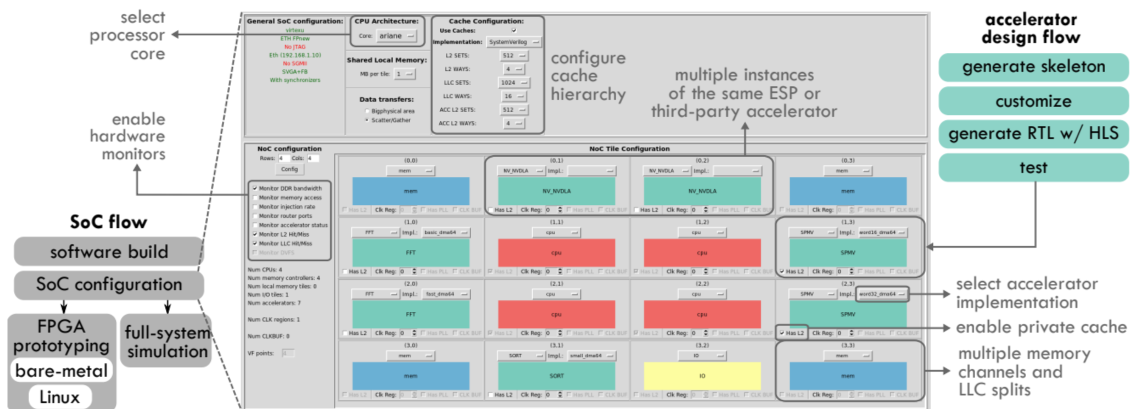

Figure 6.5: ESP Design Flow with SoC Configuration GUI [21]

After the definition of the SoC, it is possible to write the application code to test the full system. The code can be written in plain C and equipped with some ESP built-in function that leverage, for tile interactions, a device-tree and the physical address partitioning implemented by Department of Computer Science of Columbia University and showed below.

Address Map		
CPU	Leon3	Ariane/Ibex
Main Memory (1GB)	0x40000000 - 0x7FFFFFFF	0x80000000 - 0xBFFFFFFF
I/O and registers (256 MB)	0x80000000 - 0x8FFFFFFF	0x60000000 - 0x6FFFFFFF
Bootrom (128KB)	0x00000000 - 0x0001FFFF	0x00000000 - 0x0001FFFF
Frame buffer (512KB)	0x30100000 - 0x3017FFFF	0x30100000 - 0x3017FFFF
RISC-V CLINT (768KB)	n/a	0x02000000 - 0x020BFFFF
SLM scratchpad (64MB)	0x04000000 - 0x07FFFFFF	0x04000000 - 0x07FFFFFF
LPDDR scratchpad (1GB) - GF12 only	0xC0000000 - 0xFEFFFFFF	0xC0000000 - 0xFFFFFFFF
Leon3 AHB device tree area (16MB)	0xFF000000 - 0xFFFFFFFF	n/a

Figure 6.6: ESP Address Space Definition [23]

As a final step, it is possible to build the software baseline, as a baremetal or Linux application, and to run the test application in a RTL simulated environment or onto an FPGA. The framework provides a straightforward approach, based on a Makefile tree, that automatically detects the build target, peripheral addresses and accelerator attached to the NoC and maps the RAM available based on a device-tree generated at SoC tiling define time.

6.4 Fully Connected SoC Design Flow

In order to test the developed hardware with a real NN inference, it was necessary to integrate it in a SoC with at least a core that execute the code necessary to re-route the data toward the accelerator. Following the path laid out in the previous section, the choices for the application test SoC are the following:

- **NoC Tiling**: the configuration chosen for the NoC is the minimal to support the integration of an accelerator: 2x2.
- **Tiles order**: top left tile is a memory tile, top right is a CPU tile, the bottom left is the accelerator tile and the bottom right is the I/O tile.
- **Processor Architecture**: it was required to test the accelerator with the smallest low-power processor, so the choice was RISC-V 32-bit Ibex core, from lowRISC. The implementation chosen from RTL also excluded the generation of its L1 cache, branch prediction unit and CSRs.
- **Cache Hierarchy**: in order to reduce at most the size and consumption of the resulting SoC, it was chosen to exclude the generation of any additional memory. Moreover this choice allows to have more coherent time measurements across different software applications.

- **Shared-Local Memory size**: no SLM tiles were generated so it was left at default 512kb.
- **Data Transfer Addressing mode**: to ensure proper operation of the developed accelerator, the address mode was set to big-physical address space so that the hardware developed can freely access memory. This allows to simply fill the configuration registers with input, weight and output pointer.
- **Hardware monitors**: no hardware monitors were used.

Figure 6.7: Chosen SoC Configuration to test the accelerator in a real environment

After the definition of the SoC layout, it was developed a simple code baseline to test the functionality of the hardware call of the accelerator. The code is intended to testbench the whole SoC, that sequentially execute the following operations:

1. Allocate memory for input, output and weights
2. Generate test values for inputs and weights
3. Compute reference outputs with a software function
4. Write configuration and activates the accelerator
5. Stay in polling for accelerator done signal
6. Fetch output from the accelerator
7. Compare reference and accelerator outputs

8. Report test results

Collecting all the hardware and software elements developed, the whole SoC RTL can be tested in *ad hoc* environment, the chosen one was QuestaSim. The first step is to build the developed software, specifying the target processor, and generate the binaries for the ROM and RAM memories. These files are loaded into the simulated memory at runtime. After that, it is possible to open the test suite, compile the whole design, add the visibility on the waves of interest and run the simulation. The waveform of the accelerator signals during the simulation has been reported below, zooming on the time region of operation.

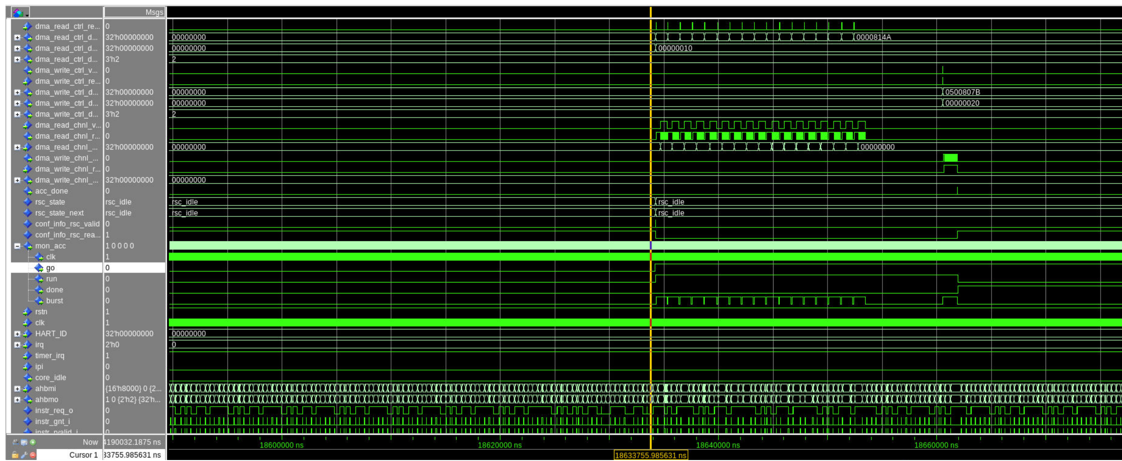

Figure 6.8: RTL simulation of the SoC hosting the GEMM Accelerator

From the image, it is distinguishable every state of operation of the accelerator:

1. Configuration setting phase: the CPU writes the configuration registers to set operation mode
2. Start of the accelerator: the CPU write the start register to activate the accelerator
3. Load of inputs: the accelerator fetches from the external memory the data required for its computations through the DMA.
4. Computation: effectively computes the results from the inputs and weights stored in the PLM. The ports of I/O of accelerator are in a steady state because the accelerator has no concurrency on stages.
5. Store of outputs: the accelerator writes back the results from the PLM of output to the external memory at the set address.
6. Notification of completion and idle: the accelerator notifies the end of the task and let the program continue its course CPU-side going in idle state.

The focus of the test was the interaction of the accelerator with its socket and the other tiles: the accelerator passed the test ad behaves exactly as intended matching the behavior with the high-level and with the standalone verification.

Chapter 7

Accelerated Kernel

In this chapter will be presented the actual TFLM implementation of the software needed to use the accelerator designed, in a fully connected kernel.

Extend TFLM Data Types Compatibility

As aforementioned in Chapter 2.3, TFLM supports a limited set of the possible integers data types and that can limit the feqatures available offered by the accelerator developed during this work. Based on the acceleration leveraging low precision parallelization provided by the hardware developed, it is a nice to have featured the extension of compatibility of the framework and OpResolver to full support low-precision data types. This work has been done by Edward Manca to support and validate his work[15], including also adaptation of the kernel parameters to STAR approach: the modified framework utter support precision homogeneous operation for *int16*, *int8* and *int4*.
Although, it was necessary to enable OpResolver for fully connected kernel to call the function, specialized for the data types required, that supports the accelerator call based on a flag settable at compilation time through the MakeFile chain. Now it is possible to dive deeper on the custom code developed where it is implemented the accelerator call, memory tiling algorithms and support functions to adapt the data handling to ST approach.

Accelerator Registers

The accelerator was designed to be interfaced easily to the framework, taking into consideration at design time parameters, as tensor data ordering for sequential memory accesses or maximum accumulation depth, to better accommodate the interface between hardware and software.
First of all it is necessary to define the configuration registers that will be used

to pilot the accelerator during the execution of the kernel. This code maps the registers described in Chapter6.2.

Source Code 7.1: Accelerator Configuration Registers Allocation

```
//Pointers to accelerator registers
#include "tensorflow/lite/kernels/internal/reference/integer_ops/star/include/fc_cxx_catapult.h"
#include "tensorflow/lite/kernels/internal/reference/integer_ops/star/include/accelerator.h"
volatile unsigned * FC_CMD = (unsigned *) (FC_ADDR + CMD_REG);//#define CMD_REG 0x00
const volatile unsigned * FC_STATUS = (unsigned *) (FC_ADDR + STATUS_REG);//#define STATUS_REG 0x04  Read
    ↪ only
volatile unsigned * FC_SELECT = (unsigned *) (FC_ADDR + SELECT_REG);//#define SELECT_REG 0x08
const volatile unsigned * FC_DEVID = (unsigned *) (FC_ADDR + DEVID_REG);//#define DEVID_REG 0x0c Read only
volatile unsigned * FC_PT_ADDRESS = (unsigned *) (FC_ADDR + PT_ADDRESS_REG);//#define PT_ADDRESS_REG 0x10
volatile unsigned * FC_PT_NCHUNK = (unsigned *) (FC_ADDR + PT_NCHUNK_REG);//#define PT_NCHUNK_REG 0x14
volatile unsigned * FC_PT_SHIFT = (unsigned *) (FC_ADDR + PT_SHIFT_REG);//#define PT_SHIFT_REG 0x18
const volatile unsigned * FC_PT_NCHUNK_MAX = (unsigned *) (FC_ADDR + PT_NCHUNK_MAX_REG);//#define
    ↪ PT_NCHUNK_MAX_REG 0x1c Read only
volatile unsigned * FC_PT_PT_ADDRESS_EXTENDED = (unsigned *) (FC_ADDR +
    ↪ PT_ADDRESS_EXTENDED_REG);//#define PT_ADDRESS_EXTENDED_REG 0x20
const volatile unsigned * FC_COHERENCE = (unsigned *) (FC_ADDR + COHERENCE_REG);//#define COHERENCE_REG
    ↪ 0x24 Read only
volatile unsigned * FC_P2P = (unsigned *) (FC_ADDR + P2P_REG);//#define P2P_REG 0x28
volatile unsigned * FC_YX = (unsigned *) (FC_ADDR + YX_REG);//#define YX_REG 0x2c
volatile unsigned * FC_SRC_OFFSET = (unsigned *) (FC_ADDR + SRC_OFFSET_REG);//#define SRC_OFFSET_REG 0x30
volatile unsigned * FC_DST_OFFSET = (unsigned *) (FC_ADDR + DST_OFFSET_REG);//#define DST_OFFSET_REG 0x34
volatile unsigned * FC_SPANDEX = (unsigned *) (FC_ADDR + SPANDEX_REG);//#define SPANDEX_REG 0x38
volatile unsigned * FC_FLAGS = (unsigned *) (FC_ADDR + FC_CXX_FLAGS_REG);//#define FC_CXX_FLAGS_REG 0x40
volatile unsigned * FC_OUT_ADD = (unsigned *) (FC_ADDR + FC_CXX_OUT_ADD_REG);//#define FC_CXX_OUT_ADD_REG
    ↪ 0x44
volatile unsigned * FC_W_ADD = (unsigned *) (FC_ADDR + FC_CXX_W_ADD_REG);//#define FC_CXX_W_ADD_REG 0x48
volatile unsigned * FC_IN_ADD = (unsigned *) (FC_ADDR + FC_CXX_IN_ADD_REG);//#define FC_CXX_IN_ADD_REG
    ↪ 0x4c
volatile unsigned * FC_N = (unsigned *) (FC_ADDR + FC_CXX_N_REG);//#define FC_CXX_N_REG 0x50
volatile unsigned * FC_M = (unsigned *) (FC_ADDR + FC_CXX_M_REG);//#define FC_CXX_M_REG 0x54
volatile unsigned * FC_OFFSET_Q_DATA = (unsigned *) (FC_ADDR + FC_CXX_OFFSET_Q_DATA_REG);//#define
    ↪ FC_CXX_OFFSET_Q_DATA_REG 0x58
volatile unsigned * FC_OFFSET_PE = (unsigned *) (FC_ADDR + FC_CXX_OFFSET_PE_REG);//#define
    ↪ FC_CXX_OFFSET_PE_REG 0x5c
volatile unsigned * FC_OPTIONS = (unsigned *) (FC_ADDR + FC_CXX_OPTIONS_REG);//#define FC_CXX_OPTIONS_REG
    ↪ 0x60
volatile unsigned * FC_ACC = (unsigned *) (FC_ADDR + FC_CXX_ACC_REG);//#define FC_CXX_ACC_REG 0x64
```

As shown in the snippet above the registers are istantiated as volatile pointers: that allows a direct access to the content by dereference, prevents TFLM to touch that memory zone and prevents GCC compiler to optimize the code and reorder instructions when the registers are used. This approach ensures the safety of this memory regions and allows a fast read/write of the contents with a RISC-V native instruction.

Accelerator Hardware Call Function

Then, the hardware call function can be defined. This functions passes directly its parameters to the configuration registers, dereferencing the pointer directly, then the accelerator is started and the CPU polls the completion of the task. After that the accelerator is deactivated and the function returns.

Source Code 7.2: Accelerator Hardware Call Function

```
void __attribute__((optimize("O0"))) HWCall_fc_ST (uint32_t in_add, uint32_t w_add, uint32_t out_add, uint8_t
    ↪ acc_flag, uint8_t relu_en,  uint8_t q_flag, uint32_t offset_q_data, int update_reg, uint8_t N,
    ↪ uint8_t M, uint32_t offset_PE, uint8_t q_confg, uint32_t precision_opt){

    volatile unsigned  done = 0x00;
    //make sure that the computation is done
    done = *(FC_STATUS);
    while ((done &= STATUS_MASK_RUN) == 0x01) {
            done = *(FC_STATUS);
```

```
    }
    *FC_CMD = 0x0;//Deactivation of accelerator
    *FC_IN_ADD = in_add;//Input address
    *FC_W_ADD = w_add;//Weight address
    *FC_OUT_ADD = out_add;//Out address
    *FC_ACC = acc_flag;//Selecting of accelerator
    *FC_FLAGS = relu_en <<2 | q_flag;//Quantization enable // this constant activates or not the quantization
         ↪ for the outputs
    *FC_OFFSET_Q_DATA = offset_q_data;//includes (bias + input_offset*filter_val)/filter_scale ->don't know
         ↪ how to compose it
    if (update_reg){
        *FC_N = N;//Input number
              *FC_M = M;//Output neurons
        *FC_OFFSET_PE = offset_PE;//Offset of PEs
        *FC_OPTIONS = (q_confg <<4)|precision_opt;//Precision of operations // if 0 just it takes one cin
             ↪ value, if 1 it takes 4 values (4 LSB) in cin , if 2 or 3 it takes 2 values (8 LSB) in cin //
             ↪ q_confg when 0 - 16X   1 - 4X    2 - 8X
    }

        // Flush (customize coherence model here)
        //esp_flush(ACC_COH_NONE); Soc created has no cache
        *FC_CMD = CMD_MASK_START; //Start of accelerator
        while (done == 0x00) {
                done = *(FC_STATUS);
        }
        done &= STATUS_MASK_DONE;//done = *FC_STATUS;
  if (done == 0x02) {
        *FC_CMD = 0x0; //Deactivation of accelerator
  }

}
```

The parameters of the function refer to the configuration registers that needs effective settings. The others are left with the reset or at least same value as before.

Fully Connected Tiling Function

This function implements the memory tiling needed to utterly pass the tensors to the accelerator. Basically it calls the accelerator based on the tensors sizes configuring the run based on memory needing and the precision required for computation. It is divided in three different cases possible:

- **Input Tensor exceeds Input PLM size**: it is also contemplated when all the tensors doesn't fit the PLMs. In this case the execution is constrained by the tensor memory order to generate one output at each call of the accelerator: the accelerator runs iteratively with maximum input size and single output size till the remaining inputs to compute are below the maximum, at this stage the remainders inputs are used in computation; all of this is iterated till all the needed output are generated.
- **Output Tensor exceeds Output PLM size**: in this case the inputs fit all in the PLM. Therefore the accelerator is called with the given number of input and maximum number of output iteratively till the number of output to generate are below the maximum supported.
- **All Tensor fit in the PLMs**: the functions simply calls the accelerator with the needed parameters.

Source Code 7.3: Accelerator Tiling Function

```c
void __attribute__((optimize("O0"))) Tiled_fc_ST_Accelerator(const int32_t *input, const int32_t *weight,
     ↪ const int64_t *output, uint8_t relu_en, uint32_t *offset_q_data, int N, int M, uint32_t offset_PE,
     ↪ uint8_t q_confg, uint32_t precision_opt, int q_mask){
        uint32_t in_p = ((uint32_t)(input) >> 2);
        uint32_t w_p =  ((uint32_t)(weight) >> 2);
        uint32_t out_p = ((uint32_t)(output) >> 2);
  int parall_shift = 1;
        int N_rem = N;
        int M_rem = M;
  if (precision_opt == 0) { //16b
    parall_shift = 1;
  }
  else if (precision_opt == 1) { //8b
    parall_shift = 2;
  }
  else { //4b
    parall_shift = 3;
  }

        if (N > N_MAX) { //The case in which (N > N_MAX && M > M_MAX) is also represented by this entry
                //It's necessary to split the number of inputs (output stationary approach)
                for (int out = 0; out < M; out++) {
                        in_p = ((uint32_t)(input) >> 2);
      uint8_t flag = 0;
                        for(int i = 0; i < N/N_MAX; i++){
                                HWCall_fc_ST (in_p, w_p, out_p, flag, relu_en, (!q_mask), offset_q_data, 1,
                                     ↪ (uint8_t) (N_MAX-1), 1, offset_PE, q_confg, precision_opt);
        // HWMockCall_fc_ST (in_p, w_p, out_p, flag, relu_en, (!q_mask), offset_q_data, 1, (uint16_t)
             ↪ (N_MAX), 1, offset_PE, q_confg, precision_opt);
        flag = 1;
                                in_p += (N_MAX >> parall_shift);
                                w_p += (N_MAX >> parall_shift);
                        }
                        //Remainder of output neurons from division N/N_MAX
                        N_rem = N % N_MAX;
                        if (N_rem != 0) {
                                HWCall_fc_ST (in_p, w_p, out_p, flag, relu_en, (!q_mask), offset_q_data, 1,
                                     ↪ (uint8_t) (N_rem-1), 1, offset_PE, q_confg, precision_opt);
        // HWMockCall_fc_ST (in_p, w_p, out_p, flag, relu_en, (!q_mask), offset_q_data, 1, (uint16_t)
             ↪ (N_rem), 1, offset_PE, q_confg, precision_opt);
                                w_p = ((uint32_t)(weight) >> 2) + (out + 1)*(N >> parall_shift); // >> 2);
                        }
                        out_p += 2;
                }
        }
        else if (M > M_MAX) {
                //It's necessary to split the number of outputs (simply dividing output neurons in chunks of
                     ↪ size M_MAX)
                for(int i = 0; i < M/M_MAX; i++){
      // printf("Calling HW function\n");
                        HWCall_fc_ST (in_p, w_p, out_p, 0, relu_en, (!q_mask), offset_q_data, 1, (uint8_t)
                             ↪ (N-1), (uint8_t) M_MAX, offset_PE, q_confg, precision_opt);
      // HWMockCall_fc_ST (in_p, w_p, out_p, 0, relu_en, (!q_mask), offset_q_data, 1, (uint16_t) N, (uint8_t)
           ↪ M_MAX, offset_PE, q_confg, precision_opt);
                        out_p += 2*M_MAX;
                        w_p += (N >> parall_shift);
                }
                //Remainder of output neurons from division M/M_MAX
                M_rem = M % M_MAX;
                if (M_rem != 0) {
      // printf("Calling HW function\n");
                        HWCall_fc_ST (in_p, w_p, out_p, 0, relu_en, (!q_mask), offset_q_data, 1, (uint8_t)
                             ↪ (N-1), (uint8_t) M_rem, offset_PE, q_confg, precision_opt);
      // HWMockCall_fc_ST (in_p, w_p, out_p, 0, relu_en, (!q_mask), offset_q_data, 1, (uint16_t) N, (uint8_t)
           ↪ M_rem, offset_PE, q_confg, precision_opt);
                        w_p += (N >> parall_shift);
                }

        }
        else{
                //PLM of accelerator is enough so it's a simple HW call
                HWCall_fc_ST (in_p, w_p, out_p, 0, relu_en, (!q_mask), offset_q_data, 1, (uint8_t) (N-1),
                     ↪ (uint8_t) M, offset_PE, q_confg, precision_opt);
    // HWMockCall_fc_ST (in_p, w_p, out_p, 0, relu_en, (!q_mask), offset_q_data, 1, (uint16_t) N, (uint8_t)
         ↪ M, offset_PE, q_confg, precision_opt);

        }
}
```

The function takes the overall tensors dimensions of tensors, the pointers and the

precision of data and tiles the memory to processes to fit into the PLMs available so that the accelerator can read contiguous memory chunks and process correctly the task.

ST Data Handling Auxiliary Functions

After that, it is necessary to define few auxiliary functions to handle the computation. This functions generally handle the ST data formatting to help the framework to comply with it.
The first function is related to compute the term related to weight-input_offset product. The functions accepts a 32-bit value of replicated offset based on the specific ST configuration, using the template specialization C++ technique, and the specific filter value for multiplication.

Source Code 7.4: ST Accelerator Offset Function

```
template<int OP_parall>
inline void __attribute__((always_inline)) AddOffset_fc_ST_Accelerator(acc_t& acc, int32_t& filter_val,
    ↪ int32_t& in_offs_arr);

template<>
inline void __attribute__((always_inline)) AddOffset_fc_ST_Accelerator<16>(acc_t& acc, int32_t& filter_val,
    ↪ int32_t& in_offs_arr)
{
  int32_t fil_val_1 = ((filter_val) & 0x0000FFFF) < 0x8000 ? ((filter_val) & 0x0000FFFF) : ((filter_val) |
      ↪ 0xFFFF0000);
  int32_t fil_val_2 = filter_val>>16; // shift already performs sign-extension
  int32_t off_val_1 = ((in_offs_arr) & 0x0000FFFF) < 0x8000 ? ((in_offs_arr) & 0x0000FFFF) : ((in_offs_arr) |
      ↪ 0xFFFF0000);
  int32_t off_val_2 = in_offs_arr>>16;

  acc += fil_val_1 * off_val_1;
  acc += fil_val_2 * off_val_2;

  return;
}
template<>
inline void __attribute__((always_inline)) AddOffset_fc_ST_Accelerator<8>(acc_t& acc, int32_t& filter_val,
    ↪ int32_t& in_offs_arr)
{
  int32_t fil_val_1 = ((filter_val) & 0x000000FF) < 0x80 ? ((filter_val) & 0x000000FF) : ((filter_val) |
      ↪ 0xFFFFFF00);
  int32_t fil_val_2 = ((filter_val>>8) & 0x000000FF) < 0x80 ? ((filter_val>>8) & 0x000000FF) :
      ↪ ((filter_val>>8) | 0xFFFFFF00);
  int32_t fil_val_3 = ((filter_val>>16) & 0x000000FF) < 0x80 ? ((filter_val>>16) & 0x000000FF) :
      ↪ ((filter_val>>16) | 0xFFFFFF00);
  int32_t fil_val_4 = filter_val>>24;
  int32_t off_val_1 = ((in_offs_arr) & 0x000000FF) < 0x80 ? ((in_offs_arr) & 0x000000FF) : ((in_offs_arr) |
      ↪ 0xFFFFFF00);
  int32_t off_val_2 = ((in_offs_arr>>8) & 0x000000FF) < 0x80 ? ((in_offs_arr>>8) & 0x000000FF) :
      ↪ ((in_offs_arr>>8) | 0xFFFFFF00);
  int32_t off_val_3 = ((in_offs_arr>>16) & 0x000000FF) < 0x80 ? ((in_offs_arr>>16) & 0x000000FF) :
      ↪ ((in_offs_arr>>16) | 0xFFFFFF00);
  int32_t off_val_4 = in_offs_arr>>24;

  acc += fil_val_2 * off_val_1;
  acc += fil_val_1 * off_val_2;
  acc += fil_val_4 * off_val_3;
  acc += fil_val_3 * off_val_4;

  return;
}
template<>
inline void __attribute__((always_inline)) AddOffset_fc_ST_Accelerator<4>(acc_t& acc, int32_t& filter_val,
    ↪ int32_t& in_offs_arr)
{
  int32_t fil_val_1 = ((filter_val) & 0x0000000F) < 0x8 ? ((filter_val) & 0x0000000F) : ((filter_val) |
      ↪ 0xFFFFFFF0);
  int32_t fil_val_2 = ((filter_val>>4) & 0x0000000F) < 0x8 ? ((filter_val>>4) & 0x0000000F) :
      ↪ ((filter_val>>4) | 0xFFFFFFF0);
```

```
  int32_t fil_val_3 = ((filter_val>>8) & 0x0000000F) < 0x8 ? ((filter_val>>8) & 0x0000000F) :
      ↪ ((filter_val>>8) | 0xFFFFFFF0);
  int32_t fil_val_4 = ((filter_val>>12) & 0x0000000F) < 0x8 ? ((filter_val>>12) & 0x0000000F) :
      ↪ ((filter_val>>12) | 0xFFFFFFF0);
  int32_t fil_val_5 = ((filter_val>>16) & 0x0000000F) < 0x8 ? ((filter_val>>16) & 0x0000000F) :
      ↪ ((filter_val>>16) | 0xFFFFFFF0);
  int32_t fil_val_6 = ((filter_val>>20) & 0x0000000F) < 0x8 ? ((filter_val>>20) & 0x0000000F) :
      ↪ ((filter_val>>20) | 0xFFFFFFF0);
  int32_t fil_val_7 = ((filter_val>>24) & 0x0000000F) < 0x8 ? ((filter_val>>24) & 0x0000000F) :
      ↪ ((filter_val>>24) | 0xFFFFFFF0);
  int32_t fil_val_8 = filter_val>>28;
  int32_t off_val_1 = ((in_offs_arr) & 0x0000000F) < 0x8 ? ((in_offs_arr) & 0x0000000F) : ((in_offs_arr) |
      ↪ 0xFFFFFFF0);
  int32_t off_val_2 = ((in_offs_arr>>4) & 0x0000000F) < 0x8 ? ((in_offs_arr>>4) & 0x0000000F) :
      ↪ ((in_offs_arr>>4) | 0xFFFFFFF0);
  int32_t off_val_3 = ((in_offs_arr>>8) & 0x0000000F) < 0x8 ? ((in_offs_arr>>8) & 0x0000000F) :
      ↪ ((in_offs_arr>>8) | 0xFFFFFFF0);
  int32_t off_val_4 = ((in_offs_arr>>12) & 0x0000000F) < 0x8 ? ((in_offs_arr>>12) & 0x0000000F) :
      ↪ ((in_offs_arr>>12) | 0xFFFFFFF0);
  int32_t off_val_5 = ((in_offs_arr>>16) & 0x0000000F) < 0x8 ? ((in_offs_arr>>16) & 0x0000000F) :
      ↪ ((in_offs_arr>>16) | 0xFFFFFFF0);
  int32_t off_val_6 = ((in_offs_arr>>20) & 0x0000000F) < 0x8 ? ((in_offs_arr>>20) & 0x0000000F) :
      ↪ ((in_offs_arr>>20) | 0xFFFFFFF0);
  int32_t off_val_7 = ((in_offs_arr>>24) & 0x0000000F) < 0x8 ? ((in_offs_arr>>24) & 0x0000000F) :
      ↪ ((in_offs_arr>>24) | 0xFFFFFFF0);
  int32_t off_val_8 = in_offs_arr>>28;

  acc += fil_val_4 * off_val_1;
  acc += fil_val_3 * off_val_2;
  acc += fil_val_2 * off_val_3;
  acc += fil_val_1 * off_val_4;
  acc += fil_val_8 * off_val_5;
  acc += fil_val_7 * off_val_6;
  acc += fil_val_6 * off_val_7;
  acc += fil_val_5 * off_val_8;

  return;
}
```

The second one is related to biasing and quantization. It is necessary to implement the same strategy used by TFLM, explained at Chapter 2.5, for this operations according to ST data formatting. It has been taken from the work developed by Edward Manca[15] that correctly implements it.

Source Code 7.5: STAR Biasing and Quantization Function Registers

```
template<int OP_parall>
inline int32_t __attribute__((always_inline)) getAccAddBiasAndQuantize_fc_STAR(acc_t& acc, const int32_t*
      ↪ bias_data, int& out_c, const int32_t* output_multiplier, const int32_t* output_shift, const int32_t&
      ↪ output_offset, const int32_t& output_activation_min, const int32_t& output_activation_max);

template<>
inline int32_t __attribute__((always_inline)) getAccAddBiasAndQuantize_fc_STAR<16>(acc_t& acc, const int32_t*
      ↪ bias_data, int& out_c, const int32_t* output_multiplier, const int32_t* output_shift, const int32_t&
      ↪ output_offset, const int32_t& output_activation_min, const int32_t& output_activation_max)
{
  #ifdef STAR_EMULATE

  int64_t acc_64 = acc;

  #else

    #ifdef FC_ST_ACCELERATOR
     int64_t acc_64 = acc;
    #else
    int32_t ext_acc;
    asm volatile("mac16sth %0, x0, x0\n":"=r"(ext_acc)::"memory");
    int64_t acc_64 = (((int64_t) ext_acc) << 32) | (((int64_t) acc) & 0x00000000FFFFFFFF);
    // int64_t acc_64 = (((int64_t) ext_acc) << 32) | acc;
    #endif

  #endif

  if (bias_data) {
      acc_64 += static_cast<int32_t>(bias_data[out_c]);
  }

  int32_t new_acc = MultiplyByQuantizedMultiplier(acc_64, output_multiplier[out_c],
                                                  output_shift[out_c]);
```

```
  new_acc += output_offset;
  new_acc = std::max(new_acc, output_activation_min);
  new_acc = std::min(new_acc, output_activation_max);

  return new_acc;
}
template<>
inline int32_t __attribute__((always_inline)) getAccAddBiasAndQuantize_fc_STAR<8>(acc_t& acc, const int32_t*
     ↪ bias_data, int& out_c, const int32_t* output_multiplier, const int32_t* output_shift, const int32_t&
     ↪ output_offset, const int32_t& output_activation_min, const int32_t& output_activation_max)
{
  int32_t new_acc = (int32_t)acc;

  if (bias_data) {
      new_acc += static_cast<int32_t>(bias_data[out_c]);
  }
  new_acc = MultiplyByQuantizedMultiplier(new_acc, output_multiplier[out_c],
                                          output_shift[out_c]);
  new_acc += output_offset;
  new_acc = std::max(new_acc, output_activation_min);
  new_acc = std::min(new_acc, output_activation_max);

  return new_acc;
}
template<>
inline int32_t __attribute__((always_inline)) getAccAddBiasAndQuantize_fc_STAR<4>(acc_t& acc, const int32_t*
     ↪ bias_data, int& out_c, const int32_t* output_multiplier, const int32_t* output_shift, const int32_t&
     ↪ output_offset, const int32_t& output_activation_min, const int32_t& output_activation_max)
{
  int32_t new_acc = (int32_t)acc;

  if (bias_data) {
      new_acc += static_cast<int32_t>(bias_data[out_c]);
  }
  new_acc = MultiplyByQuantizedMultiplier(new_acc, output_multiplier[out_c],
                                          output_shift[out_c]);
  new_acc += output_offset;
  new_acc = std::max(new_acc, output_activation_min);
  new_acc = std::min(new_acc, output_activation_max);

  return new_acc;
}
```

Finally it is necessary to implement the casting and write back function to better accommodate ST formatting: the function described performs a simple type cast using the template specialization C++ technique. The code, also taken from the same work as before[15], is shown below.

Source Code 7.6: STAR Casting Function

```
template<typename out_type, int out_Nbit>
inline void __attribute__((always_inline)) CastAndWrite_fc_STAR(int32_t& acc, out_type* output_data, int output_offs);

template<>
inline void __attribute__((always_inline)) CastAndWrite_fc_STAR<int16_t, 16>(int32_t& acc, int16_t* output_data, int output_offs)
{
  output_data[output_offs] = static_cast<int16_t>(acc);
  return;
}
template<>
inline void __attribute__((always_inline)) CastAndWrite_fc_STAR<int8_t, 8>(int32_t& acc, int8_t* output_data, int output_offs)
{
  output_data[output_offs] = static_cast<int8_t>(acc);
  return;
}
template<>
inline void __attribute__((always_inline)) CastAndWrite_fc_STAR<int8_t, 4>(int32_t& acc, int8_t* output_data, int output_offs)
{
  if (output_offs & 0x01)
  { // even (second 4bit element)
    output_offs = output_offs >> 1;
    output_data[output_offs] |= static_cast<int8_t>(acc) << 4;
  }
  else
  { // odd (first 4bit element)
    output_offs = output_offs >> 1;
    output_data[output_offs] = 0x00 | (0x0F & static_cast<int8_t>(acc));
  }
```

```
  return;
}
```

Fully Connected Kernel Function

Finally, it is possible to define the fully connected kernel function. It uses the code developed above to handle the execution and report the results complying with the framework requirements in terms of memory management and data manipulations. This kernel function comply with TFLM and interfaces the OpResolver of the framework, as explained in Chapter 2.1: collects the input tensors and output tensor pointers, the parameters and handle the whole computation. In the first stage extracts the parameters of computation, as the offsets or tensor dimensions, then loops on the batches the execution of a full kernel. The kernel execution is composed by few stages: the call of the accelerator tiling function to generate the input-weight product; then, looping both on output and input dimensions, performs the multiplication weight-input_offset and further accumulates it with the previous result; then adds the output bias and quantize the result; finally it casts the result in the correct type and writes it back to the output tensor.
The code is shown in the snippet below:

Source Code 7.7: Fully Connected Accelerated Kernel Function

```
//Fully Connected using the accelerator ST
template<int OP_parall, typename out_type, int out_Nbit> //V
void __attribute__((noinline)) FullyConnectedPerChannel_ST_Accelerator(
    const FullyConnectedParams& params, const int32_t* output_multiplier,
    const int32_t* output_shift, const RuntimeShape& input_shape,
    const int32_t* input_data, const RuntimeShape& filter_shape,
    const int32_t* filter_data, const RuntimeShape& bias_shape,
    const int32_t* bias_data, const RuntimeShape& output_shape,
    out_type* output_data) {
  #ifdef MEASURE_FC
  enable_counter();
  #endif

  const int32_t input_offset = params.input_offset;
//  const int32_t filter_offset = params.weights_offset;
  const int32_t output_offset = params.output_offset;
  const int32_t output_activation_min = params.quantized_activation_min;
  const int32_t output_activation_max = params.quantized_activation_max;
  TFLITE_DCHECK_GE(filter_shape.DimensionsCount(), 2);
  TFLITE_DCHECK_EQ(output_shape.DimensionsCount(), 2);

  TFLITE_DCHECK_LE(output_activation_min, output_activation_max);
  const int filter_dim_count = filter_shape.DimensionsCount();
  const int batches = output_shape.Dims(0);

  const int output_depth = filter_shape.Dims(filter_dim_count - 2);
  const int accum_depth = filter_shape.Dims(filter_dim_count - 1);
  constexpr int n_instr_parall = 32 / OP_parall;
  int norm_accum_depth = accum_depth / n_instr_parall;

  int32_t input_offset_array = MakeOffsetArray_fc_STAR<OP_parall>(input_offset);
  uint32_t precision_opt;
  uint8_t q_confg;
  alignas(32) static int64_t accum_acc[640] = { 0 };
      //Precision of Operations
      if (OP_parall == 16) { //16x16
        precision_opt = 0;
      }
      else if (OP_parall == 8) { //8x8
        precision_opt = 2;
      }
      else { //4x4
        precision_opt = 1;
```

```
      }
      //Quatization output bits
      if (out_Nbit == 16) { //16b
        q_confg = 0;
      }
      else if (out_Nbit == 8) { //8b
        q_confg = 2;
      }
      else { //4b
        q_confg = 1;
      }
        *FC_SELECT = *FC_DEVID;
  for (int b = 0; b < batches; ++b) {
    Tiled_fc_ST_Accelerator(&input_data[ 0 + b * norm_accum_depth], &filter_data[0], &accum_acc[0], 0,
         ↪ nullptr, accum_depth, output_depth, offset_PE, q_confg, precision_opt, 0); //Masked quantization
         ↪ and ReLu
    //Quantization is performed as stock TFLM must be packed as STAR although
      for (int out_c = 0; out_c < output_depth; ++out_c) {

        int64_t accum = accum_acc[out_c];
        for (int d = 0; d < norm_accum_depth; ++d) {
          int32_t filter_val = filter_data[out_c * norm_accum_depth + d];
          AddOffset_fc_ST_Accelerator<OP_parall>(accum, filter_val, input_offset_array);
        }
        int32_t quant_acc = getAccAddBiasAndQuantize_fc_STAR<OP_parall>(accum, bias_data, out_c,
             ↪ output_multiplier, output_shift, output_offset, output_activation_min, output_activation_max);
        int internal_output_offset = out_c + output_depth * b;
        CastAndWrite_fc_STAR<out_type, out_Nbit>(quant_acc, output_data, internal_output_offset);
      }
  }
  #ifdef MEASURE_FC
  disable_counter();
  #endif
}
#endif

}  // namespace reference_integer_ops
}  // namespace tflite

#endif  // TENSORFLOW_LITE_KERNELS_INTERNAL_REFERENCE_INTEGER_OPS_STAR_FULLY_CONNECTED_H_
```

Chapter 8

Conclusions

The SoC described in Chapter 6.4 was synthesized with Vivado and loaded on a profpga-xc7v2000t FPGA. Then the software was compiled with the TFLM libraries, customized to include the accelerated kernel, and loaded in the memory of the FPGA. Finally, it was possible to test together the overall hardware-software ecosystem. I chose, as reference models to assess the accelerator performances, two additional version of TFLM: the legacy version publicly distributed in [10] and a modified version that handles data with ST paradigm. The execution machine cycles were recorded for each configuration to have a metric of the speedup introduced.

Known Issues

The results of computation of the GEMM Accelerated kernel were incorrect. On one hand, the hardware system has been verified to assess the validity of the interface of the Ibex core with the developed accelerator. Therefore, the issue is not hardware related. Moreover, the output of the kernel is always the same stimulating the network with different inputs, therefore it must be a write-back problem to the output tensor. The same behavior was detected using a function that mock the accelerator working function. This does not compromise the integrity and reliability of the time measures: it was verified the flow of code and the effective operation of the accelerator using the monitor registers, everything go as intended except for the output values. Therefore, the results obtained can be fairly compared with the references.

Results

The three version under test were subjected to 15 inference of the MLPerf Tiny Anomaly Detection network [7][6], and the performances resulting from these runs

were averaged to obtain a comparison value of machine cycle that characterize the version. *GEMM ACCELERATED* was found to save 54.8% of cycles with the configuration INT4 compared to *STAR_EMULATE*, more than 40% compared to both *STAR_EMULATE* and *STANDARD_TFLM* in INT8 configuration and 41% respect *STAR_EMULATE* in INT16.

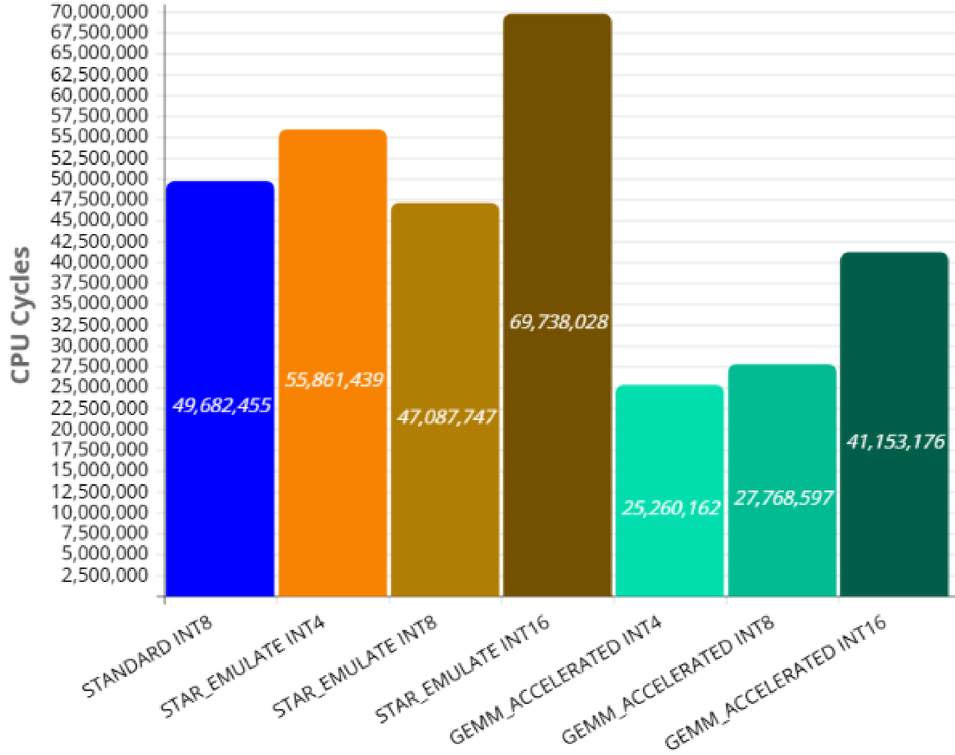

Figure 8.1: Machine Cycles of GEMM Accelerated TFLM kernel compared to the legacy and STAR emulation one

It is necessary to point out that the performances of the solution can be further improved computing, when possible, inference-constant operation offline. As an example the MAC of input offsets and weights can be performed a single time before all the execution of the network. The choice of keeping these operations inside the accelerated kernel was dictated by the comparability of results, otherwise a smart choice would have been to anticipate them in the initialization or preparation phase of the framework. On top of that, the gap of performances between the different versions would have been much wider due to the fact that the acceleration of the solution is focused only in the weight-input MAC.

Future work

The first nice to have future work may be fixing the existing issue presented of the accelerated kernel.
It's necessary further time to investigate the issue in more depth, focusing on the kernel developed and finding the write-back issue. This would help enhancing the legacy left by my predecessor in this research branch.
Another feasible work would be to extend the software compatibility of this GEMM accelerator to other applications to further assess the performances in a much more comprehensive way.
A last possible development that could be done is a full STAR-based TPU that would help to limit some issues related to large parallelism needed to preserve accuracy of models. With a much more independent ecosystem, STAR paradigm could really flourish. Obviously, my assertion should be fact checked and I would be glad to know the results.

List of Figures

1.1 Fully Connected Layer [3] 2

2.1 TFLM Model Import [8] 8
2.2 TFLM Structure [8] 9

3.1 Accelerator PLM [14] 22

4.1 Pipeline-Level Accelerator example : STAR-based Ibex core [15] . . 24
4.2 Cache-level coherency example structure[16] 25
4.3 I/O-Level Accelerator example : Google TPU [17] 26
4.4 GEMM Accelerator Block Diagram 27
4.5 GEMM Accelerator Memory Interface Block Diagram 28
4.6 GEMM Accelerator Computational Unit Block Diagram 31
4.7 GEMM Accelerator PLM Block Diagram 32

6.1 ESP Design Flow [21] 45
6.2 ESP Tiles interaction with NoC [21] 47
6.3 Results of High-Level Simulation of the GEMM Accelerator 51
6.4 RTL Simulation of the GEMM Accelerator Standalone 52
6.5 ESP Design Flow with SoC Configuration GUI [21] 53
6.6 ESP Address Space Definition [23] 54
6.7 Chosen SoC Configuration to test the accelerator in a real environment 55
6.8 RTL simulation of the SoC hosting the GEMM Accelerator 56

8.1 Machine Cycles of GEMM Accelerated TFLM kernel compared to the legacy and STAR emulation one 70

List of Source Codes

2.1 Fully Connected Stock Kernel Loops[10] 13
2.2 TFLM Multiplication Integer Quantization[10] 14
2.3 TFLM Clamping Integer Quantization[10] 15
5.1 Accelerator Memory Interface 36
5.2 Accelerator Memory Interface Directives 36
5.3 Accelerator Memory Mapped Custom Registers 36
5.4 Accelerator Input Loading Phase 37
5.5 Accelerator Weight Loading Phase 38
5.6 Accelerator Output Storing Phase 38
5.7 Accelerator Multiplier Topology Description 39
5.8 Accelerator Core Description 40
5.9 Accelerator PLM Data Type 42
5.10 Accelerator PLM sizes 42
5.11 Accelerator Input PLM Ports toward Memory Interface 43
5.12 Accelerator Weight PLM Ports toward Memory Interface 43
5.13 Accelerator Output PLM Port toward Memory Interface 43
5.14 Accelerator PLM directives 44
7.1 Accelerator Configuration Registers Allocation 60
7.2 Accelerator Hardware Call Function 60
7.3 Accelerator Tiling Function 62
7.4 ST Accelerator Offset Function 63
7.5 STAR Biasing and Quantization Function Registers 64
7.6 STAR Casting Function 65
7.7 Fully Connected Accelerated Kernel Function 66

Bibliography

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, ser. Pearson series in artificial intelligence. Pearson, 2020. [Online]. Available: https://www.pearson.com/en-us/subject-catalog/p/artificial-intelligence-a-modern-approach/P200000003500/9780134610993

[2] C. Aggarwal, *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing, 2023. [Online]. Available: https://doi.org/10.1007/978-3-031-29642-0

[3] A. Amidi and S. Amidi. [Online]. Available: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

[4] S. Wang and P. Kanwar, "Bfloat16: The secret to high performance on cloud tpus," 2019. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

[5] M. Nagel, M. Fournarakis, R. Ali Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *CoRR*, vol. abs/2106.08295, 2021. [Online]. Available: https://arxiv.org/abs/2106.08295

[6] R. Tanabe, H. Purohit, K. Dohi, T. Endo, Y. Nikaido, T. Nakamura, and Y. Kawaguchi, "MIMII DUE: sound dataset for malfunctioning industrial machine investigation and inspection with domain shifts due to changes in operational and environmental conditions," *CoRR*, vol. abs/2105.02702, 2021. [Online]. Available: https://arxiv.org/abs/2105.02702

[7] T. Nakamura, Y. Nikaido, and Y. Kawaguchi. [Online]. Available: https://github.com/mlcommons/tiny/blob/master/benchmark/training/anomaly_detection/keras_model.py

[8] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, "Tensorflow lite micro: Embedded machine learning on tinyml systems," *CoRR*, vol. abs/2010.08678, 2020. [Online]. Available: https://arxiv.org/abs/2010.08678

[9] [Online]. Available: https://github.com/google/flatbuffers

[10] [Online]. Available: https://github.com/tensorflow/tflite-micro

[11] [Online]. Available: https://ai.google.dev/edge/litert/models/ops_compatibility

[12] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *CoRR*, vol. abs/1712.05877, 2017. [Online]. Available: http://arxiv.org/abs/1712.05877

[13] Y. S. Shao and D. Brooks, *Research Infrastructures for Hardware Accelerators*. Springer International Publishing, 2016. [Online]. Available: https://doi.org/10.1007/978-3-031-01750-6

[14] E. G. Cota, P. Mantovani, and L. P. Carloni, "Exploiting private local memories to reduce the opportunity cost of accelerator integration," *Proceedings of the 2016 International Conference on Supercomputing*, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926258

[15] E. Manca, L. Urbinati, and M. R. Casu, "An end-to-end flow to deploy and accelerate tinyml mixed-precision models on risc-v mcus." [Online]. Available: 10.36227/techrxiv.173161032.20267860/v1

[16] D. Giri, P. Mantovani, and L. P. Carloni, "Accelerators and coherence: An soc perspective," *IEEE Micro*, vol. 38, pp. 36–45, 2018. [Online]. Available: https://doi.org/10.1109/MM.2018.2877288

[17] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017. [Online]. Available: http://arxiv.org/abs/1704.04760

[18] L. Urbinati and M. R. Casu, "A reconfigurable multiplier/dot-product unit for precision-scalable deep learning applications," in *Proceedings of SIE 2022*, G. Cocorullo, F. Crupi, and E. Limiti, Eds. Springer Nature Switzerland, 2023, pp. 9–14.

[19] L. Urbinati, "Accelerating quantized dnns with dedicated hardware accelerators and risc-v processors using precision-scalable multipliers," Ph.D. dissertation, Politecnico di Torino, 2024. [Online]. Available: https://hdl.handle.net/11583/2990842

[20] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., ser. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2017. [Online]. Available:

https://educate.elsevier.com/book/details/9780128119051

[21] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, and L. P. Carloni, “Agile soc development with open esp,” in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD ’20. Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3400302.3415753

[22] D. G. Bailey, “The advantages and limitations of high level synthesis for fpga based image processing,” in *Proceedings of the 9th International Conference on Distributed Smart Cameras*, ser. ICDSC ’15. Association for Computing Machinery, 2015, p. 134–139. [Online]. Available: https://doi.org/10.1145/2789116.2789145

[23] [Online]. Available: https://www.esp.cs.columbia.edu/docs/specs/esp_address_map.pdf