CONVOLVE

Seamless design of smart edge processors

GRANT AGREEMENT NUMBER: 101070374

Deliverable D2.6

# Intermediate Neural network resiliency analysis and AxC optimizations

| | |
|---|---|
| Title of the deliverable | Report on the Roadmap |
| WP contributing to the deliverable | WP 2 |
| Task contributing to the deliverable | Task 2.2 and Task 2.5 |
| Dissemination level | RE - Restricted to a group specified by the consortium |
| Due submission date | 30/04/2024 |
| Actual submission date | |
| Author(s) | Panagiotis Chaidos [ICCS]<br>Georgios Alexandris [ICCS]<br>Sotirios Xydis [ICCS] |
| Internal reviewers | Tobias Grosser [UED]<br>Egbert Jaspers [VIN]<br>Matthijs Zwemer [VIN] |

| Document Version | Date | Change |
|---|---|---|
| V0.1 | 01.04.2024 | ToC |
| V0.2 | 15.04.2024 | Initial document for internal review |
| V0.5 | 22.04.2024 | Experimental data updates, Section 2.4 and Section 3.5 |
| V0.9 | 24.04.2024 | Integration of review comments |
| V1.0 | 26.04.2024 | Final version released |

## Table of Contents

Table of Contents ........ 4
Deliverable Summary ........ 5
1. Introduction........ 6
2.Neural Network Resiliency analysis for NTC........ 7
2.1 Overview ........ 7
2.2 Related Work ........ 7
2.3 NTC for IMC assessment framework model ........ 8
2.3.1 IMC architecture template and Analytical Performance Model ........ 8
2.3.2 Assessing Near Threshold Voltage for IMC ........ 12
2.3.3 Resiliency Analysis........ 18
3. AxC Optimizations ........ 23
3.1 Overview ........ 23
3.2 Related Work ........ 24
3.3.1 Introduction ........ 24
3.3.2 YOLOv6 ........ 25
3.3.3 Approximate Multipliers ........ 26
3.4 Experimental Setup........ 26
3.4.1 Introduction ........ 26
3.4.2 CGRA Integration........ 26
3.4.3 High Level Integration ........ 27
3.5 Evaluation ........ 28
3.5.1 Introduction ........ 28
3.5.2 Configurations........ 29
3.5.3 Results ........ 29
3.6 Future Work........ 32
4 Conclusions........ 32
References........ 33

## Deliverable Summary

In this deliverable, we present the current (intermediate) status of the CONVOLVE methodologies and early assessment, exploration and optimization tools for the analysis of the error resiliency vs. energy gain trade-offs of the Neural Networks (NNs) with respect to the arithmetic approximation to be employed in an approximate Coarse Grained Reconfigurable Array (CGRA) accelerator and the timing sensitivities of near-threshold voltage operation. An early assessment design framework is presented to enable analysis of accuracy vs. energy trade-offs for characterizing the NN resiliency with respect to timing induced errors due to near-threshold voltage operation. Regarding to arithmetic approximation, we provide a systematic methodology and the corresponding automation tool-flow to explore HW-aware fine-grained arithmetic error distributions across the NN in a layer-wise and end-to-end approach, thus examining their impact on accuracy and energy. The outcome of this activity will be a map for which NN operation its error resiliency and how this resiliency could be achieved optimizing the approximate and/or near-threshold voltage computing knobs of CONVOLVE System-on-Chip (SoC). More specifically, in this deliverable we present the following contributions:

- Near Threshold Computing (NTC) energy modeling for In-Memory-Computing (IMC) accelerators with NN workloads. We extend ZigZag-IMC to estimate energy consumption under component specific NTC configurations.
- Fault injection of IMC accelerators that simulate NTC errors due to variability and critical path timing failures.
- Accuracy based resiliency analysis of approximate multiplication components in YOLOv6 use case.
- Improvements in approximate component testing framework and CGRA integration.

## 1. Introduction

In recent years Neural Network implementations tend to expand in complexity, in response to the growing field of demanding applications. With the growing complexity however, resource needs and energy consumption skyrocket as low energy techniques are becoming more and more of a requirement to platforms that handle these. To that extent, near-threshold computing (NTC) and approximate computing techniques have garnered significant attention in recent years due to their potential to mitigate these increasing energy demands of power-hungry neural network implementations. Both approaches take advantage of the inherent resiliency of neural networks to errors by trading accuracy for energy consumption. Errors can appear in NTC through bit flips and timing faults and in approximate computing through reduction of precision of operations. This warrants the possibility for exploration of tradeoffs between errors caused by the techniques and energy consumption.

NTC entails lowering of the supply voltage of the circuit at close to the threshold voltage of transistors, where they exhibit significantly lower energy consumption compared to standard operation. When considering In-Memory-Compute (IMC) accelerators, we can profile and locate the components that cause the largest percentage of energy consumption or errors, taking into account their effects in our analysis. To address these considerations, we extend ZigZag-IMC, a framework for mapping Deep Neural Networks (DNN) to IMC accelerators, so that it models accelerators in NTC regime and produces information about optimal mapping and energy consumption specifically for these. Additionally, we include leakage energy in the NTC model and tweak the cost model to effectively calculate the dynamic and leakage energy consumption separately for the multiplier array in the Digital IMC accelerators. Finally, we utilize PyTorchFI, a real-time DNN perturbation tool, in order to model errors in IMCs, taking into account cell array variability and adder tree timing errors introduced from supply voltage scaling.

In Section 3, we show Approximate Computing (AxC) and the impact of those types of operations in achieving great Power and Area gains by using hardware pruning techniques, having a certain accuracy drop. For this reason we continue our approximate arithmetic components research by evaluating two state-of-the-art [12][13] approximate multipliers in the detection stage of the YOLOv6 NN. Yolov6 model includes image processing and convolutional layers which are common fields that hardware approximation can adapt [15]. The techniques we are utilizing to make this evaluation include a high level PyTorch implementation of a behavioral simulation of those components managing to achieve a benchmarking environment that performs the same inference stage as the software implemented one, providing the outputs that analyze the **accuracy-based resiliency** of the chosen components. Finally, we are also providing an integration scheme of those components to the Blocks [14] CGRA environment.

For the DNN resiliency analysis, we work on integrating fault injection and extending ZigZag-IMC, a tool created in WP6 that targets IMC architectures of WP2, for modeling NTC. In terms of the approximate arithmetic CGRA components, this deliverable integrates implementations mentioned in WP1 for testing and evaluation as outputs of previous research tracks mentioned in deliverables of the analysis handled in Task 2.2 of WP2 and after our evaluation, the integration will continue with ZigZag experimentation and CGRA communication with the SNAX core, with outputs and methods produced in WP6.

## 2.Neural Network Resiliency analysis for NTC

### 2.1 Overview

Since DNNs can tailor gradual decreases in accuracy with exposure to error instead of catastrophic failure of results, approaches that exchange error for benefits in performance or energy are worth looking into. NTC is one that exploits circuit components operating close to the threshold voltage and thus achieving significant energy savings while introducing possible error through timing faults and variability.

In addition to improving DNN execution, utilizing IMC architectures for DNN acceleration helps minimize data movement operations and latency from them. With these in mind, we are developing an energy-performance model for IMC accelerators operating in NTC on top of the ZigZag-IMC framework. The complete framework calculates the consumption and performance of the best mapping for a DNN taking into consideration NTC parameters. Finally, we model variability errors in the cell array and timing faults in the adder tree using the PyTorchFI framework and extract the error resiliency of the network.

### 2.2 Related Work

In the work GreenTPU [3], the focus is on fault prevention for Tensor Processing Units (TPUs) operating at NTC. However, they develop a monitor/predictor that reads the inputs of an NTC TPU looking for error causing sequences, at which point the TPU is set to Super Threshold Computing (STC) to prevent errors. They provide an analysis of vulnerable paths and probability of error/fault depending on the delay. Other works focus specifically on DNNs [5] and evaluate resiliency of networks for errors/faults caused in voltage underscaling. They explore possibilities of increasing tolerance to voltage underscaling errors within the training phase of a DNN for FPGA platforms. This work is specific for DNN acceleration on IMC [6]; it develops a framework for locating worst case performance of DNNs taking into account non-volatile memory variability errors while focusing on safety critical application. It is shown that even small amounts of variation can prove detrimental for applications with very low error tolerance, while existing methods are proven ineffective or too costly.

In contrast, we provide a framework for analyzing the effects of implementing NTC techniques on IMC accelerators for DNNs, modeling performance and energy while measuring the error resiliency of the network on these techniques.

## 2.3 NTC for IMC assessment framework model

For the assessment of the NTC for IMC we present our methodology, which includes Performance, Energy and Resiliency Analysis. Figure 1 displays the complete methodology for the entire framework. We firstly incorporate our in-house energy-performance model with the use of NTC scaling factors, into the ZigZag-IMC framework. We also add capabilities for leakage energy estimation using characterization tools to extract the values. Once we have the model with our extensions, we are able to produce the optimal energy consumption and scheduling information for DNNs mapped on NTC IMC hardware. By taking into account variability of the IMCs cell array along with the timing analysis of the IMCs adder tree and mapping from the model, we produce NTC operation faults by injecting them into the DNN using the PyTorchFI tool. Depending on the results of the network, we can estimate its resiliency for NTC Errors.

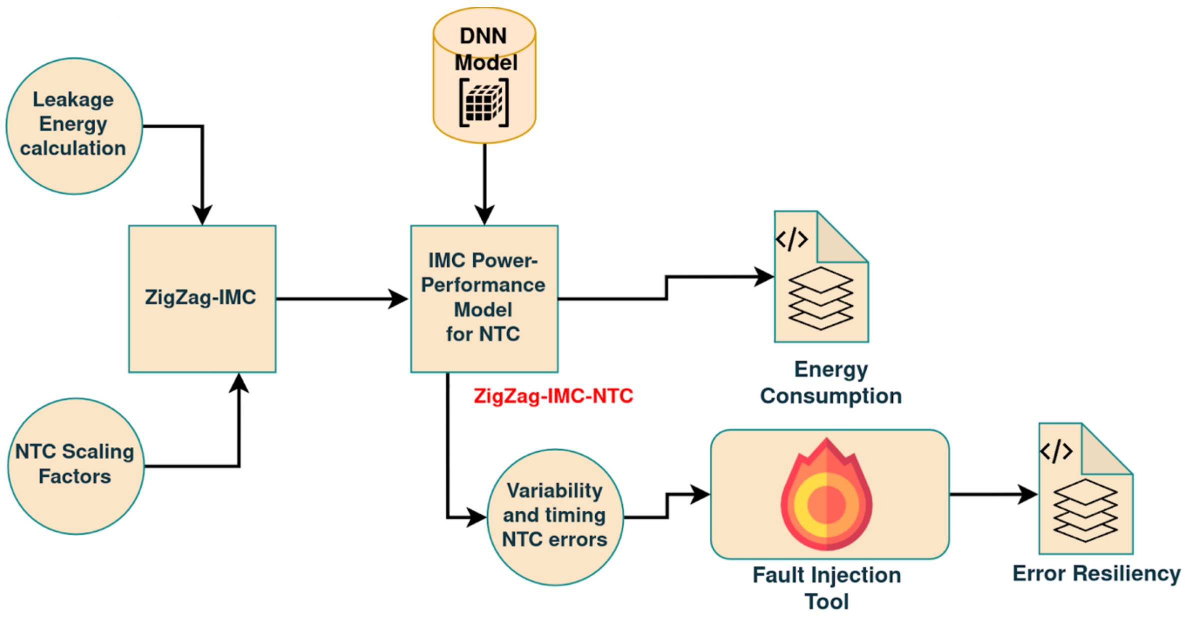


Figure 1. NTC in IMC model general methodology.

### 2.3.1 IMC architecture template and Analytical Performance Model

#### IMC Architecture and Components

The IMC NTC Framework is built on top of ZigZag-IMC and inherits the architecture for Analog and Digital IMC accelerators as shown in [1]. More specifically for each implementation:

- For Analog IMC (AIMC) accelerators, the weights are stored in the cell array with binary values while the inputs are fed into the cell array after passing through a Digital-Analog converter, in analog format. This method of computation alleviates the need for accumulation and thus the components of the AIMC accelerator are the I/O registers, DACs and ADCs, the cell array, and the digital adder tree that is right after the Analog-Digital converter at the output.

- For Digital IMC (DIMC) accelerators, the inputs and the weights stored in the cell array are in binary format and so the main components are the I/O Registers, the cell array, the multipliers and the adder tree/accumulator components. In the analytical cost model of ZigZag, all hardware components are described based on their 1-bit design implementations, e.g. 1-bit memory cell, 1-bit adder etc., which are then used to estimate their total energy consumption w.r.t. their actual sizing.

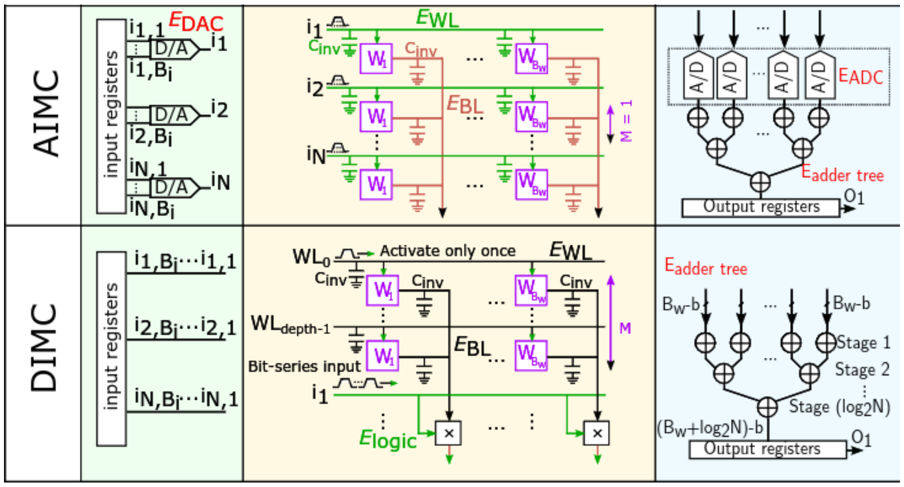


Figure 2. Architecture of IMCs as modeled in ZigZag-IMC[1].

To increase the versatility of the model, we extend it so that each component is considered as a separate voltage island and can be fed with different supply voltages. The user includes in the input hardware files a set of supply voltages for each component along with the standard hardware description. The voltages are passed in the cost model of the tool and are used to calculate the new consumptions of each 1-bit component. After the framework considers a possible mapping, it produces the total number of activations for each computing and memory component, and using the 1-bit consumptions we provide, generates the energy while taking the voltage islands into account.

| Inputs/Parameters | Description |
|---|---|
| Workload | Onnx model of DNN |
| Mapping File | Description of the mapping of operations on the specific hardware |

| | |
|---|---|
| Hardware File | • **Memory Hierarchy:** Memory sizes, attributes and consumption (can use cacti) <br> • **Technology Parameters:** Supply voltage and basic logic gates attributes (area, capacity, delay) <br> • **Processing element parameters:** Input / Weight precisions and Cell Array dimensions, operation details and operation costs (can use cacti) |
| Evaluation Metric | Return Minimal Energy, Minimal Latency or Minimal Delay-Product |

**Table 1. Standard ZigZag Inputs/Parameters**

With this extension we are allowed to explore which components can benefit more from NTC regimes and how each one impacts performance and consumption. In Section 2.3.2 we present results from this analysis for the complete energy-performance model, where we evaluate with the models from the ml_perf_tiny suite.

### Frequency Scaling

Frequency scaling in near-threshold computing is of great importance, as it can have a significant impact on performance, energy efficiency and reliability of results. While significant energy savings are enabled from voltage reduction compared to conventional approaches, components of the circuit can start to fail due to timing and variability concerns, which end up reducing the accuracy of the result. At higher frequencies, these timing faults are even more exaggerated. By adjusting the frequency while having the supply voltage in mind, we can manage to meet timing margins and improve the reliability of the HW.

We introduce the frequency scaling factors used in our framework, that we apply per component, for each of our voltage islands. The new frequency at NTC is calculated as follows, where *Vdd,STC* and *Vdd,NTC* are the operating voltages of the component in STC regime and NTC regime, *Vth* is the threshold voltage, *fSTC* is the frequency of the component in STC regime and *b* is a technology-dependent constant (≈ 1.5):

$$f_{\mathrm{NTC}} = \left(\frac{V_{\mathrm{dd,STC}}}{V_{\mathrm{dd,NTC}}}\right) \times \left(\frac{V_{\mathrm{dd,NTC}} - V_{\mathrm{th}}}{V_{\mathrm{dd,STC}} - V_{\mathrm{th}}}\right)^{b} \times f_{\mathrm{STC}}$$

ZigZag produces a maximum clock in ns that comes from the nominal delay of each computing component in the multiplication-addition-accumulation chain. We apply our frequency scaling factors, depending on the component's NTC voltage supply characteristics specified in the HW file, separately, to differentiate each island.

## Hardware Characterization of Adder Trees

To model adder trees and the errors from the adder trees in the IMC we need information about the critical paths and the longest paths in general, as those are the ones that cause errors when switching to the NTC regime.

We get these paths by synthesizing the adder trees with the Synopsys Design compiler along with the gcslib45nm library and gathering timing information. This process extracts the exact delay of the longest paths, from input bit to output bit of each adder that can cause errors in the NTC regime. Knowing the delays of the paths for each input bit, we determine error probability by using the error distribution given in [3]. Paths with delays close to the clock period are more susceptible to faults compared to shorter paths.

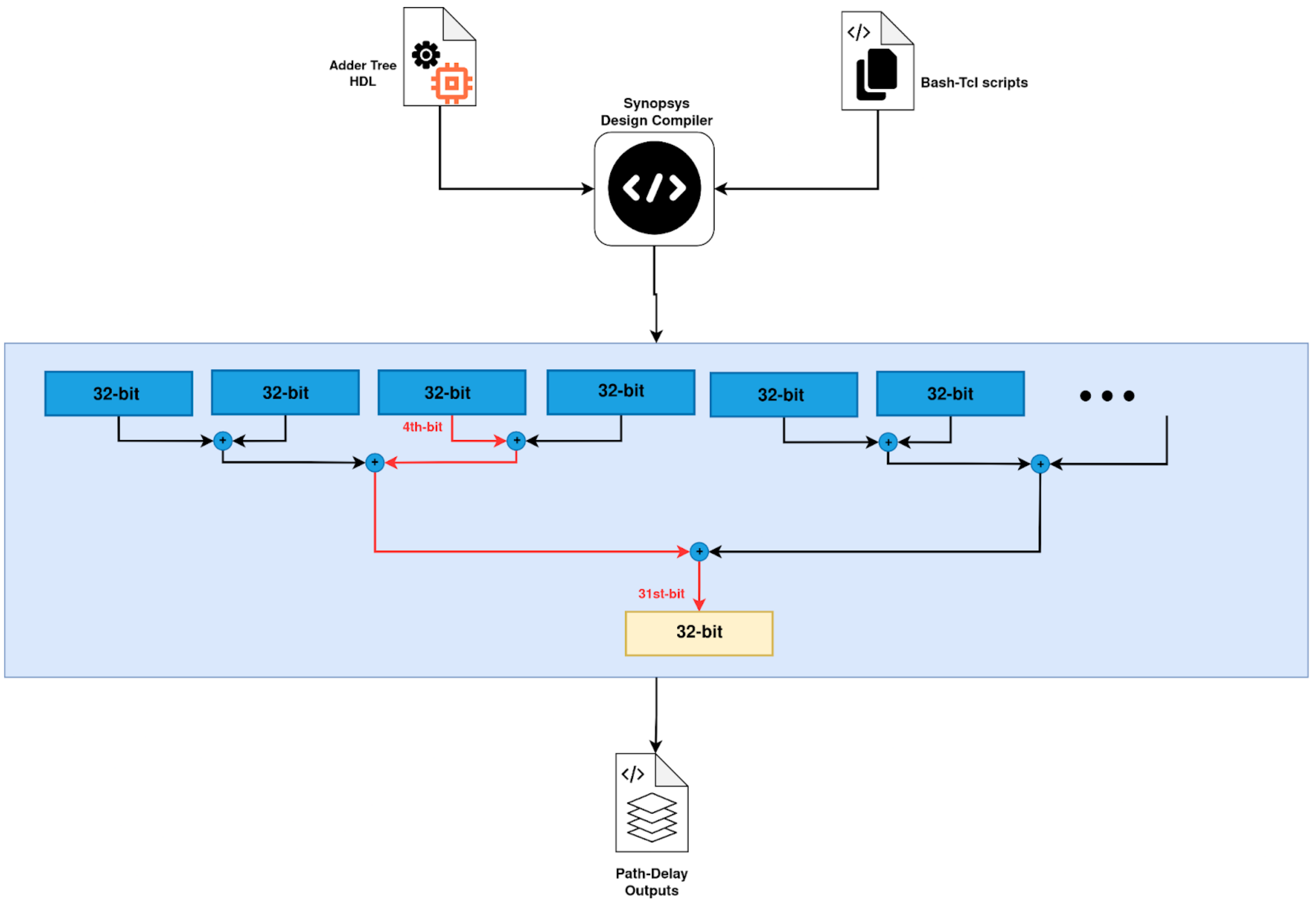


Figure 3. Locating critical paths from all output bits and measuring delays.

In order to implement such errors into PyTorchFI, we utilize its ability to model neuron errors in a network with bit precision. In this way, adder tree errors are modeled as neuron/activation map errors that are added specifically for each tested layer, causing error to the precise bits that belong to the faulty path.

Table 2 displays the critical paths for each output bit along with the delay, for adder trees with 32 inputs and 32-bit precision. Table entries in this format: Input[A][B] -> sum[C] specify that the critical path begins from input A and bit B and finishes at the C-th bit of the output. We observe that most inputs that appear on the table belong in several of the critical paths. The clock used to synthesize the adder trees is 700MHz or 1.42ns period.

| Path | Delay(ns) |
|---|---|
| Input[3][4] -> sum[12-31] | 1.04-1.42 |
| Input[3][5] -> sum[8-11] | 0.95-1.06 |
| Input[2][0] -> sum[1-3, 6-7] | 0.54-0.94 |
| Input[3][2] -> sum[4-5] | 0.78-0.81 |
| Input[20][0] -> sum[0] | 0.37 |

Table 2. Critical paths from all output bits with respective delays.

For Standard Vdd = 0.9V we achieve 700MHz or 1.42ns period which is met by the bit with the worst delay. Using the performance model we described, we can find the new delay paths for our new delays for the critical paths of the adder, after we scale the supply voltage down to Vdd=0.55V. The new frequency for NTC goes down to 50.6MHz and a period of 19.75ns, meaning that by scaling the frequency down to 50.6MHz, we introduce no error from the adder tree. By applying the same model for bit 12 of the sum, we find that the new delay for it is 14.38ns and thus can operate with a maximum frequency of 69.5MHz.

### 2.3.2 Assessing Near Threshold Voltage for IMC

For Energy modeling of IMC accelerators, we work with ZigZag-IMC, a DSE tool for mapping NN workloads on IMC hardware. A HW description along with a NN workload and a spatial mapping file (or mapping hints) that describes how the workload should be mapped to the IMC are given as input to the ZigZag Framework. ZigZag generates possible mappings for the HW accelerator and calculates the number of activations for each memory and computing component. The cost model uses the user-defined capacitances for simple logic components, supply voltage and activations from the mapping to calculate the total energy consumption.

We extend the energy model to include contribution from Leakage Energy alongside Dynamic consumption, since NTC circuits are mostly Leakage dominated. Leakage energy consumption for the computing components and registers of the cell array are extracted through Synopsys Power Compiler characterization. These include the Multipliers, Half Adders, Full Adders, and I/O Registers.

For all computing components we create scaling factors for Dynamic and Leakage Energy that allow for scaling to NTC [2]. The equations for the Scaling Factors and the calculation of the Power consumption are shown in Figure 4. The scaling factors consider the Drain-Induced Barrier Lowering effect which is related to the reduction of the threshold voltage as a function of the drain voltage. Lowering of the supply voltage causes exponential reduction in sub-threshold current. In the equations, Vdd=0.55V for the error free NTC voltage and Vth_stc = 0.5V as in [3].

$$SF_{\mathrm{DP}} = (\frac{V_{\mathrm{dd}}}{V_{\mathrm{dd,STC}}})^2 \times (\frac{f_{\mathrm{NTC}}}{f_{\mathrm{STC}}})$$

$$SF_{\mathrm{LP}} = (\frac{V_{\mathrm{dd}}}{V_{\mathrm{dd,STC}}}) \times \exp\left(\frac{V_{\mathrm{th,STC}} - V_{\mathrm{th}} + DIBL}{n \times V_{\mathrm{thermal}}}\right)$$

$$DIBL = \lambda \times (V_{\mathrm{dd}} - V_{\mathrm{dd,STC}})$$

$$Power_{\mathrm{NTC}} = Dynamic\_power_{\mathrm{NTC}} + Leakage\_power_{\mathrm{NTC}}$$

$$Dynamic\_power_{\mathrm{NTC}} = SF_{\mathrm{DP}} \times Dynamic\_power_{\mathrm{STC}}$$

$$Leakage\_power_{\mathrm{NTC}} = SF_{\mathrm{LP}} \times Leakage\_power_{\mathrm{STC}}$$

Figure 4. Scaling Factors and calculation of Power consumption [2]

We continue by decoupling the multiplier's Dynamic and Leakage Energy for Digital IMC, so that the multiplier energy is calculated as shown below, where Tot refers to Total, Dyn refers to Dynamic, Leak refers to Leakage and SF refers to Scaling Factor.

$$TotMultDynEnergy = TotMultActivations \times 1bitMultDynEnergy \times MultDynSF$$

$$TotMultLeakEnergy = TotInactiveMults \times 1bitMultLeakEnergy \times MultLeakSF$$

For the rest of the computing components, Total energy is calculated at every activation of a component and is equal to:

$$ActivationEnergy = [DynEnergy \times DynSF] + [LeakEnergy \times LeakSF]$$

$$TotEnergy = TotActivations \times ActivationEnergy$$

For the Memory components of the IMC, including the cell array, the dynamic write costs are gathered using cacti, an analytical tool that takes memory parameters as input and returns the memory's performance and energy specifications. Then, the dynamic scaling factor, calculated as explained earlier, is applied on top. We consider leakage Energy of the memories to be approximately ¼ the dynamic energy of cacti and apply our leakage scaling factor on top. Energy calculation for the memories is shown below, where ActivMemCells is the total amount of memory cells activated in a layer, calculated by ZigZag-IMC:

$$TotMemEnergy = ActivMemCells \times [MemDynEnergy \times MemDynSF] \times [MemDynEnergy \times \frac{1}{4} \times MemLeakSF]$$

Figure 5 shows exactly where our implementation changes the base ZigZag-IMC Model. More specifically, in the base HW files we include the specific Vdds for each of the components, allowing configurations where separate components are considered as different voltage islands. This is also where the scaling factors for the cell array and the memories are calculated and the write-cost to each memory is assigned. In the cost model and base architecture hardware files we change the standard calculation of energy per component and instead we use our energy values gathered from synopsys characterization for leakage, along with the standard dynamic values. For each 1-bit component, the energy is calculated using the characterization value as mentioned and the scaling factors for the energies. We modify the cost model to keep track of the not activated multipliers in each layer and assign them the leakage contribution, scaled appropriately, while the activations are multiplied with the dynamic energy per multiplier, also scaled appropriately. Finally, the extended ZigZag-IMC-NTC is able to produce optimal scheduling and energy consumption of a DNN mapped on the HW, taking into account the updated characteristics of our IMC imposed by NTC operation.

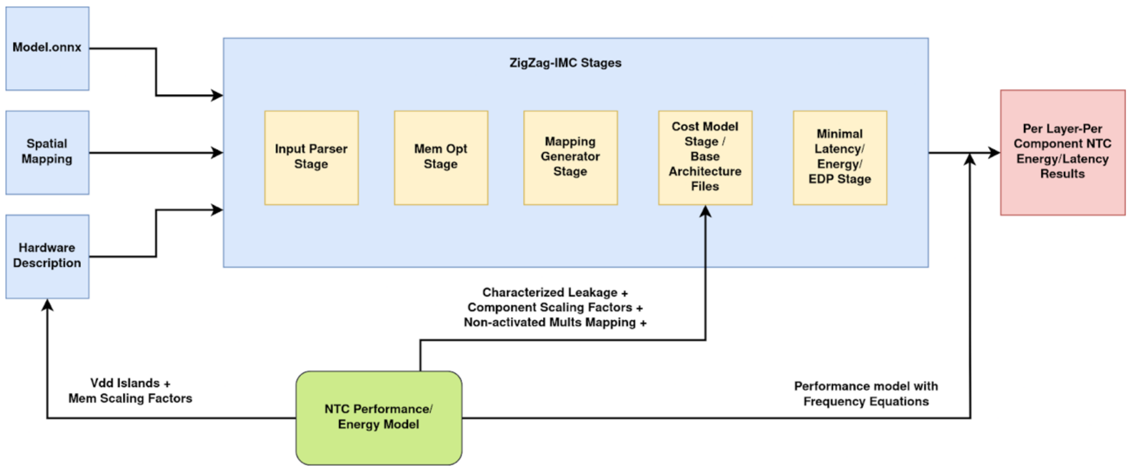

Model.onnx
Spatial Mapping
Hardware Description
ZigZag-IMC Stages
Input Parser Stage
Mem Opt Stage
Mapping Generator Stage
Cost Model Stage / Base Architecture Files
Minimal Latency/ Energy/ EDP Stage
Per Layer-Per Component NTC Energy/Latency Results
Characterized Leakage + Component Scaling Factors + Non-activated Mults Mapping +
Vdd Islands + Mem Scaling Factors
NTC Performance/ Energy Model
Performance model with Frequency Equations

Figure 5. NTC Model modifications to the base ZigZag-IMC Flow

## Demonstration of ZigZag-IMC-NTC with DIMC and AIMC Accelerators

We present Energy results when exploring several STC-NTC configurations on each available component of the IMC. We also test for several Cell-Array sizes and measure how these affect the energy results. We evaluate these configurations on the 4 ml_perf_tiny workloads provided in ZigZag including DS_CNN, Mobilenet_v1, Resnet8, DeepAutoencoder and compare. Figure 6 displays the total flow for the experimental part of ZigZag-IMC-NTC.

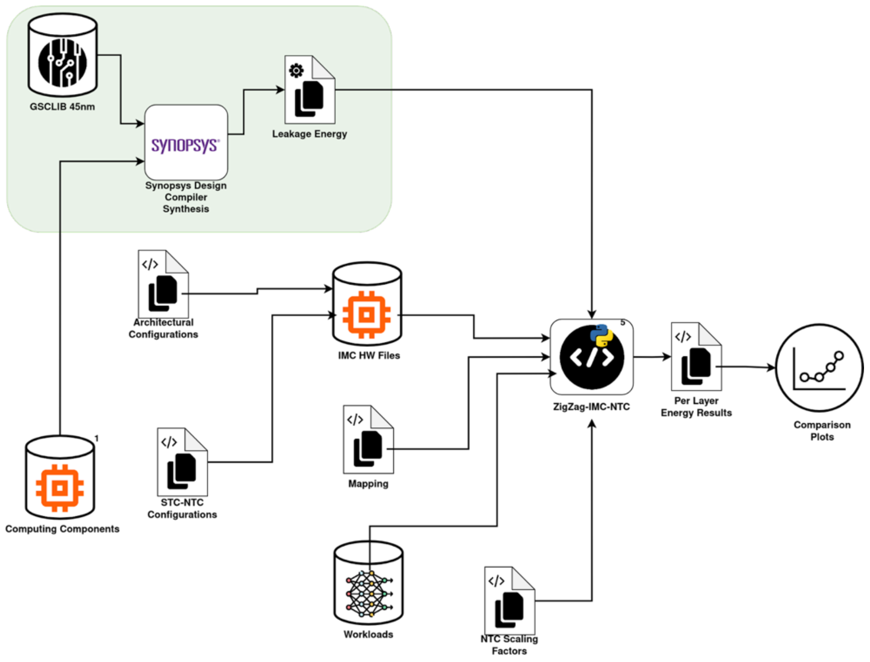
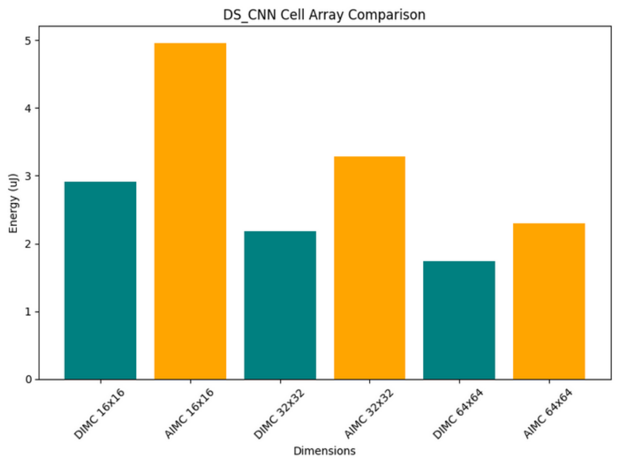
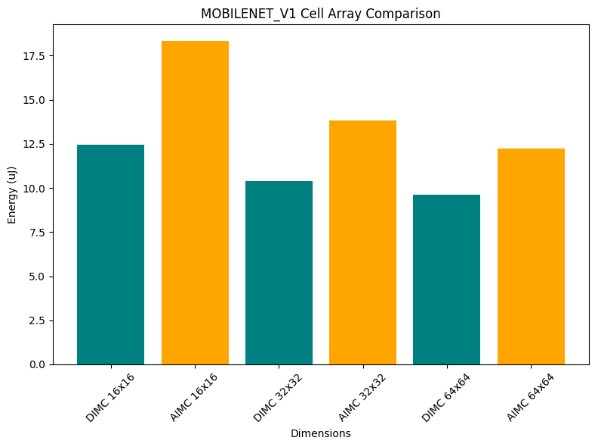


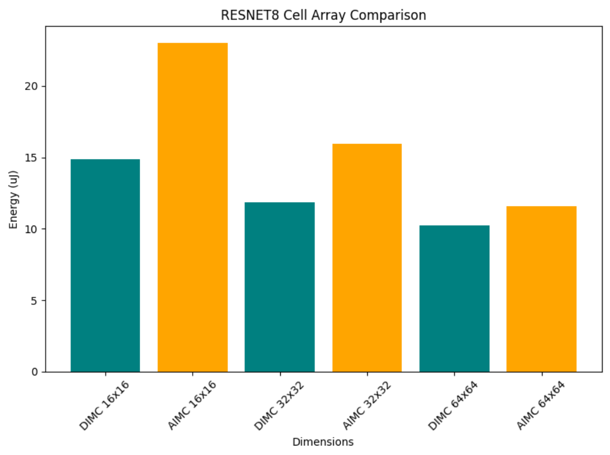
Figure 6. ZigZag-IMC-NTC Experiments Flow.

The following test cases firstly cover the cell arrays and workloads at STC, then introduce NTC operation for each workload and finally compare all 3 variables.

## Test Case 1: Cell Array and Workload Comparison at STC







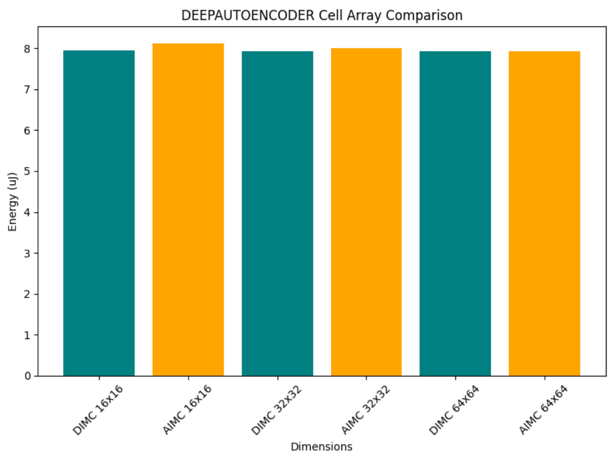
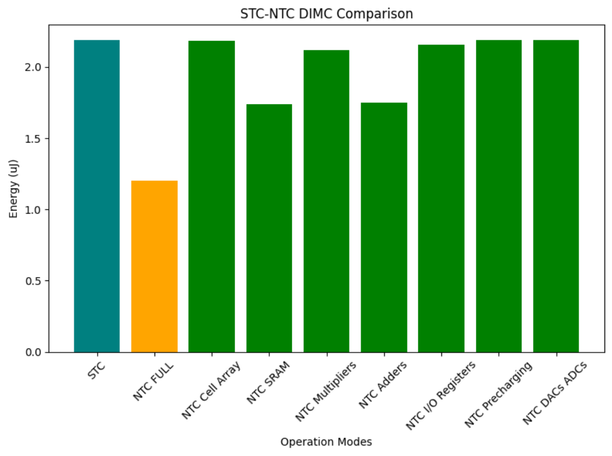


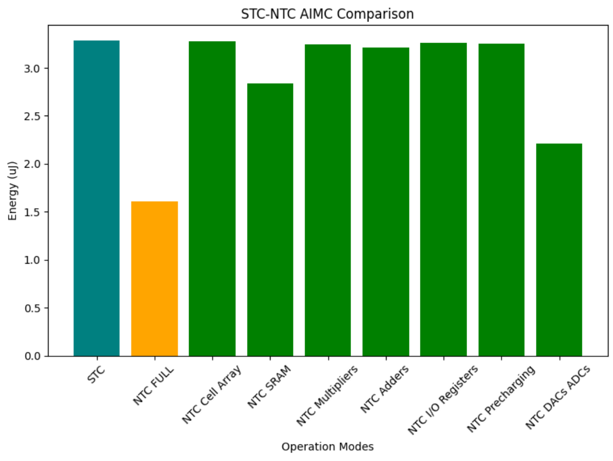
**Figure 7. Test Case 1: Energy comparison of Digital (blue) and Analog (orange) IMC Architectures for varying cell array sizes and workloads DS_CNN, MOBILENET_V1, RESNET8 and DEEPAUTOENCODER**

From the experiments presented in Figure 7 we observe that all models with the exception of the Deep Autoencoder seem to benefit from larger cell arrays up until 64x64, with energy dropping as the array size increases. We continue with the comparison of STC and NTC implementations for different components, for each workload.

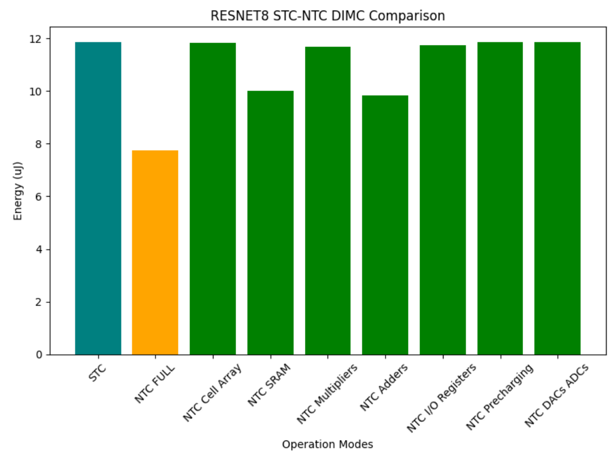
## Test Case 2: Comparison of STC and NTC

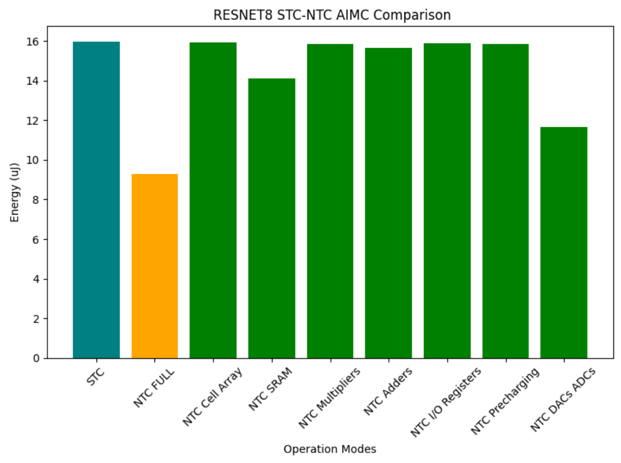
### DS_CNN:



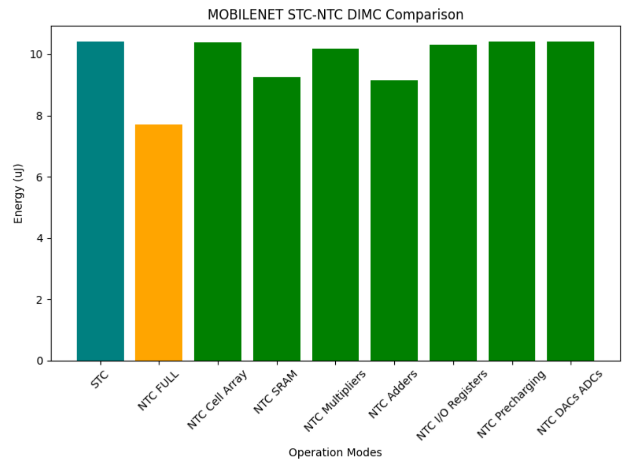


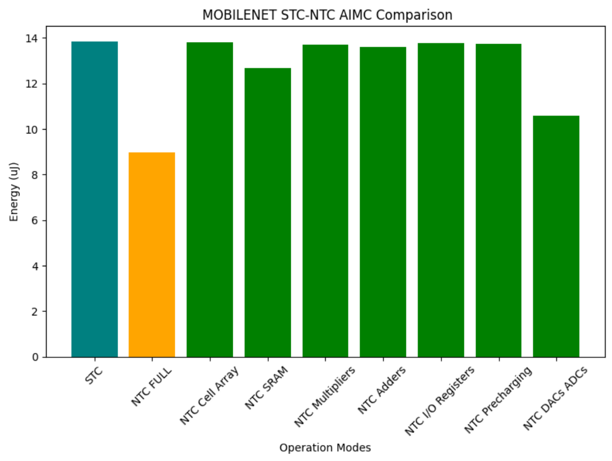
## RESNET8:



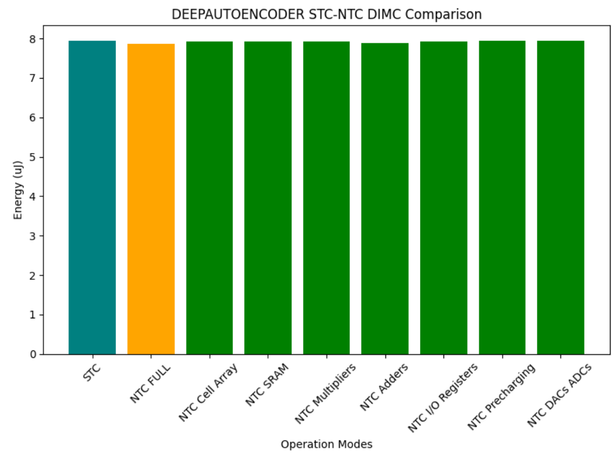


## MOBILENETV1:





## DEEPAUTOENCODER:



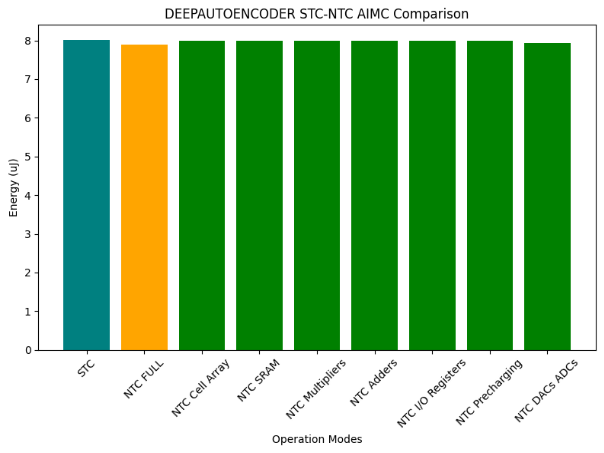


**Figure 8. Test Case 2: Energy comparison of Analog Digital (left) and Analog (right) IMC Architectures for varying voltage supply configurations on the 4 workloads. The configurations are all components on STC (blue), all components on NTC (orange) and partial NTC of a single component (green)**

Through the first comparisons we find whether larger cell arrays end up benefiting the chosen workloads.

In Figure 8, for the first 3 models we see large benefits from NTC operation and from the per component analysis we locate the more demanding parts. For both IMC architectures the SRAM memory always has high utilization and consumption. For the computing components, specifically, Digital IMCs see very large consumption from adder trees while Analog IMCs from the DAC-ADC converters.

Finally for the Deep Autoencoder, we discover from the output files that more than 90% of the consumption in most layers comes from DRAM accesses and so other computation or SRAM gains from NTC are barely observable.

### Test Case 3: Final Workloads and Cell Array Comparisons at STC-NTC

**Workload and Cell Array Comparisons:**

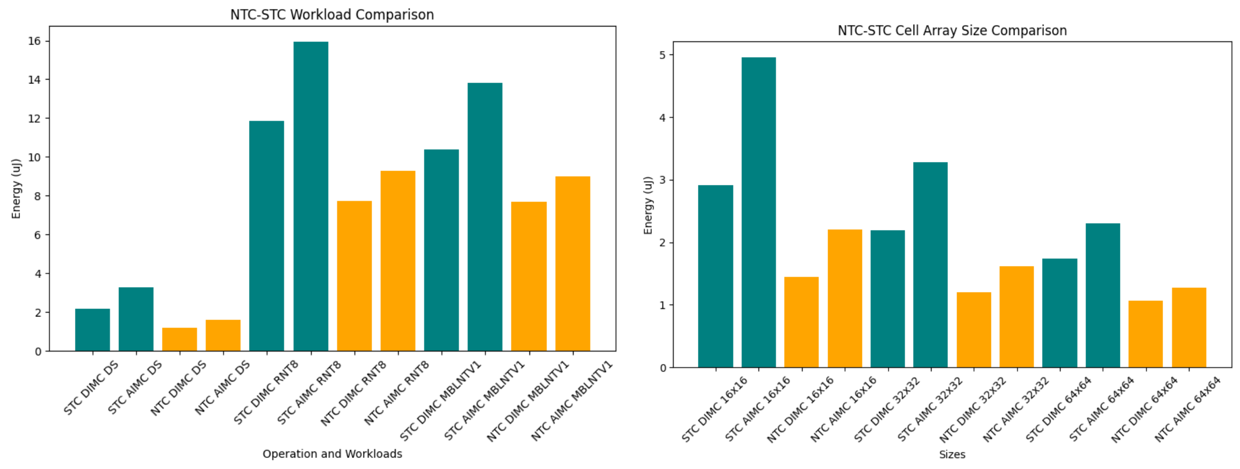


**Figure 9. Test Case 3: Energy comparison of Workloads (left) and IMC Architectures for varying cell array sizes (right) for DIMC and AIMC under full STC (blue) and full NTC (orange) regime**

In all cases presented in Figure 9 NTC designs consume less energy than their STC counterparts, despite any architectural or workload changes.

## 2.3.3 Resiliency Analysis

In order to measure network resiliency, we are required to monitor accuracy oscillations depending on the errors that each NTC configuration and design choice introduces. Employing voltage scaling of different intensities and across different components affects accuracy differently, inserting different types of errors. These effects can be modeled by inserting the knowledge for faulty paths and errors from variability distributions into a fault injection framework and comparing the results with the baseline. To achieve this, we use PyTorchFI, a runtime fault injection tool for DNNs designed for the PyTorch framework that is able to execute a faulty DNN and produce the new output.

PyTorchFI allows for perturbation on a network's weights or neurons in order to measure resiliency against errors. We use PyTorchFI to model errors caused while in NTC operation, due to the variability of the Cell array or failing path delays of the adder tree. Figure 10 shows vaguely where these 2 types of errors are inserted in the PyTorchFI framework.

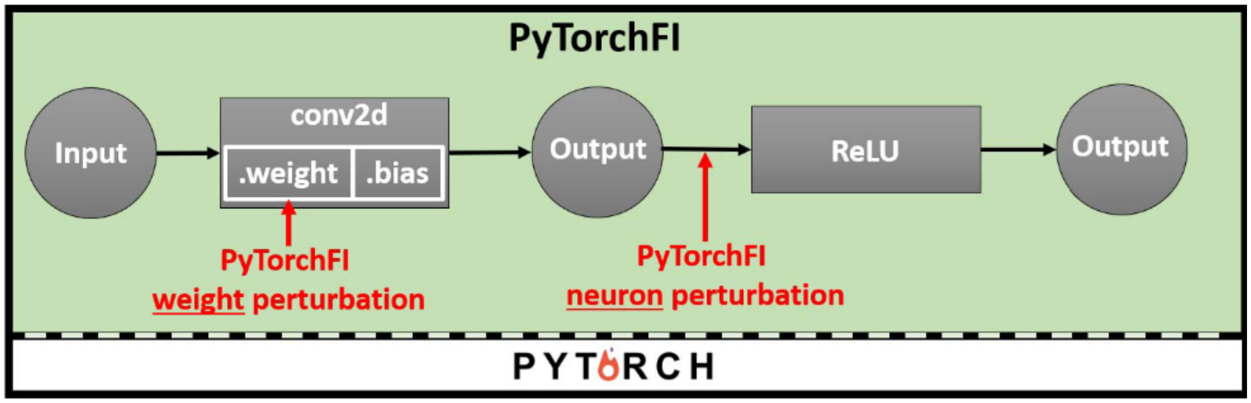


Figure 10. Perturbation types of PytorchFI and insertion points [4]

## Modeling NTC faults through weight perturbation

In the previous section we explored the energy gains from scaling the cell array's, SRAM's and adder tree's voltage, using the per-component scaling functionality from our model. However, when aggressively dropping the voltage in these components of IMC accelerators, aside from energy consumption benefits, bit-flips and erroneous calculations can occur within certain cells of the IMC's cell array. This happens due to the variability that naturally occurs from the imperfect manufacturing process, causing certain parts of the logic to be more susceptible to such NTC errors. The same is true for the SRAM memory when scaled and of course the timing failures that are introduced from scaling the adder tree.

We model these errors in the PyTorchFI framework with the intent to measure the exact impact on the network's resiliency. These errors can be modeled with the use of perturbations in the network's weights, as supported by the PyTorchFI framework. Flipping the bits in a weight of a convolution layer simulates a cell array failure/bit flip of the cell that stores the weight, since weights are loaded in the cell array where the computation takes place. Using different distributions, weight perturbations can simulate an adder tree path failure as well as an SRAM bit flip. With PytorchFI, we can run through the weights matrix and place an error in each of the weights' position and bits according to the chosen distributions. After placing the errors we can run the inference, receive the output value, check whether the computation is faulty and gather information on resiliency.

We can choose a variety of distributions with which we approach the injection process, thus modeling the variability and timing effects differently. For this analysis we test for uniform distributions with varying levels of intensity in order to simulate less and more aggressive voltage scaling.

We evaluate resiliency on Alexnet, a CNN for image classification with 5 convolutional layers, in which we inject errors in the weight matrices through PyTorchFI. In each case we evaluate with 16 different images, each one for 5 different randomly generated noise for the distributions. Alexnet is a classifier with a single scalar as the output, so with this in mind, each image is chosen to showcase a single object or animal (car, cat, dog, fish). Also, since Alexnet takes as input 224x224x3 tensors, all the images were resized before being fed into the network.

According to [3], paths with delays of 20-30% the clock period have a probability of causing error that is 10% or less, so maintaining a frequency of 90-100MHz means that the error rates stay at or under 10%. With this in mind, we test for many error rates between 0 and 0.1 in order to simulate more and less aggressive scaling.

For each weight in each convolution we produce a value from our distributions and compare it to the selected error rate; if the error rate is bigger than the value, then the weight is perturbed. We explore perturbations with intensity of 5%, 10% and 30% of the weights' value simulating 3 different error distributions from our components; each time a weight is to be perturbed we apply a uniform distribution within the limits of the percentages we mentioned, we add that to the weights current value and after this is done for the whole network we infer all the images and measure accuracy loss.

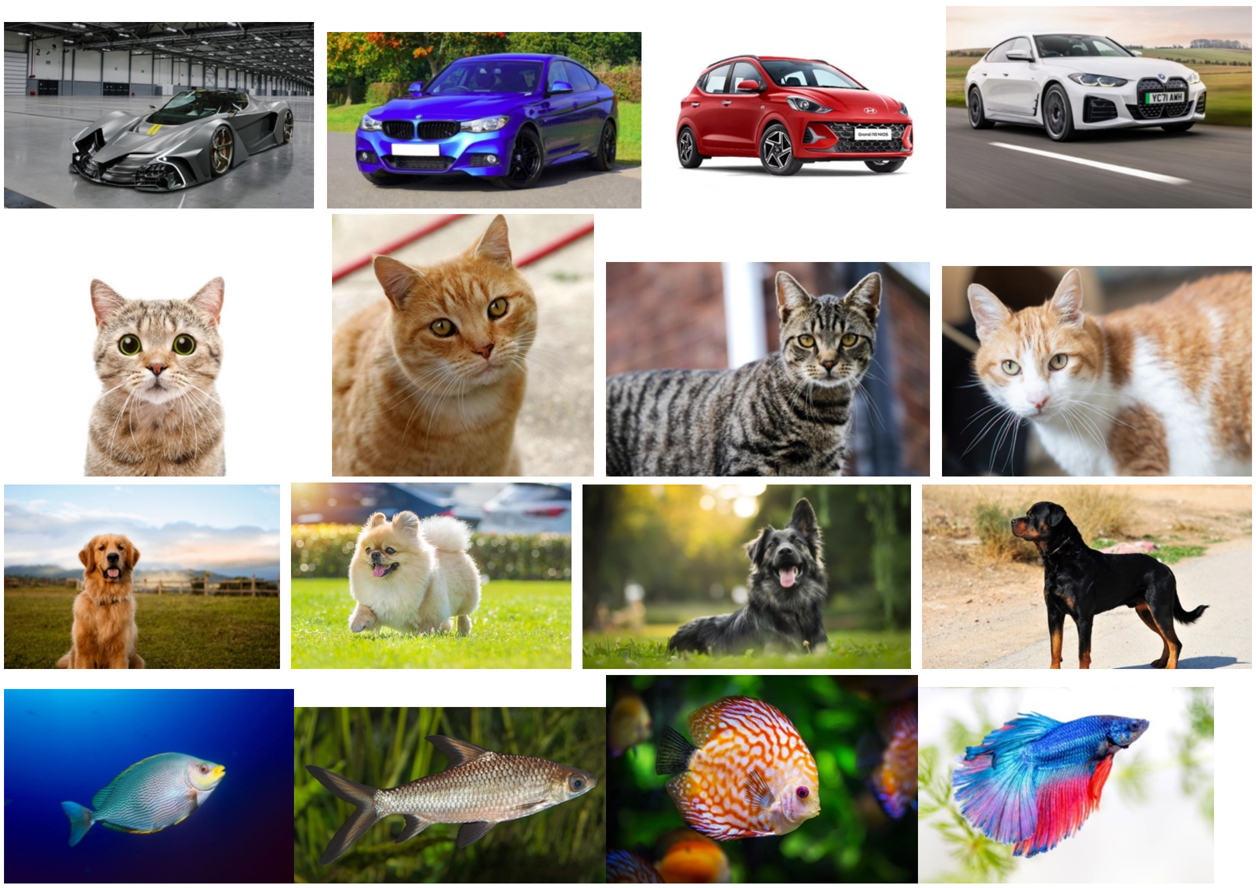


Figure 11. Test images for alexnet weight perturbation

Firstly, Tables 3 and 4 present the accuracy of Alexnet for each of the distributions of each component separately and combined. They display the resiliency of the network when inferring the chosen images with weight fault injection from the chosen distributions, error rates and weight percentage perturbation introduced. Error_p is the error rate with which a weight is perturbed and the percentage of perturbation is the amount up to which it can be changed. Different distribution types model different types of errors of components. Finally, Figure 12 is an accuracy vs error rate scatter plot containing the information of the 2 Tables. Error rate here is the probability with which we perturb a weight from each convolutional layer and Accuracy is compared to the base inference output of the model for our images.

| Error_p | Accuracy of 50% perturbation | Accuracy of 10% perturbation | Accuracy of 5% perturbation |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1.00E-05 | 1 | 1 | 1 |
| 5.00E-05 | 0.9875 | 1 | 1 |
| 1.00E-04 | 0.9875 | 1 | 1 |
| 5.00E-04 | 0.975 | 1 | 1 |
| 1.00E-03 | 0.95 | 1 | 1 |
| 5.00E-03 | 0.875 | 0.9875 | 1 |
| 1.00E-02 | 0.8 | 0.9875 | 1 |
| 5.00E-02 | 0.725 | 0.9375 | 1 |
| 0.1 | 0.475 | 0.825 | 0.9 |

Table 3. Accuracy of Alexnet under weight fault injection from uniform distributions of components seapately

| Error_p | Accuracy of combined distributions |
|---|---|
| 0 | 1 |
| 1.00E-05 | 1 |
| 5.00E-05 | 0.985 |
| 1.00E-04 | 1 |
| 5.00E-04 | 0.9375 |
| 1.00E-03 | 1 |
| 5.00E-03 | 0.9165 |
| 1.00E-02 | 0.8585 |
| 5.00E-02 | 0.5835 |
| 0.1 | 0.525 |

Table 4. Accuracy of Alexnet under weight fault injection from uniform distributions of components combined

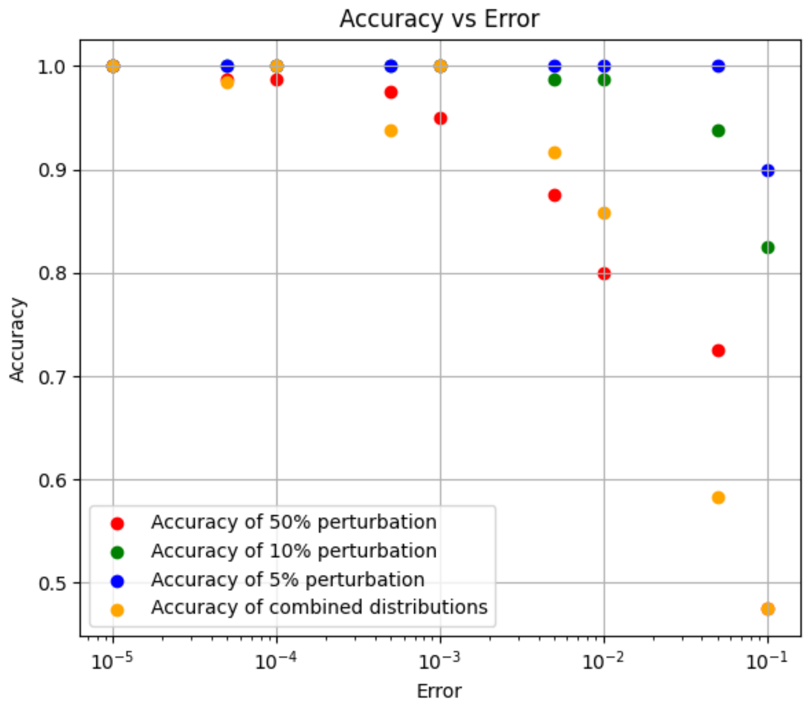


**Figure 12. Accuracy vs Error Rate plot of Alexnet for error distributions of separate and combined components. Error rate is the probability with which we perturb a weight from each convolutional layer and Accuracy is compared to the base inference output of the model**

For the uniform distribution, lowering the perturbation percentage causes significant accuracy gains across all different error rate values. It is clear that the behaviors, meaning the distributions, of the components under NTC is crucial for controlling/retaining accuracy. We see here how dropping the voltage as we explored in Section 2.3.2 introduces error in the DNN and how, depending on the distribution, it scales with intensity.

## 3. AxC Optimizations

### 3.1 Overview

The low-level operations executed in a system can vary but in the use cases that are the most interesting and adaptable in this project's RoadMap are increasing. This increase has a huge impact on the system's power as well as the area [7], meaning the cells that need to be utilized to make this complex and efficient system possible. Along with this burst of those applications and the embedded systems that evaluates them, the hardware optimizations that can prune cells of a hardware operation and/or produce more energy efficient solutions are facing a huge bottleneck (there cannot be more state-of-the-art solutions in that field).

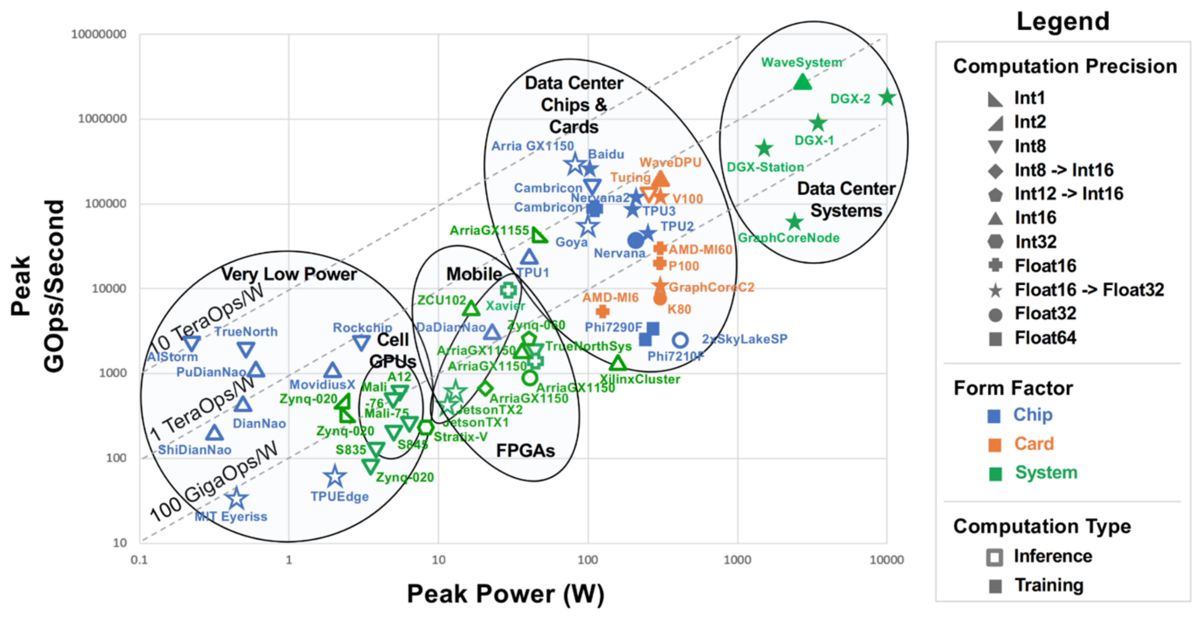


Figure 13: Power-Operations Scheme in different device families.

Figure 13 presents a survey-based analysis [7] of different device families in terms of the πεακ power and the performance during inference and training stages, along with the different datatypes which are used during those processes.

A partial solution to the huge power and area utilization in modern NN applications is introduced by **approximate computing** [8] providing energy efficiency and decreased in area operational units (adders, multipliers, etc.) but with a tradeoff in the accuracy of the output. There are some use cases where those approximations cannot be integrated due to the hard QoS constraints (medical related scenarios) but there are also cases where the accuracy-efficiency trade-off can give very positive results.

The cases where those techniques have great use are the image processing operations (filtering, edge detections, blurring) [10] and the extensions of them, which are the CNN based AI models that are commercially available in the community [11].

The target of this section is to take advantage of the approximate multiplication units tested by our research output during the **Deliverable 2.2** of this project and propose a methodology that can

- Integrate state-of-the-art approximate multipliers[12][13] in the Blocks[14] CGRA environment
- Test the multipliers logic using high level inference (PyTorch)
- Test the resiliency of the output accuracy during YOLOv6 [15] prediction stage inference

## 3.2 Related Work

The related work will mostly present the use cases of the approximate modules and not the actual circuit structure, because the target of this report is the use case evaluation of those components.

Work presented from Semeen Rehman et. Al. [10] Presents a DFS based application of error scalable multipliers and the definition of 19 different approximation and accurate configurations, which then were tested in the JPEG compression application. Those techniques are then compared through the PSNR , Power and Area metrics. The Power and Area were measured with the Synopsis Design Compiler and the actual test were performed via a behavioral model which tested those approximation techniques in C code execution environment.

Another research output presented from H. Jiang et.al. [16] measured state of the art techniques with accurate multipliers of Wallace tree and ArrayM in Image sharpening algorithm using VHDL for the behavioral component and Synopsys Design Compiler and 28nm technology library to fetch Power and Area estimations.

S. Venkatachalam presented in [17] tested also state-of-the-art multiplication techniques based on compressor trees, partial product pruning, ACM and UDM in Gaussian Noise Filtering Algorithms based on Matlab generated behavioral simulation and Synopsys Design Compiler with the TSMC 65nm library for Power Delay product generation.

Finally, in a more recent research approaches [11], Ying Wu et.al. presented an extended evaluation of approximate techniques, implementing the behavioral environment in Verilator and gathering Power and Area data from Synopsys Design Compiler using UMC40 library. The significance of this survey is that the testing of those components was performed on modern Neural Network models inference stage (such as AlexNet, SqueezeNet and MNIST).

## 3.3 Use Cases

### 3.3.1 Introduction

In this chapter will be presented the use cases of the AxC analysis improvements. The use cases include the base of the ViNotion NN use case, which includes the YOLOv6 [15] model with the yolov6s6 implementation weights and the results of the Approximate Multiplication design space exploration that has been delivered to the project (D2.2). This section aims to describe those technologies that will be used for our setup.

### 3.3.2 YOLOv6

YOLOv6 [15] is an advanced object detection NN based algorithm. It builds on the strengths of its predecessors by offering enhanced accuracy and faster processing speeds, making it highly suitable for real-time applications. YOLOv6 incorporates improvements in its architecture, such as more efficient backbone networks and optimized anchor box predictions, which contribute to its increased robustness and precision across a diverse range of object classes. Additionally, it is designed to be more scalable and easier to deploy on various platforms, from high-end servers to edge devices, ensuring wide applicability in fields like autonomous driving, surveillance, and interactive media.

In the CONVOLVE project the YOLOv6 is a part of models being built in the ViNotion use cases. Unfortunately, due to technical difficulties in accessing the exact models of the WP1 use cases, we tried to achieve a close-as-possible approach by utilizing this model.

Another reason than makes this model suitable for our research output is the many and different convolutional layers that are included in the model structure, which gives to our approximate components a lot of space to be run with different input and kernel sizes.

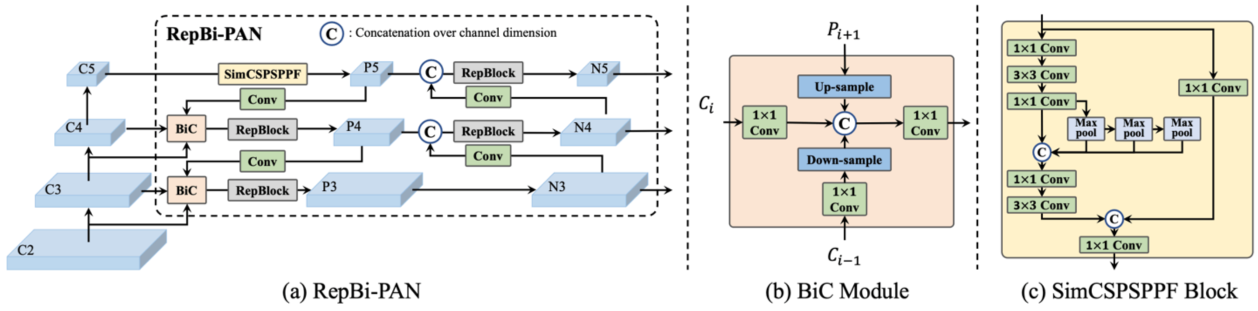


Figure 14: Overvew of YOLOv6.

In Figure 14 an overview of YOLOv6 is presented, as implemented by the developers. (a) The neck of YOLOv6 (N and S are shown). Note for M/L, RepBlocks is replaced with CSPStackRep. (b) The structure of a BiC module. (c) A SimCSPSPPF block.

### 3.3.3 Approximate Multipliers

For this project we used two state-of-the-art implementations of approximate multipliers, the **DRUM** [12] and the **ROUP** [13] (which in the bibliography is mentioned as **APR**).

| Multiplier | Parameters | Description |
|---|---|---|
| ROUP [13] (In the test data is mentioned as **APR**) | p,r | Rounding [18] and Perforation [19] combination only. The ROUP_p_r multiplier cuts the p last significant partial products and uses routing to the $r_i$-bit of the I non-perforated partial product. |
| DRUM [12] | k,n | A DRUM_k_n multiplier introduces the idea that some bits inside the two multiplication factors are not important to be calculated with accuracy. The parameter k defines the number of core-accurate bits to be calculated and the n is the bit width of the input numbers. |

Table 5: Approximate Components

In the Table 5 are presented with the approximate components we chose to include. Those multipliers came from our research output in Deliverable 2.2, as the best-behaving multipliers in Error, Power and Area utilization. In the best solution there also were multipliers from the EvoApprox8b project [20], but due to the very hard reconfigurability they provided we chose to exclude them from further analysis.

## 3.4 Experimental Setup

### 3.4.1 Introduction

This chapter will analyze the methodologies and tooling used in this exploration. The setup includes the integration of the approximate multipliers as an Blocks-CGRA compatible tile, with the proper ISA, IO and peripheral communication described, and the inference setup from which we can calculate the accuracy resiliency of each tile, utilizing behavioral simulation components with a higher level of PyTorch compatibility.

### 3.4.2 CGRA Integration

The approximate multipliers are integrated as an approximate tile into the CGRA, with the same form-factor and ISA implementation as the "vanilla" multiplication tile the Blocks CGRA implemented. The choice of the similar integration came as a conclusion of better compatibility with the implemented infrastructures as well as the adaptation of those tiles from the compilation firmware TUe provides for the CGRA.

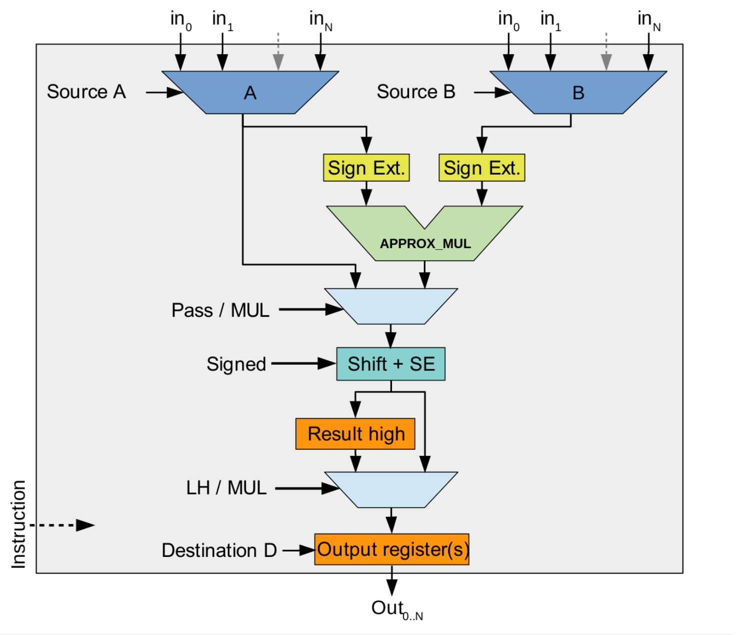


Figure 15: Approximate Multiplication Tile (left)

Figure 15 presents the implementation of an approximate multiplier tile, which came from the same template utilization like the prebuilt one with the change of the core multiplication system (mentioned as **APPROX_MUL** in Fig. 13). Due to this similarity the ISA was the same between those components.

Our research is aiming also for bigger and more runtime configurable tiles (presented in Deliverable 2.3) but are in the early modeling stages so they cannot be yet used for this type of evaluation.

## 3.4.3 High Level Integration

The goal of our first research output (D2.2) was mainly to pick the best approximate techniques among three state-of-the-art projects [12][13][20] and evaluate them having three key criteria which were **Accuracy Drop**, **Power** and **Area** utilizations. For this reason, this analysis consisted of two parts, a power modeling part held by **Synopsys Design Compiler** and provided Power and Area information about our components. The second part was the actual behavioral simulation firmware which handled the **inference of the data** and the **output activation maps** of the DenseNet121 NN.

The analysis of this section primarily focuses on the accuracy in the accuracy-based resiliency of the approximate components meaning that the output should be the impact of accuracy drop during the inference stage in NN models. To achieve that we kept the **second part** of our benchmarking flow and optimized it to fit this scope.

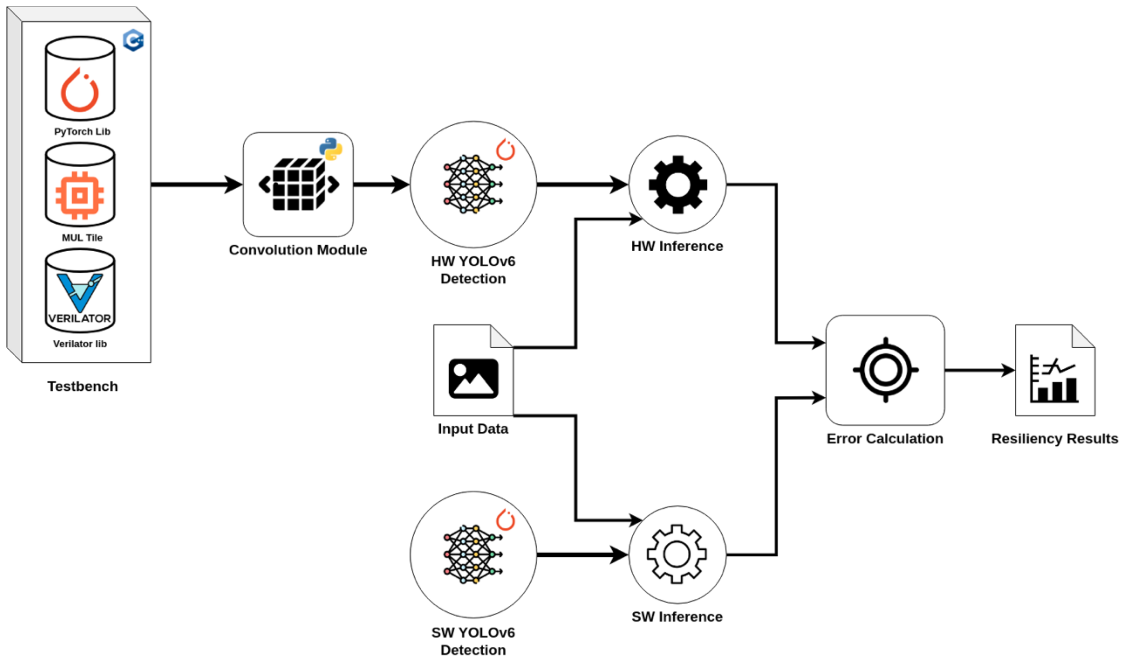


Figure 16: Approximate Tiles Benchmarking Environment

In Figure 16 the benchmarking flow is presented. The Verilog description of the approximate Tiles is being built using Verilator into the object file that is later used in the C++. Using this behavioral object along with the PyTorch shared library (libtorch) we manage to implement a **convolutional layer** that handles all the multiplication processes through our Approximate Tiles.

After this layer (the **Testbench**) is being built, it is integrated via pybind11 in the Python Runtime as a top level PyTorch Convolutional Layer Function. This function is then placed into selected layers of the YOLOv6 NN and the inference can be executed like the provided flow suggests.

Finally, the output activation maps from the behavioral simulation are compared with the preconfigured (SW tested) ones for the error to be calculated.

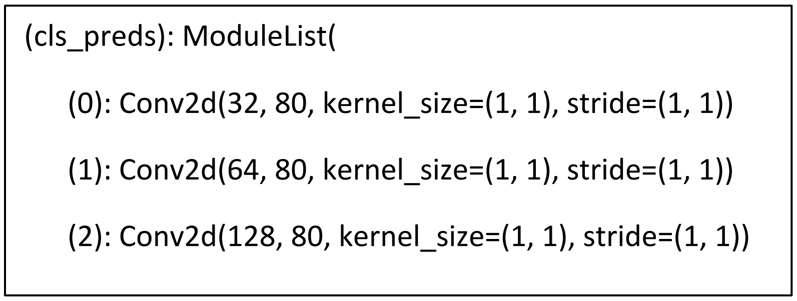
## 3.5 Evaluation

### 3.5.1 Introduction

This section concludes our output with the result production from a close to the end layer of the YOLOv6 NN with approximation multiplication that picked as described in the previous sections. The accuracy drop was measured from the MSE values of the produced feature maps on each layer in combination with the bounding box object recognition produced from the final output layer. That exploration gave an estimation of the accuracy resiliency during different approximation techniques, evaluated in a close-to-the project NN model.

## 3.5.2 Configurations

For this evaluation we focused on the prediction stage of the YOLOv6, and we traced the convolution layers with a great impact to our analysis. After this exploration we concluded that the "**cls_preds**" block in the detection stage of the NN is a great fit to demonstrate the impact of our approximate multiplier components, because with no significant error propagation we can see both the output accuracy drop and the layer error metric. An abstract schematic of this block is present in Figure 17.



```
(cls_preds): ModuleList(
    (0): Conv2d(32, 80, kernel_size=(1, 1), stride=(1, 1))
    (1): Conv2d(64, 80, kernel_size=(1, 1), stride=(1, 1))
    (2): Conv2d(128, 80, kernel_size=(1, 1), stride=(1, 1))
```

Figure 17 :cls_preds Block.

The configurations we tested were the above:

- **0 –** Apply the convolution layer **only on the layer 0**
- **1 –** Apply the convolution layer **only on the layer 1**
- **2 –** Apply the convolution layer **only on the layer 2**
- **All –** Apply the convolution layer **on all the layers (propagate the error)**

When we apply the convolution in **all the layers**, we calculate the error **only on the last output**.

In terms of the hardware configuration, we are using 16x16 bit multiplication with the output of 32-bits per operation. The adding part is handled in the software side.

For the inference we are using the weights from the **yolov6s** which came from training with the **coco** training dataset. The evaluation is performed using five images, three from the provided yolov6 repository and two random from web source.

## 3.5.3 Results

The results presented in Table 5 come from the evaluation of the five images in with DRUM (k=4,5,6,7) and with ROUP (p=1 and r= 10,12). For each multiplier tested, we present the MSE from the output Tensor compared to the software implemented one. This MSE comes as a metric across all the dimensions of the tensors produced to give a general metric of the quality of the output. The final accuracy drop is presented as compressed as possible in the form of the visual object recognition results in Figure 16.

| Image | Config | DRUM4 | DRUM5 | DRUM6 | DRUM7 | APR1_10 | APR1_12 |
|---|---|---|---|---|---|---|---|
| img1 | 0 | 0.4963 | 0.4994 | 7.83 | 0.5 | 0.5 | 0.5 |
| | 1 | 0.4905 | 0.4948 | 5.21 | 0.4953 | 0.4953 | 0.4953 |
| | 2 | 0.1569 | 0.1582 | 4.66 | 0.1587 | 0.1585 | 0.1585 |
| | all | 0.1569 | 0.1582 | 4.66 | 0.1587 | 0.1585 | 0.1585 |
| img2 | 0 | 0.6432 | 0.6506 | 8.06 | 0.6522 | 0.6523 | 0.6523 |
| | 1 | 0.6245 | 0.6290 | 5.08 | 0.6317 | 0.6317 | 0.6317 |
| | 2 | 0.1975 | 0.2006 | 4.49 | 0.2018 | 0.2015 | 0.2015 |
| | all | 0.1975 | 0.2006 | 4.49 | 0.2018 | 0.2015 | 0.2015 |
| img3 | 0 | 0.5365 | 0.5413 | 8.86 | 0.5431 | 0.5430 | 0.5430 |
| | 1 | 0.6131 | 0.6180 | 5.54 | 0.6205 | 0.6207 | 0.6207 |
| | 2 | 0.2611 | 0.2638 | 4.89 | 0.2648 | 0.2646 | 0.2646 |
| | all | 0.2611 | 0.2638 | 4.89 | 0.2648 | 0.2646 | 0.2646 |
| img4 | 0 | 0.5679 | 0.5757 | 8.56 | 0.5779 | 0.5779 | 0.5779 |
| | 1 | 0.5954 | 0.6004 | 5.38 | 0.6043 | 0.6046 | 0.6046 |
| | 2 | 0.5362 | 0.5393 | 6.26 | 0.5409 | 0.5406 | 0.5406 |
| | all | 0.5362 | 0.5393 | 6.26 | 0.5409 | 0.5406 | 0.5406 |
| img5 | 0 | 0.6806 | 0.6842 | 8.74 | 0.6853 | 0.6851 | 0.6851 |
| | 1 | 0.5938 | 0.5975 | 6.33 | 0.5995 | 0.5994 | 0.5994 |
| | 2 | 0.0880 | 0.0897 | 4.97 | 0.0902 | 0.0901 | 0.0901 |
| | all | 0.0880 | 0.0897 | 4.97 | 0.0902 | 0.0901 | 0.0901 |

Table 6 : Evaluation Results

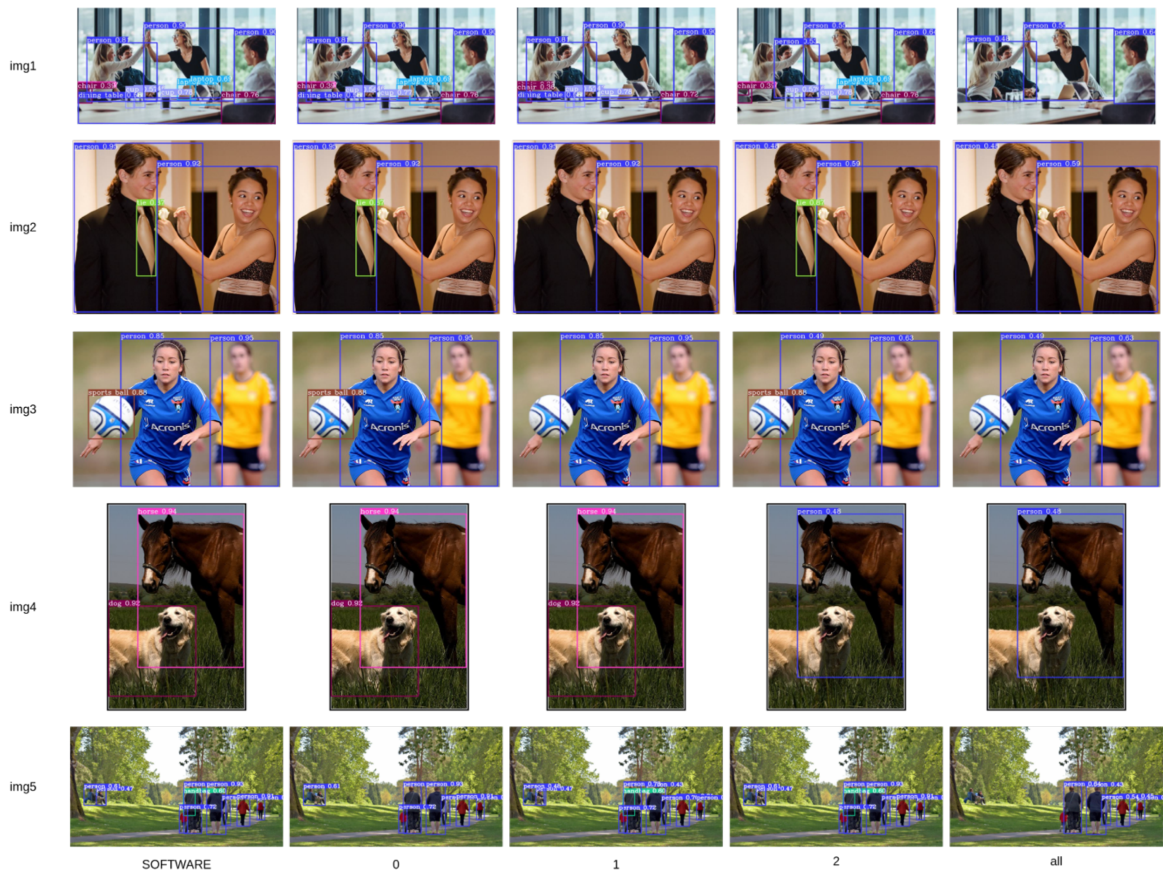


Figure 18 : Visual Approximation Results

In Table 6 all the MSE values from different configurations are presented and in Figure 18 all the different object detection that is produced from the final outputs are also provided. The approximate implementation in Figure 18 is from the ROUP1,10 multiplier, but all the other implementation with the exception of DRUM6 have the same behavior.

From the overall analysis we can see that applying the approximations techniques **in the final layers introduces smaller error**, but when comes to the detection result (Figure 18) we can see that the most information is being kept when approximations are introduced in layers 0 and 2, and if we want to cover the whole testing area we will keep **only the approximation in layer 0**, which is a valid conclusion due to the smaller sizes in that layer. Also, we can see that the best behaving multiplier in those cases is **DRUM4** which also comes with greater energy and area gains. Also, we observe that **DRUM6** performs strangely and produces lot of errors during inference stage.

Finally, we see that in the output layer when the approximation is applied in all the layers of the block, the error propagation stays the same so we can continue evaluating the whole block by applying different approximation per layer to maintain stability with the same factor of accuracy resiliency.

### 3.6 Future Work

In the future we will try to implement multiple approximation units in deferent blocks further in the YOLOv6 NN to have a better exploration of the accuracy drop in a more generalized manner. Due to time limitations, we did not explore the infrastructure of YOLOv6 to the full potential of the object detection case. So, a next step in this will be a more precise exploration in terms of different layers in this model, as also described in the WP1 by the ViNotion partner in CONVOLVE.

Furthermore, with the completion of the CGRA integration which will be also accompanied by the compiler integration then we will propose the best architecture configurations of the Blocks CGRA environment with the proper approximate tiles and the layers in both software and hardware context.

## 4 Conclusions

In conclusion, as neural network applications continue to grow in complexity and demand, innovative techniques such as near-threshold computing (NTC) and approximate computing are proving indispensable in addressing the escalating resource needs and energy consumption. By leveraging the inherent error resilience of neural networks, these approaches offer a viable trade-off between accuracy and energy efficiency. The enhancements to the ZigZag-IMC framework and the integration of PyTorchFI demonstrate our commitment to optimizing the performance of In Memory Compute (IMC) accelerators within these constraints.

Furthermore, our ongoing research into approximate arithmetic components, specifically within the YOLOv6 neural network environment, inspired by the relation with one of the projects use cases provided by the ViNotion partner, highlights our pursuit of significant power and area gains, albeit with some accuracy trade-offs. The evaluation of this methodology was a behavior simulation-based flow based on Verilator and integrated in PyTorch, and tested the multipliers as a component (Tile) of the R-Blocks CGRA accelerator, managing also instead of testing the multiplication impact, also test the component as an integratable module.

## References

[1] Houshmand, P., Sun, J., & Verhelst, M. (2023). Benchmarking and modeling of analog and digital SRAM in-memory computing architectures. arXiv preprint arXiv:2305.18335.

[2] Energy Efficient DNN Inference Through Approximate Near-Threshold Voltage Computing Master's Thesis by Nikolaos Iatridis

[3] Pandeya, Pramesh & Basu, Prabal & Chakraborty, Koushik & Roy, Sanghamitra. (2020). GreenTPU: Predictive Design Paradigm for Improving Timing Error Resilience of a Near-Threshold Tensor Processing Unit. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. PP. 1-10. 10.1109/TVLSI.2020.2985057.

[4] A. Mahmoud et al., "PyTorchFI: A Runtime Perturbation Tool for DNNs", in 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), 2020.

[5] Givaki, Kamyar & Salami, Behzad & Hojabr, Reza & Tayaranian, S. & Khonsari, Ahmad & Rahmati, Dara & Gorgin, Saeid & Cristal, Adrian & Unsal, Osman. (2019). On the Resilience of Deep Learning for Reduced-voltage FPGAs.

[6] Yan, Z., Hu, X.S., & Shi, Y. (2022). Computing-In-Memory Neural Network Accelerators for Safety-Critical Systems: Can Small Device Variations Be Disastrous? *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 1-9.

[7] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi and J. Kepner, "Survey and Benchmarking of Machine Learning Accelerators," 2019 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2019, pp. 1-9, doi: 10.1109/HPEC.2019.8916327.

[8] Z. Vasicek and L. Sekanina, "Evolutionary Approach to Approximate Digital Circuits Design," in *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 3, pp. 432-444, June 2015, doi: 10.1109/TEVC.2014.2336175.

[9] H. Jiang, C. Liu, N. Maheshwari, F. Lombardi and J. Han, "A comparative evaluation of approximate multipliers," *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, Beijing, China, 2016, pp. 191-196, doi: 10.1145/2950067.2950068.

[10] S. Rehman, W. El-Harouni, M. Shafique, A. Kumar, J. Henkel and J. Henkel, "Architectural-space exploration of approximate multipliers," *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Austin, TX, USA, 2016, pp. 1-8, doi: 10.1145/2966986.2967005.

[11] Ying Wu, Chuangtao Chen, Weihua Xiao, Xuan Wang, Chenyi Wen, Jie Han, Xunzhao Yin, Weikang Qian, and Cheng Zhuo. 2024. A Survey on Approximate Multiplier Designs for Energy Efficiency: From Algorithms to Circuits. ACM Trans. Des. Autom. Electron. Syst. 29, 1, Article 23 (January 2024), 37 pages. https://doi.org/10.1145/3610291

[12] S. Hashemi, R. I. Bahar and S. Reda, "DRUM: A Dynamic Range Unbiased Multiplier for approximate applications," *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Austin, TX, USA, 2015, pp. 418-425, doi: 10.1109/ICCAD.2015.7372600.

[13] V. Leon, K. Asimakopoulos, S. Xydis, D. Soudris and K. Pekmestzi, "Cooperative Arithmetic-Aware Approximation Techniques for Energy-Efficient Multipliers," *2019 56th ACM/IEEE Design Automation Conference (DAC)*, Las Vegas, NV, USA, 2019, pp. 1-6.

[14] M. Wijtvliet, J. Huisken, L. Waeijen and H. Corporaal, "Blocks: Redesigning Coarse Grained Reconfigurable Architectures for Energy Efficiency," *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Barcelona, Spain, 2019, pp. 17-23, doi: 10.1109/FPL.2019.00013.

[15] Chuyi Li, Lulu Li, Hongliang Jiang, Kaiheng Weng, Yifei Geng, Liang Li, Zaidan Ke, Qingyuan Li, Meng Cheng, Weiqiang Nie, Yiduo Li, Bo Zhang, Yufei Liang, Linyuan Zhou, Xiaoming Xu, Xiangxiang Chu, Xiaoming Wei, & Xiaolin Wei. (2022). YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications.

[16] H. Jiang, C. Liu, N. Maheshwari, F. Lombardi and J. Han, "A comparative evaluation of approximate multipliers," *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, Beijing, China, 2016, pp. 191-196, doi: 10.1145/2950067.2950068.

[17] S. Venkatachalam and S. -B. Ko, "Design of Power and Area Efficient Approximate Multipliers," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 5, pp. 1782-1786, May 2017, doi: 10.1109/TVLSI.2016.2643639.

[18] Kyung-Ju Cho, Kwang-Chul Lee, Jin-Gyun Chung and K. K. Parhi, "Design of low-error fixed-width modified booth multiplier," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 12, no. 5, pp. 522-531, May 2004, doi: 10.1109/TVLSI.2004.825853.

[19] G. Zervakis, K. Tsoumanis, S. Xydis, D. Soudris and K. Pekmestzi, "Design-Efficient Approximate Multiplication Circuits Through Partial Product Perforation," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 24, no. 10, pp. 3105-3117, Oct. 2016, doi: 10.1109/TVLSI.2016.2535398.

[20] V. Mrazek, R. Hrbacek, Z. Vasicek and L. Sekanina, "EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods," *Design, Automation & Test in Europe Conference &*

*Exhibition (DATE), 2017*, Lausanne, Switzerland, 2017, pp. 258-261, doi: 10.23919/DATE.2017.7926993.