*electronics*

MDPI

*Article*

# The Genesis of *AI by AI* Integrated Circuit: Where AI Creates AI

**Emilio Isaac Baungarten-Leon [1,2,*], Susana Ortega-Cisneros [1,*], Mohamed Abdelmoneum [3], Ruth Yadira Vidana Morales [3] and German Pinedo-Diaz [1]**

[1] Centro de Investigación y de Estudios Avanzados, Instituto Politécnico Nacional, Zapopan 45019, Mexico; german.pinedo@cinvestav.mx
[2] Diseño Ciencia y Tecnologia, Universidad Autónoma de Guadalajara, Ciudad Universitaria, Zapopan 45129, Mexico
[3] Intel Corporation-Intel Labs, Hillsboro, OR 97124, USA; mohamed.a.abdel-moneum@intel.com (M.A.); ruth.y.vidana.morales@intel.com (R.Y.V.M.)
* Correspondence: emilio.baungarten@cinvestav.mx (E.I.B.-L.); susana.ortega@cinvestav.mx (S.O.-C.)

**Abstract:** The typical Integrated Circuit (IC) development process commences with formulating specifications in natural language and subsequently proceeds to Register Transfer Level (RTL) implementation. RTL code is traditionally generated through manual efforts, using Hardware Description Languages (HDL) such as VHDL or Verilog. High-Level Synthesis (HLS), on the other hand, converts programming languages to HDL; these methods aim to streamline the engineering process, minimizing human effort and errors. Currently, Electronic Design Automation (EDA) algorithms have been improved with the use of AI, with new advancements in commercial (such as ChatGPT, Bard, among others) Large Language Models (LLM) and open-source tools presenting an opportunity to automate the chip design process. This paper centers on the creation of *AI by AI*, a Convolutional Neural Network (CNN) IC entirely developed by an LLM (ChatGPT-4), and its manufacturing with the first fabricable open-source Process Design Kit (PDK), SKY130A. The challenges, opportunities, advantages, disadvantages, conversation flow, and workflow involved in CNN IC development are presented in this work, culminating in the manufacturing process of *AI by AI* using a 130 nm technology, marking a groundbreaking achievement as possibly the world's first CNN entirely written by AI for its IC manufacturing with a free PDK, being a benchmark for systems that can be generated today with LLMs.

**Keywords:** convolutional neural network; hardware design; integrated circuit; large language models



**Citation:** Baungarten-Leon, E.I.; Ortega-Cisneros, S.; Abdelmoneum, M.; Vidana Morales, R.Y.; Pinedo-Diaz, G. The Genesis of *AI by AI* Integrated Circuit: Where AI Creates AI. *Electronics* **2024**, *13*, 1704. https://doi.org/10.3390/electronics13091704

Academic Editor: George A. Tsihrintzis

Received: 5 March 2024
Revised: 3 April 2024
Accepted: 4 April 2024
Published: 28 April 2024



## 1. Introduction

The history of Integrated Circuit (IC) design is marked by innovation and technological strides. It began in the late 1950s with the introduction of the transistor [1]. Texas Instruments pioneered the first IC in 1958, integrating two transistors on a silicon–germanium bar [2]. Until the arrival of Computer-Aided Design (CAD) tools in 1966, ICs were manually drawn on paper [3].

The evolution of Hardware Description Language (HDL) started in the early 1970s with Register Transfer Level (RTL), allowing for thousands of transistors per IC [4]. DEC's PDP-16 RT-Level Modules [5], Instruction Set Processor Specifications [6], and Incremental System Programming Language [7] were significant contributions. In the late 1970s, programmable logic devices increased the demand for standard languages, and in 1985, Gateway and Intrametric introduced Verilog and VHSIC Hardware Description Language (VHDL) [8,9].

Alongside Verilog and VHDL, C-based hardware description languages, known as High-Level Synthesis (HLS), emerged. SystemC allowed the use of standard C++ and a class library for HDL generation in 1999, simplifying the IC development process with HLS [10]. Today, HLS tools like LegUP, Xilinx Vivado HLS, and Intel's HLS compiler transform C++ into HDL.

On the other hand, the journey of Artificial Intelligence (AI) started in the 1960s and improved during the 1970s and 1980s with foundational concepts and algorithms of deep learning and Artificial Neural Networks (ANN) [11,12]. During the late 1980s and early 1990s, the Machine Learning (ML) and AI community experienced a wave of enthusiasm as it was discovered that ANNs could tackle certain problems in novel ways. These networks had the distinct advantage of processing raw and diverse data types and developing hierarchical structures autonomously during the training phase for predictive tasks. However, the computational power at the time was insufficient for large-scale problems, limiting the application to smaller, simpler tasks [12–14].

It was not until the end of the 2000s that technological advancements, propelled by Moore's Law, equipped computers with the necessary power to train extensive ANNs on substantial, real-world challenges, such as the Imagenet project [15]. This advancement was largely due to the advent of general-purpose computing on graphics processing units, which offered superior floating-point performance compared to Central Processing Units (CPUs) [16]. This shift enabled ANNs to achieve remarkable results on complex issues of significant importance.

The last decade has been transformative for ML, especially with the rise of deep learning techniques that utilize ANN. These advancements have significantly enhanced the precision of systems in various domains [17]. Notable progress has been made in fields such as computer vision [18–21], speech recognition [22,23], language translation [24], and other complex natural language processing tasks [25–30]. This progress is attributed to the collective efforts and breakthroughs documented in key research papers.

Additionally, reinforcement learning shows promise in automating the design of custom Application-Specific Integrated Circuit (ASIC) by solving nondeterministic polynomial-hard optimization problems that are currently reliant on human expertise. This approach could revolutionize the synthesis, placement, and routing processes in chip design, potentially outperforming human teams by rapidly generating efficient layouts [31–33]. Google's preliminary experiments with this technology have yielded encouraging results, suggesting a future where machine learning accelerates and enhances the ASIC design process [14].

Research conducted by International Business Strategies Inc. in 2014, 2018, and 2022 categorizes the IC design costs into seven components: Intellectual Property (IP), Architecture, Verification, Physical Design, Software, Prototyping, and Validation. These studies reveal that design costs fluctuate significantly due to two primary factors: the prevailing technology at the time and the nanometer scale at which it is desired to fabricate. For instance, the design cost for a 28 nm circuit was approximately USD 140 million in 2014, reduced to USD 51.3 million in 2018, and further decreased to USD 48 million in 2022. Based on the 2018 and 2022 analyses, the estimated distribution of costs is as follows: IP at 6.85%, Architecture at 5.24%, Verification at 21.24%, Physical Design at 10.2%, Software at 43.32%, Prototyping at 5.24%, and Validation at 7.92%. These percentages provide a framework for approximating the allocation of expenses in IC design.

Advancements in machine learning could streamline the entire ASIC design process, from high-level synthesis to low-level logic placement and routing. This automation could drastically cut down design time from months to weeks, changing the economic calculus by reducing costs in Prototyping, Verification, and Architecture, combined with open-source tools and IPs, design costs would be further reduced. It may be feasible to create customized chips, which are currently reserved for high-volume and high-value scenarios.

Today, commercial LLMs like OpenAI's ChatGPT [34], Google's Bard [35], and Microsoft AI chatbot [36] have been used to introduce innovative HDL generation. These methods involve feeding the LLM with the system specifications, which then automatically produce HDL code. This synergy between AI and IC development promises enhanced efficiency and opens new frontiers in the field. Nevertheless, the state-of-the-art models fall short in their ability to effectively comprehend and rectify errors introduced by these tools, making it challenging to autonomously generate comprehensive designs and testbenches with minimal initial human intervention [37–39].

This work combines different processes to increase the complexity of an IC and reduce the amount of work required. The primary research inquiry revolves around the capability of contemporary commercial LLMs to produce Convolutional Neural Network (CNN) hardware designs that are not only synthesizable, but also manufacturable using the first open-source Process Design Kits (PDKs) called SKY130A.

The development of *AI by AI*—a CNN IC engineered for MNIST dataset classification—involves the use of LLM, Vivado HLS, Verilog, OpenLane, and Caravel. *AI by AI* was entirely crafted by OpenAI's ChatGPT-4. It began as a TensorFlow (TF) CNN architecture, followed by a downscaling from Python to C++, and then was translated to Verilog using Vivado HLS. The layout design process is made by OpenLane, resulting in a layout IP of the CNN. The journey culminated with the integration of the CNN IP with Caravel, a template System on Chip (SoC) which is ready for manufacturing using ChipIgnite shuttles, a multi-project wafer program by Efabless, with the SKY130A PDK [40,41]. Throughout this paper, we delve deeply into the development of *AI by AI* IC from TF to tape-out.

The remainder of this work is organized as follows: Section 2 provides an overview of the employed tools, outlining both their advantages and disadvantages; Section 3 explains the workflow and conversation flow; Section 4 is about the implementation of *AI by AI* IC; Section 5 shows the obtained results; Section 6 presents the discussions; and, finally, Section 7 concludes this work.

## 2. Development Tools

### *2.1. Vivado HLS 2019.1*

While traditional HDLs like Verilog and VHDL are acknowledged for their efficacy, their low-level abstraction often leads to long development cycles. A divergent approach is presented by HLS, offering a faster and more agile solution for hardware description development [42].

HLS functions through an automated process, enabling the generation of synthesizable RTL code from algorithms scripted in high-level languages such as C/C++ or System C. Although the resulting RTL code is commonly implemented on a Field-Programmable Gate Array (FPGA), it can also be translated into silicon, since it is described in HDL. In this case, the attractiveness of HLS lies in the possibility of generating HW with programming languages [42–44], Table 1 shows some advantages and disadvantages of HLS.

**Table 1.** Advantages and disadvantages of HLS.

| Advantages of HLS | Disadvantages of HLS |
|---|---|
| Reduces development time and effort | Does not have the same quality of results as HDLs |
| Architecture selection and optimization | Inconveniences in the hardware description |
| Parallelism and pipelining | Does not support all the features and constructs of the input languages |
| Allocates and shares resources efficiently | May not be compatible with all the existing tools and flows |

### *2.2. OpenLane*

This software is an open-source automated flow for layout design, conformed by various tools from *OpenROAD* and *Qflow*, focusing on the RTL to Graphic Design System (GDSII) design. Initially deployed for implementing the StriVe family, a RISC-V based SoC, using free EDA tools and the first open-source PDK SKY130A.

Currently comprising over seventy scripts and utilities, *OpenLane* can be configured for customized flows, enabling the implementation of diverse designs with any technology or PDK. The flow encompasses stages like synthesis, floorplaning, placement, Clock Tree Synthesis (CTS), routing, tapeout, and signoff [45–47].

The *OpenLane* flow initiates with HDL synthesis where the *Yosys* synthesis tool optimizes the design, resulting in a netlist mapped by the PDK. During this phase, design

constraints like clock definition and boundary conditions can be integrated, and Static Timing Analysis (STA) can be executed using the *OpenSTA* tool. Subsequently, floorplanning is conducted, with *OpenROAD* tools employed for macro-related tasks, producing a Design Exchange Format (DEF) file and defining matrix and macro core sizes. The *Padring* tool is harnessed for chip-level floorplanning, optimizing core pin positions for improved pad frame and core interconnect placement.

Post-floorplanning, standard cell, and macro placement are accomplished using the *Re-PlAce* tool, with subsequent placement checks conducted via *OpenDP*. The CTS phase follows, with *TritonCTS* placing clock branches and *OpenDP* adding necessary buffers. Routing is executed through a two-step approach: an initial phase with *FastRoute*, followed by a more intricate process with *TritonRoute*. In the concluding stages, the design undergoes verifications, including Design Rule Check (DRC), Layout Versus Schematic (LVS), and STA. Successful completion of these checks deems the design suitable for approval [45,48]; a graphical representation of the described process is illustrated in Figure 1 below.

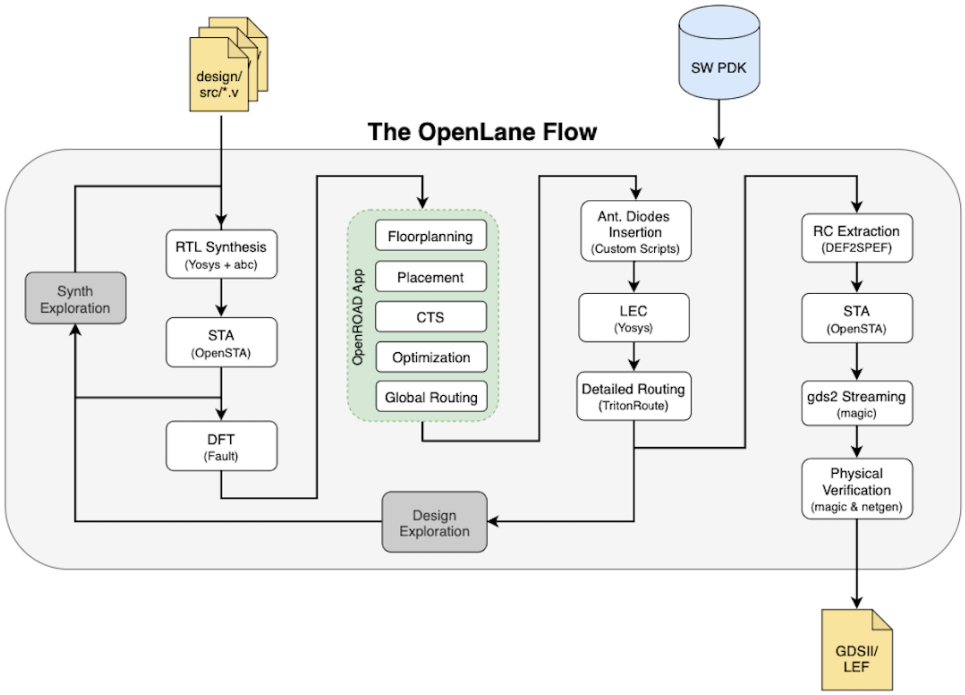


**Figure 1.** OpenLane workflow [48].

With the rise of *OpenLane*, new research has made a comparative analysis of this open-source tool with commercial tools [45–47,49,50]. Table 2 shows some of the advantages and disadvantages of *OpenLane*.

**Table 2.** Advantages and disadvantages of *OpenLane*.

| Advantages of OpenLane | Disadvantages of OpenLane |
|---|---|
| The entire flow is configured through a single configuration file | Less control over the flow compared to commercial tools |
| Automated flow, requires no manual intervention, once configured | Commercial tools have better time optimization |
| Open-source, no charge for use | *OpenLane* uses more logic cells in the design |
| Reduces the time and expertise required to obtain the GDSII | *OpenLane* generated designs tend to consume more power |

### 2.3. Caravel

*Caravel* is an SoC template developed by *Efabless* and built upon SKY130A and GF180MCUC technologies. It comprises three main sections: the template frame and two wrapper modules, known as the management area and user area [51].

The template frame is equipped with essential components, including a clocking module, Delay Locked Loop (DLL), user ID, housekeeping Serial Peripheral Interface (SPI), Power-On Reset (POR), and a General-Purpose Input/Output (GPIO) controller. The management area, housing a RISC-V based SoC, can configure and control the user area. The user area occupies a silicon space of 2.92 mm by 3.52 mm and includes 38 I/O pads, 128 Logic Analyzer (LA) signals, and four power pads. Figure 2 shows the block diagram of *Caravel* and its three sections [51].

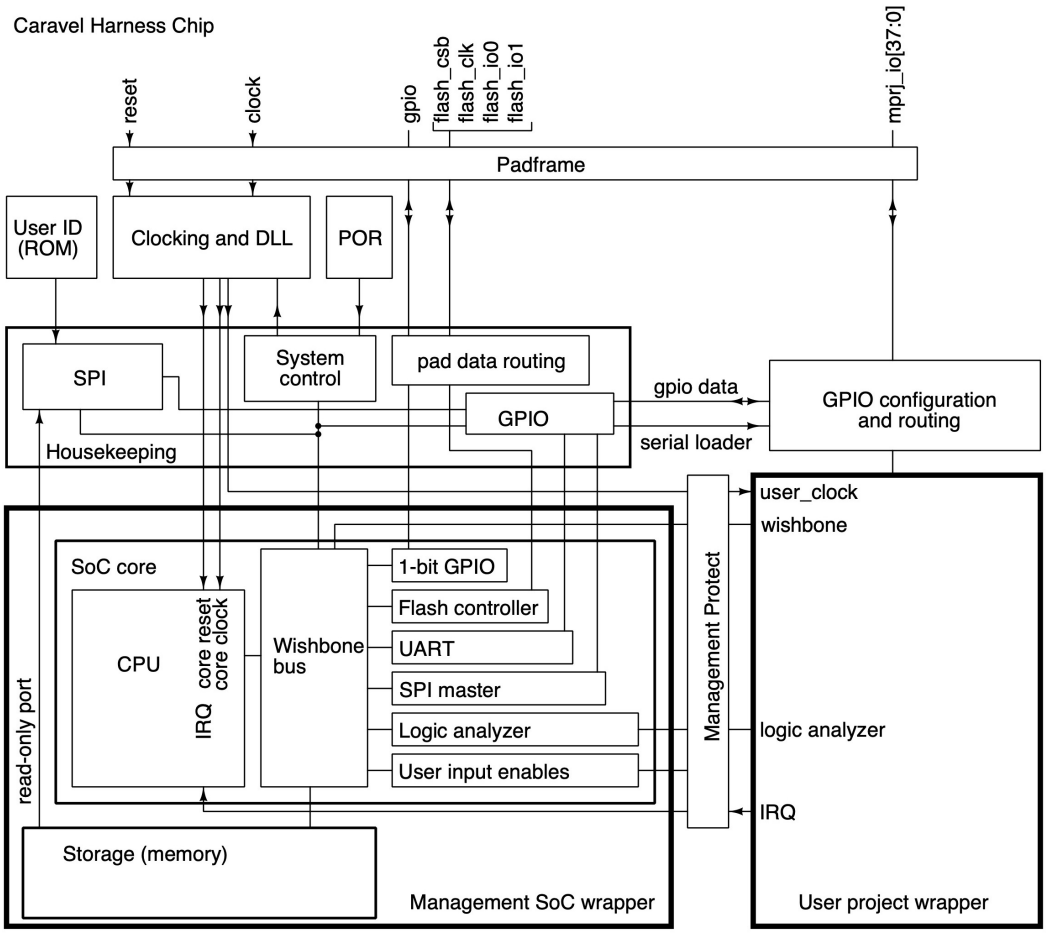


**Figure 2.** Caravel SoC architecture [51].

The very nature of a template offers great advantages when designing an IC; however, it also has some limitations. Table 3 shows these advantages and disadvantages.

**Table 3.** Advantages and disadvantages of *Caravel*.

| Advantages of Caravel | Disadvantages of Caravel |
|---|---|
| Allows low-cost and low-risk custom SoC design | Limited to SKY130A and GF180MCUC PDKs. |
| Supports various open-source tools and flows for IC design | May not be suitable for complex or high-end IC design projects |
| Enables fast SoC prototyping | Limited by 10 $mm^2$ and 38 GPIO pins |
| Enables collaboration and sharing with the open-source hardware community | |

## 3. Workflow and Conversation Flow

### *3.1. Large Language Model Conversation Flow*

The cornerstone of this work lies in the use of a commercial LLM for precise code generation, guided by the conversational flow depicted in Figure 3.

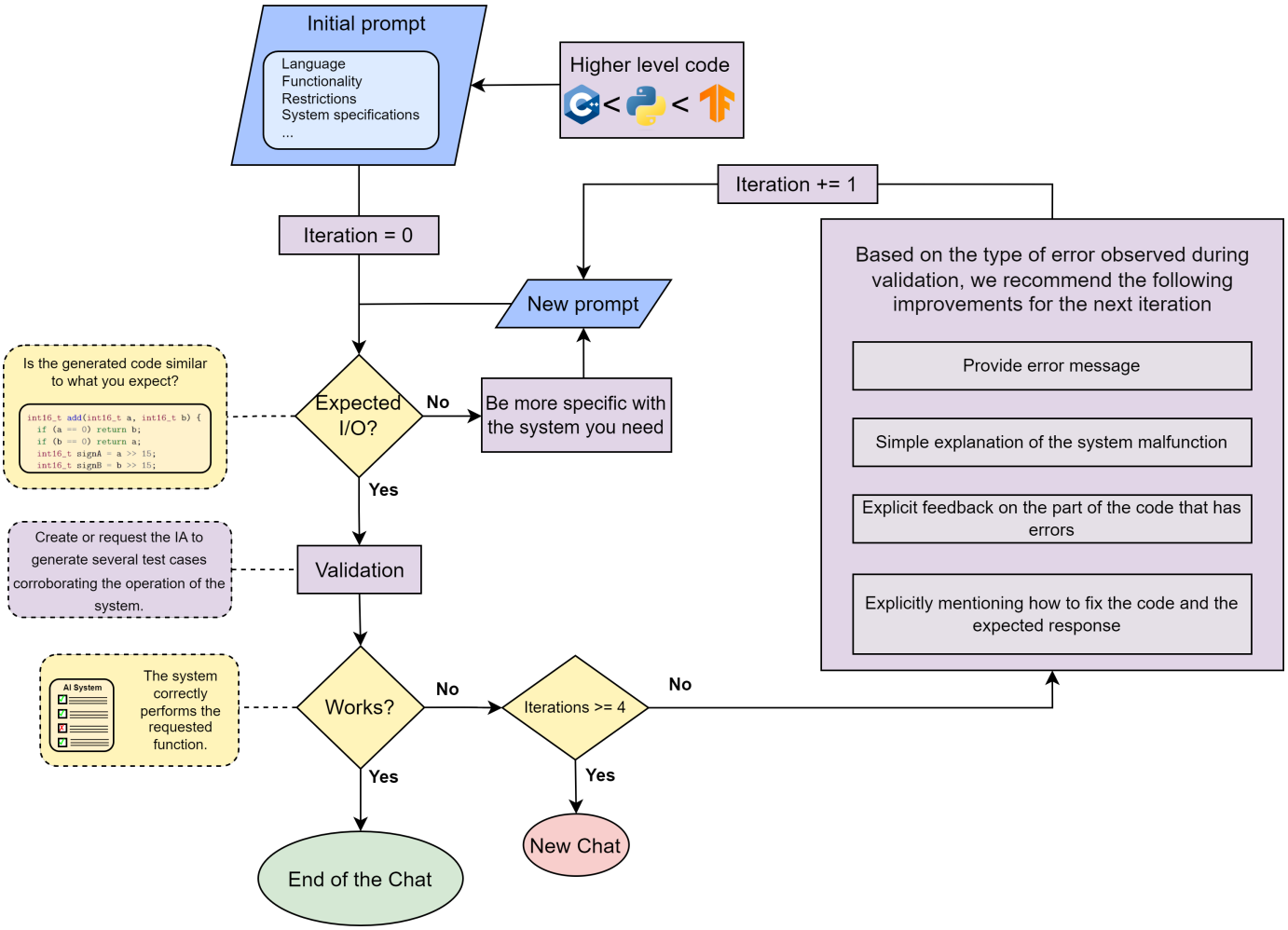


**Figure 3.** Conversation flow with LLM for code generation, highlighting the transition points and the recommendation for generation of prompts and new chat session.

The process starts by combining code from a higher abstraction level, if available, with the initial prompt. If the AI response does not meet the expected criteria, the creation of a more detailed prompt is initiated, to clarify specific requirements.

Upon receiving the expected response, the progression involves code simulation and testing, which means running the function for different cases and getting the expected result. The conversation concludes when the code functions as intended. However, in cases of code malfunction, the subsequent step entails the crafting of a new prompt incorporating error messages, heightened specificity, illustrative examples, or details regarding required code modifications, e.g., if the code does not work due to a data type error, it communicates so to the AI. After multiple iterations, when the LLM consistently produces similarly incorrect responses, it indicates the need to commence a new chat session.

### 3.2. From TensorFlow to Layout

Throughout the entire workflow, LLM played a central role in code generation, aligning with the conversation flow detailed in Section 3.1. *AI by AI* commences with the creation and training of the CNN architecture via TF. This initial phase allows training the CNN and capturing its essential weights and biases.

Considering the limitations of Caravel, we chose to implement a compact CNN with the following layers: Input layer (28 × 28 × 1), Convolutional Layer 1 (26 × 26 × 4), Max Pooling Layer 1 (6 × 6 × 4), Convolutional Layer 2 (4 × 4 × 8), Max Pooling Layer 2 (2 × 2 × 8), Flattening Layer (1 × 32), and dense layer.

Subsequently, the transformation of the TF model into a set of Python functions dedicated to executing the inference of the CNN, without the use of libraries, is initiated. A pivotal following step involves converting the Python-based forward function into C++, allowing the use of Vivado HLS.

The workflow culminates with the implementation of the CNN at the layout level, integrating it with Caravel. Figure 4 presents a visual representation of this process.

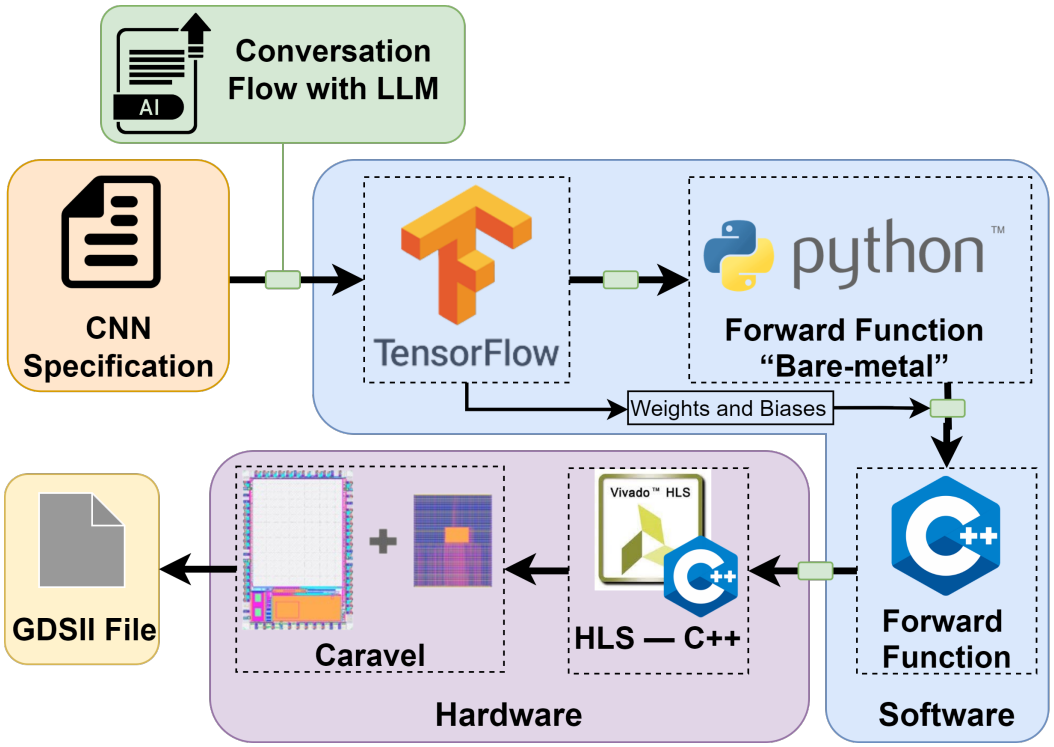


**Figure 4.** Workflow for the development of a CNN using LLM, from TF architecture to GDSII throughout Caravel integration.

## 4. Development of *AI by AI*

The development of *AI by AI* consists of a series of dialogues with ChatGPT-4, following the conversational structure outlined in Figure 3. For access to the complete conversations, the generated code, and the entire project, please refer to the following GitHub repository: https://github.com/Baungarten-CINVESTAV/AI_by_AI (accessed on 4 March 2024). Table 4 provides the ChatGPT URL of each conversation and the main topic covered in those conversations, accessed on 4 March 2024.

**Table 4.** ChatGPT-4, conversation URL.

| Subject of the Conversation | URL |
| --- | --- |
| Implementing a CNN in TF (accessed on 4 March 2024) | https://chat.openai.com/share/4e8a7cf2-a9e9-4461-a4b3-b9e8b4aa284f |
| Implementation of a forward function in Python without libraries (Bare-Metal) (accessed on 4 March 2024) | https://chat.openai.com/share/c96772be-4dac-43da-8013-c657dd935efa |
| From Python to C code I (accessed on 4 March 2024) | https://chat.openai.com/share/c96772be-4dac-43da-8013-c657dd935efa |
| From Python to C code II (accessed on 4 March 2024) | https://chat.openai.com/share/64b09191-401e-4d04-8eb5-5383b95ceea5 |
| Bias and weights as global parameters (accessed on 4 March 2024) | https://chat.openai.com/share/4b8237a4-20c3-434b-89fb-084fc5b57287 |
| From C to HLS I (accessed on 4 March 2024) | https://chat.openai.com/share/9037bfcd-8d23-4701-bafd-59eca930a822 |
| From C to HLS II (accessed on 4 March 2024) | https://chat.openai.com/share/84dd776b-0036-4fec-a878-dbcb33f6f210 |
| Add function, half-precision floating-point (accessed on 4 March 2024) | https://chat.openai.com/share/0f617bfd-f59a-49a3-a561-20b2779ca121 |
| Mult, Relu, Max function, half-precision floating-point (accessed on 4 March 2024) | https://chat.openai.com/share/2b207fc6-5952-4ef7-a562-64765e2d6722 |
| Exponent function, half-precision floating-point (accessed on 4 March 2024) | https://chat.openai.com/share/5345f69b-5e04-4fdf-a062-f29b2fcc4564 |

This chapter is structured into five distinct subsections, as visually represented in Figure 4. In each of these sections, the relevant prompts, primary challenges, key considerations, and the step-by-step development process are detailed. The journey commences with the creation of the CNN using TF, and culminated with the generation of the GDSII file ready for manufacturing.

### *4.1. CNN with TF*

The CNN was designed for image inference tasks toward the renowned MNIST dataset [52]. To harness the power of cloud computing, we opted for Google Colab [53], primarily due to its integration of TF libraries and the capacity to use GPUs.

The noteworthy prompts that emerged during the interactions with ChatGPT-4 included:

- Generate a CNN for the MNIS dataset using TF and Google Colab.
- Change the CNN model to be: 4 × 3 × 3 Conv2D, 4 × 4 MaxPool, 8 × 3 × 3 Conv2D, 2 × 2 MaxPool, and work with float16.
- Obtain the weights and biases for each layer, then write those weights on a .npy file and .bin file. Save both as a float16 data type.

The approach taken involved implementing a compact network using the following layers 4 × 3 × 3 Conv2D, 4 × 4 MaxPool, 8 × 3 × 3 Conv2D, 2 × 2 MaxPool, flatten, and finally, the dense layer, as well as the use of half-precision floating-point format to optimize resource usage. Figure 5 illustrates the CNN created.

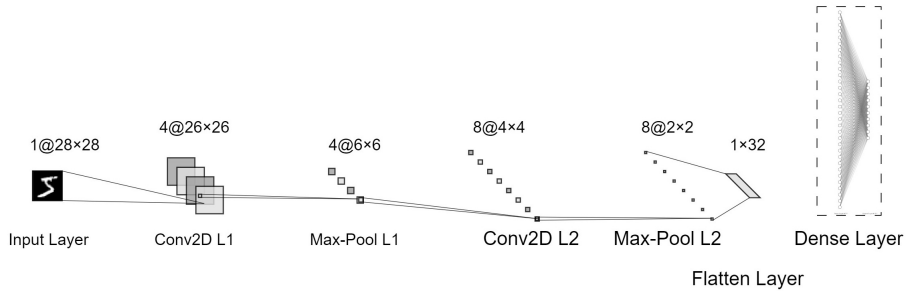


**Figure 5.** CNN architecture for the task of classifying MNIST images.

The implemented CNN utilizes a total of 666 parameters. This breakdown encompasses 36 weights and 4 biases for the initial convolutional layer, 288 weights and 8 biases for the second convolutional layer, and, finally, 320 weights and 10 biases for the dense layer. In terms of memory consumption, this results in a total of 1.332 KB required only for storing the weights and biases. At the end of the training phase, the model showed an accuracy of 99.4%. Part of the TF code of the CNN generated by the IA can be found below.

```
# Define the CNN model
model = models.Sequential()
model.add(layers.Conv2D(4, (3, 3), activation='relu', input_shape=(28,28,
↪ 1)))
model.add(layers.MaxPooling2D((4, 4)))
model.add(layers.Conv2D(8, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10, dtype='float16'))
```

### *4.2. Forward Function in Python*

Implementing the inference function in Python without the use of the TF library is a critical step in the process because, as we approach lower-level languages or avoid the use of libraries, we obtain answers with a higher number of errors. To face that problem, we provide the LLM with examples in a higher level language. In this case, ChatGPT-4 is instructed to utilize the pre-existing network, created with TF, to create the inference function using the weights and biases from previously saved NumPy files.

Key prompts from interactions with ChatGPT-4:

- Write a bare metal implementation of the CNN, just the forward function, assuming that the CNN was trained previously.
- Call the function forward based on the previous weights and biases .np file.
- Develop a functionality test of the previous code showing the selected image and its label.

The previous chat generated six essential secondary functions required for inference implementation: relu, softmax, conv2d_forwar, maxpool2d_forward, flatten, dense_forward, and a main function named forward, which calls within it the secondary functions. The following code shows the definition of the forward function and how it was used to perform the test phase.

```
def forward(X, W_conv1, b_conv1, W_conv2, b_conv2, W_dense2, b_dense2):
    out = conv2d_forward(X, W_conv1, b_conv1)
    out = relu(out)
    out = maxpool2d_forward(out, 4)
    out = conv2d_forward(out, W_conv2, b_conv2)
    out = relu(out)
    out = maxpool2d_forward(out, 2)
    out = flatten(out)
    out = dense_forward(out, W_dense2, b_dense2)
    out = softmax(out)
    return out

for i in range(10000):
    # prepare the input
    x = test_images[i].astype(np.float16)
```

```
    # Ensure that the single image has the right shape (28, 28, 1)
    single_image = np.reshape(x, (28, 28, 1))
    output = forward(single_image, weights_conv1, biases_conv1,
     ↪  weights_conv2, biases_conv2, weights_dense2, biases_dense2)
    # Find the index of the maximum value
    prediction = np.argmax(output)
    # compare the prediction with the actual label
    actual_label = np.argmax(test_labels[i])
    #actual_label = test_labels[i]
    if prediction == actual_label:
        correct_predictions += 1
# calculate the accuracy
accuracy = correct_predictions / 10000
print("The accuracy of the forward_pass function is:", accuracy)
```

Chat interactions from Sections 4.1 and 4.2 were brief, primarily due to the use of Python.

### *4.3. From Python to C++*

Utilizing a low-level programming language necessitated a more explicit approach to crafting prompts. This involved providing the entire code for the seven previously generated functions and demanded a higher number of iterations.

Main prompts obtained during interactions with ChatGPT-4:

- I develop the following CNN model in a python bare metal implementation for the mnist dataset:
  The CNN model is:
  <Here, the python code is attached>
  Rewrite the Python code on a C code: Weights and biases will be loaded from the bin file.
- Implement the whole forward function and develop the fmaxf function used in the maxpool layer.
- Create a function that convert the forward function into one hot output.
- Based on the C code create a function that compares the output of the forward function and the label:
  <Here, the C++ code is attached>

After the "From Python to C code" conversations mentioned in Table 4, we achieved a successful implementation of all the layers of the CNN in a short time. The C++ code presented below shows how the forward function is called N times for the test phase.

```
// Load the weights and biases from the binary files
    load_weights_biases();
    // For each image
    for (int i = 0; i < NUM_IMAGES; ++i) {
        // Call the forward pass function
        out_forward = forward_pass(images[i], one_Hot_output);
        printf("Predicted label = %d, Actual label = %d\n", out_forward,
         ↪  labels[i]);
        // If the predicted label matches the actual label, increment the
         ↪  count of correct predictions
        if (out_forward == labels[i]) {
            ++correct_predictions;
        }
```

```
    }
    // Calculate the accuracy
    float accuracy = ((float)correct_predictions / NUM_IMAGES) * 100.0f;
    printf("Accuracy = %.2f%%\n", accuracy);
```

Part of the forward_pass function is presented below, where each of the layers, both convolutional and maxpool, was implemented through a series of for loops, where variable *i* represents the pixel coordinate in x, variable *j* represents the pixel coordinate in y, and variable *k* represents the filter number. On the other hand variables *di* and *dj* represent the kernel, being a $3 \times 3$ kernel for the first convolutional layer.

```
int forward_pass(float* image, int* one_hot_output) {
    // Assume here that the image has a size of 28x28x1 and weights and
    ↪  biases are already loaded
    // First Conv2D layer: input is 28x28x1, filter is 3x3x1x4
    // It results in a 26x26x4 output (we are assuming VALID padding)
    float conv1[26][26][4];
    for(int i = 0; i < 26; i++)
        for(int j = 0; j < 26; j++)
            for(int k = 0; k < 4; k++) {
                conv1[i][j][k] = 0;
                // Convolution operation
                for(int di = 0; di < 3; di++)
                    for(int dj = 0; dj < 3; dj++)
                        conv1[i][j][k] += image[(i+di)*28 + (j+dj)] *
                        ↪  weights_conv1[(di*3 + dj)*4 + k];
                conv1[i][j][k] += biases_conv1[k];
                // ReLU activation
                conv1[i][j][k] = relu(conv1[i][j][k]);
            }
```

The C++ code provided by the AI can be easily scaled and customized to create various convolutional layers, changing only the ranges of the first two for loops that represent the size of the image, the third for represents the amount of filter that the layer has, and the last two for loops represent the size of the kernel. This versatility opens the opportunity to construct a wide range of CNNs, and all with the code provided by the AI.

### *4.4. Vivado HLS Considerations*

The C++ code generated by the IA uses floating data types, although Vivado HLS supports this type of data when implemented at the hardware level it uses a restricted Floating Point Units (FPUs) IP, so its use is limited only to Xilinx boards.

To face this issue, C++ functions that utilize 16-bit integer data types, but perform floating-point operations at the bit level, were developed through a series of LLM conversations, keeping in mind the IEEE® 754 half-precision floating-point format.

A total of eight functions were developed: addition, subtraction, multiplication, division, exponential, softmax, relu, and max. The addition, multiplication and division functions can be found in Appendix A.

The main prompts obtained during interactions with ChatGPT-4 are:

- Develop an [addition, subtraction, multiplication, division, relu, max] function of two numbers of 16 bits with the following structure, sign bit, 5-bit exponent, 10-bit mantissa. Generate the C code for HLS.
- Consider the case in which A and B are the same number.
- Consider the case in which A or B are equal to 0.
- Consider the case in which A and B have different signs.
- Develop an exp function of a number of 16 bits with the following structure, sign bit, 5 bits exponent, 10 bits mantissa. Generate the C code for HLS, avoid the use of floating point data type, and if you use an add, mult, div functions use:

<Here, the C++ code floating functions are attached>

The generated functions are then used to perform floating operations and used to replace the arithmetic symbols of the existing solution; e.g., instead of executing the

```
conv1[i][j][k] += image[(i+di)*28 + (j+dj)] * weights_conv1[(di*3 + dj)*4
 ↪  + k];
```

operationpresented in the forward function, the operation is executed as

```
aux_mult = multiply_custom_float(image[(i+di)*28
 ↪  +(j+dj)],weights_conv1[(di*3 + dj)*4 + k]);
conv1[i][j][k] = add(conv1[i][j][k],aux_mult);
```

where the multiplication of the pixel and the kernel is performed by the multiply_custom_floa function, and the summation of the convolution by the add function.

Due to variations in rounding methods for floating operations, the accuracy experienced a 1.4% reduction, which means that change from 99.4% to 98%. However, this error can be avoided if the floating functions created use exactly the same rounding algorithm used by TF.

### *4.5. Integration of the CNN with Caravel*

To integrate the CNN with the SoC template Caravel involves the creation of a single macro encompassing the logic of all the modules generated by HLS, because the logical density of the design utilizes the majority of the user area an external memory was employed for image storage which was connected to Caravel via GPIO ports. Meanwhile, the CNN was linked to the Caravel RISCV processor using the LA ports as Figure 6 illustrates. This connection allowed the RISCV processor to manage the initiation of the inference process, with the signal la_data_in[2], system restarts, with the signal la_data_in[1], and receive the response of the inference from the CNN, with the signal la_data_out[31:28]; Table 5 shows the connection between *AI by AI* and Caravel.

The verilog code provided to the OpenLane layout tool is just an instantiation of the IP generated by HLS connected to the Caravel ports; Appendix B shows this instantiation.

**Table 5.** Pinout of Caravel and AI by AI.

| Caravel | AI by AI | Type |
|---|---|---|
| wb_clk_i | o_mux_clk | Input |
| io_in[36] | o_mux_clk | Input |
| io_in[37] | s_mux_clk | Input |
| o_mux_clk | ap_clk | Input |
| la_data_in[1] | in_ap_rst | Input |
| io_in[35] | in_ap_rst | Input |

**Table 5.** *Cont.*

| Caravel | AI by AI | Type |
|---|---|---|
| wb_clk_i | o_mux_clk | Input |
| io_in[36] | o_mux_clk | Input |
| io_in[37] | s_mux_clk | Input |
| o_mux_clk | ap_clk | Input |
| la_data_in[1] | in_ap_rst | Input |
| io_in[35] | in_ap_rst | Input |
| la_data_out[2] | ap_start | Input |
| la_data_out[3] | ap_done | Output |
| la_data_out[4] | ap_ready | Output |
| io_out[16:5] | image_r_Addr_A | Output |
| io_out[17] | image_r_EN_A | Output |
| N/A | image_r_WEN_A | Output |
| N/A | image_r_Din_A | Output |
| io_in[33:18] | image_r_Dout_A | Input |
| io_out[34] | image_r_Clk_A | Output |
| N/A | image_r_Rst_A | Output |
| la_data_out[31:28] | ap_return | Output |

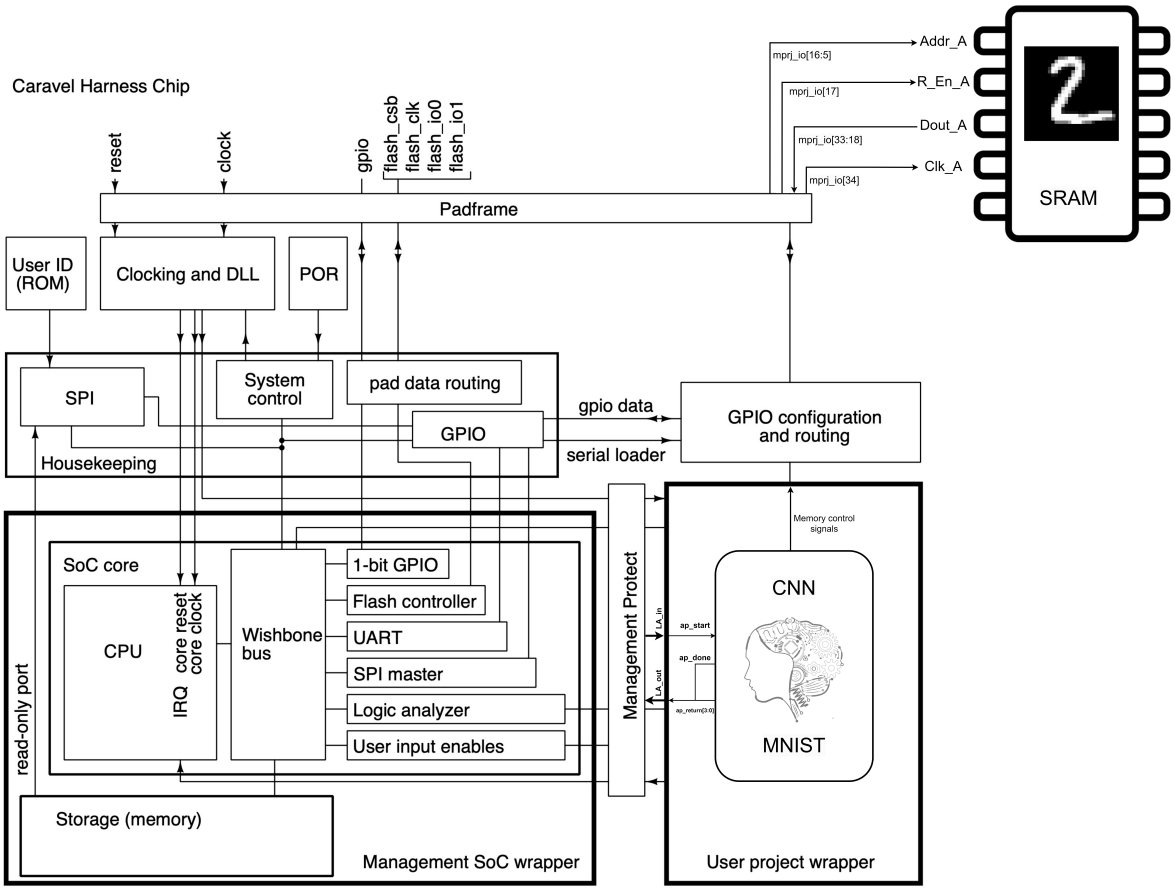


**Figure 6.** *AI by AI* and *Caravel* integration diagram.

## 5. Results

After establishing the connections between Caravel and the CNN, a testbench of the entire SoC was developed using the training data set to evaluate the performance of the CNN. Due to the RISC-V managing the SoC, some registers using C++ were configured to enable the utilization of LA ports, allowing communication between the CNN and the RISC-V processor, as well as GPIOs that enabled connectivity between the external SRAM and the SoC.

Figure 7 illustrates the SoC testbench, the image stored in memory, and the C++ code programmed in the RISC-V processor. The figure depicts the processor's handling of reset signals, start processes, the waiting period for the done signal, and the resulting inference values. After 1000 iterations, the system yields the same results as the HLS test, with an accuracy of 98%, proving that it works as intended.


**Figure 7.** Testbench of the IC *AI by AI* implemented with SKY130A standard cells.

Table 6 presents the layout specifications, with the SKY130A PDK, for the *AI by AI* system, including the gate count, die area, latency, maximum frequency, and power consumption.

**Table 6.** AI by AI layout specifications with the SKY130A PDK.

| Parameter | Value |
|---|---|
| Core area | 10.27 $mm^2$ |
| Core Utility | 8.747 $mm^2$ |
| Cells per $mm^2$ | 26,241 |
| Latency | 161.19 K |
| Maximum frequency | 40 MHz |
| Static Power | 70.5 mW |
| Switching Power | 50.5 mW |
| Buffers | 65,142 |
| Flip-Flops | 49,973 |
| Diode | 33,839 |
| Number of Cells | 94,415 |

The outcome of the RTL to GDSII conversion process, along with its integration with the RISC-V made with Caravel, is visually presented in Figure 8. It illustrates two distinct areas: the user area, representing a flat implementation of the CNN, and the management area, housing the processor and its associated peripherals.

This project was the winner of the AI-generated design competition hosted by *Efabless*, which can be accessed at this link: https://efabless.com/genai/challenges/2-winners (accessed on 4 March 2024).Additionally, the CNN SoC is currently undergoing fabrication through the multi-project wafer shuttle CI 2309, which is available at https://platform.efabless.com/shuttles/CI%202309 (accessed on 4 March 2024).

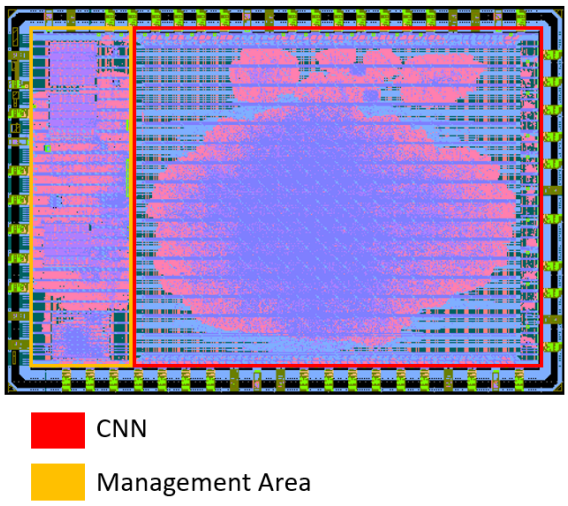


**Figure 8.** Caravel GDSII file and CNN layout details.

## 6. Discussion

The findings of this research highlight significant aspects, such as:

1. The current limitations of LLMs in generating HDL code.
2. Establishing a workflow that utilizes LLMs to generate and downscale systems from TF to HDL.
3. Introducing a new approach for converting HLS to GDSII using open-source PDKs and tools.
4. Achieving the fabrication of a CNN IC entirely created by AI.
5. Setting a precedent for current AI-generated systems by providing specific system information, such as core area, cells per square millimeter, latency, power consumption, number of flip-flops, and total number of cells.
6. Offering open-source access to the entire project, from the initial conversation with the AI to the final GDSII files generated.

These findings directly address our central research question, "are contemporary commercial LLMs capable of producing synthesizable and manufacturable CNN hardware designs using the first open-source PDKs (SKY130A)?",by providing new understanding and evidence that current commercial LLMs are not capable of directly creating a CNN in HDL; however, they are capable of creating synthesizable HLS code that can be used to generate IC with open-source tools. The paper elucidates the development of *AI by AI*, an innovative IC harnessing the power of AI. Our methodology involved the transformation of AI-generated TF code into Verilog, progressing through layout implementation and seamless integration with a RISC-V via Caravel. This process ultimately enabled us to propel *AI by AI* into the manufacturing phase through the ChipIgnite program.

*AI by AI* stands as a pioneering achievement, being the first CNN IC of its kind to be entirely conceptualized by AI and be fabricated with the open-source PDK SKY130A. Our approach harmoniously merges cutting-edge technologies, such as commercial LLMs, with more traditional ones like HLS and Verilog, creating an innovative workflow for developing intricate digital systems, particularly CNNs, and exploring the capacities of the current LLM. Frameworks like Caravel and multi-project wafer programs such as ChipIgnite have simplified and made cost-effective the layouts development and fabrication process.

While current commercial LLMs may not yet excel in rapidly and accurately producing Verilog and VHDL code, they have matured enough to proficiently handle programming tasks. The sequential transition from higher abstraction to lower abstraction languages, supplemented by tools like HLS, empowers us to generate functional Verilog code that seamlessly integrates into the silicon-level implementation process. This combination of technologies and methodologies has opened new horizons for AI-driven IC development.

## 7. Conclusions

AI is experiencing a boom in various sectors, including IC design. With LLMs such as ChatGPT, exploration in HDL generation has begun, which could reduce design costs by 31.72%—impacting prototyping, architecture, and verification phases, and compressing design timelines from months to weeks. Additionally, leveraging open-source tools and IPs could further reduce costs associated with software (43.32%) and IP (6.85%), respectively. Despite the potential, current LLMs have difficulties in producing complex HDLs systems with accurate performance, and open-source IPs are not as abundant as software libraries. Therefore, current research is focused on high-level languages such as Python and C++ to enable LLMs to efficiently create complex systems such as CNNs. HLS becomes crucial in this context for translating high-level code into HDL that, through the physical design flow, generates an IC that can be manufactured. The use of HLS causes some issues related to floating point operations, which can lead to a loss of accuracy and increased logical demands. If the loss of accuracy is significant, we recommend accessing the TF code and replicating in C++ the rounding algorithms it uses. This research establishes a benchmark for current LLM capabilities in ICs design, in particular for the design of CNNs, and is a point of comparison for evaluating future AI-generated ICs.

**Author Contributions:** Conceptualization, E.I.B.-L. and S.O.-C.; methodology, E.I.B.-L. and S.O.-C.; software, E.I.B.-L.; validation, E.I.B.-L., M.A., R.Y.V.M. and G.P.-D.; formal analysis, E.I.B.-L. and R.Y.V.M.; investigation, E.I.B.-L. and M.A.; resources, S.O.-C.; writing—original draft preparation, E.I.B.-L.; writing—review and editing, E.I.B.-L., S.O.-C., M.A., R.Y.V.M. and G.P.-D.; supervision, S.O.-C. and M.A.; project administration, E.I.B.-L. and S.O.-C.; funding acquisition, S.O.-C. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Data Availability Statement:** The data presented in this study are available or are mentioned in this paper.

**Conflicts of Interest:** Dr. Mohamed Abdelmoneum and Dr. Ruth Ruth Yadira Vidana Morales are employed by Intel Corporation. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Appendix A. Half Precision Floating Point Functions in C++

```
int16_t add(int16_t a, int16_t b) {
  if (a == 0) return b;
  if (b == 0) return a;
  int16_t signA = a >> 15;
  int16_t signB = b >> 15;
  int16_t expA = (a >> 10) & 0x1F;
  int16_t expB = (b >> 10) & 0x1F;
  int16_t mantissaA = a & 0x3FF;
  int16_t mantissaB = b & 0x3FF;
  int16_t i_loop;
  // Denormalize the mantissas
  mantissaA |= 0x400;
  mantissaB |= 0x400;
  // Align mantissas by shifting the one with the smaller exponent
  i_loop = 0;
  while (expA < expB) {
    mantissaA >>= 1;
    expA++;
    i_loop++;
```

```
  if(i_loop>50)
  {
          break;
  }
}
i_loop = 0;
while (expB < expA) {
  mantissaB >>= 1;
  expB++;
  i_loop++;
  if(i_loop>50)
  {
          break;
  }
}
// Perform add2ition or subtraction based on the signs
int32_t resultMantissa;
if (signA == signB) {
  resultMantissa = mantissaA + mantissaB;
} else if (signA) {
  resultMantissa = mantissaB - mantissaA;
} else {
  resultMantissa = mantissaA - mantissaB;
}
int16_t resultSign = (resultMantissa < 0) ? 1 : 0;
if (resultMantissa < 0) {
resultSign = 1;
resultMantissa = -resultMantissa;
} else {
  resultSign = 0;
}
// Normalize the result
i_loop = 0;
while (resultMantissa >= 0x800) {
  resultMantissa >>= 1;
  expA++;
  i_loop++;
  if(i_loop>50)
  {
          break;
  }
}
i_loop = 0;
while (resultMantissa < 0x400) {
  resultMantissa <<= 1;
  expA--;
  i_loop++;
  if(i_loop>50)
  {
          break;
  }
}
// Create the result
int16_t result = (resultMantissa & 0x3FF) | (expA << 10);
```

```
  if ((signA && signB) || resultSign) {
    result |= 0x8000;
  }
  return result;
}
int16_t multiply_custom_float2(int16_t a, int16_t b) {
    if (a == 0 || b == 0) return 0;
    // Extracting sign, exponent, and mantissa for 'a'
    int16_t sign_a = (a >> 15) & 1;
    int16_t exponent_a = (a >> 10) & (int16_t)0x1F;
    int16_t mantissa_a = a & (int16_t)0x3FF;

    // Extracting sign, exponent, and mantissa for 'b'
    int16_t sign_b = (b >> 15) & 1;
    int16_t exponent_b = (b >> 10) & (int16_t)0x1F;
    int16_t mantissa_b = b & (int16_t)0x3FF;

    // Calculating the result's sign, exponent, and mantissa
    int16_t sign_result = sign_a ^ sign_b;
    int16_t exponent_result = (exponent_a - 15) + (exponent_b - 15) + 15;
    ↪  // Remove bias, add2, then add2 bias back
    int32_t mantissa_result = (1024 + mantissa_a) * (1024 + mantissa_b);

    // Normalizing the mantissa
    if (mantissa_result >= (1 << 21)) {
        mantissa_result >>= 1;
        exponent_result += 1;
    }
    mantissa_result = (mantissa_result >> 10) - 1024; // Remove the
    ↪  implicit leading one
    // Check for underflow or overflow
    if (exponent_result < 0) return 0; // Underflow

    if (exponent_result >= 0x1F) return sign_result ? (int16_t)0x8000 :
    ↪  (int16_t)0x7FFF; // Overflow
    // Combining sign, exponent, and mantissa into a 16-bit integer
    int16_t result = (sign_result << 15) | ((exponent_result &
    ↪  (int16_t)0x1F) << 10) | (mantissa_result & (int16_t)0x3FF);
    return result;
}

int16_t divide_custom_float2(int16_t a, int16_t b) {
    int16_t sign_a = (a >> 15) & 1;
    int16_t exponent_a = (a >> 10) & (int16_t)0x1F;
    int16_t mantissa_a = (a & (int16_t)0x3FF) | (int16_t)0x400;
    int16_t sign_b = (b >> 15) & 1;
    int16_t exponent_b = (b >> 10) & (int16_t)0x1F;
    int16_t mantissa_b = (b & (int16_t)0x3FF) | (int16_t)0x400;
    if (mantissa_b == 0) return 0;
    int16_t sign_result = sign_a ^ sign_b;
    int16_t exponent_result = exponent_a - exponent_b + 15;
    int32_t remainder = mantissa_a << 10;
    int32_t divisor = mantissa_b << 10;
    int32_t quotient = 0;
```

```
    for (int i = 0; i < 10; i++) {
        remainder <<= 1;
        if (remainder >= divisor) {
            remainder -= divisor;
            quotient = (quotient << 1) | 1;
        } else {
            quotient <<= 1;
        }
    }
    if (quotient < (int16_t)0x400) {
        quotient <<= 1;
        exponent_result -= 1;
    }
    int16_t mantissa_result = quotient & (int16_t)0x3FF;
    int16_t result = (sign_result << 15) | ((exponent_result &
    ↪  (int16_t)0x1F) << 10) | mantissa_result;
    return result;
}
```

## Appendix B. Top Module Verilog Code

```
module user_project_wrapper #(
    parameter BITS = 32
) (
    // Wishbone Slave ports (WB MI A)
    input wb_clk_i,
    input wb_rst_i,
    input wbs_stb_i,
    input wbs_cyc_i,
    input wbs_we_i,
    input [3:0] wbs_sel_i,
    input [31:0] wbs_dat_i,
    input [31:0] wbs_adr_i,
    output wbs_ack_o,
    output [31:0] wbs_dat_o,
    // Logic Analyzer Signals
    input  [127:0] la_data_in,
    output [127:0] la_data_out,
    input  [127:0] la_oenb,
    // IOs
    input  [`MPRJ_IO_PADS-1:0] io_in,
    output [`MPRJ_IO_PADS-1:0] io_out,
    output [`MPRJ_IO_PADS-1:0] io_oeb,
    inout [`MPRJ_IO_PADS-10:0] analog_io,
    // Independent clock (on independent integer divider)
    input   user_clock2,
    // User maskable interrupt signals
    output [2:0] user_irq
);
/*--------------------------------------*/
/* User project is instantiated  here   */
/*--------------------------------------*/
  wire in_ap_rst;
```

```
 wire _in_ap_start;
 wire _ap_done;
 wire _ap_idle;
 wire _ap_ready;
 wire [31:0] _image_r_Addr_A;
 wire _image_r_EN_A;
 wire [1:0] _image_r_WEN_A;
 wire [15:0] _image_r_Din_A;
 wire [15:0] i_in_image_r_Dout_A;
 wire   _image_r_Clk_A;
 wire   _image_r_Rst_A;
wire  [3:0] _ap_return;
assign in_ap_rst = la_data_in[1]|io_in[30+5];
assign io_oeb[30]=1;
assign  _in_ap_start = la_data_in[2];
assign la_data_out[3]=_ap_done;
assign  la_data_out[4]=_ap_idle;
assign  la_data_out[5]=_ap_ready;
assign  la_data_out[31:28] = _ap_return;

//External memory controls
 assign io_out[11+5:0+5] = _image_r_Addr_A[11:0]; //Addres
 assign io_oeb[11+5:0+5] = 11'b0;
 assign io_out[12+5] = _image_r_EN_A; //r_Enb
 assign io_oeb[12+5]=0;
 assign i_in_image_r_Dout_A = io_in[28+5:13+5]; //Data_input
  assign io_oeb[28+5:13+5]=16'hFFFF;
 assign io_out[29+5] = _image_r_Clk_A; //CLK
  assign io_oeb[29+5]=0;
 forward_pass AI_by_AI (
   .ap_clk(wb_clk_i),
   .ap_rst(in_ap_rst),
   .ap_start(_in_ap_start),
   .ap_done(_ap_done),
   .ap_idle(_ap_idle),
   .ap_ready(_ap_ready),
   .image_r_Addr_A(_image_r_Addr_A),
   .image_r_EN_A(_image_r_EN_A),
   .image_r_WEN_A(_image_r_WEN_A), //We just read the memory dont write
    ↪  in it
   .image_r_Din_A(_image_r_Din_A),
   .image_r_Dout_A(i_in_image_r_Dout_A),
   .image_r_Clk_A(_image_r_Clk_A),
   .image_r_Rst_A(_image_r_Rst_A),
   .ap_return(_ap_return)
 );
```

## References


1. Bardeen, J.; Brattain, W.H. The transistor, a semi-conductor triode. *Phys. Rev.* **1948**, *74*, 230. [CrossRef]
2. Kilby, J.S.C. Turning potential into realities: The invention of the integrated circuit (Nobel lecture). *ChemPhysChem* **2001**, *2*, 482–489. [CrossRef] [PubMed]
3. Spitalny, A.; Goldberg, M.J. On-line operation of CADIC (computer aided design of integrated circuits). In Proceedings of the 4th Design Automation Conference, Los Angeles, CA, USA, 19–22 June 1967; pp. 7-1–7-20.

4. Barbacci, M. A Comparison of Register Transfer Languages for Describing Computers and Digital Systems. *IEEE Trans. Comput.* **1975**, *C-24*, 137–150. [CrossRef]
5. Bell, C.G.; Grason, J.; Newell, A. *Designing Computers and Digital Systems Using PDP 16 Register Transfer Modules*; Digital Press: Los Angeles, CA, USA, 1972.
6. Barbacci, M.R.; Barnes, G.E.; Cattell, R.G.G.; Siewiorek, D.P. *The ISPS Computer Description Language: The Symbolic Manipulation of Computer Descriptions*; Departments of Computer Science and Electrical Engineering, Carnegie-Mellon University: Pittsburgh, PA, USA, 1979.
7. Barbacci, M.R. *The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator*; Carnegie Mellon University, Department of Computer Science: Pittsburgh, PA, USA, 1976.
8. Huang, C.L. Method and Apparatus for Verifying Timing during Simulation of Digital Circuits. U.S. Patent 5,095,454, 10 March 1992.
9. Shahdad, M.; Lipsett, R.; Marschner, E.; Sheehan, K.; Cohen, H. VHSIC hardware description language. *Computer* **1985**, *18*, 94–103. [CrossRef]
10. Gupta, R.; Brewer, F. High-level synthesis: A retrospective. In *High-Level Synthesis: From Algorithm to Digital Circuit*; Springer: Dordrecht, The Netherlands, 2008; pp. 13–28.
11. Minsky, M.; Papert, S. *Perceptrons: An Introduction to Computational Geometry*; Massachusetts Institute of Technology: Cambridge, MA, USA, 1969; Volume 479, p. 104.
12. Rumelhart, D.E.; Hinton, G.E.; Williams, R.J. Learning representations by back-propagating errors. *Nature* **1986**, *323*, 533–536. [CrossRef]
13. Tesauro, G. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.* **1994**, *6*, 215–219. [CrossRef]
14. Dean, J. 1.1 The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design. In Proceedings of the 2020 IEEE International Solid-State Circuits Conference—(ISSCC), San Francisco, CA, USA, 16–20 February 2020; pp. 8–14. [CrossRef]
15. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Li, F.-F. Imagenet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 248–255.
16. Luebke, D.; Harris, M. General-purpose computation on graphics hardware. In Proceedings of the Workshop, SIGGRAPH, Los Angeles, CA, USA, 8–12 August 2004; Volume 33, p. 6.
17. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *Nature* **2015**, *521*, 436–444. [CrossRef] [PubMed]
18. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. *Adv. Neural Inf. Process. Syst.* **2012**, *25*, 1097–1105. [CrossRef]
19. Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; Rabinovich, A. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 1–9.
20. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
21. Tan, M.; Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In Proceedings of the International Conference on Machine Learning, PMLR, Long Beach, CA, USA, 9–15 June 2019; pp. 6105–6114.
22. Hinton, G.; Deng, L.; Yu, D.; Dahl, G.E.; Mohamed, A.r.; Jaitly, N.; Senior, A.; Vanhoucke, V.; Nguyen, P.; Sainath, T.N.; et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.* **2012**, *29*, 82–97. [CrossRef]
23. Chan, W.; Jaitly, N.; Le, Q.V.; Vinyals, O. Listen, attend and spell. *arXiv* **2015**, arXiv:1508.01211.
24. Wu, Y.; Schuster, M.; Chen, Z.; Le, Q.V.; Norouzi, M.; Macherey, W.; Krikun, M.; Cao, Y.; Gao, Q.; Macherey, K.; et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv* **2016**, arXiv:1609.08144.
25. Collobert, R.; Weston, J.; Bottou, L.; Karlen, M.; Kavukcuoglu, K.; Kuksa, P. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.* **2011**, *12*, 2493–2537.
26. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed representations of words and phrases and their compositionality. *Adv. Neural Inf. Process. Syst.* **2013**, *26*, 3111–3119.
27. Sutskever, I.; Vinyals, O.; Le, Q.V. Sequence to sequence learning with neural networks. *Adv. Neural Inf. Process. Syst.* **2014**, *27*, 3104–3112.
28. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 5998–6008.
29. Shazeer, N.; Mirhoseini, A.; Maziarz, K.; Davis, A.; Le, Q.; Hinton, G.; Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv* **2017**, arXiv:1701.06538.
30. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
31. Yu, C.; Xiao, H.; De Micheli, G. Developing synthesis flows without human knowledge. In Proceedings of the 55th Annual Design Automation Conference, San Francisco, CA, USA, 24–29 June 2018; pp. 1–6.

32. Huang, G.; Hu, J.; He, Y.; Liu, J.; Ma, M.; Shen, Z.; Wu, J.; Xu, Y.; Zhang, H.; Zhong, K.; et al. Machine learning for electronic design automation: A survey. *ACM Trans. Des. Autom. Electron. Syst.* **2021**, *26*, 1–46. [CrossRef]
33. Kahng, A.B. Machine learning applications in physical design: Recent results and directions. In Proceedings of the 2018 International Symposium on Physical Design, Monterey, CA, USA, 25–28 March 2018; pp. 68–73.
34. OpenAI. Introducing ChatGPT. 2022. Available online: https://openai.com/blog/chatgpt (accessed on 8 February 2024).
35. Pichai, S. An Important Next Step on Our AI Journey. 2023. Available online: https://blog.google/technology/ai/bard-google-ai-search-updates/ (accessed on 8 February 2024).
36. Microsoft. Microsoft Edge Features—Bing Chat. 2023. Available online: https://www.microsoft.com/en-us/edge/features/bing-chat?form=MT00D8 (accessed on 8 February 2024).
37. Chang, K.; Wang, Y.; Ren, H.; Wang, M.; Liang, S.; Han, Y.; Li, H.; Li, X. ChipGPT: How far are we from natural language hardware design. *arXiv* **2023**, arXiv:2305.14019.
38. Thakur, S.; Ahmad, B.; Fan, Z.; Pearce, H.; Tan, B.; Karri, R.; Dolan-Gavitt, B.; Garg, S. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In Proceedings of the 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 17–19 April 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 1–6.
39. Blocklove, J.; Garg, S.; Karri, R.; Pearce, H. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. *arXiv* **2023**, arXiv:2305.13243.
40. Efabless. Efabless Caravel "Harness" SoC—Caravel Harness Documentation. Available online: https://caravel-harness.readthedocs.io/en/latest/ (accessed on 8 February 2024).
41. Welcome to SkyWater SKY130 PDK's Documentation! Available online: https://skywater-pdk.readthedocs.io/en/main/ (accessed on 8 February 2024).
42. Srilakshmi, S.; Madhumati, G.L. A Comparative Analysis of HDL and HLS for Developing CNN Accelerators. In Proceedings of the 2023 Third International Conference on Artificial Intelligence and Smart Energy (ICAIS), Coimbatore, India, 2–4 February 2023; pp. 1060–1065.
43. Zhao, J.; Zhao, Y.; Li, H.; Zhang, Y.; Wu, L. HLS-Based FPGA Implementation of Convolutional Deep Belief Network for Signal Modulation Recognition. In Proceedings of the IGARSS 2020—2020 IEEE International Geoscience and Remote Sensing Symposium, Waikoloa, HI, USA, 26 September–2 October 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 6985–6988.
44. Lee, H.S.; Jeon, J.W. Comparison between HLS and HDL image processing in FPGAs. In Proceedings of the 2020 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), Seoul, Republic of Korea, 1–3 November 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–2.
45. Shalan, M.; Edwards, T. Building OpenLANE: A 130nm openroad-based tapeout-proven flow. In Proceedings of the 39th International Conference on Computer-Aided Design, San Diego, CA, USA, 2–5 November 2020; pp. 1–6.
46. Zezin, D. Modern Open Source IC Design tools for Electronics Engineer Education. In Proceedings of the 2022 VI International Conference on Information Technologies in Engineering Education (Inforino), Moscow, Russia, 12–15 April 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 1–4.
47. Charaan, S.; Nalinkumar, S.; Elavarasan, P.; Prakash, P.; Kasthuri, P. Design of an All-Digital Phase-locked loop in a 130 nm CMOS Process using open-source tools. In Proceedings of the 2022 International Conference on Electronic Systems and Intelligent Computing (ICESIC), Chennai, India, 22–23 April 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 270–274.
48. Ghazy, A.; Shalan, M. Openlane: The open-source digital asic implementation flow. In Proceedings of the Workshop on Open-Source EDA Technologies (WOSET), Online, 27 October 2020.
49. Chupilko, M.; Kamkin, A.; Smolov, S. Survey of Open-source Flows for Digital Hardware Design. In Proceedings of the 2021 Ivannikov Memorial Workshop (IVMEM), Nizhny Novgorod, Russia, 24–25 September 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 11–16.
50. Hesham, S.; Shalan, M.; El-Kharashi, M.W.; Dessouky, M. Digital ASIC Implementation of RISC-V: OpenLane and Commercial Approaches in Comparison. In Proceedings of the 2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), Lansing, MI, USA, 9–11 August 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 498–502.
51. Efabless. Homepage. Available online: https://efabless.com/ (accessed on 26 February 2024).
52. Deng, L. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Process. Mag.* **2012**, *29*, 141–142. [CrossRef]
53. Bisong, E. Google colaboratory. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*; Apress: Berkeley, CA, USA, 2019; pp. 59–64.