Combining Heterogeneous Compute Platforms

Thesis submitted by

Srijan Gupta
2020CS50444

under the guidance of

Prof. Kolin Paul

in partial fulfilment of the requirements
for the award of the degree of

Bachelor and Master of Technology

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY DELHI

June 2025

THESIS CERTIFICATE

This is to certify that the thesis titled **Combining Heterogeneous Compute Platforms**, submitted by **Srijan Gupta (2020CS50444)** to the Indian Institute of Technology, Delhi, for the award of the degree of **Bachelor and Master of Technology** in **Computer Science and Engineering**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in whole or in parts, have not been submitted to any other institution or University for the award of any degree or diploma.

Prof. Kolin Paul
Professor
Dept. of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, 110 016

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Prof. Kolin Paul, who helped introduce me to research in hardware design and FPGAs. I greatly appreciate the independence he gave me in working on this project, along with giving me constructive criticism and feedback when needed. Working with him over the past year has been a wonderful and enriching experience.

I also thank Ms. Manju Rani for ensuring I had no issues accessing the required hardware, be it an FPGA board, SD card(s), etc.

Finally, I am grateful to my family and friends for their unwavering support and constant motivation, making my journey enjoyable.

Srijan Gupta

ABSTRACT

We have demonstrated the merit of **combining heterogeneous computing platforms** using AMD's *ZCU102* development board. We show that properly splitting neural network layers between the board's compute platforms, i.e., **CPU** and **FPGA**, results in a better performance than completely implementing the network on either platform. We also compare the performance obtained with standard benchmarks for neural network deployment.

Contents

List of Tables

List of Figures

ABBREVIATIONS

GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array
AMD	Advanced Micro Devices
RTL	Register Transfer Level
HDL	Hardware Description Language
HLS	High-Level Synthesis
CNN	Convolutional Neural Network
ReLU	Rectified Linear Unit
MPSoC	Multi-Processor System on Chip
PS	Processing System
PL	Programmable Logic
SIMD	Single Instruction Multiple Data
XO	Xilinx Object
UART	Universal Asynchronous Receiver/Transmitter
DPU	Deep-Learning Processing Unit
PDM	Power Design Manager
VART	Vitis-AI Runtime

Chapter 1

Introduction

Computing workloads are growing daily, and with Moore's law finally giving way, one needs new hardware to achieve high performance on these significant problems. The most popular device for this is the GPU, which provides massive parallelism and a programming framework (CUDA) similar to standard programming. The major drawback of GPUs, however, is their power consumption.

Reconfigurable hardware arises as a possible family of devices that may be more power efficient than GPUs. However, reconfigurable hardware also has its shortcomings. The initial issue is the **low clock frequency** at which these devices operate. CPUs/GPUs run at clock frequencies of 1-3 GHz, whereas FPGA boards generally run at 100-400 MHz frequency ranges, maxing out at 600 MHz. Newer components like AI engines in AMD's Versal [1] devices may also operate at frequencies of 1 GHz. Therefore, achieving a speedup over a general-purpose implementation requires a cycle reduction of a factor of at least five.

Work exists on reconfigurable hardware performing at a higher power efficiency ([2], [3], [4], [5]) than GPU(s), with some also able to outperform the GPU(s) ([4], [5]). Outperforming GPUs is a difficult task simply due to the massive amounts of parallelism they provide; for example, an NVIDIA A40 GPU can have 2048 CUDA threads running at once, whereas a Versal VCK5000 device only has 400 AI engines, which compounds over the aforementioned frequency constraint.

The other major shortcoming, arguably the more critical issue, is the complexity in programming such devices. Programmers program FPGAs with RTL-level languages (also called HDLs). Compared to languages for CPUs (C, Python, etc.) or CUDA for GPUs, these languages are less intuitive and require more expertise.

To address the programming complexity, HLS tools have emerged in which one writes code in a high-level language (normally C/C++), and the HLS tool will automatically convert it into HDL code to program the FPGA. Similar to a compiler generating assembly, if one writes HDL code manually, it will likely be better than what an automated tool can generate; however, the unintuitiveness of these languages and relative difficulty in documentation and readability (similar to assembly) motivates programmers to proceed with HLS and try to make the HDL code generation algorithms better.

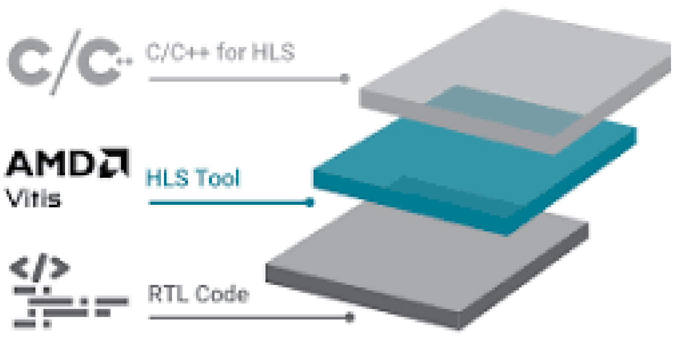


Figure 1.1: HLS tool flow

Given the drawbacks and strengths of these separate computing platforms, it makes sense that programmers choose to run portions of problems on different hardware architectures ([6], [7], [8], [9]). These works can demonstrate a higher performance achievable using just one architecture alone for computation.

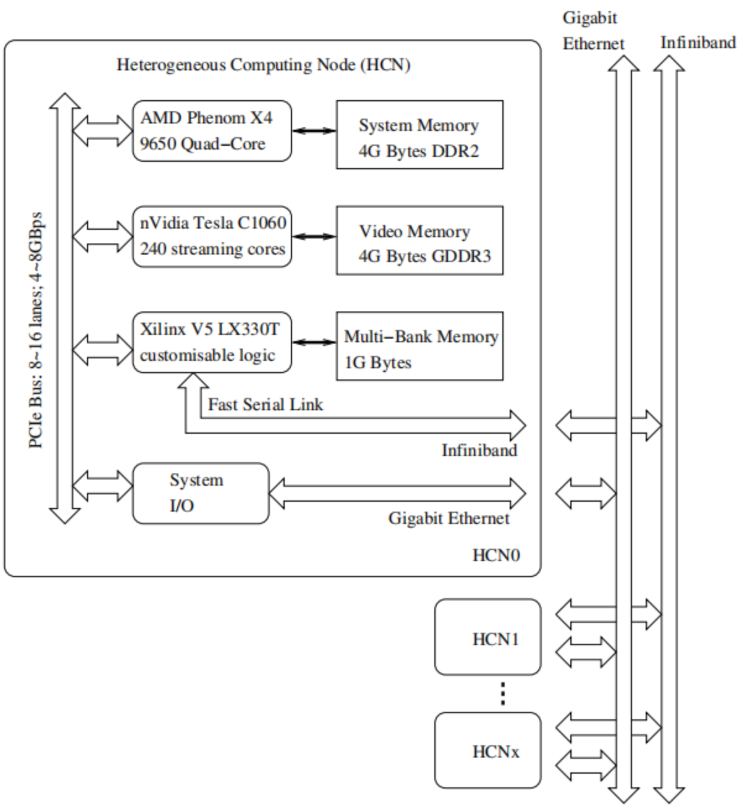

Gigabit Ethernet
Infiniband
Heterogeneous Computing Node (HCN)
PCIe Bus: 8~16 lanes; 4~8GBps
AMD Phenom X4 9650 Quad–Core
System Memory 4G Bytes DDR2
nVidia Tesla C1060 240 streaming cores
Video Memory 4G Bytes GDDR3
Xilinx V5 LX330T customisable logic
Multi–Bank Memory 1G Bytes
Fast Serial Link
Infiniband
System I/O
Gigabit Ethernet
HCN0
HCN1
HCNx

Figure 1.2: Heterogeneous architecture from [6]

The works above leave it entirely up to the programmer's intuition on how to split the problem. Though they can all achieve speedups, they do not explore other ways to divide computation. Testing different splitting methods may reveal previously unexplored performance and/or power optimization possibilities..

We demonstrate the merit of exploring this hypothesis further using a CNN [10] deployed on the ZCU102 [11] development board, which contains both a CPU and an FPGA as compute platforms. The performance obtained by splitting the problem is higher than what was obtainable using just the CPU or the FPGA. We also estimate the power consumption of our implementations to see how our partitioning affects energy efficiency.

The CNN we use is of a moderate size ($O(10^5)$ parameters), more designed for embedded devices (like the ZCU102), and we use a relatively small board for our tests. A smaller problem allows for easier exploration of division possibilities, and similarly, a smaller board allows one to develop and deploy designs quicker, and find and address bottlenecks.

Chapter 2

Background

2.1 Convolutional Neural Networks

A neural network is a machine learning algorithm. Neural networks comprise an input layer, various hidden layers, and an output layer. There are multiple types of neural networks, each used for different cases and data types. For example, recurrent neural networks are used for natural language processing, whereas convolutional neural networks [10] are utilized for image-based tasks.

Alongside the input and output layers, three main types of hidden layers are used in a CNN.

2.1.1 Convolutional Layer

A convolutional layer has the following components: input data, a filter/feature detector, and a feature map.

The feature detector is applied to an area of the input, and a dot product is calculated between the input and the filter. This dot product is then fed into an output array. The filter then shifts and repeats the process until the whole input has been traversed. The weights of the filter remain the same while traversing the input. These weights are adjusted during training. An activation function (generally ReLU) may be applied to the output array, which gives the final feature map.

Three hyperparameters affect the output size, which must be set before training.

- The **number of filters** determines the depth of the output.
- **Stride** is the distance the filter moves while traversing the input.
- **Padding** is used when the filters do not exactly fit with the input. Dummy elements outside the input range are created (generally set to 0), producing an equally sized or larger output.

2.1.2 Pooling Layer

Pooling, or downsampling, constitutes reducing the number of parameters in the input. Similar to a convolution layer, a filter traverses the input. However, instead of performing a dot product, it performs an aggregation operation over the area of the input under process.

Common pooling operations are:

- **Max Pooling:** The maximum value in the input area is output.
- **Average Pooling:** The average of all values in the input area is output.

Though information loss occurs in a pooling layer, it helps improve efficiency and prevent overfitting.

2.1.3 Fully Connected Layer

In a fully connected layer, every node in the output layer is connected to every node in the input layer. This layer is generally present after the above layers to perform the final classification task based on the extracted features.

The activation function used here is generally the softmax function, as it gives outputs between 0 and 1, showing the probability of each class.

It is essential to note that a CNN may have several more types of layers present depending on the task at hand.

2.2 AMD Zynq Ultrascale+ MPSoC ZCU102

The ZCU102 is a general-purpose evaluation board based on the Zynq Ultrascale+ XCZU9EG-2FFVB1156E MPSoC [11]. The board combines a powerful processing system (PS) and user-programmable logic (PL) on the same board.

The PS and PL can be coupled with multiple interfaces to integrate PL-based functions that are visible to the processors. Users may also use the same domain's PS I/O and PL I/O peripherals.

The ZCU102 board includes a secure digital input/output (SDIO) interface to provide access to general-purpose non-volatile SDIO memory cards and peripherals.

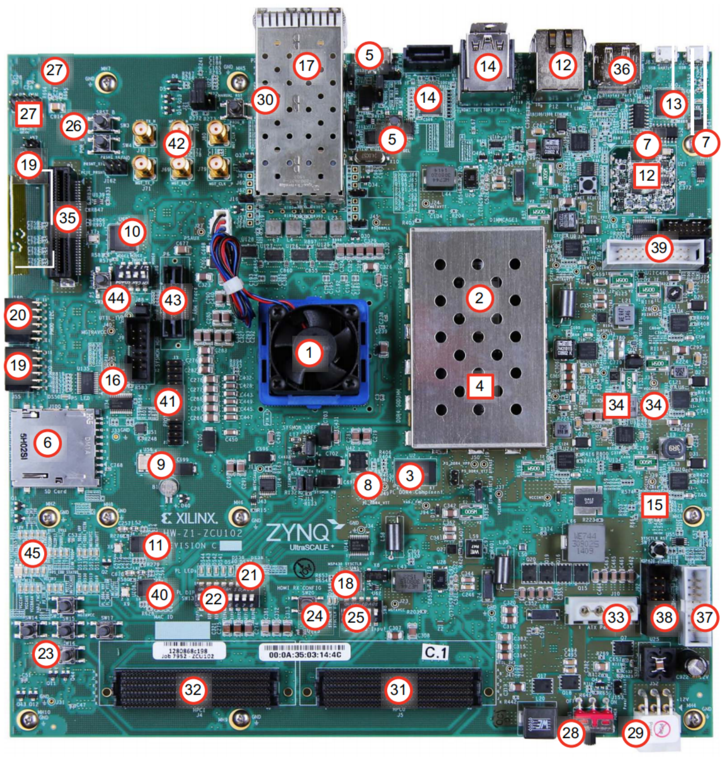


Figure 2.1: ZCU102 Evaluation Board

2.2.1 Processing System

The PS has three main processing units:

- Cortex-A53 application processing unit (APU)-Arm v8 architecture-based 64-bit quad-core multiprocessing CPU. Each core has 32 KB I/D caches and a shared 1 MB L2 cache.
- Cortex-R5 real-time processing unit (RPU)-Arm v7 architecture-based 32-bit dual real-time processing unit with dedicated tightly coupled memory (TCM).
- Mali-400 graphics processing unit (GPU)-graphics processing unit with pixel and geometry processor and 64 KB L2 cache.

The PS-side memory is a 4 GB, 64-bit DDR4 (Double Data Rate 4) SODIMM (Small Outline Dual In-line Memory Module).

The PS-side Gigabit Ethernet MAC (GEM) implements an Ethernet interface with 10/100/1000 Mb/s speeds.

The CP2108 quad USB-UART on the ZCU102 board provides four UART connections.

2.2.2 Programmable Logic

The PL of the ZCU102 is a large FPGA containing the following resources for the user to customize:

- The PL-side memory is a 4 Gb, 16-bit DDR4 memory system comprised of one 256 Mb x 16 SDRAM
- PL-side main memory is divided into several memory banks.
- **LUT (Lookup Table)**/Logic Cell: 599,550, capable of implementing any boolean function
- **FF (Flip Flop):** 548,160, used as storage units.
- **BRAM (Block RAM):** 912 cells of size 36 KB each, giving a total BRAM of 32.1 MB. Serves a similar purpose to a cache in standard hardware
- **DSP (Digital Signal Processor):** 2520, units which can perform parallel mathematical operations.

2.3 AMD Vitis Unified Software Platform

The **AMD Vitis Unified Software Platform** is a set of tools designed for FPGA and adaptive SoC development. This platform allows developers to compile, analyze, and debug applications for AMD's adaptive computing platforms without requiring extensive hardware expertise.

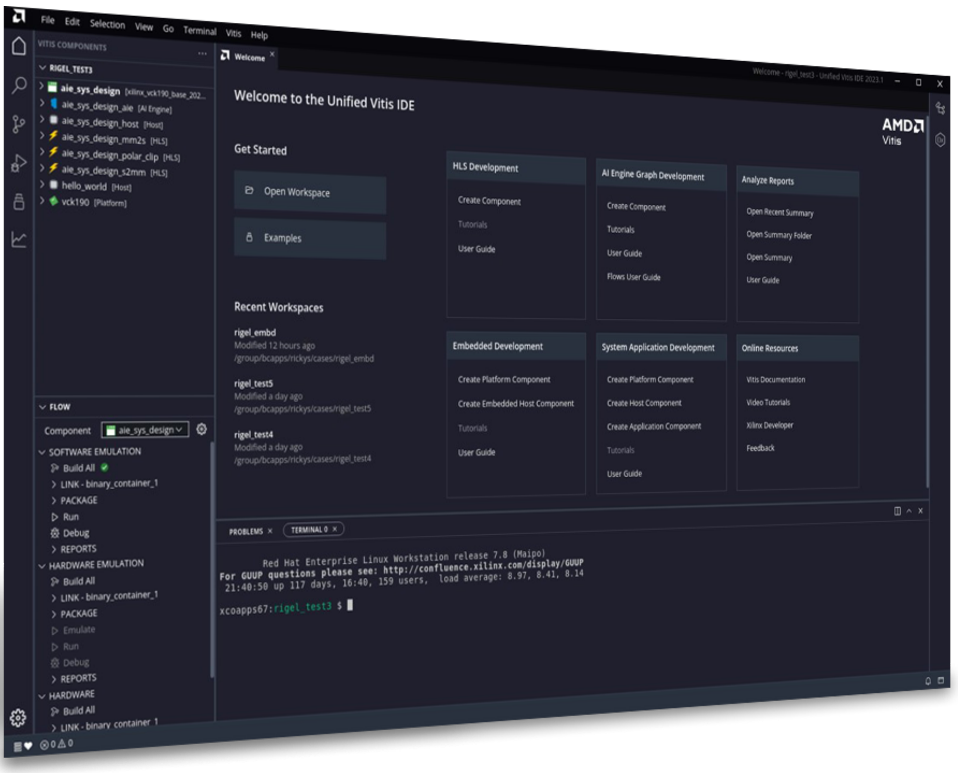
2.3.1 Components

Xilinx Runtime (XRT)

XRT is an open source library that provides a standard interface for communication between applications and FPGA kernels [12]. The runtime manages data movement between the available platforms while providing memory management and kernel control APIs.

Vitis Unified IDE

The Vitis Unified IDE [13] provides a graphical interface that unifies all available tools under the Vitis framework.



```
        Red Hat Enterprise Linux Workstation release 7.8 (Maipo)
For GUUP questions please see: http://confluence.xilinx.com/display/GUUP
 21:40:50 up 117 days, 16:40, 159 users,  load average: 8.97, 8.41, 8.14

xcoapps67:rigel_test3 $
```

Figure 2.2: Vitis Unified IDE

2.3.2 Tools

Vitis HLS

Vitis HLS [14] is a tool that compiles C/C++ code into kernels for acceleration in the PL region of AMD devices. This tool performs high-level synthesis to convert software algorithms into hardware implementations, allowing developers to create complex FPGA designs without RTL coding. It is then integrated with the Vivado design suite to generate designs for the hardware implementations.

Vivado Design Suite

The Vivado Design Suite is AMD Xilinx's FPGA development environment for design, synthesis, implementation, and debugging of AMD's adaptive computing devices [15].

Chapter 3

Model

The model used is a CNN meant to classify the images of the Fashion-MNIST dataset, i.e, the input is a grayscale image of size 28×28 and the output is a classification vector of size 10. This architecture was chosen due to the variety of layer types present, allowing for experimentation on their compatibility with the available PS and PL architectures.

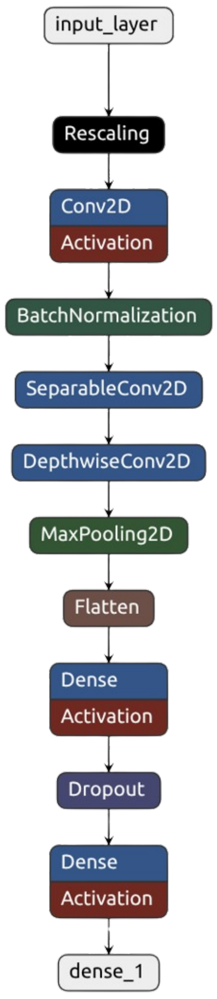


Figure 3.1: Model Used

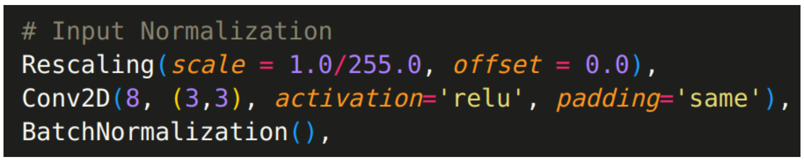
We obtained this model by slightly modifying a model suggested by querying **Perplexity** [16]. The AI did give some latency inference results, but we ignored those due to their ambiguous origin and the vast difference between them and the obtained results.

We trained the model on an NVIDIA A40 GPU [17] using the Python TensorFlow API [18]. The hyperparameters used are 15 epochs and a batch size of 64. The model had an accuracy of 95.5 % on the training data and a 91.0 % accuracy on the test data. We could have improved these metrics; however, the model's accuracy was not a priority, so we did not use further training optimization methods.

We divide the classification process into three phases:

3.1 Input Normalization

This phase introduces spatial context while maintaining a 1:1 input/output ratio to prevent information loss.



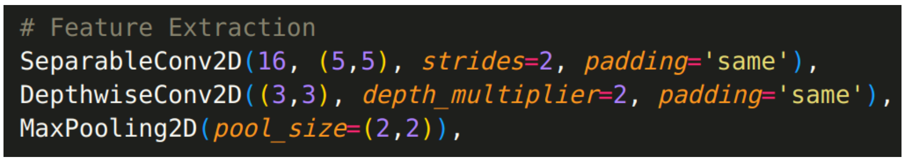
```
# Input Normalization
Rescaling(scale = 1.0/255.0, offset = 0.0),
Conv2D(8, (3,3), activation='relu', padding='same'),
BatchNormalization(),
```

Figure 3.2: Input Normalization

Instead of treating the `Conv2D` and `BatchNormalization` layers as separate entities, once the `Conv2D` layer finishes computing any element of its output, we can feed it to the `BatchNormalization`, allowing for easy pipelining.

3.2 Feature Extraction

Combines convolution types that facilitate spatial reduction and preserve feature diversity using depth multiplication. The model then pools the most *important* features.



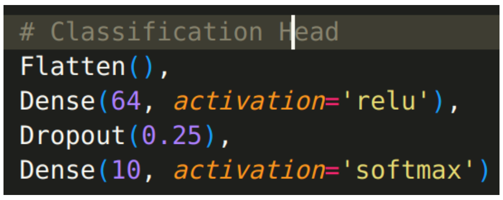
```
# Feature Extraction
SeparableConv2D(16, (5,5), strides=2, padding='same'),
DepthwiseConv2D((3,3), depth_multiplier=2, padding='same'),
MaxPooling2D(pool_size=(2,2)),
```

Figure 3.3: Feature Extraction

Similar to the previous phase, we could pipeline the layers present; however, unlike the previous phase, instead of allowing next layers to start after computing any output elements, we will need complete rows to be completed before the next layer can start due to the convolution/pooling access pattern.

3.3 Classification Head

The final phase takes the above extracted features and classifies the input, with the final layer giving the probabilities of the input belonging to each class.



```
# Classification Head
Flatten(),
Dense(64, activation='relu'),
Dropout(0.25),
Dense(10, activation='softmax')
```

Figure 3.4: Classification Head

`Dropout` layers are inactive during inference, so no implementation is required. Since we are coding in C/C++, we do not need to explicitly flatten the data received from the `Feature Extraction` phase. We can leverage SIMD capabilities to accelerate the `Dense` layers.

3.4 Layer Operation Counts

We run inference on one image at a time, so we neglect the batch size dimension in this analysis since it is always 1. Had it been used, one could simply multiply all the elements in the 'Operation Count' column by the batch size to obtain the new operation counts.

Layer	Input Shape	Output Shape	Parameter Shape(s)	Operation Count
Rescaling	$(28 \times 28 \times 1)$	$(28 \times 28 \times 1)$	-	784
Conv2D	$(28 \times 28 \times 1)$	$(28 \times 28 \times 8)$	$(3 \times 3 \times 1 \times 8)$ (8)	68992
BatchNormalization	$(28 \times 28 \times 8)$	$(28 \times 28 \times 8)$	(8) (8) (8) (8)	25088
SeparableConv2D	$(28 \times 28 \times 8)$	$(14 \times 14 \times 16)$	$(5 \times 5 \times 8 \times 1)$ $(1 \times 1 \times 8 \times 16)$ (16)	64288
DepthwiseConv2D	$(14 \times 14 \times 16)$	$(14 \times 14 \times 32)$	$(3 \times 3 \times 16 \times 2)$ (32)	56448
MaxPooling2D	$(14 \times 14 \times 32)$	$(7 \times 7 \times 32)$	-	6272
Flatten	$(7 \times 7 \times 32)$	(1568)	-	0
Dense	(1568)	(64)	(1568×64) (64)	100416
Dropout	(64)	(64)	-	0
Dense	(64)	(10)	(64×10) (10)	669

Table 3.1: Layer Complexities

Chapter 4

Hardware Design

4.1 PS Programming

The **Cortex-A53** is a quad-core CPU in which each core has one thread. We use a single core during evaluation. This is because a single compute unit is used in the PL, so we compare it to a single general-purpose compute unit.

The Cortex-A53 implements **NEON SIMD** technology, a 128-bit long vector instruction set, i.e., 4 32-bit float operations can happen simultaneously. We have written code to assist the compiler in inferring vector instructions via practices such as having continuous memory accesses in loops for vectorized load/stores.

In most cases, the `-O3` compiler flag is sufficient for `gcc` to generate vector instructions. However, NEON SIMD technology is not guaranteed to follow `IEEE 754` standards, which `gcc` prioritizes over vectorization. Therefore, the additional flag `-funsafe-math-optimizations` is required to ensure the compiler generates vector instructions for floating point computations.

The PS communicates with the PL using the XRT library. We store data that the PL requires in `xrt::bo` (buffer objects). Data shared repeatedly (inputs/outputs) is augmented with the `cacheable` flag to reduce communication overhead.

We use an `xrt::kernel` object to store a pointer to the function to be executed by the PL. This object is run by creating a `xrt::run` object, which loads the arguments of the kernel function and runs it on the PL. This run is asynchronous by nature, so we use the `wait()` method to synchronize PS/PL executions.

In the Vitis design flow, all PS-based code is written in an **Application** component.

4.2 FPGA Design

We use **Vitis HLS** to synthesize hardware implementations of the software kernels as an **HLS** component of the Vitis design flow. The compilation output is in XO (Xilinx Object) format. Vitsi then compiles this XO into an XCLBIN binary deployed to the FPGA device.

The arguments of the top-level function to synthesize are the inputs/outputs (may be intermediate layer outputs) and all network parameters required. The network parameters are constant here, but are not treated in that way to simulate a more practical situation where the parameters may change based on new data, which will raise issues if the parameters end up hard-coded.

Since top-level arguments are raw data, they can only be pointers or basic data types. Here, all arguments are pointers. The C-style arrays they point to are not the most efficient regarding access latency and parallelism. **Vitis HLS** provides specialized storage data types more amenable to FPGA synthesis and implementation A. We used `hls::stream` and `hls::vector`. Therefore, the PS transfers all parameters to objects of these classes before computation. The kernel computes the output and transfers it to the top-level output parameter. We use several compiler directives of the form `#pragma HLS <type> <args>`

The depth of `hls::stream` objects is set to the maximum number of elements they may hold theoretically. We completely partition Parameter arrays to allow for complete parallel access to components, and partially partition temporary arrays due to routing and resource issues caused by complete partitioning.

Loops which involve access of individual elements of `hls::vector` objects are entirely unrolled to exploit the parallelism provided by the data type. Vitis HLS tries to pipeline loops to improve performance, but sometimes fails to automatically infer a safe interval after which to start the next iteration. A value has been manually added to those loops when needed, and we turn off pipelining when this value becomes abnormally high.

Instead of bundling everything into the same function, we create separate functions with clear dependence relations so the HLS tool can use the `dataflow` methodology to schedule more portions of computation in parallel, and possibly even allow consumer functions to start before producer functions complete.

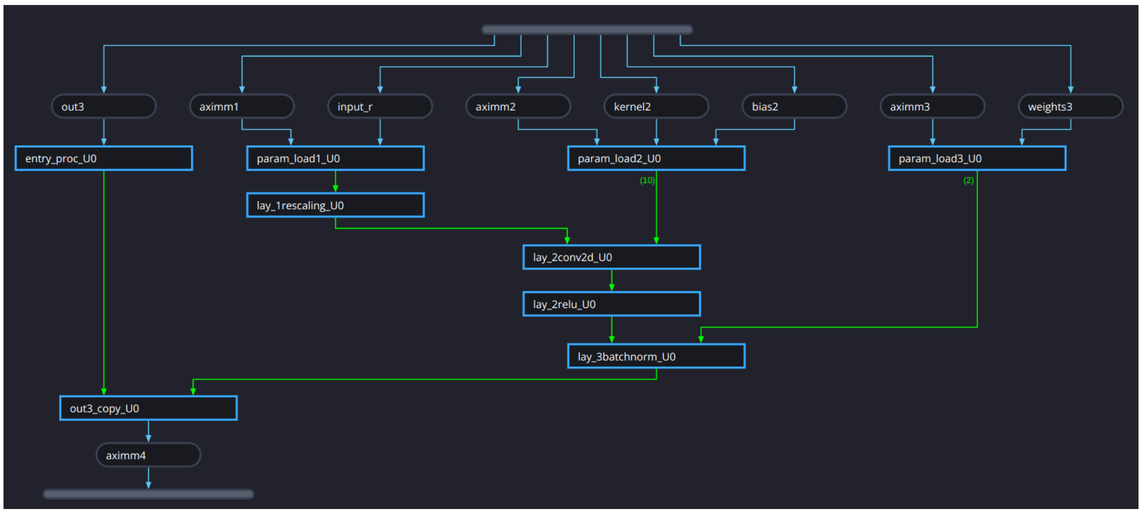


Figure 4.1: Dataflow Scheduling

To allow for parallel access of top-level parameters, they are assigned to separate memory banks. The HLS tool gives them separate group names.

The `dataflow` methodology imposes several constraints on designs when used. One must read top-level parameters assigned to the same memory bank in the same function; the same applies during writing. Any local function argument implemented as a FIFO interface A, be it an `hls::stream` or an array with the `STREAM` directive, must only be read in one function and written in one other function. This constraint allows the HLS tool to better determine producer/consumer relations and facilitates efficient FIFO use, i.e., the consumer may use data as soon as the producer gives it.

`dataflow` generally results in a higher resource utilization, as when functions have a producer/consumer relationship, they are almost running in parallel, so the functions cannot share resources, be it memory (BRAM) or compute units (DSP). The resource issue became a bottleneck for us when implementing the **Feature Extraction** phase, so much so that we could not split it into its layers due to either running out of resources or the tool's inability to route the complex designs formed.

We partition local arrays either completely or use partition factors that allow for parallel access in successive iterations, depending on the access pattern.

We were able to obtain that the maximum admissible frequency for our designs on the PL was 300 MHz.

Appendix A contains a detailed explanation of the HLS-specific data types and compiler directives used.

4.3 Integration

We use a **System Project** component provided by the Vitis design flow to deploy our implementations on the board.

The system project contains both the HLS and application components, it creates the `.xclbin` file mentioned above makes a boot image we flash onto an SD card.

Though we assign top-level parameters to separate memory banks, the tool assigns them all to the same bank by default while creating the `.xclbin`. Therefore, we manually specify the bank for each argument, ensuring consistency with the initial assignment.

Vitis HLS also provides options to further optimize the hardware after synthesis into an `XO` object (analogous to *IR* in compilers), of which we use the `-O3` option, which provides the most hardware optimization, similar to the flag in `gcc`.

Chapter 5

Evaluation

5.1 Experimental Setup

The ZCU102 board is connected via its USB-UART port to a desktop to display the CLI of the booted OS, and an external Ethernet port to allow for remote communication with the system.

The boot mode of the board is SD-boot mode, with the required files (binary container, inputs, weights, etc.) stored in the boot partition, and a separate partition containing the filesystem. We use the PetaLinux operating system on the board, detailed in Appendix B.

The input file contains 100 images stored in it (in binary form), which we read, compute the inference, and compare the inferred class with the model's expected output. We measure the runtime of each inference using the built-in `std::chrono` library, with the measurement unit being microseconds (μs).

We run each test five times. We convert the average inference latency from μs to ms for consistency with neural network latency reports.

5.2 Cases Considered

One may compute the model's three phases on either the PS or the PL. A total of eight cases arise from this.

A three-bit number represents each case. Input normalization, feature extraction, and classification head correspond to the 0^{th}, 1^{st}, and 2^{nd} bits, respectively. A 0 or 1 represents computing the corresponding PS or PL phase.

We expect the classification head phase to perform better on the PS due to the relatively small sizes of the matrices involved, so a large portion of the weight matrices may be present in the cache simultaneously, allowing the higher frequency PS to perform at a high efficiency. Therefore, we expect case 3 (011) to be the fastest.

Case 5 (101) is expected to be the slowest as it involves sending and receiving data to/from the PL twice as opposed to never in case 0 and once in the other cases.

5.3 Benchmarks

5.3.1 Vitis-AI (DPU)

Vitis-AI [19] is an open-source [20] framework for deploying deep learning models on AMD's adaptive computing boards. This framework does involve problem partition; however, it is need-based rather than performance-based. The reason is that not all commonly used model layers are supported by the DPU generated on the PL of the board, forcing developers to perform some computations on the CPU. This framework also requires the model to be quantized, which is, in essence, changing the domain of the initial problem statement and retraining the model to maintain accuracy.

We expect our implementations to at least provide comparable performance to Vitis-AI. We hesitate to expect speedup because of the integer operations' inherent higher performance than floating-point operations. Therefore, we treat this as our main benchmark for comparison.

5.3.2 GPU

We implemented a **CUDA**-based inference implementation involving a similar I/O strategy for the same NVIDIA **A40 GPU** on which we trained the model. The implementation used 1024 **CUDA** threads.

We do not expect our implementations to outperform a GPU simply due to the massive amount of parallelism (1024 threads) on the GPU compared to the maximum achievable on the ZCU102, four parallel floating-point operations on the PS, and a maximum of 128 parallel floating-point operations on the PL (which requires significant data manipulation). Another reason that compounds on top of this is the difference in frequency. The A40 GPU runs at a frequency of 1.74 GHz, whereas the PS runs at 1.2 GHz and the PL at 300 MHz.

5.4 Results

Case	Inference Latency (ms)
DPU	0.697612
0	0.890686
1	0.843006
2	0.641896
3	0.521244
4	0.818262
5	0.785238
6	0.612966
7	0.67664
GPU	0.0705

Table 5.1: Speed Results

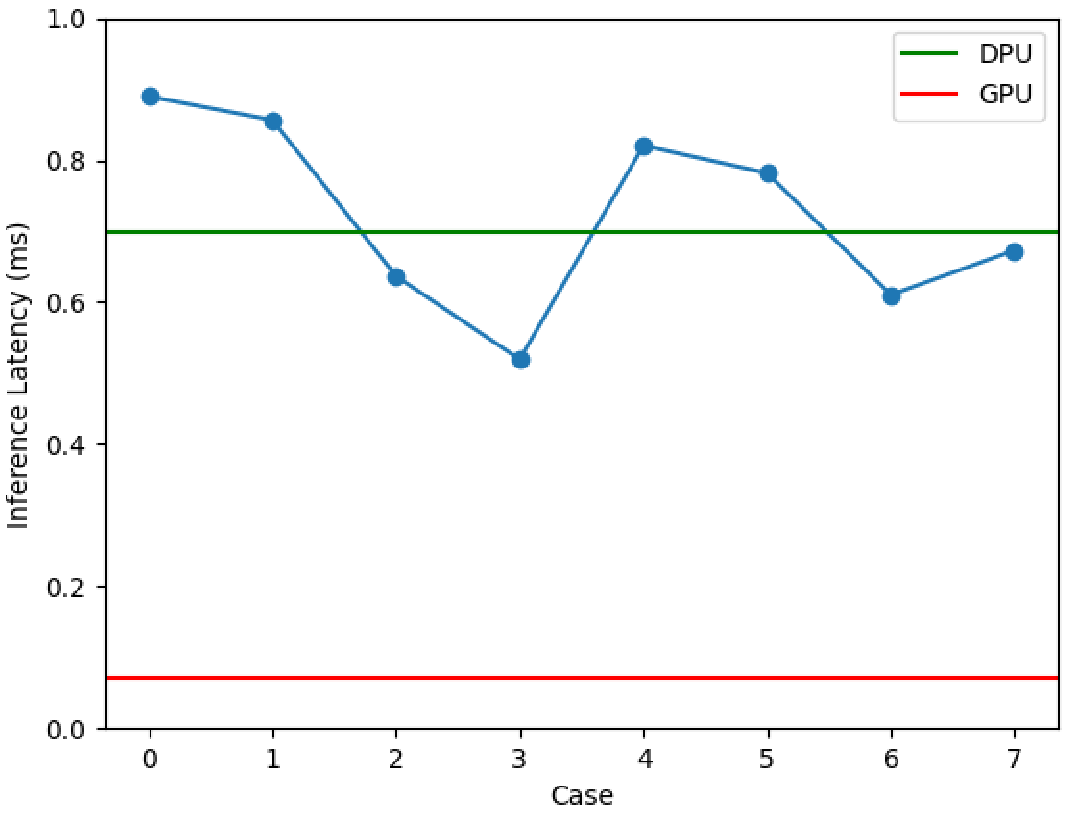


Figure 5.1: Inference Latencies

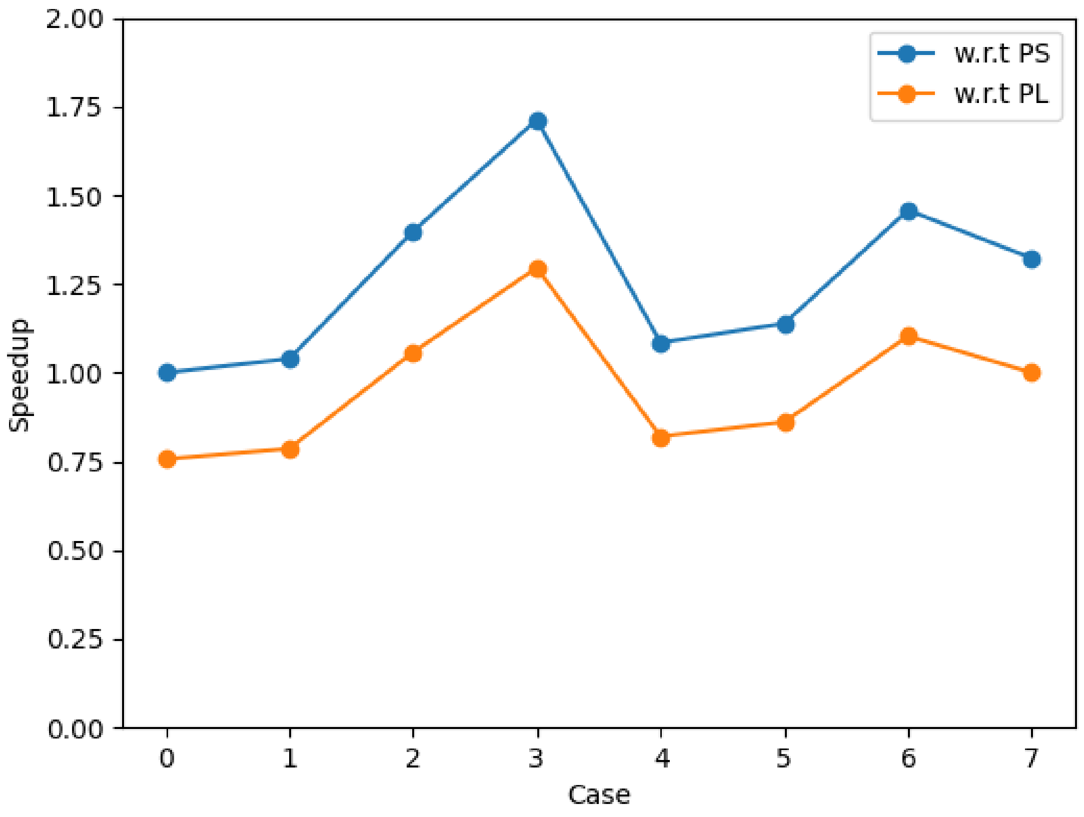


Figure 5.2: Speedup Achieved

The data proves our initial hypothesis that splitting the network among the PS and the PL could lead to better performance than achievable by either computing alone. As visible in 5.2, cases 2, 3, and 6 have lower latencies than the 'pure' implementations (cases 0 and 7) and the Vitis-AI DPU-based implementation.

We could not apply all the optimizations used in cases 1-6 to case 7 due to the increase in complexity of the design, leading to either resource exhaustion or failure in logic routing. Therefore, case 7 is less optimized compared to other cases.

As expected, we could not even come close to the performance of a (datacenter style) GPU, which is not surprising given our use of an embedded device.

The hypothesis on the best-performing case turned out to be right, i.e., case 3 had the lowest latency, though the guess for the worst case was wrong. Case 5 outperformed both case 2 and case 4, likely because the designs for the **input normalization** and **classification head** phases were smaller compared to **feature extraction**, and we could optimize separately without leading to implementation failures. Therefore, it overcame the communication overhead theorized to be a bottleneck.

Chapter 6

Power Estimation

6.1 AMD Power Design Manager

The AMD Power Design Manager (PDM) [21] is a tool in the Vitis framework used to estimate the power consumption of designs created for AMD's adaptive computing devices. Vivado does have a 'report power' option to generate a power estimate, in the form of a `.xpe` file, however, AMD recommends using PDM as it gives better power estimates compared to Vivado. To simplify the use of PDM, it allows for importing the `.xpe` file generated for a design, so a user does not have to specify the whole configuration manually.

6.2 Estimated Results

We had to settle for power estimation tools instead of simply profiling the design itself because the ZCU102 board does not support power profiling.

Vitis-AI only provides the final `.xclbin` file and not the complete design, and due to the aforementioned issue we were unable to provide any power estimates/measurements for it.

We obtained the power consumption of the GPU using the command `nvidia-smi`

Case	Estimated Power (W)	Inference Energy (mJ)
0	3.446	3.065451328
1	5.312	4.548846208
2	10.92	6.95073288
3	12.339	6.409444194
4	4.73	3.87972574
5	6.763	5.285460338
6	11.845	7.22800852
7	10.221	6.877240734
GPU	73.41	5.175405

Table 6.1: Power Results

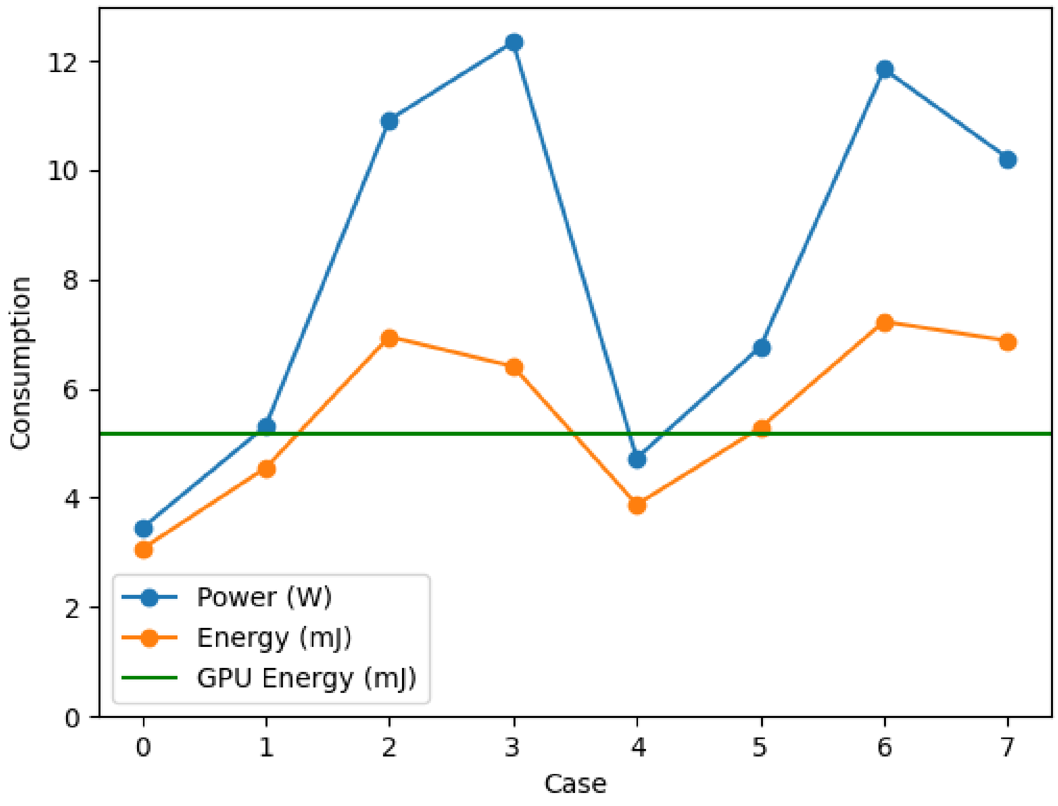


Figure 6.1: Power and Energy Consumption

As expected when comparing a GPU and an embedded device, all the ZCU102 implementations are far more power efficient than the GPU. However, since the GPU is much faster than our board, it is more energy efficient than in five of the eight cases. Unexpectedly, the three more energy-efficient cases than the GPU have the worst performance.

We expected case 0 to be the most efficient because the PS configuration does not change, so the estimated power of the PS remains constant, with PDM adding the estimations for the PL onto the same.

Even though we were able to speed up the inference compared to case 0, we could not make it fast enough to make it as/more energy efficient as not using the PL.

Chapter 7

Conclusions, Limitations, and Future Work

Using the PS and PL of AMD's ZCU102 board, we have shown that dividing up a problem and testing different cases for assigning each part to an available hardware architecture is a strategy that merits further exploration.

We obtained several cases where partitioning allowed us to perform better than we could using just the PS or the PL. In the best case, we received a maximum speedup of 1.72× over the PS and 1.29× over the PL.

Our most efficient partition is able to provide a 1.34× speedup over a Vitis-AI implementation of the same model, which is the standard for deploying models on AMD's adaptive computing devices. This is significant because Vitis-AI quantized the model to use integer operations, which are both faster and less resource-intensive compared to the floating-point operations of our implementation(s).

Though this result is a notable milestone, there are some asterisks behind it. Vitis-AI did not support running the `DepthwiseConv2D` layer on the DPU, forcing the framework to split up the model and perform some computations on the CPU, and increase the amount of PS-PL communication. The latency of the VART [19] communication APIs may have also been a factor due to the complex format in which the data is transferred, as opposed to the raw data transfer we used via the XRT library.

Our approach is not without limitations. We chose the partitions of the problem manually. This may not always be possible. For example, problems involving large amounts of communication may not have visible boundaries where one can decide to switch architectures.

We could not test several optimization approaches due to either resource exhaustion or logic routing failures. These constraints could be part of a 'partition optimization' problem.

Though our approach improved the performance of the CNN, it degraded its energy efficiency, sacrificing energy for speed. 'Partition optimization' should include power and energy as part of the problem constraints.

Future work involves developing heuristics to improve the quality of results of the 'partition optimization' problem, avoiding having to run the available possibilities due to an exponential blowup in their count as problem size increases.

As we succeeded in partitioning problems for embedded devices, this approach should be extended to combine larger devices like GPUs and AMD's Versal boards, where factors like power and energy efficiency are far more significant. The approach for these devices will be more complex compared to our approach, as the extra factor of how and how many of the available compute units to use will arise.

Appendix A

HLS Data Types and Pragmas

Vitis HLS provides many FPGA-specific constructs in the `hls` namespace, allowing for more efficient implementations on the platforms it supports. Documentation of these is available at [14].

A.1 Data Types

Vitis provides certain storage data types more amenable to FPGA synthesis than standard C-style arrays or C++ containers.

A.1.1 stream⟨T⟩

Vitis HLS provides a C++ template class `hls::stream<T>` to model streaming data structures. They are read from and written to sequentially, i.e., after reading data from a stream, one may not reread it.

`T` specifies the data type of the stream. A second optional parameter `N` specifies the depth of the FIFO. Verification uses this specified depth, whereas one should set the depth for synthesis using the `STREAM` directive

By default, the implementation of a `stream` object ais an AXI4-Stream Interface `(axis)` in the Vitis kernel flow.

A.1.2 vector⟨T, N⟩

The vector data type is used to model and synthesize SIMD-type vector operations easily. The parameter `T` specifies a data type with certain operations defined for it, and `N` is an integer specifying the number of elements in the vector.

For any vector of the type `hls::vector<T,N>`, the storage is guaranteed to be contiguous of size `sizeof(T)*N` and aligned to the next power of 2. A vector performs the best when both `sizeof(T)` and `N` are powers of 2.

After declaration, a vector can be treated as a primitive variable to perform arithmetic and logic operations.

A.2 Pragmas

A `#pragma` statement in C/C++ code is a compiler-specific directive that tells the compiler to perform implementation-specific tasks. Vitis HLS supports several compiler `#pragmas` with them taking the form `#pragma HLS <type> <args>`.

A.2.1 ARRAY_PARTITION

This directive partitions an array into smaller arrays or individual elements. It effectively provides more read and write ports, possibly increasing the throughput of the design, while requiring more memory instances or registers.

The syntax for the ARRAY_PARTITION directive is:

```
#pragma HLS ARRAY_PARTITION variable=<name> type=<type> factor=<int>
                          dim=<int> off=true
```

A.2.2 DATAFLOW

This directive enables task-level parallelism, allowing functions and loops to overlap during execution, increasing the concurrency and throughput of the design.

The syntax for the DATAFLOW directive is:

```
#pragma HLS DATAFLOW [disable_start_propagation]
```

A.2.3 INTERFACE

This directive is only supported for top-level function arguments and specifies how Vitis HLS creates ports from the function arguments during synthesis.

The syntax for the INTERFACE directive is:

```
#pragma HLS INTERFACE mode=<mode> port=<name> direct_io=<value>[OPTIONS]
```

A.2.4 PIPELINE

This directive reduces the initiation interval (II) of a function/loop by allowing iterations to execute concurrently. A pipelined function/loop can process new inputs every N cycles, where N is the II of the function/loop.

The syntax for the PIPELINE directive is:

```
#pragma HLS PIPELINE II=<int> off rewind style=<value>
```

A.2.5 STREAM

By default, array variables are implemented in RAM. If the data in the array is accessed sequentially, a more efficient implementation is to use streaming data, where FIFOs are used instead of RAMs. It may also specify the depth of `hls::stream` objects.

The syntax for the STREAM directive is:

```
#pragma HLS STREAM variable=<variable> type=<type> depth=<int>
```

A.2.6 UNROLL

One may unroll loops to create multiple independent operations rather than a single collection of operations. This directive creates multiple copies of the loop body, allowing some/all iterations to occur in parallel.

The syntax for the UNROLL directive is:

```
#pragma HLS UNROLL factor=<N> skip_exit_check off=true
```

Appendix B

PetaLinux

PetaLinux [22] is an embedded Linux Software Development Kit (SDK) targeting FPGA-based system-on-a-chip (SoC) designs or FPGA designs.

The PetaLinux tool contains the following:

- **Yocto Extensible SDK (eSDK)**: This component contains architecture-specific scripts (based on the Yocto Project [23]) to build a Linux-based system of the specified architecture. The supported architectures are `aarch64`, `arm` and `microblaze`.
- **XSCT and Toolchains**: PetaLinux uses XSCT [24] to configure and build all embedded software applications.
- **PetaLinux Command Line Interface (CLI) tools**: Contains all the required commands. The available commands are:
  - `petalinux-create`
  - `petalinux-config`
  - `petalinux-build`
  - `petalinux-util`
  - `petalinux-package`
  - `petalinux-upgrade`
  - `petalinux-devtool`
  - `petalinux-boot`

References

[1] Versal Adaptive SoC Technical Reference Manual (AM011) https://docs.amd.com/r/en-US/am011-versal-acap-trm

[2] Nick Brown. Exploring the Versal AI Engines for Accelerating Stencil-based Atmospheric Advection Simulation. *2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*

[3] Zhuoping Yang, Jinming Zhuang, Jiaqi Yin, Cunxi Yu, Alex K. Jones, and Peipei Zhou. AIM: Accelerating Arbitrary-precision Integer Multiplication on Heterogeneous Reconfigurable Computing Platform Versal ACAP. *2023 IEEE International Conference on Computer-Aided Design*

[4] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. *2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*

[5] Jinming Zhuang, Zhuoping Yang, Shixin Ji, Heng Huang, Alex K. Jones, Jingtong Hu, Yiyu Shi, and Peipei Zhou. SSR: Spatial Sequential Hybrid Architecture for Latency Throughput Tradeoff in Transformer Acceleration. *2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*

[6] Kuen Hung Tsoi, and Wayne Luk. Axel: A Heterogeneous Cluster with FPGAs and GPUs. *2010 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*

[7] Walther Carballo-Hernández, Maxime Pelcat, and François Berry. Why is FPGA-GPU Heterogeneity the Best Option for Embedded Deep Neural Networks? *DATE Friday Workshop on System-level Design Methods for Deep Learning on Heterogeneous Architectures (SLOHA 2021)*

[8] Bruno da Silva, An Braeken, Erik H. D'Hollander, Abdellah Touhafi, Jan G. Cornelis, and Jan Lemeire. Comparing and Combining GPU and FPGA Accelerators in an Image Processing Context. *2013 23rd International Conference on Field programmable Logic and Applications*

[9] Wentao Liang, Norihisa Fujita, Ryohei Kobayashi, and Taisuke Boku. Using Intel oneAPI for Multi-hybrid Acceleration Programming with GPU and FPGA Coupling. *2024 International Conference on High Performance Computing in Asia-Pacific Region*

[10] What are convolutional neural networks? https://www.ibm.com/think/topics/convolutional-neural-networks

[11] ZCU102 Evaluation Board User Guide (UG1182) https://docs.amd.com/v/u/en-US/ug1182-zcu102-eval-bd

[12] Xilinx Runtime (XRT) Architecture https://xilinx.github.io/XRT/master/html/index.html

[13] Vitis Reference Guide (UG1702) https://docs.amd.com/r/en-US/ug1702-vitis-accelerated-reference

[14] Vitis High-Level Synthesis User Guide (UG1399) https://docs.amd.com/r/en-US/ug1399-vitis-hls

[15] Vivado Design Suite User Guide: Using the Vivado IDE (UG893) https://docs.amd.com/r/en-US/ug893-vivado-ide

[16] Give a basic neural network model useful for benchmarking embedded devices https://www.perplexity.ai/search/give-a-basic-neural-network-mo-d.8NYBC_R4qboVgqwsl_6g

[17] NVIDIA A40 Data Center GPU for Visual Computing | NVIDIA https://www.nvidia.com/en-in/data-center/a40/

[18] Module: tf | TensorFlow v2.16.1 https://www.tensorflow.org/api_docs/python/tf

[19] Vitis AI — Vitis AI 3.0 documentation https://xilinx.github.io/Vitis-AI/3.0/html/index.html

[20] GitHub - Xilinx/Vitis-AI: Vitis AI is Xilinx's development stack for AI inference on Xilinx hardware platforms, including both edge devices and Alveo cards. https://github.com/Xilinx/Vitis-AI

[21] Power Design Manager User Guide (UG1556) https://docs.amd.com/r/en-US/ug1556-power-design-manager

[22] PetaLinux Tools Documentation: Reference Guide (UG1144) https://docs.amd.com/r/en-US/ug1144-petalinux-tools-reference-guide

[23] Yocto Project Reference Manual — The Yocto Project 5.2.999 documentation https://docs.yoctoproject.org/ref-manual/index.html

[24] Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400) https://docs.amd.com/r/en-US/ug1400-vitis-embedded