

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA ĐIỆN - ĐIỆN TỬ
BỘ MÔN VIỄN THÔNG

—00—



ĐỒ ÁN TỐT NGHIỆP

ĐỀ TÀI:

NHẬN DIỆN BIẾN BÁO GIAO THÔNG
TRÊN FPGA PYNQ Z2

GVHD : TS. VÕ QUẾ SƠN
SVTH : HUỲNH THỊNH PHÁT
MSSV : 2114369

TP.HỒ CHÍ MINH, THÁNG 4/2025

Số: _____/BKDT

Khoa: Điện - Điện tử

Bộ Môn: Viễn thông

NHIỆM VỤ LUẬN VĂN TỐT NGHIỆP

1. HỌ VÀ TÊN: Huỳnh Thịnh Phát MSSV: 2114369
2. NGÀNH: ĐIỆN TỬ - VIỄN THÔNG LỚP: DD21DV3
3. Đề tài: Nhận diện biển báo giao thông trên FPGA PYNQ Z2
4. Nhiệm vụ (Yêu cầu về nội dung và số liệu ban đầu):
 - Tìm hiểu về quy trình thiết kế trên FPGA.
 - Nghiên cứu về CNN, BNN và ứng dụng trong Objecs Detection.
 - Triển khai CNN, BNN trên FPGA PYNQ Z2.
 - Dánh giá về kết quả chạy thực nghiệm.
5. Ngày giao nhiệm vụ luận văn: 03/02/2025
6. Ngày hoàn thành nhiệm vụ: 26/04/2025
7. Họ và tên người hướng dẫn: Phần hướng dẫn
TS. Võ Quê Sơn _____

Nội dung và yêu cầu LVTN đã được thông qua Bộ Môn.

Tp.HCM, ngày tháng năm 2025

CHỦ NHIỆM BỘ MÔN

NGƯỜI HƯỚNG DẪN CHÍNH

PHẦN DÀNH CHO KHOA, BỘ MÔN:

Người duyệt (chấm sơ bộ): _____

Đơn vị: _____

Ngày bảo vệ: _____

Điểm tổng kết: _____

Nơi lưu trữ luận văn: _____

**KHÓA LUẬN TỐT NGHIỆP ĐƯỢC HOÀN THÀNH TẠI
TRƯỜNG ĐẠI HỌC BÁCH KHOA - ĐHQG - HCM**

Cán bộ hướng dẫn Khóa luận tốt nghiệp: TS. Võ Quê Sơn

Cán bộ chấm phản biện: PGS.TS. Hồ Văn Khuong

Khóa luận tốt nghiệp được bảo vệ tại Trường Đại học Bách Khoa, ĐHQG Tp.HCM
ngày 30 tháng 5 năm 2025.

Thành phần Hội đồng đánh giá khóa luận tốt nghiệp gồm:

(Ghi rõ họ, tên, học hàm, học vị của Hội đồng chấm bảo vệ khóa luận tốt nghiệp)

1. PGS.TS. Hồ Văn Khuong
2. TS. Nguyễn Chí Ngọc
3. TS. Võ Quê Sơn

Xác nhận của Chủ tịch Hội đồng đánh giá khóa luận tốt nghiệp và Chủ nhiệm Bộ môn sau khi
luận văn đã được sửa chữa (nếu có).

CHỦ TỊCH HỘI ĐỒNG

CHỦ NHIỆM BỘ MÔN VIỄN THÔNG

LỜI NÓI ĐẦU

Trong bối cảnh công nghệ phát triển mạnh mẽ hiện nay, các hệ thống giao thông thông minh ngày càng trở thành một yếu tố quan trọng trong việc đảm bảo an toàn và hiệu quả cho người tham gia giao thông. Một trong những ứng dụng then chốt của hệ thống giao thông thông minh là nhận diện biển báo giao thông. Việc phát triển hệ thống nhận diện biển báo giao thông tự động không chỉ giúp nâng cao mức độ an toàn mà còn góp phần cải thiện hiệu quả giao thông trong các thành phố lớn.

Với mong muốn đóng góp vào sự phát triển của các ứng dụng trong lĩnh vực giao thông thông minh, em lựa chọn đề tài "Nhận diện biển báo giao thông bằng FPGA PYNQ-Z2". Đề tài này tập trung vào việc nghiên cứu và ứng dụng nền tảng FPGA PYNQ-Z2 để xây dựng một hệ thống nhận diện biển báo giao thông hiệu quả, sử dụng các phương pháp tối ưu cho xử lý hình ảnh và phân loại biển báo.

PYNQ-Z2 là một nền tảng mạnh mẽ, hỗ trợ các công cụ và tài nguyên cần thiết để thiết kế và triển khai các ứng dụng vi mạch số. Đề tài này sẽ giới thiệu các phương pháp và thuật toán để nhận diện biển báo giao thông trên FPGA, từ đó đánh giá và tối ưu các giải pháp về hiệu suất, chi phí và độ chính xác của hệ thống.

Em xin gửi lời cảm ơn chân thành đến thầy Võ Quế Sơn vì đã nhiệt tình hướng dẫn và hỗ trợ em trong suốt quá trình thực hiện đề tài này. Nhờ sự giúp đỡ tận tâm của thầy, em đã có thể hoàn thành đề tài và hy vọng rằng kết quả nghiên cứu sẽ góp phần vào sự phát triển của các hệ thống giao thông thông minh trong tương lai.

TP. Hồ Chí Minh, ngày 25 tháng 5 năm 2025
Huỳnh Thịnh Phát

Mục lục

Lời nói đầu	2
Danh sách bảng	6
Danh sách hình vẽ	8
1 GIỚI THIỆU	9
1.1 Giới thiệu đề tài	9
1.2 Cấu trúc của bài báo cáo	10
1.3 Mục tiêu đề tài	10
2 CƠ SỞ LÝ THUYẾT	11
2.1 Giới thiệu về mô hình học sâu (Deep Learning Models)	11
2.2 Giới thiệu về Convolutional Neural Networks - CNN	12
2.2.1 Cấu trúc của CNN:	12
2.2.2 Lợi ích của CNN:	16
2.2.3 Ứng dụng của CNN:	16
2.3 Giới thiệu về Binarized Neural Networks - BNN	17
2.3.1 Các khái niệm cơ bản trong BNN	18
2.3.2 Cách BNN hoạt động	19
2.3.3 Lợi ích của BNN	20
2.3.4 Hạn chế của BNN	20
2.3.5 Ứng dụng của BNN	21
2.4 Tổng quan về FPGA	21
2.4.1 Ưu điểm của FPGA	22
2.4.2 Nhược điểm của FPGA	22
2.4.3 Ứng dụng của FPGA	23
2.5 Tổng quan về kit FPGA PYNQ-Z2	23
2.5.1 Thông số kỹ thuật kit PYNQ-Z2	24
2.5.2 Lợi ích và ứng dụng của PYNQ Z2	25
2.5.3 PYNQ Framework (PYNQ Python API)	26
2.5.4 Triển khai CNN trên PYNQ Z2	26
2.5.5 Cách kết nối kit PYNQ-Z2	27
2.6 Bộ dữ liệu GTSRB (German Traffic Sign Recognition Benchmark)	29
3 Huấn luyện tập dữ liệu với Python	31
3.1 Sơ đồ giải thuật tổng quát	31
3.2 Core Quantization (quantization.py)	32
3.3 Network Layers (quantized_net.py)	34

3.4	Network Architectures (cnv.py)	35
3.4.1	Mục đích thiết kế	35
3.4.2	Kiến trúc mạng CNN thiết kế	36
3.4.3	Các tham số quan trọng	36
3.4.4	Chức năng lượng tử hoá	36
3.5	Data Pipeline (gtsrb.py, augmentors.py, readTrafficSigns.py)	37
3.5.1	Đọc ảnh và nhãn từ file CSV (readTrafficSigns.py)	38
3.5.2	Xoay và cắt ngẫu nhiên (augmentors.py)	38
3.5.3	Khôi tiền xử lý (gtsrb.py)	39
3.6	Weight Conversion (gtsrb-gen-binary-weights.py, finnthesizer.py)	40
3.6.1	Thư viện finnthesizer.py:	41
3.6.2	gtsrb-gen-binary-weights.py	41
3.6.3	Training	42
3.6.4	Các kết quả sau huấn luyện	43
4	Thiết kế kiến trúc HLS CNN	45
4.1	Thiết kế CNV BNN cho Hardware	46
4.1.1	Kiến trúc HLS CNV BNN cho Hardware	46
4.1.2	Lập sơ đồ giải thuật	47
4.1.3	Chức năng của các hàm chính	48
4.1.4	Chi tiết mã nguồn sử dụng	48
4.1.5	Các thư viện và tệp kèm được sử dụng	50
4.2	Thiết kế CNV BNN cho Software	52
4.2.1	Sơ đồ giải thuật kiến trúc HLS CNV BNN cho Software	52
4.2.2	Tổng quan hệ thống	53
4.2.3	Chi tiết các hàm được sử dụng	53
4.3	Synthesis HLS và xuất RTL Design	55
4.4	Tạo block Design bằng Vivado	55
4.5	Dánh giá kiến trúc HLS đã thiết kế	58
5	Chạy kiểm nghiệm trên JupyterNotebook	59
5.1	Khởi tạo Classifier	59
5.2	Danh sách các lớp phân loại	59
5.3	Mở và hiển thị hình ảnh để phân loại	60
5.4	Khởi chạy BNN trên phần cứng	61
5.5	Khởi chạy BNN trên phần mềm	62
5.6	Phát hiện đối tượng trong cảnh	64
5.6.1	Đọc và hiển thị hình ảnh	64
5.6.2	Các bước thực hiện nhận diện trong cảnh	64

5.7	Nhận xét kết quả chạy trên Jupyter	66
6	Kết luận và hướng phát triển	67
6.1	Kết luận	67
6.2	Phát triển	68

Danh sách bảng

Bảng 1	Chi tiết cấu trúc mô hình	36
Bảng 2	Các tham số quan trọng	36
Bảng 3	Chi tiết các chức năng mô hình gtsrb.py	40

Danh sách hình vẽ

Hình 1	Cấu trúc CNN	12
Hình 2	Ví dụ thực tế về cấu trúc của 1 model BNN	17
Hình 3	Cách BNN thực hiện tính toán mô hình	18
Hình 4	Minh họa lớp sign và Straight-Through Estimator (STE)	19
Hình 5	Thiết kế FPGA	22
Hình 6	Hình ảnh kit PYNQ-Z2	24
Hình 7	Thông số kỹ thuật kit PYNQ-Z2	25
Hình 8	PYNQ Framework	26
Hình 9	Chèn image OS vào thẻ SD	27
Hình 10	Hình ảnh kết nối kit PYNQ-Z2	28
Hình 11	Set IP và DNS cho kit	28
Hình 12	Hình ảnh kit đã kết nối thành công	29
Hình 13	Tập dữ liệu German Traffic Sign Recognition Benchmark	30
Hình 14	Biểu đồ số tệp training	30
Hình 15	Các bước thiết kế	31
Hình 16	Sơ đồ giải thuật tổng quát	31
Hình 17	Luồng hoạt động chính	31
Hình 18	Sơ đồ giải thuật quantization.py	32
Hình 19	Sơ đồ giải thuật quantization_net.py	34
Hình 20	Sơ đồ giải thuật cnv.py	35
Hình 21	Luồng xử lý khối Data Pipeline	37
Hình 22	Sơ đồ giải thuật readTrafficSigns.py	38
Hình 23	Sơ đồ giải thuật augmentors.py	38
Hình 24	Sơ đồ giải thuật gtsrb.py	39
Hình 25	Sơ đồ giải thuật Weight Conversion	40
Hình 26	Kết quả Training khi chưa áp dụng các biện pháp	42
Hình 27	Biểu đồ quá trình Training của mô hình	43
Hình 28	Accuracy của mô hình sau train	44
Hình 29	Kiến trúc CNV BNN	46
Hình 30	Sơ đồ giải thuật truyền dữ liệu tổng quát bằng HLS	47
Hình 31	Kiến trúc HLS CNV BNN cho Software	52
Hình 32	Kiến trúc tổng quát hệ thống	53
Hình 33	Synthesis và Export HLS thành công	55
Hình 34	Block Design bộ CNV BNN	55
Hình 35	Xuất Bitstream thành công	56
Hình 36	Xuất file .tcl	56
Hình 37	Utilization reports	57

Hình 38	Khởi tạo Classifier	59
Hình 39	Danh sách các lớp phân loại	59
Hình 40	Mở và hiển thị hình ảnh để phân loại	60
Hình 41	Kết quả chạy trên phần cứng	61
Hình 42	Kết quả chạy trên phần mềm	62
Hình 43	Biểu đồ so sánh HW và SW	62
Hình 44	Kết quả tính toán thời gian tăng tốc phần cứng	63
Hình 45	Đọc và hiển thị biển báo "STOP"	64
Hình 46	Phát hiện đối tượng trong cảnh sau	65
Hình 47	Phát hiện biển báo cấm trong ảnh	66
Hình 48	QR Source code của đồ án	71
Hình 49	Giao diện mở và tạo Project trên Vitis	72
Hình 50	Giao diện thêm file cấu hình Hardware C++	72
Hình 51	Giao diện chọn Top Function cho mô hình	73
Hình 52	Giao diện chọn Part của Kit	73
Hình 53	Thêm các file header	73
Hình 54	Giao diện chính Vitis HLS	74
Hình 55	Synthesis code C++	74
Hình 56	Xuất RTL	74
Hình 57	Mở và tạo Project Vivado	75
Hình 58	Chọn board trên Vivado	75
Hình 59	Các bước gọi IP vừa thiết kế	76
Hình 60	Tạo Block Design mới	76
Hình 61	Block Design hoàn chỉnh	77
Hình 62	Các bước xuất ra Bitfile	77
Hình 63	Xuất block design	78
Hình 64	Truy cập vào thẻ SD của PYNQ-Z2	78
Hình 65	Truy cập Jupyter Notebook	79
Hình 66	Mở cửa sổ Notebook mới	79

ĐỒ ÁN TỐT NGHIỆP - CHUYÊN NGÀNH ĐIỆN TỬ VIỄN THÔNG

1 GIỚI THIỆU

1.1 Giới thiệu đề tài

Đề tài "Nhận diện biển báo giao thông trên FPGA PYNQ-Z2" nghiên cứu và phát triển một hệ thống sử dụng công nghệ FPGA kết hợp với các phương pháp học sâu để nhận diện biển báo giao thông. Việc áp dụng FPGA giúp tối ưu hóa quá trình xử lý và phân loại biển báo giao thông trong thời gian thực. Hệ thống sử dụng Mạng Nơ-ron Tích chập (CNN) để tự động trích xuất các đặc trưng từ ảnh biển báo, giúp phân loại chính xác các loại biển báo giao thông từ bộ dữ liệu GTSRB. Việc triển khai trên nền tảng FPGA giúp giảm độ trễ và tiết kiệm năng lượng, điều này rất quan trọng trong các ứng dụng giao thông thông minh.

Để giải quyết vấn đề về tài nguyên bộ nhớ và tính toán khi triển khai các mô hình học sâu trên phần cứng hạn chế, nghiên cứu đã áp dụng Mạng Nơ-ron Nhị phân (BNN). BNN giúp nhị phân hóa trọng số và kích hoạt của mô hình, chuyển đổi các giá trị thực thành các giá trị nhị phân (+1 hoặc -1). Phương pháp này không chỉ giảm kích thước mô hình mà còn giúp tăng tốc độ tính toán nhờ vào các phép toán nhị phân đơn giản, rất phù hợp với các hệ thống có tài nguyên tính toán hạn chế như FPGA.

Hệ thống nhận diện biển báo giao thông sử dụng FPGA PYNQ-Z2 có khả năng nhận diện chính xác nhiều loại biển báo trong thời gian thực. Các biển báo như biển báo tốc độ, biển báo dừng, và các biển báo cảnh báo khác đều có thể được phân loại nhanh chóng, giúp cải thiện hiệu quả giao thông và đảm bảo an toàn cho người tham gia. Việc áp dụng BNN giúp hệ thống hoạt động hiệu quả hơn, tiết kiệm bộ nhớ và năng lượng, đồng thời tăng tốc quá trình xử lý, mở ra cơ hội ứng dụng trong các xe tự lái và các hệ thống giao thông thông minh.

Kết quả nghiên cứu đã chứng minh tính khả thi của việc triển khai hệ thống nhận diện biển báo giao thông trên FPGA PYNQ-Z2 với độ chính xác cao và hiệu suất vượt trội. Hệ thống có thể được áp dụng rộng rãi trong các ứng dụng giao thông thông minh, giúp tối ưu hóa việc điều khiển giao thông và giám sát an toàn. Hướng phát triển tiếp theo có thể là tối ưu hóa mô hình học sâu hơn nữa và mở rộng ứng dụng của hệ thống vào các nền tảng giao thông tự động hóa, từ đó đóng góp vào sự phát triển của các thành phố thông minh.

1.2 Cấu trúc của bài báo cáo

Cấu trúc của bài báo cáo bao gồm các phần sau:

- Lời mở đầu: Giới thiệu về đề tài, lý do chọn đề tài và mục tiêu nghiên cứu.
- Tổng quan lý thuyết: Cung cấp kiến thức nền tảng về FPGA, học sâu, CNN và BNN, cũng như các ứng dụng của chúng trong nhận diện biển báo giao thông.
- Phương pháp nghiên cứu: Trình bày cách tiếp cận và các phương pháp sử dụng, bao gồm quá trình huấn luyện mô hình CNN, áp dụng BNN và triển khai trên FPGA PYNQ-Z2.
- Kết quả và thảo luận: Dưa ra kết quả đạt được từ việc triển khai và thử nghiệm hệ thống, cùng với phân tích hiệu suất và độ chính xác.
- Kết luận và hướng phát triển: Tổng kết các kết quả nghiên cứu, những ứng dụng thực tế và gợi ý cho nghiên cứu tiếp theo.
- Tài liệu tham khảo: Liệt kê các nguồn tài liệu đã tham khảo trong quá trình nghiên cứu.

1.3 Mục tiêu đề tài

Mục tiêu của đề tài "Nhận diện biển báo giao thông trên FPGA PYNQ-Z2" là nghiên cứu và phát triển một hệ thống nhận diện biển báo giao thông sử dụng công nghệ FPGA kết hợp với các phương pháp học sâu, đặc biệt là Mạng Nơ-ron Tích chập (CNN) và Mạng Nơ-ron Nhị phân (BNN). Cụ thể, mục tiêu của đề tài bao gồm:

1. Phát triển hệ thống nhận diện, phân loại biển báo giao thông: Sử dụng FPGA PYNQ-Z2 để tối ưu hóa quá trình xử lý hình ảnh và phân loại các loại biển báo giao thông.
2. Áp dụng các thuật toán học sâu: Sử dụng CNN để trích xuất đặc trưng từ hình ảnh và phân loại biển báo, đồng thời áp dụng BNN để giảm yêu cầu bộ nhớ và tính toán khi triển khai mô hình trên phần cứng có tài nguyên hạn chế.
3. Tối ưu hóa hiệu suất và tiết kiệm tài nguyên: Thiết kế hệ thống với khả năng nhận diện biển báo giao thông trong thời gian thực, giảm độ trễ và tiết kiệm năng lượng.
4. Đánh giá hiệu quả của hệ thống: Đo lường độ chính xác, tốc độ nhận diện và khả năng triển khai hệ thống trên phần mềm và phần cứng FPGA, đánh giá, so sánh khả năng thực hiện và tốc độ thực thi trên hai nền tảng này.

2 CƠ SỞ LÝ THUYẾT

2.1 Giới thiệu về mô hình học sâu (Deep Learning Models)

Mô hình học sâu (Deep Learning - DL) là một nhánh con của học máy (machine learning), được phát triển dựa trên các mạng nơ-ron nhân tạo (artificial neural networks). Các mô hình học sâu được gọi là "sâu" vì chúng sử dụng nhiều lớp ẩn (hidden layers) để trích xuất các đặc trưng phức tạp từ dữ liệu. Mô hình học sâu có khả năng học từ dữ liệu một cách tự động mà không cần phải xác định trước các đặc trưng quan trọng, điều này giúp chúng đặc biệt mạnh mẽ trong việc xử lý các vấn đề phức tạp như nhận diện hình ảnh, xử lý ngôn ngữ tự nhiên, nhận dạng tiếng nói và nhiều ứng dụng khác.

Một trong những yếu tố quan trọng giúp học sâu trở nên mạnh mẽ là khả năng học được các biểu diễn dữ liệu phức tạp thông qua việc truyền dữ liệu qua các lớp khác nhau trong mô hình. Học sâu đã giúp đạt được những bước tiến đáng kể trong nhiều lĩnh vực, nhưng cũng có những thách thức về yêu cầu tính toán và tài nguyên bộ nhớ khi triển khai trên các thiết bị có hạn chế tài nguyên, chẳng hạn như các thiết bị di động hoặc hệ thống nhúng. Các mô hình học sâu, đặc biệt là Mạng Nơ-ron Tích chập (CNN), được sử dụng rộng rãi trong các bài toán nhận diện hình ảnh, như nhận diện biển báo giao thông, phân loại động vật, hay nhận dạng khuôn mặt. Bên cạnh đó, Mạng Nơ-ron Nhị phân (BNN) giúp tối ưu hóa mô hình học sâu bằng cách nhị phân hóa trọng số và kích hoạt, từ đó giảm yêu cầu về bộ nhớ và tính toán khi triển khai trên phần cứng hạn chế.

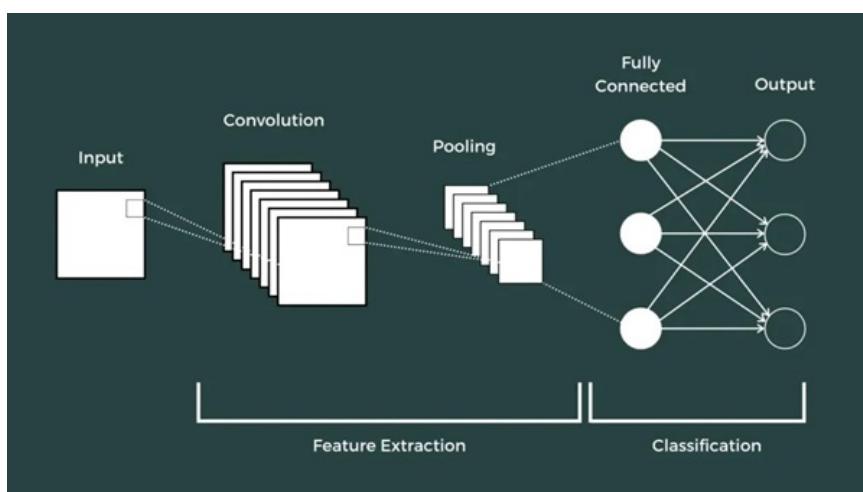
Ngoài CNN và BNN, mô hình học sâu còn có nhiều ứng dụng quan trọng khác, đặc biệt là trong Xử lý ngôn ngữ tự nhiên (NLP). Các mô hình như RNN và Transformers được ứng dụng trong các hệ thống dịch ngôn ngữ tự động, phân loại văn bản, và trợ lý ảo. Học sâu cũng đóng vai trò quan trọng trong Nhận dạng giọng nói, với các hệ thống chuyển đổi giọng nói thành văn bản hoặc nhận diện âm thanh trong môi trường công nghiệp. Hơn nữa, học sâu đã được áp dụng trong Dự báo tài chính, giúp dự đoán xu hướng thị trường chứng khoán hoặc phân tích dữ liệu kinh tế. Những ứng dụng này cho thấy sự linh hoạt và tiềm năng của mô hình học sâu trong việc giải quyết các vấn đề phức tạp ở nhiều lĩnh vực khác nhau.

2.2 Giới thiệu về Convolutional Neural Networks - CNN

Mạng Convolutional (Convolutional Neural Networks - CNN) là một trong những mô hình học sâu thành công nhất, đặc biệt trong việc xử lý hình ảnh và video. CNN được thiết kế để nhận diện các đặc trưng không gian trong dữ liệu có cấu trúc như ảnh hoặc video. CNN sử dụng các lớp convolutional để tự động trích xuất các đặc trưng từ ảnh mà không cần phải xác định các đặc trưng thủ công, điều này làm cho CNN cực kỳ hiệu quả trong việc nhận dạng hình ảnh và đối tượng.

CNN hoạt động thông qua ba loại lớp cơ bản: lớp tích chập (Convolutional Layer), lớp pooling (Pooling Layer), và lớp fully connected (FC Layer).

2.2.1 Cấu trúc của CNN:



Hình 1: Cấu trúc CNN

- **Lớp tích chập (Convolutional Layer):**

Lớp tích chập là lớp quan trọng nhất trong CNN. Mục tiêu của lớp này là trích xuất các đặc trưng từ ảnh đầu vào thông qua việc sử dụng một bộ lọc (hay còn gọi là kernel). Bộ lọc này sẽ "quét" qua toàn bộ ảnh và tính toán phép toán tích chập giữa ảnh đầu vào và bộ lọc. Kết quả của phép toán tích chập là các bản đồ đặc trưng (feature maps), trong đó mỗi giá trị trong bản đồ đặc trưng phản ánh một đặc trưng học được từ ảnh đầu vào.

Lớp tích chập không chỉ giúp phát hiện các đặc trưng đơn giản như cạnh và góc, mà còn giúp phát hiện các hình dạng phức tạp hơn trong các lớp sau của mạng. Cùng với quá trình huấn luyện, các bộ lọc trong lớp tích chập sẽ học các đặc trưng phù hợp nhất để phân biệt các đối tượng trong ảnh.

Phép toán tích chập giữa ảnh đầu vào X và bộ lọc W được tính bằng công thức sau:

$$Y(m, n) = \sum_{i=0}^{S-1} \sum_{j=0}^{S-1} W(i, j) \cdot X(m-i, n-j)$$

Trong đó:

- (m, n) là giá trị của pixel tại vị trí (m, n) trong bản đồ đặc trưng sau phép tích chập.
- $W(i, j)$ là trọng số trong bộ lọc (kernel) với kích thước $S \times S$.
- $X(m-i, n-j)$ là giá trị pixel trong ảnh đầu vào tại vị trí $(m-i, n-j)$.
- S là kích thước của bộ lọc (kernel), thông thường có kích thước 3×3 , 5×5 , hoặc 7×7 .

Ví dụ minh họa: Giả sử bộ lọc có một ảnh đầu vào có kích thước 5×5 :

$$X = \begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 4 & 5 & 6 & 1 & 0 \\ 7 & 8 & 9 & 2 & 1 \\ 2 & 3 & 4 & 3 & 1 \\ 1 & 0 & 1 & 4 & 2 \end{bmatrix}$$

Và bộ lọc (kernel) W có kích thước 3×3 :

$$W = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Khi thực hiện phép toán tích chập, quét bộ lọc qua ảnh và tính giá trị tích chập tại mỗi vị trí. Ví dụ, tại vị trí $Y(1, 1)$:

$$Y(1, 1) = (1 \times 1) + (2 \times 0) + (3 \times -1) + (4 \times 1) + (5 \times 0) + (6 \times -1) + (7 \times 1) + (8 \times 0) + (9 \times -1) = -6$$

Sau khi tính toán các giá trị tại các vị trí $Y(1, 1)$, $Y(1, 2)$, $Y(2, 1)$, và $Y(2, 2)$, chúng ta có bản đồ đặc trưng như sau:

$$Y = \begin{bmatrix} -6 & 12 \\ -6 & 10 \end{bmatrix}$$

Quá trình này giúp tạo ra các bản đồ đặc trưng mà sau này sẽ được sử dụng trong các lớp tiếp theo.

- **Lớp Pooling (Pooling Layer):**

Lớp pooling giúp giảm kích thước của bản đồ đặc trưng và giảm thiểu số lượng tham số, từ đó giúp giảm độ phức tạp của mô hình và tăng tốc độ tính toán. Mục đích chính của lớp pooling là giảm độ phân giải không gian của bản đồ đặc trưng khi vẫn giữ lại các đặc trưng quan trọng.

Có hai loại pooling phổ biến:

- Max pooling: Giữ lại giá trị lớn nhất trong một cửa sổ nhỏ.
- Average pooling: Giữ lại giá trị trung bình trong một cửa sổ.

Lớp pooling giúp làm số lượng tham số, đồng thời giảm độ phức tạp tính toán, đồng thời giữ lại các đặc trưng quan trọng và giúp mạng học được tính chất bất biến về vị trí và góc quay của đối tượng trong ảnh.

Công thức cho max pooling là:

$$P(i, j) = \max_{z \in F} z$$

Trong đó:

- $P(i, j)$ là giá trị của pixel sau khi thực hiện max pooling tại vị trí (i, j) .
- F là một cửa sổ nhỏ trong bản đồ đặc trưng.
- \max là phép toán lấy giá trị lớn nhất trong cửa sổ F .

Còn đối với average pooling, công thức là:

$$P(i, j) = \frac{1}{|F|} \sum_{z \in F} z$$

Trong đó:

$|F|$ là số phần tử trong cửa sổ F .

Ví dụ minh họa: Giả sử có một bản đồ đặc trưng sau lớp tích chập có kích thước 4×4 :

$$S = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Sử dụng max pooling với cửa sổ 2×2 và bước nhảy (stride) là 2. Ta sẽ quét qua bản đồ đặc trưng và lấy giá trị lớn nhất trong mỗi cửa sổ 2×2 . Kết quả pooling sẽ là:

$$P = \begin{bmatrix} 6 & 8 \\ 14 & 16 \end{bmatrix}$$

Với average pooling, sử dụng cửa sổ 2×2 và bước nhảy 2. Ta tính giá trị trung bình trong mỗi cửa sổ 2×2 . Ví dụ, tại vị trí $P(1, 1)$, giá trị trung bình của cửa sổ đầu tiên 2×2 là:

$$P(1, 1) = \frac{1 + 3 + 5 + 6}{4} = 3.75$$

Kết quả pooling sẽ là:

$$P = \begin{bmatrix} 3.75 & 5 \\ 10.5 & 12 \end{bmatrix}$$

- **Lớp Fully Connected (FC Layer):**

Lớp fully connected (FC) nằm ở cuối mạng CNN, nơi các đặc trưng đã được trích xuất từ các lớp trước đó sẽ được sử dụng để phân loại. Các đặc trưng này được làm phẳng thành một vector và được đưa qua một hoặc nhiều lớp fully connected. Mỗi neuron trong lớp fully connected sẽ kết nối với tất cả các neuron trong lớp trước đó. Mục tiêu của lớp FC là phân loại và dự đoán đầu ra dựa trên các đặc trưng đã học được từ các lớp trước.

Trong lớp FC, quá trình học và phân loại được thực hiện bằng cách sử dụng các trọng số và bias để tính toán đầu ra cho từng lớp (label). Sau khi tính toán, các giá trị đầu ra sẽ được đưa qua một hàm kích hoạt (activation function) như ReLU, sigmoid hoặc softmax để xác định xác suất cho mỗi lớp.

Công thức tính toán trong lớp fully connected là:

$$y = f \left(\sum_i w_i \cdot x_i + b \right)$$

Trong đó:

- y là đầu ra của lớp FC, có thể là giá trị phân loại.
- w_i là trọng số của liên kết giữa các neuron trong lớp FC.
- x_i là đầu vào từ lớp trước.
- b là bias của neuron trong lớp FC.
- f là hàm kích hoạt (activation function), ví dụ như ReLU, sigmoid, hoặc softmax.

Ví dụ minh họa: Giả sử sau khi qua các lớp trước, ta có một vector đặc trưng sau khi

làm phẳng (flatten) có chiều dài là 6:

$$x = [1 \ 2 \ 3 \ 4 \ 5 \ 6]$$

Lớp fully connected sẽ tính toán đầu ra từ vector này.

Ví dụ với trọng số $w = [0.5 \ 0.2 \ 0.4 \ 0.3 \ 0.1 \ 0.6]$ và bias $b = 0.1$, công thức tính toán là:

$$y = f((0.5 \times 1) + (0.2 \times 2) + (0.4 \times 3) + (0.3 \times 4) + (0.1 \times 5) + (0.6 \times 6)) + 0.1$$

$$y = f(0.5 + 0.4 + 1.2 + 1.2 + 0.5 + 3.6) + 0.1 = f(7.8)$$

Giải hàm kích hoạt là ReLU, kết quả sẽ là:

$$y = \text{ReLU}(7.8) = 7.8$$

Lớp FC giúp đưa ra kết quả phân loại cuối cùng từ các đặc trưng đã học được từ các lớp trước.

2.2.2 Lợi ích của CNN:

- Khả năng học đặc trưng tự động: CNN có khả năng tự động trích xuất các đặc trưng từ dữ liệu mà không cần phải xác định trước các đặc trưng quan trọng, giúp giảm chi phí thời gian và công sức so với các phương pháp học máy truyền thống.
- Tính bất biến với vị trí: Nhờ vào cấu trúc của các lớp convolutional, CNN có khả năng nhận diện các đối tượng trong ảnh dù chúng xuất hiện ở vị trí nào, giúp cải thiện độ chính xác trong nhận diện.
- Giảm yêu cầu tính toán: Bằng cách sử dụng chuyển giao trọng số (weight sharing) trong các lớp convolutional, CNN giúp giảm số lượng tham số cần học, giúp tiết kiệm bộ nhớ và tăng tốc độ tính toán.

2.2.3 Ứng dụng của CNN:

CNN đặc biệt hữu ích trong các bài toán nhận diện hình ảnh và video. Các ứng dụng điển hình bao gồm:

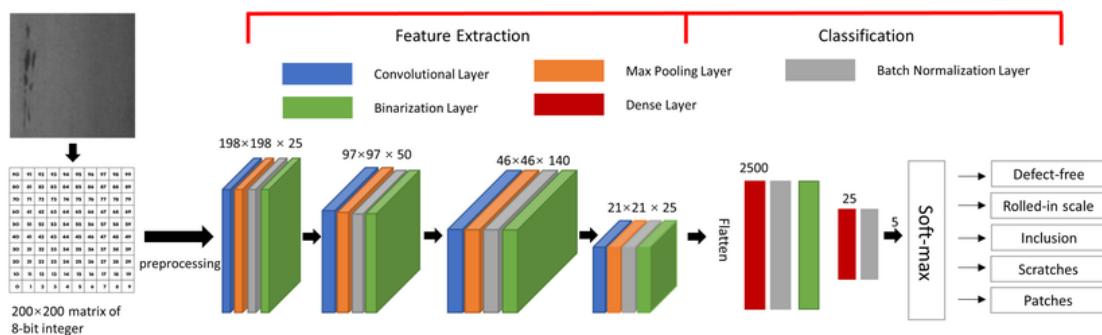
- Phân loại hình ảnh: Ví dụ, phân loại ảnh thành các nhóm như chó, mèo, hoặc nhận diện các vật thể cụ thể trong ảnh, video.
- Nhận diện khuôn mặt trong các hệ thống bảo mật.
- Nhận diện biển báo giao thông trong các hệ thống giao thông thông minh.

2.3 Giới thiệu về Binarized Neural Networks - BNN

Mạng Nơ-ron nhị phân (Binarized Neural Networks - BNNs) là một phương pháp tối ưu hóa các mô hình học sâu (Deep Neural Networks - DNNs), với mục đích giảm thiểu yêu cầu về bộ nhớ và tính toán trong khi vẫn giữ được khả năng học mạnh mẽ của các mô hình DNN truyền thống. BNN đặc biệt hữu ích khi triển khai trên các hệ thống có tài nguyên hạn chế, chẳng hạn như các thiết bị di động, hệ thống nhúng hoặc các phần cứng như FPGA và ASIC. BNN hoạt động bằng cách nhị phân hóa cả trọng số (weights) và kích hoạt (activations) của mô hình học sâu, nghĩa là chúng chỉ sử dụng hai giá trị nhị phân (+1 hoặc -1) thay vì các giá trị thực (floating-point values).

Trong khi các mô hình học sâu như DNNs và CNNs có khả năng học mạnh mẽ và linh hoạt, chúng lại yêu cầu rất nhiều bộ nhớ và tài nguyên tính toán. Điều này làm cho việc triển khai các mô hình này trên các nền tảng có tài nguyên hạn chế (như IoT devices, mobile devices, hoặc edge devices) trở nên khó khăn. BNNs được phát triển để giải quyết vấn đề này, cho phép các mô hình học sâu có thể hoạt động nhanh hơn và tiết kiệm bộ nhớ bằng cách sử dụng các giá trị nhị phân thay cho các giá trị thực.

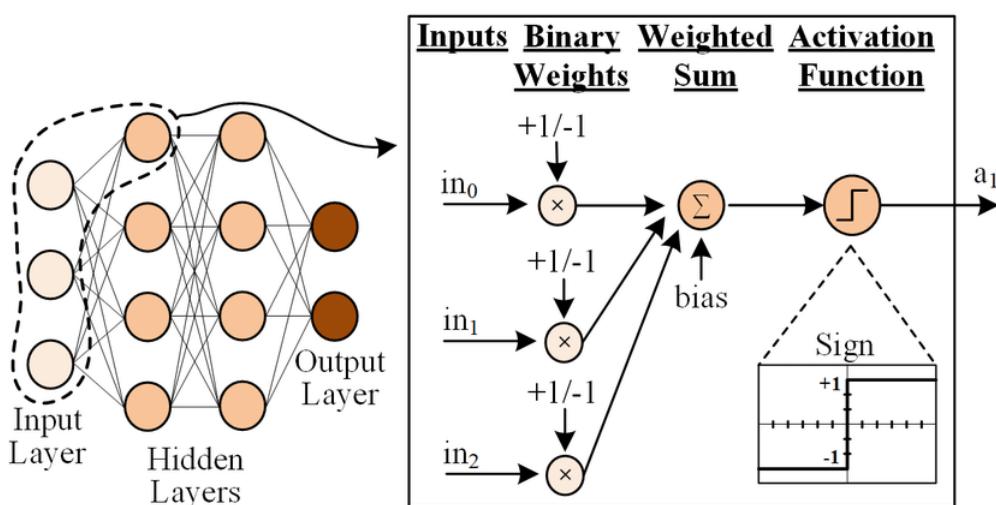
BNN được xây dựng dựa trên các phương pháp quy chuẩn hóa (quantization) và nhị phân hóa (binarization) trong học sâu, cho phép giảm kích thước mô hình mà không làm giảm quá nhiều độ chính xác. Các nghiên cứu về BNN đã chỉ ra rằng việc nhị phân hóa trọng số và kích hoạt có thể mang lại các lợi ích rõ rệt về hiệu suất và tiết kiệm tài nguyên trong quá trình suy luận (inference), cũng như huấn luyện (training).



Hình 2: Ví dụ thực tế về cấu trúc của 1 model BNN

2.3.1 Các khái niệm cơ bản trong BNN

1. Trọng số (Weights): Trong BNN, trọng số là các giá trị học được và được sử dụng trong phép toán dot product với các giá trị kích hoạt từ các lớp trước. Tuy nhiên, thay vì sử dụng các giá trị thực, trọng số trong BNN được nhị phân hóa, nghĩa là chúng chỉ có thể nhận một trong hai giá trị: +1 hoặc -1. Mặc dù trong quá trình huấn luyện, các trọng số thực được sử dụng, nhưng sau khi huấn luyện xong, các trọng số nhị phân sẽ được sử dụng trong quá trình suy luận.
2. Kích hoạt (Activations): Kích hoạt là các giá trị đầu ra của hàm kích hoạt trong mỗi neuron. Trong BNN, các giá trị kích hoạt cũng được nhị phân hóa thành +1 hoặc -1, thay vì sử dụng các giá trị thực. Hàm kích hoạt được sử dụng trong BNN là hàm sign function, có nhiệm vụ chuyển giá trị thực thành giá trị.
3. Dot Product: Trong mạng nơ-ron, dot product là một phép toán nhân và cộng được thực hiện giữa các trọng số và các giá trị kích hoạt. BNN sử dụng các phép toán nhị phân đơn giản hơn thay vì các phép toán số học thực tế, giúp tiết kiệm bộ nhớ và tính toán.
4. Độ lệch (Bias) và Hệ số tăng cường (Gain)
 - Bias là một giá trị thêm vào trong mạng nơ-ron, giúp mô hình học được các đặc trưng tốt hơn.
 - Gain là một hệ số tăng cường được học trong BNN, tương tự như bias, và nó được áp dụng sau khi thực hiện phép toán dot product giữa trọng số và kích hoạt.
5. Topology và Architecture: Topology là cấu trúc của các lớp trong mô hình mạng nơ-ron, còn architecture đề cập đến việc bố trí các thành phần phần cứng khi triển khai mạng nơ-ron.



Hình 3: Cách BNN thực hiện tính toán mô hình

2.3.2 Cách BNN hoạt động

BNN được xây dựng trên nguyên lý nhị phân hóa cả trọng số và kích hoạt. Điều này giúp giảm bớt yêu cầu bộ nhớ vì mỗi trọng số và mỗi kích hoạt chỉ cần 1 bit để lưu trữ, thay vì 32 bit như trong các mô hình DNN thông thường. Điều này làm giảm kích thước mô hình và tăng tốc độ tính toán nhờ vào các phép toán nhị phân đơn giản (cộng và nhân nhị phân).

1. Nhị phân hóa trọng số (Binarization of Weights):

Trọng số thực W_R được chuyển thành trọng số nhị phân W_B thông qua hàm sign function:

$$W_B = \text{sign}(W_R)$$

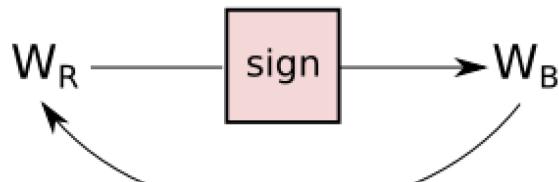
Nếu trọng số thực $W_R \geq 0$, thì $W_B = +1$; nếu $W_R < 0$, thì $W_B = -1$.

- Tuy nhiên, trong quá trình huấn luyện, trọng số thực W_R vẫn được cập nhật thông qua các phương pháp tối ưu hóa như Stochastic Gradient Descent (SGD) hoặc Adam, còn trong suy luận, các trọng số nhị phân W_B sẽ được sử dụng.
- Việc tính toán gradient đối với các trọng số nhị phân thông qua backpropagation gặp khó khăn vì gradient của hàm sign bằng 0 hoặc không xác định. Để giải quyết vấn đề này, phương pháp Straight-Through Estimator (STE) được sử dụng, giúp xấp xỉ gradient của hàm sign bằng cách bỏ qua gradient của lớp hiện tại và sử dụng hàm identity:

$$\frac{\partial L}{\partial W_R} = \frac{\partial L}{\partial W_B}$$

- Nhờ vào phương pháp STE, các trọng số thực W_R có thể được cập nhật trong quá trình huấn luyện mà không ảnh hưởng đến các trọng số nhị phân W_B .

Forward pass of the weights



Backward pass of the gradient

Hình 4: Minh họa lớp sign và Straight-Through Estimator (STE)

2. Nhị phân hóa kích hoạt (Binarization of Activations)

Hàm sign sẽ chuyển giá trị kích hoạt thực thành các giá trị nhị phân:

$$a_B = \text{sign}(a_R)$$

Nếu $a_R \geq 0$, thì $a_B = +1$; nếu $a_R < 0$, thì $a_B = -1$.

Trong quá trình huấn luyện, để đạt được kết quả tốt, Courbariaux et al. nhận thấy rằng cần hủy bỏ gradient trong quá trình lan truyền ngược nếu đầu vào của hàm kích hoạt quá lớn. Điều này có thể được thực hiện thông qua một hàm chỉ thị (indicator function) để đặt gradient bằng 0 nếu đầu vào của hàm kích hoạt lớn hơn một giá trị giới hạn nhất định. Cụ thể:

$$\frac{\partial L}{\partial a_R} = \frac{\partial L}{\partial a_B} \times 1(|a_R| \leq 1)$$

Trong đó:

- a_R là giá trị thực đầu vào hàm kích hoạt.
- a_B là giá trị nhị phân hóa đầu ra hàm kích hoạt.
- $1(|a_R| \leq 1)$ là hàm chỉ thị, đánh giá là 1 nếu $|a_R| \leq 1$, và 0 nếu ngược lại. Điều này có nghĩa là nếu giá trị đầu vào hàm kích hoạt quá lớn (ngoài phạm vi [-1, 1]), gradient sẽ được làm bằng 0, ngừng việc cập nhật trong quá trình lan truyền ngược.

Quá trình này phần lớn được khắc phục thông qua việc sử dụng STE giúp mạng có thể huấn luyện hiệu quả hơn, đồng thời giữ được tính chính xác trong suy luận nhờ vào việc bù đắp gradient của hàm kích hoạt.

2.3.3 Lợi ích của BNN

- Tiết kiệm bộ nhớ: Việc sử dụng 1 bit thay vì các giá trị thực 32 bit giúp giảm kích thước mô hình và tiết kiệm bộ nhớ đáng kể, đặc biệt khi triển khai trên các hệ thống có tài nguyên hạn chế như IoT devices và FPGA.
- Tăng tốc tính toán: Các phép toán nhị phân có thể thực hiện nhanh chóng và hiệu quả trên phần cứng, giúp tăng tốc độ suy luận. Các phép toán như cộng và nhân nhị phân đơn giản hơn nhiều so với các phép toán với số thực.
- Tiết kiệm năng lượng: Các phép toán nhị phân tiêu thụ ít năng lượng hơn, điều này rất quan trọng khi triển khai trên các thiết bị di động hoặc hệ thống nhúng.

2.3.4 Hạn chế của BNN

- Giảm độ chính xác: Việc nhị phân hóa trọng số và kích hoạt có thể làm giảm độ chính xác so với các mô hình sử dụng số thực. Điều này đặc biệt dễ nhận thấy khi bài toán yêu cầu độ chính xác cao trong việc phân loại hoặc nhận diện.
- Khó khăn trong huấn luyện: Việc nhị phân hóa trong quá trình huấn luyện có thể gặp khó khăn vì các gradient từ hàm sign không thể thay đổi nhỏ. Tuy nhiên, phương pháp STE đã giúp giải quyết vấn đề này một cách hiệu quả.

2.3.5 Ứng dụng của BNN

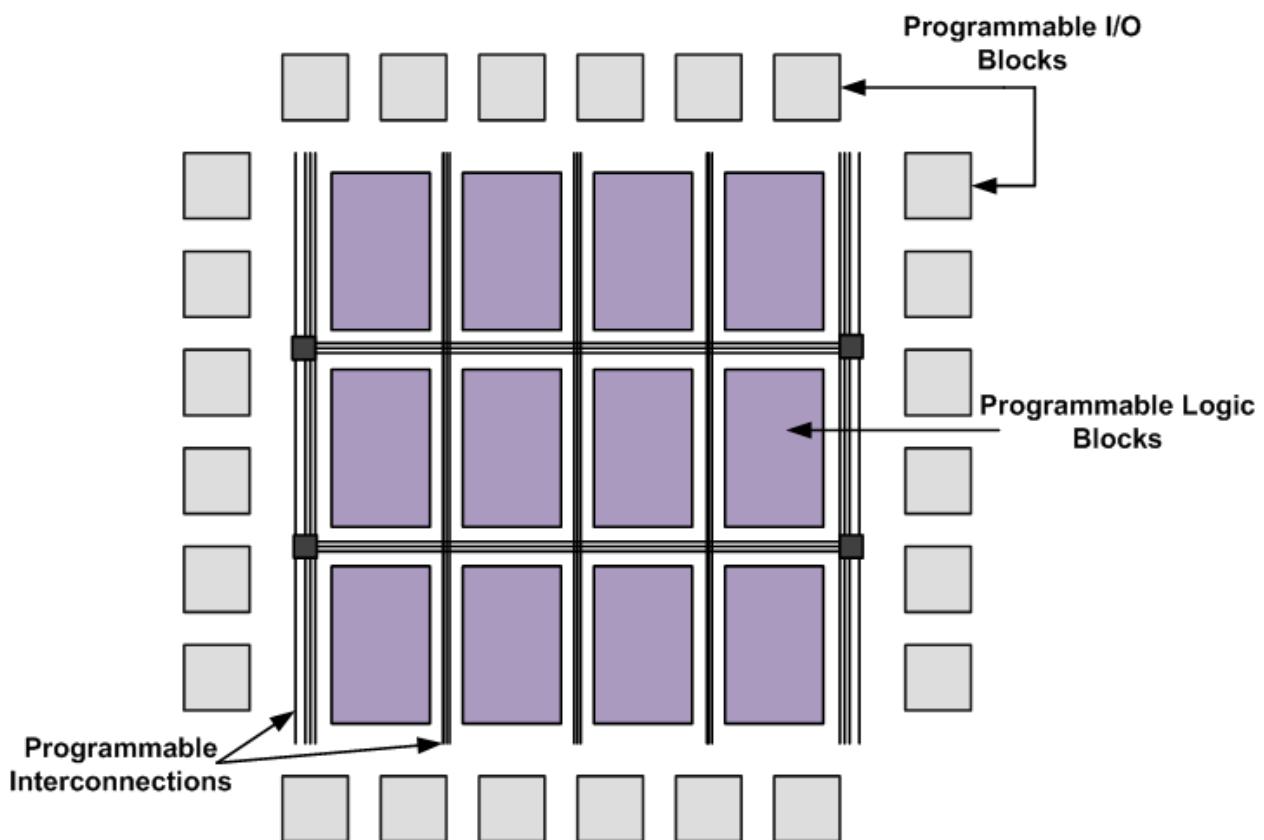
BNN rất thích hợp cho các hệ thống có tài nguyên hạn chế nhưng vẫn cần hiệu suất học sâu mạnh mẽ, như các hệ thống nhúng hoặc phần cứng đặc thù như FPGA và ASIC. Các ứng dụng điển hình của BNN bao gồm:

- Nhận diện biển báo giao thông: BNN có thể giúp nhận diện các biển báo giao thông trong các hệ thống tự lái hoặc các hệ thống giám sát giao thông.
- Nhận diện đối tượng trong video: BNN giúp xử lý video trong thời gian thực, nhận diện các đối tượng hoặc hành vi từ các đoạn video.
- Nhận diện hình ảnh trong IoT devices: Các thiết bị IoT có thể triển khai BNN để nhận diện các hình ảnh hoặc dữ liệu mà không cần yêu cầu quá nhiều bộ nhớ và tính toán.

2.4 Tổng quan về FPGA

FPGA (Field-Programmable Gate Array) là một loại mạch tích hợp có thể lập trình lại sau khi sản xuất, cho phép người dùng thiết kế phần cứng tùy chỉnh cho các ứng dụng cụ thể. FPGA bao gồm các khối logic có thể cấu hình (CLBs - Configurable Logic Blocks) và các đường truyền có thể lập trình (Programmable Interconnects), giúp người thiết kế kết nối các khối logic và cấu hình chúng để thực hiện các phép toán từ các cổng logic đơn giản đến các chức năng phức tạp. Một trong những ưu điểm nổi bật của FPGA là khả năng xử lý song song, giúp thực hiện các tác vụ tính toán nhanh chóng và hiệu quả hơn so với các bộ xử lý truyền thống.

Các FPGA hiện nay thường được sử dụng trong nhiều ứng dụng khác nhau, đặc biệt là trong các hệ thống xử lý tín hiệu, hình ảnh và các thuật toán học máy. FPGA có thể lập trình lại nhiều lần, giúp người dùng linh hoạt thay đổi chức năng của mạch mà không cần thay thế phần cứng. Tuy nhiên, thiết kế FPGA yêu cầu kiến thức sâu về các ngôn ngữ mô tả phần cứng như VHDL hoặc Verilog, điều này làm cho việc phát triển phần cứng trên FPGA có phần phức tạp hơn so với lập trình phần mềm.



Hình 5: Thiết kế FPGA

2.4.1 Ưu điểm của FPGA

Dưới đây là một số lợi ích của FPGA:

- Xử lý song song: FPGA có khả năng thực hiện các phép toán song song, giúp tăng tốc các tác vụ tính toán nặng như xử lý tín hiệu số và học máy.
- Linh hoạt: FPGA có thể được lập trình lại bất kỳ lúc nào, giúp người dùng thay đổi chức năng của nó mà không phải thay phần cứng. Điều này giúp tiết kiệm chi phí và thời gian phát triển.
- Hiệu suất cao: FPGA có thể xử lý các tác vụ phức tạp với hiệu suất cao, nhờ vào khả năng tối ưu hóa phần cứng cho từng ứng dụng cụ thể.
- Tiết kiệm chi phí: So với ASIC (mạch tích hợp dành riêng), FPGA có chi phí thấp hơn trong việc phát triển và không cần phải chi tiền cho việc thiết kế phần cứng riêng biệt.
- Dễ dàng lập trình: FPGA có thể được lập trình thông qua phần mềm sử dụng các ngôn ngữ như VHDL hoặc Verilog, giúp dễ dàng thiết kế và kiểm tra các hệ thống phần cứng.

2.4.2 Nhược điểm của FPGA

Dưới đây là một số hạn chế của FPGA:

- Yêu cầu kiến thức phần cứng: Để lập trình FPGA, người phát triển cần có kiến thức về các ngôn ngữ mô tả phần cứng như VHDL hoặc Verilog, điều này đòi hỏi thời gian học hỏi và làm quen.

- Tiêu thụ điện năng: FPGA có thể tiêu tốn nhiều năng lượng hơn so với các bộ xử lý chuyên dụng như ASIC hoặc GPU.
- Giới hạn về tài nguyên: Khi chọn một FPGA cho một dự án, người phát triển phải làm việc với tài nguyên của FPGA đã chọn, và đôi khi phải tìm cách tối ưu hóa tài nguyên hạn chế này.
- Đắt khi sản xuất số lượng lớn: FPGA thích hợp với prototyping (thử nghiệm mẫu) và sản xuất số lượng nhỏ, nhưng khi cần sản xuất số lượng lớn, chi phí cho FPGA sẽ cao hơn so với các giải pháp ASIC.

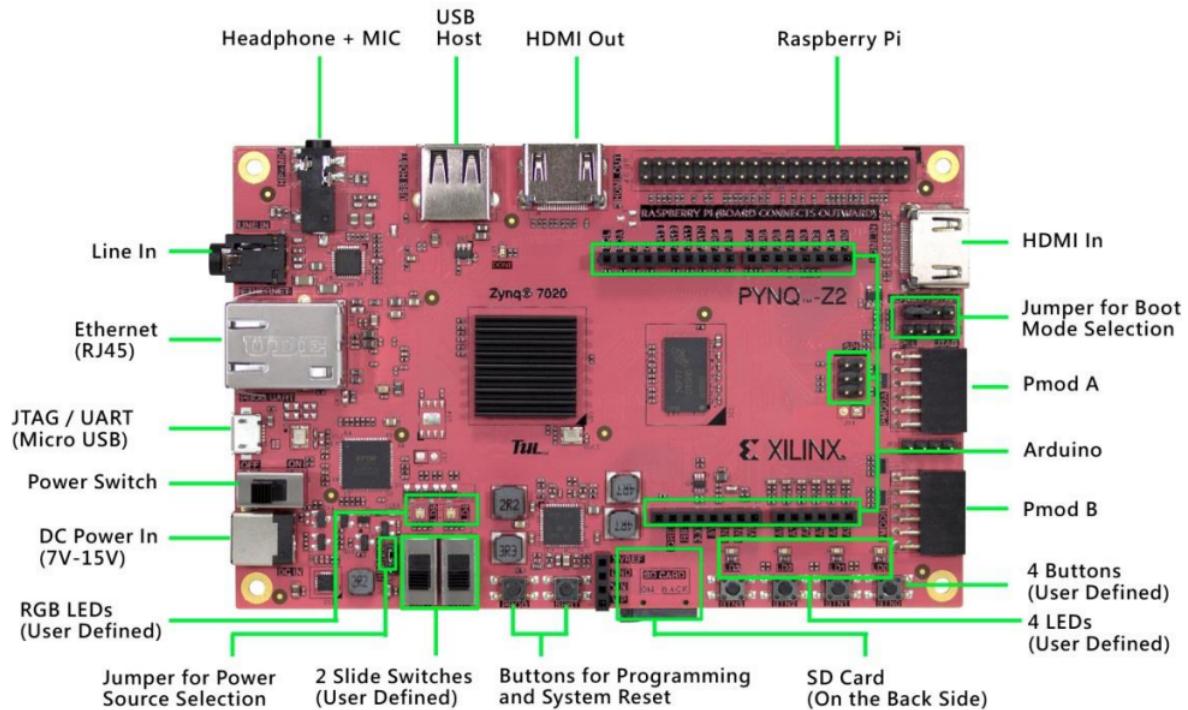
2.4.3 Ứng dụng của FPGA

FPGA được sử dụng trong rất nhiều lĩnh vực và ứng dụng, từ các hệ thống đơn giản đến các cấu trúc phức tạp. Những ứng dụng tiêu biểu của FPGA bao gồm:

- Phát triển phần cứng cho vi điều khiển: FPGA có thể được sử dụng để phát triển các bo mạch phát triển và giao diện cho các vi điều khiển, giúp các nhà thiết kế dễ dàng thử nghiệm và phát triển sản phẩm.
- Tự động hóa và IoT (Internet of Things): FPGA được ứng dụng trong các hệ thống điều khiển và tự động hóa, đặc biệt trong các thiết bị IoT để xử lý dữ liệu nhanh chóng và hiệu quả.
- Thị giác máy tính và xử lý hình ảnh: FPGA là một công cụ lý tưởng cho các ứng dụng xử lý hình ảnh và video nhờ vào khả năng xử lý song song và hiệu suất cao.
- Mã hóa và bảo mật: FPGA được sử dụng trong các hệ thống mã hóa dữ liệu, đặc biệt là trong các ứng dụng đòi hỏi tốc độ xử lý cao và bảo mật.
- Ứng dụng trong hàng không và quốc phòng: Các hệ thống trên FPGA có thể đáp ứng được yêu cầu khắt khe trong ngành hàng không và quốc phòng, nơi mà tính linh hoạt và hiệu suất là rất quan trọng.

2.5 Tổng quan về kit FPGA PYNQ-Z2

PYNQ-Z2 là một nền tảng phần cứng phát triển dựa trên công nghệ FPGA (Field-Programmable Gate Array) của Xilinx, được thiết kế để giúp các nhà phát triển và nghiên cứu viên dễ dàng triển khai và thử nghiệm các ứng dụng học máy và xử lý tín hiệu số. Kit PYNQ-Z2 được trang bị FPGA Xilinx Zynq-7000 với tính năng ARM-based và có khả năng xử lý song song mạnh mẽ, làm cho nó trở thành một công cụ lý tưởng để phát triển các hệ thống nhúng với yêu cầu tính toán cao và tiêu thụ năng lượng thấp.



Hình 6: Hình ảnh kit PYNQ-Z2

2.5.1 Thông số kỹ thuật kit PYNQ-Z2

Kit PYNQ-Z2 là một nền tảng phát triển mạnh mẽ, được trang bị với những tính năng đáng chú ý sau:

- **FPGA Xilinx Zynq-7000:** FPGA Zynq-7000 cung cấp khả năng tính toán song song, giúp tăng tốc các tác vụ tính toán phức tạp. FPGA này tích hợp cả CPU ARM và phần cứng FPGA, giúp tận dụng sức mạnh của cả hai.
 - **Bộ xử lý ARM Cortex-A9:** PYNQ-Z2 tích hợp bộ xử lý ARM Cortex-A9, cho phép thực thi các tác vụ tính toán cao cấp và điều khiển các thiết bị ngoài vi một cách hiệu quả.
 - **Cổng kết nối và giao tiếp linh hoạt:** PYNQ-Z2 cung cấp các giao tiếp mạnh mẽ như HDMI, USB, Ethernet và GPIO, giúp kết nối dễ dàng với các thiết bị bên ngoài và via mạng.
 - **Bộ nhớ DDR3 1GB:** Kit này trang bị bộ nhớ DDR3 1GB cho phép xử lý dữ liệu nhanh chóng, đặc biệt là khi làm việc với các mô hình học sâu hoặc xử lý tín hiệu.
 - **Các đầu vào/ra linh hoạt (I/O):** PYNQ-Z2 cung cấp nhiều cổng I/O có thể cấu hình, giúp kết nối các thiết bị bên ngoài như cảm biến, màn hình và các bộ xử lý tín hiệu.

<p>ZYNQ XC7Z020-1CLG400C</p> <ul style="list-style-type: none"> • 650MHz dual-core Cortex-A9 processor • DDR3 memory controller with 8 DMA channels and 4 High Performance AXI3 Slave ports • High-bandwidth peripheral controllers: 1G Ethernet, USB 2.0, SDIO • Low-bandwidth peripheral controller: SPI, UART, CAN, I2C • Programmable from JTAG, Quad-SPI flash, and microSD card • Programmable logic equivalent to Artix-7 FPGA <ul style="list-style-type: none"> • 13,300 logic slices, each with four 6-input LUTs and 8 flip-flops • 630 KB of fast block RAM • 4 clock management tiles, each with a phase-locked loop (PLL) and mixed-mode clock manager (MMCM) • 220 DSP slices • On-chip analog-to-digital converter (XADC) <p>Memory</p> <ul style="list-style-type: none"> • 512MB DDR3 with 16-bit bus @ 1050Mbps • 16MB Quad-SPI Flash with factory programmed 48-bit globally unique EUI-48/64™ compatible identifier • microSD slot <p>Power</p> <ul style="list-style-type: none"> • Powered from USB or 7V-15V external power source 	<p>USB and Ethernet</p> <ul style="list-style-type: none"> • Gigabit Ethernet PHY • Micro USB-JTAG Programming circuitry • Micro USB-UART bridge • USB 2.0 OTG PHY (supports host only) <p>Audio and Video</p> <ul style="list-style-type: none"> • HDMI sink port (input) • HDMI source port (output) • I2S interface with 24bit DAC with 3.5mm TRRS jack • Line-in with 3.5mm jack <p>Switches, Push-buttons and LEDs</p> <ul style="list-style-type: none"> • 4 push-buttons • 2 slide switches • 4 LEDs • 2 RGB LEDs <p>Expansion Connectors</p> <ul style="list-style-type: none"> • Two standard Pmod ports <ul style="list-style-type: none"> • 16 Total FPGA I/O (8 shared pins with Raspberry Pi connector) • Arduino Shield connector <ul style="list-style-type: none"> • 24 Total FPGA I/O • 6 Single-ended 0-3.3V Analog inputs to XADC • Raspberry Pi connector <ul style="list-style-type: none"> • 28 Total FPGA I/O (8 shared pins with Pmod A port)
---	---

Hình 7: Thông số kỹ thuật kit PYNQ-Z2

2.5.2 Lợi ích và ứng dụng của PYNQ Z2

Kit PYNQ-Z2 là nền tảng lý tưởng cho các ứng dụng trong nhiều lĩnh vực như học máy, xử lý tín hiệu số và hệ thống nhúng. Các ứng dụng điển hình bao gồm:

- Phát triển hệ thống học máy và AI:

PYNQ-Z2 giúp tăng tốc các mô hình học sâu (deep learning models) và các thuật toán AI nhờ khả năng xử lý song song của FPGA, giúp giảm thời gian tính toán và tăng hiệu quả.

- Ứng dụng trong xử lý tín hiệu số (DSP):

Kit có thể được sử dụng trong các hệ thống radar, nhận diện giọng nói, xử lý hình ảnh thời gian thực, nhờ vào khả năng xử lý song song và tốc độ cao.

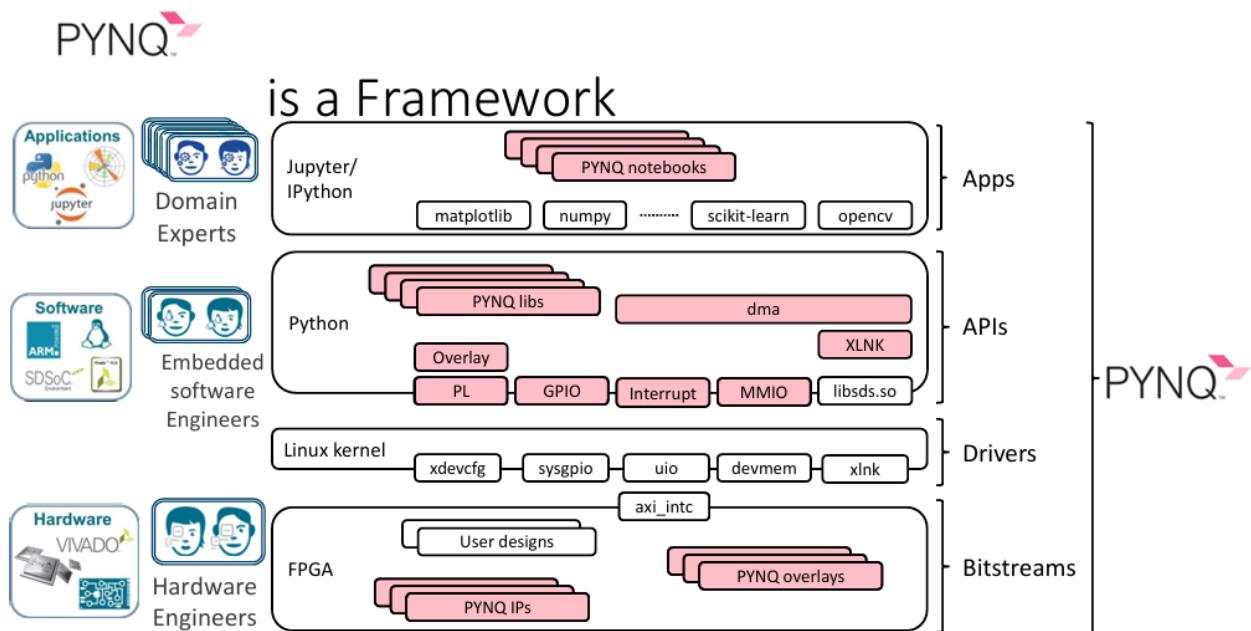
- Hệ thống nhúng:

Với khả năng xử lý mạnh mẽ và tiêu thụ năng lượng thấp, PYNQ-Z2 là công cụ lý tưởng cho các ứng dụng trong tự động hóa, IoT và các hệ thống điều khiển trong công nghiệp.

- Giáo dục và nghiên cứu:

Kit PYNQ-Z2 rất phù hợp cho các nghiên cứu viên và sinh viên trong lĩnh vực kỹ thuật và khoa học máy tính, giúp họ dễ dàng tiếp cận và làm việc với FPGA và các ứng dụng học sâu.

2.5.3 PYNQ Framework (PYNQ Python API)



Hình 8: PYNQ Framework

Một điểm nổi bật của PYNQ-Z2 là PYNQ Framework, cho phép lập trình viên phát triển ứng dụng FPGA mà không cần phải viết mã phần cứng trực tiếp. Việc sử dụng Python, một ngôn ngữ lập trình phổ biến và dễ sử dụng, giúp các nhà phát triển dễ dàng tương tác với FPGA và tối ưu hóa các ứng dụng.

- Phát triển ứng dụng với Python: PYNQ Framework hỗ trợ sử dụng Python để lập trình FPGA, giúp các nhà phát triển không cần kiến thức chuyên sâu về VHDL hay Verilog.
- Tối ưu hóa phần cứng: Các tác vụ tính toán có thể được triển khai trên phần cứng FPGA, trong khi phần mềm điều khiển có thể được phát triển bằng Python.
- Giao diện đồ họa (GUI): PYNQ-Z2 hỗ trợ giao diện người dùng đồ họa, giúp dễ dàng triển khai và kiểm thử các ứng dụng mà không cần kiến thức về FPGA.

2.5.4 Triển khai CNN trên PYNQ Z2

PYNQ-Z2 rất phù hợp cho việc triển khai các mô hình học sâu, đặc biệt là các mô hình CNN (Mạng Nơ-ron Tích chập), nhờ vào khả năng tăng tốc tính toán của FPGA và bộ xử lý ARM.

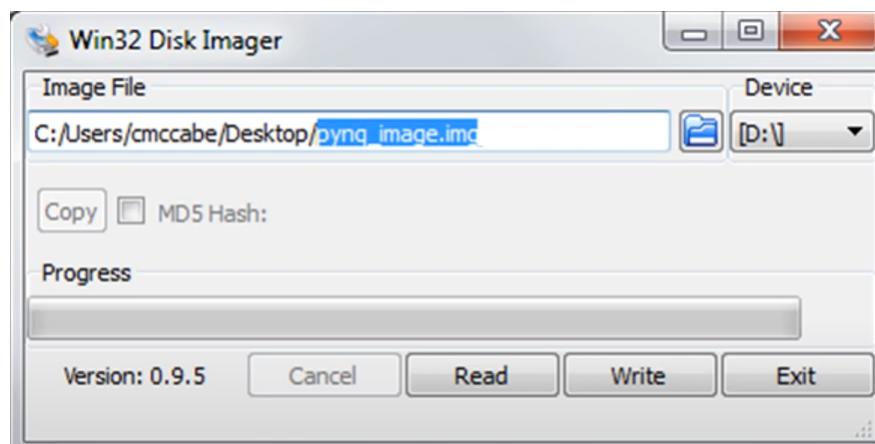
- Tăng tốc tính toán: FPGA giúp tăng tốc các phép toán nhị phân và các phép toán phức tạp trong CNN, như phép toán tích chập và pooling.

- Giảm độ trễ: Khả năng xử lý song song của FPGA giúp giảm độ trễ khi thực hiện các tác vụ tính toán phức tạp, quan trọng trong các ứng dụng nhận diện hình ảnh thời gian thực.
- Hỗ trợ phát triển nhanh chóng: PYNQ Framework giúp các nhà phát triển dễ dàng triển khai các mô hình học sâu mà không cần viết mã phần cứng từ đầu.

2.5.5 Cách kết nối kit PYNQ-Z2

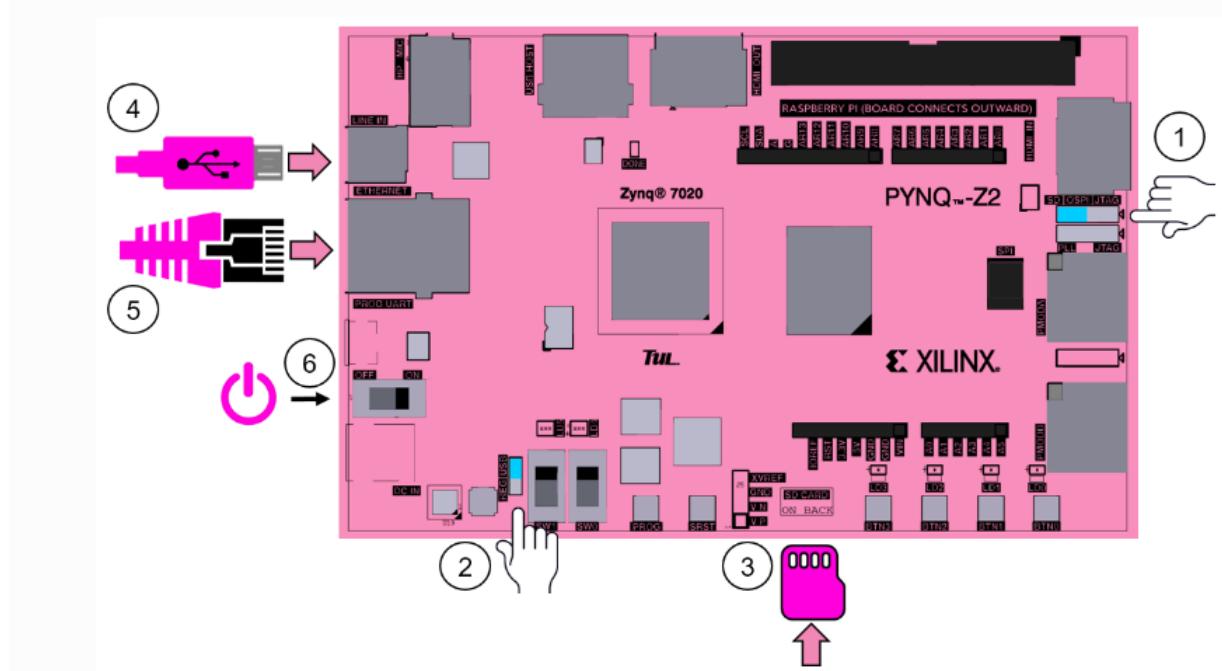
Chuẩn bị:

- PYNQ-Z2 board
- Máy tính có sẵn trình duyệt thông dụng
- Dây cáp Ethernet
- Dây micro USB
- 1 thẻ SD đã được tải Image có OS cấu hình sẵn. Ta download image từ web "pynq.io" và sử dụng "Win32DiskImager utility" để chèn image OS vào thẻ SD.



Hình 9: Chèn image OS vào thẻ SD

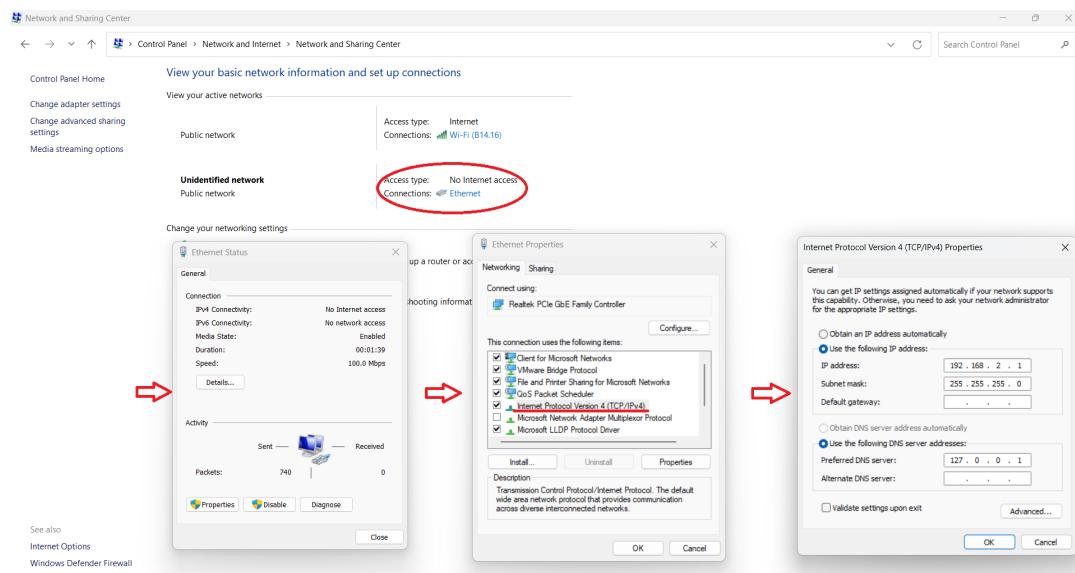
Tiến hành Setup:



Hình 10: Hình ảnh kết nối kit PYNQ-Z2

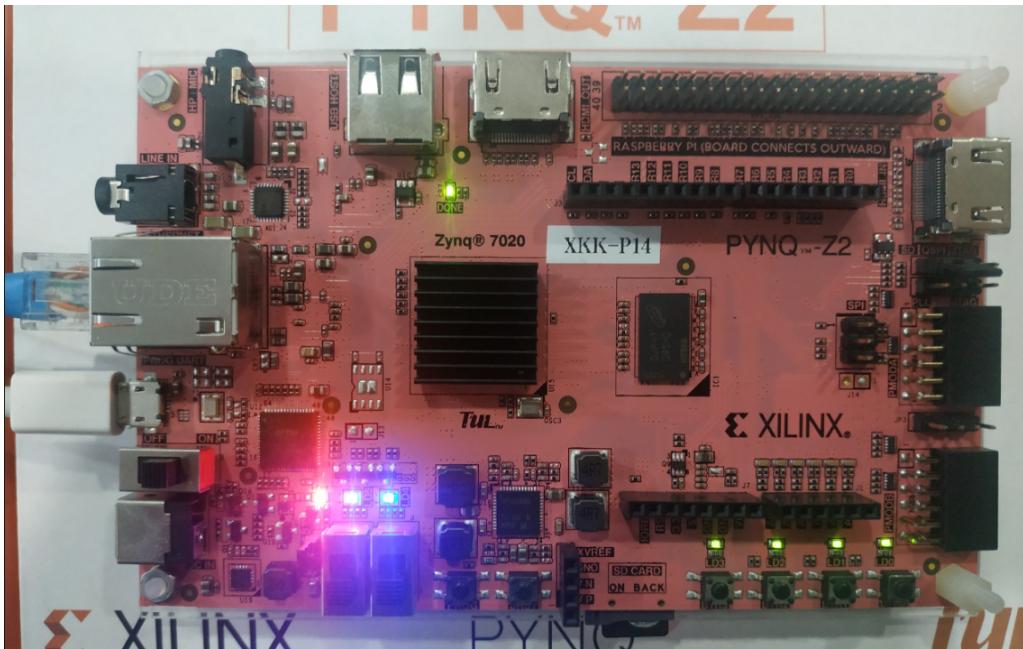
Ta thực hiện các kết nối Micro USB, Ethernet, thẻ SD và gắn các jumpers nguồn và SD theo hình ảnh trên.

Bật nguồn kit và mở cửa sổ Network and Sharing Center, set lại địa chỉ IP và DNS theo hướng dẫn sau:



Hình 11: Set IP và DNS cho kit

Sau khi set địa chỉ IP và DNS cho kit ta đợi khoảng 1 phút cho đến khi đèn "DONE" màu vàng sáng, sau đó 2 LED RCB chớp nháy, 4 LED TÙ 0-3 sẽ sáng đèn. Khi này ta đã kết nối kit với PC thành công.



Hình 12: Hình ảnh kit đã kết nối thành công

2.6 Bộ dữ liệu GTSRB (German Traffic Sign Recognition Benchmark)

Bộ Dữ Liệu GTSRB là một bộ dữ liệu nổi tiếng được sử dụng để huấn luyện và thử nghiệm các mô hình nhận diện biển báo giao thông. Bộ dữ liệu này bao gồm hàng nghìn ảnh biển báo giao thông được chụp tại Đức, và các ảnh này được phân loại thành nhiều nhóm biển báo giao thông khác nhau, ví dụ như biển báo giao nhau, biển báo tốc độ, biển báo cấm, và nhiều loại biển báo khác. GTSRB là một bộ dữ liệu quan trọng trong nghiên cứu nhận diện hình ảnh, đặc biệt là trong các hệ thống tự lái và các ứng dụng giao thông thông minh.

Bộ dữ liệu GTSRB chứa các loại biển báo giao thông khác nhau và được phân chia thành các nhóm:

- Biển báo giao nhau: Các biển báo báo hiệu cho người lái xe về các ngã ba, ngã tư.
- Biển báo tốc độ: Biển báo giới hạn tốc độ cho phép trong khu vực.
- Biển báo cấm: Cấm các hành vi giao thông cụ thể như cấm rẽ trái, rẽ phải hoặc đi vào đường cấm.

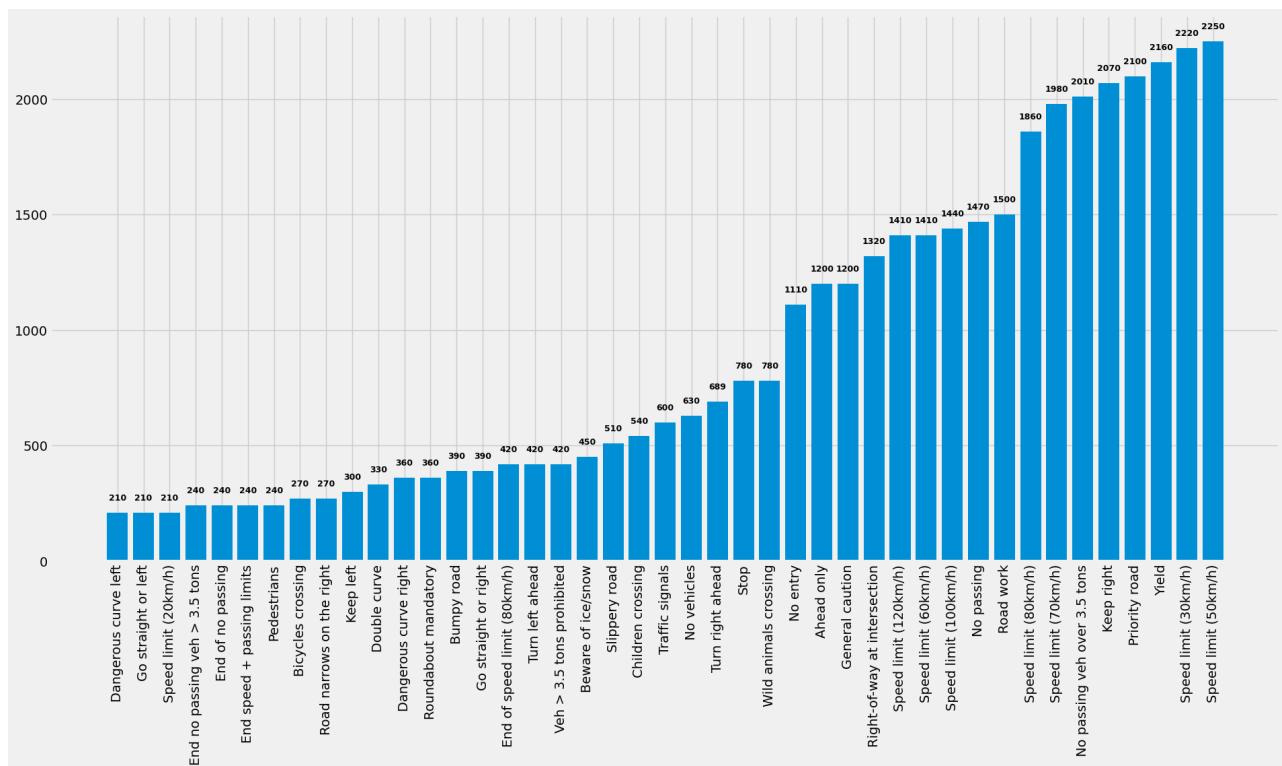
Các ảnh trong bộ dữ liệu GTSRB có độ phân giải cao và được gắn nhãn chính xác với các loại biển báo, giúp mô hình học cách phân loại biển báo giao thông một cách hiệu quả.



Hình 13: Tập dữ liệu German Traffic Sign Recognition Benchmark

Ta có kích thước tập Training sử dụng trong bài là 39210 bức hình và tập Test là 12630 bức hình.

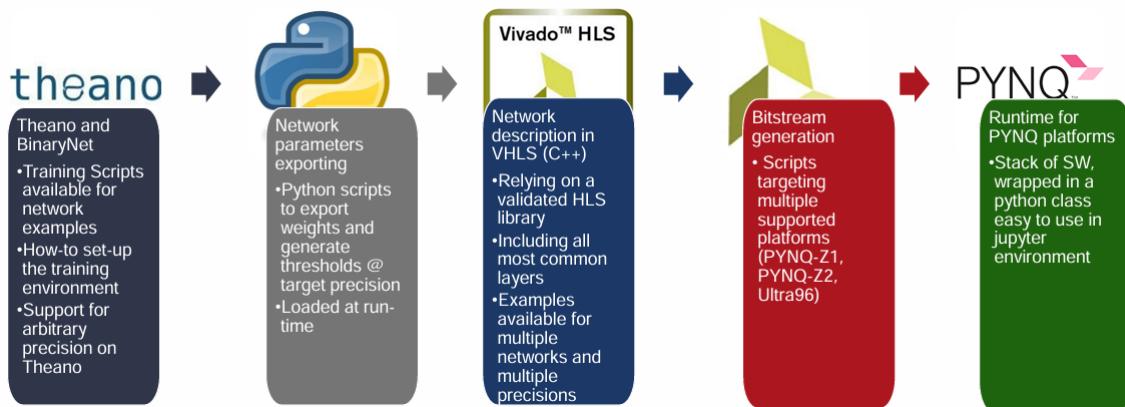
Biểu đồ số tệp training của GTSRB được sử dụng



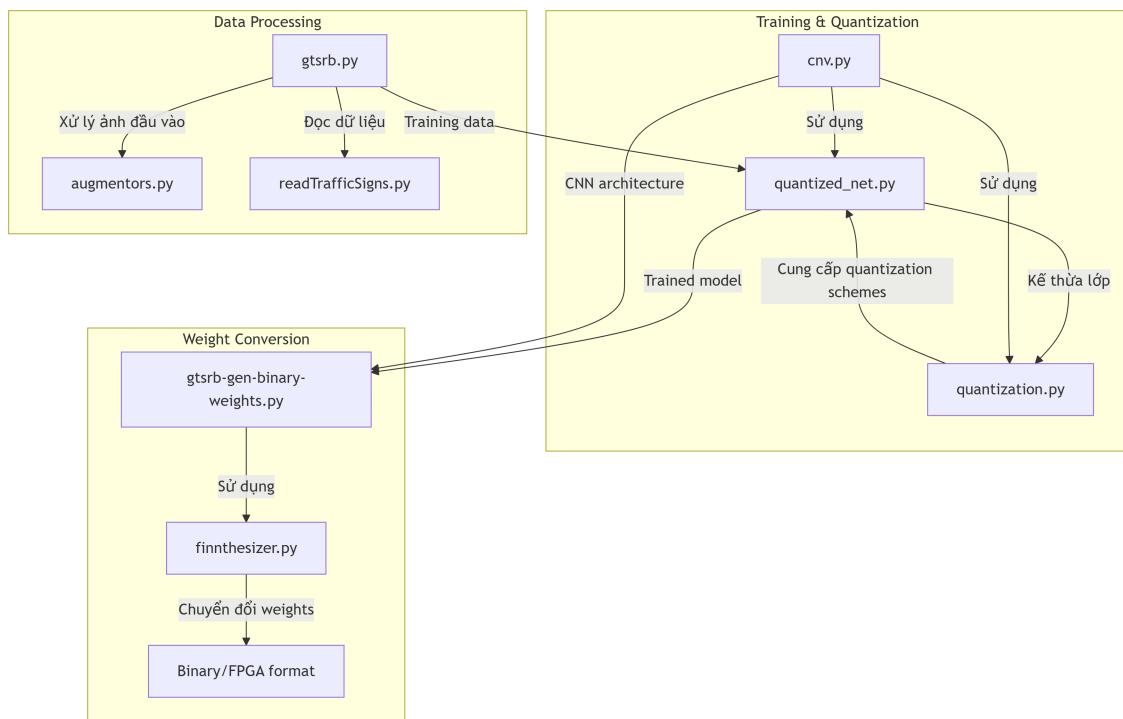
Hình 14: Biểu đồ số tệp training

3 Huấn luyện tập dữ liệu với Python

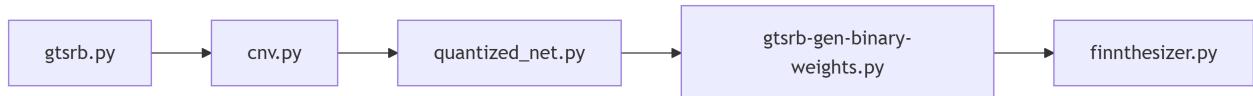
3.1 Sơ đồ giải thuật tổng quát



Hình 15: Các bước thiết kế



Hình 16: Sơ đồ giải thuật tổng quát



Hình 17: Luồng hoạt động chính

1. Core Quantization (quantization.py):

- Cung cấp các lớp quantization (QuantizationBinary, QuantizationFixed) để chuyển đổi weights/activations sang dạng nhị phân hoặc fixed-point.
- Implement các hàm quantization như binary_tanh_unit, hard_sigmoid.

2. Network Layers (quantized_net.py):

- Kế thừa từ Lasagne để tạo các layer quantized (DenseLayer, Conv2DLayer).
- Sử dụng quantization schemes từ quantization.py.

3. Network Architectures:

- cnv.py: Xây dựng kiến trúc CNN binary cho GTSRB.

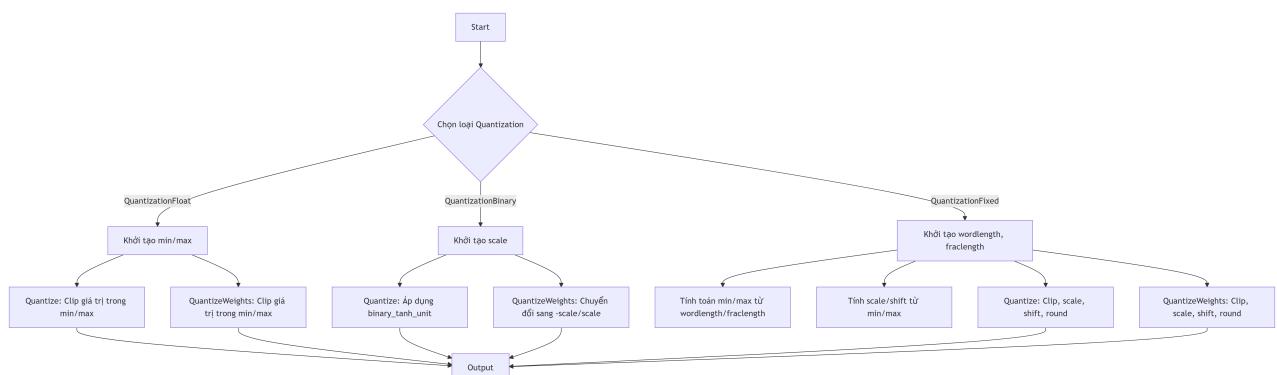
4. Data Pipeline:

- gtsrb.py: Xử lý dataset GTSRB (resize, normalize, augment).
- augmentors.py: Thực hiện data augmentation (xoay ảnh, crop).
- readTrafficSigns.py: Đọc dữ liệu ảnh từ thư mục.

5. Weight Conversion:

- gtsrb-gen-binary-weights.py: Script chính để chuyển đổi weights sang định dạng FPGA.
- finntesizer.py: Thư viện của các hàm:
 - Đóng gói weights thành binary format.
 - Tạo threshold memories.
 - Sinh file config cho HLS.

3.2 Core Quantization (quantization.py)

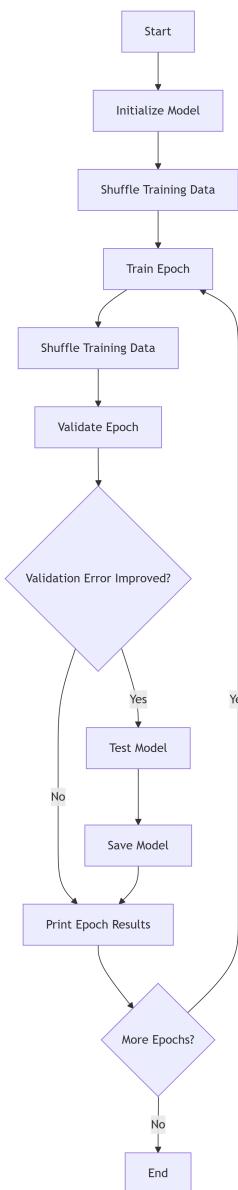


Tệp này cung cấp các lớp và hàm để thực hiện quantization (làm tròn dữ liệu trọng số và kích hoạt) cho mạng nơ-ron:

- QuantizationFloat:
 - Đơn giản clip giá trị đầu vào trong khoảng [min, max].
 - Không có phép làm tròn, giữ nguyên dạng float.

- QuantizationBinary: Lớp này thực hiện quantization nhị phân (chuyển đổi trọng số và kích hoạt thành các giá trị nhị phân -1 hoặc +1). Hàm quantize sử dụng một hàm binary_tanh_unit để chuyển đổi các giá trị đầu vào thành giá trị nhị phân bằng cách áp dụng hàm kích hoạt hard_sigmoid và sau đó làm tròn chúng.
- QuantizationFixed: Lớp này thực hiện quantization fixed-point (một kiểu làm tròn sang các giá trị cố định với một độ phân giải cụ thể). quantizeWeights và quantize sử dụng phương pháp làm tròn có độ chính xác cố định cho trọng số và kích hoạt.
- hard_sigmoid và binary_tanh_unit: Các hàm này là các hàm kích hoạt được sử dụng trong mạng BNN. binary_tanh_unit thực hiện hàm kích hoạt nhị phân, với hard_sigmoid là bước trung gian.

3.3 Network Layers (quantized_net.py)

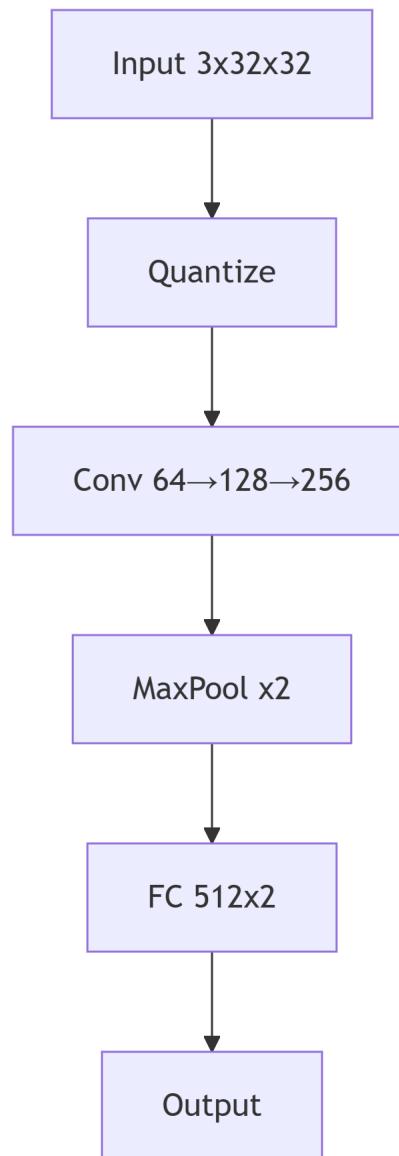


Hình 19: Sơ đồ giải thuật quantization_net.py

Trong tệp này, các lớp mạng được mở rộng từ Lasagne để hỗ trợ quantized layers:

- DenseLayer và Conv2DLayer: Các lớp này mở rộng các lớp DenseLayer và Conv2DLayer trong Lasagne để hỗ trợ quantization cho trọng số (weights) và kích hoạt (activations).
- Các lớp này sử dụng các phương thức từ quantization.py để quantize trọng số và kích hoạt trong quá trình huấn luyện. Quá trình này đảm bảo rằng các trọng số và kích hoạt trong mạng sẽ được lưu dưới dạng nhị phân hoặc fixed-point, tối ưu cho việc triển khai phần cứng.
- Các lớp get_output_for và get_output trong các lớp này đảm bảo rằng kết quả tính toán sẽ được thực hiện với trọng số đã được quantized.

3.4 Network Architectures (cnv.py)



Hình 20: Sơ đồ giải thuật cnv.py

3.4.1 Mục đích thiết kế

cnv.py: Tệp này xây dựng kiến trúc mạng nơ-ron tích chập (CNN) nhị phân để nhận diện biển báo giao thông từ GTSRB (German Traffic Sign Recognition Benchmark). Mạng CNN này sử dụng các lớp Conv2DLayer và DenseLayer đã được quantized để tiết kiệm tài nguyên phục vụ việc thực hiện việc nhận diện. Đảm bảo yêu cầu thiết kế một CNN nhỏ gọn phù hợp với phần cứng FPGA/Xilinx.

3.4.2 Kiến trúc mạng CNN thiết kế

Tên	Chức năng
Input	Ảnh đầu vào 3 kênh (RGB), kích thước 32x32.
Tiền xử lý	Lượng tử hóa đầu vào về 8-bit (FixedHardTanH).
Conv Blocks	6 tầng tích chập ($64 \rightarrow 128 \rightarrow 256$ filters), kèm BatchNorm và Activation.
Pooling	2 tầng MaxPool (giảm kích thước không gian).
FC Layers	2 tầng fully-connected (512 units).
Output	Lớp đầu ra với $num_outputs$ units (tùy chỉnh).

Bảng 1: Chi tiết cấu trúc mô hình

3.4.3 Các tham số quan trọng

Tham số	Mô tả
$num_outputs$	Số lượng lớp đầu ra (1–64).
learning_parameters	Chứa cấu hình: $weight_bits$, $activation_bits$, W_LR_scale , ...
epsilon, alpha	Dùng cho BatchNorm (tránh chia 0, điều chỉnh đạo hàm).

Bảng 2: Các tham số quan trọng

3.4.4 Chức năng lượng tử hóa

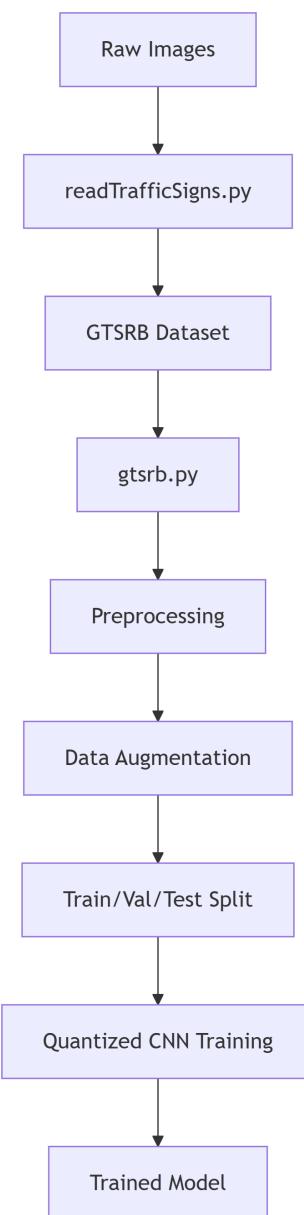
- Đầu vào: Lượng tử hóa về 8-bit (dải giá trị rộng).
- Trọng số:
 - Có thể dùng 1-bit (binary) hoặc n-bit fixed-point (tùy $weight_bits$).
 - Ví dụ: Nếu $weight_bits=1$, trọng số chỉ nhận giá trị -1 hoặc +1.
- Activation:
 - Dùng FixedHardTanH (lượng tử hóa sau hàm kích hoạt).

3.5 Data Pipeline (gtsrb.py, augmentors.py, readTrafficSigns.py)

Khối này mô tả quy trình xử lý dữ liệu và huấn luyện mô hình nhận diện biển báo giao thông, từ khi đọc ảnh gốc đến khi tạo ra mô hình đã được lượng tử hóa. Các bước chính bao gồm tiền xử lý ảnh, tăng cường dữ liệu, chia tập train/val/test và huấn luyện mô hình CNN.

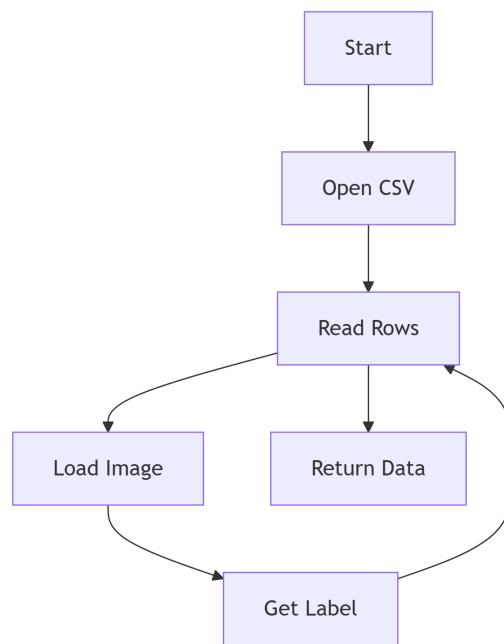
Input: Ảnh traffic sign từ GTSRB dataset.

Output: Dữ liệu đã chuẩn hóa + augmented, sẵn sàng cho training.



Hình 21: Luồng xử lý khối Data Pipeline

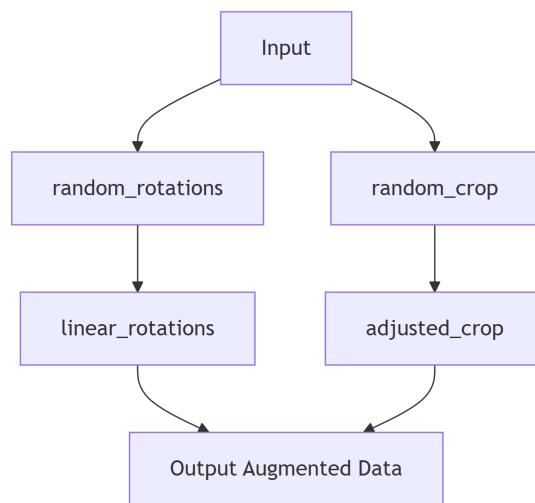
3.5.1 Đọc ảnh và nhãn từ file CSV (readTrafficSigns.py)



Hình 22: Sơ đồ giải thuật readTrafficSigns.py

Đọc dữ liệu ảnh từ thư mục và lưu trữ nhãn của biển báo giao thông (tập tin CSV chứa thông tin nhãn).

3.5.2 Xoay và cắt ngẫu nhiên (augmentors.py)



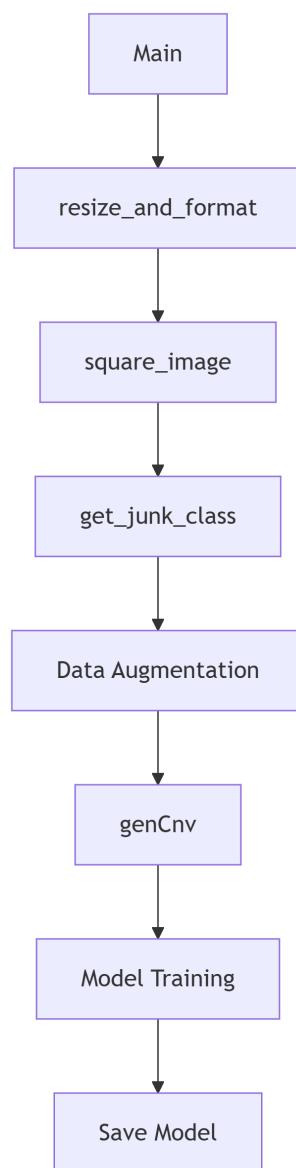
Hình 23: Sơ đồ giải thuật augmentors.py

Tệp cấp các hàm data augmentation (tăng cường dữ liệu) như xoay ảnh (rotations) và cắt ảnh (cropping) để tăng cường tính đa dạng của dữ liệu huấn luyện.

3.5.3 Khối tiền xử lý (gtsrb.py)

File gtsrb.py là một pipeline hoàn chỉnh để xử lý và huấn luyện mô hình nhận diện biển báo giao thông trên tập dữ liệu GTSRB. Nó thực hiện các bước từ tiền xử lý ảnh (resize, normalize), augment dữ liệu (xoay, cắt ngẫu nhiên) đến xây dựng và huấn luyện mô hình CNN lượng tử hóa.

- Đầu vào:
 - Thư mục ảnh GTSRB (cấu trúc thư mục chuẩn).
 - Các tham số huấn luyện (bit-width, learning rate,...).
- Đầu ra:
 - File model .npz (vd: gtsrb-1w-1a.npz).
 - Log quá trình training (loss/accuracy).

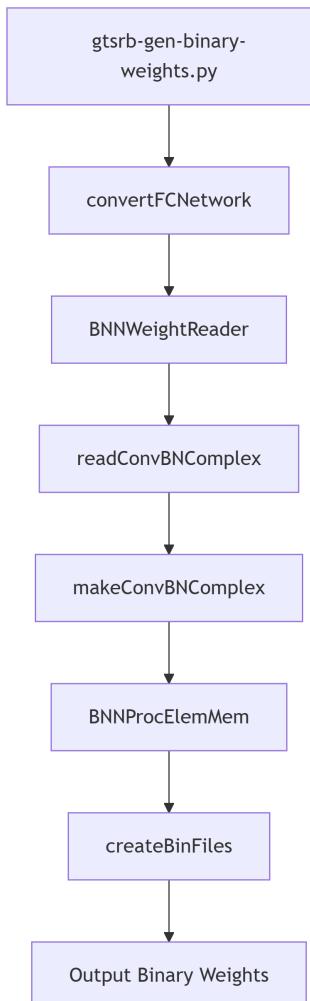


Hình 24: Sơ đồ giải thuật gtsrb.py

Chức năng	Mô tả
Xử lý dữ liệu	<ul style="list-style-type: none"> - Đọc ảnh từ thư mục (sử dụng <i>readTrafficSigns.py</i>) - Resize về 32x32, chuẩn hóa pixel về [-1, 1] - Tạo ảnh vuông bằng zero-padding
Data Augmentation	Áp dụng xoay ngẫu nhiên ($\pm 7^\circ$) và cắt ảnh (dùng <i>augmentors.py</i>)
Chia tập dữ liệu	Chia <i>train/val/test</i> theo tỉ lệ 80%/10%/10%, thêm lớp "junk" (nhiều)
Xây dựng model	<ul style="list-style-type: none"> - Tạo kiến trúc CNN từ <i>cnn.py</i> với: - Lượng tử hóa 1-bit cho weights/activations - Hàm mất mát Squared Hinge Loss
Huấn luyện	Sử dụng Adam optimizer, learning rate decay, và batch normalization
Lưu model	Xuất file <i>.npz</i> chứa trọng số đã huấn luyện

Bảng 3: Chi tiết các chức năng mô hình gtsrb.py

3.6 Weight Conversion (gtsrb-gen-binary-weights.py, finnthesizer.py)



Hình 25: Sơ đồ giải thuật Weight Conversion

3.6.1 Thư viện finnthesizer.py:

Thư viện này giúp tạo các binary weights và thresholds cần thiết để tải vào bộ nhớ của FPGA. Nó đóng vai trò quan trọng trong việc tối ưu hóa mô hình cho FPGA, bao gồm cả việc tạo các tệp khởi tạo cho Vivado HLS và đóng gói trọng số và threshold thành các tệp nhị phân cho việc khởi tạo trong quá trình runtime.

3.6.2 gtsrb-gen-binary-weights.py

Khối gtsrb-gen-binary-weights.py chủ yếu thực hiện việc đọc và xử lý các trọng số đã huấn luyện từ file .npz, sau đó chuyển đổi chúng thành định dạng nhị phân để có thể sử dụng trong các hệ thống phần cứng FPGA. Các chức năng chính bao gồm:

- Đọc trọng số từ file: Tải trọng số của mô hình từ file .npz.
- Xử lý các lớp tích chập và fully-connected: Đọc các trọng số của các lớp tích chập (Convolutional Layers) và các lớp fully-connected (FC Layers) của mạng nơ-ron.
- Chuyển đổi trọng số và ngưỡng: Trọng số và ngưỡng (thresholds) được xử lý và chuyển đổi sang dạng nhị phân hoặc các dạng lưu trữ khác phù hợp với FPGA.
- Tạo file cấu hình HLS: File cấu hình HLS được tạo ra để khởi tạo bộ nhớ cho các lớp trong mô hình, sử dụng trong quá trình tạo bitstream cho FPGA.
- Lưu trọng số dưới dạng nhị phân: Tạo các file nhị phân để khởi tạo bộ nhớ trong quá trình chạy mô hình trên FPGA.

Khối gtsrb-gen-binary-weights.py sử dụng thư viện finnthesizer để đọc và chuyển đổi các trọng số của mô hình. Cụ thể:

- Thư viện finnthesizer cung cấp các hàm để đọc trọng số từ các file .npz (ví dụ như hàm BNNWeightReader) và thực hiện các phép toán như lượng tử hóa trọng số và các ngưỡng.
- Chức năng chính của thư viện finnthesizer trong khối gtsrb-gen-binary-weights.py là chuyển đổi các trọng số và ngưỡng của mô hình thành các dạng dữ liệu phù hợp với FPGA (dưới dạng nhị phân hoặc các định dạng chuẩn khác) để có thể sử dụng trong hệ thống phần cứng.

3.6.3 Training

Ta chia tệp training thành 70% cho train và 30% cho validation.

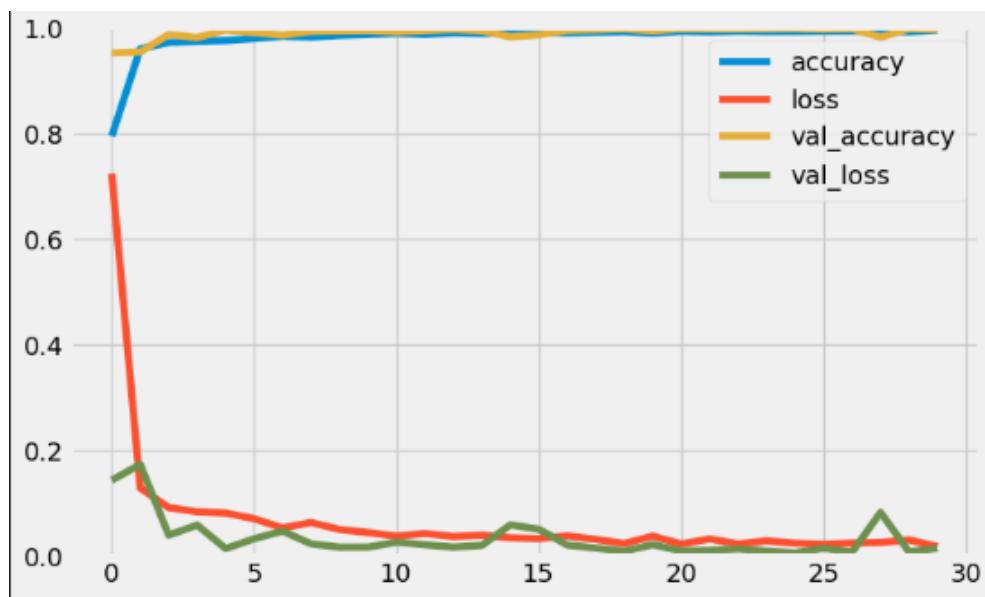
Như ở hình 13, ta thấy sự phân chia dữ liệu hình ảnh ở các Class là không đồng đều, có Class chỉ hơn 200 bức hình nhưng mà có Class lại đến hơn 2000 bức hình, điều này có thể dẫn đến các vấn đề xảy ra như:

- **Mất cân bằng giữa các lớp:**

- Class chiếm ưu thế (2000 ảnh) sẽ khiến mô hình:
 - * Tập trung học features của class này nhiều hơn, dẫn đến dự đoán thiên lệch (bias).
 - * "Lười" học các class ít dữ liệu (200 ảnh) vì sai số từ chúng ít ảnh hưởng đến loss function tổng thể.
- Class thiểu số (200 ảnh) sẽ:
 - * Khó được nhận diện chính xác, dễ bị xếp nhầm vào class đa số.
 - * Mô hình có thể overfit do lượng dữ liệu quá ít để tổng quát hóa.

- **Hậu quả cụ thể khi training:**

- Độ chính xác giả mạo: Accuracy cao (ví dụ 90%) nhưng thực tế mô hình chỉ đoán đúng class đa số, class ít bị "bỏ rơi".
- F1-score thấp cho class thiểu số do precision/recall đều kém.
- Khả năng tổng quát hóa yếu: Mô hình hoạt động kém trên dữ liệu thực tế nếu class thiểu số quan trọng.



Hình 26: Kết quả Training khi chưa áp dụng các biện pháp

Dựa vào biểu đồ ta thấy có hiện tượng Overfitting nhẹ và có thể là mô hình đang học các đặc trưng một cách "thuộc lòng" chứ không tổng quát hóa.

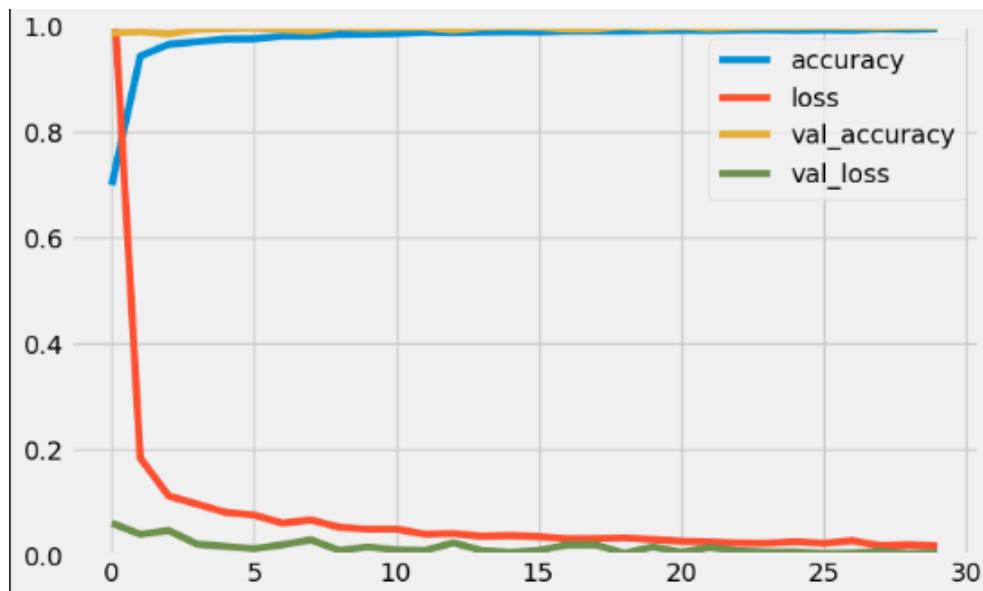
3 kỹ thuật được đưa ra để chống overfitting sử dụng trong bài này:

1. Dropout: Chống overfitting bằng cách "tắt ngẫu nhiên" một tỷ lệ neuron nhất định trong quá trình training. Tác dụng:
 - Buộc mô hình học các features phân tán thay vì phụ thuộc vào vài neuron cụ thể.
 - Hiệu quả với dataset nhỏ hoặc mô hình phức tạp.
2. Batch Normalization (BatchNorm): Ôn định phân phối đầu vào giữa các layer, giúp mô hình hội tụ nhanh hơn. Tác dụng:
 - Giảm hiện tượng "internal covariate shift".
 - Cho phép sử dụng learning rate lớn hơn mà không lo phân kỳ.
3. Data Augmentation: Tăng kích thước dataset ảo bằng cách biến đổi ảnh gốc (xoay, dịch chuyển, zoom,...). Tác dụng:
 - Giảm overfitting khi dataset gốc nhỏ.
 - Cải thiện khả năng tổng quát hóa, đặc biệt với biển báo giao thông (GTSRB) ở các góc độ khác nhau.

3.6.4 Các kết quả sau huấn luyện

Sau khi huấn luyện hoàn thành, tệp NPZ chứa trọng số sẽ được chuyển đổi thành các tệp nhị phân thông qua gtsrb-gen-binary-weights.py và finnthesizer.py. Các tệp này sẽ được sử dụng để tải vào bộ nhớ của FPGA, nơi mô hình sẽ được triển khai và thực hiện nhận diện biển báo giao thông trong thời gian thực.

Biểu đồ quá trình Training của mô hình



Hình 27: Biểu đồ quá trình Training của mô hình

Ta thấy sau khi áp dụng các kỹ thuật, hiện tượng overfitting hầu như không còn xuất hiện đáng kể nữa và mô hình đã có thể học các đặc trưng một cách tổng quát hơn, chính xác hơn.

1. ACCURACY (Độ chính xác)

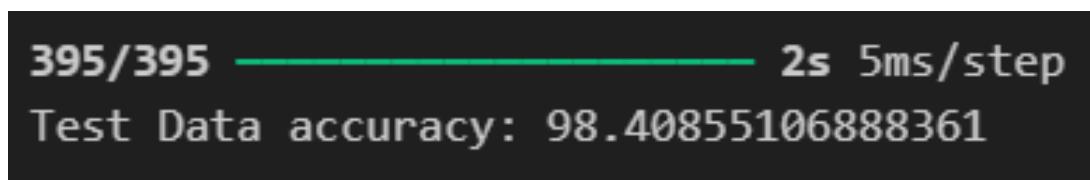
- Training Accuracy (đường xanh)
 - Rất tốt: Tăng mạnh và đạt giá trị khá cao ngay từ các epoch đầu
 - Ổn định: Duy trì mức cao và không dao động
 - Hội tụ nhanh: Chỉ cần 2-3 epochs để đạt hiệu suất tối ưu
- Validation Accuracy (đường vàng)
 - Rất tốt: Đạt giá trị tiệm cận 1 và gần bằng 1 ở các epoch đầu.
 - Nhất quán: Gần như trùng khớp với training accuracy
 - Không overfitting: Không có dấu hiệu suy giảm

2. LOSS (Hàm mất mát)

- Training Loss (đường đỏ)
 - Giảm mạnh: Từ 1.0 xuống 0.1 và bé hơn trong 5 epochs đầu
 - Hội tụ tốt: Tiến về 0 một cách ổn định
- Validation Loss (đường xanh lá)
 - Tốt: Luôn thấp hơn training loss
 - Ổn định: Dao động nhẹ (không đáng kể)
 - Không tăng: Không có dấu hiệu overfitting

Kết quả test độ chính xác

Ta kiểm tra lại tính đúng đắn của mô hình sau khi Train bằng cách kiểm tra nó với tập Test có sẵn của GTSRB. Tập test chứa 12631 dữ liệu hình ảnh được phân bố ngẫu nhiên các Class. Ta



Hình 28: Accuracy của mô hình sau train

thấy độ chính xác của mô hình là khoảng 98.4%, đây là một độ chính xác khá cao. Nhờ vậy nó sẽ giúp đảm bảo được tính chính xác và độ tin cậy khi triển khai mô hình này lên FPGA mà không sợ xảy ra quá nhiều sai số.

4 Thiết kế kiến trúc HLS CNN

Thiết kế kiến trúc High-Level Synthesis (HLS) cho một mạng Convolutional Neural Network (CNN) sử dụng các trọng số nhị phân (1-bit weights) và kích hoạt nhị phân (1-bit activations). Mạng này được thiết kế để chạy trên phần cứng, sử dụng mô hình dữ liệu AXI Lite cho việc tải tham số và xử lý dataflow architecture cho việc suy luận hình ảnh (image inference).

Ta sẽ lập trình xử lý phân loại ảnh bằng phần cứng và phần mềm trên kit PYNQ Z2

1. HARDWARE - Sử dụng khối PL (Programmable Logic)

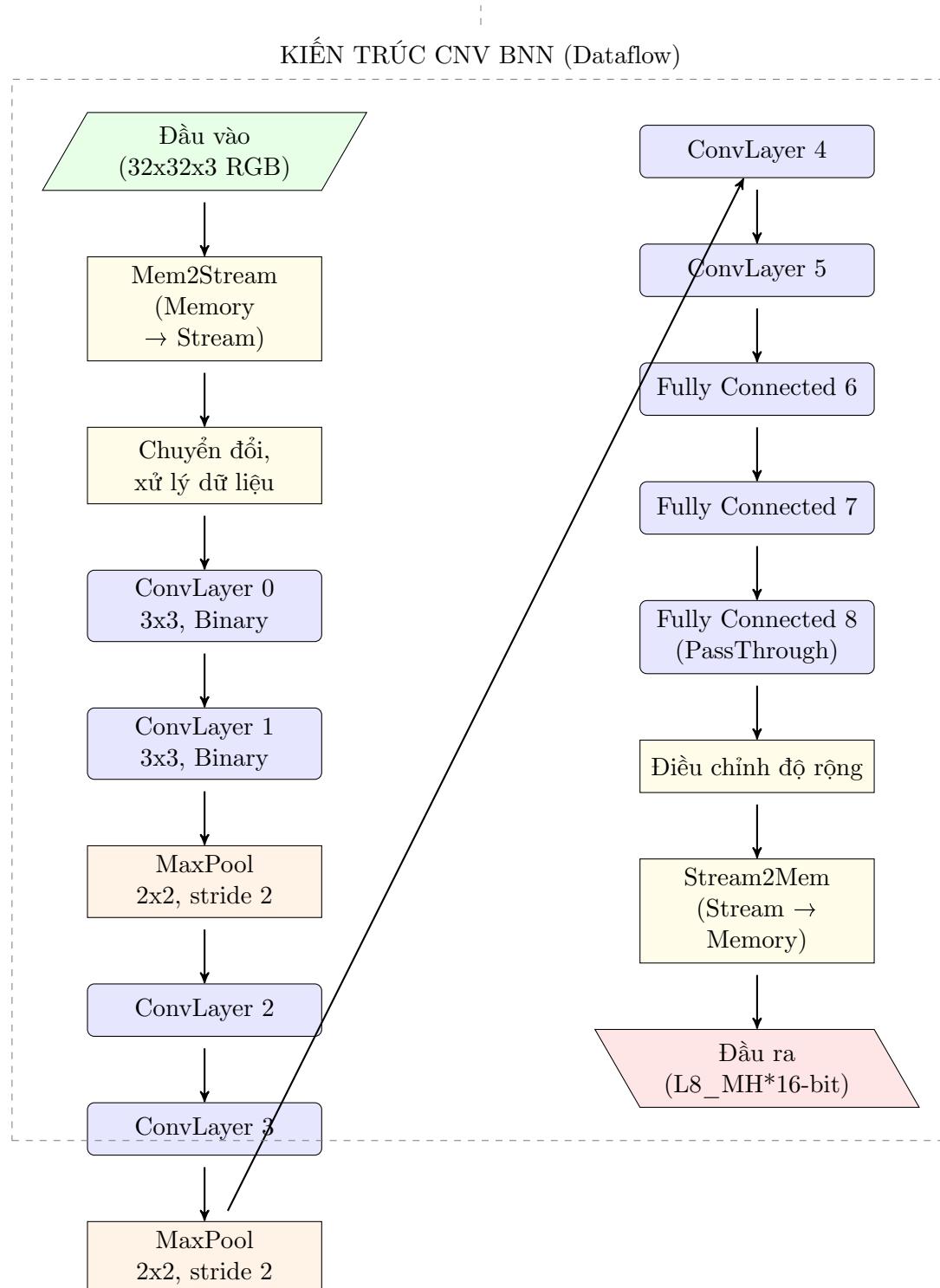
- Khối chính:
 - FPGA Fabric (Xilinx Zynq-7020)
 - Logic cells, LUTs, Flip-flops
 - DSP slices cho tính toán số học
 - Block RAM (BRAM) cho lưu trữ tạm
- Thành phần cụ thể:
 - DSP48E1 slices: Tính toán MAC (Multiply-Accumulate)
 - BRAM: Lưu weights, feature maps
 - AXI Interconnect: Giao tiếp giữa PS-PL
- Ưu điểm:
 - Tốc độ cao: Xử lý song song
 - Tiết kiệm năng lượng: Tối ưu cho inference
 - Latency thấp: Real-time processing

2. SOFTWARE - Sử dụng khối PS (Processing System)

- Khối chính:
 - ARM Cortex-A9 Dual-core (667MHz)
 - DDR3 Memory (512MB)
 - Cache L1/L2
- Thành phần cụ thể:
 - Có các thư viện hỗ trợ sẵn của nhà sản xuất (tiny cnn, cnn)
 - Deep Learning Libraries:
- Ưu điểm:
 - Linh hoạt: Dễ lập trình và debug
 - Ecosystem phong phú: Nhiều thư viện

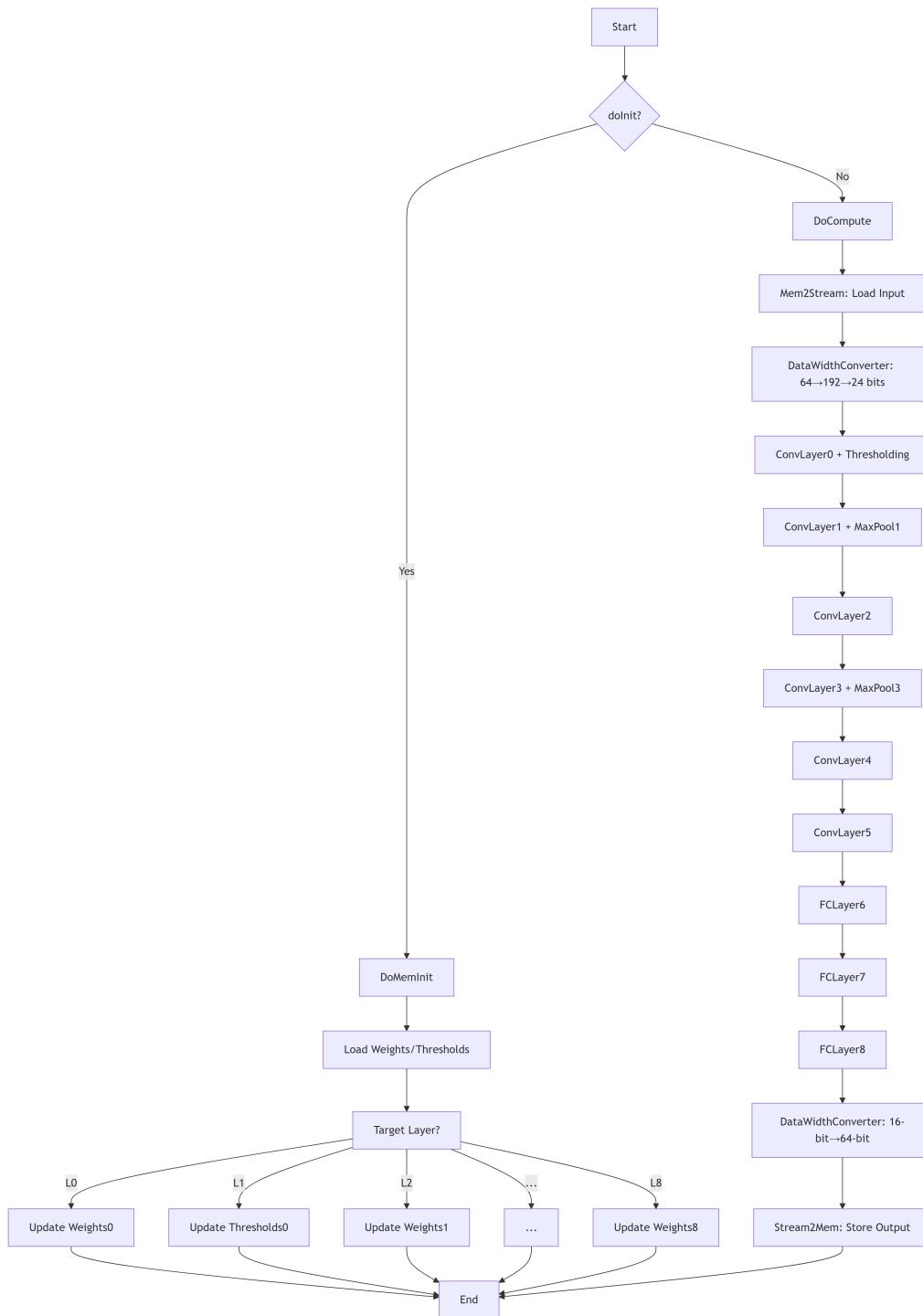
4.1 Thiết kế CNV BNN cho Hardware

4.1.1 Kiến trúc HLS CNV BNN cho Hardware



Hình 29: Kiến trúc CNV BNN

4.1.2 Lập sơ đồ giải thuật



Hình 30: Sơ đồ giải thuật truyền dữ liệu tổng quát bằng HLS

Tổng quan kiến trúc:

- Loại mạng: Binary Neural Network (BNN) - Tối ưu cho FPGA với phép tính XNOR + Bitcount
- Luồng dữ liệu: Pipeline qua 9 lớp (5 Conv + 3 FC) với DATAFLOW
- Tối ưu hóa: Sử dụng Streaming, Parallel Processing (SIMD/PE), và Fixed-point Arithmetic

4.1.3 Chức năng của các hàm chính

- *DoMemInit*: Hàm này được sử dụng để khởi tạo các tham số của mạng, bao gồm trọng số (weights) và ngưỡng (thresholds) cho các lớp trong mạng. Các giá trị này được tải vào bộ nhớ theo yêu cầu của người dùng.
- *DoCompute*: Hàm này thực hiện phép toán suy luận trên mạng nơ-ron tích chập, xử lý các đầu vào và tạo ra đầu ra qua các lớp khác nhau của mạng. Nó bao gồm các lớp chập (convolution layers), lớp pool (max pooling), và các lớp kết nối đầy đủ (fully connected layers).
- *BlackBoxJam*: Hàm này là một giao diện cho việc kết nối với phần mềm thông qua giao thức AXI Lite, cho phép điều khiển việc tải tham số và thực hiện suy luận.

4.1.4 Chi tiết mã nguồn sử dụng

1. Các khai báo tĩnh (static) và đối tượng:

BinaryWeights và ThresholdsActivation là các lớp mô tả các đối tượng trọng số và ngưỡng. Mỗi lớp tương ứng với một lớp trong mạng nơ-ron và sẽ lưu trữ trọng số hoặc ngưỡng của lớp đó.

Các đối tượng như weights0, weights1, threshs0, threshs1 chứa các trọng số và ngưỡng cho mỗi lớp.

2. Hàm paddedSizeHW:

Hàm này tính toán kích thước đã được đệm cho dữ liệu đầu vào sao cho nó phù hợp với kích thước yêu cầu của phần cứng, đảm bảo rằng kích thước của dữ liệu chia hết cho giá trị cần thiết.

3. Hàm DoMemInit:

Hàm này sẽ nhận các tham số xác định lớp (targetLayer), bộ nhớ (targetMem), chỉ mục (targetInd), ngưỡng (targetThresh), và giá trị (val) để lưu trữ giá trị vào bộ nhớ của mạng. Nó sẽ xác định lớp nào và vị trí nào cần được khởi tạo dựa trên tham số truyền vào, sau đó gán giá trị vào bộ nhớ tương ứng.

4. Hàm DoCompute:

Đây là hàm chính thực hiện phép toán suy luận hình ảnh trên mạng. Các bước chính trong hàm này bao gồm:

- Mem2Stream_Batch: Chuyển dữ liệu từ bộ nhớ vào các luồng dữ liệu.
- StreamingDataWidthConverter_Batch: Chuyển đổi độ rộng của dữ liệu giữa các luồng.

- ConvLayer_Batch: Áp dụng các lớp tích chập (convolution layers) vào các luồng dữ liệu.
- StreamingMaxPool_Batch: Áp dụng lớp max pooling.
- StreamingFCLayer_Batch: Áp dụng các lớp kết nối đầy đủ (fully connected layers).
- Cuối cùng, dữ liệu đầu ra được chuyển từ các luồng vào bộ nhớ.

5. Hàm BlackBoxJam:

Đây là một hàm giao diện kết nối với phần mềm qua giao thức AXI Lite. Hàm này có hai chức năng chính:

- Nếu doInit là true, hàm sẽ gọi DoMemInit để khởi tạo các tham số mạng.
- Nếu doInit là false, hàm sẽ gọi DoCompute để thực hiện phép toán suy luận hình ảnh.

Các tham số này có thể được truyền qua các giao diện phần cứng (AXI) để kiểm soát quá trình khởi tạo và tính toán.

6. Các pragma HLS:

- Các chỉ thị HLS như #pragma HLS INTERFACE dùng để xác định các giao diện phần cứng, như giao diện AXI Lite hoặc giao diện AXI Master, để truyền và nhận dữ liệu giữa phần mềm và phần cứng.
- Các chỉ thị HLS ARRAY_PARTITION được sử dụng để phân vùng mảng (arrays) để tăng tốc độ truy cập bộ nhớ trong quá trình thực thi.
- Đây là một mô hình high-level synthesis (HLS) cho một mạng nơ-ron tích chập sử dụng trọng số và kích hoạt nhị phân. Mạng này được tối ưu hóa để chạy trên phần cứng với các phép toán dữ liệu chảy (dataflow architecture) và hỗ trợ khởi tạo tham số qua giao diện AXI Lite.
- Các hàm như DoMemInit và DoCompute thực hiện các bước quan trọng trong việc khởi tạo và tính toán mạng.

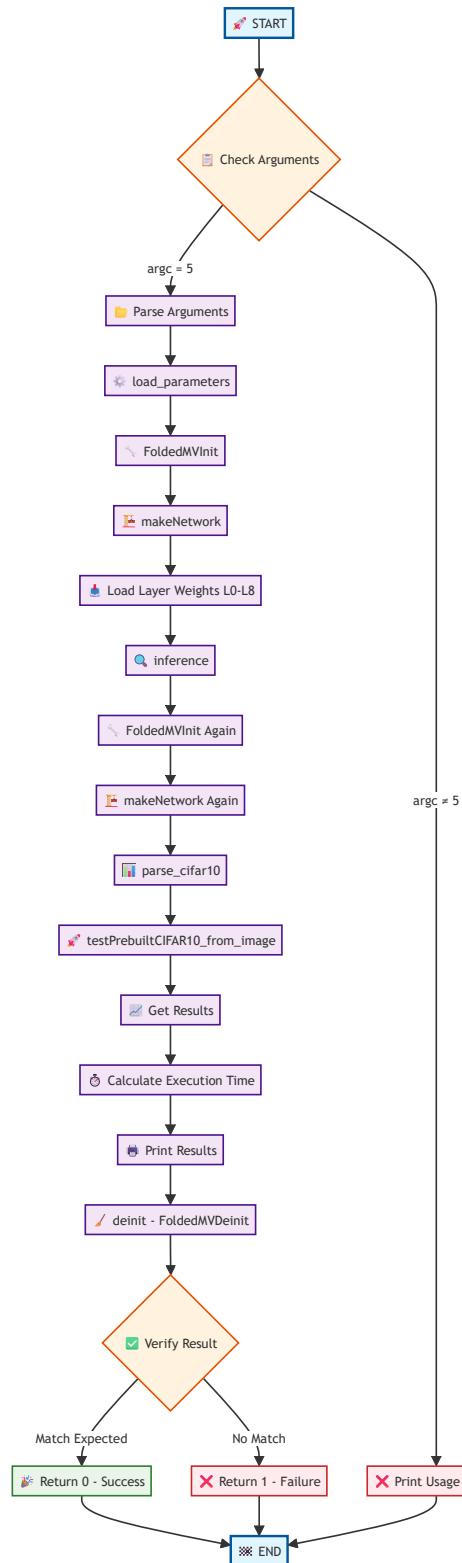
4.1.5 Các thư viện và tệp kèm được sử dụng

1. **config.h:** Tệp này chứa các tham số cấu hình cho các lớp khác nhau của Mạng Nơ-ron Nhị phân (BNN). Nó định nghĩa các hằng số để xác định các thuộc tính của từng lớp, chẳng hạn như số lượng bản đồ đặc trưng đầu vào và đầu ra (kênh), kích thước của các bản đồ đặc trưng, số lượng phần tử xử lý (PE), kích thước bộ nhớ, và các cài đặt phần cứng khác. Những cấu hình này được sử dụng để khởi tạo các lớp với các tham số cụ thể cho việc triển khai trên FPGA. Tệp này được thu hoạch từ phần 4.5 ở trên.
2. **bnn-library.h:** Đây là thư viện các hàm HLS (High-Level Synthesis) cho việc triển khai BNN. Nó bao gồm nhiều hàm mẫu để triển khai các thành phần cơ bản của BNN trên phần cứng (như lớp tích chập, lớp kết nối đầy đủ, hàm kích hoạt, v.v.). Nó cũng bao gồm một số macro để xử lý lỗi (CASSERT_DATAFLOW) và các thư viện để xử lý dòng dữ liệu và bộ nhớ. Tệp này định nghĩa một số thành phần chính như ConvLayer_Batch, StreamingMaxPool, v.v., được sử dụng để mô tả tính toán và di chuyển dữ liệu trong BNN.
3. **maxpool.h:** Tệp này triển khai phép toán max-pooling cho Mạng Nơ-ron Nhị phân. Max-pooling là một phép toán giảm mẫu thường được sử dụng trong các Mạng Nơ-ron Tích chập (CNN) để giảm kích thước không gian của các bản đồ đặc trưng. Các mẫu StreamingMaxPool và StreamingMaxPool_Batch chịu trách nhiệm xử lý các dòng dữ liệu đầu vào và thực hiện phép toán max-pooling.
4. **activations.hpp:** Tệp này có thể chứa các định nghĩa cho các hàm kích hoạt khác nhau, được sử dụng trong các mạng nơ-ron để thêm tính phi tuyến. Mặc dù nội dung cụ thể chưa rõ, nhưng các hàm kích hoạt điển hình bao gồm ReLU (Rectified Linear Unit), sigmoid hoặc tanh. Nó cũng có thể chứa các phép toán ngưỡng cho các giá trị kích hoạt nhị phân.
5. **interpret.hpp:** Tệp này có thể được sử dụng để định nghĩa cách các giá trị nhị phân (như trọng số và kích hoạt nhị phân) được giải thích trong phần cứng. Nó có thể định nghĩa các kiểu và phương thức để xử lý dữ liệu nhị phân một cách tối ưu cho tính toán trên phần cứng, chẳng hạn như FPGA. Tuy nhiên, cần phân tích kỹ hơn nội dung của tệp này để có thông tin chi tiết hơn.
6. **dma.h:** Tệp này cung cấp các hàm để xử lý Truy cập Bộ nhớ Trực tiếp (DMA) giữa bộ nhớ và các dòng dữ liệu trong các triển khai dựa trên FPGA. Nó bao gồm các hàm để chuyển dữ liệu giữa bộ nhớ AXI4 và các dòng HLS, chẳng hạn như Mem2Stream, Stream2Mem, và các phiên bản theo lô của chúng. Những hàm này đảm bảo việc di chuyển dữ liệu hiệu quả cho các ứng dụng tăng tốc phần cứng.

7. **fclayer.h:** Tệp này triển khai các lớp kết nối đầy đủ (FC) trong BNN. Các lớp kết nối đầy đủ chịu trách nhiệm thực hiện phép toán nhân ma trận-vecto, vốn là phép toán cốt lõi của lớp FC, với thêm việc chuyển đổi độ rộng của dòng dữ liệu. Nó bao gồm các hàm để xử lý dữ liệu qua các lớp kết nối đầy đủ bằng cách sử dụng các tài nguyên tối ưu hóa cho phần cứng.
8. **convlayer.h:** Tệp này triển khai các lớp tích chập cho BNN. Các lớp tích chập chịu trách nhiệm áp dụng các phép toán tích chập trên các bản đồ đặc trưng đầu vào (chẳng hạn như hình ảnh) để trích xuất các đặc trưng không gian. Hàm ConvLayer_Batch thực hiện phép toán tích chập, bao gồm việc sinh cửa sổ trượt (im2col), nhân ma trận-vecto, và tính toán hàm kích hoạt. Tệp này hỗ trợ việc triển khai tối ưu hóa các phép toán tích chập trên phần cứng như FPGA.

4.2 Thiết kế CNV BNN cho Software

4.2.1 Sơ đồ giải thuật kiến trúc HLS CNV BNN cho Software



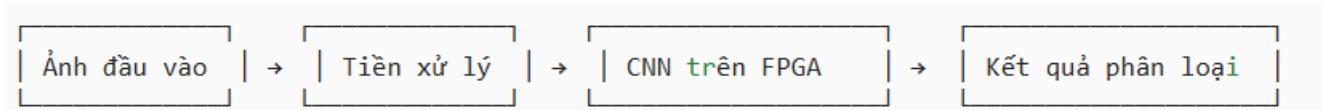
Hình 31: Kiến trúc HLS CNV BNN cho Software

4.2.2 Tổng quan hệ thống

1. Mục đích của chương trình:

- Chức năng chính: Phân loại ảnh CIFAR-10 sử dụng mạng neural tích chập (CNN) với tăng tốc phần cứng
- Mô hình: Mạng neural nhị phân (BNN - Binarized Neural Network) với bộ tăng tốc FoldedMV
- Ứng dụng: Nhận dạng ảnh thời gian thực với hiệu suất cao

2. Kiến trúc hệ thống



Hình 32: Kiến trúc tổng quát hệ thống

3. Thông số kỹ thuật

- Trọng số: 8-bit (nhị phân hóa)
- Activation: 16-bit
- Số lớp: 9 lớp (L0-L8)
- Kiến trúc: CNN với fully-connected cuối

4. Quản lý bộ nhớ

- Static: Mảng scores[64] cho kết quả
- Dynamic: Cấp phát động cho batch processing
- Hardware: Bộ nhớ chuyên dụng trên FPGA

4.2.3 Chi tiết các hàm được sử dụng

1. Hàm makeNetwork() - Tạo kiến trúc mạng

- Đầu vào: 3 kênh màu \times 32 \times 32 pixel (định dạng CIFAR-10)
- Lớp 1: chaninterleave_layer - Sắp xếp lại dữ liệu kênh màu để tối ưu truy cập bộ nhớ
- Lớp 2: offloaded_layer - Lớp được xử lý trên phần cứng FPGA
 - Input: 3072 neuron ($3 \times 32 \times 32$)
 - Output: 10 lớp (classes)
 - Template: <8, 1, ap_int<16>> nghĩa là 8-bit trọng số, 16-bit activation
- Đầu ra: 10 giá trị tương ứng với 10 lớp CIFAR-10

2. Hàm load_parameters() - Tải trọng số

- Khởi tạo: Bộ tăng tốc FoldedMV với cấu hình "cnvW1A1"
- Tải trọng số: Cho 9 lớp (L0-L8), mỗi lớp có:
 - PE: Số lượng Processing Elements (đơn vị xử lý song song)
 - WMEM: Bộ nhớ trọng số
 - TMEM: Bộ nhớ ngưỡng (threshold)
 - API: Giao diện độ chính xác activation

3. Hàm inference() - Suy luận đơn ảnh

Quy trình xử lý:

- Tiền xử lý: Chuẩn hóa ảnh về khoảng [-1, 1]
- Suy luận: Chạy trên bộ tăng tốc với template <8, 16, ap_int<16>
- Hậu xử lý: Tìm lớp có xác suất cao nhất
- Do hiệu suất: Tính thời gian thực thi (microseconds)

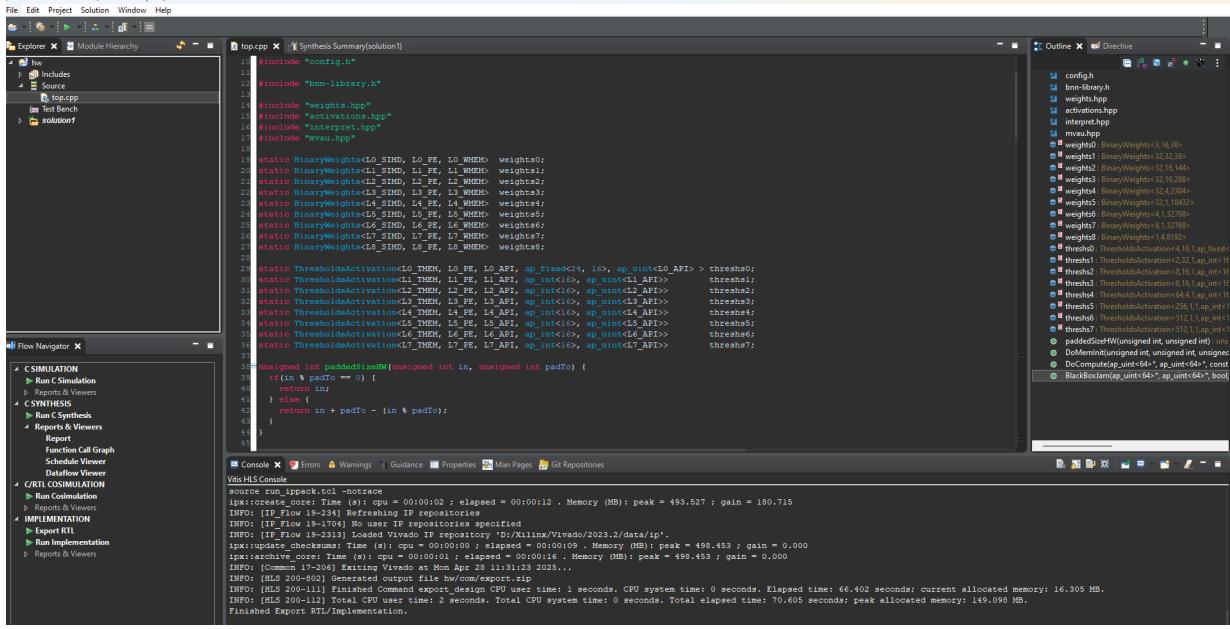
4. Hàm inference_multiple() - Suy luận nhiều ảnh

Tính năng:

- Batch processing: Xử lý nhiều ảnh cùng lúc
- Chi tiết kết quả: Có thể trả về điểm số chi tiết hoặc chỉ kết quả cuối
- Quản lý bộ nhớ: Cấp phát động, cần giải phóng bằng free_results()

4.3 Synthesis HLS và xuất RTL Design

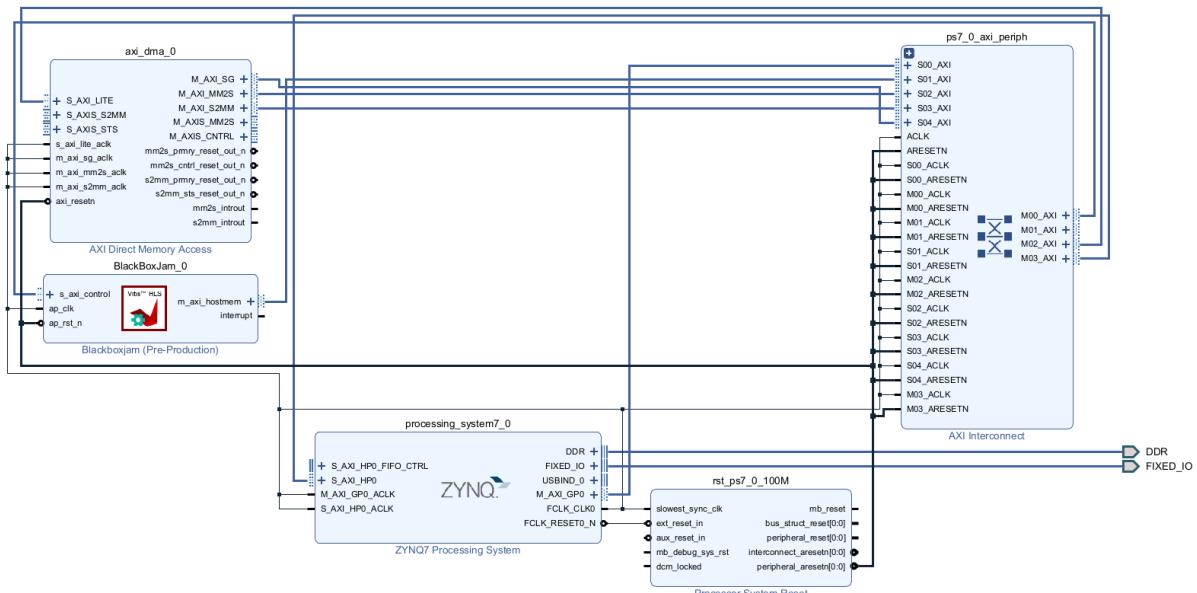
Ta tạo file chứa source code C+ và các thư viện, tệp kèm vào chung 1 folder. Tạo project thêm các file vào, chọn top Function là "BlackBoxJam", tiến hành chạy Synthesis và Export RTL.



Hình 33: Synthesis và Export HLS thành công

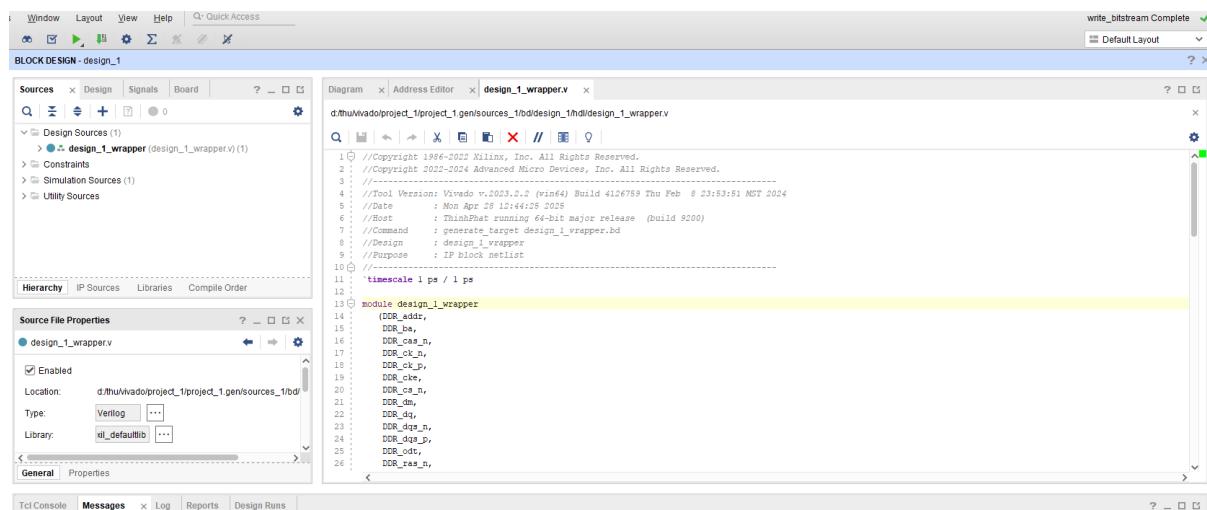
4.4 Tạo block Design bằng Vivado

Từ IP được đóng góp (Export RTL) từ Vitis đã thực hiện ở phần trên, ta tiến hành tạo block Design.



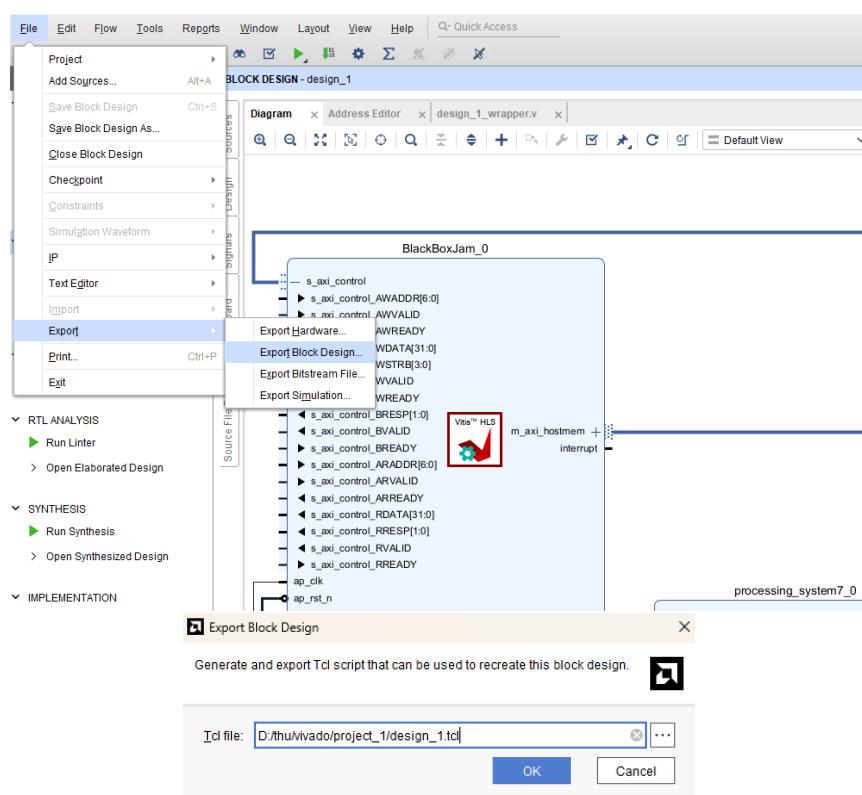
Hình 34: Block Design bộ CNV BNN

Tiến hành các bước Synthesis → Implementation → Generate Bitstream



Hình 35: Xuất Bitstream thành công

Xuất file .TCL (Tool Command Language), tệp lưu trữ các cấu hình dùng cho Overlay.



Hình 36: Xuất file .tcl

Thống kê tài nguyên đã sử dụng khi Implementation lên FPGA.

Design Timing Summary					
WNS(ns)	TNS(ns)	TNS Failing Endpoints	TNS Total Endpoints	WHS(ns)	THS(ns)
0.285	0.000	0	108705	0.020	0.000
All user specified timing constraints are met.					
Resource Utilization Reports					
Site Type	Used	Fixed	Available	Util%	
Slice LUTs	26072	0	53200	49.01	
LUT as Logic	24047	0	53200	45.20	
LUT as Memory	2025	0	17400	11.64	
LUT as Distributed RAM	1578	0			
LUT as Shift Register	447	0			
Slice Registers	41312	0	106400	38.83	
Register as Flip Flop	41312	0	106400	38.83	
Register as Latch	0	0	106400	0.00	
F7 Muxes	892	0	26600	3.35	
F8 Muxes	241	0	13300	1.81	
Site Type	Used	Fixed	Available	Util%	
Block RAM Tile	124	0	140	88.57	
RAMB36/FIFO*	76	0	140	54.29	
RAMB36E1 only	76				
RAMB18	96	0	280	34.29	
RAMB18E1 only	96				
Site Type	Used	Fixed	Available	Util%	
DSPs	24	0	220	10.91	
DSP48E1 only	24				

Hình 37: Utilization reports

Nhận xét về các thông số trong kết quả Design Timing Summary của bộ BNN được triển khai trên Kit PYNQ Z2:

1. WNS (Worst Negative Slack) và TNS (Total Negative Slack)

- WNS (Worst Negative Slack) là độ trễ tiêu cực lớn nhất trong thiết kế. Với giá trị WNS = 0.285 ns, điều này cho thấy có một độ trễ tối thiểu tại một số điểm trong thiết kế. Tuy nhiên, WNS vẫn ở mức dương (dù rất nhỏ), cho thấy mô hình có thể thực thi theo yêu cầu thời gian, nhưng có thể cần phải điều chỉnh nhỏ để cải thiện hiệu suất.
- TNS (Total Negative Slack): Đây là giá trị tổng của tất cả các độ trễ tiêu cực trong thiết kế. TNS bằng 0 cho thấy không có độ trễ tiêu cực tổng thể, điều này là một điểm tích cực vì nó có nghĩa là toàn bộ thiết kế không có điểm nào vượt quá các ràng buộc về thời gian.

2. Sử Dụng Tài Nguyên FPGA:

- Slice LUTs (LUTs Logic, LUTs as Memory, LUTs as Distributed RAM, LUTs as Shift Register): Used = 26872 và Available = 53200, tỷ lệ sử dụng 49.01% cho thấy rằng phần tài nguyên LUTs được sử dụng một cách hiệu quả, còn dư một lượng lớn tài nguyên cho các mô hình hoặc mở rộng trong tương lai.
- Slice Registers: Used = 43112 và Available = 106440, sử dụng khoảng 38.83%. Điều này cũng cho thấy tài nguyên Slice Registers không bị quá tải, vẫn còn dư khả năng mở rộng.
- F7 Muxes: Các bộ chọn (multiplexers) này dùng để lựa chọn giữa các tín hiệu khác nhau trong thiết kế. Used = 892 và Available = 26600. Việc chỉ sử dụng 3.35% của F7 Muxes cho thấy rằng chúng vẫn chưa được sử dụng được nhiều, và còn khả năng để mở rộng đáng kể.
- F8 Muxes: Các F8 Muxes ít quan trọng hơn trong việc lựa chọn tín hiệu, nên chỉ sử dụng 1.81%.

3. Block RAM (BRAM):

Used = 124 và Available = 140, sử dụng 88.57% cho thấy phần bộ nhớ BRAM đang sử dụng khá nhiều tài nguyên, nhưng vẫn còn một phần nhỏ để mở rộng.

4. DSPs (Digital Signal Processors):

Used = 24 và Available = 220, tỷ lệ sử dụng chỉ 10.91% cho thấy DSPs có thể chưa được tận dụng hết, cho phép mở rộng mô hình mà không gặp phải vấn đề về tài nguyên này.

4.5 Đánh giá kiến trúc HLS đã thiết kế

- Sự tối ưu hóa tài nguyên: Thiết kế đã sử dụng hợp lý các tài nguyên FPGA như LUTs, Slice Registers, và BRAMs, mà vẫn còn một phần lớn tài nguyên chưa được sử dụng hết. Điều này cho thấy rằng hệ thống có thể mở rộng hoặc cải tiến mà không gặp phải vấn đề về tài nguyên.
- Tốc độ và thời gian thực thi: Với WNS và TNS bằng 0, việc triển khai này đáp ứng hoàn hảo yêu cầu về thời gian, tức là tất cả các tín hiệu và phép toán được thực hiện đúng thời gian mà không gặp phải độ trễ tiêu cực. Điều này là rất quan trọng trong các ứng dụng thời gian thực như nhận diện biển báo giao thông.

5 Chạy kiểm nghiệm trên JupyterNotebook

Ta tiến hành kết nối kit và load các module, thư viện lên Jupyter.

Dữ liệu được sử dụng là tập dữ liệu biển báo giao thông của Đức. Mô hình này có thể phân loại 42 loại biển báo giao thông khác nhau.

5.1 Khởi tạo Classifier

```
In [1]: import bnn
print(bnn.available_params(bnn.NETWORK_CNVW1A1))

classifier = bnn.CnvClassifier(bnn.NETWORK_CNVW1A1, 'road-signs', bnn.RUNTIME_HW)
```

Hình 38: Khởi tạo Classifier

- Thư viện bnn được nhập vào, sau đó gọi hàm bnn.available_params() để hiển thị các tham số có sẵn cho mạng BNN.
- Mạng nơ-ron được khởi tạo thông qua bnn.CnvClassifier với mô hình NETWORK_CNVW1A1, tập dữ liệu "road-signs", và môi trường thực thi là phần cứng (Pynq FPGA).

5.2 Danh sách các lớp phân loại

Sau khi khởi tạo classifier, lệnh này sẽ in ra danh sách các lớp biển báo giao thông mà mô hình có thể phân loại. Có tổng cộng 42 lớp biển báo, ví dụ như "20 Km/h", "Stop", "Pedestrians in road ahead", và nhiều lớp khác.

```
In [2]: print(classifier.classes)

['20 Km/h', '30 Km/h', '50 Km/h', '60 Km/h', '70 Km/h', '80 Km/h', 'End 80 Km/h', '100 Km/h', '120 Km/h', 'No overtaking', 'No overtaking for large trucks', 'Priority crossroad', 'Priority road', 'Give way', 'Stop', 'No vehicles', 'Prohibited for vehicle s with a permitted gross weight over 3.5t including their trailers, and for tractors except passenger cars and buses', 'No entr y for vehicular traffic', 'Danger Ahead', 'Bend to left', 'Bend to right', 'Double bend (first to left)', 'Uneven road', 'Road slippery when wet or dirty', 'Road narrows (right)', 'Road works', 'Traffic signals', 'Pedestrians in road ahead', 'Children cr ossing ahead', 'Bicycles prohibited', 'Risk of snow or ice', 'Wild animals', 'End of all speed and overtaking restrictions', 'T urn right ahead', 'Turn left ahead', 'Ahead only', 'Ahead or right only', 'Ahead or left only', 'Pass by on right', 'Pass by on left', 'Roundabout', 'End of no-overtaking zone', 'End of no-overtaking zone for vehicles with a permitted gross weight over 3. 5t including their trailers, and for tractors except passenger cars and buses', 'Not a roadsign']
```

Hình 39: Danh sách các lớp phân loại

5.3 Mở và hiển thị hình ảnh để phân loại

```
In [3]: from PIL import Image
import numpy as np
from os import listdir
from os.path import isfile, join
from IPython.display import display

imgList = [f for f in listdir("/home/xilinx/jupyter_notebooks/bnn/pictures/road_signs/") if isfile(join("/home/xilinx/jupyter_no
images = []

for imgFile in imgList:
    img = Image.open("/home/xilinx/jupyter_notebooks/bnn/pictures/road_signs/" + imgFile)
    images.append(img)
    img.thumbnail((64, 64), Image.ANTIALIAS)
    display(img)
```

Hình 40: Mở và hiển thị hình ảnh để phân loại

- Mã này tìm và mở các hình ảnh từ thư mục chứa biển báo giao thông, rồi hiển thị chúng.
- imgList: Tạo danh sách các tệp hình ảnh trong thư mục road_signs.
- images.append(img): Tải từng hình ảnh vào danh sách images sau khi mở chúng.
- img.thumbnail((64, 64), Image.ANTIALIAS): Thay đổi kích thước hình ảnh xuống 64x64 pixels, giúp giảm khối lượng tính toán khi phân loại.
- display(img): Hiển thị hình ảnh trong notebook.

5.4 Khởi chạy BNN trên phần cứng

- classifier.classify_images(images): Phân loại các hình ảnh đã tải vào danh sách images. Kết quả trả về là các chỉ số lớp mà mỗi hình ảnh được phân loại vào.
- classifier.class_name(index): Chuyển đổi chỉ số lớp thành tên lớp biển báo giao thông tương ứng.

```
In [4]: results = classifier.classify_images(images)
print("Identified classes: {}".format(results))
for index in results:
    print("Identified class name: {}".format((classifier.class_name(index))))
```

Packing and interleaving CIFAR-10 inputs...
Running prebuilt CIFAR-10 test for 13 images...
Inference took 6148 microseconds, 472.923 usec per image
Classification rate: 2114.51 images per second
Inference took 6148.00 microseconds, 472.92 usec per image
Classification rate: 2114.51 images per second
Identified classes: [5 13 8 8 41 14 30 5 38 6 13 27 33]
Identified class name: 80 Km/h
Identified class name: Give way
Identified class name: 120 Km/h
Identified class name: 120 Km/h
Identified class name: End of no-overtaking zone
Identified class name: Stop
Identified class name: Risk of snow or ice
Identified class name: 80 Km/h
Identified class name: Pass by on right
Identified class name: End 80 Km/h
Identified class name: Give way
Identified class name: Pedestrians in road ahead
Identified class name: Turn right ahead

Hình 41: Kết quả chạy trên phần cứng

Kết quả:

- Thời gian suy luận: 6148 microseconds (473 microseconds mỗi ảnh).
- Tỷ lệ phân loại: 2114 ảnh mỗi giây, cho thấy tốc độ xử lý rất cao trên phần cứng.
- Trạng thái dự đoán: Đúng với toàn bộ biển báo đưa vào.

5.5 Khởi chạy BNN trên phần mềm

- bnn.CnvClassifier(...): Tạo một đối tượng classifier nhưng lần này sử dụng RUNTIME_SW để chạy trên phần mềm thay vì phần cứng.
- sw_class.classify_images(images): Phân loại các hình ảnh giống như ở trên, nhưng lần này mô hình chạy trên phần mềm ARM thay vì FPGA.

```
In [5]: sw_class = bnn.CnvClassifier(bnn.NETWORK_CNVW1A1,"road-signs", bnn.RUNTIME_SW)

results = sw_class.classify_images(images)
print("Identified classes: {0}".format(results))
for index in results:
    print("Identified class name: {0}".format((classifier.class_name(index))))
```

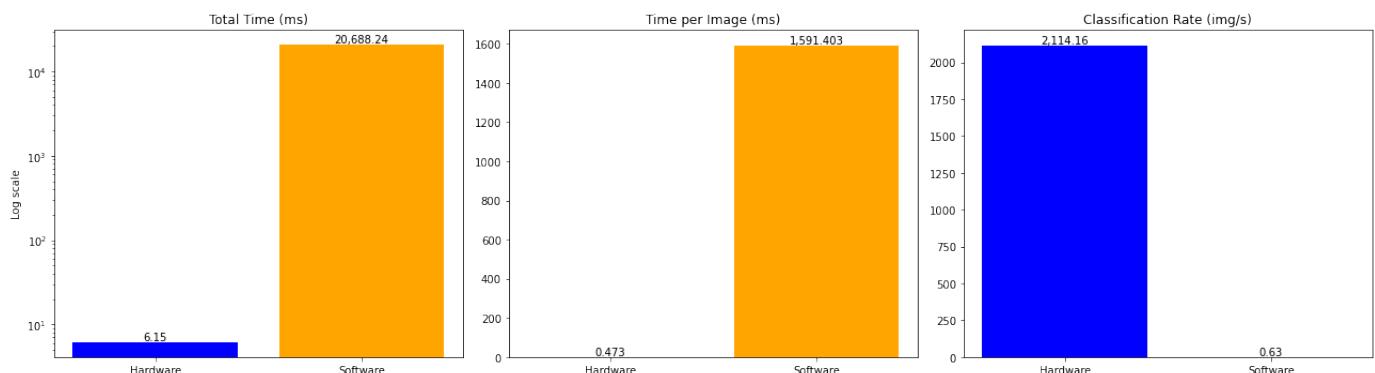
Setting network weights and thresholds in accelerator...
Packing and interleaving CIFAR-10 inputs...
Running prebuilt CIFAR-10 test for 13 images...
Inference took 20874554 microseconds, 1.60573e+06 usec per image
Classification rate: 0.622768 images per second
Inference took 20874553.38 microseconds, 1605734.88 usec per image
Classification rate: 0.62 images per second
Identified classes: [5 13 8 8 41 14 30 5 38 6 13 27 33]
Identified class name: 80 Km/h
Identified class name: Give way
Identified class name: 120 Km/h
Identified class name: 120 Km/h
Identified class name: End of no-overtaking zone
Identified class name: Stop
Identified class name: Risk of snow or ice
Identified class name: 80 Km/h
Identified class name: Pass by on right
Identified class name: End 80 Km/h
Identified class name: Give way
Identified class name: Pedestrians in road ahead
Identified class name: Turn right ahead

Hình 42: Kết quả chạy trên phần mềm

Kết quả:

- Thời gian suy luận: 28874554 microseconds, lâu hơn nhiều so với khi chạy trên phần cứng.
- Tỷ lệ phân loại: 0.6 ảnh mỗi giây, chậm hơn nhiều so với khi chạy trên FPGA.
- Trạng thái dự đoán: Đúng với toàn bộ biến báo đưa vào.

Biểu đồ so sánh HW và SW



Hình 43: Biểu đồ so sánh HW và SW

Tính toán thời gian tăng tốc phần cứng

```
In [9]: speedup_total = sw_time_total / hw_time_total
speedup_per_image = sw_time_per_image / hw_time_per_image
speedup_rate = hw_rate / sw_rate

print(f"Speedup Factors:")
print(f"- Total time: {speedup_total:.1f}x faster in hardware")
print(f"- Time per image: {speedup_per_image:.1f}x faster in hardware")
print(f"- Classification rate: {speedup_rate:.1f}x higher in hardware")

Speedup Factors:
- Total time: 3364.5x faster in hardware
- Time per image: 3364.5x faster in hardware
- Classification rate: 3355.8x higher in hardware
```

Hình 44: Kết quả tính toán thời gian tăng tốc phần cứng

Dựa trên kết quả thực nghiệm ở trên (Hình 43, 44), có thể rút ra các nhận xét về hiệu năng giữa mô hình chạy trên phần cứng (FPGA PYNQ-Z2) và phần mềm (CPU ARM) như sau:

1. Tổng thời gian xử lý
 - Phần cứng (Hardware): Tổng thời gian xử lý cho toàn bộ tập dữ liệu chỉ 6,15 ms.
 - Phần mềm (Software): Tổng thời gian xử lý lên tới 20.688,24 ms.
 - Nhận xét: Chạy trên FPGA nhanh hơn chạy trên CPU ARM tới 3364,5 lần về tổng thời gian xử lý.
2. Thời gian xử lý trên mỗi ảnh
 - Phần cứng: Trung bình chỉ mất 0,473 ms/ảnh.
 - Phần mềm: Trung bình mất 1.591,403 ms/ảnh.
 - Nhận xét: Thời gian xử lý mỗi ảnh trên FPGA nhanh hơn 3364,5 lần so với CPU ARM. Điều này cho phép hệ thống đáp ứng tốt các yêu cầu nhận diện thời gian thực.
3. Tốc độ phân loại (Classification Rate)
 - Phần cứng: Đạt tốc độ 2114,16 ảnh/giây.
 - Phần mềm: Chỉ đạt 0,63 ảnh/giây.
 - Nhận xét: Tốc độ phân loại trên FPGA cao hơn 3355,8 lần so với CPU ARM, vượt trội cho các ứng dụng cần tốc độ cao và xử lý song song.
4. Kết luận chung:
 - Việc triển khai mô hình BNN trên FPGA PYNQ-Z2 mang lại hiệu suất vượt trội so với chạy trên CPU ARM, cả về thời gian xử lý tổng, thời gian xử lý từng ảnh và tốc độ phân loại.
 - Kết quả này minh chứng cho lợi thế lớn của phần cứng chuyên dụng (FPGA) khi xử lý các tác vụ học sâu dạng nhị phân, đặc biệt trong các ứng dụng nhúng, thời gian thực hoặc yêu cầu tiết kiệm năng lượng.
 - Tuy nhiên, việc phát triển và tổng hợp mô hình cho phần cứng sẽ phức tạp và tốn thời gian hơn so với phát triển phần mềm, nhưng hoàn toàn xứng đáng khi cần tối ưu hiệu năng hệ thống.

5.6 Phát hiện đối tượng trong cảnh

5.6.1 Đọc và hiển thị hình ảnh

```
In [6]: from PIL import Image
image_file = "/home/xilinx/jupyter_notebooks/bnn/pictures/street_with_stop.JPG"
im = Image.open(image_file)
im
```

Out[6]:



Hình 45: Đọc và hiển thị biển báo "STOP"

Hình ảnh này là một cảnh giao thông với biển báo "STOP".

5.6.2 Các bước thực hiện nhận diện trong cảnh

1. Chia hình ảnh thành các tile (Mảnh nhỏ):

Phần này chia ảnh thành các mảnh nhỏ (tiles), mỗi mảnh ảnh sẽ được phân loại riêng biệt để tìm kiếm các vật thể, ví dụ như biển báo giao thông "STOP". Các bước cụ thể bao gồm:

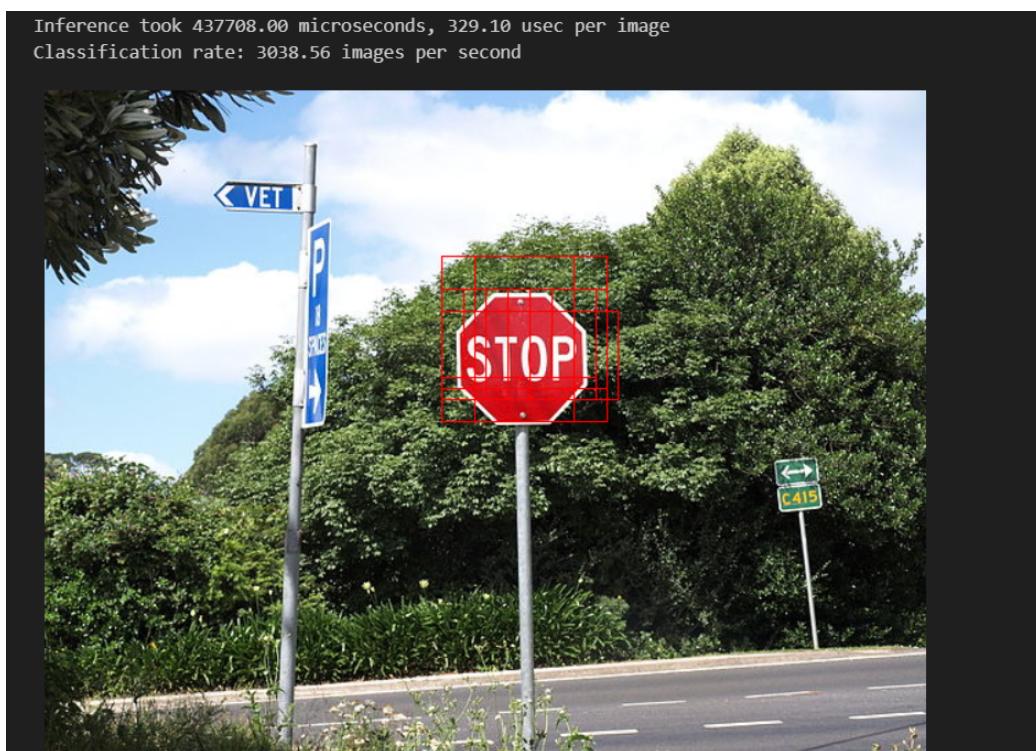
- Chia ảnh thành các tile: Đầu tiên, kích thước stride được xác định, rồi sau đó tính toán số lượng tile theo chiều rộng và chiều cao của ảnh.
- Dánh giá kết quả: Việc chia ảnh thành các mảnh nhỏ (tile) giúp tăng độ chính xác của việc nhận diện vật thể, vì mỗi tile có thể chứa một phần nhỏ của biển báo, giúp hệ thống dễ dàng nhận diện chính xác hơn. Tuy nhiên, số lượng tile có thể ảnh hưởng đến thời gian xử lý và cần tối ưu hóa.

2. Phân loại và vẽ hộp bao quanh biển báo:

Sau khi chia ảnh thành các tile, việc phân loại được thực hiện và các tile có biển báo "STOP" sẽ được vẽ hộp bao quanh.

```
results = classifier.classify_images(images)
stop = results == 14
```

Ở đây, mỗi tile được phân loại, và nếu kết quả phân loại là biển báo "STOP" (lớp 14), một hình chữ nhật được vẽ bao quanh vị trí của biển báo trên ảnh.



Hình 46: Phát hiện đối tượng trong cảnh sau

Hình ảnh sau khi được phân loại và vẽ hộp cho thấy rằng biển báo "STOP" đã được nhận diện chính xác, với các chi tiết rõ ràng.

Kết quả cho thấy thời gian xử lý 437708 microseconds (hoặc 329.1 microseconds per image), với tốc độ phân loại đạt 3038,56 ảnh mỗi giây.

Dánh giá hiệu suất:

- Tốc độ phân loại rất cao: 3038,56 ảnh mỗi giây là một tốc độ xử lý khá tốt, cho thấy rằng hệ thống hoạt động rất hiệu quả trong việc nhận diện biển báo giao thông.
- Thời gian suy luận là khá nhanh, mặc dù có thể cải thiện thêm nếu giảm thiểu số lượng tile hoặc tăng độ phân giải để đạt được độ chính xác cao hơn.

Tương tự ta có kết quả khi phát hiện biển báo cấm



Hình 47: Phát hiện biển báo cấm trong ảnh

5.7 Nhận xét kết quả chạy trên Jupyter

Phần cứng FPGA cho thấy hiệu suất vượt trội về tốc độ suy luận, đạt 2114 ảnh mỗi giây với thời gian suy luận chỉ 473 microseconds mỗi ảnh. Điều này làm nổi bật khả năng của FPGA trong việc xử lý các mô hình học sâu phức tạp một cách nhanh chóng.

Phần mềm (ARM), mặc dù cung cấp kết quả chính xác, nhưng tốc độ xử lý thấp hơn nhiều, chỉ 0.62 ảnh mỗi giây với thời gian suy luận 1605734 microseconds mỗi ảnh. Điều này cho thấy phần mềm không thể đạt được hiệu suất cao như phần cứng FPGA khi triển khai các mô hình học sâu.

FPGA mang lại sự cải thiện rõ rệt về tốc độ suy luận và khả năng xử lý hình ảnh trong các ứng dụng yêu cầu tính toán thời gian thực như nhận diện biển báo giao thông.

Hệ thống đã thực hiện thành công Post-processing là một bước quan trọng trong việc tối ưu hóa kết quả phân loại, giúp nâng cao độ chính xác và hiệu quả của hệ thống.

Việc triển khai mô hình trên ARM vẫn có thể hữu ích trong các trường hợp không đòi hỏi tốc độ xử lý cao hoặc khi phần cứng FPGA không khả dụng.

6 Kết luận và hướng phát triển

6.1 Kết luận

Đề tài "Nhận diện biển báo giao thông trên FPGA PYNQ-Z2" đã thành công trong việc phát triển một hệ thống nhận diện biển báo giao thông sử dụng nền tảng FPGA PYNQ-Z2, kết hợp với các phương pháp học sâu như Mạng Nơ-ron Tích chập (CNN) và Mạng Nơ-ron Nhị phân (BNN). Hệ thống này giúp giảm độ trễ và tiết kiệm năng lượng khi xử lý và phân loại các loại biển báo giao thông trong thời gian thực. Việc áp dụng FPGA vào hệ thống không chỉ nâng cao hiệu suất mà còn tối ưu hóa tài nguyên phần cứng, đáp ứng yêu cầu khắt khe về tốc độ và độ chính xác trong các ứng dụng giao thông thông minh.

Tuy nhiên, nghiên cứu này vẫn có một số hạn chế cần được khắc phục. Đầu tiên, bộ dữ liệu GTSRB (German Traffic Sign Recognition Benchmark) được sử dụng cho việc huấn luyện và thử nghiệm hệ thống khá hạn chế, chủ yếu chỉ bao gồm các biển báo giao thông phổ biến trong môi trường lý tưởng. Điều này gây khó khăn trong việc kiểm tra tính hiệu quả của hệ thống khi triển khai trong các tình huống giao thông thực tế, nơi có thể gặp phải các yếu tố như ánh sáng yếu, thời tiết xấu hoặc biển báo bị mờ.

Hệ thống hiện tại cũng chưa được thử nghiệm trong các điều kiện giao thông phức tạp hoặc với tốc độ cao, điều này có thể ảnh hưởng đến khả năng nhận diện chính xác khi đối mặt với nhiều biển báo và yếu tố nhiễu. Do đó, việc mở rộng bộ dữ liệu để bao gồm nhiều loại biển báo giao thông trong các điều kiện thực tế là một yêu cầu quan trọng. Ngoài ra, việc tối ưu hóa thêm các thuật toán học sâu, đặc biệt là các mô hình nhị phân (BNN), có thể giúp hệ thống hoạt động hiệu quả hơn, giảm thiểu yêu cầu về bộ nhớ và tính toán.

Để phát triển hệ thống trong tương lai, cần tập trung vào việc cải thiện độ chính xác và tính linh hoạt của mô hình học sâu trong các điều kiện thực tế. Tăng cường thêm khả năng nhận diện thực tế (Real-time) để có thể nhận diện các biển báo ngoài thực tế với thời gian thực. Nếu làm được điều này sẽ mang lại rất nhiều ứng dụng đặc biệt là trong các hệ thống thông minh hỗ trợ tự động của các phương tiện giao thông như là ADAS.

6.2 Phát triển

Do còn nhiều hạn chế về kiến thức nên đề án vẫn còn nhiều hướng có thể phát triển, tối ưu thêm như:

1. Mở rộng bộ dữ liệu huấn luyện

Để tăng độ chính xác và khả năng nhận diện trong các điều kiện thực tế, cần bổ sung bộ dữ liệu huấn luyện với các tình huống như thời tiết xấu, ánh sáng yếu hoặc biển báo bị mờ. Điều này giúp hệ thống hoạt động hiệu quả hơn trong môi trường giao thông phức tạp.

2. Tối ưu hóa mô hình học sâu

Cải thiện các mô hình Mạng Nơ-ron Nhị phân (BNN) để giảm thiểu sự giảm độ chính xác do nhị phân hóa trọng số và kích hoạt. Tập trung vào việc tối ưu hóa các phương pháp nhị phân hóa giúp tiết kiệm bộ nhớ và tính toán mà không làm giảm hiệu suất.

3. Mở rộng ứng dụng trong giao thông thông minh và xe tự lái

Hệ thống có thể được tích hợp vào các xe tự lái và các nền tảng giao thông thông minh, giúp nhận diện biển báo và tối ưu hóa việc điều khiển giao thông trong các thành phố thông minh, giảm thiểu ùn tắc và tai nạn.

4. Cải thiện giao diện và khả năng tương tác

Tích hợp tính năng cảnh báo trực quan cho người lái xe khi phát hiện biển báo, và kết nối hệ thống với các hệ thống điều khiển giao thông để tối ưu hóa dòng chảy giao thông và giảm tắc nghẽn.

5. Tăng cường hiệu quả năng lượng và tài nguyên phần cứng

Nghiên cứu các kỹ thuật tiết kiệm năng lượng và tối ưu hóa tài nguyên phần cứng, giúp hệ thống hoạt động hiệu quả hơn trên các nền tảng có tài nguyên hạn chế, mở rộng khả năng ứng dụng trong môi trường thực tế.

6. Ứng dụng thực tế

Hệ thống nhận diện biển báo giao thông trên FPGA PYNQ-Z2 có thể được ứng dụng rộng rãi trong các hệ thống giao thông thông minh, đặc biệt là trong xe tự lái và các thành phố thông minh. Việc tích hợp hệ thống này vào xe tự lái giúp xe nhận diện và tuân thủ các biển báo giao thông một cách chính xác, giảm thiểu nguy cơ tai nạn và nâng cao sự an toàn cho người tham gia giao thông. Ngoài ra, hệ thống cũng có thể được ứng dụng trong các thành phố thông minh để giám sát và điều phối giao thông, giúp tối ưu hóa lưu lượng xe cộ, giảm ùn tắc và cải thiện hiệu quả vận hành của các cơ sở hạ tầng giao thông.

Tài liệu

- [1] A Review of Binarized Neural Networks, Taylor Simons, 2019. *Electronics*, vol. 8, no. 6, p. 661. *Truy cập online:* <https://www.mdpi.com/2079-9292/8/6/661>
- [2] Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs, 2017. *Truy cập online:* <https://www.cs.cornell.edu/~zhiruz/pdfs/bnn-fpga2017.pdf>
- [3] Accelerating Image Processing with PYNQ & OpenMV Cam, Adam Taylor, 2019. *Truy cập online:* <https://www.hackster.io/adam-taylor/accelerating-image-processing-with-pynq-openmv-cam-50ba7a>
- [4] BNN-PYNQ: Baking a custom BNN for the Zybo-Z7, Franco Caspe, 2020. *Truy cập online:* <https://www.hackster.io/franco-caspe/bnn-pynq-baking-a-custom-bnn-for-the-zybo-z7-f0bbe3#toc-section-3--hls--amp--vivado-synthesis-4>
- [5] Creating a simple Overlay for PYNQ-Z1 board from Vivado HLx, 2017. *Truy cập online:* <https://yangtavaresblog.wordpress.com/2017/07/31/creating-a-simple-overlay-for-pynq-z1-board-from-vivado-hlx/>
- [6] Documentation Navigator, 2024. *Truy cập online:* <https://www.xilinx.com/support/documentation-navigation/overview.html>
- [7] FINN, 2020. *Truy cập online:* <https://finn.readthedocs.io/en/latest/>
- [8] FINN Fast, Scalable Quantized Neural Network Inference on FPGAs, 2020. *Truy cập online:* <https://github.com/Xilinx/finn>
- [9] FPGA-Based Acceleration on Additive Manufacturing Defects Inspection, Yawen Luo, Yuhua Chen, 2021. *Truy cập online:* https://www.researchgate.net/publication/350183602_FPGA-Based_Acceleration_on_Additive_Manufacturing_Defects_Inspection
- [10] Giới thiệu về FPGA và Ngôn ngữ mô tả phần cứng, nguyễn-thanh, 2014. *Truy cập online:* <https://vimach.net/threads/gioi-thieu-ve-fpga-va-ngon-ngu-mo-ta-phan-cung.27/>
- [11] GTSRB - German Traffic Sign Recognition Benchmark, 2018. *Truy cập online:* <https://www.kaggle.com/datasets/meowmeowmeowmeow/gtsrb-german-traffic-sign>

- [12] Guarding Machine Learning Hardware Against Physical Side-Channel Attacks, Anuj Dubey, Rosario Cammarota, 2021. *Truy cập online:* https://www.researchgate.net/publication/354310846_Guarding_Machine_Learning_Hardware_Against_Physical_Side-Channel_Attacks
- [13] Paper Explanation: Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1, Natsu, 2018. *Truy cập online:* <https://mohitjain.me/2018/07/14/bnn/>
- [14] PYNQ, 2017. *Truy cập online:* <https://github.com/Xilinx/PYNQ>
- [15] PYNQ: Python productivity for Adaptive Computing platforms, 2022. *Truy cập online:* <https://pynq.readthedocs.io/>
- [16] PYNQ-Z2 Reference Manual v1.1, 2019. TuL PYNQ-Z2.
- [17] Thuật toán CNN là gì? Hướng dẫn chọn tham số cho CNN. *Truy cập online:* <https://evbn.org/thuat-toan-cnn-la-gi-huong-dan-chon-tham-so-cho-cnn-1678015007/>
- [18] Tutorial: Creating a new Verilog Module Overlay, RogerPease, 2020. *Truy cập online:* <https://discuss.pynq.io/t/tutorial-creating-a-new-verilog-module-overlay/1530>
- [19] Vivado Design Suite User Guide, AMD XILINX, 2022.
- [20] Vitis Accelerated Libraries, 2021. *Truy cập online:* https://github.com/Xilinx/Vitis_Libraries

Phụ lục

Source của đồ án



Hình 48: QR Source code của đồ án

Các thư viện cần chuẩn bị

1. Thư viện Python chính:

- numpy
- matplotlib
- pillow (PIL)
- opencv-python
- jupyter
- ipywidgets

2. PYNQ Framework

3. Deep Learning Libraries

Các phần cứng cần chuẩn bị

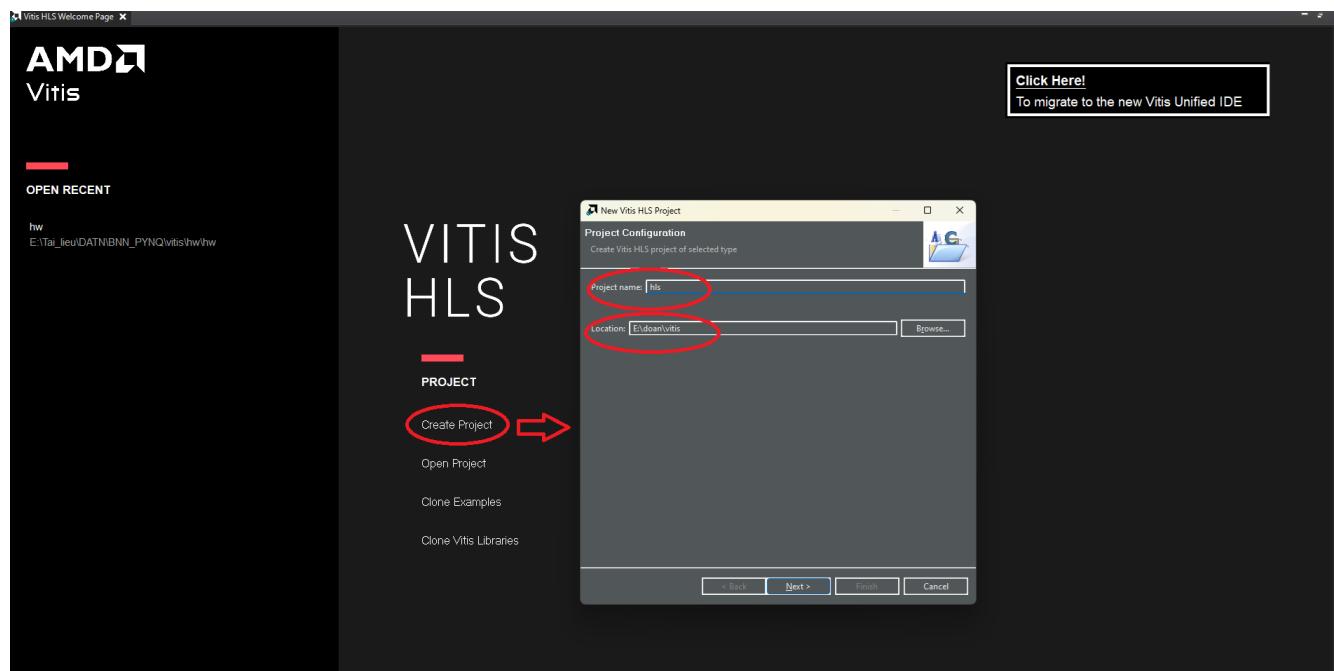
1. PYNQ-Z1 hoặc PYNQ-Z2 board
2. SD card với PYNQ image
3. Cáp kết nối mạng và cáp Micro USB

Lưu ý quan trọng

1. PYNQ Image: Cần flash PYNQ OS image lên SD card
2. Bitstream files: Project cần các file .bit và .hwh cho FPGA
3. Network: Cần kết nối mạng để access Jupyter server thông qua web interface

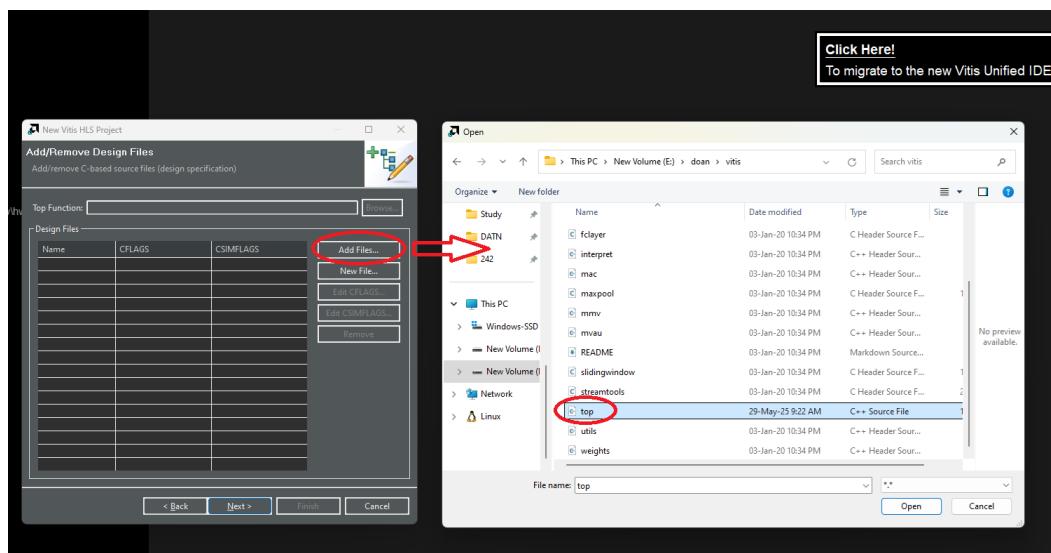
Hướng dẫn cách dùng Vitis HLS

Mở và tạo Project mới trên Vitis HLS (Lưu ý tên và các đường dẫn đến Project đều phải viết liền không dấu để tránh xung đột của phần mềm)



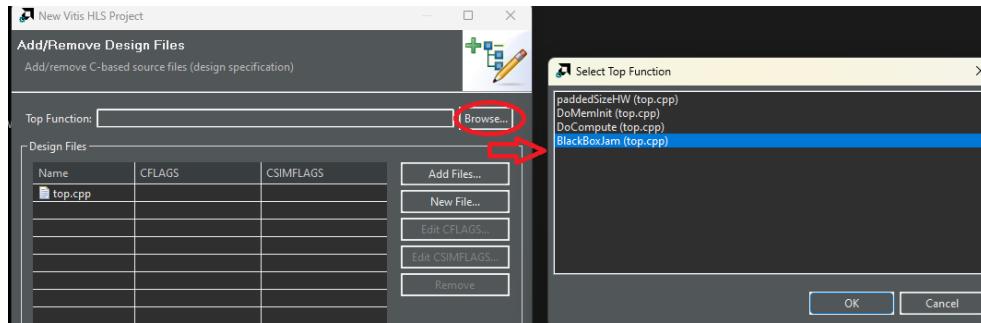
Hình 49: Giao diện mở và tạo Project trên Vitis

Thêm file cấu hình Hardware C++



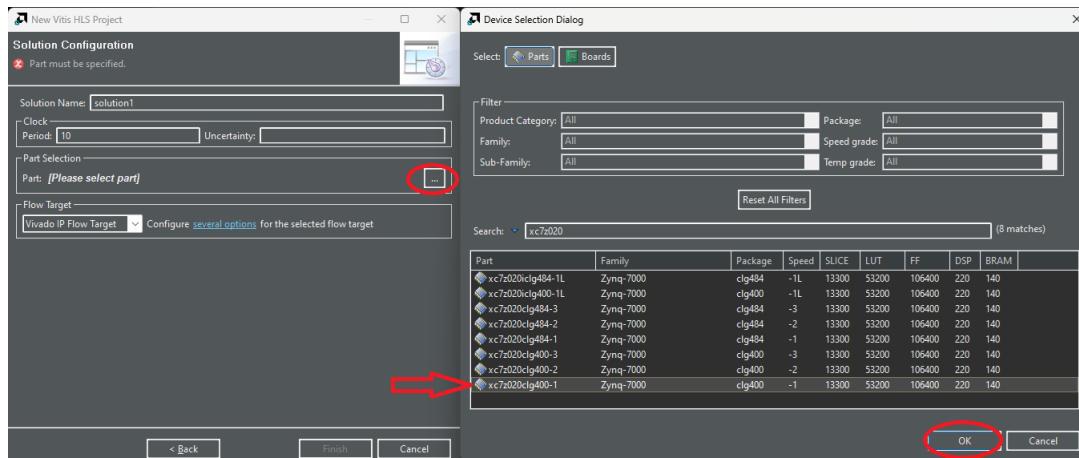
Hình 50: Giao diện thêm file cấu hình Hardware C++

Vào Browse để chọn Top Function cho mô hình.



Hình 51: Giao diện chọn Top Function cho mô hình

Chọn Part của Kit, ở đây sử dụng kit PYNQ-Z2 nên chọn Part là xc7z020clg400-1, nếu xài kit khác có thể check datasheet của kit và chọn Part phù hợp.

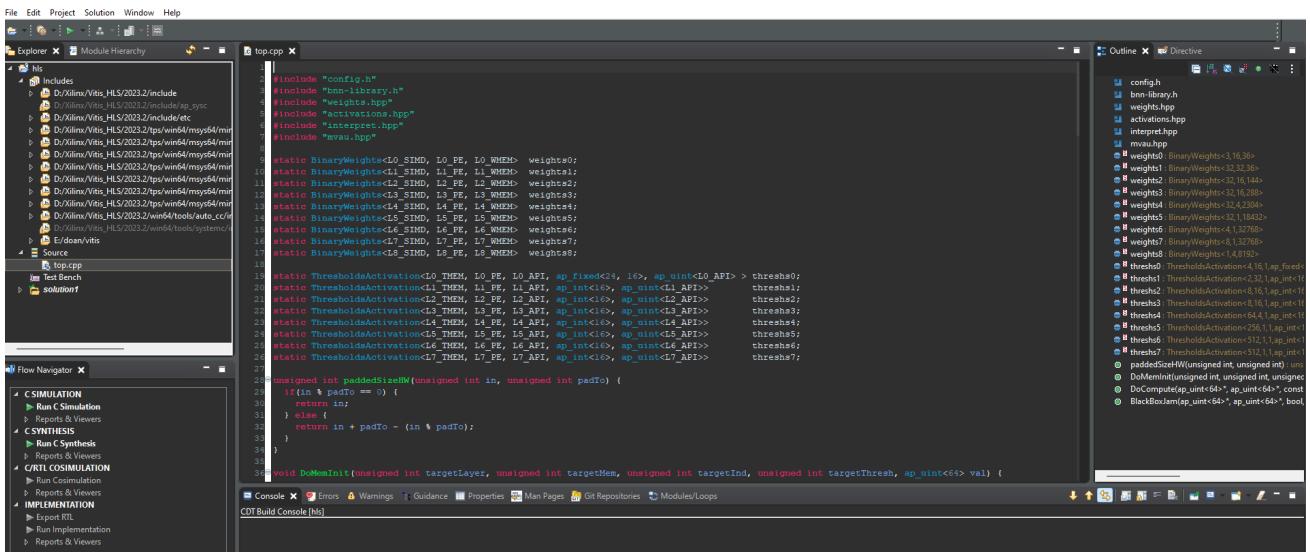


Hình 52: Giao diện chọn Part của Kit

Sau đó chọn Finish, giao diện Vitis HLS sẽ được hiện ra, ta thêm các file Header vào chung folder với file C++ tránh việc chương trình không tìm được các file Header này.

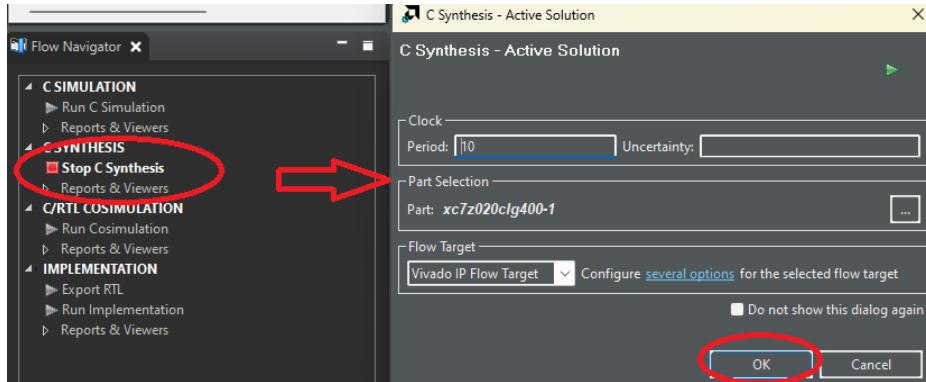
hw	28-Apr-25 11:30 AM	File folder	
activations	03-Jan-20 10:34 PM	C++ Header Sour...	8 KB
bnn-library	03-Jan-20 10:34 PM	C Header Source F...	3 KB
config	04-Jan-22 5:26 PM	C Header Source F...	4 KB
CONTRIBUTING	03-Jan-20 10:34 PM	Markdown Source...	1 KB
convlayer	03-Jan-20 10:34 PM	C Header Source F...	10 KB
dma	03-Jan-20 10:34 PM	C Header Source F...	8 KB
fclayer	03-Jan-20 10:34 PM	C Header Source F...	6 KB
interpret	03-Jan-20 10:34 PM	C++ Header Sour...	9 KB
Jenkinsfile	03-Jan-20 10:34 PM	File	5 KB
LICENSE	03-Jan-20 10:34 PM	File	2 KB
mac	03-Jan-20 10:34 PM	C++ Header Sour...	9 KB
maxpool	03-Jan-20 10:34 PM	C Header Source F...	15 KB
mmv	03-Jan-20 10:34 PM	C++ Header Sour...	3 KB
mrvu	03-Jan-20 10:34 PM	C++ Header Sour...	7 KB
slidingwindow	03-Jan-20 10:34 PM	C Header Source F...	14 KB
streamtools	03-Jan-20 10:34 PM	C Header Source F...	28 KB
top	28-Apr-25 11:45 AM	C++ Source File	11 KB
utils	03-Jan-20 10:34 PM	C++ Header Sour...	5 KB
vitis_hls	28-Apr-25 11:31 AM	Text Document	4 KB

Hình 53: Thêm các file header



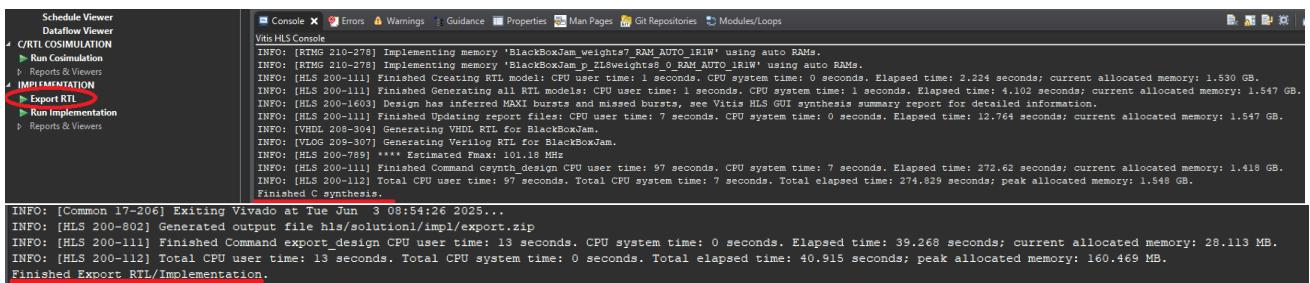
Hình 54: Giao diện chính Vitis HLS

Ta tiến hành bước Synthesis để tổng hợp code C++ thành ngôn ngữ phần cứng HDL. Sau chi



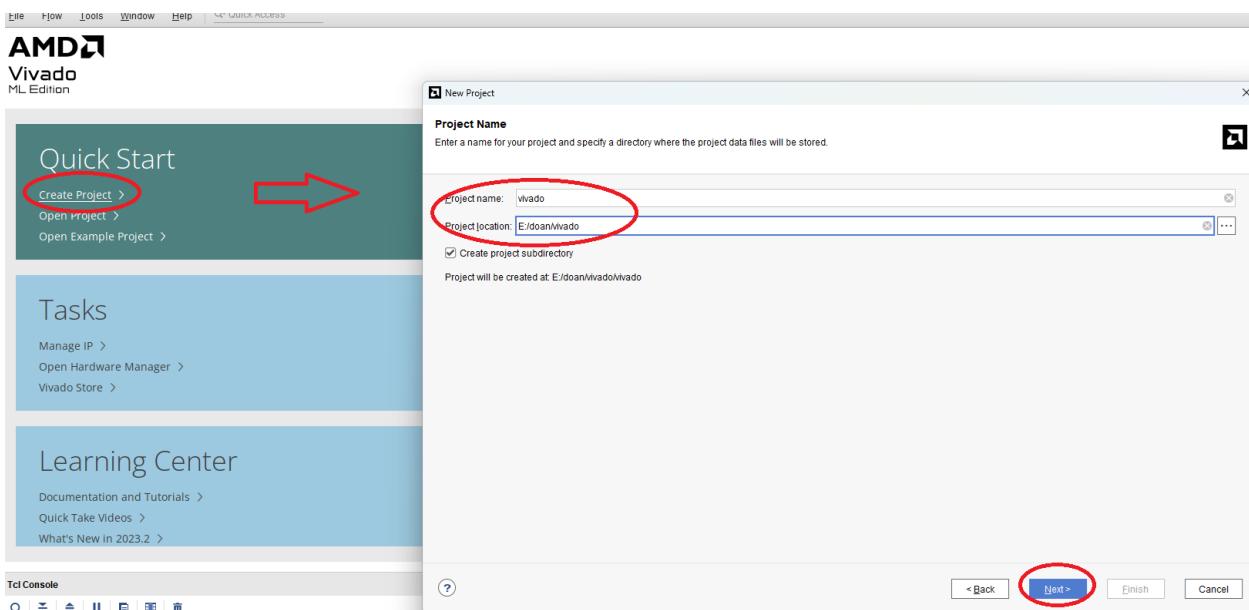
Hình 55: Synthesis code C++

Synthesis code C++ xong, ta chọn Export RTL, để nó xuất kết quả vừa Synthesis được thành 1 IP core dùng để thiết kế Block Design ở Vivado.



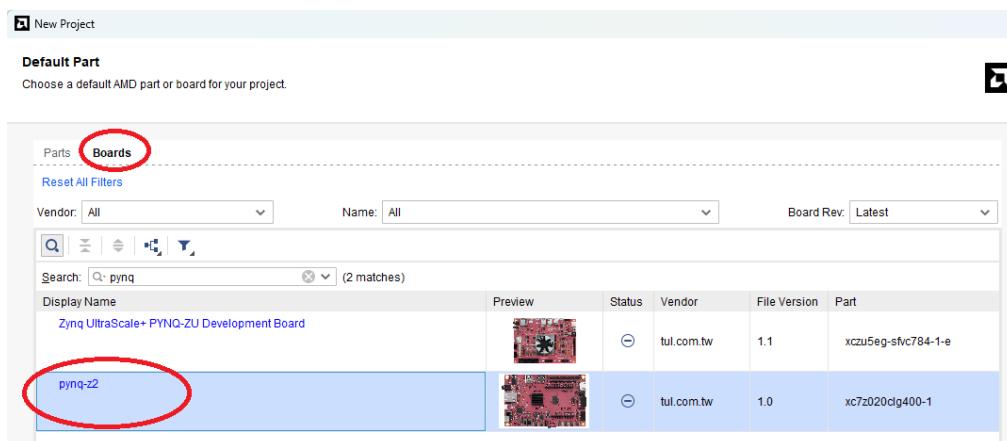
Hình 56: Xuất RTL

Hướng dẫn cách dùng Vivado



Hình 57: Mở và tạo Project Vivado

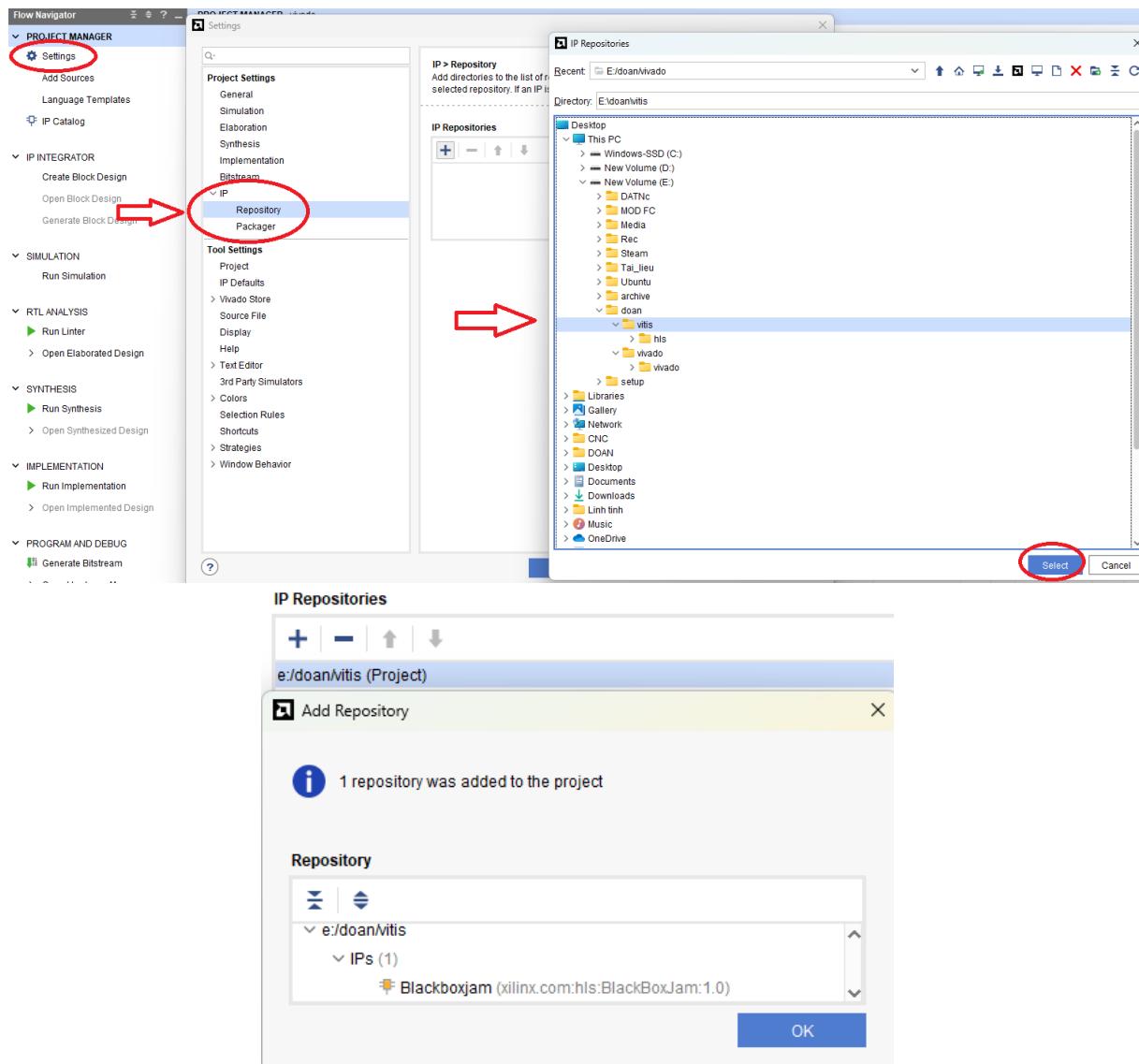
Chọn mục Board -> Chọn kit PYNQ-Z2 (Nếu không có sẵn có thể chọn Part tương tự bên Vitis lại hoặc chọn Download từ trang web của nhà sản xuất).



Hình 58: Chọn board trên Vivado

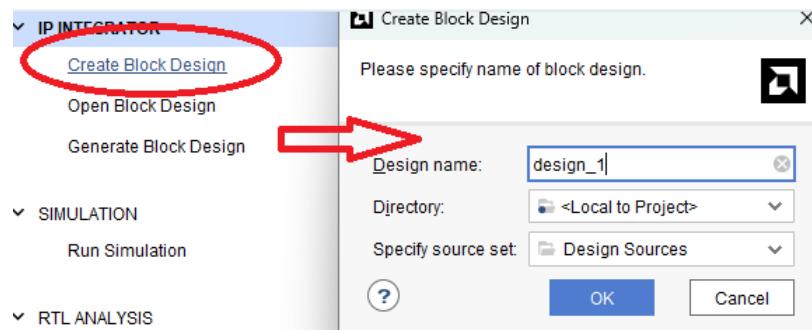
Sau đó giao diện chính của Vivado sẽ hiện lên.

Tiến hành các bước sau để gọi IP vừa thiết kế bên Vitis ra.



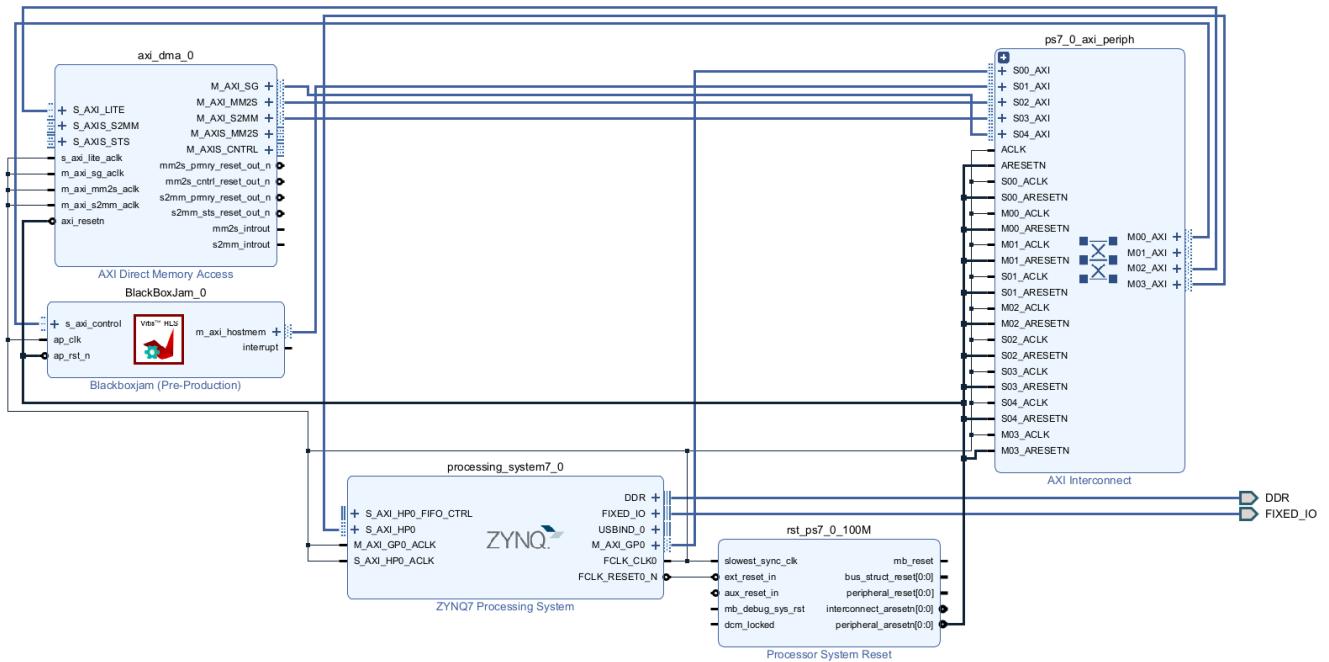
Hình 59: Các bước gọi IP vừa thiết kế

Tạo Block Design mới



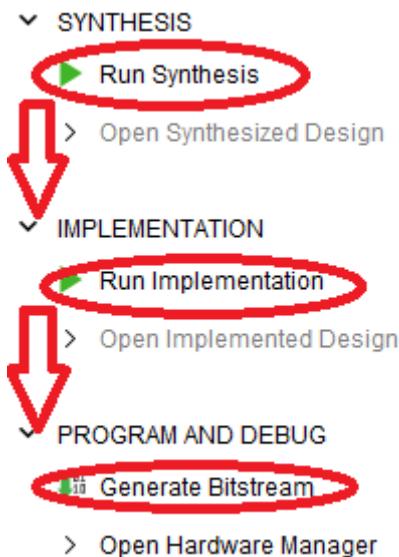
Hình 60: Tạo Block Design mới

Tiến hành gọi các khối Processing System, Direct Memory Access và khối IP vừa thiết kế ra, và dùng tool tự động kết nối nó lại, thu được khối Block Design hoàn chỉnh.



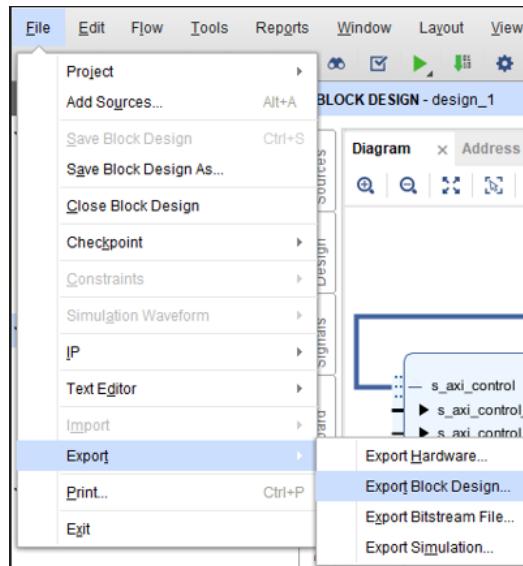
Hình 61: Block Design hoàn chỉnh

Thực hiện các bước Synthesis -> Implementation -> Generate Bitstream



Hình 62: Các bước xuất ra Bitfile

Xuất block design



Hình 63: Xuất block design

Để nạp cho thư viện kit và chạy trên Overlays ta cần chuẩn bị 3 file sau:

- File .tcl đã xuất ở bước 4
- File bitstream (.bit) theo đường dẫn "Thư mục lưu project/tên project.runs/impl_1"
- File .hwh theo đường dẫn:
"Thư mục lưu project/tên project.gen/sources_1/bd/design_1/hw_handoff"

Hướng dẫn cách dùng Jupyter Notebook trên kit

Cách truy cập vào vùng lưu trữ của thẻ SD trên kit

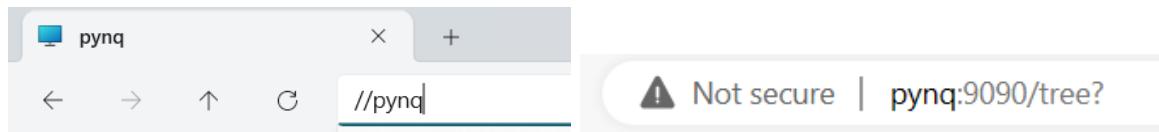


Hình 64: Truy cập vào thẻ SD của PYNQ-Z2

Kết nối kit vào PC và truy cập Jupyter Notebook bằng một trong 2 cách:

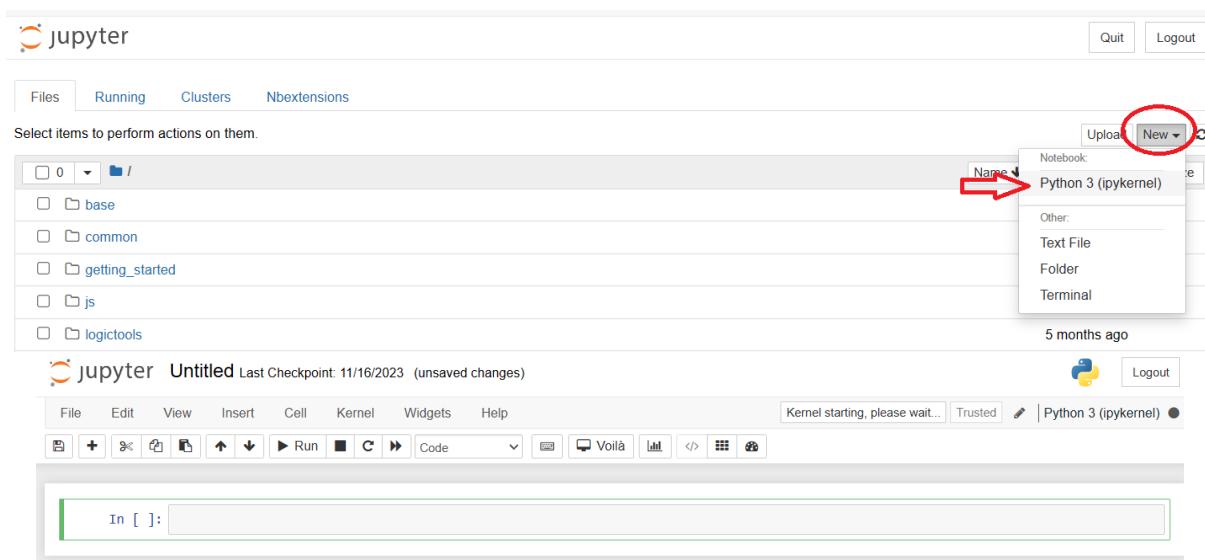
- Thông qua File Explorer bằng cách gõ lệnh "//pynq" vào thanh địa chỉ.
- Thông qua trình duyệt Web theo đường dẫn "pynq:9090".

Với mật khẩu là: xilinx



Hình 65: Truy cập Jupyter Notebook

Tạo một Notebook mới để chạy phần mềm.



Hình 66: Mở cửa sổ Notebook mới

Để có chi tiết hơn các vấn đề thiết kế có thể tham khảo file Đồ án 1 được kèm chung theo Source code.