TRƯỜNG ĐẠI HỌC BÁCH KHOA TP.HCM
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

ĐỒ ÁN THIẾT KẾ
KỸ THUẬT MÁY TÍNH

Thiết kế SoC RISC-V tích hợp EdgeAI cho ứng dụng IoT

Học kỳ 251

GVHD: PGS. TS. Trần Ngọc Thịnh
ThS. Huỳnh Phúc Nghị

STT	Họ và tên	MSSV	Ghi chú
1	Lâm Nữ Uyển Nhi	2212429	
2	Vũ Đức Lâm	2211824	

TP. Hồ Chí Minh, Tháng 12/2025

Mục lục

Danh sách hình vẽ

Danh sách bảng biểu

Danh mục Ký hiệu và Chữ viết tắt

Ký hiệu	**Ý nghĩa**
H, W	Chiều cao và chiều rộng của đặc trưng đầu vào (Input Feature Map)
C	Số lượng kênh đầu vào (Input Channels)
N_f	Số lượng bộ lọc / Số kênh đầu ra (Number of Filters / Output Channels)
H_{out}, W_{out}	Chiều cao và chiều rộng của đặc trưng đầu ra (Output Feature Map)
R, S	Chiều cao và chiều rộng của bộ lọc (Kernel Height, Kernel Width)
P	Kích thước vùng đệm (Padding)
Str (hoặc U)	Bước trượt (Stride)
T_h	Chiều cao của mảnh dữ liệu đầu vào trong một Pass (Tile Height)
T_c	Số kênh đầu vào được xử lý song song trong một Pass (Tile Input Channels)
T_m	Số bộ lọc được tính toán song song trong một Pass (Tile Output Channels)
T_{ho}	Chiều cao hợp lệ của mảnh dữ liệu đầu ra trong một Pass

Ký hiệu	**Ý nghĩa**
b	Số chu kỳ đồng hồ để truyền một giá trị dữ liệu (Cycles per Data Transfer)
T_{comp}	Thời gian tính toán (Computation time)
T_{load}	Thời gian nạp dữ liệu (Load time)
T_{store}	Thời gian ghi dữ liệu (Store time)
T_{pass}	Thời gian hoàn thành một Pass
I	Tensor dữ liệu đầu vào
O	Tensor dữ liệu đầu ra
W	Tensor trọng số (Weights)
B	Vector hệ số chệch (Bias)
μ, σ	Giá trị trung bình (Mean) và Phương sai (Variance) trong Batch Norm
γ, β	Tham số tỉ lệ (Scale) và dịch chuyển (Shift) trong Batch Norm

Viết tắt	**Ý nghĩa**
AI	Trí tuệ nhân tạo (Artificial Intelligence)
CNN	Mạng nơ-ron tích chập (Convolutional Neural Network)
DNN	Mạng nơ-ron sâu (Deep Neural Network)
FPGA	Mảng cổng lập trình được dạng trường (Field-Programmable Gate Array)
SoC	Hệ thống trên chip (System-on-Chip)
RTL	Mức chuyển giao thanh ghi (Register Transfer Level)
IFM	Đặc trưng đầu vào (Input Feature Map)
OFM	Đặc trưng đầu ra (Output Feature Map)
PE	Phần tử xử lý (Processing Element)
PU	Đơn vị xử lý (Processing Unit - Chứa nhiều PE)
PA	Mảng xử lý (Process Array - Chứa nhiều PU)
MAC	Phép tính Nhân-Cộng tích lũy (Multiply-Accumulate)

Ký hiệu	**Ý nghĩa**
DMA	Truy cập bộ nhớ trực tiếp (Direct Memory Access)
AXI-Lite	Giao diện mở rộng nâng cao rút gọn (Advanced eXtensible Interface Lite)
AXI-Stream	Giao diện luồng dữ liệu mở rộng nâng cao (Advanced eXtensible Interface Stream)
BRAM	Block RAM (Bộ nhớ nội trên FPGA)
DMA	Truy cập bộ nhớ trực tiếp (Direct Memory Access)
AXI	Giao diện mở rộng nâng cao (Advanced eXtensible Interface)
DSP	Digital Signal Processing (Khối xử lý tín hiệu số trên FPGA)
LUT	Bảng tra (Look-Up Table)
FF	Flip-Flop
OSPI	Giao diện ngoại vi nối tiếp 8 kênh (Octal Serial Peripheral Interface)
SPI	Giao diện ngoại vi nối tiếp (Serial Peripheral Interface)
UART	Bộ truyền nhận dữ liệu nối tiếp bất đồng bộ (Universal Asynchronous Receiver-Transmitter)
I2C	Giao thức giao tiếp giữa các vi mạch (Inter-Integrated Circuit)
DVP	Cổng dữ liệu hình ảnh kỹ thuật số (Digital Video Port)
GPIO	Cổng vào/ra đa dụng (General Purpose Input/Output)

Chương 1

Giới thiệu đề tài

Chương này trình bày tổng quan về bối cảnh nghiên cứu, xác định mục tiêu cụ thể, phạm vi thực hiện.

1.1 Tổng quan về đề tài

Tên đề tài: Thiết kế SoC RISC-V tích hợp EdgeAI cho ứng dụng IoT.

Đề tài tập trung vào việc thiết kế và phát triển một hệ thống trên chip (SoC) dựa trên vi xử lý RISC-V tích hợp phần tăng tốc EdgeAI (Accelerator), nhằm xử lý các tác vụ trí tuệ nhân tạo ngay tại biên. Hệ thống sẽ được triển khai thử nghiệm trên nền tảng FPGA, với kiến trúc được tối ưu hóa nhằm hướng tới khả năng chuyển đổi sang thiết kế ASIC (Application-Specific Integrated Circuit) trong tương lai.

Bên cạnh việc thiết kế phần cứng, đề tài cũng bao gồm quá trình thử nghiệm hiệu suất hệ thống với một tập dữ liệu cố định và triển khai một số ứng dụng IoT thực tế làm "case study" (ví dụ: nhận diện hình ảnh từ camera) nhằm đánh giá tính khả thi và hiệu quả hoạt động của hệ thống trong môi trường thực tế.

Hệ thống hoàn chỉnh sẽ bao gồm các thành phần chính:

- Lõi vi xử lý RISC-V (CPU Core).
- Bộ tăng tốc mạng nơ-ron tích chập (CNN Accelerator).
- Hệ thống Bus giao tiếp nội bộ (AXI-Lite, AXI-Stream).
- Bộ truy cập bộ nhớ trực tiếp (DMA).
- Các giao tiếp I/O với ngoại vi (OSPI, SPI, UART, I2C, DVP, GPIO, TIMER,...).

1.2 Mục tiêu và Nhiệm vụ nghiên cứu

Mục tiêu chính của đề tài là nghiên cứu, thiết kế và hiện thực một hệ thống trên chip (SoC) hoàn chỉnh tích hợp lõi vi xử lý RISC-V và bộ tăng tốc phần cứng (Hardware Accelerator) cho các tác vụ trí tuệ nhân tạo tại biên (EdgeAI). Cụ thể, đề tài hướng tới các mục tiêu sau:

- **Về kiến trúc hệ thống:** Xây dựng kiến trúc SoC tối ưu năng lượng, sử dụng chuẩn giao tiếp AXI để kết nối giữa vi xử lý trung tâm và khối tăng tốc tính toán.
- **Về xử lý AI:** Thiết kế khối Accelerator chuyên dụng hỗ trợ các phép toán trọng yếu của mạng nơ-ron tích chập (CNN) như AlexNet, VGG16, MobileNetv1, nhằm giảm tải cho CPU và tăng tốc độ xử lý thực tế.
- **Về ứng dụng thực tế:** Tích hợp đầy đủ các giao tiếp ngoại vi (Camera/HDMI DVP, UART, SPI, OSPI, I2C, GPIO, TIMER) để xây dựng một ứng dụng IoT trọn vẹn (ví dụ: nhận diện vật thể hoặc phân loại ảnh) chạy trực tiếp trên nền tảng FPGA và hướng tới ASIC.

- **Về quy trình thiết kế:** Làm chủ quy trình thiết kế từ mức RTL (Verilog) đến mô phỏng (Simulation), tổng hợp (Synthesis) và kiểm tra trên phần cứng thực (FPGA Prototyping).

1.3 Phạm vi đề tài

Để đảm bảo tính khả thi trong khuôn khổ thời gian của đồ án, nhóm thực hiện xác định phạm vi nghiên cứu như sau:

1.3.1 Phạm vi và Giới hạn đề tài

- **Vi xử lý:** Sử dụng lõi PicoRV32 (RISC-V 32-bit - RV32I) mã nguồn mở, tập trung vào việc tích hợp và xây dựng hệ thống bus (System Interconnect) thay vì thiết kế lại kiến trúc nhân CPU.
- **Mô hình AI:** Tập trung hỗ trợ các mạng CNN cơ bản (như LeNet-5, MobileNet dạng rút gọn) đã được lượng tử hóa (Quantization) xuống 8-bit integer để phù hợp với tài nguyên phần cứng, không đi sâu vào việc huấn luyện (training) các mô hình lớn.
- **Nền tảng phần cứng:** Hệ thống được thiết kế bằng ngôn ngữ Verilog và kiểm chứng trên Kit FPGA (AMD Virtex™ 7 FPGA VC707, Arty A7-100T Artix-7 FPGA). Chưa bao gồm các bước thiết kế vật lý (Physical Design) để ra chip ASIC thực tế (Layout, GDSII).

1.3.2 Đối tượng và Công cụ nghiên cứu

- Ngôn ngữ thiết kế: Verilog, C/C++.
- Công cụ mô phỏng và tổng hợp: Vivado Design Suite.
- Framework AI hỗ trợ: PyTorch/TensorFlow (để trích xuất trọng số mô hình).

1.4 Phân chia công việc

Đồ án được thực hiện bởi hai thành viên với sự phân chia công việc cụ thể dựa trên kiến trúc hệ thống như sau:

Bảng 1.1: Bảng phân chia công việc của các thành viên

STT	**Thành viên**	**Nội dung thực hiện**
1	**Lâm Nữ Uyển Nhi** (Chịu trách nhiệm về Accelerator)	• Nghiên cứu lý thuyết về mạng CNN và các kỹ thuật tối ưu phần cứng. • Thiết kế kiến trúc khối CNN Accelerator (PE Array, Buffer Controller). • Hiện thực các khối tính toán: Convolution, Pooling, ReLU. • Viết Testbench kiểm tra chức năng khối Accelerator.
2	**Vũ Đức Lâm** (Chịu trách nhiệm về SoC & System)	• Nghiên cứu kiến trúc RISC-V và chuẩn bus AMBA AXI. • Thiết kế hệ thống SoC: Tích hợp CPU, Interconnect, Memory Controller. • Thiết kế các giao tiếp ngoại vi: UART, SPI, OSPI, I2C, GPIO, TIMER, DVP (Camera/HDMI). • Phát triển Firmware/Driver để điều khiển hệ thống. • Tổng hợp hệ thống lên FPGA và đo đạc hiệu năng.

1.5 Cấu trúc báo cáo

Báo cáo được trình bày trong 7 chương với nội dung cụ thể như sau:

- **Chương 1 - Giới thiệu đề tài:** Trình bày tổng quan, mục tiêu, phạm vi và kế hoạch thực hiện đồ án.
- **Chương 2 - Cơ sở lý thuyết:** Cung cấp kiến thức nền tảng về kiến trúc tập lệnh RISC-V, mạng nơ-ron tích chập (CNN), các chuẩn giao tiếp (AXI, UART, SPI,...) và công nghệ FPGA.
- **Chương 3 - Phân tích và Kiến trúc hệ thống:** Phân tích yêu cầu bài toán, từ đó đề xuất kiến trúc tổng thể của SoC và sơ đồ khối chi tiết.
- **Chương 4 - Thiết kế Bộ tăng tốc AI (AI Accelerator):** Trình bày chi tiết thiết kế phần cứng của khối xử lý CNN, bao gồm kiến trúc mảng tính toán và quản lý dữ liệu.
- **Chương 5 - Hiện thực SoC và Tích hợp hệ thống:** Mô tả quá trình tích hợp các module vào hệ thống bus, thiết kế bộ nhớ và các ngoại vi, cũng như quy trình tổng hợp trên FPGA.
- **Chương 6 - Đánh giá kết quả:** Trình bày phương pháp kiểm thử, kết quả tài nguyên sử dụng (Resource Utilization), công suất tiêu thụ và so sánh hiệu năng thực tế.
- **Chương 7 - Kết luận và Hướng phát triển:** Tóm tắt các kết quả đạt được và đề xuất các hướng cải tiến trong tương lai.

Chương 2

Cơ sở lý thuyết

Chương này cung cấp các kiến thức nền tảng về kiến trúc tập lệnh RISC-V, mô hình mạng nơ-ron tích chập (CNN), các chuẩn giao tiếp dữ liệu (AXI, UART, SPI, OSPI, I2C, Camera/HDMI DVP) và công nghệ FPGA được sử dụng trong đề tài.

2.1 Kiến trúc tập lệnh RISC-V

2.1.1 Tổng quan về kiến trúc RISC-V

RISC-V là một kiến trúc tập lệnh (ISA - Instruction Set Architecture) mã nguồn mở, ra đời vào năm 2010 tại Đại học California, Berkeley. Khác với các kiến trúc thương mại phổ biến như x86 (Intel) hay ARM, RISC-V được thiết kế dựa trên nguyên lý máy tính tập lệnh rút gọn (RISC) thuần túy, loại bỏ các gánh nặng tương thích ngược của các kiến trúc cũ để tối ưu hóa hiệu năng và năng lượng.

Đặc điểm cốt lõi của RISC-V là tính mô-đun hóa và khả năng mở rộng. Kiến trúc này không định nghĩa một tập lệnh khổng lồ duy nhất, mà chia thành:

- **Tập lệnh cơ sở (Base ISA):** Là phần cứng tối thiểu bắt buộc phải

có để một vi xử lý được gọi là RISC-V. Đối với các ứng dụng nhúng 32-bit, chuẩn này là **RV32I** (Base Integer). Nó cung cấp đầy đủ các lệnh để thực thi tính toán nguyên, truy cập bộ nhớ và điều khiển luồng chương trình.

- **Các phần mở rộng (Extensions):** Là các mô-đun tùy chọn để tăng cường sức mạnh xử lý. Ví dụ: M (Integer Multiplication/Division), A (Atomic instructions), F (Single-precision Floating-point), C (Compressed instructions - nén lệnh 16-bit để tiết kiệm bộ nhớ).

Sự kết hợp này tạo nên chuỗi định danh cho vi xử lý, ví dụ **RV32IMAC** biểu thị vi xử lý 32-bit có hỗ trợ nhân chia, thao tác nguyên tử và lệnh nén.

2.1.2 Mô hình lập trình và Tập thanh ghi

Theo đặc tả của RV32I, trạng thái kiến trúc của một luồng xử lý (Hart - Hardware Thread) bao gồm hai thành phần chính: bộ đếm chương trình (PC) và tập thanh ghi mục đích chung (GPR).

2.1.2.1 Bộ đếm chương trình (Program Counter - PC)

PC là một thanh ghi 32-bit lưu trữ địa chỉ của lệnh đang được thực thi. Trong RISC-V, PC không phải là một thanh ghi mục đích chung (không thể đánh địa chỉ trực tiếp như GPR). Giá trị của PC chỉ có thể thay đổi thông qua các lệnh rẽ nhánh, nhảy hoặc lệnh hệ thống. Khi khởi động (Reset), PC sẽ được nạp một địa chỉ cố định (Reset Vector) để bắt đầu chu trình nạp lệnh.

2.1.2.2 Tập thanh ghi mục đích chung (General Purpose Registers)

RV32I cung cấp 32 thanh ghi, được đánh số từ `x0` đến `x31`, mỗi thanh ghi rộng 32-bit (XLEN=32). Để đảm bảo chương trình phần mềm hoạt động chính xác với phần cứng, đặc biệt khi sử dụng bộ công cụ biên dịch **RISC-V GNU Toolchain (GCC)**, các thanh ghi này phải tuân thủ chuẩn Giao diện Nhị phân Ứng dụng (ABI - Application Binary Interface). Trình biên dịch GCC sử dụng các tên quy ước (như `sp`, `ra`, `a0`...) thay vì tên phần cứng (`x2`, `x1`, `x10`...) để quản lý việc gọi hàm và truyền tham số. Chi tiết chức năng được trình bày trong Bảng 2.1.

Bảng 2.1: Tập thanh ghi mục đích chung của RISC-V (RV32I)

Tên thanh ghi	**Tên ABI**	**Mô tả chức năng**	**Lưu bởi**
`x0`	`zero`	Luôn bằng 0 (Hardwired zero)	N/A
`x1`	`ra`	Địa chỉ trả về (Return Address)	Caller
`x2`	`sp`	Con trỏ ngăn xếp (Stack Pointer)	Callee
`x3`	`gp`	Con trỏ toàn cục (Global Pointer)	N/A
`x4`	`tp`	Con trỏ luồng (Thread Pointer)	N/A
`x5`	`t0`	Thanh ghi tạm thời / Địa chỉ trả về thay thế	Caller
`x6 - x7`	`t1 - t2`	Thanh ghi tạm thời (Temporaries)	Caller
`x8`	`s0 / fp`	Thanh ghi lưu trữ / Con trỏ khung (Frame Pointer)	Callee
`x9`	`s1`	Thanh ghi lưu trữ (Saved register)	Callee
`x10 - x11`	`a0 - a1`	Đối số hàm / Giá trị trả về	Caller
`x12 - x17`	`a2 - a7`	Đối số hàm (Function Arguments)	Caller
`x18 - x27`	`s2 - s11`	Thanh ghi lưu trữ (Saved registers)	Callee
`x28 - x31`	`t3 - t6`	Thanh ghi tạm thời (Temporaries)	Caller

Trong đó:

- **Caller-saved:** Giá trị không được bảo toàn qua lời gọi hàm (hàm con có thể ghi đè).
- **Callee-saved:** Giá trị phải được bảo toàn (nếu hàm con muốn dùng, phải lưu ra stack trước và khôi phục lại trước khi return).

2.1.3 Đặc tả tập lệnh cơ sở RV32I

Tập lệnh RV32I bao gồm 47 lệnh cơ bản. Một điểm đặc biệt trong thiết kế của RISC-V là việc cố định độ dài lệnh ở 32-bit và căn chỉnh bộ nhớ theo từ (word-aligned), giúp đơn giản hóa mạch giải mã lệnh và dự đoán rẽ nhánh.

2.1.3.1 Định dạng lệnh (Instruction Formats)

RISC-V sử dụng 6 định dạng lệnh cơ bản (R, I, S, B, U, J). Điểm tối ưu trong thiết kế định dạng lệnh của RISC-V là vị trí của các trường thanh ghi nguồn (`rs1`, `rs2`) và thanh ghi đích (`rd`) luôn được giữ cố định tại các bit giống nhau trong mọi định dạng lệnh (xem Hình 2.1).
Điều này cho phép bộ giải mã (Decoder) có thể bắt đầu đọc dữ liệu từ tập thanh ghi (Register File) ngay lập tức mà không cần phải chờ xác định xong loại lệnh (Opcode), giúp giảm độ trễ trong đường ống xử lý.

Bit	**31...25**	**24...20**	**19...15**	**14...12**	**11...7**	**6...0**
R-type	funct7	rs2	rs1	funct3	rd	opcode
I-type	imm[11:0]		rs1	funct3	rd	opcode
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
B-type	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U-type	imm[31:12]				rd	opcode
J-type	imm[20|10:1|11|19]				rd	opcode

Hình 2.1: Cấu trúc bit của các định dạng lệnh RV32I

Dưới đây là giải thích chi tiết ý nghĩa của từng loại định dạng lệnh:

- **R-type (Register):** Dùng cho các lệnh thao tác trực tiếp giữa thanh ghi và thanh ghi (ví dụ: `add x1, x2, x3`).
- **I-type (Immediate):** Dùng cho các lệnh thao tác với hằng số ngắn (Immediate) và các lệnh nạp dữ liệu (Load) từ bộ nhớ.

- **S-type (Store):** Dùng chuyên biệt cho các lệnh lưu dữ liệu từ thanh ghi vào bộ nhớ.
- **B-type (Branch):** Dùng cho các lệnh rẽ nhánh có điều kiện (ví dụ: so sánh bằng, so sánh lớn hơn).
- **U-type (Upper Immediate):** Dùng để thao tác với các hằng số lớn (20-bit cao), thường dùng để nạp địa chỉ nền.
- **J-type (Jump):** Dùng cho các lệnh nhảy vô điều kiện (dùng trong gọi hàm hoặc vòng lặp).

Một kỹ thuật quan trọng khác là việc mã hóa giá trị tức thời (Immediate Encoding). Trong các lệnh dạng S và B, các bit giá trị tức thời bị phân mảnh và xáo trộn. Tuy nhiên, việc xáo trộn này được thiết kế có chủ đích để các bit này luôn tương ứng với cùng một vị trí bit đầu ra của bộ tạo giá trị tức thời (Immediate Generator), giúp giảm số lượng tầng logic (Fan-out) trong phần cứng.

2.1.3.2 Phân nhóm chức năng chi tiết

1. Lệnh tính toán số nguyên (Integer Computational Instructions):

Nhóm lệnh này thực hiện các phép toán số học và logic. Chúng không gây ra ngoại lệ số học và không thay đổi bất kỳ cờ trạng thái nào (RISC-V không sử dụng thanh ghi cờ như ARM/x86).

- **Tính toán với hằng số (I-Type):** `ADDI`, `ANDI`, `ORI`, `XORI`, `SLTI` (Set Less Than Immediate). Lệnh `LUI` (Load Upper Immediate) dùng để nạp 20-bit cao vào thanh ghi.
- **Tính toán giữa các thanh ghi (R-Type):** `ADD`, `SUB`, `AND`, `OR`, `XOR`. Lệnh `SLT/SLTU` so sánh hai thanh ghi và ghi giá trị 1 vào đích nếu nhỏ hơn, ngược lại ghi 0.

- **Dịch bit:** `SLL/SLLI` (Dịch trái logic), `SRL/SRLI` (Dịch phải logic - chèn 0), `SRA/SRAI` (Dịch phải số học - giữ nguyên dấu).

2. Lệnh truy cập bộ nhớ (Load and Store Instructions):
RISC-V sử dụng kiến trúc Load-Store thuần túy. Việc tính toán địa chỉ bộ nhớ luôn thông qua công thức: $Address = rs1 + sign\_extend(imm)$.

- **Load:** `LW` (32-bit), `LH` (16-bit), `LB` (8-bit). Các biến thể `LHU` và `LBU` dùng để nạp dữ liệu không dấu, trong đó phần bit cao của thanh ghi đích sẽ được điền 0 (Zero-extension) thay vì mở rộng dấu (Sign-extension).
- **Store:** `SW`, `SH`, `SB`. Lệnh store chỉ lấy các bit thấp tương ứng trong thanh ghi nguồn để ghi vào bộ nhớ.

3. Lệnh điều khiển luồng (Control Transfer Instructions):
RISC-V khác biệt so với các kiến trúc cũ ở chỗ lệnh rẽ nhánh thực hiện so sánh trực tiếp hai thanh ghi.

- **Rẽ nhánh có điều kiện (Branch):** `BEQ` (Bằng), `BNE` (Không bằng), `BLT/BGE` (So sánh có dấu), `BLTU/BGEU` (So sánh không dấu). Việc tách biệt so sánh có dấu và không dấu giúp lập trình viên kiểm soát chính xác các cấu trúc điều khiển.
- **Nhảy vô điều kiện (Jump):**
  - `JAL` (Jump and Link): Nhảy đến địa chỉ tương đối so với PC, đồng thời lưu địa chỉ lệnh kế tiếp (PC+4) vào thanh ghi `rd` (thường là `ra`).
  - `JALR` (Jump and Link Register): Nhảy đến địa chỉ tuyệt đối được tính từ thanh ghi cơ sở + offset. Lệnh này hỗ trợ việc gọi hàm qua con trỏ hoặc quay về từ hàm (Return).

4. Lệnh môi trường hệ thống (System Environment):
Hai lệnh quan trọng nhất là `ECALL` (Environment Call) dùng để tạo yêu cầu

phục vụ từ hệ điều hành (System Call) và `EBREAK` (Environment Break) dùng để chuyển quyền kiểm soát cho trình gỡ lỗi (Debugger). Ngoài ra, các lệnh `CSRRW, CSRRS, CSRRC` dùng để đọc/ghi các thanh ghi trạng thái điều khiển (CSR) nhằm quản lý ngắt và cấu hình hệ thống.

2.1.4 Vi xử lý PicoRV32

Trong đồ án này, nhóm thực hiện lựa chọn lõi vi xử lý **PicoRV32** để làm bộ xử lý trung tâm cho hệ thống SoC.

PicoRV32 là một hiện thực phần cứng (CPU Core) của kiến trúc RISC-V, hỗ trợ đầy đủ tập lệnh cơ sở **RV32I**. Đặc điểm nổi bật của PicoRV32 là sự tối ưu hóa về mặt diện tích và tài nguyên trên FPGA, thay vì tập trung vào hiệu năng đường ống (Pipeline) phức tạp. Nó hoạt động dựa trên máy trạng thái đa chu kỳ, cho phép đạt tần số hoạt động cao và dễ dàng tích hợp vào các thiết kế SoC nhỏ gọn phục vụ ứng dụng IoT. Ngoài ra, PicoRV32 cung cấp giao diện đồng xử lý (PCPI), cho phép mở rộng khả năng tính toán thông qua các bộ tăng tốc phần cứng bên ngoài.

2.2 Tổng quan về Mạng nơ-ron tích chập (CNN)

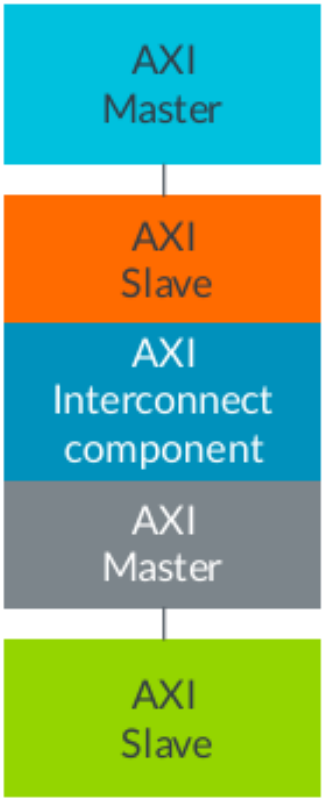
2.3 Các chuẩn giao tiếp hệ thống

2.3.1 Chuẩn giao tiếp AMBA AXI4

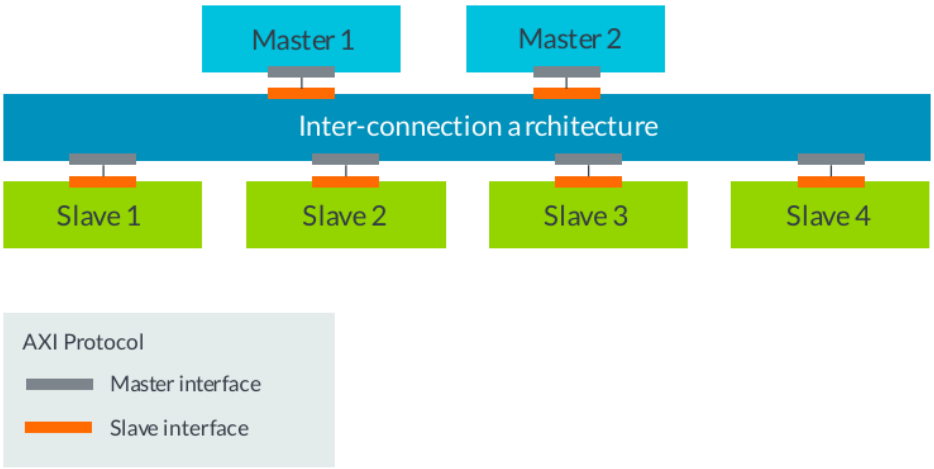
AMBA (Advanced Microcontroller Bus Architecture) là tiêu chuẩn kết nối trên chip (On-Chip Interconnect) phổ biến nhất hiện nay, được phát triển bởi ARM. Trong đó, giao thức AXI (Advanced eXtensible Interface) là chuẩn giao tiếp hiệu năng cao, được thiết kế cho các hệ thống SoC yêu cầu băng thông lớn và độ trễ thấp.

Phiên bản AXI4 (được giới thiệu trong AMBA 4.0) hỗ trợ các tính năng vượt trội so với các thế hệ trước:

- Tách biệt hoàn toàn pha địa chỉ/điều khiển và pha dữ liệu.
- Hỗ trợ giao dịch dữ liệu không thẳng hàng (Unaligned data transfers).
- Cho phép phát hành nhiều địa chỉ chờ (Outstanding addresses) trước khi dữ liệu hoàn tất.
- Hỗ trợ hoàn thành giao dịch không theo thứ tự (Out-of-order completion) thông qua ID.



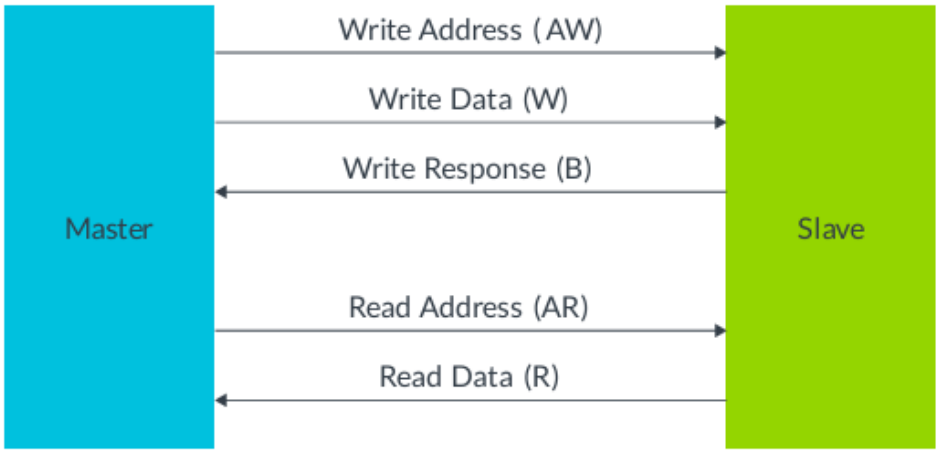
Hình 2.2: a. Tổng quan giao thức AXI4



Hình 2.3: b. Tổng quan giao thức AXI4

2.3.1.1 Kiến trúc 5 kênh độc lập (Channel Architecture)

AXI chia nhỏ một giao dịch truyền thông thành 5 kênh riêng biệt hoạt động song song. Kiến trúc này cho phép đường truyền dữ liệu hai chiều (Full-duplex), nghĩa là Master có thể ghi dữ liệu vào Slave trong khi đang đọc dữ liệu từ Slave khác.



Hình 2.4: Mô hình 5 kênh giao tiếp của AXI4

Năm kênh tín hiệu bao gồm:

1. **Write Address Channel (AW):** Master gửi địa chỉ bắt đầu và thông tin điều khiển (loại burst, độ dài) cho giao dịch ghi. Các tín

hiệu bắt đầu bằng `AW...` (ví dụ: `AWADDR`, `AWVALID`).

2. **Write Data Channel (W):** Master truyền dữ liệu thực tế tới Slave. Kênh này hỗ trợ tín hiệu `WSTRB` (Strobe) để đánh dấu các byte hợp lệ trong một word (hỗ trợ ghi từng byte). Các tín hiệu bắt đầu bằng `W...`.

3. **Write Response Channel (B):** Slave gửi phản hồi trạng thái (OKAY, ERROR) cho Master sau khi toàn bộ dữ liệu đã được ghi thành công. Tín hiệu bắt đầu bằng `B...`.

4. **Read Address Channel (AR):** Master gửi địa chỉ bắt đầu cho giao dịch đọc. Tín hiệu bắt đầu bằng `AR...`.

5. **Read Data Channel (R):** Slave trả về dữ liệu yêu cầu cùng với trạng thái đọc. Tín hiệu bắt đầu bằng `R...`.

2.3.1.2 Cơ chế bắt tay (Handshake Mechanism)

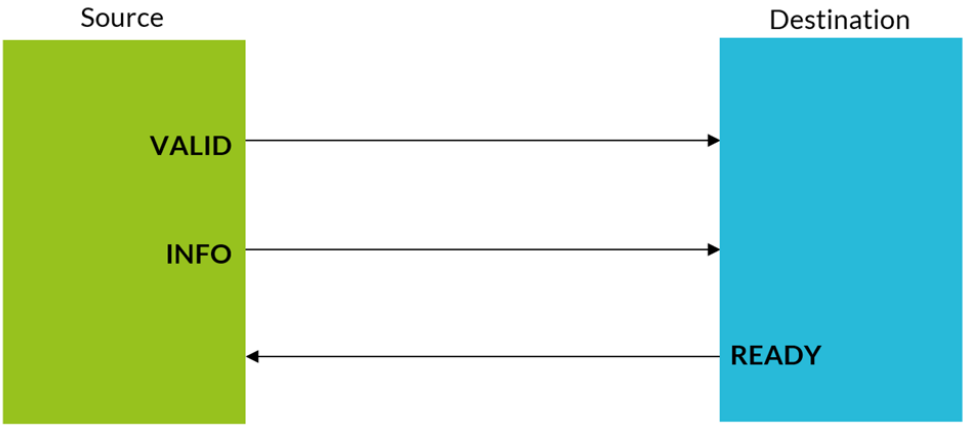
Toàn bộ 5 kênh AXI đều sử dụng chung một cơ chế bắt tay hai chiều `VALID/READY` để điều khiển luồng dữ liệu:

- **VALID (từ Bên gửi):** Báo hiệu rằng dữ liệu hoặc địa chỉ trên đường truyền đã hợp lệ và ổn định.
- **READY (từ Bên nhận):** Báo hiệu rằng bên nhận đã sẵn sàng chấp nhận dữ liệu mới.

Hình 2.5: Cơ chế bắt tay VALID/READY trong AXI

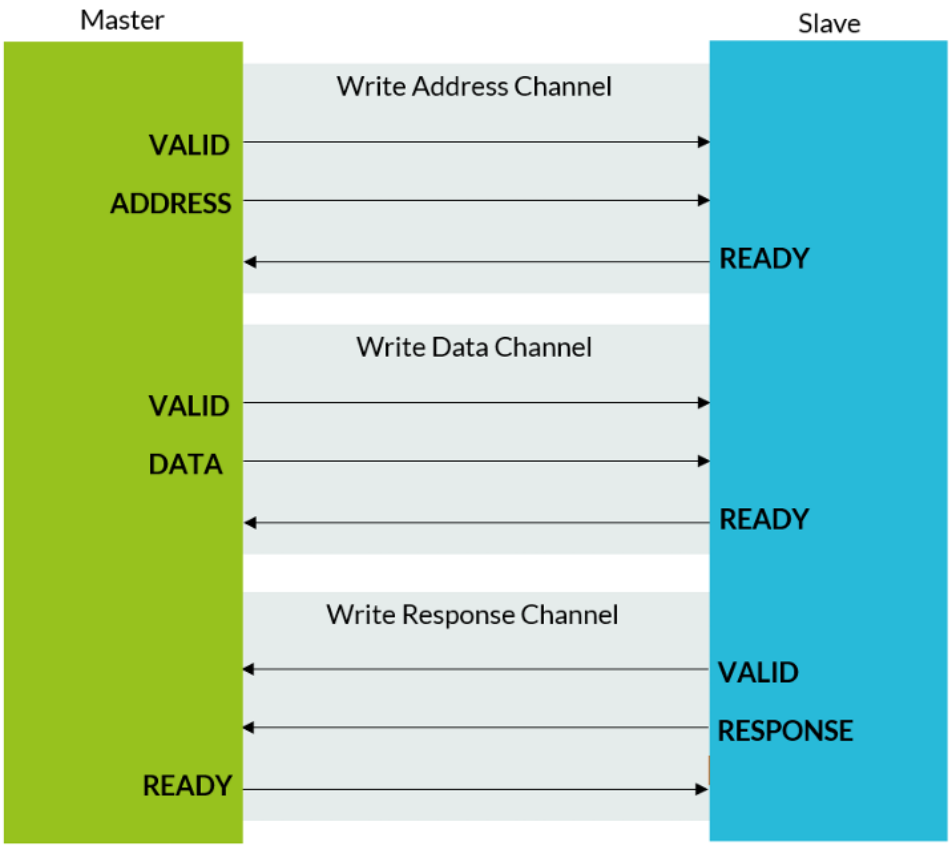
Giao dịch chỉ thực sự diễn ra tại cạnh dương của xung nhịp khi và chỉ khi cả `VALID` và `READY` đều ở mức cao (High). Cơ chế này cho phép bên nhận có thể "kìm" (back-pressure) bên gửi nếu bộ đệm bị đầy, hoặc bên gửi có thể đợi chuẩn bị dữ liệu xong mới phát tín hiệu.
Dựa trên cơ chế bắt tay này, chuẩn AXI định nghĩa hai cấp độ truyền tải dữ liệu cần phân biệt rõ:

- **Transfer (hoặc Beat):** Là một lần trao đổi dữ liệu đơn lẻ thành công (một lần bắt tay VALID/READY = 1). Trong một chuỗi dữ liệu (Burst), mỗi nhịp truyền một gói tin (ví dụ 32-bit) được gọi là một Transfer.



Hình 2.6: Minh họa một Transfer trong AXI

- **Transaction (Giao dịch):** Là một hoạt động đọc hoặc ghi hoàn chỉnh. Một Transaction bao gồm toàn bộ quá trình: gửi địa chỉ (Address Phase), truyền một hoặc nhiều dữ liệu (Data Phase - gồm nhiều Transfers) và nhận phản hồi (Response Phase).



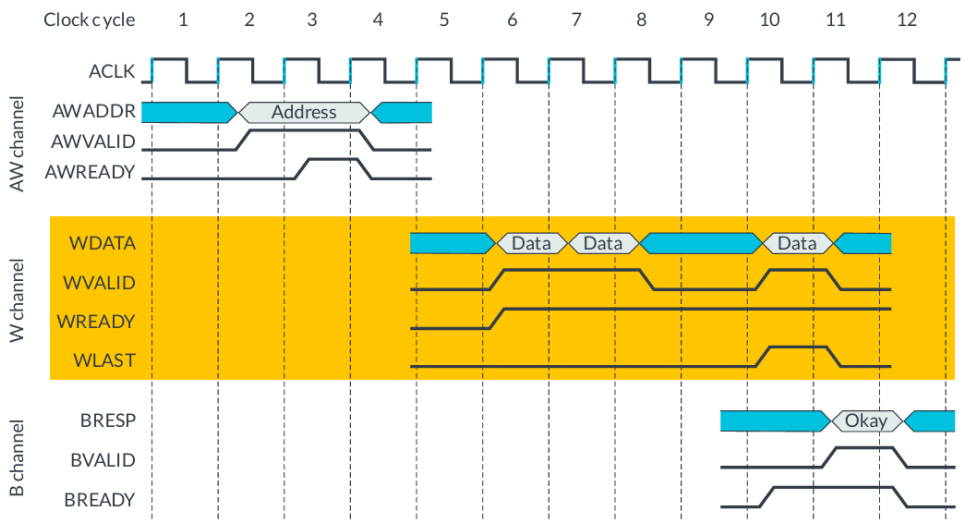
Hình 2.7: Minh họa một Transaction trong AXI

2.3.1.3 Quy trình thực hiện giao dịch chi tiết (Transaction Steps)

Để đảm bảo toàn vẹn dữ liệu, chuẩn AXI quy định chặt chẽ về hướng đi của tín hiệu và trình tự bắt tay giữa Master và Slave. Dưới đây là mô tả chi tiết các tín hiệu tham gia vào hai loại giao dịch cơ bản.

1. Giao dịch Ghi (Write Transaction)

Quá trình ghi dữ liệu diễn ra qua 3 pha, sử dụng các kênh AW, W và B.



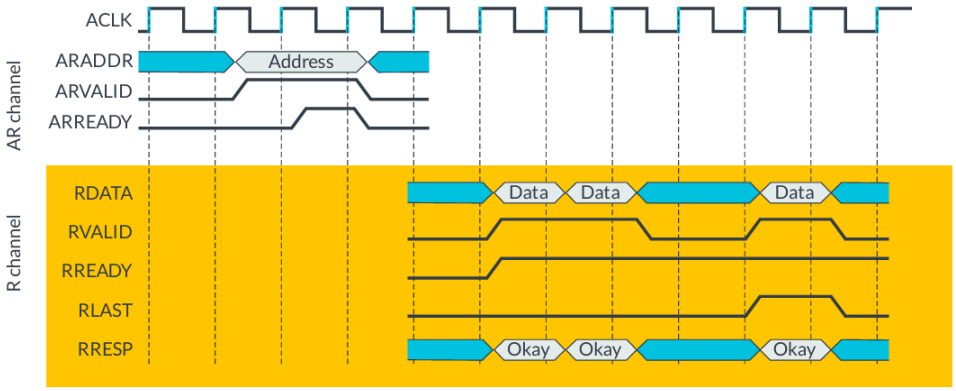
Hình 2.8: Giản đồ tín hiệu chi tiết của giao dịch Ghi

- **Pha địa chỉ (Write Address Channel):**
  - **Master → Slave:** Master đặt địa chỉ lên bus `AWADDR` và các thông tin điều khiển (Burst type, length) lên `AWLEN`, `AWSIZE`... sau đó xác lập tín hiệu `AWVALID = 1`.
  - **Slave → Master:** Khi Slave sẵn sàng nhận địa chỉ, nó bật `AWREADY = 1`. Giao dịch địa chỉ hoàn tất.
- **Pha dữ liệu (Write Data Channel):**
  - **Master → Slave:** Master đưa dữ liệu lên bus `WDATA`. Nếu đây là gói cuối cùng trong Burst, Master bật tín hiệu `WLAST = 1`. Đồng thời, Master xác lập `WVALID = 1`.
  - **Slave → Master:** Slave bật `WREADY = 1` để báo hiệu đã nhận gói dữ liệu đó. Quá trình lặp lại cho đến hết Burst.
- **Pha phản hồi (Write Response Channel):**
  - **Slave → Master:** Sau khi nhận đủ dữ liệu và hoàn tất việc ghi vào bộ nhớ, Slave gửi trạng thái (ví dụ: OKAY) qua bus `BRESP` và xác lập `BVALID = 1`.

– **Master → Slave:** Master xác nhận đã nhận được phản hồi bằng cách bật `BREADY = 1`. Kết thúc giao dịch.

2. Giao dịch Đọc (Read Transaction)

Quá trình đọc dữ liệu diễn ra qua 2 pha, sử dụng kênh AR và R.



Hình 2.9: Giản đồ tín hiệu chi tiết của giao dịch Đọc

- **Pha địa chỉ (Read Address Channel):**
  - **Master → Slave:** Master đặt địa chỉ cần đọc lên bus `ARADDR` cùng các tham số điều khiển, sau đó bật `ARVALID = 1`.
  - **Slave → Master:** Slave chấp nhận địa chỉ bằng cách bật `ARREADY = 1`.
- **Pha dữ liệu (Read Data Channel):**
  - **Slave → Master:** Slave truy xuất dữ liệu và đưa lên bus `RDATA`. Nếu thành công, Slave gửi kèm trạng thái OKAY trên bus `RRESP`. Tại gói dữ liệu cuối cùng, Slave bật `RLAST = 1`. Tín hiệu `RVALID = 1` được xác lập khi dữ liệu trên bus là hợp lệ.
  - **Master → Slave:** Master nhận dữ liệu bằng cách bật `RREADY = 1`.

2.3.1.4 Cấu trúc giao dịch Burst (Burst Transaction)

AXI là giao thức dựa trên Burst, nghĩa là chỉ cần gửi một địa chỉ khởi đầu, Master có thể truyền liên tiếp một chuỗi dữ liệu (tức là thực hiện một Transaction gồm nhiều Transfers). Các tham số chính điều khiển Burst bao gồm:

- **Burst Length (AxLEN):** Số lượng gói dữ liệu (beat/transfer) trong một burst. AXI4 hỗ trợ lên đến 256 beat cho kiểu INCR.
- **Burst Size (AxSIZE):** Số byte trong mỗi beat (ví dụ: 4 bytes cho hệ thống 32-bit).
- **Burst Type (AxBURST):** Xác định cách tính địa chỉ cho các beat tiếp theo:
  - *FIXED:* Địa chỉ giữ nguyên (dùng cho FIFO).
  - *INCR (Incrementing):* Địa chỉ tăng dần (dùng cho RAM). Đây là kiểu phổ biến nhất.
  - *WRAP:* Địa chỉ tăng đến giới hạn biên rồi quay vòng (dùng cho Cache Line fill).

2.3.1.5 Các biến thể giao thức trong thiết kế

Trong phiên bản AXI4, chuẩn AMBA định nghĩa thêm các biến thể rút gọn để phù hợp với từng mục đích sử dụng cụ thể:

1. Giao thức AXI4-Lite (AXI-Lite)

AXI4-Lite là một phiên bản rút gọn của AXI4, được thiết kế cho các giao tiếp điều khiển đơn giản, không yêu cầu truyền dữ liệu tốc độ cao (Burst transfer). Đặc điểm chính của AXI4-Lite bao gồm:

- Mỗi giao dịch chỉ truyền một gói dữ liệu đơn lẻ (Burst length = 1).
- Dữ liệu thường có độ rộng 32-bit hoặc 64-bit cố định.

- Đơn giản hóa logic điều khiển, giảm diện tích phần cứng.

Nhờ sự đơn giản này, AXI4-Lite thường được sử dụng làm giao diện cấu hình cho các thanh ghi điều khiển (Control Registers) bên trong các khối IP (Intellectual Property).

2. Giao thức AXI4-Stream (AXI-Stream)

AXI4-Stream được thiết kế chuyên biệt cho việc truyền tải các luồng dữ liệu liên tục tốc độ cao (Streaming data) mà không cần sử dụng địa chỉ. Khác với AXI4-Lite hay AXI4-Full (Memory Mapped), AXI4-Stream chỉ tập trung vào việc đẩy dữ liệu từ nguồn (Master) đến đích (Slave) nhanh nhất có thể.

- Không có kênh địa chỉ (Address Channel), giảm đáng kể số lượng dây tín hiệu.
- Hỗ trợ truyền dữ liệu liên tục không giới hạn độ dài Burst.
- Thích hợp cho dữ liệu video, âm thanh hoặc dữ liệu mạng nơ-ron (Feature maps).

2.3.1.6 Áp dụng trong hệ thống đề tài

Trong khuôn khổ đồ án thiết kế SoC RISC-V tích hợp EdgeAI này, nhóm thực hiện áp dụng kết hợp cả hai chuẩn giao tiếp trên để tối ưu hóa hiệu năng và tài nguyên:

- **Sử dụng AXI4-Lite:** Đóng vai trò là kênh điều khiển (Control Plane). Vi xử lý PicoRV32 (Master) sẽ sử dụng AXI4-Lite để ghi vào các thanh ghi cấu hình của khối ngoại vi, khối Accelerator và DMA, thiết lập các thông số như kích thước ảnh, địa chỉ bộ nhớ và tín hiệu bắt đầu (Start).
- **Sử dụng AXI4-Stream:** Đóng vai trò là kênh dữ liệu (Data Plane). Dữ liệu hình ảnh từ Camera và các ma trận trọng số (Weights) sẽ

được truyền trực tiếp từ DMA vào khối Accelerator thông qua AXI4-Stream. Việc loại bỏ overhead của kênh địa chỉ giúp tối đa hóa băng thông xử lý cho mạng CNN.

2.3.2 Giao thức truyền thông UART

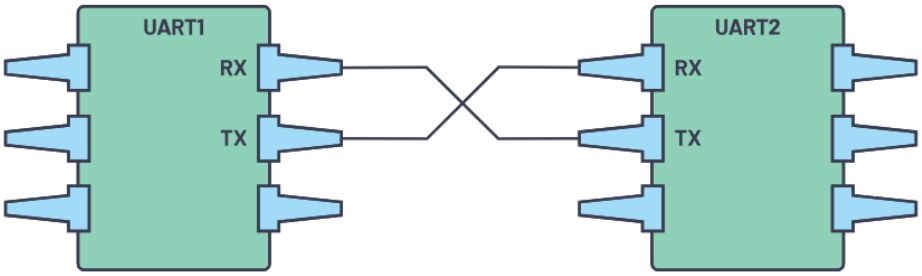
UART (Universal Asynchronous Receiver-Transmitter) là một vi mạch phần cứng dùng để truyền tải dữ liệu nối tiếp giữa hai thiết bị. Khác với các giao thức đồng bộ như SPI hay I2C, UART hoạt động theo cơ chế bất đồng bộ (Asynchronous), nghĩa là không cần tín hiệu xung nhịp (Clock) chung để đồng bộ hóa việc truyền nhận giữa bên gửi và bên nhận. Trong các thiết kế SoC, UART thường được tích hợp như một khối ngoại vi (Peripheral) để phục vụ việc gỡ lỗi (Debug), in log hệ thống hoặc giao tiếp với máy tính.

2.3.2.1 Nguyên lý hoạt động

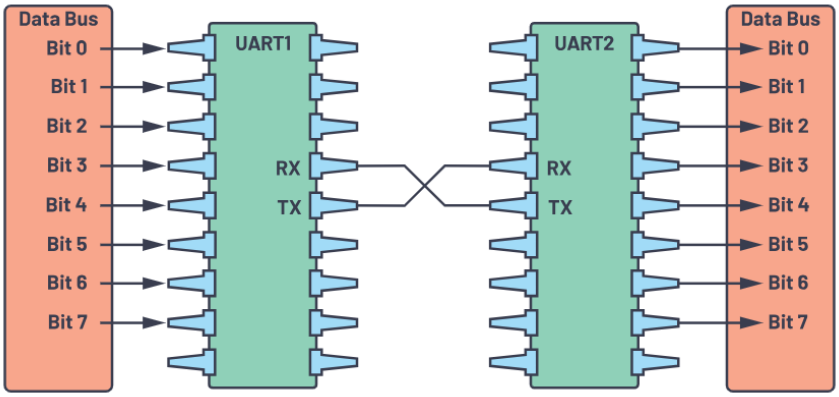
Giao thức UART truyền dữ liệu trên hai dây tín hiệu riêng biệt:

- **TX (Transmit):** Chân truyền dữ liệu đi.
- **RX (Receive):** Chân nhận dữ liệu về.

Để giao tiếp thành công, chân TX của thiết bị này phải được nối với chân RX của thiết bị kia và ngược lại. Quá trình truyền tin diễn ra bằng cách chuyển đổi dữ liệu song song (Parallel data) từ bus hệ thống thành luồng dữ liệu nối tiếp (Serial bit stream) tại phía phát, và khôi phục lại thành song song tại phía thu.



Hình 2.10: Minh họa chân kết nối truyền nhận dữ liệu UART



Hình 2.11: Chuyển đổi dữ liệu song song thành nối tiếp và ngược lại trong UART

2.3.2.2 Cấu trúc khung dữ liệu (Data Frame)

Do không có xung nhịp đồng bộ, UART sử dụng các bit điều khiển đặc biệt để đánh dấu điểm bắt đầu và kết thúc của một gói tin. Một khung dữ liệu chuẩn bao gồm các thành phần sau:

1. **Trạng thái nghỉ (Idle State):** Khi không có dữ liệu truyền, đường truyền luôn được giữ ở mức điện áp cao (Logic 1).

2. **Start Bit:** Để bắt đầu một phiên truyền, thiết bị phát sẽ kéo đường truyền từ mức cao xuống mức thấp (Logic 0) trong một chu kỳ bit. Bên thu phát hiện cạnh xuống này để bắt đầu quá trình đồng bộ.

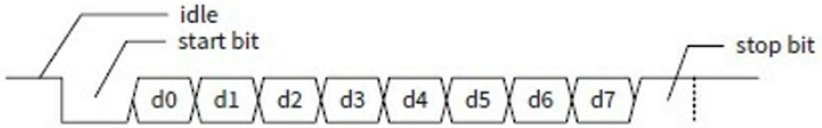
3. **Data Bits:** Chứa thông tin thực tế cần truyền, thường có độ dài từ 5 đến 9 bit (phổ biến nhất là 8 bit). Theo quy ước, bit có trọng số

nhỏ nhất (LSB) được truyền đi trước.

4. **Parity Bit (Tùy chọn):** Dùng để kiểm tra lỗi đơn giản. Bit này có thể được cấu hình là chẵn (Even), lẻ (Odd) hoặc không sử dụng (None). Nếu sử dụng, tổng số bit '1' trong gói dữ liệu (bao gồm cả parity) phải thỏa mãn quy tắc chẵn/lẻ đã thiết lập.

5. **Stop Bit:** Đánh dấu kết thúc gói tin bằng cách kéo đường truyền về mức cao (Logic 1). Độ dài có thể là 1, 1.5, hoặc 2 bit thời gian. Stop bit đảm bảo đường truyền quay về trạng thái nghỉ để sẵn sàng cho Start bit tiếp theo.

Hình 2.12: Khung dữ liệu UART



Hình 2.13: Ví dụ khung dữ liệu UART với 8bit dữ liệu, không parity và 1 stop bit

2.3.2.3 Tốc độ Baud (Baud Rate)

Vì thiếu xung nhịp đồng bộ, hai thiết bị UART phải thống nhất trước một tốc độ truyền nhận, gọi là Baud Rate (đơn vị: bit/giây - bps).

- Bên phát sẽ đẩy từng bit dữ liệu ra đường truyền với chu kỳ $T = 1/BaudRate$.
- Bên thu sẽ lấy mẫu tín hiệu (sample) tại điểm giữa của mỗi chu kỳ bit dự kiến để đọc dữ liệu.

Theo khuyến cáo kỹ thuật, độ sai lệch tốc độ Baud giữa hai thiết bị không được vượt quá 10% để đảm bảo dữ liệu được đọc chính xác. Các tốc độ phổ biến thường dùng là 9600, 19200, 115200 bps.

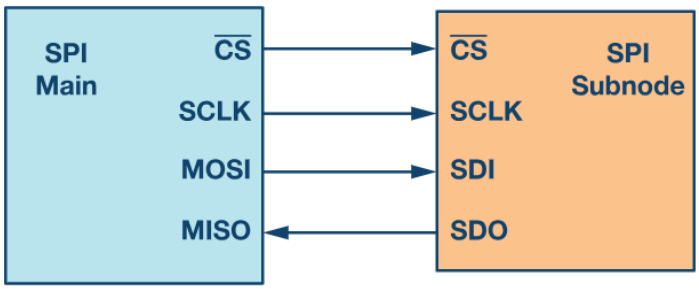
2.3.3 Giao thức truyền thông SPI

SPI (Serial Peripheral Interface) là chuẩn giao tiếp nối tiếp đồng bộ tốc độ cao, hoạt động ở chế độ song công toàn phần (Full-duplex). Chuẩn này được Motorola giới thiệu vào giữa những năm 1980 và hiện nay đã trở thành tiêu chuẩn công nghiệp để kết nối vi xử lý với các thiết bị ngoại vi như cảm biến, bộ nhớ Flash (SPI Flash), màn hình LCD, hoặc bộ chuyển đổi ADC/DAC.

Khác với UART (bất đồng bộ) hay I2C (bán song công, tốc độ thấp), SPI sử dụng đường xung nhịp riêng biệt và kiến trúc Master-Slave chặt chẽ, cho phép đạt băng thông truyền tải rất cao (có thể lên tới hàng chục MHz).

2.3.3.1 Cấu hình tín hiệu vật lý

Một bus SPI tiêu chuẩn (4-wire mode) bao gồm 4 đường tín hiệu logic kết nối giữa Master và Slave.



Hình 2.14: Sơ đồ kết nối tín hiệu chuẩn 4 dây của SPI

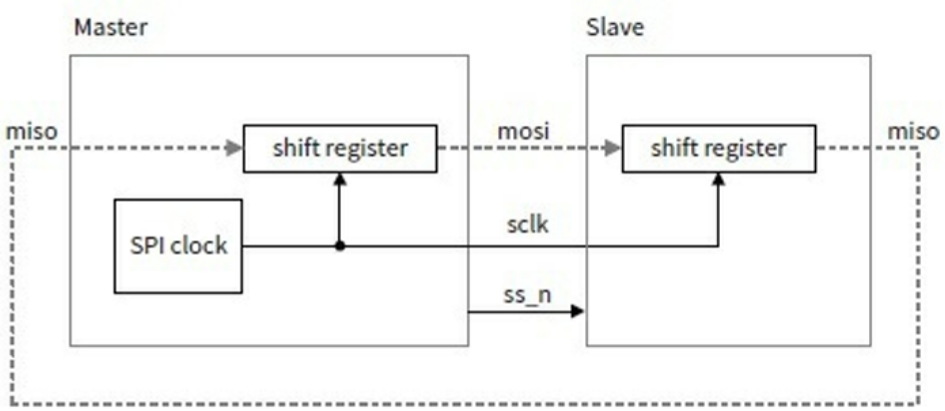
Chức năng các chân tín hiệu bao gồm:

- **SCLK (Serial Clock):** Tín hiệu xung nhịp do Master tạo ra. Toàn bộ quá trình truyền nhận dữ liệu được đồng bộ theo cạnh lên hoặc cạnh xuống của xung này. Slave không được phép tạo xung Clock.

- **MOSI (Master Out Slave In):** Đường truyền dữ liệu từ Master đến Slave.
- **MISO (Master In Slave Out):** Đường truyền dữ liệu từ Slave về Master. Nếu chỉ có Master gửi dữ liệu (ví dụ điều khiển LCD), chân này có thể bỏ qua.
- **CS/SS (Chip Select / Slave Select):** Tín hiệu chọn thiết bị, thường hoạt động ở mức thấp (Active Low). Master kéo chân này xuống 0V để bắt đầu giao dịch với một Slave cụ thể.

2.3.3.2 Cơ chế hoạt động: Thanh ghi dịch (Shift Register)

Cốt lõi của giao thức SPI là cấu trúc thanh ghi dịch vòng tròn (Circular Shift Register).



Hình 2.15: Cơ chế trao đổi dữ liệu dùng thanh ghi dịch trong SPI

Quá trình truyền nhận diễn ra như sau:

1. Master và Slave mỗi bên đều có một thanh ghi dịch (thường là 8-bit hoặc 16-bit).
2. Tại mỗi chu kỳ xung nhịp SCLK:
    - 1 bit dữ liệu từ Master được đẩy ra đường MOSI và dịch vào thanh ghi của Slave.

- Đồng thời, 1 bit dữ liệu từ Slave được đẩy ra đường MISO và dịch vào thanh ghi của Master.

3. Sau N chu kỳ xung nhịp (với N là độ rộng dữ liệu), giá trị trong thanh ghi của Master và Slave được trao đổi hoàn toàn cho nhau.

2.3.3.3 Các chế độ hoạt động (Clock Polarity & Phase)

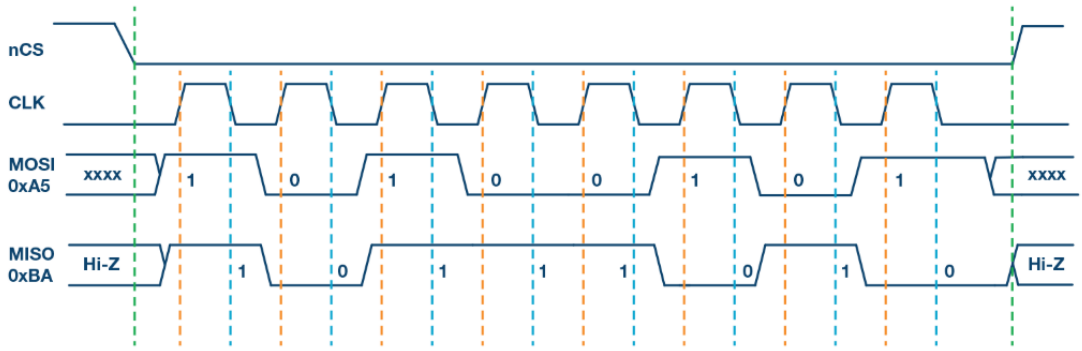
SPI định nghĩa 4 chế độ hoạt động (Modes) dựa trên trạng thái của xung Clock, được quy định bởi hai tham số:

- **CPOL (Clock Polarity):** Trạng thái nghỉ của đường SCLK (0 hoặc 1).
- **CPHA (Clock Phase):** Cạnh lên hoặc xuống của xung dùng để lấy mẫu (Sample) và dùng để thay đổi dữ liệu (Shift).

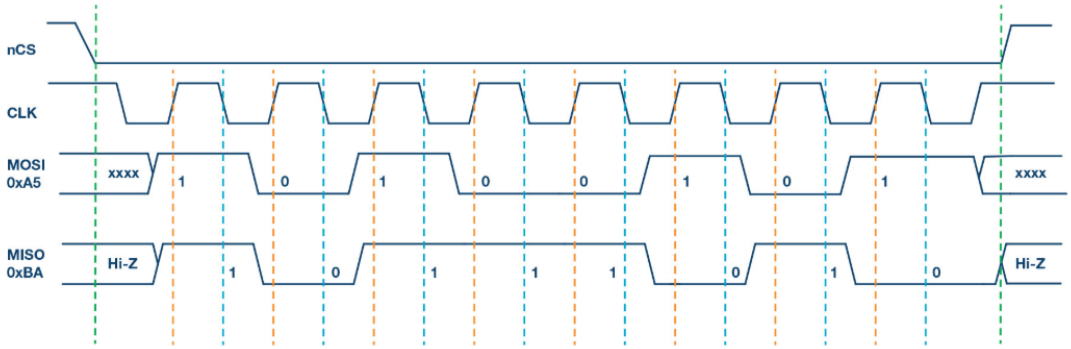
SPI Mode	CPOL	CPHA	Clock Polarity in Idle State	Clock Phase Used to Sample and/or Shift the Data
0	0	0	Logic low	Data sampled on rising edge and shifted out on the falling edge
1	0	1	Logic low	Data sampled on the falling edge and shifted out on the rising edge
2	1	0	Logic high	Data sampled on the falling edge and shifted out on the rising edge
3	1	1	Logic high	Data sampled on the rising edge and shifted out on the falling edge

Hình 2.16: 4 chế độ hoạt động của SPI(CPOL/CPHA)

Lưu ý: Mode 0 và Mode 3 là hai cấu hình phổ biến nhất. Master và Slave phải được cấu hình cùng một Mode để giao tiếp thành công.



Hình 2.17: SPI MODE 0 (CPOL=0, CPHA=0), trạng thái SCLK ban đầu ở mức low, dữ liệu được lấy mẫu tại cạnh lên của SCLK và dịch ở cạnh xuống



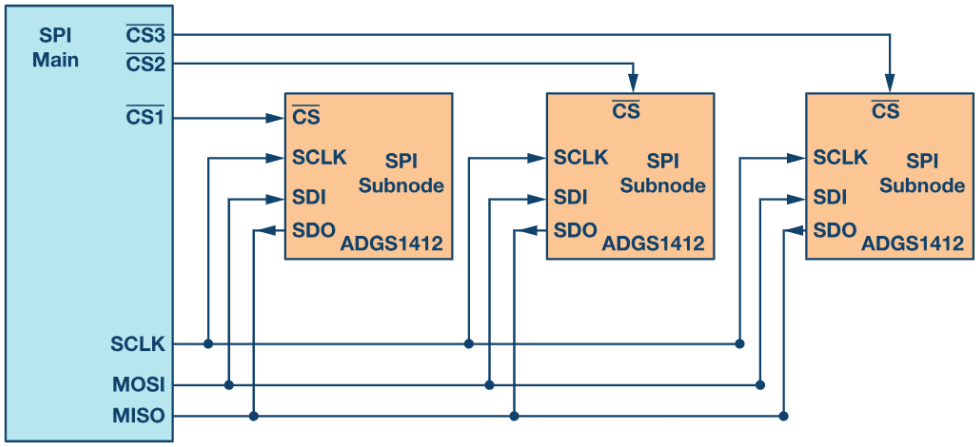
Hình 2.18: SPI MODE 3 (CPOL=1, CPHA=1), trạng thái SCLK ban đầu ở mức high, dữ liệu được lấy mẫu tại cạnh lên của SCLK và dịch ở cạnh xuống

2.3.3.4 Các mô hình kết nối đa thiết bị

SPI cho phép một Master giao tiếp với nhiều Slave thông qua hai cấu hình chính:

1. Cấu hình Slave độc lập (Independent Slaves):

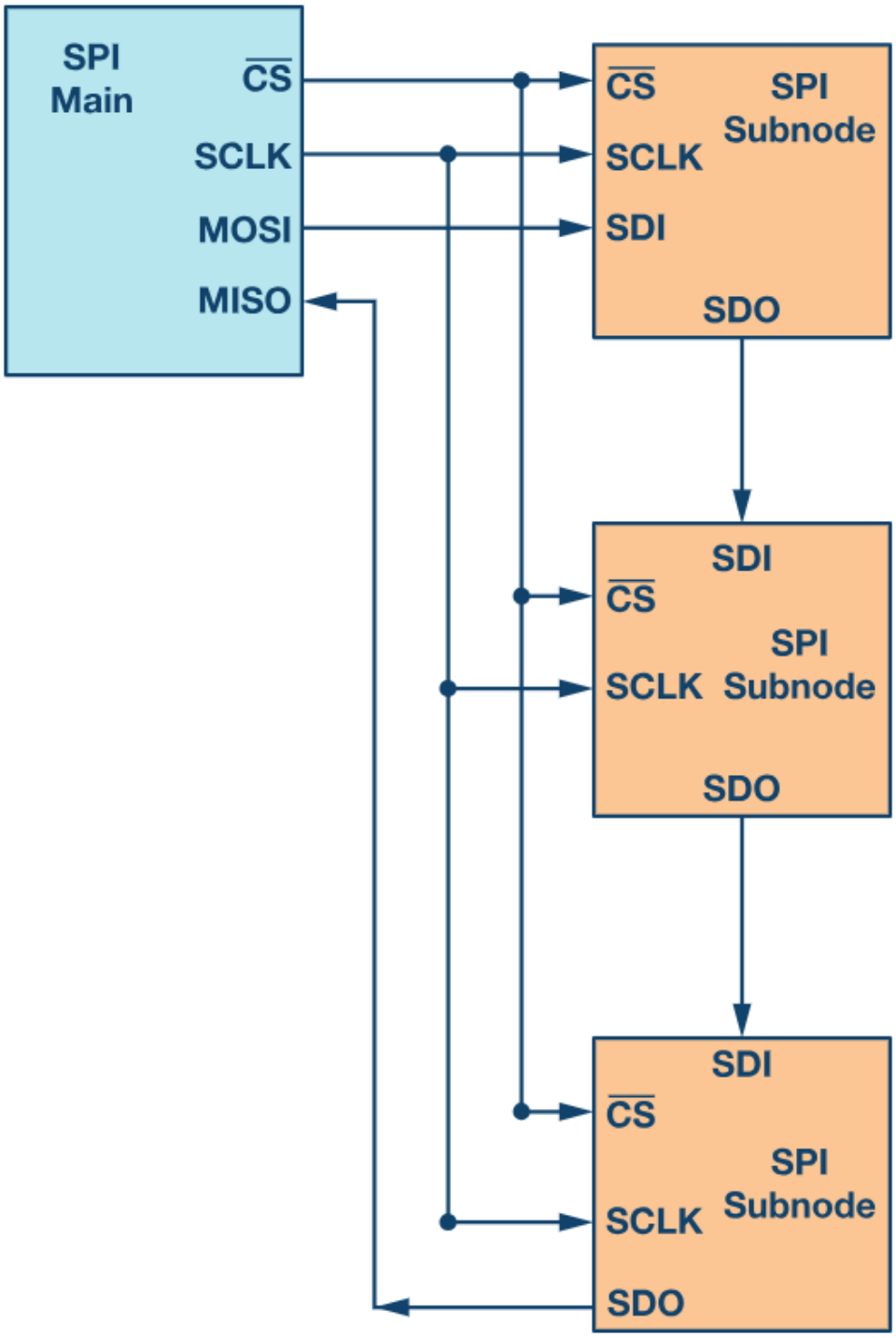
Master sử dụng các chân CS riêng biệt ($CS_1, CS_2, ...$) cho từng Slave. Đây là cấu hình phổ biến giúp tối ưu băng thông.



Hình 2.19: Cấu hình Slave độc lập trong SPI

2. Cấu hình Chuỗi (Daisy Chain):

Các Slave được nối tiếp nhau (MISO của Slave này nối vào MOSI của Slave kia). Dữ liệu đi qua chuỗi các thiết bị, giúp tiết kiệm chân điều khiển của Master nhưng làm giảm tốc độ truyền tổng thể.



Hình 2.20: Cấu hình Chuỗi (Daisy Chain) trong SPI

SPI có tốc độ truyền cao nhất so với UART và I2C, phần cứng đơn giản, hỗ trợ Full-duplex. Nhưng tốn nhiều dây tín hiệu, khoảng cách truyền ngắn, không có cơ chế xác nhận lỗi (ACK) như I2C.

2.4 Công nghệ FPGA và Quy trình thiết kế

Chương 3

Phân tích và Kiến trúc hệ thống

Dựa trên cơ sở lý thuyết đã trình bày, chương này đi sâu vào phân tích các yêu cầu kỹ thuật, từ đó đề xuất kiến trúc tổng thể của hệ thống SoC (System-on-Chip). Đồng thời, chương này cũng xác định đặc tả chức năng của từng khối thành phần và quy hoạch không gian địa chỉ bộ nhớ (Memory Map) cho toàn hệ thống.

3.1 Phân tích yêu cầu thiết kế

3.2 Kiến trúc tổng thể SoC

3.3 Đặc tả các khối chức năng chính

3.4 Tổ chức bộ nhớ và Bản đồ địa chỉ (Memory Map)

Chương 4

Thiết kế Bộ tăng tốc AI (AI Accelerator):

Chương này trình bày chi tiết thiết kế của lõi IP Accelerator, bao gồm phân tích toán học, chiến lược tối ưu dòng dữ liệu và kiến trúc vi mô.

4.1 Phân tích toán học của phép tính tích chập

Để đảm bảo tính linh hoạt cho kiến trúc phần cứng, giúp hệ thống có khả năng hỗ trợ đa dạng các mô hình mạng nơ-ron từ kinh điển (như VGG16) đến các mô hình tối ưu cho thiết bị biên (như MobileNet), nhóm thực hiện đề tài đã tập trung phân tích đặc tả toán học của hai loại phép tính cốt lõi: **Standard Convolution** và **Depthwise Separable Convolution**. Việc hiểu rõ bản chất toán học và cấu trúc dữ liệu của các phép tính này (bao gồm cả cơ chế xử lý biên - Padding) là cơ sở quan trọng để chúng tôi thiết kế nên một kiến trúc thống nhất (Unified Architecture).

4.1.1 Standard Convolution (Tích chập tiêu chuẩn)

Đây là phép tính nền tảng trong hầu hết các mạng CNN truyền thống. Về mặt toán học, tích chập tiêu chuẩn thực hiện việc trượt bộ lọc (filter) trên không gian đầu vào (H, W), đồng thời tích lũy giá trị qua toàn bộ chiều sâu của kênh (Channels).

4.1.1.1 Công thức toán học tổng quát

Xét một lớp tích chập với đầu vào I có kích thước $C \times H_{in} \times W_{in}$ và bộ trọng số W có kích thước $M \times C \times R \times S$. Tham số Padding (P) được sử dụng để giữ nguyên kích thước không gian hoặc kiểm soát việc giảm kích thước. Giá trị đầu ra O tại kênh m, vị trí (h, w) được xác định bởi:

$$O[m][h][w] = B[m] + \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} I[c][h \cdot U + r - P][w \cdot U + s - P] \times W[m][c][r][s] \tag{4.1}$$

Trong đó:

- U: Bước trượt (Stride).
- P: Số lượng điểm ảnh đệm thêm vào mỗi cạnh (Padding).
- Điều kiện biên: Nếu chỉ số truy cập I nằm ngoài phạm vi $[0, H_{in} - 1]$ hoặc $[0, W_{in} - 1]$, giá trị trả về là 0 (Zero-padding).

4.1.1.2 Cấu trúc vòng lặp (Loop Nest)

Với giả thiết kích thước batch $N = 1$, chúng tôi mô hình hóa phép tính này dưới dạng 6 vòng lặp lồng nhau. Việc xử lý Padding thường được thực hiện bằng phần cứng chuyên dụng (Padding Logic) để tránh truy cập bộ nhớ ngoài vùng cho phép.

Algorithm 1: Standard Convolution (Standard Conv2D)

Input: $I[C][H_{in}][W_{in}]$, $W[M][C][R][S]$, Padding P, Stride U

Output: $O[M][H_{out}][W_{out}]$

for $m = 0$ ***to*** $M-1$ **do**
  for $c = 0$ ***to*** $C-1$ **do**
    for $h = 0$ ***to*** $H_{out}-1$ **do**
      for $w = 0$ ***to*** $W_{out}-1$ **do**
        for $r = 0$ ***to*** $R-1$ **do**
          for $s = 0$ ***to*** $S-1$ **do**
            $h_{in} = h \cdot U + r - P$
            $w_{in} = w \cdot U + s - P$
            if $h_{in} \geq 0 \wedge h_{in} < H_{in} \wedge w_{in} \geq 0 \wedge w_{in} < W_{in}$ **then**
              $val = I[c][h_{in}][w_{in}]$
            else
              $val = 0$ `/* Zero Padding */`
            end
            $O[m][h][w] \leftarrow O[m][h][w] + val \times W[m][c][r][s]$
          end
        end
      end
    end
  end
end

4.1.2 Depthwise Separable Convolution

Để giảm chi phí tính toán cho các thiết bị biên, các mô hình như MobileNet sử dụng kỹ thuật **Depthwise Separable Convolution**, tách phép chập chuẩn thành hai bước: **Depthwise (DW)** và **Pointwise (PW)**.

4.1.2.1 Depthwise Convolution (DW)

Phép tính này áp dụng bộ lọc riêng cho từng kênh đầu vào. Công thức tính toán bao gồm tham số Padding như sau:

$$O_{dw}[c][h][w] = \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} I[c][h \cdot U + r - P][w \cdot U + s - P] \times W_{dw}[c][r][s] \quad (4.2)$$

Nhận xét: Việc xử lý Padding trong Depthwise cũng tương tự như Standard Conv, tuy nhiên do tính độc lập giữa các kênh, bộ điều khiển (Controller) cần đảm bảo logic Padding hoạt động chính xác cho từng luồng tính toán song song.

Algorithm 2: Depthwise Convolution (với Padding)

```
for c = 0 to C − 1                                   /* Parallel Channels */
 do
    for h = 0 to H_out − 1 do
        for w = 0 to W_out − 1 do
            for r = 0 to R − 1 do
                for s = 0 to S − 1 do
                    h_in = h · U + r − P
                    w_in = w · U + s − P
                    if h_in ∈ [0, H_in) ∧ w_in ∈ [0, W_in) then
                        O_dw[c][h][w] += I[c][h_in][w_in] × W_dw[c][r][s]
                    end
                end
            end
        end
    end
end
```

4.1.2.2 Pointwise Convolution (PW)

Pointwise Convolution là tích chập chuẩn với kernel 1×1. Do kích thước kernel là 1×1, tham số Padding thường được đặt bằng 0 ($P = 0$) và Stride $U = 1$ để giữ nguyên kích thước không gian (H, W), chỉ thay đổi số kênh từ C sang M.

$$O_{pw}[m][h][w] = \sum_{c=0}^{C-1} I[c][h][w] \times W_{pw}[m][c] \tag{4.3}$$

4.1.3 Yêu cầu đối với Kiến trúc thống nhất (Unified Architecture)

Từ các phân tích trên, nhóm nhận thấy rằng để bộ tăng tốc hoạt động hiệu quả cho cả hai trường hợp, kiến trúc phần cứng cần giải quyết được bài toán "kép":

1. **Cơ chế xử lý Padding động:** Phần cứng cần có khối logic để tự động chèn giá trị 0 khi chỉ số tính toán $(h \cdot U + r - P)$ bị âm hoặc vượt quá kích thước ảnh, thay vì phải tốn tài nguyên bộ nhớ để lưu trữ các viền số 0 thực tế.

2. **Tính linh hoạt của Mảng PE:** Các đơn vị tính toán cần có khả năng chuyển đổi chế độ giữa tích lũy theo không gian (Standard/Pointwise) và tính toán độc lập theo kênh (Depthwise).

4.1.4 Kỹ thuật Gập Batch Normalization (BN Folding)

Trong các mạng CNN hiện đại như MobileNet, lớp Batch Normalization (BN) thường được đặt ngay sau lớp Convolution để chuẩn hóa phân phối dữ liệu, giúp mạng hội tụ nhanh hơn. Công thức tính toán của lớp BN trong quá trình suy luận (Inference) cho một kênh m là:

$$y = \frac{x - \mu_m}{\sqrt{\sigma_m^2 + \epsilon}} \cdot \gamma_m + \beta_m \tag{4.4}$$

Trong đó:

- x: Giá trị đầu ra từ lớp Convolution (trước khi qua hàm kích hoạt).
- μ_m, σ_m: Giá trị trung bình (mean) và phương sai (variance) động (running statistics) của kênh m.
- γ_m, β_m: Tham số tỉ lệ (scale) và dịch chuyển (shift) được học trong quá trình huấn luyện.
- ϵ: Hằng số nhỏ để tránh chia cho 0.

Việc thực hiện trực tiếp công thức này trên phần cứng rất tốn kém do yêu cầu các phép toán phức tạp như căn bậc hai và phép chia. Tuy nhiên, do tại thời điểm suy luận, các tham số $\mu, \sigma, \gamma, \beta$ đều là hằng số, chúng tôi áp dụng kỹ thuật **BN Folding** để gộp toàn bộ phép tính BN vào trong trọng số (W) và bias (B) của lớp Convolution đi trước nó.

4.1.4.1 Công thức biến đổi trọng số

Giả sử đầu ra của lớp Convolution là $x = W_{orig} \cdot Input + B_{orig}$. Khi thay vào công thức BN, ta có:

$$y = \frac{(W_{orig} \cdot Input + B_{orig}) - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta \tag{4.5}$$

Phương trình trên có thể được viết lại dưới dạng một phép Convolution mới với trọng số W' và bias B':

$$y = W' \cdot Input + B' \tag{4.6}$$

Trong đó, các tham số mới được tính toán trước (offline) bởi phần mềm

(driver) trước khi nạp xuống phần cứng:

$$W'[m][c][r][s] = W_{orig}[m][c][r][s] \cdot \frac{\gamma_m}{\sqrt{\sigma_m^2 + \epsilon}} \tag{4.7}$$

$$B'[m] = \left(B_{orig}[m] - \mu_m\right) \cdot \frac{\gamma_m}{\sqrt{\sigma_m^2 + \epsilon}} + \beta_m \tag{4.8}$$

Kết luận thiết kế: Nhờ kỹ thuật BN Folding, kiến trúc phần cứng của chúng tôi **không cần** thiết kế khối chức năng riêng cho Batch Normalization. Accelerator chỉ cần thực hiện phép tính Convolution bình thường với bộ trọng số (W', B') đã được tinh chỉnh, giúp tiết kiệm đáng kể tài nguyên DSP và giảm độ trễ xử lý.

4.2 Chiến lược phân mảnh và Dòng dữ liệu đề xuất

Để xử lý các Feature Map kích thước lớn trên tài nguyên phần cứng giới hạn, chúng tôi áp dụng chiến lược phân mảnh dữ liệu (Tiling) không chồng lấn. Mục này sẽ trình bày chi tiết cách thức chia nhỏ không gian dữ liệu và thuật toán điều phối các bước tính toán (Passes).

4.2.1 Định nghĩa khái niệm "Tile" (Mảnh dữ liệu)

Trong kiến trúc này, một "Tile" được định nghĩa là một phần nhỏ của khối dữ liệu gốc với kích thước được tối ưu hóa cho dung lượng bộ nhớ on-chip. Hệ thống xử lý ba loại Tile chính tương ứng với ba luồng dữ liệu. Đầu tiên, Input Tile (Mảnh đầu vào) là khối dữ liệu được cắt ra từ Input Feature Map gốc với kích thước $T_c \times T_h \times W$. Việc giới hạn số kênh nạp vào là T_c và chiều cao là T_h giúp dữ liệu vừa vặn với bộ nhớ đệm, trong khi chiều rộng W được giữ nguyên để tận dụng tính liên tục của dữ liệu trong bộ nhớ (Burst Read). Tiếp theo, Weight Tile (Mảnh trọng số) tập hợp các bộ lọc

cần thiết để xử lý cho Input Tile hiện tại, có kích thước $T_m \times T_c \times R \times S$. Cuối cùng, kết quả tính toán tương ứng tạo ra Output Tile (Mảnh đầu ra) với kích thước $T_m \times T_h \times W$.

4.2.2 Phương pháp Phân mảnh không gian dữ liệu (Space Partitioning)

Không gian tính toán của một lớp tích chập được định nghĩa bởi ba chiều chính: Chiều cao không gian (H), Chiều sâu kênh đầu vào (C), và Số lượng bộ lọc/kênh đầu ra (M). Chúng tôi chia nhỏ không gian này thành các khối (Block/Tile) độc lập dựa trên tham số phần cứng.

4.2.2.1 Công thức chia khối (Block Calculation)

Giả sử phần cứng có khả năng xử lý song song một khối dữ liệu kích thước $T_c \times T_h \times W$ và tạo ra T_m kênh đầu ra. Số lượng khối (Blocks) trên mỗi chiều được tính như sau:

- **Số khối theo chiều dọc (N_h):** Ảnh đầu vào chiều cao H được cắt thành N_h phần không chồng lấn.

$$N_h = \lceil \frac{H}{T_h} \rceil \tag{4.9}$$

  Ví dụ: Với $H = 21, T_h = 11$, ta có $N_h = 2$ khối (Khối 0: hàng 0-10; Khối 1: hàng 11-20).

- **Số khối kênh đầu vào (N_c):** Tổng C kênh được chia thành N_c nhóm.

$$N_c = \lceil \frac{C}{T_c} \rceil \tag{4.10}$$

- **Số khối kênh đầu ra (N_m):** Tổng M bộ lọc được chia thành N_m nhóm.

$$N_m = \lceil \frac{M}{T_m} \rceil \tag{4.11}$$

Một đơn vị xử lý cơ sở, gọi là **1 Pass**, chính là quá trình hệ thống xử lý hoàn tất cho một cặp Input Tile và Weight Tile để cập nhật giá trị cho một Output Tile.

4.2.3 Mô hình hóa toán học và Tham số thiết kế

Để hiện thực hóa chiến lược phân mảnh, chúng tôi xây dựng mô hình toán học cho lớp tích chập thứ i. Các ký hiệu và tham số thiết kế được chuẩn hóa trong Bảng 4.1.

Bảng 4.1: Bảng tham số thiết kế và ánh xạ ký hiệu

Nhóm tham số	**Ký hiệu**	**Mô tả**
Filter	S, R	Độ rộng (w_f) và Độ dài (h_f) của bộ lọc
	N_f	Tổng số bộ lọc (Filters)
Feature Map	W, H, C	Kích thước Input Feature Map (Rộng, Dài, Số kênh)
	W_{out}, H_{out}, N_f	Kích thước Output Feature Map
Tiling (Pass)	T_h	Chiều cao IFM nạp trong 1 pass (h)
	T_c	Số kênh IFM tính toán song song (k)
	T_m	Số bộ lọc tính toán song song (m)
Output Tile	T_{ho}	Chiều cao OFM hợp lệ tạo ra trong 1 pass (h_o)
Khác	P, Str	Padding và Stride

4.2.3.1 Công thức tính kích thước không gian toàn cục

Dựa trên nguyên lý tích chập, một layer cần xử lý khối dữ liệu vào $W \times H \times C$. Kích thước không gian của Output Feature Map (OFM) toàn cục được tính như sau:

$$W_{out} = \left\lfloor \frac{W - S + 2P}{Str} \right\rfloor + 1; \quad H_{out} = \left\lfloor \frac{H - R + 2P}{Str} \right\rfloor + 1 \tag{4.12}$$

4.2.3.2 Tính toán kích thước Output Tile theo cơ chế Ping-Pong

Trong chiến lược phân mảnh đề xuất, chiều cao đầu ra hợp lệ (T_{ho}) không cố định mà phụ thuộc vào việc hệ thống có tận dụng được dữ liệu dôi ra (Residual Data) từ pass trước đó hay không.

Giả sử bước trượt $Str = 1$ và Padding được xử lý tại biên ảnh gốc, số lượng hàng Output hợp lệ ghi xuống DRAM trong mỗi Pass được xác định bởi công thức:

$$T_{ho} = \begin{cases} T_h - R + 1 & \text{nếu là Tile đầu tiên } (Tile\_idx = 0) \\ T_h & \text{nếu là các Tile tiếp theo } (Tile\_idx > 0) \end{cases} \tag{4.13}$$

Giải thích:

- **Tile đầu tiên ($T_{ho} = T_h - R + 1$):** Do không có dữ liệu tích lũy từ phía trên, $R - 1$ hàng cuối cùng không đủ dữ liệu lân cận để hoàn thành phép tính tích chập, trở thành dữ liệu dôi ra (Residual) được lưu vào Buffer.
- **Các Tile tiếp theo ($T_{ho} = T_h$):** Hệ thống nạp T_h hàng input mới, kết hợp với $R - 1$ hàng residual từ tile trước đó. Điều này cho phép hoàn thiện $R - 1$ hàng biên cũ và tính trọn vẹn phần thân mới, tạo ra đủ T_h hàng output hợp lệ.

4.2.3.3 Phân tích số lượng Pass và Dữ liệu biên

Tổng số pass cần thiết được xác định bởi tích số các phân mảnh trên 3 chiều:

$$Total\_Pass = \underbrace{\left\lceil \frac{C}{T_c} \right\rceil}_{N_c} \times \underbrace{\left\lceil \frac{H}{T_h} \right\rceil}_{N_h} \times \underbrace{\left\lceil \frac{N_f}{T_m} \right\rceil}_{N_m} \tag{4.14}$$

Do kích thước layer thường không chia hết cho kích thước Tile, lượng dữ liệu xử lý trong các pass cuối (Boundary Passes) sẽ khác biệt:

- **Phân mảnh theo kênh (C):** Để hoàn thiện một phần output, cần thực hiện $\lceil C/T_c \rceil$ pass tích lũy. Pass cuối cùng xử lý phần dư: $T_h \times W \times (C \pmod{T_c})$.
- **Phân mảnh theo chiều dọc (H):** Cần $\lceil H/T_h \rceil$ bước trượt dọc.
  - Pass cuối cùng (Last Tile) xử lý chiều cao input dư: $H_{rem} = H \pmod{T_h}$.
  - Số hàng Output hợp lệ của Pass cuối cũng tuân theo logic Ping-Pong: Nếu H_{rem} đủ lớn, nó sẽ tạo ra H_{rem} hàng output (do thừa hưởng biên trên).
- **Phân mảnh theo số bộ lọc (N_f):** Pass cuối xử lý số bộ lọc dư: $N_f \pmod{T_m}$.

4.2.3.4 Trường hợp Depthwise Convolution

Đối với Depthwise Convolution ($N_f = C$), quy trình được đơn giản hóa do không có vòng lặp tích lũy kênh ($N_c = 1$):

- Tổng số Pass: $\lceil H/T_h \rceil \times \lceil N_f/T_m \rceil$.
- Kích thước Output Tile T_{ho} vẫn tuân theo quy tắc (3) nêu trên.

4.2.4 Thuật toán Điều phối Pass (Pass Scheduling)

Trình tự thực hiện các Pass phụ thuộc vào loại tích chập (Standard hay Depthwise) để tối ưu hóa việc tái sử dụng dữ liệu biên (như đã phân tích ở mục Dữ liệu dôi ra).

4.2.4.1 Thuật toán cho Standard Convolution

Trong Standard Convolution, một điểm ảnh đầu ra cần tổng hợp dữ liệu từ **tất cả** các khối kênh đầu vào (N_c). Do đó, ta cần vòng lặp tích lũy (Reduction Loop) chạy qua N_c trước khi chuyển sang khối chiều cao khác.

Algorithm 3: Lịch trình Pass cho Standard Convolution

Input: N_m (Output Blocks), N_h (Height Blocks), N_c (Input Blocks)

for $m = 0$ ***to*** $N_m - 1$ **do**

  1. Load Weights for Output Block m (Weight Stationary)

  for $h = 0$ ***to*** $N_h - 1$ **do**

    for $c = 0$ ***to*** $N_c - 1$ **do**

      Pass (m, h, c)**:**

      - Nạp Input Block (c, h) kích thước $T_c \times T_h$

      - Tính toán với Weight Block (m, c)

      - Cộng dồn kết quả vào Buffer hiện tại (A hoặc B)

    end

    2. Xử lý biên & Ghi Output:

    - Sau khi cộng đủ N_c passes: Output Block (m, h) đã hoàn tất (Valid).

    - Ghi phần Valid xuống DRAM.

    - Hoán đổi Ping-Pong Buffer (để dùng phần Dôi ra cho $h+1$).

  end

end

4.2.4.2 Thuật toán cho Depthwise Convolution

Trong Depthwise Convolution, kênh Input thứ i chỉ tính toán với kênh Filter thứ i. Do đó $N_c = N_m$ (số nhóm kênh Input bằng số nhóm kênh Output) và không có sự cộng dồn chéo giữa các nhóm. Vòng lặp tích lũy biến mất.

Algorithm 4: Lịch trình Pass cho Depthwise Convolution

Input: N_m (Channel Groups), N_h (Height Blocks)

for $g = 0$ ***to*** $N_m - 1$ **do**

1. Load Weights for Group g

for $h = 0$ ***to*** $N_h - 1$ **do**

Pass (g, h)**:**

- Nạp Input Block (g, h) kích thước $T_c \times T_h$

- Tính toán với Weight Block g

- Tạo ra ngay kết quả Output Block (g, h) (không cần cộng dồn)

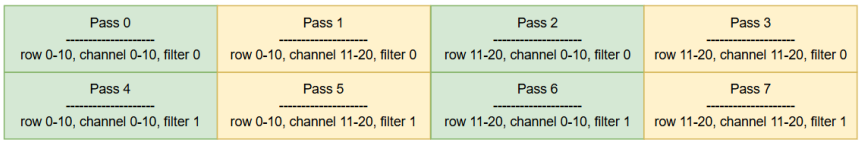
2. Xử lý biên & Ghi Output:

- Ghi ngay phần Valid xuống DRAM.

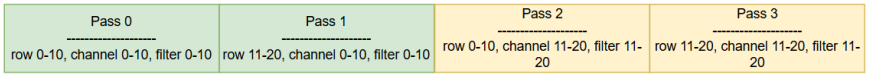
- Hoán đổi Ping-Pong Buffer (lưu phần Dôi ra cho $h+1$).

end

end



(a) Standard Convolution $(H = 21, M = 2)$



(b) Depthwise Convolution $(H = 21, M = 21)$

Hình 4.1: Minh họa chiến lược phân chia Pass cho hai loại tích chập với $T_h = 11$. (a) Standard Convolution chia thành 2 phần theo chiều dọc và tích lũy theo chiều sâu. (b) Depthwise Convolution xử lý độc lập từng nhóm kênh và chia 2 phần theo chiều dọc.

4.2.5 Phân tích vấn đề tại biên và Dữ liệu dôi ra

Mặc dù chiến lược phân mảnh dữ liệu được áp dụng trên cả chiều kênh và chiều không gian, tác động của chúng lên luồng dữ liệu là khác nhau.

- Việc chia nhỏ chiều kênh (C, M) dẫn đến bài toán tích lũy tổng riêng

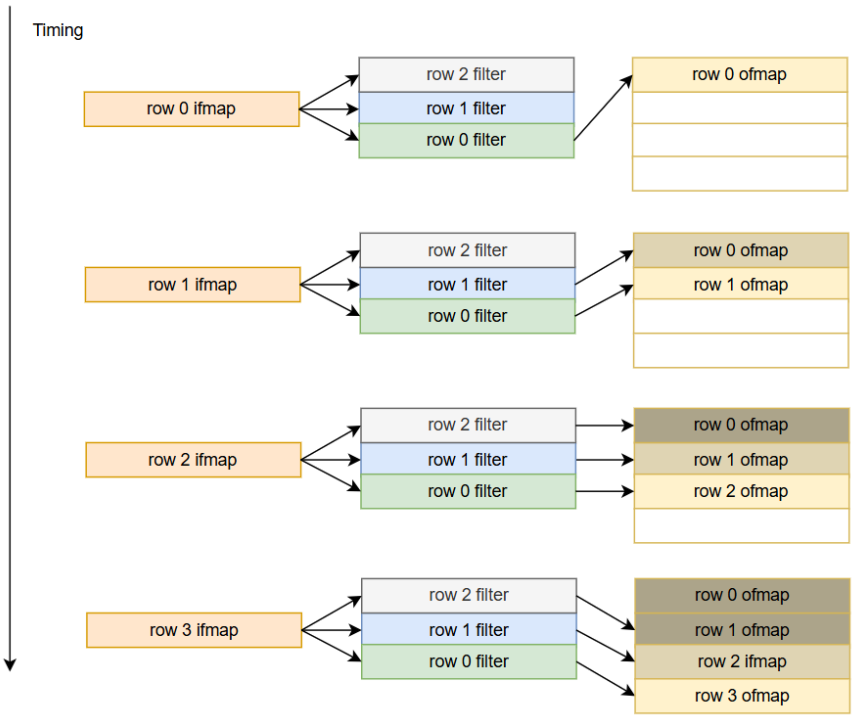
(Partial Sum Accumulation).

- Việc chia nhỏ chiều không gian (H) dẫn đến bài toán thiếu dữ liệu vùng lân cận cho cửa sổ trượt (Sliding Window Boundary).

Mục này tập trung phân tích vấn đề tại biên không gian, nguyên nhân chính dẫn đến sự cần thiết của cơ chế Ping-Pong Buffer đặc thù.

4.2.5.1 Cơ sở hình thành Dữ liệu dôi ra

Khi bộ lọc trượt theo chiều dọc, tại các hàng cuối cùng của một Tile không gian (gọi là Tile H_k), bộ lọc cần dữ liệu của các hàng tiếp theo (thuộc Tile H_{k+1}) để hoàn thành phép tính.



Hình 4.2: Sơ đồ minh họa quá trình tính toán tích chập và sự hình thành dữ liệu dôi ra (Residual Data) trong một pass với tile đầu vào $T_h = 4$ và bộ lọc kích thước 3×3 trong trường hợp số kênh của ifmap feature là 1.

Xét ví dụ cụ thể với Tile đầu vào có kích thước chiều cao $T_h = 4$ (các hàng 0, 1, 2, 3) và bộ lọc kích thước 3×3. Khi thực hiện tích chập:

- **Hàng 0, 1:** Có đầy đủ dữ liệu lân cận (trong phạm vi Tile hiện tại)

→ Tạo ra kết quả hoàn chỉnh (*Valid Output*).

- **Hàng 2:** Cần dữ liệu hàng [2, 3, **4**]. Thiếu hàng 4 (thuộc Tile H_{k+1}) → Kết quả chưa hoàn thiện.
- **Hàng 3:** Cần dữ liệu hàng [3, **4**, **5**]. Thiếu hàng 4, 5 (thuộc Tile H_{k+1}) → Kết quả chưa hoàn thiện.

Các kết quả tại hàng 2 và 3 được gọi là **Dữ liệu dôi ra (Residual Data)**. Số lượng hàng dôi ra luôn là $R-1$. Để đảm bảo tính đúng đắn mà không cần nạp lại phần dữ liệu Input [2, 3] khi xử lý Tile H_{k+1}, hệ thống cần lưu trữ các giá trị dôi ra này và cộng dồn chúng với kết quả tính toán từ Tile tiếp theo.

4.2.6 Cơ chế Ping-Pong Buffer và Logic xử lý hàng hợp lệ

Để xử lý dữ liệu dôi ra (Residual Data) tại biên dưới của mỗi tile mà không cần nạp lại Input, hệ thống sử dụng hai bộ đệm đầu ra $Buffer_A$ và $Buffer_B$ hoạt động luân phiên.
Điểm quan trọng trong chiến lược này là số lượng hàng đầu ra hợp lệ (Valid Rows) sẽ khác nhau giữa Tile đầu tiên và các Tile tiếp theo:

- **Tile đầu tiên ($h = 0$):** Do không có dữ liệu tích lũy từ phía trên, bộ lọc trượt qua T_h hàng đầu vào chỉ tạo ra được $T_h - R + 1$ hàng đầu ra hoàn chỉnh. $R-1$ hàng cuối cùng là dữ liệu dôi ra.
- **Các Tile tiếp theo ($h > 0$):** Nhờ tận dụng $R-1$ hàng dôi ra từ bước trước (đã lưu trong Buffer), hệ thống sẽ hoàn thiện được các hàng này. Tổng số hàng hoàn chỉnh được ghi xuống DRAM trong bước này là đủ T_h hàng.

4.2.7 Thuật toán Điều phối và Xoay vòng bộ nhớ

Thuật toán 5, 6 và 7 mô tả chi tiết quy trình quản lý bộ nhớ và luồng dữ liệu, minh họa rõ sự khác biệt khi xử lý Tile đầu tiên và các Tile sau.

Algorithm 5: Lịch trình Pass cho Standard Conv (Phần 1: Tích lũy)

Input: N_m, N_h, N_c

for $m = 0$ ***to*** $N_m - 1$ **do**

  1. Load Weights...

  for $h = 0$ ***to*** $N_h - 1$ **do**

    for $c = 0$ ***to*** $N_c - 1$ **do**

      Pass (m, h, c)**:** ...

      ... (Code phần tích lũy) ...

    end

    (Xem tiếp xử lý biên ở Thuật toán 6)

  end

end

Algorithm 6: Lịch trình Pass cho Standard Conv (Phần 2: Xử lý biên)

...Tiếp tục từ vòng lặp h của Thuật toán 5

foreach *Tile h đã hoàn tất tích lũy* **do**

  2. Xử lý biên & Ghi Output:

  - Kiểm tra điều kiện biên...

  - Ghi phần Valid xuống DRAM.

  - Hoán đổi Ping-Pong Buffer.

end

Algorithm 7: Lịch trình Pass cho Depthwise Convolution

Input: N_m (Số nhóm kênh), N_h (Số khối dọc)

Output: DRAM (Valid Output Feature Map)

Initialize pointers: $Buf_{curr} \leftarrow A$, $Buf_{next} \leftarrow B$

for $m = 0$ ***to*** $N_m - 1$ **do**

  1. Load Weights for Group m

  for $h = 0$ ***to*** $N_h - 1$ **do**

    Pass (m, h)**:**

    - Nạp Input Tile (m, h) kích thước $T_c \times T_h$

    - Tính toán Depthwise (1-to-1 mapping)

    2. Xử lý biên & Quản lý Ping-Pong:

    if $h == 0$ **then**

      // Trường hợp Tile đầu tiên

      - Lưu $T_h - R + 1$ hàng Output hợp lệ vào Buf_{curr}

      - Lưu phần dư ($R - 1$ hàng cuối) vào Buf_{next}

      - **Drain:** Ghi Buf_{curr} xuống DRAM

    else

      // Các Tile tiếp theo (Tận dụng Residual)

      - Hoàn thiện $R - 1$ hàng biên (từ Buf_{curr} cũ)

      - Tạo thêm các hàng thân mới (đủ T_h hàng)

      - Lưu phần dư mới vào Buf_{next}

      - **Drain:** Ghi toàn bộ T_h hàng hợp lệ xuống DRAM

    end

    3. Chuẩn bị cho tile tiếp theo:

    - Clear Buf_{curr}

    - **Swap pointers:** $Buf_{curr} \leftrightarrow Buf_{next}$

  end

end

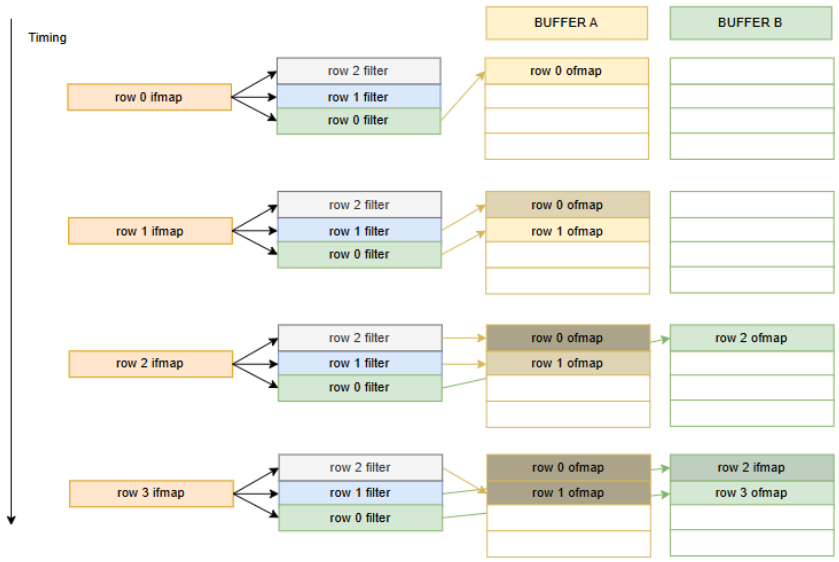
Giải thích cơ chế:

1. Tại vòng lặp h, Buf_{curr} đóng vai trò tích lũy kết quả chính, còn Buf_{next} đóng vai trò hứng các giá trị dôi ra (Residual) cho tương lai.

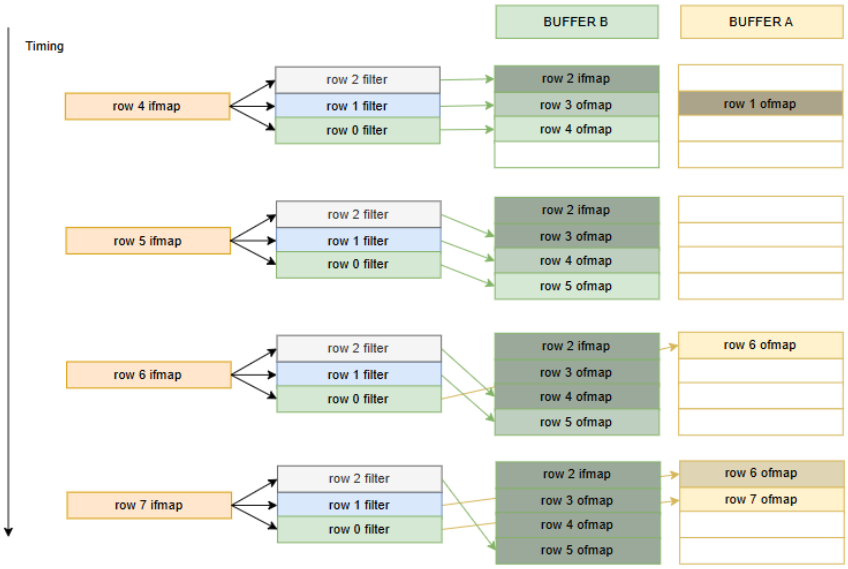
2. Khi $h = 0$: Chúng ta ghi các hàng dôi ra vào đầu Buf_{next}.

3. Khi chuyển sang $h = 1$: Ta thực hiện Swap. Lúc này Buf_{curr} (vốn là Buf_{next} cũ) đã có sẵn dữ liệu dôi ra ở các hàng đầu. Việc tính toán tiếp tục cộng dồn vào đó, biến chúng thành kết quả hoàn chỉnh (Valid).

4. Quá trình ghi xuống DRAM (Drain) ở $h > 0$ sẽ ghi toàn bộ T_h hàng, bao gồm cả những hàng vừa được hoàn thiện từ dữ liệu dôi ra.



(a) Giai đoạn 1 (Xử lý Tile H_k): Buffer A tích lũy kết quả Valid, Buffer B lưu trữ dữ liệu Dôi ra (Residual).



(b) Giai đoạn 2 (Xử lý Tile H_{k+1}): Buffer B hoàn thiện kết quả biên (từ Residual cũ), Buffer A lưu trữ dữ liệu Dôi ra mới.

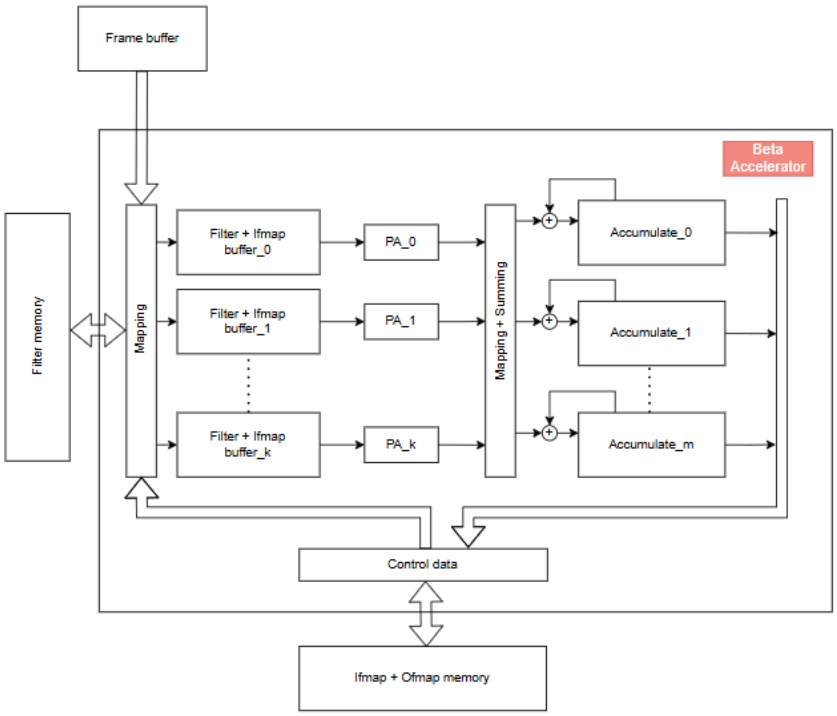
Hình 4.3: Sơ đồ luồng dữ liệu minh họa cơ chế Ping-Pong Buffer dùng để quản lý vùng dữ liệu dôi ra (Residual Data). Hệ thống luân phiên vai trò của Buffer A và B để đảm bảo tính liên tục của phép tính biên mà không cần nạp lại dữ liệu đầu vào.

4.3 Thiết kế kiến trúc vi mô (Micro-architecture)

Dựa trên chiến lược dòng dữ liệu đã phân tích, nhóm đề xuất kiến trúc phần cứng chuyên dụng mang tên **Beta Accelerator**. Kiến trúc này được thiết kế để tối ưu hóa khả năng tính toán song song ở mức bộ lọc (Filter parallelism) và mức kênh (Channel parallelism). Đặc biệt, hệ thống sử dụng kiến trúc bộ nhớ tách biệt (Separate Memory Architecture) cho Trọng số và Dữ liệu để tối đa hóa băng thông, đồng thời hỗ trợ cơ chế quản lý bộ nhớ Ping-Pong để che giấu độ trễ truy cập.

4.3.1 Sơ đồ khối tổng quát hệ thống

Sơ đồ tổng thể của Beta Accelerator được trình bày trong Hình 4.4. Hệ thống bao gồm các khối chức năng chính sau:



Hình 4.4: Sơ đồ khối tổng quát kiến trúc Beta Accelerator với bus dữ liệu tách biệt

- **Khối Control Data (Controller):** Đóng vai trò bộ điều khiển trung

tâm và quản lý giao tiếp với bộ nhớ ngoài.

  - Hệ thống được thiết kế với hai giao tiếp bộ nhớ độc lập: **Weight Memory Interface** (dành cho trọng số) và **Activation Memory Interface** (dành cho IFM và OFM).
  - Do IFM (Input) và OFM (Output) được lưu trữ trên cùng một không gian bộ nhớ Dữ liệu (Activation Memory), khối này chịu trách nhiệm điều phối (arbitration) tài nguyên bus dữ liệu này, quyết định thời điểm thực hiện nạp dữ liệu đầu vào (Load IFM) hoặc ghi kết quả đầu ra (Store OFM) để tránh xung đột. Trong khi đó, luồng nạp Weight có thể diễn ra song song nhờ đường bus riêng.

- **Khối Mapping (Dispatcher):** Chịu trách nhiệm phân phối dữ liệu từ hai nguồn bus riêng biệt (Weight Bus và Activation Bus) tới các bộ nhớ đệm cục bộ của từng đơn vị tính toán, đảm bảo tính đồng bộ giữa dữ liệu và trọng số.

- **Hệ thống Bộ đệm (Filter + Ifmap Buffer):** Được tổ chức theo cơ chế **Ping-Pong Buffer** (Double Buffering) để cho phép nạp dữ liệu cho Pass $k+1$ trong khi Pass k đang được tính toán.

  - **IFM Buffer:** Lưu trữ một tile IFM kích thước $T_h \times W$ của 1 kênh, được nạp từ Activation Memory.
  - **Weight Buffer:** Lưu trữ bộ trọng số kích thước $S \times R \times T_m$, được nạp từ Weight Memory riêng biệt.

- **Mảng xử lý (Process Array - PA):** Là trái tim tính toán của hệ thống, bao gồm T_c khối PA hoạt động song song. Mỗi khối PA phụ trách xử lý 1 kênh đầu vào (Input Channel) và T_m bộ lọc tương ứng.

- **Khối Tổng hợp (Reduction Unit - Mapping + Summary):** Thực hiện chức năng cộng dồn (Reduction) kết quả từ T_c khối PA.

Do tích chập là phép tổng trọng số qua các kênh, khối này sẽ cộng giá trị Partial Sum từ các kênh IFM khác nhau để tạo ra T_m giá trị OFM bán hoàn chỉnh.

- **Khối Tích lũy (Accumulator):** Sử dụng bộ nhớ Ping-Pong để lưu trữ và cộng dồn kết quả qua các Pass (theo chiều sâu kênh C). Khi một điểm ảnh OFM đã được tích lũy đủ số kênh cần thiết, nó sẽ được gửi đi thông qua Activation Bus để ghi vào vùng nhớ Output (OFM) và vị trí nhớ đó sẽ được reset về 0.

4.3.2 Tổ chức Mảng tính toán (Processing Hierarchy)

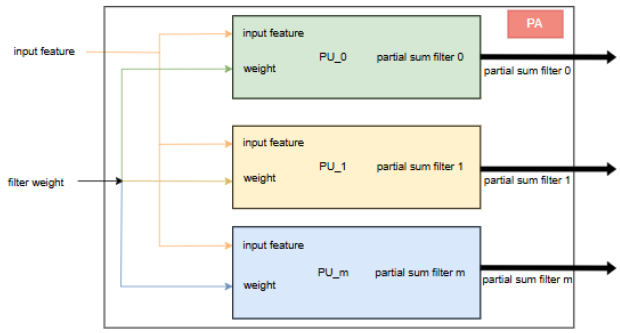
Kiến trúc tính toán được tổ chức theo mô hình phân cấp gồm 3 tầng: Process Array (PA), Process Unit (PU) và Process Element (PE).

4.3.2.1 Mảng xử lý (Process Array - PA)

Khối PA (Hình 4.5) được thiết kế để khai thác tính song song mức kênh đầu ra (Output Channel Parallelism).

- Mỗi PA chịu trách nhiệm tính toán cho 1 kênh đầu vào (Input Channel) duy nhất nhưng tạo ra kết quả cho T_m bộ lọc (Filters) khác nhau.

- **Luồng dữ liệu:** Trọng số đầu vào (Input Filter Weights) từ Weight Memory được rẽ nhánh (demultiplex) tới các PU cụ thể. Ngược lại, dữ liệu IFM từ Activation Memory được quảng bá (broadcast) dùng chung cho tất cả các PU trong cùng một PA, giúp tối ưu hóa việc tái sử dụng dữ liệu IFM.

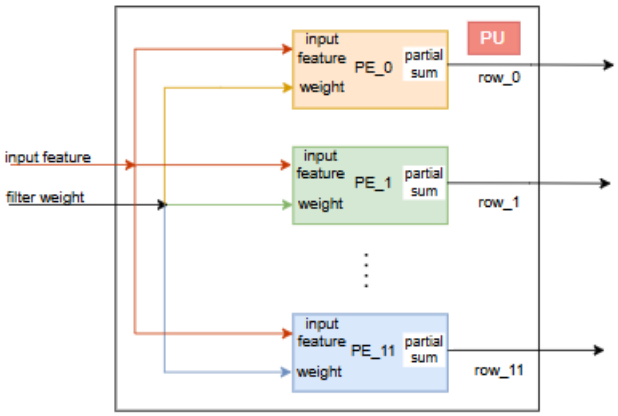


Hình 4.5: Kiến trúc bên trong khối Process Array (PA)

4.3.2.2 Đơn vị xử lý (Process Unit - PU)

Mỗi PU (Hình 4.6) bao gồm 11 phần tử xử lý (PE) hoạt động song song, tương ứng với khả năng hỗ trợ kích thước bộ lọc tối đa là 11×11 (chiều cao $R = 11$).

- Mỗi PE trong PU chịu trách nhiệm tính toán tích chập cho **1 hàng** của bộ lọc (Filter Row).
- Các PE hoạt động đồng bộ. Sau mỗi khoảng thời gian ΔT chu kỳ, PU sẽ tạo ra một cột kết quả gồm R giá trị tương ứng với R hàng của bộ lọc.

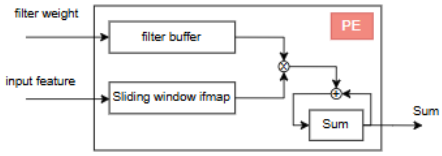


Hình 4.6: Kiến trúc khối Process Unit (PU) với các PE hoạt động song song

4.3.2.3 Phần tử xử lý (Process Element - PE)

PE là đơn vị tính toán cơ sở nhỏ nhất (Hình 4.7), thực hiện phép tính nhân-cộng (MAC).

- **Filter Buffer:** Lưu trữ S giá trị trọng số của một hàng filter $(1 \times S)$. Buffer này hoạt động theo chế độ Weight Stationary, giữ giá trị không đổi trong suốt quá trình thực hiện 1 Pass.
- **Sliding Window Register:** Chứa S giá trị IFM $(1 \times S)$. Đây là thanh ghi dịch, sau mỗi ΔT chu kỳ, dữ liệu sẽ dịch đi 1 vị trí (stride $= 1$) để thực hiện phép trượt cửa sổ.
- Vì mỗi PE chứa 1 bộ nhân và 1 bộ cộng, để tính tích chập 1 hàng kích thước S, hệ thống cần $\Delta T = S$ chu kỳ.



Hình 4.7: Cấu trúc bên trong một Process Element (PE)

4.3.3 Đánh giá thời gian thực thi (Performance Estimation)

Thời gian thực thi của hệ thống phụ thuộc vào loại lớp tích chập (Standard hay Depthwise) do sự khác biệt trong chiến lược luồng dữ liệu.

4.3.3.1 Thời gian xử lý một Pass cơ sở (T_{pass})

Dựa trên kiến trúc Pipeline của các Process Element (PE), thời gian để hoàn thành tính toán cho một tile có chiều cao T_h và độ rộng OFM W_{out} được xác định bởi:

$$T_{pass} = \Big[(W_{out} - 1) \times (S + U - 1) + S\Big] \times T_h \tag{4.15}$$

Trong đó:

- S: Kích thước bộ lọc (Filter width).
- U: Bước trượt (Stride).
- W_{out}: Chiều rộng của OFM.
- $(S+U-1)$: Số chu kỳ trung bình để tính một điểm ảnh tiếp theo nhờ tối ưu hóa Pipeline (khi $U = 1$, thời gian này là S).

4.3.3.2 Tổng thời gian thực thi (T_{total})

Trường hợp 1: Standard Convolution

Với tích chập tiêu chuẩn, mỗi điểm ảnh đầu ra là tổng hợp của tất cả C kênh đầu vào. Hệ thống phải thực hiện vòng lặp tích lũy qua các khối kênh T_c.

$$T_{total\_std} = \underbrace{\left\lceil \frac{N_f}{T_m} \right\rceil}_{\text{Output Blocks}} \times \underbrace{\left\lceil \frac{C}{T_c} \right\rceil}_{\text{Input Blocks}} \times \underbrace{\left\lceil \frac{H}{T_h} \right\rceil}_{\text{Height Blocks}} \times T_{pass} \tag{4.16}$$

Trường hợp 2: Depthwise Convolution

Với tích chập chiều sâu, các kênh hoạt động độc lập ($N_f = C$). Hệ thống không cần thực hiện vòng lặp tích lũy kênh đầu vào ($\lceil C/T_c \rceil$ bị loại bỏ). Các nhóm kênh được xử lý song song dựa trên khả năng của phần cứng (T_m).

$$T_{total\_dw} = \underbrace{\left\lceil \frac{N_f}{T_m} \right\rceil}_{\text{Channel Groups}} \times \underbrace{\left\lceil \frac{H}{T_h} \right\rceil}_{\text{Height Blocks}} \times T_{pass} \tag{4.17}$$

Nhận xét: So với Standard Convolution, Depthwise Convolution giảm được hệ số $\lceil C/T_c \rceil$ lần số lượng tính toán, giúp tăng tốc độ xử lý đáng kể

đối với các mạng nhẹ (Lightweight CNNs) như MobileNet.

4.3.4 Chiến lược Che giấu độ trễ và Mô hình hiệu năng toàn hệ thống

Để tối ưu hóa hiệu năng, Beta Accelerator áp dụng kỹ thuật **Che giấu độ trễ (Latency Hiding)** thông qua cơ chế Ping-Pong Buffer. Mục tiêu là thực hiện song song quá trình tính toán (Computation) và quá trình truyền tải dữ liệu (Data Transfer).

4.3.4.1 Cơ chế hoạt động với Bộ nhớ tách biệt

Nhờ việc tách biệt bộ nhớ Weight và bộ nhớ Activation (IFM/OFM), hệ thống có lợi thế lớn về băng thông:

- **Weight Loading:** Quá trình nạp Weight diễn ra trên bus riêng, do đó hoàn toàn không xung đột với việc nạp IFM hay ghi OFM. Việc nạp Weight cho Pass $i+1$ luôn được thực hiện song song và che giấu bởi thời gian tính toán Pass i.
- **Activation Loading/Storing:** IFM và OFM chia sẻ băng thông của Activation Memory. Do đó, vẫn tồn tại sự tranh chấp tài nguyên giữa việc nạp IFM cho Pass tiếp theo và ghi OFM của Pass trước đó.

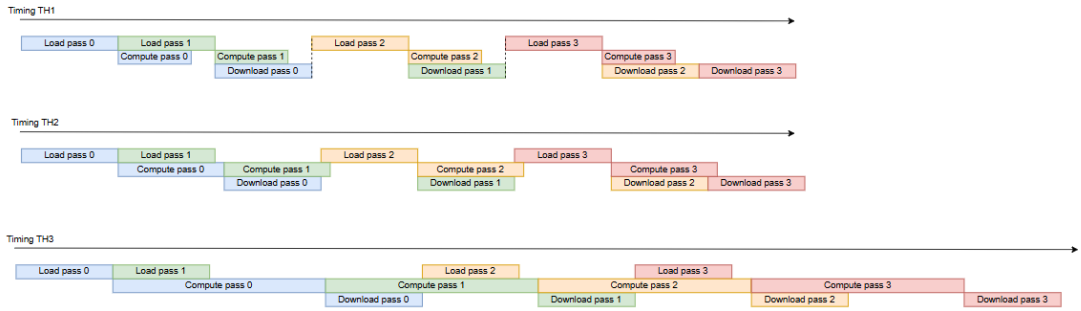
Quy trình hoạt động theo nguyên lý "gối đầu":

- Trong khi lõi tính toán đang xử lý Pass i, bộ điều khiển DMA đồng thời nạp IFM và Weight cho Pass $i+1$ vào nửa còn lại của Buffer.
- Đồng thời, kết quả OFM của Pass $i-1$ (nếu đã hoàn tất) được ghi trả về Activation Memory.

4.3.4.2 Các kịch bản hiệu năng (Performance Scenarios)

Do Weight Bus tách biệt, độ trễ nạp Weight thường được che giấu hoàn toàn. Điểm nghẽn (bottleneck) chủ yếu nằm ở sự cân bằng giữa tính toán và truy cập Activation Memory (IFM/OFM).

Gọi T_{load_act} là thời gian nạp IFM, T_{store_act} là thời gian ghi OFM, và T_{comp} là thời gian tính toán 1 Pass.



Hình 4.8: Biểu đồ thời gian thực thi trong 3 trường hợp cân bằng tải

Trường hợp 1: Activation Memory Bound (Nghẽn băng thông dữ liệu)

Xảy ra khi tổng thời gian truy cập Activation Memory lớn hơn thời gian tính toán ($T_{load\_act} + T_{store\_act} \geq T_{comp}$). Lõi tính toán phải chờ bus dữ liệu hoàn thành tác vụ.

- **Đối với Standard Convolution:** Cần nạp lại IFM nhiều lần cho các nhóm Filter khác nhau:

$$T_{total} \approx \left[\left(H \times W \times C \times \left\lceil \frac{N_f}{T_m} \right\rceil \right) + \left(H_{out} \times W_{out} \times N_f \right) \right] \times b_{act} \quad (4.18)$$

  Trong đó b_{act} là số chu kỳ để truyền 1 đơn vị dữ liệu trên Activation Bus.

- **Đối với Depthwise Convolution:** IFM chỉ cần nạp 1 lần duy nhất

do ánh xạ 1-1 giữa kênh Input và Filter:

$$T_{total} \approx \Big[(H \times W \times C) + (H_{out} \times W_{out} \times C)\Big] \times b_{act} \tag{4.19}$$

Trường hợp 2: Compute Bound (Nghẽn tính toán)

Xảy ra khi thời gian tính toán lớn hơn tổng thời gian nạp và ghi dữ liệu ($T_{comp} > T_{load\_act} + T_{store\_act}$). Lúc này, toàn bộ thời gian truyền tải dữ liệu (Activation và Weight) được che giấu hoàn toàn.

Công thức tổng quát:

$$T_{total} = T_{load\_first\_pass} + \sum_{all\_passes} T_{comp} + T_{store\_residual} \tag{4.20}$$

Việc tách biệt bộ nhớ Weight giúp giảm đáng kể khả năng rơi vào trạng thái Memory Bound so với kiến trúc Bus chung, đặc biệt là ở các lớp Fully Connected hoặc Convolution có số lượng tham số lớn.

4.3.4.3 Tổng thời gian toàn mạng (Model Latency)

Thời gian thực thi của toàn bộ mô hình (Model) bao gồm N lớp tích chập là tổng thời gian của từng lớp:

$$T_{model} = \sum_{i=1}^{N} T_{total}^{(i)} \tag{4.21}$$

4.4 Tối ưu hóa Tham số và Cơ chế Sinh mã cấu hình

Sau khi xây dựng mô hình ước lượng hiệu năng, bước tiếp theo là xác định bộ tham số cấu hình tối ưu cho từng lớp mạng và chuyển đổi chúng thành chuỗi lệnh điều khiển (Instruction Stream) mà Controller có thể thực thi.

4.4.1 Bài toán Tối ưu hóa Không gian thiết kế

Với mỗi lớp tích chập thứ i, mục tiêu là tìm ra bộ ba tham số phân mảnh $\mathbf{S}_i = \{T_{h,i}, T_{c,i}, T_{m,i}\}$ sao cho thời gian thực thi tổng thể (T_{total}) là nhỏ nhất.

4.4.1.1 Các ràng buộc phần cứng (Hardware Constraints)

Bộ tham số được chọn bắt buộc phải thỏa mãn các giới hạn vật lý của FPGA. Các ràng buộc chính bao gồm:

1. **Dung lượng bộ nhớ on-chip (BRAM):** Tổng kích thước của các Tile (bao gồm cả cơ chế Ping-Pong nhân hệ số 2) không được vượt quá dung lượng BRAM (4096 bytes) dành riêng cho từng loại dữ liệu.

$$\left\lceil (T_c \times T_h \times W/2048) \right\rceil \leq BRAM_{IFM\_MAX} \tag{4.22}$$

$$\left\lceil (T_m \times T_c \times R \times S/2048) \right\rceil \leq BRAM_{WGT\_MAX} \tag{4.23}$$

2. **Tài nguyên tính toán:** Số lượng bộ lọc tính toán song song (T_m) không được vượt quá số lượng mảng xử lý (PA) vật lý có trên chip.

$$1 \leq T_m \leq N_{PA\_MAX} \tag{4.24}$$

3. **Tính hợp lệ:** Kích thước Tile không được lớn hơn kích thước gốc của Feature Map.

$$1 \leq T_h \leq H; \quad 1 \leq T_c \leq C \tag{4.25}$$

4.4.2 Chiến lược Tìm kiếm và Sinh mã (Search & Generate)

Do không gian tìm kiếm tham số cho một lớp là hữu hạn, chúng tôi phát triển một công cụ phần mềm (Software Tool) chạy trên máy tính chủ (Host

PC) để thực hiện quy trình sau:

- **Bước 1 - Quét tham số (Exhaustive Search):** Thuật toán duyệt qua toàn bộ các tổ hợp (T_h, T_c, T_m) khả dĩ.
- **Bước 2 - Kiểm tra ràng buộc:** Loại bỏ ngay các tổ hợp vi phạm ràng buộc BRAM hoặc tài nguyên tính toán nêu trên.
- **Bước 3 - Đánh giá hiệu năng:** Với các tổ hợp hợp lệ, phần mềm áp dụng các công thức mô hình hiệu năng (Mục 4.3.4) để tính T_{total} dự kiến. Cấu hình cho T_{total} nhỏ nhất sẽ được chọn.
- **Bước 4 - Đóng gói (Packing):** Các tham số tối ưu được đóng gói thành một chuỗi bit nhị phân (Binary Descriptor) để gửi xuống phần cứng.

4.4.3 Cấu trúc Lệnh cấu hình (Layer Descriptor)

Để Controller phần cứng hiểu và vận hành theo tham số đã tìm được, chúng tôi định nghĩa một cấu trúc dữ liệu điều khiển (Descriptor). Mỗi lớp mạng tương ứng với một Descriptor được lưu trong bộ nhớ lệnh.

Bảng 4.2: Cấu trúc dữ liệu cấu hình cho một Layer

Offset	**Trường thông tin**	**Mô tả chức năng**
0x00	Layer Info	Chứa H, W, C, N_f (Kích thước gốc)
0x04	Kernel Info	Chứa R, S, P, Str (Bộ lọc, Padding, Stride)
0x08	**Tiling Config**	**Chứa T_h, T_c, T_m (Tham số tối ưu)**
0x0C	IFM Base Addr	Địa chỉ bắt đầu của Input Feature Map
0x10	WGT Base Addr	Địa chỉ bắt đầu của Weights
0x14	OFM Base Addr	Địa chỉ bắt đầu của Output Feature Map
0x18	Control Flags	Loại lớp (Std/Depthwise), Activation (ReLU)...

Khối **Control Data** (Mục 4.3) sẽ đọc Descriptor này, giải mã và cài đặt các giá trị T_h, T_c, T_m vào các thanh ghi đếm của máy trạng thái (FSM). Nhờ

đó, phần cứng có thể linh hoạt xử lý nhiều kích thước mạng khác nhau mà không cần thiết kế lại mạch RTL.

Chương 5

Hiện thực SoC và Tích hợp hệ thống

Chương này trình bày chi tiết quá trình hiện thực hệ thống SoC trên nền tảng FPGA. Nội dung bao gồm việc tích hợp lõi vi xử lý RISC-V, thiết kế các khối ngoại vi (Camera, DMA, UART), xây dựng hệ thống Bus kết nối và phát triển lớp phần mềm điều khiển (Firmware) để vận hành toàn bộ hệ thống.

5.1 Môi trường và Công cụ hiện thực

5.2 Tích hợp Lõi RISC-V và Hệ thống Bus

5.3 Thiết kế và Tích hợp các khối Ngoại vi

5.4 Phát triển Firmware và Trình điều khiển (Driver)

5.5 Quy trình Tổng hợp và Triển khai trên FPGA

Chương 6

Đánh giá và Thảo luận kết quả

Chương này trình bày các kết quả thực nghiệm thu được từ mô hình ước lượng hiệu năng của kiến trúc đề xuất. Các đánh giá tập trung vào hiệu quả của thuật toán tối ưu tham số (Codegen) và khả năng xử lý của phần cứng đối với các lớp mô hình đại diện: AlexNet, VGG-16 và MobileNetV1. Đồng thời, chúng tôi thực hiện so sánh với kiến trúc Eyeriss để làm rõ ưu nhược điểm của giải pháp.

6.1 Môi trường và Phương pháp thực nghiệm

Do giới hạn về thời gian tổng hợp phần cứng (Synthesis) và tài nguyên FPGA thực tế, trong phạm vi đồ án này, nhóm thực hiện đánh giá hiệu năng kiến trúc thông qua mô hình ước lượng (Analytical Estimator) được xây dựng bằng C++. Mô hình này hiện thực các công thức mà chương 4 đã mô tả.

Các tham số cấu hình cho mô phỏng được thiết lập như sau:

- **Tần số hoạt động (Frequency):** 200 MHz.

- **Số lượng đơn vị xử lý (PEs):** 165 PEs (Cấu hình tiêu chuẩn, tương đương mảng 11×15).
- **Phạm vi đánh giá:** Chỉ tập trung đo đạc thời gian thực thi của các lớp Tích chập (Convolutional Layers).
- **Chiến lược xử lý:** Batch Size = 1 (Tối ưu độ trễ cho xử lý từng ảnh đơn lẻ).
- **Cấu hình Bộ nhớ:** Giả lập kiến trúc bộ nhớ tách biệt (Separate Off-chip Memory), trong đó Trọng số (Weights) và Dữ liệu (Activations) được truy xuất trên các kênh độc lập.

6.2 Hiệu quả của Thuật toán Tối ưu trên AlexNet

Để chứng minh tính hiệu quả của công cụ sinh mã, chúng tôi so sánh thời gian thực thi giữa việc chọn tham số ngẫu nhiên và tham số tối ưu. Bảng 6.1 trình bày chi tiết cấu hình và thời gian thực thi từng lớp của mạng AlexNet sau khi tối ưu.

Bảng 6.1: Chi tiết hiệu năng từng lớp của AlexNet (Mô phỏng với 165 PEs)

Layer	Filter Size	Ifmap Size	Ofmap Size	Tiling Parameters			Cycles	Latency (ms)
				T_c	T_m	T_h		
Conv1	$11 \times 11 \times 3 \times 96$	$227 \times 227 \times 3$	$55 \times 55 \times 96$	1	15	11	3,611,128	18.06
Conv2	$5 \times 5 \times 48 \times 256$	$27 \times 27 \times 48$	$27 \times 27 \times 256$	1	15	5	3,172,860	15.86
Conv3	$3 \times 3 \times 256 \times 384$	$13 \times 13 \times 256$	$13 \times 13 \times 384$	1	15	3	1,189,760	5.95
Conv4	$3 \times 3 \times 192 \times 384$	$13 \times 13 \times 192$	$13 \times 13 \times 384$	1	15	3	1,752,192	8.76
Conv5	$3 \times 3 \times 192 \times 256$	$13 \times 13 \times 192$	$13 \times 13 \times 256$	1	15	3	1,211,392	6.06
Total	-	-	-	-	-	-	**10,937,332**	**54.69**

Nhận xét: Thuật toán đã tự động đẩy tham số T_m lên 15 ở các lớp sau để tận dụng tối đa số lượng PE, giúp tổng thời gian thực thi giảm xuống còn **54.69 ms**.

6.3 Đánh giá chi tiết trên MobileNetV1

Chúng tôi tiến hành chạy mô phỏng trên MobileNetV1 để đánh giá khả năng hỗ trợ các mô hình sử dụng *Depthwise Separable Convolution*. Bảng 6.2 phân tích chi tiết hiệu năng và độ tận dụng tài nguyên (PE Utilization) qua từng lớp.

Bảng 6.2: Chi tiết hiệu năng MobileNetV1 (165 PEs) - Phân tách Depthwise/Pointwise

Layer	Type	Stride	Filter	Input Size	PEs Used	Util. (%)	Latency (ms)	State
L0	Conv Std	2	$3 \times 3 \times 3 \times 32$	224×224	99	60.0	4.26	Mem
L1	**Depthwise**	1	$3 \times 3 \times 32$	112×112	**9**	**5.4**	4.01	Mem
L2	Pointwise	1	$1 \times 1 \times 32 \times 64$	112×112	52	31.5	14.05	Mem
L3	**Depthwise**	2	$3 \times 3 \times 64$	112×112	**9**	**5.4**	5.02	Mem
L4	Pointwise	1	$1 \times 1 \times 64 \times 128$	56×56	60	36.3	11.04	Mem
L5	**Depthwise**	1	$3 \times 3 \times 128$	56×56	9	5.4	4.01	Mem
L6	Pointwise	1	$1 \times 1 \times 128 \times 128$	56×56	60	36.3	20.07	Mem
...	...	...	...	...	...	...	...	...
L25	**Depthwise**	2	$3 \times 3 \times 1024$	14×14	9	5.4	0.50	Mem
L26	Pointwise	1	$1 \times 1 \times 1024 \times 1024$	7×7	60	36.3	17.56	Mem
Total			**Toàn bộ mạng**		**-**	**-**	**229.56**	**-**

Phân tích: Tại các lớp Depthwise (L1, L3,...), hệ thống chỉ sử dụng được 9/165 PEs (5.4%), gây lãng phí tài nguyên lớn. Đây là nguyên nhân chính khiến thời gian suy luận của MobileNet cao hơn AlexNet dù khối lượng tính toán lý thuyết nhỏ hơn.

6.4 Đánh giá khả năng xử lý trên VGG-16

Để kiểm chứng khả năng chịu tải của hệ thống với các mạng nơ-ron sâu và nặng, chúng tôi thực hiện mô phỏng trên VGG-16. Kết quả chi tiết được trình bày trong Bảng 6.3.

Bảng 6.3: Chi tiết hiệu năng từng lớp của VGG-16 (Mô phỏng với 165 PEs)

Layer	**Filter / Channels**	**Map Size**	**Optimized Config**			**Latency (ms)**	**Bottleneck**
			$\mathbf{T_c}$	$\mathbf{T_m}$	$\mathbf{T_h}$		
Conv1_1	$3 \times 3/64$	224×224	1	13	3	19.82	Memory
Conv1_2	$3 \times 3/64$	224×224	1	13	3	96.34	Memory
Conv2_1	$3 \times 3/128$	112×112	1	15	3	44.15	Memory
Conv2_2	$3 \times 3/128$	112×112	1	15	3	80.28	Memory
Conv3_1	$3 \times 3/256$	56×56	1	15	3	40.14	Memory
Conv3_2	$3 \times 3/256$	56×56	1	15	3	76.27	Memory
Conv3_3	$3 \times 3/256$	56×56	1	15	3	76.27	Memory
Conv4_1	$3 \times 3/512$	28×28	1	15	3	37.13	Memory
Conv4_2	$3 \times 3/512$	28×28	1	15	3	72.25	Memory
Conv4_3	$3 \times 3/512$	28×28	1	15	3	72.25	Memory
Conv5_1	$3 \times 3/512$	14×14	1	15	3	18.06	Memory
Conv5_2	$3 \times 3/512$	14×14	1	15	3	18.06	Memory
Conv5_3	$3 \times 3/512$	14×14	1	15	3	18.06	Memory
Total	-	-	-	-	-	**669.10**	-

Nhận xét: Với VGG-16, toàn bộ các lớp đều hoạt động ở trạng thái *Memory Bound*. Tuy nhiên, nhờ chiến lược quản lý bộ nhớ tách biệt (Separate Memory) cho Weight và Activation, hệ thống vẫn duy trì được thời gian thực thi ổn định, không xảy ra hiện tượng tắc nghẽn cục bộ.

6.5 So sánh với các Nghiên cứu liên quan

Để đánh giá vị thế của giải pháp, chúng tôi so sánh kết quả ước lượng với **Eyeriss** [Chen et al., ISSCC 2016].

Bảng 6.4: So sánh hiệu năng xử lý Convolution với Eyeriss

Thông số	**Đề xuất (Ours)**	**Eyeriss [Chen et al.]**
Số lượng PE	165	168
Kiến trúc bộ nhớ	**Separate Off-chip Memory**	Shared DRAM
Chiến lược xử lý	Batch Size = 1	Batch Size = 3-4
AlexNet (Conv Only)	54.69 ms	**28.57 ms** (35 fps)
VGG16 (Conv Only)	**669.10 ms**	1428.57 ms (0.7 fps)

Thảo luận:

- **Đối với VGG16:** Kiến trúc đề xuất nhanh hơn **2.1 lần** nhờ thiết kế tách biệt bộ nhớ trọng số, giúp loại bỏ xung đột băng thông khi xử lý các Feature Map lớn.
- **Đối với AlexNet:** Eyeriss nhanh hơn nhờ xử lý Batch ($N = 4$), giúp tái sử dụng trọng số. Thiết kế của chúng tôi tuy chậm hơn về thông lượng (Throughput) nhưng có lợi thế về độ trễ (Latency) cho các ứng dụng thời gian thực.

Chương 7

Kết luận và Hướng phát triển

Chương này tổng kết các kết quả đạt được trong Giai đoạn 1 và đề ra kế hoạch chi tiết cho việc hiện thực và kiểm thử trong Giai đoạn 2.

7.1 Đánh giá mức độ hoàn thành Giai đoạn 1

7.2 Kế hoạch thực hiện Giai đoạn 2

7.3 Tiến độ dự kiến