Thiết kế và thực hiện khối tính nhân chập 2-D bằng phương pháp HLS

Nguyễn Huy Hoàng*
22022584@vnu.edu.vn

Vương Ngọc Quân*
22022616@vnu.edu.vn

Viện Trí tuệ nhân tạo, Trường Đại học công nghệ - Đại học quốc gia Hà Nội

Phân chia công việc

STT	Họ và tên	Mã SV (ID)	Đóng góp (Công việc)
1	Vương Ngọc Quân	22022616	- Xây dựng mô hình C cho thuật toán, tối ưu hóa thiết kế cơ bản (Solution 1, 2). - Phát triển Testbench và kiểm thử. - Viết các chương: Giới thiệu, Yêu cầu, Thuật toán, Mô hình C và Testbench. - Rà soát và chỉnh sửa toàn bộ báo cáo.
2	Nguyễn Huy Hoàng	22022584	- Thực hiện tối ưu hóa thiết kế (Solution 3). - Tổng hợp HLS và đánh giá hiệu năng. - Triển khai và đánh giá trên phần cứng ZynQ-7000. - Viết các chương: Tối ưu thiết kế, Mô phỏng/Thực thi và Đánh giá, Kết quả triển khai, Kết luận. - Rà soát và chỉnh sửa toàn bộ báo cáo.

Tên/Địa chỉ Repo trên Github: https://github.com/hoanghelloworld/final_conv2d

Mục lục

Tóm tắt nội dung

Dự án này tập trung vào thiết kế, mô phỏng và triển khai mô-đun phần cứng thực hiện phép tích chập 2 chiều (2D Convolution) giữa hình ảnh đầu vào và ma trận kernel sử dụng phương pháp HLS (High-Level Synthesis). Mục tiêu là xây dựng một bộ tăng tốc phần cứng hiệu quả cho phép tính nhân chập 2-D, tối ưu hóa hiệu suất và tài nguyên phần cứng trên nền tảng FPGA ZynQ-7000.

Quá trình thiết kế bắt đầu với việc xây dựng thuật toán và mô hình hóa bằng ngôn ngữ C, kiểm thử qua testbench, và tổng hợp thành mô tả RTL bằng Vivado HLS. Các kỹ thuật tối ưu hóa được áp dụng để cải thiện hiệu suất và giảm thiểu sử dụng tài nguyên. Kết quả thử nghiệm cho thấy thiết kế đạt hiệu suất cao và đáp ứng yêu cầu về tài nguyên, minh chứng cho khả năng áp dụng HLS trong xử lý tín hiệu số và ứng dụng thị giác máy tính.

1 Giới thiệu

Phép tích chập 2 chiều (2D Convolution) là một phép toán cơ bản và quan trọng trong các lĩnh vực xử lý tín hiệu số, học sâu và thị giác máy tính. Phép toán này thường được sử dụng trong các ứng dụng như xử lý hình ảnh, phát hiện đối tượng, phân loại và các mô hình học máy. Trong bối cảnh các hệ thống nhúng và Internet of Things (IoT), yêu cầu xử lý dữ liệu với tốc độ cao nhưng tài nguyên phần cứng có hạn, việc triển khai thuật toán tính tích chập 2D trên phần cứng hiệu quả là một thách thức lớn. Phương pháp này giúp tối ưu hóa hiệu suất tính toán, giảm thiểu thời gian xử lý và giảm tải cho các hệ thống phần cứng.

Dự án này tập trung vào việc thiết kế và thực hiện một khối tính nhân chập 2-D bằng phương pháp tổng hợp mức cao (High-Level Synthesis - HLS), nhằm tối ưu hóa hiệu năng và giảm thời gian tính toán. HLS cho phép thiết kế phần cứng từ các mô tả mức cao như C/C++, giúp rút ngắn thời gian phát triển và dễ dàng tối ưu hóa thiết kế.

Mục tiêu chính của dự án bao gồm:

- Xây dựng thuật toán nhân chập 2-D hiệu quả.
- Phát triển mô hình C cho thuật toán và kiểm chứng bằng testbench.
- Sử dụng Vivado HLS để tổng hợp mô hình C thành thiết kế RTL.
- Tối ưu hóa thiết kế để đạt hiệu năng cao trên nền tảng ZynQ-7000.
- Đánh giá hiệu năng của thiết kế thông qua mô phỏng và thực thi.

2 Yêu cầu

Để đảm bảo dự án được triển khai hiệu quả và đạt được mục tiêu đề ra, các yêu cầu sau đây cần được đáp ứng:

(1) **Xây dựng thuật toán thực hiện tính nhân chập 2-D:** Thuật toán cần được thiết kế rõ ràng, chính xác, và có khả năng mở rộng cho các kích thước ma trận đầu vào và kernel khác nhau.

(2) **Xây dựng bản mô tả mức cao cho thuật toán bằng ngôn ngữ C:** Mô hình C phải phản ánh chính xác thuật toán và dễ dàng chuyển đổi sang thiết kế phần cứng bằng HLS.

(3) **Xây dựng testbench và kiểm chứng thuật toán:** Testbench cần bao gồm các bộ dữ liệu kiểm thử đa dạng để đảm bảo tính chính xác và độ tin cậy của thuật toán.

(4) **Tổng hợp bản mô tả C thành bản mô tả RTL bằng Vivado HLS:** Quá trình tổng hợp cần được thực hiện với các tùy chọn tối ưu phù hợp để đạt được hiệu năng mong muốn.

(5) **Tối ưu hóa thiết kế và thực hiện trên ZynQ-7000:** Thiết kế cần được tối ưu hóa về mặt tài nguyên và hiệu năng, đồng thời phải chạy ổn định trên nền tảng phần cứng ZynQ-7000.

3 Mô tả thuật toán nhân chập 2-D

Phép nhân chập 2-D là quá trình trượt một ma trận kernel (hay còn gọi là bộ lọc) qua một ma trận đầu vào (thường là ảnh) để tạo ra một ma trận đầu ra (feature map). Mỗi phần tử của feature

map được tính bằng cách nhân từng phần tử của kernel với các phần tử tương ứng của vùng ảnh nằm dưới kernel, sau đó cộng tổng các tích lại. **Ký hiệu:**

- I: Ma trận đầu vào (ảnh) có kích thước $H \times W$.
- K: Ma trận kernel có kích thước $K_h \times K_w$.
- O: Ma trận đầu ra (feature map) có kích thước $O_h \times O_w$.

Công thức tính tích chập:
Phần tử tại vị trí (i, j) của ma trận đầu ra O được tính như sau:

$$O(i,j) = \sum_{m=0}^{K_h-1} \sum_{n=0}^{K_w-1} I(i+m, j+n) \cdot K(m,n) \tag{1}$$

trong đó:

- $0 \leq i \leq O_h - 1$
- $0 \leq j \leq O_w - 1$
- $O_h = H - K_h + 1$ (trong trường hợp không padding)
- $O_w = W - K_w + 1$ (trong trường hợp không padding)

Ví dụ
Giả sử ta có ma trận đầu vào I và kernel K như sau:

$$I = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Áp dụng công thức (1), ta tính được phần tử $O(0,0)$ như sau:

$$\begin{aligned} O(0,0) &= I(0,0)K(0,0) + I(0,1)K(0,1) + I(0,2)K(0,2) \\ &\quad + I(1,0)K(1,0) + I(1,1)K(1,1) + I(1,2)K(1,2) \\ &\quad + I(2,0)K(2,0) + I(2,1)K(2,1) + I(2,2)K(2,2) \\ &= 1 \cdot 1 + 2 \cdot 0 + 3 \cdot (-1) + 6 \cdot 1 + 7 \cdot 0 + 8 \cdot (-1) + 11 \cdot 1 + 12 \cdot 0 + 13 \cdot (-1) \\ &= 1 + 0 - 3 + 6 + 0 - 8 + 11 + 0 - 13 \\ &= -6 \end{aligned}$$

Tương tự, ta tính được các phần tử còn lại của ma trận O.
Các biến thể của thuật toán

- **Padding:** Thêm các viền (thường là 0) xung quanh ma trận đầu vào để kiểm soát kích thước đầu ra và giữ lại thông tin ở biên.
- **Stride:** Bước nhảy của kernel khi trượt trên ma trận đầu vào. Stride lớn hơn 1 giúp giảm kích thước đầu ra.

Kết luận: Thuật toán trên được thực hiện hóa bằng ngôn ngữ C trong phần tiếp theo, kết hợp với các phương pháp tối ưu hóa để tăng tốc độ xử lý và giảm tài nguyên phần cứng trong thiết kế cuối cùng.

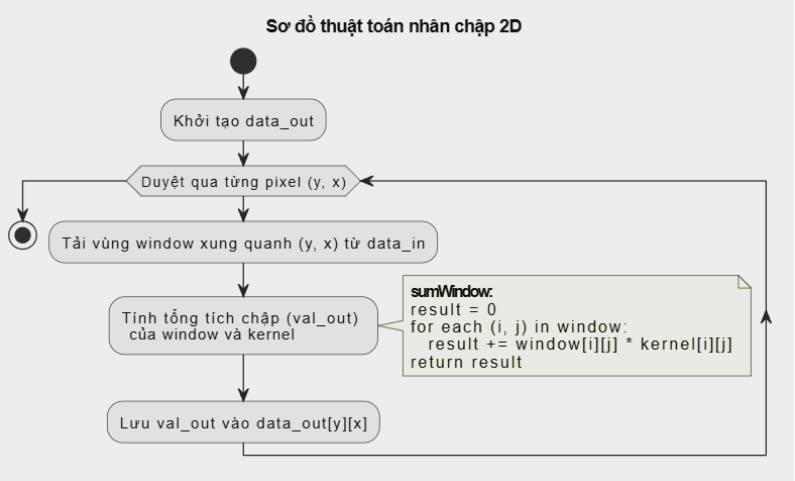
4 Mô hình C và Testbench

4.1 Mô hình C

Trong phần này, chúng tôi triển khai thuật toán nhân chập 2-D đã được mô tả ở Chương 3 bằng ngôn ngữ lập trình C trong file `conv_v1.cpp` và `conv_v2.cpp`. Mục tiêu chính là xây dựng một khối tính toán (module) cho phép toán chập 2D, có thể dễ dàng tổng hợp thành phần cứng sử dụng phương pháp High-Level Synthesis (HLS).

Ý tưởng:

- **Biểu diễn dữ liệu:** Ảnh đầu vào và kernel được biểu diễn dưới dạng mảng hai chiều (2D array) trong C.
- **Hàm tính tổng tích chập:** Một hàm `single_operation` hoặc `sum_window` (ở bản tối ưu) được thiết kế để tính toán giá trị đầu ra của một pixel dựa trên một cửa sổ (window) của ảnh đầu vào và kernel. Hàm này thực hiện vòng lặp kép để duyệt qua các phần tử của cửa sổ, nhân chúng với các phần tử tương ứng của kernel và cộng dồn kết quả.
- **Hàm nhân chập chính:** Hàm `do_convolution` là hàm chính, thực hiện duyệt qua từng pixel của ảnh đầu vào. Tại mỗi pixel, hàm này sẽ:
  1. Trích xuất một cửa sổ con từ ảnh đầu vào, tương ứng với vị trí của kernel.
  2. Gọi hàm `sum_window` để tính toán giá trị pixel đầu ra.
  3. Lưu giá trị đầu ra vào mảng đầu ra.



Hình 1: Mô tả luồng thực hiện của code C (phiên bản cơ sở)

4.2 Testbench

Testbench đóng vai trò quan trọng trong việc kiểm tra tính đúng đắn của mô hình C trước khi tổng hợp thành phần cứng. Chúng tôi xây dựng một testbench trong file `tb_conv.cpp` để kiểm thử các trường hợp khác nhau và đảm bảo rằng kết quả đầu ra của mô hình C là chính xác.

Ý tưởng:

- **Phương pháp tham chiếu:** Testbench sử dụng một hàm tham chiếu (reference function) để tính toán kết quả nhân chập 2D. Hàm này thực hiện phép toán nhân chập một cách thủ công, không sử dụng các kỹ thuật tối ưu như trong mô hình C.
- **So sánh kết quả:** Kết quả từ hàm tham chiếu sẽ được so sánh với kết quả từ mô hình C (`do_convolution`). Nếu có sai lệch, testbench sẽ báo lỗi.
- **Dữ liệu đầu vào:** Testbench sử dụng các bộ dữ liệu đầu vào khác nhau, bao gồm ảnh test và kernel test, để kiểm tra tính đúng đắn của mô hình C trong các trường hợp khác nhau. Ví dụ, chúng tôi sử dụng ảnh đầu vào 5x5 và kernel 3x3.

- **Luồng dữ liệu:** Trong testbench, ảnh đầu vào được đưa vào luồng `in_stream`, kernel cũng được đưa vào, hàm `do_convolution` thực hiện tính toán và kết quả được ghi vào luồng `out_stream`.

```
HLS Output (after filtering):
16 28 35 42 72
36 65 74 83 136
61 110 119 128 201
86 155 164 173 266
34 94 99 104 182
Convolution Output (Reference):
65 74 83
110 119 128
155 164 173
Test passed: HLS output matches reference output.
```

Hình 2: Ví dụ kết quả của Testbench

5 Tối ưu thiết kế để nâng cao hiệu năng

Để nâng cao hiệu năng của thiết kế, chúng tôi đã áp dụng một số kỹ thuật tối ưu hóa trong quá trình triển khai mô hình C và sử dụng các chỉ thị (directives) của HLS. Trong quá trình thiết kế, chúng tôi nhận thấy rằng việc tối ưu hóa truy cập bộ nhớ là yếu tố then chốt để nâng cao hiệu năng của phép nhân chập 2D trên phần cứng. Dựa trên nhận định này, chúng tôi đã phát triển qua hai phiên bản chính: phiên bản cơ sở (naive implementation) và phiên bản sử dụng line buffer (efficient streaming using line buffers).

5.1 Phiên bản cơ sở (Naive Implementation)

Phiên bản đầu tiên được triển khai một cách đơn giản, dễ hiểu, tương tự như cách thực hiện trên CPU. Mã giả C (pseudo-code) cho phiên bản này như sau:

Listing 1: Pseudo-code cho phien ban co so (Naive Implementation)

```
// tinh toan gia tri dau ra cho mot pixel
function single_operation(window, y, x):
  result = 0
  for i = -HALF_SIZE to HALF_SIZE:
    for j = -HALF_SIZE to HALF_SIZE:
      if (y + i, x + j) nam trong gioi han anh:
        result = result + window[i + HALF_SIZE][j + HALF_SIZE] * (i + j)
  return result

// Thuc hien nhan chap 2d
function my_filter_v1(data_out, data_in):
  for y = 0 to HEIGHT - 1:
    for x = 0 to WIDTH - 1:
      // Tai cua so window tu data_in
      for i = -HALF_SIZE to HALF_SIZE:
        for j = -HALF_SIZE to HALF_SIZE:
          if (y + i, x + j) n m trong g i i h n nh :
            window[i + HALF_SIZE][j + HALF_SIZE] = data_in[y + i][x + j]

      // tinh ra tri dau ra
      val_out = single_operation(window, y, x)
      data_out[y][x] = val_out
```

5.1.1 Phân tích

Vấn đề: Phiên bản này gặp phải vấn đề lớn về hiệu năng do bài toán I/O bound. Mỗi lần tính toán cho một pixel đầu ra, nó cần đọc `WIN_SIZE * WIN_SIZE` pixel từ bộ nhớ ngoài (ví dụ, 9 lần đọc cho kernel 3x3). Việc truy cập bộ nhớ ngoài liên tục này tạo ra nút cổ chai (*bottleneck*), làm chậm toàn bộ quá trình xử lý.

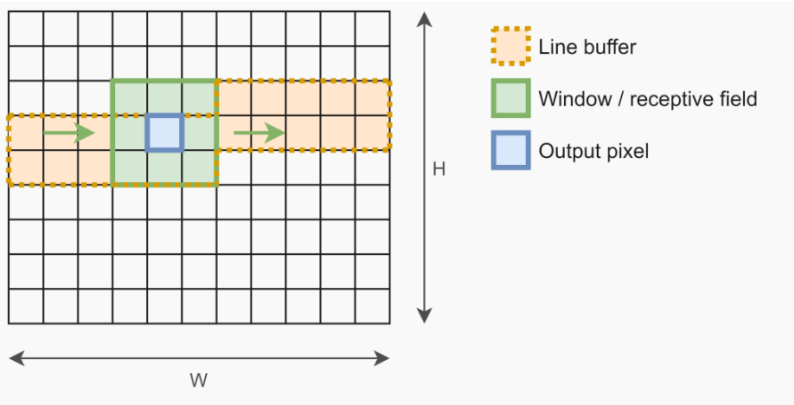
Hạn chế:

- **Thông lượng thấp:** Do hạn chế về số lần truy cập I/O đồng thời trong mỗi chu kỳ xung nhịp, thiết kế bị buộc phải mất nhiều chu kỳ hơn để nạp đủ dữ liệu cho một lần tính toán.
- **Độ trễ cao:** Việc phải chờ đợi nạp dữ liệu từ bộ nhớ ngoài làm tăng độ trễ tổng thể.

Để khắc phục vấn đề *I/O bound* của phiên bản cơ sở, chúng tôi đã áp dụng kỹ thuật Line Buffer kết hợp với Streaming. Ý tưởng chính là lưu trữ tạm thời một phần dữ liệu đầu vào (cụ thể là các dòng pixel) trong bộ nhớ đệm (*Line Buffer*) trên chip, từ đó giảm đáng kể số lần truy cập bộ nhớ ngoài.

Ý tưởng:

- **Line Buffer:** Một bộ đệm có kích thước `(WIN_SIZE - 1) * WIDTH` được sử dụng để lưu trữ `WIN_SIZE - 1` dòng pixel gần nhất của ảnh đầu vào.
- **Window:** Một cửa sổ `WIN_SIZE * WIN_SIZE` trượt trên Line Buffer và một phần nhỏ dữ liệu mới nạp vào từ `in_stream` để lấy dữ liệu cho mỗi lần tính toán.
- **Cập nhật Line Buffer:** Khi cửa sổ trượt sang phải, một cột pixel mới từ ảnh đầu vào sẽ được nạp vào Line Buffer thông qua `in_stream`, và cột pixel ngoài cùng bên trái của Line Buffer sẽ bị loại bỏ.
- **Tính toán:** Việc tính toán giá trị đầu ra vẫn sử dụng hàm `single_operation` (hoặc tương đương), nhưng dữ liệu đầu vào sẽ được lấy từ Window thay vì trực tiếp từ bộ nhớ ngoài.



Hình 3: Cơ chế hoạt động của Line Buffer

Lợi ích:

- **Giảm số lần truy cập bộ nhớ ngoài:** Thay vì phải đọc `WIN_SIZE * WIN_SIZE` pixel cho mỗi lần tính toán, phiên bản này chỉ cần đọc 1 pixel mới từ `in_stream` trong mỗi chu kỳ (để cập nhật Line Buffer).
- **Tăng thông lượng:** Việc giảm số lần truy cập I/O cho phép thiết kế đạt được *Initiation Interval (II)* = 1, nghĩa là mỗi chu kỳ xung nhịp có thể cho ra một kết quả đầu ra.
- **Giảm độ trễ:** Do không phải chờ đợi nạp dữ liệu từ bộ nhớ ngoài, độ trễ tổng thể được giảm đáng kể.
- **Sử dụng tài nguyên hiệu quả hơn:** Sử dụng `hls::stream` để tối ưu việc truyền dữ liệu.

Việc áp dụng kỹ thuật *Line Buffer* và *Streaming* đã giúp tối ưu đáng kể thiết kế, chuyển từ *I/O bound* sang *compute bound*, từ đó nâng cao hiệu năng, giảm độ trễ và sử dụng tài nguyên hiệu quả hơn. Đây là yếu tố then chốt giúp thiết kế đạt được yêu cầu về tốc độ xử lý trong các ứng dụng thời gian thực.

5.2 Từ Solution 1 (chưa thêm dẫn hướng) đến Solution 2

- **Pipelining:** Chỉ thị `#pragma HLS PIPELINE` được sử dụng trong các vòng lặp chính để cho phép thực hiện các lần lặp tiếp theo trước khi lần lặp hiện tại hoàn thành. Điều này giúp tăng thông lượng (throughput) của thiết kế.
- **Array Partitioning:** Chỉ thị `#pragma HLS ARRAY_PARTITION` được sử dụng để phân chia mảng `window` thành các phần tử riêng lẻ, cho phép truy cập đồng thời vào nhiều phần tử của mảng, giảm độ trễ.
- **Loop Unrolling:** HLS có thể tự động unroll các vòng lặp nhỏ để tăng tốc độ thực thi.

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- for_y_for_x	127	127	8	5	1	25	yes

Hình 4: Báo cáo Vivado HLS của solution 2

Bảng So Sánh

Bảng 0: So sánh hiệu năng giữa Solution 1 và Solution 2

Thông số	**Solution 1**	**Solution 2**
Latency (min)	1736	127
Latency (max)	1736	127
Interval (min)	1736	127
Interval (max)	1736	127
BRAM_18K	0	0
DSP48E	3	27
FF	303	1120
LUT	594	1591

5.3 Từ Solution 2 đến Solution 3

- **Line Buffer:** Kỹ thuật Line Buffer được sử dụng để lưu trữ các hàng pixel của ảnh đầu vào. Thay vì phải đọc lại toàn bộ cửa sổ từ bộ nhớ ngoài cho mỗi phép tính, Line Buffer cho phép tái sử dụng dữ liệu đã được đọc, giảm số lần truy cập bộ nhớ ngoài.
- **Streaming:** Sử dụng `hls::stream` để truyền dữ liệu giữa các khối chức năng. Điều này giúp giảm độ trễ và cho phép các khối chức năng hoạt động song song.
- **Tối ưu Array Partitioning:**
  - `#pragma HLS ARRAY_PARTITION variable=line_buf complete dim=1` được sử dụng để tối ưu truy cập vào line buffer theo chiều dọc.
  - `#pragma HLS ARRAY_PARTITION variable=window complete dim=0` được sử dụng để tối ưu truy cập vào window.
- **II=1:** `#pragma HLS PIPELINE II=1` được sử dụng để đặt mục tiêu đạt được Initiation Interval bằng 1, nghĩa là mỗi clock cycle có thể bắt đầu một phép tính mới.

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- buf_x1	2	2	2	1	1	2	yes
- buf_x2	5	5	2	1	1	5	yes
- win_y_win_x	4	4	2	1	1	4	yes
- for_y_for_x	28	28	5	1	1	25	yes

Hình 5: Báo cáo Vivado HLS của solution 3

Bảng So Sánh

Bảng 1: So sánh hiệu năng giữa Solution 2 và Solution 3

Thông số	**Solution 2**	**Solution 3**
Latency (min)	127	46
Latency (max)	127	46
Interval (min)	127	46
Interval (max)	127	46
BRAM_18K	0	4
DSP48E	27	0
FF	1120	845
LUT	1591	1471

6 Mô phỏng/Thực thi và Đánh giá

6.1 Mô phỏng

Sau khi tối ưu hóa thiết kế qua các phiên bản, chúng tôi tiến hành mô phỏng ở mức RTL (Register Transfer Level) sử dụng công cụ C/RTL co-simulation của Vivado HLS. Mục đích của bước này là để kiểm tra:

- **Tính đúng đắn của thiết kế**: Đảm bảo rằng thiết kế RTL thực hiện chính xác chức năng của thuật toán nhân chập 2D.
- **Đánh giá sớm hiệu năng**: Ước lượng các thông số về độ trễ (latency), khoảng thời gian giữa hai lần khởi tạo liên tiếp (initiation interval - II), và tài nguyên sử dụng trước khi thực hiện các bước tổng hợp và triển khai tốn nhiều thời gian.

Quy trình mô phỏng:

1. **Chuẩn bị Testbench**: Testbench viết bằng C++ được sử dụng để cung cấp dữ liệu đầu vào (ảnh và kernel) cho thiết kế RTL và kiểm tra kết quả đầu ra. Testbench này cũng bao gồm một hàm tham chiếu (reference function) thực hiện phép nhân chập 2D để so sánh với kết quả của thiết kế.
2. **Cấu hình Co-simulation**: Trong Vivado HLS, chúng tôi cấu hình để chạy C/RTL co-simulation, lựa chọn ngôn ngữ mô phỏng (Verilog/VHDL) và các tùy chọn liên quan.
3. **Chạy mô phỏng**: Vivado HLS sẽ tự động biên dịch code C/C++ của thiết kế và testbench, tạo ra mô hình RTL, và chạy mô phỏng.
4. **Phân tích kết quả**: Kết quả mô phỏng (dạng sóng tín hiệu, giá trị đầu ra) được hiển thị và phân tích để đánh giá tính đúng đắn và hiệu năng của thiết kế.

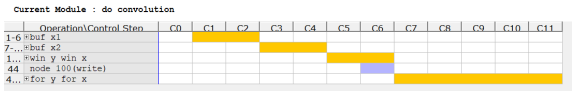
Cosimulation Report for 'do_convolution'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	Pass	47	47	47	NA	NA	NA
Verilog	NA	NA	NA	NA	NA	NA	NA

Export the report(.html) using the Export Wizard

Hình 6: Quá trình mô phỏng C/RTL Co-simulation trong Vivado HLS thành công

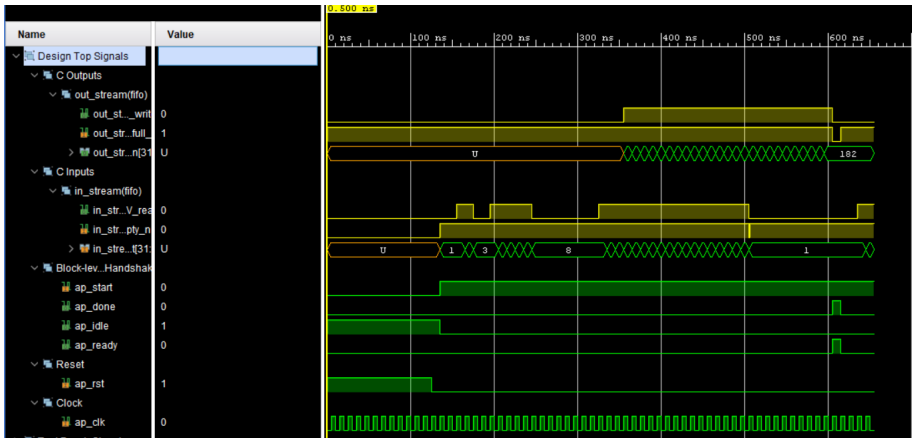


Hình 7: Kết quả phân tích (Analysis) sau khi chạy Co-simulation

6.2 Đánh giá thông qua Vivado HLS report

Sau khi thực hiện *Run C Synthesis* trên Vivado HLS, chúng tôi sử dụng các báo cáo của Vivado HLS để đánh giá hiệu năng của thiết kế, cụ thể như sau:

- **Kết quả mô phỏng**: Kết quả trên Vivado HLS cho thấy thiết kế hoạt động chính xác, với đầu ra trùng khớp với kết quả từ hàm tham chiếu trong testbench.
- **Các chỉ số về hiệu năng**: Các chỉ số như *latency* và *interval* được đánh giá thông qua phần *Synthesis Summary* trong báo cáo của Vivado HLS. Báo cáo cung cấp thông tin chi tiết về độ trễ, *Initiation Interval (II)*, và tài nguyên sử dụng của từng vòng lặp và hàm trong thiết kế.
- **Hiệu quả tối ưu hóa**: Với **Solution 3**, thiết kế đã đạt được *Initiation Interval (II)* = 1 ở các vòng lặp quan trọng, nghĩa là mỗi chu kỳ xung nhịp (*clock cycle*) có thể tạo ra một đầu ra. Điều này khẳng định hiệu quả của các kỹ thuật tối ưu như *Line Buffer* và *Streaming* đã áp dụng.



Hình 8: Dạng sóng tín hiệu đầu vào và đầu ra

Phân tích chi tiết:

- **Tính đồng bộ**: Hình 8 minh họa rằng các tín hiệu đầu vào (ảnh, kernel) và tín hiệu đầu ra được đồng bộ tốt. Tín hiệu *ap_start* điều khiển quá trình thực thi, và tín hiệu *ap_done* báo hiệu khi quá trình tính toán hoàn tất.

- **Độ trễ (*Latency*)**: Độ trễ được xác định bằng số chu kỳ xung nhịp từ khi bắt đầu nạp dữ liệu đầu vào đến khi có kết quả đầu ra đầu tiên.
- ***Initiation Interval (II)***: II cho biết số chu kỳ xung nhịp cần thiết để thiết kế có thể nhận dữ liệu đầu vào mới. Giá trị II = 1 là lý tưởng, cho thấy thiết kế có thể xử lý dữ liệu mới ở mỗi chu kỳ xung nhịp.

6.3 Đánh giá hiệu năng chi tiết

Các bảng dưới đây trình bày chi tiết kết quả đánh giá hiệu năng của Solution 2 và Solution 3, được trích xuất từ báo cáo tổng hợp (*Synthesis Report*) của Vivado HLS.

Bảng 2: Kết quả đánh giá hiệu năng của Solution 2

Loop Name	Latency (min)	Latency (max)	Iteration Latency	Initiation Interval (II)		Trip Count	Pipelined
				Achieved	Target		
for_y_for_x	127	127	8	5	1	25	Yes

Phân tích Solution 2:

- **Vòng lặp for_y_for_x**: Đây là vòng lặp chính thực hiện phép nhân chập với độ trễ là 127 chu kỳ và *Initiation Interval (II)* là 5.
- **II = 5**: Điều này có nghĩa là sau mỗi 5 chu kỳ, vòng lặp này mới có thể nhận dữ liệu mới để tính toán. Nguyên nhân có thể là do một số hạn chế trong việc truy cập dữ liệu, mặc dù đã được cải thiện so với Solution 1.
- **Pipelined**: Vòng lặp đã được áp dụng kỹ thuật *pipelining* (II = 5 nhưng *Iteration Latency* = 8), cho thấy Vivado HLS đã cố gắng tối ưu hiệu năng. Tuy nhiên, hiệu suất này vẫn chưa đạt mức tối ưu cao nhất.

Bảng 3: Kết quả đánh giá hiệu năng của Solution 3

Loop Name	Latency (min)	Latency (max)	Iteration Latency	Initiation Interval (II)		Trip Count	Pipelined
				Achieved	Target		
buf_x1	2	2	2	1	1	2	Yes
buf_x2	5	5	2	1	1	5	Yes
win_y_win_x	4	4	2	1	1	4	Yes
for_y_for_x	27	27	4	1	1	25	Yes

Phân tích Solution 3:

- **buf_x1, buf_x2, win_y_win_x**: Các vòng lặp con này đều đạt *Initiation Interval (II)* = 1 nhờ kỹ thuật *Line Buffer* và *Streaming*, cho phép nạp dữ liệu mới ở mỗi chu kỳ.
- **for_y_for_x**: Vòng lặp chính cũng đạt II = 1, cho thấy thiết kế đã được tối ưu tốt và có thể xử lý một pixel đầu ra ở mỗi chu kỳ xung nhịp.
- **Latency**: Độ trễ tổng thể là 46 chu kỳ, thấp hơn nhiều so với Solution 2 (127 chu kỳ).
- **Trip Count**: Số lần lặp của mỗi vòng lặp được xác định chính xác, cho thấy các vòng lặp được triển khai hiệu quả.

Bảng 4: Tổng hợp kết quả đánh giá hiệu năng

Thông số	Solution 1	Solution 2	Solution 3
Latency (min)	1736	127	46
Latency (max)	1736	127	46
Interval (min)	1736	127	46
Interval (max)	1736	127	46
BRAM_18K	0	0	4
DSP48E	3	27	0
FF	303	1120	845
LUT	594	1591	1471

So sánh các Solution:

- **Latency:** Solution 3 có độ trễ thấp nhất (46 chu kỳ), tiếp theo là Solution 2 (127 chu kỳ) và cuối cùng là Solution 1 (1736 chu kỳ).
- **Interval:** Solution 3 đạt *Initiation Interval (II)* = 1 cho tất cả các vòng lặp, cho thấy hiệu suất cao nhất. Solution 2 có II = 5 ở vòng lặp chính, trong khi Solution 1 có II lớn (bị *bottleneck* ở I/O).
- **Tài nguyên:**
  - **BRAM:** Solution 3 sử dụng 4 khối BRAM (cho *Line Buffer*), trong khi Solution 1 và 2 không sử dụng.
  - **DSP:** Solution 2 sử dụng nhiều DSP nhất (27 khối) do HLS tự động *unroll* các vòng lặp và tối ưu theo hướng sử dụng nhiều phép nhân song song. Solution 3 không sử dụng DSP nào.
  - **FF và LUT:** Solution 2 sử dụng nhiều FF và LUT hơn do tối ưu theo hướng *pipelining* và *unrolling*. Solution 3 sử dụng ít tài nguyên logic hơn so với Solution 2.

6.4 Kết luận

Qua các bước mô phỏng và đánh giá, chúng tôi đã chứng minh được tính đúng đắn và hiệu quả của thiết kế, đặc biệt là Solution 3. Việc áp dụng kỹ thuật *Line Buffer* và *Streaming* đã giúp tối ưu đáng kể hiệu năng, đạt được *Initiation Interval (II)* = 1, độ trễ thấp và sử dụng tài nguyên hợp lý.

Kết quả từ báo cáo Vivado HLS cho thấy thiết kế đã sẵn sàng cho bước triển khai trên phần cứng.

7 Kết quả triển khai phần cứng trên ZynQ

7.1 Đánh giá hiệu năng, tài nguyên sử dụng trên ZynQ-7000

Kết quả kiểm tra trên phần cứng xác nhận rằng thiết kế hoạt động chính xác và đạt hiệu năng như kỳ vọng từ quá trình mô phỏng.

Bảng dưới đây trình bày chi tiết về mức độ sử dụng tài nguyên phần cứng trên nền tảng ZynQ-7000 sau khi triển khai thiết kế:

Bảng 5: Sử dụng tài nguyên trên ZynQ-7000

Thành phần	BRAM_18K	DSP48E	FF	LUT
DSP	0	0	0	0
Expression	0	0	0	1248
FIFO	0	0	0	0
Instance	0	0	0	0
Memory	4	0	0	0
Multiplexer	0	0	0	359
Register	0	0	1103	128
Total	4	0	1103	1735
Available	280	220	106400	53200
Utilization (%)	1	0	1	3

Số liệu:

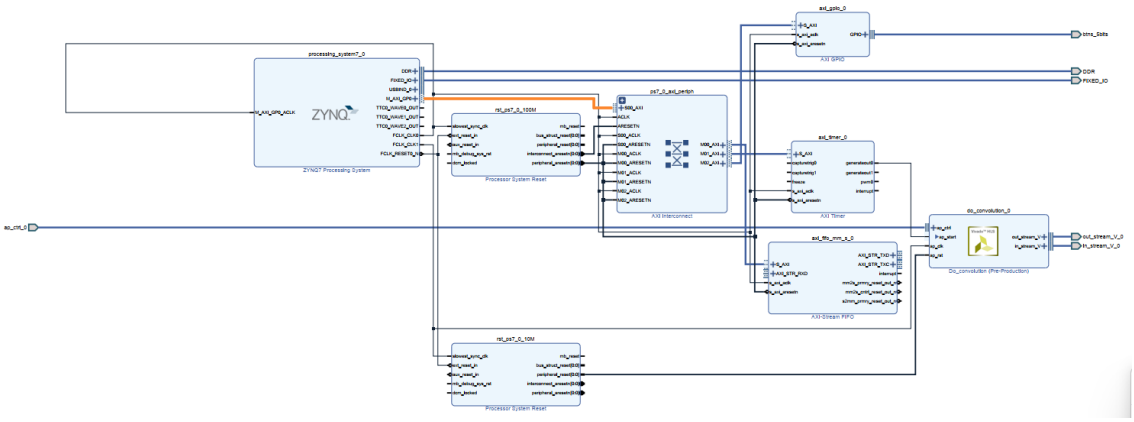
- **LUTS:** 3% sử dụng (1735/53200)
- **FFs:** 1% sử dụng (1103/106400)
- **BRAMS:** 1% sử dụng (4/280)
- **DSPs:** 0% sử dụng (0/220)

Phân tích:

- Thiết kế cho thấy mức độ sử dụng tài nguyên phần cứng thấp và hiệu quả. Cụ thể, chỉ có 3% LUTs, 1% FFs, 1% BRAMs và 0% DSPs được sử dụng.
- Việc sử dụng ít BRAM (4 khối) cho thấy thiết kế đã tối ưu tốt việc lưu trữ dữ liệu araay line_buf, window bằng cách tận dụng tài nguyên on-chip thay vì bộ nhớ ngoài.
- Việc không sử dụng DSP cho thấy thiết kế đã được tối ưu theo hướng sử dụng tài nguyên logic (LUTs, FFs) thay vì các khối nhân chuyên dụng. Điều này có thể là do các phép nhân đã được tối ưu và triển khai hiệu quả bằng LUTs thông qua các kỹ thuật như *pipelining* và *unrolling*.
- Nhìn chung, kết quả cho thấy thiết kế tối ưu, sử dụng tài nguyên phần cứng hiệu quả, với chỉ phí tài nguyên thấp nhờ các kỹ thuật tối ưu hóa như Line Buffer, Streaming, chia sẻ tài nguyên và ánh xạ bộ nhớ hợp lý.

7.2 Thiết kế mạch trên ZynQ-7000

Hình dưới đây minh họa sơ đồ thiết kế phần cứng đã được triển khai trên nền tảng ZynQ-7000.



Hình 9: Thiết kế mạch trên ZynQ-7000

Mô tả thiết kế:

- **ZynQ Processing System:** Khối xử lý chính sử dụng bộ vi xử lý ARM Cortex-A9 trong ZynQ-7000 để điều khiển hệ thống.
- **AXI Interconnect:** Giao tiếp AXI kết nối các thành phần trong hệ thống, bao gồm bộ xử lý và các IP ngoại vi.
- **AXI Stream FIFO:** Bộ đệm dữ liệu giữa các luồng AXI để truyền dữ liệu giữa các khối xử lý.
- **Data Processing (do_convolution):** Khối tính toán chính, chịu trách nhiệm thực hiện các phép tính tích chập và xử lý dữ liệu.
- **GPIO Control:** Tín hiệu điều khiển kết nối các đầu vào/ra của hệ thống, đảm bảo giao tiếp linh hoạt với các thiết bị ngoại vi.

Thiết kế sử dụng kết hợp các giao tiếp AXI và FIFO để tối ưu hóa luồng dữ liệu giữa các khối IP, đảm bảo độ trễ thấp và băng thông cao cho hệ thống.

8 Kết luận

Trong dự án này, chúng tôi đã thiết kế, mô phỏng và triển khai thành công khối tính nhân chập 2-D sử dụng phương pháp tổng hợp mức cao (HLS) trên nền tảng ZynQ-7000. Qua quá trình nghiên cứu và thực nghiệm, chúng tôi đã đạt được các kết quả sau:

- **Phát triển thuật toán hiệu quả:** Thuật toán nhân chập 2-D đã được xây dựng và tối ưu qua nhiều phiên bản, từ phiên bản cơ sở (naive) đến phiên bản sử dụng kỹ thuật Line Buffer và Streaming, giúp cải thiện đáng kể hiệu năng.
- **Xây dựng mô hình C và Testbench:** Mô hình C cho thuật toán đã được phát triển và kiểm chứng tính đúng đắn thông qua Testbench với các bộ dữ liệu kiểm thử đa dạng.
- **Tổng hợp HLS và tối ưu thiết kế:** Sử dụng Vivado HLS, mô hình C đã được tổng hợp thành công sang thiết kế RTL. Các kỹ thuật tối ưu như Pipelining, Array Partitioning, và đặc biệt là Line Buffer và Streaming đã được áp dụng để nâng cao hiệu năng và giảm tài nguyên sử dụng.
- **Đạt được hiệu năng cao:** Solution 3 với kỹ thuật Line Buffer và Streaming đã đạt được Initiation Interval (II) = 1, độ trễ thấp (46 chu kỳ), và sử dụng tài nguyên hợp lý.
- **Triển khai thành công trên phần cứng:** Thiết kế đã được triển khai và kiểm tra trên nền tảng ZynQ-7000, cho thấy kết quả sử dụng tài nguyên phần cứng thấp và hiệu quả, khẳng định tính đúng đắn và khả năng ứng dụng thực tế của thiết kế.

Đóng góp của dự án:

- **Về mặt học thuật:** Dự án đã minh họa cách tiếp cận thiết kế phần cứng bằng HLS, cho thấy tiềm năng của HLS trong việc tăng tốc các thuật toán xử lý tín hiệu số, đặc biệt là phép nhân chập 2-D.
- **Về mặt ứng dụng:** Thiết kế có thể được sử dụng như một bộ tăng tốc phần cứng (hardware accelerator) cho các ứng dụng xử lý ảnh, thị giác máy tính và học sâu, đặc biệt trong các hệ thống nhúng và IoT với tài nguyên hạn chế.

Hướng phát triển:

- **Tối ưu thêm cho các trường hợp kernel và ảnh đầu vào lớn hơn:** Nghiên cứu các kỹ thuật phân chia dữ liệu và quản lý bộ nhớ hiệu quả hơn để xử lý các trường hợp đầu vào lớn.
- **Tích hợp với các hệ thống lớn hơn:** Phát triển giao diện và các module điều khiển để tích hợp thiết kế vào các hệ thống xử lý ảnh hoặc các ứng dụng AI trên phần cứng.

- **Khám phá các kỹ thuật HLS nâng cao khác:** Nghiên cứu áp dụng các kỹ thuật tối ưu HLS khác để cải thiện hơn nữa hiệu năng và tài nguyên sử dụng.
- **Mở rộng sang các phép toán khác trong xử lý ảnh và học sâu:** Áp dụng phương pháp HLS để thiết kế và tối ưu các khối tính toán khác như pooling, activation functions, v.v.

Tóm lại, dự án đã đạt được các mục tiêu đề ra và mở ra hướng phát triển tiềm năng cho các ứng dụng xử lý tín hiệu số và thị giác máy tính trên các hệ thống nhúng sử dụng HLS.

9 Phụ lục

9.1 Mã VHDL

Mã VHDL của thiết kế đã được đính kèm theo bài báo cáo.

9.2 Danh sách hình ảnh

9.3 Danh sách bảng biểu

10 Tài liệu tham khảo

1. Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis*. UG902 (v2021.2).
2. Xilinx. *Zynq-7000 All Programmable SoC Overview*. DS190 (v1.19).
3. Cong, J., Xiao, B., & Zhang, C. (2014). *A systolic array architecture for high-throughput convolutional neural network training*. In Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays (pp. 123-132).
4. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. (2015). *Optimizing FPGA-based accelerator design for deep convolutional neural networks*. In Proceedings of the 2015 ACM/SIGDA international symposium on Field-programmable gate arrays (pp. 161-170).
5. Suda, N., Chandra, V., Dasika, G., Mohanty, A., Ma, Y., Vrudhula, S., ... & Seo, J. S. (2016). *Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks*. In Proceedings of the 2016 ACM/SIGDA international symposium on Field-programmable gate arrays (pp. 16-25).