Goals

The system application of the HW accelerator is to accelerate Yolov5 image detection inference application for traffic violation detection. Specifically Helmet Detection. The inference application is typically running on a camera with AI enabled HW that is mounted at a traffic signal.

Architecture Analysis

AI Algorithm

- Investigation of the best ML algorithm for the stated goal.
- Initial attempt with Keras based binary classification model. Did not provide results with sufficient accuracy
- With additional research determined that a prebuilt yolov5 based model would provide higher accuracy.
- Ran the model on Kaggle servers to determine the accuracy and ensured that it provides stated results.

Algorithm Analysis

- Yolov5 is a prebuilt AI model. The user simply selects from various sizes of the prebuilt model (nano, small, medium, large etc) based on system goals
- Yolov5 architecture is described on https://docs.ultralytics.com/yolov5/tutorials/architecture_description/
- The block diagram provides the basic algorithm used.

HW Accelerator Analysis

In a YoloV5 inference the 16x16 vector multiplication would dominate the workflow. It takes approx. 85% of the total CPU cycles.

A typical CPU is expected to take approx. 100 clock cycles to complete the operation. The HW accelerator actually performs the same operation in 16 clock cycles. So the multiplication portion would speed up by 6X

So the final speed up with the addition of the accelerator would be approx. 500%. (600% *0.85)

Yolo5 Block Diagram

YOLOv5's architecture consists of three main parts:

- **Backbone**: This is the main body of the network. For YOLOv5, the backbone is designed using the `CSPDarknet53` structure, a modification of the Darknet architecture used in previous versions.
- **Neck**: This part connects the backbone and the head. In YOLOv5, `SPPF` (Spatial Pyramid Pooling - Fast) and `PANet` (Path Aggregation Network) structures are utilized.
- **Head**: This part is responsible for generating the final output. YOLOv5 uses the `YOLOv3 Head` for this purpose.

The structure of the model is depicted in the table below.

Block Name	Abbreviation (Common)	Primary Components	Purpose/Functionality	Typical Location in YOLOv5	Notes / Special Features
Convolutional Block	**Conv / CBS**	Conv2d (Convolutional Layer) + BatchNorm2d (Batch Normalization) + SiLU (Activation Function)	Basic building block for feature extraction and spatial downsampling (if stride > 1).	Throughout Backbone & Neck	SiLU (Sigmoid Linear Unit) is the default activation function in YOLOv5, known for its smoothness and improved performance over ReLU in many cases.
Bottleneck	**Bottleneck / BottleneckCSP**	Conv2d (1x1) -> Conv2d (3x3) with residual connection	Reduces computational cost by first compressing channels (1x1 conv) then expanding them (3x3 conv). Includes a shortcut connection to help with gradient flow in deep networks.	Used within C3 blocks in Backbone and Neck	In YOLOv5, it's often referred to as BottleneckCSP or Bottleneck and is the core component within the C3 module. There might be slight variations (e.g., backbone=True/False influencing residual connections).

C3 Block	**C3**	Main Branch: Conv2d + Bottleneck modules (repeated 'n' times) Residual Branch: Conv2d (shortcut) Final: Concatenation of main and residual branches	**Cross Stage Partial (CSP) Network** module. Splits feature map into two parts, one through n Bottleneck modules, the other as a shortcut. Concatenating them reduces computational bottleneck and increases gradient path diversity, improving learning efficiency and feature reuse.	Heavily used in both **Backbone** and **Neck**	The n parameter controls the number of Bottleneck modules in the main branch, influencing model depth and capacity. Different versions of YOLOv5 (n, s, m, l, x) vary n.
Spatial Pyramid Pooling - Fast	**SPPF**	A series of MaxPool2d operations (e.g., 5x5, 9x9, 13x13) applied *sequentially* followed by Concatenation.	Aggregates feature maps at different scales by pooling operations. This helps the network to capture contextual information from various receptive fields, making it more robust to object scale variations. It's an optimized version of the original SPP.	End of the **Backbone**	The "Fast" in SPPF comes from using a sequence of smaller max-pooling kernels (e.g., three 5x5 max-pools) instead of parallel larger ones, which is mathematically equivalent but computationally more efficient.
Upsample	**Upsample**	nn.Upsample (typically bilinear interpolation or nearest neighbor)	Increases the spatial dimensions of a feature map.	In the **Neck** (PANet part) for top-down pathway	Used to bring lower-resolution, semantically rich feature maps to higher resolutions for concatenation with finer-grained features.
Concat	**Concat**	Element-wise concatenation along the channel dimension (dim=1)	Merges feature maps from different paths or scales.	Primarily in the **Neck** (PANet, FPN)	Essential for feature fusion, combining features from backbone to neck, and different

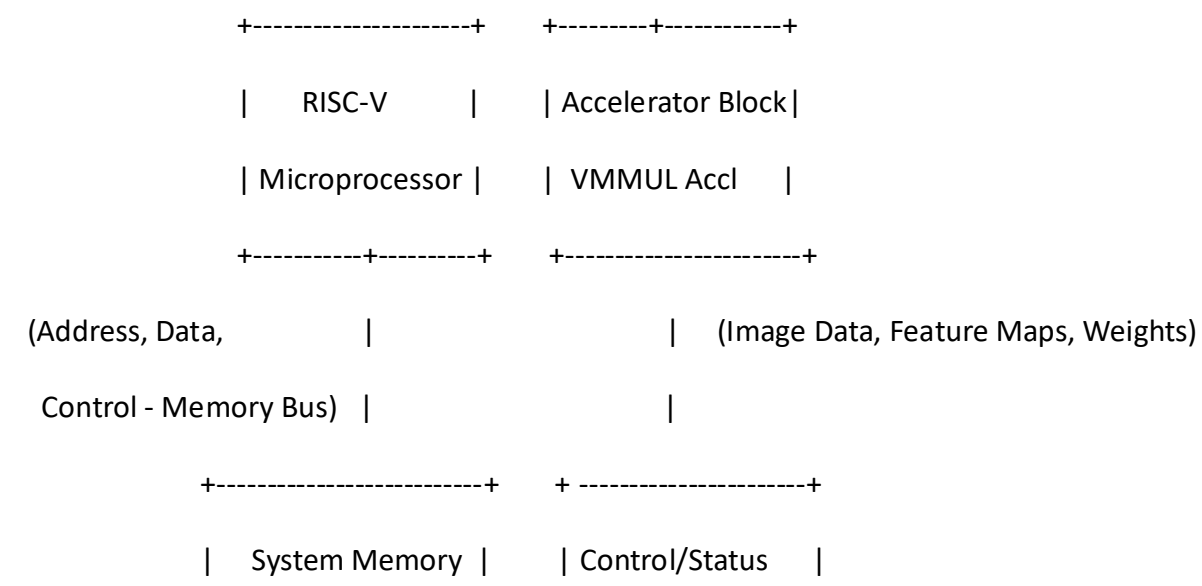
					scales within the neck.
Head (Detection Layer)	**Detect**	Conv2d (1x1) + Sigmoid/Activation	Final prediction layer that maps feature maps to raw object detection outputs (bounding box coordinates, objectness score, class probabilities).	**Head** (final layers, three branches for different scales)	Contains the 1x1 convolutions that output the raw predictions for each anchor box at each grid cell on the multi-scale feature maps from the Neck.

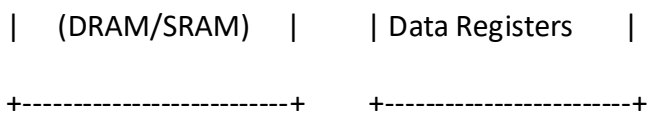
This table covers the most common and fundamental CNN building blocks found in YOLOv5's architecture. The exact configuration (number of layers, channels, repetitions of C3 blocks) will vary between the different YOLOv5 models (e.g., YOLOv5n, YOLOv5s, YOLOv5m, YOLOv5l, YOLOv5x) which are essentially scaled versions of the same core architecture.

HW Accelerator Analysis

Building an accelerator for the entire Yolov5 pipeline would not be practical. Ideally a general approach would be more suitable. The block diagrams below show a step by step analysis to reach the HW accelerator (16x16 vector multiplier systolic array using 2x2 vector multiplier as a PE)

Overview





Explanation of the Blocks and Interfaces:

RISC-V Microprocessor: The main processor that controls the system. It communicates with memory and peripherals (including our accelerator) via a memory bus.

System Memory (DRAM/SRAM): This is the main memory where the input image, kernel weights, intermediate feature maps, and final output will reside.

Accelerator Block: This encapsulates custom hardware for accelerating the first (and potentially subsequent) convolutional layers.

MMIO Mapped Registers: A set of control and status registers within the accelerator that are mapped into the microprocessor's memory address space. The RISC-V core interacts with the accelerator by reading and writing to these specific memory addresses.

Control Registers: Used by the RISC-V core to configure the accelerator (e.g., source and destination memory addresses for DMA, kernel parameters, start signals).

Status Registers: Used by the accelerator to report its status (e.g., idle, busy, done, error) back to the RISC-V core.

DMA Controller: A dedicated hardware unit within the accelerator that can directly transfer data to and from the System Memory without constant intervention from the RISC-V core.

The RISC-V core programs the DMA controller with the source address, destination address, and the amount of data to transfer.

The DMA controller then handles the memory transfers autonomously.

Internal Buffers: On-chip memory within the accelerator to temporarily store the data needed for processing (e.g., the current input image patch, the relevant kernel weights, and the intermediate or final output of the vmmul and accumulation units). These buffers help to reduce latency and improve data locality for the processing core.

Processing Core: This is the core of the accelerator, containing the logic for:

Data Reshape: Rearranging the input data into the required matrix formats.

vmmul Unit: Performing the parallel 2D matrix multiplications.

Accumulation: Summing the results.

Workflow:

Data Loading (by RISC-V): The RISC-V core loads the input image and kernel weights into the System Memory (DRAM/SRAM).

Accelerator Configuration (by RISC-V): The RISC-V core writes to the MMIO Mapped Registers within the Accelerator Block to configure it for the current layer:

Specifies the source address in System Memory for the input feature map (or image).

Specifies the source address in System Memory for the kernel weights.

Specifies the destination address in System Memory for the output feature map.

Provides parameters like kernel size, stride, and channel information.

Starts the accelerator by writing to a control register.

DMA Transfer (by Accelerator): Upon receiving the start signal, the DMA Controller within the Accelerator Block reads the necessary input data (image patches and kernel weights) from the System Memory (as specified in the MMIO registers) and stores them in the Internal Buffers.

Processing (by Accelerator): The Processing Core within the accelerator reads the data from the Internal Buffers, performs the data reshaping, vmmul operations, and accumulation. The results are stored in the internal output buffers.

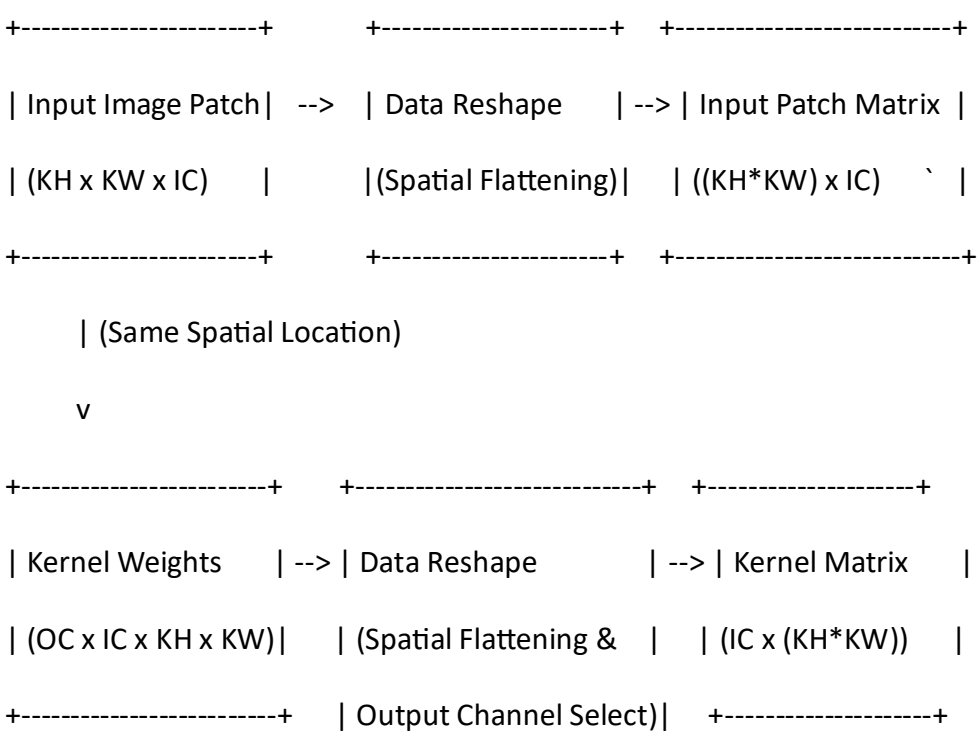
DMA Transfer (by Accelerator): Once the processing is complete, the DMA Controller writes the output feature map data from the Internal Buffers back to the System Memory at the destination address specified in the MMIO registers.

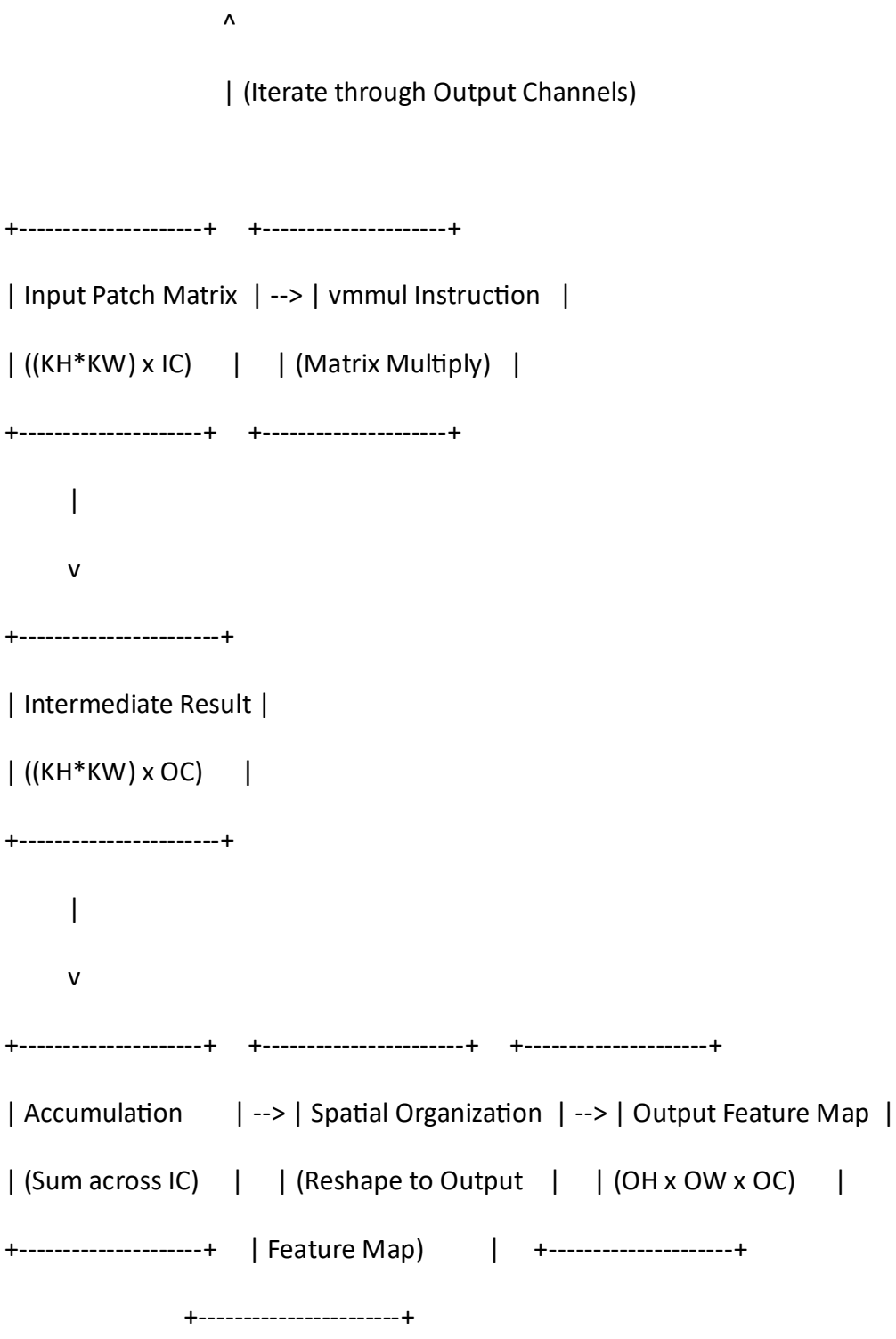
Status Reporting (by Accelerator): The accelerator updates its status registers (accessible via MMIO) to indicate that the operation is complete.

Next Layer (by RISC-V): The RISC-V core checks the status registers. Once the accelerator is done, it can configure the accelerator for the next layer, pointing the input DMA to the output of the previous layer in System Memory, and providing the new kernel weights and output address.

This block diagram provides a more standard view of how a hardware accelerator interacts with a microprocessor in a system-on-a-chip (SoC) design. The MMIO interface allows the software (running on the RISC-V core) to control the hardware, and the DMA controller enables efficient data transfers without constant CPU intervention.

VMMUL Block Accelerator (Conceptual)





Explanation of the Blocks:

Input Image Patch (KH x KW x IC): This represents a small 3D block of the input image corresponding to the spatial extent of the convolutional kernel (Kernel Height - KH, Kernel Width - KW) and all Input Channels (IC) for a specific output spatial location.

Kernel Weights (OC x IC x KH x KW): This represents the 4D tensor of convolutional kernel weights. For processing a single output spatial location, we focus on the weights for all Input Channels and the current Kernel Height and Width, for each Output Channel (OC).

Data Reshape (Spatial Flattening):

Input Patch: The (KH x KW x IC) patch is reshaped into a 2D matrix of size ((KH * KW) x IC). This flattens the spatial dimensions into the rows of the matrix, with each column corresponding to an input channel.

Kernel Weights: For a specific output channel, the (IC x KH x KW) kernel is reshaped into a 2D matrix of size (IC x (KH * KW)). Here, each row corresponds to an input channel, and the columns represent the flattened spatial kernel weights.

Input Patch Matrix ((KH*KW) x IC): The 2D representation of the input image patch.

Kernel Matrix (IC x (KH*KW)): The 2D representation of the kernel weights for a specific output channel.

vmmul Instruction (Matrix Multiply): Your specialized 2D matrix multiplication instruction takes the Input Patch Matrix and the Kernel Matrix as input. The result of this multiplication will be an Intermediate Result matrix of size ((KH*KW) x (KH*KW)).

Intermediate Result ((KH*KW) x (KH*KW)): This matrix contains the element-wise products and sums that need to be further processed to get the final output feature value.

Accumulation (Sum across Input Channel): To get the single output feature value for a specific output spatial location and a specific output channel, you need to perform a reduction (summation) across the appropriate dimensions of the Intermediate Result. The exact dimensions to sum over depend on the precise reshaping done in step 3 and the semantics of your vmmul.

More Direct Reshape and Multiply (Revised Flow): A more direct reshape of the Kernel Matrix to ((KH*KW) x IC) would lead to an Intermediate Result of ((KH*KW) x OC) directly after vmmul (if we process all output channels in parallel within the vmmul). Then, the accumulation would happen along the first dimension (KH*KW) to get the (1 x OC) output for that spatial location.

Output Feature Map (OH x OW x OC): After processing all spatial locations and output channels, the accumulated results are organized into the output feature map with dimensions Output Height (OH), Output Width (OW), and Output Channels (OC).

Spatial Organization (Reshape to Output Feature Map): This block represents the process of taking the individual output values calculated for each spatial location and arranging them into the final output feature map.

Iteration:

The process within the dashed box (Data Reshape -> vmmul -> Accumulation) is repeated for each output spatial location (sliding the kernel across the input image with the specified stride).

The "Iterate through Output Channels" arrow indicates that the kernel weights are selected for each output channel, and the vmmul and accumulation are performed to compute the corresponding output feature map channel.

Key Idea:

The diagram highlights how you're breaking down the 3D convolution into a series of 2D matrix multiplications using your specialized vmmul. The efficiency depends heavily on how well you can structure the data in the Input Patch Matrix and Kernel Matrix to maximize the utilization of your vmmul instruction and minimize the overhead of data reshaping and accumulation.

This block diagram provides a visual representation of the data flow and the key processing steps involved in using a 2D vmmul for a 3D convolution. Remember that the specific dimensions and reshaping steps might need to be adjusted based on the exact design of your vmmul instruction and your desired data processing flow.

Final Decision of Accelerator to build

To build a PE (Processing Element) array for a YOLOv5-class ASIC or FPGA accelerator, the best practice is to design a flexible, schedule-aware PE array capable of efficiently supporting a variety of dataflows and quantizations. Here's a concise, research-backed summary of what you should build:

Recommended PE Array Architecture

1. Flexible, Schedule-Aware PE Array

Array Structure:

Use a square grid (e.g., 16×16 or 32×32) of PEs for balance between scalability and control logic simplicity.

Each PE is a Versatile Processing Element (VPE), optimized for MAC (Multiply-Accumulate) operations, with support for INT8/INT4 and possibly FP16 for flexibility.

The array should be runtime-configurable: able to switch between input-stationary, weight-stationary, output-stationary, or mixed dataflows depending on the current layer's needs.

Local Storage:

Each PE includes small local register files (RFs) for input feature maps, weights, partial sums, and output activations, maximizing data reuse and minimizing SRAM/DRAM access.

Sparsity Support:

Integrate logic for sparsity acceleration: use bitmaps to skip zero activations/weights, reducing unnecessary MACs and saving power.

Partial Sum Accumulation:

Use a flexible adder tree (e.g., FlexTree) for efficient accumulation of partial sums across the PE array, with runtime-configurable depth to match the layer's requirements.

2. Data Movement and Control

Schedule-Aware Tensor Distribution Network:

Design a data distribution network that can efficiently load and drain data between on-chip SRAM and the PE array, dynamically adapting to the dataflow and layer schedule.

Configuration Registers:

Expose hardware knobs (configuration descriptors) that allow the compiler or runtime to program the optimal dataflow and PE array behavior for each layer.

3. Array Size

16×16 PE array (256 PEs) is a common, practical starting point for modern DNN accelerators.

Larger arrays (e.g., 32×32) may be used if silicon area and memory bandwidth allow, but control complexity and routing overhead increase with size.