Overview

+-----------------+

| RISC-V |

| Microprocessor |

+--------+--------+

| (Address, Data, Control - Memory Bus)

|

v

+---------------------------------------------------------------------+

| System Memory (DRAM/SRAM) |

+---------------------------------------------------------------------+

^ ^

| (Image Data, Feature Maps, Weights) |

| |

| (Address, Data, Control - Memory Bus) |

|

+---------+---------+

| Accelerator Block |

+-------------------+

| |

| +---------------+ | +-----------------+

| | MMIO Mapped | |<--> | Control/Status |

| | Registers | | | Registers |

| +---------------+ | +-----------------+

| |

| +---------------+ |

| | DMA Controller| |

| +---------------+ |

| ^ |

| | (Address, Control)

| | |

+--------+- -------+

| (Data Read/Write Channel)

|

+-------------------+

| Internal Buffers | (e.g., for Input Patch, Weights, Output)

+-------------------+

|

v

+-------------------+

| Processing Core | (Your vmmul-based logic)

| (Data Reshape, |

| vmmul, Accum.) |

+-------------------+

Explanation of the Blocks and Interfaces:

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 your 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.

VMMUL Block Accelerator

+---------------------+ +-----------------------+ +---------------------+

| Input Image Patch | --> | Data Reshape | --> | Input Patch Matrix |

| (KH x KW x IC) | | (Spatial Flattening) | | ((KH\*KW) x IC) |

+---------------------+ +-----------------------+ +---------------------+

|

| (Same Spatial Location)

v

+---------------------+ +-----------------------+ +---------------------+

| Kernel Weights | --> | Data Reshape | --> | Kernel Matrix |

| (OC x IC x KH x KW) | | (Spatial Flattening & | | (IC x (KH\*KW)) |

+---------------------+ | Output Channel Select)| +---------------------+

^

| (Iterate through Output Channels)

+---------------------+ +---------------------+

| Input Patch Matrix | --> | vmmul Instruction |

| ((KH\*KW) x IC) | | (Matrix Multiply) |

+---------------------+ +---------------------+

|

v

+-----------------------+

| Intermediate Result |

| ((KH\*KW) x OC) |

+-----------------------+

|

v

+---------------------+ +-----------------------+ +---------------------+

| Accumulation | --> | Spatial Organization | --> | Output Feature Map |

| (Sum across IC) | | (Reshape to Output | | (OH x OW x OC) |

+---------------------+ | Feature Map) | +---------------------+

+-----------------------+

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.

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.

Block Diagram (Textual)

+-----------------------+

| Schedule-Aware |

| Tensor Distribution |

| Network |

+-----------+-----------+

|

+-------+-------+

| PE Array | <-- 16x16 or 32x32 VPEs

+-------+-------+

|

+-------+-------+

| FlexTree | <-- Flexible adder tree for partial sum accumulation

+---------------+

If you were building an ASIC (Application-Specific Integrated Circuit) for YOLOv5 inference, the hardware architecture would be tailored for high throughput, low latency, and energy efficiency—much more so than an FPGA or GPU. Here’s how a typical ASIC accelerator for YOLOv5 would be architected, based on best practices in the field and the architectural patterns seen in high-performance FPGA accelerators:

Recommended ASIC Hardware Architecture for YOLOv5

1. Processing Element (PE) Array

Massively parallel PE array: The core of the ASIC would be a large array of processing elements, each capable of performing MAC (multiply-accumulate) operations.

Optimized for low-precision arithmetic: For YOLOv5, 4-bit or 8-bit quantized operations are common, so each PE would efficiently support INT8/INT4 MACs.

DSP Packing: Like advanced FPGA designs, ASICs can pack multiple low-bitwidth MACs into a single physical unit, maximizing silicon efficiency.

2. Hierarchical On-Chip Memory

Local SRAM buffers: Each PE or PE cluster has local SRAM to store input tiles, weights, and partial sums, minimizing expensive DRAM access.

Global SRAM: Larger shared buffers for feature maps and weights, supporting double-buffering for overlapping computation and data movement.

Memory Tiling and Loop Tiling: Data is processed in tiles to maximize data reuse and minimize bandwidth bottlenecks.

3. Dedicated Function Units

Convolution Engine: Highly pipelined and parallelized for 2D convolutions, the dominant operation in YOLOv5.

Activation Units: Hardware blocks for Leaky ReLU (used in YOLOv5), sigmoid, and other activation functions.

Pooling/Upsample Units: Specialized logic for max pooling and upsampling layers.

Elementwise Units: For add, multiply, and other per-pixel operations.

4. Task Scheduler and Controller

Central scheduler: Manages layer execution, memory transfers, and dataflow between units.

DMA Engines: For efficient movement of data between on-chip memory and external DRAM.

5. Quantization Support

Native support for INT8/INT4 arithmetic: Reduces area and power, and enables higher parallelism.

Flexible data paths: To accommodate different quantization schemes and layer types.

6. High-Throughput Memory Interfaces

Wide DRAM interfaces: For fast access to weights and feature maps.

AXI or custom high-speed buses: For internal data movement.

Block Diagram (Textual)

text

+----------------------+

| Global SRAM Buffer |

+----------------------+

|

+----------------------+

| Task Scheduler |

+----------------------+

|

+------------+-------------+

| | |

+------------+ +------------+ +------------+

| Conv Units | | Pool Units | | Act Units |

+------------+ +------------+ +------------+

| | |

+------------+-------------+

|

+----------------------+

| PE Array (MACs) |

+----------------------+

|

+----------------------+

| Local SRAM Buffers |

+----------------------+

|

+----------------------+

| External DRAM (DDR) |

+----------------------+

Block diagram for a single INT8 PE:

text

+----------------------+

| Input Register (8b) |

| Weight Register(8b) |

+----------------------+

| |

v v

+-----------------+

| Multiplier | (8b x 8b)

+-----------------+

|

v

+-----------------+

| Accumulator | (16b or 32b)

+-----------------+

|

v

[Optional: Activation/Bias]

|

v

+-----------------+

| Output Register |

+-----------------+

2. 16×16 Systolic Array Organization

8 rows × 8 columns = 64 PEs

Systolic array: Data (inputs and weights) flows rhythmically across the array, enabling high-throughput, pipelined computation.

Each PE communicates with its neighbors (e.g., passing partial sums right and down).

Block diagram (simplified):

Inputs x weights -->

+---+---+---+---+---+---+---+---+---+

|PE |PE |PE |PE |PE |PE |PE|PE|

+---+---+---+---+---+---+---+---+---+

|... |

+-----------------------------------------+

|PE |PE |PE |PE |PE |PE |PE |PE|

+---+---+---+---+---+---+---+---+-----+

Expected speedup

Understanding the "expected number of 16x16 multiplier operations" for a YOLOv5 inference is primarily relevant for hardware acceleration, especially on platforms like FPGAs or ASICs that feature dedicated DSP (Digital Signal Processor) blocks optimized for fixed-point arithmetic, often including 16x16-bit multipliers.

Here's how to approach this:

1. **Relate to MACs/FLOPs:** The computational complexity of a neural network is typically measured in **Multiply-Accumulate operations (MACs)** or **Floating Point Operations (FLOPs)**.
   * One MAC operation (one multiplication followed by one addition) is often approximated as **2 FLOPs** (one for multiplication, one for addition).
   * Many hardware accelerators, especially those targeting integer or fixed-point arithmetic, perform MACs efficiently. A single MAC on 16-bit integers would consume one "16x16 multiplier operation" if the hardware has such a dedicated unit.
2. **YOLOv5 Model Variants:** YOLOv5 comes in several sizes, each with different computational requirements:
   * **YOLOv5n** (Nano)
   * **YOLOv5s** (Small)
   * **YOLOv5m** (Medium)
   * **YOLOv5l** (Large)
   * **YOLOv5x** (Extra Large)

The computational cost (FLOPs/MACs) increases significantly with model size. For an input image size of **640x640 pixels**, the typical GFLOPs (billions of FLOPs) are:

* + **YOLOv5n:** ~4.5 GFLOPs
  + **YOLOv5s:** ~16.5 GFLOPs
  + **YOLOv5m:** ~49.0 GFLOPs
  + **YOLOv5l:** ~109.1 GFLOPs
  + **YOLOv5x:** ~205.7 GFLOPs

1. **Translating GFLOPs to 16x16 Multiplier Operations:**

Assuming that most of the FLOPs in a CNN come from convolutional layers (which are dominated by multiply-accumulate operations) and assuming you are targeting a 16-bit fixed-point or integer inference:

* + **Approximation:** Each MAC operation (which counts as 2 FLOPs) would correspond to **one 16x16 multiplier operation** on a hardware unit designed for that precision.

Therefore, to get an *estimated* number of 16x16 multiplier operations:

Number of 16x16 Multiplier Operations≈2GFLOPs​×109

Let's calculate for each YOLOv5 model at 640x640 input:

* + **YOLOv5n:** 24.5 GFLOPs​×109=2.25 billion 16x16 multiplier operations
  + **YOLOv5s:** 216.5 GFLOPs​×109=8.25 billion 16x16 multiplier operations
  + **YOLOv5m:** 249.0 GFLOPs​×109=24.5 billion 16x16 multiplier operations
  + **YOLOv5l:** 2109.1 GFLOPs​×109=54.55 billion 16x16 multiplier operations
  + **YOLOv5x:** 2205.7 GFLOPs​×109=102.85 billion 16x16 multiplier operations

**Important Considerations:**

* **Quantization:** These estimations assume the model has been quantized to 16-bit precision (e.g., FP16, INT16). If the model runs in FP32 (single-precision float), dedicated 16x16 integer multipliers wouldn't be directly used, or the operations would be emulated, which is less efficient.
* **Layer-Specific Operations:** While most operations are MACs, some layers (e.g., activation functions like SiLU, pooling layers, normalization layers) involve other operations (additions, divisions, comparisons, exponentials) that may not directly map to a 16x16 multiplier. However, for a high-level estimate of computational load, MACs (or FLOPs) are dominant.
* **Input Size:** The FLOPs (and thus multiplier operations) scale with the square of the input image size. For instance, a 1280x1280 input would quadruple the operations compared to a 640x640 input for the same model.
* **Batch Size:** These figures are per image. If you process a batch of images, the total operations will multiply by the batch size.

**Percentage of Time:** In this scenario, for any workload with significant multiplication (like a convolutional layer), the multiplication operations would dominate the execution time.

* **Estimate: 80% to 95%+ of the time.**
* **Reason:** If one multiplication takes 50-100 cycles, and an addition or load takes 1-3 cycles, the multiplication itself is disproportionately expensive. Even if you have 2 loads, 1 add, and 1 store for every multiplication, the multiplication cost would still be the overwhelming factor.

Actual Speedup

Instead of 100 clocks the accelerator takes 16 clocks. So acceleration of 85% x 6 = 500%

So effectively we will get a 5x acceleration on a RISCV Yolov5 inference operation.