Efficient Edge AI Inference on RISC-V based NPU (Neural Processing Unit)

Table of Contents

[0 Introduction 5](#_Toc203631631)

[0.1 Scope & Purpose of this book 5](#_Toc203631632)

[0.2 Why are we writing this book 5](#_Toc203631633)

[0.3 Target Audience 6](#_Toc203631634)

[0.4 What will you get by reading this 6](#_Toc203631635)

[1 Chapter 1 - Hardware Architecture 7](#_Toc203631636)

[1.1 Hardware Block Diagram 7](#_Toc203631637)

[1.2 Scalar Engine 8](#_Toc203631638)

[1.3 Vector Processing Unit (VPU) 9](#_Toc203631639)

[1.4 Matrix Processing Unit (MPU) 11](#_Toc203631640)

[1.5 Memory Hierarchy 13](#_Toc203631641)

[1.5.1 TCM vs Cache 14](#_Toc203631642)

[1.5.2 DDR Bandwidth 14](#_Toc203631643)

[1.5.3 On Chip RAM 14](#_Toc203631644)

[1.6 Data Movement Engines 14](#_Toc203631645)

[1.6.1 CPU Load/Store Unit (LSU) 14](#_Toc203631646)

[1.6.2 Vector LSU 15](#_Toc203631647)

[1.6.3 Matrix LSU 15](#_Toc203631648)

[1.6.4 Direct Memory Access (DMA) Engines 15](#_Toc203631649)

[1.6.5 Bringing it all together 16](#_Toc203631650)

[2 Chapter 2 - Inference vs Training 16](#_Toc203631651)

[2.1 Key differences 16](#_Toc203631652)

[2.2 Why is edge inference important 17](#_Toc203631653)

[2.3 Why is edge inference challenging 17](#_Toc203631654)

[3 Chapter 3 - WTF is “TOPS” and “Flops”? Does it matter? 19](#_Toc203631655)

[3.1 What is it 19](#_Toc203631656)

[3.2 How is it calculated 19](#_Toc203631657)

[3.3 Is it relevant 19](#_Toc203631658)

[3.4 Why are things not perfect? 20](#_Toc203631659)

[4 Chapter 4 – Structure of Neural Net 22](#_Toc203631660)

[Do we need this section? Readers may already know this. Let’s think what value we can add here 22](#_Toc203631661)

[4.1 What is a layer/op 22](#_Toc203631662)

[4.2 What is a graph 22](#_Toc203631663)

[5 Chapter 5 - Data Types and impact on performance 22](#_Toc203631664)

[5.1 Integer formats 23](#_Toc203631665)

[5.2 Floating point formats 24](#_Toc203631666)

[6 Chapter 6 - Tensor Layouts and impact on performance 25](#_Toc203631667)

[6.1 NCHW 25](#_Toc203631668)

[6.2 NHWC 25](#_Toc203631669)

[6.3 Others 25](#_Toc203631670)

[7 Chapter 7 - How are NN operations accelerated on NPU 25](#_Toc203631671)

[7.1 General matrix multiply (GeMM) 25](#_Toc203631672)

[7.1.1 GeMM on a scalar 25](#_Toc203631673)

[7.1.1.1 Gustavson’s method 26](#_Toc203631674)

[7.1.1.2 Outer Product Method 26](#_Toc203631675)

[7.1.2 GeMM on a VPU 28](#_Toc203631676)

[7.1.3 GeMM on an MPU 29](#_Toc203631677)

[7.2 2D Convolutions 31](#_Toc203631678)

[7.2.1 Accelerating Conv2D on MPU 32](#_Toc203631679)

[7.2.1.1 Im2col method 32](#_Toc203631680)

[There is an implicit im2col method, but we leave to the reader to explore it. 32](#_Toc203631681)

[7.2.1.2 Conv2D using Outer Product 33](#_Toc203631682)

[7.2.2 Other types of 2D convolutions 33](#_Toc203631683)

[7.2.2.1 Pointwise Convolutions 33](#_Toc203631684)

[7.2.2.2 Strided Convolutions 33](#_Toc203631685)

[7.2.2.3 Dilated Convolutions 33](#_Toc203631686)

[7.2.2.4 Grouped Convolutions 34](#_Toc203631687)

[7.2.2.5 Depthwise Convolutions 34](#_Toc203631688)

[7.2.2.6 Depthwise Separable Convolutions 34](#_Toc203631689)

[7.2.2.7 Point-Wise Convolution with Residual input layer 34](#_Toc203631690)

[7.2.2.8 Transposed Convolutions 34](#_Toc203631691)

[7.3 Activations 35](#_Toc203631692)

[7.4 Attention 35](#_Toc203631693)

[7.4.1 Multi Head Attention 35](#_Toc203631694)

[7.4.2 Group Query Attention 35](#_Toc203631695)

[7.4.3 Deformable Attention 35](#_Toc203631696)

[7.5 MLP 35](#_Toc203631697)

[7.6 Layernorm 35](#_Toc203631698)

[7.7 Softmax 35](#_Toc203631699)

[7.8 Transpose 35](#_Toc203631700)

[7.9 Others 35](#_Toc203631701)

[8 Chapter 8 – Optimization Techniques 36](#_Toc203631702)

[8.1 Graph modifications for performance 36](#_Toc203631703)

[8.1.1 Layer Fusion 36](#_Toc203631704)

[8.1.2 Constant Folding 36](#_Toc203631705)

[8.1.3 Pruning 36](#_Toc203631706)

[8.1.4 Quantization 36](#_Toc203631707)

[8.1.5 What else?? 36](#_Toc203631708)

[8.2 Compression 37](#_Toc203631709)

[8.2.1 Weight Compression 37](#_Toc203631710)

[8.2.2 Activation Compression 37](#_Toc203631711)

[8.3 Using Sparsity 37](#_Toc203631712)

[8.3.1 What is sparsity and where does it come from 37](#_Toc203631713)

[8.3.2 Structured vs unstructured 37](#_Toc203631714)

[8.3.3 Ways to leverage sparsity for acceleration 37](#_Toc203631715)

[8.4 Graph Sequencing Techniques 37](#_Toc203631716)

[8.4.1 Layerwise execution 37](#_Toc203631717)

[8.4.2 Depthwise execution 37](#_Toc203631718)

[8.4.3 Hybrid (assuming our disclosure if filed by the time this is published) 37](#_Toc203631719)

[8.5 Scalability 37](#_Toc203631720)

[8.5.1 Load distribution across multiple NPUs 37](#_Toc203631721)

[8.5.1.1 How does a GPU do it 37](#_Toc203631722)

[8.5.1.2 Why this is not right for NPU 37](#_Toc203631723)

[8.5.1.3 Then what is right for NPU 37](#_Toc203631724)

[8.5.2 Other scalability challenges 37](#_Toc203631725)

[9 Chapter 9 - Software 38](#_Toc203631726)

[9.1 The role of software 38](#_Toc203631727)

[9.2 Popular AI software frameworks 38](#_Toc203631728)

[9.3 Compiler based 38](#_Toc203631729)

[9.3.1.1 LLVM 38](#_Toc203631730)

[9.3.1.2 MLIR 38](#_Toc203631731)

[9.3.1.3 TVM 38](#_Toc203631732)

[9.4 How does software get best hardware entitlement 38](#_Toc203631733)

[10 Chapter 10 - Case Studies 39](#_Toc203631734)

[10.1 A simple 2-layer NN 39](#_Toc203631735)

[10.2 CNN – Alexnet/Resnet50/…. 39](#_Toc203631736)

[10.3 ViT 39](#_Toc203631737)

[10.4 LLMs 39](#_Toc203631738)

[10.4.1 Whisper 39](#_Toc203631739)

[10.4.2 Llama or DeepSeek or similar. Pick something suitable for Edge. 39](#_Toc203631740)

[10.4.3 DETR 39](#_Toc203631741)

[10.5 Some ADAS/Robotics networks 39](#_Toc203631742)

[10.5.1 BEVFormer 39](#_Toc203631743)

[10.5.2 BEVFusion 39](#_Toc203631744)

[11 Appendix A 39](#_Toc203631745)

[11.1 DDR bandwidth utilization 39](#_Toc203631746)

[11.2 Comparison of how a GeMM operation executes on scalar, vector and matrix engine. 39](#_Toc203631747)

[11.3 Arithmetic Intensity. 41](#_Toc203631748)

[12 References 42](#_Toc203631749)

# Introduction

## Scope & Purpose of this book

This book talks about performance architecture of **Embedded Inferencing NPUs for Edge AI SoCs**. We will take a holistic system view which will include hardware computation blocks, memory subsystem and of course software. While there are many hardware architecture options, we will assume a RISC-V based NPU for discussing architecture tradeoffs and performance analysis. The primary reason for that choice is the open nature of RISC-V which allows us to discuss it in an open forum like this. However, the contents of this book will be applicable to a wide range of matrix accelerator based NPUs (for e.g. [Texas Instruments C7 MMA](https://forum.digikey.com/t/ti-edge-ai-am6xa-processors-with-deep-learning-accelerators-and-its-efficiency/39813), [Qualcomm's Hexagon DSP based NPU](file:///C:/Users/a0875454/Desktop/Qualcomm's%20Hexagon%20DSP%20And%20NPU))

This book is organized into several chapters. In every chapter we will take one topic and explain it in an easy to understand way. We know you get bored fast ☺ . Therefore, each chapter is structured to be a few minutes of reading time. Over the course of the next few chapters, we will have learned about

1. Challenges and care-abouts of Embedded EdgeAI Inference.
2. NPU hardware architecture and how it interacts with an SoC.
3. Best practices to get maximum performance at lowest power.
4. Software’s role in achieving high performance.

We understand there are numerous other alternative NPU architectures for AI inferencing. To name a few

* GPU
* In-Memory Compute
* Quantum Compute
* Neuromorphic Compute

This book will not include them, but maybe a future one will.

## Why are we writing this book

While the internet is full of blogs, e-books, papers and videos on AI, we have felt that Embedded AI Inferencing is a topic covered very sparsely. This also applies to courses taught at universities and online education platforms. Most of the online literature focusses on training and/or GPU based architectures. At the same time we believe that low power, low latency inferencing on the edge will be the fastest growing area. Enabling this ecosystem will need a new generation of engineers who have deep knowledge of how the math and logic of AI is brought to life on low cost, small form factor hardware running on

* The cellphone in your pocket
* Or the car you are driving
* Or the watch you are wearing
* Or any other AI enabled device around you

Unfortunately this knowledge is limited to a few practitioners (like us ☺)) who have learned it the hard way trying to optimize AI models on embedded NPUs. With this book, our goal is to disseminate this knowledge and also get feedback from other experts in the field.

## Target Audience

This will be a great resource if you are

* A student of AI algorithms and computer architecture
* A practicing engineer, trying to dive deep into Edge AI
* A systems or SoC architect, with interest in PPA (power, performance, area) optimization of your IP/SoC
* Or you are just curious
* Or you are looking to hire us for your next big startup idea ☺

It is expected that the reader is already familiar with basic concepts of AI/ML and fundamentals of computer architecture. Knowledge of RISC-V architecture will be helpful, but not mandatory. There are some links in the references section which maybe good resources for deeper dives into these topics.

## What will you get by reading this

We know you are busy and there is a ton of content on AI already available. This book is designed to be unique in the sense that it will go deep into performance optimization of AI inferencing on resource constrained embedded SoCs. By the time you are done reading this book, you would have developed a good understanding of

* AI inference performance bottlenecks on embedded SoCs.
* Hardware design tradeoffs like Matmul Array Size, Vector Length, External Memory bandwidth, Internal SRAM size etc.
* AI model tradeoffs like pruning, quantization, layer fusion and hardware specific graph modifications.
* Best practices and software defined data-flows to achieve the most juice from the hardware.

# Chapter 1 - Hardware Architecture

## Hardware Block Diagram

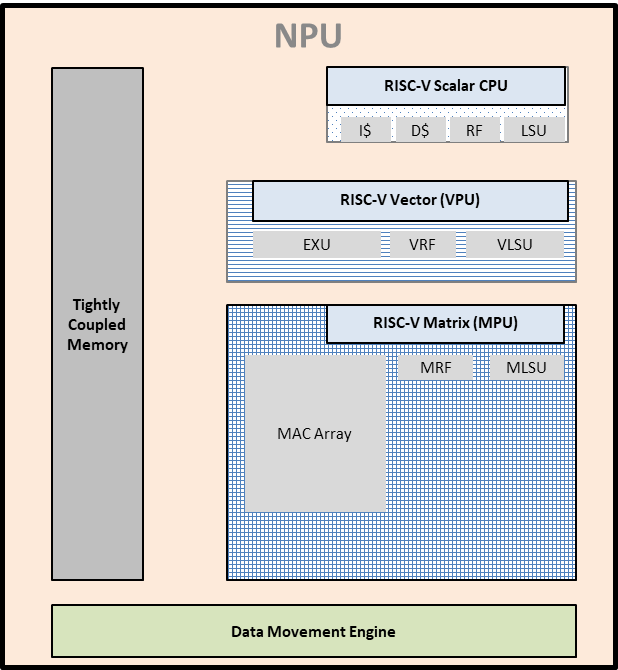


Figure 1 : Block Diagram of a RISC-V based NPU

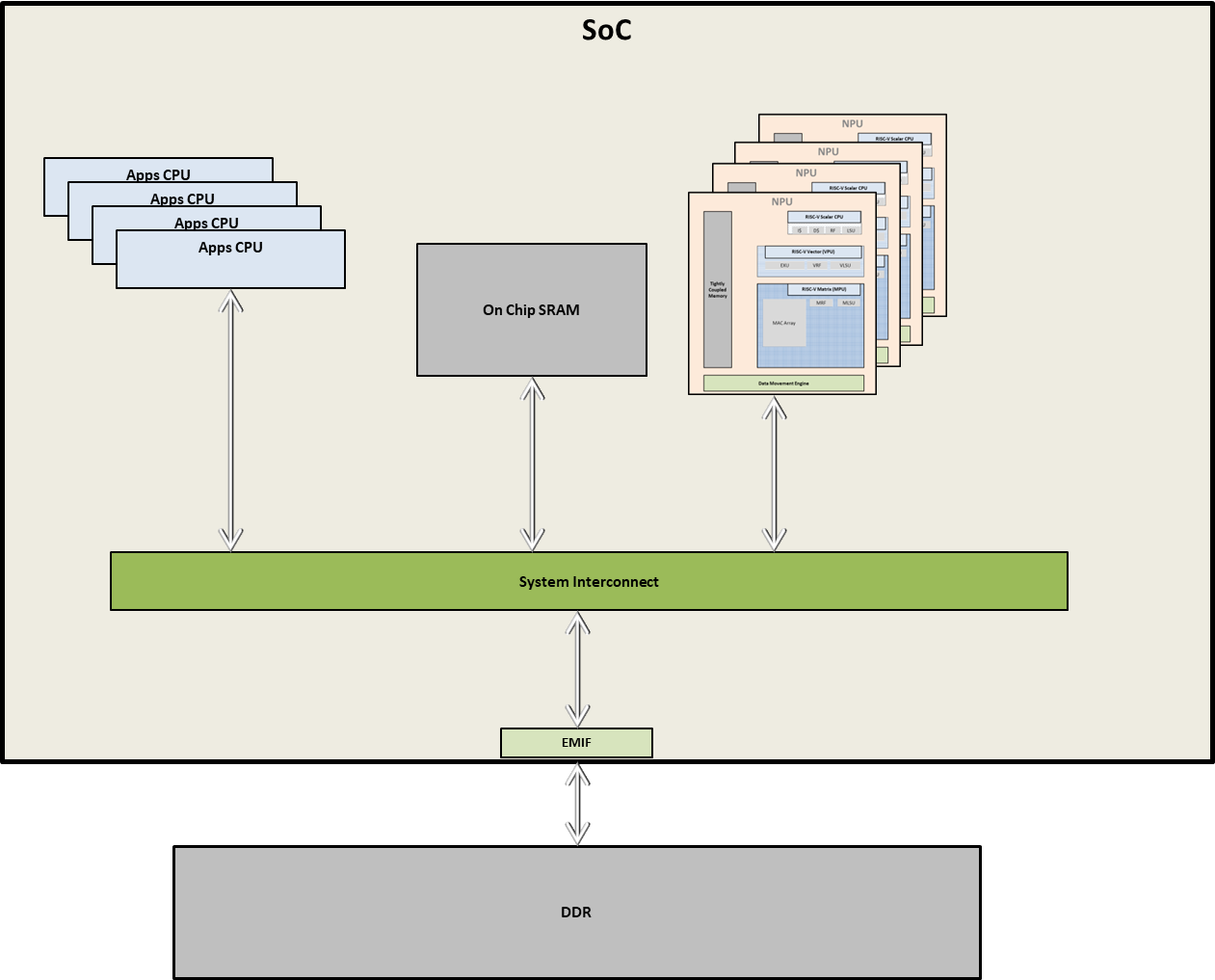


Figure 2 : Example of a SoC with an Apps CPU Cluster, NPU cluster, On-Chip SRAM and External Memory Interface to DDR

In the architecture assumed here, the NPU acts as an accelerator which offloads the Apps CPU from the heavy computational load. An example of an Apps CPU is ARM Cortex-A, or a 64-bit RISC-V core, typically running an Operating System like Linux, QNX or RTOS. In the typical usage model, Apps CPU hosts the application (e.g. Face ID), receives the data (e.g. camera frames) and sends it to the NPU for inference computation (e.g. MTCNN face recognition). Apps CPU then goes back to running OS and apps while the NPU is running math heavy computations. Once the NPU is done, it informs the Apps CPU and the processed results are passed on to the application which requested it.

In the meanwhile, NPU will need additional data from off-chip memory (e.g. DDR) or from on-chip SRAM. It accesses the memory using its own data movement engines so that the Apps CPU is not disturbed.

The choice of architecture is motivated by the following principles:

* In a typical AI model, computational load is heavily dominated by matrix multiplication operations. Matrix Multiplication Engine (**MME**) is a hardware block custom designed to process 2D matrices, with **M**x**N** operations in parallel. Hence it can perform these operations at a very high speed and low power. *M and N govern the matrix MAC array dimension and are important design decisions affecting PPA.*
* Since MME is custom designed for matrix multiplication, it cannot process other operations like addition, min/max etc. These operations are offloaded to a Vector (**VPU**) which offers 1-dimensional parallelism.
* VPU has a limited instruction set, and many operations are not good candidates for data parallelism. This is where **scalar** comes handy. Additionally, scalar core is responsible to run control software and moving data from cache/TCM to register files.
* AI operations are memory intensive. To achieve full utilization of compute elements, data must be moved in/out of registers fast enough. **TCM** is a memory sitting close to the CPU, often with same latency as a cache hit. However, it has lower complexity than cache because of no cache management and coherence to worry about.
* Data Movement Engine (DME) is responsible for bringing data from On Chip SRAM to TCM. DME takes care of bringing small tiles from big tensors which may be stored in a multi-dimensional layout.

In the next section, we take a closer look at each of the hardware modules.

## Scalar Engine

As described earlier, scalar is responsible for graph execution and issuing instructions for scalar’s compute elements (ALU, FPU etc.) as well as VPU and MME. In some cases, scalar’s LSU (Load store unit) may also be responsible for moving data from TCM/cache to/from VPU’s register files. Therefore, the scalar plays a pivotal role in AI graph execution. Further, there are many operations which do not fit within VPU or MME’s instruction set. Hence, these must be executed on the scalar. Some examples of such operations are

* Non Maximal Suppression
* ArgMax, ArgMin
* AffineGrid
* Logical Operators (And/Or/Xor)
* Reshape
* Flatten
* ..and many more

For best performance, it is expected that an AI graph will have very few of such operations. For e.g. NMS is used at the end of the graph to find the best bounding box. Most of the heavy lifting is done by MPU and VPU, therefore the scalar engine does not need be an ultra-high performance CPU core, but there are a few attributes which can help get maximum performance from the NPU

1. Instruction level parallelism – helps in executing load/store, scalar, vector and matrix instructions in a pipelined fashion, keeping all HW blocks occupied as much as possible.
2. Multi-threading – Low latency context switches.
3. FPU – Floating Point Unit for generating final probabilities and classification scores.

Some of the NPUs use a DSP as the scalar engine, which commonly use VLIW for Instruction Level Parallelism. Similar results can be achieved on RISC-V cores with hyper-scalar pipelines.

*For a comparison of VLIW vs SMT, please refer to* [*https://www.cs.umd.edu/~meesh/411/CA-online/chapter/multiple-issue-processors-ii/index.html*](https://www.cs.umd.edu/~meesh/411/CA-online/chapter/multiple-issue-processors-ii/index.html)

Some DSPs also support multi-threading, for e.g. Qualcomm Snapdragon 8 Gen 2, Hexagon has 6-way SMT.

RISC-V supports up to 8-way SMT. <https://www.youtube.com/watch?v=QR4niawYSVI>

## Vector Processing Unit (VPU)

Vector processing is a type of data level parallelism where a single instruction operates on multiple data elements simultaneously. In concept, it is very similar to SIMD (e.g. ARM Neon, Intel AVX-512) but there are some subtle but critical differences. Refer to [RISC-V Vector vs ARM Neon SIMD](RISC-V%20Vector%20vs%20ARM%20Neon%20SIMD)

Key design time attributes of a VPU based on RISC-V are

**VLEN** – Register width (in bits).

**ELEN** – Widest supported element size.

**LMUL** – Lane multiplier for register grouping. Allows packing multiple registers together to operate more number of elements in parallel.

**SEW** (Selected Element Width) is a runtime configuration to choose the bitwidth of the operands. This gives us the maximum number of elements which can be processed together:

**VLMAX = (VLEN\*LMUL)/SEW**

VLEN, ELEN and LMUL are important design decisions, impacting performance, die area and therefore the cost & power of the VPU.

Vector engines offer not only faster execution but also code size reduction. Let’s take an example of adding two arrays of length n.

void vvaddint32(size\_t n, const int\*x, const int\*y, int\*z){

for (size\_t i=0; i<n; i++)

z[i]=x[i]+y[i];

}

Figure 3 : C code of array addition on scalar

This loop has n additions, increments of i, comparisons between i and n, and branching logic. When compiled, it will result in 36 instructions.

On RISC-V Vector, it compiles into 11 instructions as shown below

vvaddint32:

vsetvli t0, a0, e32, ta, ma # Set vector length based on 32-bit vectors

vle32.v v0, (a1) # Get first vector

sub a0, a0, t0 # Decrement number done

slli t0, t0, 2 # Multiply number done by 4 bytes

add a1, a1, t0 # Bump pointer

vle32.v v1, (a2) # Get second vector

add a2, a2, t0 # Bump pointer

vadd.vv v2, v0, v1 # Sum vectors

vse32.v v2, (a3) # Store result

add a3, a3, t0 # Bump pointer

bnez a0, vvaddint32 # Loop back

ret # Finished

Figure 4 : Assembly code of array addition on RISC-V Vector

<https://github.com/riscvarchive/riscv-v-spec/blob/master/example/vvaddint32.s>

We see more than 3x improvement in code density. On a 256-bit vector, vadd.vv instruction processes 8 elements at a time since data type is int32.

A lot of AI operations like “*add*”, “*concat*” etc. can be done effectively on vector. RISC-V Vector also supports permute instructions which can be used for table lookup. This is extremely helpful in running ops which include exponentiation and logarithmic functions. Softmax, used heavily in transformer networks, is a good example.

## Matrix Processing Unit (MPU)

Also known as “Tensor Processor” on some NPUs, this block offers data parallelism across two dimensions compared to a vector which offers 1D parallelism. Matrix array size is a key design decision. Larger matrices can help boost performance, but result in higher area and power. When matrix array becomes very large, feeding it with valid input can become challenging and may cause under-utilization of processing elements. We will visit this topic in more detail in section 8.1.

Basic operation of an MPU is explained below,

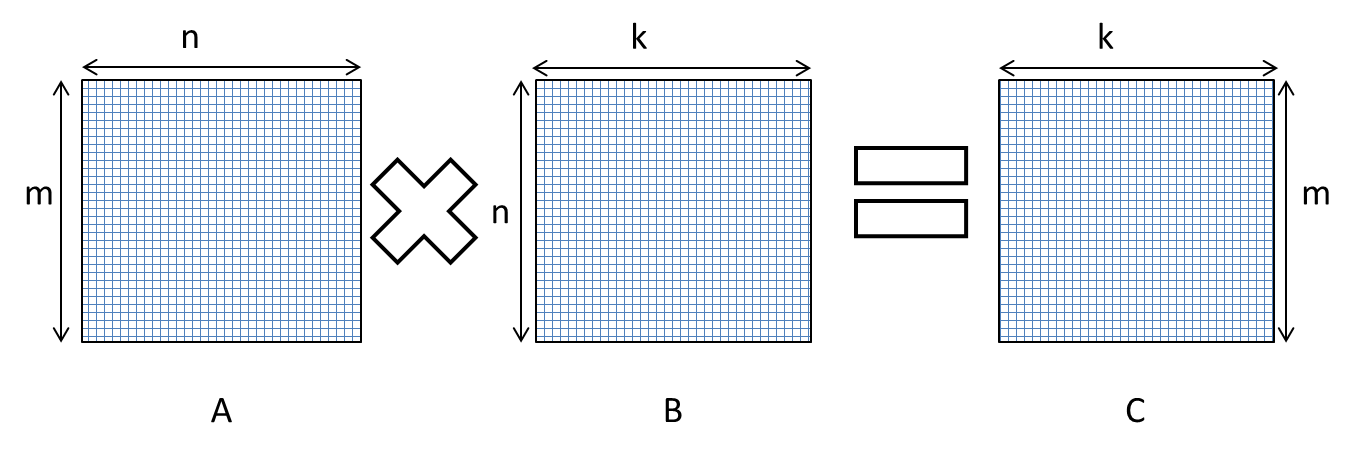


Figure 5 : MPU matmul

There are several matrix extensions proposed in RISC-V forum. The two most prominent are

1. **IME (Integrated Matrix Extension)** – Closely coupled with Vector. Matrix and Vector registers are shared. Matrix introduces no additional architecture state, making it easier for adoption in OSes (e.g. Linux). MAC array size is constrained to vector length, and it becomes a challenge to process computationally intensive workloads. Despite these limitations, this architecture is well suited for AI enabled Apps CPUs.
2. **AME (Attached Matrix Extension)** – Decoupled from Vector. Matrix has its own registers and accumulators. Designers have more freedom to implement schemes like “outer-product matmul” on larger MAC arrays. More suitable for NPU accelerators where high performance (/watt and /mm2) is the goal.

A lot of topics we will discuss in this book are applicable to both the architectures, but the authors assume AME wherever there is a divergence.

[RISC-V AME spec 0.5b](https://lists.riscv.org/g/tech-attached-matrix-extension/attachment/210/1) defines:

1. **ELEN** – Widest supported element size. Similar to ELEN in Vector.
2. **MLEN** – Matrix tile size (in bits).
3. **RLEN** – Matrix row size (in bits).
4. **AMUL** - multiple of length for matrix accumulation registers. Similar to LMUL in vector. AMUL can be used to group registers to operate on wider elements, up to 8x. TBD : Confirm

**SEW** = Selected element width, similar to as in Vector.

An Accumulation Register has MLEN\*AMUL bits of state, where each row has RLEN\*AMUL bits. As a result, there are MLEN/RLEN rows for each accumulation register in logic.

This leads to the following constraints

1. **TMMAX**, max value of m in Figure 5 = MLEN/RLEN
2. **TNMAX**, max value of n in Figure 5 = RLEN/SEW
3. **TKMAX**, max value of k in Figure 5 = min(TMMAX, TNMAX)

As an example, for MLEN=512, RLEN=64, SEW=8

TMMAX = 8, TNMAX = 8, TKMAX = 8. This means that MPU is designed to perform 8x8x8 matmul on INT8/FP8 data.

<https://lists.riscv.org/g/tech-attached-matrix-extension/attachment/210/1/riscv-matrix-spec-v0.5b-64bit-encoding.pdf>

Before MME can process the matrices, they must be loaded to Matrix Register File. This is done by matrix LSU. Since register file space is small and precious, entire matrices are not loaded at once. Instead big matrices are divided into tiles of size *mtile\_m x mtile\_k x mtile\_n*. The loads are done row-wise for matrix A and column-wise for matrix B.

The basic operational steps for compute dot product of 1 tile are

1. Load row #i, from tile A.
2. Load column #j from tile B.
3. Compute outer product. The result in accumulated in matrix registers.
4. Repeat steps 1-3 *mtile\_k* number of times.

Step #3 above is where computation happens. A typical MPU can execute this in a single cycle. Hence, the loop above computes tile A and B dot product in *mtile\_m* x *mtile\_n* cycles.

The total number of computations for a tile are *mtile\_m* \* *mtile\_k* \* *mtile\_n* multiplies and adds i.e. total **2\* *mtile\_m* \* *mtile\_k* \* *mtile\_n*** math operations, done in ***mtile\_k*** cycles. Thus MPU achieves a speedup of **2\* *mtile\_m* \* *mtile\_n*** ops/cycle. Full throughput is achieved when

* *mtile\_m =* TMMAX/SEW
* *mtile\_k =* TKMAX
* *mtile\_n =* TNMAX

Substituting the values of TMMAX, TNMAX and TKMAX, we compute maximum throughput achievable =

2\* ((MLEN/RLEN)/SEW) \* (RLEN/SEW) = **2\*MLEN/SEW2**

A MPU with 4096-bit MLEN, running at 1GHz, operating on 8-bit data has a peak performance rating of

*2 \* 642 \* 1G = 8 Trillion Operations Per Second, commonly expressed as* ***8 TOPS****.*

## Memory Hierarchy

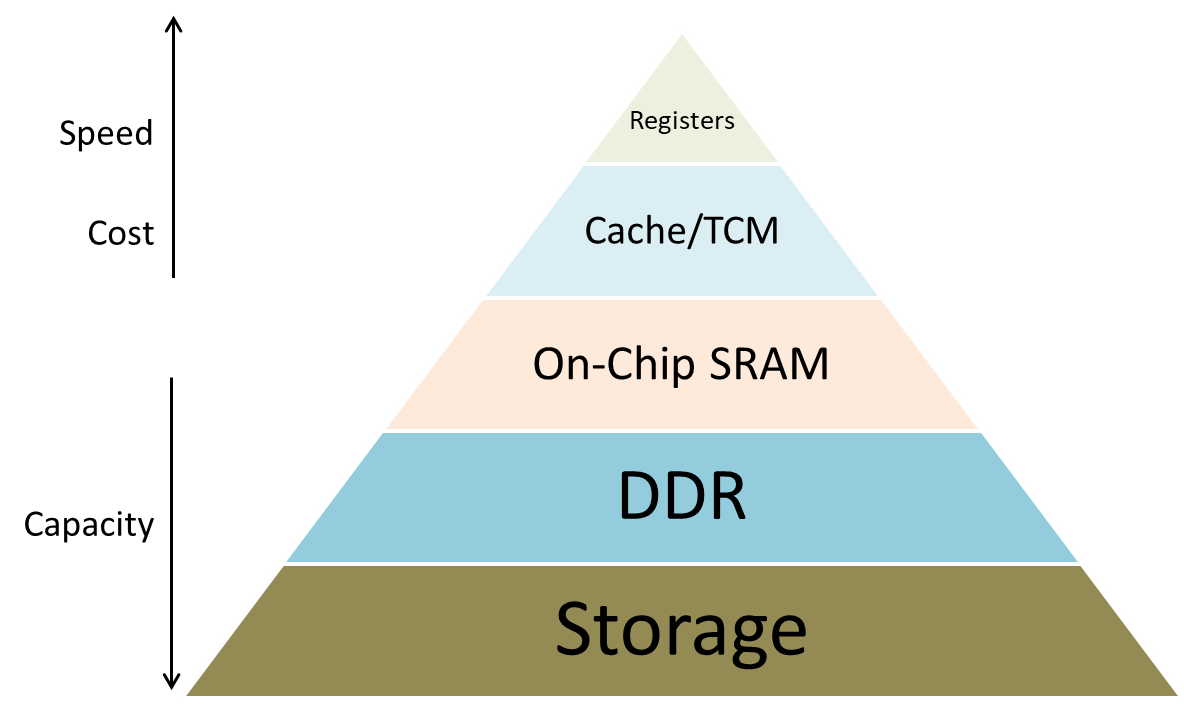


Figure 6 : Typical Memory Hierarchy in an embedded SoC

A hierarchical memory organization helps in a balanced PPA by arranging different types of memory in layers. The table below shows the speed vs cost vs data movement energy comparison of different types of memories

| **Memory Type** | **Cost/byte** | **Speed** | **Energy/byte** |
| --- | --- | --- | --- |
| Registers | Highest | Fastest | Lowest |
| TCM/Cache (L1,L2,L3) | High | Fast | Low |
| On Chip RAM/LLC | Mid | Mid | Mid |
| DDR | Low | Slow | High |
| Storage | Lowest | Slowest | Highest |

This places an economical (and also physical) limit on capacities of registers files, caches and on-chip RAM. To achieve best performance, data must be moved in small chunks from big memories to fast memories close to execution units. The data must be evicted when it is no longer needed so that space can be created for the next tile of data. Detailed discussion of different types of memory is outside the scope of this book. However, we will discuss a few important topics which we believe are highly important to understand performance challenges of AI inference.

### TCM vs Cache

A TCM (Tightly Coupled Memory) sits close to the execution units and offers access latency comparable to L1 cache hit. The major differences between TCM and cache are

1. TCMs are mapped in system memory map, while cache is not.
2. Cache can be accessed only by the CPU, while TCMs can be accessed by other masters, including DMA engines in the SoC.
3. TCMs can be multi-port, which allows other cores to access TCM of one core, although at a higher latency.
4. TCMs have lower overheads, since there is no coherence, no eviction policy and no tags to be maintained.

Therefore, TCMs are preferred to caches to store AI data. The execution order in an AI graph is highly predictable and software controlled management is quite effective.

### DDR Bandwidth

DDR bandwidth, expressed in GB/s, is the rate at which data can be transferred between the SoC and the DDR. This is a critical factor in any embedded SoC, but becomes even more critical in AI because of heavy data intensive nature of workloads. If the data cannot be moved fast enough to execution units, it cause stalls which results in lower performance. DDR bandwidth is a shared resource between all the subsystems inside a SoC including Apps CPU, ISP (Image Signal Processor), DSS (Display Sub System), VSS (Video Subsystem), NPU and many more. Therefore an NPU which can minimize off-chip data transfers usually results in higher performance in real-world use cases.

### On Chip RAM

On Chip RAM acts as a buffer between TCM and DDR. This memory can be shared by all the subsystems, including all the NPUs (*if the SoC has multiple NPUs*). To best utilize DDR bandwidth (as discussed in the previous section), it is beneficial to

1. Access DDR in big chunks. Helps more efficient reads and writes.
2. Increase data reuse to keep the tiles longer in the SoC internal memory. This will be clearer after the tiling chapter.
3. Share data (weights, intermediate results) between NPUs.

Thus, a well sized SRAM can reduce DDR traffic and make DDR access more optimal. However, SRAM has cost and area impact which imposes constraints on how big SRAM can be placed inside the SoC. Therefore, SRAM sizing is an important design decision affecting PPA. In section <TBD> we explore a HW-SW co-design methodology to come up with an optimal SRAM size for the desired system performance.

## Data Movement Engines

### CPU Load/Store Unit (LSU)

In a load-store architecture, CPU performs logic and math operations on the operands in the registers. LSU is responsible for

1. Loading the operands from memory to CPU’s register files.
2. Writing the results from register files to memory.

In most modern CPUs ld/st instructions are pipelined to run concurrently with math and logic instructions, but they can still cause pipeline stalls for e.g. in case of a cache miss.

### Vector LSU

VLSU is a fully pipelined LSU unit to move a block of data between memory and Vector Register File. It supports three types of addressing

1. Unit Stride – Fastest
2. Non-unit, but constant stride
3. Indexed (gather-scatter) – Slowest, but helpful for operations like table lookup etc. For e.g. activation functions like *tanh*, *sigmoid* etc. can be specified through a lookup table and executed on vector via indexed addressing.

### Matrix LSU

Keeping the MAC arrays fed with data is a challenging task. The usual way of CPU’s LSU reading from cache will not be able to provide sufficient bandwidth and will often cause stalls because of cache misses. Therefore, a better solution is needed.

AI data has some special attributes, for e.g.

1. Multi-dimensional tensors, strided access. Not cache friendly.
2. Skip pattern reads.
3. Packed formats (e.g. two INT4 weight packed in one byte).
4. Memory access pattern is predictable. Therefore it is a good practice to bypass data caches.

This creates the need for an AI specific data movement engine which can read a “tile” of data from memory and feed the MPU registers without involving the CPU. The tile is expected to be a small portion of a multi-dimensional tensor; hence the data movement engine must be capable of address generation using tensor dimensions.

RISC-V (proposed) matrix extension defines matrix load and store instructions but does not specify LSU behavior. For best performance it is critical that Matrix LSU can fetch the data directly from memory, offloading the CPU. It is also recommended that Matrix LSU reads/writes to a TCM or on-chip SRAM instead of DDR. How does the data move between DDR and TCM/SRAM? See the next section.

It must be noted that for VLSU and MLSU to feed the execution units at the desired rate, Memory system must provide enough sustained bandwidth of (# lanes x SEW) /clock cycle.

### Direct Memory Access (DMA) Engines

A DMA engine can move data from almost anywhere to anywhere in the SoC, as long as source and destination are mapped in system memory map. In most high performance compute (e.g. multimedia, AI etc.), DMAs are commonly used to move inputs, outputs and parameters between DDR and SoC internal memories (e.g. SRAM). This is a faster, cheaper and power efficient method to offload the CPU and other processing elements from data movement. DMA can work in parallel with compute, which allows a pipelined architecture.

An example of pipelining is where a tensor is broken down into small tiles. At any given time:

1. Compute elements are processing tile #N, reading and writing to SRAM.
2. DMA is writing out the results of tile #N-1 from SRAM to DDR.
3. DMA is proactively fetching input(s) for tile #N+1 from DDR to SRAM.

If there is only one DMA, steps #2 and #3 may be done sequentially. The total execution time will be

*max(t1, t2+t3)*

Therefore, to get best utilization of the precious compute elements, memory movement must be faster than compute. This is where DMAs excel.

### Bringing it all together

1. The full AI model (millions or billions of parameters), resides in DDR.
2. Every few milliseconds, a new input tensor like camera image is written to DDR.
3. SoC DMA moves a block of weights and activations from DDR to SRAM.
4. Vector/Matrix LSU moves smaller tiles from SRAM to register files.
5. Computations are done in the register space.
6. Vector/Matrix LSU stores the results in SRAM. These results may be reused for next layer of processing. Therefore, the system tries to keep the intermediate results in SRAM as long as possible.
7. When SRAM is full, or the final results are available, SoC SMA writes a big block of data to DDR.

This is how multi-level memory hierarchy and data movement architecture results in most optimal performance, power and cost.

# Chapter 2 - Inference vs Training

## Key differences

AI Inference is the process of running new (previously unseen) data through a pre-trained model to ‘infer’ what the model was trained to do. For e.g. a model to detect human faces in an image would have been trained on a dataset of positive and negative training images. In inference, the system takes a completely new image (e.g. from a live camera) and feeds it to the pre-trained model. The model will try to infer if the image contains any human faces, and the size/position of the faces found. From a computational complexity and system performance point of view, the key differences between training and inference are

1. Training has a backward and a forward pass. Inference has only **forward pass**.
2. Training has many many many epochs. In each epoch the model weights are updated, model outputs are computed and compared with the reference to evaluate the cost function which is to be minimized. In inference, you get only one chance to get it right. Therefore, inference computation complexity is lower, but timing and power budgets are much tighter.
3. Training is done on **large batches**. Batching can result in better utilization of compute elements and therefore higher throughput, but comes at the cost of increased latency. Inference usually requires low latency and therefore batches are small, commonly set = 1. This is a tradeoff which embedded system designers must be aware of.
4. **Data formats** – training is usually done on floating point data. The most important reason behind this is that most optimizers (like Gradient Descent, Adam etc.) need differentiable cost functions. Integer functions are not differentiable and therefore floating point is a better choice. For inference, we don’t have that constraint which makes it possible to quantize the weights and activations to fixed-point. Fixed point hardware is simpler and cheaper than floating-point. This allows designers to pack more multiply-add units in the same area and power budget. FP32 is the most commonly used format for training, but newer formats like BF16, FP8 etc. are gaining popularity. In inference, INT8 has been the most commonly used format, but INT4 is gaining traction and many inference NPUs now support INT4 format. More on that in chapter 5.

## Why is edge inference important

While training gets all the media attention, inference is really the end goal of AI. This is where AI solves real world problems like self-driving cars, speech recognition, health-monitoring and generating new content.

Why can’t all the inference run on cloud? A few reasons

1. **Latency** – For e.g. a self-driving car has only a few 100s of milliseconds to get an image from front camera, detect objects in its path and take a corrective action (braking, lane change etc.). It simply does not have the time to send the image to a cloud server and wait to get the results. All of the decision making must be done quickly and the only way to do it is by running inference on an onboard AI chip.
2. **Privacy** – If you are comfortable with the big tech collecting your voice, camera and financial data then cloud inference is for you. If you are as paranoid as I am, you need Edge Inference where your data never leaves your personal device.
3. **Cost** – Did you look at your AWS bill last month? What if you could invest in a device with a highly capable inference accelerator running your chosen models and you don’t have to make the Mag7 richer than they already are.

## Why is edge inference challenging

While model complexity is increasing and new frontier models are released every week, computational capacity is not increasing as quickly as before. <jensen-huang-says-moores-law-is-dead>

How does it affect Edge AI inferencing? Let’s look at the constraints in an embedded SoC

1. **Limited Computational Resources** – Training a model is a one-time cost. Inference is an operational cost that is incurred with every request. To keep the cost down, inference chips have very tight area budgets. This impacts all the computational elements in the chip including CPU, Vector, Matrix and the internal memory subsystem.
2. **Memory Bandwidth** – AI workloads have always been memory intensive. Let’s take example of a Resnet-50 with 224x224x3 input. This model generates <TBD> bytes of intermediate outputs per inference. To run this model @ 30 frames/second requires <TBD> of memory bandwidth. To make things worse,
   1. Input resolutions are increasing. Gone are the days of 224x224 (0.05 Mega Pixel) images. For e.g. a self-driving car uses Front Camera with a resolution somewhere between 5-8 Mega Pixels.
   2. Newer models (like Transformer based) have a much lower [arithmetic intensity](https://www.iguazio.com/glossary/arithmetic-intensity/) than CNNs. WTF is arithmetic intensity? In simple words, it is the ratio of math operations vs memory access in a given workload. The impact of low arithmetic intensity is that the SoC spends more time moving data and there is a risk that the precious compute elements sit idle waiting to be fed.
   3. Memory bandwidth is a **shared resource** in a SoC. Usually, there are higher priority masters on the bus which get precedence over NPU. An example is a camera capture device (e.g. CSI-2) writing to DDR, or a Display device reading from DDR. These are real-time masters which usually get priority over NPU.
3. **Latency** – As we discussed in the previous section, keeping low latency is critical in many applications. This requires system designers to balance the competing priorities between throughput and latency to design a well-balanced system.
4. **Power** – Power is arguably the most important factor is designing an embedded system.
   1. Very often such systems are battery operated and SoC power draw is among the most significant contributors to **battery drain**.
   2. **Thermal management** – High power results in heat dissipation which can damage the electronics, or reduce the lifetime. Therefore the devices are either actively or passively cooled. The system cost increases significantly with the cooling method. Liquid cooling removes heat the fastest but is also most expensive. Next comes air cooling (fans) followed by passive cooling (heat sinks). Designing a low power SoC helops achieve longer lifespan with simpler cooling system, resulting in very high cost savings.
   3. **PMIC cost** – Sometimes SoC need an external chip called PMIC (Power Management IC) to provide power. If the SoC needs more power (hence higher current) it may need multiple PMICs, or more expensive ones. Therefore, a low power SoC can result in PMIC cost savings as well.

# Chapter 3 - WTF is “TOPS” and “Flops”? Does it matter?

## What is it

* **FLOPS** = Floating Point Operations Per Second. This is the peak throughput of an NPU/GPU when all the operations are done on floating point operands.
* **TOPS** = Trillion Operations Per Second. Same as Flops but more generic since the operands maybe floating-point or fixed-point.

Both of these are metrics used by HW vendors to advertise the computational capability their AI accelerator. However, FLOPS is more relevant for training accelerators while TOPS is used commonly for inference accelerators and almost always refers to INT8 TOPS. The reason being INT8 multipliers are the smallest (compared to INT16 and floating point) and therefore a higher spec can be achieved with a smaller die.

## How is it calculated

Number of INT8 operations possible in a MPU + VPU based NPU

* MPU : 2 x M2 x F
* VPU : 2 x (VLEN/8) x F

Where

M = MPU MAC array size in INT8 elements

VLEN = VPU Vector Length in bits

F = Frequency

The multiplier “2” comes from the assumption that a MAC operation can be done in a single cycle. MAC includes a multiply and add, and therefore is equivalent to 2 math operations.

An example of an NPU with 64x64 MPU and 512-bit VPU operating @ 2GHz has a TOPS rating of

*((2 x 642 ) + (2 x 512/8) x 2x109)/1012 =* ***16.64 TOPS***

As you can see, most of the TOPS come from MPU and a small fraction from VPU. Technically, scalar can also run some math operations but the overall contribution is negligible and therefore ignored from TOPS calculations.

## Is it relevant

The short answer is Yes. This is a relevant metric because it gives an indication of how much computational workload the NPU is capable of if *everything else is perfect*. What does “everything else” mean? Since the TOPS spec is dominated by MPU throughput achieving high utilization is possible only if

1. MPU always has sufficient parallel workload to utilize M2 multipliers
2. MPU is never starved for data
3. MPU, VPU, Scalar, LSUs and DMAs are perfectly parallelized and MPU is the long pole. [Amdahl's law](https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/speedups.and.amdahls.law.1pp.pdf)

## Why are things not perfect?

Let’s take an example. Shown below is the architecture of Resnet50 which is a Convolutional Neural Net designed for image classification.

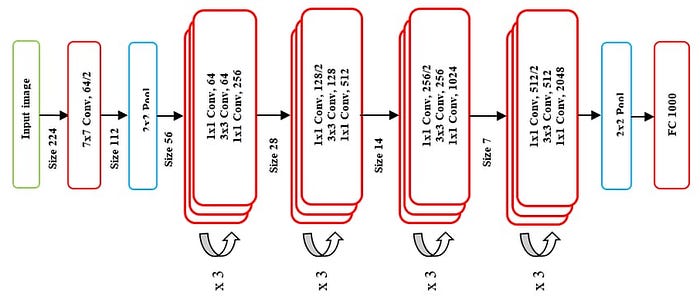


Figure 7 : Resnet50 Model Architecture

*Source : https://medium.com/data-science/the-annotated-resnet-50-a6c536034758*

The model takes a RGB image of size 224x224 pixels and outputs class probabilities. If we add up all the math operations in the model, it comes up to roughly 4 Billion MACs = 8 Million Operations per inference. So if we run this network on an 8 TOPS accelerator, we should get 1000 inferences/second, right?

Wrong!!!!!!!!!!!!

Let’s look at a few examples

1. NVidia H200 has 3958 INT8 TOPS. Resnet50 expected performance is 494750 fps. Actual performance (<https://developer.nvidia.com/deep-learning-performance-training-inference/ai-inference>) is 21253 fps, a utilization of almost 4%. And this is with batchsize=8. Throughput with batchsize=1 is expected to be much lower.
2. Texas Instruments’ appnote (<https://www.ti.com/lit/an/spracz2/spracz2.pdf>) shows that Resnet50 performance on TDA4VM 8TOPS NPU is 162fps. This is 16.2% of what we would expect.
3. Qualcomm Snapdragon 8 Elite Mobile features a 45 TOPS NPU. Resnet50 expected performance is 5625 fps. Actual performance as per <https://aihub.qualcomm.com/models/resnet50> is 1876 fps, which shows about 33% utilization.

Why is the utilization not 100%? Here are a few reasons

1. A real world AI workload will include many operations, not only matrix multiplication. Non-matmul operations (like Add, Pool etc.) must be done on VPU. Pipelined architectures allow VPU and MPU to run concurrently (as in hyper-scalar, VLIW etc.) in which case the long pole becomes the bottleneck. Therefore tuning VPU and MPU design parameters to the workload becomes critical to achieving high TOPS utilization.
2. Larger array helps achieve high TOPS rating (remember 2xM2xF) but if the input matrices are small, a big MAC array is not fully utilized and the real work done is a small fraction of the theoretical limit. For e.g. in the model architecture shown above, last few layers have a very low spatial resolution (28x28, 14x14, 7x7). On a 64x64 array, the utilization will be less than 50%, 25% and 12%. Imagine what happens if the marketing manager decides to increase MAC array size to 128x128!!! It will increase TOPS spec by 4x, but the % utilization will drop further, resulting in almost no increase in real world performance.
3. And finally the memory wall problem. See the section 11.2 on Arithmetic Intensity.

Processing huge tensors with millions/billions of parameters requires moving the inputs, parameters, intermediate results and final outputs between NPU and memory. As we saw in memory hierarchy section, DDR is the largest but also the slowest. Unfortunately, on-chip memories are not large enough to hold the entire model and input/output tensors. Therefore, model parameters and input/output must be broken down into small “tiles” and moved back and forth between on-chip memory and external DDR. Just to give an example, Resnet-50 example above generates <TBD> Mega Bytes of intermediate data per inference. Running this @30 inferences/second will result in <TBD> GB/s traffic. Sounds scary, right? Now imagine a self-driving car where the input resolution goes to a few Mega Pixels and there are multiple cameras feeding to the NPU cluster simultaneously.

Memory bandwidth, both within the chip and between the chip and DDR, is limited. Therefore, data movement sometimes takes longer than compute and NPU has to sit idle waiting for data to arrive.

A detailed analysis of hardware design choices and the workload characteristics is recommended for a balanced NPU. Design decisions without considering system-level challenges often result in additional area & power with no improvement in performance. As an example, simply doubling the frequency will double the TOPS spec (remember 2xM2xF) but if nothing is done about memory bandwidth, frequency increase will not result in proportionate increase in real world performance.

In a nutshell, TOPS (or TFlops) is a relevant metric, but it does not paint the complete picture. NPU architecture and the SoC architecture play a big role in system level performance. A more relevant metric is “inferences/second”. ***Inference*** can refer to a camera image in case of image classification, or a token in case of a language model.

Having said that, inferences/sec is not a standardized unit of work since every NN model will have a different computational load, different arithmetic intensity and a different ratio of matmul vs other ops. Therefore, architecting or selecting the ideal NPU is often very application specific. An NPU which is efficient for a CopilotTM PC may be extremely inefficient for a self-driving car, even though the TOPS rating maybe the same.

# Chapter 4 – Structure of Neural Net

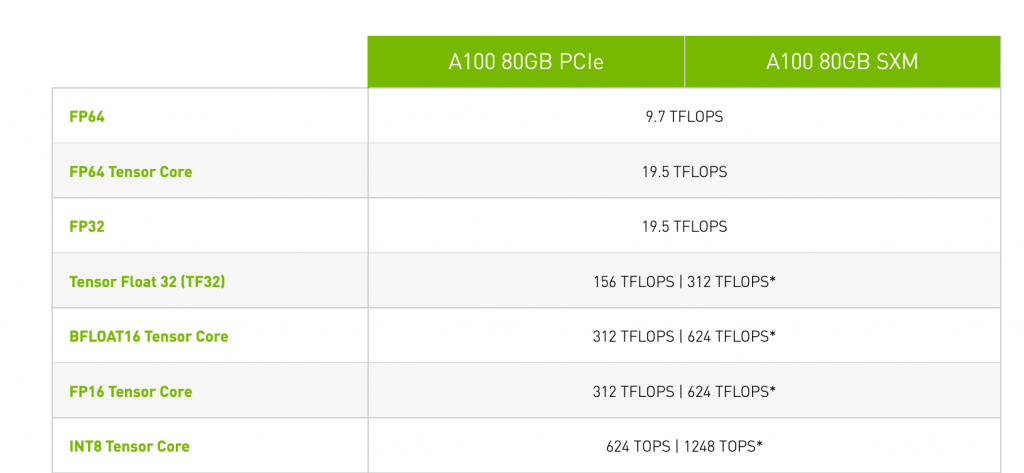
## Do we need this section? Readers may already know this. Let’s think what value we can add here

## What is a layer/op

## What is a graph

# Chapter 5 - Data Types and impact on performance

Let’s start by looking at NVidia A100’s performance spec for different data types



https://frankdenneman.nl/2022/07/26/training-vs-inference-numerical-precision/

Why does throughput vary with data type?

1. As we observed in our Vector and Matrix discussion in chapter 1

* Vector ops/cycle = VLEN\*LMUL/SEW
* Matrix ops/cycle = MLEN\*AMUL/(SEW2)

Matrix ops/cycle is inversely proportional to the element width.

1. Hardware multiplier size is proportional to square of the element width. Therefore, a 16-bit multiplier is 4x the size of an 8-bit multiplier. A 32-bit multiplier is 4x the size of a 16-bit multiplier i.e. 16x the size of an 8-bit multiplier. And a 64-bit multiplier, well you guessed it right, is 64x the size of an 8-bit multiplier. Therefore,
2. Floating point math units are big and power hungry. [IEEE 754 standard](http://www.dsc.ufcg.edu.br/~cnum/modulos/Modulo2/IEEE754_2008.pdf) defines many corner-case scenarios, non-ordinary values, rounding modes and exceptions which the FPU hardware must handle. As per the paper <https://arxiv.org/pdf/2303.17951>, FP8 hardware is 53% higher gate count than INT8.

Both inference and training can benefit from using lower precision formats to gain speed and power advantage, though it may come at the cost of accuracy. We will discuss this in more detail in chapter 8.1.4 on quantization.

The table below lists some of the common formats in AI inferencing. This is by no means a comprehensive list.

## Integer formats

| **Format** | **Description** | **Comments** |
| --- | --- | --- |
| (U)INT32 | (unsigned) 32-bit integer | Integer formats, can also be used to represent factional fixed point numbers by using a scaling factor. |
| (U)INT16 | (unsigned) 16-bit integer |
| (U)INT8 | (unsigned) 8-bit integer |
| (U)INT4 | (unsigned) 4-bit integer | Not a standard format, but gaining popularity. Reduces weight memory when two weights are packed in one byte. |

## Floating point formats

| **Format** | **Description** | **Comments** |
| --- | --- | --- |
| IEEE FP64 | Double Precision FP | Not commonly used in inferencing, or even training. |
| IEEE FP32 | Single Precision FP | Most common format in training, though new formats are gaining traction. |
| IEEE FP16 | Half Precision FP | Faster than FP32, but BF16 is giving it a run for the money |
| Brain Float16 | 1 Sign + 5 Exp + 7 Mantissa bits  Same dynamic range as FP32, but lower precision. Considered a drop-in replacement for FP32. | Not a standard format, but gaining popularity. Half the size of FP32 with zero accuracy loss since AI inference is more sensitive to dynamic range and less to precision. |
| Tensor Float 32 | 1 Sign + 8 Exp + 10 Mantissa bits | 19-bit data within a 32-bit word.  Enables AI training to use tensor cores on NVidia GPUs |
| FP8-E4M3 | 8-bit Floating Point  1 Sign + 4 Exp + 3 Mantissa bits | Balanced range and precision  Not a standard format, but gaining popularity. |
| FP8-E5M2 | 8-bit Floating Point  1 Sign + 5 Exp + 2 Mantissa bits | Wider range, lower precision  Not a standard format, but gaining popularity. |
| Block Float  <https://arxiv.org/pdf/1709.07776> | Hybrid of floating-point and fixed-point arithmetic where a block  of “n” numbers shares an exponent.  FP representation of *1+Lm +Le* bits is compressed to *1 + Lm + Le/n* | All multiply-accumulate operations can be carried out in fixed-point. |
| FP4-E2M1 | 1 Sign + 2 Exp + 1 Mantissa bit | Ultra-low precision. Optimized for weight storage in LLMs and ViTs. |

# Chapter 6 - Tensor Layouts and impact on performance

## NCHW

## NHWC

## Others

Here we should make a recommendation for which layout we think is best for the NPU architecture we are assuming. Rest of the document will be around that layout.

# Chapter 7 - How are NN operations accelerated on NPU

## General matrix multiply (GeMM)

GeMM is the foundation for many operators in an AI graph. Some examples are Fully Connected aka dense layers, Feed Forward Networks and Multi-Layer Perceptron. A GeMM operator has a form

C = A.B

Where, A, B and C are matrices of shape MxK, KxN and MxN respectively.

Let’s see how this operation will be executed on hardware. We will look at three scenarios –

* The first one is a simple scalar CPU with no vector or matrix,
* Second is Scalar + VPU and
* Third will be a full blown NPU with Scalar + Vector + MPU.

In all scenarios, we will assume that A, B, C are large matrices and reside in external memory DDR. Therefore, compute performance will be sensitive to memory b/w and access patterns.

### GeMM on a scalar

Nobody in their right mind would dream of running GeMM on a scalar. However, understanding this piece is important for appreciating the challenges of parallelization on VPU and finally on MPU.

#### Naive method

A GeMM operation of form C = A.B can be implemented as 3-level nested for loop, where output element {i,j} is a dot product of ith row of A and jth row of B.

for m=0:M   
 for n=0:N   
 for k=0:K   
 C[m,n] += A[m,k] \* B[k,n]

Figure 8 : GeMM scalar pseudocode

The inner loop does the following

1. Loads A[m,k] and B[k,j] from memory to CPU register files.
2. Computes the product and accumulates into another register. Most CPUs have FMA (fused-multiply-add) instruction which can do multiply & add in a single step.

After the inner loop is complete, C[m,n] is ready to be written out to memory. CPU may do a reduction or saturation to convert it to a lower data type. The inner loop is repeated for all *MxN* outputs in matrix *C*. Hence, the full loop takes *M\*N\*K* FMAs. This method is known as **Inner Product** method. It generates a single full sum at a time, with no merging of partial sums. This requires a hardware intersection unit to align effectual inputs. This is fairly efficient from computation point of view, ***if*** all load/store and other instructions can be pipelined then the runtime will be dominated by FMA cycles.

But that’s a big **If**. The biggest problem we see here is access pattern of matrix B resulting is an extremely high load time of B[k,n]. To understand this better, let’s note that the change variable in the inner loop is *k*, which means that for successive values of *k****,*** B[k,n] is read from a different row in the matrix. This results in very inefficient memory access for two reasons

1. B[k,n] and B[k+1,n] are not in the same cache line. Hence caching or prefetching results in worse performance than uncached access.
2. The matrix is stored in DDR is row-major layout. B[k,n] and B[k+1,n] reside in different DDR pages. This results in extremely inefficient DDR access and low utilization of DDR bandwidth. For more details on DDR bandwidth utilization, please see Appendix 11.1.

**Is there a better way? There are actually two.**

#### Gustavson’s method

We can move inner loop to middle as shown below

for m=0:M   
 for k=0:K   
 for n=0:N   
 C[m,n] += A[m,k] \* B[k,n]

Figure 9 : GeMM pseudocode for Gustavson’s method

The inner loop generates partial sums of *n* output elements. *n* final outputs will be ready after the end of the middle loop. This **improves spatial locality in input matrix B**, but side-effect is that we need *n* registers for accumulation.

#### Outer Product Method

We can move inner loop outside as shown below

for k=0:K   
 for m=0:m   
 for n=0:N   
 C[m,n] += A[m,k] \* B[k,n]

Figure 10 : GeMM pseudocode for OP method

The inner loop does a partial update of mx*n* output elements. mx*n* final outputs will be ready after the end of the outer loop. This improves spatial locality in matrix B but **also increases data reuse for matrix A**. Note that A[m,k] is independent of the inner loop change variable *n*. Hence, a row of A needs to be loaded only once in the middle loop and gets reused for all iterations of the inner loop. This reduces number of A loads by a factor of k.

But nothing comes free. The cost of doing this reordering is that we need mx*n* registers for accumulation. This is not economical on CPU scalar, but we will revisit this later when we discuss matmul on MPUs.

### GeMM on a VPU

We take loop Gustavson’s method and modify it for parallelization along the row axis.

for m=0:M   
 for k=0:K   
 for n=0:N/VL   
 C[m,n:**n+VL**] += A[m,k] \* B[k,n:n+VL]

Figure 11 : GeMM pseudocode for Gustavson’s method on VPU

The inner loop does *VL* FMAs in parallel. Remember from chapter 1.3,

VL <= VLMAX = (VLEN\*LMUL)/SEW

For VLEN=512, SEW=8, VPU can process 64 FMAs (=128 ops) per cycle. However, achieving this throughput requires load/store to run at least as fast as compute. This is not possible if A, B and C matrices reside in DDR. The SoC may have an internal SRAM (or TCM) but if the matrices are too big, SRAM may not be enough.

**Solution** – Tiling comes to our rescue. We divide matrices into smaller submatrices. In general, GEMM of form **C = A.B** can be broken down into submatrices

* A11….. AMK, Each submatrix is of size mtile\_m x mtile\_k
* B11….. BKN,  Each submatrix is of size mtile\_k x mtile\_n
* C11….. CMN, Each submatrix is of size mtile\_m x mtile\_n

Total number of submatrices = (M/mtile\_m)\*(N/mtile\_n)\*(K/mtile\_k)

C can be computed tile-wise as Cmn = Σdotproduct(Amk , Bkn )

Submatrix dimensions (m,n,k) are chosen such that the fit in internal SRAM and Σ is done by accumulating through all the tiles.

Diagram : TBD

The for loop now can be written as

for m=0:M:mtile\_m   
 for k=0:K:mtile\_k   
 for n=0:N:mtile\_n   
 m\_ul = m + mtile\_m   
 n\_ul = n + mtile\_n   
 k\_ul = k + mtile\_k   
 for x=m:m\_ul  
 for z=k:k\_ul  
 for y=n:n\_ul:VL  
 C[x,y:y+VL] += A[x,z] \* B[z,y:y+VL]

Figure 12 : GeMM pseudocode for Gustavson’s method with tiling

### GeMM on an MPU

**Option 1: Inner Product**: Use Inner Product method and extend the parallelization to two dimensions. We also set tile size to match with the matrix panel size on MPU

1. mtile\_m = TMMAX = MLEN/RLEN
2. mtile\_n = TNMAX = RLEN/SEW
3. mtile\_k = TKMAX = min(TMMAX, TNMAX)

Let’s assume square tiles of size TxT

for m=0:M:T   
 for n=0:N:T   
 for k=0:K:T

tile\_A = A[m:m+T,k:k+T]

tile\_B = B[k:k+T, n:n+T]

C[m:m+T,n:n+T] = tiled\_inner\_product(tile\_A, tile\_B)

tiled\_inner\_product is computed row by row, taking the inner product of a row of A, with multiple columns of B.

for x=0:T   
 C\_tile[x] += inner(tile\_A[x], tile\_B)

Figure 13 : Tiled GeMM pseudocode on MPU using IP method

*A[x]* is the xth row of tile A. “*inner*” is a vector-matrix dot product function which is done on MPU hardware in a single cycle. The output is generated row by row. The entire tile is computed in T cycles. Thus 2T3 arithmetic operations are done in T cycles, resulting in a speedup of 2T2.

Need to double check : In this method, to compute T outputs, MPU needs to load 1 row of A and T columns of B, each of size T elements. This is repeated T times to compute the tile. Thus, total number of loads per tile = T\*(T+T2) = T2 + T3

Not bad, but can we do better?

**Option 2: Outer Product**: Now let’s take the outer product method and extend the parallelization to two dimensions, as follows

for m=0:M:T   
 for n=0:N:T   
 for k=0:K:T

tile\_A = A[m:m+T,k:k+T]

tile\_B = B[k:k+T, n:n+T]

C[m:m+T,n:n+T] = tiled\_outer\_product(tile\_A, tile\_B)

tiled\_outer\_product is computed step by step for all the elements in the tile, taking the outer product of a column of A, with one row of B.

for x=0:T   
 C\_tile += outer(tile\_A[:,x], tile\_B[x,:])

Figure 14 : Tiled GeMM pseudocode on MPU using OP method

*A[:,x]* is the xth column of tile A. *B[x,:]* is the xth row of tile B. “*outer*” is a vector-vector outer product function which is done on MPU hardware in a single cycle. This generates TxT partial outputs which are accumulated for all values of x. The entire tile is computed in T cycles. Thus 2T3 arithmetic operations are done in T cycles, resulting in a speedup of 2T2. Same as inner product method.

Then what’s the difference?

Need to double check : The difference lies in the number of loads. Every iteration of the loop requires 2T loads. This is repeated T times for a total of T2 loads. Significant improvement from T2 + T3 in case of IME.

However, nothing comes for free. Outer product requires TxT accumulator array, which can be quite costly for large T values. This method has a higher overhead, but it gives system designers to create a high performance NPU with balanced load/store with compute. Therefore it is a good choice for high performance AI accelerators.

For rest of this book we will assume outer product based MPU.

TBD : Balanced load/store with compute analysis

## 2D Convolutions

2D convolution is the backbone of a CNN (Convolutional Neural Network). Convolutions have been used in image processing for a long time. Some examples are edge detection, noise filtering, blurring, sharpening etc. Mathematically speaking, a convolution is an integral that expresses the amount of overlap of one function f as it is shifted over another function g. In image processing and computer vision, convolutions have been used for various purposes like edge detection (Sobel filter), sharpen/blur images etc. In Machine Learning, convolutions are used to extract low level features from an input signal. This is done by sliding a set of filter kernels on the input tensor. The figure below is an example of a 3x3 kernel applied on a 3-channel RGB image.

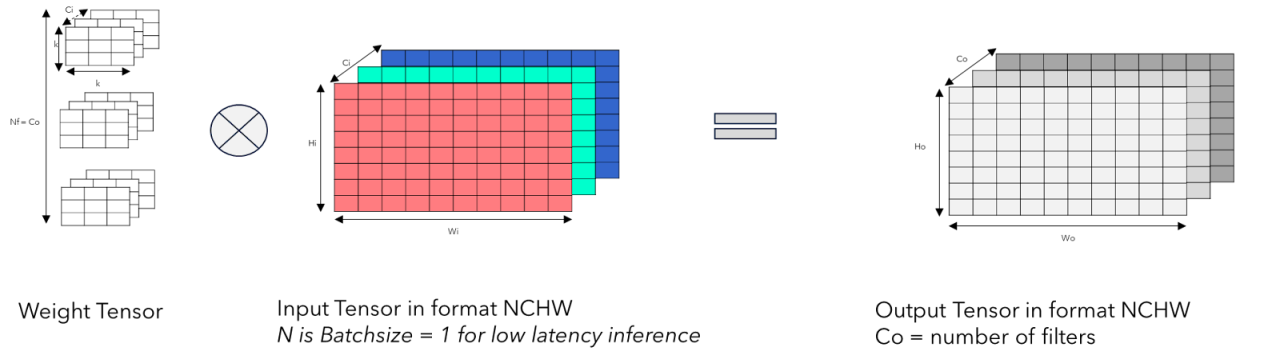


Figure 15 : 2D Convolution on an RGB Image

A CNN graph has several convolution layers, with each layer having several learnable kernels. After training, each kernel learns to detect some unique features in the image which can later be used for feature patching, pattern recognition and even content generation.

Mathematical formula for conv2d is

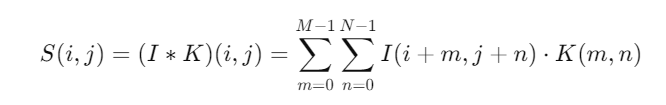


Figure 16 : 2D Convolution formula

*Source :* [*https://medium.com/@shubhamzope\_dev/understanding-how-conv2d-layers-work-in-ai-advanced-concepts-and-formulas-ea208b5c8ad7*](https://medium.com/@shubhamzope_dev/understanding-how-conv2d-layers-work-in-ai-advanced-concepts-and-formulas-ea208b5c8ad7)

Input ***I*** is a 3D tensor of size Wi x Hi x Ci

Kernel ***K*** is a 4D tensor of size kx x ky x Ci x Co

Output ***S*** is a 3D tensor of size Wo x Ho x Co

With unity stride and no dilation, the following relationship applies

* Wo = Wi - kx + padx - 1
* Ho = Hi – ky + pady - 1

Convolution is the most computationally expensive operation in a neural net.

* One convolution takes *kx\*ky\*Ci* multiplications and additions. Total *2\* kx\*ky \*Ci* operations.
* There are *Co* filters in a layer.
* *Wo\*Ho\*Co* outputs need to be generated.
* *2\* kx\*ky \*Ci \* Co \* Wo\*Ho* operations for 1 convolution layer.

2D Convolution can be viewed as a level nested for loop

for n=0:batchsize   
 for cout=0:Co   
 for cin=0:Ci   
 for h=0:Ho

for w=0:Wo

for k\_v = 0:ky

for k\_h = 0:kx

*O[n][co][h][w] += kernel[co][ci][k\_h][ k\_w] \* input[n][ci][ h + k\_h][ w + k\_w]*]

This needs to be repeated for all the convolution layers in a network.

All of this needs to be repeated ips times in a second, where ips is the inference per second requirement.

**Example** – Darknet-53 (640x640) requires 155 Billion operations for 1 inference.

@30 ips, the number of computations needed is 4.65 Trillion operations per second.

Therefore, an efficient implementation of convolution on the hardware is critical to achieving high performance at low system cost and power.

### Accelerating Conv2D on MPU

#### Im2col method

My least favorite method. Requires the input image (HxWxC) to be expanded by a factor of k2 and reshaped to HW x k2Ci. Kernel tensor is in shape k2Ci x Co. This results in a GeMM of form

C=AMxK . BKxN

where

* M = H\*W
* K = k2\*Ci
* N = Co

Output is a matrix of shape H\*W x Co. On a VPU, parallelization can be done along the M dimension or N dimension. In either case, this achieves 100% utilization when m (or N) is a multiple of VLMAX, (and of-course memory is not the bottleneck).

On an MPU, the two dimensions for parallelization chosen are M and N. This achieves 100% MAC array utilization when M is a multiple of MAC array width (TMMAX) and N is a multiple of MAC array height (TNMAX). For large tensors this is not difficult to achieve, though there are exceptions.

If 100% PE utilization is achievable, where is the problem? The expansion step is where the problem lies. This has two major disadvantages

1. Memory footprint increases by a factor of **k2**.
2. VPU and MPU are not designed for this operation. Hence, it falls back to scalar and becomes a performance bottleneck.

## There is a proposed [implicit im2col](https://arxiv.org/abs/2110.03901) method, but we leave to the reader to explore it.

#### Conv2D using Outer Product

We are all engineers. Why not jump right into the pseudocode.

for i=0:Ho   
 for j=0:Wo:TNMAX   
 for cout=0:Co:TMMAX

for k\_v = 0:ky

for k\_h = 0:kx

input\_slice = input[i+y, j\*TNMAX+x:(j+1)\*TNMAX+x, ch] #input vector of length T elements

kernel\_slice = kernel[y, x, cout\*TMMAX:(cout+1)TMMAX] #weight vector of length T elements

ACC += outer(input\_slice, kernel\_slice) #Vector - Vector Outer Product and accumulation

out\_tileij = ACC #Write TxT convolution result to output tensor

Important thing to note is that we have chosen the dimensions Wo and Co for parallelization, as highlighted in the code above. This allows us to achieve 100% MAC array utilization without any im2col like transformation, as long as

* Wo is a multiple of TNMAX
* Co is a multiple of TMMAX

### Other types of 2D convolutions

So far we have considered the simplest form of Conv2d. There are some variations which can potentially have a performance impact. Let’s look at some of them

#### Pointwise Convolutions

The only difference is that the kernel is of size 1x1xCi. Has no impact on performance.

#### Strided Convolutions

Default is unit stride where every element in the input image is used for convolution. With stride=s, every sth element is used i.e. s-1 elements are skipped over. Do note that stride applies only to W and H dimension, not to the C dimension.

This is a load problem more than a compute problem. RISC-V AME supports instruction *mlsae* for strided load of A matrix. This instruction selectively pick the required operands from memory to registers. Once the operands are in memory, convolution operation will be done as normal. The performance impact comes from load operation. Strided loads are slower than unit-stride loads by a factor of s. TO BE CONFIRMED. If load bandwidth is on the critical path, it may stall the MPU and cause performance degradation.

#### Dilated Convolutions

Similar to strided convolutions, but the skipping is done the filter kernels. Strided load instructions RISC-V Matrix help here as well, with the same performance impact as before. RISC-V AME defines *mlsbe* instruction for strided load of B matrix.

#### Grouped Convolutions

Regular convolution convolves each of Co filters across *all the input channels* and concatenates the results to generate Co output channels. In a grouped convolution, input channels and filters are divided into *g* groups. Each group of input channels (of size Ci/g) is convolved with Co/g filters. This results in reduced number of parameters and MAC operations.

|  | **Standard Convolution** | **Grouped Convolution** |
| --- | --- | --- |
| # Parameters | Co\*Ci\*kx\*ky | (Co/g)\*Ci\*kx\*ky |
| #MACs | Co\*Ci\*kx\*ky\*Wo\*Ho | (Co/g)\*(Ci/g)\*kx\*ky)\*g |

Each group has *g*2 fewer MACs than the original convolution, and there are *g* such groups. Hence, the overall MAC count goes down by a factor of *g*.

Does this mean inference time of a grouped convolution should be *g* times lower? Not necessarily. Recall that while accelerating conv2d on MPU, we used Co as one of the parallelization dimensions. This works well if Co is a multiple of matrix MAC array X size. In grouped convolution, MAC array utilization will be low if (Co/g) is not a multiple of matrix MAC array X size. Consider the following scenario:

* Co = 128
* Matrix MAC Array = 64x64
* g = 4

In regular convolution, weight tensor gets divided into two tiles of 64 channels each which fully utilizes the X dimension of MAC array.

In grouped convolution, each group has 128/4 = 32 channels. MAC array X dimension will be only 50% utilized. Therefore, total number of MACs reduces by a factor of 4, but because of 50% efficiency drop, we expect only a 2x increase in inference speed.

For larger MAC arrays the utilization will be even lower. For e.g. 128x128 MAC will achieve 100% utilization on regular conv2d but only 25% on grouped convolution in the scenario above. **This is one of the examples where high “TOPS” spec does not convert to high throughput in real world usecases.**

#### Depthwise Convolutions

Depthwise convolution treats each input channel separately. It applies a different filter to each channel. It can be seen a special case of grouped convolution where *g* = 1.

Does https://arxiv.org/pdf/2206.12124 help ? Unlikely, but read again

Same for <https://ojs.aaai.org/index.php/AAAI/article/view/6159/6015>

#### Depthwise Separable Convolutions

Logically a two-step process – depthwise convolution, followed by a pointwise convolution.

#### Point-Wise Convolution with Residual input layer

TBD. < https://software-dl.ti.com/mctools/nnc/mcu/users\_guide/layer\_configs.html >

#### Transposed Convolutions

TBD.

## Activations

Take a few popular ones (ReLu, Gelu, Tanh, Sigmoid…)

Do HW mapping of each one and mention how to get best performance.

## Attention

### Multi Head Attention

### Group Query Attention

### Deformable Attention

## MLP

## Layernorm

## Softmax

## Transpose

## Others

While we cannot cover each and every op, it will help to have a note explaining how reshape, concat etc. shall be done.

# Chapter 8 – Optimization Techniques

## Graph modifications for performance

### Layer Fusion

### Constant Folding

### Pruning

### Quantization

TBD : To rewrite

Quantization is a powerful trick to reduce power and boost throughput, without a huge increase in system cost. Smaller data width also helps in reducing memory traffic and storage, which helps in cases where performance bottleneck is on memory bandwidth. For e.g. w8a16 (8-bit weights, 16-bit activations) will take same number of computation time as w16a16, but w8a16 reduces weights memory by half and indirectly improves performance.

Energy consumed by data movement dominates the total energy in AI inferencing. Moving fewer bytes directly helps reduce the power.

Floating point is good when you have a problem where you don't know the range of the input data beforehand. In most AI (and DSP problems) you know the range of the input data beforehand The good news is that fractional numbers can be represented using a scaling factor and integer ALUs can be used efficiently.

Therefore, to save area and power, it is a common practice to design NPUs with fewer floating point execution units than integer executions. There are NPUs which support only integer math in vector and matrix. All the floating point computations are offloaded to the scalar unit which makes it very slow. For such NPUs, neural network must be quantized to use integer formats for all weights and activations. Therefore, when it comes to inference, floating point data must be used very carefully. For training, you must use floating point formats, but you gain throughput by choosing low precision FP formats like TF32, BF16, FP8 or FP4.

### What else??

## Compression

### Weight Compression

### Activation Compression

## Using Sparsity

### What is sparsity and where does it come from

### Structured vs unstructured

### Ways to leverage sparsity for acceleration

## Graph Sequencing Techniques

### Layerwise execution

### Depthwise execution

### Hybrid (assuming our disclosure if filed by the time this is published)

## Scalability

### Load distribution across multiple NPUs

#### How does a GPU do it

#### Why this is not right for NPU

#### Then what is right for NPU

### Other scalability challenges

# Chapter 9 - Software

## The role of software

## Popular AI software frameworks

## Compiler based

#### LLVM

#### MLIR

#### TVM

## How does software get best hardware entitlement

# Chapter 10 - Case Studies

Here we should show how the models will be executed on our assumed NPU and what performance can be expected. Having the roofline tool would have helped here.

## A simple 2-layer NN

## CNN – Alexnet/Resnet50/….

## ViT

## LLMs

### Whisper

### Llama or DeepSeek or similar. Pick something suitable for Edge.

### DETR

## Some ADAS/Robotics networks

### BEVFormer

### BEVFusion

# Appendix A

## DDR bandwidth utilization

TBD

## Comparison of how a GeMM operation executes on scalar, vector and matrix engine.

Assume input and output matrices are 8x8. Vector has 8 lanes and Matrix has an 8x8 MAC array.

**Scalar Pseudocode**

for i in range 0 ->7

for j in range 0 ->7

for k in range 0 ->7

C[i,j] += A[i,k] \* B[k,j] #Multiply Accumulate

Most CPUs have a Fused Multiply Add (FMA) instruction. The loop above is expected to run in 512 cycles, assuming no loop overheads and no memory bottlenecks.

**VPU Pseudocode**

for i in range 0 -> 7

for j in range 0 -> 1

for k in range 0 -> 7

C[i,j:j+8] += A[i,k] \* B[k,j:j+8] #Multiply Accumulate

Major differences from scalar are:

1. The middle loop runs only once. This is because vector engine can process 8 elements at a time.
2. In the inner loop
   * 8 elements are read from jth row of matrix B.
   * 1 element is read from kth row and ith column in matrix A.
   * 8 partial outputs of C are computed.
   * The partial outputs are accumulated into vector registers. This is possible because every VPU lane has registers allocated to it.
3. Once the inner loop is completed, we have 8 elements of matrix C which can be committed to memory.

The inner loop computes 8 elements in 8 cycles. Matrix Multiplication finishes in 64 cycles, which is an 8x speedup from scalar.

**MPU Pseudocode**

for i in range 0 -> 7

for j in range 0 -> 1

for k in range 0 -> 1

C[i:i+8,j:j+8] += A[i:i+8,k] \* B[k,j:j+8] #Multiply Accumulate

Major differences from VPU are:

1. Middle loop and inner loop run only once. This is because MPU has an 8x8 array of MAC units, all working in parallel.
2. In the inner loop,
   * 8 elements are read from jth row of matrix B.
   * 8 elements are read from ith column in matrix A.
   * 8x8 partial outputs of C are computed using 8x8 MAC array.
   * The partial outputs are accumulated into matrix registers, or internal accumulators.
3. Once the inner loop is completed, we have 64 elements of matrix C which can be committed to memory.

The inner loop computes 64 elements in 8 cycles. Matrix Multiplication finishes in 8 cycles, which is an 8x speedup from VPU and 64x speedup from scalar.

Point to Note: This method of doing matmul is called **inner-product method**, mainly because the inner loop is essentially a dot product (aka inner product) of ith column in matrix A with jth row of matrix B. Next we briefly look at another method which is more hardware friendly. This method is called **outer-product method**.

Here we simply re-order the loops, as shown below

for k in range 0 -> 1

for i in range 0 -> 7

for j in range 0 -> 1

C[i:i+8,j:j+8] += A[i:i+8,k] \* B[k,j:j+8] #Multiply Accumulate

The numerical output remains the same, but a few things happen as a result of this re-ordering

1. In the inner loop, elements of matrix A do not change, since the code loops on the variable j and A column is a function of i and k. This means that in the inner loop we do not need to reload A column every time.
2. A column can be loaded to registers in the middle loop and it is reused for all the iterations in the inner loop. This brings down the number of loads by a factor of (almost) 8. Data movement is the most energy intensive step in AI inference and this optimization goes a long way in reducing it.
3. It must also be noted that the elements of matrix C are not ready to be committed to memory after the inner loop ends, as was the case in inner-product method. Instead, all the elements of C get a partial update every iteration and are ready to written to memory after all the loops complete. This requires more internal memory closer to MAC arrays.

Thus, outer product method reduces power at the cost of additional hardware. For more details, please refer to RISC-V IME and AME resources linked in the references section.

## Arithmetic Intensity.

TBD.

# References

1. [Texas Instruments C7 MMA](https://forum.digikey.com/t/ti-edge-ai-am6xa-processors-with-deep-learning-accelerators-and-its-efficiency/39813)
2. [Qualcomm's Hexagon DSP And NPU](Qualcomm's%20Hexagon%20DSP%20And%20NPU)
3. VLIW vs SMT : <https://www.cs.umd.edu/~meesh/411/CA-online/chapter/multiple-issue-processors-ii/index.html>
4. RISC-V SMT: <https://www.youtube.com/watch?v=QR4niawYSVI>
5. Vector vs SIMD: <https://medium.com/swlh/risc-v-vector-instructions-vs-arm-and-x86-simd-8c9b17963a31>
6. <https://people.eecs.berkeley.edu/~pattrsn/252S98/Lec06-vector.pdf>
7. <https://www.cs.sfu.ca/~ashriram/Courses/CS7ARCH/hw/hw4.html#dense-matrix-dense-matrix-tiled-multiplication-gemm>
8. <https://arxiv.org/pdf/2301.10852>
9. implicit im2col - <https://arxiv.org/abs/2110.03901>
10. RISC-V AME Matrix Proposal : <https://lists.riscv.org/g/tech-attached-matrix-extension/attachment/300/0/AME_proposal_from_Alibaba.pdf>
11. RISC-V AME Matrix Proposal : <https://github.com/riscv-stc/riscv-matrix-spec/blob/matrix64/pdfs/riscv-matrix-spec-v0.5b-64bit-encoding.pdf>
12. [RISC-V AME spec 0.5b](https://lists.riscv.org/g/tech-attached-matrix-extension/attachment/210/1)
13. RISC-V AME Matrix ZVMA Proposal:<https://lists.riscv.org/g/tech-attached-matrix-extension/attachment/257/0/zvma-20241218.pdf>
14. <https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/speedups.and.amdahls.law.1pp.pdf>
15. NVidia H100 Architecture: <https://lists.riscv.org/g/tech-attached-matrix-extension/attachment/25/0/NVIDIA%20Hopper%20Architecture%20In-Depth%20_%20NVIDIA%20Technical%20Blog.pdf>
16. NVidia A100 Architecture: <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>
17. <https://medium.com/@zhaodongyu/optimize-sgemm-on-risc-v-platform-b0098630b444>