AI Domain Specific Processor Design

北京希姆计算科技有限公司
Stream Computing Inc.

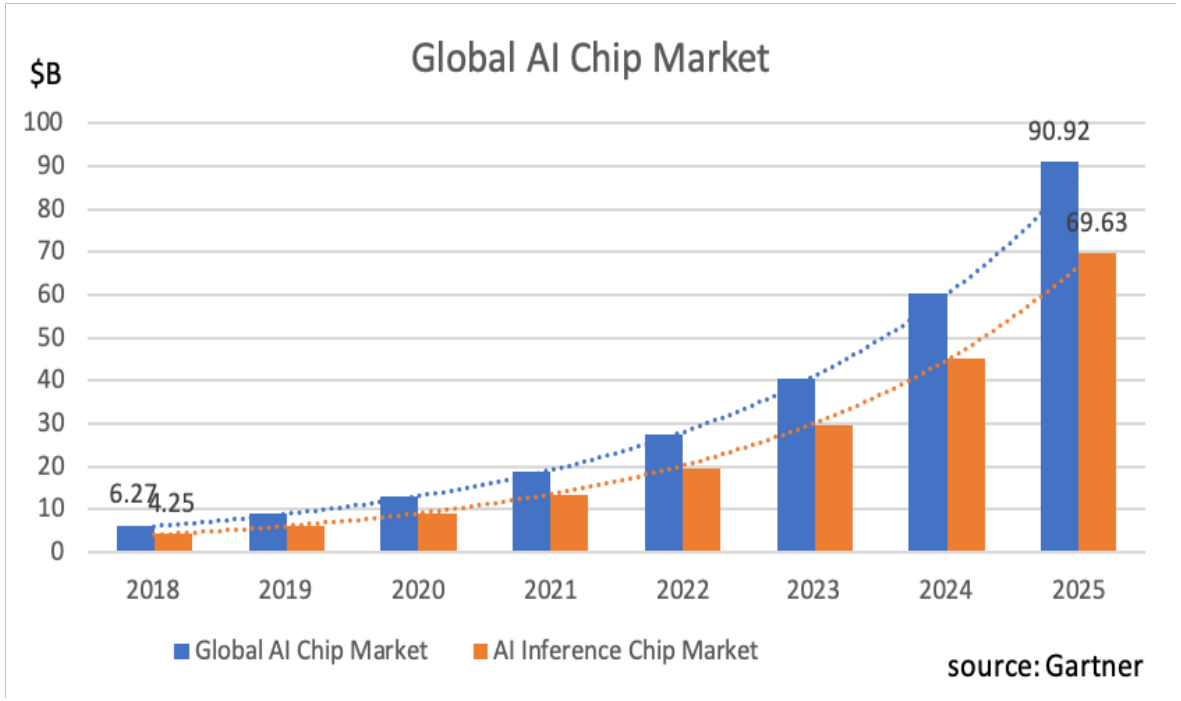
Mark Zhan
OS2ATC 2019@ShenZhen

Agenda

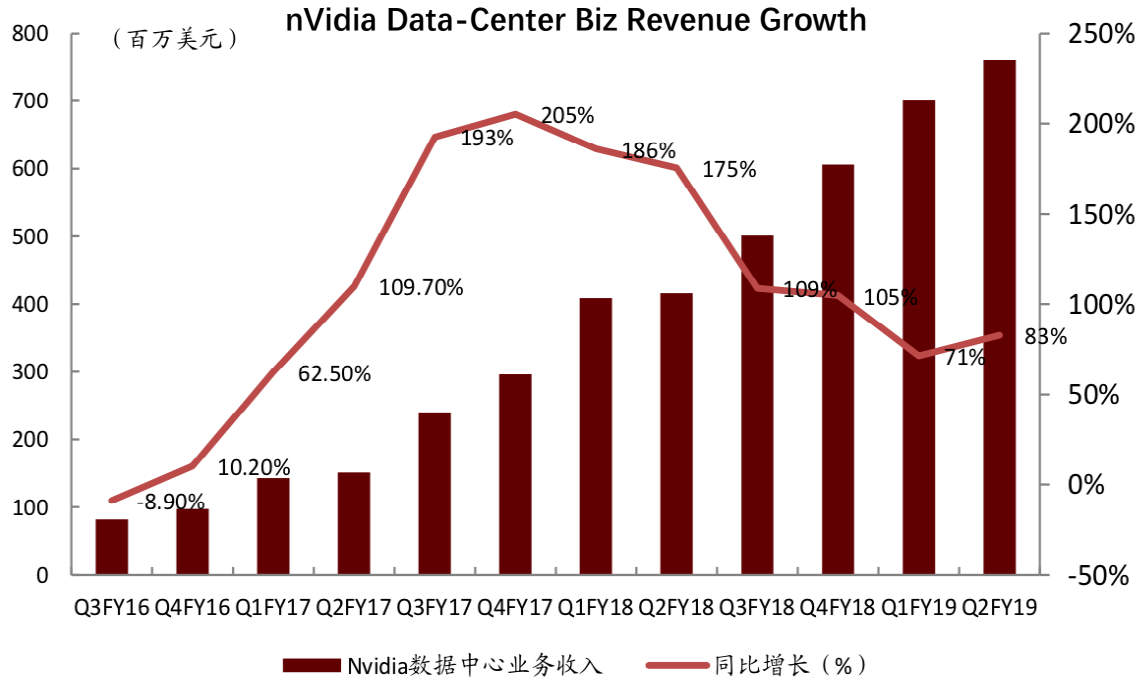
- Market Analysis of AI Processor
- Architecture of AI Processor
  - Evolution of AI Processor Architecture
  - Key Metrics of AI Processor Architecture
- Software of AI Processor
  - Program Parallelism Strategy
  - Data Sharding of Matrix Multiplication
  - Graph Compiler
- Summary

Global AI Chip Market Trend

- AI Chip Market will grow > 10 times in next 5 years
  - The total AI chip market is projected to attain $90.35 billion by 2025 at 45.2% CAGR
  - AI Inference Chip (server-side & edge-computing) will have bigger market share than AI training chip

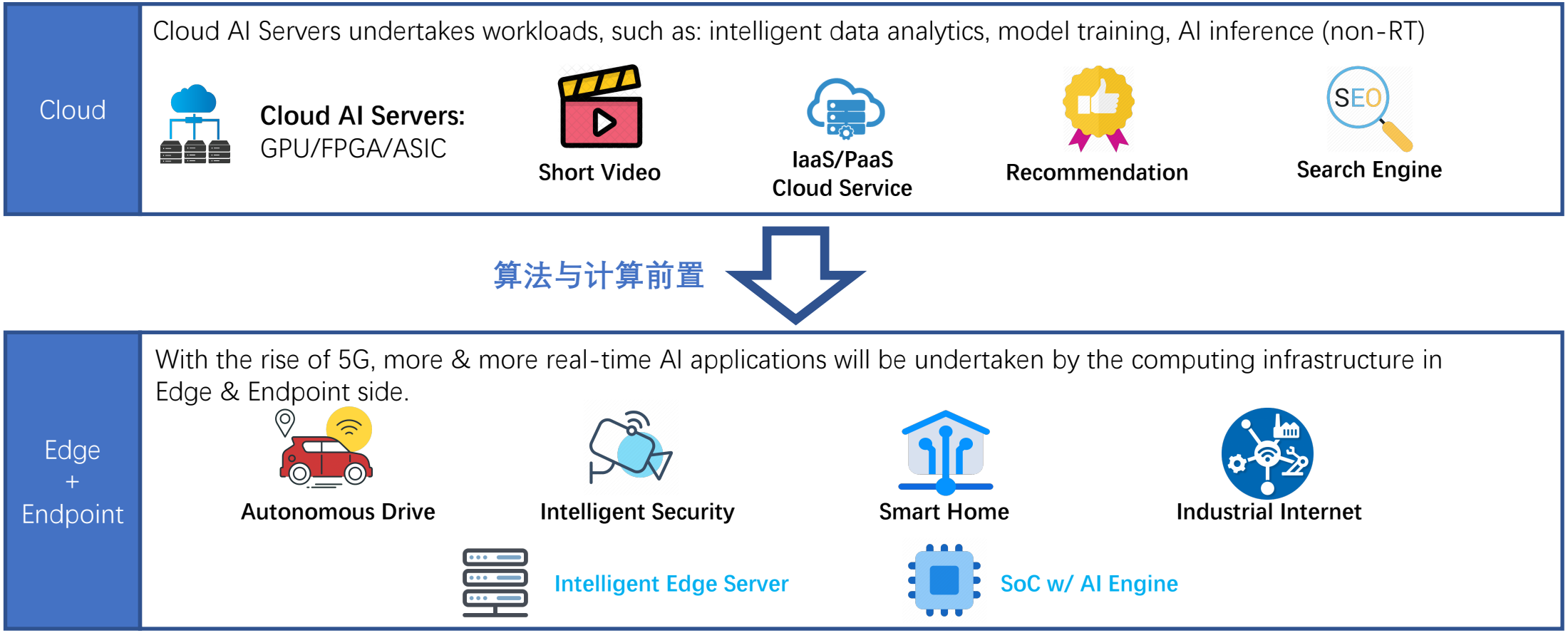


- NVIDIA FY20 Q3 Financial Report shows Data-Center Biz Revenue continues to grow rapidly, driven by the rise of:
  - Inference: The revenue of GPU T4 exceeds GPU V100
  - Conversational AI



What is driving the endless need to AI Computing Power?

- For internet companies, various value-added services have been powered by AI technology in cloud-side
- The rise of 5G and AIoT is pushing more computing to edge-side, w/ better real-time performance and better data security



What is blocking broader adoption of AI Computing Power?

- Total Cost of Ownership (TCO) of AI Computing Infrastructure includes:
  - Server Hardware (NRE H/W Cost)
  - Software: Licenses and Services
  - Operation expenses: Building Infrastructure、Electricity (Cooling、backup power)、etc.

- TCO of AI Server Deployment is still very high today!

	Performance (fps)	CPU (Intel Xeon 6140 x2)	GPU (x4)	Memory (64GB x2)	SSD (960G)	DISK (10T)	Server	Total	H/W TCO (¥/f)
nVidia T4 (ResNet-50)	20000							¥200,000	¥10
nVidia V100 (BERT)	12000							¥450,000	¥37.5

Agenda

- Market Analysis of AI Processor
- Architecture of AI Processor
  - Evolution of AI Processor Architecture
  - Key Metrics of AI Processor Architecture
- Software of AI Processor
  - Program Parallelism Strategy
  - Data Sharding of Matrix Multiplication
  - Graph Compiler
- Summary

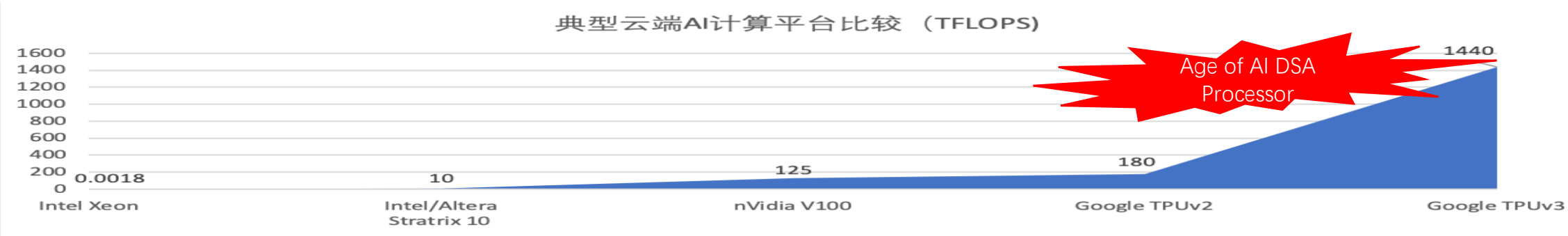
Evolution of AI Computing Engine

- Before 2006, CPU has been the major computing platform for logic-based applications.
- In General CPU, the majority of transistors are used for control logic and memory unit, only very small portion of transistors are used for ALUs.
- Being closer to the limitation of silicon process, Moore Law becomes invalid — The performance improvement of processor becomes slow!

- In 2006, NVIDIA released its epochal GPU product “G80” and ”Fermi” architecture.
- GPU is designed for computing: the majority of transistors are used for ALUs.
- Powerful Massive Parallel computing ability opens the door of AI age.

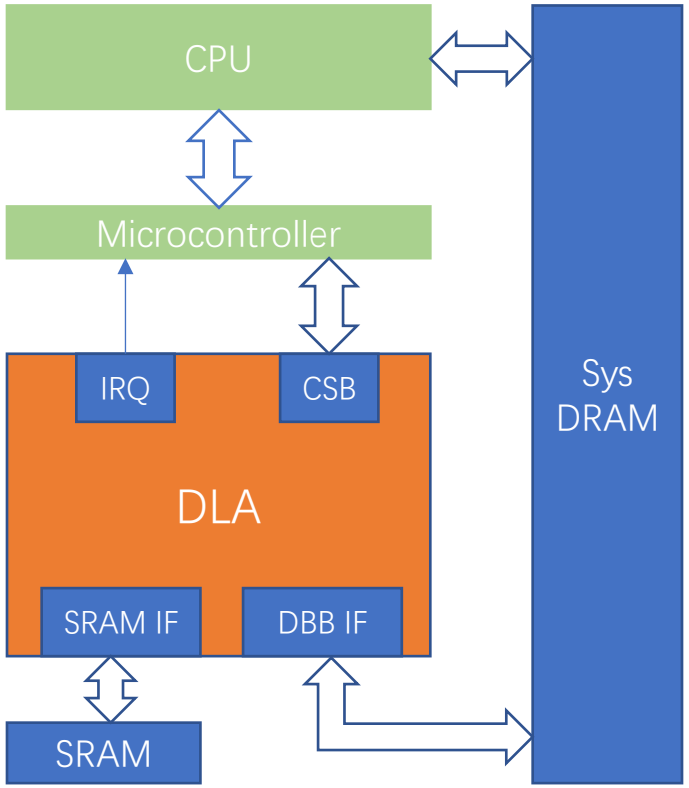
Evolution of AI Computing Engine

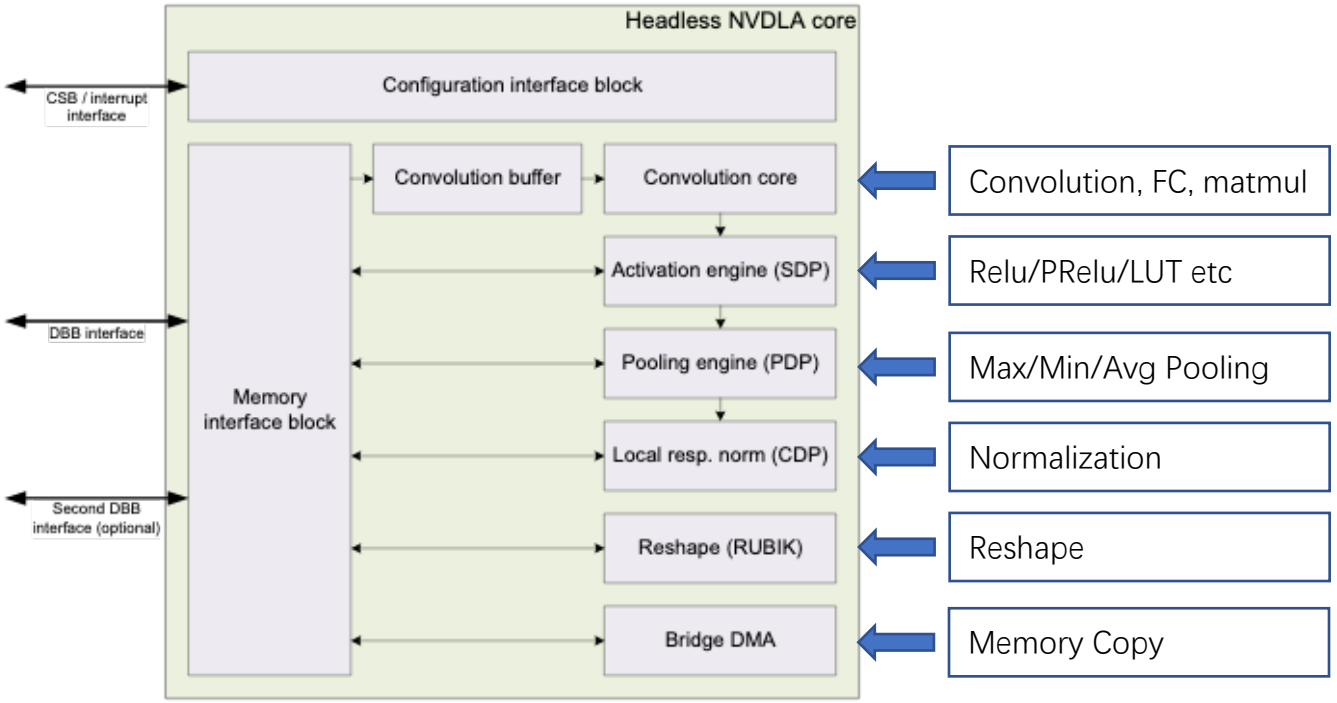
- GPU has been the defacto computing platform for AI. But:
  - GPU is very expensive — Very high TCO for customers
  - GPU is not designed for AI computing, e.g: redundant transistors for computer graphics, double-precision etc.
  - Not sufficient memory design: Too small on-chip SRAM causes big latency.
  - Poor performance for small batch size



	CPU	FPGA	GPU	DSA TPU/NPU
pros	• Good Programmability • General Purpose: Almost can do everything	• Very flexible programmability. • Suitable for quick algorithm iteration	• Compared with CPU, GPU has massive parallel computing ability • Compared with FPGA, good universality	• AI-Oriented Domain Specific Architecture Design and Optimization • Best Trade-off for chip PPA • Low cost for silicon large volume shipment
cons	• High Difficulty and Cost of Silicon Development • Low Computing Power. Not suitable for computing intensive workload	• Limited Computing Power. • Speed and Power are not competitive with ASIC • High cost for large volume shipment.	• Expensive —— High TCO • High power consumption • Redundant design for AI Inference Computing (e.g: useless FP64)	• High design and development difficulty • DSA design cause limited universality

NVIDIA Open Source NVDLA

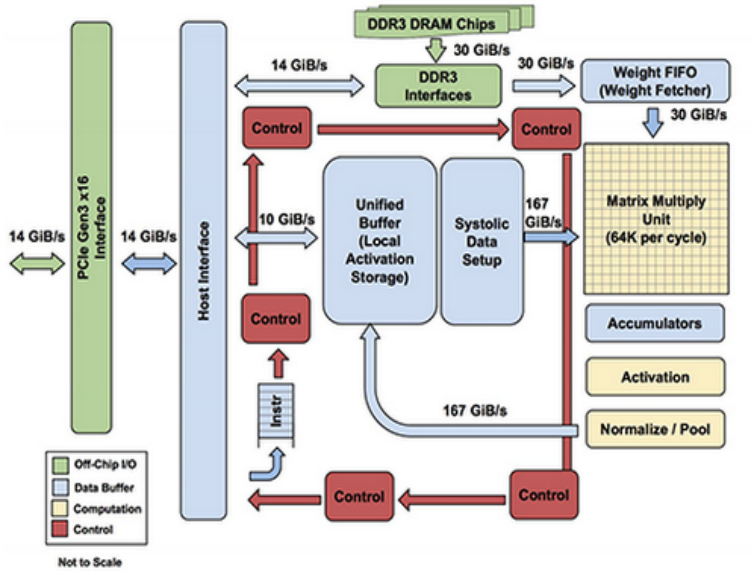


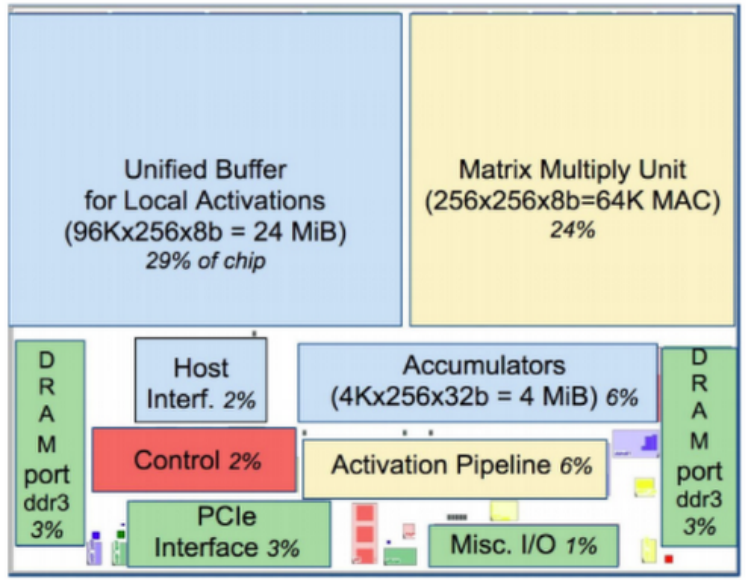


- A complete design, both s/w and h/w are open source
- Good for beginners

Google TPU v1

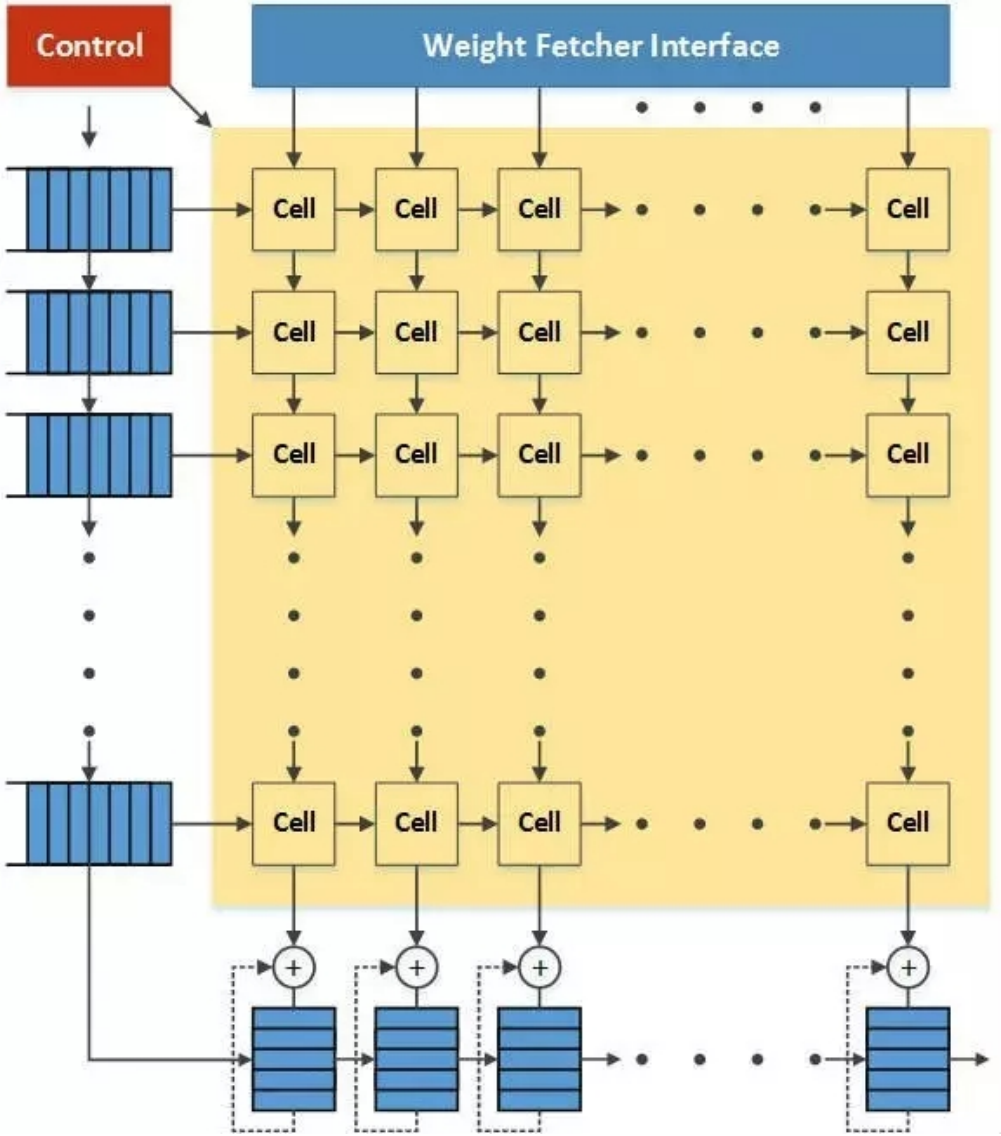
- The Matrix Unit (~1/4 chip area):
  - 256x256 **Systolic Array** of 8-bit MAC
  - 700MHz, Peak Performance 90TOPS
- Big size of on-chip SRAM (>1/4 chip area): 24MiB activation memory
- Host controls TPUv1 through 5 CMD Instructions:
  - Read_Host_Memory, Write_Host_Memory, Read_Weights, MatrixMultiply/Conv, Activate(ReLU,Sigmoid,Maxpool,LRN,...)
  - 4-stage overlapped execution, 1 instruction type / stage
- Software Complexity:
  - no branches, in-order issue, s/w controlled buffers, SW controlled pipeline synchronization.
- Overall Design:
  - As a slave computation accelerator
  - Only Meet the current need (at that time) of DNN models
  - Limited Flexibility to meet future need of future DNN models





Google TPU v1 — Systolic Array

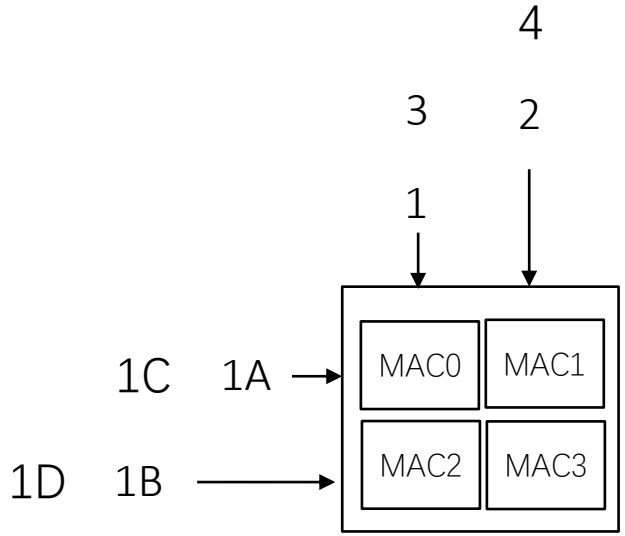
- Problem: energy/time for repeated SRAM accesses of matrix multiply
- Solution: “Systolic Execution” to compute data on the fly in buffers by pipelining control and data
  - Control and Data are pipelined
  - Relies on data from different directions arriving at cells in an array at regular intervals and being combined
- Major benefit of systolic arrays:
  - all operand data and partial results are stored within (passing through) the cell array
  - There is no need to access external main memory or internal caches during each operation as is the case with Von Neumann or Harvard sequential machines

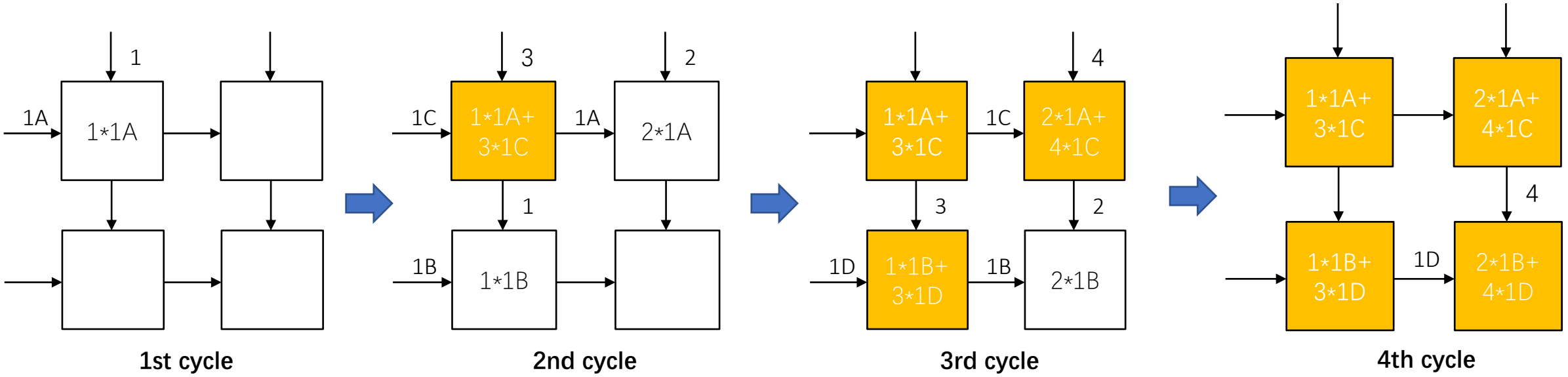


Google TPU v1 — Systolic Array Inside

- Use a generic 2*2 matrix multiply to show how systolic array works:

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} * \begin{bmatrix} 1A & 1B \\ 1C & 1D \end{bmatrix} = \begin{bmatrix} 1*1A+3*1C & 1*1B+3*1D \\ 2*1A+4*1C & 2*1B+4*1D \end{bmatrix}$$

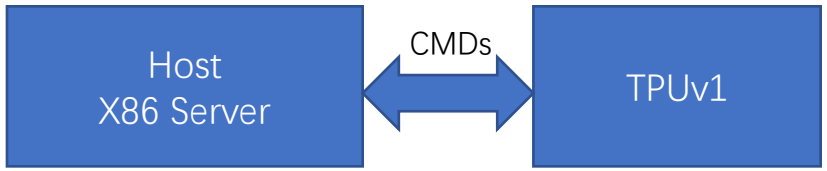




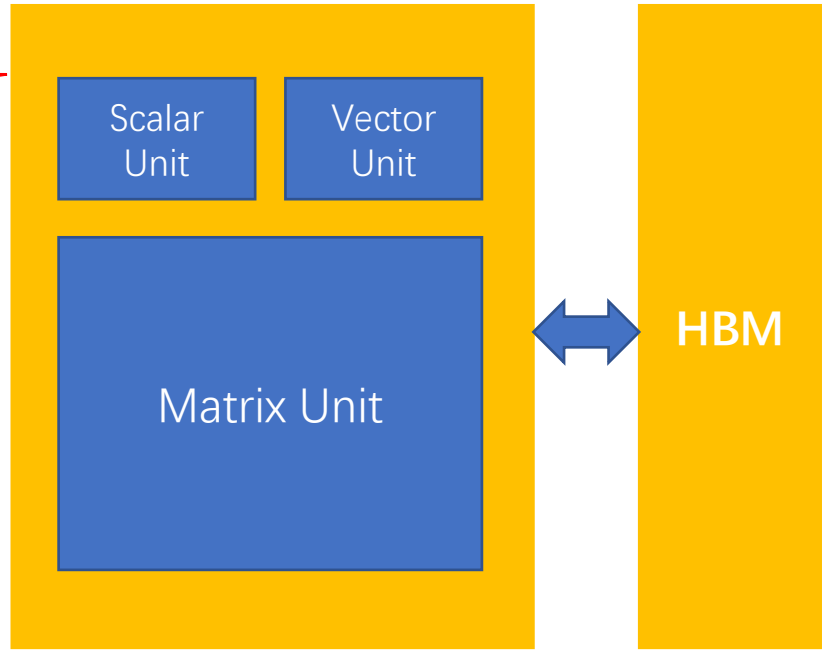
Google TPU v2/v3

- The change of design objectives impacts H/W & S/W Codesign:
  - Inference (TPUv1) => Training (TPUv2/v3)
  - meet our current and future needs:
    - Enough flexibility & Programmability to handle future models
    - Choosing the right amount of flexibility is central to the codesign process
- Tensor Processing Core:
  - Scalar + SIMD Vector + Matrix GEMM
- SW & HW IF: Traditional CPU Architecture ISA
  - Build a NN Supercomputer, rather than a NN coprocessor chip
- Matrix Unit:
  - 128 x 128 Systolic MAC Array
  - bfloat16 multiplies
  - float32 accumulate

Huge Impact to other successors



TPUv1 as a slave coprocessor of X86 Host Server



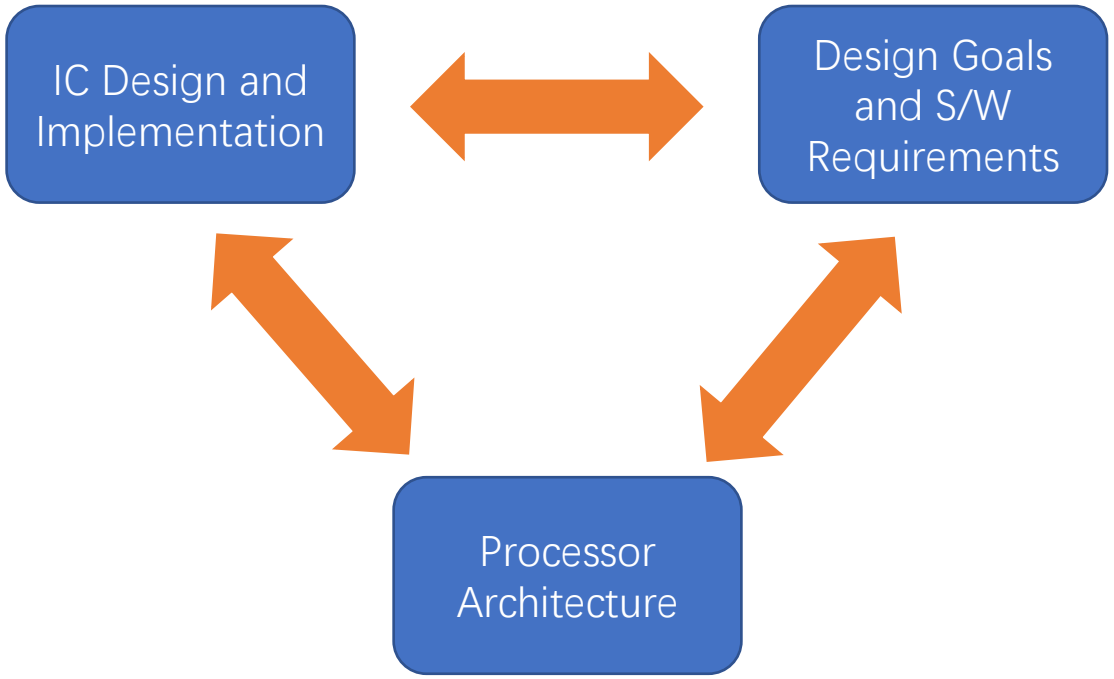
TPUv2 Core — A Turing-Complete Processor Core

DSA Design —— S/W & H/W Codesign

David Patterson: Next Decade is A New Golden Age for Domain Specific Architecture

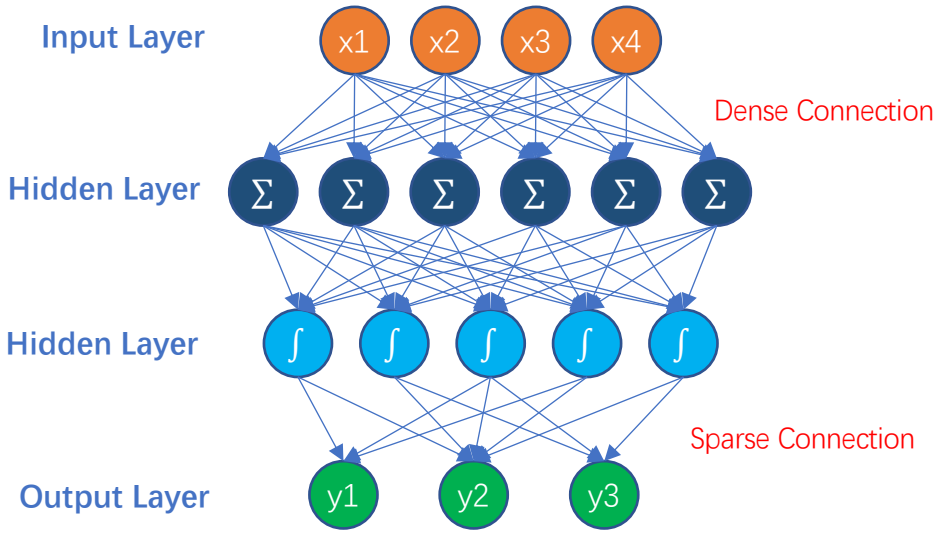
- What is DSA Processor?
  - A processor runs a few categories of specific workloads, but does very well (high-performance).

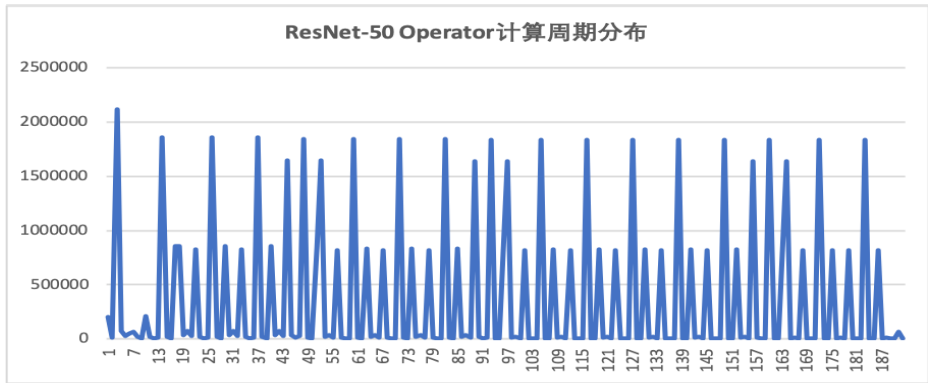
- DSA Processor S/W & H/W Codesign Guidelines:
  - Control Path: Analyze the control logic of your program
  - Data Path: Analyze the data movement pattern
  - Analyze the computation intensity
  - What is your PPA goal of your processor?
  - Other Design Goals…

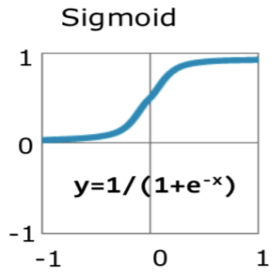


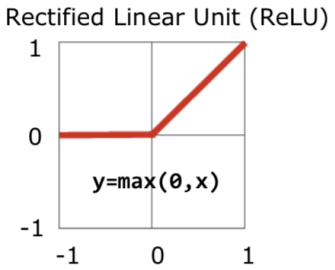
DSA Design —— NN Computation Graph Program Analysis

- Characters of Neural Network Computation Workload Programs:









- Graph-based workload and dataflow
- Intensive Tensor-based Computation
- Heavy Memory Access (Weight Params Load)
- Non-Linear Activation Function: sqrt, exp, recip etc.
- Asymmetrical Computation Intensity:
  - Dense vs Sparse

DSA Design —— Tradeoff among various design factors

- Target User Scenario & Performance:
  - Training or Inference?
  - Silicon PPA Target?
- How much chip resources can be invested into PEs?
  - How do you design your PE Array?
- What is Memory Model Architecture? How to solve Memory Wall Problem?
  - How much on-chip SRAM?
  - How your data is moved? What is your memory bandwidth need?
- Supported Data Types:
  - Training: FP32 is must-have. Support BFP16?
  - Inference: FP16? INT16?
  - INT8 Quantization? Or even INT4?
- Parallelism Strategy?
  - Instruction Level Parallelism: VLIW or SIMD? Superscalar, Single or Multi issue?
  - What is your program parallelism model?
- How to reduce data movement?
  - Batches: Reuse weights once fetched from memory across multiple inputs
  - Apply compression to exploit redundancy in data —— requires additional hardware
- How to increase PE utilization rate?
  - What is your DNN model mapping solution?
- Reduce time and energy per MAC
  - Reduce precision => If precision varies, what is the impact to accuracy?
- Reduce unnecessary MACs?
  - Exploit sparsity => requires additional hardware; impact on accuracy
  - Exploit redundant operations => requires additional hardware

DSA Design —— Key Metrics to Evaluate

- Peak Performance:
  - TOPS/TFLOPS: # of MACs * Freq
- Real Performance —— Throughput & Latency:
  - Real ResNet-50/BERT throughput (fps) with different batch size?
  - Latency with different batch size?
  - MLPerf Performance Benchmark Suite
- Energy and Power:
  - What is your peak power consumption (# Watt)?
  - What is your “throughput/watt”?
- Computation Accuracy
  - What is your precision/accuracy loss? (compared with golden data on CPU/GPU)
- Hardware Cost ($$$)?
- Flexibility and Programmability:
  - Range of supported DNN Models and Workloads?
  - Easy to customize operator function?
- Scalability:
  - Scaling of performance with amount of resources?

Agenda

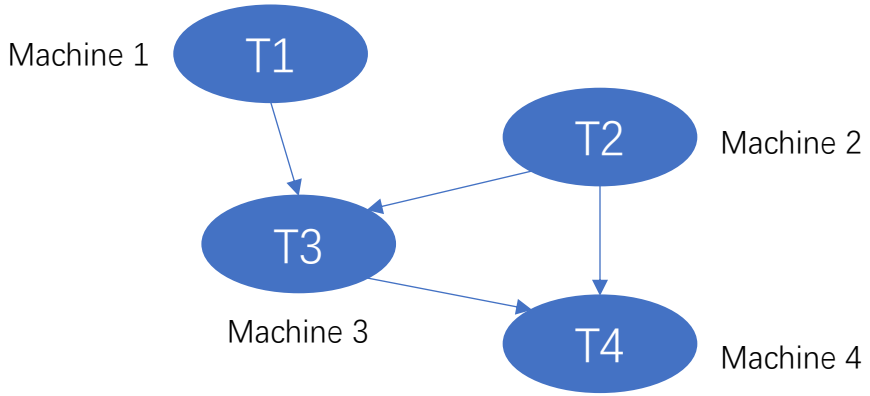
- Market Analysis of AI Processor
- Architecture of AI Processor
  - Evolution of AI Processor Architecture
  - Key Metrics of AI Processor Architecture
- Software of AI Processor
  - Program Parallelism Strategy
  - Data Sharding of Matrix Multiplication
  - Graph Compiler
- Summary

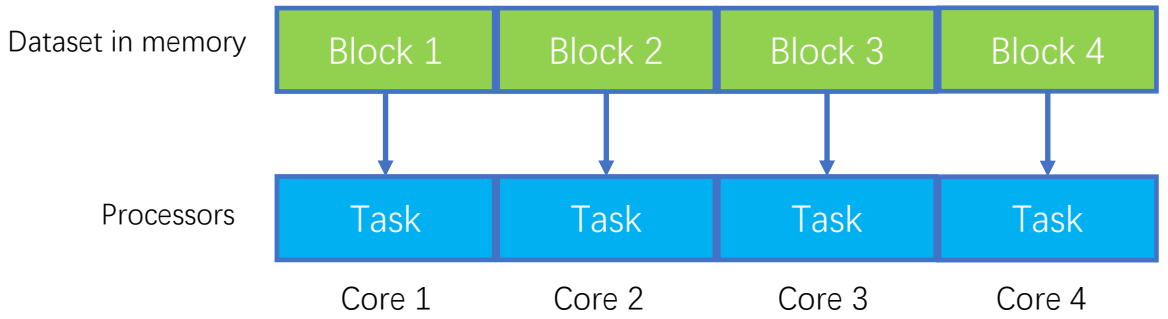
Parallelism Strategy — Task Decomposition vs Data Decomposition

- How do we parallelize a task? We need to *decompose* problems to achieve parallelism:
  - We can decompose a task on either *the control* or *the data*

Task Decomposition Parallelism	• Decompose a big task into a few small sub-tasks. • The output of a previous sub-task will be the input of next sub-task. • Running all sub-tasks on different machines/processors • aka: *sub-task pipeline parallelism*

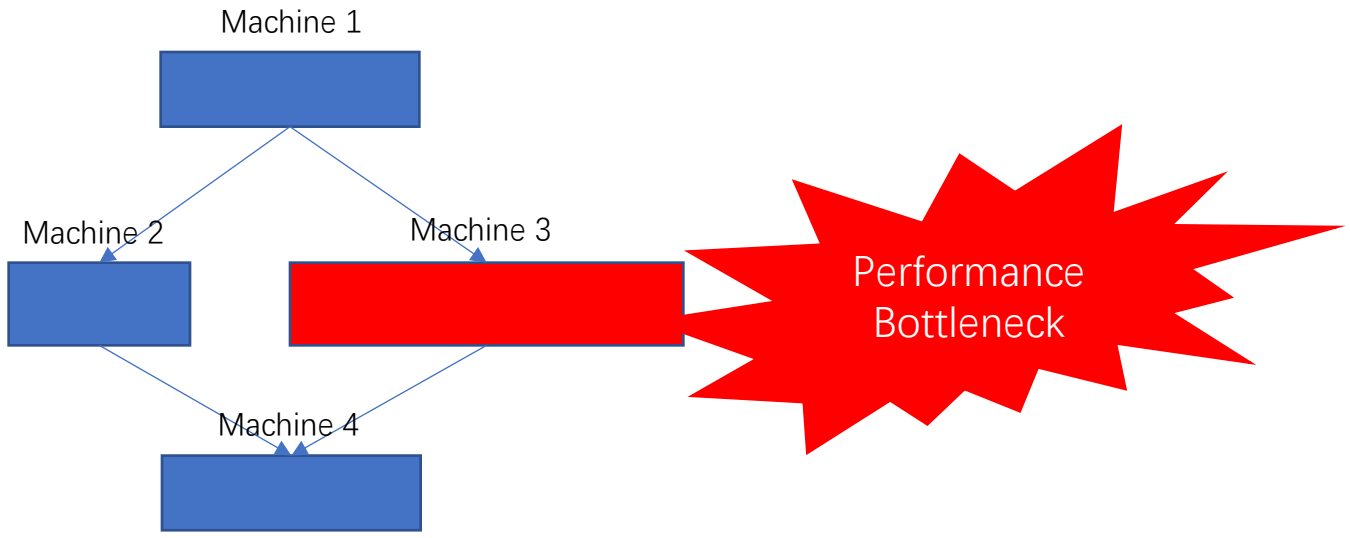
Data Decomposition Parallelism	• All machines execute the same task, but handle different partition of the whole dataset. • The degree of parallelism comes from the data decomposition and grows with the number of parallel machines • very high speedups, and usually very fine-grained (e.g., SIMD vectorization)

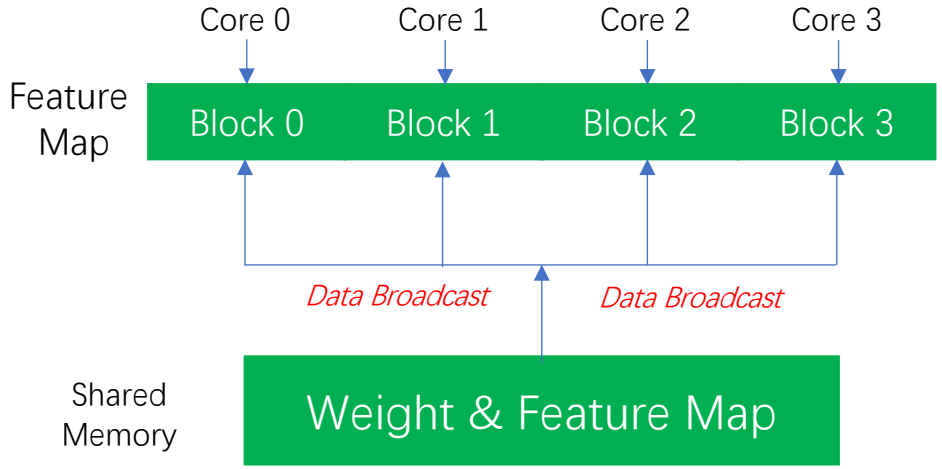




Parallelism Strategy — Task Decomposition vs Data Decomposition

	Task Decomposition Parallelism	Data Decomposition Parallelism
pros	• In Theory, by *handy* decomposition we can achieve maximum performance	• Easiest & Consistent Parallelism Model • Can achieve high speedups, and fine granularity • No handy sub-tasks balance issues
cons	• No Unique Decomposition Solution • The number of sub-tasks determines the parallelism degree. • Extra communication overhead is introduced by data exchange among sub-tasks • Tasks Granularity is the key: *fine-grained* vs coarse grained decomposition	• Memory Bandwidth Pressure by Parallel Cores access shared memory • Latency will grow with the bigger dataset or batch size





Parallelism Strategy — Amdahl Law: Principle to evaluate your program performance

- The fundamental law of parallel computing：define the theoretical speedup that a program can be parallelized:

$$\text{Speedup} = \frac{1}{\frac{P}{N} + S}$$

Where: (1) P — the percentage of the execution time in a program can be parallelized. (2) S — the percentage of the sequential execution time in a program, which will not benefit from the improved parallel resources. S = 1 - P. (3)N — The number of parallel computing resources.

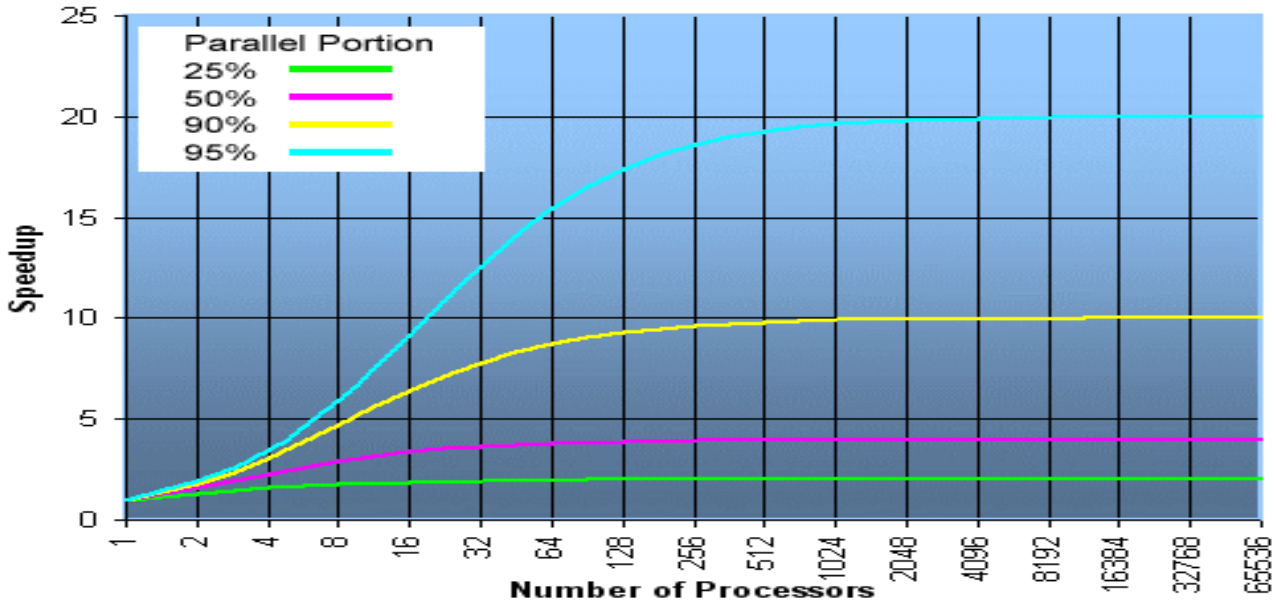
- The upper limitation of parallel speedup：

$$\text{Max Speedup} = \frac{1}{1 - P}$$

- Amdahl Law Famous Hint: *You can spend a lifetime getting 95% of your code to be parallel, and never achieve better than 20x speedup no matter how many processors you throw at it!*

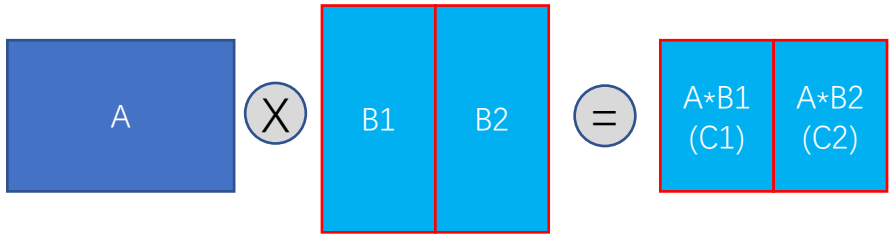
NOTICE: ***P*** of NN Model Program is important to us!!!

Parallelism Speedup (Various parallelized fraction "P")				
N	0.50	0.90	0.95	0.99
4	1.60	3.08	3.48	3.88
8	1.78	4.71	5.93	7.48
16	1.88	6.40	9.14	13.91
32	1.94	7.80	12.55	24.43
64	1.97	8.77	15.42	39.26
100	1.98	9.17	16.81	50.25
1000	2.00	9.91	19.63	90.99
10000	2.00	9.99	19.96	99.02
100000	2.00	10.00	20.00	99.90



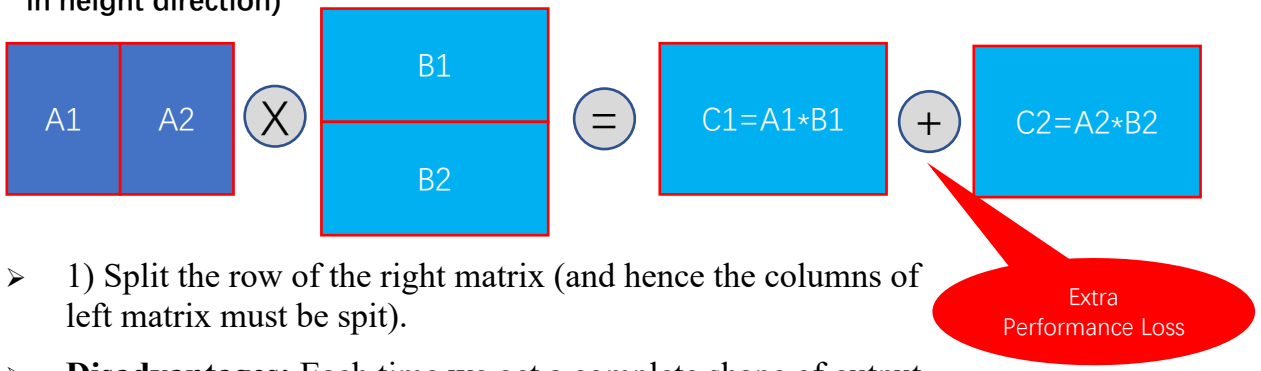
Matrix Multiplication Data-Sharding Solutions

Option #1: Split the columns of right matrix



- 1) Only split the columns of right matrix, no split to the left matrix; 2) Reduce the size of input right matrix, and hence reduce the size of output matrix, but each time we get a complete output result.
- **Disadvantages:** For each block of right matrix and output, memory address are not continuous.

Option #2: Split the height of right matrix (and hence the left matrix needs to be split in height direction)

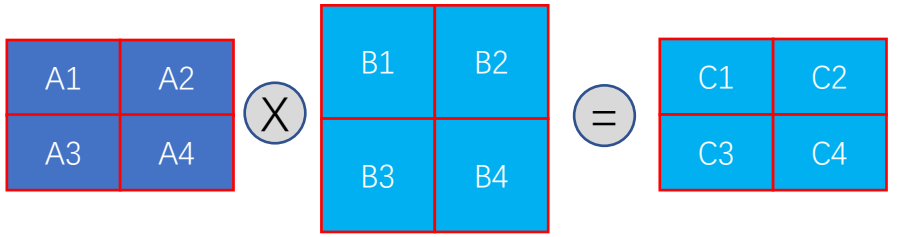


- 1) Split the row of the right matrix (and hence the columns of left matrix must be spit).
- **Disadvantages:** Each time we get a complete shape of output matrix, but only partial result. Extra accumulation is needed.

Option #3: Split the height of left matrix

- 1) Split the row of the left matrix; 2) Each time we get half of final output matrix; 3) The memory address is continuous for all blocks (A1, A2, C1, C2)
- The option is useful when size of left matrix is very big

Option #4: Practical Solution

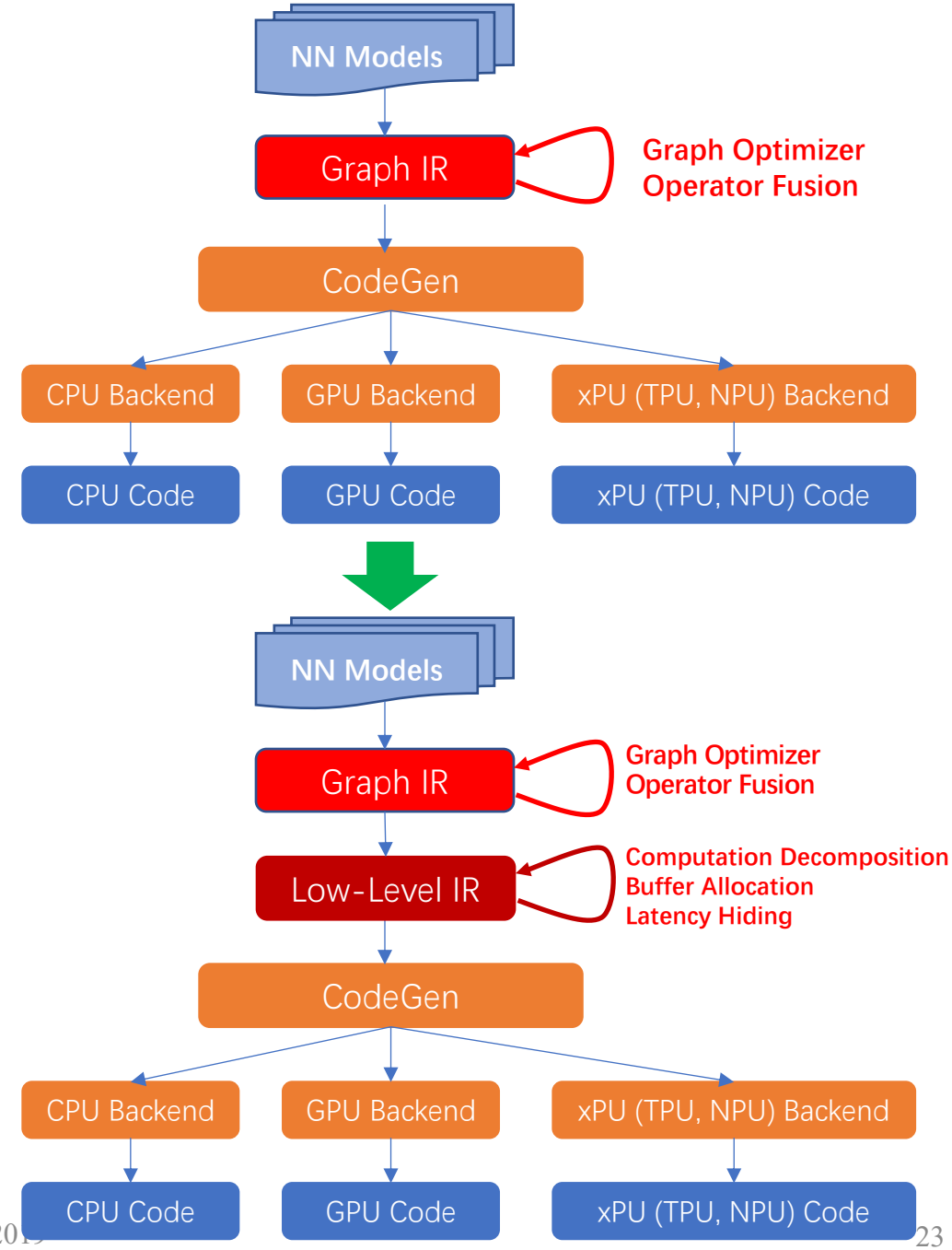


$$C1 = A1 * B1 + A2 * B3$$
$$C2 = A1 * B2 + A2 * B4$$
$$C3 = A3 * B1 + A4 * B3$$
$$C4 = A3 * B2 + A4 * B4$$

- Practical Sharding solution to feed data to your Matrix MAC Array
- Impact your operator implementation, performance, buffer allocation, etc.

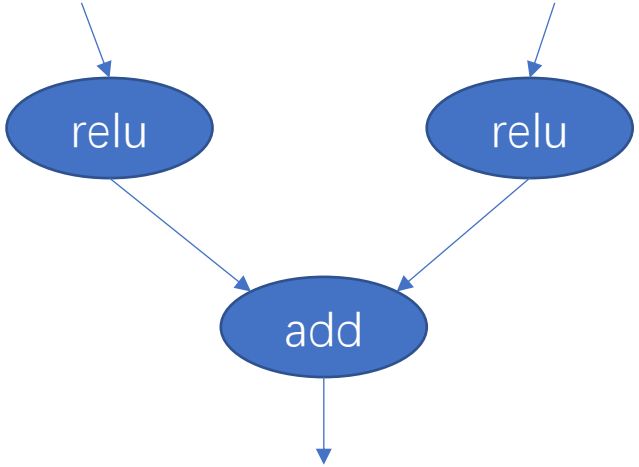
Neural Network Compiler (aka: Graph Compiler)

- A key component for AI Processor S/W stack:
  - Map NN Workloads to low-level AI hardware
- Graph IR (e.g: NNVM/Relay IR etc)
  - Generate an unified graph representation
  - General Graph Optimization:
    - target-independent op fusion
    - Dead code elimination
    - Constant propagation
- Challenges: How to generate optimized codes for each operators for different target backend? The problems to rely on ops library like: cuDNN/mkl-DNN:
  - Can't do target-dependent further op fusion optimization
  - Can't do optimization using compile-time constants (e.g: tensor shape)
  - Can't do graph-level optimization depending on target architecture
- Solution: Low-Level IR
  - Address the huge difference between Graph-IR and Target Codes
  - Target Dependent Optimization through Progressive IR Lowering



Operator Fusion Example

- Operator Fusion is a popular optimization mechanism for almost all NN Compilers
- Operator Fusion is very efficient for Element-Wise Operation in NN Graph:
  - Improve the ratio of Computation/MemAccess
  - Reduce the times to access SRAM
- Example Pattern: Two Relu Ops + Add Op, N * 64 FP16, Vector 1024bit




```
/* Pseudo codes for relu */
Loop:
      vle.v       v0, (t0)
      addi        t0, t0, 128
      vfmax.vf v0, v0, r0
      vse.v       v0, (t0)
```

```
/* Pseudo codes for add */
Loop:
      vle.v       v0, (t0)
      vle.v       v1, (t1)
      addi        t0, t0, 128
      addi        t1, t1, 128
      vfadd.vv    v0, v0, v1
      vse.v       v0, (t2)
      addi        t2, t2, 128
```

- Each Relu Op (Assume 1 CPI)
  - Cycles: N * 4
  - Mem Access: N * 2
- Add Op:
  - Cycles: N * 7
  - Mem Access: N * 3
- Overall:
  - Cycles: 2 * (N*4) + (N*7) = 15 * N
  - Mem Access: 7 * N

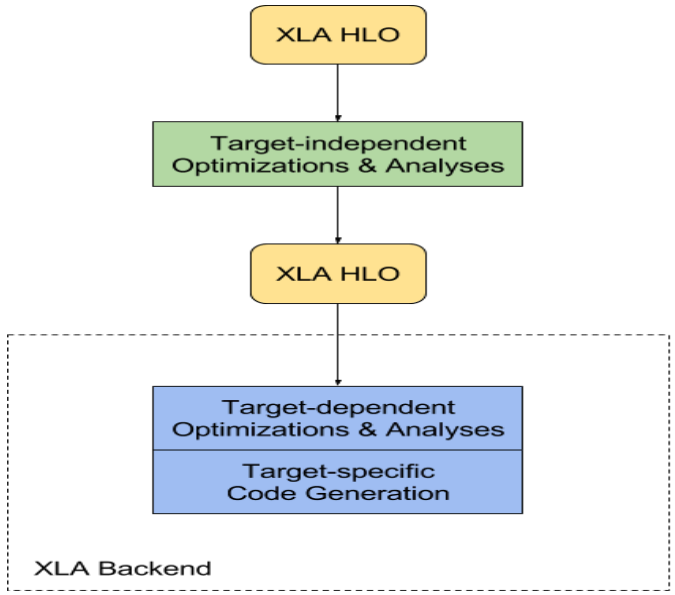
Less Cycles

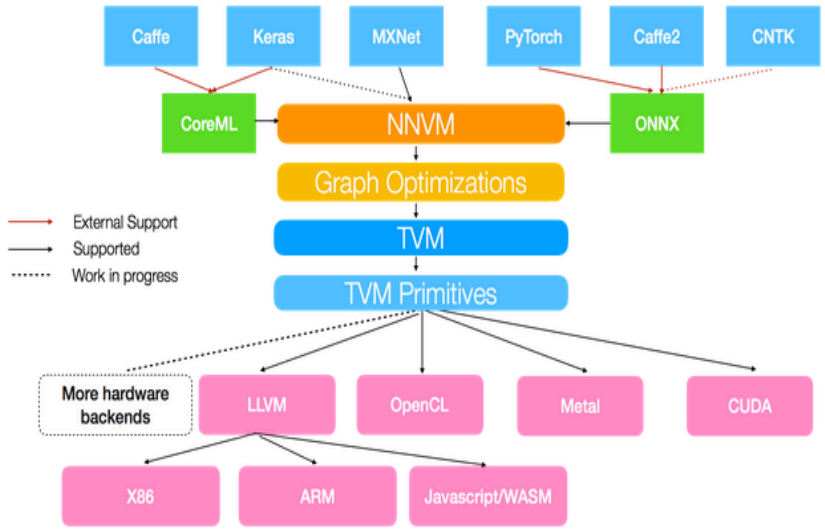
Redundant MemAccess are eleminated

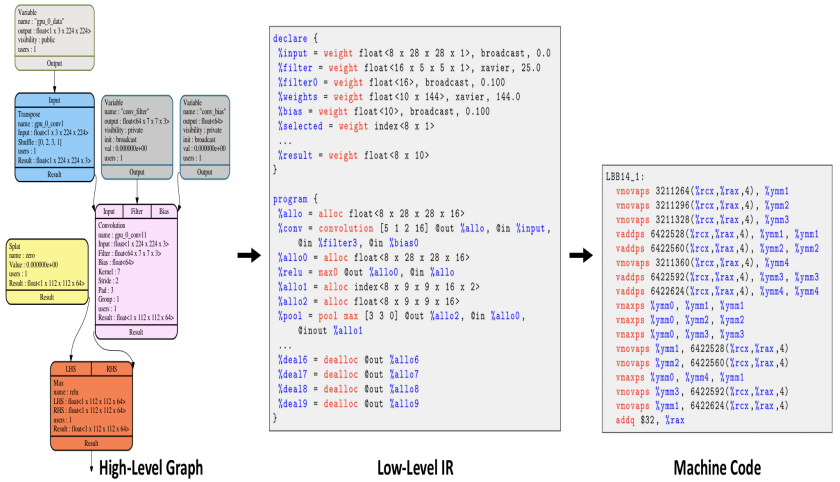
```
/* Pseudo codes for fused ops:
Loop:
      vle.v       v0, (t0)
      vle.v       v1, (t1)
      addi        t0, t0, 128
      addi        t1, t1, 128
      vfmax.vf    v0, v0, r0
      vfmax.vf    v1, v1, r0
      vfadd.vv    v0, v0, v1
      vse.v       v0, (t2)
      addi        t2, t2, 128
```

- Fused Op ((Assume 1 CPI):
  - Cycles: 9 * N
  - Mem Access: 3 * N

Overview of NN Compilers in Community







Tensorflow XLA

- HLO IR: DSL to represent BAISC Linear Algebra in various NN operators
  - E.g: A softmax op can be split and represented by a combination of basic "exp, div, reduce"operations.
- Current XLA Framework is lack of low-level IR:
  - Highly depends on manually optimized ops library

TVM

- A end-2-end complete NN Compiler stack
- Graph-IR: NNVM/Relay, supports different NN Models (Tensorflow, PyTorch etc)
- Low-level IR: Halide IR extension, based on Polyhedral model, A "compute/schedule" separate design.
- TVM Weakness:
  - Can't do graph global optimization in low-level Halide IR, schedule optimization only can be in operator-level
  - Limited Schedule Rules

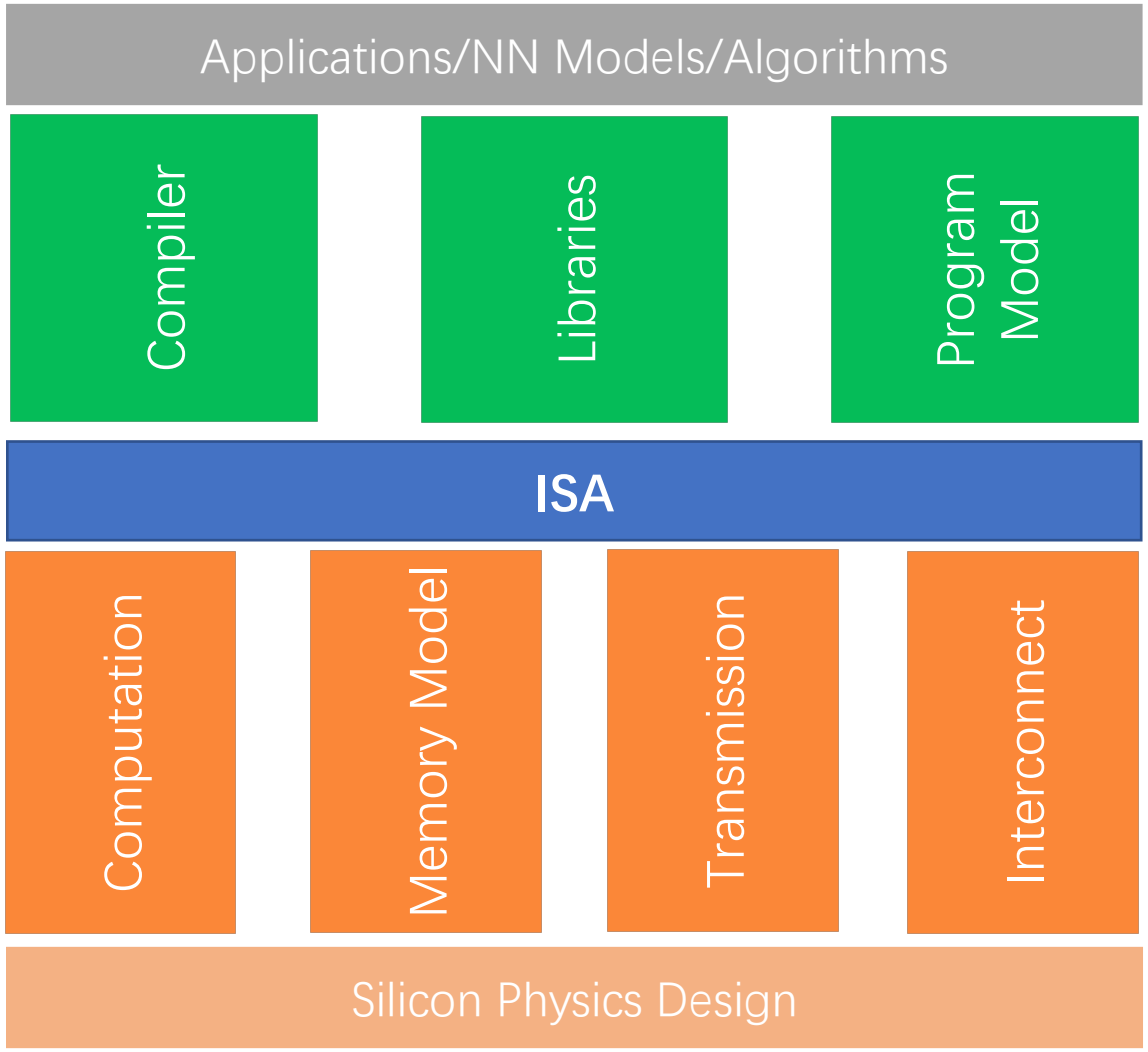
PyTorch Glow

- A rising star: NN Compiler for PyTorch
- Glow High-level IR: Graph-Level IR
- Glow Lowered High-Level IR: Tensor IR
- Glow Low-level IR: w/ global graph info
  - Limited. Only a Memory-level IR
  - No loop-level IR, computation decomposition ability

Google MLIR — Multi-Level IR Framework

- From Compiler GURU “Chris Lattner”!
- Multi-Level IR Framework, Major Design goals: Extend the LLVM’s Codegen framework & Arch-independent optimizations into domain-specific IRs, e.g: Neural Network Compiler is one of the major target domains.
  - Progressive Lowering Multi-level IR framework
  - Help compiler engineer to design your own IR (aka: dialect) in different level to address different problems/needs.
- The following facilities are provided by MLIR Framework:
  - An unified IR framework from high-level IR to low-level IR
  - General compilation optimization algorithms common to different level dialects
  - IR definition and transform framework based on LLVM TableGen, to simplify IR definition, optimization and lowering implementation
  - Common types in neural network domain, e.g: tensor, memref etc
  - Polyhedral-based algorithms implementation for Code transformation, buffer management, DMA code injection etc.
- **Highlight**: MLIR is NOT a complete NN compiler implementation, it is a framework to help you to implement your own NN compiler.

Summary of AI Processor Architecture



Q&A

Thanks!