Advanced Computer Architecture

高级计算机体系结构

第17讲：TLP (3)

张献伟

xianweiz.github.io

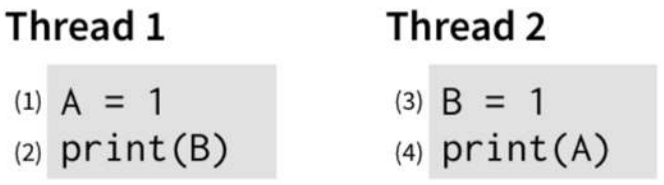
DCS5637, 12/21/2022

Review Questions

- Limits of snooping protocol
  Broadcast-based, hard to scale for many processors.
- For directory-based protocol, what is the entry content?
  Dirty bit + presence bits.
- Explain the storage overhead of dir-protocol?
  One entry per memory line, presence bits for all nodes/processors.
- Schemes to reduce storage overhead?
  Limited pointer scheme (P), sparse directory (M).
- What is ‘home node’ in dir-protocol?
  Node with memory holding the corresponding data for the line
- Coherence vs. consistency?
  Same vs different location, eventually vs when, cache vs. mem, …

Memory Consistency Model[一致性模型]

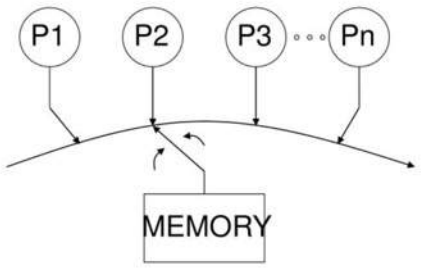
- **Memory consistency model** (or just “**memory model**”) defines the allowed orderings of multiple threads on a multiprocessor
  - SC is one such model
    - E.g., orderings that print 01/11 are allowed, but not 00
- A memory consistency model is a <u>contract</u> between the hw and sw
  - The hw promises to only reorder operations in ways allowed by the model[硬件承诺]
  - In return, the sw acknowledges that all such reorderings are possible and that is needs to account for them[软件认可]



```
Thread 1            Thread 2

(1) A = 1           (3) B = 1
(2) print(B)        (4) print(A)
```

Issues of SC[问题]

- SC: just like a switch to select thread to run, and runs its next event completely
  - Events happen in program order



- SC presents a simple programming paradigm
- But, SC reduces potential performance
  - Especially in a multiprocessor with a large number of processors or long interconnect delays
- Simplest way to implement SC
  - A processor delays the completion of any memory access until all the invalidations caused by that access are completed
  - Example: for a write miss, four processors share a block
    - 170 cycles for write: 50 cycles to establish ownership, then 10 cycles to issue each invalidate, and 80 cycles for an invalidate to complete and be acknowledged (50 + 40 + 80)

Optimizations[优化]

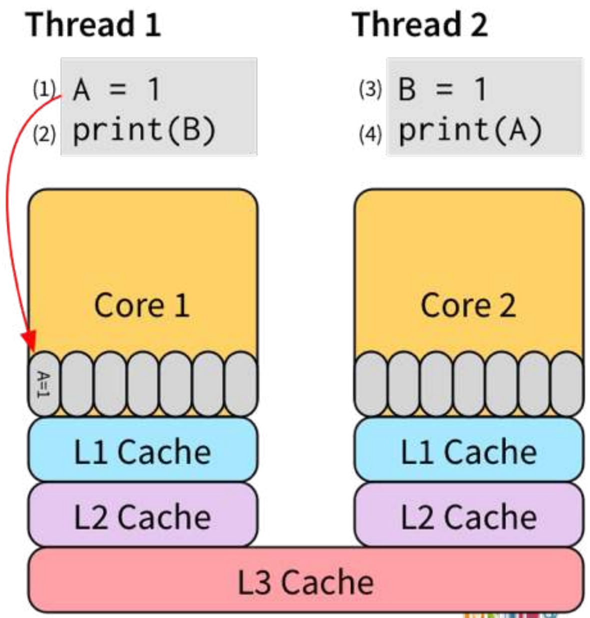
- **Goal**: develop a model that is simple to explain and yet allows a high performance implementation[好理解、高性能]

- **Solution-1**: develop ambitious implementations that preserve SC but use latency-hiding techniques to reduce the penalty[保持SC、隐藏时延]

- **Solution-2**: develop less restrictive memory consistency models that allow for faster hw[放宽顺序要求]
  - Such models can affect how the programmer sees the multiprocessor

The Example

- SC maintains a single view of memory
  - Cannot run *(2)* until *(1)* has become visible to every other thread
- No reason why *(2)* needs to wait until *(1)* completes
  - *(2)*: a read from B, *(1)*: a write to A
  - They don't interfere with each other at all
    - So should be allowed to run in parallel
  - Note that event *(1)* is very slow
    - A very high overhead
- SC greatly hurts performance
  - The model should be relaxed!!!
    - Event *(2)* should not wait for *(1)*

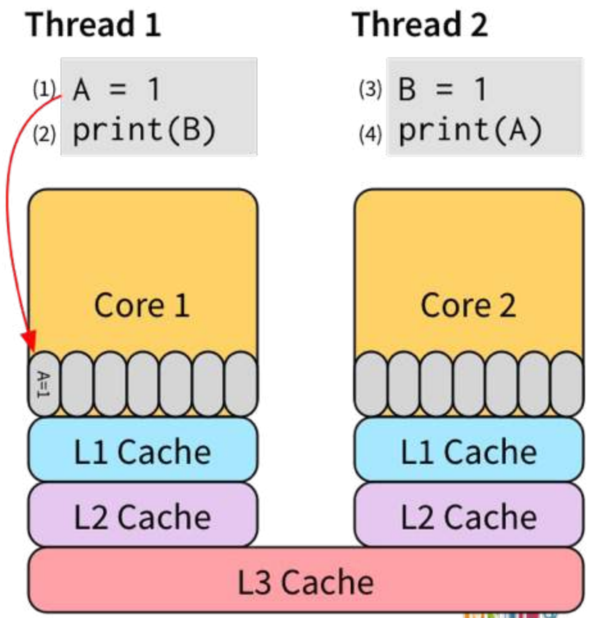


https://www.cs.utexas.edu/~bornholt/post/memory-models.html

The Example (cont.)

- Place *write(1)* into a store buffer, rather than waiting for it to become visible
  - Then *(2)* could start immediately, rather than waiting for *(1)* to reach the L3
  - The store buffer is on-core: very fast to access
  - At some time in the future, the cache hierarchy will pull the write from the store buffer and propagate it through the L3 so that it becomes visible to other threads
- The buffer helps hide the write latency
- Preserves single-threaded behavior
  - Access: store buffer → memory

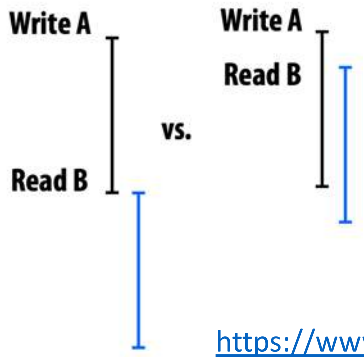
Is it still cache coherent?
Is the result correct?

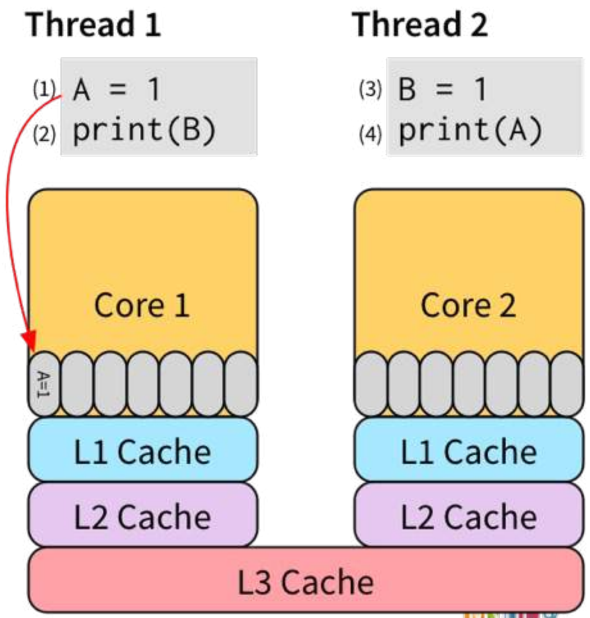


https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Total Store Ordering[TSO一致性]

- TSO mostly preserves the same guarantees as SC, except that it allows the use of store buffers
  - There buffers hide write latency, making execution significantly faster
- Retains ordering among writes (that's why called 'total store ordering')[保证写顺序]
  - Relaxed only the W→R ordering
- Performance gain
  - Allow processor to hide latency of writes when later read is independent



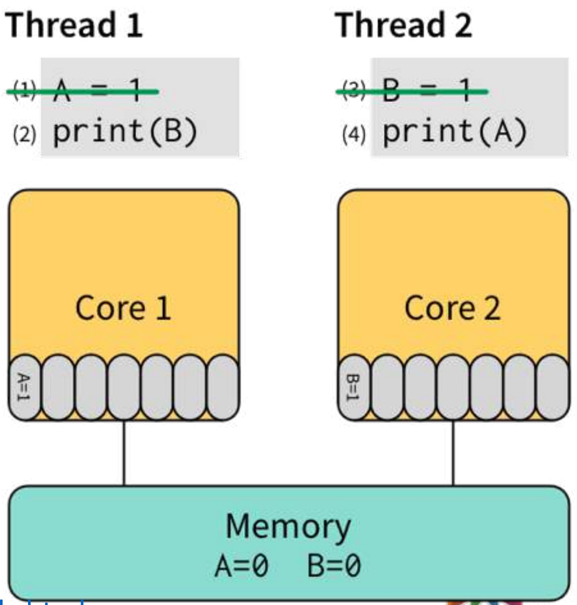


https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Total Store Ordering (cont.)

- While boosting performance, TSO allows behaviors that SC does not
  - I.e., programs running on TSO hw can exhibit behavior that programmers would find suprising

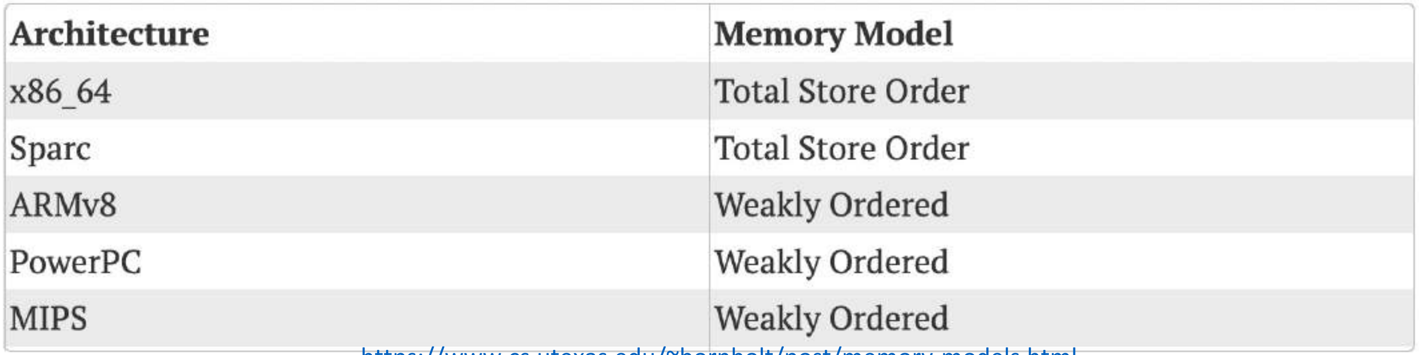
- The example: both threads first check their local store buffer, but fails to locate and then fetches from memory
  - This program can print 00
    - B=1 not in Core-1's buffer
    - A=1 not in Core-2's buffer
  - TSO cannot put into practices ???



https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Ordering on Different Architectures

- Actually, <u>every</u> modern architecture includes a store buffer, and so has a memory model at least as weak as TSO
  - x86 specifies a memory model that is very close to TSO
    - among the most well-behaved architectures in terms of the crazy behaviors it allows
  - ARM memory model is notoriously underspecified, but is essentially a form of *weak ordering*, gives very few guarantees
    - RISC-V: "RVWMO" (RISC-V Weak Memory Ordering)
    - Weak ordering allows almost any operation to be reordered, good for hardware optimizations but nightmare to program at the lowest levels



Architecture	Memory Model
x86_64	Total Store Order
Sparc	Total Store Order
ARMv8	Weakly Ordered
PowerPC	Weakly Ordered
MIPS	Weakly Ordered

https://www.cs.utexas.edu/~bornholt/post/memory-models.html
https://kernelgo.org/memory-model.html

Partial Store Ordering[PSO一致性]

- In TSO, only W→R order is relaxed
  - The W→W constraint still exists
    - Writes by the same thread are not reordered (they occur in program order)
- In **partial store ordering** (PSO), W → W is also relaxed
- Example: *A* and *flag* are initially 0s
  - SC: print '1' (when flag is 1, *A* must be 1 already)
  - TSO: print '1' (ditto)
  - PSO: may print '0' (when *flag* is 1, *A* can be 0 or 1)

Thread 1 (on P1)	Thread 2 (on P2)

A = 1;
flag = 1;
	while (flag == 0); // spinning if flag is 0
	print A;

Aggressive Memory Ordering???

- SC maintains all four memory operation orderings
- Certain orderings can be violated ???
  - W→R: store buffer to allow read execute earlier
  - W→W: reorder writes in the store buffer
    - Earlier write is a cache miss, later is a hit
  - R→W, R→R: processor may reorder independent instructions
    - Out-of-order execution
  - Note that all are valid optimizations if a program consists of a single instruction stream[对单线程都有效]

- What if we discard all four memory orderings?
  - Still a memory consistency model (**Release Consistency**)

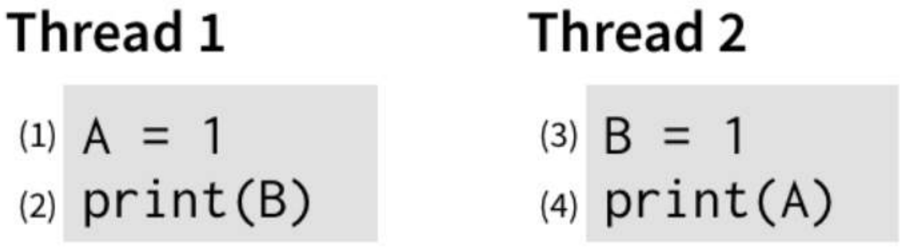
Release Consistency[RC一致性]

- Release Consistency (RC)
  - Processors support special synchronization operations
  - Memory accesses before memory fence instruction must complete before the fence issues
  - Memory accesses after fence cannot begin until fence instruction is complete
  - 硬件不再对一致性做过多保证，需要软件介入以控制执行行为

```
reorderable reads and writes here
...
MEMORY FENCE
...
reorderable reads and writes here
...
MEMORY FENCE
```

Express Synchronization[同步]

- ’00’ is not allowed in SC (the example)
  - Suppose architecture is of RC model, how to get the same effect with SC (i.e., no ‘00’)?
- All modern architectures include **synchronization** operations to bring their relaxed memory models under control when necessary
  - Most common operation: barrier (or fence)
- A barrier inst forces all memory operations before it to complete before any memory operation after it can begin
  - I.e., a barrier inst effectively <u>reinstates SC at a particular point</u> in program execution

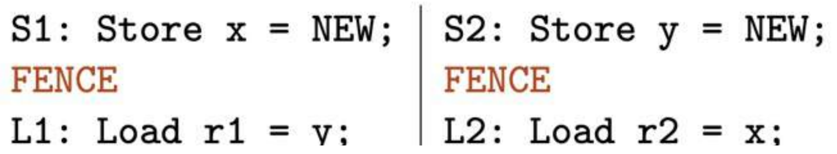


Thread 1

```
(1) A = 1
(2) print(B)
```

Thread 2

```
(3) B = 1
(4) print(A)
```

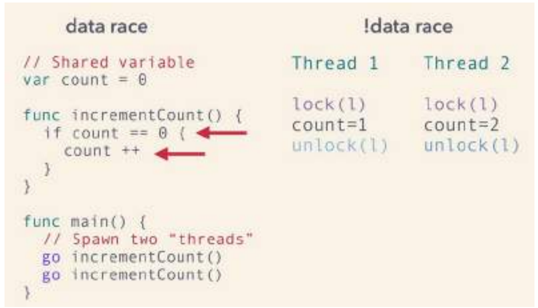


```
S1: Store x = NEW;  | S2: Store y = NEW;
FENCE               | FENCE
L1: Load r1 = y;    | L2: Load r2 = x;
```

FENCE: S1/S2 must be completely done before L1/L2

Synchronized Programs[同步程序]

- Two memory accesses by different processors conflict if
  - They access the same memory location
  - At least one is a write



data race

```
// Shared variable
var count = 0

func incrementCount() {
  if count == 0 {
    count ++
  }
}

func main() {
  // Spawn two "threads"
  go incrementCount()
  go incrementCount()
}
```

!data race

```
Thread 1      Thread 2

lock(l)       lock(l)
count=1       count=2
unlock(l)     unlock(l)
```

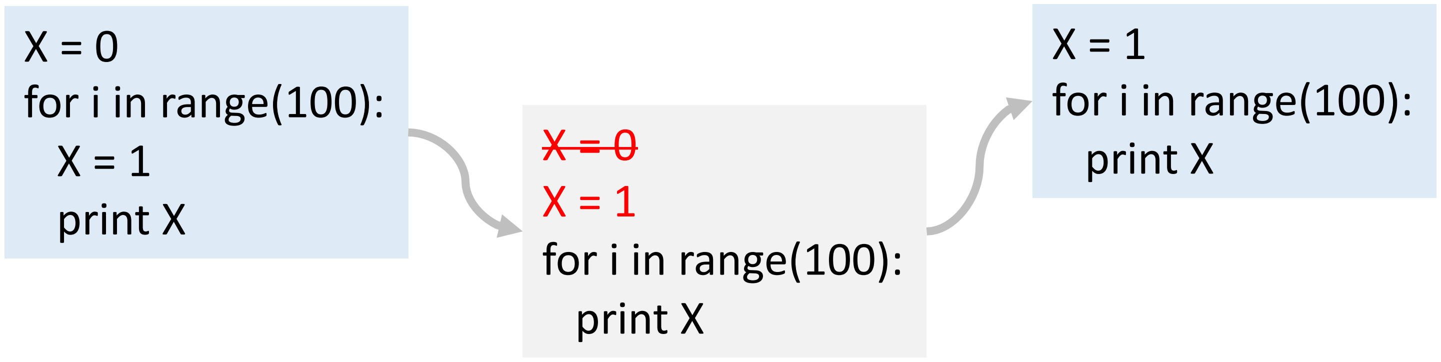
- Unsynchronized program
  - Conflicting accesses not ordered by synchronization (e.g., a fence, operation with release/acquire semantics, barrier, etc.)
  - Unsynchronized programs contain **data race**s: the output of the program depends on relative speed of processors (non-deterministic program results)
- In practice, most programs are synchronized (via locks, barriers, etc. implemented in synchronization libraries)
- Synchronized programs yield SC results on non-SC systems[程序在Relaxed Consistency上跑和在SC上跑结果一样]
  - Synchronized programs are **data-race-free** (DRF)

Summary: Relaxed Consistency

- Motivation: obtain higher performance by allowing reordering of memory operations (reordering is not allowed by SC)
  - Relaxed consistency models differ in which memory ordering constraints they ignore (e.g., TSO, PSO, RC)
- One cost is software complexity: programmer or compiler must correctly insert synchronization to ensure certain specific operation orderings when needed
  - Optimize for the <u>common case</u>: most memory accesses are not conflicting, so don't design a system that pays the cost as if they are
  - But in practice complexities encapsulated in libraries that provide intuitive primitives like lock/unlock, barrier (or lower level primitives like fence)

Compiler Reordering

- Besides hardware, compilers can also reorder memory operations
  - Example: the program prints a string of 100 '1's (always)
  - Possible to optimize the code?
    - Loop-invariant code motion: move the write outside the loop
    - Dead store elimination: remove X = 0
  - These two programs are totally equivalent
    - Produce the same output



```
X = 0
for i in range(100):
   X = 1
   print X
```

```
~~X = 0~~
X = 1
for i in range(100):
   print X
```

```
X = 1
for i in range(100):
   print X
```

Compiler Reordering (cont.)

- Now suppose there's another thread running in parallel with the program, and it performs a single write to X
  - The first program
    - It can print strings like 11101111 …, so long as there's only one single zero (because it will reset X = 1 on the next iteration)
  - The second program
    - It can print strings like 1110000 …, where once it starts printing 0s it never goes back to 1s
  - The first can never print 1110000…; the second cannot print 11011111…

With parallelism, the compiler optimization no longer produces an equivalent program.

T0
```
X = 0
for i in range(100):
   X = 1
   print X
```

T1
```
X = 0
```

T0
```
X = 1
for i in range(100):
   print X
```

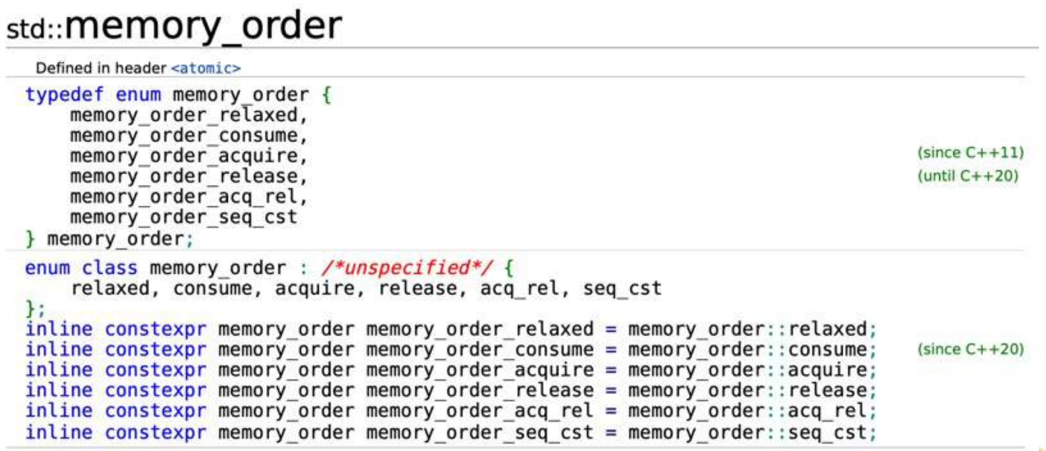
T1
```
X = 0
```

https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Languages' Memory Models[语言内存模型]

- The compiler optimization is effectively reordering
  - It's rearranging (and removing some) memory accesses in ways that may or may not be visible to programmers
- To preserve intuitive behavior, programming languages need memory models of their own,
  - To provide a contract to programmers about how their memory operations will be reordered
- Memory consistency at the program level

The memory model means that C++ code now has a **standardized library** to call regardless of who made the compiler and on what platform it's running. There's a standard way to control how different threads talk to the processor's memory.


std::memory_order

Defined in header <atomic>

```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
} memory_order;
```
(since C++11)
(until C++20)

```
enum class memory_order : /*unspecified*/ {
    relaxed, consume, acquire, release, acq_rel, seq_cst
};
inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
inline constexpr memory_order memory_order_consume = memory_order::consume;
inline constexpr memory_order memory_order_acquire = memory_order::acquire;
inline constexpr memory_order memory_order_release = memory_order::release;
inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
```
(since C++20)

https://www.cs.utexas.edu/~bornholt/post/memory-models.html

C++ Atomic[原子操作]

- Multithreading: concurrency, data race, thread sync
- Synchronization primitives: synchronize code to avoid race conditions in multithreading
  - std::**mutex**: very annoying, be cautious of deadlock
  - std::**atomic**: lock-free, efficient, usually for variables

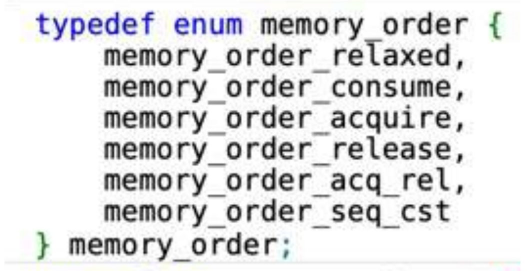
```
std::mutex mtx;
int num = 0;

void inc() {
  std::lock_guard<std::mutex> lock(mtx);
  num++;
}

int main() {
  std::thread t1(inc), t2(inc);
  return 0;
}
```

```
std::atomic<int> num = 0;

void inc() {
  num++;
}

int main() {
  std::thread t1(inc), t2(inc);
  return 0;
}
```

https://www.internalpointers.com/post/lock-free-multithreading-atomic-operations
http://www.max-sperling.bplaced.net/?p=1759

C++ Memory Model

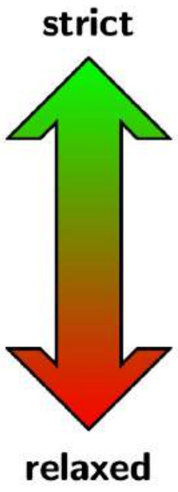
- The six memory orders can be combined with each other to achieve three ordering models
  - **Sequential consistent** ordering: achieves synchronization and guarantees a single total order
    - memory_order_seq_cst
  - **Acquire-release** ordering: implements synchronization, but does not guarantee global order consistency
    - memory_order_acquire / load
    - memory_order_release / store
    - memory_order_acq_rel / load, store, read-modify-write
    - memory_order_consume
  - **Relaxed ordering**: does not implement synchronization, but only guarantees atomicity
    - memory_order_relaxed



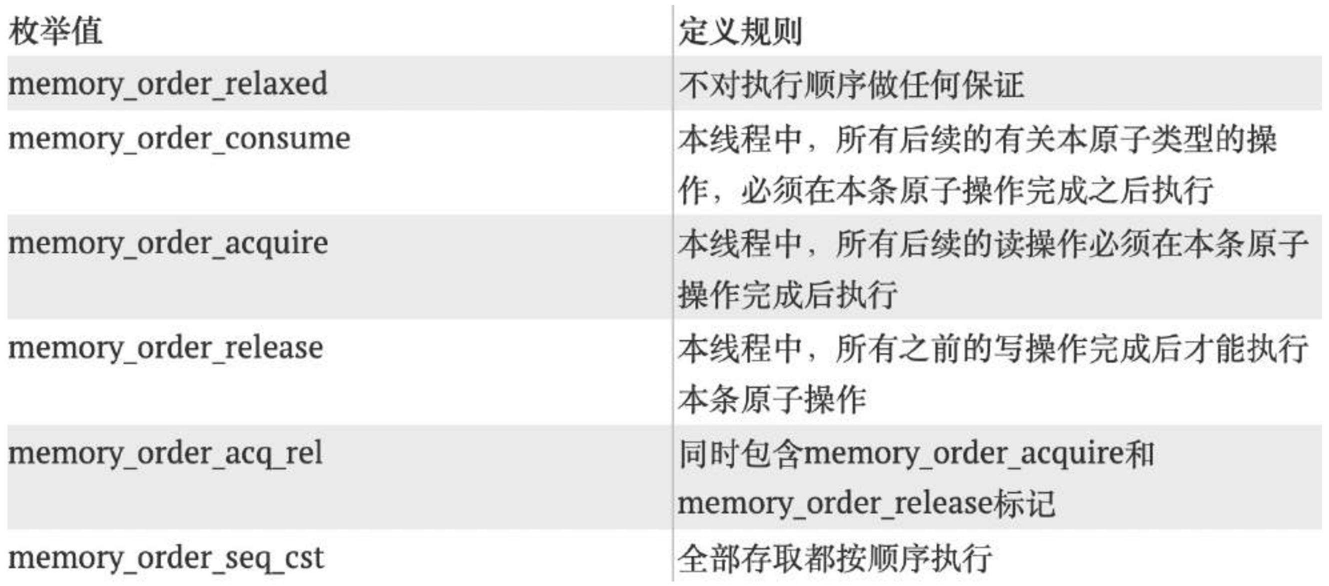
```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
} memory_order;
```

https://www.sobyte.net/post/2022-06/cpp-memory-order/

C++ Memory Model (cont.)



seq_cst	total order (SC for DRF code)
release / acquire	message passing
release / consume	
relaxed	no synchronization



枚举值	定义规则
memory_order_relaxed	不对执行顺序做任何保证
memory_order_consume	本线程中，所有后续的有关本原子类型的操作，必须在本条原子操作完成之后执行
memory_order_acquire	本线程中，所有后续的读操作必须在本条原子操作完成后执行
memory_order_release	本线程中，所有之前的写操作完成后才能执行本条原子操作
memory_order_acq_rel	同时包含memory_order_acquire和memory_order_release标记
memory_order_seq_cst	全部存取都按顺序执行

https://kernelgo.org/memory-model.html

Example

```
std::atomic<bool> x{false}, y{false};

void thread1() {
    x.store(true, std::memory_order_relaxed); // (1)
    y.store(true, std::memory_order_relaxed); // (2)
}

void thread2() {
    while (!y.load(std::memory_order_relaxed)); // (3)
    assert(x.load(std::memory_order_relaxed)); // (4)
}
```

no determined order between (1) and (2)

when loop exits, y has been true. I.e., (2) happened;
but possible that (1) has not been done → (4) may fail

```
std::atomic<bool> x{false}, y{false};

void thread1() {
    x.store(true, std::memory_order_relaxed); // (1)
    y.store(true, std::memory_order_release); // (2)
}

void thread2() {
    while (!y.load(std::memory_order_acquire)); // (3)
    assert(x.load(std::memory_order_relaxed)); // (4)
}
```

(1) happens before (2):
if (2) is visible, then all before release are visible

when loop exits, y has been true. I.e., (2) happened
(3) before (4) → (1) before (4) → (4) never fails

Summary of TLP

- Multiprocessors with thread-level parallelism
  - Sharing memory, having private caches
- Cache coherence
  - **Snooping**: every cache block is accompanied by the sharing status of that block
    - All cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary
  - **Directory-based**: a single location (directory) keeps track of the sharing status of a block of memory
    - Reduce storage and communication overheads
- Memory consistency
  - **Sequential consistency**: maintains all four memory operation orderings (W→R, R→R, R→W, W→W)
  - **Relaxed consistency**: allows certain orderings to be violated
    - TSO, PSO, RC

Advanced Computer Architecture

高 级 计 算 机 体 系 结 构

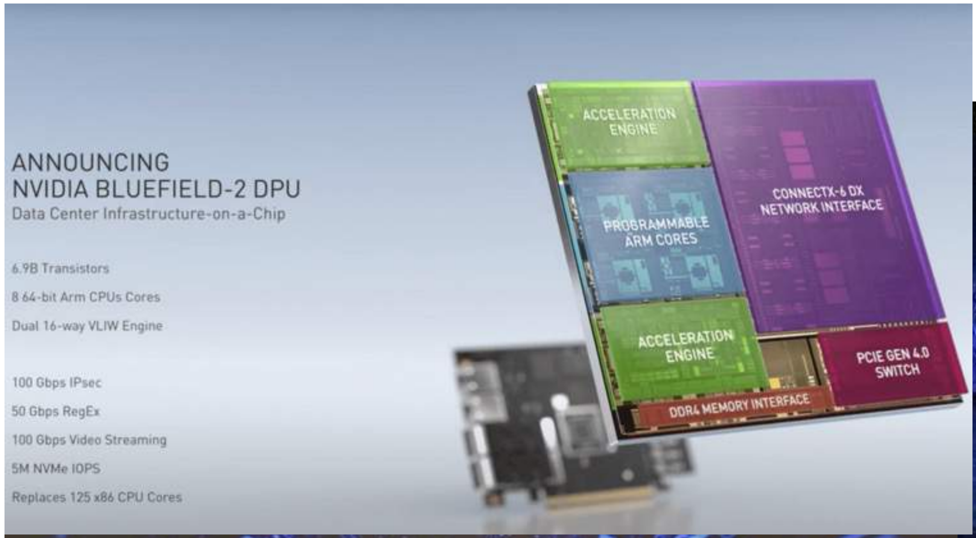
第17讲：Domain Specific Arch (1)

张献伟

xianweiz.github.io

DCS5637, 12/21/2022

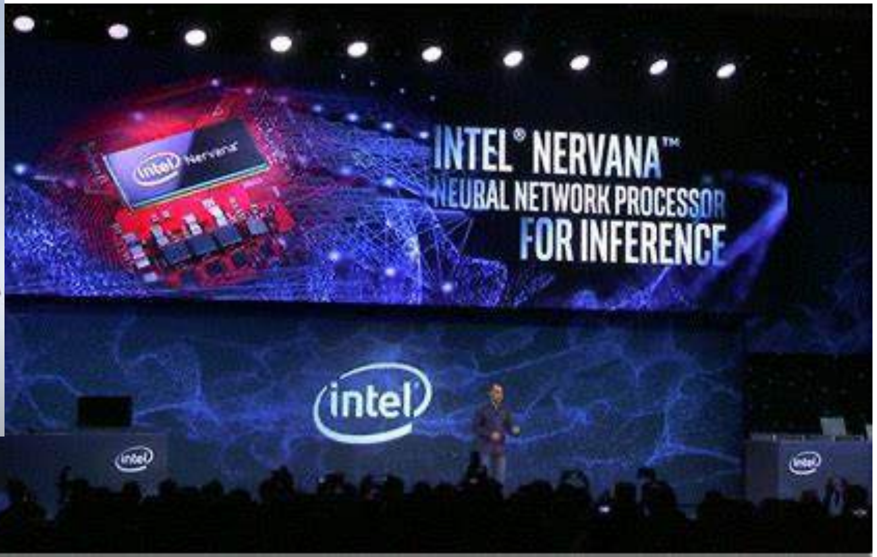
HW Companies Building Custom Chips


ANNOUNCING
NVIDIA BLUEFIELD-2 DPU
Data Center Infrastructure-on-a-Chip

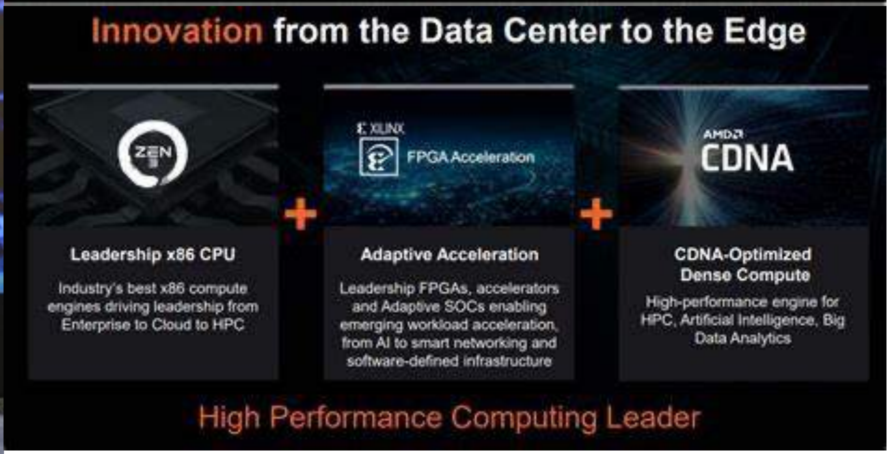
6.9B Transistors
8 64-bit Arm CPUs Cores
Dual 16-way VLIW Engine

100 Gbps IPsec
50 Gbps RegEx
100 Gbps Video Streaming
5M NVMe IOPS
Replaces 125 x86 CPU Cores

ACCELERATION ENGINE
PROGRAMMABLE ARM CORES
CONNECTX-6 DX NETWORK INTERFACE
ACCELERATION ENGINE
PCIE GEN 4.0 SWITCH
DDR4 MEMORY INTERFACE



Ascend 910
Meet the world's most powerful AI processor
算力最强的AI处理器 — 昇腾910正式推出
HUAWEI


Innovation from the Data Center to the Edge

ZEN	XILINX FPGA Acceleration	AMD CDNA
Leadership x86 CPU	Adaptive Acceleration	CDNA-Optimized Dense Compute
Industry's best x86 compute engines driving leadership from Enterprise to Cloud to HPC	Leadership FPGAs, accelerators and Adaptive SOCs enabling emerging workload acceleration, from AI to smart networking and software-defined infrastructure	High-performance engine for HPC, Artificial Intelligence, Big Data Analytics

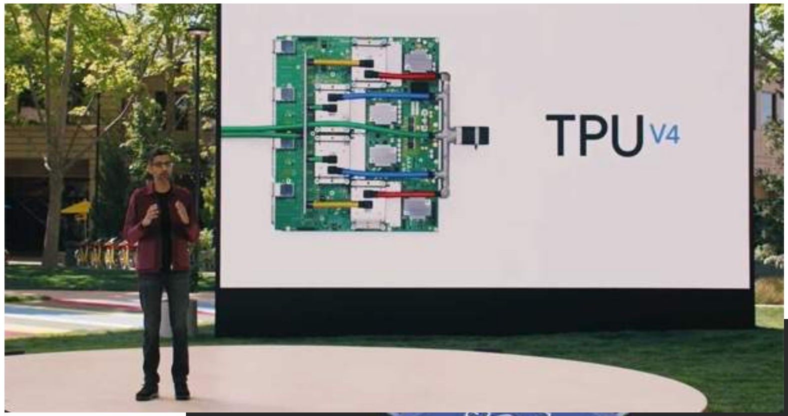
High Performance Computing Leader

SW Companies are Building HW

Chips Off the Old Block: Computers Are Taking Design Cues From Human Brains *(September 16, 2017)*

After training a speech-recognition algorithm, for example, Microsoft offers it up as an online service, and it actually starts identifying commands that people speak into their smartphones. **G.P.U.s are not quite as efficient during this stage of the process. So, many companies are now building chips specifically to do what the other chips have learned.**

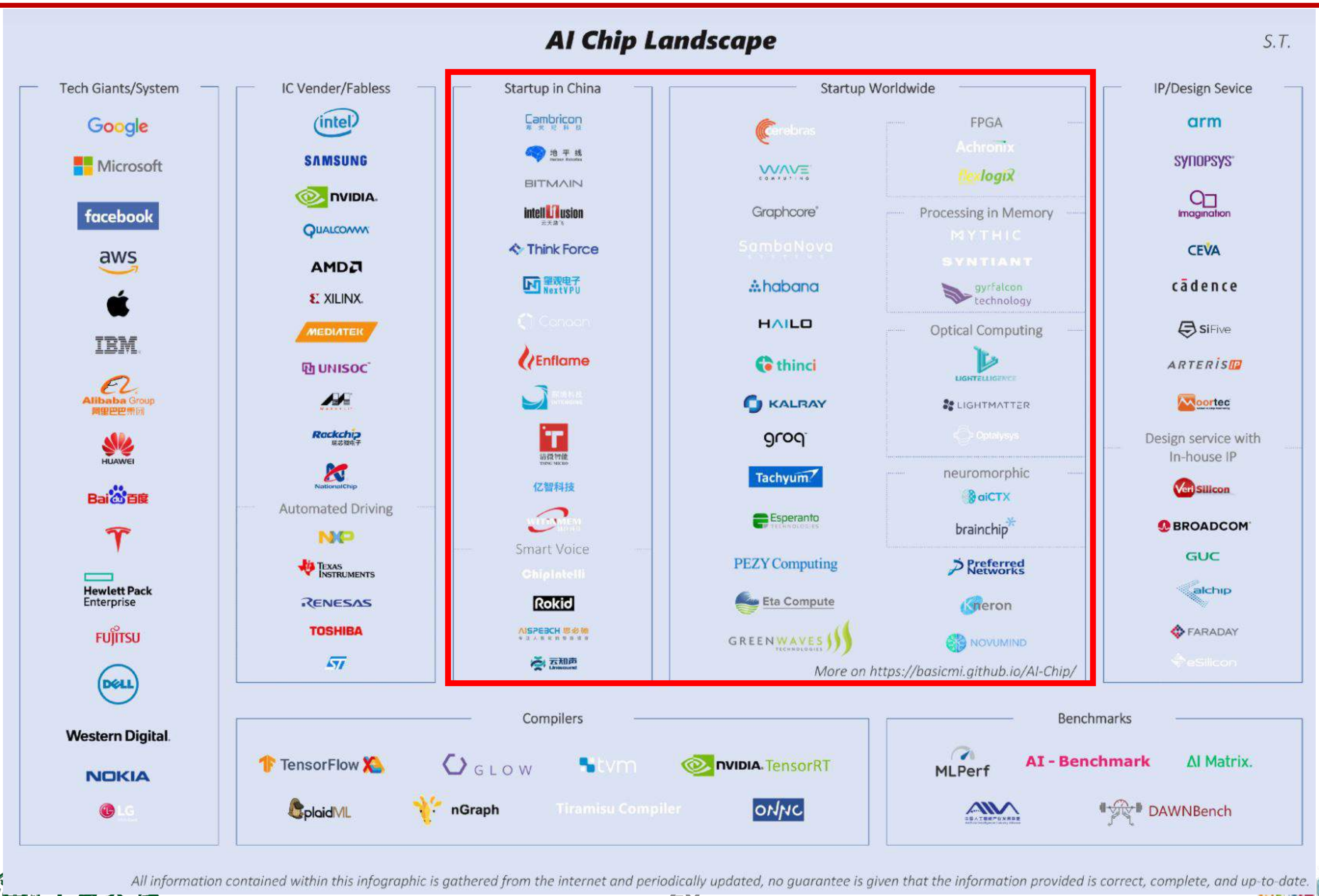
Google built its own specialty chip, a Tensor Processing Unit, or T.P.U. Nvidia is building a similar chip. **And Microsoft has reprogrammed specialized chips from Altera**, which was acquired by Intel, so that it too can run neural networks more easily.



AWS Inferentia
High performance machine learning inference chip, custom designed by AWS

平头哥
全球最高性能AI推理芯片含光800发布
Hanguang 800 - Meet the World's Most Powerful AI Inference Chip
78563 IPS 峰值性能
500 IPS/W 峰值能效
For Resnet

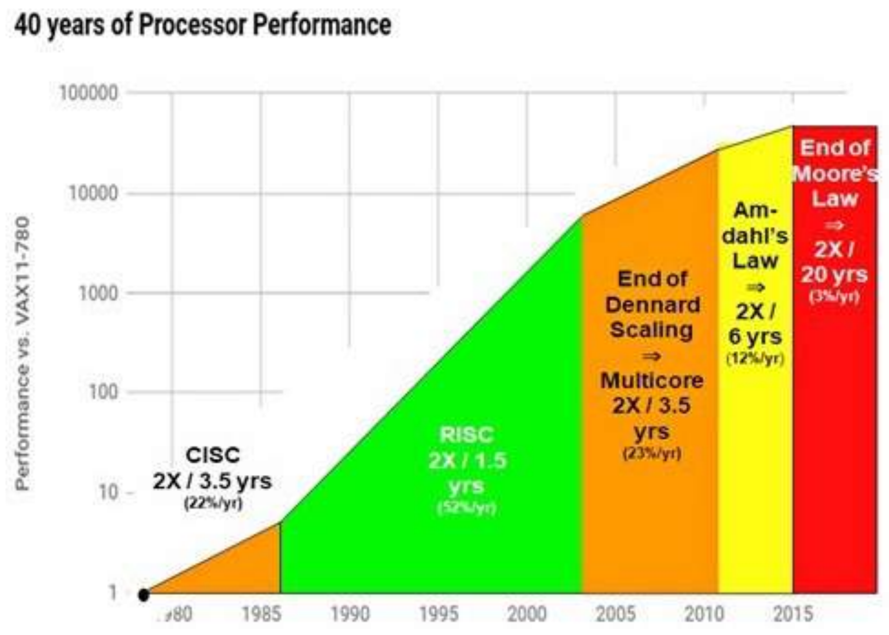
Startups Building Custom Hardware



https://www.servethehome.com/dell-emc-talks-deep-learning-and-ai-q3-2019/

Past General-Purpose[通用]

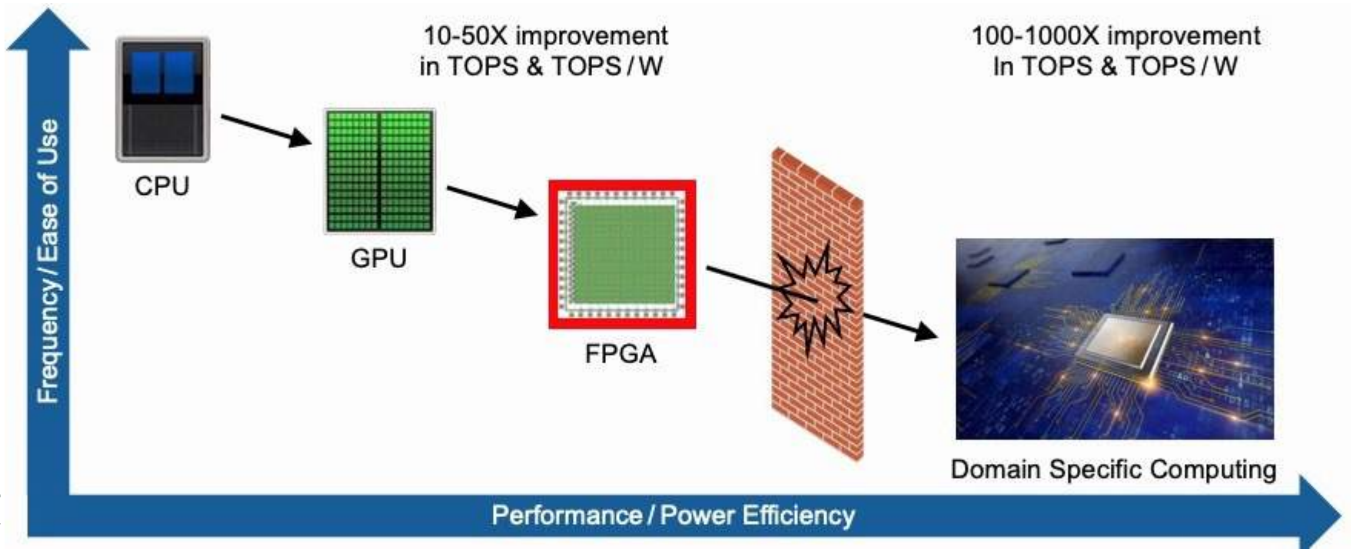
- Moore’s Law enabled:
  - Deep memory hierarchy
  - Wide SIMD units
  - Deep pipelines
  - Branch prediction
  - Out-of-order execution
  - Speculative prefetching
  - Multithreading
  - Multiprocessing



- The sophisticated architectures targeted general-purpose code
  - Architects treated code as black boxes
  - Extract performance from software that is oblivious to architecture

Domain-Specific Architecture[领域专用]

- Hard to keep improving performance
  - More transistors means more power
  - Energy budget is limited: higher performance → lower energy/operation
  - Enhancing existing cores may only boost 10% performance
- Need factor of 100 improvements in number of operations per instruction
  - Requires domain specific architectures



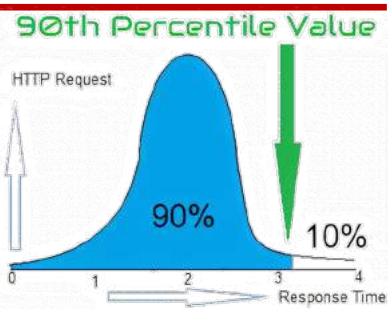
Domain-Specific Architecture (cont.)

- Computers will be much more heterogeneous[异构]
  - Standard processors to run conventional large programs
    - E.g., operating system
  - Domain-specific processors doing only a narrow range of tasks
    - But they do them extremely well
- DSA opportunities[机遇]
  - Preceding architecture from the past may not be a good match to some domains
    - E.g., caches are excellenet general-purpose architectures but not necessarily for DSAs
  - Domain-specific algorithms are almost always for small compute-intensive kernels of larger systems
    - DSAs should focus on the subset and not plan to run the entire program

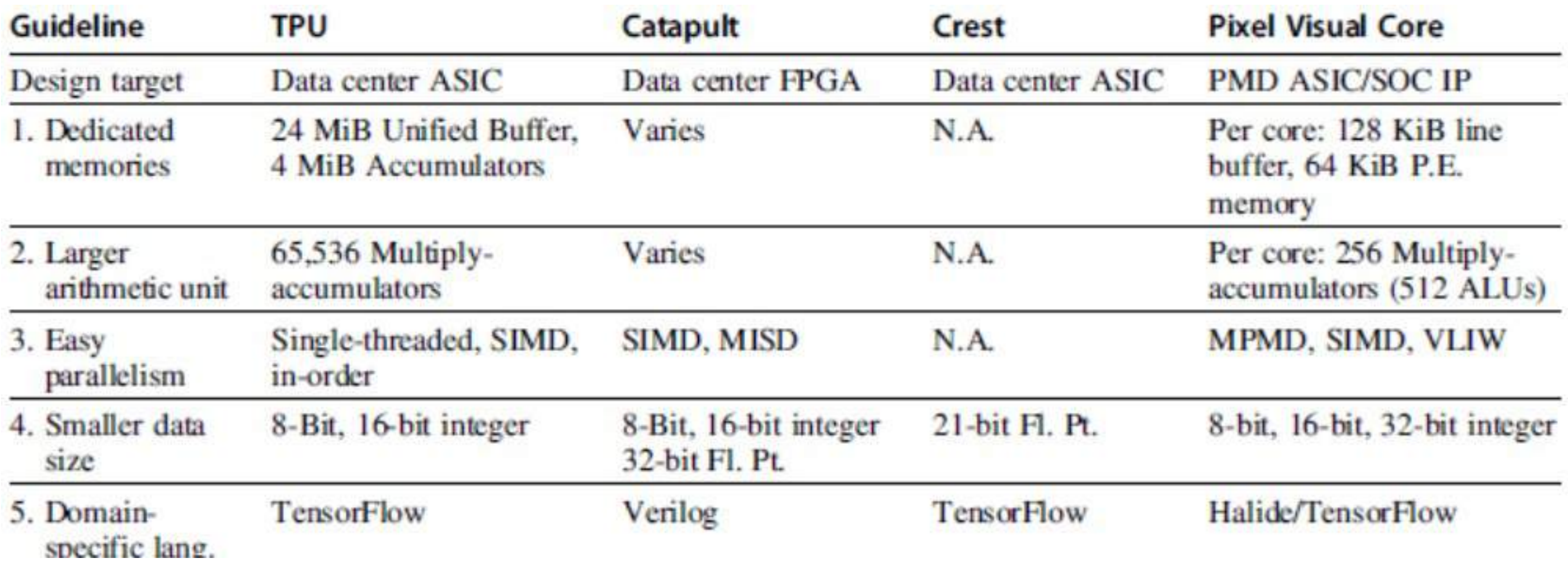
DSA Challenges[挑战]

- Architects must expand their areas of expertise
  - Must now learn application domains and algorithms
- *Nonrecurring engineering* (NRE) costs[一次性工程成本]
  - Find a target whose demand is large enough to justify allocating dedicated silicon on an SOC or even a custom chip
    - The costs are amortized over the number of chips manufactured, so unlikely to make economic sense if you need only 1000 chips
  - For smaller volume applications, use reconfigurable chips such as FPGAs
    - Several different applications may reuse the same reconfigurable hardware to amortize costs
    - However, the hardware is less efficient than custom chips, so the gains from FPGAs are more modest
- Port software[移植软件]
  - Programming languages and compilers

DSA Design Guidelines[设计准则]



- Why guidelines?
  - Lead to increased area and energy efficiency
  - Provide two valuable bonus effects
    - Lead to simpiler designs, reducing the cost of NRE of DSAs
    - For user-facing apps, better match the 99th-percentile response-time deadlines



Guideline	TPU	Catapult	Crest	Pixel Visual Core
Design target	Data center ASIC	Data center FPGA	Data center ASIC	PMD ASIC/SOC IP
1. Dedicated memories	24 MiB Unified Buffer, 4 MiB Accumulators	Varies	N.A.	Per core: 128 KiB line buffer, 64 KiB P.E. memory
2. Larger arithmetic unit	65,536 Multiply-accumulators	Varies	N.A.	Per core: 256 Multiply-accumulators (512 ALUs)
3. Easy parallelism	Single-threaded, SIMD, in-order	SIMD, MISD	N.A.	MPMD, SIMD, VLIW
4. Smaller data size	8-Bit, 16-bit integer	8-Bit, 16-bit integer 32-bit Fl. Pt.	21-bit Fl. Pt.	8-bit, 16-bit, 32-bit integer
5. Domain-specific lang.	TensorFlow	Verilog	TensorFlow	Halide/TensorFlow

DSA Design Guidelines (cont.)

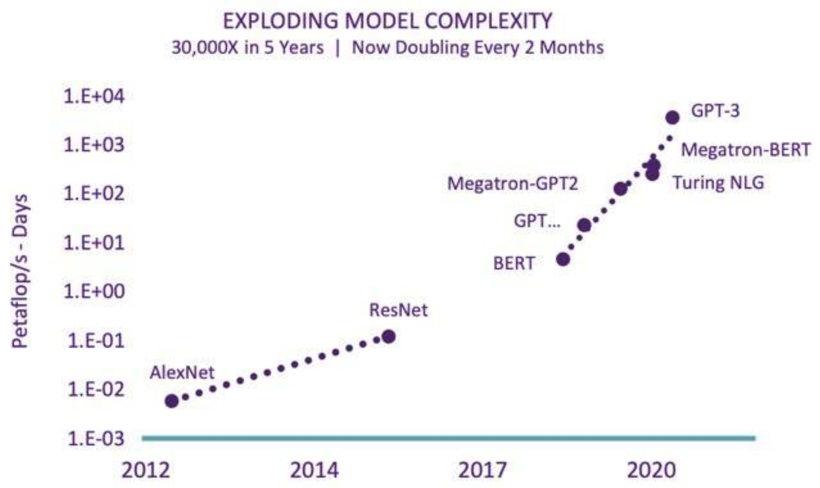
- *Use dedicated memories to minimize the distance over which data is moved*
  - Hardware cache → software-controlled scratchpad
    - Compiler writers and programmers of DSAs understand their domain
    - Software-controlled memories are much more energy efficient
- *Invest the resources saved from dropping advanced u-arch optimizations into more arithmetic units or bigger memories*
  - Owing to the superior understanding of the execution of programs
- *Use the easiest form of parallelism that matches the domain*
  - Target domains for DSAs almost always have inherent parallelism
    - How to utilize that parallelism and how to expose it to the software?
  - Design the DSA around the natural granularity of the parallelism and expose that parallelism simply in the programming model
    - SIMD > MIMD (i.e., DLP > TLP), VLIW > OoO

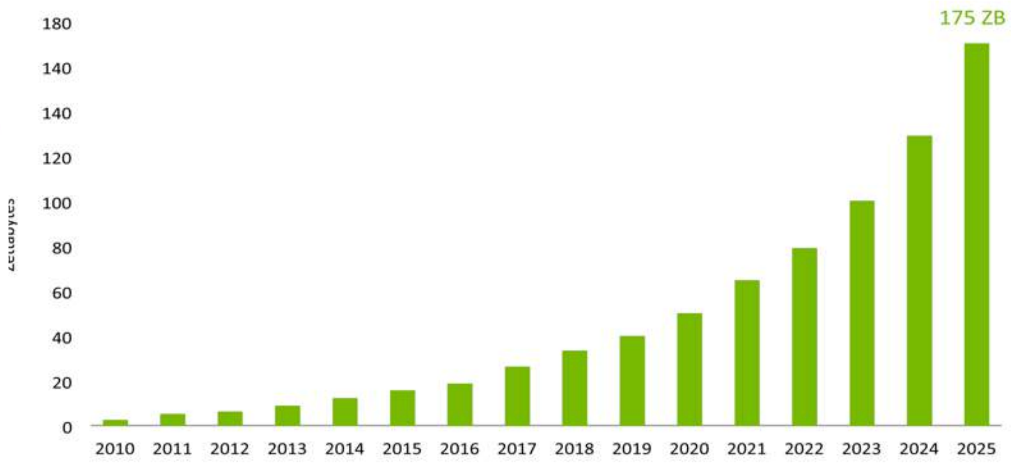
DSA Design Guidelines (cont.)

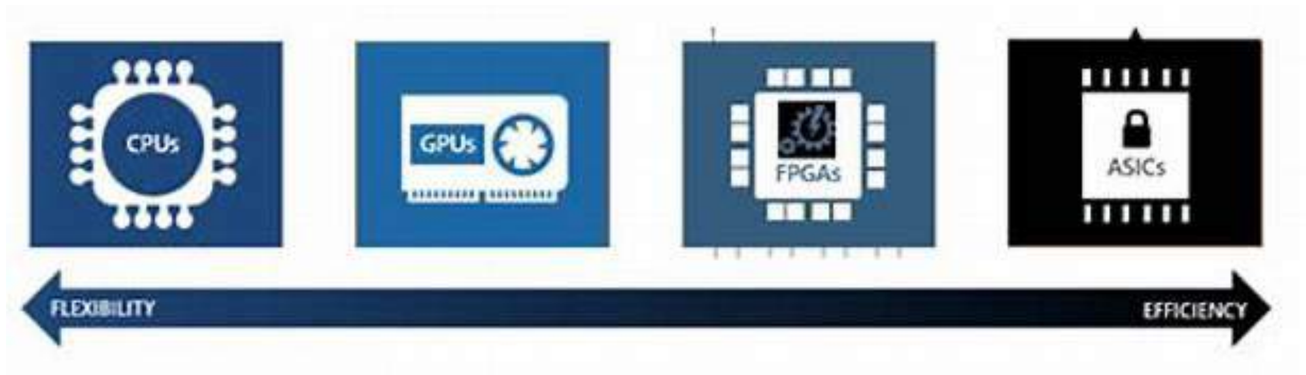
- *Reduce data size and type to the simplest needed for the domain*
  - Apps in many domains are typically memory-bound, using narrower data types helps increase the effective memory bandwidth and on-chip memory utilizations
  - Narrower and simpler data also enable to pack more arithmetic units into the same chip area
- *Use a domain-specific programming language to port code to the DSA*
  - WRONG: you new arch is so attractive that programmers will rewrite their code just for you hw
  - Fortunately, domain-specific languages were popular even before architects' switched attentions
    - Halide for vision processing, TensorFlow for DNNs

The Trend

- The ABC of AI: Algorithm + Big-data + Computing



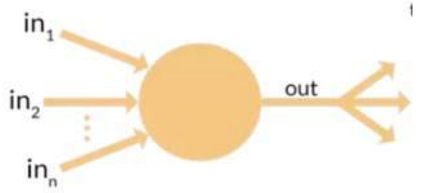




HPC
BIG DATA
AI

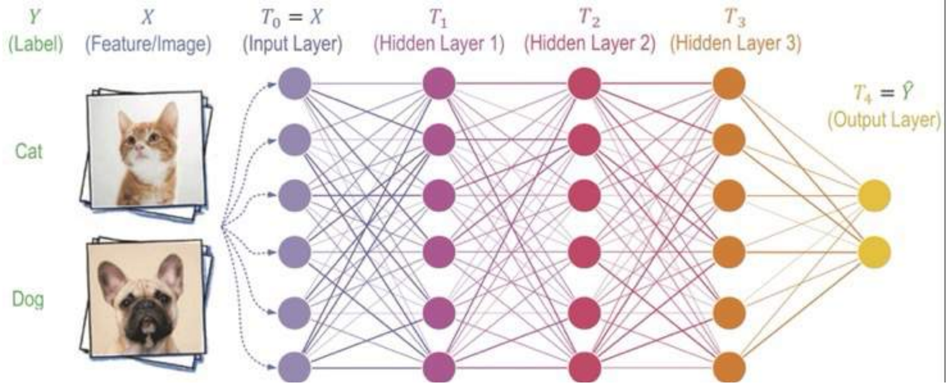
Example Domain

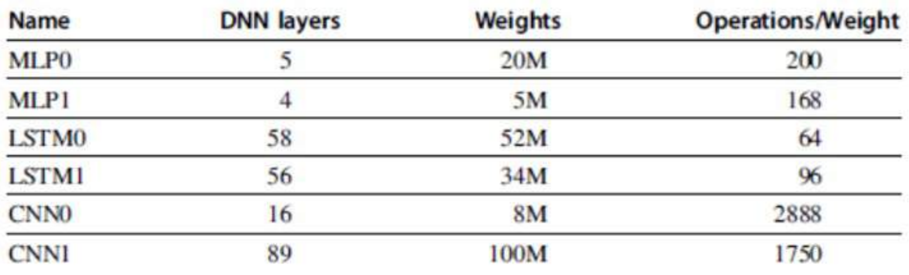
- Deep neural networks (DNNs)
  - Revolutioning many areas of computing today
  - Are applicable to a wide range of problems
    - So, a DNN-specific arch can be reused for solutions in speech, vision, language, translation, search ranking, and many more areas
- DNN structure
  - Inspired by neuron of the brain
    - Each neuron simply computes the sum over a set of products of weights or parameters and data values
      - E.g., pixels for image-processing
  - The sum is then put through a nonlinear function to determine its output
    - E.g., f(x) = max(x, 0) --- *rectified linear unit* (ReLU)
    - Output is called *activation*
      - The output of the neuron that has been "activated"



DNNs

- Most practitioners will choose an existing design
  - Topology
  - Data type
- Training (learning)[训练]
  - Calculate weights using backpropagation algorithm
  - Supervised learning: stochastic gradient descent[随机梯度下降]
- Inference[推理]
  - Use neural network for classification

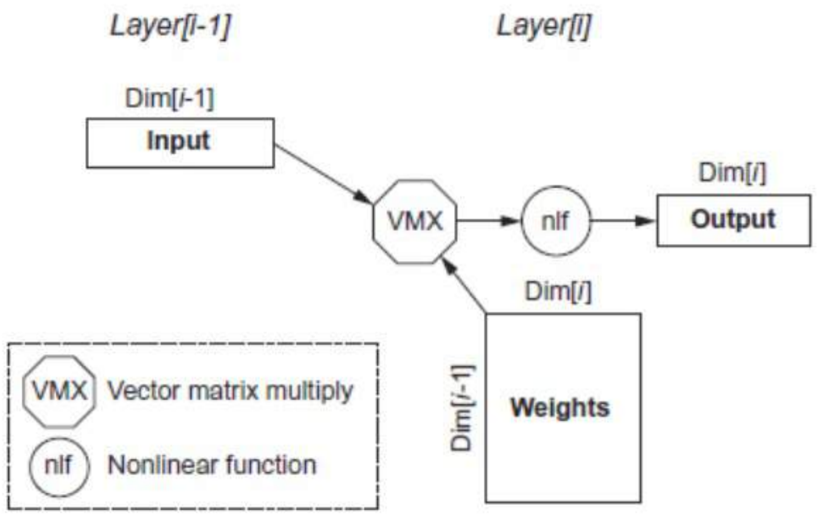




Name	DNN layers	Weights	Operations/Weight
MLP0	5	20M	200
MLP1	4	5M	168
LSTM0	58	52M	64
LSTM1	56	34M	96
CNN0	16	8M	2888
CNN1	89	100M	1750

Multilayer Perceptron[多层感知机]

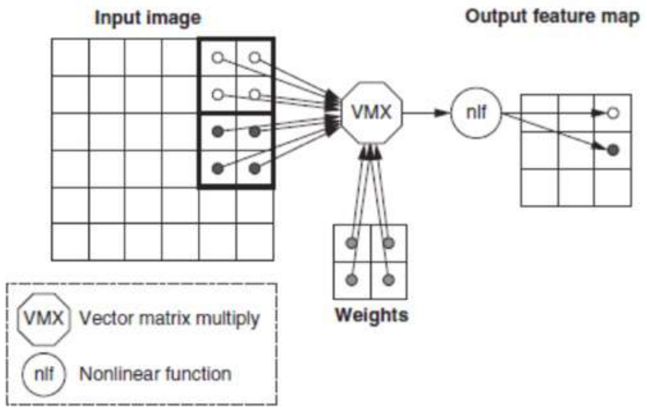
- Feed-forward neural networks
  - The units are arranged into a graph without any cycles
    - so that all the computation can be done sequentially
  - Fully connected: every unit in one layer is connected to every unit in the next layer
- MLP, the original DNNs, is just a vector matrix multiply of the input vector times the weights array



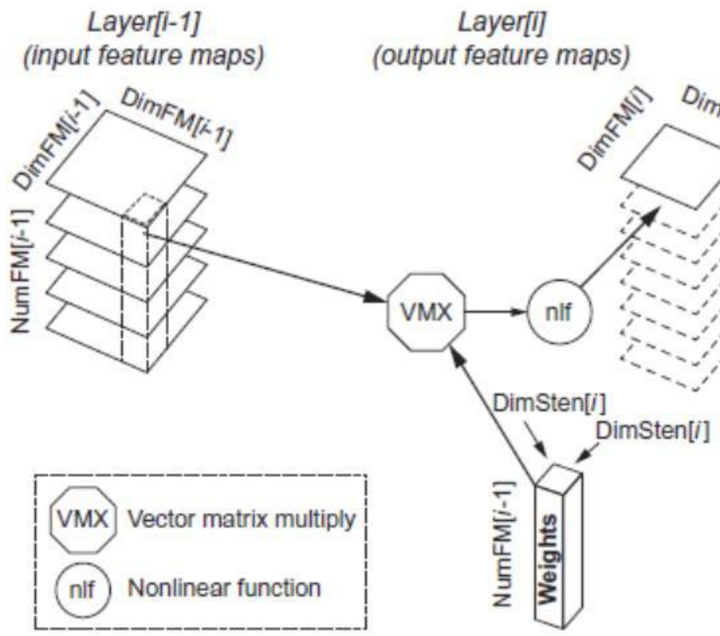
Parameters:
Dim[i]: number of neurons
Dim[i-1]: dimension of input vector
Number of weights: Dim[i-1] x Dim[i]
Operations: 2 x Dim[i-1] x Dim[i]
Operations/weight: 2

Convolutional Neural Network[卷积]

- CNNs are widely used for computer vision applications
- Each layer raises the level of abstraction
  - Lines → corners → shapes → …
- **Feature map**[特征图]: a set of 2D maps produced by each neural layer
  - Each cell is identifying one feature in the area of the input
- **Stencil computation**[模版计算]: uses neighboring cells in a fixed pattern to update all the elements of an array
  - 循环运算：遍历计算区域，每个位置均执行相同的计算操作



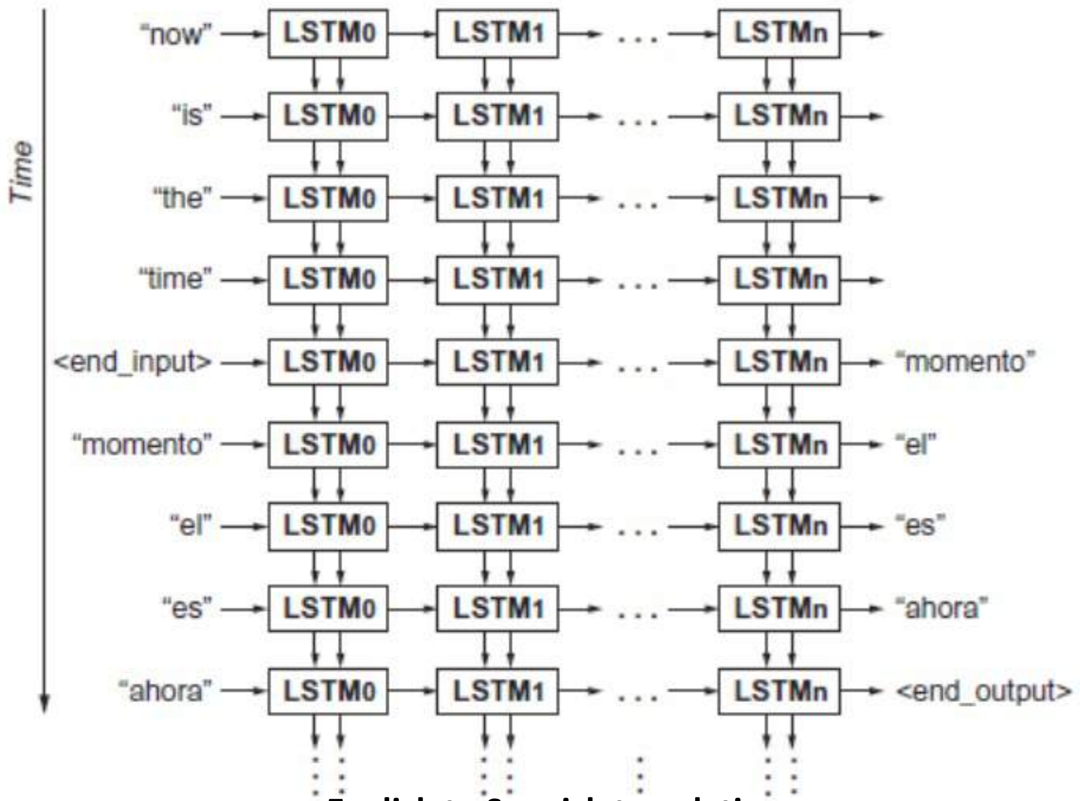
Convolutional Neural Network (cont.)



- Parameters:
  - DimFM[i-1]: Dimension of the (square) input Feature Map
  - DimFM[i]: Dimension of the (square) output Feature Map
  - DimSten[i]: Dimension of the (square) stencil
  - NumFM[i-1]: Number of input Feature Maps
  - NumFM[i]: Number of output Feature Maps
  - Number of neurons: NumFM[i] x DimFM[i]^2
  - Number of weights per output Feature Map: NumFM[i-1] x DimSten[i]^2
  - Total number of weights per layer: NumFM[i] x Number of weights per output Feature Map
  - Number of operations per output Feature Map: 2 x DimFM[i]^2 x Number of weights per output Feature Map
  - Total number of operations per layer: NumFM[i] x Number of operations per output Feature Map = 2 x DimFM[i]^2 x NumFM[i] x Number of weights per output Feature Map = 2 x DimFM[i]^2 x Total number of weights per layer
  - Operations/Weight: 2 x DimFM[i]^2

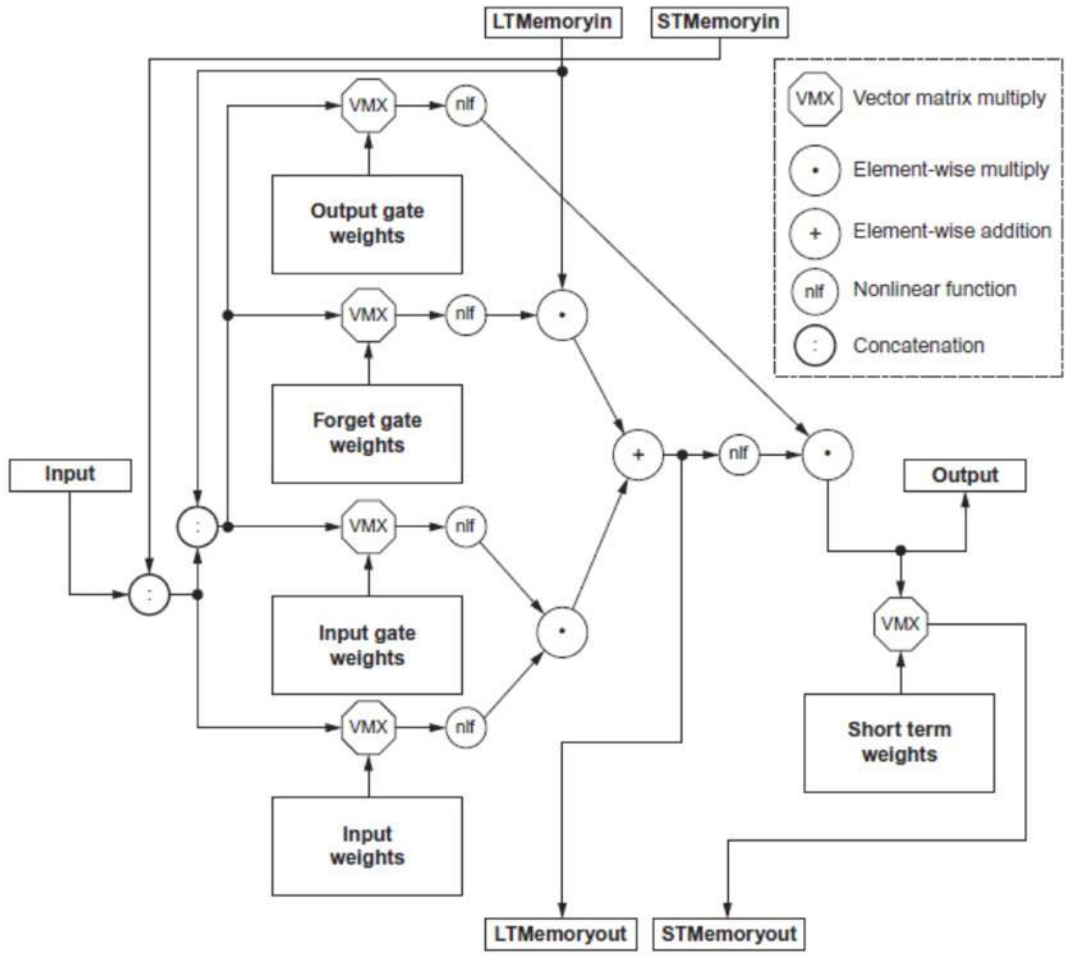
Recurrent Neural Network[循环]

- Popular for speech recognition on language translations
- RNNs can remember facts
  - Long short-term memory (LSTM) network



English to Spanish translation

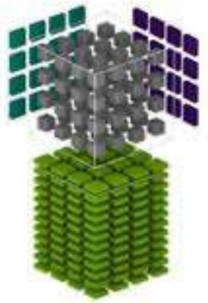
Recurrent Neural Network (cont.)

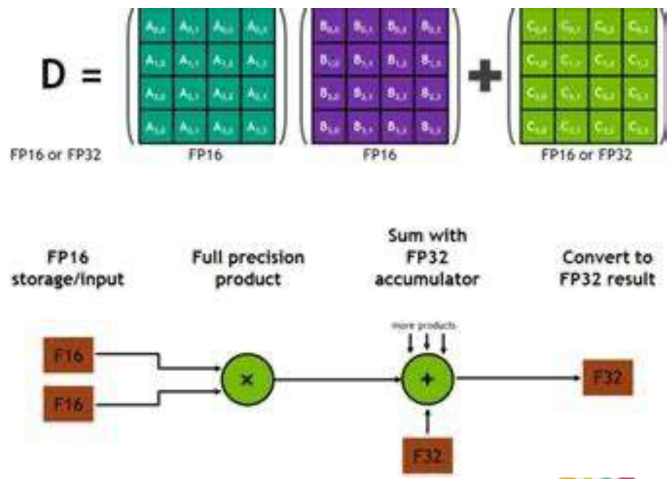


- Parameters:
  - Number of weights per cell: 3 x (3 x Dim x Dim)+(2 x Dim x Dim) + (1 x Dim x Dim) = 12 x Dim^2
  - Number of operations for the 5 vector-matrix multiplies per cell: 2 x Number of weights per cell = 24 x Dim^2
  - Number of operations for the 3 element-wise multiplies and 1 addition (vectors are all the size of the output): 4 x Dim
  - Total number of operations per cell (5 vector-matrix multiplies and the 4 element-wise operations): 24 x Dim^2 + 4 x Dim
  - Operations/Weight: ~2

Example Domain: DNNs

- Batches[批]
  - Reuse weights once fetched from memory across multiple inputs
    - Increases operational intensity
- Quantization[量化]
  - Numerical precision is less important for DNNs than for many applications
    - Use 8- or 16-bit fixed point
- Summary: need the following kernels
  - Matrix-vector multiply
  - Matrix-matrix multiply
  - Stencil
  - ReLU
  - Sigmoid
  - Hyperbolic tangent[双曲正切]





Tensor Processing Unit (TPU)

- Google's first custom ASIC DSA for WSCs
  - Its domain is the inference phase of DNNs
  - It is programmed using the TensorFlow framework
  - The first TPU was been deployed in 2015
    - Originated as far back as 2006, to improve perf by 10x over GPUs

TPU v1

Launched in 2015

Inference only

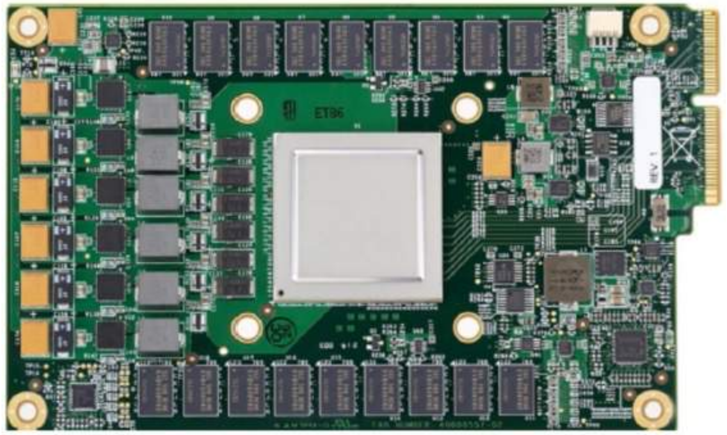
TPU v2

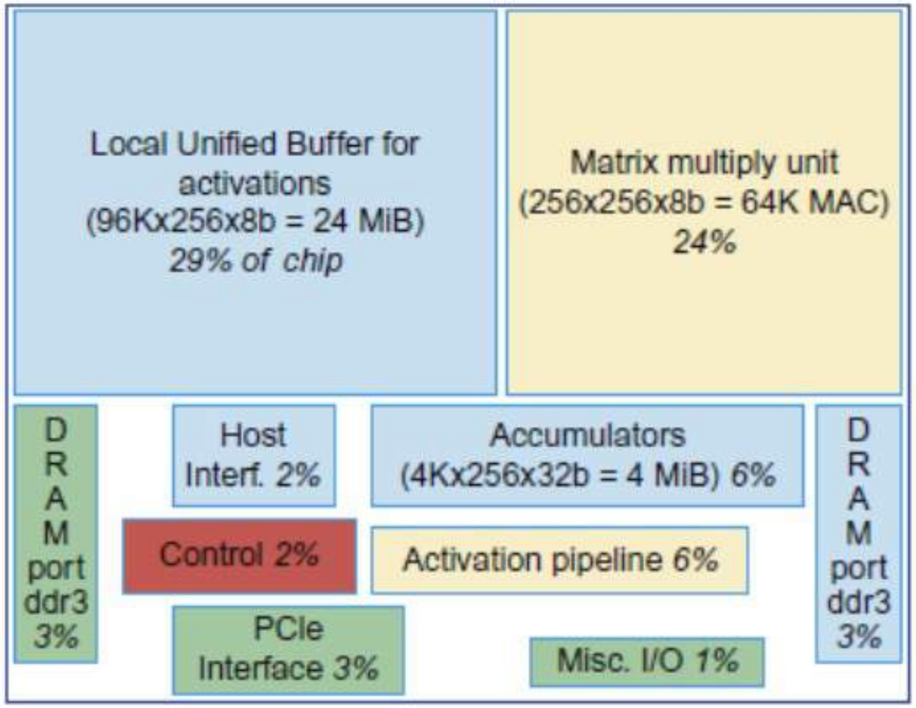
Launched in 2017

Inference and training

TPU Chip Overview

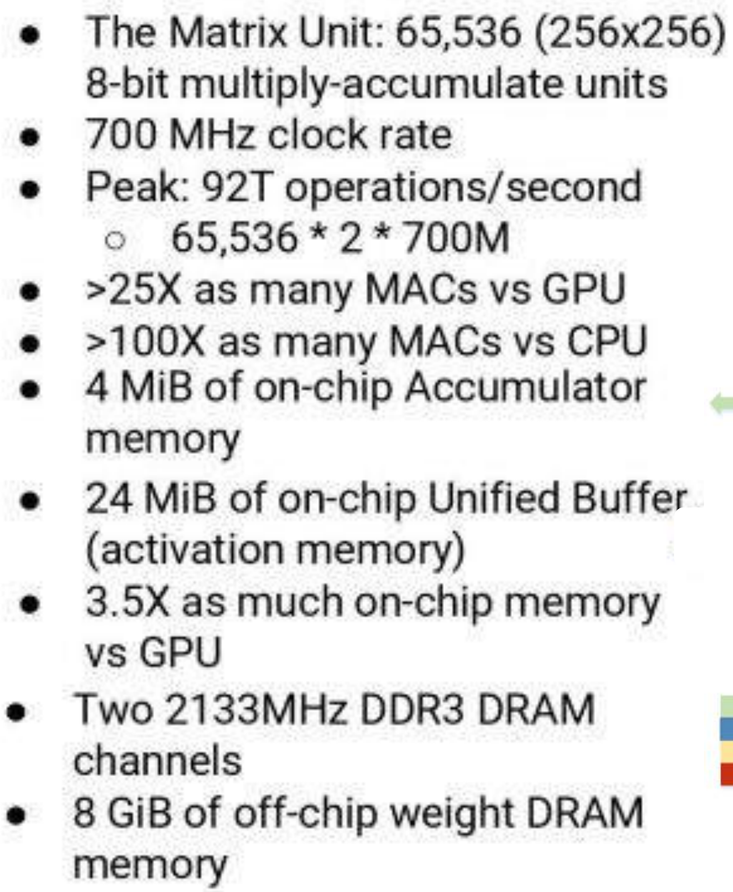
- TPU chip is half the size of the other chips
  - 28 nm process with a die size ≤ 331 mm^2
  - This is partially due to simplification of control logic
- Floor plan of TPU die
  - 50%+ on arithmetic and memory



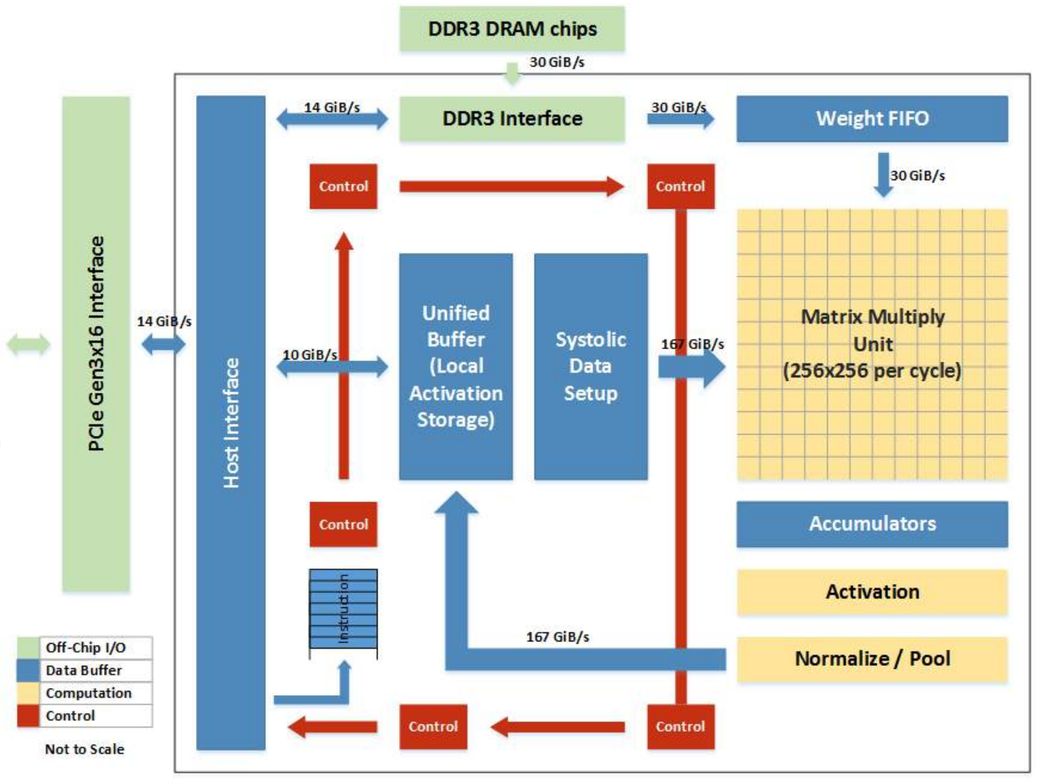

Local Unified Buffer for activations (96Kx256x8b = 24 MiB) *29% of chip*
Matrix multiply unit (256x256x8b = 64K MAC) *24%*
DRAM port ddr3 *3%*
Host Interf. *2%*
Accumulators (4Kx256x32b = 4 MiB) *6%*
DRAM port ddr3 *3%*
Control *2%*
Activation pipeline *6%*
PCIe Interface *3%*
Misc. I/O *1%*

TPU Architecture[架构]

- A coprocessor on the PCIe I/O bus
- A large software-managed on-chip memory



- The Matrix Unit: 65,536 (256x256) 8-bit multiply-accumulate units
- 700 MHz clock rate
- Peak: 92T operations/second
  - 65,536 * 2 * 700M
- >25X as many MACs vs GPU
- >100X as many MACs vs CPU
- 4 MiB of on-chip Accumulator memory
- 24 MiB of on-chip Unified Buffer (activation memory)
- 3.5X as much on-chip memory vs GPU
- Two 2133MHz DDR3 DRAM channels
- 8 GiB of off-chip weight DRAM memory

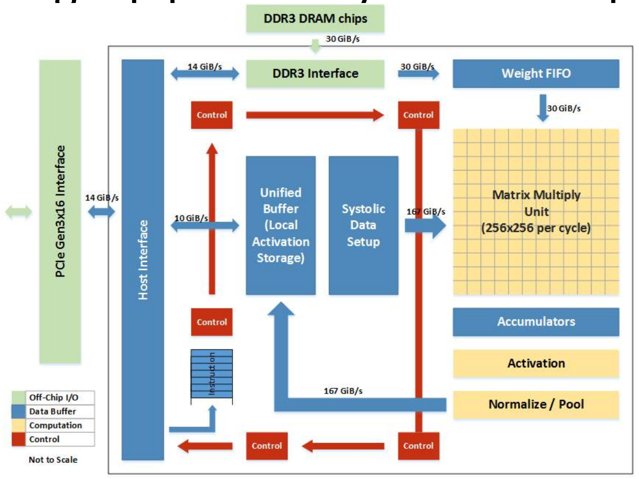


TPU ISA[指令]

- The host CPU sends TPU instructions over the PCIe bus into an instruction buffer[指令发送]
  - TPU has no PC, and it has no branch instructions
  - 5 main (CISC) instructions (11 in total)
    - Other 6: alternate host memory read/write, set configuration, two versions of synchronization, interrupt host, debug-tag, nop and halt
- Instruction execution[指令执行]
  - Average clock cycles per instruction: > 10
  - 4-stage overlapped execution, 1 instruction type/stage
    - Execute other instructions while matrix multiplier busy
- Complexity in software[软件复杂性]
  - No branches, in-order issue
  - SW controlled buffers, SW controlled pipeline synchronization

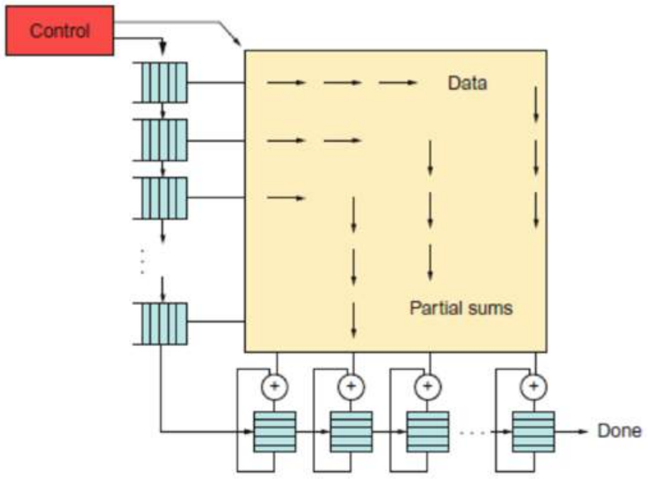
TPU ISA (cont.)

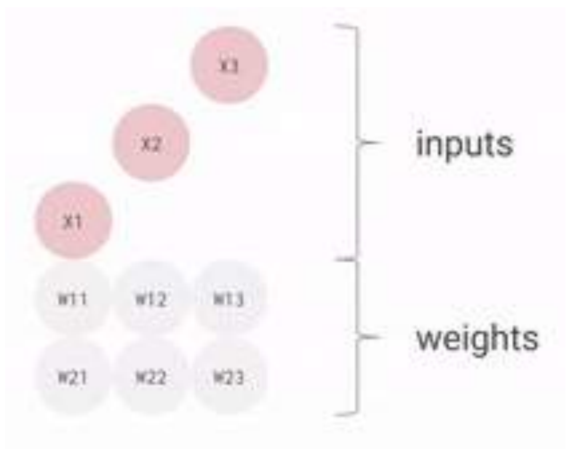
- **Read_Host_Memory**
  - Reads data from the CPU memory into the unified buffer
- **Read_Weights**
  - Reads weights from the Weight Memory into the Weight FIFO as input to the Matrix Unit
- **MatrixMultiply/Convolve**
  - Perform a matrix-matrix multiply, a vector-matrix multiply, an element-wise matrix multiply, an element-wise vector multiply, or a convolution from the Unified Buffer into the accumulators
    - Takes a variable-sized B*256 input, multiplies it by a 256x256 constant input, and produces a B*256 output, taking B pipelined cycles to complete
- **Activate**
  - Computes activation function
- **Write_Host_Memory**
  - Writes data from unified buffer into host memory

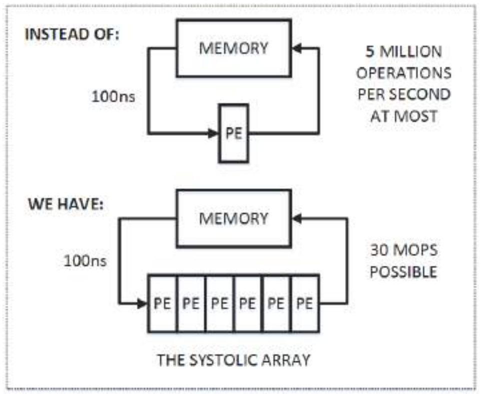


TPU Microarchitecture[微架构]

- The *u*-arch philosophy of TPU is to keep the Matrix Multiply Unit busy
  - Hide the execution of the other insts by overlapping with the MatrixMultiply inst
    - Each of the other 4 insts have separate execution hw
- 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[脉动阵列执行]



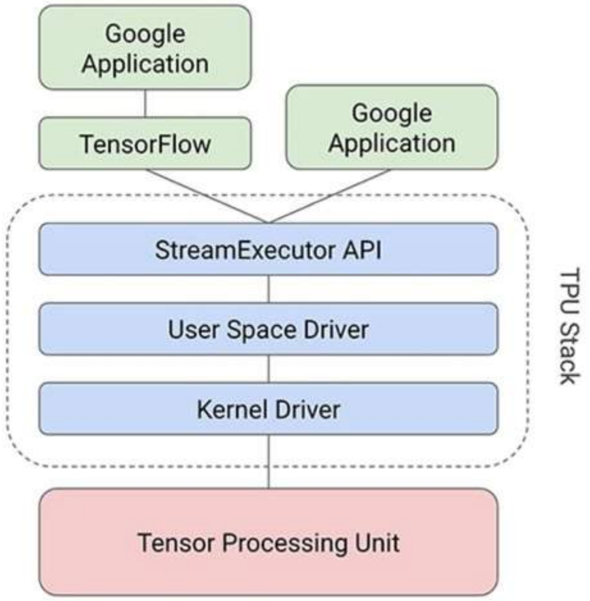




脉动阵列 - 因Google TPU获得新生，
https://zhuanlan.zhihu.com/p/26522315

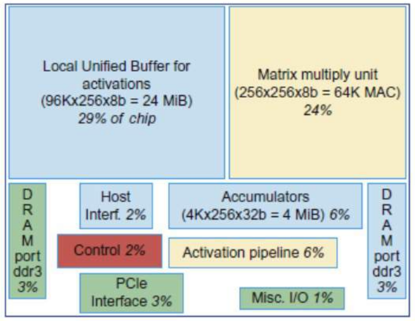
TPU Software[软件]

- Software stack had to be compatible with CPUs/GPUs[兼容]
  - So that applications could be ported quickly
  - The portion of the app run on the TPU is typically written using TensorFlow and is compiled into an API that can run on CPUs/GPUs
- Like GPUs, the TPU stack is split into[分层]
  - **Kernel Driver**: lightweight and handles only memory management and interrupts
    - Designed for long-term stability
  - **Use Space Driver**: changes frequently, and handles the following
    - Sets up and controls TPU execution
    - Reformats data into TPU order
    - Translates API calls into TPU insts and turns them into an app binary



How TPU Follows the Guidelines

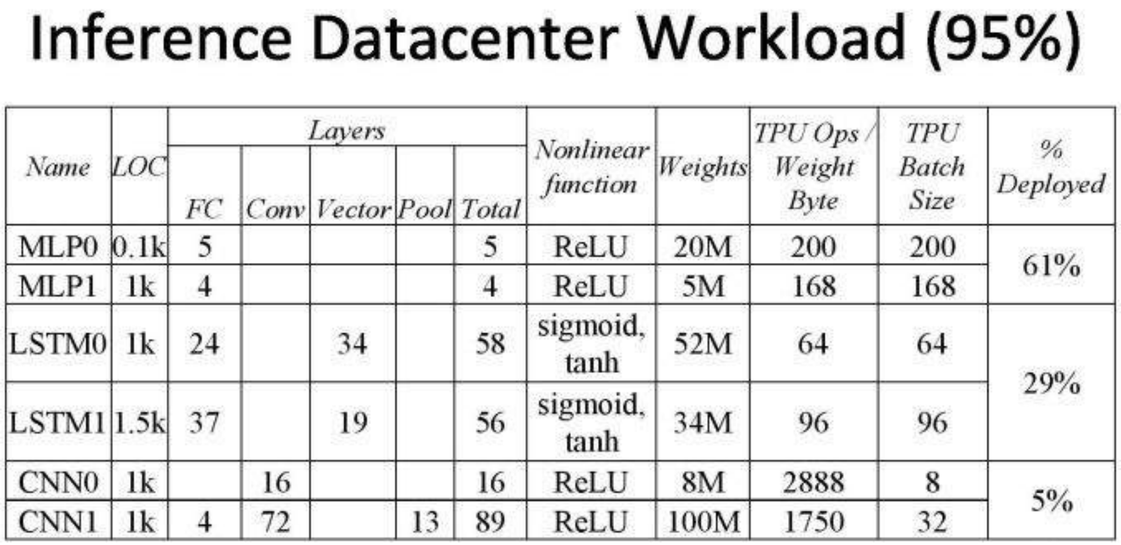
- *Use dedicated memories*
  - 24 MB dedicated buffer, 4 MB accumulator buffers
- *Invest resources in arithmetic units and dedicated memories*
  - 60% of the memory and 250X the arithmetic units of a server-class CPU
- *Use the easiest form of parallelism that matches the domain*
  - Exploits 2D SIMD parallelism
- *Reduce the data size and type needed for the domain*
  - Primarily uses 8-bit integers
- *Use a domain-specific programming language*
  - Uses TensorFlow


Local Unified Buffer for activations (96Kx256x8b = 24 MiB) 29% of chip
Matrix multiply unit (256x256x8b = 64K MAC) 24%
DRAM port ddr3 3%
Host Interf. 2%
Accumulators (4Kx256x32b = 4 MiB) 6%
DRAM port ddr3 3%
Control 2%
Activation pipeline 6%
PCIe Interface 3%
Misc. I/O 1%

SUN YAT-SEN UNIVERSITY

TPU Performance[性能]

- Compare using six benchmarks
  - Representing 95% of TPU inference workload in Google data center in 2016
  - Typically written in TensorFlow, pretty short (100-1500 LOCs)
- Chips/servers being compared
  - CPU server: Intel 18-core, dual-socket Haswell; host server for GPUs/TPUs
  - GPU accelerator: Nvidia K80



Inference Datacenter Workload (95%)

Name	LOC	Layers					Nonlinear function	Weights	TPU Ops / Weight Byte	TPU Batch Size	% Deployed
		FC	Conv	Vector	Pool	Total					
MLP0	0.1k	5				5	ReLU	20M	200	200	61%
MLP1	1k	4				4	ReLU	5M	168	168	
LSTM0	1k	24		34		58	sigmoid, tanh	52M	64	64	29%
LSTM1	1.5k	37		19		56	sigmoid, tanh	34M	96	96	
CNN0	1k		16			16	ReLU	8M	2888	8	5%
CNN1	1k	4	72		13	89	ReLU	100M	1750	32	