

# Learning on GPUs *and* ML Accelerators

CS4787/5777 Lecture 26 — Fall 2022

# Recap: ML on Hardware

- Over the past three lectures, we've been talking about the architecture of the CPU and how it affects performance of machine learning models.
- However, the CPU is not the only type of hardware that machine learning models are trained or run on.
- In fact, most modern DNN training happens not on CPUs but on GPUs.
- In this lecture, we'll look at **why GPUs became dominant for machine learning training**, and we'll explore what makes their architecture uniquely well-suited to large-scale numerical computation.

# Modern ML Hardware

- CPUs
- GPUs

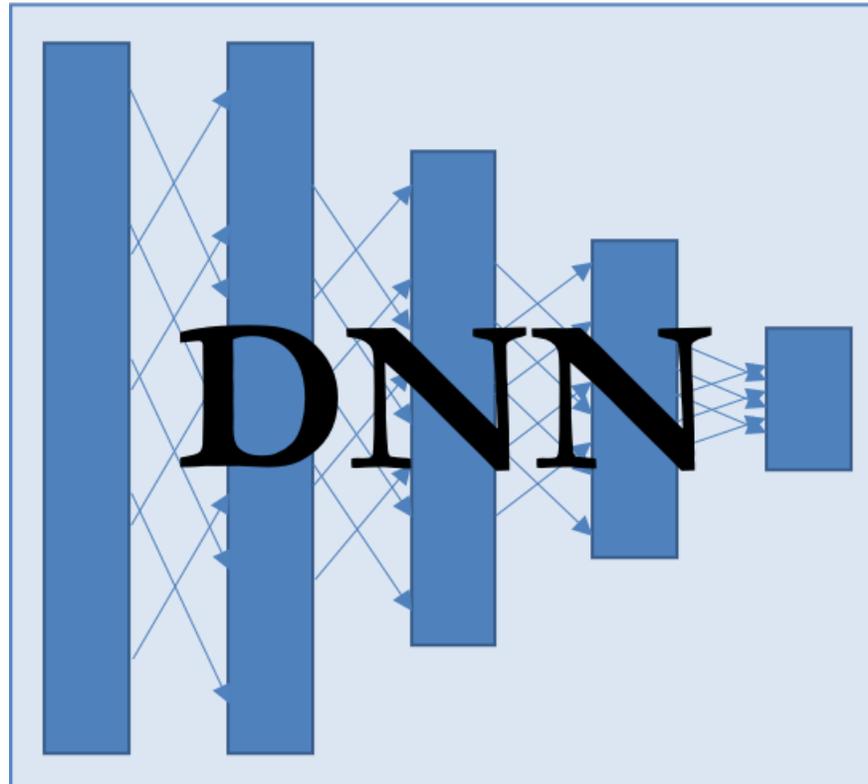


- FPGAs
- Specialized accelerators

Right now, **GPUs** are dominant for ML training...we'll get to why later

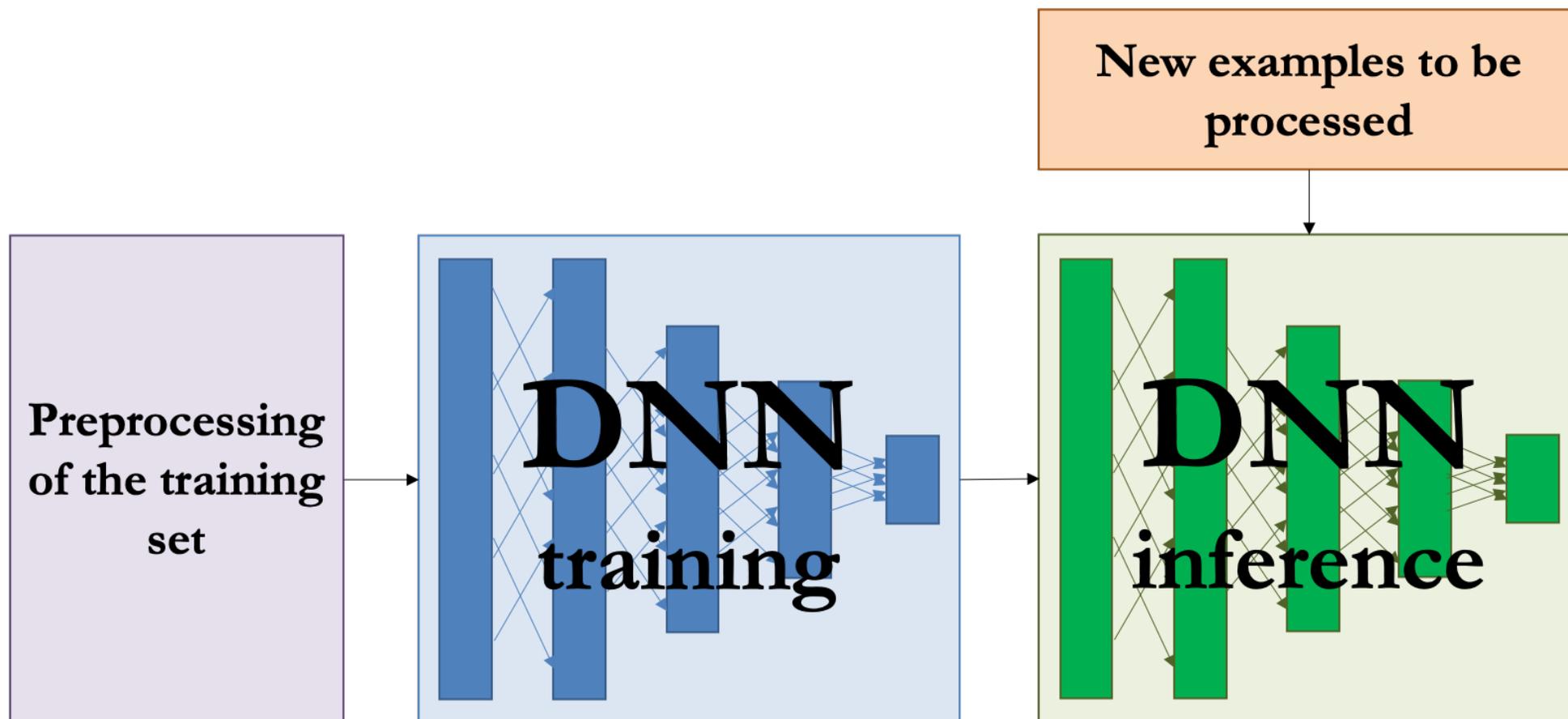
# What does the modern ML pipeline look like?

- It's not just training a neural network.



# What does the modern ML pipeline look like?

- The pipeline has many different components



**Where can hardware help?**

# Everywhere!

There's interest in hardware everywhere in the pipeline

- both adapting existing hardware architectures, and
- developing new ones

Active Learning Q: **What improvements can we get?**

- Lower latency inference
  - To be able to make real-time predictions
- Higher throughput training
  - To train on larger datasets to produce more accurate models
- Lower power cost
  - Especially important as data and model sizes scale

# How can hardware help?

## **Speed up the basic building blocks of machine learning**

- Major building block: matrix-matrix multiply
- Other major building blocks: convolution, attention

## **Add data/memory paths specialized to machine learning computations and workloads**

- Example: having a local cache to store network weights
- Create application-specific functional units
- Not for general ML, but for a specific domain

**Why are GPUs so popular for machine learning?**

# Why are GPUs so popular for training deep neural networks?

To answer this...we need to look at GPU architectures.

# A brief history of GPU computing

- GPUs were originally designed to support the 3D graphics pipeline, driven by demand for videogames with ever-increasing graphical fidelity.

# Important properties of 3d graphics rendering:

- Lots of opportunities for parallelism
  - rendering different pixels/objects in the scene can be done simultaneously
- Lots of **numerical computations**
  - 3d rendering is based on geometry
  - Mostly polygons in space which are transformed, animated, and shaded
  - All of this is **numerical computation: mostly linear algebra**

# GPU computing history (continued)

- The first era of GPUs ran a **fixed-function graphics pipeline**. They weren't programmed, but instead just configured to use a set of fixed functions designed for specific graphics tasks: mostly drawing and shading polygons in 3d space.
  - Even these GPUs were a little programmable via OpenGL calls, but you couldn't *program* them...in the sense of writing an ordinary imperative program that would run there

# Programmable GPUs

In the early 2000s, there was a shift towards **programmable GPUs**.

- Programmable GPUs allowed customization certain stages in the graphics pipeline by writing small programs called **shaders** which let developers process the vertices and pixels of the polygons they wanted to render in custom ways.
- And there was powerful economic incentive for hardware designers to increase the throughput of these GPU **shaders** so that game designers could render models with more polygons and higher resolution.

# What did shaders need from hardware?

- These shaders needed to be **capable of very high-throughput parallel processing**
  - So that they could process and render very large numbers of polygons in a single frame of a 3d animation.
  - Intuition: if I am rendering multiple independent objects in a scene, I can process them in parallel.
- The shaders also needed **high throughput to high latency texture memory**
  - Texture memory accesses are essentially random
  - Needed lots of parallel contexts to be able to hide that latency

# GPU Parallelism

- The GPU supported parallel programs that were more parallel than those of the CPU.
- Unlike multicore CPUs, which supported computing different functions at the same time, the GPU computes the **same function on multiple elements of data**.
  - Example application: want to render a large number of triangles, each of which is lit by the same light sources.
  - Example application: want to transform all the objects in the scene based on the motion of the player. That is, the GPU could run the same function on a bunch of triangles in parallel, but couldn't easily compute a different function for each triangle.

# GPU Memory

- The GPU needed high bandwidth access to large texture memory to draw textures on models in real time.
  - This memory has **high latency**
- Unlike multicore CPUs, which have deep cache hierarchies, the GPU used parallelism to **hide latency**
  - Example application: when a shader reaches a lookup into texture memory, the GPU “core” will request the data; while the request is resolving, it runs other threads.
  - But actually switching to an independent program would be too expensive, so instead it does same program/different data

# GPU Computing and Types of Parallelism

Two types of parallelism:

- **Data parallelism** involves the same operations being computed in parallel on many different data elements.
- **Task parallelism** involves different operations being computed in parallel
  - on either the same data or different data

Active Learning Question: **How are the types of parallelism we've discussed categorized according to this distinction?**

- SIMD/Vector parallelism?
- Multi-core/multi-thread parallelism?
- Distributed computing?

# The general-purpose GPU

- Eventually, people started to use GPUs for tasks other than graphics rendering.
- However, **working within the structure of the graphics pipeline of the GPU placed limits on this.**
  - OpenGL is great for graphics, but annoying for ML
- To better support general-purpose GPU programming, in 2007 NVIDIA released **CUDA**, a parallel programming language/computing platform for general-purpose computation on the GPU.
  - Other companies (e.g. Intel/AMD) have competing products as well.
- Now, programmers no longer needed to use the 3d graphics API to access the parallel computing capabilities of the GPU.

# A Revolution in GPU Computing

Several new applications:

- Training deep neural networks
- Cryptocurrencies

All based on running small highly parallel programs called “**kernels**” on the GPU.

- Not to be confused with the kernels of kernel learning!

# An Illustration of CUDA

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

This syntax launches N threads, each of which performs a single addition.

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

# It's straightforward to implement other ops

```
// Kernel definition
__global__ void VecMul(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] * B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecMul<<<1, N>>>(A, B, C);
    ...
}
```

# The general-purpose GPU: upsides

- Importantly, spinning up hundreds or thousands of threads like this could be reasonably fast on a GPU, while on a CPU this would be **way** too slow due to the overhead of creating new threads.
  - Because GPU threads run the same program, so they share resources for their program context
- Additionally, GPUs support many many more parallel threads running than a CPU.
  - Typical thread count for a GPU: tens of thousands.
  - Typical thread count for a CPU: at most a few dozen.

# The general-purpose GPU: downsides

- Important downside of GPU threads of this type: they're data-parallel only.
  - You couldn't direct each of the individual threads to "do its own thing" or run its own independent computation
- Importantly: more **modern GPUs now support some level of task-parallelism.**
  - Threads are bound together into "warps" (of ~32 threads) that all execute the same instruction.
  - But threads in different warps have limited ability to run independently.

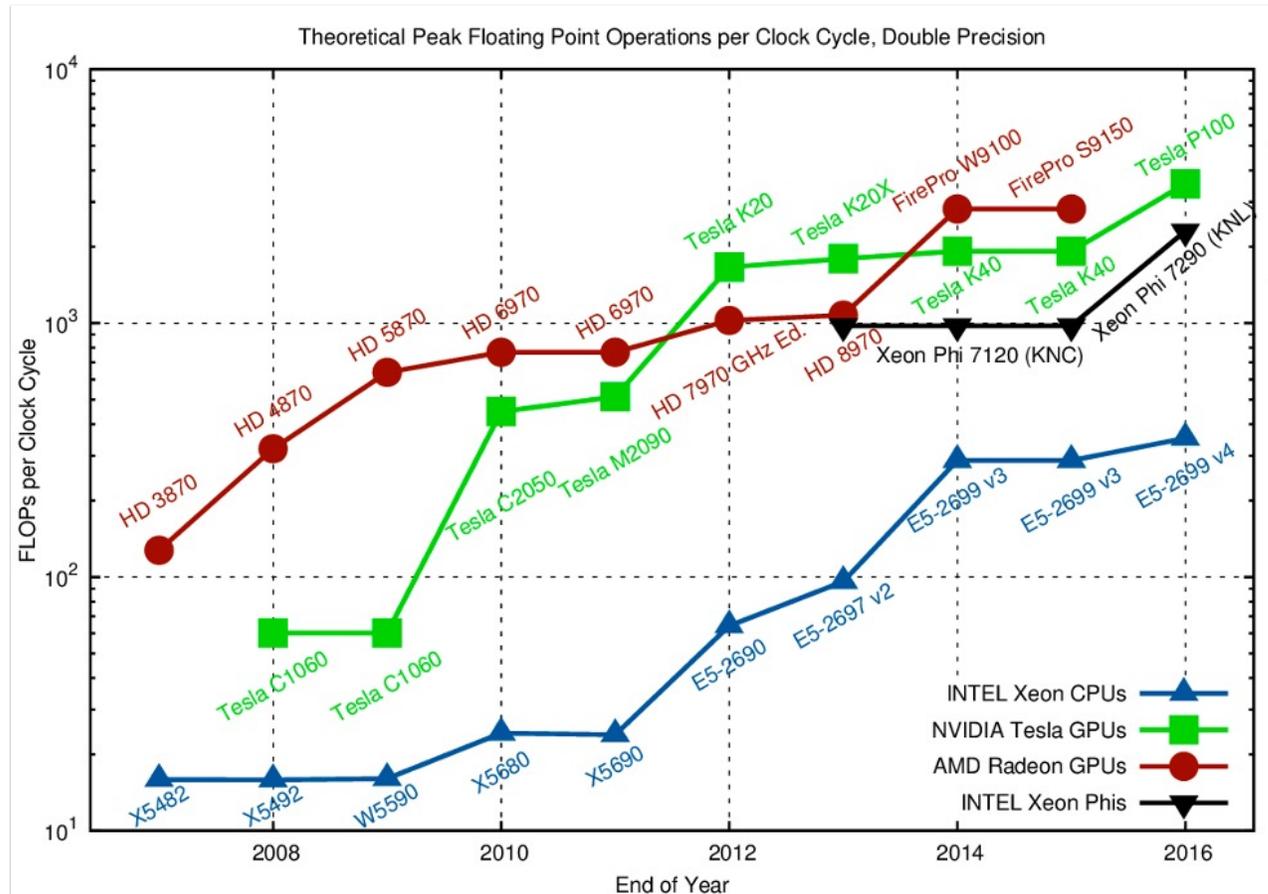
# Summary: GPU vs CPU

- CPU is a general purpose processor
  - Modern CPUs spend most of their area on deep caches
  - This makes the CPU a great choice for applications with random or non-uniform memory accesses
- GPU is optimized for
  - more compute intensive workloads
  - streaming memory models

**Machine learning workloads are compute intensive and often support streaming data flows.**

# FLOPS: GPU vs CPU

- FLOPS: floating point operations per second



**GPU FLOPS consistently exceed CPU FLOPS**

From Karl Rupp's blog  
<https://www.karlrupp.net/2016/08/flops-per-cycle-for-cpus-gpus-and-xeon-phis/>

This was the best diagram I could find that shows trends over time.

# Memory Bandwidth: GPU vs CPU

- GPUs have higher memory bandwidths than CPUs
  - NVIDIA A100 80GB GPU has 2 TB/s memory bandwidth
  - Whereas an Intel Xeon Platinum CPU has only about 120 GB/s memory bandwidth at about the same time
- **But, this comparison is a bit unfair!**
  - GPU memory bandwidth is the **bandwidth to GPU memory**
  - e.g. on a PCIe2, bandwidth to system memory was only 32 GB/s for a GPU

# What limits deep learning?

- Active learning question: **Is deep learning more compute bound or more memory bound?**
  - i.e. is it more limited by the computational capabilities or memory capabilities of the hardware?

# What limits deep learning?

- **Ideally: for MLPs it's compute bound**

- Why? Matrix-matrix multiply over  $n \times n$  matrices takes  $O(n^2)$  memory but  $O(n^3)$  compute
- So as we scale wider, costs will be dominated by that compute

- **But often it is memory/communication bound**

- When we are doing convolutions with small filter sizes, compute becomes less dominant
- Especially when we have to load a lot of data
- Especially when we are running at large scale on a cluster
- Especially for transformers
- Especially when data/models do not fit in caches on the hardware

# GPUs in machine learning

- Because of their high data parallelism, GPUs provide **an ideal substrate for large-scale numerical computation.**
  - In particular, GPUs can perform matrix multiplies very fast.
  - Just like BLAS on the CPU, there's an optimized library from NVIDIA "cuBLAS" that does matrix multiples efficiently on their GPUs.
  - There's even a specialized library of primitives designed for deep learning: cuDNN.
- Machine learning frameworks, such as TensorFlow and PyTorch, are designed to support computation on GPUs.
  - And training a deep net on a GPU can decrease training time by an order of magnitude.

**How do ML frameworks support GPU computing?**

# Machine Learning Frameworks and GPUs

A core feature of ML frameworks is GPU support. Approach:

- Represent the compute graph as linear-algebra operations.
- For each operation, call a hand-optimized kernel that computes that op on the GPU.
  - i.e. matrix-matrix multiply
  - i.e. convolution for a CNN
- When GPU code is not available for an op, run it on the CPU
  - but this happens rarely, only with uncommon architectures
- Stitch those ops together by calling them from Python
- and eventually make the results available back on the CPU

**So should we always use GPUs?**

**Will we always use GPUs?**

Maybe not!

# Challengers to the GPU

- **More compute-intensive CPUs**
  - Like Intel's Phi line — promise same level of compute performance and better handling of sparsity
- **Low-power devices**
  - Like mobile-device-targeted chips
- **Configurable hardware like FPGAs and CGRAs**
- **Accelerators that speed up matrix-matrix multiply**
  - Like Google's Version 1 TPU
- **Accelerators special-designed to accelerate ML computations beyond just matrix-matrix multiply**
  - Version 2/3 TPUs
- Many competitors in this space

# ML Accelerators

# Effect of Hardware on Statistical Performance

- Sometimes, when we change device, we expect the same results: same learned model, same accuracy. **But this is not always the case. Why?**
- One reason: FP arithmetic is not associative, so reordering to get better performance on different hardware can change the results slightly!
  - This can even be the case for multiple runs on the same hardware.
- Another reason: randomness.
  - Different hardware platforms could use different pseudo-random generators or use random numbers in a different order.
- Another reason: hardware can do approximation to trade-off precision for hardware efficiency.
  - E.g. if the hardware runs on 16-bit floats, it can give different results.

# Effect of Hardware on Statistical Performance

- Sometimes, when we change device, we expect the same results: same learned model, same accuracy. **But this is not always the case. Why?**
- **Generally, we expect specialized ML hardware to produce learned models that are similar in quality, but not necessarily exactly the same, as baseline chips (CPU/GPU).**

# Programming ML Hardware

- One major issue when developing new hardware for ML, and the question we should always be asking: **how is the device programmed?**
- To be useful for DNN applications, we should be able to map a high-level program written in something like TensorFlow / PyTorch / MxNet / Keras to run on the hardware.
  - Ideal scenario: users don't need to reason about the hardware that their code is running on. They just see the speedup.
  - Currently: users still need to give *some* thought to the hardware.
  - E.g. if the accelerator uses low-precision arithmetic, the performance of an algorithm in full precision on a CPU may differ from the performance of the algorithm in low-precision on the accelerator.
  - But it is possible to get TensorFlow/PyTorch code to run on an accelerator with minimal modifications!

# GPU as ML accelerator

- The only real distinction between GPUs and ML accelerators is that GPUs weren't originally designed for AI/ML.
  - But there's nothing in the architecture that separates GPUs from all purpose-built ML accelerators.
- ...although individual accelerators do usually have features that GPUs lack.
- As ML tasks capture more of the market for GPUs, GPU designers have been adjusting their architectures to fit ML applications.
  - For example, by supporting low-precision arithmetic.
- As GPU architectures become more specialized to AI tasks, it becomes more accurate to think of them as ML accelerators.

# FPGA as ML accelerator

- All computer processors are basically integrated circuit: electronic circuits etched on a single piece of silicon.
- Usually this circuit is fixed when the chip is designed.
- A **field-programmable gate array** or FPGA is a type of chip that allows the end-user to reconfigure the circuit it computes in the field (hence the name).

# FPGA as ML accelerator (continued)

FPGAs consist of an array of programmable circuits that can each individually do a small amount of computation, as well as a programmable interconnect that connects these circuits together. The large number of programmable gates in the FPGA makes it a **naturally highly parallel device**.

- You can program it by specifying the circuit you want it to compute in terms of logic gates: basic AND, OR, NOT, etc.
  - Although in practice higher-level languages like Verilog are used.
- This doesn't actually involve physical changes to the circuit that's actually etched on the physical silicon of the FPGA: that's fixed. Rather, the FPGA constructs a logical circuit that is reconfigurable.
  - FPGAs were used historically for circuit simulation.

# Why would we want to use an FPGA instead of a GPU/GPU?

- An important property of FPGAs that distinguishes them from CPUs/GPUs: you can choose to have data flow through the chip however you want!
  - Unlike CPUs which are tied to their cache-hierarchy-based data model, or GPUs which perform best under a streaming model.
- FPGAs often use less power to accomplish the same work compared with other architectures.
  - But they are also typically slower.

# Why would we want to use an FPGA instead of an ASIC?

- Pro for FPGA: Much cheaper to program and FPGA than the design an ASIC.
- Pro for FPGA: A single FPGA costs much less than the first ASIC you synthesize.
- Pro for FPGA: FPGA designs can be adjusted on the fly.
- Pro for ASIC: The marginal cost of producing an additional ASIC is lower if you really want to synthesize millions or billions of them.
- Pro for ASIC: ASICs can typically achieve higher speed and lower power.

# Users of FPGAs for Machine Learning

e.g. Microsoft's Project Catapult/Project Brainwave.

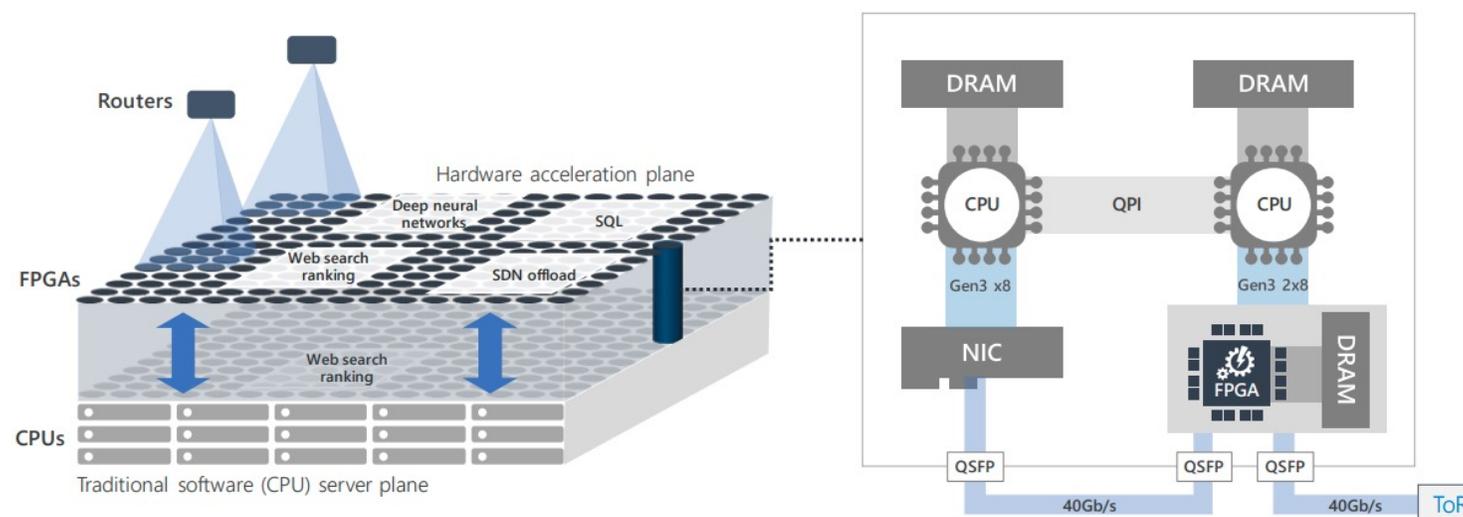


Figure 1. The first generation of Catapult-enhanced servers in production (right) consists of dual Xeon CPUs with a PCIe-attached FPGA. Each FPGA sits in-line between the 40Gbps server NIC and the TOR, enabling in-situ processing of network packets and point-to-point connectivity with up to hundreds of thousands of other FPGAs at datacenter scale. Multiple FPGAs can be allocated as a single shared hardware microservice with no software in the loop (left), enabling scalable workloads and better load balancing between CPUs and FPGAs.

from the paper "*Serving DNNs in RealTime at Datacenter Scale with Project Brainwave*"

An example of a designed-for-ML accelerator

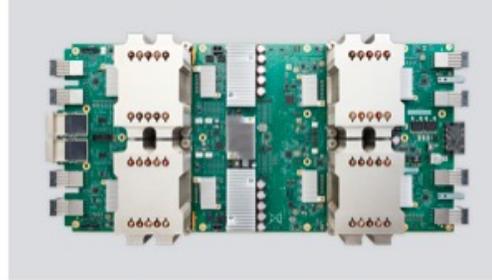
# The Tensor Processing Unit (TPU)

Google's Tensor Processing Unit (TPU) in 2015 was one of the first specialized architectures for machine learning and AI applications.

- The original version focused on **fast inference via high-throughput 8-bit arithmetic.**
- Most of the chip is dedicated to **accelerating 8-bit integer dense-matrix-dense-matrix multiplies**
  - Note that even though the numbers it multiplies are in 8-bit, it uses 32-bit accumulators to sum up the results.
  - This larger accumulator is common in ML architectures that use low precision.
- ...with a little bit of logic on the side to apply activation functions.  
**The second- and third-generation TPUs are designed to also support training and can calculate in float/bfloat16.**

Now ML accelerators are found in the cloud!

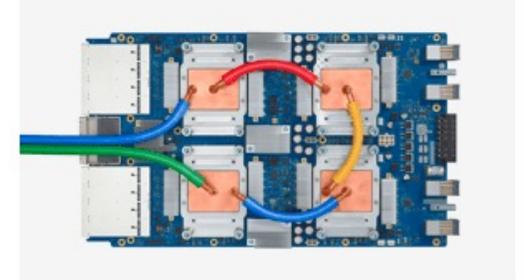
## Cloud TPU offering



**Cloud TPU v2**

180 teraflops

64 GB High Bandwidth Memory (HBM)



**Cloud TPU v3**

420 teraflops

128 GB HBM

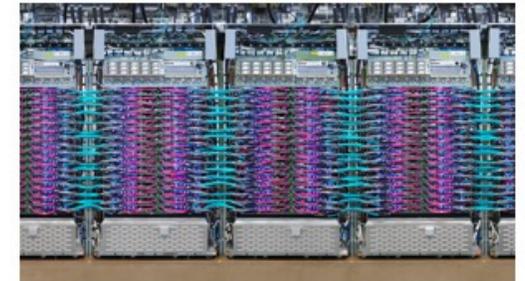


**Cloud TPU v2 Pod**

11.5 petaflops

4 TB HBM

2-D toroidal mesh network



**Cloud TPU v3 Pod**

100+ petaflops

32 TB HBM

2-D toroidal mesh network

# How do we program a TPU

- Essentially the same as for a GPU
- Google supports running ML kernels on TPUs.
  - A TPU is just another accelerator the ML framework can compute on and store data on
  - This makes it (relatively) easy to train and infer deep neural networks using tools you're already familiar with.

# Hybrid Systems

- Recent systems-on-a-chip developed for personal devices (laptops, phones) now tend to include AI accelerators.
  - You can think of this as like a CPU+GPU combo
- Examples include the M1 and M2 chips from Apple.

M2\_chip\_layout.png

# How to choose your accelerator?

- Different accelerators provide different trade-offs
- Need to ask yourself some questions
  - Am I running training or inference?
  - What is my budget?
  - Where am I running? Am I limited to one cloud provider? Am I running on an edge device?
  - What did other people with similar tasks use?
  - Do I care more about throughput or latency? Or energy?