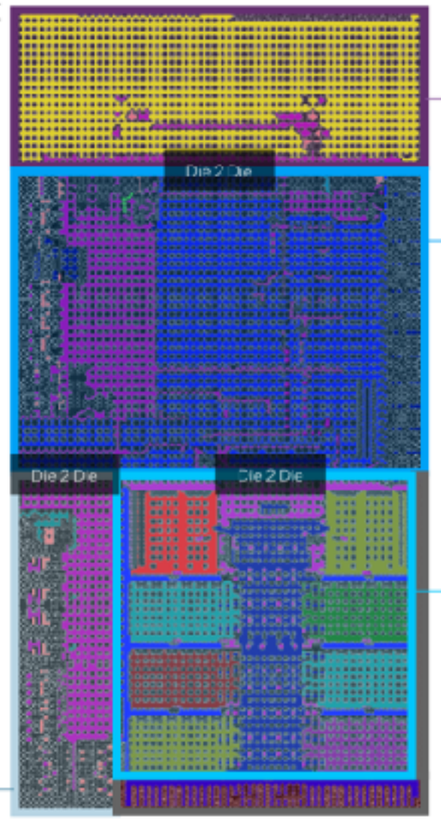
General purpose computing on hardware accelerators

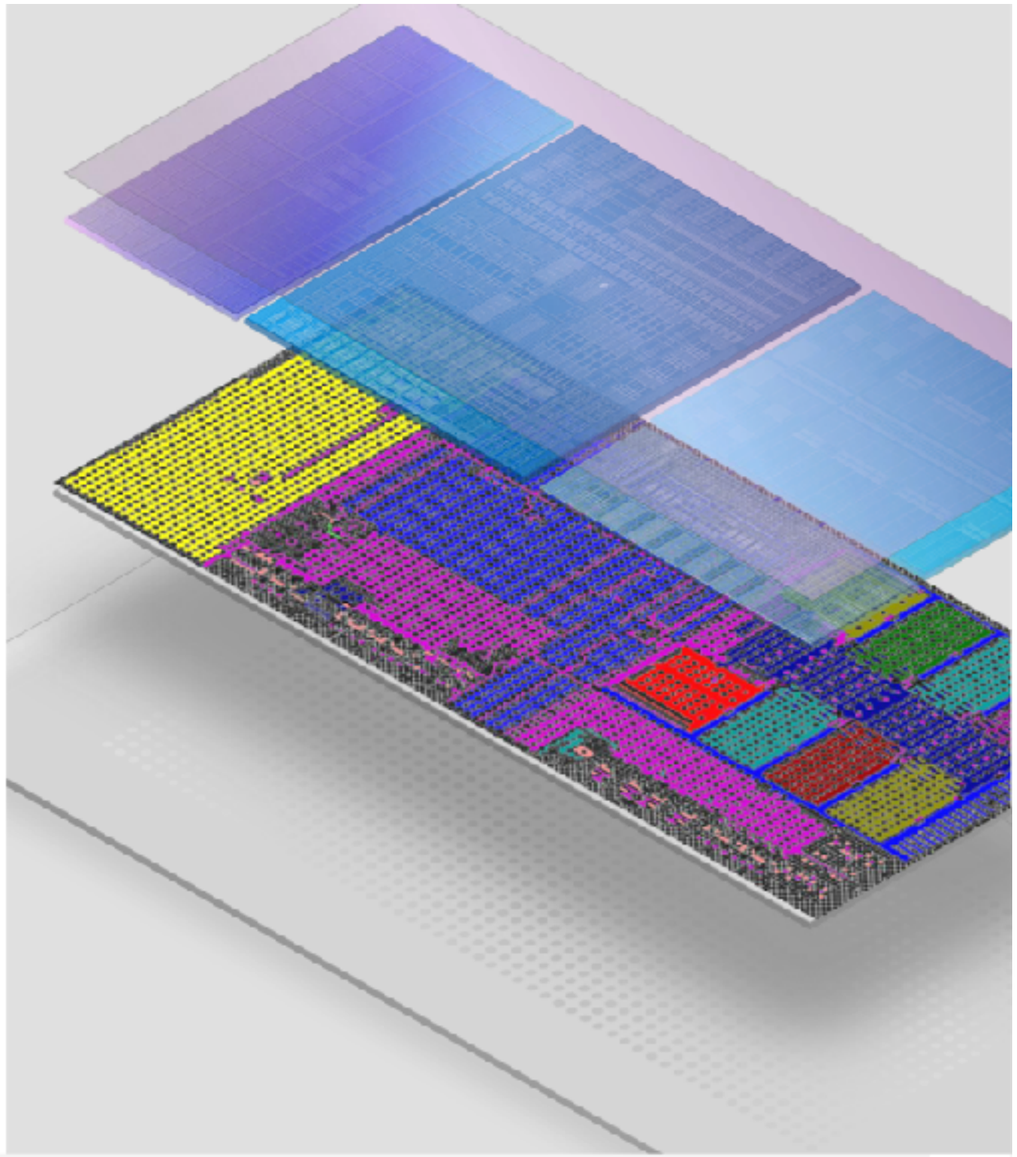
Hung-Wei Tseng

Intel Meteor Lake

Meteor Lake

Colors Represent
3D Capacitors,
Voltage Islands

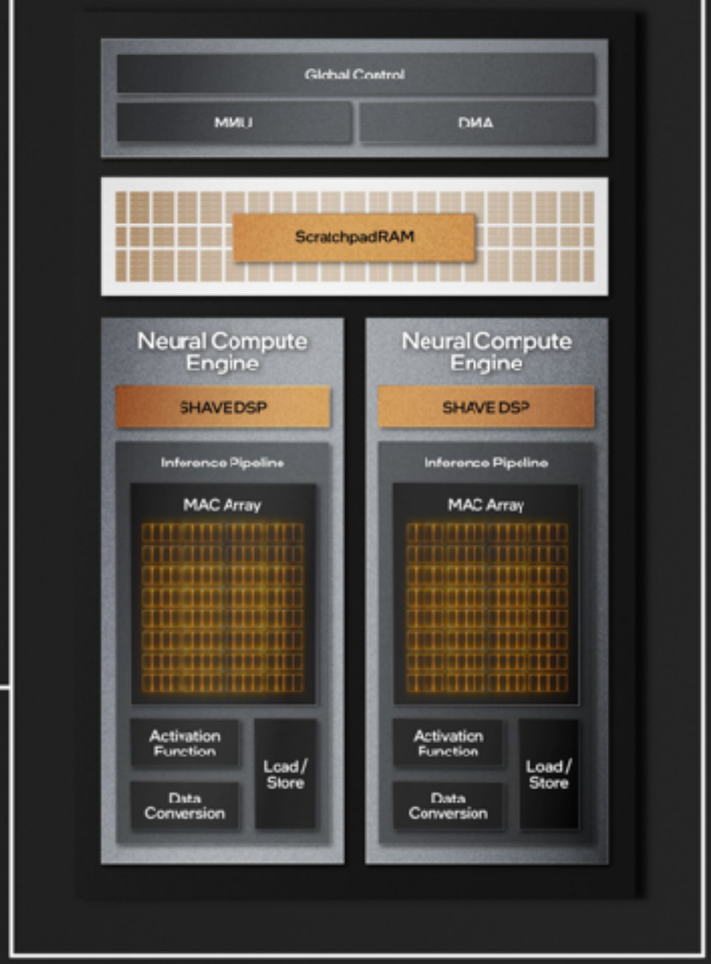




AI Accelerator is part of a CPU now

METEOR LAKE

NPU Architecture



Streaming Multprocessors in NVIDIA Blackwell

RTX Blackwell Neural Rendering Architecture

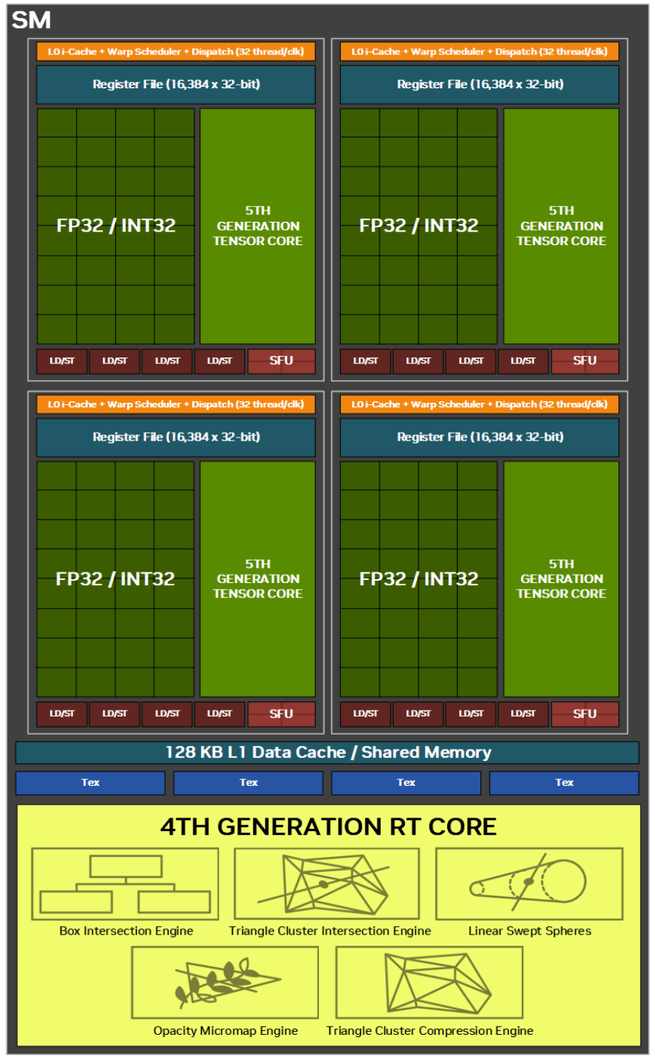
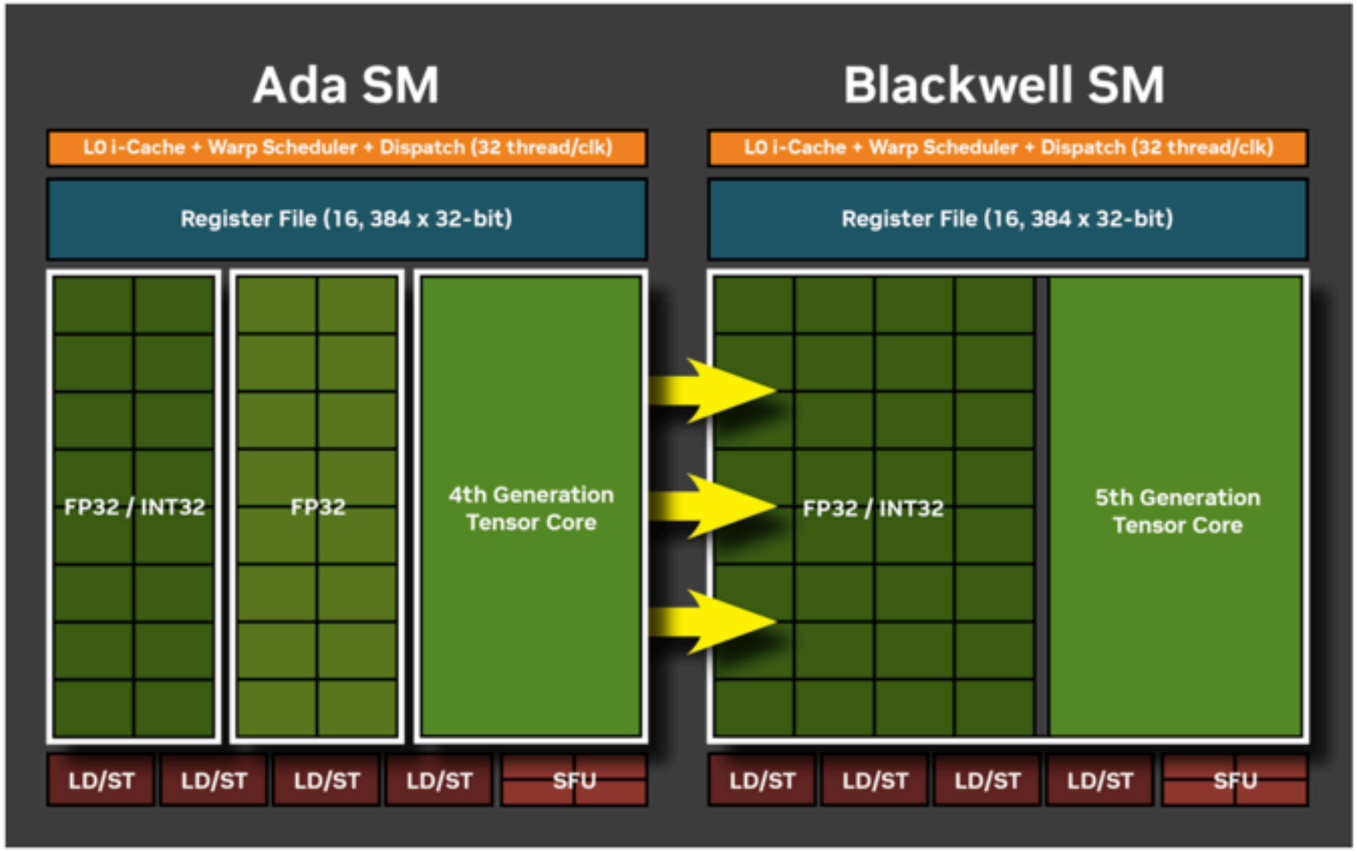
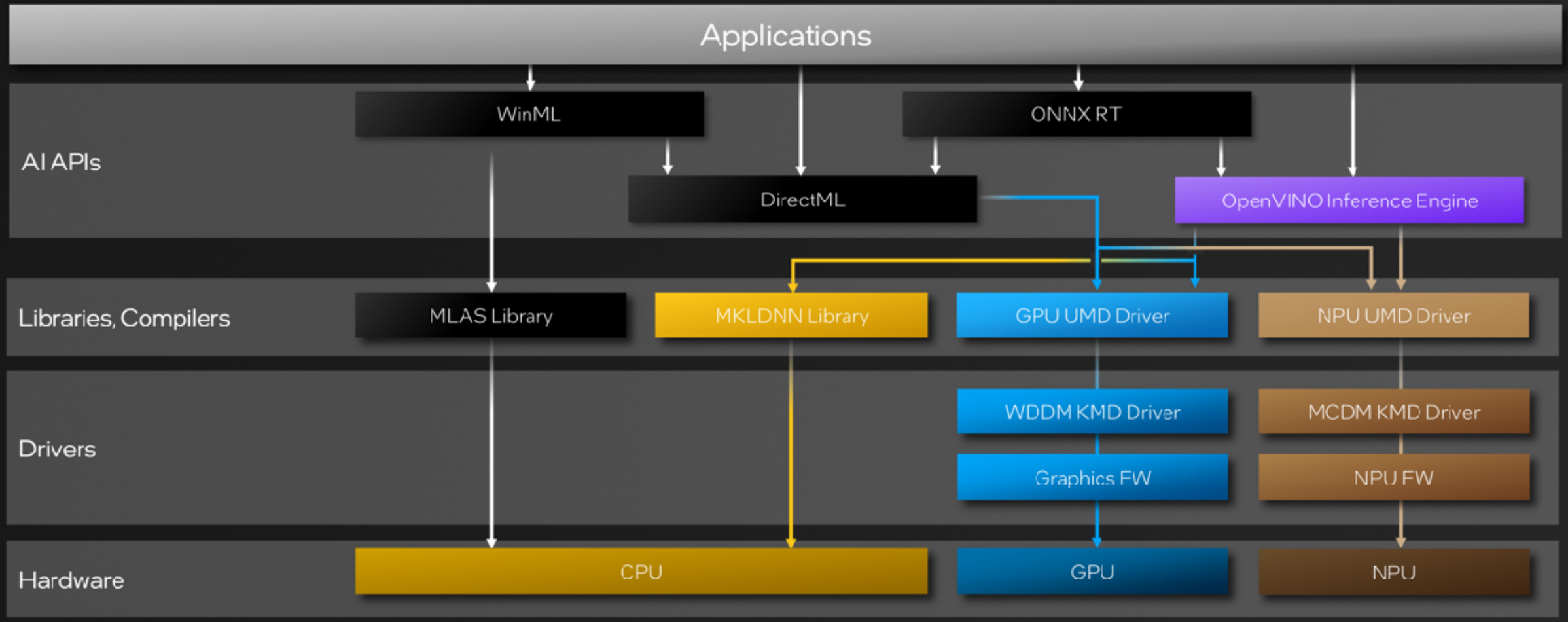


Figure 5. The Blackwell Streaming Multiprocessor (SM)

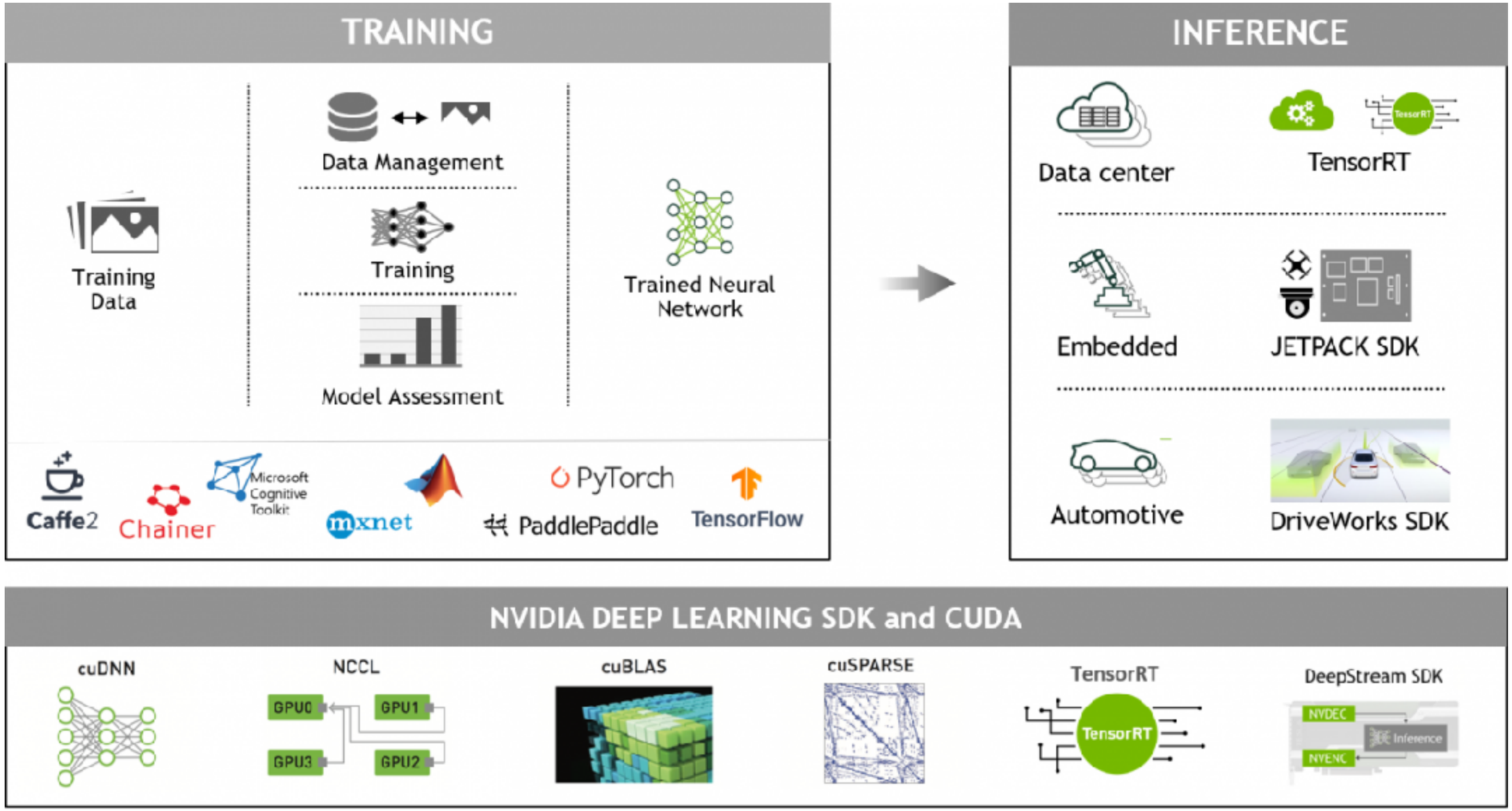
Note that the number of possible INT32 integer operations in Blackwell are doubled compared to Ada, by fully unifying them with FP32 cores, as depicted in Figure 6 below. However, the unified cores can only operate as either FP32 or INT32 cores in any given clock cycle. Figure 6 below shows how the SM architecture evolved between Ada and Blackwell.



AI Software Stack



NVIDIA's system stack



Outline

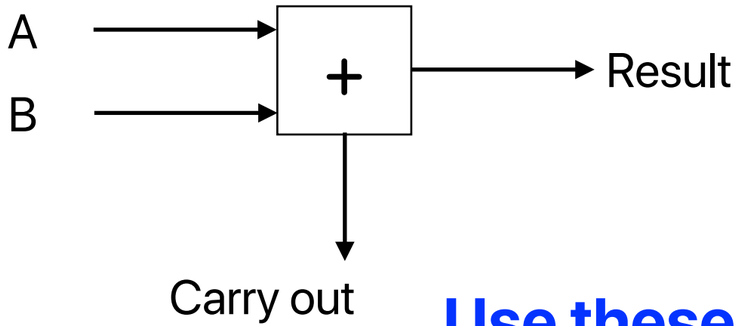
- FPGAs in data centers
- General-purpose programming on hardware accelerators
  - Neural Acceleration for General-Purpose Approximate Programs (Approximate computing on NPUs)
  - Edge Tensor Processing Units
  - Accelerating Applications using Edge Tensor Processing Units (GP Edge TPU)
  - TCUDB: Accelerating Database with Tensor Processors
  - RTNN: accelerating neighbor search using hardware ray tracing

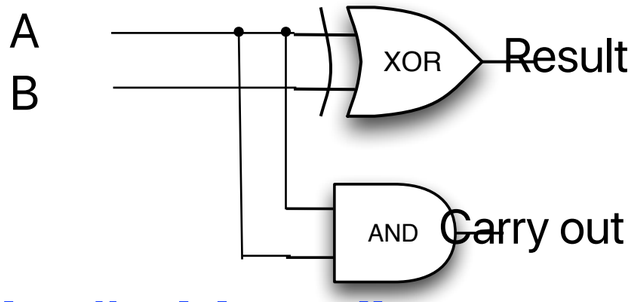
A Cloud-Scale Acceleration Architecture

Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, Doug Burger
Microsoft

How to implement a half adder?

- What gates do you need to implement a half adder?





Use these as the "address"

This is what we really want for a certain feature!

A	B	Result	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

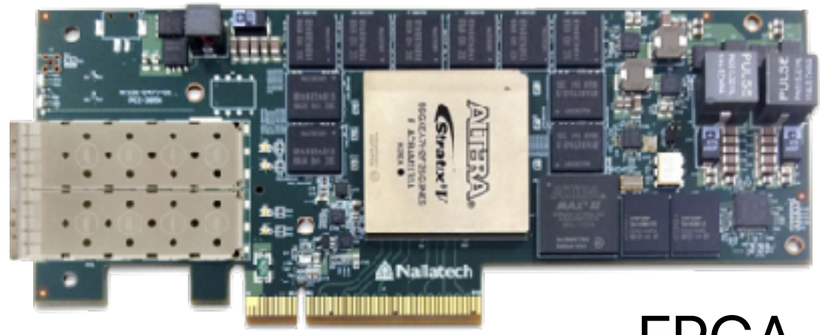
Use SRAM cells to keep these

Lookup table

A	B	Result	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

SRAM Cell Value
00
10
10
01

FPGA

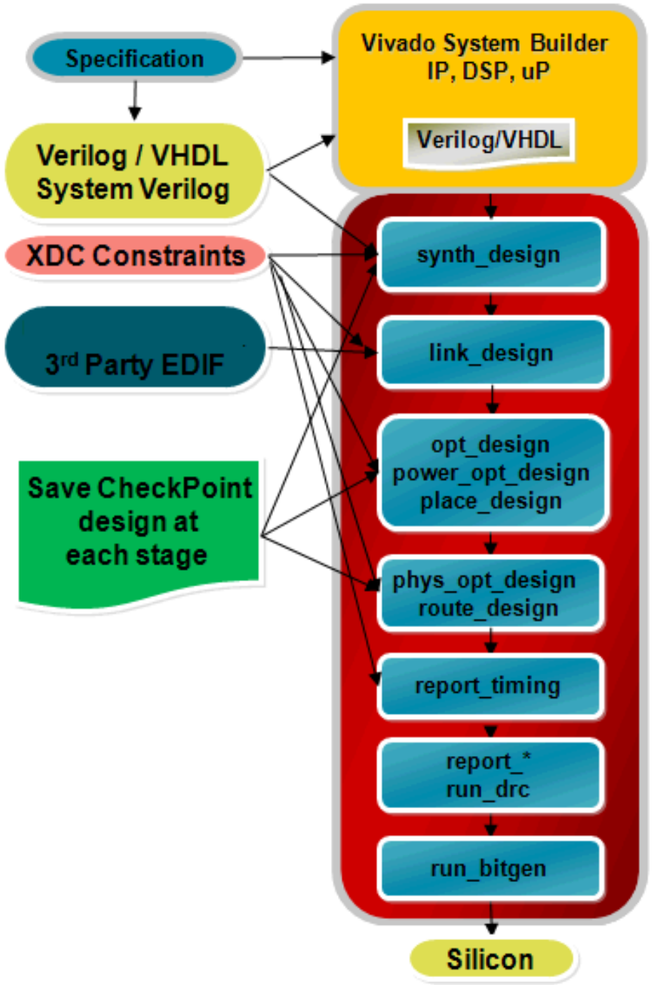


FPGA

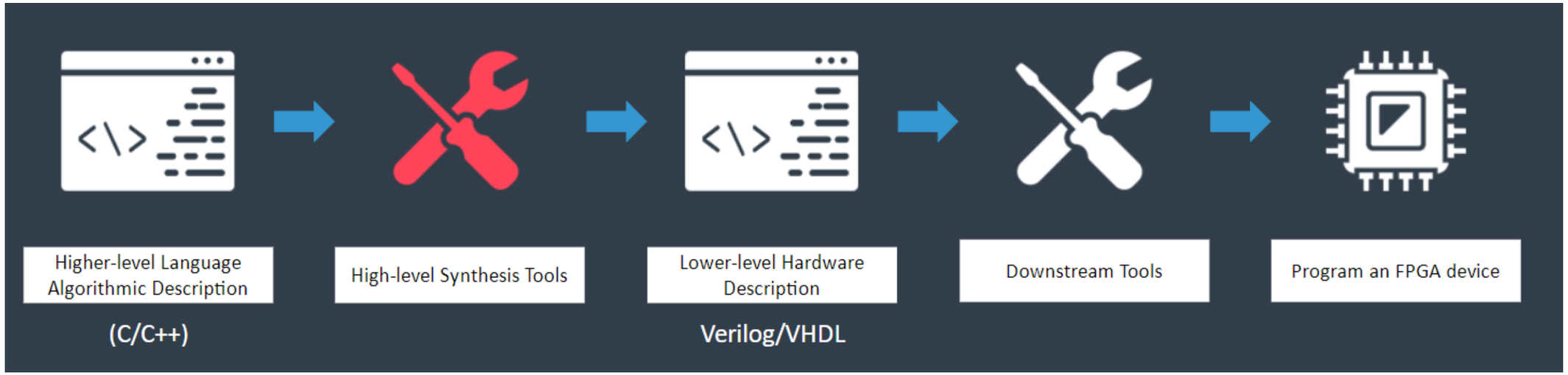
- Field Programmable Gate Array
  - An array of “Lookup tables (LUTs)”
  - Reconfigurable wires or say interconnects of LUTs
  - Registers
- An LUT
  - Accepts a few inputs
  - Has SRAM memory cells that store all possible outputs
  - Generates outputs according to the given inputs
- As a result, you may use FPGAs to emulate any kind of gates or logic combinations, and create an ASIC-like processor

FPGA Design Flow

- **Interactive IP plug-n-play environment**
  - AXI4, IP_XACT
- **Common constraint language (XDC) throughout the flow**
  - Apply constraints at any stage
- **Reporting at any stage**
  - Robust Tcl API
- **Common data model throughout the flow**
  - “In memory” model improves speed
  - Generate reports at all stages
- **Save checkpoint designs at any stage**
  - Netlist, constraints, place and route results



Also possible to programm an FPGA using C/C++



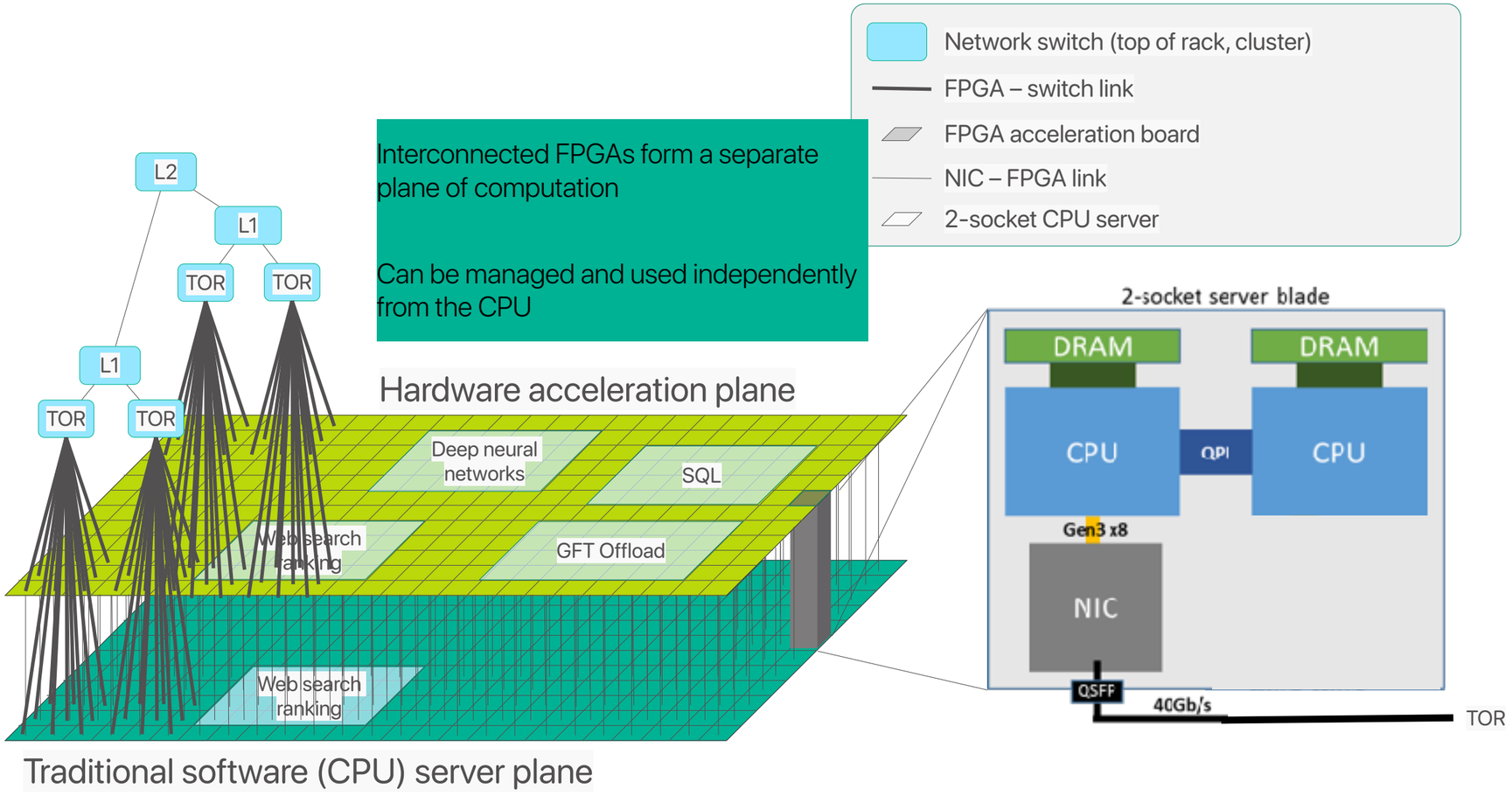
- Retain the advantages of a programming language to write efficient code that can be translated into hardware
- Additional work (rewriting the code) require for the desired performance goals even if the C/C++ code can be automatically converted into hardware.

Producer-Consumer Paradigm

Stream data Paradigm

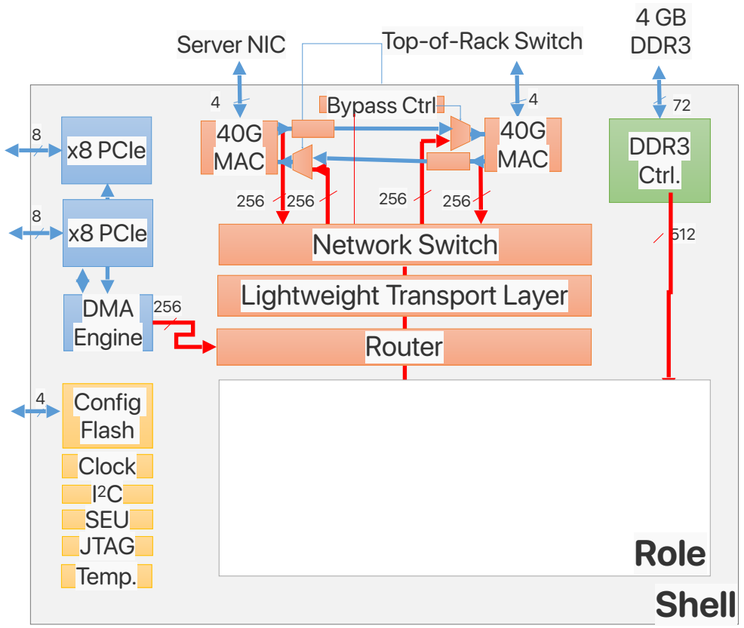
Pipelining Paradigm

Configurable cloud



Gen2 shell

- Foundation for all accelerators
  - Includes PCIe, Networking and DDR IP
  - Common, well tested platform for development
- Lightweight Transport Layer
  - Reliable FPGA-to-FPGA Networking
  - Ack/Nack protocol, retransmit buffers
  - Optimized for lossless network
  - Minimized resource usage



3:00

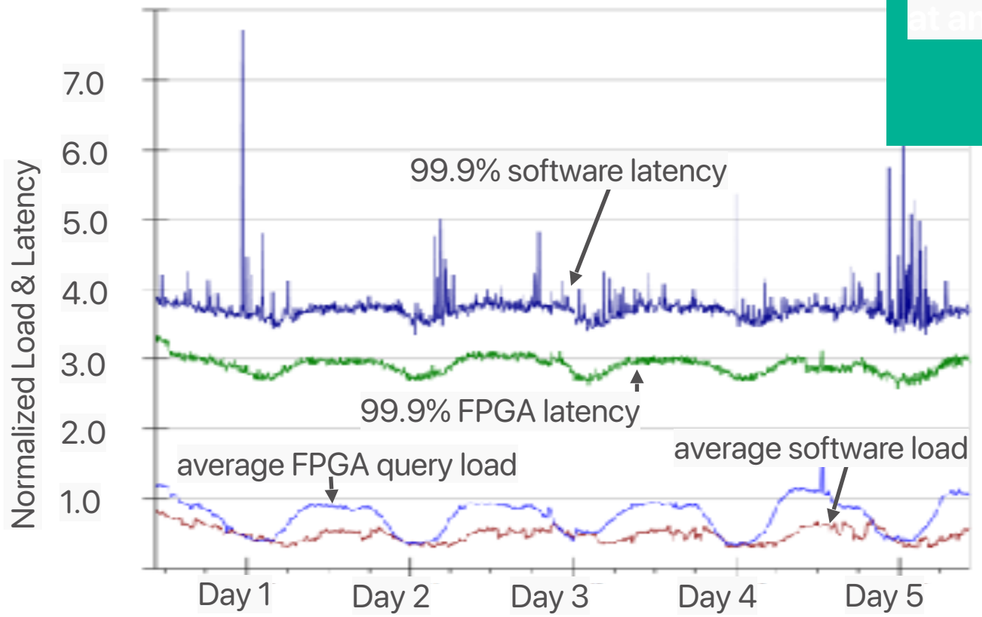
Why does M$ integrate FPGAs in their data centers?

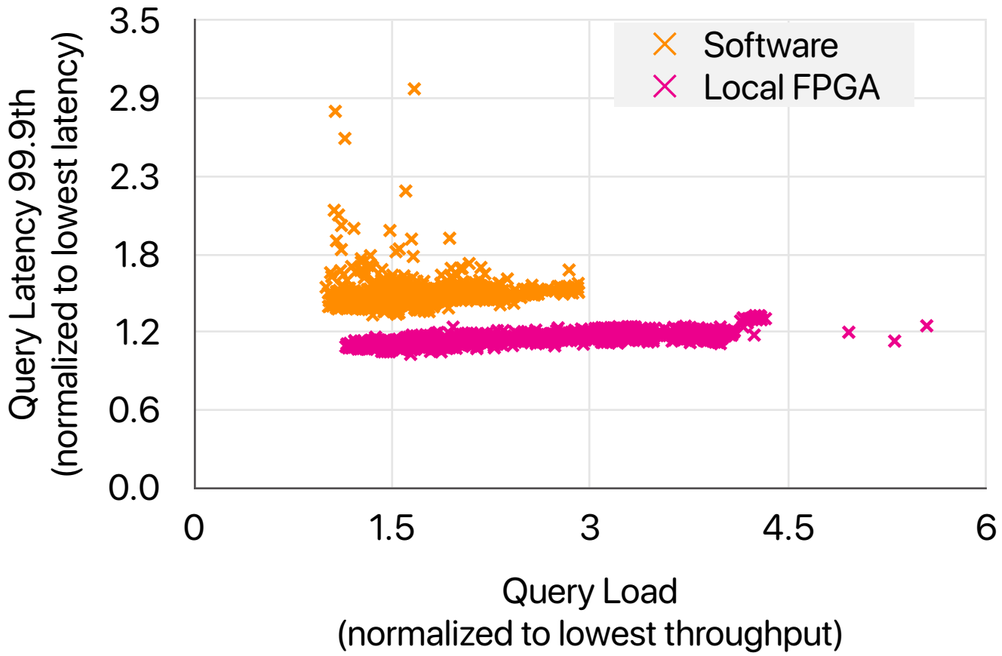
Use cases

- Local: Great service acceleration
- Infrastructure: Fastest cloud network
- Remote: Reconfigurable app fabric (DNNs)

5 day bed-level latency

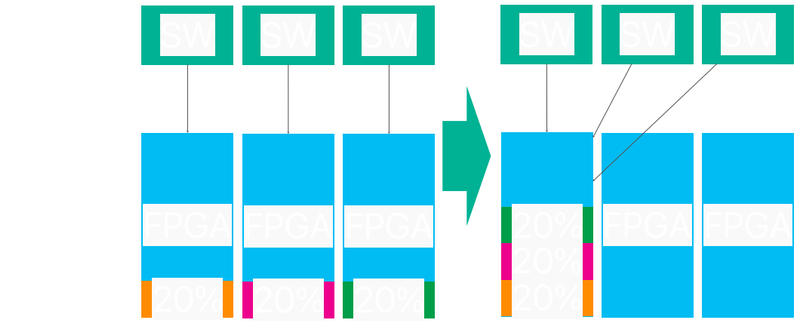
- Lower & more consistent 99.9th tail latency
- In production for years

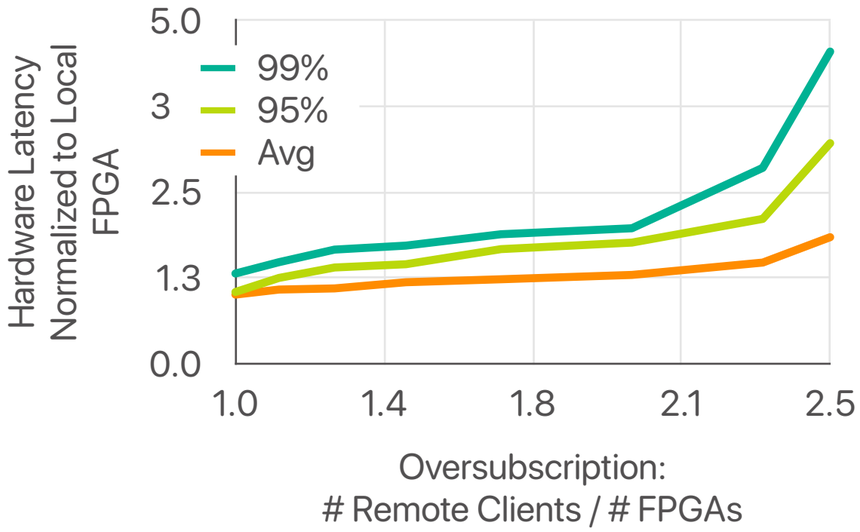




Shared DNN

- Economics: consolidation
  - Most accelerators have more throughput than a single host requires
  - Share excess capacity, use fewer instances
  - Frees up FPGAs for other use services
- DNN accelerator
  - Sustains 2.5x busy clients in microbenchmark, before queuing delay drives latency up



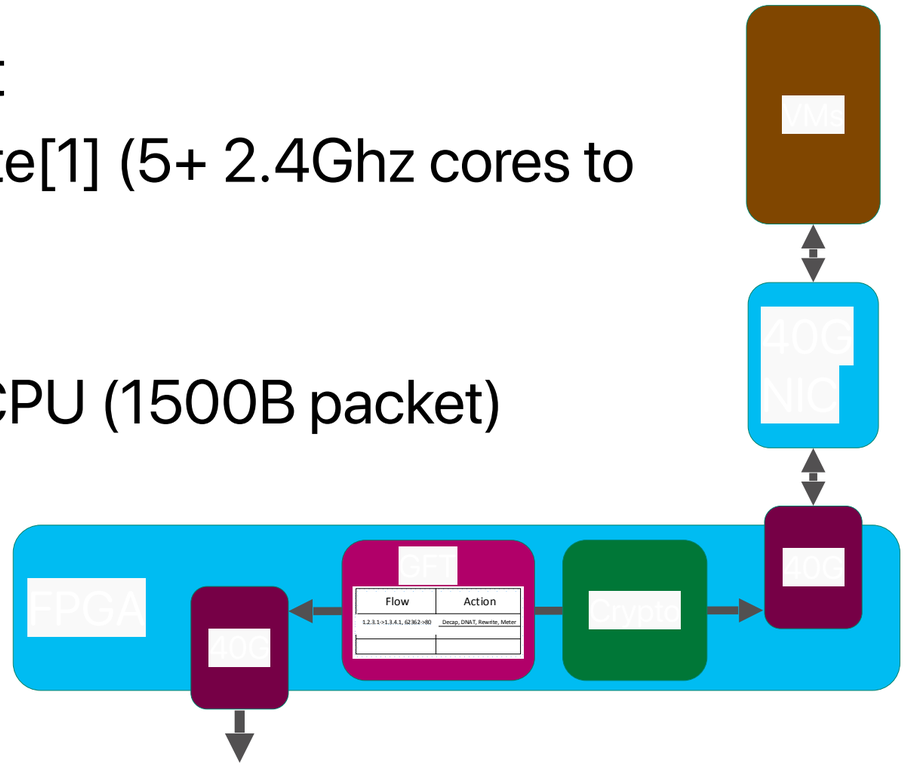


Accelerated networking

- Software defined networking
  - Generic Flow Table (GFT) rule based pack
  - 10x latency reduction vs software, CPU lo
  - 25Gb/s throughput at 25µs latency – the fastest cloud network

> Our FPGA implementation supports full 40 Gb/s encryption and decryption. The worst case half-duplex FPGA crypto latency for AES-CBC-128-SHA1 is 11 μs for a 1500B packet, from first flit to first flit. In software, based on the Intel numbers, it is approximately 4 μs. AES-CBC-SHA1 is, however,

- Capable of 40 Gb line rate encrypt and decrypt
  - On Haswell, AES GCM-128 costs 1.26 cycles/byte[1] (5+ 2.4Ghz cores to sustain 40Gb/s)
  - CBC and other algorithms are more expensive
  - AES CBC-128-SHA1 is 11µs in FPGA vs 4µs on CPU (1500B packet)
  - **Higher latency, but significant CPU savings**



Why FPGA?

This model offers significant flexibility. From the local perspective, the FPGA is used as a compute or a network accelerator. From the global perspective, the FPGAs can be managed as a large-scale pool of resources, with acceleration

hyperscale infrastructure. The acceleration system we describe is sufficiently flexible to cover three scenarios: local compute acceleration (through PCIe), network acceleration, and global application acceleration, through configuration as pools of remotely accessible FPGAs. Local acceleration handles high-

These programmable architectures allow for hardware homogeneity while allowing fungibility via softwa[illegible]ifferent services. They must be highly *flexible* at the [illegible]stem [illegible]

Flexible

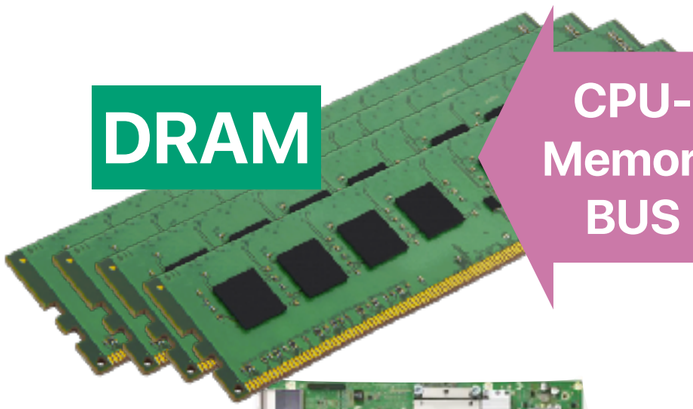
This paper described Configurable Clouds, a datacenter-scale acceleration architecture, based on FPGAs, that is both scalable and flexible. By putting in FPGA cards both in I/O

In addition to architectural requirements that provide sufficient flexibility to justify scale production deployment, there are also physical restrictions in current infrastructures that

3:00

Refresh our minds. Can we compare GPUs/Accelerators(ASICs)/FPGAs?

Recap: The landscape of modern computers



CPU-Memory BUS

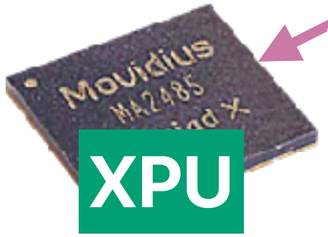
DDR4

Memory Controller

High-performance, but very specialized

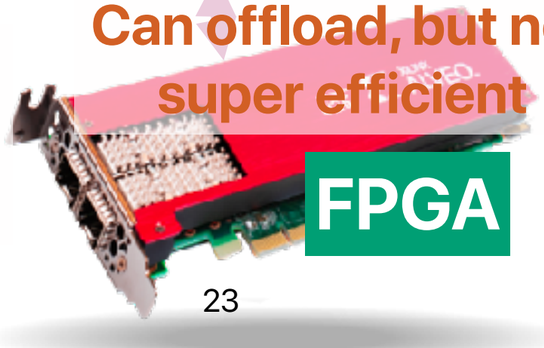
PCI EXPRESS

PCIe Root Complex



General-purpose for "regular" workloads, but can be power consuming

Can offload, but not super efficient



Can we take both advantages of programmability, energy-efficiency and performance?

3:00

If the only operator available to you is matrix multiplications and accumulations, how're you going to write your programs?

Neural Acceleration for General-Purpose Approximate Programs

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, Doug Burger*
University of Washington and Microsoft*
In 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2012)

How can we understand a paper within an hour?

Why, what, and how

- Why — the most important thing of a research paper
  - The big problem/question this paper is trying to address/answer
  - Why should you, as a reader, care about this paper
- What — novel ideas or interesting things to the why
  - Novel solutions to the why
  - Novel high-level design of the system
  - Novel use of an existing mechanism to solve a new problem
- How — implementation of the paper (not a contribution, but make the paper more convincing)
  - How does the implementation reflects the what?

3:00

What's the why of "Neural Acceleration for General-Purpose Approximate Programs"

Why "Neural Acceleration for General-Purpose Approximate Programs"

- ASICs/accelerators are energy-efficient, but hard-to-program
- FPGAs are slow if memory accesses are frequent
- GPUs do not work well if the workload is not regular
- Many applications tolerate inexact computation

1. Introduction

Energy efficiency is a primary concern in computer systems. The cessation of Dennard scaling has limited recent improvements in transistor speed and energy efficiency, resulting in slowed general-purpose processor improvements. Consequently, architectural innovation has become crucial to achieve performance and efficiency gains [10].

However, there is a well-known tension between efficiency and programmability. Recent work has quantified three orders of magnitude of difference in efficiency between general-purpose processors and ASICs [21, 36]. Since designing ASICs for the massive base of quickly changing, general-purpose applications is currently infeasible, practitioners are increasingly turning to programmable accelerators such as GPUs and FPGAs. Programmable accelerators provide an intermediate point between the efficiency of ASICs and the generality of conventional processors, gaining significant efficiency for restricted domains of applications.

Programmable accelerators exploit some characteristic of an application domain to achieve efficiency gains at the cost of generality. For instance, FPGAs exploit copious, fine-grained, and irregular parallelism but perform poorly when complex and frequent accesses to memory are required. GPUs exploit many threads and SIMD-style parallelism but lose efficiency when threads diverge. Emerging accelerators, such as BERET [19], Conservation Cores and Qs-

Tolerance to approximation is one such program characteristic that is growing increasingly important. Many modern applications—such as image rendering, signal processing, augmented reality, data mining, robotics, and speech recognition—can tolerate inexact computation in substantial portions of their execution [7, 14, 28, 41]. This tolerance can be leveraged for substantial performance and energy gains.

What this paper proposed?

- Replacing the original algorithm with a machine learning model
- Learning algorithm, langage/compilation framework, architectural interface

This paper introduces a new class of programmable accelerators that exploit approximation for better performance and energy efficiency. The key idea is to *learn* how an original region of approximable code behaves and replace the original code with an efficient computation of the learned model. This approach contrasts with previous work on approximate computation that extends conventional microarchitectures to support selective approximate execution, incurring instruction bookkeeping overheads [1, 8, 11, 29], or requires vastly different programming paradigms [4, 24, 26, 32]. Like emerging flexible accelerators [18, 19, 47, 48], our technique automatically offloads code segments from programs written in mainstream languages; but unlike prior work, it leverages changes in the semantics of the offloaded code.

We have identified three challenges that must be solved to realize effective trainable accelerators:

1. A **learning algorithm** is required that can accurately and efficiently mimic imperative code. We find that neural networks can approximate various regions of imperative code and propose the Parrot transformation, which exploits this finding (Section 2).
2. A **language and compilation framework** should be developed to transform regions of imperative code to neural network evaluations. To this end, we define a programming model and implement a compilation workflow to realize the Parrot transformation (Sections 3 and 4). The Parrot transformation starts from regions of approximable imperative code identified by the programmer, collects training data, explores the topology space of neural networks, trains them to mimic the regions, and finally replaces the original regions of code with trained neural networks.
3. An **architectural interface** is necessary to call a neural processing unit (NPU) in place of the original code regions. The NPU we designed is tightly integrated with a speculative out-of-order core. The low-overhead interface enables acceleration even when fine-grained regions of code are transformed. The core communicates both the neural configurations and run-time invocations to the NPU through extensions to the ISA (Sections 5 and 6).

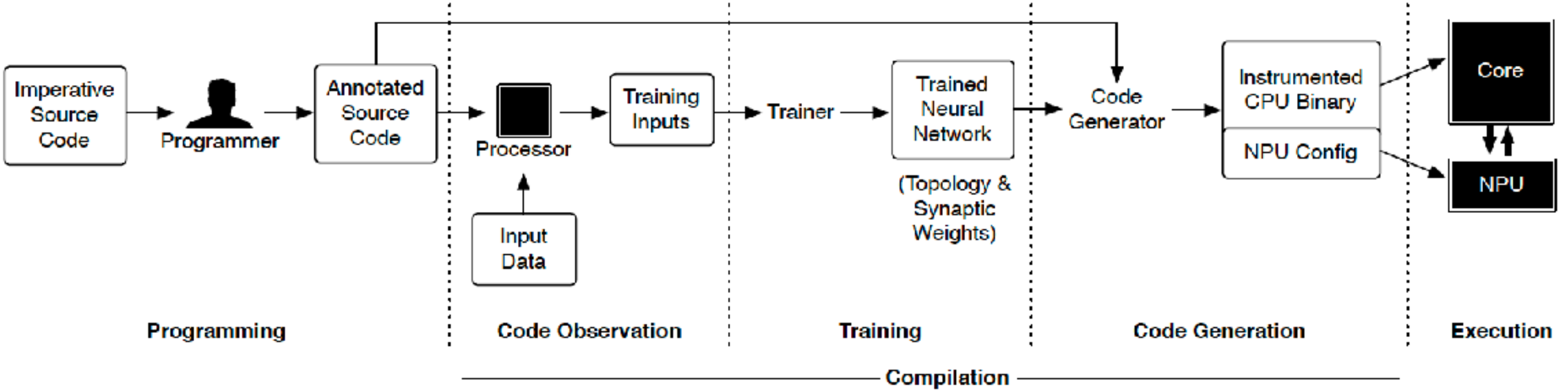


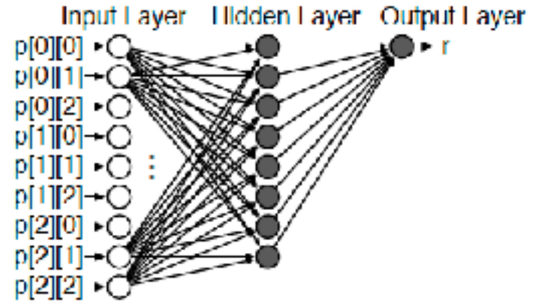
Figure 1: The Parrot transformation at a glance: from annotated code to accelerated execution on an NPU-augmented core.

The overall architecture

```
1  float sobel [[PARROT]] (float[3][3] p){
     float x, y, r;
3    x = (p[0][0] + 2 * p[0][1] + p[0][2]);
     x = (p[2][0] + 2 * p[2][1] + p[2][2]);
5    y = (p[0][2] + 2 * p[1][2] + p[2][2]);
     y = (p[0][0] + 2 * p[1][1] + p[2][0]);
7    r = sqrt(x * x + y * y);
     if (r >= 0.7071) r = 0.7070;
9    return r;
   }
```

```
   void edgeDetection(Image& srcImg, Image& dstImg){
2    float[3][3] p; float pixel;
     for(int y = 0; y < srcImg.height; ++y)
4      for(int x = 0; x < srcImg.width; ++x)
         srcImg.toGrayeScale(x,y);
6    for(int y = 0; y < srcImg.height; ++y)
       for(int x = 0; x < srcImg.width; ++x){
8        p = srcImg.build3x3Window(x, y);
         pixel = sobel(p);
10       dstImg.setPixel(x, y, pixel);
       }
12 }
```

(a) Original implementation of the Sobel filter



(b) The sobel function transformed to a $9 \rightarrow 8 \rightarrow 1$ NN

```
   void edgeDetection(Image& srcImg, Image& dstImg){
2    float[3][3] p; float pixel;
     for(int y = 0; y < srcImg.height; ++y)
4      for(int x = 0; x < srcImg.width; ++x)
         srcImg.toGrayeScale(x,y);
6    for(int y = 0; y < srcImg.height; ++y)
       for(int x = 0; x < srcImg.width; ++x){
8        p = srcImg.build3x3Window(x, y);
         NPU_SEND(p[0][0]); NPU_SEND(p[0][1]); NPU_SEND(p[0][2]);
10       NPU_SEND(p[1][0]); NPU_SEND(p[1][1]); NPU_SEND(p[1][2]);
         NPU_SEND(p[2][0]); NPU_SEND(p[2][1]); NPU_SEND(p[2][2]);
12       NPU_RECEIVE(pixel);
         dstImg.setPixel(x, y, pixel);
14     }
   }
```

(c) parrot transformed code; an NPU invocation replaces the function call

Figure 2: Three stages in the transformation of an edge detection algorithm using the Sobel filter.

How

- Something hard to find in introduction, but we don't care that much unless we want to implement/reproduce that
- They simulated the system

Code annotation. The C source code for each benchmark was annotated as described in Section 3: we identified a single pure function with fixed-size inputs and outputs. No algorithmic changes were made to the benchmarks to accommodate the Parrot transformation. There are many choices for the selection of target code and, for some programs, multiple NPUs may even have been beneficial. For the purposes of this evaluation, however, we selected a single target region per benchmark that was easy to identify, frequently executed as to allow for efficiency gains, and amenable to learning by a neural network. Qualitatively, we found it straightforward to identify a reasonable candidate function in each benchmark.

7.1. Benchmarks and the Parrot Transformation

Table 1 lists the benchmarks used in this evaluation. These benchmarks are all written in C. The application domains—signal processing, robotics, gaming, compression, machine learning, and image processing—are selected for their usefulness to general applications and tolerance to imprecision. The domains are commensurate with evaluations of previous work on approximate computing [1, 11, 28, 29, 41, 43].

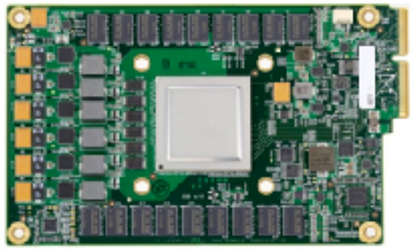
Table 1 also lists the input sets used for performance, energy, and accuracy assessment. These input sets are different from the ones used during the training phase of the Parrot transformation. For applications with random inputs we use a different random input set. For applications with image input, we use a different image.

7.2. Experimental Setup

Cycle-accurate simulation. We use the MARSSx86 cycle-accurate x86-64 simulator [35] to evaluate the performance effect of the Parrot transformation and NPU acceleration. Table 2 summarizes the microarchitectural parameters for the core, memory subsystem, and NPU. We configure the simulator to resemble Intel's Penryn microarchitecture, which is an aggressive out-of-order design. We augment MARSSx86 with a cycle-accurate NPU simulator and add support for NPU queue instructions through unused x86 opcodes. We use C assembly inlining to add the NPU invocation code. We compile the benchmarks using GCC version 4.4.6 with the -O3 flag to enable aggressive compiler optimizations. The baseline in all of the reported results is the execution of the entire benchmark on the core without the Parrot transformation.

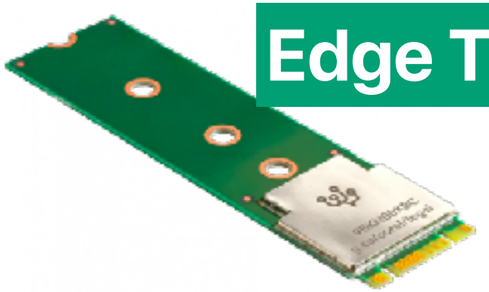
Edge TPUs

Variants of Google TPUs

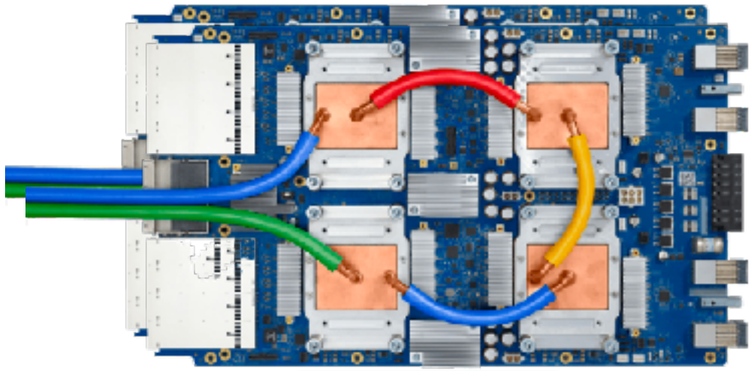


TPU v1

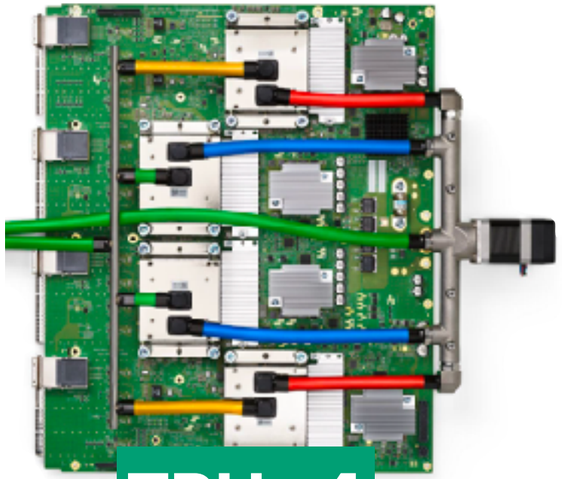
TPU v2



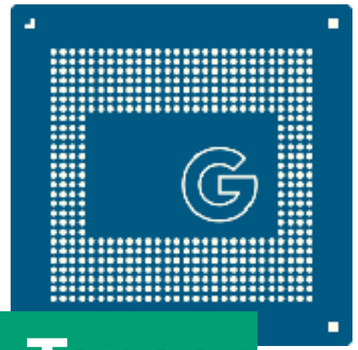
Edge TPU



TPU v3

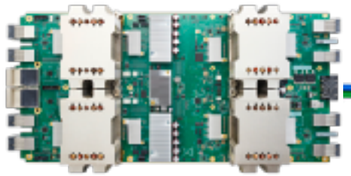
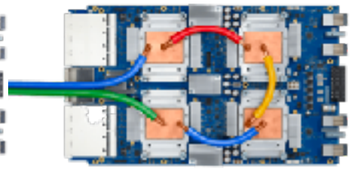
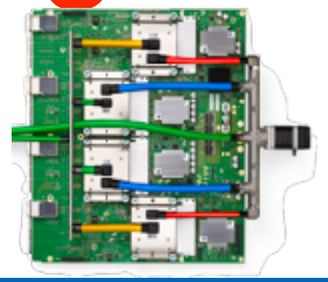


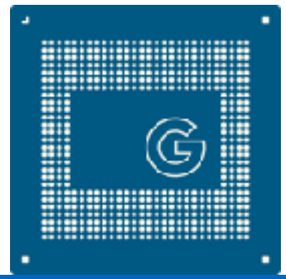
TPU v4



Google Tensor

Variants of Google TPUs

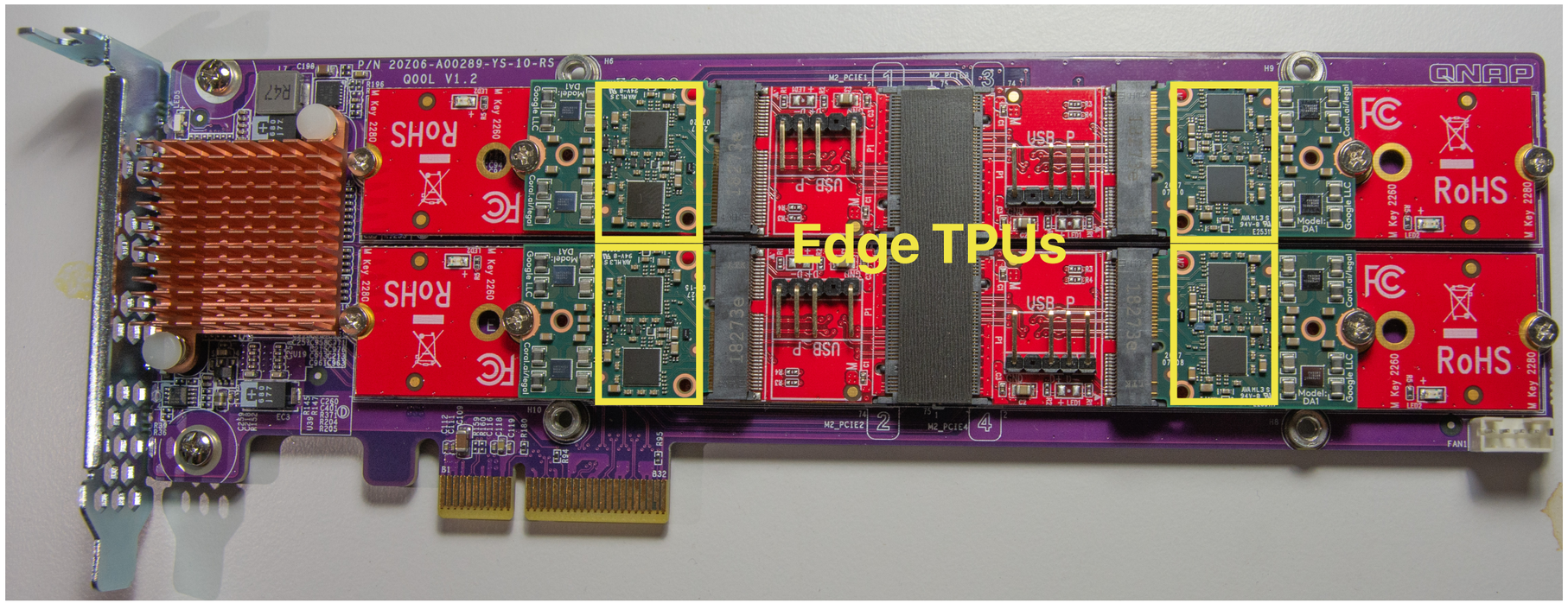







	TPU v1	TPU v2	TPU v3	TPU v4	Edge TPU	Tensor
Year	2016	2017	2018	2021	2018	2021
Precision	INT8	BF16	BF16	BF16	INT8	?
Accessibility	Google Cloud Only	Google Cloud Only	Google Cloud Only	Google Cloud Only	Public Available	Only in Google Pixels
Programming Framework	Tensorflow/PyTorch	Tensorflow/PyTorch	Tensorflow/PyTorch	Tensorflow/PyTorch	Tensorflow/TFLite/ C++	TFLite/NNAPI
TOPS	95	45	129	275	4	27
Power (Watt)	75	280	220	170	2	5
TOPS/W	1.27	0.16	0.59	1.62	2	5.4
Device Memory	24 MB	64 GB	128 GB	32 GB	8MB	N/A

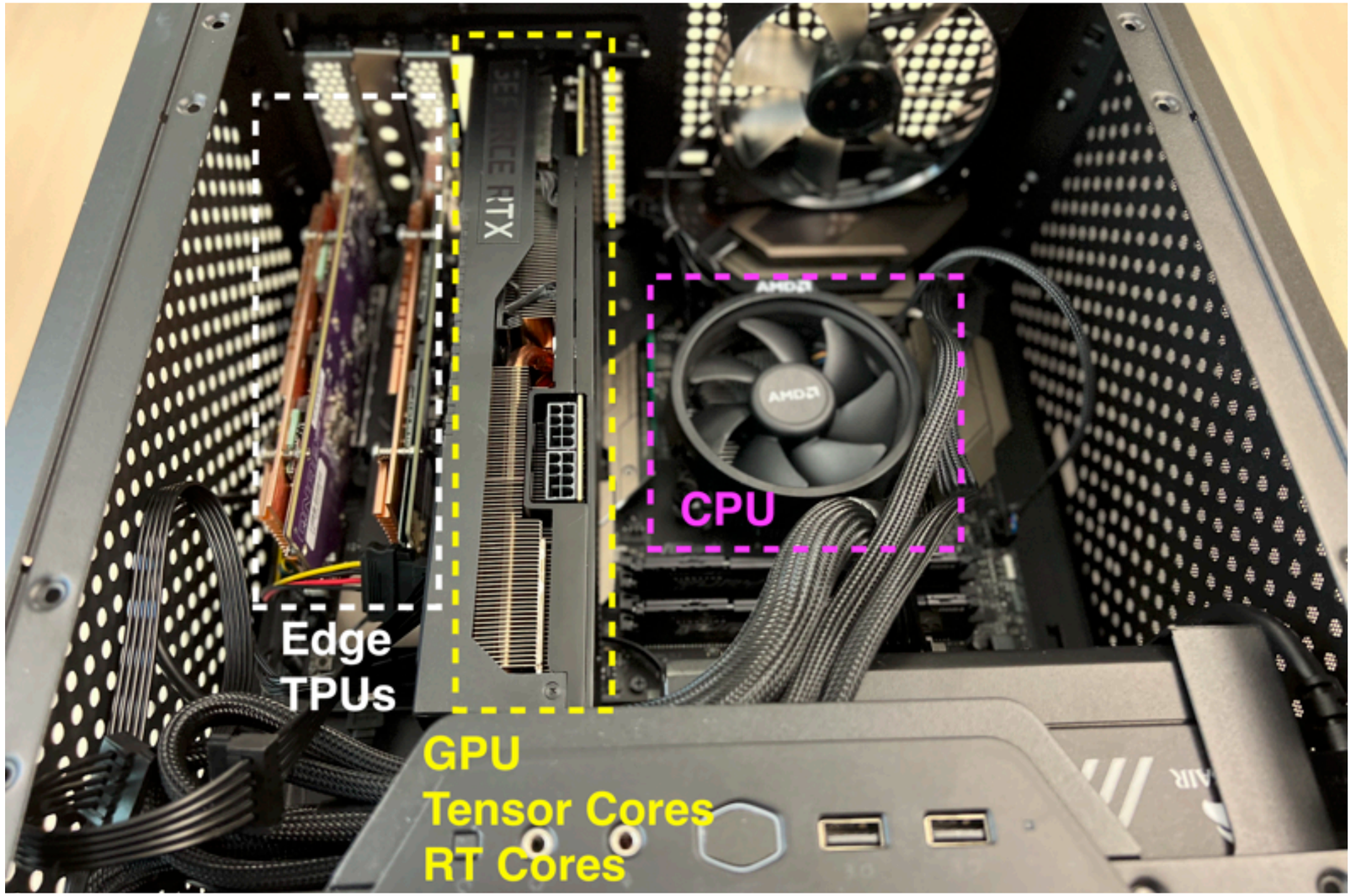
* https://www.reddit.com/r/GooglePixel/comments/1723547/just_how_fast_is_the_google_tensor_3/

Our Edge TPU Expansion Card



- Each TPU can deliver up to 4 Trillion Operations (TOPS) with 2 Watts of Power and cost USD 29
- Each card contains 4 TPUs — 16 TOPS
- GPUs can deliver 130 TOPS — GPU is still more powerful, but more power consuming, costs a lot more
- The only type of TPUs that you can order online — both in M.2 and USB

Heterogeneous computer with Edge TPU card



The concept of approximate computing on NPUs

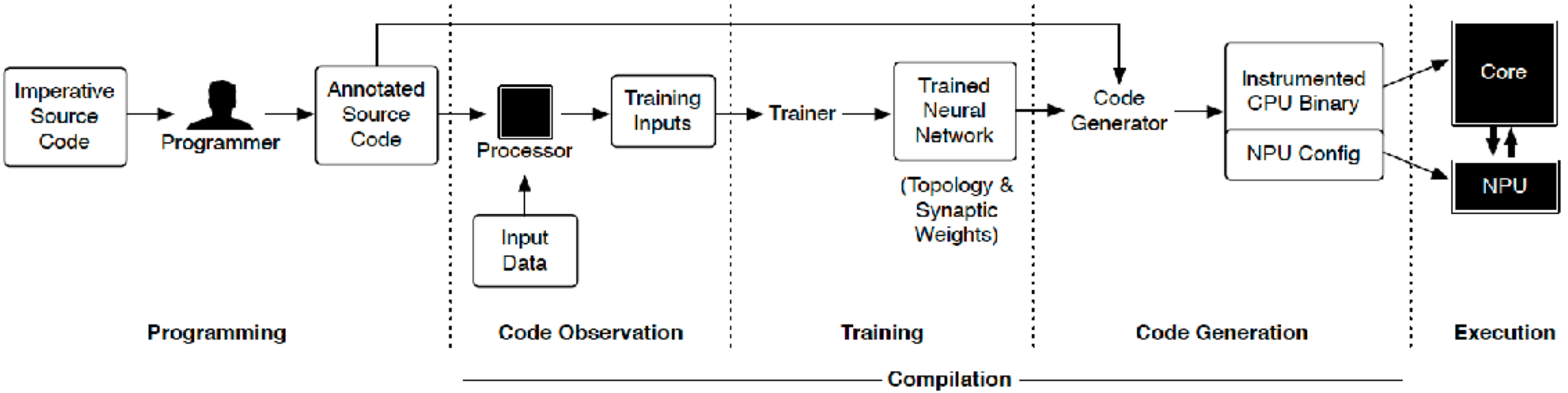


Figure 1: The Parrot transformation at a glance: from annotated code to accelerated execution on an NPU-augmented core.

Roogle Project Presentations

- Make an appointment on 4/29 and 5/1 through the Google Calendar
- 15 minute presentation with 3 minute Q & A
- Why & what & how!!! — considering you're giving a presentation at Apple's keynote
  - 7-minute why — why should everyone care about this problem? Why is this still a problem?
  - 5-minute what — what are you proposing in this project to address the problem?
  - 3-minute how — expected platforms/engineering efforts, milestones and workload distribution among members
  - Please reference this article to make a good presentation https://cseweb.ucsd.edu//~swanson/GivingTalks.html

Electrical
Computer Science
Engineering

277

つづく