7

Domain-Specific Architectures

Moore's Law can't continue forever … We have another 10–20 years before we reach a fundamental limit.

Gordon Moore,
Intel Cofounder (2005)

Computer Architecture. https://doi.org/10.1016/B978-0-443-15406-5.00007-9
Copyright © 2026 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

7.1 Introduction

Not only did Gordon Moore predict the amazing growth of transistors per chip in 1965, but the opening chapter quote shows that he also predicted its near collapse 50 years later. As evidence, Figure 1.24 in Chapter 1 shows that even the company he founded—which for decades proudly used Moore's law as a guideline for capital investment—is slowing its development of new semiconductor processes.

During the semiconductor boom time, architects rode Moore's law to create novel mechanisms that could turn the cornucopia of transistors into higher performance. The resources for a five-stage pipeline, 32-bit RISC processor—which needed as little as 25,000 transistors in the 1980s—grew by a factor of 1,000,000 to enable features that accelerated general-purpose code on general-purpose processors, as earlier chapters document:

- 1st-level, 2nd-level, 3rd-level, and even 4th-level caches
- 512-bit SIMD floating-point units
- 15+ stage pipelines
- Branch prediction
- Out-of-order execution
- Speculative prefetching
- Multithreading
- Multiprocessing with hundreds of processors per socket

These sophisticated architectures targeted million-line programs written in efficient languages like C and C++. Architects treated such code as black boxes, generally without understanding either the internal structure of the programs or even what they were trying to do. Benchmark programs like those in SPEC2017 were just artifacts to measure and accelerate. Compiler writers were the people at the hardware-software interface, which dates back to the RISC revolution in the 1980s, but they have limited understanding of the high-level application behavior; that's why compilers cannot even bridge the semantic gap between C or C++ and the architecture of GPUs.

As Chapter 1 describes, Dennard scaling ended much earlier than Moore's law. Thus, more transistors switching now means more power. The energy budget is not increasing, and we've already replaced the single inefficient processor with multiple efficient cores. Hence, we have no other major architecture innovations to continue major improvements in cost-performance and energy efficiency for general-purpose architectures. Because the energy budget is limited (because of electromigration, mechanical and thermal limits of chips), if we want higher performance (higher operations/second), we need to lower the energy per operation.

Figure 7.1 is another take on the relative energy costs of memory and logic mentioned in Chapter 1, this time calculated as overhead for an arithmetic

RISC instruction		Overhead	ALU	125 pJ
Load/Store	D-$	Overhead	ALU	150 pJ
SP floating point			+	15–20 pJ
32-bit addition			+	7 pJ
8-bit addition			+	0.2–0.5 pJ

Figure 7.1 Energy costs in pico-Joules for a 90 nm process to fetch instructions or access a data cache compared to the energy cost of arithmetic operations (Qadeer et al., 2015).

instruction. Given this overhead, minor twists to existing cores may get us 5%−10% improvements, but if we want large improvements while offering programmability, we need to increase the number of arithmetic operations per instruction from one to hundreds. To achieve that level of efficiency, we need a drastic change in computer architecture from general-purpose cores to *domain-specific architectures (DSAs)*.

Thus, just as the field switched from uniprocessors to multiprocessors in the early 2000s out of necessity, desperation is the reason architects are now working on DSAs. The new normal is that a computer will consist of standard processors to run conventional large programs such as operating systems along with domain-specific processors that do only a narrow range of tasks, but they do them extremely well. Thus such computers will be much more heterogeneous than the homogeneous multicore chips of the past. Indeed, smartphone SOCs already embrace heterogeneity with accelerators that improve energy efficiency of signal processing, graphics generation, image conversion, and so on.

Part of the argument is that the preceding architecture innovations from the past few decades that leveraged Moore's law (caches, out-of-order execution, etc.) may not be a good match to some domains—especially in terms of energy usage—so their resources can be recycled to make the chip a better match to the domain. For example, caches are excellent for general-purpose architectures but not necessarily for DSAs; for applications with easily predictable memory access patterns or huge datasets like video that have little data reuse, multilevel caches are overkill, hoarding data that might not be reused, wasting area and energy. Therefore the promise of DSAs is both improved silicon efficiency and better energy efficiency, with the latter typically being the more important attribute today.

Architects probably won't create a DSA for a large C++ program like a compiler, as found in the SPEC2017 benchmark. Domain-specific algorithms are almost always for small compute-intensive kernels of larger systems, such as for object recognition or speech understanding. DSAs should target the subset and not plan to run the entire program. In addition, changing the code of the benchmark is no longer breaking the rules; it is a perfectly valid source of speedup for DSAs, in part because the code being accelerated is small. Consequently, if they are going to make useful contributions, architects interested in DSA must now shed their blinders and learn application domains and algorithms.

In addition to needing to expand their areas of expertise, a challenge for domain-specific architects is to find a target whose demand is large enough to justify allocating dedicated silicon on an SOC or even a custom chip. The *nonrecurring engineering (NRE)* costs of a custom chip and supporting software are amortized over the number of chips manufactured, so it is unlikely to make economic sense if you need only 10,000 chips (see Figure 1.17 in Chapter 1).

One way to accommodate smaller volume applications is to use reconfigurable chips such as Field Programmable Gate Arrays (FPGAs) because they have lower NRE than custom chips *and* because several different applications may be able to reuse the same reconfigurable hardware to amortize its costs. However, since the hardware is much less efficient than custom chips, FPGAs gains are more modest.

Another DSA challenge is porting software. Familiar programming environments like the C++ programming language and compiler are rarely the right vehicles for a DSA.

The rest of this chapter provides five guidelines for the design of DSAs and then a tutorial on our example domain, which is *deep neural networks* (*DNNs*). We chose DNNs because they are revolutionizing many areas of computing in the past decade. Unlike some hardware targets, DNNs are applicable to a wide range of problems, so we can reuse a DNN-specific architecture for solutions in speech, vision, language, translation, search ranking, and many more areas.

We follow with six examples of DSAs: five custom chips for the data center that accelerate DNNs and one DSA that accelerates DNNs for *personal mobile devices* (*PMDs*). Next is the cost-performance of the DSAs using DNN benchmarks, concluding with recognition of the ongoing renaissance in computer architecture.

7.2 Guidelines for DSAs

Here are five principles that generally guided the designs of the six DSAs we'll see in Sections 7.4–7.7. Not only do these five guidelines lead to increased area and energy efficiency, but they also provide two valuable bonus effects. First, they lead to simpler designs, which reduce the NRE of DSAs (see the fallacy in Section 7.11). Second, for user-facing applications that are commonplace with DSAs, accelerators that follow these principles are a better match to the 99th-percentile response-time deadlines than the time-varying performance optimizations of traditional processors, as we will see in Section 7.10. Figure 7.2 shows how the six DSAs followed these guidelines.

1. *Use dedicated memories to minimize the distance over which data is moved.* The many levels of caches in general-purpose microprocessors use a great deal of area and energy trying to move data optimally for a program. For example, a two-way set associative cache uses 2.5 times as much energy as an equivalent software-controlled scratchpad memory. By definition, the compiler writers

Guideline	Google TPUv4 lite	Google TPUv4	NVIDIA A100	NVIDIA T4	Graphcore IPU Bow	Samsung NPUv2
Domain	DNN inference	DNN training & inference	DNN training & inference	DNN inference	DNN training & inference	DNN inference
Design target	Data center ASIC	Data center ASIC	Data center ASIC	Data center ASIC	Data center ASIC	PMD SOC
1. Dedicated memories	16 MiB VMEM, 128 MiB CMEM	32 MiB VMEM, 128 MiB CMEM	27 MiB multi-thread register file, 40 MiB L2 cache	10 MiB multi-thread register file, 18 MiB L2 cache	897 MiB In-processor memory	2 MiB Tightly Coupled Memory
2. Larger arithmetic unit	65,536 Multiply-accumulators	131,072 Multiply-accumulators	110,592 Multiply-accumulators	20,480 Multiply-accumulators	94,208 Multiply-accumulators	8192 Multiply-accumulators
3. Easy parallelism	Single-threaded, VLIW, SIMD, in-order	Single-threaded, VLIW, SIMD, in-order	Multithreaded, SIMD	Multithreaded, SIMD	Multithreaded, SIMD	Single-threaded, SIMD
4. Smaller data size	8-bit int, bf16	8-bit int, bf16	1-bit, 4-bit, 8-bit int, fp16, bf16, tf32	4-bit int, 8-bit int, fp16	fp16	4-bit int, 8-bit int, 16-bit int, fp16
5. Domain-specific lang.	JAX, PyTorch, TensorFlow	JAX, PyTorch, TensorFlow	JAX, PyTorch, TensorFlow	JAX, PyTorch, TensorFlow	JAX, PyTorch, TensorFlow	TensorFlow, TensorFlow Lite

Figure 7.2 The six DSAs in this chapter and how closely they followed the five guidelines. Note that any chip that can do training can also do inference. Both Google and NVIDIA reduced the cost of their inference offering by reducing the number of cores and reducing the cost of external memory as compared to their training versions.

and programmers of DSAs understand their domain, so there is no need for the hardware to try to move data for them. Instead, data movement is reduced with software-controlled memories that are dedicated to and tailored for specific functions within the domain.

2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.*
   As Section 7.1 describes, architects turned the bounty from Moore's law into resource-intensive optimizations for CPUs and GPUs (out-of-order execution, multithreading, multiprocessing, prefetching, address coalescing, etc.). Given the superior understanding of running programs in these narrower domains, these resources are much better spent on more processing units or larger on-chip memory.
3. *Use the easiest form of parallelism that matches the domain.*
   Target domains for DSAs almost always have inherent parallelism. The key decisions for a DSA are how to take advantage of that parallelism and how to expose it to the software. Design the DSA around the natural granularity of the parallelism of the domain and expose that parallelism simply in the programming model. For example, with respect to data-level parallelism, if SIMD

works in the domain, it's certainly easier for the programmer and the compiler writer than MIMD. Similarly, if Very Long Instruction Word (VLIW) can express the instruction-level parallelism for the domain, the design can be smaller and more energy-efficient than speculative out-of-order execution. Moreover, both VLIW and SIMD approaches improve the ratio of computation per instruction access.

4. *Reduce data size and type to the simplest needed for the domain.*
   As we will see, applications in many domains are typically memory bound, so you can increase the effective memory bandwidth and on-chip memory utilization by using narrower data types. Narrower and simpler data also lets you pack more arithmetic units into the same chip area and have more energy-efficient arithmetic operations.
5. *Use a domain-specific programming language to write code for a DSA.*
   As Section 7.1 mentions, a classic challenge for DSAs is getting applications to run on your novel architecture. A long-standing fallacy is the assumption that your new computer is so attractive that programmers will rewrite their code just for your hardware. Fortunately, domain-specific programming languages and libraries were becoming popular even before architects were forced to switch their attention to DSAs. Examples are JAX, PyTorch, and TensorFlow for DNNs (Paszke et al., 2019; Abadi et al., 2016) and SQL for database systems. They make porting applications to your DSA much more feasible. As previously mentioned, only a small, compute-intensive portion of the application needs to run on the DSA in some domains, which also simplifies porting.

DSAs introduce many new terms, mostly from the new domains but also from novel architecture mechanisms not seen in conventional processors. As we did in Chapter 4, Figure 7.3 lists the new acronyms, terms, and short explanations to aid the reader.

There are many domains that already have DSAs (like GPUs for graphics) and others that are worth investigating further: data analytics, image processing, security, and privacy. However, the driving example of this chapter is artificial intelligence.

7.3 Example Domain: Deep Neural Networks

Artificial intelligence (AI) is not only the next big wave in computing—it's the next major turning point in human history… the Intelligence Revolution will be driven by data, neural networks, and computing power. Intel is committed to AI [thus]… we've added a set of leading-edge accelerants required for the growth and widespread adoption of AI.

Brian Krzanich,
Intel CEO (2016)

Area	Term	Acronym	Short explanation
General	Domain-specific architectures	DSA	A special-purpose processor designed for a particular domain. It relies on other processors to handle processing outside that domain
	Intellectual property block	IP	A portable design block that can be integrated into an SOC. They enable a marketplace where organizations offer IP blocks to others who compose them into SOCs
	System on a chip	SOC	A chip that integrates all the components of a computer; commonly found in PMDs
Deep neural networks	Activation	—	Result of "activating" the artificial neuron; the output of the nonlinear functions
	Batch	—	A collection of datasets processed together to lower the cost of fetching weights
	Convolutional neural network	CNN	A DNN that takes as inputs a set of nonlinear functions of spatially nearby regions of outputs from the prior layer, which are multiplied by the weights
	Deep neural network	DNN	A sequence of layers that are collections of artificial neurons, which consist of a nonlinear function applied to products of weights times the outputs of the prior layer
	Inference	—	The production phase of DNNs; also called *serving*
	MultiLayer perceptron	MLP	A DNN that takes as inputs a set of nonlinear functions of all outputs from the prior layer multiplied by the weights. These layers are called *fully connected*
	Rectified linear unit	*ReLU*	A nonlinear function that performs `f(x)=max(x,0)`. Other popular nonlinear functions are sigmoid and hyperbolic tangent (tanh)
	Recurrent neural network	RNN	A DNN whose inputs are from the prior layer *and* the previous state
	Training	—	The development phase of DNNs; also called *learning*
	Transformer	—	A DNN that relies on an attention mechanism to provide the context for positions in the input sequence. Like RNNs, it is popular for natural language translation and text summarization, but unlike RNNs it processes the input all at once rather than sequentially. This change means the operations can occur in parallel, so it processes much larger data
	Weights	—	The values learned during training that are applied to inputs; also called *parameters*
	Large Language Model	LLM	Also known as Frontier models, LLMs are based on the decode phase of the Transformer model and typically have billions to trillions of parameter.

Figure 7.3 A handy guide to DSA terms used in this chapter.

Artificial intelligence (AI) has made a dramatic comeback since the turn of the century. Instead of *building* artificial intelligence as a large set of logical rules, the focus switched to *machine learning* (ML) from example data as the path to artificial intelligence. It turned out that it was harder to program a computer to be intelligent than to program a computer to *learn* to be intelligent. However, the amount of data needed to learn was much greater than thought. The warehouse-scale computers (WSCs) of this century, which harvest and store petabytes of information

found on the Internet from the billions of users and their smartphones, supply ample data. We also underestimated the amount of computation needed to learn from the massive data, but GPUs—which have excellent single-precision floating-point cost-performance—embedded in the thousands of servers of WSCs deliver sufficient computing.

One part of machine learning, called DNNs, has been the AI and ML star for the past decade. Example DNN breakthroughs are in language translation, which DNNs improved more in a single leap than all the advances from the prior decade (Tung, 2016; Lewis-Kraus, 2016); the switch to DNNs reduced the error rate in an image recognition competition from 26%–3.5% over only 5 years (Krizhevsky et al., 2012; Szegedy et al., 2015; He et al., 2016); and in 2016, DNNs enabled a computer program for the first time to beat a human champion at Go (Silver et al., 2016). In 2022, a DNN predicted the three-dimensional shape of all 200 M known proteins from their one-dimensional amino acid sequences, a problem that had vexed biologists for decades. *Science* magazine honored the achievement as the biggest breakthrough of the year (Jumper et al., 2021), and they won a Nobel Prize in 2024. Significant new DNN results appear nearly every month, so some label this era the "golden decade of deep learning" (Dean, 2022). Although many of these examples run in the cloud, they have also enabled applications like Google Translate on smartphones, as described in Chapter 1.

Readers interested in learning more about DNNs than found in this section should download and try the tutorials for JAX and PyTorch or, for the less adventurous, consult a free online textbook on DNNs (Nielsen, 2016). The Google crash course on ML/AI is also great!

The Neurons of DNNs

DNNs were inspired by the brain's neuron. The artificial neuron used for neural networks simply computes the sum over a set of products of *weights* or *parameters* and data values that is then put through a nonlinear function to determine its output. As we will see, each artificial neuron has a large fan-in and a large fan-out.

For an image-processing DNN, the input data would be the pixels of a photo, with the pixel values multiplied by the weights. Although many nonlinear functions have been tried, a popular one today is simply `f(x) = max(x,0)`, which returns 0 if the `x` is negative or the original value if positive or zero. (This simple function goes by the complicated name *rectified linear unit* or *ReLU*). The output of a nonlinear function is called an *activation*, in that it is the output of the artificial neuron that has been "activated."

A cluster of artificial neurons might process different portions of the input, and the output of that cluster becomes the input to the next layer of artificial neurons. The layers between the input layer and the output layer are called *hidden layers*. For image processing, you can think of each layer as looking for different types of features, going from lower-level ones like edges and angles to higher-level ones like eyes and ears. If the image-processing application was trying to decide if the image contained a dog, the output of the last layer could be a

DNN model (type and # layers)	# parameters	Size and type of data set	Training platform and time
BERT-large (24 layers of Transformer encoders) [Devlin et al., 2018]	340 M	BookCorpus (800 M words) and English Wikipedia (2500 M words)	64 TPU v2 * 4 days (~1500 TPU v2 hours)
Stable Diffusion v2 Base (latent diffusion model with UNet and cross-attention, OpenCLIP-ViT/H text encoder) [Rombach et al., 2022]	860 M UNet and 123 M text encoder	Filtered LAION-5B (5.85 billion image-text pairs)	256 A100 GPUs * 33 days (~200,000 A100 hours)
Switch-Transformer Base (12 layers of Transformer encoder-decoders with mixture-of-experts) [Fedus et al., 2022]	8000 M	Colossal Clean Crawled Corpus 356 billion tokens	16 TPU v3 * 2.5–3 days (~70 TPU v3 hours)
GPT-3 (96 layers of Transformer decoders) [Brown et al., 2020]	175,000 M	Common Crawl 410 billion tokens, WebText2 19 billion tokens, Books1 12 billion, Books2 55 billion, Wikipedia 3 billion	Estimated 1024 A100 GPUs * 34 days (~840,000 A100 hours)

Figure 7.4 Example DNN sizes, showing name, number of parameters, training data set, and time to train and hardware used. To reduce the effort of benchmarking, MLPerf Training version 3.1 uses some of the same size models with much smaller data sets. For example, Stable Diffusion v2 with only 0.4B image-text pairs instead of 5.9B in the figure takes less than 1 hour on eight of the newer H100 GPUs, or about 6 H100 hours in total, versus 200,000 A100 hours. Production training production of Stable Diffusion takes 10,000X longer than MLPerf training even if a system based on H100s was 3.3X faster than using A100s.

probability number between 0 and 1 or perhaps a list of probabilities corresponding to a list of dog breeds.

The number of layers gave deep neural networks their name. The original lack of data and computing horsepower kept most neural networks relatively shallow, perhaps two hidden layers. In 2024 some DNNs have scores of layers using hundreds of billions of weights. Figure 7.4 shows the number of layers, parameters, and training times for a variety of DNNs.

Training Versus Inference

The preceding discussion concerns DNNs that are in production. DNN development starts by defining the neural network architecture, picking the number and type of layers, the dimensions of each layer, and the size of the data. Although experts may develop new neural network architectures, most practitioners will choose among the many existing designs (e.g., Figure 7.4) that have been shown to perform well on problems similar to theirs.

Once the neural architecture has been selected, the next step is to learn the weights associated with each edge in the neural network graph. The weights determine the model's behavior. Depending on the choice of neural architecture, there can be anywhere from thousands to hundreds of billions of weights in a

single model. Training is the costly process of tuning these weights so that the DNN approximates the complex function (e.g., mapping from pictures to the objects in that picture) described by the training data.

This development phase is universally called *training* or *learning*, whereas the production phase has many names: *inference*, *serving, prediction*, *scoring*, or *implementation*. An educational analogy would be that training is like going to college to learn a field and serving is like the graduate performing tasks that depend on that learning.

Most DNNs use *supervised learning* in that they are given a training set to learn from where the data is preprocessed to have the correct labels. Thus, in the ImageNet DNN competition (Russakovsky et al., 2015), the training set consists of 1.2 million photos, and each photo has been labeled as one of 1000 categories. Several of these categories are quite detailed, such as specific breeds of dogs and cats. The winner is determined by evaluating a separate secret set of 50,000 photos to see which DNN has the lowest error rate.

Setting the weights is an iterative process that goes *backward* through the neural network using the training set. This process is called *backpropagation*. For example, because you know the breed of a dog in the image training set, you see what your DNN says about the image, and then you adjust the weights to improve the answer. Amazingly, the weights at the start of the training process should be set to random data, and you just keep iterating until you're satisfied with the DNN accuracy from the training set.

For the mathematically inclined, the goal of learning is to find a function that maps the inputs to the correct outputs over the multilayer neural network architecture. Backpropagation means "back propagation of errors." It calculates a gradient over all the weights as input to an optimization algorithm that tries to minimize the errors by updating the weights. The most popular optimization algorithm for DNNs is *stochastic gradient descent*. It adjusts the weights proportionally to maximize the descent of the gradient obtained from backpropagation. Readers interested in learning more should see Nielsen (2016).

Training can take weeks or months of computation on thousands of chips. The inference phase is often below 100 ms per data sample, which is a million times quicker. Although training takes much longer than a single inference, the total compute time for inference is a product of the number of customers of the DNN and how frequently they invoke it.

After training, you deploy your DNN, hoping that your training set is representative of the real world, and that your DNN will be so popular that your users will spend much more time employing it than you've put into developing it!

There are tasks that don't have training datasets, such as when trying to predict the future of some real-world event. Although we won't cover it here, *reinforcement learning* (*RL*) is a popular algorithm for such learning. Instead of a training set to learn from, RL acts on the real world and then gets a signal from a reward function, depending on whether that action made the situation better or worse.

DNN Model	TPU v1 July 2016 (Inference)	TPU v3 April 2019 (Training & Inference)	TPU v4 lite February 2020 (Inference)	TPU v4 October 2022 (Training)
MLP/DLRM	61%	27%	25%	24%
RNN	29%	21%	29%	2%
CNN	5%	24%	18%	12%
Transformer	–	21%	28%	57%
(BERT)			*(28%)*	*(26%)*
(Large Language Model)				*(31%)*

Figure 7.5 Popularity of production DNNs at Google over time. The paper introducing Transformer was published in 2017, and 2 years later, it was 21% of the workload. BERT was published in 2018 and the GPT-3 paper that inspired the work on large transformer models appeared in 2020, and by 2022, they represented nearly 60% of the workload on TPU v4.

Although it's hard to imagine a faster-changing field, four types of DNNs reign as most popular in 2024: *Multilayer perceptrons* (*MLPs*), *convolutional neural networks* (*CNNs*), *recurrent neural networks* (*RNNs*), and *transformer* models. They are all examples of supervised learning, relying on training sets. To give an idea of how fast models change, Figure 7.5 shows how dramatically the production DNN workload at Google changed over only 6 years.

Multilayer Perceptron

MLPs were the original DNNs. Each new layer is a set of nonlinear functions F of weighted sum of all outputs from a prior one $y_n = F(W \times y_{n-1})$. The weighted sum consists of a vector-matrix multiply of the outputs with the weights (see Figure 7.6). Such a layer is called *fully connected* because each output neuron result depends on *all* input neurons of the prior layer.

We can calculate the number of neurons, operations, and weights per layer for each of the DNN types. The easiest is MLP because it is just a vector-matrix multiply of the input vector times the weights array. Here are the parameters and the equations to determine weights and operations for inference (we count multiply and add as two operations):

- Dim[*i*]: Dimension of the output vector, which is the number of neurons
- Dim[*i*– 1]: Dimension of the input vector
- Number of weights: Dim[*i*– 1] × Dim[*i*]
- Operations: 2 × Number of weights
- Operations/Weight: 2

This final term is the *operational intensity* from the Roofline model discussed in Chapter 4. We use operations per *weight* because there can be billions of

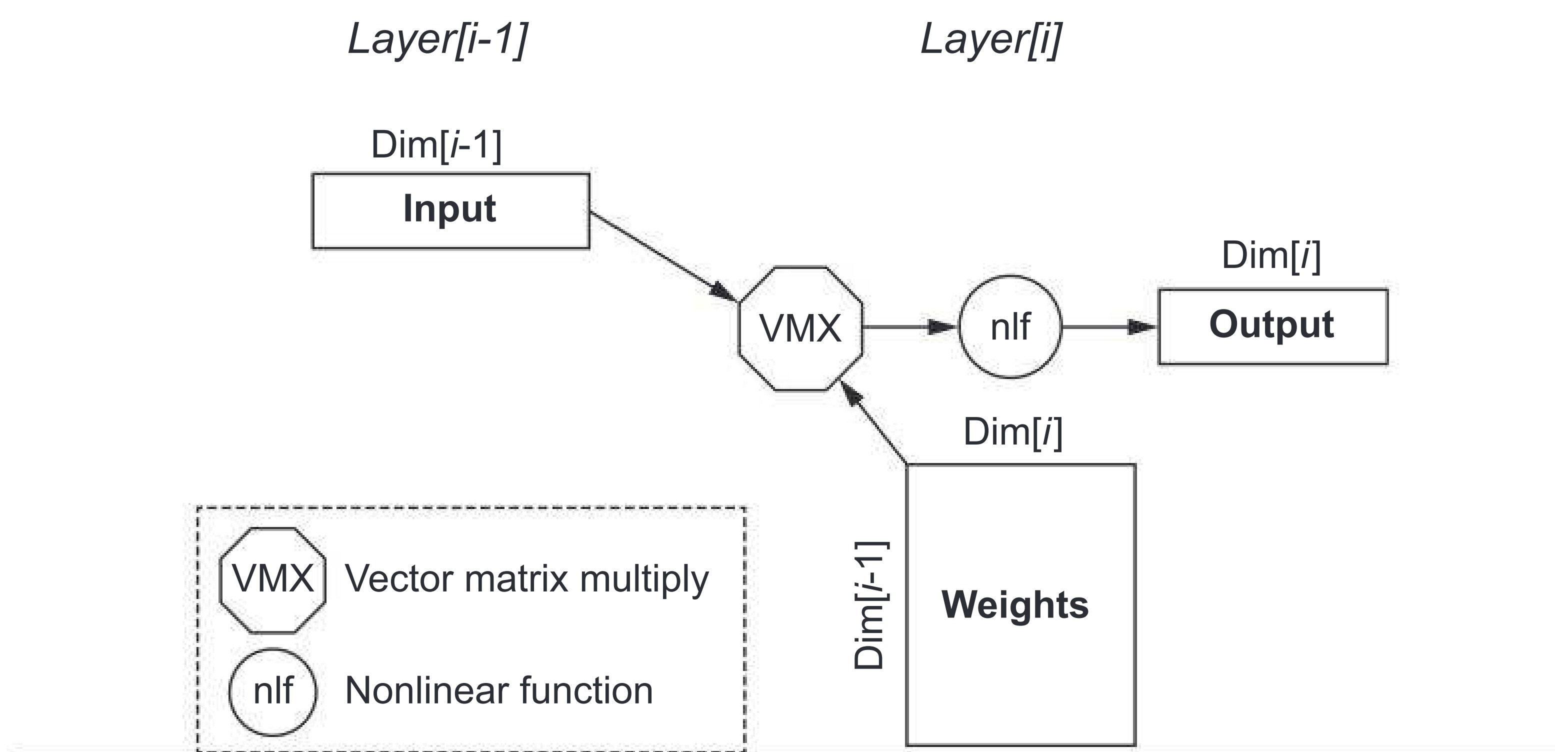

Figure 7.6 MLP showing the input Layer[i − 1] on the left and the output Layer[i] on the right. ReLU is a popular nonlinear function for MLPs. The dimensions of the input and output layers are often different. Such a layer is called fully connected because it depends on all the inputs from the prior layer, even if many of them are zeros. One study suggested that 44% were zeros, which presumably is in part because ReLU turns negative numbers into zeros.

weights, which usually don't fit on the chip. For example, the dimensions of one stage of an example MLP has Dim[$i-1$] = 4096 and Dim[i] = 2048, so for that layer, the number of neurons is 2048, number of weights is 8,388,608, the number of operations is 16,777,216, and the operational intensity is 2. As we recall from the Roofline model, low operational intensity makes it harder to deliver high performance.

Convolutional Neural Network

CNNs are widely used for computer vision applications. As images have a two-dimensional structure, neighboring pixels are the natural place to look to find relationships. CNNs take as inputs a set of nonlinear functions from spatially nearby regions of outputs from the prior layer and then multiply by the weights, which reuses the weights many times.

The idea behind CNNs is that each layer raises the level of abstraction of the image. For example, the first layer might identify only horizontal lines and vertical lines. The second layer might combine them to identify corners. The next step might be rectangles and circles. The following layer could use that input to detect portions of a dog, like eyes or ears. The higher layers would be trying to identify characteristics of different breeds of dogs.

Each neural layer produces a set of two-dimensional *feature maps*, where each cell of the two-dimensional feature map is trying to identify one feature in the corresponding area of the input.

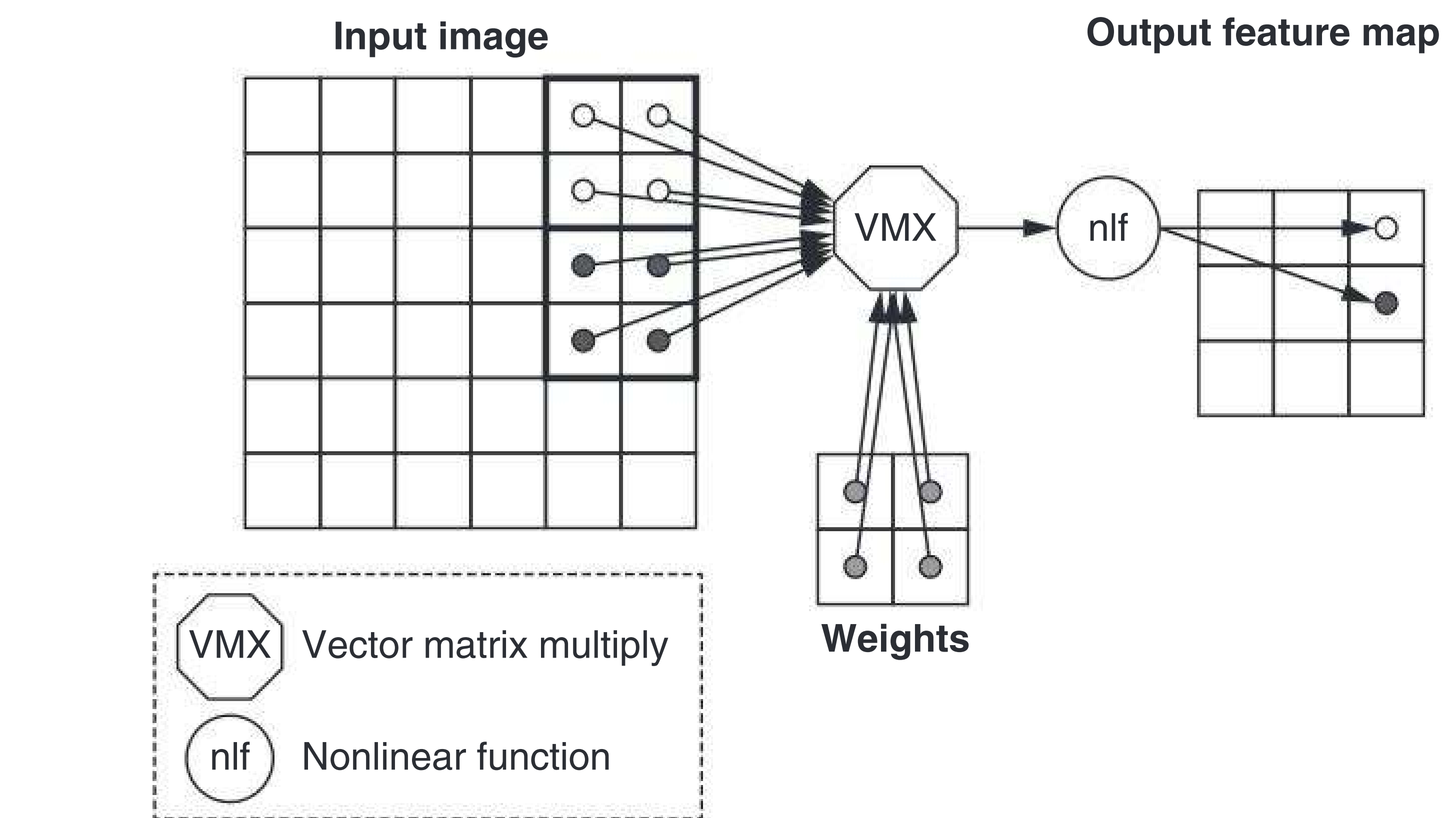

Figure 7.7 Simplified first step of a CNN. In this example every group of four pixels of the input image are multiplied by the same four weights to create the cells of the output feature map. The pattern depicted shows a stride of two between the groups of input pixels, but other strides are possible. To relate this figure to MLP, you can think of each 2×2 convolution as a tiny, fully connected operation to produce one point of the output feature map. Figure 7.7 shows how multiple feature maps turn the points into a vector in the third dimension.

Figure 7.7 shows the starting point where a 2×2 stencil computation from the input image creates the elements of the first feature map. A *stencil computation* uses neighboring cells in a fixed pattern to update all the elements of an array. The number of output feature maps will depend on how many different features you are trying to capture from the image and the stride used to apply the stencil.

The process is actually more complicated because the image is usually not just a single, flat, two-dimensional layer. Typically, a color image will have three levels for red, green, and blue. For example, a 2×2 stencil will access 12 elements: 2×2 of red pixels, 2×2 of green pixels, and 2×2 of blue pixels. In this case you need 12 weights per output feature map for a 2×2 stencil on three input levels of an image.

Figure 7.8 shows the general case of an arbitrary number of input and output feature maps, which occurs after that first layer. The calculation is a three-dimensional stencil over all the input feature maps with a set of weights to produce one output feature map.

For the mathematically oriented, if the number of input feature maps and output feature maps both equal 1 and the stride is 1, then a single layer of a two-dimensional CNN is the same calculation as a two-dimensional discrete convolution.

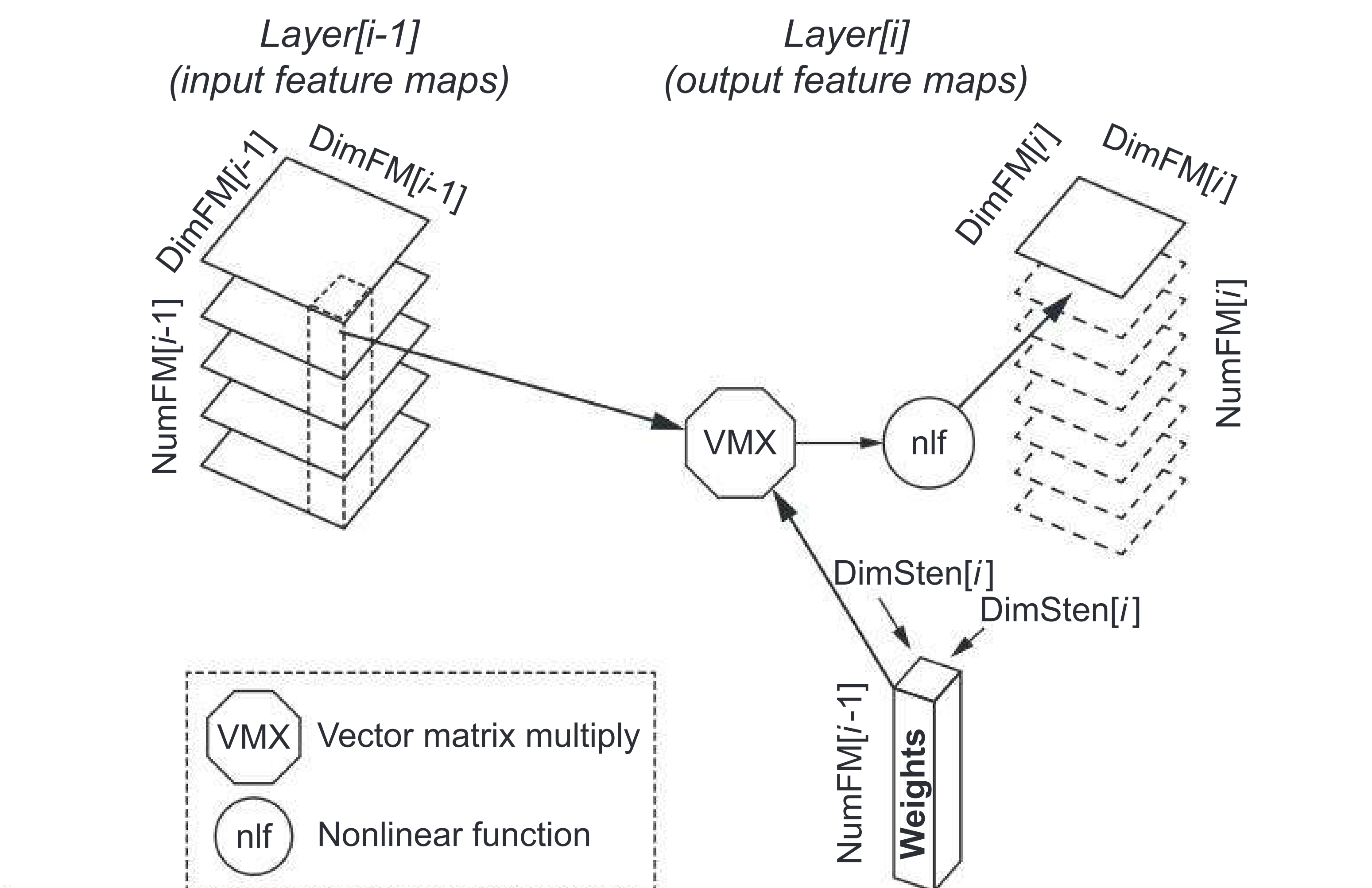

Figure 7.8 CNN general step showing input feature maps of Layer[*i* − 1] on the left, the output feature maps of Layer[*i*] on the right, and a three-dimensional stencil over input feature maps to produce a single output feature map. Each output feature map has its own unique set of weights, and the vector-matrix multiply happens for every one. The dotted lines show future output feature maps in this figure. As this figure illustrates, the dimensions and number of the input and output feature maps are often different. As with MLPs, ReLU is a popular nonlinear function for CNNs.

As we see in Figure 7.8, CNNs are more complicated than MLPs. Here are the parameters and the equations to calculate the weights and operations:

- 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*] $\times$ DimFM[*i*]^2
- Number of weights per output Feature Map: NumFM[*i*− 1] $\times$ DimSten[*i*]^2
- Total number of weights per layer: NumFM[*i*] $\times$ Number of weights per output Feature Map
- Number of operations per output Feature Map: 2 $\times$ DimFM[*i*]^2 $\times$ Number of weights per output Feature Map

- Total number of operations per layer: NumFM[i] × Number of operations per output Feature Map = 2 × DimFM[i]^2 × NumFM[i] × Number of weights per output Feature Map = 2 × DimFM[i]^2 × Total number of weights per layer
- Operations/Weight: 2 × DimFM[i]^2

A CNN might have a layer with DimFM[$i-1$] = 28, DimFM[i] = 14, DimSten[i] = 3, NumFM[$i-1$] = 64 (number of input feature maps), and NumFM[i] = 128 (number of output feature maps). That layer has 25,088 neurons and 73,728 weights, does 28,901,376 operations, and has an operational intensity of 392. As our example indicates, CNN layers generally have fewer weights and greater operational intensity than the fully connected layers found in MLPs.

Recurrent Neural Network

The third type of DNN is RNNs, which were popular for speech recognition or language translation. RNNs add the ability to explicitly model sequential inputs by adding state to the DNN model so that RNNs can remember facts. It's analogous to the difference in hardware between combinational logic and a state machine. For example, you might learn the gender of the person, which you would want to pass along to remember later when translating words. Each layer of an RNN is a collection of weighted sums of inputs from the prior layer and the previous state. The weights are reused across time steps.

One example of an RNN is the *long short-term memory* (*LSTM*) model, which mitigates a problem that previous RNNs had with their inability to remember important long-term information. While LSTMs were once popular, they've largely been replaced by Transformers.

Transformer Neural Network

Transformer models appeared about the same time as the last edition of this book, and they have already become an extremely popular and important DNN model. Transformers rely on *self-attention*, in that it weighs each part of the input data differently. Like RNNs, it is popular for natural language translation and text summarization, but unlike RNNs it processes the input all at once rather than sequentially. This change means the operations can occur in parallel, which in turn means Transformer models can process much larger data sets since they can process inputs faster.

They rely on the attention mechanism to provide the context for positions in the input sequence. The attention layer can access all previous states in any point along the input sequence. Indeed, the title of the article introducing Transformers is "Attention is all you need" (Vaswani, 2017). Figures 7.9 and 7.10 show the architecture of a Transformer model.

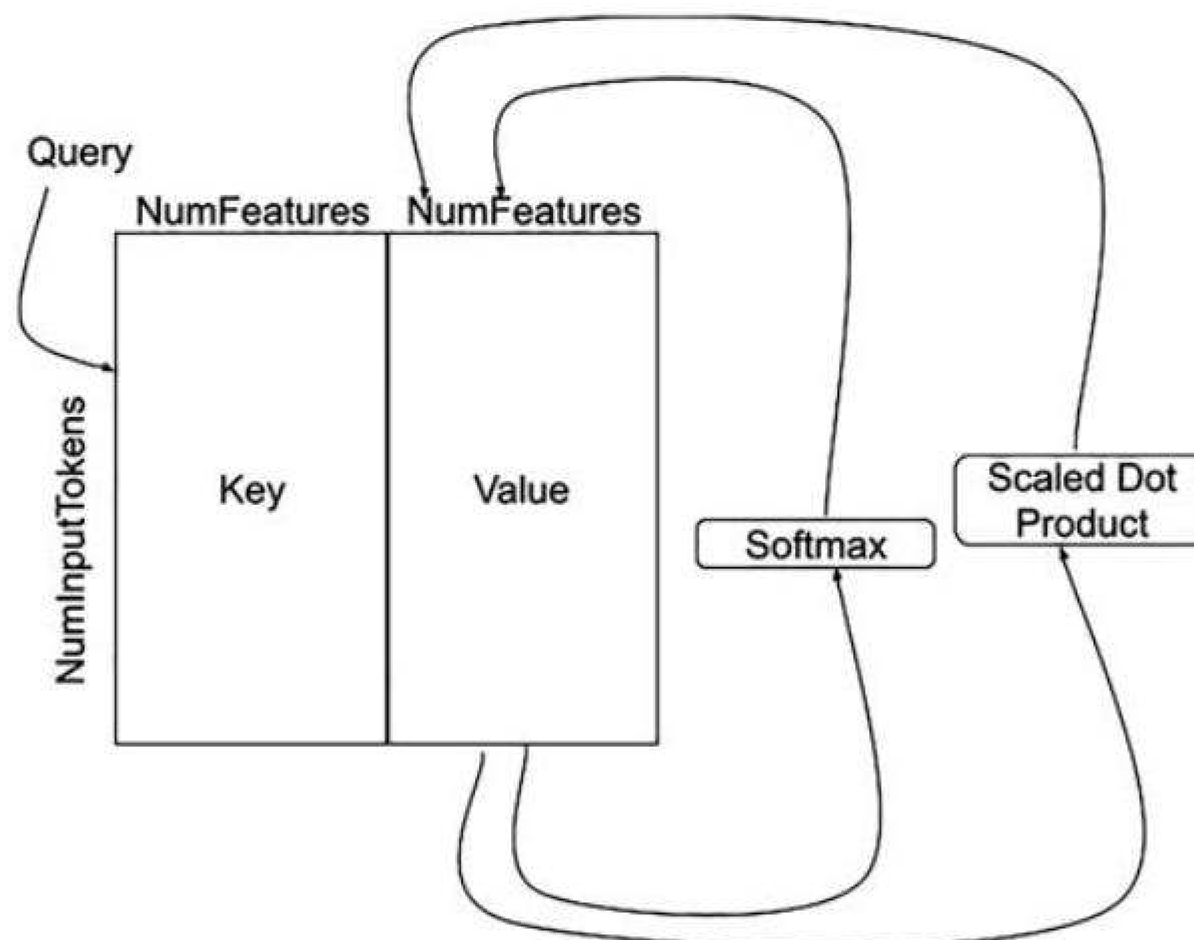

Figure 7.9 A portion of the architecture of the Transformer model (Vaswani, 2017). *Attention* is based on a key-value table. The width of the key and value is the number of features and the length is the number of input tokens. Attention makes multiple passes over the table. *SoftMax* normalizes the output of a DNN to a probability distribution over the predicted output classes. It maps the inputs to values between -1 and +1 that sum to 1. The scaled dot-product enhances some parts of the input while diminishing other parts, such as longer-term factors versus short-term factors.

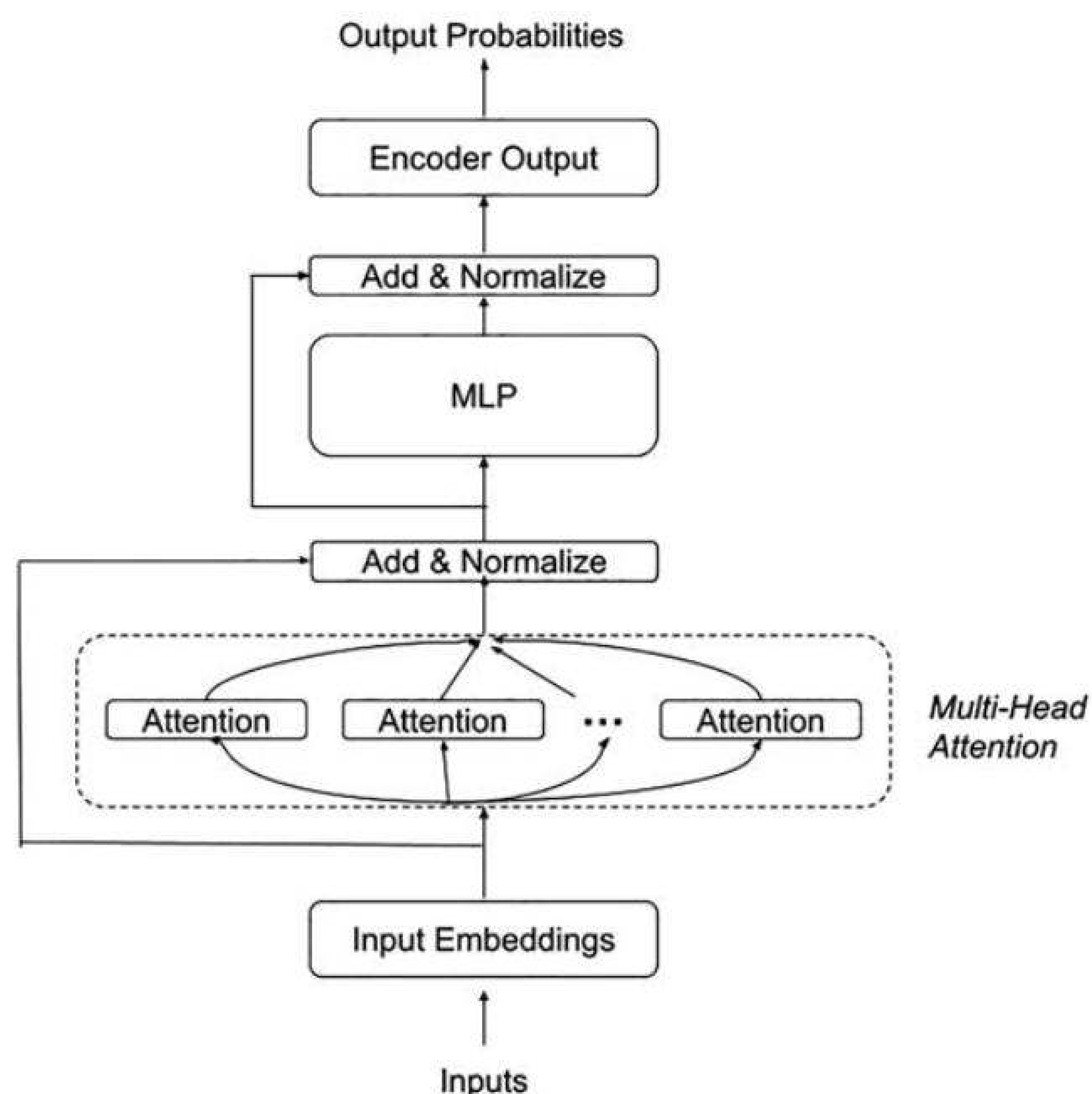

Figure 7.10 The full Transformer model. *Multihead Attention* runs through the attention mechanism in Figure 7.9 several times in parallel. The collective outputs are concatenated and then transformed into the expected dimension. The next step is to send it to an MLP.

One popular version is BERT (Bidirectional Encoder Representations from Transformers) that refines the original Transformer by having two models that are pretrained for natural language processing using the BookCorpus and English Wikipedia (Devlin et al., 2018). Another popular Transformer model is GPT-3 (Generative Pretrained Transformer 3), which is trained to predict what the next token is (called *generative pretraining*). GPT-3 stands out for its large size, with 175B parameters. Its predecessor, GPT-2, has 1.5B parameters. GPT-3 has inspired a new class of models known collectively as *large language models (LLM)* or Frontier Models. Since 2021, exploring the potential of LLMs has received considerable attention from the ML community including industry. Today Transformer models have largely replaced RNNs for natural language processing (see Figure 7.5).

Batches

Because DNNs can have many weights, a performance optimization is to reuse the weights once they have been fetched from memory across a set of inputs, thereby increasing effective operational intensity. For example, an image-processing DNN might work on a set of 32 images at a time to reduce the effective cost of fetching weights by a factor of 32. Such collections are called *batches* or *minibatches*. In addition to improving the performance of inference, backpropagation needs a batch of examples instead of one at a time to train well.

Looking at an MLP in Figure 7.6, a batch can be seen as a sequence of input row vectors, which you can think of as a matrix with a height dimension that matches the batch size. Computing them as matrices instead of sequentially as independent vectors improves computing efficiency.

Quantization

Numerical precision is less important for DNNs than many applications. For example, there is no need for double-precision floating-point arithmetic, which is the standard bearer of high-performance computing. In 2024 DNN many accelerators support floating point data that are 8 or 16 bits wide. Recent DNN DSAs offer 6-bit and even 4-bit floating point data. Nor do you need the full accuracy of the IEEE 754 floating-point standard, which aims to be accurate within one-half of a unit in the last place of the floating-point significand.

To take advantage of the flexibility in numerical precision, developers often use fixed point instead of floating point for the inference phase. (Training is conventionally done in floating-point arithmetic.) This conversion from floating point to integer is called *quantization*, and such a transformed application is said to be *quantized* (Vanhoucke et al., 2011). The fixed-point data width is 1, 2, 4, 8, or 16 bits, with the standard multiply-add operation accumulating at twice the width of the multiplies. This transformation typically occurs after training, and it can reduce DNN accuracy by a few percentage points (Bhattacharya and Lane, 2016).

These narrower data help DNNs because they increase operational intensity *and* because the reduce the memory capacity of weights, which are often limited as we shall see.

Summary of DNNs

Even this quick overview suggests that DSAs for DNNs will need to perform at least these matrix-oriented operations well: vector-matrix multiply, matrix-matrix multiply, and stencil computations. They will also need support for the nonlinear functions, which include at a minimum ReLU, Sigmoid, and tanh. These modest requirements still leave open a very large design space, which the next sections explore.

7.4 Google's Tensor Processing Unit v4 and v4 lite, Data Center DNN Accelerators

Starting as far back as 2006, Google engineers had discussions about deploying GPUs, FPGAs, or custom ASICs in their data centers. They concluded that the few applications that could run on special hardware could be done virtually for free using the excess capacity of the large data centers, and it's hard to improve on free. The conversation changed in 2013 when it was projected that if people used voice search for 3 minutes a day using speech recognition DNNs, it would have required Google's data centers to double in order to meet computation demands. That would take a long time and be very expensive to satisfy with conventional CPUs.

Google then started a high-priority project to quickly produce a custom ASIC for inference (and bought off-the-shelf GPUs for training). The goal was to improve cost-performance by 10X over GPUs. Tensor Processing Unit (TPU) v1 delivered on its goal, offering 80X the performance per Watt of contemporary CPUs and 30X that of contemporary GPUs. Given the rise in importance of ML, CPUs and GPUs now include optimizations specifically for ML, so the gap between them and DNN DSAs is now smaller.

The first TPU generation was a single-core design for inference, and the second and third generations were dual-core designs for both training and inference. They are programmed using the JAX, PyTorch, and TensorFlow frameworks, which were designed for DNNs, using the custom XLA compiler.

For the fourth generation[1], Google realized that it could get two chips from a single design by using a *single*-core chip for inference (like TPU v1) and a *dual*-core chip for training (like TPU v2 and TPU v3), as long as both chips used the same core. Figure 7.11 lists the features of both TPUs covered in this section. The same team developed both chips in a unified codebase.

[1] This section is based on the paper "Ten lessons from three generations shaped Google's TPU v4" (Jouppi et al., 2021), of which one of your book authors was a coauthor.

	Google TPU v4 Lite	Google TPU v4
Production deployment	2020	2020
Peak TFLOPS	138 (bf16 or 8b int)	275 (bf16 or int8)
DNN Target	Inference only	Training or Inference
Clock Rate	1050 MHz	1050 MHz
Tech. node, Die size	7 nm, <400 mm^2	7 nm, <700 mm^2
Transistor count	16 billion	22 billion
Chips per CPU host	8	4
Thermal Design Power	175 W	N.A.
Idle, min/mean/max power	55 W, N.A.	90 W, 121/170/192 W
Inter-Chip Interconnect	2 @ 50 GB/s	6 links @ 50 GB/s
Largest scale configuration	8 chips	4096 chips
Cloud Availability	Google	Google
Processor Style	Single Instruction 2D Data	Single Instruction 2D Data
Arithmetic Units/Core	65,536 (bf16)	65,536 (bf16)
Cores/Chip	1	2
Threads/Core	1	1
On-Chip Memory	144 MiB	160 MiB
DRAM memory capacity, BW	8 GiB, 614 GB/s	32 GiB, 1200 GB/s

Figure 7.11 Key features of the TPU v4 lite and TPU v4 DNN accelerators.

We'll start with *TPU v4 lite* that is aimed at inference, and then describe the changes to *TPU v4* for training.

TPU v4 lite Architecture

Figure 7.12 shows the TPU v4 lite block diagram, including the *uncore*, which is everything on the chip except the TensorCore. (TPU v4 includes scaled-up versions of the same uncore found in the inference chip.) At the center of the design is a vector unit along with a 16 MB vector memory. The vector unit performs the activation functions of the DNN. Google invented "brain float 16" (Bfloat16) floating-point format starting with TPU v2 since it is a better match to DNNs than IEEE 754 fp16. Unlike fp16, bf16 has the same number of exponent bits as IEEE 754 fp32 but fewer mantissa bits, which matter less for DNNs. Attached to the vector unit are four matrix multiply units (MXUs). Each performs 128×128 matrix multiplies per clock cycle, either in bf16 that accumulate in IEEE 754 fp32 or in 8-bit integers that accumulate in 32 bits. Each MXU has 16,384 multiply-accumulate units, which sets TPU v4 lite's peak performance at 138 TeraFLOPS/second. The Cross Lane Unit (XLU) performs 128×128 matrix transpositions, reductions, and permutations across the VPU lanes.

The MXU uses systolic execution to save energy by reducing the reads and writes to vector memory (Kung and Leiserson, 1980; Ramacher et al., 1991). A *systolic array* is a two-dimensional collection of arithmetic units that each independently compute a partial result as a function of inputs from other arithmetic

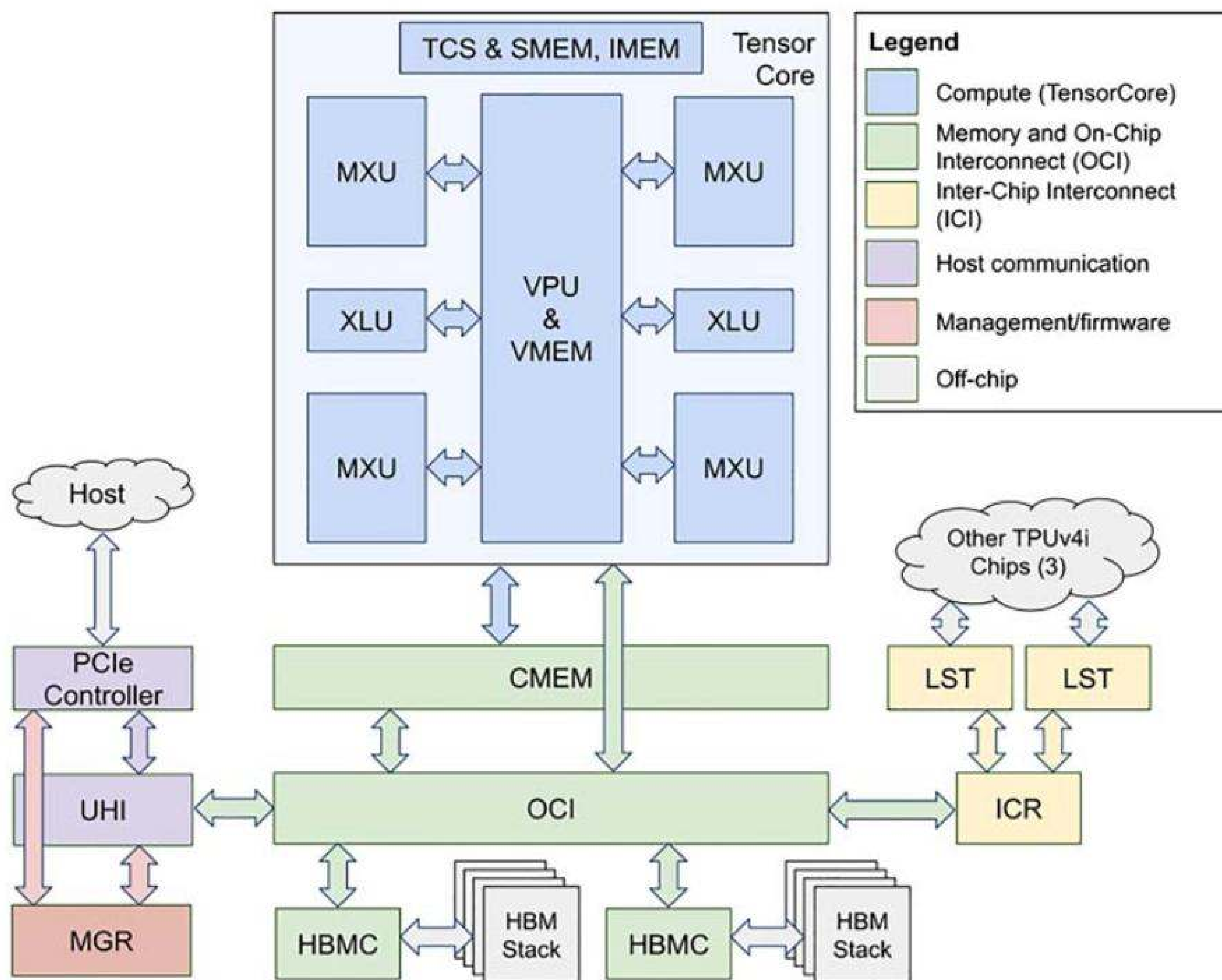

Figure 7.12 TPU v4 lite Block Diagram. Architectural memories are Common Memory (CMEM), Vector Memory (VMEM), Scalar Memory (SMEM), Instruction Memory (IMEM), and High Bandwidth Memory (HBM). HBM is a standard DRAM product in a separate package that many DNN DSAs rely upon. The data path includes the Matrix Multiply Unit (MXU), Vector Processing Unit (VPU), Cross-Lane Unit (XLU), and TensorCore Sequencer (TCS). The uncore—everything not the large shaded box at the top middle—includes the On-Chip Interconnect (OCI), ICI Router (ICR), ICI Link Stack (LST), HBM Controller (HBMC), Unified Host Interface (UHI), and Chip Manager (MGR).

units that are considered upstream to each unit. It relies on data from different directions arriving at cells in an array at regular intervals where they are combined. Because the data flows through the array as an advancing wave front, it is similar to blood being pumped through the human circulatory system by the heart, which is the origin of the systolic name.

Figure 7.13 demonstrates how a systolic array works. The six circles at the bottom are the multiply-accumulate units that are initialized with the weights *wi*. The staggered input data *xi* are shown coming into the array from above. The 10 steps of the figure represent 10 clock cycles moving down from top to bottom of the page. The systolic array passes the inputs down and the products and sums to the right. The desired sum of products emerges as the data completes its path through the systolic array. Note that in a systolic array, the input data is read only once from memory, and the output data is written only once to memory.

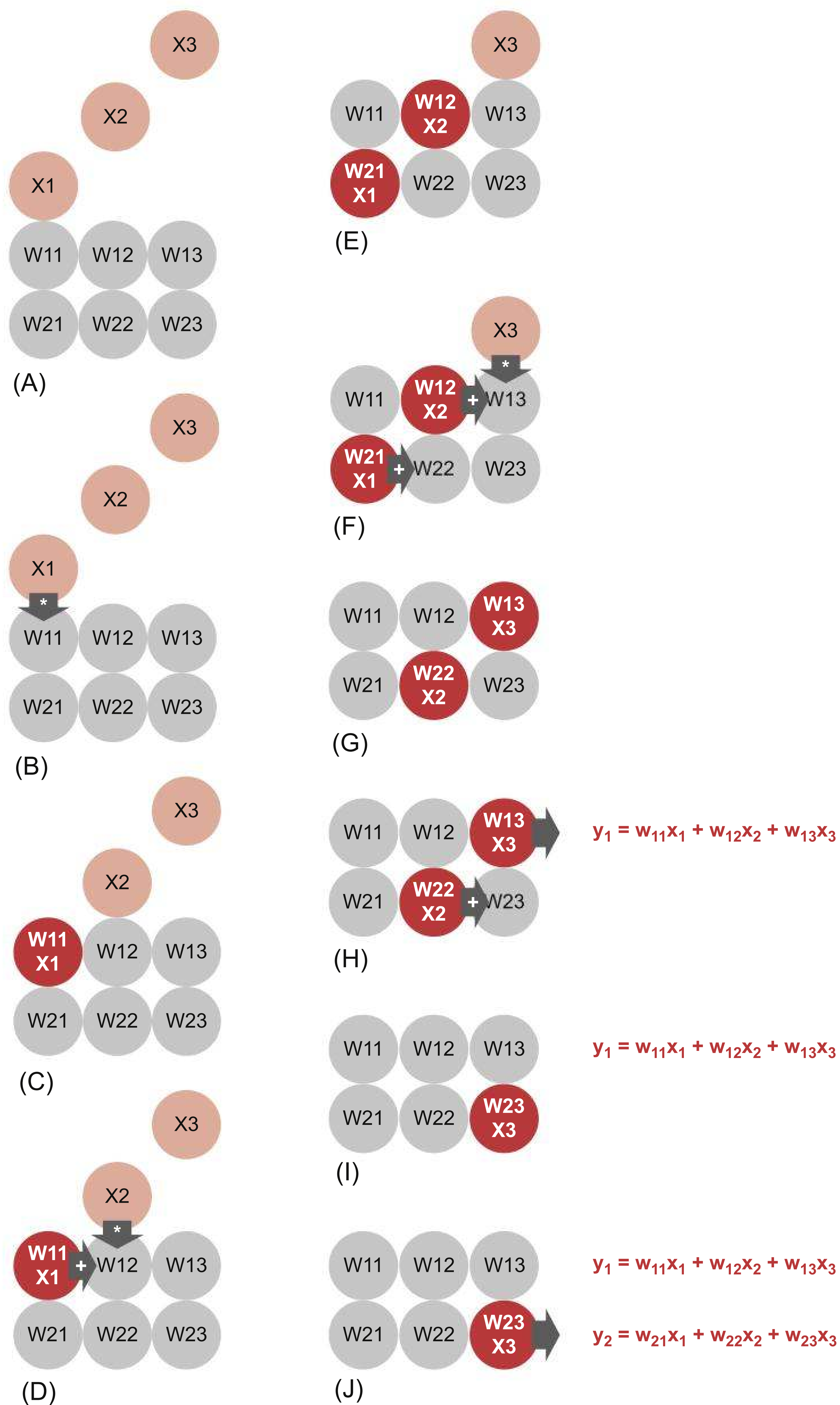

Figure 7.13 Example of systolic array in action, top to bottom on the page. In this example the six weights are already inside the multiply-accumulate units, as is the norm for the TPU. The three inputs are staggered in time to get the desired effect and, in this example, are shown coming in from the top. (In the TPU the data actually comes in from the left.) The array passes the data down to the next element and the result of the computation to the right to the next element. At the end of the process, the sum of products is found to the right. Drawings courtesy of Yaz Sato.

In the TPU the systolic array is rotated. Figure 7.13 shows that the weights are loaded from the top and the input data flows into the array from the left. A given 128-element multiply-accumulate operation moves through the matrix as a diagonal wave front. The weights are preloaded and take effect with the advancing wave alongside the first data of a new block. Control and data are pipelined to give the illusion that the 128 inputs are read at once, and after a feed delay, they update one location of each of the 128 accumulator memories. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it must worry about the latency of the unit.

Figure 7.14 shows the floor plan of the TPU v4 lite die. The TPU chip was fabricated using a 7-nm process with a clock rate of 1.05 GHz. Although the exact die size is not revealed, it is less than 400 mm^2.

The largest block in Figure 7.14 is Common Memory (CMEM), representing 28% of the chip. There are large data structures needed for DNNs that don't fit in the 16 MB vector memory, so TPU v4 expands the memory hierarchy by adding a 128 MB memory to reduce the number of accesses to the slowest and least energy-efficient main memory (High Bandwidth Memory or HBM).

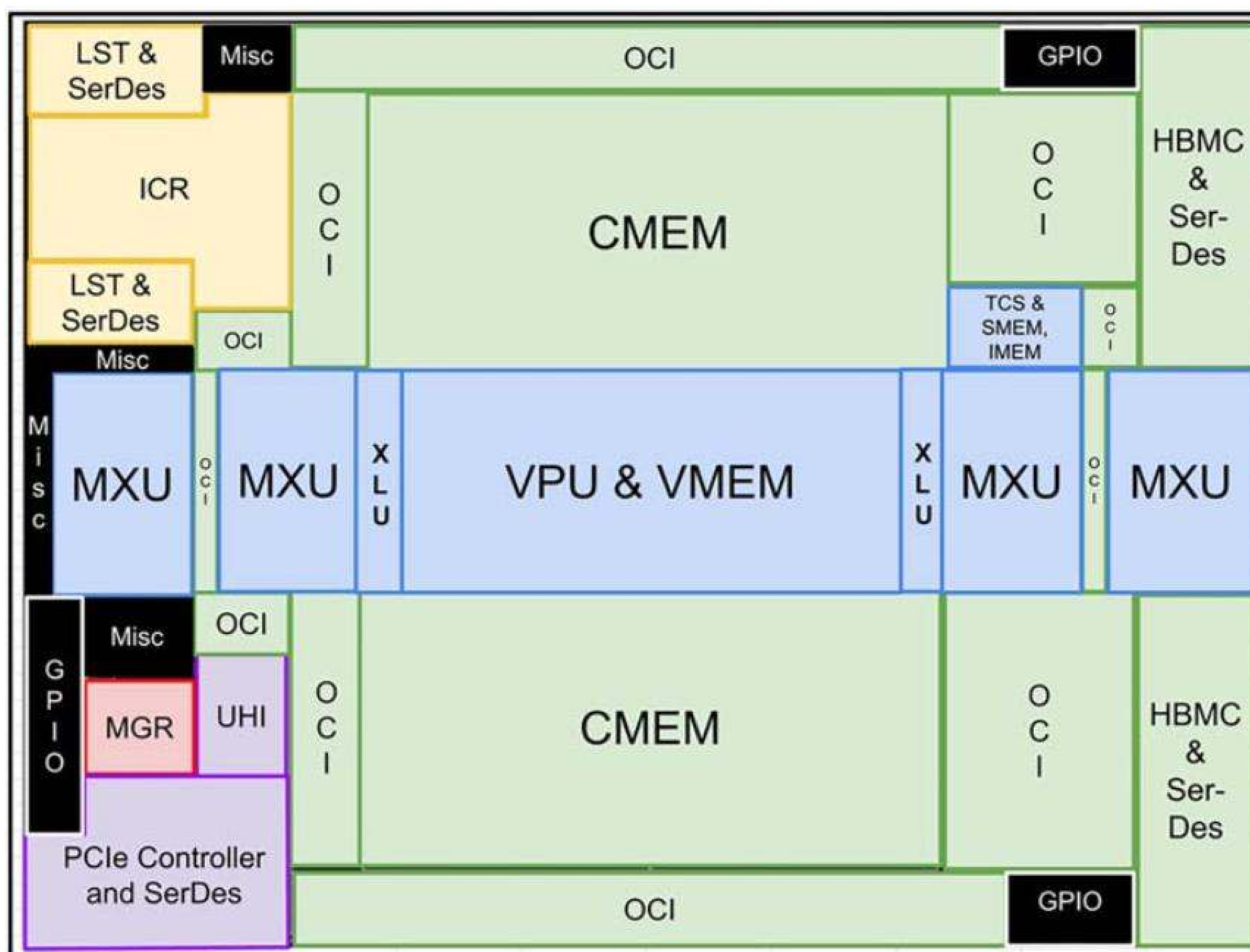

Figure 7.14 Floor plan of TPU v4 lite die. The shading follows Figure 7.11. The die is <400 mm^2. CMEM is 28% of the area. OCI blocks are stretched to fill space in the abutted floorplan because the die dimensions and overall layout are dominated by the TensorCore, CMEM, and SerDes locations. The TensorCore and CMEM block arrangements are derived from the TPU v4 floorplan.

TPU v4 Lite Instruction Set Architecture

Rather than try to be binary compatible with the VLIW instruction set of TPU v2 and TPU v3, Google engineers chose instead to be *compiler compatible,* for a few reasons:

- The original argument for VLIW was enabling more hardware resources over time by recompiling the apps with the compiler controlling the new instruction level parallelism, which binary compatibility restricts.
- Some Google engineers had worked on Itanium compilers, where they learned the drawbacks of binary compatibility for a VLIW compiler and hardware.
- The XLA compiler accepts JAX and PyTorch as well as TensorFlow, so TPUs could rely on one compiler versus having an interface that works for many compilers.
- TPU software is maintained and distributed in source code, not binary code.

The TPU v4 instruction set is based on ~400-bit VLIW instruction bundles instead of the 322-bit bundles of TPU v2 and TPU v3, with the extra bits to control the extra features, like more MXUs.

XLA produces high-level operations (HLOs) that are machine-independent and low-level operations (LLOs) that are machine-dependent. Optimizations at the HLO level apply to all platforms. If a new TPU restricts the needed compiler changes to LLOs, for example, a wider VLIW, the HLO optimizations are unaffected.

The XLA compiler controls all forms of parallelism explicitly:

- *Instruction-level parallelism*: XLA uses standard compilation techniques including loop unrolling, instruction scheduling, and software pipelining to keep all compute units busy.
- *Data-level parallelism*: With 2D vector registers and compute units, the layout of data in both compute units and memory is critical to performance, perhaps more than for a vector or SIMD processor. For the MXU, two 2D inputs interact to produce a 2D output. Each operand has a memory layout, which gets transformed into a layout in 2D registers, which in turn must be fed at the exact moment to meet systolic array timing in the MXU.
- *Memory-level parallelism*: TPU v4 lite has no caches, so all memory transfers between the different levels of the memory hierarchy are scheduled by the compiler. TPU v4 lite contains tensor DMA engines that are distributed throughout the chip's uncore to mitigate the impact of interconnect latency and wire scaling challenges. The tensor DMA engines function as

coprocessors that fully decode and execute TensorCore DMA instructions. The TPU v4 lite four-dimensional DMA unifies the DMA architecture across local (on-chip), remote (chip-to-chip), and host (host-to-chip and chip-to-host) transfers to simplify scaling of applications from a single chip to a full pod.

- *Thread-level parallelism:* XLA maps the parallelism from a TensorFlow program across thousands of chips in a supercomputer. The goal is to run seamlessly on systems from 4 to 4096 chips.

Changes for TPU v4 Versus TPU v4 lite

As mentioned earlier, the biggest change to the chip is that TPU v4 has two tensor cores (TCs) instead of one, which doubles the peak performance to 275 TeraFLOPS/second. The two cores share access to CMEM, hence the name Common Memory.

From a systems perspective, the two TPUs are very different. TPU v4 lite is optimized for inference, configured as eight TPU v4 lites per CPU host. Given the greater compute demand on the host for training, each CPU host has only four TPU v4s. Since TPU v4s are intended to be connected into a supercomputer that Google calls *pods*, they have six Inter-Chip Interconnect (ICI) links, whereas TPU v4 lite has only two. Note that TPU v3 has four ICI links. The extra links allow TPU v4 to be configured in network topologies that have greater bisection bandwidth (3D torus). Whereas the maximum pod size for TPU v3 was 1024 chips, the TPU v4 can have up to 4096. The peak performance of the pod is therefore over 1 ExaFLOPS/second for bf16. Unlike the TOP500 list that ranks individual HPC supercomputers, Google deploys dozens of these pods in the cloud.

The final major change to TPU v4 that helps training is to increase the capacity of off-chip HBM memory from 8 GiB in TPU v4 lite to 32 GiB in TPU v4.

Summary: How TPU v4 Follows the Guidelines

Let's see how TPU v4 and TPU v4 lite followed Section 7.2 guidelines.

1. *Use dedicated memories to minimize the distance over which data is moved.* TPU v4 has a two-level on-chip SRAM memory that is controlled by the program. This SRAM is 20 times as energy efficient as off-chip HBM memory.
2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.*
   TPU v4 offers 160 MiB of software-controlled scratchpad memory (128 MiB of CMEM and two 16 MiB VMEMs) and 131,072 16-bit/32-bit ALUs, which means it has an order of magnitude more ALUs than a contemporary server-class CPU. TPU v4 lite has half the ALUs and nearly the same amount of on-chip memory (128 MiB of CMEM and one 16 MiB VMEM).

3. *Use the simplest form of parallelism that matches the domain.*
   The TPU delivers its performance via a two-dimensional SIMD parallelism with its four 128 × 128 MXUs per TC, which is internally pipelined with a systolic organization. In addition to exploiting data-level parallelism, it has two TCs per socket and up to 4096 chips per supercomputer. It exploits instruction-level parallelism using its ~400-bit VLIW instructions. Contemporary CPUs relied on multiprocessing, out-of-order execution, multithreading, and one-dimensional SIMD.
4. *Reduce data size and type to the simplest needed for the domain.*
   The TPU computes primarily on 16-bit floating-point data, although it accumulates in 32-bit floating point and supports 8-bit integers. The primary benefit is that the narrower data reduces the demands on memory bandwidth and on memory capacity. CPUs also support 64-bit integers and 32-bit and 64-bit floating point.
5. *Use a domain-specific programming language to port code to the DSA.*
   The TPU is programmed using JAX, PyTorch, and TensorFlow programming frameworks. CPUs must run virtually every programming language.

7.5 The NVIDIA A100 and T4 GPUs, Graphics, and DNN Accelerators for the Data Center

As Chapter 4 shows, the NVIDIA GPU is fundamentally a multiprocessor architecture. The processor is called an *SM* for *Streaming Multiprocessor*. GPUs scale cost and performance by varying the number of SMs for a given generation from scores of SMs to more than 100. They also leverage the multiprocessor architecture to improve the yield of GPU chips by having redundant SMs.

GPUs have memory hierarchies featuring both cache and scratchpad memories, but they primarily hide memory latency using many threads and quickly switch among them. The primary hardware to support fast content switching is very large register files. The GPU architecture encourages CUDA programmers to expose many threads so that there are always threads ready to execute when some threads wait on memory. GPUs gang 32 threads at a time into *warps*. GPUs switch to a new warp if one warp is waiting for operands from HBM memory. GPUs only get full memory bandwidth when the warps are *coalesced*, which means loads or stores are to sequential locations in memory.

GPUs were originally designed for graphics, which involve numerically intensive calculations on data-intensive algorithms that have regular memory accesses. DNNs have similar characteristics, which is why GPUs became popular for ML a decade ago.

With the increasing popularity of ML on GPUs, recent generations have added features that specifically help DNNs, in addition to the architecture enhancements that help both. In the latter category new server-class GPUs:

- Increase bandwidth and capacity of main memory.
- Have larger cache memories.
- Increase the number of SMs per socket.
- Enhance Turbo boost clock rate.

NVIDIA A100 Tensor Core Architecture

Section 4.4 in Chapter 4 shows that NVIDIA GPUs use a tiled architecture, with each tile called a *Streaming Multiprocessor* (*SM*). In Chapter 4 we instead use the more descriptive term *Multithreaded SIMD processor*. The number of SMs increases per GPU generation, from 56 SMs in Pascal to 132 in Hopper (see Figure 4.21). Volta, the GPU contemporary to Google TPU v3 (Jouppi 2020), was the first NVIDIA GPU with direct support for ML. NVIDIA reports that depending on the size of the matrices, it sped up peak ML performance on GPUs by 6x–9x.

The CUDA operation for matrix multiply is D = A x B+C, where A, B, C, and D can be tiles of larger matrices (Raihan, 2019). The size of each tile is MxNxK, where:

- MxK is the dimension of A,
- KxN is the dimension of B, and
- MxN is the dimension of C and D,

which is exactly what one would expect for matrix multiplication. Volta added four TC PTX instructions to support these operations: LoadA, LoadB, MultiplyAddA, and StoreD. Each instruction has many options based on type, shape, stride, data layout, and synchronization.

To map the computation to the multithreaded nature of SMs, each tile is further subdivided into *fragments*, mapped into the registers of one per thread. For example, a Warp of 32 threads for a 16×16 tile might map a $16 \times 16/32 = 8$ element fragment into 8 separate general-purpose registers. NVIDIA has not published details on how it does the mapping of fragments to registers, but for readers interested in more details, Raihan (2019] reverse-engineered how it was done to show that matrix C has a different distribution of data within a warp than matrices A and B.

Each Volta SM consists of four *subcores*, which Figure 7.15 describes. Each Volta subcore in turn has two *tensor cores (TCs)* [Choquette 2018]. (Google uses the name TensorCore for the processor in its TPUs, but names are coincidental.) Volta has 80 SMs, so it contains $80 \times 4 \times 2 = 640$ TCs. The NVIDIA Ampere

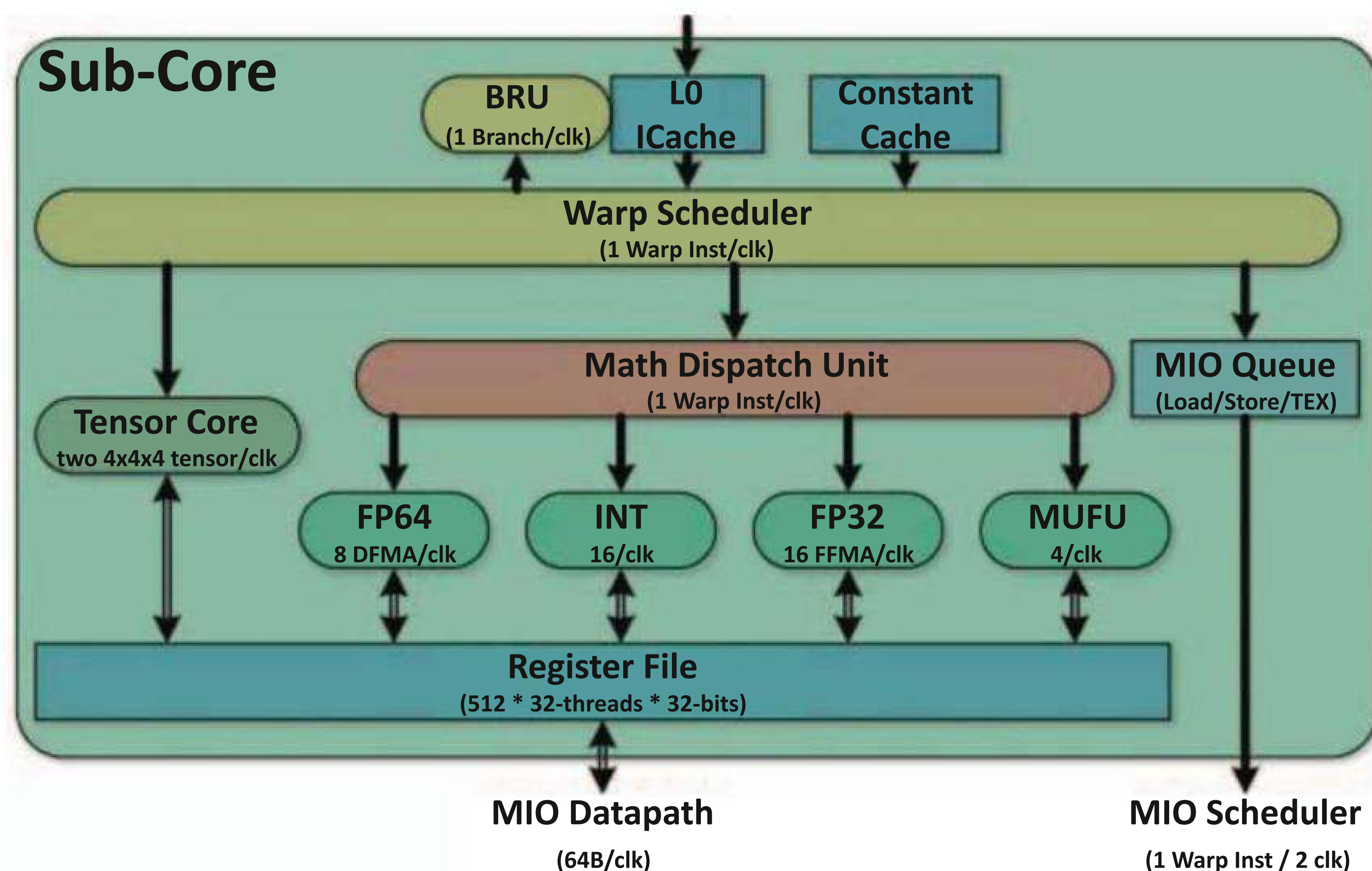

Figure 7.15 NVIDIA Volta SM subcore. Reproduced from Figure 3 in [Choquette, J., Giroux, O. and Foley, D., 2018. Volta: Performance and programmability. IEEE Micro, 38(2), pp. 42–52].

generation has only one TC per subcore, so its total is 108×4×1 = 432 TCs. Hopper returns to two TCs per subcore.

Each TC performs matrix multiplication. One TC operates on 1024 bits per clock cycle and can operate on operands from 1-bit integer to 16-bit float point: 1024 1-bit integers, 256 4-bit integers, 128 8-bit integers, or 64 16-bit floating point numbers. The Warp Scheduler in Figure 7.15 sends multiply-add operations to a TC, which fetches operands from the Register File, completes the 4×4×4 matrix multiply, and writes the result back to the Register File. Each TC completes one 4×4×4 matrix multiply-add per clock cycle, requiring 64 floating-point operations.

While NVIDIA has not published a block diagram of the TC, Raihan (2019) ran microbenchmarks to try to reveal its underlying microarchitecture. Figure 7.16 is their estimate of the Volta TC Microarchitecture.

TPU hardware for matrix multiplication relies upon systolic arrays. Since Volta is only calculating 4×4 matrices instead of 128×128, it does not require systolic arrays. The Volta multiply-add hardware is simply combinational logic, relying on good circuit design, logic design, pipelining, and physical design to complete all 64 floating-point operations within a single clock cycle.

The Volta TC can perform fp16 or mixed fp16 and fp32 arithmetic operations. Figure 7.17 shows subsequent GPUs changed the data types that TCs support. *tf32* in Ampere stands for *TensorFloat-32*, which uses the same exponent size as fp32 (8 bits) and the same mantissa size as fp16 (10 bits). This 19-bit format requires 32

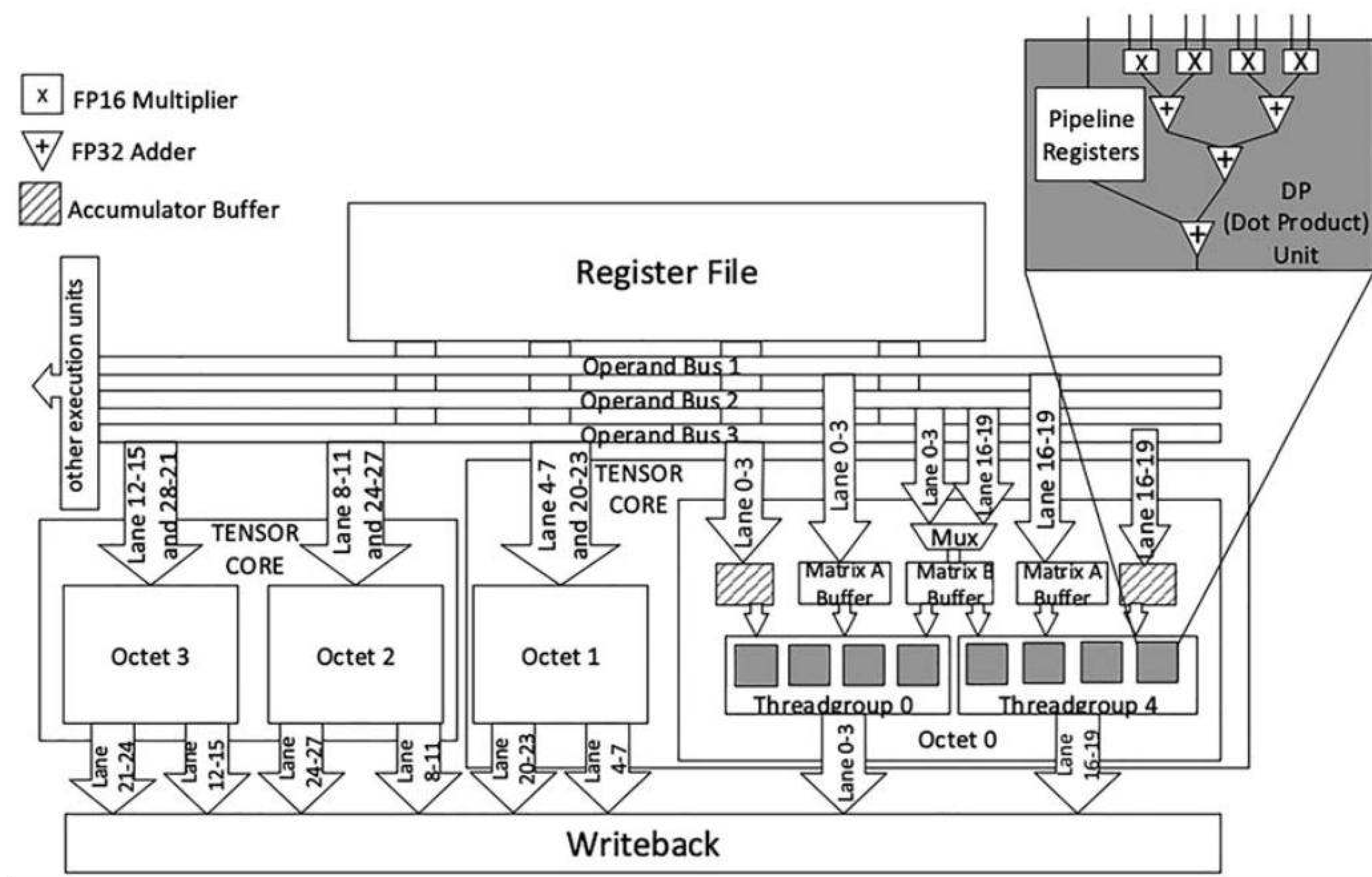

Figure 7.16 Proposed NVIDIA Volta Tensor Core microarchitecture. A *threadgroup* is four consecutive threads within a warp. An *octet* is the pair of threadgroups that compute 8×8 subtiles of the result. Each Volta warp has four octets. Reproduced from Figure 13 in [Raihan, M.A., Goli, N. and Aamodt, T.M., 2019, March. Modeling deep learning accelerator enabled GPUs. In 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (pp. 79—92). IEEE].

GPU	*Integer*	*Floating Point*
Volta	n.a.	fp16, fp32
Turing	Int8	fp16
Ampere	Int1, Int4, Int8	fp16, fp32, bf16, tf32, fp64
Hopper	Int 8	fp16, fp32, bf16, tf32, fp64, fp8
Blackwell	Int 8	fp16, fp32, bf16, tf32, fp64, fp8, fp6, fp4

Figure 7.17 Data types that tensor cores support across NVIDIA GPU generations.

bits when stored in memory. It leads to more efficient computation than fp32 when the 7-bit mantissa of bf16 or the 5-bit exponent of fp16 is insufficient. The fp8 floating point format in Hopper has two flavors: sign, 4-bit exponent and 3-bit mantissa or sign, 5-bit exponent and 2-bit mantissa. The expectation is that the forward pass of training needs more precision, so it needs the larger mantissa, but the

backward pass needs a bigger range and hence the larger exponent. The Blackwell fp4 format has only 2 bits of exponent and 1 bit of mantissa, which is amazingly a floating point number that has only 16 possible values.

On the A100, each SM has 4 TCs, and each TC operates on 4096 bits per clock cycle. The Ampere generation also introduced support for fine-grained structured sparsity. It applies to contiguous blocks of four values, when any two of the four values are zero. The A100 adds a new PTX instruction that performs 2:4 sparse matrix multiplication.

The TCs only operate on the two nonzero values, so they can complete the same computation in half the time. This optimization doubles peak performance when the data fits this pattern. DNNs are fairly adaptable, so if the ML frameworks can try to ensure that a section of data obeys 2:4 sparsity after training. NVIDIA shows an example performance gain of 1.10x−1.18x, depending on the batch size.

Besides adding fp8 (see Figure 7.17), the Hopper generation added hardware specifically for the Transformer DNN, which illustrates its importance. The *Transformer Engine* automatically chooses between fp8 and fp16 data types for each layer of a Transformer DNN by monitoring the statistics of the out values of each TC. When possible, it uses fp8 to accelerate performance. The hardware scales between fp8 and fp16 during execution so that the programmer need not manage the transition. The goal of the Transformer Engine is to allow every layer in the DNN to use the smallest data range it needs.

Hopper dropped Int1 and Int4 data types of the A100 (see Figure 7.17), presumably because the architects thought they were no longer needed. Given the fast-changing nature of ML, it is hard to know what data types will be important even only three years from the start of a design of a DSA for DNNs to when it is deployed.

Figure 7.18 shows key features of the NVIDIA A100 and T4 GPUs. Like the comparison of TPU v4 lite to TPU v4 in the previous section, for inference, NVIDIA packs eight T4s per CPU host versus four A100s per CPU host. Similarly, at the system level, the A100 scales to 4096 chips but the T4 only scales to 8 chips. The A100 uses 12 NVLinks to seamlessly scale to an 8-socket system called DGX and then uses cluster networking switches and links to scale to 512 DGX systems together to reach systems with a total of 4096 chips.

Figures 7.19 and 7.20 show the chip layout of the two GPUs. When comparing A100 and T4 chips, the size, power, and main memories are quite different. The T4 die is about two-thirds of the A100 die area but only holds about 40% of the SMs, as it is in a larger technology node (12 nm vs. 7 nm). Remarkably, T4 Thermal Design Power (TDP) is 4x less than A100 (70 W vs. 300 W) despite the base clock rates being only 1.3x slower (585 MHz vs. 765 MHz). Moreover, that ratio reverses for the highest Turbo boost mode, with T4 being 1.1x faster (1590 MHz vs. 1410 MHz).

The next biggest difference is the use of graphics DRAM (GDDR6) for T4 instead of high bandwidth memory in A100 (HBM2E). The capacity ratio is again about 40% (16 GiB vs. 40 GiB), but the memory bandwidth difference is substantial, nearly a factor of 5X (320 GB/s vs. 1555 GB/s.)

	NVIDIA A100	NVIDIA T4	Graphcore MK2 IPU
Production deployment	2020	2018	2021
Peak TFLOPS	312 (bf16), 624 (int8)	65 (IEEE fp16)/130 (8b int)	250 (bf16)
DNN Target	Training or Inference	Inference only	Training or Inference
Clock Rate	765/turbo 1410 MHz	585/Turbo 1590	1850 MHz
Tech. node, Die size	7 nm, 826 mm^2	12 nm, 545 mm^2	7 nm, 832 mm^2
Transistor count	54 billion	14 billion	59 billion
Chips per CPU host	4	8	4
Thermal Design Power	300 W	70 W	300 W
Inter Chip Interconnect	12 links @ 25 GB/s	--	3 links @ 64 GB/s
Largest scale configuration	4096 chips	8 chips	256 chips
Cloud Availability	AWS, Azure, Google	AWS, Azure, Google	AWS, Azure, Google
Processor Style	Single Instruction Multiple Threads	Single Instruction Multiple Threads	Multithreaded Single Instruction Multiple Data
Arithmetic Units/ Processor	1024 (fp16)	512 (fp16)	64 (fp16)
Processors/Chip	108	40	1472
Threads/Core	32	32	6
On-Chip Memory	40 MiB	18 MiB	897 MiB
DRAM memory type	HBM2E	GDDR6	None
DRAM memory capacity, BW	40 GiB, 1555 GB/s	16 GiB, 320 GB/s	0

Figure 7.18 Key features of the NVIDIA A100, NVIDIA T4, and the Graphcore MK2 IPU DNN accelerators.

Figure 7.19 Floorplan of the NVIDIA A100. While it shows 128 SMPs, NVIDIA enables only 108 of them. The L2 cache holds 40 MB. The A100 can compress data from DRAM into the cache to accelerate unstructured sparsity and other compressible data patterns. The claim is that compression yields up to 4X of the DRAM read/write bandwidth, up to 4X of the L2 read bandwidth, and up to 2X of the L2 cache capacity.

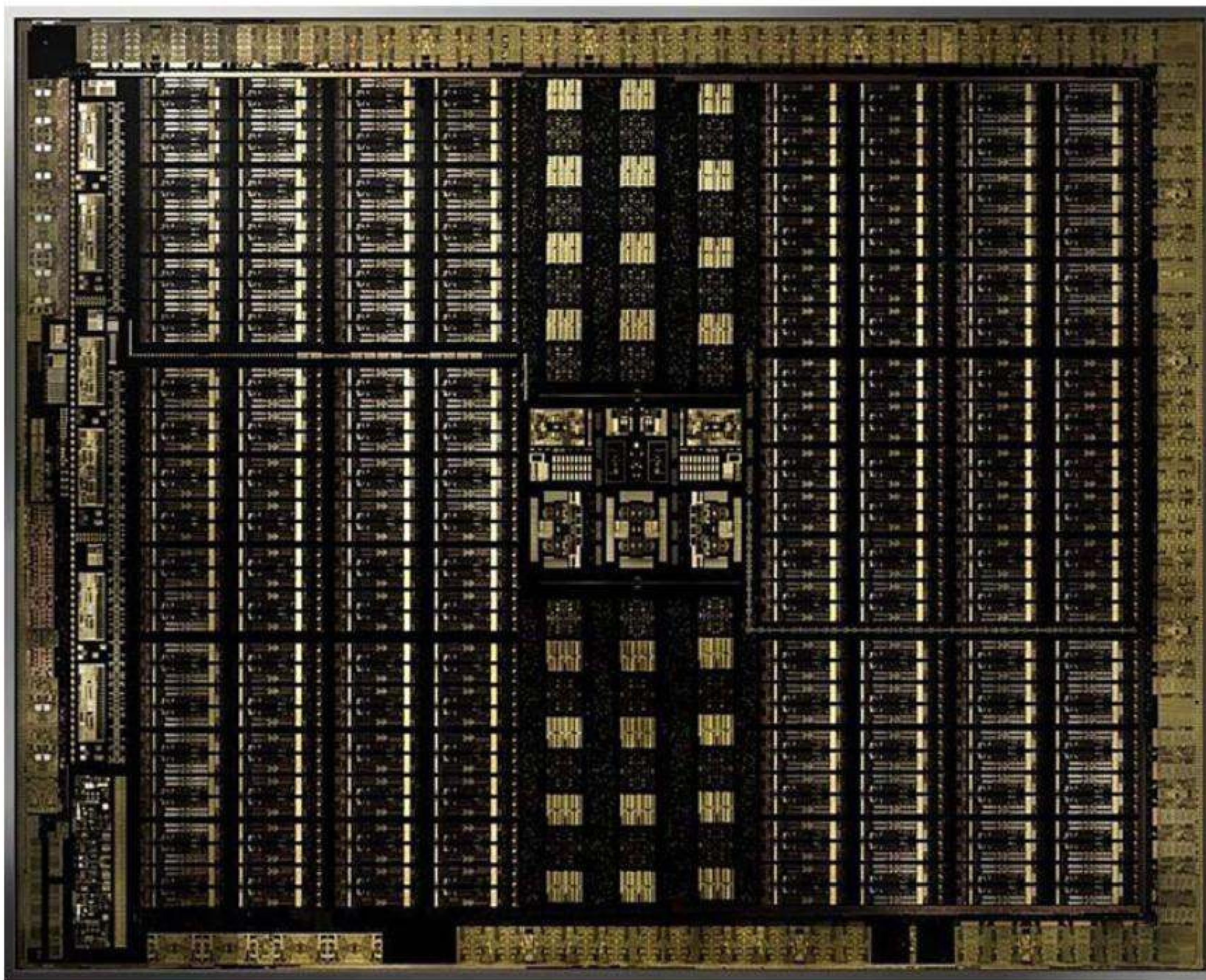

Figure 7.20 Die photo of NVIDIA T4. While it shows 48 SMPs, NVIDIA enables only 40 of them.

A final difference is how they handle main memory error correction (ECC). Like server CPUs, A100 uses wider HBM so that ECC can be checked or stored simultaneously with the data access: 72 bits for 64 bits of data plus 8 bits of ECC. T4 GDDR memory is 64 bits wide, so the memory controller stores the ECC bits elsewhere in the DRAM chip if ECC is turned on. (The former style is called side band ECC and the latter is called in band ECC.) ECC on T4 is thus optional for the customer. Google data centers require ECC on main memory, which reduces T4 performance on average about 5% (see Section 7.10).

Summary: How A100 and T4 Follow the Guidelines

Let's see how TPU v4 followed Section 7.2 guidelines.

1. *Use dedicated memories to minimize the distance over which data is moved.* The GPUs have very large register files to support large-scale multithreading: 27 MiB for A100 and 10 MiB for T4. These registers are comparable in size to their L2 caches, which are 40 MiB and 18 MiB, respectively. While the

multithreading is helpful when the GPUs are used as DSAs for graphics, it's unclear how much they help ML.

2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.*
   The A100 has 110,592 16-bit/32-bit ALUs and the T4 has 20,480 ALUs, once again many times what is found in contemporary CPUs.
3. *Use the easiest form of parallelism that matches the domain.*
   GPUs rely on multiprocessing, multithreading, and two-dimensional SIMD (although the multiplier units are smaller than those of TPUs).
4. *Reduce data size and type to the simplest needed for the domain.*
   To let the programmer pick the optimum size, GPUs offer the widest range of data types: 4-bit and 8-bit integers and IEEE floating-point at 16 and 32 bits, with A100 adding 1-bit and 16-bit integers and bf16, tf32, and fp64 floating-point formats.
5. *Use a domain-specific programming language to port code to the DSA.*
   The GPU is programmed using the JAX, PyTorch, and TensorFlow programming frameworks for ML plus CUDA and OpenCL. Graphics uses libraries like DirectX, Metal, and OpenGL.

7.6 Graphcore IPU Bow, a Data Center Accelerator for Training

Figure 7.18 gives the key characteristics of the Graphcore *Intelligence Processing Unit* (*IPU*) and Figure 7.21 shows a die photo. The IPU is a massively parallel processor in a chip, with the second-generation IPU Bow containing 1472 IPU cores. Each core can compute 64 16-bit floating point operations per clock cycle. Like GPUs, IPUs have redundant cores (1 in 24) to improve yield of their large 832 mm^2 chip. The IPU uses 32-bit scalar instructions of Graphcore's own design. Like CPUs and GPUs, it includes hardware multithreading that directly supports 6 threads per core. Thus, like for GPUs, programmers are encouraged to use many threads to ensure that IPUs have plenty of work to keep them well utilized.

The Exchange Unit in the middle of the chip provides the communication between the tiles. Like TPU v4 and the A100, the IPU has on-chip links that allow IPU chips to be connected together in larger systems. While the theoretical limit is 64,000 IPUs, the largest system submitted for the MLPerf benchmark was 256 chips (see Section 7.10). The Exchange Unit on each chip works with the IPU links so that the programmer can seamlessly send messages to any tile on any chip. Measurements of all-to-all communication show very effective bandwidth within a single chip and between chips.

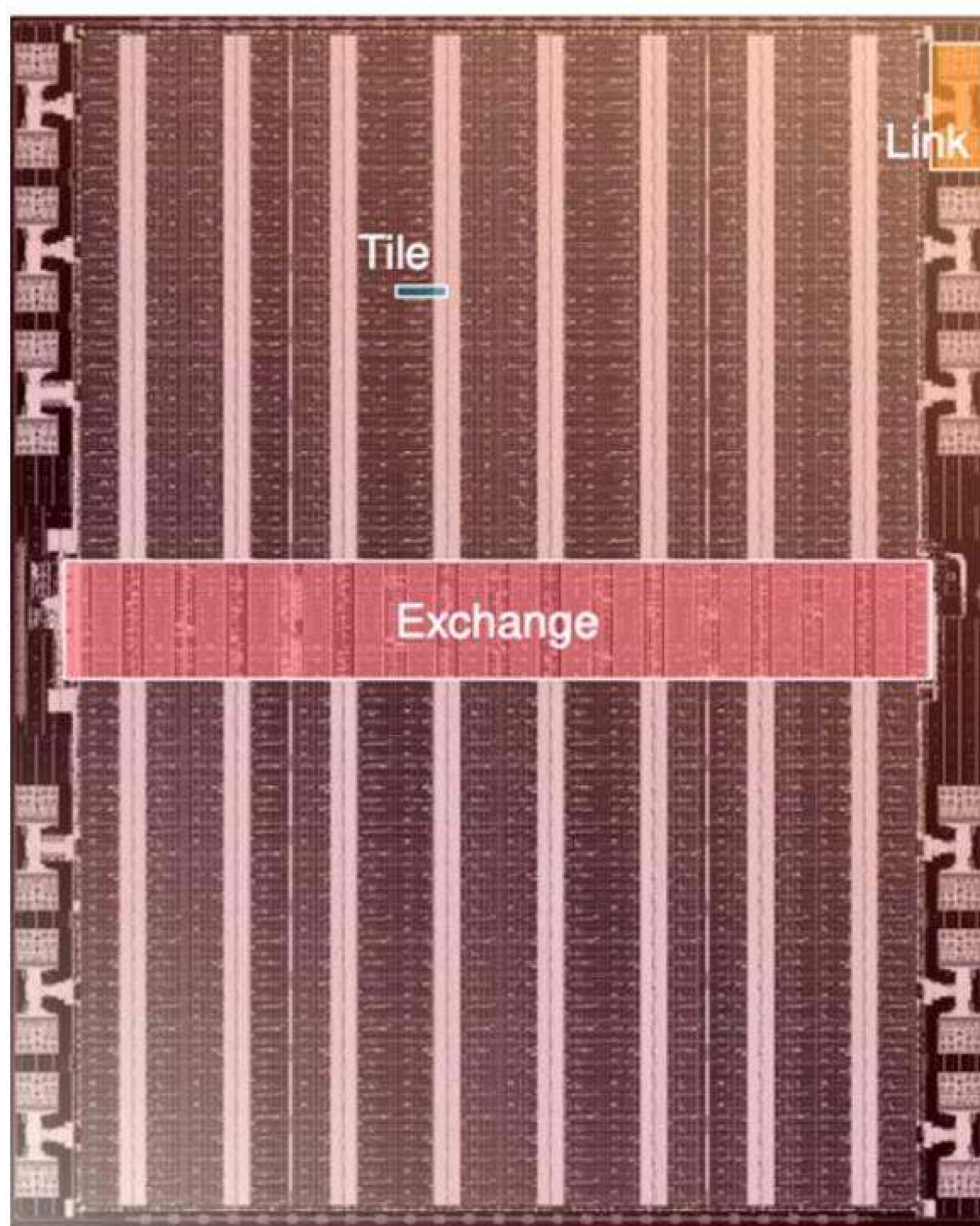

Figure 7.21 Die photo of Graphcore IPU MK2. Manufactured in a 7 nm, the die size is 832 mm^2. It holds 1492 tiles, each with an IPU core and 624 KiB of SRAM, yielding a total of 897 MiB per die. The exchange performs all communication between tiles. Each die has 3 IPU links, which enable large-scale systems; the largest benchmarked so far has 256 chips. Tile memory is about half of the die area, tile logic is about a quarter, the exchange is about one-tenth, and the rest is uncore. From Sparsity-Aware and Reconfigurable NPU.

To simplify design and programming, the IPU relies on the Bulk Synchronous Parallel (BSP) model for parallel computing. BSP simplifies the parallel computing challenge by dividing it into a sequence of larger steps. Each large step contains a computation phase, followed by a communication phase, and then a barrier synchronization. Thus all threads must synchronize before the program proceeds to the next step.

The most unusual feature is the IPU memory system. Each IPU Bow tile has 624 KiB of SRAM, or a remarkable total of 897 MiB of SRAM per chip for all 1472 cores. To put that into perspective, it is 3X the size of the first generation IPU, nearly 6X the on-chip SRAM of TPU v4, and 22X the L2 caches of the A100. Equally bold is that there is no DRAM attached to the IPU; if the

application needs more memory than fits in the IPU, it must go across a gateway to use the DDR4 DRAM attached to the host ARM CPU. The IPU architect's bet was that the on-chip SRAM is sufficient for most DNNs (see Section 7.11).

Summary: How the IPU Bow Follows the Guidelines

The IPU Bow adheres to Section 7.2 guidelines as follows.

1. *Use dedicated memories to minimize the distance over which data is moved.*
   The IPU Bow deploys 624 KiB per core with a total of 897 GiB per chip, trying to keep data local as much as possible. The only DRAM is on the CPU host, which offers much lower bandwidth than the DRAM memories of the earlier DSAs.
2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.*
   The IPU Bow has 94,208 16-bit ALUs, once again vastly outpacing contemporary CPUs.
3. *Use the easiest form of parallelism that matches the domain.*
   IPUs rely on multiprocessing, multithreading, and one-dimensional SIMD.
4. *Reduce data size and type to the simplest needed for the domain.*
   Unlike the other DSAs, IPU Bow only supports fp16 and fp32.
5. *Use a domain-specific programming language to port code to the DSA.*
   Like TPUs and GPUs, the IPU is programmed using JAX, PyTorch, and TensorFlow programming frameworks.

7.7 The Samsung Neural Processing Unit (NPU), a Smartphone Inference Accelerator

A smartphone DSA is designed to be part of a *system on a chip* (*SOC*). Figure 7.22 shows the equivalent features for the Samsung *Neural Processing Unit* (*NPU v1*) to those for the data center accelerators in Figures 7.11 and 7.18, and Figure 7.23 shows its die photo. Its budgets for energy and die area are two orders of magnitude smaller than the data center inference DSAs.

Each NPUv1 contains three cores, whose block diagram is in Figure 7.24. Each core contains two NPU engines that share a 1 MiB scratchpad memory called *Tightly Coupled Memory* (*TCM*). At the heart of the NPU engine are four 16×16 multiply accumulate units in the tensor unit, similar to the four MXU in the TPU v4 lite. They are presumably also organized as systolic arrays. They operate primarily on 8-bit integers, although they can also operate on 16-bit integers using multiple clock cycles per operation.

	Samsung NPUv1	Samsung NPUv2	Intel Core i7-11375H
Production deployment	2021	2022	2021
Peak TOPS/sec (8-bit int)	14.6 (8b)	39.3 (4b), 19.7 (8b), 9.8 (fp16)	1.3 (8b), 0.6 (fp16)
DNN Target	Inference only	Inference only	General Purpose
Clock Rate Base (MHz)	332	332	3300
Clock Rate Turbo (MHz)	1196	1196	5000
Clock Rate AVX/SIMD (MHz)	N.A.	N.A.	
Tech. node, Die area	5 nm, 5.4 mm2	4 nm, 4.7 mm2	10 nm, 122 mm2
Transistor count	N.A.	N.A.	N.A.
Thermal Design Power (Watts)	0.33 @0.6V, 0.79@0.9V	0.38–5.1 @0.55V	28-35
Processor Style	Single Instruction 2D Data	Single Instruction 2D Data	Multithreaded SIMD
Arithmetic Units/Core	2048 (8b)	4096 (8b)	64 (8b)
Cores/Chip	3	2	4
Threads/Core	1	1	2
On Chip Memory	3 MiB	2 MiB	17 MiB
DRAM memory type	N.A.	N.A.	DDR4, LPDDR4
DRAM memory capacity, BW	N.A.	N.A.	64 GiB, 26-34 GB/s

Figure 7.22 Key features of the Samsung NPUv1 and NPUv2 DNN inference accelerators for smartphones and the Intel Core i7, which are compared in Section 7.9. From Architecture for Samsung Flagship Mobile SoC.

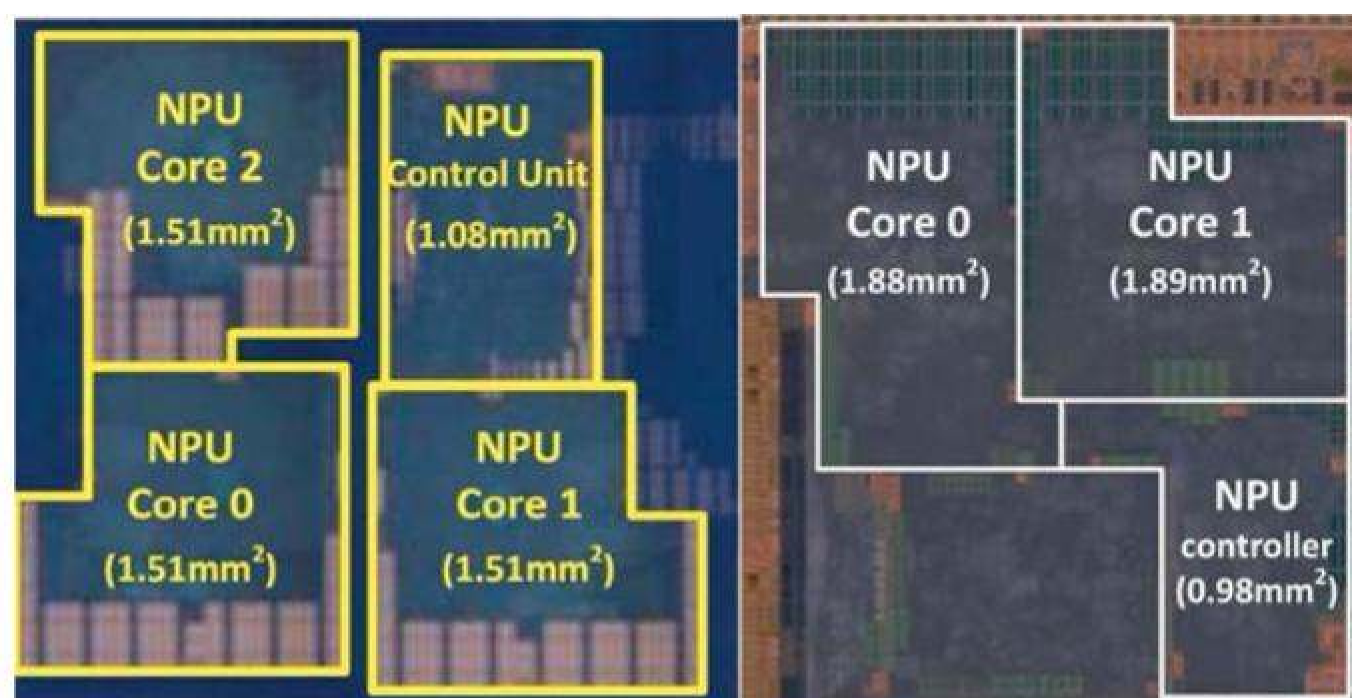

Figure 7.23 Die photos of the Samsung NPUv1 (*left*) and NPUv2 (*right*). NPUv1 is manufactured in 5 nm, the die size is 5.4 mm^2, and it holds 3 cores. NPUv2 is manufactured in 4 nm, the die size is 4.7 mm^2, and it holds 2 cores. Each core contains 1 MiB of SRAM in both versions. The two die photos show the actual relative sizes of the two versions. From 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).

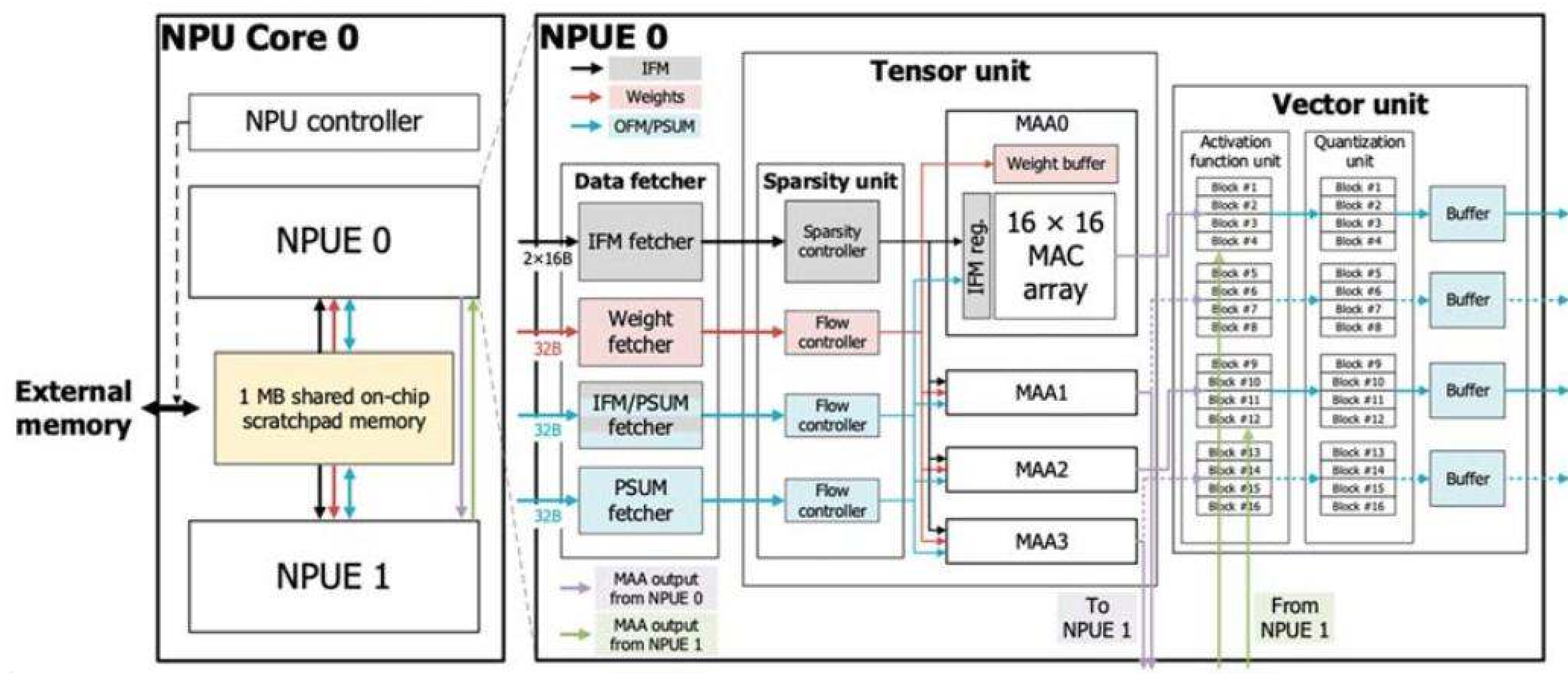

Figure 7.24 Block diagram of an NPUv1 core. From A Multi-Mode 8K-MAC HW-Utilization-Aware Neural Processing Unit with a Unified Multi-Precision Datapath in 4nm Flagship Mobile SoC.

The Data Fetcher transfers data from the TCM to the tensor unit on its right. It consists of subunits that fetch the input feature maps (IFU), weights (WFU), and partial sums (PSUM). It also has a unit that can either fetch feature maps or partial sums (IFM/PSUM) depending on the needs of the computation. All fetch subunits can operate simultaneously over separate 256-bit wide paths to the TCM.

Perhaps the most novel feature of NPUv1 is that it leverages the sparsity of the input matrices to save energy and increase performance. The Sparsity unit on the left side of the tensor unit in Figure 7.24 generates a dense matrix by skipping zero values in the input. These dense matrices are then transferred to the four MAC units. Next to the tensor unit is a vector unit, which like the vector unit in TPU v4 lite performs the nonlinear activation operation on the output of the MAC units.

A year after the debut of NPUv1, Samsung revealed the next generation of the NPU (see Figures 7.22 and 7.23). Similar in spirit to NPUv1, NPUv2 has the same clock rates and same TCM size per core. Here are the major differences:

- Each NPUv2 core has a single NPU engine instead of two.
- Each NPUv2 tensor unit supports 16-bit floating point data alongside 8-bit integer data. It also supports 4-bit integer data.
- Each NPUv2 tensor unit supports 4096 8-bit MACs, thereby doubling the number of MACs per core.
- To reduce impact of the larger area of the NPUv2 cores, each NPUv2 has two cores instead of three.

- NPUv2 has several operational modes, including priority for low power consumption for always-on features like voice-triggering that allows the other units to be asleep and uses only a single core. Low power mode can save 89% of power compared to standard modes. In low latency mode both cores operate at high clock rates on the DNN to process the model as quickly as possible. The power can be considerably higher than NPUv1.
- NPUv2 also has a gathering unit that can access data in the TCM of the other core. Compared to software solutions, the gathering unit significantly reduces redundant data transfers and overall latency for low latency mode.
- NPUv2 was fabricated in 4 nm instead of 5 nm and has a 15% smaller die area.

Summary: How the Samsung NPU Follows the Guidelines

The NPU embraces Section 7.2 guidelines as follows:

1. *Uses dedicated memories to minimize the distance over which data is moved.*
   The NPU has a 1 MiB scratchpad memory per core, which occupies 51% of the core.
2. *Invests the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.*
   The NPUv1 has 6,144 8-bit MACs and NPUv2 has 8192 8-bit MACs.
3. *Uses the easiest form of parallelism that matches the domain.*
   NPUs rely on multiprocessing and two-dimensional SIMD.
4. *Reduces data size and type to the simplest needed for the domain.*
   While NPUv1 supports only 8-bit and 16-bit integers, NPUv2 adds 4-bit integers and 16-bit floating point.
5. *Uses a domain-specific programming language to port code to the DSA.*
   The NPU is programmed using the TensorFlow and TensorFlow Lite programming frameworks.

7.8 Cross-Cutting Issues

Heterogeneity and System on a Chip

The easy way to incorporate DSAs into a system is over the I/O bus, which is the approach of the data center accelerators in this chapter. To avoid fetching memory operands over the slow I/O bus, these accelerators have local DRAM.

Amdahl's law reminds us that the performance of an accelerator can be limited by the frequency of shipping data between the host memory and the accelerator memory. There will surely be applications that would benefit from the host CPU and the accelerators to be integrated into the same *system on a chip* (*SOC*).

Such a design is called an *IP block*, with IP standing for *Intellectual Property*, but a more descriptive name might be portable design block. IP blocks are typically specified in a hardware description language like Verilog or VHDL to be integrated into the SOC. IP blocks enable a marketplace where many companies make IP blocks that other companies can buy to build the SOCs for their applications without having to design everything themselves. Figure 7.32 below indicates the importance of IP blocks by plotting the number of IP blocks across generations of Apple PMD SOCs; they tripled in just four years. Another indication of the importance of IP blocks is that the Apple CPU and GPU use only one-third of the area of the Apple SOCs, with IP blocks occupying the remainder (Shao and Brooks, 2015).

Designing an SOC is like city planning, where independent groups lobby for limited resources, and finding that the right compromise is difficult. CPUs, GPUs, caches, video encoders, and so on have adjustable designs that can shrink or expand to use more or less area and energy to deliver more or less performance. Budgets will vary depending on whether the SOC is for tablets or for IoT. Thus an IP block must be scalable in area, energy, and performance. Moreover, it is especially important for a new IP block to offer a small resource version because it may not already have a well-established foothold in the SOC ecosystem; adoption is much easier if the initial resource request can be modest.

It will be interesting to see whether the attractiveness of integration leads to most data center processors coming from traditional CPU companies with IP accelerators integrated into the CPU die, or whether systems companies will continue deploying accelerators over I/O buses, which has been the norm from 2017 to 2024.

Energy and Carbon Emissions of Machine Learning Training

Concerns about global climate change have rightfully led to investigation of the environmental impact of ML. Figure 7.25 shows how by following best practices we can reduce energy use by up to 100x and emissions by up to 1000x (Patterson 2022). The example starts with the famous Transformer model in 2017 that was trained on a GPU in the average data center. The figures show the savings of applying best practices in 2019 and then again in 2021. Paying tribute to the 3Cs mnemonic of caches, the *4Ms of ML Energy Efficiency* are:

1. **Model**: The first M is the DNN model, starting with the Transformer model. The best practice is to pick the latest and most efficient model that solves the same problem equally well. In 2019, Google researchers performed a

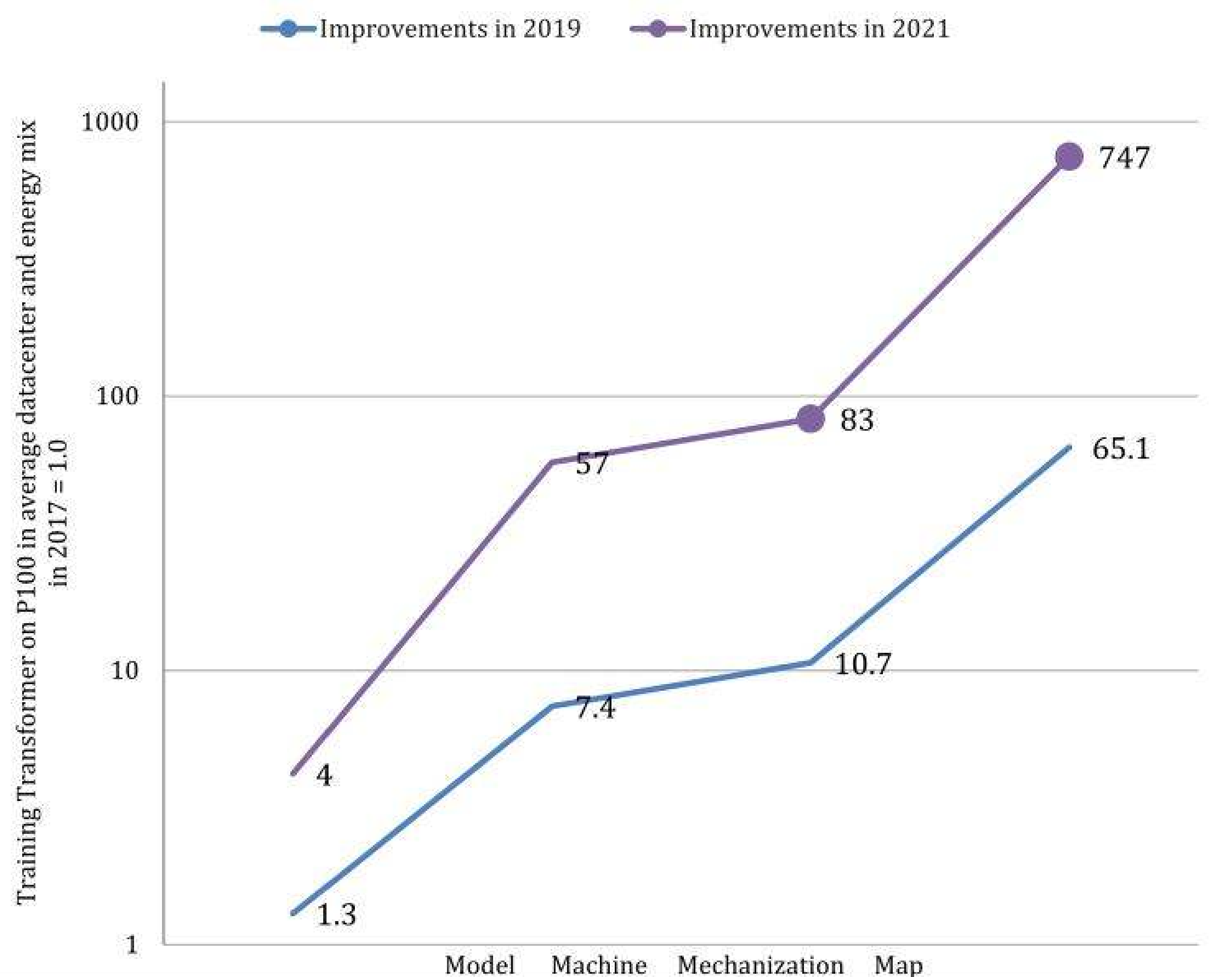

Figure 7.25 Improvements in gross energy and CO_2 emissions since 2017 from applying the best practices factored into the 4 Ms (Patterson, 2022).

Neural Architecture Search (NAS) to discover the Evolved Transformer model that is 1.3X faster but of equal quality (So, 2019). In 2021 the same researchers did another NAS to discover Primer, which is 4.2x faster than Transformer (3.2x faster than Evolved Transformer) again with equal quality (So, 2021). Note that NASs are done rarely, once per problem domain and architecture search space, not once per model training. The results are traditionally published and the model is made open source so that all ML practitioners can benefit from the NAS. The energy and emissions for the NAS are more than paid back by the subsequent reuse of the more efficient model discovered by the NAS.

2. **Machine.** This practice is to pick the most efficient hardware on which to train the model, so the second M is for Machine. Compared to the P100 GPU in 2017, TPU v2 in 2019 is 5.7x faster. In 2021 TPU v4 is 14x faster (2.6x vs. TPU v2). Combined with the first M, the energy and emissions reduction over Transformer on a P100 are 7.4x in 2019 and 57x in 2021.

3. **Mechanization (data center PUE).** Google cloud data centers have a Power Utilization Efficiency (PUE) of ~1.1, while the average data center has a PUE of ~1.6, so the gain is ~1.4. Combined with the first two Ms, the energy and emissions reduction over Transformer on a P100 in a data center with an average PUE are 10.7x in 2019 and 83x in 2021.

4. **Map.** As mentioned in Chapter 1, the location of a data center has a huge impact on how clean its energy supply is, up to 5x–10x. Compared to the average energy supply, the Google Iowa data center in 2019 had 6x lower emissions per kilowatt-hour. The Google Oklahoma data center in 2021 was 9x lower (1.5x vs. Iowa in 2019.) Combined with the first three Ms, the emissions reduction over Transformer on a P100 in a data center with an average PUE and average cleanliness of energy are 65x in 2019 and 747x in 2021.

Following the 4 M best practices gives us the best chance to deliver on the amazing potential of ML in a sustainable way.

An Open Instruction Set

One challenge for designers of DSAs is determining how to collaborate with a CPU to run the rest of the application. If it's going to be on the same SOC, then a major decision is which CPU instruction set to choose, because until recently, virtually every instruction set belonged to a single company. Previously, the practical first step of an SOC was to sign a contract with a company to lock in the instruction set.

The alternative was to design your own custom RISC processor and to port a compiler and libraries to it. The cost and hassle of licensing IP cores led to a surprisingly large number of do-it-yourself simple RISC processors in SOCs. One AMD engineer estimated that there were 12 instruction sets in a modern microprocessor!

RISC-V offers a third path: a viable free and open instruction set with plenty of opcode space reserved for adding instructions for domain-specific coprocessors, which enables the previously mentioned tighter integration between CPUs and DSAs. To support DSAs, RISC-V has four opcodes reserved for custom instructions, each of which can be extended with a 10-bit function code, leaving room for literally thousands of such instructions. SOC designers can now select a standard instruction set that comes with a large base of support software without having to sign a contract.

They still have to pick the instruction set early in the design, but they don't have to pick one company and sign a contract. They can design a RISC-V core themselves, they can buy one from the several companies that sell RISC-V IP blocks, or they can download one of the free open-source RISC-V IP blocks developed by others. The last option is analogous to open-source software, which offers web browsers, compilers, operating systems, and so on that volunteers maintain for users to download and use for free.

As a bonus, the open nature of the instruction set improves the business case for small companies offering RISC-V technology because customers don't have to worry about the long-term viability of a company with its own unique instruction set.

Another attraction of RISC-V for DSAs is that the instruction set is not as important as it is for general-purpose processors. If DSAs are programmed at higher levels using abstractions like data acyclic graphs (DAGs) or parallel patterns—as is the case for JAX, PyTorch, and TensorFlow—then there is less to do at the instruction set level. Moreover, in a world where performance-cost and energy-cost advances come from adding DSAs, binary compatibility may not play as important a role as in the past.

At the time of this writing, RISC-V is likely the third most important instruction set, following ARM and x86. The future of the open RISC-V instruction set appears promising. (We wish we could peer into the future and learn the status of RISC-V from now to the next edition of this book!)

7.9 Putting It All Together: Comparing DNN Accelerators

We now use the DNN domain to compare the cost-performance of the accelerators in this chapter.[2] We start with a thorough comparison of TPU v4 lite to T4 for inference benchmarks and then compare TPU v4, A100, and IPU Bow on training. We then add a brief performance evaluation of the Samsung NPU used in smartphones by comparing it to an x86 CPU used in notebooks.

Inference: TPU v4 lite Versus T4

Figure 7.26 shows performance in Google data centers of TPU v4 lite relative to T4. The three inference benchmarks come from MLPerf. There are two ratings for each benchmark. For the server version, performance is queries per second within a latency bound that is tens to hundreds of milliseconds. For the offline version, performance is queries per second with no latency bound.

TPU v4 lite is 1.3X as fast as T4 for server benchmarks and 1.6X for offline benchmarks.

Figure 7.27 shows the difference between reported MLPerf performance and the results for Google data centers. As the T4 Turbo boost clock rate is 2.6X faster than the base clock rate, the cooling system is critical to T4 performance since the clock may slow when temperature rises. (Both T4 and TPU v4 lite are air cooled.) MLPerf Inference runs are required to last at least 1 minute, which is near the time it takes to heat up a chip and its heatsink fully. Google experiments found that an idle T4 at the lowest clock rate ran at 35°C. When running MLPerf on T4 at the maximum of 1.6 GHz, the temperature rises to 75°C in 30 seconds. Thereafter the chip stays at 75°C, with the clock speed varying between 0.9 and 1.3 GHz. Given the Google data center environment plus the need for ECC—which was optional for MLPerf Inference—Figure 7.27 shows the impact of running MLPerf

[2] This section is based in part upon the paper "Ten lessons from three generations shaped Google's TPU v4" (Jouppi et al., 2021), of which one of your book authors was a coauthor.

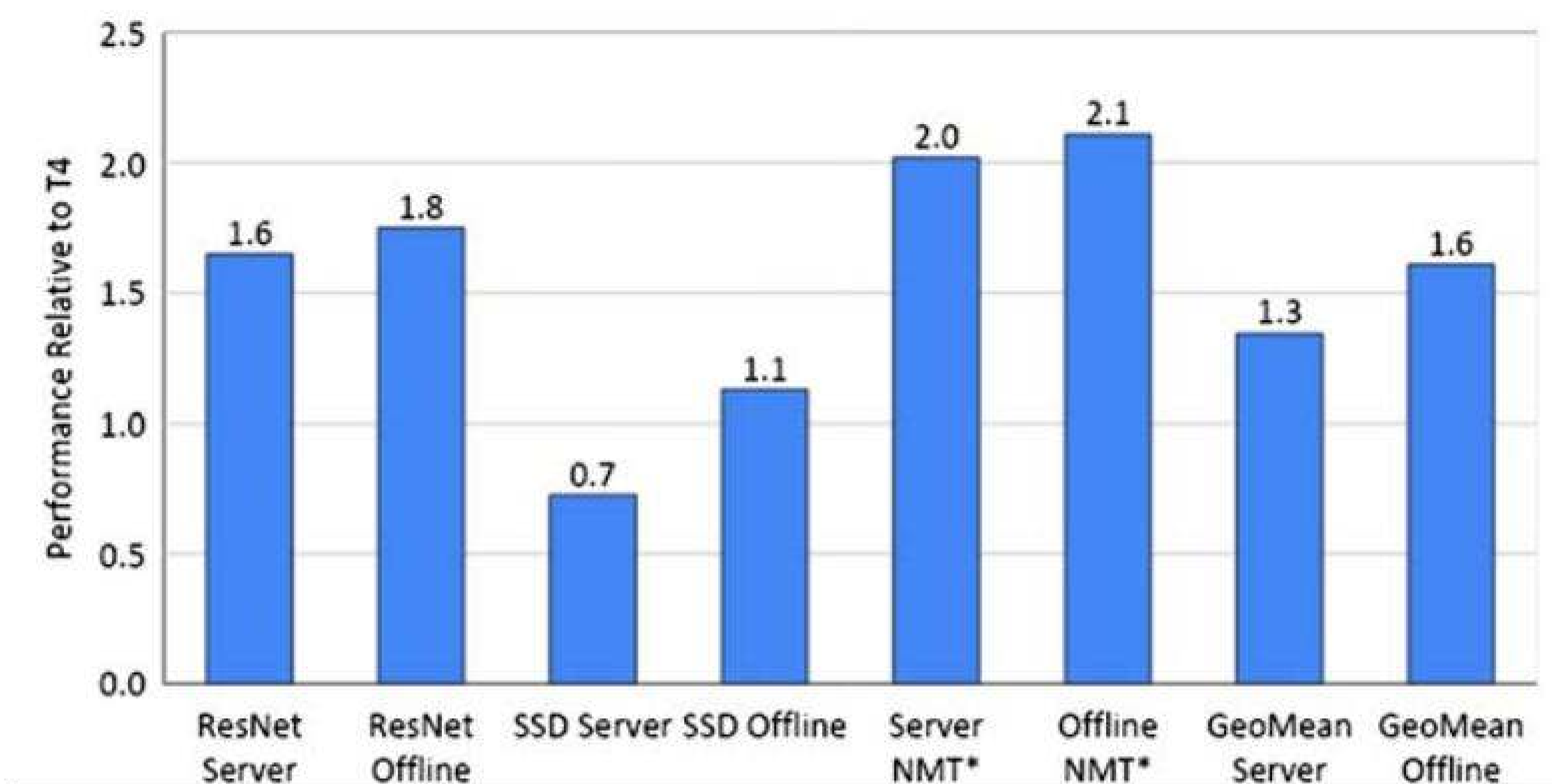

Figure 7.26 TPU v4 lite performance relative to T4 when running in Google data centers (Section 7.5). Note that the T4 uses int8 for ResNet and SSD and fp16 for NMT. MLPerf Inference 0.7 omits NMT, so the figure uses MLPerf Inference 0.5 results for it and 0.7 numbers for ResNet and SSD.

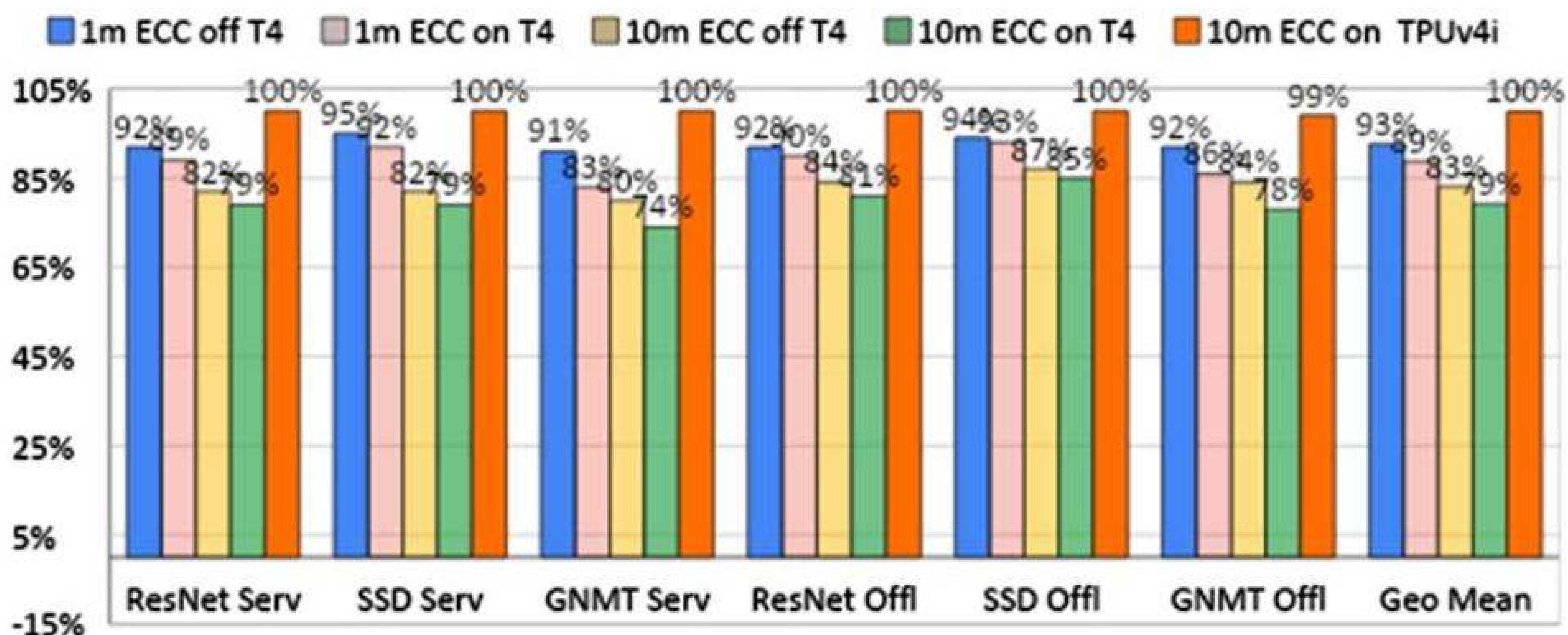

Figure 7.27 T4 and TPU v4 lite running unverified MLPerf Inference benchmarks 0.5/0.7 at Google with memory ECC off and on. NVIDIA's MLPerf score is 100% for T4 and 100% for TPU v4i is an (unverified) MLPerf score running in our data center. For the 1-minute case, the T4 was idle initially. It then ran MLPerf with ECC off and on for 1 minute at the fastest clock rate. (T4 offers inline ECC, which uses memory bandwidth.) For the 10-minute case, the machines are not idle beforehand. TPU v4i's speed is unchanged whether ECC is on or off or how long it runs.

Inference for 10 minutes with ECC enabled: T4 performance drops 19%–26% below its MLPerf Inference ratings.

In contrast, TPU v4i starts at ∼37°C and rises only ∼5°C over 10 minutes. Google purposely provisions enough power and cooling in datacenters to keep TPU v4i latency constant and includes extra memory for ECC so that it can always be on at no performance penalty.

Training: TPU v4 Versus A100 Versus IPU Bow

Figures 7.28 and 7.29 show performance for the three training systems as the systems scale for two MLPerf benchmarks: Resnet-50 and BERT. Vendors are free to pick the size of the system for which they want to report training results, so the

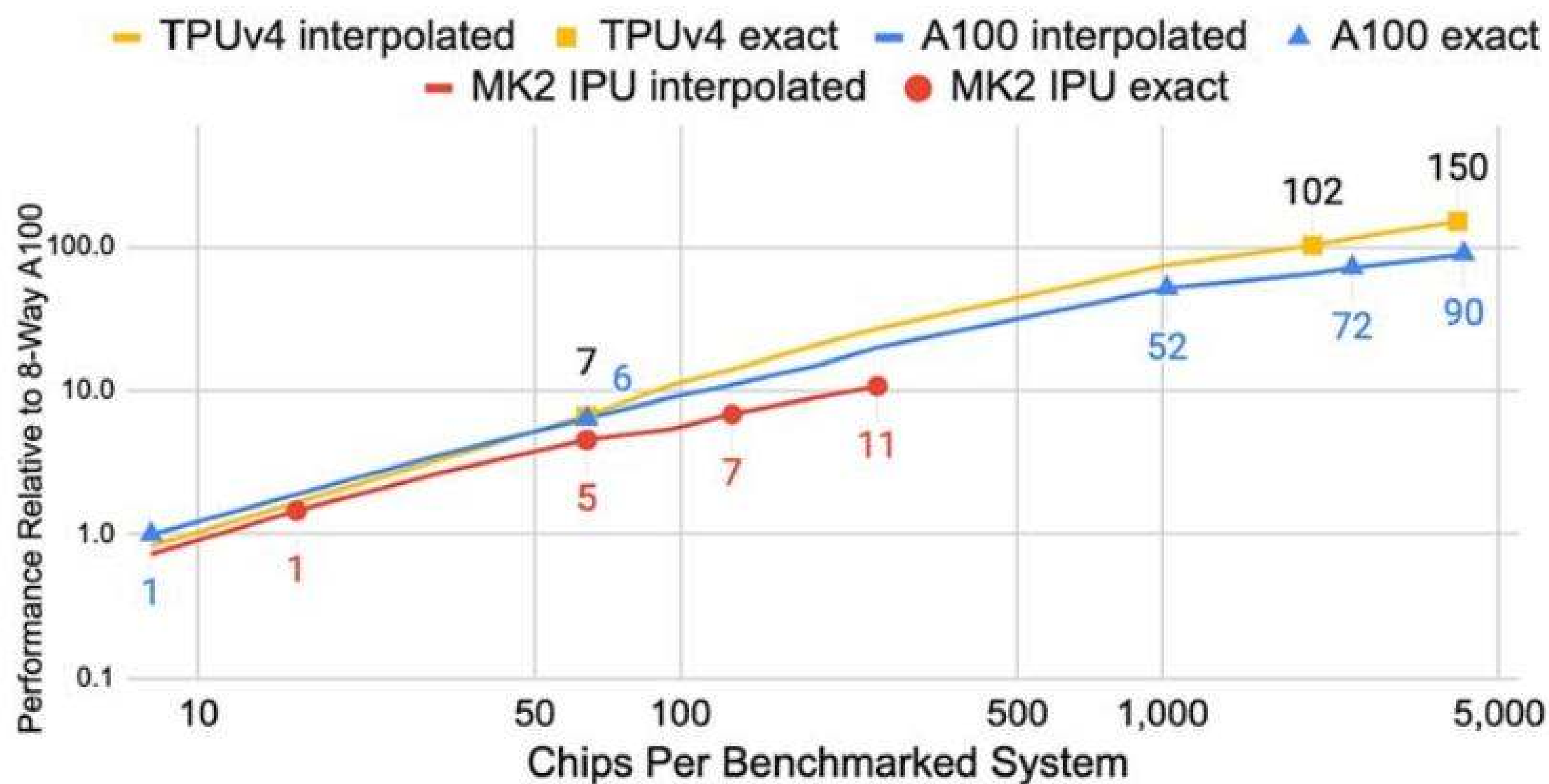

Figure 7.28 MLPerf training 2.0 performance for ResNet relative to 8-Way A100 GPU. The points are the reported results, and the lines are extrapolations of performance for intermediate sizes systems. For TPU v4, the results below 2048 chips are from MLPerf Training 1.0; the rest of the points for all systems are from MLPerf Training 2.0. Note that the axes are on a log scale.

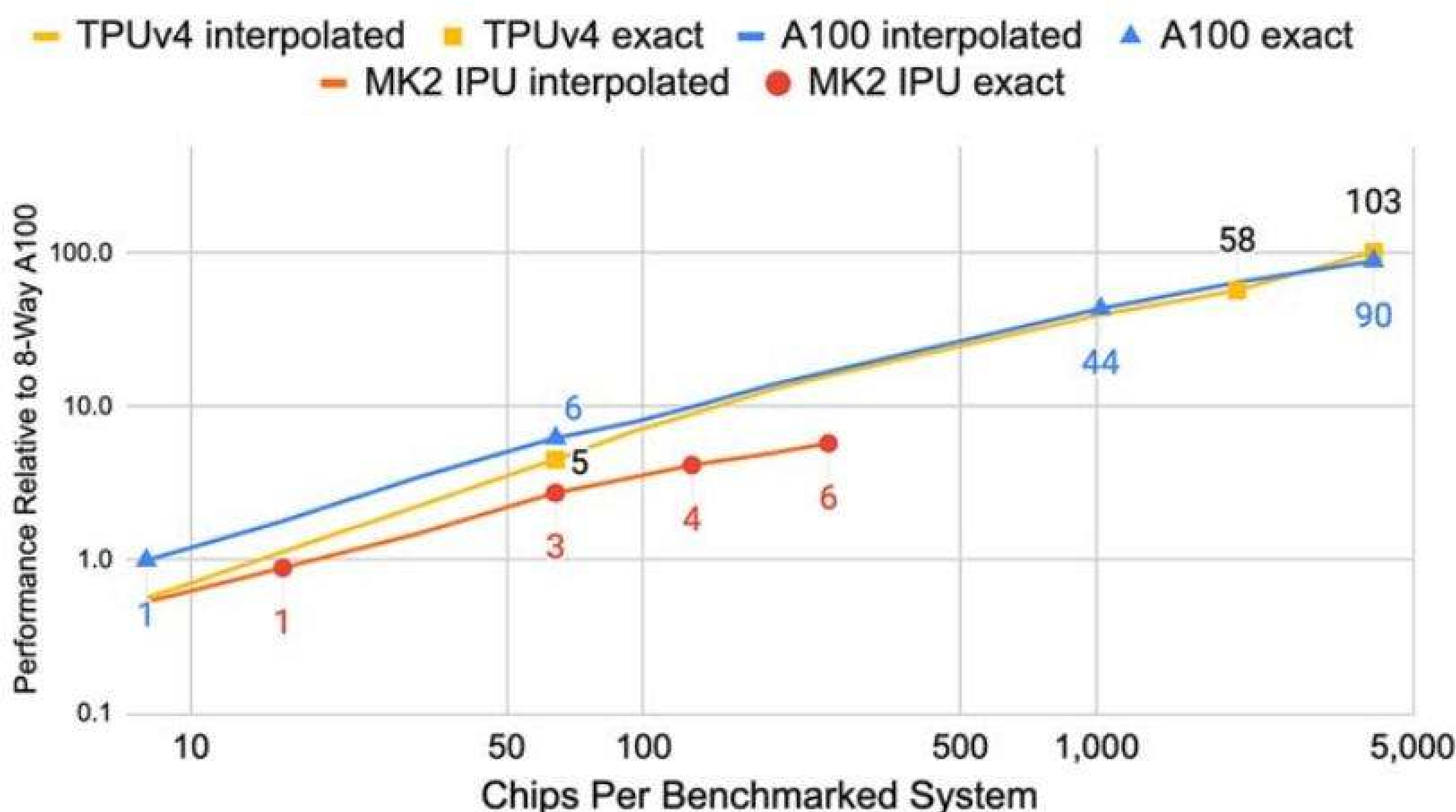

Figure 7.29 MLPerf training 2.0 performance for BERT relative to 8-Way A100 GPU. The points are the reported results, and the lines are extrapolations of performance for intermediate sizes systems. For TPU v4, the results below 2048 chips are from MLPerf Training 1.0; the rest of the points for all systems are from MLPerf Training 2.0. Note that the axes are on a log scale.

comparison is limited by the choices made by the three companies. We want to compare systems of equal size or cost in this section. The figures show the reported results as large points, while the lines between the points are extrapolations. MLPerf results for TPU v4 and A100 both scale to much larger systems than the IPU (4096 vs. 256 chips). TPU v4 is faster for ResNet-50 than the A100, and the A100 is faster than the IPU. For BERT, TPU v4 is a little faster than the A100, and the A100 is much faster than the IPU.

Inference: Samsung NPUv1 and NPUv2 Versus Intel Core i7 plus Iris Xe GPU

MLPerf also offers a set of inference benchmarks for mobile devices. The mobile benchmarks generally use fewer weights and have lower quality targets than the data center benchmarks to reduce the computation demands. Figure 7.30 shows the results of two MLPerf categories: phones for NPUv1 and NPUv2 and notebooks for the Intel Core i7.

This comparison is an apples-and-oranges one in that notebooks have higher budgets for power and die area and much faster clock rates (see Figure 7.22) by factors of 5x−25x. Nevertheless, the domain-specific NPU accelerators outperform the CPU by factors of 1.2x and 3.4x, respectively. One reason is the zero skipping in sparse matrices, which improves performance on average by a factor of 1.4x.

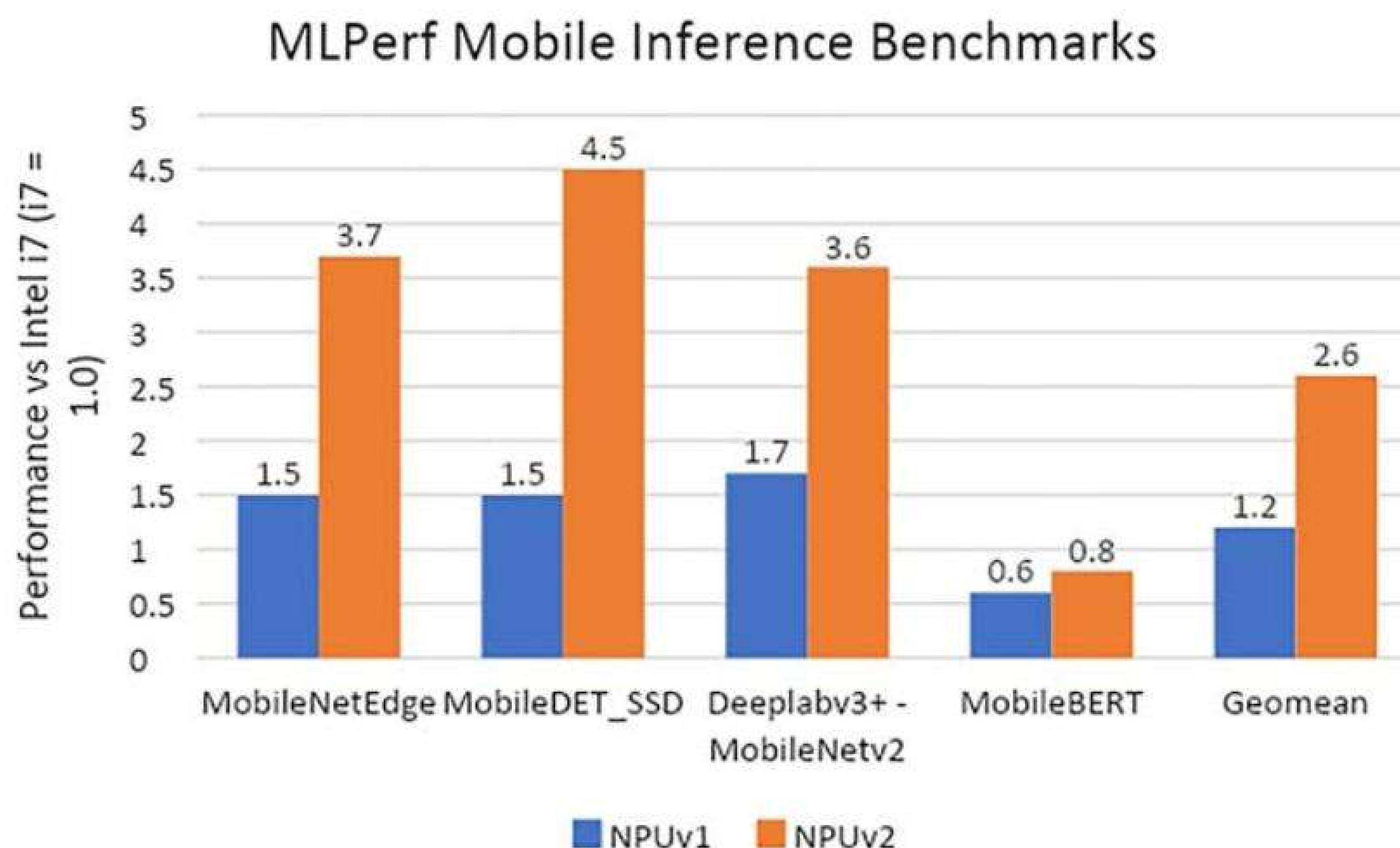

Figure 7.30 Performance of NPUv1 and NPUv2 relative to the Intel Core i7-11375H for the MLPerf Mobile Inference Single-Stream Benchmarks. (NPUv1 results are from MLPerf 1.0, Core i7 results are from MLPerf 1.1, and NPUv2 scores are from MLPerf 2.0.).

Note that MobileBERT underperforms on NPUs compared to the three other benchmarks. This benchmark requires computing using fp16. As NPUv1 doesn't support fp16, it was actually run using a DSP that was also in the SOC. It was run on NPUv2, but the performance advantage over the i7 is much less for fp16 than for integer data types.

7.10 Fallacies and Pitfalls

In this first decade of commercial DSAs for DNNs, fallacies and pitfalls abound.

Pitfall *Unlike standard software, DNNs can change rapidly.*

Figure 7.5 shows the change in the mix for inference of production DNN models at Google over 4 years. The first paper on BERT was published in 2018, and two years later it was more than a quarter of the workload. Moreover, even the model types are consistently growing rapidly. Over 4 years the inference applications grew in memory footprint and computation demand by on average 1.5x annually. For training, the growth has been even faster. In 2024 there is a great deal of excitement in LLM models, which have hundreds of billions to trillions of parameters. Two years after the GPT-3 paper, LLM models make up a quarter of the TPU v4 workload.

The challenge for the DSA architect is to specialize to increase performance for DNNs while remaining flexible enough to support rapid growth in existing models and to accommodate new DNNs as they are discovered.

Fallacy *For DNN hardware, inferences per second (IPS) is a fair summary performance metric.*

IPS is not appropriate as a single, overall performance summary for DNN inference hardware because it's simply the inverse of the complexity of the typical inference in the application (e.g., the number, size, and type of DNN layers). For example, TPU v1 ran the 4-layer MLP1 at 360,000 IPS but the 89-layer CNN1 at only 4700 IPS; thus TPU IPS varies by 75X! Therefore, using IPS as the single-speed summary is even more misleading for DNN accelerators than MIPS or FLOPS is for traditional processors, so IPS should be even more disparaged.

Unlike the last edition of this book, today we have MLPerf, an industry-standard benchmark for ML workloads. There is no longer an excuse to quote indirect indicators like IPS or peak operations per second when making performance claims. We used MLPerf results to act as a guide as to which DSAs we would cover in this chapter; all DSAs in this chapter have MLPerf scores.

Pitfall *It takes a long time for compilers for new DSAs to mature.*

The open-source compilers for traditional programming languages make it fairly easy to evaluate new conventional CPUs, but that is not the case for DSAs for DNNs. Companies like Google and NVIDIA have teams that built their own novel compiler and libraries for JAX, PyTorch, and TensorFlow. Although there have been numerous startup companies developing novel DSAs for DNNs since 2017, most startups have not yet submitted a single MLPerf result. Presumably, the reason is that their software stacks do not yet demonstrate the performance that their architects expected.

Fallacy *The low latency needs of inference means batch size must be 1.*

Figure 7.31 shows the latency constraints of MLPerf benchmarks and production applications along with their batch size on TPU v4 lite. Sensibly enough, the latency constraint is expressed as a time limit to which 99% of the queries must meet. TPU v4 lite was able to meet the latency constraints with some batch sizes into the hundreds. To keep things in perspective, unlike CPUs where microsecond latencies can sometimes lead to performance problems, inference latency is measured in milliseconds.

Pitfall *Not supporting multitenancy.*

Like CPUs, inference DSAs should support multitenancy. Sharing across an application can lower costs and reduce latency if an application uses many models. For example, translation DNNs need many language pairs and speech DNNs must handle several dialects. Another reason is simply good software engineering practices. Examples include trying new features on a fraction of customers, or slowly deploying a new release to reduce the chances of problematic software updates. Six of eight Google production inference workloads need multitenancy. The

Type	*DNN*	*Latency constraint*	*Batch size*
MLPerf	Resnet50	15 ms	16
	SSD	100 ms	4
	GNMT	250 ms	16
Production	MLP0	7 ms	512
	MLP1	20 ms	128
	CNN0	1 ms	16
	CNN1	32 ms	16
	RNN0	60 ms	8
	RNN1	10 ms	32
	BERT0	5 ms	128
	BERT1	10 ms	64

Figure 7.31 Latency constraints versus batch size on TPU v4 lite for MLPerf and production applications (Jouppi, 2021).

maximum memory footprint for three of these applications was 1300-3000 MB, far more than can fit in the SRAM of a single socket in 2024. And as mentioned above, the expectation is that the size of DNNs will get even bigger over time and that SRAM will improve slowly in future technologies (see Figure 1.18 in Chapter 1). One consequence of multitenancy for devices deployed in the cloud is that security features like enclaves are becoming requirements for protecting customers' data. For example, A100 and H100 GPUs from NVIDIA support enclaves.

Pitfall *Being ignorant of architecture history when designing a DSA.*

Ideas that didn't fly for general-purpose computing may be ideal for DSAs; thus history-aware architects could have a competitive edge. For DNN accelerators, two important architectural features date back to the 1980s: systolic arrays (Kung and Leiserson, 1980) and Very Long Instruction Words (Fisher, 1983). The former reduced the area and power of the large MXU in TPUs and the latter simplified exploitation of instruction-level parallelism. We advise mining the historical perspectives sections at the end of every chapter of this book to discover jewels that could enhance DSAs that you design.

Fallacy *There is no I/O during DNN training.*

Facebook developed a data storage and integration pipeline that feeds their ML clusters that consists of tens of thousands of GPUs (Zhao, 2022). *Deep learning recommendation models* (*DLRM*s) are widely used at Facebook and consume most of the ML training cycles. The demands for data storage and integration have grown rapidly; between 2020 and 2022 storage capacity grew 2X and the online ingestion bandwidth grew 4X. Energy use for DLRM across three clusters at Facebook averages 28% for storage, 22% for preprocessing, and 50% for training.

Fallacy *The training of a DNN model is equal to the lifetime emissions of five cars.*

This headline appeared in *MIT Review* and *New Scientist* in 2019 and has since appeared in several research papers. This claim was made for the training of the Evolved Transformer (see Section 7.9 and Figure 7.25). The origin of this claim started with a paper (Strubell 2019) that tried to estimate the carbon emissions for the NAS that discovered Evolved Transformer, which the original paper (So 2019) did not include. As they did not have access to the 4Ms data for Google (where the NAS was performed), they instead used a GPU and average numbers for data center PUE and energy cleanliness. This understandable substitution raised the emissions estimate by 5X over what the NAS actually was for Google using TPU v2 and Google data centers. There was also confusion about the computational cost of NAS. The Evolved Transformer NAS used a small proxy task to search for the best models to save time, money, and energy and then scaled up the found models to full size. However, the external researchers assumed the search was done with full-size tasks. The resulting computation estimate for the

NAS was another 18.7x too high. The overall overshoot was 88x (5 × 18.7) higher than Google's actual NAS. Unfortunately, the journalists for *MIT Review* and *New Scientist* and many subsequent authors that cite the paper exacerbated the error by confusing the one-time cost of the Evolved Transformer NAS with the relatively tiny "every-time" cost that is incurred from training. The NAS produced 1300x as many emissions to discover the Evolved Transformer as it does to train it. Thus the training of the Evolved Transformer was actually ~120,000x smaller (5 × 18.7 × 1300) than the claimed five times the lifetime emissions of a car (2.4 kg instead of 284,019 kg). If we factor in the further improvements using the Primer model using TPU v4 running in Google's Oklahoma data center (see Figure 7.25), the emissions from training a model equivalent to Transformer drop to 0.2 kg, or 1,400,000x smaller than five car lifetime emissions.

7.11 Concluding Remarks

In this chapter we've seen several commercial examples of the recent shift from the traditional goal of improving general-purpose computers so that all programs benefit to accelerating a subset of programs with DSAs. For at least the past decade, architecture researchers have been publishing innovations based on simulations using limited benchmarks claiming improvements for general-purpose processors of *10% or less,* while companies are now reporting gains for DSA hardware products of *3−10 times or more*. Not only do they provide impressive speedups, but they also significantly reduce the energy and emissions impact of ML.

We think that is a sign that the field is undergoing a transformation, and we are seeing a renaissance in architecture innovation because of:

- the historic end of both Dennard scaling and Moore's law, which means improving cost-energy-performance will require innovation in computer architecture; and
- the potential upside of DSAs and their synergy with domain-specific programming languages.

In particular, at the time of this writing of this seventh edition, the excitement of the potential of artificial intelligence has shifted the focus of data center investments from primarily being towards CPU servers to primarily being for AI DSAs.

We believe that many architecture researchers will build DSAs that will raise the bar still higher than those discussed in this chapter. And we can't wait to see what the computer architecture world will look like by the next edition of this book!

7.12 Historical Perspectives and References

Section M.9 (available online) covers the development of DSAs.

Case Studies and Exercises by Cliff Young

Case Study 1: Matrix Multiplication

Concepts illustrated by this case study

- Structure of matrix multiplication operations

Matrix multiplication, which you might have encountered in high-school algebra, is the fundamental compute-intensive operation of the current deep learning revolution. Like scalar multiplication, it has two inputs and one output, but in this case each of the three operands is a two-dimensional matrix. We usually describe the shape of a matrix by giving its count of rows followed by its count of columns. So, for example, a 3×5 matrix has 3 rows and 5 columns, with 15 elements in total.

When we multiply matrices, the shapes must match up. Following FORTRAN conventions, we usually talk about a matrix multiplication $C = AB$ with result matrix C and source matrices A and B. C is an $m \times n$ matrix, while A and B are $m \times k$ and $k \times n$ matrices, respectively. The key thing to note is that just three scalar variables, m, k, and n, determine everything about the geometry of the matrix multiplication operation. The descriptions of the shapes mean that C and A have the same number of rows, m; that C and B have the same number of columns, n; and that A must have the same number of columns as B has rows.

This soup of letters can be hard to keep track of. One can visualize a matrix multiplication operation as corresponding to a rectangular prism whose faces are the three matrices A, B, and C (Figure 7.33). The three scalars determine the length of the edges of the rectangular prism. The work done by matrix multiplication corresponds to the volume of the prism. Think of each value in the input matrices as being broadcast through the prism in a direction perpendicular to the face of the prism. Wherever two values from A and B meet, we multiply them. Then what do we do to construct the output, C? For each location in the output, take the set of products that are in the line perpendicular to C's face of the prism and add them up, to produce that value in C.

The code for a very simple matrix multiplication, sometimes called a GEMM for GEneral Matrix Multiplication, follows. GEMM is part of the Basic Linear Algebra Subroutines (BLAS, pronounced "blahs") libraries, a standard library for linear algebra.

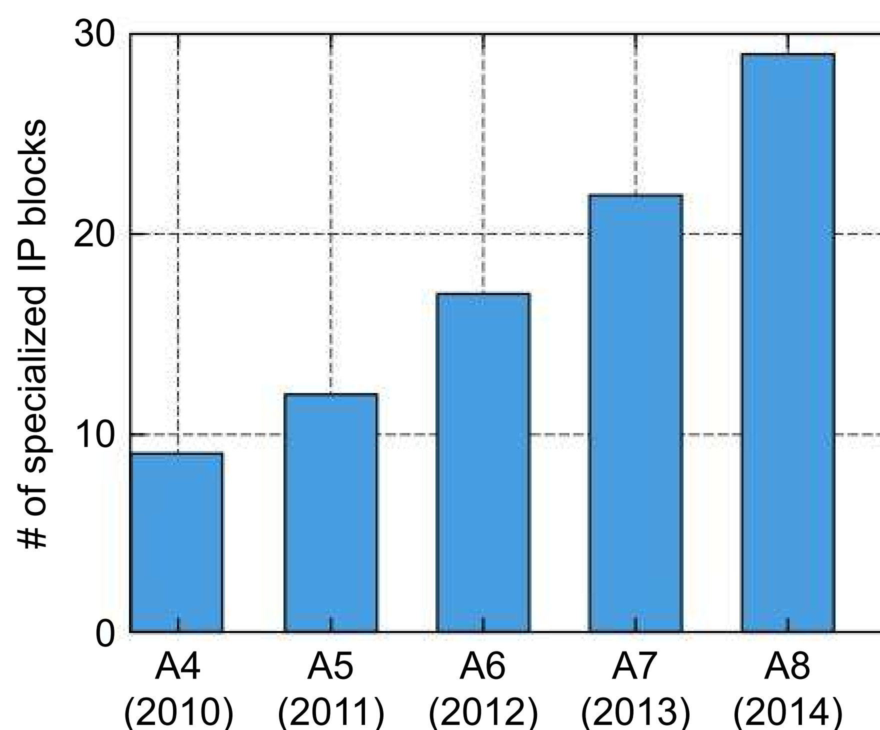

Figure 7.32 Number of IP blocks in Apple SOCs for the iPhone and iPad between 2010 and 2014. From Shao, Yakun Sophia, and David Brooks. Research infrastructures for hardware accelerators. Springer Nature, 2022.

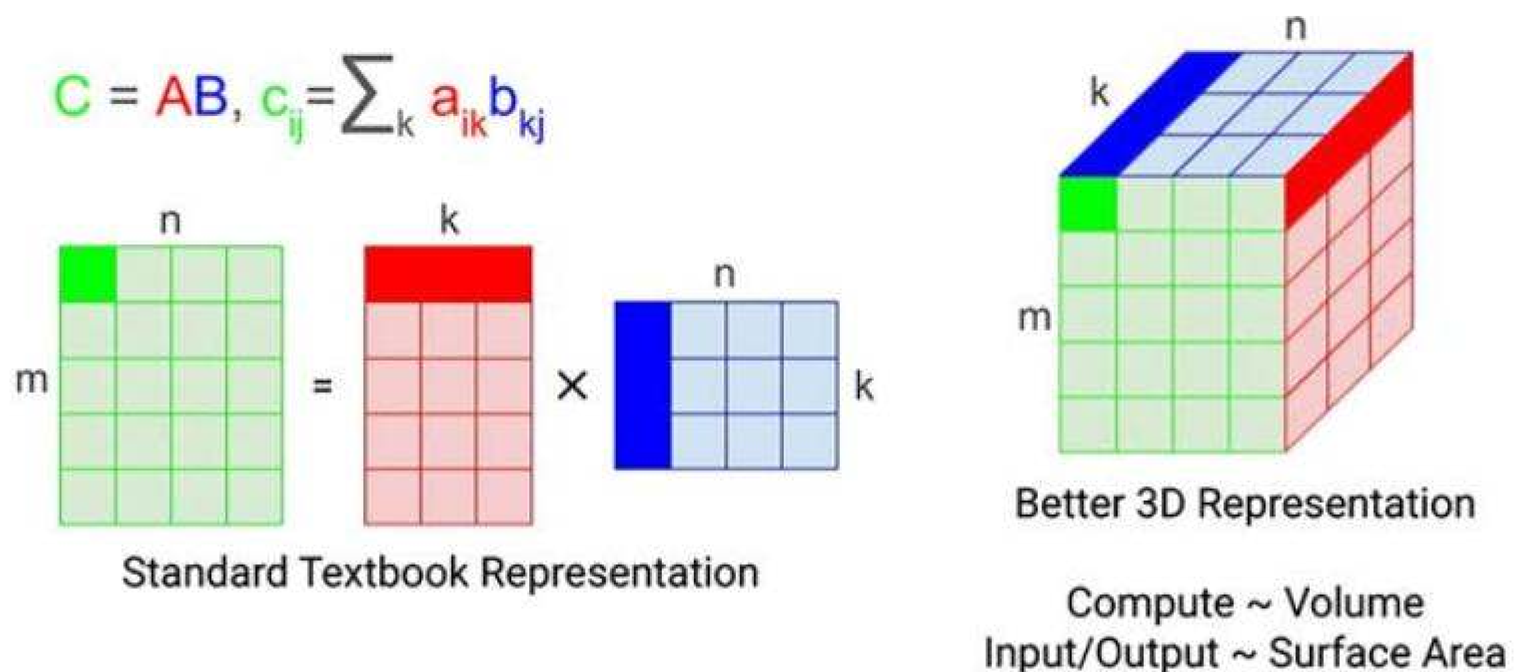

Figure 7.33 Illustration of matrix multiplication of matrix A (m × k) by matrix B (k × n) producing matrix C (m × n).

```
const int M, N, K;
float a[M][K], b[K][N], c[M][K];

void GEMM(const float &a[M][], const float &b[K][], float &c[M][]) {
 for (int i = 0; i < M; ++i)
 for (int j = 0; j < N; ++j)
  for (int k = 0; k < K; ++k)
  c[i][j] += a[i][k] * b[k][j];
}
```

7.1 [10/20/10/10/20] < 7.3,7.4 >

a. [10] Suppose that M, N, and K are all equal. What is the asymptotic complexity in time of this algorithm (if you don't know big-O notation, write a formula for the number of adds and multiplications in terms of M, N, and K)? What is the asymptotic complexity in space of the arguments? What does this mean for the operational intensity of matrix multiplication as M, N, and K grow large?

b. [20] Suppose that M=3, N=4, and K=5 so that each of the dimensions are relatively prime. Write out the order of accesses to memory locations in each of the three matrices A, B, and C (you might start with two-dimensional indices, then translate those to memory addresses or offsets from the start of each matrix). For which matrices are the elements accessed sequentially? Which are not? Assume row-major (C-language) memory ordering.

c. [10] Suppose that you build a new matrix B^T (indicating the transpose of B) by swapping the rows and columns so that they are B[N][K] instead. So now the innermost statement of the loop looks like:

```
c[i][j] += a[i][k] * b_t[j][k];
```

Now, for which matrices are the elements accessed sequentially?

The inner loop body of the matrix multiplication:

```
c[i][j] += a[i][k] * b[k][j];
```

is called a *multiply-accumulate* operation, or MAC for short. We can write it as a short subroutine:

```
void MAC(float &d, const float &s1, const float &s2) {
 d += s1 * s2;
}
```

The high-performance computing standard is to count a MAC as two operations (presumably the multiplication and the addition), even if the ISA specifies a MAC as a single opcode (as IBM POWER does, requiring three source operands for the instruction). We can rewrite the matrix multiplication to use the MAC instead:

```
void GEMM_MAC(const float &a[M][], const float &b[K][], float &c[M][]) {
 for (int i = 0; i < M; ++i)
 for (int j = 0; j < N; ++j)
  for (int k = 0; k < K; ++k)
  MAC(c[i][j], a[i][k], b[k][j]);
}
```

d. [10] For our M=3, N=4, K=5 case, how many MACs does the routine perform? How many operations?

e. [20] Suppose we had a hardware unit that could multiply two 100×100 matrices per cycle. If that unit ran at 1GHz, how many MACs per second would it perform? How many operations per second would that be?

Decompositions

As a step toward the next formulation of matrix multiplication, let's rewrite our GEMM_MAC to take the B argument transposed, which is to say, with all the same values but with its rows and columns swapped. Since B is a KxN matrix, its transpose, B^T, is an NxK matrix.

```
void GEMM_MAC_BT(const float &a[M][], const float &b_t[N][], float &c[M][]) {
 for (int i = 0; i < M; ++i)
 for (int j = 0; j < N; ++j)
  for (int k = 0; k < K; ++k)
  MAC(c[i][j], a[i][k], b_t[j][k]);
```

A nice side-effect of this transposition is that the minor dimension (fastest-incrementing) of the A and B matrices in memory is now k for both matrices.

Dot or Inner Product

Consider the *dot product* operation. Dot takes two vectors, multiplies them pair-wise element-by-element, adds up the element-wise products, and returns the sum.

```
 float dot(const float &x[], const float &y[], int len) {
 float sum = 0.;
 for (int i = 0; i < len; ++i)
 sum += x[i] * y[i];
 return sum;
}

void GEMM_dot(const float &a[M][], const float &b_t[N][], float &c[M][]) {
 for (int i = 0; i < M; ++i)
 for (int j = 0; j < N; ++j)
  c[i][j] = dot(a[i], b_t[j], K);
}
```

7.2 [25/25/25] < 7.3,7.4 >

a. [25] Rewrite the GEMM_MAC_BT routine so that instead of using MAC, it uses a dot product in the loop nest. How many times will you call the dot routine?
b. [25] Suppose that instead of a software version of dot that takes a *len* parameter, you have a hardware version that always requires len to be a fixed length, say 2. Rewrite your code to use such a dot2 routine. How many times does it get called? What should you do if K is not even?

SAXPY

A slightly different decomposition uses *SAXPY* (for *Single-precision Alpha times X Plus Y*). SAXPY takes an input vector and a scalar, scales each element of the

input vector by the scalar, then positionally adds the resulting products into the output.

```
void saxpy(float alpha, const float &x[], float &y[], int len) {
 for (int i = 0; i < len; ++i)
 y[i] += alpha * x[i];
}
```

c. [25] Rewrite the GEMM routine so that instead of using MAC, it uses saxpy in the loop nest.

Outer Product

The *outer product* of two vectors is a matrix that consists of all possible pairs of products between the two vectors. As with the GEMM routine, the convention is to reuse the result matrix as both an input and output, adding the outer product to the matrix that was passed in. Also note that we are applying a bit of C indexing sleight-of-hand in this routine, by passing *result* as a raw pointer and then manually doing the index calculations for *result*.

```
void outer_product(const float &x[], const float &y[], int len_x, int len_y, float *result) {
 for (int i = 0; i < len_x; ++i)
 for (int j = 0; j < len_y; ++j)
  result[i * len_y + j][29][30][31] += x[i] * y[j];
}
```

7.3 [25/25] < 7.3,7.4 >

a. [25] Rewrite the GEMM routine so that instead of using MAC, it uses outer_-product in the loop nest. The specific index formulation in the example does not matter—feel free to rewrite the outer_product routine to be friendly to the interface that you choose.
b. [25] Redraw Figure 7.33 to show the decomposition of the rectangular prism into dot-product, SAXPY, and outer product operations. You might explode the subcubes in the prism slightly to show the grouping into the three kinds of operations.

Case Study 2: Neural Network Layers

Concepts illustrated by this case study

- Structure of Deep Neural Networks

Figure 7.34 is a copy of Figure 2 from the classic AlexNet paper (Krizhevsky, 2012). The paper appeared in 2012 and yet contains almost all of the components that are still relevant in modern neural network design.

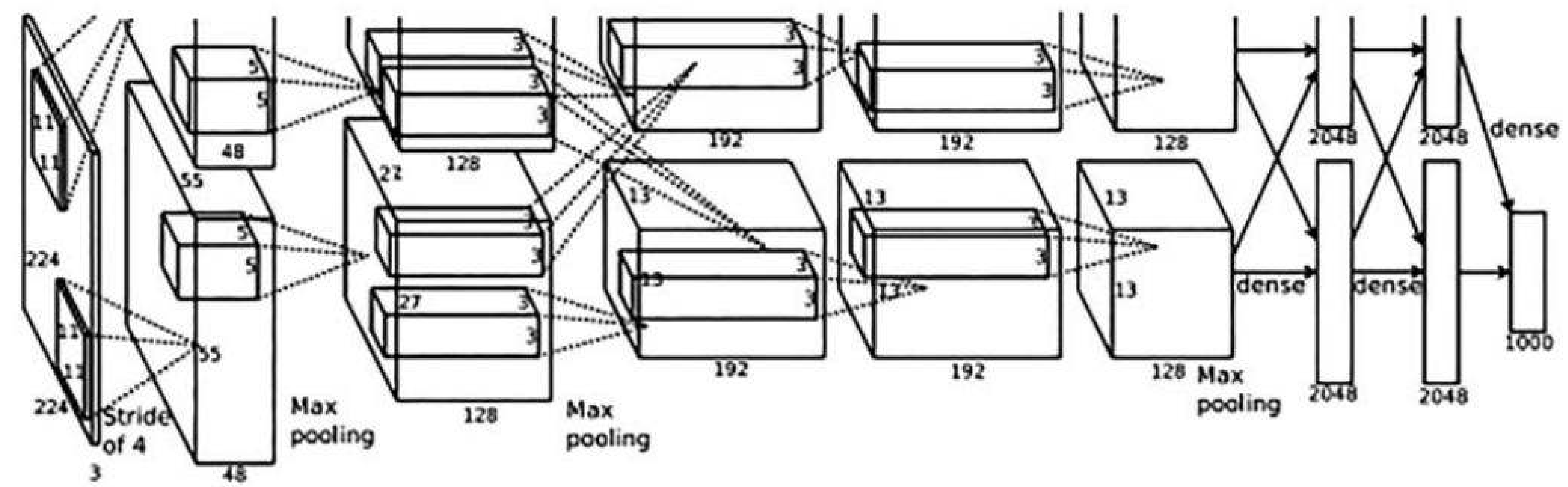

Figure 7.34 AlexNet software architecture (From Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems 25 (2012)). This figure shows "the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440—186,624—64,896—64,896—43,264—4096—4096—1000."

The diagram warrants a bit of explanation. For ease of reading, let's break it down piece by piece, and we will limit ourselves to the basics.

- The rectangular prisms (left and middle) and rectangles (right) in the diagram show *activations* (also called *feature maps*), which sometimes have three-dimensional structure (the input to AlexNet is a 224×224, RGB (x3) image) and sometimes have just one-dimensional structure (the five rectangles on the right are one-dimensional vectors of length 2048 or 1000). The last vector of activations, with its 1000 values, is the "output classifier" of AlexNet, which lists the probability that the input image corresponds to each of the 1000 categories of the ILSVRC contest.
- The *layers* and *weights* of AlexNet are not depicted explicitly. Instead they are in between each of the explicitly shown activations, and their shapes can be derived from the shape of the activations, as we will explain presently.
- The first five layers are between rectangular prisms and are *convolutional layers*, where each point in the output rectangular prism depends on a *receptive field* (the smaller prisms inside of each big prism). The number under each set of convolutional inputs or outputs (3, 48, 128, 192, 192, and 128) is the *depth* or feature depth of the layer. RGB for the input activations, but more depth representing more abstract concepts at each subsequent layer. The prisms are also labeled with a *spatial extent*, 224×224 for the input, but dropping rapidly to 55×55 after the first layer, 22×22 after the second layer, and 13×13 after the third and fourth layers.
- There are two mechanisms that reduce the spatial extent (which is kind of like image resolution) in the first few layers. One is striding: we only evaluate the

receptive field every 4 pixels rather than at every pixel location. A second is max pooling, which takes a 2×2 spatial set of activations and returns the maximum value among them.

- The last three layers are *fully connected layers*, which are basically matrix-vector multiplications. The sixth layer flattens the 13×13×128 into a 1D vector of two sets of 21632 activations (for a total of 43264), then does a matrix multiplication to project those down to 4096 activations. The seventh layer goes from 4096 to 4096 activations. The classifier layer further projects down from 4096 to the final 1000 output categories.
- At the classifier layer, a SoftMax function transforms the 1000-element vector of activations into a set of 1000 probabilities. SoftMax is a vector function, where the formula for it is: $v_i = e^{z_i} \Big/ \sum_{j=1}^{K} e^{x_j}$, where the K elements of the input vector are x_i and the K elements of the output vector are v_i. Don't be terribly intimidated by this formula—the denominator is the same for every output; it is a rescaling factor that ensures that the values in the output vector add up to 1.
- The diagram also shows parallelization over two GPUs, with limited communication between them. We'll come back to parallelization later.

7.4 [10/20/15//25/25/25/25] < 7.3,7.4 >

a. [10] How many weights are there in each convolutional layer? How many weights must be used to produce a single output activation?

b. [20] Derive the neuron (activation) counts for the convolutional layers, given the totals supplied in the caption of the diagram: 253,440 for the output of the first convolutional layer, and so on.

c. [15] The sixth, seventh, and eighth layers of AlexNet start with their data split across the two GPUs. How much data must be sent from each GPU to the other for these calculations? How many weights are resident on each GPU for these layers? How many operations are performed by these layers, for a single image?

d. [25] One unfortunate reality of deep learning practice is that the clean mathematical formulas that we might have learned for convolutions don't always fit perfectly in practice. We can see this at work in the layers that reduce spatial extent: convolutional kernel striding and max pooling. We might have expected a convolution of a k-length filter over x-length inputs to produce x-k+1 outputs, but some of the time a specification of a neural network layer appears to produce more than this expected number. The usual fix is to provide some amount of "zero padding" in the input to increase its effective size so that the desired output size is reached. For the striding and max pool layers in the diagram, when do you need to add X or Y padding to make the accounting work out? Note that the max pooling might also need to have a stride.

e. [25] The SoftMax function might look like a complicated formula, but a small amount of playing with tiny examples can fuel intuition. The inputs to SoftMax are called *logits*; evaluate SoftMax of:

- a vector of four values, all equal to 2.0.
- the vector [1,2,3,4]
- a one-hot vector of length four
- a four-vector with two large values and two small values

After this, experiment with scaling the whole vector, adding the same value to the whole vector, and changing the relative sizes of elements in the vector. Qualitatively, what can you say about the effects of SoftMax on its inputs? Where will the bulk of the value of the output vector be concentrated?

f. [25] Generate some random 1000-vectors using uniform, Gaussian, and exponential distributions, then compute the SoftMax of those vectors. Graph the CDF (cumulative distribution function) of these outputs. How many inputs account for 50%, 95%, and 99% of the CDF?

g. [25] SoftMax can be numerically unstable, because the exponential function used in its evaluation grows quickly, so relatively small inputs to it can result in values that exceed the representable range of floating-point. How can you rewrite SoftMax to remove this numerical instability?

The one major element missing from AlexNet is the *attention* layer, which figures prominently in the paper that introduced the Transformer architecture, "Attention Is All You Need" (Vishwani, 2017). Algebraically attention looks like the formula

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Figure 7.35 illustrates attention graphically:

But this is a bit terse for us. Here's an explanation:

- The three inputs, Q, K, and V, stand for "Query," "Key," and "Value," respectively. These names are our hint that attention works like a context-addressable memory (CAM), a hardware structure that has a memory that holds key-value pairs, accepts a query (instead of an address in a normal memory), and, in the case where a key equals the query, returns the value that corresponds to the key. But unlike the Boolean, exact comparison and selection of a CAM, attention is softer, which allows backpropagation to work through it as it learns.
- To start, think of Q as a vector, not a matrix. One puzzling thing about deep learning is that an activation vector doesn't necessarily have a human-accessible "meaning;" researchers state this in a fancy way by saying that vector is in a *latent* space. The bottom MatMul in the figure, representing QK^T, is

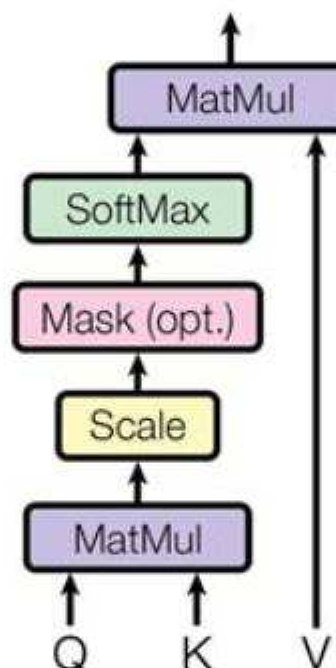

Figure 7.35 Graphical representation of attention (Part of Figure 2 from Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." Advances in neural information processing systems 30 (2017)).

(for now) a matrix-vector multiplication with K as the matrix and Q as the vector. Now view K as a set of vectors, each of which represents a key in the same latent space as the query vector. One way to look at QK^T is as computing the dot-product between Q and each of the vectors in the set of keys. And in this case, we are using dot-product like a comparison function—when two vectors point in mostly the same direction, their dot product is large, while if they point in orthogonal directions, their dot product is zero. So all that the first MatMul is doing is comparing the query vector Q, to the set of keys, and calculating how similar Q is to each key.

- The "scale" and optional "mask" steps are element-wise operations, which we'll skip over.
- The SoftMax does what we described in the previous section and turns the logits from the QK^T calculation into a probability distribution, which tells us how much weight to assign to each KV pair.
- The last matrix multiplication, SoftMax(...)V, scales the value vectors by the weights chosen by SoftMax. So if the SoftMax came out one-hot, this is like a content-addressable memory saying, "here's the hit." And if the SoftMax has a mixture of nonzero values, then it blends the best cases whose keys were close to the query, in proportion to how close they were. So the second MatMul returns a weighted blend of the values based on the query.
- Lastly, we can relax our starting simplification that we thought of Q as a vector and instead think of Q as a set of queries, where we process a whole set of queries in parallel and return a blended set of values, one for each query, from the whole attention block.

7.5 [25/15/15] < 7.3,7.4 >

a. [25] Let's explore how dot product works like a comparison function. Once again, to keep things simple, we'll work on four-vectors. It should be clear that the canonical orthogonal basis set of vectors, {[1,0,0,0], [0,1,0,0], [0,0,1,0], and [0,0,0,1]}, have the property that any pair of them have a dot product of zero. But the vectors that come out of our neural networks are rarely so clean, because they are learned through random processes. Randomly choose a 4-vector, and then construct yourself an orthogonal basis set that includes that four-vector. The four vectors in your set need not have unit length, but they do have to have the property that pairwise, their dot products are always zero.
b. [15] What happens when one key vector is equal to twice another key vector? What will the ratio of their dot products be?
c. [15] Similarly, what happens when one key vector is the negation of another? What will the ratio of their dot products be? And what happens to the negative one in the SoftMax?

Case Study 3: Numerical Representations

Concepts illustrated by this case study

- Numerical Representation
- Floating-Point Arithmetic

While matrix multiplication is the key large-scale operation in today's neural networks, *reduced precision* is part of the recipe that makes it affordable, in both chip area and power, to perform tera- and peta-operations per second. AlexNet was written for 32-bit floating-point values and ran on GPUs, but part of the high operation count of TPUv1 came from its use of 8-bit fixed-point arithmetic. For the sake of simplicity in doing exercises, we will restrict ourselves to 4-bit numerical representations, which means we'll be able to enumerate every representable value in each of the formats we discuss.

Integers

An unsigned integer is just an n-bit string of bits, where each bit is worth the corresponding power of two. So, if we name our four bits from most to least significant a, b, c, and d, then the value of a four-bit unsigned integer is:

$$uint4 = 8a + 4b + 2c + d$$

and we can use the four bits to represent integers in the range 0..15.

To represent signed integers, the *twos complement format* treats the most significant bit as representing not 8, but -8, with this formula:

$$int4 = -8a + 4b + 2c + d$$

and instead our four bits represent integers in the range -8..7. The most significant bit is also called the *sign bit*.

A third, more rarely used format is called *Excess-k*. In this format the binary representation looks like the formula for the unsigned formula, but with a "bias" term, k, which changes the interpretation of the bits in the value:

$$excessk = 8a + 4b + 2c + d + k$$

The bias term is not necessarily represented in hardware—it might be managed entirely by software, changing the interpretation of the value. An unsigned integer is the same as an excess-0 value.

Much more rarely, integers can be represented using a sign-magnitude representation, where the first bit indicates the sign while the remaining bits take their usual form:

$$signmag = (-1)^a \bullet (4b + 2c + d)$$

A 4-bit sign-magnitude representation has the interesting property that there are two representations of zero (effectively, positive and negative zero). This makes comparison tricky, since the two zeros should compare as equal even though they have different bit patterns.

7.6 [5/10/10/10/20] <7.4,7.5 >

a. [5]. Enumerate all the 4-bit integer values, for both unsigned and signed representations. This can just be a table; it's useful to do the representations in binary.
b. [10]. Negating a sign-magnitude representation is easy: just flip the sign bit. For the signed values, is there a common pattern or formula that relates x to -x? Ignore -8, as its negation, 8, isn't representable in a 4-bit signed value.
c. [10]. For each of the formats, if we add two 4-bit values, how many bits do we need to represent the sum?
d. [10]. If we multiply two 4-bit unsigned values, how many bits do we need to exactly represent the unsigned product?
e. [20]. Suppose we want to build a multiplier that takes two 4-bit inputs where the inputs are both unsigned, both signed, or one signed and one unsigned. What do we need to do to ensure correct outputs in each of the three cases?

Floating Point

If you remember scientific notation from high-school chemistry, it represented numbers such as Avogadro's number, the number of atoms or molecules in a mole, in a format that looked like 6.02×10^{23}. Floating-point representation is essentially the same thing done on computers, but instead of using decimals for the significant digits and the exponents, we use integers, which can be represented as bits.

Floating-point values have three sections:

- a *sign bit*, just like sign-magnitude integer representations.
- 0 or more *fractional bits*, which correspond to the ".02" part of Avogadro's number. They represent values to the right of the radix point (binary point in this case; decimal point in scientific notation). Taken as a group, the fraction bits look a lot like an unsigned integer. What happened to the 6 in 6.02? That turns out to be a key part of how floating-point efficiently represents real numbers, which we'll discuss more below.
- 0 or more *exponent bits*, which mostly represent a base-2 exponent in an excess-k format, although there can be some special values that capture corner cases of the format.

For so-called *normal* floating-point numbers, this formula gives the value represented by the number:

$$(-1)^{sign} \times 1.fraction \times 2^{exponent}$$

The sign term just sets the sign of the overall expression, positive or negative. Note that the middle term has an "implicit leading one," before the bits that make up the fraction. This implicit leading one means that this middle term represents values in the half-open interval [1.0,2.0) on the number line, where the formatting of the fractional bits means that we never represent 2.0, although with a large number of fraction bits (53 in IEEE double-precision format), we can get very close to it. The combination of the implicit leading one and the exponent term gives us the huge dynamic range of the floating-point format: each successive exponent value covers twice as much span on the real number line as the value before it.

You might ask: how do we represent zero in floating-point? That's where one of the special exponent values comes in, typically the all-zero exponent pattern. For that special pattern, we use the revised formula:

$$(-1)^{sign} \times 0.fraction \times 2^{minexp}$$

where *minexp* is the exponent value from the smallest normal floating-point number. Otherwise, all that has changed is that the implicit leading one has become an implicit leading zero. For nonzero fractions, the values in this format

are called *subnormal* or *denormal* values, and they play a key role in the numerical stability and error analysis of floating-point numbers.

7.7 [20/20/20/5/10] <7.4,7.5 >

a. [20]. In a 4-bit format with 1 sign bit, 1 fraction bit, and 2 exponent bits, list the values that all 16 possible 4-bit values correspond to on the real number line. Assume that the exponent is excess-(-2) so that an exponent of 0 indicates the subnormal range, 1 indicates an exponent of -1, 2 indicates an exponent of 0, and 3 indicates an exponent of 1.
b. [20]. Graph the 16 values on the real number line (it's fine to do only the positive or negative half, since they're symmetric). What role do the subnormals play on the number line?
c. [20]. Graph the representable values for a format with 1 sign bit, no fraction bits, and 3 exponent bits. What is the distribution of values that it can represent? For this format, how would you implement multiplication?
d. [5]. Graph the representable values for a format with 1 sign bit, 3 fraction bits, and no exponent bits. What does this format look like?
e. [10]. What about the format with 1 sign bit, 2 fraction bits, and 1 exponent bit? What is the single exponent bit doing in this case? How does this format differ from the previous one?

Case Study 4: Speeds, Feeds, and Scale

Concepts illustrated by this case study

- Capacities of memories and rates of computations ("speeds and feeds") for a simple neural network model
- Construction of a DSA for DNNs

Perhaps the best article in terms of system design wisdom per word is Jon Bentley's *The Back of the Envelope* (Bentley, 1999). In it, Bentley describes a style of computer system design that relies on pencil and paper (or literally, an old mailing envelope) and high-school algebra to quickly check whether a design is sane or even possible. Computer architects sometimes call such analyses "Speeds and Feeds." This section contains a number of such examples.

The first Tensor Processing Unit (TPU) from Google included a 256×256 systolic array matrix multiplier that peaked at 92 tera-operations per second, a figure that was astonishing for its time, 30x that of the contemporary two-chip K80 GPU and 90x that of contemporary hyperscaler CPUs. In the paper describing the TPU (Jouppi, 2017) the authors also identify the TPU's memory bandwidth as the crucial bandwidth bottleneck: with two channels of DDR3 memory, the TPU could only load about 30 GB/s of single-byte weights.

7.8 [10/10/10/10] < 7.3,7.4,7.5 >

a. [10]. Calculate the clock rate at which the TPU ran, assuming each cell of the systolic array performs one multiply-accumulate per cycle, which counts as two operations.
b. [10]. How long did it take for the TPU to load a weight tile of 64 KiB weights?
c. [10]. *Arithmetic Intensity* is defined as the number of times a value loaded is reused (usually as an operand to many different ALU operations). In steady-state operation how much arithmetic intensity must a weight enjoy for the TPU to reach both peak memory bandwidth and peak compute operations per second?

The later TPUv4 lite uses two HBM stacks to reach 614 GB/s of memory bandwidth (Jouppi, 2021). It has four 128×128 systolic arrays, but let's ignore that and treat the compute resources like they were a monolithic 256×256 systolic array.

d. [10], For a bf16 weight size, which is 2 bytes (compared to TPUv1's single byte per weight), how long would TPUv4 lite take to load 64 KiB weights? What arithmetic intensity does it require to run at both compute and memory peak?

According to the specifications at https://www.nvidia.com/en-us/data-center/h100/, the fastest version of the H100 GPU (SXM form factor) peaks at 32 TFLOPS/second of fp64, 367 TFLOPS of fp32, 1.979 TFLOPS of bf16 on its tensor core, and 3958 TFLOPS/second in either fp8 or int8 mode on the tensor core. There are reported to be 132 streaming multiprocessors in the H100, with clock rates somewhere between a base rate near 1 GHz and a boost rate near 2 GHz.

Another DNN DSA is the Cerebras WSE-2, which is an engineering marvel—it uses an entire manufacturing wafer to build a single compute unit, with an area of around 50 normal maximum-sized dies (which top out around 800 mm^2 of die area). The on-chip specifications are quite impressive: 40 GB of on-chip SRAM memory delivering 20 PB/s of bandwidth and 20 kW of power dissipation. Those huge resources are connected to the outside world by 12, 100-Gigabit Ethernet links.

7.9 [10/10/10] < 7.3,7.4,7.5 >

a. [10], Consider peak fp32 TFLOPS/second, which are likely measured at the peak clock of the GPU. How many fp32 functional units are there likely to be per SM, and in the whole H100 chip?
b. [10], Given how much higher the tensor core bf16 operations per second are on H100, what do you think the ratio of tensor core functional units to SIMD core functional units is?

c. [10] GPT-3 has 175 billion parameters, so even at one byte per weight, a GPT-3 instance needs 175 GB alone. How long would it take to stream those parameters from a memory system separate from the WSE onto the wafer?

Exercises

7.10. [5/5/10/10/10/10/15] <7.9>

In 2024 it costs \$2−\$4 per hour to rent one NVIDIA H100 GPU in the cloud. Press reports suggest that the purchase price (CAPEX) is around \$30,000. One part of OPEX is electricity costs. The TDP of an H100 is 700 W and the retail price of electricity in 2023 varied from \$0.10−\$0.20 per kWh, depending on the location in the United States.

a. [5] What does H100 rental annualize to, assuming the device is rented continuously?
b. [5] If the purchase price were the only cost to the cloud vendor for the H100, how long would it take to pay back the purchase price?
c. [10] What's the worst-case electric bill for operating such a machine for 1 year?
d. [10] How does the electricity cost change the payback time? What other factors might affect the lifetime total cost of ownership of a GPU?
e. [10] Assuming those other factors account for another operating expense equal to the electricity cost, what is the payback time now?
f. [10] Suppose that the GPU has a useful lifetime of about 8 years. How much profit might a cloud vendor make? How does that profit change if the H100 rental rate approximately halves, say, every 2 years when a new generation of GPUs becomes available?
g. [15] So far, we've ignored practical issues that affect profitability of H100 rental. For income, we've assumed one can rent the hardware 100% of the time over many years. What might be a more realistic percentage? How might that percentage change if you're using the H100s for inference versus training? For expenses, we omitted the CAPEX and OPEX of the CPU hosts that house the H100s, the cost of personnel of the company that rents those H100s, and the *cost of money*, which is the opportunity cost of holding cash instead of investing it. Estimate how these factors would affect predicted profits. If the revised profit estimate doesn't make economic sense, what might be plausible explanations?

7.11. [10/10/10/10] <7.2>

The first Anton molecular dynamics supercomputer typically simulated a box of water that was 64 Angstroms on a side. The computer itself might be approximated as a box with 1 meter side length. A single simulation step represented 2.5 femtoseconds of simulation time and took about 10 microseconds of wall-clock time.

The physics models used in molecular dynamics act as if every particle in the system exerts a force on every other particle in the system on each (“outer”) time step, requiring what amounts to a global synchronization across the entire computer.

a. [10] Calculate the spatial expansion factor from simulation space to hardware in real space.
b. [10] Calculate the temporal slowdown factor from simulated time to wall-clock time.
c. [10] These two previous answers come out surprisingly close. Is this just a coincidence, or is there some other limit that constrains them in some way? (Hint: the speed of light applies to both the simulated chemical system and the hardware that does the simulation.)
d. [10] Given these limits, what would it take to use a warehouse-scale supercomputer to perform molecular dynamics simulations at Anton rates? That is, what’s the fastest simulation step time that might be achieved with a machine 10^2 or 10^3 meters on a side? What about simulating on a world-spanning cloud service?

References

Bentley, J., 1999. The back of the envelope. IEEE Software 16 (5), 121.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., et al., 2020. Language models are few-shot learners. Adv. Neural Inf. Process. Syst. 33, 1877−1901.

Choquette, J., Giroux, O., Foley, D., 2018. Volta: performance and programmability. IEEE Micro 38 (2), 42−52.

Dean, J., 2022. A golden decade of deep learning: computing systems & applications. Daedalus 151 (2), 58−74.

Devlin, J., Chang, M.W., Lee, K., Toutanova, K., 2018. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), pp. 4171−4186.

Fedus, W., Zoph, B., Shazeer, N., 2022. Switch transformers: scaling to trillion parameter models with simple and efficient sparsity. J. Mach. Learn Res. 23 (120), 1−39.

Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., et al., 2017. In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 1−12.

Jouppi, N.P., Yoon, D.H., Kurian, G., Li, S., Patil, N., Laudon, J., et al., 2020. A domain-specific supercomputer for training deep neural networks. Commun. ACM 63 (7), 67−78.

Jouppi, N.P., Yoon, D.H., Ashcraft, M., Gottscho, M., Jablin, T.B., Kurian, G., et al., 2021. Ten lessons from three generations shaped Google’s TPU v4. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 1−14.

Jouppi, N., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., et al., 2023. TPU v4: an optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In: Proceedings of the 50th Annual International Symposium on Computer Architecture, pp. 1−14.

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., et al., 2021. Highly accurate protein structure prediction with AlphaFold. Nature 596 (7873), 583−589.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. 25.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al., 2019. Pytorch: an imperative style, high-performance deep learning library. Adv. Neural Inf. Process. Syst. 32.

Patterson, D., Gonzalez, J., Hölzle, U., Le, Q., Liang, C., Munguia, L.M., et al., 2022. The carbon footprint of machine learning training will plateau, then shrink. Computer 55 (7), 18–28.

Raihan, M.A., Goli, N., Aamodt, T.M., 2019. Modeling deep learning accelerator enabled GPUs. In: 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, pp. 79–92.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., Ommer, B., 2022. High-resolution image synthesis with latent diffusion models. In: Proceedings of the IEEE/CVF Conference On Computer Vision and Pattern Recognition, pp. 10684–10695.

So, D., Le, Q., Liang, C., 2019. The evolved transformer. In: International Conference on Machine Learning, pp. 5877–5886.

So, D., Mańke, W., Liu, H., Dai, Z., Shazeer, N., Le, Q.V., 2021. Searching for efficient transformers for language modeling. Adv. Neural Inf. Process. Syst. 34, 6010–6022.

Strubell, E., Ganesh, A., McCallum, A., 2019. Energy and policy considerations for deep learning in NLP. Annual Meeting of the Association for Computational Linguistics.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., et al., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.

Zhao, M., Agarwal, N., Basant, A., Gedik, B., Pan, S., Ozdal, M., et al., 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In: Proceedings of the 49th Annual International Symposium on Computer Architecture, pp. 1042–1057.

A.1	Introduction	A-2
A.2	Classifying Instruction Set Architectures	A-3
A.3	Memory Addressing	A-7
A.4	Type and Size of Operands	A-13
A.5	Operations in the Instruction Set	A-15
A.6	Instructions for Control Flow	A-16
A.7	Encoding an Instruction Set	A-21
A.8	Cross-Cutting Issues: The Role of Compilers	A-24
A.9	Putting It All Together: The RISC-V Architecture	A-33
A.10	Fallacies and Pitfalls	A-42
A.11	Concluding Remarks	A-46
A.12	Historical Perspective and References	A-47
	Exercises by Gregory D. Peterson	A-47