Chapter 7
Domain-Specific Architectures

© 2019 Elsevier Inc. All rights reserved.

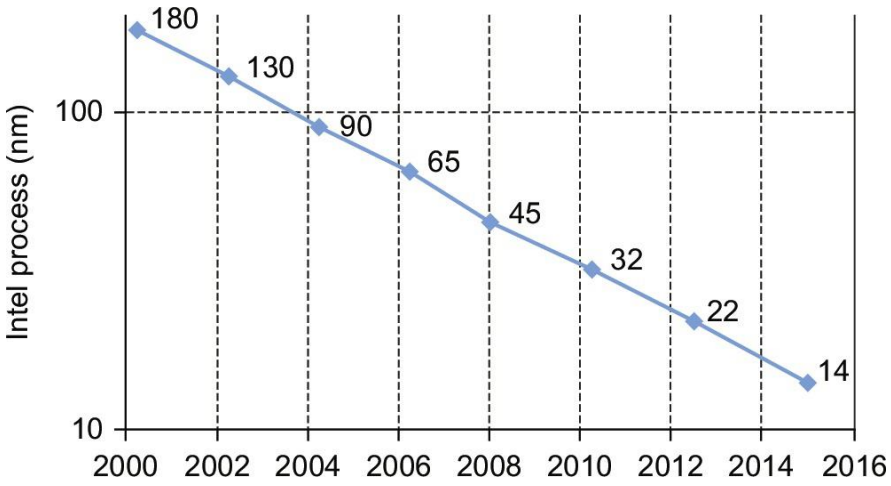

Figure 7.1 Time before new Intel semiconductor process technology measured in nm. The *y*-axis is log scale. Note that the time stretched previously from about 24 months per new process step to about 30 months since 2010.

© 2019 Elsevier Inc. All rights reserved.

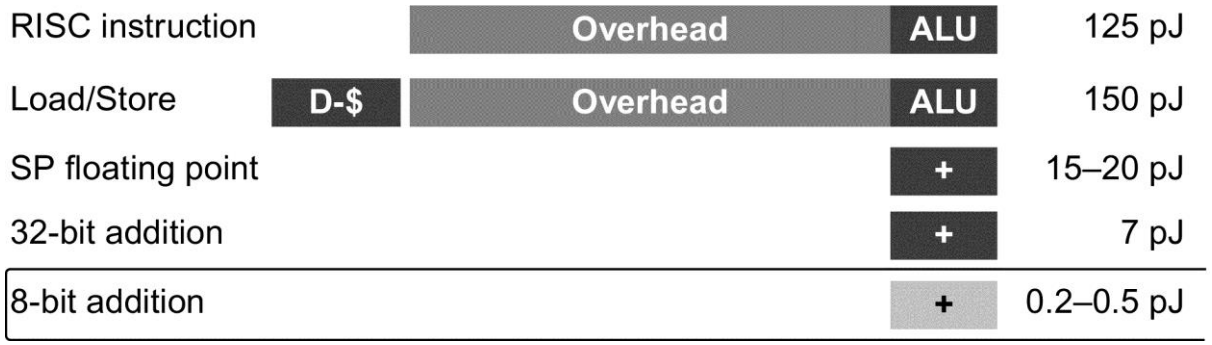

Figure 7.2 Energy costs in picoJoules for a 90 nm process to fetch instructions or access a data cache compared to the energy cost of arithmetic operations (Qadeer et al., 2015).

© 2019 Elsevier Inc. All rights reserved.

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

Figure 7.3 The four DSAs in this chapter and how closely they followed the five guidelines. Pixel Visual Core typically has 2–16 cores. The first implementation of Pixel Visual Core does not support 8-bit arithmetic.

© 2019 Elsevier Inc. All rights reserved.

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 *prediction*
	Long short-term memory	LSTM	An RNN well suited to classify, process, and predict time series. It is a hierarchical design consisting of modules called *cells*
	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*
	Weights	—	The values learned during training that are applied to inputs; also called *parameters*
TPU	Accumulators	—	The 4096 256 × 32-bit registers (4 MiB) that collect the output of the MMU and are input to the Activation Unit
	Activation unit	—	Performs the nonlinear functions (ReLU, sigmoid, hyperbolic tangent, max pool, and average pool). Its input comes from the Accumulators and its output goes to the Unified Buffer
	Matrix multiply unit	MMU	A systolic array of 256 × 256 8-bit arithmetic units that perform multiply-add. Its inputs are the Weight Memory and the Unified Buffer, and its output is the Accumulators
	Systolic array	—	An array of processing units that in lockstep input data from upstream neighbors, compute partial results, and pass some inputs and results to downstream neighbors
	Unified buffer	UB	A 24 MiB on-chip memory that holds the activations. It was sized to try to avoid spilling activations to DRAM when running a DNN
	Weight memory	—	An 8 MiB external DRAM chip containing the weights for the MMU. Weights are transferred to a *Weight FIFO* before entering the MMU

Figure 7.4 A handy guide to DSA terms used in Sections 7.3–7.6. Figure 7.29 on page 472 has a guide for Section 7.7.

© 2019 Elsevier Inc. All rights reserved.

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

Figure 7.5 Six DNN applications that represent 95% of DNN workloads for inference at Google in 2016, which we use in Section 7.9. The columns are the DNN name, the number of layers in the DNN, the number of weights, and operations per weight (operational intensity). Figure 7.41 on page 595 goes into more detail on these DNNs.

© 2019 Elsevier Inc. All rights reserved.

Type of data	Problem area	Size of benchmark's training set	DNN architecture	Hardware	Training time
text [1]	Word prediction (word2vec)	100 billion words (Wikipedia)	2-layer skip gram	1 NVIDIA Titan X GPU	6.2 hours
audio [2]	Speech recognition	2000 hours (Fisher Corpus)	11-layer RNN	1 NVIDIA K1200 GPU	3.5 days
images [3]	Image classification	1 million images (ImageNet)	22-layer CNN	1 NVIDIA K20 GPU	3 weeks
video [4]	activity recognition	1 million videos (Sports-1M)	8-layer CNN	10 NVIDIA GPUs	1 month

Figure 7.6 Training set sizes and training time for several DNNs (Iandola, 2016).

© 2019 Elsevier Inc. All rights reserved.

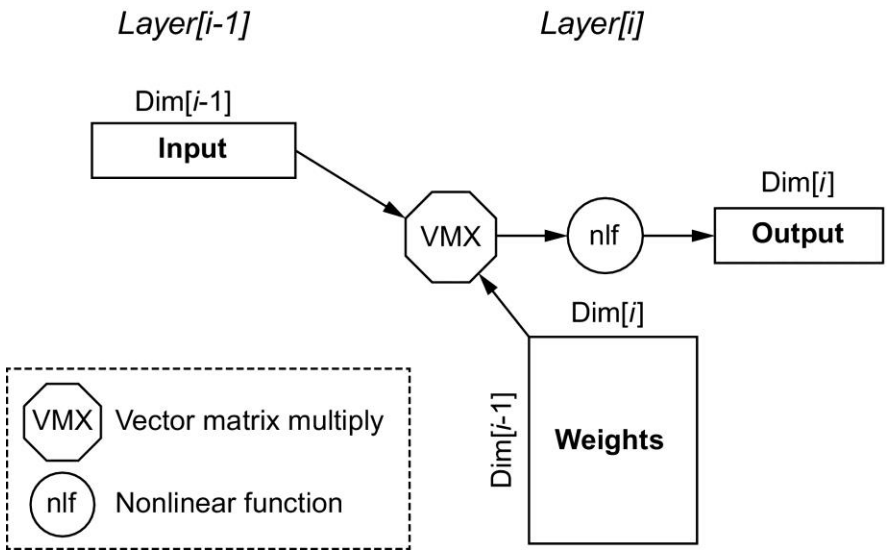

Figure 7.7 MLP showing the input Layer[*i*− 1] on the left and the output Layer[*i*] on the right. ReLU is a popular nonlinear function for MLPs. The dimensions of the input and output layers are often different. Such a layer is called fully connected because it depends on all the inputs from the prior 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.

© 2019 Elsevier Inc. All rights reserved.

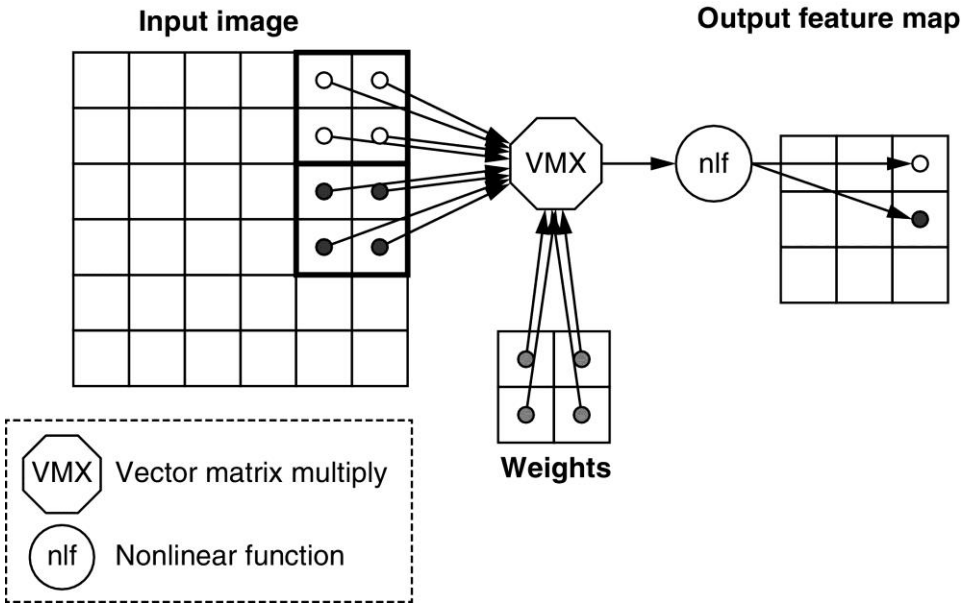

Figure 7.8 Simplified first step of a CNN. In this example, every group of four pixels of the input image are multiplied by the same four weights to create the cells of the output feature map. The pattern depicted shows a stride of two between the groups of input pixels, but other strides are possible. 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 7.9 shows how multiple feature maps turn the points into a vector in the third dimension.

© 2019 Elsevier Inc. All rights reserved.

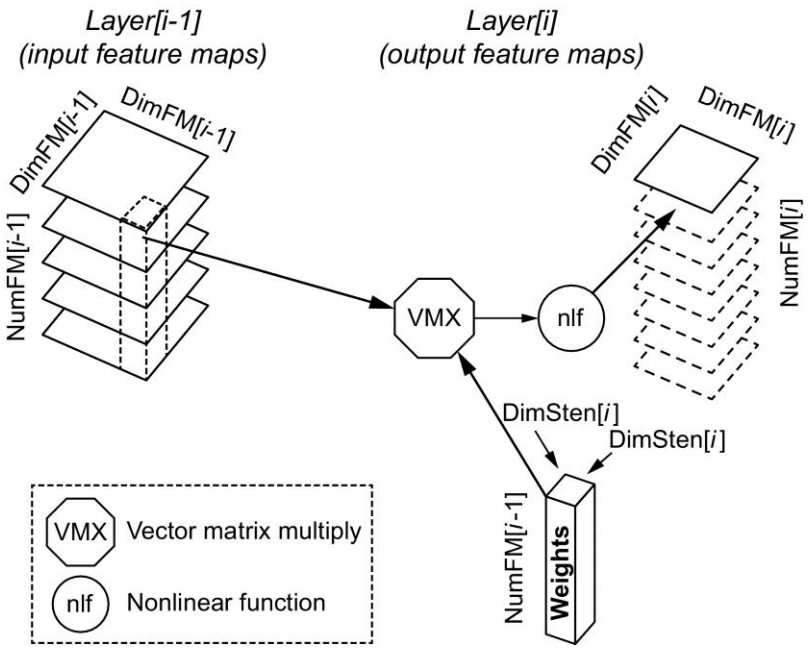

Figure 7.9 CNN general step showing input feature maps of Layer[*i*− 1] on the left, the output feature maps of Layer[*i*] on the right, and a three-dimensional stencil over input feature maps to produce a single output feature map. Each output feature map has its own unique set of weights, and the vector-matrix 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.

© 2019 Elsevier Inc. All rights reserved.

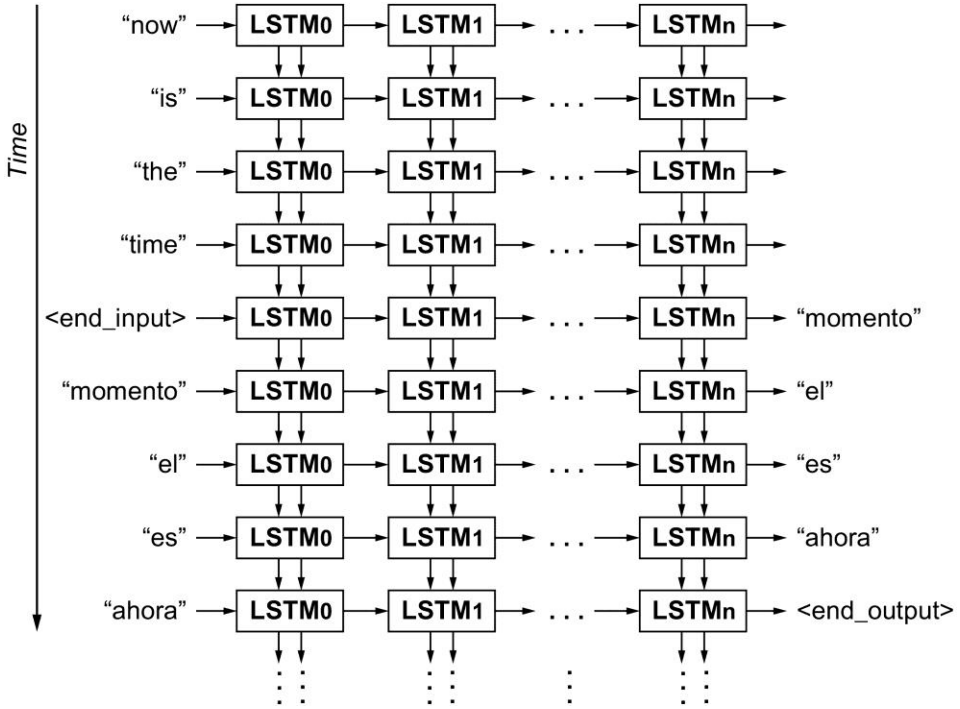

Figure 7.10 LSTM cells connected together. The inputs are on the left (English words), and the outputs are on the right (the translated Spanish words). The cells can be thought of as being unrolled over time, from top to bottom. Thus the short-term and long-term memory of LSTM is implemented by passing information top-down between unrolled cells. They are unrolled enough to translate whole sentences or even paragraphs. Such sequence-to-sequence translation models delay their output until they get to the end of the input (Wu et al., 2016). They produce the translation in *reverse order*, using the most recent translated word as input to the next step, so "now is the time" becomes "ahora es el momento." (This figure and the next are often shown turned 90 degrees in LSTM literature, but we've rotated them to be consistent with Figures 7.7 and 7.8.)

© 2019 Elsevier Inc. All rights reserved.

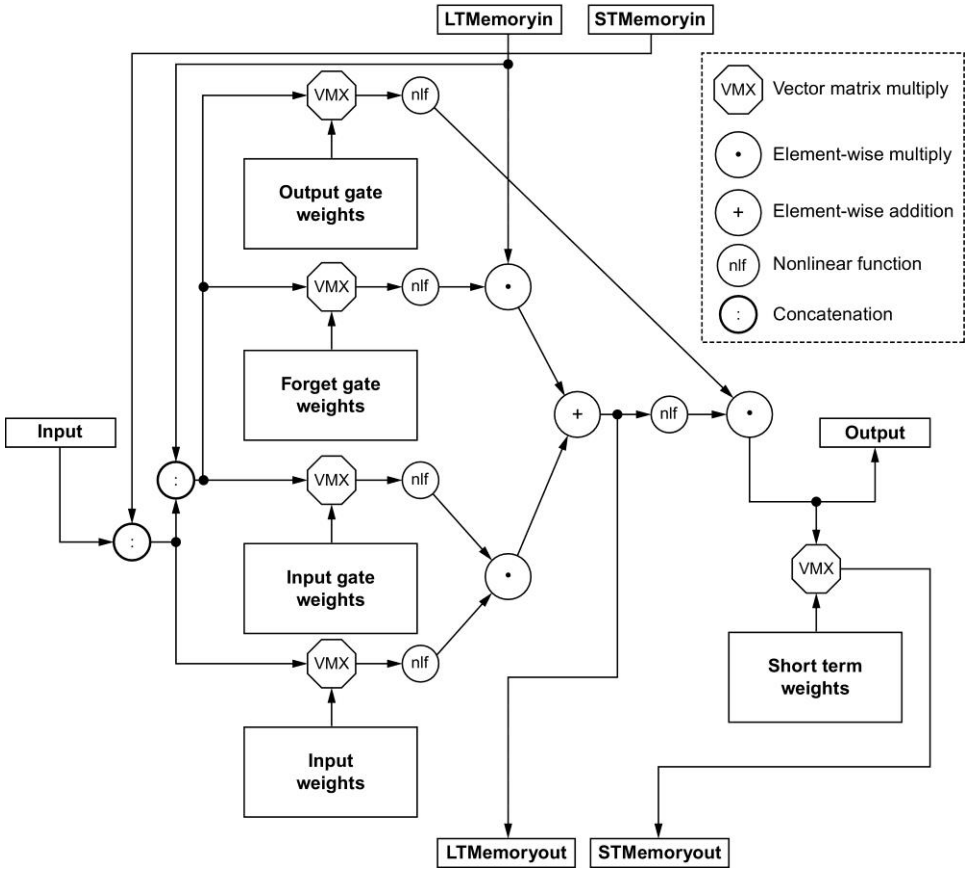

Figure 7.11 This LSTM cell contains 5 vector-matrix multiplies, 3 element-wise multiplies, 1 element-wise add, and 6 nonlinear functions. The standard input and short-term memory input are concatenated to form the vector operand for the input vector-matrix multiply. The standard input, long-term memory input, and short-term memory input are concatenated to form the vector that is used in three of the other four vector-matrix multiplies. The nonlinear functions for the three gates are Sigmoids $f(x) = 1/(1 + \exp(-x))$; the others are hyperbolic tangents. (This figure and the previous one are often shown turned 90 degrees in LSTM literature, but we've rotated them to be consistent with Figures 7.7 and 7.8.)

© 2019 Elsevier Inc. All rights reserved.

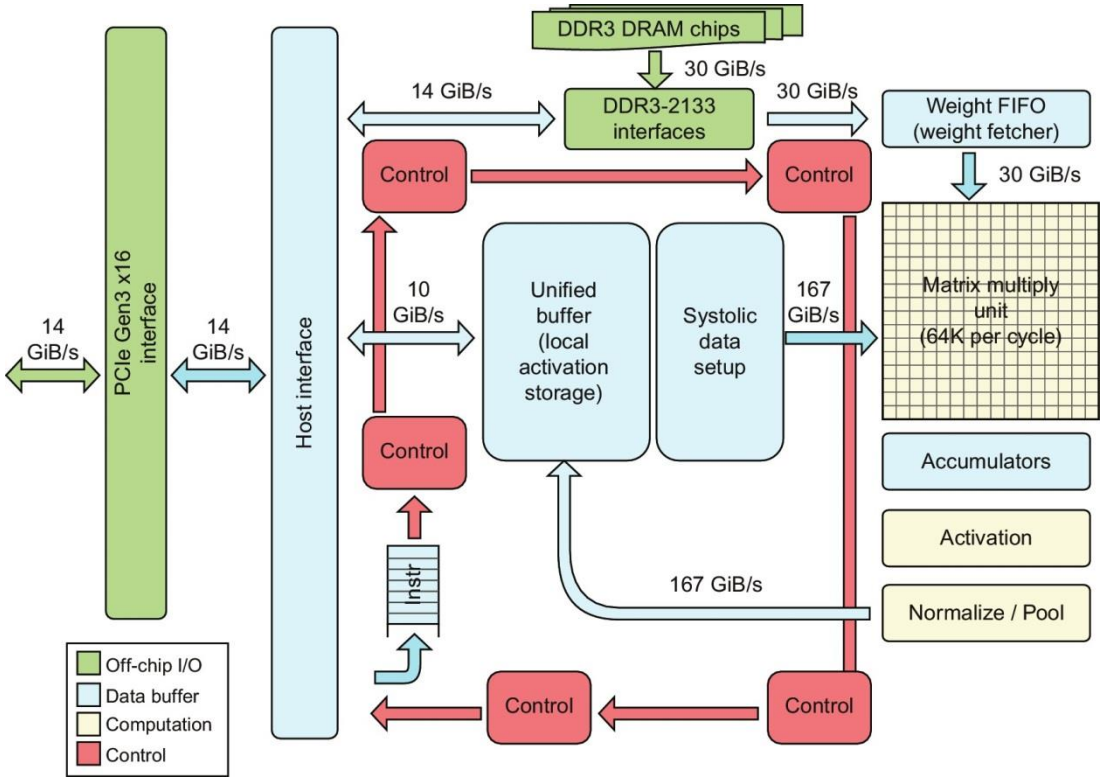

Figure 7.12 TPU Block Diagram. The PCIe bus is Gen3 × 16. The main computation part is the light-shaded Matrix Multiply Unit in the upper-right corner. Its inputs are the medium-shaded Weight FIFO and the medium-shaded Unified Buffer and its output is the medium-shaded Accumulators. The light-shaded Activation Unit performs the nonlinear functions on the Accumulators, which go to the Unified Buffer.

© 2019 Elsevier Inc. All rights reserved.

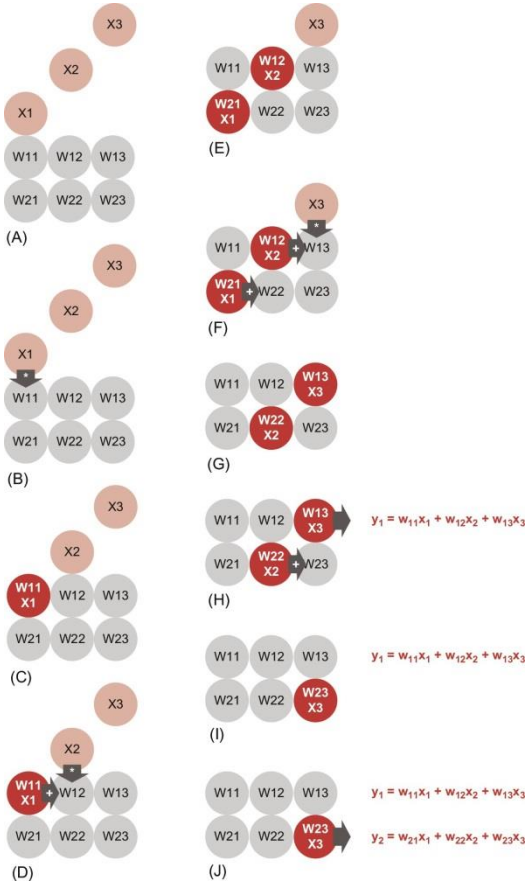

Figure 7.13 Example of systolic array in action, from top to bottom on the page. In this example, the six weights are already inside the multiply-accumulate units, as is the norm for the TPU. The three inputs are staggered in time to get the desired effect, and in this example are shown coming in from 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.

© 2019 Elsevier Inc. All rights reserved.

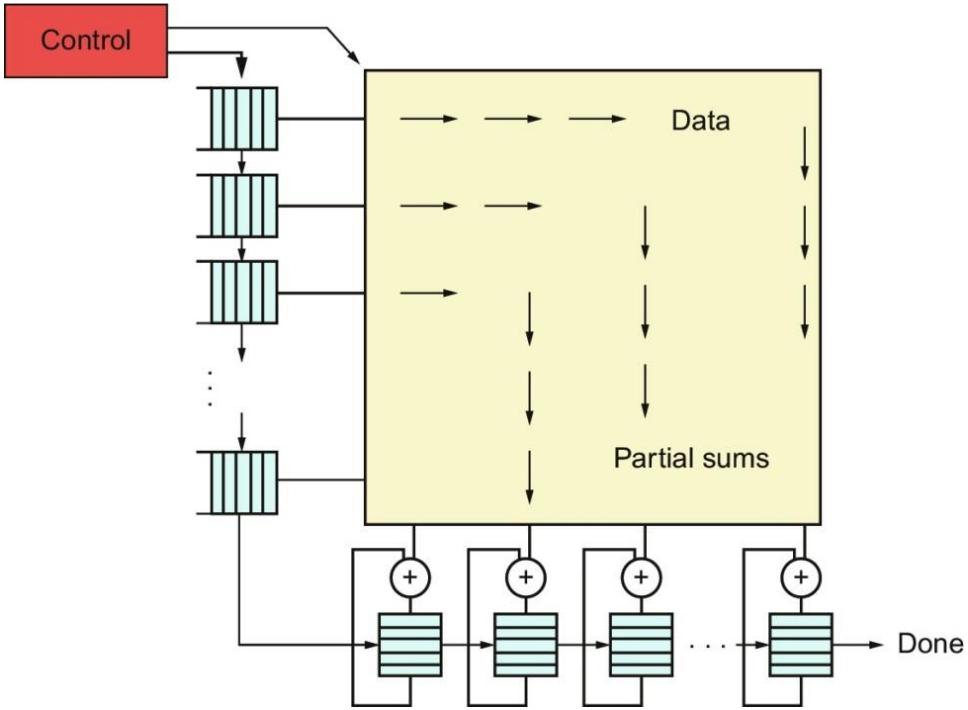

Figure 7.14 Systolic data flow of the Matrix Multiply Unit.

© 2019 Elsevier Inc. All rights reserved.

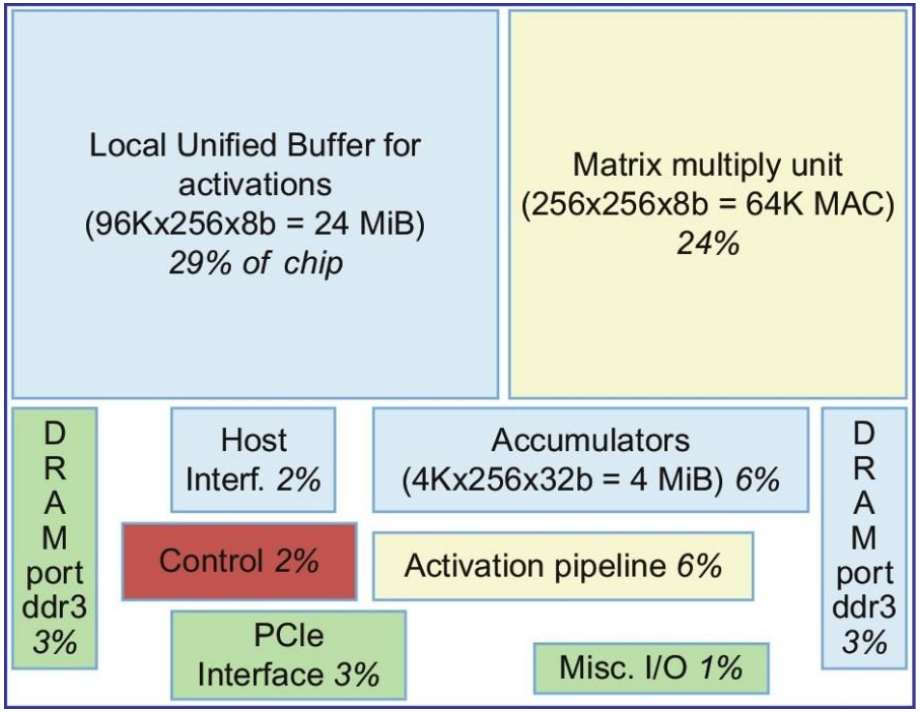

Figure 7.15 Floor plan of TPU die. The shading follows Figure 7.14. The light data buffers are 37%, the light computation units are 30%, the medium I/O is 10%, and the dark control is just 2% of the die. Control is much larger (and much more difficult to design) in a CPU or GPU. The unused white space is a consequence of the emphasis on time to tape-out for the TPU.

© 2019 Elsevier Inc. All rights reserved.

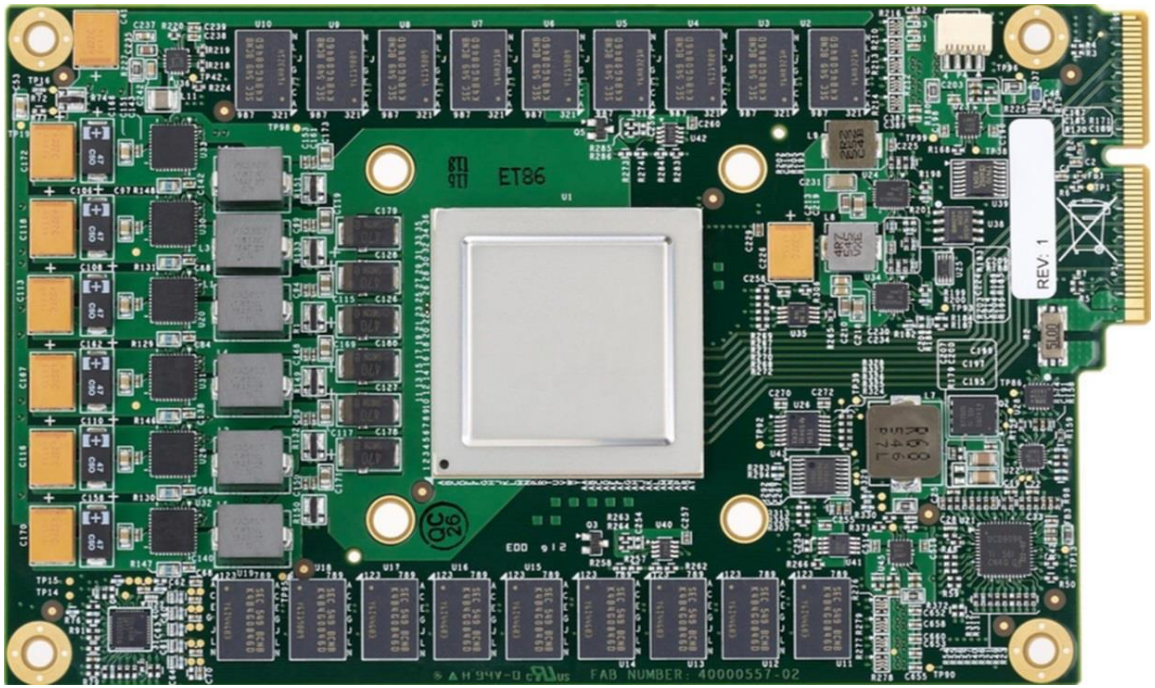

Figure 7.16 TPU printed circuit board. It can be inserted into the slot for an SATA disk in a server, but the card uses the PCIe bus.

© 2019 Elsevier Inc. All rights reserved.

```
# Network Parameters
n_hidden_1 = 256 # 1st layer number of features
n_hidden_2 = 256 # 2nd layer number of features
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])

# Create model
def multilayer_perceptron(x, weights, biases):
    # Hidden layer with ReLU activation
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    # Hidden layer with ReLU activation
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)
    # Output layer with linear activation
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

Figure 7.17 Portion of the TensorFlow program for the MNIST MLP. It has two hidden 256 × 256 layers, with each layer using a ReLU as its nonlinear function.

© 2019 Elsevier Inc. All rights reserved.

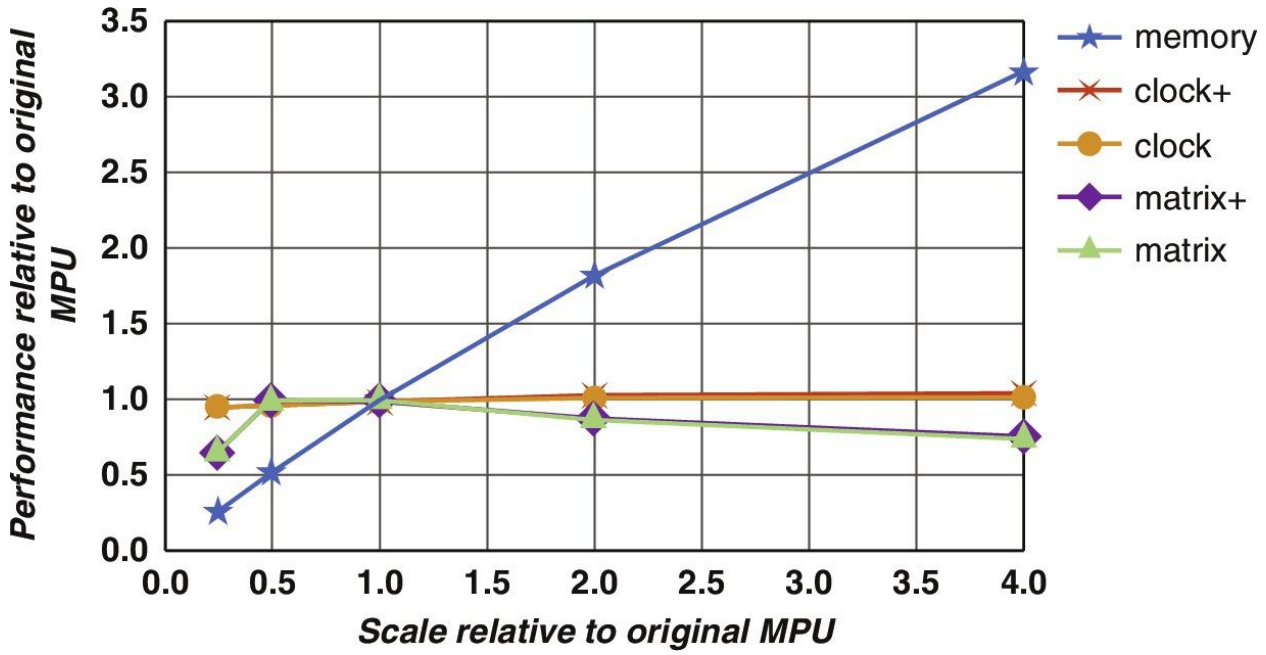

Figure 7.18 Performance as metrics scale from 0.25 × to 4 ×: memory bandwidth, clock rate + accumulators, clock rate, matrix unit dimension + accumulators, and one dimension of the square matrix unit. This is the average performance calculated from six DNN applications in Section 7.9. The CNNs tend to be computation-bound, but the MLPs and LSTMs are memory-bound. Most applications benefit from a faster memory, but a faster clock makes little difference, and a bigger matrix unit actually hurts performance. This performance model is only for code running inside the TPU and does not factor in the CPU host overhead.

© 2019 Elsevier Inc. All rights reserved.

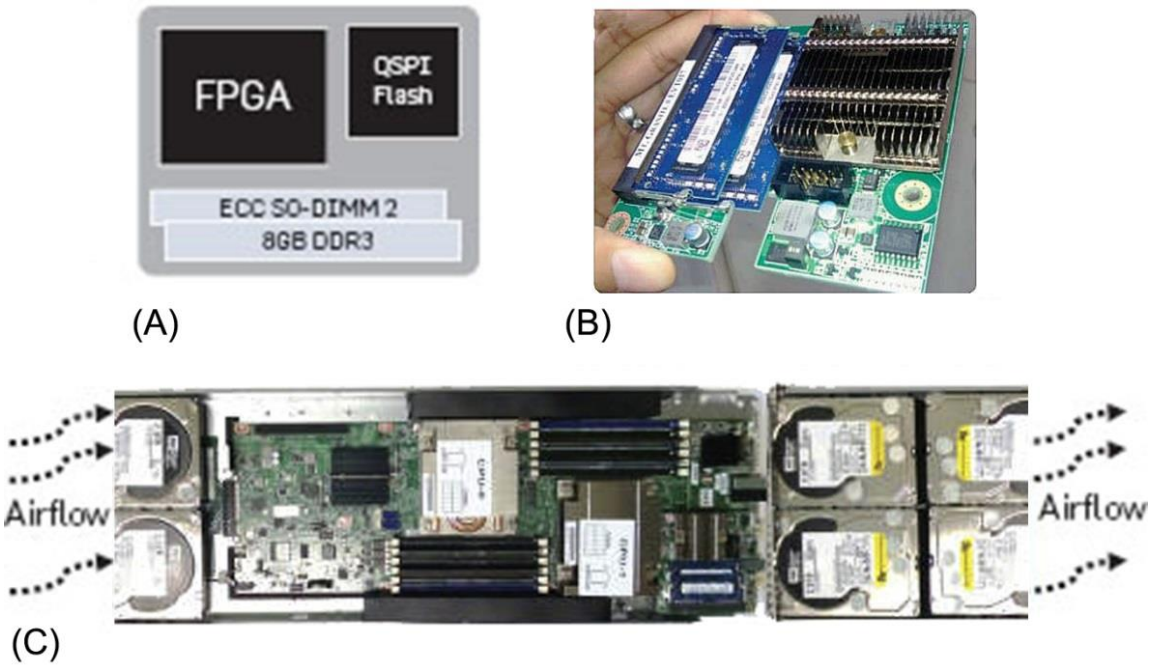

Figure 7.19 The Catapult board design. (A) shows the block diagram, and (B) is a photograph of both sides of the board, which is 10 cm × 9 cm × 16 mm. The PCIe and inter-FPGA network are wired to a connector on the bottom of the board that plugs directly into the motherboard. (C) is a photograph of the server, which is 1U (4.45 cm) high and half a standard rack wide. Each server has two 12-core Intel Sandy Bridge Xeon CPUs, 64 GiB of DRAM, 2 solid-state drives, 4 hard-disk drives, and a 10-Gbit Ethernet network card. The highlighted rectangle on the right in (C) shows the location of the Catapult FPGA board on the server. The cool air is sucked in from the left in (C), and the hot air exhausts to the right, which passes over the Catapult board. This hot spot and the amount of the power that the connector could deliver mean that the Catapult board is limited to 25 watts. Forty-eight servers share an Ethernet switch that connects to the data center network, and they occupy half of a data center rack.

© 2019 Elsevier Inc. All rights reserved.

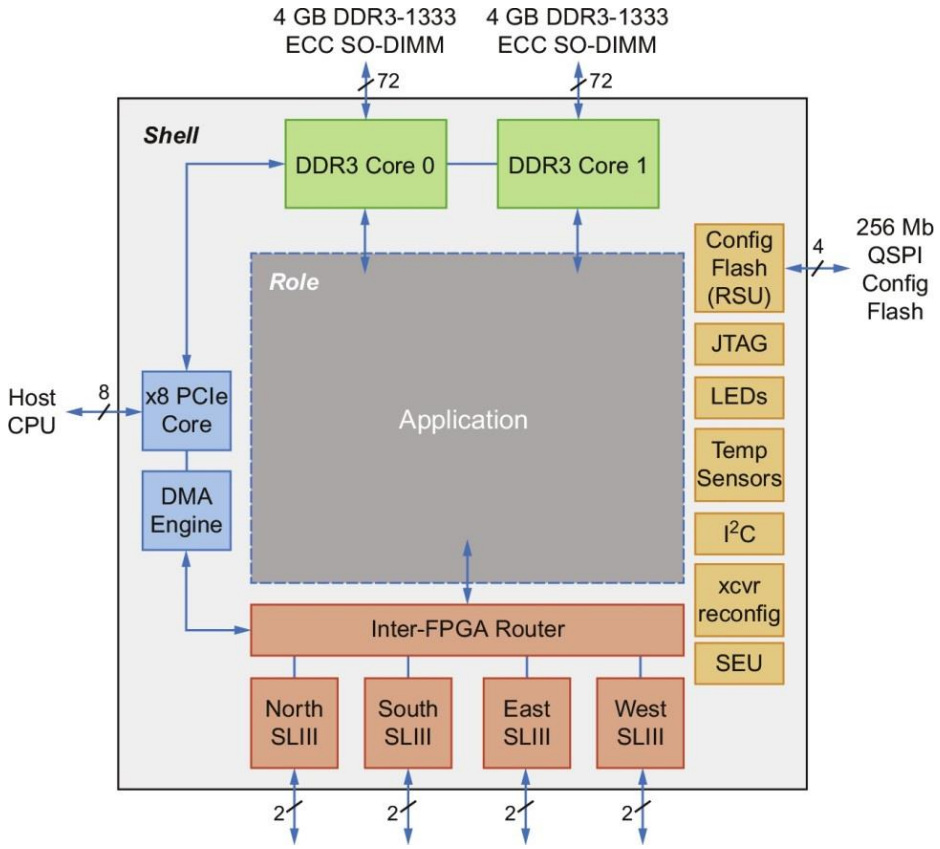

Figure 7.20 Components of Catapult shell and role split of the RTL code.

© 2019 Elsevier Inc. All rights reserved.

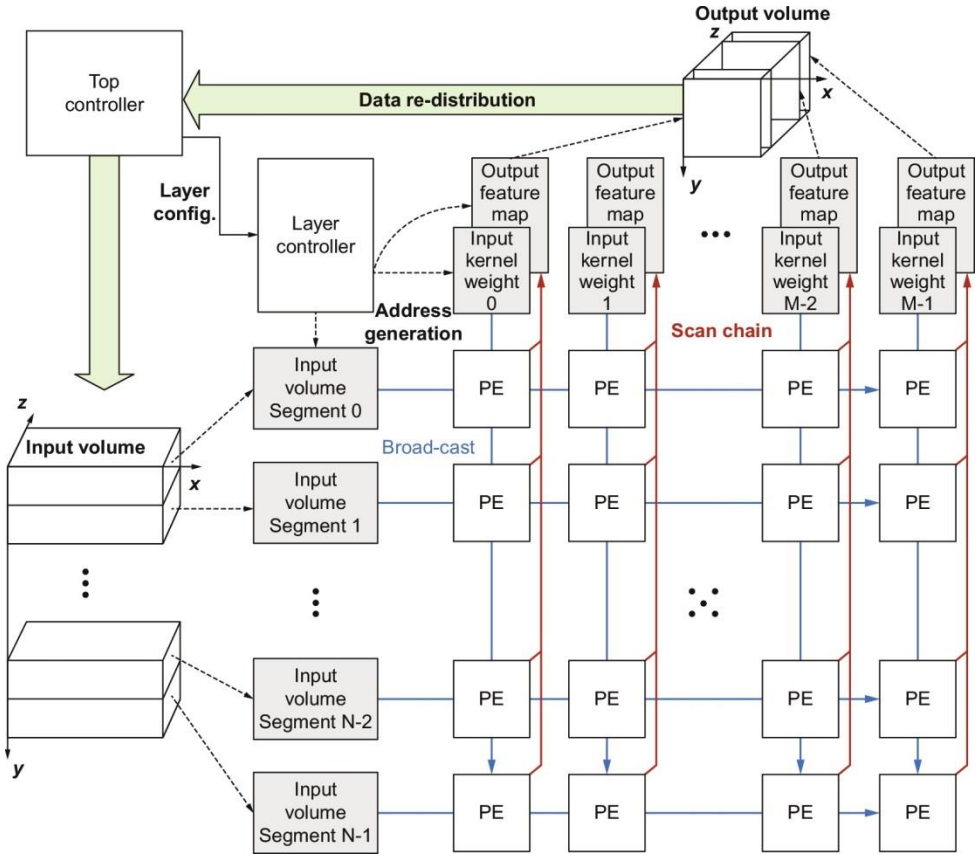

Figure 7.21 CNN Accelerator for Catapult. The Input Volume of the left correspond to Layer[*i*− 1] on the left of Figure 7.20, with NumFM[*i*− 1] corresponding to y and DimFM[*i*− 1] corresponding to *z*. Output Volume at the top maps to Layer[*i*], with z mapping to NumFM[*i*] and DimFM[*i*] mapping to *x*. The next figure shows the inside of the Processing Element (PE).

© 2019 Elsevier Inc. All rights reserved.

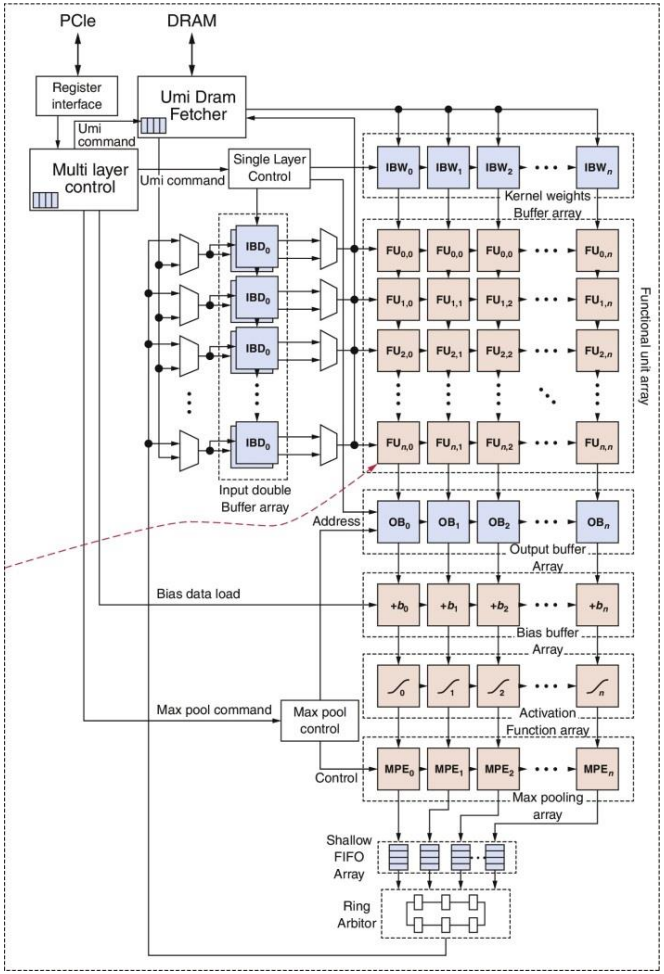

Figure 7.22 The Processing Element (PE) of the CNN Accelerator for Catapult in Figure 7.21. The two-dimension Functional Units (FU) consist of just an ALU and a few registers.

© 2019 Elsevier Inc. All rights reserved.

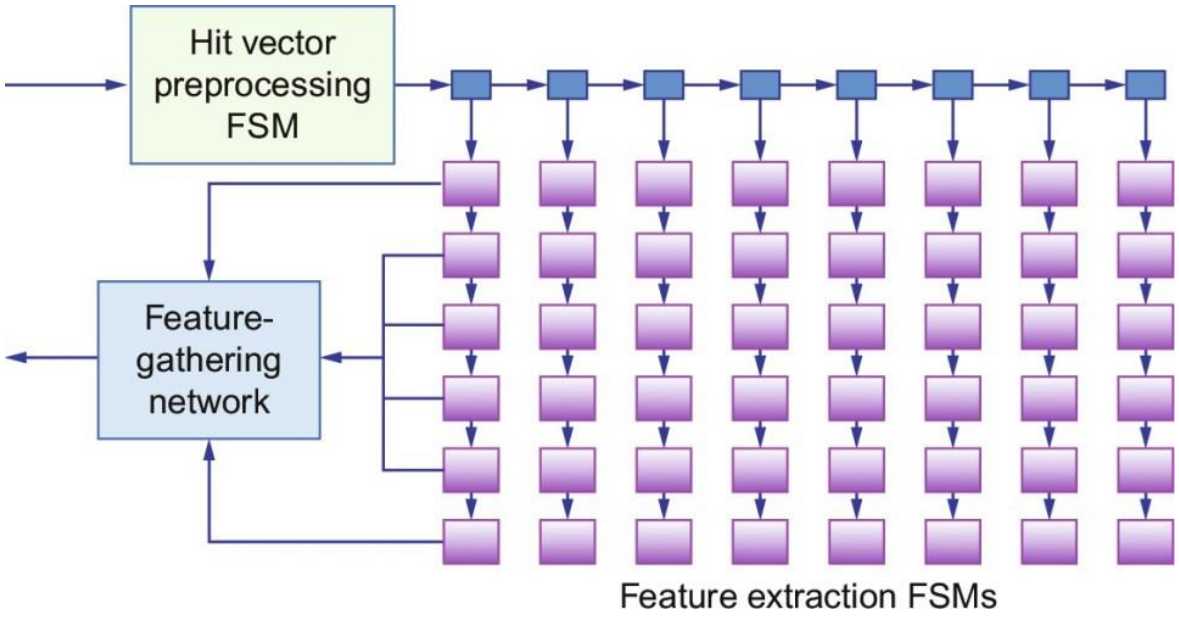

Figure 7.23 The architecture of FPGA implementation of the Feature Extraction stage. A hit vector, which describes the locations of query words in each document, is streamed into the hit vector preprocessing state machine and then split into control and data tokens. These tokens are issued in parallel to the 43 unique feature state machines. The feature-gathering network collects generated feature and value pairs and forwards them to the following Free-Form Expressions stage.

© 2019 Elsevier Inc. All rights reserved.

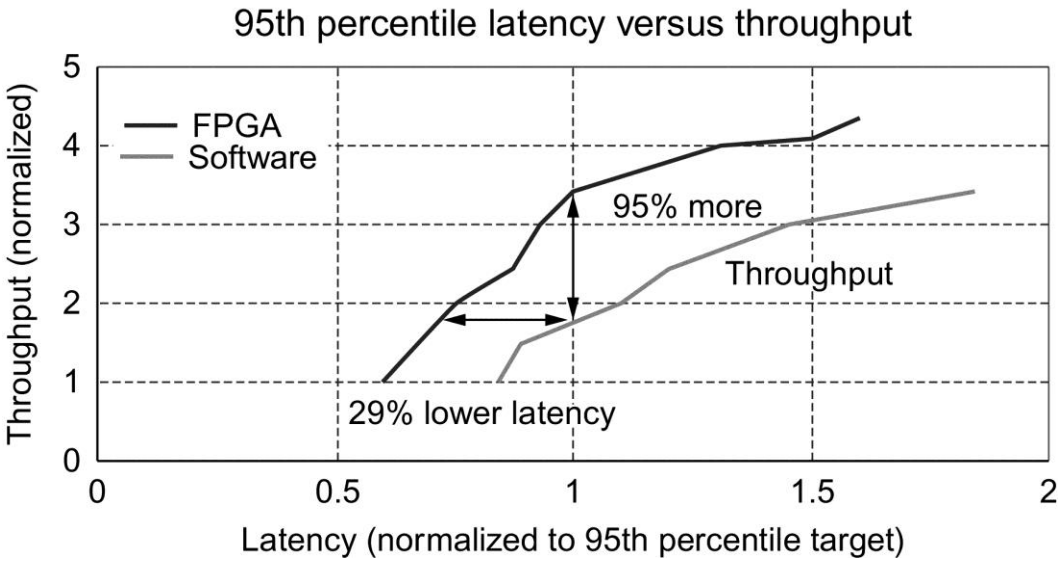

Figure 7.24 Performance for the ranking function on Catapult for a given latency bound. The *x*-axis shows the response time for the Bing ranking function. The maximum response time at the 95th percentile for the Bing application on the *x*-axis is 1.0, so data points to the right may have a higher throughput but arrive too late to be useful. The *y*-axis shows the 95% throughputs on Catapult and pure software for a given response time. At a normalized response time of 1.0, Catapult has 1.95 the throughput of Intel server running in pure software mode. Stated alternatively, if Catapult matches the throughput that the Intel server has at 1.0 normalized response time, Catapult's response time is 29% less.

© 2019 Elsevier Inc. All rights reserved.

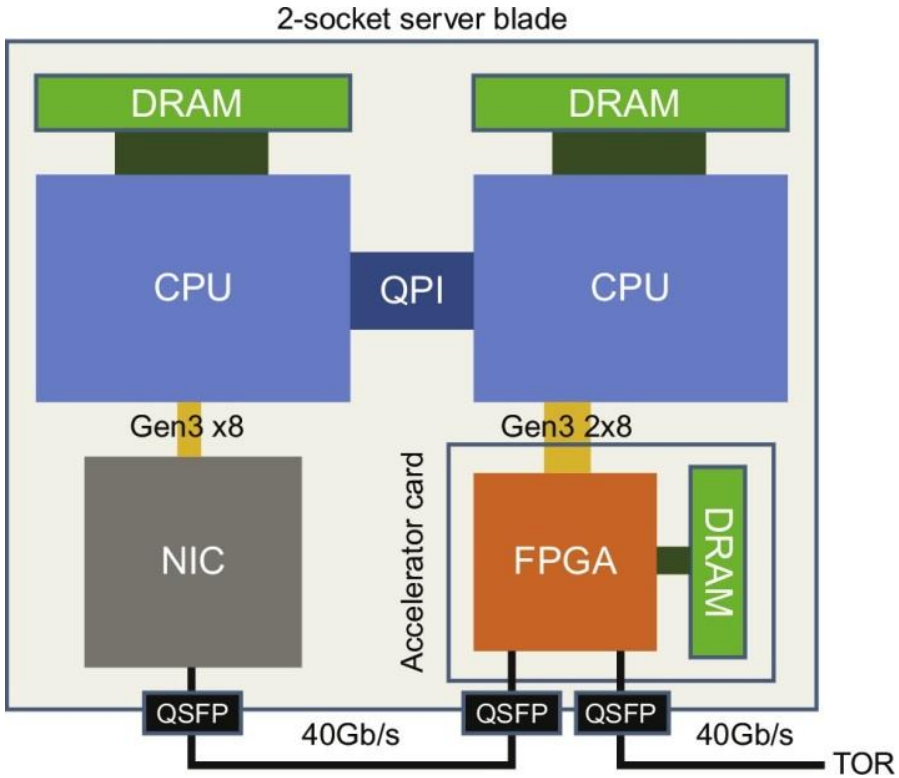

Figure 7.25 The Catapult V2 block diagram. All network traffic is routed through the FPGA to the NIC. There is also a PCIe connector to the CPUs, which allows the FPGA to be used as a local compute accelerator, as in Catapult V1.

© 2019 Elsevier Inc. All rights reserved.

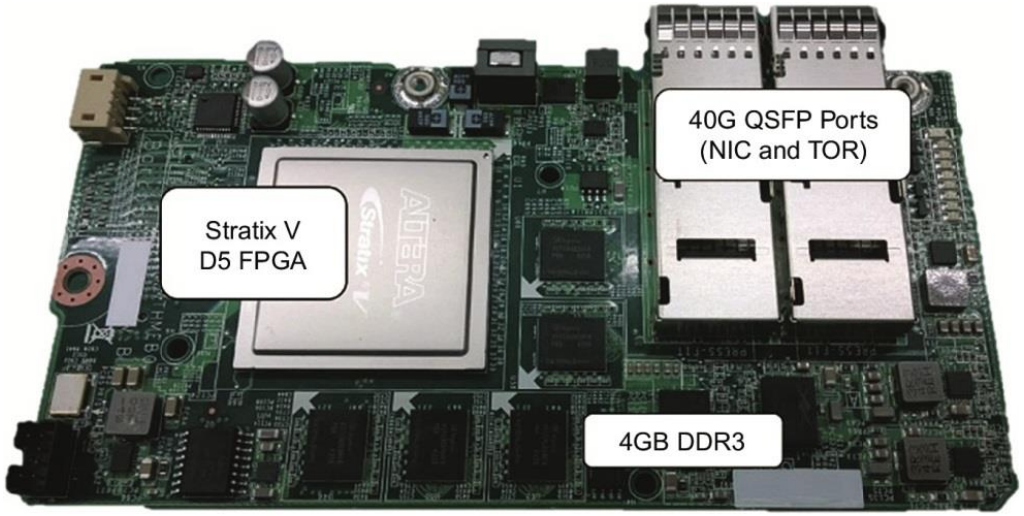

Figure 7.26 The Catapult V2 board uses a PCIe slot. It uses the same FPGA as Catapult V1 and has a TDP of 32 W. A 256-MB Flash chip holds the *golden image* for the FPGA that is loaded at power on, as well as one application image.

© 2019 Elsevier Inc. All rights reserved.

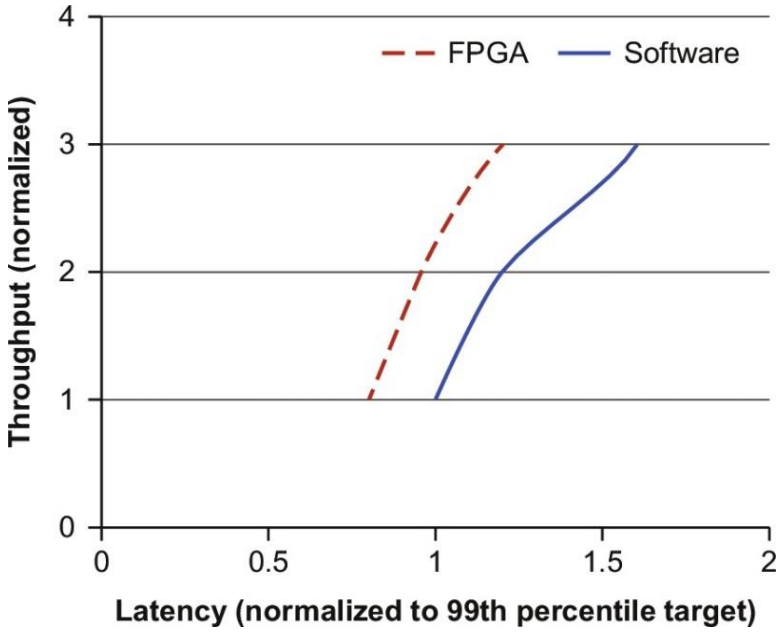

Figure 7.27 Performance for the ranking function on Catapult V2 in the same format as Figure 7.24. Note that this version measures 99th percentile while the earlier figure plots 95th percentile.

© 2019 Elsevier Inc. All rights reserved.

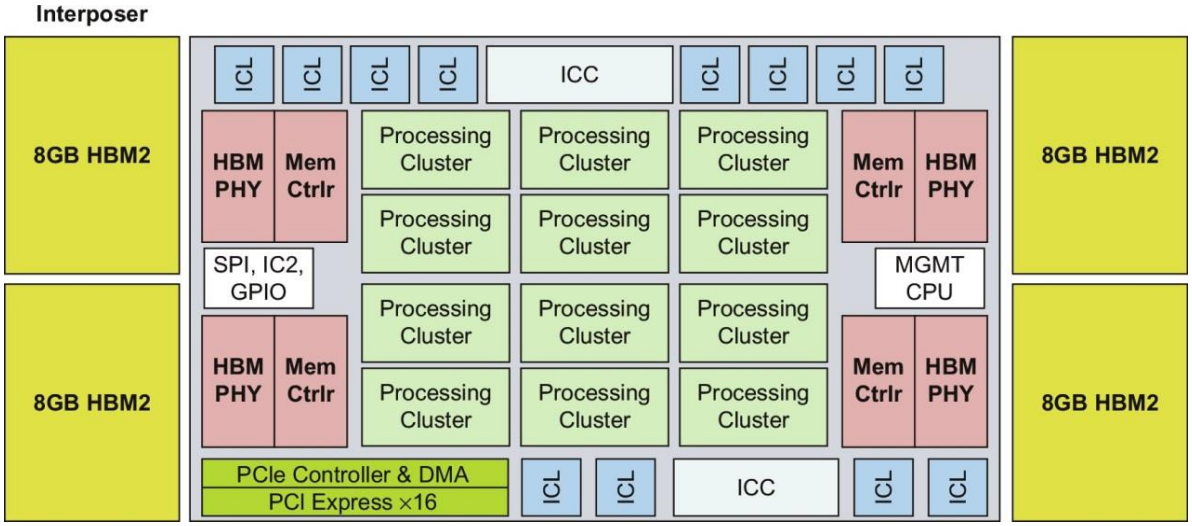

Figure 7.28 Block diagram of the Intel Lake Crest processor. Before being acquired by Intel, Crest said that the chip is almost a full reticle in TSMC 28 nm, which would make the die size 600–700 mm^2. This chip should be available in 2017. Intel is also building Knights Crest, which is a hybrid chip containing Xeon x 86 cores and Crest accelerators.

© 2019 Elsevier Inc. All rights reserved.

Term	Acronym	Short explanation
Core	–	A processor. Pixel Visual Core can have 2–16 cores. The first implementation has 8; also called *stencil processor (STP)*
Halide	–	A domain-specific programming language for image processing that separates the algorithm from its execution schedule
Halo	–	An extended region around the 16 × 16 computation array to handle stencil computation near the borders of the array. It holds values, but doesn't compute
Image signal processors	ISP	A fixed function ASIC that improves the visual quality of an image; found in virtually all PMDs with cameras
Image processing unit	IPU	A DSA that solves the inverse problem of a GPU: it analyzes and modifies an *input* image in contrast to generating an *output* image
Line buffer pool	LB	A line buffer is designed to capture a sufficient number of full lines of an intermediate image to keep the next stage busy. Pixel Visual Core uses two-dimensional line buffers, each Change 64 to 128 KiB. The *Line Buffer Pool* contains one LB per core plus one LB for DMA
Network on chip	NOC	The network that connects the cores in Pixel Visual Core
Physical ISA	pISA	The Pixel Visual Core instruction set architecture (ISA) that is executed by the hardware
Processing element array	–	The 16 × 16 array of Processing Elements plus the halo that performs the 16-bit multiply-add operations. Each Processing Element includes a Vector Lane and local memory. It can shift data en mass to neighbors in any of four directions
Sheet generator	SHG	Does memory accesses of blocks of 1 × 1 to 31 × 31 pixels, which are called *sheets*. The different sizes allow the option of including the space for the halo or not
Scalar lane	SCL	Same operations as the Vector Lane except it adds instructions that handle jumps, branches, and interrupts, controls instruction flow to the vector array, and schedules all the loads and stores for the sheet generator. It also has a small instruction memory. It plays the same role as the scalar processor in a vector architecture
Vector lane	VL	Portion of the Processing Element that performs the computer arithmetic
Virtual ISA	vISA	The Pixel Visual Core ISA generated by the compiler. It is mapped to pISA before execution

Figure 7.29 A handy guide to Pixel Visual Core terms in Section 7.7. Figure 7.4 on page 437 has a guide for Sections 7.3–7.6.

© 2019 Elsevier Inc. All rights reserved.

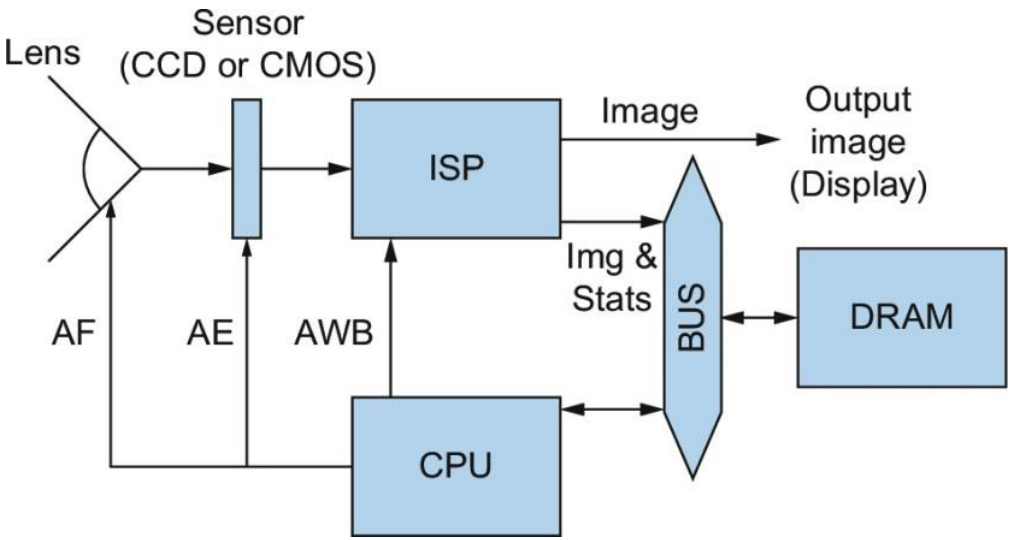

Figure 7.30 Diagram showing interconnection of the Image Signal Processor (ISP), CPU, DRAM, lens, and sensor. The ISP sends statistics to the CPU as well as the improved image either to the display or to DRAM for storage or later processing. The CPU then processes the image statistics and sends information to let the system adapt: *Auto White Balance* (AWB) to the ISP, *Auto Exposure* (AE) to the sensor, and *Auto Focus* (AF) to the lens, known as the *3As*.

© 2019 Elsevier Inc. All rights reserved.

```
Func buildBlur(Func input) {
       // Functional portion (independent of target processor)
      Func blur_x("blur_x"), blur_y("blur_y");
      blur_x(x,y) = (input(x−1,y) + input(x,y)*2 + input(x+1,y)) / 4;
      blur_y(x,y) = (blur_x(x,y−1) + blur_x(x,y)*2 + blur_x(x,y+1)) / 4;

      if (has_ipu) {
        // Schedule portion (directs how to optimize for target processor)
        blur_x.ipu(x,y);
        blur_y.ipu(x,y);
}
      return blur_y;
}
```

Figure 7.31 Portion of a Halide example to blur an image. The `ipu(x,y)` suffix schedules the function to Pixel Visual Core. A blur has the effect of looking at the image through a translucent screen, which reduces noise and detail. A Gaussian function is often used to blur the image.

© 2019 Elsevier Inc. All rights reserved.

Operation	*Energy (pJ)*	Operation	Energy (pJ)	Operation	Energy (pJ)
8b DRAM LPDDR3	125.00	8b SRAM	1.2–17.1	16b SRAM	2.4–34.2
32b Fl. Pt. muladd	2.70	8b int muladd	0.12	16b int muladd	0.43
32b Fl. Pt. add	1.50	8b int add	0.01	16b int add	0.02

Figure 7.32 Relative energy costs per operation in picoJoules assuming TSMC 28-nm HPM process, which was the process Pixel Visual Core used [17][18][19][20]. The absolute energy cost are less than in Figure 7.2 because of using 28 nm instead of 90 nm, but the relative energy costs are similarly high.

© 2019 Elsevier Inc. All rights reserved.

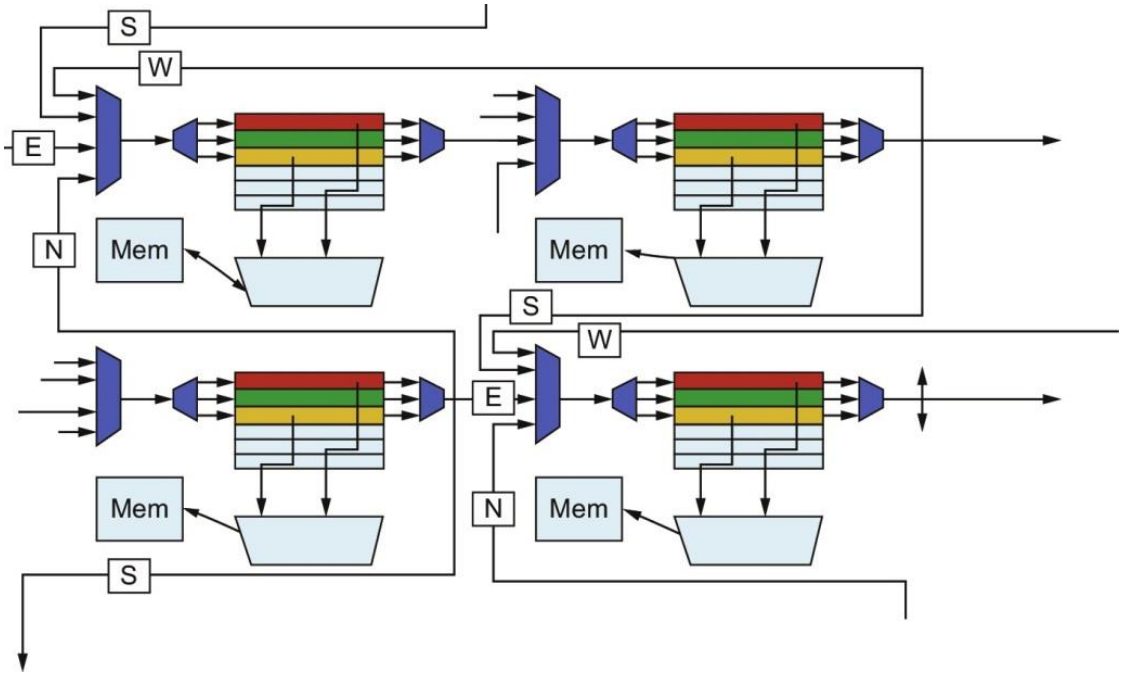

Figure 7.33 The two-dimensional SIMD includes two-dimensional shifting "N," "S," "E," "W," show the direction of the shift (North, South, East, West). Each PE has a software-controlled scratchpad memory.

© 2019 Elsevier Inc. All rights reserved.

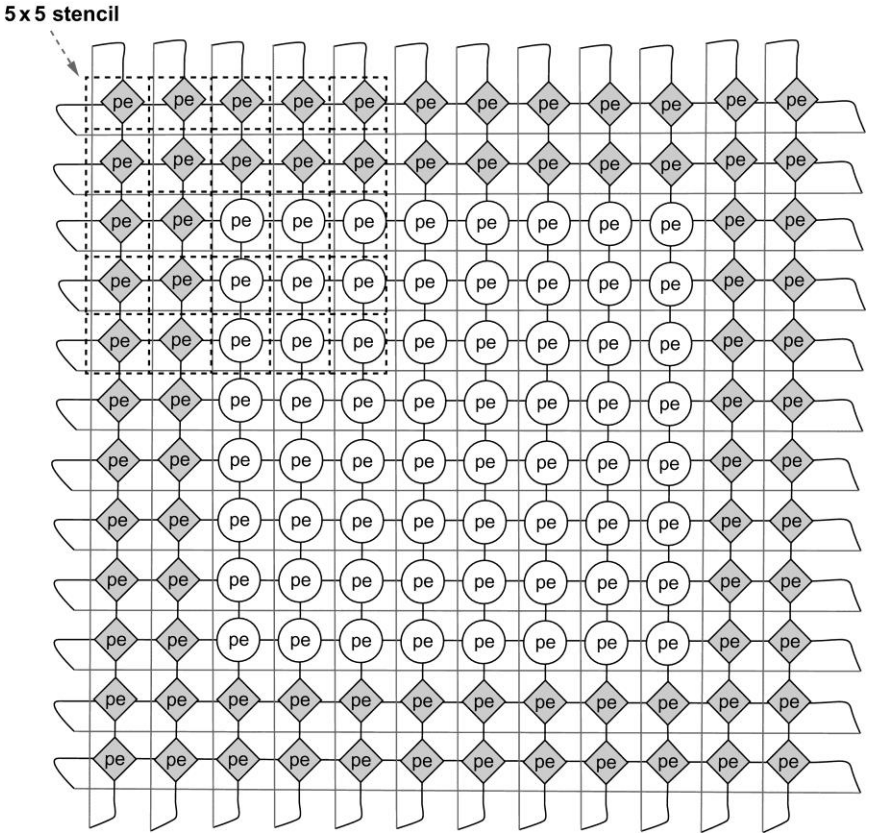

Figure 7.34 The two-dimensional array of full processing elements *(shown as unshaded circles)* surrounded by two layers of simplified processing elements *(shaded diamonds)* called a *halo*. In this figure, there are 8 × 8 or 64 full PEs with 80 simplified PEs in the halo. (Pixel Visual Core actually has 16 × 16 or 256 full PEs and two layers in its halo and thus 144 simplified PEs.) The edges of the halo are connected *(shown as gray lines)* to form a torus. Pixel Visual Core does a series of two-dimensional shifts across all processing elements to move the neighbor portions of each stencil computation into the center PE of the stencil. An example 5 × 5 stencil is shown in the upper-left corner. Note that 16 of the 25 pieces of data for this 5 × 5 stencil location come from halo processing elements.

© 2019 Elsevier Inc. All rights reserved.

Field	Scalar	Math	Memory	Imm	MemImm
# Bits	43	38	12	16	10

Figure 7.35 VLIW format of the 119-bit pISA instruction.

© 2019 Elsevier Inc. All rights reserved.

```
// vISA inner loop blur in x dimension
input.b16  t1 <- _input[x*1+(-1)][y*1+0][0]; // t1 = input[x-1,y]
input.b16  t2 <- _input[x*1+0][y*1+0][0];  // t2 = input[x,y]
mov.b16    st3 <- 2;
mul.b16    t4 <- t2, st3;  //t4 = input[x,y] * 2
add.b16    t5 <- t1, t4;  //t5 = input[x-1,y] + input[x,y]*2
input.b16  t6 <- _input[x*1+1][y*1+0][0];  // t6 = input[x+1,y]
add.b16    t7 <- t5, t6;  //t7 = input[x+1,y]+input[x,y]+input[x-1,y]*2
mov.b16    st8 <- 4;
div.b16    t9 <- t7, st8;  //t9 = t7/4
output.b16 _blur_x[x*1+0][y*1+0][0] <- t9; // blur_x[x,y] = t7/4
// vISA inner loop blur in y dimension
input.b16  t1 <- _blur_x[x*1+0][y*1+(-1)][0]; // t1 = blur_x[x,y-1]
input.b16  t2 <- _blur_x[x*1+0][y*1+0][0];  // t2 = blur_x[x,y]
mov.b16    st3 <- 2;
mul.b16    t4 <- t2, st3;  //t4 = blur_x[x,y] * 2
add.b16    t5 <- t1, t4;  //t5 = blur_x[x,y-1] + blur_x[x,y]*2
input.b16  t6 <- _blur_x[x*1+0][y*1+1][0];  // t6 = blur_x[x,y+1]
add.b16    t7 <- t5, t6;  //t7 = blurx[x,y+1]+blurx[x,y-1]+blurx[x,y]*2
mov.b16    st8 <- 4;
div.b16    t9 <- t7, st8;  //t9 = t7/4
output.b16 _blur_y[x*1+0][y*1+0][0] <- t9; // blur_y[x,y] = t7/4
```

Figure 7.36 Portion of the vISA instructions compiled from the Halide Blur code in Figure 7.31. This vISA code corresponds to the functional part of the Halide code.

© 2019 Elsevier Inc. All rights reserved.

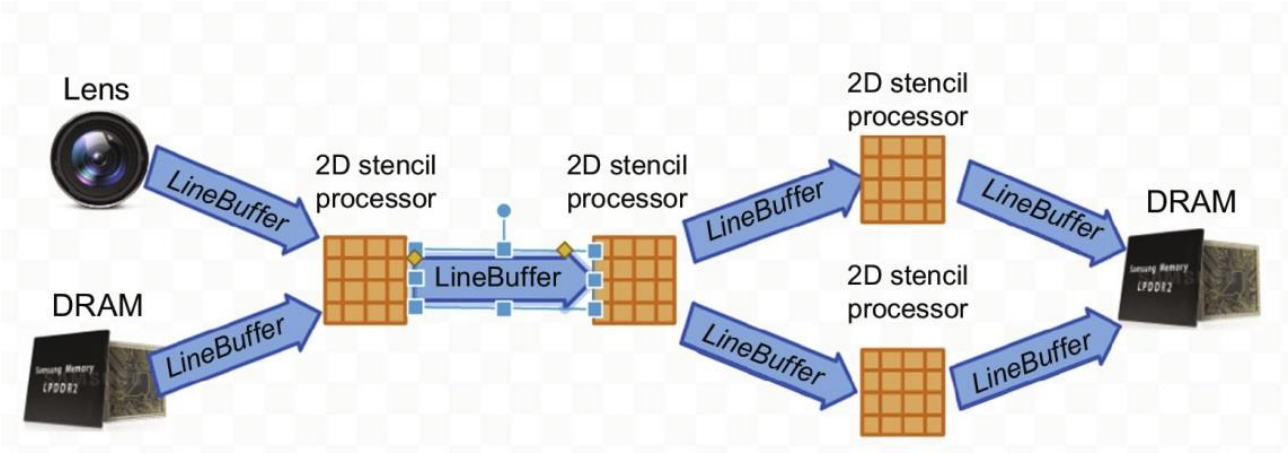

Figure 7.37 Programmer view of Pixel Visual Core: a directed-acyclic graph of kernels.

© 2019 Elsevier Inc. All rights reserved.

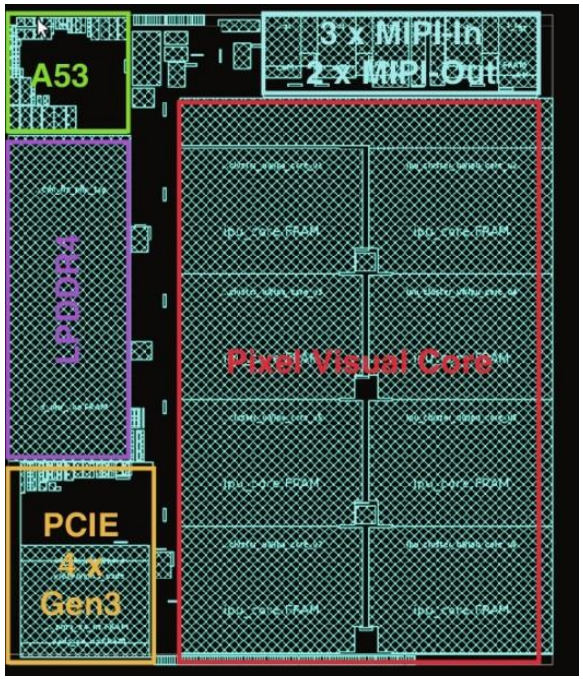

Figure 7.38 Floor plan of the 8-core Pixel Visual Core chip. A53 is an ARMv7 core. LPDDR4 is a DRAM controller. PCIE and MIPI are I/O buses.

© 2019 Elsevier Inc. All rights reserved.

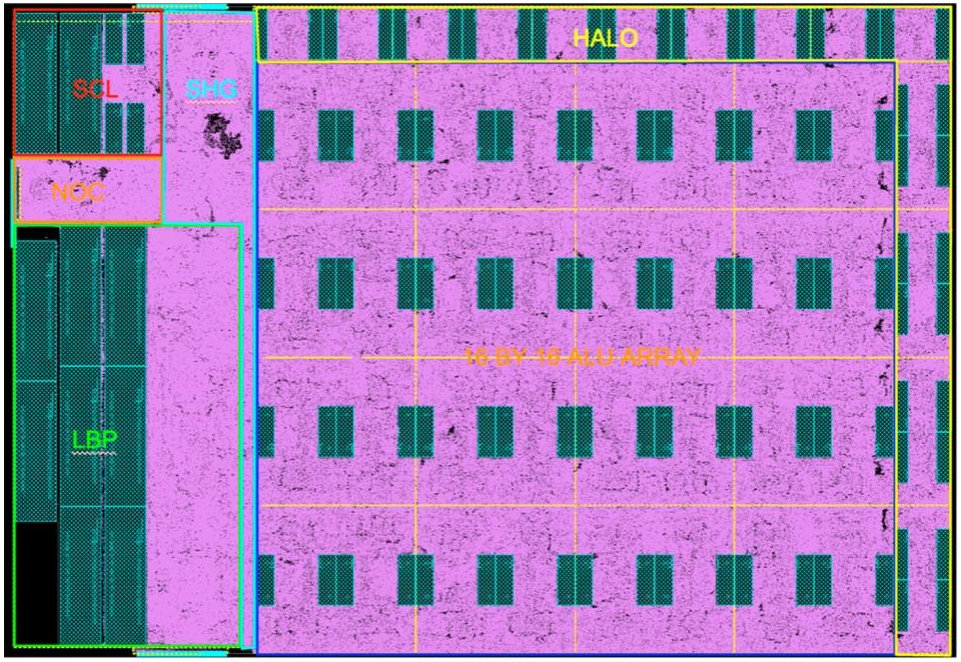

Figure 7.39 Floor plan of a Pixel Visual Core. From left to right, and top down: the scalar lane (SCL) is 4% of the core area, NOC is 2%, the line buffer pool (LBP) is 15%, the sheet generator (SHG) is 5%, the halo is 11%, and the processing element array is 62%. The torus connection of the halo makes each of the four edges of the array logical neighbors. It is more area-efficient to collapse the halo to just two sides, which preserves the topology.

© 2019 Elsevier Inc. All rights reserved.

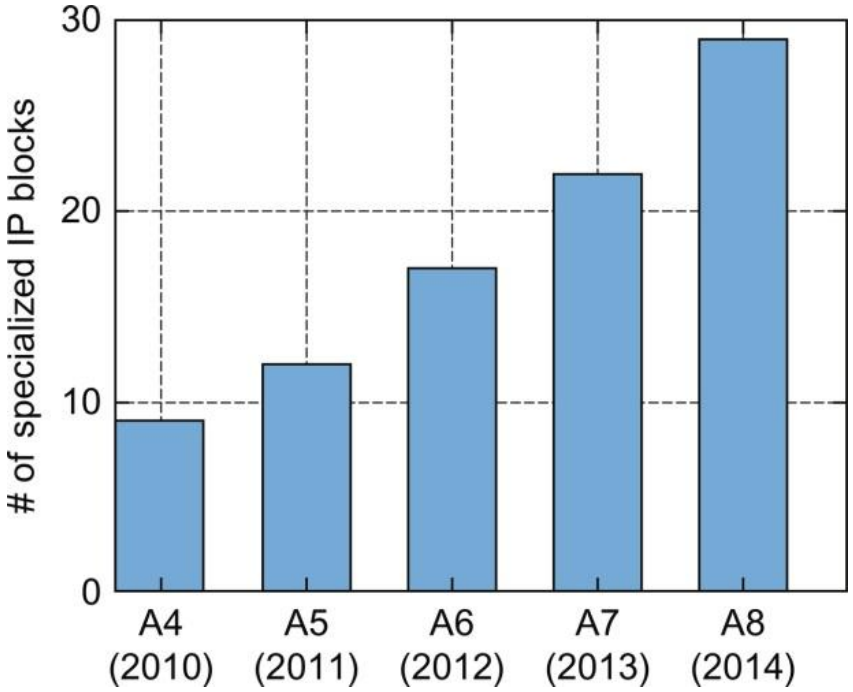

Figure 7.40 Number of IP blocks in Apple SOCs for the iPhone and iPad between 2010 and 2014 (Shao and Brooks, 2015).

© 2019 Elsevier Inc. All rights reserved.

		DNN layers							
Name	LOC	FC	Conv	Element	Pool	Total	Weights	TPU Ops/Weight	% deployed TPUs 2016
MLP0	100	5				5	20M	200	61%
MLP1	1000	4				4	5M	168	
LSTM0	1000	24		34		58	52M	64	29%
LSTM1	1500	37		19		56	34M	96	
CNN0	1000		16			16	8M	2888	5%
CNN1	1000	4	72		13	89	100M	1750	

Figure 7.41 Six DNN applications (two per DNN type) that represent 95% of the TPU's workload. The 10 columns are the DNN name; the number of lines of code; the types and number of layers in the DNN (FC is fully connected; Conv is convolution; Element is element-wise operation of LSTM, see Section 7.3; and Pool is pooling, which is a downsizing stage that replaces a group of elements with its average or maximum); the number of weights; TPU operational intensity; and TPU application popularity in 2016. The operational intensity varies between TPU, CPU, and GPU because the batch sizes vary. The TPU can have larger batch sizes while still staying under the response time limit. One DNN is RankBrain (Clark, 2015), one LSTM is GNM Translate (Wu et al., 2016), and one CNN is DeepMind AlphaGo (Silver et al., 2016; Jouppi, 2016).

© 2019 Elsevier Inc. All rights reserved.

					Measured		TOPS/s			
Chip model	mm^2	nm	MHz	TDP	Idle	Busy	8b	FP	GB/s	On-chip memory
Intel Haswell	662	22	2300	145W	41W	145W	2.6	1.3	51	51 MiB
NVIDIA K80	561	28	560	150W	25W	98W	–	2.8	160	8 MiB
TPU	<331*	28	700	75W	28W	40W	92	–	34	28 MiB

*The TPU die size is less than half of the Haswell die size.

Figure 7.42 The chips used by the benchmarked servers are Haswell CPUs, K80 GPUs, and TPUs. Haswell has 18 cores, and the K80 has 13 SMX processors.

© 2019 Elsevier Inc. All rights reserved.

				Measured power	
Server	Dies/Server	DRAM	TDP	Idle	Busy
Intel Haswell	2	256 GiB	504W	159W	455W
NVIDIA K80 (2 dies/card)	8	256 GiB (host) + 12 GiB × 8	1838W	357W	991W
TPU	4	256 GiB (host) + 8 GiB × 4	861W	290W	384W

Figure 7.43 Benchmarked servers that use the chips in Figure 7.42. The low-power TPU allows for better rack-level density than the high-power GPU. The 8 GiB DRAM per TPU is Weight Memory.

© 2019 Elsevier Inc. All rights reserved.

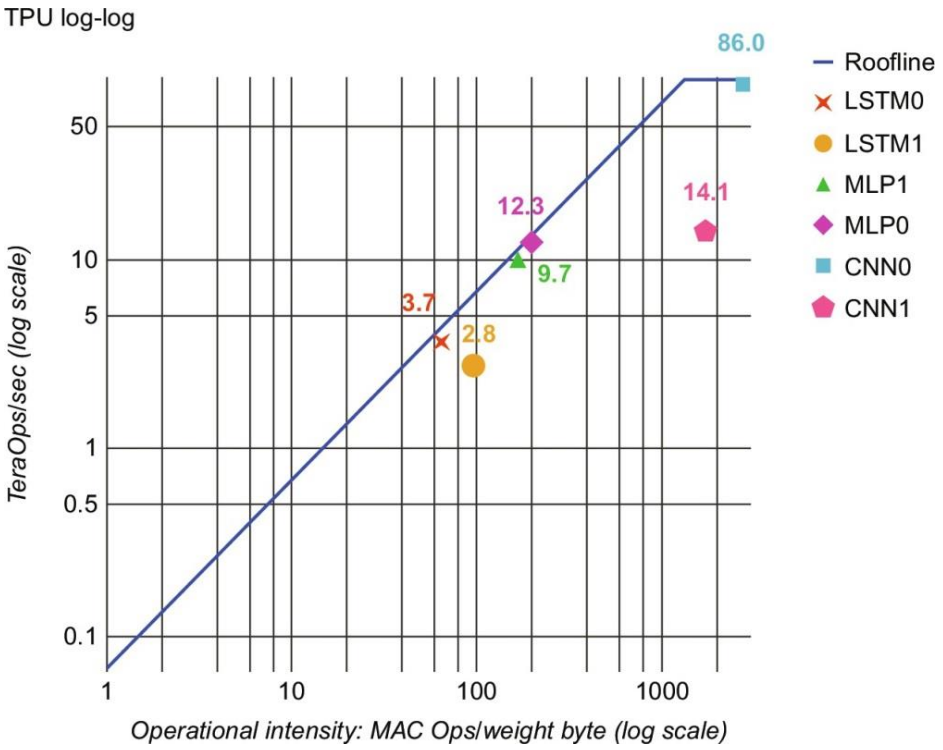

Figure 7.44 TPU Roofline. Its ridge point is far to the right at 1350 multiply-accumulate operations per byte of weight memory. CNN1 is much further below its Roofline than the other DNNs because it spends about a third of the time waiting for weights to be loaded into the matrix unit and because the shallow depth of some layers in the CNN results in only half of the elements within the matrix unit holding useful values (Jouppi et al., 2017).

© 2019 Elsevier Inc. All rights reserved.

Application	MLP0	MLP1	LSTM0	LSTM1	CNN0	CNN1	Mean	Row
Array active cycles	12.7%	10.6%	8.2%	10.5%	78.2%	46.2%	28%	1
Useful MACs in 64K matrix (% peak)	12.5%	9.4%	8.2%	6.3%	78.2%	22.5%	23%	2
Unused MACs	0.3%	1.2%	0.0%	4.2%	0.0%	23.7%	5%	3
Weight stall cycles	53.9%	44.2%	58.1%	62.1%	0.0%	28.1%	43%	4
Weight shift cycles	15.9%	13.4%	15.8%	17.1%	0.0%	7.0%	12%	5
Non-matrix cycles	17.5%	31.9%	17.9%	10.3%	21.8%	18.7%	20%	6
RAW stalls	3.3%	8.4%	14.6%	10.6%	3.5%	22.8%	11%	7
Input data stalls	6.1%	8.8%	5.1%	2.4%	3.4%	0.6%	4%	8
TeraOp/s (92 Peak)	12.3	9.7	3.7	2.8	86.0	14.1	21.4	9

Figure 7.45 Factors limiting TPU performance of the NN workload based on hardware performance counters. Rows 1, 4, 5, and 6 total 100% and are based on measurements of activity of the matrix unit. Rows 2 and 3 further break down the fraction of 64K weights in the matrix unit that hold useful weights on active cycles. Our counters cannot exactly explain the time when the matrix unit is idle in row 6; rows 7 and 8 show counters for two possible reasons, including RAW pipeline hazards and PCIe input stalls. Row 9 (TOPS) is based on measurements of production code while the other rows are based on performance-counter measurements, so they are not perfectly consistent. Host server overhead is excluded here. The MLPs and LSTMs are memory-bandwidth limited, but CNNs are not. CNN1 results are explained in the text.

© 2019 Elsevier Inc. All rights reserved.

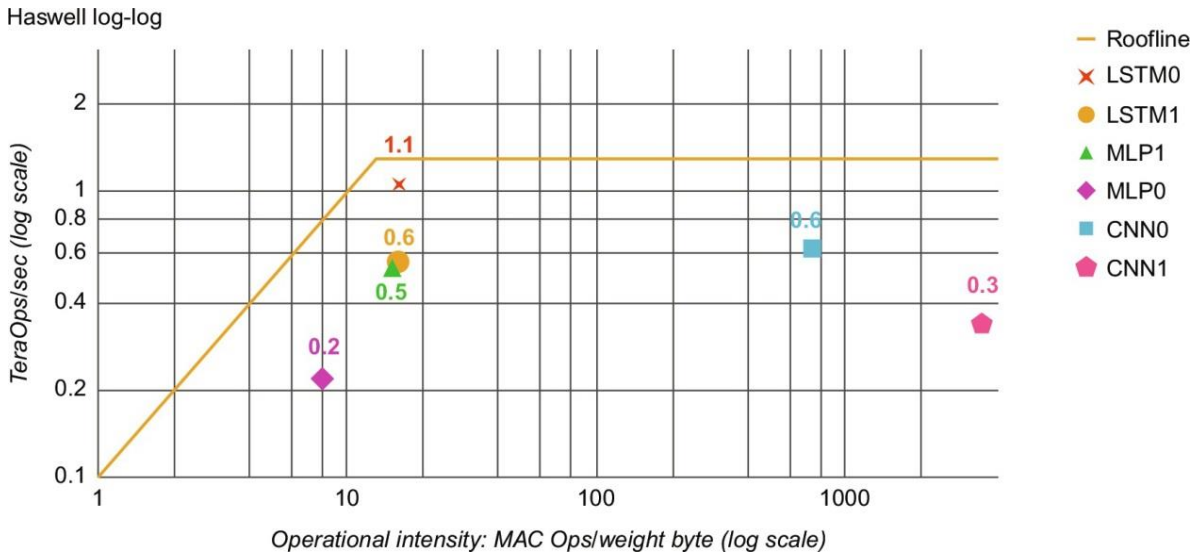

Figure 7.46 Intel Haswell CPU Roofline with its ridge point at 13 multiply-accumulate operations/byte, which is much further to the left than in Figure 7.44.

© 2019 Elsevier Inc. All rights reserved.

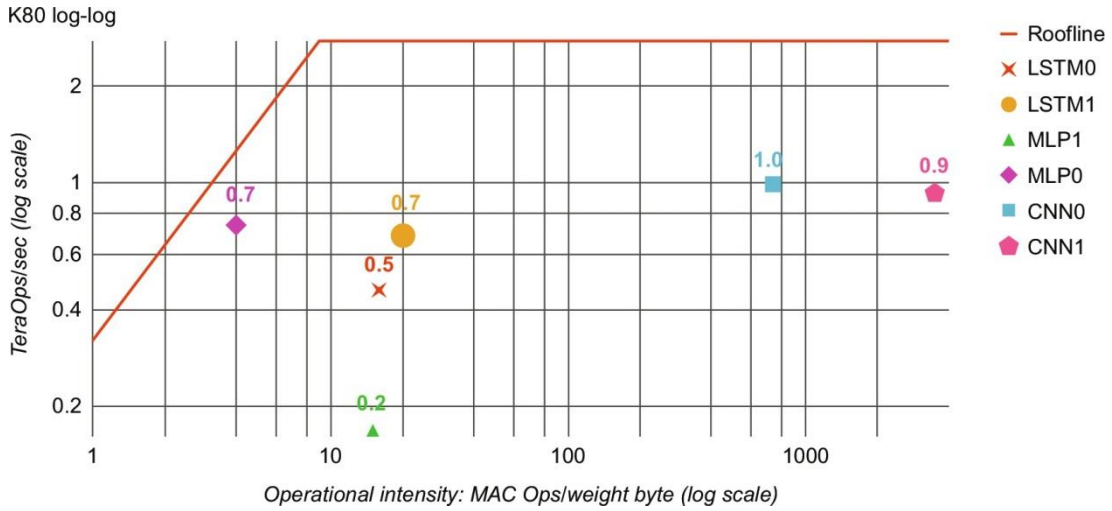

Figure 7.47 NVIDIA K80 GPU die Roofline. The much higher memory bandwidth moves the ridge point to 9 multiply-accumulate operations per weight byte, which is even further to the left than in Figure 7.46.

© 2019 Elsevier Inc. All rights reserved.

Type	Batch	99th% response	Inf/s (IPS)	% max IPS
CPU	16	7.2 ms	5482	42%
CPU	64	21.3 ms	13,194	100%
GPU	16	6.7 ms	13,461	37%
GPU	64	8.3 ms	36,465	100%
TPU	200	7.0 ms	225,000	80%
TPU	250	10.0 ms	280,000	100%

Figure 7.48 99th% response time and per die throughput (IPS) for MLP0 as batch size varies. The longest allowable latency is 7 ms. For the GPU and TPU, the maximum MLP0 throughput is limited by the host server overhead.

© 2019 Elsevier Inc. All rights reserved.

Type	MLP0	MLP1	LSTM0	LSTM1	CNN0	CNN1	Mean
GPU	2.5	0.3	0.4	1.2	1.6	2.7	1.9
TPU	41.0	18.5	3.5	1.2	40.3	71.0	29.2
Ratio	16.7	60.0	8.0	1.0	25.4	26.3	15.3

Figure 7.49 K80 GPU and TPU performance relative to CPU for the DNN workload. The mean uses the actual mix of the six applications in Figure 7.41. Relative performance for the GPU and TPU includes host server overhead. Figure 7.48 corresponds to the second column of this table (MLP0), showing relative IPS that meet the 7-ms latency threshold.

© 2019 Elsevier Inc. All rights reserved.

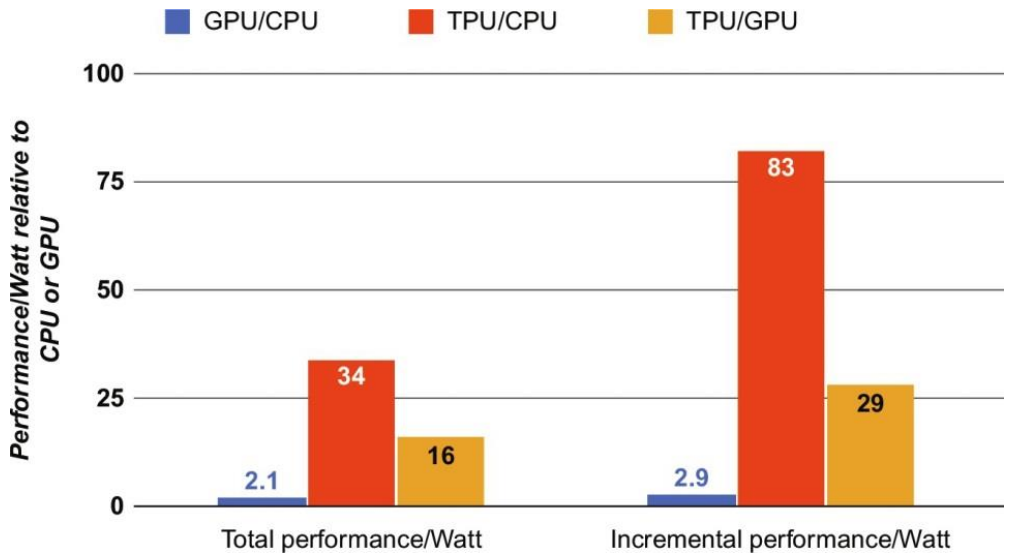

Figure 7.50 Relative performance/watt of GPU and TPU servers to CPU or GPU servers. Total performance/watt includes host server power, but incremental doesn't. It is a widely quoted metric, but we use it as a proxy for performance/TCO in the data center.

© 2019 Elsevier Inc. All rights reserved.

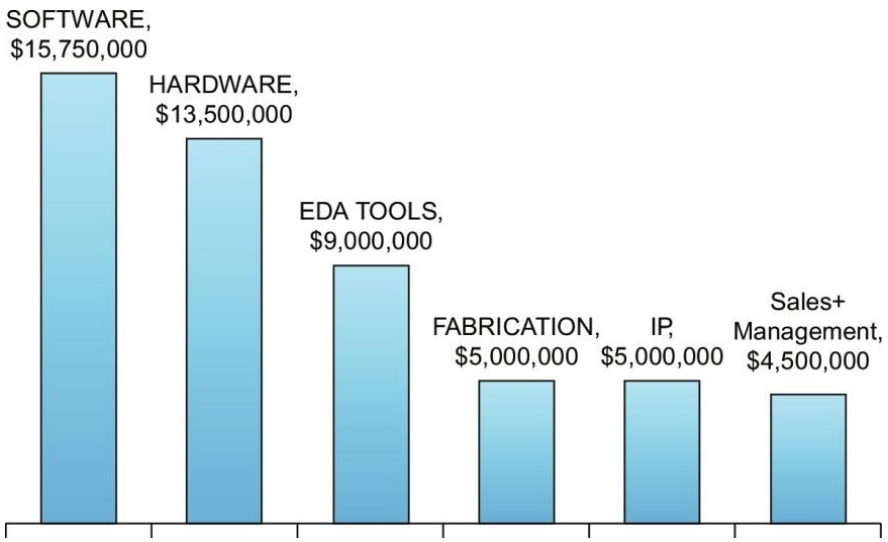

Figure 7.51 The breakdown of the $50 million cost of a custom ASIC that came from surveying others (Olofsson, 2011). The author wrote that his company spent just $2 million for its ASIC.

© 2019 Elsevier Inc. All rights reserved.