7

Domain-Specific Architectures

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

Gordon Moore,
Intel Co-Founder (2005)

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00007-9

7.1 Introduction

Gordon Moore not only predicted the amazing growth of transistors per chip in 1965, but the opening chapter quote shows that he also predicted its demise 50 years later. As evidence, Figure 7.1 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 100,000 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

These sophisticated architectures targeted million-line programs written in efficient languages like C++. Architects treated such code as black boxes, generally

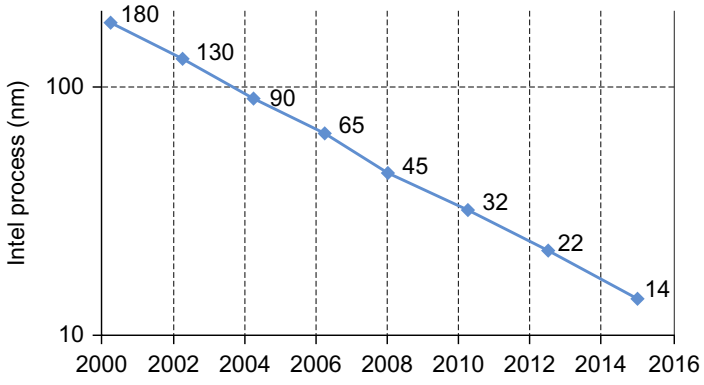


Figure 7.1 Time before new Intel semiconductor process technology measured in nm. The *y*-axis is log scale. Note that the time stretched previously from about 24 months per new process step to about 30 months since 2010.

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 described, 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 nothing left up our sleeves 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 7.2 is another take on the relative energy costs of memory and logic mentioned in Chapter 1, this time calculated as overhead for an arithmetic instruction. Given this overhead, minor twists to existing cores may get us 10% improvements, but if we want order-of-magnitude 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 past decade 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.

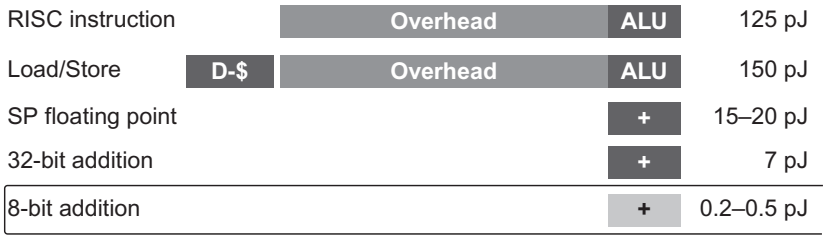


Figure 7.2 Energy costs in picoJoules for a 90 nm process to fetch instructions or access a data cache compared to the energy cost of arithmetic operations (Qadeer et al., 2015).

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 data sets like video that have little data reuse, multilevel caches are overkill, hording area and energy that could be put to better use. 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 focus on 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. 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 1000 chips.

One way to accommodate smaller volume applications is to use reconfigurable chips such as 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 (see Section 7.5). However, since the hardware is less efficient than custom chips, the gains from FPGAs are more modest.

Another DSA challenge is how to port software to it. 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 today. 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 four examples of DSAs: two custom chips for the data center that accelerate DNNs, an FPGA for the data center that accelerates many domains, and an image-processing unit designed for *personal mobile devices* (*PMDs*). We then compare the cost-performance of the DSAs along with CPUs and GPUs using DNN benchmarks, and conclude with a prediction of an upcoming renaissance for computer architecture.

7.2 Guidelines for DSAs

Here are five principles that generally guided the designs of the four DSAs we'll see in Sections 7.4–7.7. Not only do these five guidelines lead to increased area and energy efficiency, they also provide two valuable bonus effects. First, they lead to simpler designs, which reduce the cost of NRE of DSAs (see the fallacy in Section 7.10). 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 7.9. Figure 7.3 shows how the four 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 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 the resource-intensive optimizations for CPUs and GPUs (out-of-order execution, multithreading, multiprocessing, prefetching, address coalescing, etc.).

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.

Given the superior understanding of the execution of 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 VLIW can express the instruction-level parallelism for the domain, the design can be smaller and more energy-efficient than out-of-order execution.
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 let's you pack more arithmetic units into the same chip area.
5. *Use a domain-specific programming language to port code to the 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 assuming that your new computer is so attractive that programmers will rewrite their code just for your hardware. Fortunately, domain-specific programming languages were becoming popular even before architects were forced to switch their attention to DSAs. Examples are Halide for vision processing and TensorFlow for DNNs (Ragan-Kelley et al., 2013; Abadi et al., 2016). Such languages 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 7.4 lists the new acronyms, terms, and short explanations to aid the reader.

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 *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.

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* from example data as the path to artificial intelligence. 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 the 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 star for the past five years. 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 in the past five years reduced the error rate in an image recognition competition from 26% to 3.5% (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). Although many of these run in the cloud, they have also enabled Google Translate on smartphones, which we described in Chapter 1. In 2017 new, significant DNN results appear nearly every week.

Readers interested in learning more about DNNs than found in this section should download and try the tutorials in TensorFlow (TensorFlow Tutorials, 2016), or for the less adventurous, consult a free online textbook on DNNs (Nielsen, 2016).

The Neurons of DNNs

DNNs were inspired by the neuron of the brain. 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

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.

contained a dog, the output of the last layer could be a probability number between 0 and 1 or perhaps a list of probabilities corresponding to a list of dog breeds.

The number of layers gave DNNs their name. The original lack of data and computing horsepower kept most neural networks relatively shallow. Figure 7.5 shows the number of layers for a variety of recent DNNs, the number of weights, and the number of operations per weight fetched. In 2017 some DNNs have 150 layers.

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 7.5) 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 behavior of the model. Depending on the choice of neural architecture, there can be anywhere from thousands to hundreds of millions of weights in a single model (see Figure 7.5). 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*, *prediction*, *scoring*, *implementation*, *evaluation*, *running*, or *testing*. Most DNNs use *supervised learning* in that they are given a training set to learn from where the data is preprocessed in order 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 image in the 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 using 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 stands for "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) or TensorFlow Tutorials (2016).

Training can take weeks of computation, as Figure 7.6 shows. The inference phase is often below 100 ms per data sample, which is a million times less. 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!

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).

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 in 2017. 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.

Although it's hard to imagine a faster changing field, only three types of DNNs reign as most popular in 2017: *MultiLayer Perceptrons* (*MLPs*), *Convolutional Neural Networks* (*CNNs*), and *Recurrent Neural Networks* (*RNNs*). They are all examples of supervised learning, which rely on training sets.

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 7.7). 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

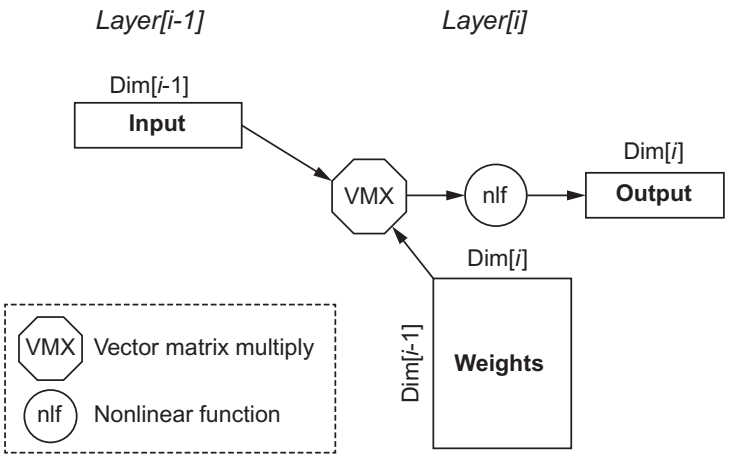


Figure 7.7 MLP showing the input Layer[*i*−1] on the left and the output Layer[*i*] on the right. ReLU is a popular nonlinear function for MLPs. The dimensions of the input and output layers are often different. Such a layer is called fully connected because it depends on all the inputs from the prior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

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[$i-1$] $\times$ Dim[i]
- Operations: $2 \times$ 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 millions of weights, which usually don't fit on the chip. For example, the dimensions of one stage of an MLP in Section 7.9 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 multiplies 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 7.8 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

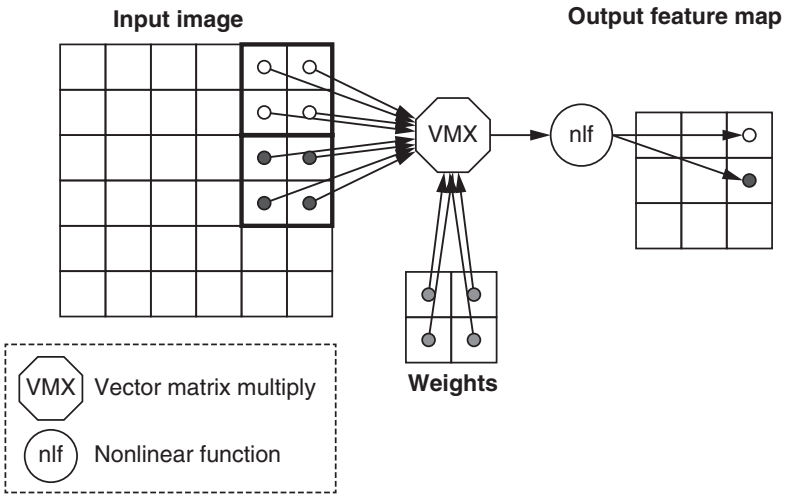


Figure 7.8 Simplified first step of a CNN. In this example, every group of four pixels of the input image are multiplied by the same four weights to create the cells of the output feature map. The pattern depicted shows a stride of two between the groups of input pixels, but other strides are possible. 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.

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 7.9 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.

As we see in Figure 7.9, CNNs are more complicated than MLPs. Here are the parameter 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

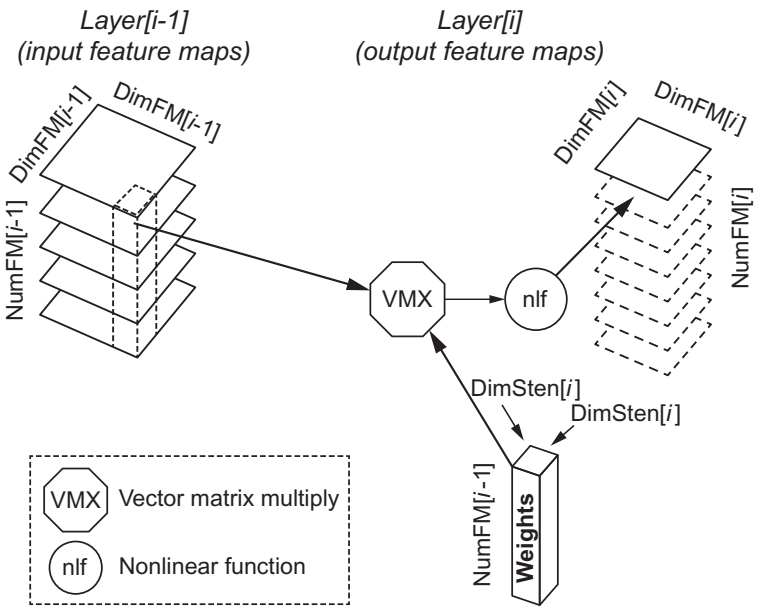


Figure 7.9 CNN general step showing input feature maps of Layer[$i-1$] on the left, the output feature maps of Layer[i] on the right, and a three-dimensional stencil over input feature maps to produce a single output feature map. Each output feature map has its own unique set of weights, and the vector-matrix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

- Number of weights per output Feature Map: $\text{NumFM}[i-1] \times \text{DimSten}[i]^2$
- Total number of weights per layer: $\text{NumFM}[i] \times$ Number of weights per output Feature Map
- Number of operations per output Feature Map: $2 \times \text{DimFM}[i]^2 \times$ Number of weights per output Feature Map
- Total number of operations per layer: $\text{NumFM}[i] \times$ Number of operations per output Feature Map $= 2 \times \text{DimFM}[i]^2 \times \text{NumFM}[i] \times$ Number of weights per output Feature Map $= 2 \times \text{DimFM}[i]^2 \times$ Total number of weights per layer
- Operations/Weight: $2 \times \text{DimFM}[i]^2$

A CNN in Section 7.9 has a layer with $\text{DimFM}[i-1] = 28$, $\text{DimFM}[i] = 14$, $\text{DimSten}[i] = 3$, $\text{NumFM}[i-1] = 64$ (number of input feature maps), and $\text{NumFM}[i] = 128$ (number of output feature maps). That layer has 25,088 neurons, 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 are 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.

Long short-term memory (*LSTM*) is by far the most popular RNN today. LSTMs mitigate a problem that previous RNNs had with their inability to remember important long-term information.

Unlike the other two DNNs, LSTM is a hierarchical design. LSTM consists of modules called *cells*. You can think of cells as templates or macros that are linked together to create the full DNN model, similar to how layers of an MLP line up to form a complete DNN model.

Figure 7.10 shows how the LSTM cells are linked together. They are hooked up from left to right, connecting the output of one cell to the input of the next. They are also unrolled in time, which runs top down in Figure 7.10. Thus a sentence is input a word at a time per iteration of the unrolled loop. The long-term and short-term memory information that gives the LSTM its name is also passed top-down from one iteration to the next.

Figure 7.11 shows the contents of an LSTM cell. As we would expect from Figure 7.10, the input is on the left, the output is on the right, the two memory inputs are at the top, and the two memory outputs are at the bottom.

Each cell does five vector-matrix multiplies using five unique sets of weights. The matrix multiply on the input is just like the MLP in Figure 7.7. Three others are called *gates* in that they gate or limit how much information from one source is passed along to the standard output or the memory output. The amount of information sent per gate is set by their weights. If the weights are mostly zeros or small values, then little gets through; conversely, if they are mostly large, then the gate lets most information flow. The three gates are called the *input gate*, the *output gate*, and the *forget gate*. The first two filter the input and output, and the last one determines what to forget along the long-term memory path.

The short-term memory output is a vector-matrix multiply using the Short Term Weights and the output of this cell. The short-term label is applied because it does not directly use any of the inputs to the cell.

Because the LSTM cell inputs and outputs are all connected together, the size of the three input-output pairs must be the same. Looking inside the cell, there are enough dependencies that all of the inputs and outputs are often the same size. Let's assume they are all the same size, called *Dim*.

Even so, the vector-matrix multiplies are not all the same size. The vectors for the three gate multiplies are $3 \times \text{Dim}$, because the LSTM concatenates all three inputs. The vector for the input multiply is $2 \times \text{Dim}$, because the LSTM

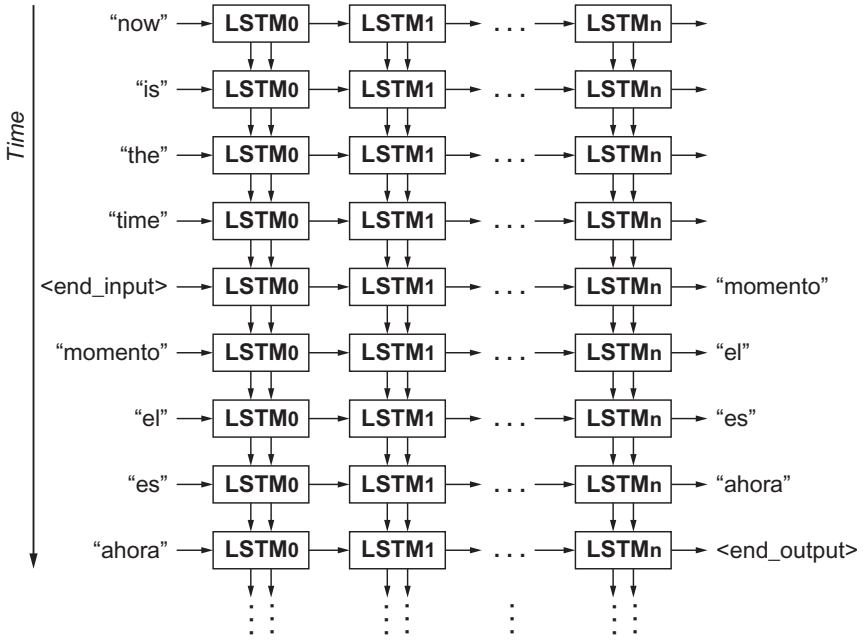


Figure 7.10 LSTM cells connected together. The inputs are on the left (English words), and the outputs are on the right (the translated Spanish words). The cells can be thought of as being unrolled over time, from top to bottom. Thus the short-term and long-term memory of LSTM is implemented by passing 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.)

concatenates the input with the short-term memory input as the vector. The vector for the last multiply is just $1 \times \text{Dim}$, because it is just the output.

Now we can finally calculate the weights and operations:

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

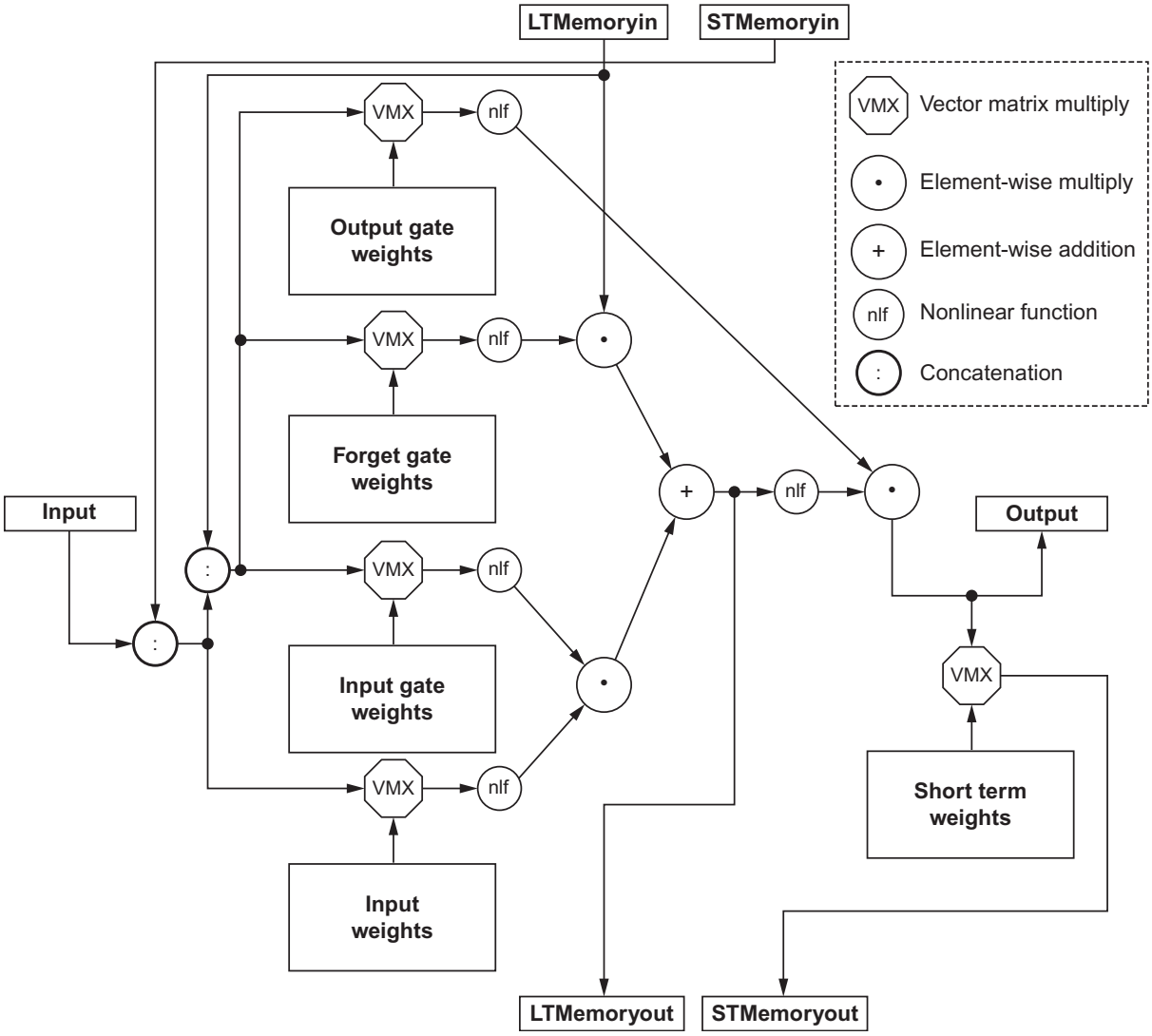


Figure 7.11 This LSTM cell contains 5 vector-matrix multiplies, 3 element-wise multiplies, 1 element-wise add, and 6 nonlinear functions. The standard input and short-term memory input are concatenated to form the vector operand for the input vector-matrix multiply. The standard input, long-term memory 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.)

Dim is 1024 for one of the six cells of an LSTM in Section 7.9. Its number of weights is 12,582,912, its number of operations is 25,169,920, and its operational intensity is 2.0003. Thus LSTMs are like MLPs in that they typically have more weights and a lower operational intensity than CNNs.

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 datasets 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 in order to train well.

Looking at an MLP in Figure 7.7, 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. A sequence of row vector inputs to the five matrix multiplies of LSTMs in Figure 7.11 can also be considered a matrix. In both cases, computing them as matrices instead of sequentially as independent vectors improves computing efficiency.

Quantization

Numerical precision is less important for DNNs than for many applications. For example, there is no need for double-precision floating-point arithmetic, which is the standard bearer of high-performance computing. It's even unclear that 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, some developers use fixed point instead of floating point for the inference phase. (Training is almost always done in floating-point arithmetic.) This conversion is called *quantization*, and such a transformed application is said to be *quantized* (Vanhoucke et al., 2011). The fixed-point data width is usually 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).

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 four sections explore.

7.4 Google's Tensor Processing Unit, an Inference Data Center Accelerator

The Tensor Processing Unit (*TPU*)[1] is Google's first custom ASIC DSA for WSCs. Its domain is the inference phase of DNNs, and it is programmed using the TensorFlow framework, which was designed for DNNs. The first TPU was been deployed in Google data centers in 2015.

The heart of the TPU is a 65,536 (256×256) 8-bit ALU Matrix Multiply Unit and a large software-managed on-chip memory. The TPU's single-threaded, deterministic execution model is a good match to the 99th-percentile response-time requirement of the typical DNN inference application.

TPU Origin

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 three 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 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 $10\times$ over GPUs. Given this mandate, the TPU was designed, verified (Steinberg, 2015), built, and deployed in data centers in just 15 months.

TPU Architecture

To reduce the chances of delaying deployment, the TPU was designed to be a coprocessor on the PCIe I/O bus, which allows it to be plugged into existing servers. Moreover, to simplify hardware design and debugging, the host server sends instructions over the PCIe bus directly to the TPU for it to execute, rather than having the TPU fetch the instructions. Thus the TPU is closer in spirit to an FPU (floating-point unit) coprocessor than it is to a GPU, which fetches instructions from its memory.

Figure 7.12 shows the block diagram of the TPU. The host CPU sends TPU instructions over the PCIe bus into an instruction buffer. The internal blocks are typically connected together by 256-byte-wide (2048-bits) paths. Starting in the upper-right corner, the *Matrix Multiply Unit* is the heart of the TPU. It contains

[1]This section is based on the paper "In-Datacenter Performance Analysis of a Tensor Processing Unit" Jouppi et al., 2017, of which one of your book authors was a coauthor.

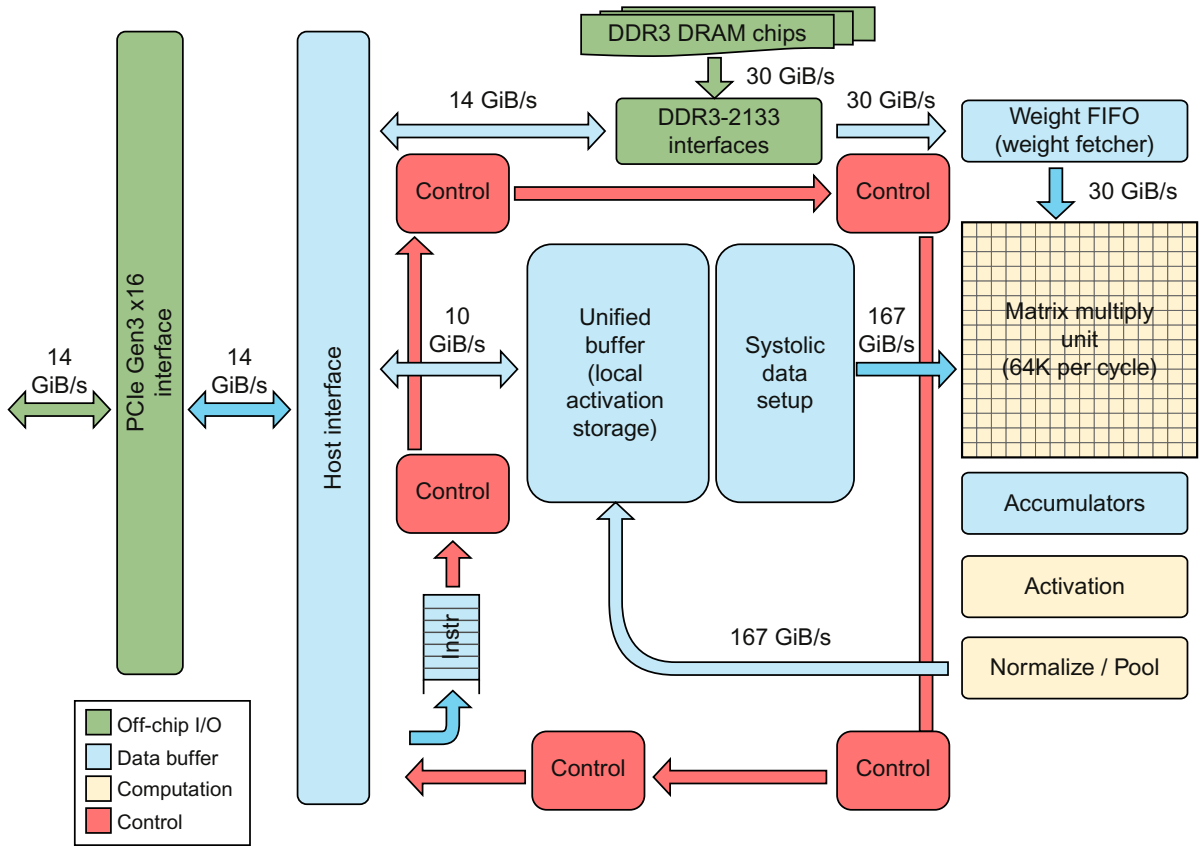


Figure 7.12 TPU Block Diagram. The PCIe bus is Gen3 ×16. The main computation part is the light-shaded Matrix Multiply Unit in the upper-right corner. Its inputs are the medium-shaded Weight FIFO and the medium-shaded Unified Buffer and its output is the medium-shaded Accumulators. The light-shaded Activation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

256 × 256 ALUs that can perform 8-bit multiply-and-adds on signed or unsigned integers. The 16-bit products are collected in the 4 MiB of 32-bit *Accumulators* below the matrix unit. When using a mix of 8-bit weights and 16-bit activations (or vice versa), the Matrix Unit computes at half-speed, and it computes at a quarter-speed when both are 16 bits. It reads and writes 256 values per clock cycle and can perform either a matrix multiply or a convolution. The nonlinear functions are calculated by the *Activation* hardware.

The weights for the matrix unit are staged through an on-chip *Weight FIFO* that reads from an off-chip 8 GiB DRAM called *Weight Memory* (for inference, weights are read-only; 8 GiB supports many simultaneously active models). The intermediate results are held in the 24 MiB on-chip *Unified Buffer*, which can serve as inputs to the Matrix Multiply Unit. A programmable DMA controller transfers data to or from CPU Host memory and the Unified Buffer.

TPU Instruction Set Architecture

As instructions are sent over the relatively slow PCIe bus, TPU instructions follow the CISC tradition, including a repeat field. The TPU does not have a program counter, and it has no branch instructions; instructions are sent from the host CPU. The clock cycles per instruction (CPI) of these CISC instructions are typically 10–20. It has about a dozen instructions overall, but these five are the key ones:

1. `Read_Host_Memory` reads data from the CPU host memory into the Unified Buffer.
2. `Read_Weights` reads weights from Weight Memory into the Weight FIFO as input to the Matrix Unit.
3. `MatrixMultiply/Convolve` causes the Matrix Multiply Unit to perform a matrix-matrix multiply, a vector-matrix multiply, an element-wise matrix multiply, an element-wise vector multiply, or a convolution from the Unified Buffer into the Accumulators. A matrix operation takes a variable-sized B*256 input, multiplies it by a 256×256 constant input, and produces a B*256 output, taking B pipelined cycles to complete. For example, if the input were 4 vectors of 256 elements, B would be 4, so it would take 4 clock cycles to complete.
4. `Activate` performs the nonlinear function of the artificial neuron, with options for ReLU, Sigmoid, tanh, and so on. Its inputs are the Accumulators, and its output is the Unified Buffer.
5. `Write_Host_Memory` writes data from the Unified Buffer into the CPU host memory.

The other instructions are alternate host memory read/write, set configuration, two versions of synchronization, interrupt host, debug-tag, nop, and halt. The CISC MatrixMultiply instruction is 12 bytes, of which 3 are Unified Buffer address; 2 are accumulator address; 4 are length (sometimes 2 dimensions for convolutions); and the rest are opcode and flags.

The goal is to run whole inference models in the TPU to reduce interactions with the host CPU and to be flexible enough to match the DNN needs of 2015 and beyond, instead of just what was required for 2013 DNNs.

TPU Microarchitecture

The microarchitecture philosophy of the TPU is to keep the Matrix Multiply Unit busy. The plan is to hide the execution of the other instructions by overlapping their execution with the `MatrixMultiply` instruction. Thus each of the preceding four general categories of instructions have separate execution hardware (with read and write host memory combined into the same unit). To increase instruction parallelism further, the `Read_Weights` instruction follows the decoupled access/

execute philosophy (Smith, 1982b) in that they can complete after sending their addresses but before the weights are fetched from Weight Memory. The matrix unit has not-ready signals from the Unified Buffer and the Weight FIFO that will cause the matrix unit to stall if their data are not yet available.

Note that a TPU instruction can execute for many clock cycles, unlike the traditional RISC pipeline with one clock cycle per stage.

Because reading a large SRAM is much more expensive than arithmetic, the Matrix Multiply Unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer (Kung and Leiserson, 1980; Ramacher et al., 1991; Ovtcharov et al., 2015b). 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 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.

In the TPU, the systolic array is rotated. Figure 7.14 shows that the weights are loaded from the top and the input data flows into the array in from the left. A given 256-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 256 inputs are read at once, and after a feed delay, they update one location of each of 256 accumulator memories. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit.

TPU Implementation

The TPU chip was fabricated using the 28-nm process. The clock rate is 700 MHz. Figure 7.15 shows the floor plan of the TPU. Although the exact die size is not revealed, it is less than half the size of an Intel Haswell server microprocessor, which is 662 mm^2.

The 24 MiB Unified Buffer is almost a third of the die, and the Matrix Multiply Unit is a quarter, so the datapath is nearly two-thirds of the die. The 24 MiB size was picked in part to match the pitch of the Matrix Unit on the die and, given the

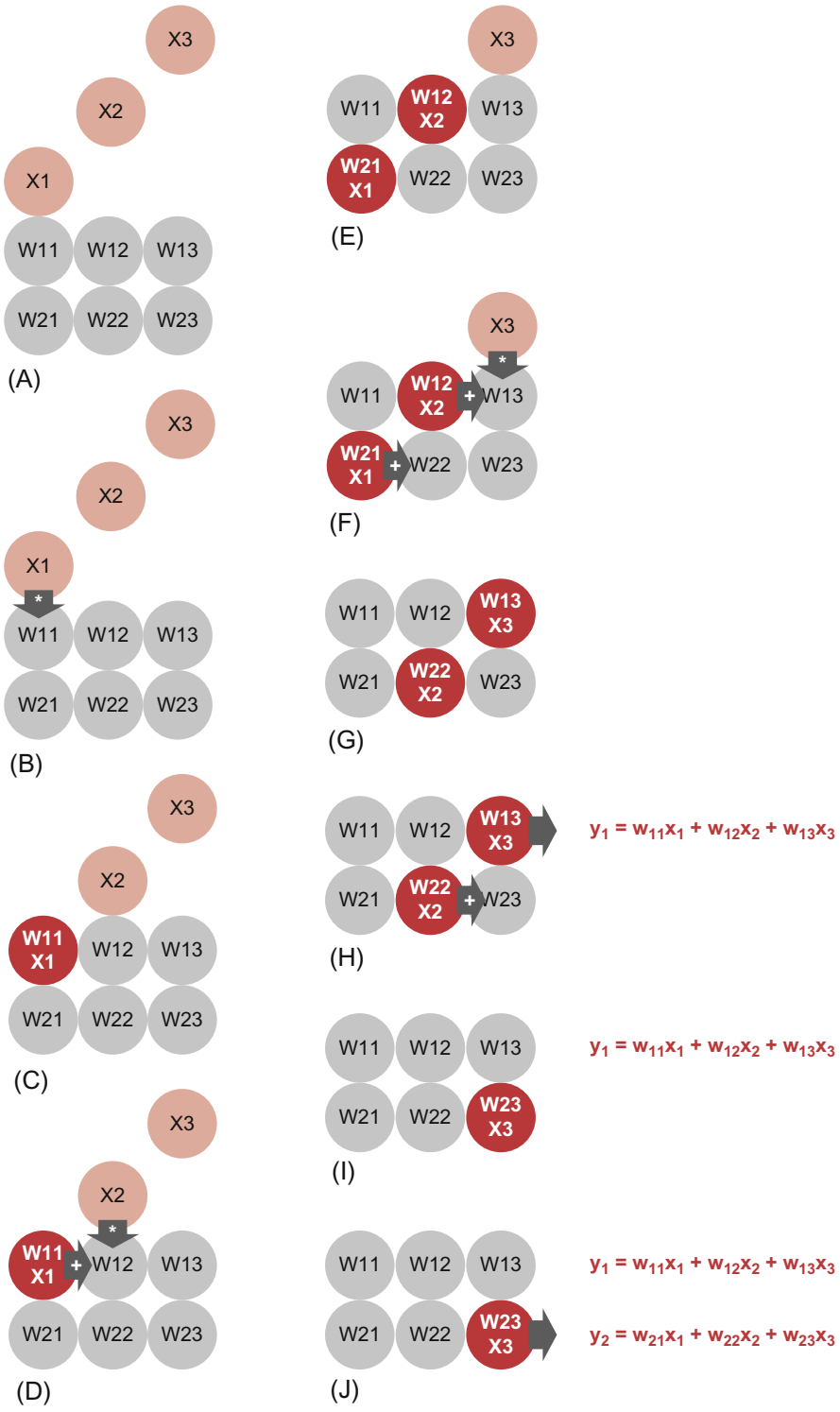


Figure 7.13 Example of systolic array in action, from top to bottom on the page. In this example, the six weights are already inside the multiply-accumulate units, as is the norm for the TPU. The three inputs are staggered in time to get the desired effect, and in this example are shown coming in from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

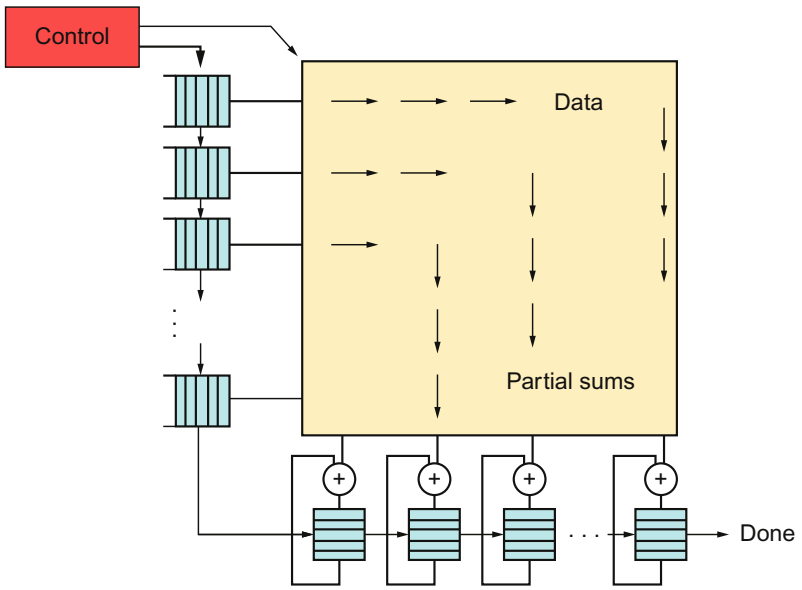


Figure 7.14 Systolic data flow of the Matrix Multiply Unit.

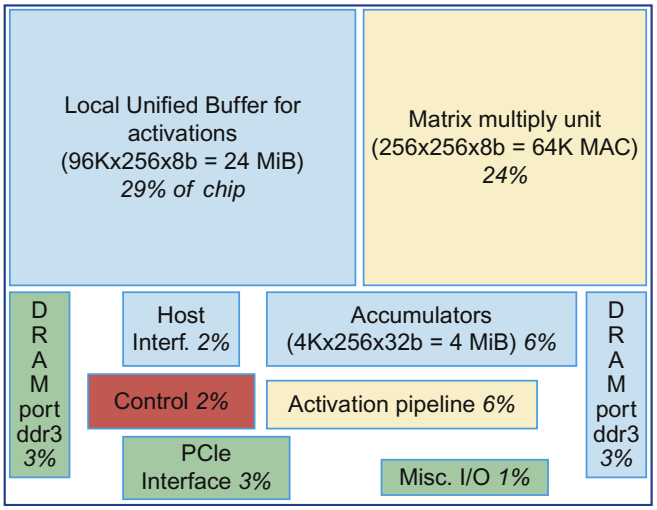


Figure 7.15 Floor plan of TPU die. The shading follows Figure 7.14. The light data buffers are 37%, the light computation units are 30%, the medium I/O is 10%, and the dark control is just 2% of the die. Control is much larger (and much more difficult to design) in a CPU or GPU. The unused white space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

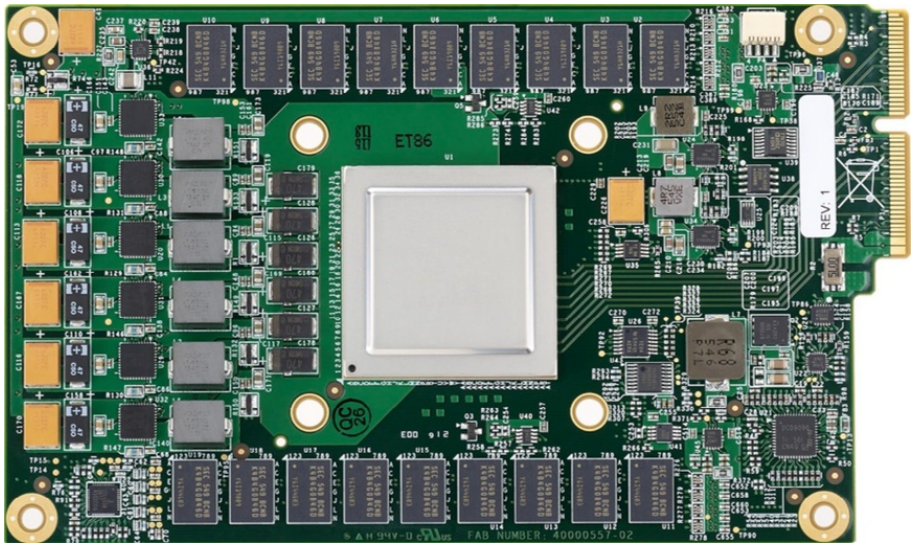


Figure 7.16 TPU printed circuit board. It can be inserted into the slot for an SATA disk in a server, but the card uses the PCIe bus.

short development schedule, in part to simplify the compiler. Control is just 2%. Figure 7.16 shows the TPU on its printed circuit card, which inserts into existing servers in a SATA disk slot.

TPU Software

The TPU software stack had to be compatible with that developed for CPUs and GPUs so that applications could be ported quickly. The portion of the application run on the TPU is typically written using TensorFlow and is compiled into an API that can run on GPUs or TPUs (Larabel, 2016). Figure 7.17 shows TensorFlow code for a portion of an MLP.

Like GPUs, the TPU stack is split into a User Space Driver and a Kernel Driver. The Kernel Driver is lightweight and handles only memory management and interrupts. It is designed for long-term stability. The User Space driver changes frequently. It sets up and controls TPU execution, reformats data into TPU order, and translates API calls into TPU instructions and turns them into an application binary. The User Space driver compiles a model the first time it is evaluated, caching the program image and writing the weight image into the TPU Weight Memory; the second and following evaluations run at full speed. The TPU runs most models completely from inputs to outputs, maximizing the ratio of TPU compute time to I/O time. Computation is often done one layer at a time, with overlapped execution allowing the matrix unit to hide most noncritical path operations.

```
# 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.

Improving the TPU

The TPU architects looked at variations of the microarchitecture to see whether they could have improved the TPU.

Like an FPU, the TPU coprocessor has a relatively easy microarchitecture to evaluate, so the TPU architects created a performance model and estimated performance as the memory bandwidth, the matrix unit size, and the clock rate and number of accumulators varied. Measurements using TPU hardware counters found that the modeled performance was on average within 8% of the hardware.

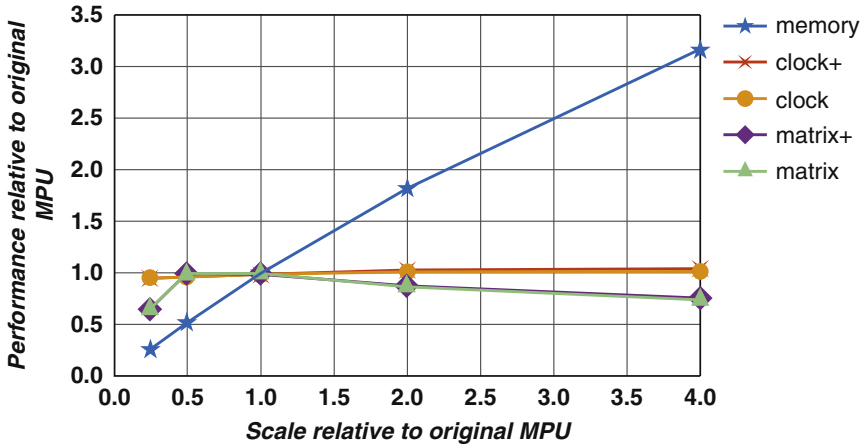


Figure 7.18 Performance as metrics scale from 0.25× to 4×: memory bandwidth, clock rate + accumulators, clock rate, matrix unit dimension + accumulators, and one dimension of the square matrix unit This is the average performance calculated from six DNN applications in Section 7.9. The CNNs tend to be 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.

Figure 7.18 shows the performance sensitivity of the TPU as these parameters scale over the range for 0.25 × to 4 ×. (Section 7.9 lists the benchmarks used.) In addition to evaluating the impact of only raising clock rates (*clock* in Figure 7.18), Figure 7.18 also plots a design (*clock+*) that increases the clock rate and scales the number of accumulators correspondingly so that the compiler can keep more memory references in flight. Likewise, Figure 7.18 plots matrix unit expansion if the number of accumulators increase with the square of the rise in one dimension (*matrix+*), because the matrix grows in both dimensions, as well as only increasing the matrix unit (*matrix*).

First, increasing memory bandwidth (*memory*) has the biggest impact: performance improves 3 × on average when memory bandwidth increases 4 ×, because it reduces the time waiting for weight memory. Second, clock rate has little benefit on average with or without more accumulators. Third, the average performance in Figure 7.18 slightly *degrades* when the matrix unit expands from 256 × 256 to 512 × 512 for all applications, whether or not they get more accumulators. The issue is analogous to internal fragmentation of large pages, only worse because it's in two dimensions.

Consider the 600 × 600 matrix used in LSTM1. With a 256 × 256 matrix unit, it takes nine steps to tile 600 × 600, for a total of 18 µs of time. The larger 512 × 512 unit requires only four steps, but each step takes four times longer, or 32 µs of time. The TPU's CISC instructions are long, so decode is insignificant and does not hide the overhead of loading from the DRAM.

Given these insights from the performance model, the TPU architects next evaluated an alternative and hypothetical TPU that they might have designed in the same process technology if they'd had more than 15 months to do so. More aggressive logic synthesis and block design might have increased the clock rate by 50%. The architects found that designing an interface circuit for GDDR5 memory, as used by the K80, would improve Weight Memory bandwidth by more than a factor of five. As Figure 7.18 shows, increasing clock rate to 1050 MHz, but not helping memory, made almost no change in performance. If the clock is left at 700 MHz, but it uses GDDR5 instead for Weight Memory, performance is increased by 3.2 ×, even accounting for the host CPU overhead of invoking the DNN on the revised TPU. Doing both does not improve average performance further.

Summary: How TPU Follows the Guidelines

Despite living on an I/O bus and having relatively little memory bandwidth that limits full utilization of the TPU, a small fraction of a big number can, nonetheless, be relatively large. As we will see in Section 7.9, the TPU delivered on its goal of a tenfold improvement in cost-performance over the GPU when running DNN inference applications. Moreover, a redesigned TPU with the only change being a switch to the same memory technology as in the GPU would be three times faster.

One way to explain the TPU's success is to see how it followed the guidelines in Section 7.2.

1. *Use dedicated memories to minimize the distance over which data is moved.*
   The TPU has the 24 MiB Unified Buffer that holds the intermediate matrices and vectors of MLPs and LSTMs and the feature maps of CNNs. It is optimized for accesses of 256 bytes at a time. It also has the 4 MiB Accumulators, each 32-bits wide, that collect the output of the Matrix Unit and act as input to the hardware that calculates the nonlinear functions. The 8-bit weights are stored in a separate off-chip weight memory DRAM and are accessed via an on-chip weight FIFO. In contrast, all these types and sizes of data would exist in redundant copies at several levels of the inclusive memory hierarchy of a general-purpose CPU.
2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.*
   The TPU offers 28 MiB of dedicated memory and 65,536 8-bit ALUs, which means it has about 60% of the memory and 250 times as many ALUs as a server-class CPU, despite being half its size and power (see Section 7.9). Compared to a server-class GPU, the TPU has 3.5 times the on-chip memory and 25 times as many ALUs.
3. *Use the easiest form of parallelism that matches the domain.*
   The TPU delivers its performance via a two-dimensional SIMD parallelism with its 256 × 256 Matrix Multiply Unit, which is internally pipelined with a systolic organization, plus a simple overlapped execution pipeline of its

instructions. GPUs rely instead on multiprocessing, multithreading, and one-dimensional SIMD, and CPUs rely on multiprocessing, out-of-order execution, and one-dimensional SIMD.

4. *Reduce data size and type to the simplest needed for the domain*.
   The TPU computes primarily on 8-bit integers, although it supports 16-bit integers and accumulates in 32-bit integers. CPUs and GPUs 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 the TensorFlow programming framework, whereas GPUs rely on CUDA and OpenCL and CPUs must run virtually everything.

7.5 Microsoft Catapult, a Flexible Data Center Accelerator

At the same time that Google was thinking about deploying a custom ASIC in its data centers, Microsoft was considering accelerators for theirs. The Microsoft perspective was that any solution had to follow these guidelines:

- It had to preserve homogeneity of servers to enable rapid redeployment of machines and to avoid making maintenance and scheduling even more complicated, even if that notion is a bit at odds with the concept of DSAs.
- It had to scale to applications that might need more resources than could fit into a single accelerator without burdening all applications with multiple accelerators.
- It needed to be power-efficient.
- It couldn't become a dependability problem by being a single point of failure.
- It had to fit within the available spare space and power in existing servers.
- It could not hurt data center network performance or reliability.
- The accelerator had to improve the cost-performance of the server.

The first rule prevented deploying an ASIC that helped only some applications on some servers, which was the decision that Google made.

Microsoft started a project called Catapult that placed an FPGA on a PCIe bus board into data center servers. These boards have a dedicated network for applications that need more than one FPGA. The plan was to use the flexibility of the FPGA to tailor its use for varying applications both on different servers and to reprogram the same server to accelerate distinct applications over time. This plan increased the return on its investment of the accelerator. Another advantage of FPGAs is that they should have lower NRE than ASICs, which could again improve return on investment. We discuss two generations of Catapult, showing how the design evolved to meet the needs of WSCs.

One interesting upside of FPGAs is that each application—or even each phase of an application—can be thought of as its own DSA, so in this section, we get to see many examples of novel architectures in one hardware platform.

Catapult Implementation and Architecture

Figure 7.19 shows a PCIe board that Microsoft designed to fit within its servers, which limited power and cooling to 25 W. This constraint led to the selection of the 28-nm Altera Stratix V D5 FPGA for its first implementation of Catapult. The board also has 32 MiB of flash memory and includes two banks of DDR3-1600 DRAM with a total capacity of 8 GiB. The FPGA has 3926 18-bit ALUs, 5 MiB of on-chip memory, and 11 GB/s bandwidth to DDR3 DRAMs.

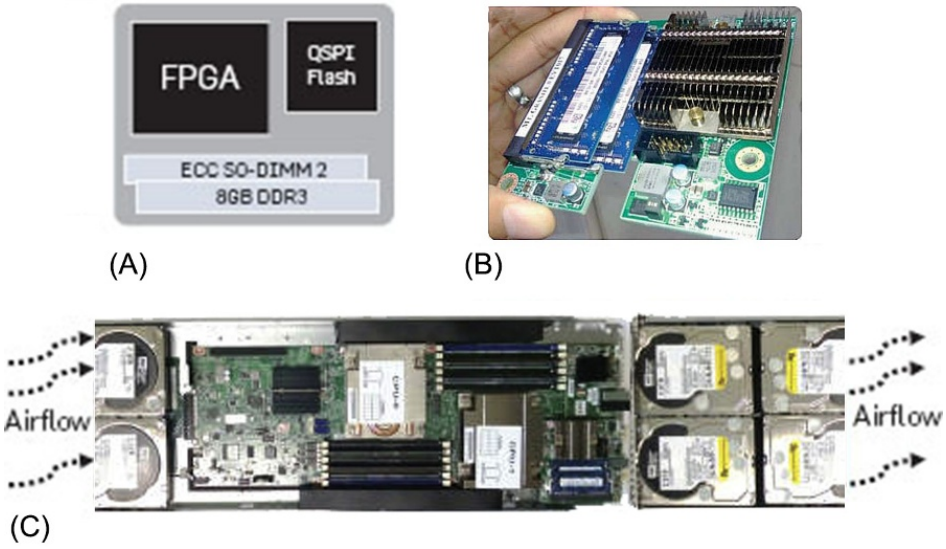


Figure 7.19 The Catapult board design. (A) shows the block diagram, and (B) is a photograph of both sides of the board, which is 10 cm × 9 cm × 16 mm. The PCIe and inter-FPGA network are wired to a connector on the bottom of the board that plugs directly into the motherboard. (C) is a photograph of the 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.

Each of the 48 servers in half of a data center rack contains a Catapult board. Catapult follows the preceding guidelines about supporting applications that need more than a single FPGA without affecting the performance of the data center network. It adds a separate low-latency 20 Gbit/s network that connects 48 FPGAs. The network topology is a two-dimensional 6×8 torus network.

To follow the guideline about not being a single point of failure, this network can be reconfigured to operate even if one of the FPGAs fails. The board also has SECDED protection on all memories outside the FPGA, which is required for large-scale deployment in a data center.

Because FPGAs use a great deal of memory on the chip to deliver programmability, they are more vulnerable than ASICs to *single-event upsets (SEUs)* because of radiation as the process geometries shrink. The Altera FPGA in Catapult boards includes mechanisms to detect and correct SEUs inside the FPGA and reduces the chances of SEUs by periodically scrubbing the FPGA configuration state.

The separate network has an added benefit of reducing the variability of communication performance as compared to a data center network. Network unpredictability increases tail latency—which is especially detrimental for applications that face end users—so a separate network makes it easier to successfully offload work from the CPU to the accelerator. This FPGA network can run a much simpler protocol than in the data center because the error rates are considerably lower and the network topology is well defined.

Note that resiliency requires care when reconfiguring FPGAs so that they neither appear as failed nodes nor crash the host server or corrupt their neighbors. Microsoft developed a high-level protocol for ensuring safety when reconfiguring one or more FPGAs.

Catapult Software

Possibly the biggest difference between Catapult and the TPU is having to program in a hardware-description language such as Verilog or VHDL. As the Catapult authors write (Putnam et al., 2016):

> Going forward, the biggest obstacle to widespread adoption of FPGAs in the datacenter is likely to be programmability. FPGA development still requires extensive hand-coding in Register Transfer Level and manual tuning.

To reduce the burden of programming Catapult FPGAs, the Register Transfer Level (RTL) code is divided into the *shell* and the *role*, as Figure 7.20 shows. The shell code is like the system library on an embedded CPU. It contains the RTL code that will be reused across applications on the same FPGA board, such as data marshaling, CPU-to-FPGA communication, FPGA-to-FPGA communication, data movement, reconfiguration, and health monitoring. The shell RTL code is 23% of the Altera FPGA. The role code is the application logic, which the Catapult programmer writes using the remaining 77% of the FPGA resources. Having a shell has the added benefit of offering a standard API and standard behavior across applications.

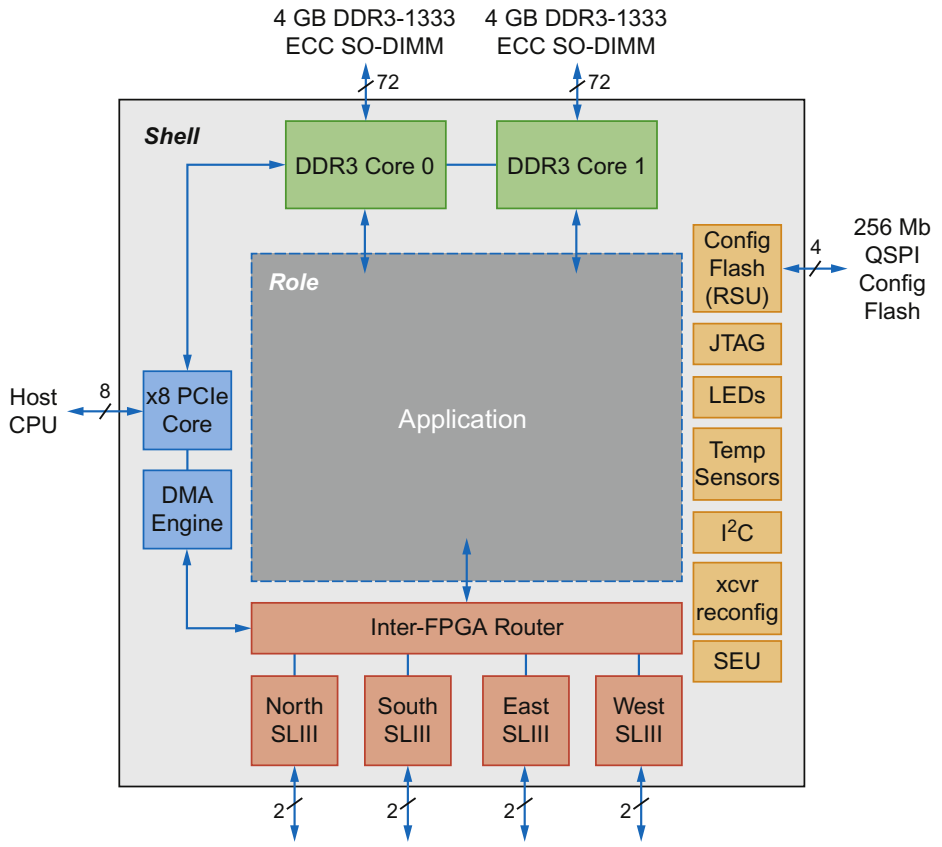


Figure 7.20 Components of Catapult shell and role split of the RTL code.

CNNs on Catapult

Microsoft developed a configurable CNN accelerator as an application for Catapult. Configuration parameters include the number of neural network layers, the dimension of those layers, and even the numerical precision to be used. Figure 7.21 shows the block diagram of the CNN accelerator. Its key features are:

- Run-time configurable design, without requiring recompilation using the FPGA tools.
- To minimize memory accesses, it offers efficient buffering of CNN data structures (see Figure 7.21).
- A two-dimensional array of Processing Elements (PEs) that can scale up to thousands of units.

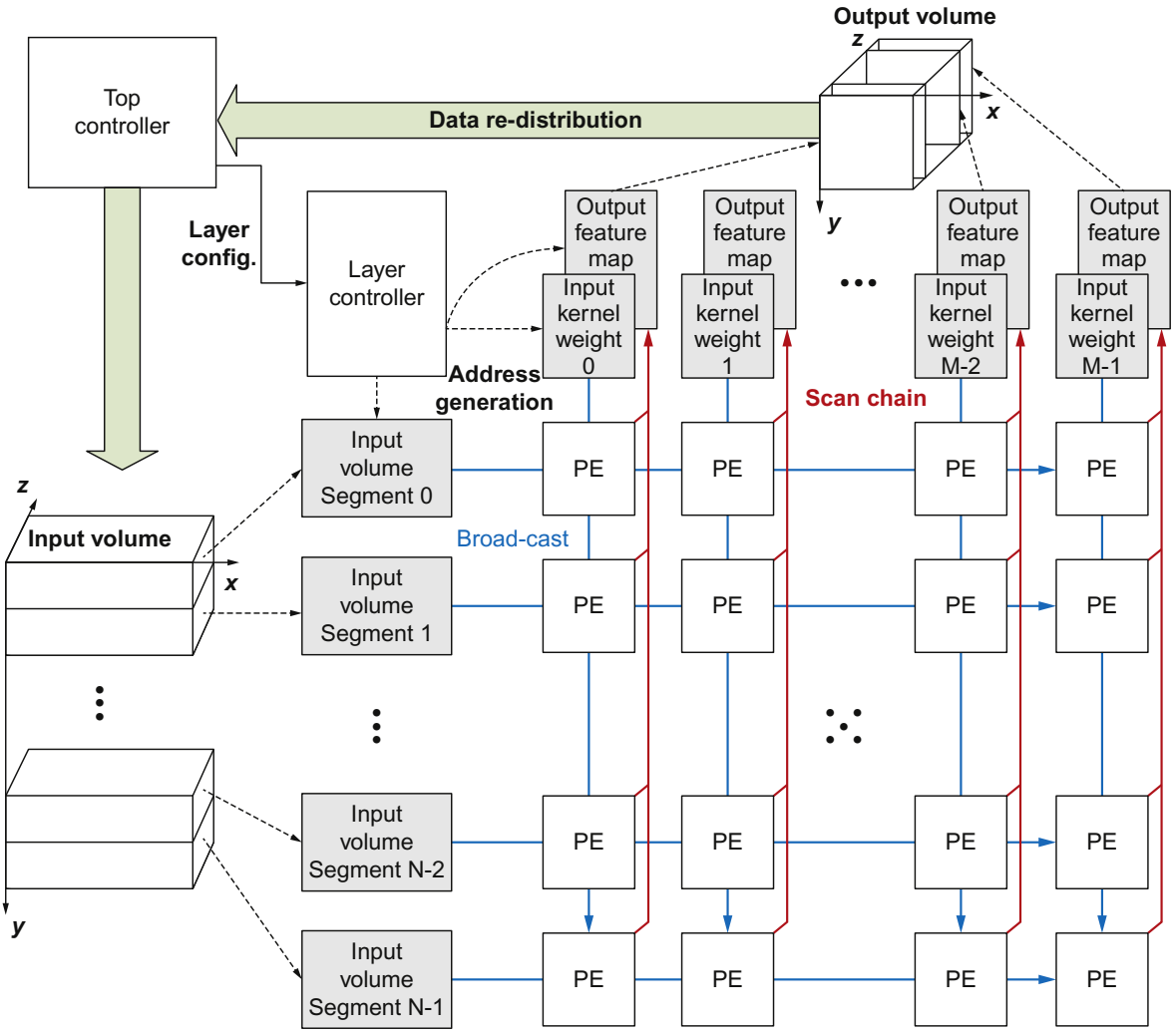


Figure 7.21 CNN Accelerator for Catapult. The Input Volume of the left correspond to Layer[$i-1$] on the left of Figure 7.20, with NumFM[$i-1$] corresponding to y and DimFM[$i-1$] corresponding to z. Output Volume at the top maps to Layer[i], with z mapping to NumFM[i] and DimFM[i] mapping to x. The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

Images are sent to DRAM and then input into a multibank buffer in the FPGA. The inputs are sent to multiple PEs to perform the stencil computations that produce the output feature maps. A controller (upper left in Figure 7.21) orchestrates the flow of data to each PE. The final results are then recirculated to the input buffers to compute the next layer of the CNN.

Like the TPU, the PEs are designed to be used as a systolic array. Figure 7.22 shows the details of the PE design.

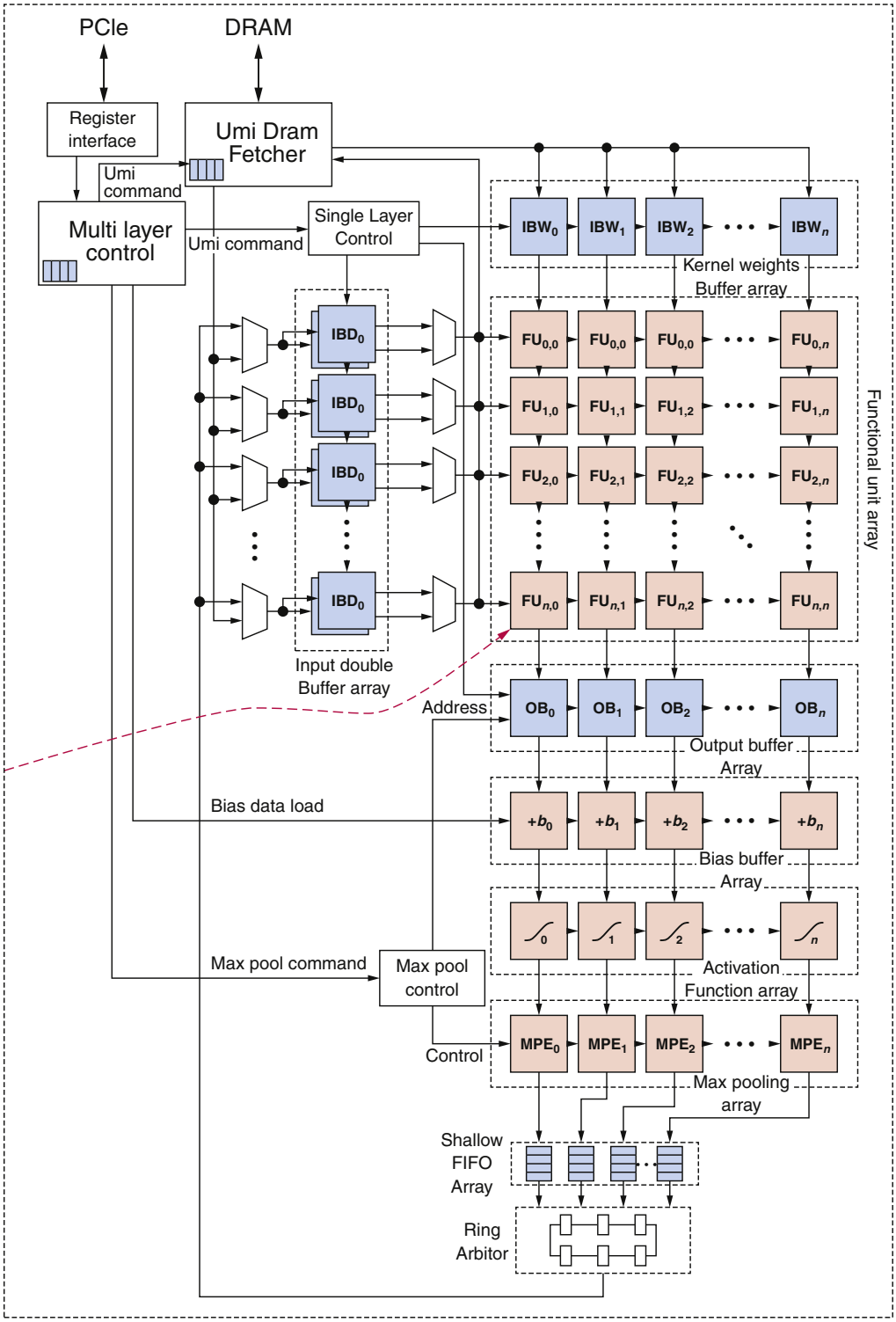


Figure 7.22 The Processing Element (PE) of the CNN Accelerator for Catapult in Figure 7.21. The two-dimension Functional Units (FU) consist of just an ALU and a few registers.

Search Acceleration on Catapult

The primary application to test the return on investment of Catapult was a critical function of the Microsoft Bing search engine called *ranking*. It ranks the order of the results from a search. The output is a document score, which determines the position of the document on the web page that is presented to the user. The algorithm has three stages:

1. *Feature Extraction* extracts thousands of interesting features from a document based on the search query, such as the frequency that the query phrase appears in a document.
2. *Free-Form Expressions* calculates thousands of combinations of features from the prior stage.
3. *Machine-Learned Scoring* uses machine-learning algorithms to evaluate the features from the first two stages to calculate a floating-point score of a document that is returned to the host search software.

The Catapult implementation of ranking produces identical results to equivalent Bing software, even reproducing known bugs!

Taking advantage of one of the preceding guidelines, the ranking function does not have to fit within a single FPGA. Here is how the ranking stages are split across eight FPGAs:

- One FPGA does Feature Extraction.
- Two FPGAs do Free-Form Expressions.
- One FPGA does a compression stage that increases scoring engine efficiency.
- Three FPGA do Machine-Learned Scoring.

The remaining FPGA is a spare used to tolerate faults. Using multiple FPGAs for one application works well because of the dedicated FPGA network.

Figure 7.23 shows the Feature Extraction stage organization. It uses 43 feature-extraction state machines to compute in parallel 4500 features per document-query pair.

Next is the following Free-Form Expressions stage. Rather than implement the functions directly in gates or in state machines, Microsoft developed a 60-core processor that overcomes long-latency operations with multithreading. Unlike a GPU, Microsoft's processor does not require SIMD execution. It has three features that let it match the latency target:

1. Each core supports four simultaneous threads where one can stall on a long operation but the others can continue. All functional units are pipelined, so they can accept a new operation every clock cycle.
2. Threads are statically prioritized using a priority encoder. Expressions with the longest latency use thread slot 0 on all cores, then the next slowest is in slot 1 on all cores, and so on.

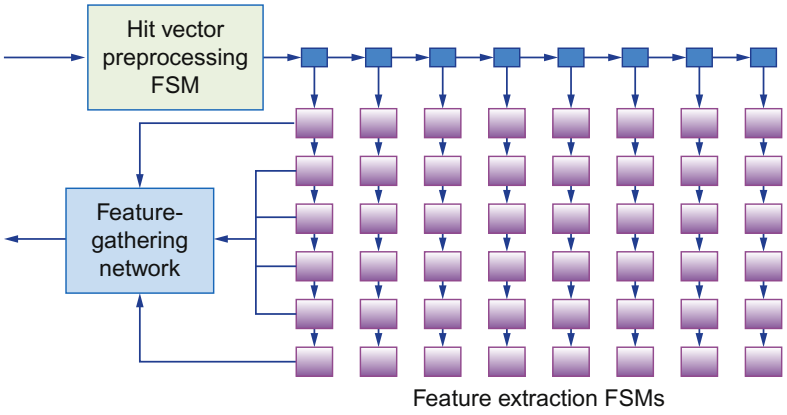


Figure 7.23 The architecture of FPGA implementation of the Feature Extraction stage. A hit vector, which describes the locations of query words in each document, is streamed into the hit vector preprocessing state machine and then split into control and data tokens. These tokens are issued in parallel 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.

3. Expressions that are too large to fit in the time allocated for a single FPGA can be split across the two FPGAs used for free-form expressions.

One cost of the reprogrammability in an FPGA is a slower clock rate than custom chips. Machine-Learned Scoring uses two forms of parallelism to try to overcome that disadvantage. The first is to have a pipeline that matches the available pipeline parallelism in the application. For ranking, the limit is 8 µs per stage. The second version of parallelism is the rarely seen *multiple instruction streams, single data stream (MISD)* parallelism, where a large number of independent instruction streams operate in parallel on a single document.

Figure 7.24 shows the performance of the ranking function on Catapult. As we will see in Section 7.9, user-facing applications often have rigid response times; it doesn't matter how high the throughput is if the application misses the deadline. The *x*-axis shows the response-time limit, with 1.0 as the cutoff. At this maximum latency, Catapult is 1.95 times as fast as the host Intel server.

Catapult Version 1 Deployment

Before populating a whole warehouse-scale computer with tens of thousands of servers, Microsoft did a test deployment of 17 full racks, which contained $17 \times 48 \times 2$ or 1632 Intel servers. The Catapult cards and network links were tested at manufacture and system integration, but at deployment, seven of the 1632 cards failed (0.43%), and one of the 3264 FPGA network links (0.03%) was defective. After several months of deployment, nothing else failed.

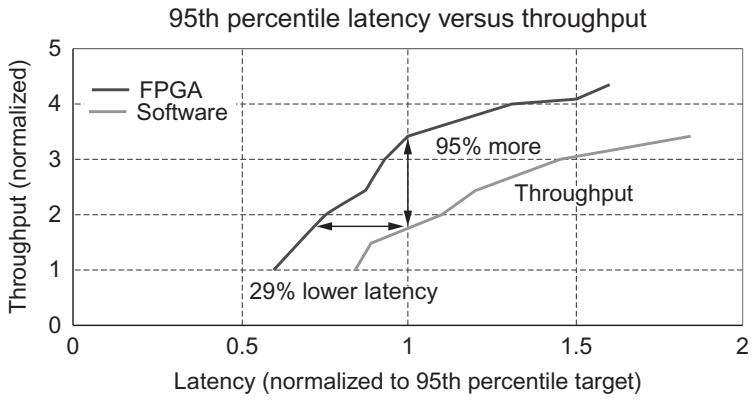


Figure 7.24 Performance for the ranking function on Catapult for a given latency bound. The *x*-axis shows the response time for the Bing ranking function. The maximum response time at the 95th percentile for the Bing application on the *x*-axis is 1.0, so data points to the right may have a higher throughput 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.

Catapult Version 2

Although the test deployment was successful, Microsoft changed the architecture for the real deployment to enable both Bing and Azure Networking to use the same boards and architecture (Caulfield et al., 2016). The main problem with the V1 architecture was that the independent FPGA network did not enable the FPGA to see and process standard Ethernet/IP packets, which prevented it from being used to accelerate the data center network infrastructure. In addition, the cabling was expensive and complicated, it was limited to 48 FPGAs, and the rerouting of traffic during certain failure patterns reduced performance and could isolate nodes.

The solution was to place the FPGA logically between the CPU and NIC, so that all network traffic goes through the FPGA. This "bump-on-a-wire" placement removes many weaknesses of the FPGA network in Catapult V1. Moreover, it enables the FPGAs to run their own low-latency network protocol that allows them to be treated as a global pool of all the FPGAs in the data center and even across data centers.

Three changes occurred between V1 and V2 to overcome the original concerns of Catapult applications interfering with data center network traffic. First, the data center network was upgraded from 10 Gbit/s to 40 Gbit/s, increasing the headroom. Second, Catapult V2 added a rate limiter for FPGA logic, ensuring that an FPGA application could not overwhelm the network. The final and perhaps most

important change was that the networking engineers would now had their own use cases for the FPGA, given its bump-in-the-wire placement. That placement transformed these former interested bystanders into enthusiastic collaborators.

By deploying Catapult V2 in the majority of its new servers, Microsoft essentially has a second supercomputer composed of distributed FPGAs that shares the same network wires as the CPU servers and is at the same scale, as there is one FPGA per server. Figures 7.25 and 7.26 show the block diagram and the board for Catapult V2.

Catapult V2 follows the same shell and role split of the RTL to simplify programming, but at the time of publication, the shell uses almost half of the FPGA resources (44%) because of the more complicated network protocol that shares the data center network wires.

Catapult V2 is used for both Ranking acceleration and function network acceleration. In Ranking acceleration, rather than perform nearly all of the ranking function inside the FPGA, Microsoft implemented only the most compute-intensive portions and left the rest to the host CPU:

- The *feature functional unit (FFU)* is a collection of finite state machines that measure standard features in search, such as counting the frequency of a particular search term. It is similar in concept to the Feature Extraction stage of Catapult V1.

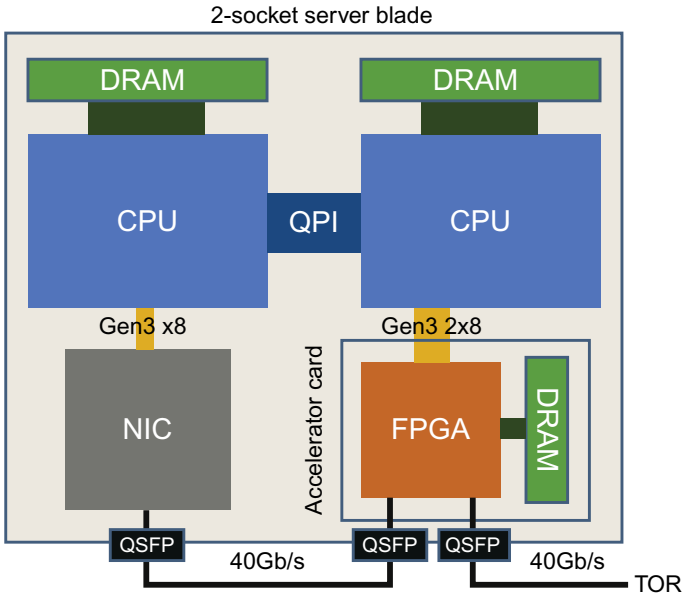


Figure 7.25 The Catapult V2 block diagram. All network traffic is routed through the FPGA to the NIC. There is also a PCIe connector to the CPUs, which allows the FPGA to be used as a local compute accelerator, as in Catapult V1.

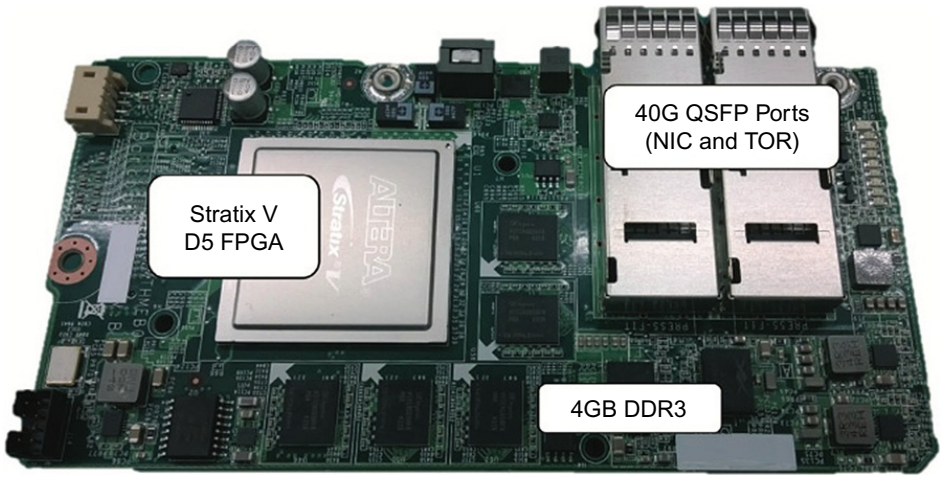


Figure 7.26 The Catapult V2 board uses a PCIe slot. It uses the same FPGA as Catapult V1 and has a TDP of 32 W. A 256-MB Flash chip holds the *golden image* for the FPGA that is loaded at power on, as well as one application image.

- The *dynamic programming feature (DPF)* creates a Microsoft proprietary set of features using dynamic programming and bears some similarity to the Free-Form Expressions stage of Catapult V1.

Both are designed so that they can use non-local FPGAs for these tasks, which simplifies scheduling.

Figure 7.27 shows the performance of Catapult V2 compared to software in a format similar to Figure 7.24. The throughput can now be increased 2.25× without endangering latency, whereas the speedup was previously 1.95×. When ranking was deployed and measured in production, Catapult V2 had better tail latencies than software; that is, the FPGA latencies never exceeded the software latencies at any given demand despite being able to absorb twice the workload.

Summary: How Catapult Follows the Guidelines

Microsoft reported that adding Catapult V1 to the servers in the pilot phase increased the total cost of ownership (TCO) by less than 30%. Thus, for this application, the net gain in cost-performance for Ranking was at least 1.95/1.30, or a return on investment of about 1.5. Although no comment was made about TCO concerning Catapult V2, the board has a similar number of the same type of chips, so we might guess that the TCO is no higher. If so, the cost-performance of Catapult V2 is about 2.25/1.30, or 1.75 for Ranking.

Here is how Catapult followed the guidelines from Section 7.2.

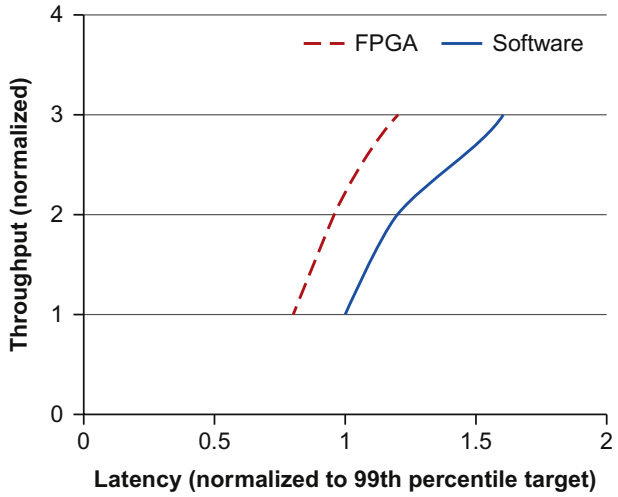


Figure 7.27 Performance for the ranking function on Catapult V2 in the same format as Figure 7.24. Note that this version measures 99th percentile while the earlier figure plots 95th percentile.

1. *Use dedicated memories to minimize the distance over which data is moved.* The Altera V FPGA has 5 MiB of memory on-chip, which an application can customize for its use. For example, for CNNs, it is used for the input and output feature maps of Figure 7.21.
2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.* The Altera V FPGA also has 3926 18-bit ALUs that are tailored to the application. For CNNs, they are used to create the systolic array that drives the Processing Elements in Figure 7.22, and they form the datapaths of the 60-core multiprocessor used by Free Form Expression stage of ranking.
3. *Use the easiest form of parallelism that matches the domain.* Catapult picks the form of parallelism that matches the application. For example, Catapult uses two-dimensional SIMD parallelism for the CNN application and MISD parallelism in the Machine Scoring phase stream Ranking.
4. *Reduce data size and type to the simplest needed for the domain.* Catapult can use whatever size and type of data that the application wants, from an 8-bit integer to a 64-bit floating point.
5. *Use a domain-specific programming language to port code to the DSA.* In this case, programming is done in the hardware register-transfer language (RTL) Verilog, which is an even less productive language than C or C++. Microsoft did not (and possibly could not) follow this guideline given its use of FPGAs.

Although this guideline concerns the one-time porting of an application from software to FPGA, applications are not frozen in time. Almost by definition, software evolves to add features or fix bugs, especially for something as important as web search.

Maintenance of successful programs can be most of software's development costs. Moreover, when programming in an RTL, software maintenance is even more burdensome. The Microsoft developers, like all others who use FPGAs as accelerators, hope that future advances in domain-specific languages and systems for hardware-software co-design will reduce the difficulty of programming FPGAs.

7.6 Intel Crest, a Data Center Accelerator for Training

The quotation by the Intel CEO that opens Section 7.3 came from the press release announcing that Intel was going to start shipping DSAs ("accelerants") for DNN. The first example was Crest, which was announced while we were writing this edition. Despite the limited details, we include it here because of the significance of a traditional microprocessor manufacturer like Intel taking this bold step of embracing DSAs.

Crest is aimed at DNN training. The Intel CEO said the goal is to accelerate DNN training a hundredfold over the next three years. Figure 7.6 shows that training can take a month. There is likely to be a demand to decrease the DNN training to just eight hours, which would be 100 times quicker than the CEO predicted. DNNs will surely become even more complex over the next 3 years, which will require a much greater training effort. Thus there seems little danger that a 100× improvement in training is overkill.

Crest instructions operate on blocks of 32 × 32 matrices. Crest uses a number format called *flex point*, which is a scaled fixed-point representation: 32 × 32 matrices of 16-bit data share a single 5-bit exponent that is provided as part of the instruction set.

Figure 7.28 shows the block diagram of the Lake Crest chip. To compute on these matrices, Crest uses the12 processing clusters of Figure 7.28. Each cluster includes a large SRAM, a big linear algebra processing unit, and a small amount of logic for on- and off-chip routing. The four 8 GiB HBM2 DRAM modules offer 1 TB/s of memory bandwidth, which should lead to an attractive Roofline model for the Crest chip. In addition to high-bandwidth paths to main memory, Lake Crest supports high bandwidth interconnects directly between compute cores inside the processing clusters, which facilitates quick core-to-core communication without passing through shared memory. Lake Crest's goal is a factor of 10 improvement in training over GPUs.

Figure 7.28 shows 12 Inter-Chip Links (ICLs) and 2 Inter-Chip Controllers (ICCs), so Crest is clearly designed to allow many Crest chips to collaborate, similar in spirit to the dedicated network connecting the 48 FPGAs in Catapult. It's likely that the 100× improvement in training will require ganging together several Crest chips.

7.7 Pixel Visual Core, a Personal Mobile Device Image Processing Unit

Pixel Visual Core is a programmable, scalable DSA intended for image processing and computer vision from Google, initially for cell phones and tablets running the

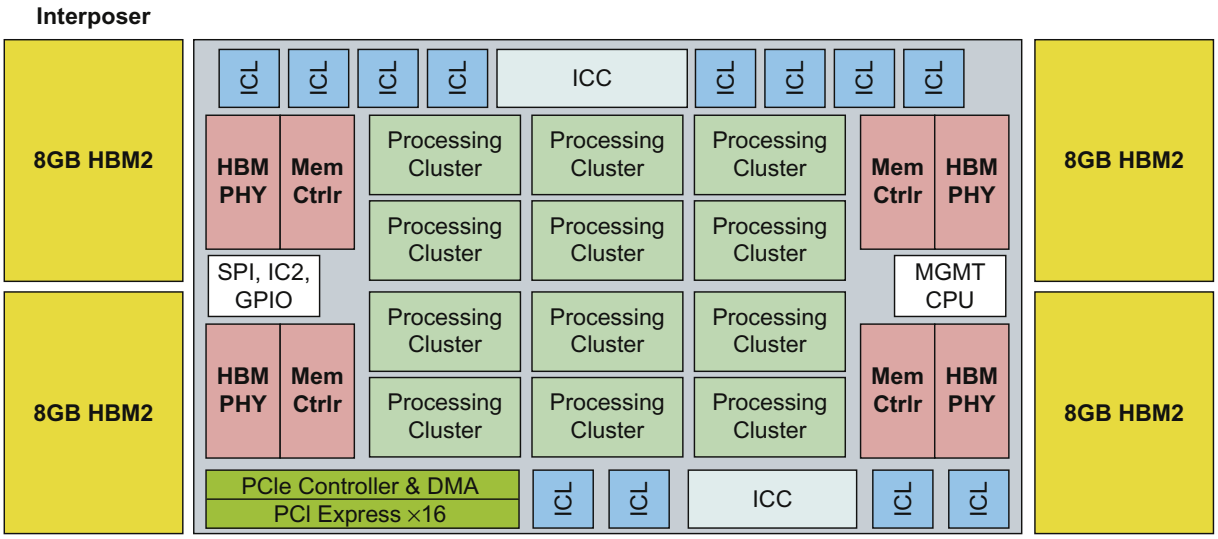


Figure 7.28 Block diagram of the Intel Lake Crest processor. Before being acquired by Intel, Crest said that the chip is almost a full reticle in TSMC 28 nm, which would make the die size 600–700 mm^2. This chip should be available in 2017. Intel is also building Knights Crest, which is a hybrid chip containing Xeon x86 cores and Crest accelerators.

Android operating system, and then potentially for Internet of Things (IoT) devices. It is a multicore design, supporting between 2 and 16 cores to deliver a desired cost-performance. It is designed either to be its own chip or to be part of a *system on a chip* (*SOC*). It has a much smaller area and energy budget than its TPU cousin. Figure 7.29 lists terms and acronyms found in this section.

Pixel Visual Core is an example of a new class of domain specific architectures for vision processing that we call *image processing units* (*IPUs*). IPUs solve the inverse problem of GPUs: they analyze and modify an input image in contrast to generating an output image. We call them IPUs to signal that, as a DSA, they do not need to do everything well because there will also be CPUs (and GPUs) in the system to perform non-input-vision tasks. IPUs rely on stencil computations mentioned above for CNNs.

The innovations of Pixel Visual Core include replacing the one-dimensional SIMD unit of CPUs with a two-dimensional array of processing elements (PEs). They provide a two-dimensional shifting network for the PEs that is aware of the two-dimensional spatial relationship between the elements, and a two-dimensional version of buffers that reduces accesses to off-chip memory. This novel hardware makes it easy to perform stencil computations that are central to both vision processing and CNN algorithms.

ISPs, the Hardwired Predecessors of IPUs

Most portable mobile devices (PMDs) have multiple cameras for input, which has led to hardwired accelerators called *image signal processors* (*ISPs*) for enhancing

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.

input images. The ISP is usually a fixed function ASIC. Virtually every PMD today includes an ISP.

Figure 7.30 shows a typical organization of an image-processing system, including the lens, sensor, ISP, CPU, DRAM, and display. The ISP receives images, removes artifacts in images from the lens and the sensor, interpolates missing colors, and significantly improves the overall visual quality of the image. PMDs tend to have small lens and thus tiny noisy pixels, so this step is critical to producing high-quality photos and videos.

An ISP processes the input image in raster scan order by calculating a series of cascading algorithms via software configurable hardware building blocks, typically organized as a pipeline to minimize memory traffic. At each stage of the pipeline and for each clock cycle, a few pixels are input, and a few are output. Computation is typically performed over small neighborhoods of pixels (*stencils*). Stages are connected by buffers called *line buffers*. The line buffers help

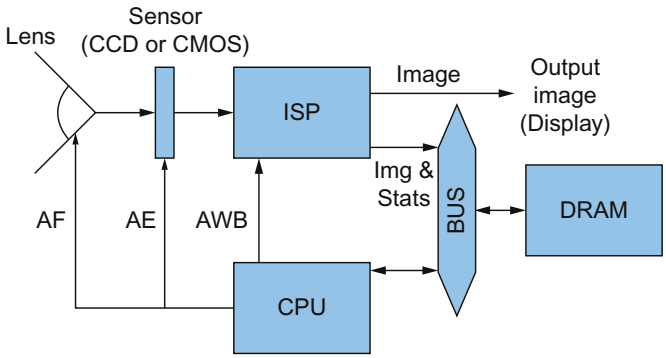


Figure 7.30 Diagram showing interconnection of the Image Signal Processor (ISP), CPU, DRAM, lens, and sensor. The ISP sends statistics to the CPU as well as the improved image either to the display or to DRAM for storage or later processing. The CPU then processes the image statistics and sends information 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*.

keep the processing stages utilized via spatial locality by capturing just enough full lines of an intermediate image to facilitate the computation required by the next stage.

The enhanced image is either sent to a display or to DRAM for storage or for later processing. The ISP also sends statistics about the image (e.g., color and luma histograms, sharpness, and so on) to the CPU, which in turn it processes and sends information to help the system adapt.

Although efficient, ISPs have two major downsides. Given the increasing demand for improved image quality in handheld devices, the first is the inflexibility of an ISP, especially as it takes years to design and manufacture a new ISP within an SOC. The second is that these computing resources can be used only for the image-enhancing function, no matter what is needed at the time on the PMD. Current generation ISPs handle workloads at up to 2 Tera-operations per second on a PMD power budget, so a DSA replacement has to achieve similar performance and efficiency.

Pixel Visual Core Software

Pixel Visual Core generalized the typical hardwired pipeline organization of kernels of an ISP into a *directed acyclic graph* (*DAG*) of kernels. Pixel Visual Core image-processing programs are typically written in Halide, which is a domain-specific functional programming language for image processing. Figure 7.31 is a Halide example that blurs an image. Halide has a functional section to express the function being programmed and a separate schedule section to specify how to optimize that function to the underlying hardware.

```
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.

Pixel Visual Core Architecture Philosophy

The power budget of PMDs is 6–8 W for bursts of 10–20 seconds, dropping down to tens of milliwatts when the screen is off. Given the challenging energy goals of a PMD chip, the Pixel Visual Core architecture was strongly shaped by the relative energy costs for the primitive operations mentioned in Chapter 1 and made explicit in Figure 7.32. Strikingly, a single 8-bit DRAM access takes as much energy as 12,500 8-bit additions or 7–100 8-bit SRAM accesses, depending on the organization of the SRAM. The 22× to 150× higher cost of IEEE 754 floating-point operations over 8-bit integer operations, plus the die size and energy benefits of storing narrower data, strongly favor using narrow integers whenever algorithms can accommodate them.

In addition to the guidelines from Section 7.2, these observations led to other themes that guided the Pixel Visual Core design:

- *Two-dimensional is better than one-dimensional*: Two-dimensional organizations can be beneficial for processing images as it minimizes communication distance and because the two- and three-dimensional nature of image data can utilize such organizations.
- *Closer is better than farther*: Moving data is expensive. Moreover, the relative cost of moving data to an ALU operation is increasing. And of course DRAM time and energy costs far exceed any local data storage or movement.

A primary goal in going from an ISP to an IPU is to get more reuse of the hardware via programmability. Here are the three main features of the Pixel Visual Core:

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.

1. Following the theme that two-dimensional is better than one-dimensional, Pixel Visual Core uses a two-dimensional SIMD architecture instead of one-dimensional SIMD architecture. Thus it has a two-dimensional array of independent *processing element*s (*PEs*), each of which contains 2 16-bit ALUs, 1 16-bit MAC unit, 10 16-bit registers, and 10 1-bit predicate registers. The 16-bit arithmetic follows the guideline of providing only the precision needed by the domain.
2. Pixel Visual Core needs temporary storage at each PE. Following the guideline from Section 7.2 of avoiding caches, this PE memory is a compiler-managed scratchpad memory. The logical size of each PE memory is 128 entries of 16 bits, or just 256 bytes. Because it would be inefficient to implement a separate small SRAM in each PE, Pixel Visual Core instead groups the PE memory of 8 PEs together in a single wide SRAM block. Because the PEs operate in SIMD fashion, Pixel Visual Core can bind all the individual reads and writes together to form a "squarer" SRAM, which is more efficient than narrow and deep or wide and shallow SRAMs. Figure 7.33 shows four PEs.
3. To be able to perform simultaneous stencil computations in all PEs, Pixel Visual Core needs to collect inputs from nearest neighbors. This communication pattern requires a "NSEW" (North, South, East, West) shift network: it can shift data en masse between the PEs in any compass direction. So that it doesn't lose pixels along the edges as it shifts images, Pixel Visual Core connects the endpoints of the network together to form a torus.

Note that the shift network is in contrast with the systolic array of processing element arrays in the TPU and Catapult. In this case, software explicitly moves the data in the desired direction across the array, whereas the systolic approach is a hardware-controlled, two-dimensional pipeline that moves data as a wavefront that is invisible to the software.

The Pixel Visual Core Halo

A 3×3, 5×5, or 7×7 stencil is going to get inputs from 1, 2, or 3 external pixels at the edges of the two-dimensional subset being computed (half of the dimension of the stencil minus one-half). That leaves two choices. Either Pixel Visual Core

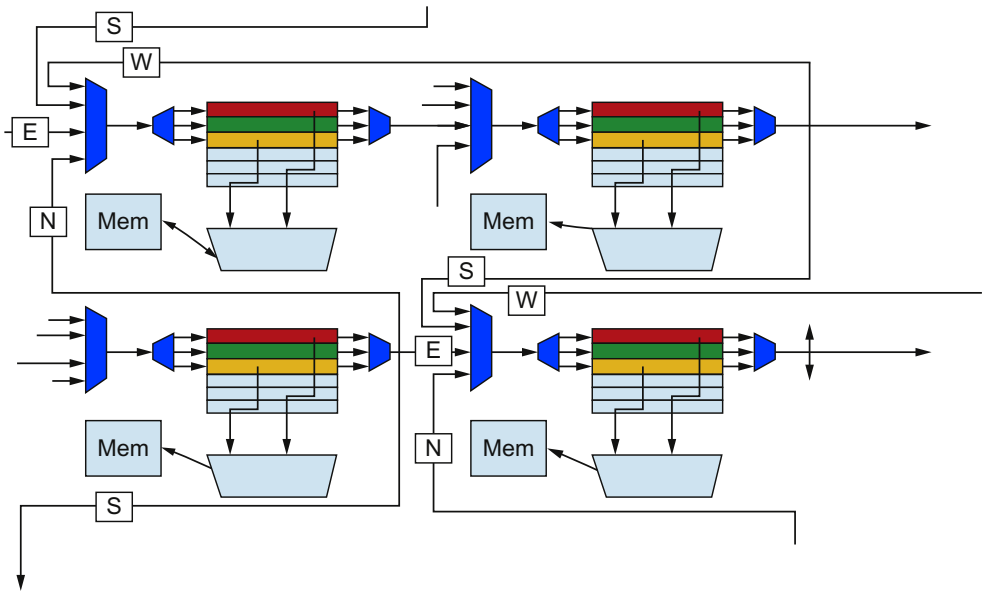


Figure 7.33 The two-dimensional SIMD includes two-dimensional shifting "N," "S," "E," "W," show the direction of the shift (North, South, East, West). Each PE has a software-controlled scratchpad memory.

under utilizes the hardware in the elements near the border, because they only pass input values, or Pixel Visual Core slightly extends the two-dimensional PEs with simplified PEs that leave out the ALUs. Because the difference in size between a standard PE and a simplified PE is about 2.2 ×, Pixel Visual Core has an extended array. This extended region is called the *halo*. Figure 7.34 shows two rows of a halo surrounding an 8x8 PE array and illustrates how an example 5 × 5 stencil computation in the upper-left corner relies on the halo.

A Processor of the Pixel Visual Core

The collection of 16 × 16 PEs and 4 halo lanes in each dimension, called the *PE array* or *vector array*, is the main computation unit of the Pixel Visual Core. It also has a load-store unit called a *Sheet Generator* (*SHG*). SHG refers to memory accesses of blocks of 1 × 1 to 256 × 256 pixels, which are called *sheets*. This happens during downsampling, and typical values are 16 × 16 or 20 × 20.

An implementation of Pixel Visual Core can have any even number of 2 or more cores, depending on the resources available. Thus it needs a network to connect them together, so every core also has an interface to the Network on Chip (NOC). A typical NOC implementation for Pixel Visual Core will not be an expensive cross switch, however, because those require data to travel a long distance, which is expensive. Leveraging the pipeline nature of the application, the NOC typically needs to communicate only to neighboring cores. It is implemented as a two-dimensional mesh, which allows power gating of pairs of cores under software control.

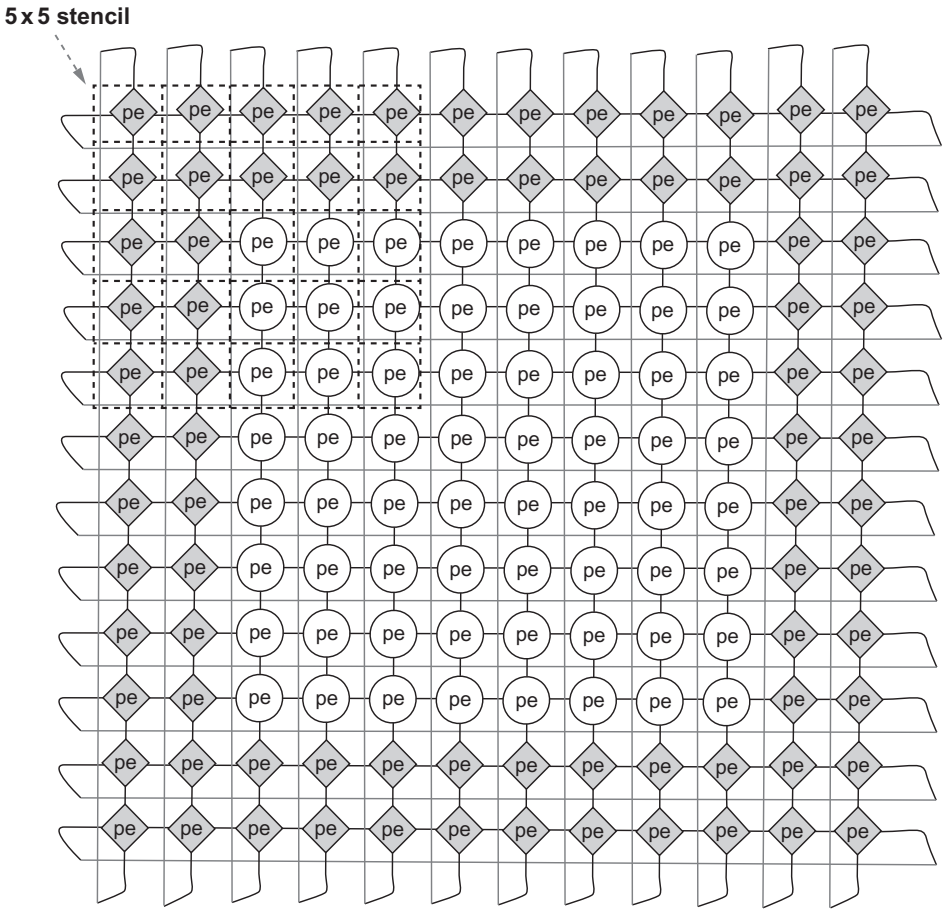


Figure 7.34 The two-dimensional array of full processing elements *(shown as unshaded circles)* surrounded by two layers of simplified processing elements *(shaded diamonds)* called a *halo*. In this figure, there are 8×8 or 64 full PEs with 80 simplified PEs in the halo. (Pixel Visual Core actually 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.

Finally, the Pixel Visual Core also includes a scalar processor that is called a *scalar lane* (*SCL*). It is identical to 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. Note that Pixel Visual Core has a single instruction stream that controls the scalar and vector units, similar to how a CPU core has a single instruction stream for its scalar and SIMD units.

In addition to cores, there is also a DMA engine to transfer data between DRAM and the line buffers while efficiently converting between image memory layout formats (e.g., packing/unpacking). As well as sequential DRAM accesses, the DMA engines perform vector-like gather reads of DRAM as well as sequential and strided reads and writes.

Pixel Visual Core Instruction Set Architecture

Like GPUs, Pixel Visual Core uses a two-step compilation process. The first step is compiling the programs from the target language (e.g., Halide) into *vISA* instructions. The Pixel Visual Core *vISA* (*virtual Instruction Set Architecture*) is inspired in part by the RISC-V instruction set, but it uses an image-specific memory model and extends the instruction set to handle image processing, and in particular, the two-dimensional notion of images. In vISA, the two-dimensional array of a core is infinite, the number of register is unbounded, and memory size is similarly unlimited. vISA instructions contain pure functions that don't directly access DRAM (see Figure 7.36), which greatly simplifies mapping them onto the hardware.

The next step is to compile the vISA program into a *pISA* (*physical Instruction Set Architecture*) program. Using vISA as the target of compilers allows the processor to be software-compatible with past programs and yet accept changes to the pISA instruction set, so vISA plays the same role that PTX does for GPUs (see Chapter 4).

Lowering from vISA to pISA takes two steps: compilation and mapping with early-bound parameters, and then patching the code with late-bound parameters. The parameters that must be bound include STP size, halo size, number of STPs, mapping of line buffers, mapping of kernels to processors, as well as register and local memory allocations.

Figure 7.35 shows that pISA is a very long instruction word (VLIW) instruction set with 119-bit-wide instructions. The first 43-bit field is for the Scalar Lane, the next 38-bit field specifies the computation by the two-dimensional PE array, and the third 12-bit field specifies the memory accesses by the two-dimensional PE array. The last two fields are immediates for computation or addressing. The operations for all the VLIW fields are what you'd expect: two's complement integer arithmetic, saturating integer arithmetic, logical operations, shifts, data transfers, and a few special ones like divide iteration and count leading zeros. The Scalar Lane supports a superset of the operations in the two-dimensional PE array, plus it adds instructions for control-flow and sheet-generator control. The 1-bit Predicate registers mentioned above enables conditional moves to registers (e.g., $A = B$ if C).

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

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

Although the pISA VLIW instruction is very wide, Halide kernels are short, often just 200–600 instructions. Recall that as an IPU, it only needs to execute the compute-intensive portion of an application, leaving the rest of the functionality to CPUs and GPUs. Thus the instruction memory of a Pixel Visual Core holds just 2048 pISA instructions (28.5 KiB).

The Scalar Lane issues sheet generator instructions that access line buffers. Unlike other memory accesses within Pixel Visual Core, the latency can be more than 1 clock cycle, so they have a DMA-like interface. The lane first sets up the addresses and transfer size in special registers.

Pixel Visual Core Example

Figure 7.36 shows the vISA code that is output from the Halide compiler for the blur example in Figure 7.31, with comments added for clarity. It calculates a blur first in the *x* direction and then in the y direction using 16-bit arithmetic. The vISA code matches the functional part of the Halide program. This code can be thought of as executing across all the pixels of an image.

Pixel Visual Core Processing Element

One of the architectural decisions was how big to build the halo. Pixel Visual Core uses 16×16 PEs, and it adds a halo of 2 extra elements, so it can support 5×5

```
// 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.

stencils directly. Note that the bigger the array of PEs, the less the halo overhead to support a given stencil size.

For Pixel Visual Core, the smaller size of the halo PEs and the 16×16 arrays means it only costs 20% more area for the halo. For a 5×5 stencil, Pixel Visual Core can calculate 1.8 times as many results per clock cycle ($16^2/12^2$), and the ratio is 1.3 for a 3×3 stencil ($16^2/14^2$).

The design of the arithmetic unit of the PE is driven by multiply-accumulate (MAC), which is a primitive of stencil computation. Pixel Visual Core native MACs are 16-bits wide for the multiplies, but they can accumulate at a 32-bit width. Pipelining MAC would use energy unnecessarily because of the reading and writing of the added pipeline register. Thus the multiply-add hardware determines the clock cycle. The other operations, previously mentioned, are the traditional logical and arithmetic operations along with saturating versions of the arithmetic operations and a few specialized instructions.

The PE has two 16-bit ALUs that can operate in a variety of ways within a single clock cycle:

- Independently, producing two 16-bit results: A op B, C op D.
- Fused, producing just one 16-bit result: A op (C op D).
- Joined, producing one 32-bit result: A:C op B:D.

Two-Dimensional Line Buffers and Their Controller

Because DRAM accesses use so much energy (see Figure 7.32), the Pixel Visual Core memory system was carefully designed to minimize the number of DRAM accesses. The key innovation is the *two-dimensional line buffer*.

Kernels are logically running on separate cores, and they are connected in a DAG with input from the sensor or DRAM and output to DRAM. The line buffers hold portions of the image being calculated between kernels. Figure 7.37 shows the logical use of line buffers in Pixel Visual Core.

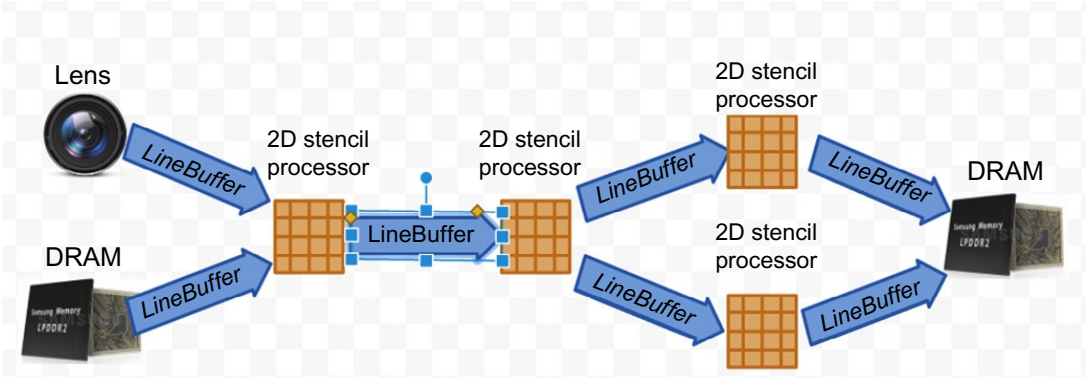


Figure 7.37 Programmer view of Pixel Visual Core: a directed-acyclic graph of kernels.

Here are four features that the two-dimensional line buffer must support:

1. It must support two-dimensional stencil computations of various sizes, which are unknown at design time.
2. Because of the halo, for the 16×16 PE array in Pixel Visual Core, the STPs will want to read 20×20 blocks of pixels from the line buffer and write 16×16 blocks of pixels to the line buffer. (As previously mentioned, they call these blocks of pixels *sheets*.)
3. Because the DAG is programmable, we need line buffers that can be allocated by software between any two cores.
4. Several cores may need to read data from the same line buffer. Thus a line buffer should support multiple consumers, although it needs just one producer.

Line buffers in Pixel Visual Core are really a multi-reader, two-dimensional FIFO abstraction on top of a relatively large amount of SRAM: 128 KiB per instance. It contains temporary "images" that are used just once, so a small, dedicated local FIFO is much more efficient than a cache for data in distant memory.

To accommodate the size mismatch between reading 20×20 blocks of pixels and writing 16×16 blocks, the fundamental unit of allocation in the FIFO is a group of 4×4 pixels. Per stencil processor, there is one *Line Buffer Pool* (*LBP*) that can have eight logical line buffers (*LB*), plus one LBP for DMA of I/O. The LBP has three levels of abstraction:

1. At the top, the LBP controller supports eight LBs as logical instances. Each LB has one FIFO producer and up to eight FIFO consumers per LB.
2. The controller keeps track of a set of head and tail pointers for each FIFO. Note that the sizes of the line buffers inside the LBP are flexible and up to the controller.
3. At the bottom are many physical memory banks to support the bandwidth requirements. Pixel Visual Core has eight physical memory banks, each having a 128-bit interface and 16 KiB of capacity.

The controller for the LBP is challenging because it must fulfill the bandwidth demands of the STPs and I/O DMAs as well as schedule all their reads and writes to the banks of physical SRAM memory. The LBP controller is one of the most complicated pieces of Pixel Visual Core.

Pixel Visual Core Implementation

The first implementation of Pixel Visual Core was as a separate chip. Figure 7.38 shows the floorplan of the chip, which has 8 cores. It was fabricated in a TSMC 28 nm technology in 2016. The chip dimensions are 6×7.2 mm, it runs at 426 MHz, it is stacked with 512 MB DRAM as Silicon in Package, and consumes

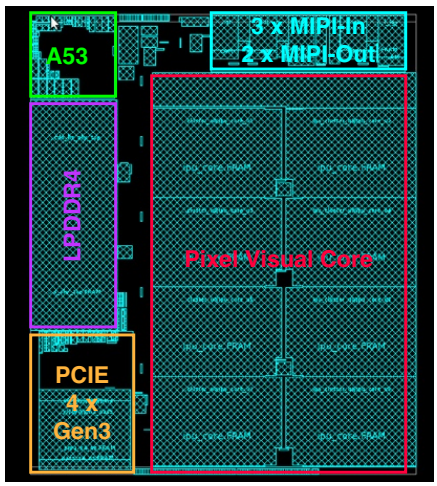


Figure 7.38 Floor plan of the 8-core Pixel Visual Core chip. A53 is an ARMv7 core. LPDDR4 is a DRAM controller. PCIE and MIPI are I/O buses.

(including the DRAM) 187–4500 mW depending on the workload. About 30% of the power for the chip is for an ARMv7 A53 core for control, the MIPI, the PCIe, the PCIe, and the LPDDR interfaces, interface is just over half this die at 23 mm^2. Power for Pixel Visual Core running a worst case "power virus" can go as high as 3200 mW. Figure 7.39 shows the floor plan of a core.

Summary: How Pixel Visual Core Follows the Guidelines

Pixel Visual Core is a multicore DSA for image and vision processing intended as a stand-alone chip or as an IP block for mobile device SOCs. As we will see in Section 7.9, its performance per watt for CNNs are factors of 25–100 better than CPUs and GPUs. Here is how the Pixel Visual core followed the guidelines in Section 7.2.

1. *Use dedicated memories to minimize the distance over which data is moved.* Perhaps the most distinguishing architecture feature of Pixel Visual Core is the software-controlled, two-dimensional line buffers. At 128 KiB per core, they are a significant fraction of the area. Each core also has 64 KiB of software-controlled PE memory for temporary storage.
2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.* Two other key features of Pixel Visual Core are a 16×16 two-dimensional array of processing elements per core and a two-dimensional shifting network between the processing elements. It offers a halo region that acts as a buffer to allow full utilization of its 256 arithmetic units.

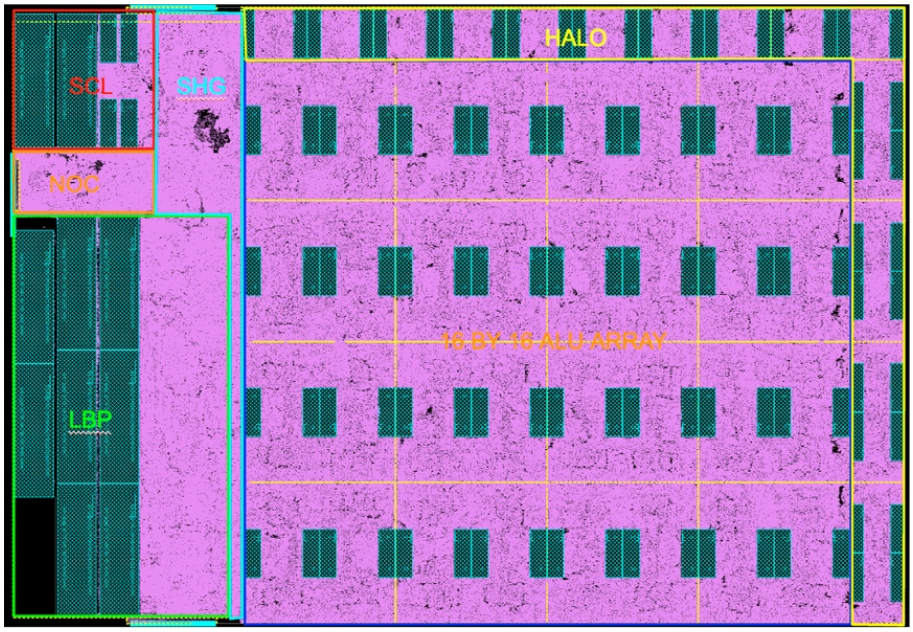


Figure 7.39 Floor plan of a Pixel Visual Core. From left to right, and top down: the scalar lane (SCL) is 4% of the core area, NOC is 2%, the line buffer pool (LBP) is 15%, the sheet generator (SHG) is 5%, the halo is 11%, and the processing element array is 62%. The torus connection of the halo makes 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.

3. *Use the easiest form of parallelism that matches the domain.*
   Pixel Visual Core relies on two-dimensional SIMD parallelism using its PE array, VLIW to express instruction-level parallelism, and multiple program multiple data (MPMD) parallelism to utilize multiple cores.
4. *Reduce data size and type to the simplest needed for the domain.*
   Pixel Visual Core relies primarily on 8-bit and 16-bit integers, but it also works with 32-bit integers, albeit more slowly.
5. *Use a domain-specific programming language to port code to the DSA.*
   Pixel Visual Core is programmed in the domain-specific language Halide for image processing and in TensorFlow for CNNs.

7.8 Cross-Cutting Issues

Heterogeneity and System on a Chip (SOC)

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 is 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*), which is one of the goals of Pixel Visual Core and eventually the Intel Crest.

Such a design is called an *IP block*, 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 7.40 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 CPU and GPU get 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 differ 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. The Pixel Visual Core approach is a multicore design, allowing the SOC engineer to choose between 2 and 16 cores to match the area and power budget and desired performance.

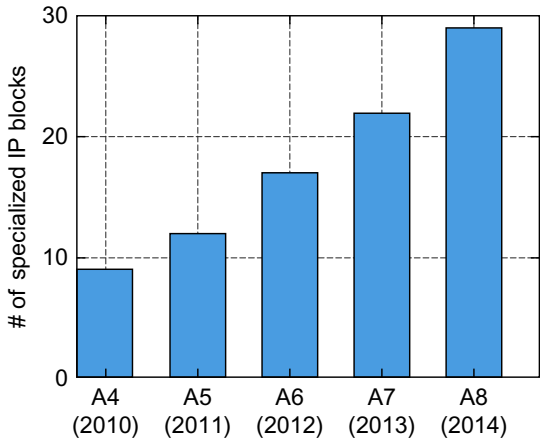


Figure 7.40 Number of IP blocks in Apple SOCs for the iPhone and iPad between 2010 and 2014 (Shao and Brooks, 2015).

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 designing their own accelerators and include IP CPUs in their ASICs.

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 choice: 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. 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 case 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 DAGs or parallel patterns, as is the case for Halide 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, 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: CPUs Versus GPUs Versus 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 the TPU to standard CPUs and GPUs and then add brief comparisons to Catapult and Pixel Visual Core.

Figure 7.41 shows the six benchmarks we use in this comparison. They consist of two examples of each of the three types of DNNs in Section 7.3. These six benchmarks represent 95% of TPU inference workload in Google data centers in 2016. Typically written in TensorFlow, they are surprisingly short: just 100–1500 lines of code. They are small pieces of larger applications that run on the host server, which can be thousands to millions of lines of C++ code. The applications are typically user-facing, which leads to rigid response-time limits, as we will see.

Figures 7.42 and 7.43 show the chips and servers being compared. They are server-class computers deployed in Google data centers at the same time that TPUs were deployed. To be deployed in Google data centers, they must at least check for internal memory errors, which excluded some choices, such as the Nvidia Maxwell GPU. For Google to purchase and deploy them, the machines had to be sensibly configured, and not awkward artifacts assembled solely to win benchmarks.

The traditional CPU server is represented by an 18-core, dual-socket Haswell processor from Intel. This platform is also the host server for GPUs or TPUs.

		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).

[2]This section is also largely based upon the paper "In-Datacenter Performance Analysis of a Tensor Processing Unit" Jouppi et al., 2017, of which one of your book authors was a coauthor.

					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.

				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.

Haswell is fabricated in an Intel 22-nm process. Both the CPU and GPU are very large dies: about 600 mm^2!

The GPU accelerator is the Nvidia K80. Each K80 card contains two dies and offers SECDED on internal memory and DRAM. Nvidia states that (Nvidia, 2016)

> the K80 Accelerator dramatically lowers datacenter cost by delivering application performance with fewer, more powerful servers.

DNN researchers frequently used K80s in 2015, which is when they were deployed at Google. Note that K80s were also chosen for new cloud-based GPUs by Amazon Web Services and by Microsoft Azure in late 2016.

Because the number of dies per benchmarked server varies between 2 and 8, the following figures show results normalized per die, except for Figure 7.50, which compares the performance/watt of whole servers.

Performance: Rooflines, Response Time, and Throughput

To illustrate the performance of the six benchmarks on the three processors, we adapt the Roofline performance model in Chapter 4. To use the Roofline model for the TPU, when DNN applications are quantized, we first replace floating-point operations with integer multiply-accumulate operations. As weights do not normally fit in on-chip memory for DNN applications, the second change is to redefine operational intensity to be integer operations per byte of weights read (Figure 7.41).

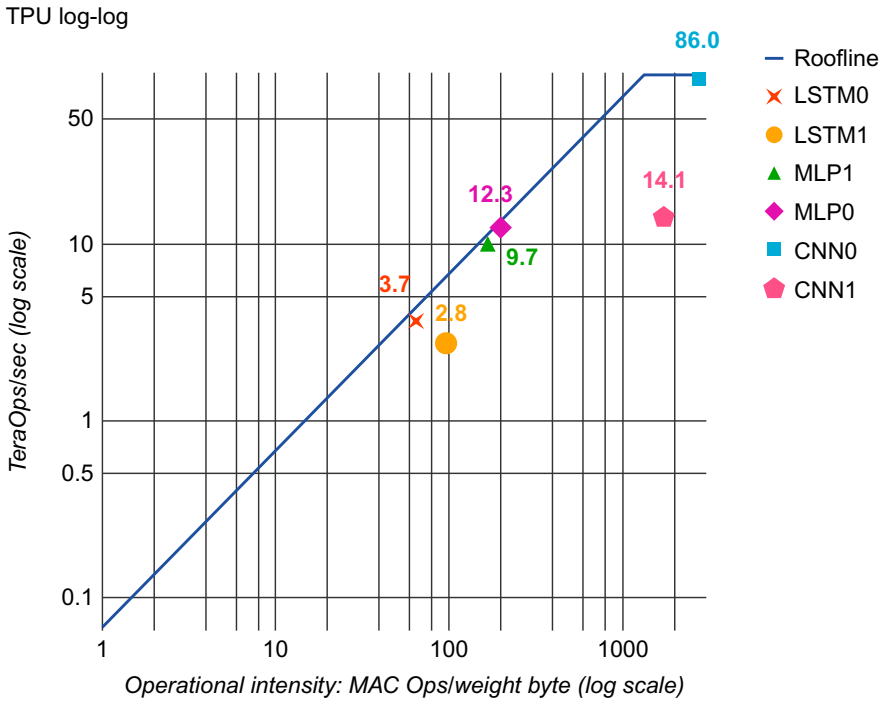


Figure 7.44 TPU Roofline. Its ridge point is far to the right at 1350 multiply-accumulate operations per byte of weight memory. CNN1 is much further below its Roofline than the other DNNs because it spends about a third of the time waiting for weights to be loaded into the matrix unit and because the 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).

Figure 7.44 shows the Roofline model for a single TPU on log-log scales. The TPU has a long "slanted" part of its Roofline, where operational intensity means that performance is limited by memory bandwidth rather than by peak compute. Five of the six applications are happily bumping their heads against the ceiling: the MLPs and LSTMs are memory-bound, and the CNNs are computation-bound. The single DNN that is not bumping its head against the ceiling is CNN1. Despite CNNs having very high operational intensity, CNN1 is running at only 14.1 Tera Operations Per Second (TOPS), while CNN0 runs at a satisfying 86 TOPS.

For readers interested into a deep dive into what happened with CNN1, Figure 7.45 uses performance counters to give partial visibility into the utilization of the TPU. The TPU spends less than half of its cycles performing matrix operations for CNN1 (column 7, row 1). On each of those active cycles, only about half of the 65,536 MACs hold useful weights because some layers in CNN1 have shallow feature depths. About 35% of cycles are spent waiting for weights to load from memory into the matrix unit, which occurs during the four fully connected layers that run at an operational intensity of just 32. This leaves roughly 19% of cycles not

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.

explained by the matrix-related counters. Because of overlapped execution on the TPU, we do not have exact accounting for those cycles, but we can see that 23% of cycles have stalls for RAW dependences in the pipeline and that 1% are spent stalled for input over the PCIe bus.

Figures 7.46 and 7.47 show Rooflines for Haswell and the K80. The six NN applications are generally further below their ceilings than the TPU in Figure 7.44. Response-time limits are the reason. Many of these DNN applications are parts of services that are part of end-user-facing services. Researchers have demonstrated that small increases in response time cause customers to use a service less (see Chapter 6). Thus, although training may not have hard response-time deadlines, inference usually does. That is, inference cares about throughput only while it is maintaining the latency bound.

Figure 7.48 illustrates the impact of the 99th percentile response-time limit of 7 ms for MLP0 on Haswell and the K80, which was required by the application developer. (The inferences per second and 7-ms latency include the server host time as well as the accelerator time.) They can operate at 42% and 37%, respectively, with the highest throughput achievable for MLP0, if the response-time limit is relaxed. Thus, although CPUs and GPUs have potentially much higher throughput, it's wasted if they don't meet the response-time limit. These bounds affect the TPU as well, but at 80% in Figure 7.48, it is operating much closer to its highest MLP0 throughput. As compared with CPUs and GPUs, the single-threaded TPU has none of the sophisticated microarchitectural features discussed in Section 7.1 that consume transistors and energy to improve the average case but not the 99th-percentile case.

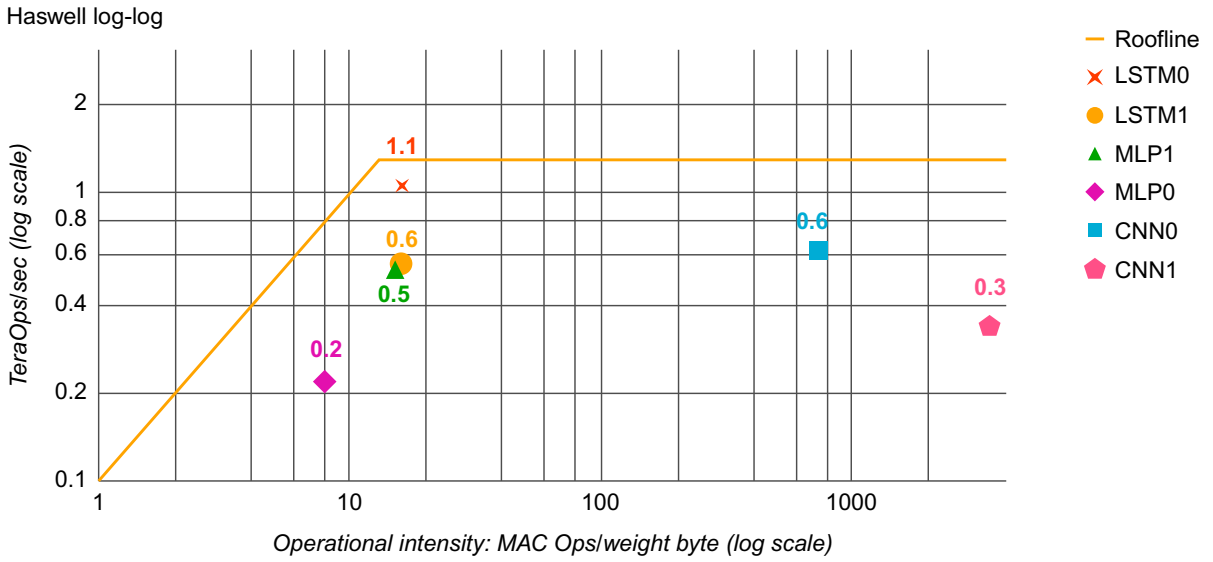


Figure 7.46 Intel Haswell CPU Roofline with its ridge point at 13 multiply-accumulate operations/byte, which is much further to the left than in Figure 7.44.

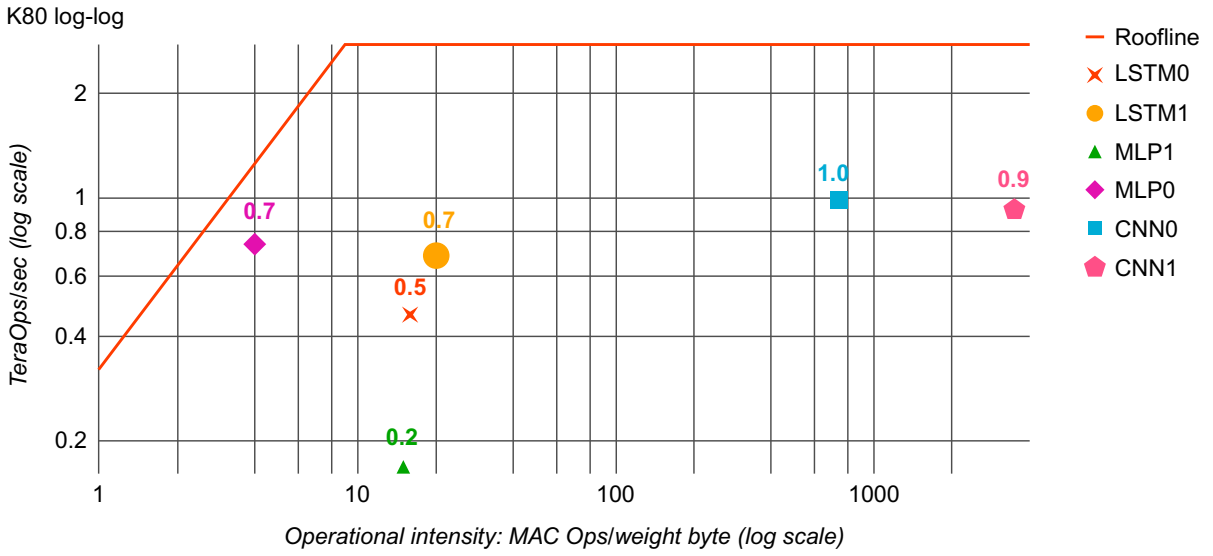


Figure 7.47 NVIDIA K80 GPU die Roofline. The much higher memory bandwidth moves the ridge point to 9 multiply-accumulate operations per weight byte, which is even further to the left than in Figure 7.46.

Figure 7.49 gives the bottom line of relative inference performance per die, including the host server overhead for the two accelerators. Recall that architects use the geometric mean when they don't know the actual mix of programs that will be run. For this comparison, however, we *do* know the mix (Figure 7.41). The

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.

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.

weighted mean in the last column of Figure 7.49 using the actual mix makes the GPU up to 1.9 times, and the TPU is 29.2 times as fast as the CPU, so the TPU is 15.3 times as fast as the GPU.

Cost-Performance, TCO, and Performance/Watt

When buying computers by the thousands, cost-performance trumps general performance. The best cost metric in a data center is total cost of ownership (TCO). The actual price Google pays for thousands of chips depends on negotiations between the companies involved. For business reasons, Google can't publish such price information or data that might let them be deduced. However, power is correlated with TCO, and Google can publish watts per server, so we use performance/watt as our proxy for performance/TCO. In this section, we compare servers (Figure 7.43) rather than single dies (Figure 7.42).

Figure 7.50 shows the weighted mean performance/watt for the K80 GPU and TPU relative to the Haswell CPU. We present two different calculations of performance/watt. The first ("total") includes the power consumed by the host CPU server when calculating performance/watt for the GPU and TPU. The second ("incremental") subtracts the host CPU server power from the total for the GPU and TPU beforehand.

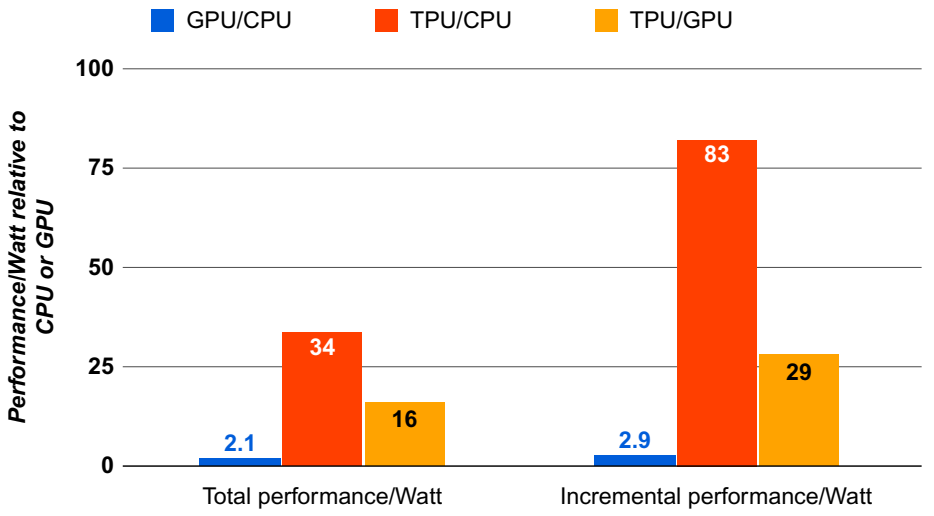


Figure 7.50 Relative performance/watt of GPU and TPU servers to CPU or GPU servers. Total performance/watt includes host server power, but incremental doesn't. It is a widely quoted metric, but we use it as a proxy for performance/TCO in the data center.

For total-performance/watt, the K80 server is 2.1 × Haswell. For incremental-performance/watt, when Haswell power is omitted, the K80 server is 2.9 ×.

The TPU server has 34 times better total-performance/watt than Haswell, which makes the TPU server 16 times the performance/watt of the K80 server. The relative incremental-performance/watt—which was Google's justification for a custom ASIC—is 83 for the TPU, which lifts the TPU to 29 times the performance/watt of the GPU.

Evaluating Catapult and Pixel Visual Core

Catapult V1 runs CNNs 2.3 × as fast as a 2.1 GHz, 16-core, dual-socket server (Ovtcharov et al., 2015a). Using the next generation of FPGAs (14-nm Arria 10), performance goes up 7 ×, and perhaps even 17 × with more careful floorplanning and scaling up of the Processing Elements (Ovtcharov et al., 2015b). In both cases, Catapult increases power by less than 1.2 ×. Although it's apples versus oranges, the TPU runs its CNNs 40 × to 70 × versus a somewhat faster server (see Figures 7.42, 7.43, and 7.49).

Because Pixel Visual Core and the TPU are both made by Google, the good news is that we can directly compare performance for CNN1, which is a common DNN, although it had to be translated from TensorFlow. It runs with batch size of 1 instead of 32 as in the TPU. The TPU runs CNN1 about 50 times as fast as Pixel Visual Core, which makes Pixel Visual Core about half as fast as the GPU and a little faster than Haswell. Incremental performance/watt for CNN1 raises Pixel Visual Core to about half the TPU, 25 times the GPU, and 100 times the CPU.

Because the Intel Crest is designed for training rather than inference, it wouldn't be fair to include it in this section, even if it were available to measure.

7.10 Fallacies and Pitfalls

In these early days of both DSAs and DNNs, fallacies abound.

Fallacy *It costs $100 million to design a custom chip.*

Figure 7.51 shows a graph from an article that debunks the widely quoted $100-million myth that it was "only" $50 million, with most of the cost being salaries (Olofsson, 2011). Note that the author's estimate is for sophisticated processors that include features that DSAs by definition omit, so even if there were no improvement to the development process, you would expect the cost of a DSA design to be less.

Why are we more optimistic six years later, when, if anything, mask costs are even higher for the smaller process technologies?

First, software is the largest category, at almost a third of the cost. The availability of applications written in domain-specific languages allows the compilers to do most of the work of porting the applications to your DSA, as we saw for the TPU and Pixel Visual Core. The open RISC-V instruction set will also help reduce the cost of getting system software as well as cut the large IP costs.

Mask and fabrication costs can be saved by having multiple projects share a single reticle. As long as you have a small chip, amazingly enough, for $30,000 anyone can get 100 untested parts in 28-nm TSMC technology (Patterson and Nikolić, 2015).

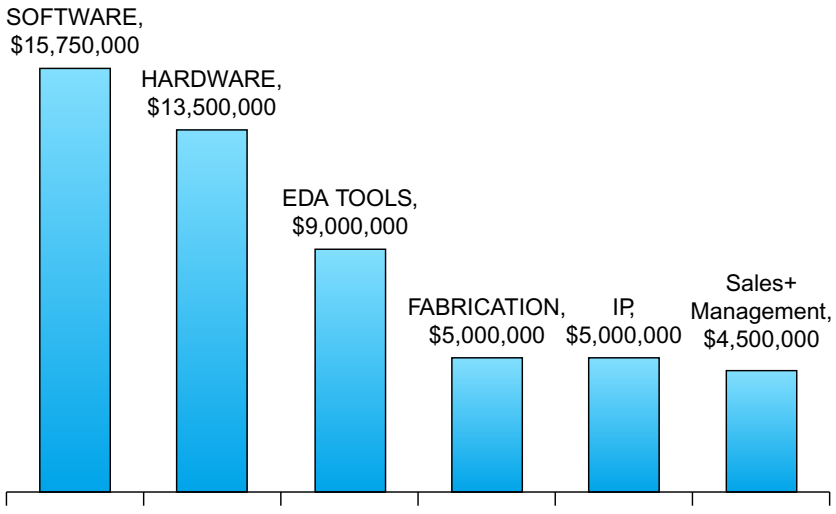


Figure 7.51 The breakdown of the $50 million cost of a custom ASIC that came from surveying others (Olofsson, 2011). The author wrote that his company spent just $2 million for its ASIC.

Perhaps the biggest change is to hardware engineering, which is more than a quarter of the cost. Hardware engineers have begun to follow their software colleagues to use agile development. The traditional hardware process not only has separate phases for design requirements, architecture, logical design, layout, verification, and so on, but also it uses different job titles for the people who perform each of the phases. This process is heavy on planning, documentation, and scheduling in part because of the change in personnel each phase.

Software used to follow this "waterfall" model as well, but projects were so commonly late, over budget, and even canceled that it led to a radically different approach. The Agile Manifesto in 2001 basically said that it was much more likely that a small team that iterated on an incomplete but working prototype shown regularly to customers would produce useful software on schedule and on budget more than the traditional plan-and-document approach of the waterfall process would.

Small hardware teams now do agile iterations (Lee et al., 2016). To ameliorate the long latency of a chip fabrication, engineers do some iterations using FPGAs because modern design systems can produce both the EDIF for FPGAs and chip layout from a single design. FPGA prototypes run 10–20 times slower than chips, but that is still much faster than simulators. They also do "tape-in" iterations, where you do all the work of a tape-out for your working but incomplete prototype, but you don't pay the costs of fabricating a chip.

In addition to an improved development process, more modern hardware design languages to support them (Bachrach et al., 2012), and advances in automatic generation of hardware from high-level domain-specific languages (Canis et al., 2013; Huang et al., 2016; Prabhakar et al., 2016). Open source cores that you can download for free and modify should also lower the cost of hardware design.

Pitfall *Performance counters added as an afterthought for DSA hardware.*

The TPU has 106 performance counters, and the designers wanted even more (see Figure 7.45). The *raison d'être* for *DSAs* is performance, and it is way too early in their evolution to have a good idea about what is going on.

Fallacy *Architects are tackling the right DNN tasks.*

The architecture community is paying attention to deep learning: 15% of the papers at ISCA 2016 were on hardware accelerators for DNNs! Alas, all nine papers looked at CNNs, and only two mentioned other DNNs. CNNs are more complex than MLPs and are prominent in DNN competitions (Russakovsky et al., 2015), which might explain their allure, but they are only about 5% of the Google data center NN workload. It seems wise try to accelerate MLPs and LSTMs with at least as much gusto.

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 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 NN layers). For example, the

TPU runs 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 much more misleading for NN accelerators than MIPS or FLOPS is for traditional processors, so IPS should be even more disparaged. To compare DNN machines better, we need a benchmark suite written at a high level to port it to the wide variety of DNN architectures. Fathom is a promising new attempt at such a benchmark suite (Adolf et al., 2016).

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 the TPU, three important architectural features date back to the early 1980s: systolic arrays (Kung and Leiserson, 1980), decoupled-access/execute (Smith, 1982b), and CISC instructions (Patterson and Ditzel, 1980). The first reduced the area and power of the large Matrix Multiply Unit, the second fetched weights concurrently during operation of the Matrix Multiply Unit, and the third better utilized the limited bandwidth of the PCIe bus for delivering instructions. We advise mining the historical perspectives sections at the end of every chapter of this book to discover jewels that could embellish DSAs that you design.

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.

Both versions of Catapult preserved data-center homogeneity by designing a small, low-power FPGA board that could fit inside a server. The hope is that the flexibility of FPGAs will allow Catapult to be useful to many current applications and the new ones that appeared after deployment. Catapult runs search rank and CNNs faster than GPUs, offering a 1.5–1.75 gain in performance/TCO for ranking over CPUs.

The TPU project actually began with FPGAs but abandoned them when the designers concluded that the FPGAs of that time were not competitive in performance compared to the GPUs. They also believed the TPU would use much less power than GPUs, while being as fast or faster, potentially making the TPU much better than FPGAs and GPUs. Finally, the TPU was not the device that broke data center homogeneity at Google because some servers in its data centers already had GPUs. The TPU basically followed in the footsteps of the GPU and was just another type of accelerator.

The nonrecurring engineering costs were likely much higher for the TPU than for Catapult, but the rewards were also greater: both performance and performance/watt were much higher for an ASIC than for an FPGA. The risk was that the TPU was appropriate only for DNN inference, but as we mentioned, DNNs are an attractive target because they can potentially be used for many applications. In 2013

Google's management took a leap of faith by trusting that the DNN requirements in 2015 and beyond would justify investment in the TPU.

The deterministic execution model of both Catapult and the TPU is a better match to the response-time deadline of user-facing applications than are the time-varying optimizations of CPUs and GPUs (caches, out-of-order execution, multithreading, multiprocessing, prefetching, etc.) that help average throughput more than latency. The lack of such features helps explain why, despite having myriad ALUs and a big memory, the TPU is relatively small and low powered. This achievement suggests a "Cornucopia Corollary" to Amdahl's Law: *low utilization of a huge, cheap resource can still deliver high, cost-effective performance*.

In summary, the TPU succeeded for DNNs because of the large matrix unit; the substantial software-controlled on-chip memory; the ability to run whole inference models to reduce dependence on host CPU; a single-threaded, deterministic execution model that proved to be a good match to 99th-percentile response-time limits; enough flexibility to match the DNNs of 2017 as well as of 2013; the omission of general-purpose features that enabled a small and low-power die despite the larger datapath and memory; the use of 8-bit integers by the quantized applications; and the fact that applications were written using TensorFlow, which made it easy to port them to the DSA at high-performance rather than having to rewrite them in order for them to run well on the very different hardware.

Pixel Visual Core demonstrated the constraints of designing a DSA for a PMD in terms of die size and power. Unlike the TPU, it is a separate processor from the host that fetches its own instructions. Despite being aimed primarily at computer vision, Pixel Visual Core can run CNNs one to two orders of magnitude better in performance/watt than the K80 GPU and the Haswell CPU.

It's too early to render judgment on the Intel Crest, although its enthusiastic announcement by the Intel CEO signals a shift in the computing landscape.

An Architecture Renaissance

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 *10 times or more*.

We think that is a sign that the field is undergoing a transformation, and we expect to see a renaissance in architecture innovation in the next decade 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;
- the productivity advances in building hardware from both Agile hardware development and new hardware design languages that leverage advances in modern programming languages;

- the reduced cost of hardware development because of free and open instruction sets, open-source IP blocks, and commercial IP blocks (which so far is where most DSAs are found);
- the improvements mentioned above in productivity and cost of development means researchers can afford to demonstrate their ideas by building them in FPGAs or even in custom chips, instead of trying to convince skeptics with simulators; and
- the potential upside of DSAs and their synergy with domain-specific programming languages.

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: Google's Tensor Processing Unit and Acceleration of Deep Neural Networks

Concepts illustrated by this case study

- Structure of matrix multiplication operations
- Capacities of memories and rates of computations ("speeds and feeds") for a simple neural network model
- Construction of a special-purpose ISA
- Inefficiencies in mapping convolutions to TPU hardware
- Fixed-point arithmetic
- Function approximation

7.1 [10/20/10/25/25] <7.3,7.4> Matrix multiplication is a key operation supported in hardware by the TPU. Before going into details of the TPU hardware, it's worth analyzing the matrix multiplication calculation itself. One common way to depict matrix multiplication is with the following triply nested loop:

```
float a[M][K], b[K][N], c[M][N];
// M, N, and K are constants.
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];
```

a. [10] Suppose that M, N, and K are all equal. What is the asymptotic complexity in time of this algorithm? 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 transpose matrix B, swapping its indices 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[j][k];
```

Now, for which matrices are the elements accessed sequentially?

d. [25] The innermost (k-indexed) loop of our original routine performs a dot-product operation. Suppose that you are a given a hardware unit that can perform an 8-element dot-product more efficiently than the raw C code, behaving effectively like this C function:

```
void hardware_dot(float *accumulator,
   const float *a_slice, const float *b_slice) {
       float total = 0.;
       for (int k = 0; k < 8; ++k) {
              total += a_slice[k] * b_slice[k];
       }
       *accumulator += total;
}
```

How would you rewrite the routine with the transposed B matrix from part (c) to use this function?

e. [25] Suppose that instead, you are given a hardware unit that performs an 8-element "saxpy" operation, which behaves like this C function:

```
void hardware_saxpy(float *accumulator,
   float a, const float *input) {
        for (int k = 0; k < 8; ++k) {
               accumulator[k] += a * input[k];
        }
}
```

Write another routine that uses the saxpy primitive to deliver equivalent results to the original loop, without the transposed memory ordering for the B matrix.

7.2 [15/10/10/20/15/15/20/20] <7.3,7.4> Consider the neural network model MLP0 from Figure 7.5. That model has 20 M weights in five fully connected layers (neural network researchers count the input layer as if it were a layer in the stack, but it has no

weights associated with it). For simplicity, let's assume that those layers are each the same size, so each layer holds 4 M weights. Then assume that each layer has identical geometry, so each group of 4 M weights represents a 2 K*2 K matrix. Because the TPU typically uses 8-bit numerical values, 20 M weights take up 20 MB.

a. [15] For batch sizes of 128, 256, 512, 1024, and 2048, how big are the input activations for each layer of the model (which, except for the input layer, are also the output activations of the previous layer)? Now considering the whole mode (i.e., there's just the input to the first layer and the output from the last layer), for each batch size, what is the transfer time for input and output over PCIe Gen3 x16, which has a transfer speed of about 100 Gibit/s?

b. [10] Given the memory system speed of 30 GiB/s, give a lower bound for the time the TPU takes to read the weights of MLP0 from memory. How much time does it take for the TPU to read a 256×256 "tile" of weights from memory?

c. [10] Show how to calculate the TPU's 92 T operations/second, given that we know that the systolic array matrix multiplier has 256×256 elements, each of which performs an 8-bit multiply-accumulate operation (MAC) each cycle. In high-performance-computing marketing terms, a MAC counts as two operations.

d. [20] Once a weight tile has been loaded into the matrix unit of the TPU, it can be reused to multiply a 256-element input vector by the 256×256 weight matrix represented by the tile to produce a 256-element output vector every cycle. How many cycles pass during the time it takes to load a weight tile? This is the "break-even" batch size, where compute and memory-load times are equal, also known as the "ridge" of the roofline.

e. [15] The compute peak for the Intel Haswell x86 server is about 1 T FLOPS, while the compute peak for the NVIDIA K80 GPU is about 3 T FLOPS. Supposing that they hit these peak numbers, calculate their best-case compute time for batch size 128. How do these times compare to the time the TPU takes to load all 20 M weights from memory?

f. [15] Assuming that the TPU program does not overlap computation with I/O over PCIe, calculate the time elapsed from when the CPU starts to send the first byte of data to the TPU until the time that the last byte of output is returned. What fraction of PCIe bandwidth is used?

g. [20] Suppose that we deployed a configuration where one CPU was connected to five TPUs across a single PCIe Gen3 x16 bus (with appropriate PCIe switches). Assume that we parallelize by placing one layer of MLP0 on each TPU, and that the TPUs can communicate directly with each other over PCIe. At batch = 128, what is the best-case latency for calculating a single inference, and what throughput, in terms of inferences per second, would such a configuration deliver? How does this compare to a single TPU?

h. [20] Suppose that each example in a batch of inferences requires 50 core-microseconds of processing time on the CPU. How many cores on the host CPU will be required to drive a single-TPU configuration at batch = 128?

7.3 [20/25/25/25/Discussion] <7.3,7.4> Consider a pseudo-assembly language for the TPU, and consider the program that handles a batch of size 2048 for a tiny fully

connected layer with a 256×256 weight matrix. If there were no constraints on the sizes or alignments of computations in each instruction, the entire program for that layer might look like the following:

```
read_host u#0, 256*2048
read_weights w#0, 256*256
// matmul weights are implicitly read from the FIFO.
activate u#256*2048, a#0, 256*2048
write_host, u#256*2048, 256*2048
```

In this pseudo-assembly language, a prefix of "u#" refers to a memory address in the unified buffer; a prefix of "w#" refers to a memory address in the off-chip weight DRAM, and a prefix of "a#" refers to an accumulator address. The last argument of each assembly instruction describes the number of bytes to be operated upon.

Let's walk through the program instruction by instruction:

- The read_host instruction reads 512 KB of data from host memory, storing it at the very beginning of the unified buffer (u#0).
- The read_weights instruction tells the weight fetching unit to read 64 KB of weights, loading them into the on-chip weight FIFO. These 64 KB of weights represent a single, 256×256 matrix of weights, which we will call a "weight tile."
- The matmul instruction reads the 512 KB of input data from address 0 in the unified buffer, performs a matrix multiplication with the tile of weights, and stores the resulting 256*2048 = 524,288, 32-bit activations at accumulator address 0 (a#0). We have intentionally glossed over the details of the ordering of weights; the exercise will expand on these details.
- The activate instruction takes those 524,288 32-bit accumulators at a#0, applies an activation function to them, and stores the resulting 524,288, 8-bit output values at the next free location in the unified buffer, u#524288.
- The write_host instruction writes the 512 KB of output activations, starting at u#524288, back to the host CPU.

We will progressively add realistic details to the pseudo-assembly language to explore some aspects of TPU design.

a. [20] While we have written our pseudo-code in terms of bytes and byte addresses (or in the case of the accumulators, in terms of addresses to 32-bit values), the TPU operates on a natural vector length of 256. This means that the unified buffer is typically addressed at 256-byte boundaries, the accumulators are addressed in groups of 256 32-bit values (or at 1 KB boundaries), and weights are loaded in groups of 65,536 8-bit values. Rewrite the program's addresses and transfer sizes to take these vector and weight-tile lengths into account. How many 256-element vectors of input activations will be read by the program? How many bytes of accumulated values will be used while computing the results? How many 256-element vectors of output activations will be written back to the host?

b. [25] Suppose that the application requirements change, and instead of a multiplication by a 256×256 weight matrix, the shape of the weight matrix now becomes 1024×256. Think of the matmul instruction as putting the weights as the right argument of the matrix multiplication operator, so 1024 corresponds to K, the dimension in which the matrix multiplication adds up values. Suppose that there are now two variants of the accumulate instruction, one of which overwrites the accumulators with its results, and the other of which adds the matrix multiplication results to the specified accumulator. How would you change the program to handle this 1024×256 matrix? Do you need more accumulators? The size of the matrix unit remains the same at 256×256; how many 256×256 weight tiles does your program need?

c. [25] Now write the program to handle a multiplication by a weight matrix of size 256×512. Does your program need more accumulators? Can you rewrite your program so that it uses only 2048, 256-entry accumulators? How many weight tiles does your program need? In what order should they be stored in the weight DRAM?

d. [25] Next, write the program to handle a multiplication by a weight matrix of size 1024×768. How many weight tiles does your program need? Write your program so that it uses only 2048, 256-entry accumulators. In what order should the weight tiles be stored in the weight DRAM? For this calculation, how many times did each input activation get read?

e. [Discussion] What would it take to build an architecture that reads each 256-element set of input activations just once? How many accumulators would that require? If you did it that way, how big would the accumulator memory have to be? Contrast this approach with the TPU, which uses 4096 accumulators, so that one set of 2048 accumulators can be written by the matrix unit while another is being used for activations.

7.4 [15/15/15] <7.3,7.4>Consider the first convolutional layer of AlexNet, which uses a 7×7 convolutional kernel, with an input feature depth of 3 and an output feature depth of 48. The original image width is 220×220.

a. [15] Ignore the 7×7 convolutional kernel for the moment, and consider just the center element of that kernel. A 1×1 convolutional kernel is mathematically equivalent to a matrix multiplication, using a weight matrix that is input_depth $\times$ output_depth in dimensions. With these depths, and using a standard matrix multiplication, what fraction of the TPU's 65,536 ALUs can be used?

b. [15] For convolutional neural networks, the spatial dimensions are also sources of weight reuse, since the convolutional kernel gets applied to many different (x,y) coordinate positions. Suppose that the TPU reaches balanced compute and memory at a batch size of 1400 (as you might have computed in exercise 1d). What is the smallest square image size that the TPU can process efficiently at a batch size of 1?

c. [15] The first convolutional layer of AlexNet implements a *kernel stride* of 4, which means that rather than moving by one X or Y pixel at each application,

the 7×7 kernel moves by 4 pixels at a time. This striding means that we can permute the input data from $220 \times 220 \times 3$ to be $55 \times 55 \times 48$ (dividing the X and Y dimensions by 4 and multiplying the input depth by 16), and simultaneously we can restack the $7 \times 7 \times 3 \times 48$ convolutional weights to be $2 \times 2 \times 48 \times 48$ (just as the input data gets restacked by 4 in X and Y, we do the same to the 7×7 elements of the convolutional kernel, ending up with ceiling(7/4) = 2 elements in each of the X and Y dimensions). Because the kernel is now 2×2, we need to perform only four matrix multiplication operations, using weight matrices of size 48×48. What is the fraction of the 65,536 ALUs that can be used now?

7.5 [15/10/20/20/20/25] <7.3> The TPU uses *fixed-point arithmetic* (sometimes also called *quantized arithmetic*, with overlapping and conflicting definitions), where integers are used to represent values on the real number line. There are a number of different schemes for fixed-point arithmetic, but they share the common theme that there is an affine projection from the integer used by hardware to the real number that the integer represents. An affine projection has the form r = i*s + b, where i is the integer, r is the represented real value, and s and b are a scale and bias. You can of course write the projection in either direction, from integers to reals or vice versa (although you need to round when converting from reals to integers).

a. [15] The simplest activation function supported by the TPU is "ReLUX," which is a rectified linear unit with a maximum of X. For example, ReLU6 is defined by Relu6(x) = { 0, when x < 0; x, when 0 <= x <= 6; and 6, when x > 6 }. So 0.0 and 6.0 on the real number line are the minimum and maximum values that Relu6 might produce. Assume that you use an 8-bit unsigned integer in hardware, and that you want to make 0 map to 0.0 and 255 map to 6.0. Solve for s and b.

b. [10] How many values on the real number line are exactly representable by an 8-bit quantized representation of ReLU6 output? What is the real-number spacing between them?

c. [20] The difference between representable values is sometimes called a "unit in the least place," or *ulp*, when performing numerical analysis. If you map a real number to its fixed-point representation, then map back, you only rarely get back the original real number. The difference between the original number and its representation is called the *quantization error*. When mapping a real number in the range [0.0,6.0] to an 8-bit integer, show that the worst-case quantization error is one-half of an ulp (make sure you round to the nearest representable value). You might consider graphing the errors as a function of the original real number.

d. [20] Keep the real-number range [0.0,6.0] for an 8-bit integer from the last step. What 8-bit unsigned integer represents 1.0? What is the quantization error for 1.0? Suppose that you ask the TPU to add 1.0 to 1.0. What answer do you get back, and what is the error in that result?

e. [20] If you pick a random number uniformly in the range [0.0, 6.0], then quantize it to an 8-bit unsigned integer, what distribution would you expect to see for the 256 integer values?

f. [25] The hyperbolic tangent function, *tanh*, is another commonly used activation function in deep learning: $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$

Tanh also has a bounded range, mapping the entire real number line to the interval $(-1.0, 1.0)$. Solve for s and b for this range, using an 8-bit unsigned representation. Then solve for s and b using an 8-bit two's complement representation. For both cases, what real number does the integer 0 represent? Which integer represents the real number 0.0? Can you imagine any issues that might result from the quantization error incurred when representing 0.0?

7.6 [20/25/15/15/30/30/30/40/40/25/20/Discussion] <7.3> In addition to tanh, another s-shaped smooth function, the logistic sigmoid function $y = 1/(1+\exp(-x))$,

$$logistic\_sigmoid(x) = \frac{1}{1+e^{-x}}$$

is commonly used as an activation function in neural networks. A common way to implement them in fixed-point arithmetic uses a piecewise quadratic approximation, where the most significant bits of the input value select which table entry to use. Then the least significant bits of the input value are sent to a degree-2 polynomial that describes a parabola that is fit to the subrange of the approximated function.

a. [20] Using a graphing tool (we like www.desmos.com/calculator), draw the graphs for the logistic sigmoid and tanh functions.

b. [25] Now draw the graph of $y = \tanh(x/2)/2$. Compare that graph with the logistic sigmoid function. How much do they differ by? Build an equation that shows how to transform one into the other. Prove that your equation is correct.

c. [15] Given this algebraic identity, do you need to use two different sets of coefficients to approximate logistic sigmoid and tanh?

d. [15] Tanh is an odd function, meaning that $f(-x) = -f(x)$. Can you exploit this fact to save table space?

e. [30] Let's focus our attention on approximating *tanh* over the interval $x \in [0.0, 6.4]$ on the number line. Using floating-point arithmetic, write a program that divides the interval into 64 subintervals (each of length 0.1), and then approximates the value of *tanh* over each subinterval using a single constant floating-point value (so you'll need to pick 64 different floating-point values, one for each subinterval). If you spot-check 100 different values (randomly chosen is fine) within each subinterval, what is the worst-case approximation error you see over all subintervals? Can you choose your constant to minimize the approximation error for each subinterval?

f. [30] Now consider building a floating-point linear approximation for each subinterval. In this case, you want to pick a pair of floating-point values m and b, for the traditional line equation $y = mx + b$, to approximate each of the 64 subintervals. Come up with a strategy that you think is reasonable to build this linear interpolation over 64 subintervals for tanh. Measure the worst-case approximation error over the 64 intervals. Is your approximation monotonic when it reaches a boundary between subintervals?

g. [40] Next, build a quadratic approximation, using the standard formula $y = ax^2 + bx + c$. Experiment with a number of different ways to fit the formula. Try fitting the parabola to the endpoints and midpoint of the bucket, or using a Taylor approximation around a single point in the bucket. What worst-case error do you get?

h. [40] (extra credit) Let's combine the numerical approximations of this exercise with the fixed-point arithmetic of the previous exercise. Suppose that the input $x \in [0.0, 6.4]$ is represented by a 15-bit unsigned value, with 0x0000 representing 0.0 and 0x7FFF representing 6.4. For the output, similarly use a 15-bit unsigned value, with 0x0000 representing 0.0 and 0x7FFF representing 1.0. For each of your constant, linear, and quadratic approximations, calculate the combined effect of approximation and quantization errors. Since there are so few input values, you can write a program to check them exhaustively.

i. [25] For the quadratic, quantized approximation, is your approximation monotonic within each subinterval?

j. [20] A difference of one ulp in the output scale should correspond to an error of 1.0 / 32767. How many ulps of error are you seeing in each case?

k. [Discussion] By choosing to approximate the interval [0.0, 6.4], we effectively clipped the "tail" of the hyperbolic tangent function, for values of $x > 6.4$. It's not an unreasonable approximation to set the output value for all of the tail to 1.0. What's the worst-case error, in terms of both real numbers and ulps, of treating the tail this way? Is there a better place we might have clipped the tail to improve our accuracy?

Exercises

7.7 [10/20/10/15] <7.2,7.5> One popular family of FPGAs, the Virtex-7 series, is built by Xilinx. A Virtex-7 XC7VX690T FPGA contains 3,600 25x18-bit integer multiply-add "DSP slices." Consider building a TPU-style design on such an FPGA.

a. [10] Using one 25×18 integer multiplier per systolic array cell, what's the largest matrix multiplication unit one could construct? Assume that the matrix multiplication unit must be square.

b. [20] Suppose that you could build a rectangular, nonsquare matrix multiplication unit. What implications would such a design have for hardware and software? (Hint: think about the vector length that software must handle.)

c. [10] Many FPGA designs are lucky to reach 500 MHz operation. At that speed, calculate the peak 8-bit operations per second that such a device might achieve. How does that compare to the 3 T FLOPS of a K80 GPU?

d. [15] Assume that you can make up the difference between 3600 and 4096 DSP slices using LUTs, but that doing so will reduce your clock rate to 350 MHz. Is this a worthwhile trade-off to make?

7.8 [15/15/15] <7.9> Amazon Web Services (AWS) offers a wide variety of "computing instances," which are machines configured to target different applications and scales. AWS prices tell us useful data about the Total Cost of Ownership (TCO) of various computing devices, particularly as computer equipment is often depreciated[1] on a 3-year schedule. As of July 2017, a dedicated, compute-oriented "c4" computing instance includes two x86 chips with 20 physical cores in total. It rents on-demand for $1.75/hour, or $17,962 for 3 years. In contrast, a dedicated "p2" computing instance also has two x86 chips but with 36 cores in total, and adds 16 NVIDIA K80 GPUs. A p2 rents on-demand for $15.84/hour, or $184,780 for 3 years.

a. [15] The c4 instance uses Intel Xeon E5-2666 v3 (Haswell) processors. The p2 instance uses Intel Xeon E5-2686 v4 (Broadwell) processors. Neither part number is listed officially on Intel's product website, which suggests that these parts are specially built for Amazon by Intel. The E5-2660 v3 part has a similar core count to the E5-2666 v3 and has a street price of around $1500. The E5-2697 v4 part has a similar core count to the E5-2686 v4 and has a street price of around $3000. Assume that the non-GPU portion of the p2 instance would have a price proportional to the ratio of street prices. What is the TCO, over 3 years, for a single K80 GPU?

b. [15] Suppose that you have a compute- and throughput-dominated workload that runs at rate 1 on the c4 instance and at rate T on the GPU-accelerated p2 instance. How large must T be for the GPU-based solution to be more cost-effective? Suppose that each general-purpose CPU core can compute at a rate of about 30G single-precision FLOPS. Ignoring the CPUs of the p2 instance, what fraction of peak K80 FLOPs would be required to reach the same rate of computation as the c4 instance?

c. [15] AWS also offers "f1" instances that include 8 Xilinx Ultrascale + VU9P FPGAs. They rent at $13.20/hour, or $165,758 for 3 years. Each VU9P device includes 6840 DSP slices, which can perform 27×18-bit integer multiply-accumulate operations (recall that one multiply-accumulate counts as two "operations"). At 500 MHz, what is the peak multiply-accumulate operations/cycle that an f1-based system might achieve, counting all 8 FPGAs toward the computation total? Assuming that the integer operations on the FPGAs can substitute for floating-point operations, how does this compare to the peak single-precision multiply-accumulate operations/cycle of the GPUs of the p2 instance? How do they compare in terms of cost-effectiveness?

7.9 [20/20/25] <7.7> As shown in Figure 7.34 (but simplified to fewer PEs), each Pixel Visual Core includes a 16×16 set of full processing elements, surrounded

[1] Capital expenses are accounted for over the lifetime of an asset, using a "depreciation schedule." Rather than taking a one-time charge at the point where an asset is acquired, standard accounting practice spreads out the capital cost over the lifetime of the asset. So one might account for a $30,000 device that has a useful life of 3 years by assigning $10,000 in depreciation to each year.

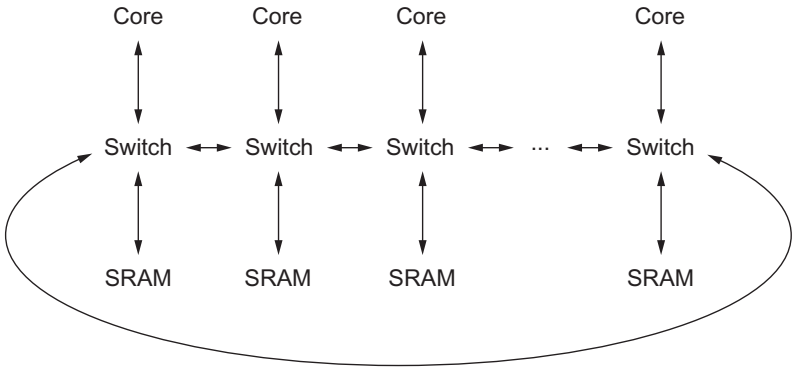
by an additional two layers of "simplified" processing elements. Simplified PEs can store and communicate data but omit the computation hardware of full PEs. Simplified PEs store copies of data that might be the "home data" of a neighboring core, so there are $(16+2+2)^2=400$ PEs in total, 256 full and 144 simplified.

a. [20] Suppose that you wanted to process a 64×32 grayscale image with a 5×5 stencil using 8 Pixel Visual Cores. For now, assume that the image is laid out in raster-scan order (pixels that are adjacent in X are adjacent in memory, while pixels that are adjacent in Y are 64 memory locations apart). For each of the 8 cores, describe the memory region that the core should import to handle its part of the image. Make sure to include the halo region. Which parts of the halo region should be zeroed by software to ensure correct operation? You may find it convenient to refer to subregions of the image using a 2D slice notation, where for example image[2:5][6:13] refers to the set of pixels whose x component is $2 <= x < 5$ and whose y component is $6 <= y < 13$ (the slices are half-open following Python slicing practice).

b. [20] If we change to a 3×3 stencil, how do the regions imported from memory change? How many halo-simplified PEs go unused?

c. [25] Now consider how to support a 7×7 stencil. In this case, we don't have as many hardware-supported simplified PEs as we need to cover the three pixels worth of halo data that "belong to" neighboring cores. To handle this, we use the outermost ring of full PEs as if they were simplified PEs. How many pixels can we handle in a single core using this strategy? How many "tiles" are now required to handle our 64×32 input image? What is the utilization of our full PEs over the complete processing time for the 7×7 stencil over the 64×32 image?

7.10 [20/20/20/25/25] <7.7> Consider a case in which each of the eight cores on a Pixel Visual Core device is connected through a four-port switch to a 2D SRAM, forming a core + memory unit. The remaining two ports on the switch link these units in a ring, so that each core is able to access any of the eight SRAMs. However, this ring-based network-on-chip topology makes some data access patterns more efficient than others.



a. [20] Suppose that each link in the NOC has the same bandwidth B, and that each link is full-duplex, so it can simultaneously transfer bandwidth B in each direction. Links connect the core to the switch, the switch to SRAM, and pairs of switches in the ring. Assume that each local memory has at least B bandwidth, so it can saturate its link. Consider a memory access pattern where each of the eight PEs access only the closest memory (the one connected via the switch of the core + memory unit). What is the maximum memory bandwidth that the core will be able to achieve?

b. [20] Now consider an off-by-one access pattern, where core i accesses memory $i+1$, going through three links to reach that memory (core 7 will access memory 0, because of the ring topology). What is the maximum memory bandwidth that the core will be able to achieve in this case? To achieve that bandwidth, do you need to make any assumptions about the capabilities of the 4-port switch? What if the switch can only move data at rate B?

c. [20] Consider an off-by-two access pattern, where core i access memory $i+2$. Once again, what is the maximum memory bandwidth that the core will be able to achieve in this case? Where are the bottleneck links in the network-on-chip?

d. [25] Consider a uniform random memory access pattern, where each core uses each of the SRAMs for ⅛ of its memory requests. Assuming this traffic pattern, how much traffic traverses a switch-to-switch link, compared to the amount of traffic between a core and its associated switch or between an SRAM and its associated switch?

e. [25] (advanced) Can you conceive of a case (workload) where this network can deadlock? From the standpoint of software-only solutions, what should the compiler do to avoid such a scenario? If you can make changes to hardware, what changes in routing topology (and routing scheme) would guarantee no deadlocks?

7.11 <7.2> The first Anton molecular dynamics supercomputer typically simulated a box of water that was 64 Å on a side. The computer itself might be approximated as a box with 1 m side length. A single simulation step represented 2.5 fs of simulation time, and took about 10 μs 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. Calculate the spatial expansion factor from simulation space to hardware in real space.

b. Calculate the temporal slowdown factor from simulated time to wall-clock time.

c. These two numbers 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. 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 m on a side? What about simulating on a world-spanning Cloud service?

7.12 <7.2> The Anton communication network is a 3D, $8 \times 8 \times 8$ torus, where each node in the system has six links to neighboring nodes. Latency for a packet to transit single link is about 50 ns. Ignore on-chip switching time between links for this exercise.

a. What is the diameter (maximum number of hops between a pair of nodes) of the communication network? Given that diameter, what is the shortest latency required to broadcast a single value from one node of the machine to all 512 nodes of the machine?

b. Assuming that adding up two values takes zero time, what is the shortest latency to add up a sum over 512 values to a single node, where each value starts on a different node of the machine?

c. Once again assume that you want to perform the sum over 512 values, but you want each of the 512 nodes of the system to end up with a copy of the sum. Of course you could perform a global reduction followed by a broadcast. Can you do the combined operation in less time? This pattern is called an *all-reduce*. Compare the times of your all-reduce pattern to the time of a broadcast from a single node or a global sum to a single node. Compare the bandwidth used by the all-reduce pattern with the other patterns.

A

Instruction Set Principles

A *n* Add the number in storage location *n* into the accumulator.

E *n* If the number in the accumulator is greater than or equal to zero execute next the order which stands in storage location *n*; otherwise proceed serially.

Z Stop the machine and ring the warning bell.

Wilkes and Renwick,
Selection from the List of 18 Machine Instructions for the EDSAC (1949)

A.1 Introduction

In this appendix we concentrate on instruction set architecture—the portion of the computer visible to the programmer or compiler writer. Most of this material should be review for readers of this book; we include it here for background. This appendix introduces the wide variety of design alternatives available to the instruction set architect. In particular, we focus on four topics. First, we present a taxonomy of instruction set alternatives and give some qualitative assessment of the advantages and disadvantages of various approaches. Second, we present and analyze some instruction set measurements that are largely independent of a specific instruction set. Third, we address the issue of languages and compilers and their bearing on instruction set architecture. Finally, the "Putting It All Together" section shows how these ideas are reflected in the RISC-V instruction set, which is typical of RISC architectures. We conclude with fallacies and pitfalls of instruction set design.

To illustrate the principles further and to provide a comparison with RISC-V, Appendix K also gives four examples of other general-purpose RISC architectures (MIPS, Power ISA, SPARC, and Armv8), four embedded RISC processors (ARM Thumb2, RISC-V Compressed, microMIPS), and three older architectures (80x86, IBM 360/370, and VAX). Before we discuss how to classify architectures, we need to say something about instruction set measurement.

Throughout this appendix, we examine a wide variety of architectural measurements. Clearly, these measurements depend on the programs measured and on the compilers used in making the measurements. The results should not be interpreted as absolute, and you might see different data if you did the measurement with a different compiler or a different set of programs. We believe that the measurements in this appendix are reasonably indicative of a class of typical applications. Many of the measurements are presented using a small set of benchmarks, so that the data can be reasonably displayed and the differences among programs can be seen. An architect for a new computer would want to analyze a much larger collection of programs before making architectural decisions. The measurements shown are usually *dynamic*—that is, the frequency of a measured event is weighed by the number of times that event occurs during execution of the measured program.

Before starting with the general principles, let's review the three application areas from Chapter 1. *Desktop computing* emphasizes the performance of programs with integer and floating-point data types, with little regard for program size. For example, code size has never been reported in the five generations of SPEC benchmarks. *Servers* today are used primarily for database, file server, and Web applications, plus some time-sharing applications for many users. Hence, floating-point performance is much less important for performance than integers and character strings, yet virtually every server processor still includes floating-point instructions. *Personal mobile devices* and *embedded applications* value cost and energy, so code size is important because less memory is both cheaper and lower energy, and some classes of instructions (such as floating point) may be optional to reduce chip costs, and a compressed version of the instructions set designed to save memory space may be used.

Thus, instruction sets for all three applications are very similar. In fact, architectures similar to RISC-V, which we focus on here, have been used successfully in desktops, servers, and embedded applications.

One successful architecture very different from RISC is the 80x86 (see Appendix K). Surprisingly, its success does not necessarily belie the advantages of a RISC instruction set. The commercial importance of binary compatibility with PC software combined with the abundance of transistors provided by Moore's Law led Intel to use a RISC instruction set internally while supporting an 80x86 instruction set externally. Recent 80x86 microprocessors, including all the Intel Core microprocessors built in the past decade, use hardware to translate from 80x86 instructions to RISC-like instructions and then execute the translated operations inside the chip. They maintain the illusion of 80x86 architecture to the programmer while allowing the computer designer to implement a RISC-style processor for performance. There remain, however, serious disadvantages for a complex instruction set like the 80x86, and we discuss these further in the conclusions.

Now that the background is set, we begin by exploring how instruction set architectures can be classified.

A.2 Classifying Instruction Set Architectures

The type of internal storage in a processor is the most basic differentiation, so in this section we will focus on the alternatives for this portion of the architecture. The major choices are a stack, an accumulator, or a set of registers. Operands may be named explicitly or implicitly: The operands in a *stack architecture* are implicitly on the top of the stack, and in an *accumulator architecture* one operand is implicitly the accumulator. The *general-purpose register architectures* have only explicit operands—either registers or memory locations. Figure A.1 shows a block diagram of such architectures, and Figure A.2 shows how the code sequence C = A + B would typically appear in these three classes of instruction sets. The explicit operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of architecture and choice of specific instruction.

As the figures show, there are really two classes of register computers. One class can access memory as part of any instruction, called *register-memory* architecture, and the other can access memory only with load and store instructions, called *load-store* architecture. A third class, not found in computers shipping today, keeps all operands in memory and is called a *memory-memory* architecture. Some instruction set architectures have more registers than a single accumulator but place restrictions on uses of these special registers. Such an architecture is sometimes called an *extended accumulator* or *special-purpose register* computer.

Although most early computers used stack or accumulator-style architectures, virtually every new architecture designed after 1980 uses a load-store register architecture. The major reasons for the emergence of general-purpose register (GPR) computers are twofold. First, registers—like other forms of storage internal to the processor—are faster than memory. Second, registers are more efficient for a

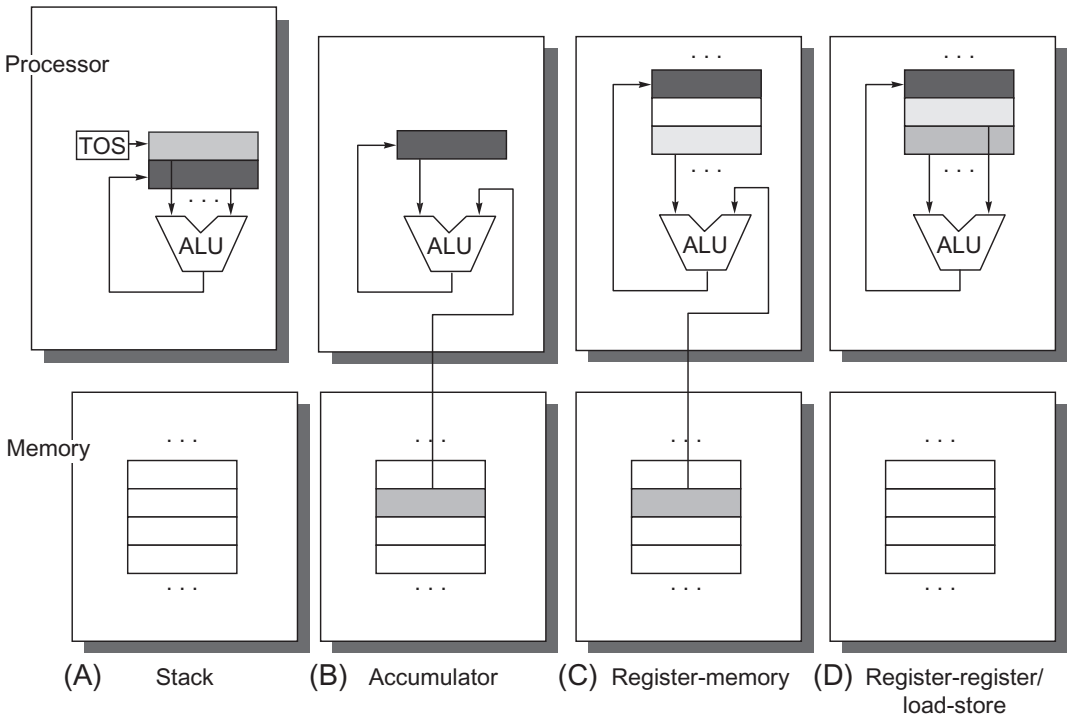


Figure A.1 Operand locations for four instruction set architecture classes. The arrows indicate whether the operand is an input or the result of the arithmetic-logical unit (ALU) operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result. In (A), a top of stack (TOS) register points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit. In (B), the accumulator is both an implicit input operand and a result. In (C), one input operand is a register, one is in memory, and the result goes to a register. All operands are registers in (D) and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (A) and load or store for (D).

Stack	Accumulator	Register (register-memory)	Register (load-store)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R3,R1,B`	`Load R2,B`
`Add`	`Store C`	`Store R3,C`	`Add R3,R1,R2`
`Pop C`			`Store R3,C`

Figure A.2 The code sequence for `C = A + B` for four classes of instruction sets. Note that the `Add` instruction has implicit operands for stack and accumulator architectures and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure A.1 shows the `Add` operation for each class of architecture.

compiler to use than other forms of internal storage. For example, on a register computer the expression `(A * B) + (B * C) - (A * D)` may be evaluated by doing the multiplications in any order, which may be more efficient because of the location of the operands or because of pipelining concerns (see Chapter 3). Nevertheless, on a stack computer the hardware must evaluate the expression in only one order, because operands are hidden on the stack, and it may have to load an operand multiple times.

More importantly, registers can be used to hold variables. When variables are allocated to registers, the memory traffic reduces, the program speeds up (because registers are faster than memory), and the code density improves (because a register can be named with fewer bits than can a memory location).

As explained in Section A.8, compiler writers would prefer that all registers be equivalent and unreserved. Older computers compromise this desire by dedicating registers to special uses, effectively decreasing the number of general-purpose registers. If the number of truly general-purpose registers is too small, trying to allocate variables to registers will not be profitable. Instead, the compiler will reserve all the uncommitted registers for use in expression evaluation.

How many registers are sufficient? The answer, of course, depends on the effectiveness of the compiler. Most compilers reserve some registers for expression evaluation, use some for parameter passing, and allow the remainder to be allocated to hold variables. Modern compiler technology and its ability to effectively use larger numbers of registers has led to an increase in register counts in more recent architectures.

Two major instruction set characteristics divide GPR architectures. Both characteristics concern the nature of operands for a typical arithmetic or logical instruction (ALU instruction). The first concerns whether an ALU instruction has two or three operands. In the three-operand format, the instruction contains one result operand and two source operands. In the two-operand format, one of the operands is both a source and a result for the operation. The second distinction among GPR architectures concerns how many of the operands may be memory addresses in ALU instructions. The number of memory operands supported by a typical ALU instruction may vary from none to three. Figure A.3 shows combinations of these two attributes with examples of computers. Although there are seven

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Figure A.3 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers. Computers with no memory reference per ALU instruction are called load-store or register-register computers. Instructions with multiple memory operands per typical ALU instruction are called register-memory or memory-memory, according to whether they have one or more than one memory operand.

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C)	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs, which may have some instruction cache effects
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density	Operands are not equivalent because a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

Figure A.4 Advantages and disadvantages of the three most common types of general-purpose register computers. The notation (*m*, *n*) means *m* memory operands and *n* total operands. In general, computers with fewer alternatives simplify the compiler's task because there are fewer decisions for the compiler to make (see Section A.8). Computers with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. The number of registers also affects the instruction size because you need $\log_2$ (number of registers) for each register specifier in an instruction. Thus, doubling the number of registers takes three extra bits for a register-register architecture, or about 10% of a 32-bit instruction.

possible combinations, three serve to classify nearly all existing computers. As we mentioned earlier, these three are load-store (also called register-register), register-memory, and memory-memory.

Figure A.4 shows the advantages and disadvantages of each of these alternatives. Of course, these advantages and disadvantages are not absolutes: they are qualitative and their actual impact depends on the compiler and implementation strategy. A GPR computer with memory-memory operations could easily be ignored by the compiler and used as a load-store computer. One of the most pervasive architectural impacts is on instruction encoding and the number of instructions needed to perform a task. We see the impact of these architectural alternatives on implementation approaches in Appendix C and Chapter 3.

Summary: Classifying Instruction Set Architectures

Here and at the end of Sections A.3–A.8 we summarize those characteristics we would expect to find in a new instruction set architecture, building the foundation for the RISC-V architecture introduced in Section A.9. From this section we should clearly expect the use of general-purpose registers. Figure A.4, combined with Appendix C on pipelining, leads to the expectation of a load-store version of a general-purpose register architecture.

With the class of architecture covered, the next topic is addressing operands.

A.3 Memory Addressing

Independent of whether the architecture is load-store or allows any operand to be a memory reference, it must define how memory addresses are interpreted and how they are specified. The measurements presented here are largely, but not completely, computer independent. In some cases the measurements are significantly affected by the compiler technology. These measurements have been made using an optimizing compiler, because compiler technology plays a critical role.

Interpreting Memory Addresses

How is a memory address interpreted? That is, what object is accessed as a function of the address and the length? All the instruction sets discussed in this book are byte addressed and provide access for bytes (8 bits), half words (16 bits), and words (32 bits). Most of the computers also provide access for double words (64 bits).

There are two different conventions for ordering the bytes within a larger object. *Little Endian* byte order puts the byte whose address is "x … x000" at the least-significant position in the double word (the little end). The bytes are numbered:

7	6	5	4	3	2	1	0

Big Endian byte order puts the byte whose address is "x … x000" at the most-significant position in the double word (the big end). The bytes are numbered:

0	1	2	3	4	5	6	7

When operating within one computer, the byte order is often unnoticeable—only programs that access the same locations as both, say, words and bytes, can notice the difference. Byte order is a problem when exchanging data among computers with different orderings, however. Little Endian ordering also fails to match the normal ordering of words when strings are compared. Strings appear "SDRAWKCAB" (backwards) in the registers.

A second memory issue is that in many computers, accesses to objects larger than a byte must be *aligned*. An access to an object of size s bytes at byte address A is aligned if $A \bmod s = 0$. Figure A.5 shows the addresses at which an access is aligned or misaligned.

Why would someone design a computer with alignment restrictions? Misalignment causes hardware complications, because the memory is typically aligned on a multiple of a word or double-word boundary. A misaligned memory access may, therefore, take multiple aligned memory references. Thus, even in computers that allow misaligned access, programs with aligned accesses run faster.

	Value of three low-order bits of byte address							
Width of object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

Figure A.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order three bits of the address.

Even if data are aligned, supporting byte, half-word, and word accesses requires an alignment network to align bytes, half words, and words in 64-bit registers. For example, in Figure A.5, suppose we read a byte from an address with its 3 low-order bits having the value 4. We will need to shift right 3 bytes to align the byte to the proper place in a 64-bit register. Depending on the instruction, the computer may also need to sign-extend the quantity. Stores are easy: only the addressed bytes in memory may be altered. On some computers a byte, half-word, and word operation does not affect the upper portion of a register. Although all the computers discussed in this book permit byte, half-word, and word accesses to memory, only the IBM 360/370, Intel 80x86, and VAX support ALU operations on register operands narrower than the full width.

Now that we have discussed alternative interpretations of memory addresses, we can discuss the ways addresses are specified by instructions, called *addressing modes*.

Addressing Modes

Given an address, we now know what bytes to access in memory. In this subsection we will look at addressing modes—how architectures specify the address

of an object they will access. Addressing modes specify constants and registers in addition to locations in memory. When a memory location is used, the actual memory address specified by the addressing mode is called the *effective address*.

Figure A.6 shows all the data addressing modes that have been used in recent computers. Immediates or literals are usually considered memory addressing modes (even though the value they access is in the instruction stream), although registers are often separated because they don't usually have memory addresses. We have kept addressing modes that depend on the program counter, called *PC-relative addressing*, separate. PC-relative addressing is used primarily for specifying code addresses in control transfer instructions, discussed in Section A.6.

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4]←Regs[R4] + Regs[R3]`	When a value is in a register
Immediate	`Add R4,3`	`Regs[R4]←Regs[R4]+3`	For constants
Displacement	`Add R4,100(R1)`	`Regs[R4]←Regs[R4] + Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	`Add R4,(R1)`	`Regs[R4]←Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address
Indexed	`Add R3,(R1+R2)`	`Regs[R3]←Regs[R3] + Mem[Regs[R1]+Regs [R2]]`	Sometimes useful in array addressing: `R1` = base of array; `R2` = index amount
Direct or absolute	`Add R1,(1001)`	`Regs[R1]←Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	`Add R1,@(R3)`	`Regs[R1]←Regs[R1] + Mem[Mem[Regs[R3]]]`	If `R3` is the address of a pointer p, then mode yields $*p$
Autoincrement	`Add R1,(R2)+`	`Regs[R1]←Regs[R1] + Mem[Regs[R2]]` `Regs[R2]←Regs[R2]+`d	Useful for stepping through arrays within a loop. `R2` points to start of array; each reference increments `R2` by size of an element, d
Autodecrement	`Add R1, -(R2)`	`Regs[R2]←Regs[R2] -`d `Regs[R1]←Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1]←Regs[R1] + Mem[100+Regs[R2] + Regs[R3] *`d`]`	Used to index arrays. May be applied to any indexed addressing mode in some computers

Figure A.6 Selection of addressing modes with examples, meaning, and usage. In autoincrement/-decrement and scaled addressing modes, the variable *d* designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode. The extensions to C used as hardware descriptions are defined on page A.38.

Figure A.6 shows the most common names for the addressing modes, though the names differ among architectures. In this figure and throughout the book, we will use an extension of the C programming language as a hardware description notation. In this figure, only one non-C feature is used: the left arrow (←) is used for assignment. We also use the array `Mem` as the name for main memory and the array `Regs` for registers. Thus, `Mem[Regs[R1]]` refers to the contents of the memory location whose address is given by the contents of register 1 (`R1`). Later, we will introduce extensions for accessing and transferring data smaller than a word.

Addressing modes have the ability to significantly reduce instruction counts; they also add to the complexity of building a computer and may increase the average clock cycles per instruction (CPI) of computers that implement those modes. Thus, the usage of various addressing modes is quite important in helping the architect choose what to include.

Figure A.7 shows the results of measuring addressing mode usage patterns in three programs on the VAX architecture. We use the old VAX architecture for a few measurements in this appendix because it has the richest set of addressing

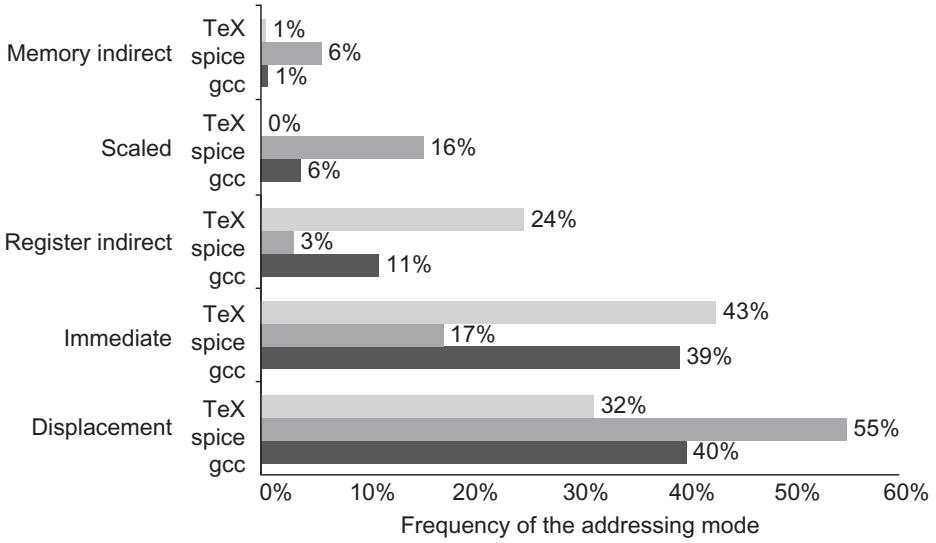


Figure A.7 Summary of use of memory addressing modes (including immediates). These major addressing modes account for all but a few percent (0%–3%) of the memory accesses. Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. Of course, the compiler affects what addressing modes are used; see Section A.8. The memory indirect mode on the VAX can use displacement, autoincrement, or autodecrement to form the initial memory address; in these programs, almost all the memory indirect references use displacement mode as the base. Displacement mode includes all displacement lengths (8, 16, and 32 bits). The PC-relative addressing modes, used almost exclusively for branches, are not included. Only the addressing modes with an average frequency of over 1% are shown.

modes and the fewest restrictions on memory addressing. For example, Figure A.6 on page A.9 shows all the modes the VAX supports. Most measurements in this appendix, however, will use the more recent register-register architectures to show how programs use instruction sets of current computers.

As Figure A.7 shows, displacement and immediate addressing dominate addressing mode usage. Let's look at some properties of these two heavily used modes.

Displacement Addressing Mode

The major question that arises for a displacement-style addressing mode is that of the range of displacements used. Based on the use of various displacement sizes, a decision of what sizes to support can be made. Choosing the displacement field sizes is important because they directly affect the instruction length. Figure A.8

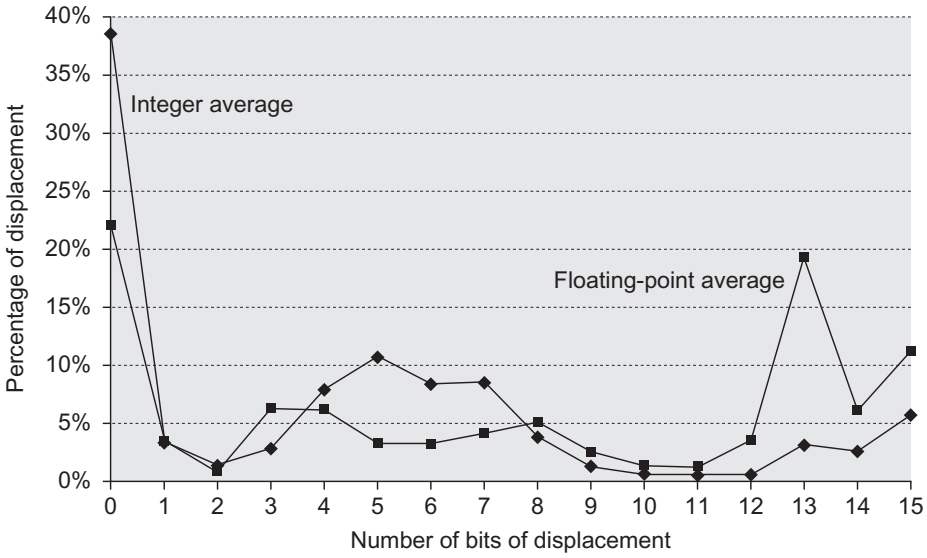


Figure A.8 Displacement values are widely distributed. There are both a large number of small values and a fair number of large values. The wide distribution of displacement values is due to multiple storage areas for variables and different displacements to access them (see Section A.8) as well as the overall addressing scheme the compiler uses. The *x*-axis is $\log_2$ of the displacement, that is, the size of a field needed to represent the magnitude of the displacement. Zero on the *x*-axis shows the percentage of displacements of value 0. The graph does not include the sign bit, which is heavily affected by the storage layout. Most displacements are positive, but a majority of the largest displacements (14+ bits) are negative. Because these data were collected on a computer with 16-bit displacements, they cannot tell us about longer displacements. These data were taken on the Alpha architecture with full optimization (see Section A.8) for SPEC CPU2000, showing the average of integer programs (CINT2000) and the average of floating-point programs (CFP2000).

shows the measurements taken on the data access on a load-store architecture using our benchmark programs. We look at branch offsets in Section A.6—data accessing patterns and branches are different; little is gained by combining them, although in practice the immediate sizes are made the same for simplicity.

Immediate or Literal Addressing Mode

Immediates can be used in arithmetic operations, in comparisons (primarily for branches), and in moves where a constant is wanted in a register. The last case occurs for constants written in the code—which tend to be small—and for address constants, which tend to be large. For the use of immediates it is important to know whether they need to be supported for all operations or for only a subset. Figure A.9 shows the frequency of immediates for the general classes of integer and floating-point operations in an instruction set.

Another important instruction set measurement is the range of values for immediates. Like displacement values, the size of immediate values affects instruction length. As Figure A.10 shows, small immediate values are most heavily used. Large immediates are sometimes used, however, most likely in addressing calculations.

Summary: Memory Addressing

First, because of their popularity, we would expect a new architecture to support at least the following addressing modes: displacement, immediate, and register indirect. Figure A.7 shows that they represent 75%–99% of the addressing modes used

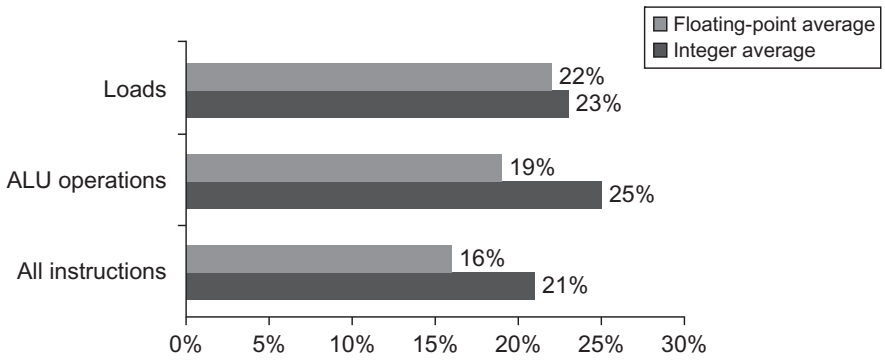


Figure A.9 About one-quarter of data transfers and ALU operations have an immediate operand. The bottom bars show that integer programs use immediates in about one-fifth of the instructions, while floating-point programs use immediates in about one-sixth of the instructions. For loads, the load immediate instruction loads 16 bits into either half of a 32-bit register. Load immediates are not loads in a strict sense because they do not access memory. Occasionally a pair of load immediates is used to load a 32-bit constant, but this is rare. (For ALU operations, shifts by a constant amount are included as operations with immediate operands.) The programs and computer used to collect these statistics are the same as in Figure A.8.

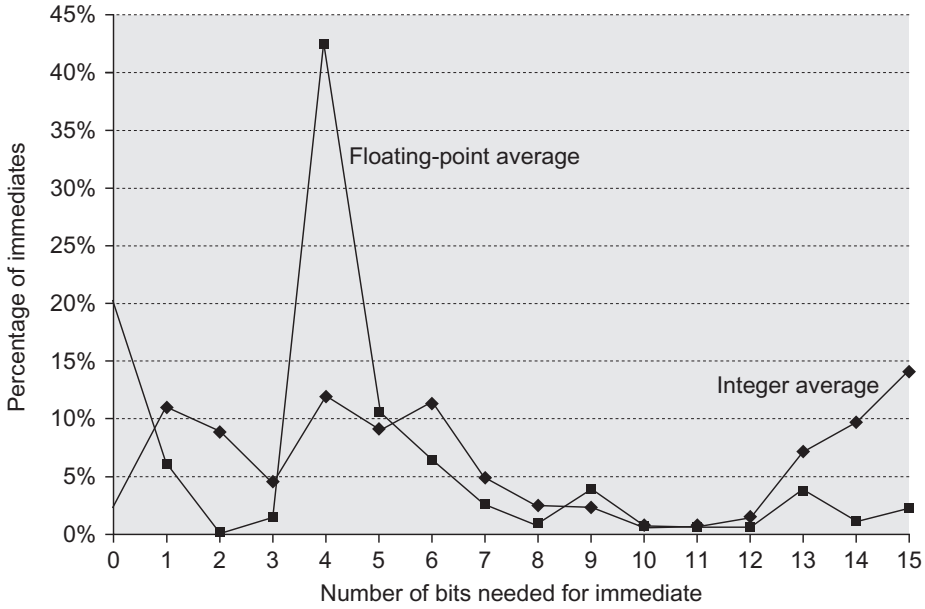


Figure A.10 The distribution of immediate values. The *x*-axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The majority of the immediate values are positive. About 20% were negative for CINT2000, and about 30% were negative for CFP2000. These measurements were taken on an Alpha, where the maximum immediate is 16 bits, for the same programs as in Figure A.8. A similar measurement on the VAX, which supported 32-bit immediates, showed that about 20%–25% of immediates were longer than 16 bits. Thus, 16 bits would capture about 80% and 8 bits about 50%.

in our measurements. Second, we would expect the size of the address for displacement mode to be at least 12–16 bits, because the caption in Figure A.8 suggests these sizes would capture 75%–99% of the displacements. Third, we would expect the size of the immediate field to be at least 8–16 bits. This claim is not substantiated by the caption of the figure to which it refers.

Having covered instruction set classes and decided on register-register architectures, plus the previous recommendations on data addressing modes, we next cover the sizes and meanings of data.

A.4 Type and Size of Operands

How is the type of an operand designated? Usually, encoding in the opcode designates the type of an operand—this is the method used most often. Alternatively, the data can be annotated with tags that are interpreted by the hardware. These tags specify the type of the operand, and the operation is chosen accordingly. Computers with tagged data, however, can only be found in computer museums.

Let's start with desktop and server architectures. Usually the type of an operand—integer, single-precision floating point, character, and so on—effectively

gives its size. Common operand types include character (8 bits), half word (16 bits), word (32 bits), single-precision floating point (also 1 word), and double-precision floating point (2 words). Integers are almost universally represented as two's complement binary numbers. Characters are usually in ASCII, but the 16-bit Unicode (used in Java) is gaining popularity with the internationalization of computers. Until the early 1980s, most computer manufacturers chose their own floating-point representation. Almost all computers since that time follow the same standard for floating point, the IEEE standard 754, although this level of accuracy has recently been abandoned in application-specific processors. The IEEE floating-point standard is discussed in detail in Appendix J.

Some architectures provide operations on character strings, although such operations are usually quite limited and treat each byte in the string as a single character. Typical operations supported on character strings are comparisons and moves.

For business applications, some architectures support a decimal format, usually called *packed decimal* or *binary-coded decimal*—4 bits are used to encode the values 0–9, and 2 decimal digits are packed into each byte. Numeric character strings are sometimes called *unpacked decimal*, and operations—called *packing* and *unpacking*—are usually provided for converting back and forth between them.

One reason to use decimal operands is to get results that exactly match decimal numbers, as some decimal fractions do not have an exact representation in binary. For example, 0.10_{10} is a simple fraction in decimal, but in binary it requires an infinite set of repeating digits: $0.0001100110\overline{0011}_{\ldots 2}$. Thus, calculations that are exact in decimal can be close but inexact in binary, which can be a problem for financial transactions. (See Appendix J to learn more about precise arithmetic.)

The SPEC benchmarks use byte or character, half-word (short integer), word (integer and single precision floating point), double-word (long integer), and floating-point data types. Figure A.11 shows the dynamic distribution of the sizes of objects referenced from memory for these programs. The frequency of access to

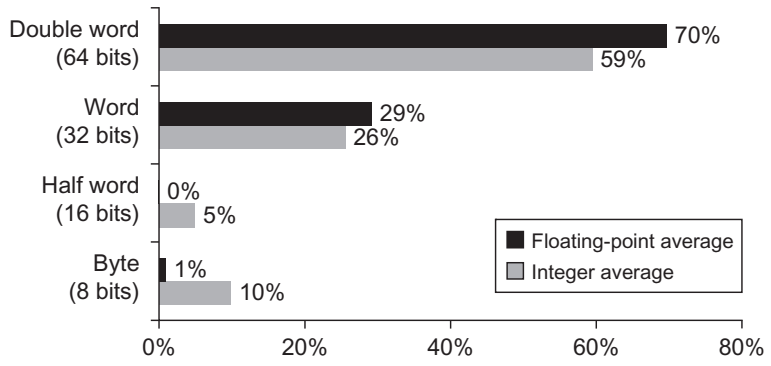


Figure A.11 Distribution of data accesses by size for the benchmark programs. The double-word data type is used for double-precision floating point in floating-point programs and for addresses, because the computer uses 64-bit addresses. On a 32-bit address computer the 64-bit addresses would be replaced by 32-bit addresses, and so almost all double-word accesses in integer programs would become single-word accesses.

different data types helps in deciding what types are most important to support efficiently. Should the computer have a 64-bit access path, or would taking two cycles to access a double word be satisfactory? As we saw earlier, byte accesses require an alignment network: how important is it to support bytes as primitives? Figure A.11 uses memory references to examine the types of data being accessed.

In some architectures, objects in registers may be accessed as bytes or half words. However, such access is very infrequent—on the VAX, it accounts for no more than 12% of register references, or roughly 6% of all operand accesses in these programs.

A.5 Operations in the Instruction Set

The operators supported by most instruction set architectures can be categorized as in Figure A.12. One rule of thumb across all architectures is that the most widely executed instructions are the simple operations of an instruction set. For example, Figure A.13 shows 10 simple instructions that account for 96% of instructions executed for a collection of integer programs running on the popular Intel 80x86. Hence, the implementor of these instructions should be sure to make these fast, as they are the common case.

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

Figure A.12 Categories of instruction operators and examples of each. All computers generally provide a full set of operations for the first three categories. The support for system functions in the instruction set varies widely among architectures, but all computers must have some instruction support for basic system functions. The amount of support in the instruction set for the last four categories may vary from none to an extensive set of special instructions. Floating-point instructions will be provided in any computer that is intended for use in an application that makes much use of floating point. These instructions are sometimes part of an optional instruction set. Decimal and string instructions are sometimes primitives, as in the VAX or the IBM 360, or may be synthesized by the compiler from simpler instructions. Graphics instructions typically operate on many smaller data items in parallel—for example, performing eight 8-bit additions on two 64-bit operands.

Rank	80x86 instruction	Integer average % total executed)
1	Load	22%
2	Conditional branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move register-register	4%
9	Call	1%
10	Return	1%
Total		**96%**

Figure A.13 The top 10 instructions for the 80x86. Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.

As mentioned before, the instructions in Figure A.13 are found in every computer for every application—desktop, server, embedded—with the variations of operations in Figure A.12 largely depending on which data types the instruction set includes.

A.6 Instructions for Control Flow

Because the measurements of branch and jump behavior are fairly independent of other measurements and applications, we now examine the use of control flow instructions, which have little in common with the operations of the previous sections.

There is no consistent terminology for instructions that change the flow of control. In the 1950s they were typically called *transfers*. Beginning in 1960 the name *branch* began to be used. Later, computers introduced additional names. Throughout this book we will use *jump* when the change in control is unconditional and *branch* when the change is conditional.

We can distinguish four different types of control flow change:

- Conditional branches
- Jumps
- Procedure calls
- Procedure returns

We want to know the relative frequency of these events, as each event is different, may use different instructions, and may have different behavior. Figure A.14 shows the frequencies of these control flow instructions for a load-store computer running our benchmarks.

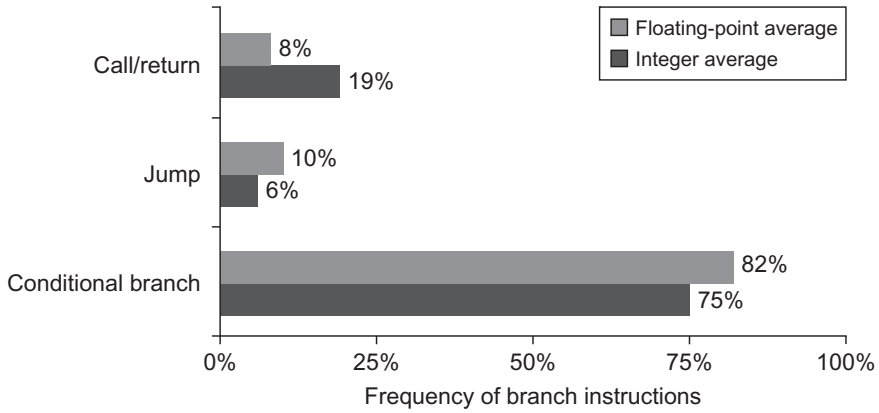


Figure A.14 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches. Conditional branches clearly dominate. Each type is counted in one of three bars. The programs and computer used to collect these statistics are the same as those in Figure A.8.

Addressing Modes for Control Flow Instructions

The destination address of a control flow instruction must always be specified. This destination is specified explicitly in the instruction in the vast majority of cases—procedure return being the major exception, because for return the target is not known at compile time. The most common way to specify the destination is to supply a displacement that is added to the *program counter* (PC). Control flow instructions of this sort are called *PC-relative*. PC-relative branches or jumps are advantageous because the target is often near the current instruction, and specifying the position relative to the current PC requires fewer bits. Using PC-relative addressing also permits the code to run independently of where it is loaded. This property, called *position independence*, can eliminate some work when the program is linked and is also useful in programs linked dynamically during execution.

To implement returns and indirect jumps when the target is not known at compile time, a method other than PC-relative addressing is required. Here, there must be a way to specify the target dynamically, so that it can change at runtime. This dynamic address may be as simple as naming a register that contains the target address; alternatively, the jump may permit any addressing mode to be used to supply the target address.

These register indirect jumps are also useful for four other important features:

- *Case* or *switch* statements, found in most programming languages (which select among one of several alternatives).
- *Virtual functions* or *methods* in object-oriented languages like C++ or Java (which allow different routines to be called depending on the type of the argument).

- *High-order functions* or *function pointers* in languages like C or C++ (which allow functions to be passed as arguments, giving some of the flavor of object-oriented programming).
- *Dynamically shared libraries* (which allow a library to be loaded and linked at runtime only when it is actually invoked by the program rather than loaded and linked statically before the program is run).

In all four cases the target address is not known at compile time, and hence is usually loaded from memory into a register before the register indirect jump.

As branches generally use PC-relative addressing to specify their targets, an important question concerns how far branch targets are from branches. Knowing the distribution of these displacements will help in choosing what branch offsets to support, and thus will affect the instruction length and encoding. Figure A.15 shows the distribution of displacements for PC-relative branches in instructions. About 75% of the branches are in the forward direction.

Conditional Branch Options

Because most changes in control flow are branches, deciding how to specify the branch condition is important. Figure A.16 shows the three primary techniques in use today and their advantages and disadvantages.

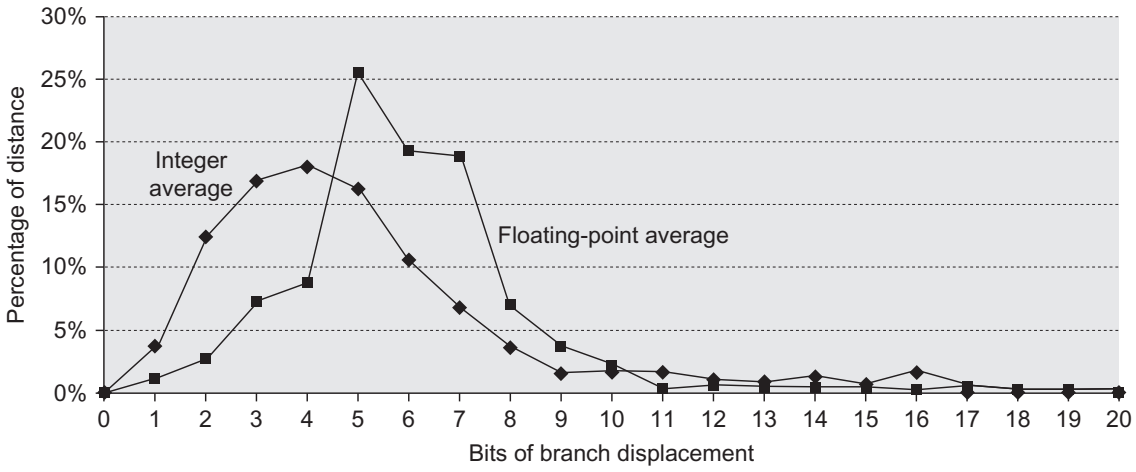


Figure A.15 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in the integer programs are to targets that can be encoded in 4–8 bits. This result tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load-store computer (Alpha architecture) with all instructions aligned on word boundaries. An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. However, the number of bits needed for the displacement may increase if the computer has variable-length instructions to be aligned on any byte boundary. The programs and computer used to collect these statistics are the same as those in Figure A.8.

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructionsbecause they pass information from one instruction to a branch
Condition register/ limited comparison	Alpha, MIPS	Tests arbitrary register with the result of a simple comparison (equality or zero tests)	Simple	Limited compare may affect critical path or require extra comparison for general condition
Compare and branch	PA-RISC, VAX, RISC-V	Compare is part of the branch. Fairly general compares are allowed (greater then, less then)	One instruction rather than two for a branch	May set critical path for branch instructions

Figure A.16 The major methods for evaluating branch conditions, their advantages, and their disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Computers with compare and branch often limit the set of compares and use a separate operation and register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This dichotomy is reasonable because the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

One of the most noticeable properties of branches is that a large number of the comparisons are simple tests, and a large number are comparisons with zero. Thus, some architectures choose to treat these comparisons as special cases, especially if a *compare and branch* instruction is being used. Figure A.17 shows the frequency of different comparisons used for conditional branching.

Procedure Invocation Options

Procedure calls and returns include control transfer and possibly some state saving; at a minimum the return address must be saved somewhere, sometimes in a special link register or just a GPR. Some older architectures provide a mechanism to save many registers, while newer architectures require the compiler to generate stores and loads for each register saved and restored.

There are two basic conventions in use to save registers: either at the call site or inside the procedure being called. *Caller saving* means that the calling procedure must save the registers that it wants preserved for access after the call, and thus the called procedure need not worry about registers. *Callee saving* is the opposite: the called procedure must save the registers it wants to use, leaving the caller unrestrained. There are times when caller save must be used because of access patterns to globally visible variables in two different procedures. For example, suppose we have a procedure P1 that calls procedure P2, and both procedures manipulate the

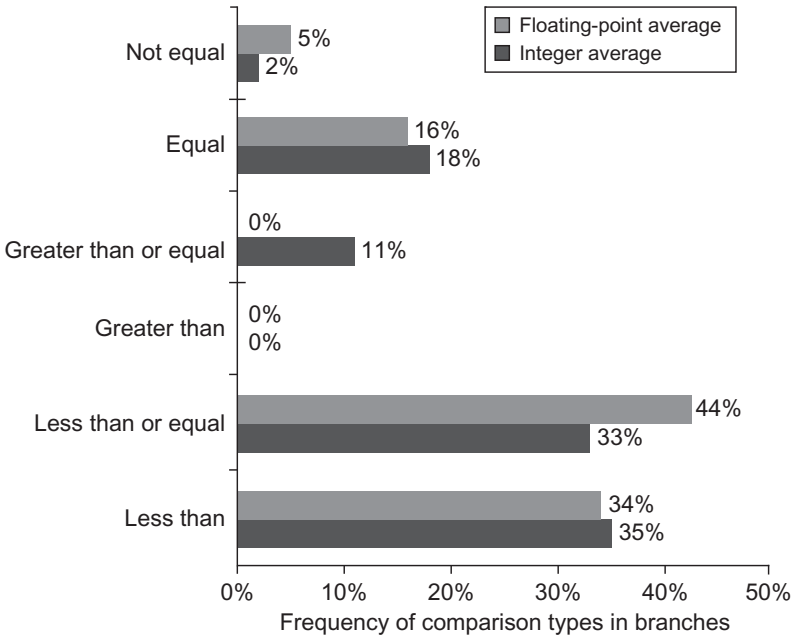


Figure A.17 Frequency of different types of compares in conditional branches. Less than (or equal) branches dominate this combination of compiler and architecture. These measurements include both the integer and floating-point compares in branches. The programs and computer used to collect these statistics are the same as those in Figure A.8.

global variable x. If P1 had allocated x to a register, it must be sure to save x to a location known by P2 before the call to P2. A compiler's ability to discover when a called procedure may access register-allocated quantities is complicated by the possibility of separate compilation. Suppose P2 may not touch x but can call another procedure, P3, that may access x, yet P2 and P3 are compiled separately. Because of these complications, most compilers will conservatively caller save *any* variable that may be accessed during a call.

In the cases where either convention could be used, some programs will be more optimal with callee save and some will be more optimal with caller save. As a result, most real systems today use a combination of the two mechanisms. This convention is specified in an application binary interface (ABI) that sets down the basic rules as to which registers should be caller saved and which should be callee saved. Later in this appendix we will examine the mismatch between sophisticated instructions for automatically saving registers and the needs of the compiler.

Summary: Instructions for Control Flow

Control flow instructions are some of the most frequently executed instructions. Although there are many options for conditional branches, we would expect

branch addressing in a new architecture to be able to jump to hundreds of instructions either above or below the branch. This requirement suggests a PC-relative branch displacement of at least 8 bits. We would also expect to see register indirect and PC-relative addressing for jump instructions to support returns as well as many other features of current systems.

We have now completed our instruction architecture tour at the level seen by an assembly language programmer or compiler writer. We are leaning toward a load-store architecture with displacement, immediate, and register indirect addressing modes. These data are 8-, 16-, 32-, and 64-bit integers and 32- and 64-bit floating-point data. The instructions include simple operations, PC-relative conditional branches, jump and link instructions for procedure call, and register indirect jumps for procedure return (plus a few other uses).

Now we need to select how to represent this architecture in a form that makes it easy for the hardware to execute.

A.7 Encoding an Instruction Set

Clearly, the choices mentioned herein will affect how the instructions are encoded into a binary representation for execution by the processor. This representation affects not only the size of the compiled program but also the implementation of the processor, which must decode this representation to quickly find the operation and its operands. The operation is typically specified in one field, called the *opcode*. As we shall see, the important decision is how to encode the addressing modes with the operations.

This decision depends on the range of addressing modes and the degree of independence between opcodes and modes. Some older computers have one to five operands with 10 addressing modes for each operand (see Figure A.6). For such a large number of combinations, typically a separate *address specifier* is needed for each operand: the address specifier tells what addressing mode is used to access the operand. At the other extreme are load-store computers with only one memory operand and only one or two addressing modes; obviously, in this case, the addressing mode can be encoded as part of the opcode.

When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the register field and addressing mode field may appear many times in a single instruction. In fact, for most instructions many more bits are consumed in encoding addressing modes and register fields than in specifying the opcode. The architect must balance several competing forces when encoding the instruction set:

1. The desire to have as many registers and addressing modes as possible.
2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.

3. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation. (The value of easily decoded instructions is discussed in Appendix C and Chapter 3.) As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary bit length. Many desktop and server architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size.

Figure A.18 shows three popular choices for encoding the instruction set. The first we call *variable*, because it allows virtually all addressing modes to be with all operations. This style is best when there are many addressing modes and operations. The second choice we call *fixed*, because it combines the operation and the addressing mode into the opcode. Often fixed encoding will have only a single

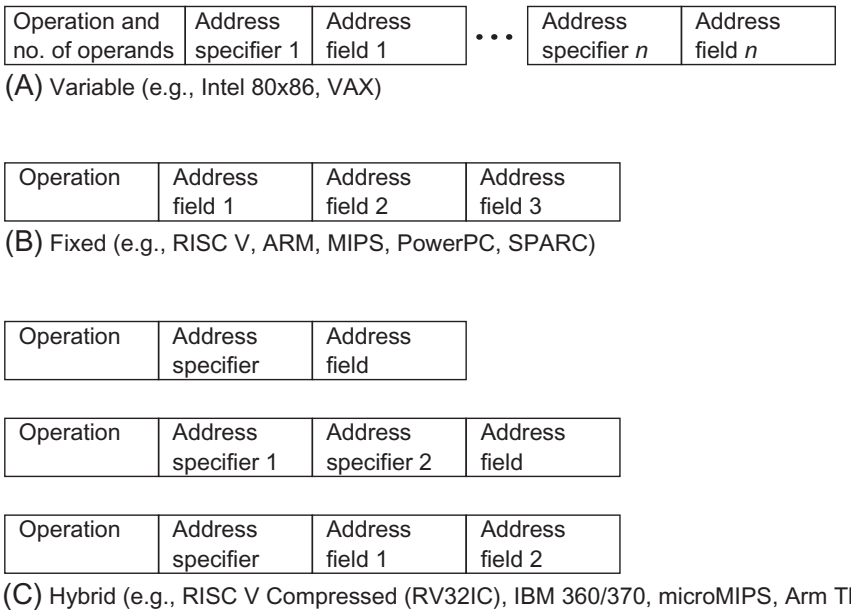


Figure A.18 Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, because unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode. It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.

size for all instructions; it works best when there are few addressing modes and operations. The trade-off between variable encoding and fixed encoding is size of programs versus ease of decoding in the processor. Variable tries to use as few bits as possible to represent the program, but individual instructions can vary widely in both size and the amount of work to be performed.

Let's look at an 80x86 instruction to see an example of the variable encoding:

```
add EAX,1000(EBX)
```

The name `add` means a 32-bit integer add instruction with two operands, and this opcode takes 1 byte. An 80x86 address specifier is 1 or 2 bytes, specifying the source/destination register (`EAX`) and the addressing mode (displacement in this case) and base register (`EBX`) for the second operand. This combination takes 1 byte to specify the operands. When in 32-bit mode (see Appendix K), the size of the address field is either 1 byte or 4 bytes. Because1000 is bigger than 2^8, the total length of the instruction is

$$1 + 1 + 4 = 6 \text{ bytes}$$

The length of 80x86 instructions varies between 1 and 17 bytes. 80x86 programs are generally smaller than the RISC architectures, which use fixed formats (see Appendix K).

Given these two poles of instruction set design of variable and fixed, the third alternative immediately springs to mind: reduce the variability in size and work of the variable architecture but provide multiple instruction lengths to reduce code size. This *hybrid* approach is the third encoding alternative, and we'll see examples shortly.

Reduced Code Size in RISCs

As RISC computers started being used in embedded applications, the 32-bit fixed format became a liability because cost, and hence smaller code, are important. In response, several manufacturers offered a new hybrid version of their RISC instruction sets, with both 16-bit and 32-bit instructions. The narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and the two-address format rather than the classic three-address format of RISC computers. RISC-V offers such an extension, called RV32IC, the C standing for compressed. Common instruction occurrences, such as intermediates with small values and common ALU operations with the source and destination register being identical, are encoded in 16-bit formats. Appendix K gives two other examples, the ARM Thumb and microMIPS, which both claim a code size reduction of up to 40%.

In contrast to these instruction set extensions, IBM simply compresses its standard instruction set and then adds hardware to decompress instructions as they are fetched from memory on an instruction cache miss. Thus, the instruction cache contains full 32-bit instructions, but compressed code is kept in main memory, ROMs, and the disk. The advantage of a compressed format, such as RV32IC,

microMIPS and Thumb2 is that instruction caches act as if they are about 25% larger, while IBM's CodePack means that compilers need not be changed to handle different instruction sets and instruction decoding can remain simple.

CodePack starts with run-length encoding compression on any PowerPC program and then loads the resulting compression tables in a 2 KB table on chip. Hence, every program has its own unique encoding. To handle branches, which are no longer to an aligned word boundary, the PowerPC creates a hash table in memory that maps between compressed and uncompressed addresses. Like a TLB (see Chapter 2), it caches the most recently used address maps to reduce the number of memory accesses. IBM claims an overall performance cost of 10%, resulting in a code size reduction of 35%–40%.

Summary: Encoding an Instruction Set

Decisions made in the components of instruction set design discussed in previous sections determine whether the architect has the choice between variable and fixed instruction encodings. Given the choice, the architect more interested in code size than performance will pick variable encoding, and the one more interested in performance than code size will pick fixed encoding. RISC-V, MIPS, and ARM all have an instruction set extension that uses 16-bit instruction, as well as 32-bit; applications with serious code size constraints can opt to use the 16-bit variant to decrease code size. Appendix E gives 13 examples of the results of architects' choices. In Appendix C and Chapter 3, the impact of variability on performance of the processor will be discussed further.

We have almost finished laying the groundwork for the RISC-V instruction set architecture that will be introduced in Section A.9. Before we do that, however, it will be helpful to take a brief look at compiler technology and its effect on program properties.

A.8 Cross-Cutting Issues: The Role of Compilers

Today almost all programming is done in high-level languages for desktop and server applications. This development means that because most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target. In earlier times for these applications, architectural decisions were often made to ease assembly language programming or for a specific kernel. Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to designing and efficiently implementing an instruction set.

Once it was popular to try to isolate the compiler technology and its effect on hardware performance from the architecture and its performance, just as it was popular to try to separate architecture from its implementation. This separation is essentially impossible with today's desktop compilers and computers. Architectural choices affect the quality of the code that can be generated for a computer and the complexity of building a good compiler for it, for better or for worse.

In this section, we discuss the critical goals in the instruction set primarily from the compiler viewpoint. It starts with a review of the anatomy of current compilers. Next we discuss how compiler technology affects the decisions of the architect, and how the architect can make it hard or easy for the compiler to produce good code. We conclude with a review of compilers and multimedia operations, which unfortunately is a bad example of cooperation between compiler writers and architects.

The Structure of Recent Compilers

To begin, let's look at what optimizing compilers are like today. Figure A.19 shows the structure of recent compilers.

A compiler writer's first goal is correctness—all valid programs must be compiled correctly. The second goal is usually speed of the compiled code. Typically, a whole set of other goals follows these two, including fast compilation, debugging support, and interoperability among languages. Normally, the passes

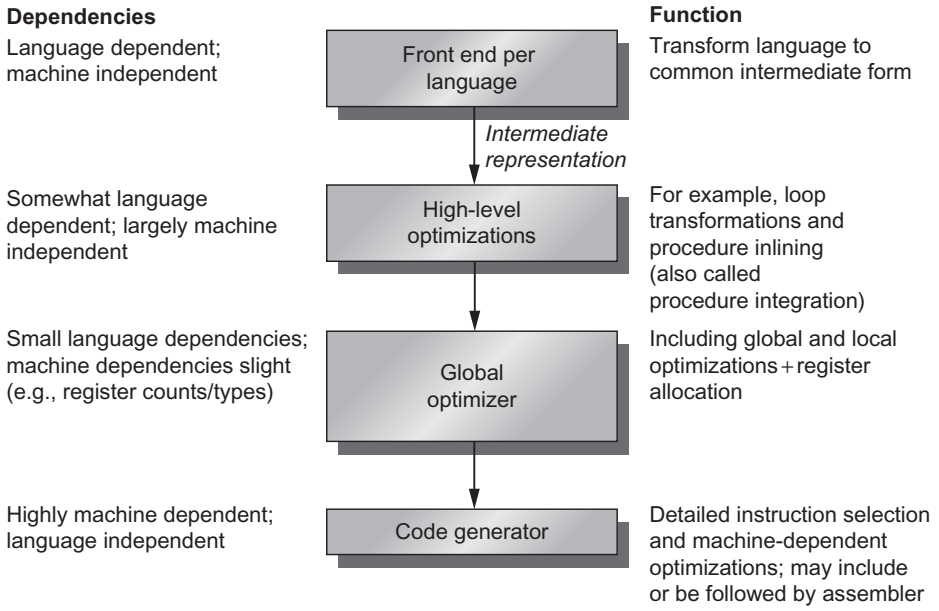


Figure A.19 Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower-quality code is acceptable. A *pass* is simply one phase in which the compiler reads and transforms the entire program. (The term *phase* is often used interchangeably with *pass*.) Because the optimizing passes are separated, multiple languages can use the same optimizing and code generation passes. Only a new front end is required for a new language.

in the compiler transform higher-level, more abstract representations into progressively lower-level representations. Eventually it reaches the instruction set. This structure helps manage the complexity of the transformations and makes writing a bug-free compiler easier.

The complexity of writing a correct compiler is a major limitation on the amount of optimization that can be done. Although the multiple-pass structure helps reduce compiler complexity, it also means that the compiler must order and perform some transformations before others. In the diagram of the optimizing compiler in Figure A.19, we can see that certain high-level optimizations are performed long before it is known what the resulting code will look like. Once such a transformation is made, the compiler can't afford to go back and revisit all steps, possibly undoing transformations. Such iteration would be prohibitive, both in compilation time and in complexity. Thus, compilers make assumptions about the ability of later steps to deal with certain problems. For example, compilers usually have to choose which procedure calls to expand inline before they know the exact size of the procedure being called. Compiler writers call this problem the *phase-ordering problem*.

How does this ordering of transformations interact with the instruction set architecture? A good example occurs with the optimization called *global common subexpression elimination*. This optimization finds two instances of an expression that compute the same value and saves the value of the first computation in a temporary. It then uses the temporary value, eliminating the second computation of the common expression.

For this optimization to be significant, the temporary must be allocated to a register. Otherwise, the cost of storing the temporary in memory and later reloading it may negate the savings gained by not recomputing the expression. There are, in fact, cases where this optimization actually slows down code when the temporary is not register allocated. Phase ordering complicates this problem because register allocation is typically done near the end of the global optimization pass, just before code generation. Thus, an optimizer that performs this optimization must *assume* that the register allocator will allocate the temporary to a register.

Optimizations performed by modern compilers can be classified by the style of the transformation, as follows:

- *High-level optimizations* are often done on the source with output fed to later optimization passes.
- *Local optimizations* optimize code only within a straight-line code fragment (called a *basic block* by compiler people).
- *Global optimizations* extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.
- *Register allocation* associates registers with operands.
- *Processor-dependent optimizations* attempt to take advantage of specific architectural knowledge.

Register Allocation

Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important—if not the most important—of the optimizations. Register allocation algorithms today are based on a technique called *graph coloring*. The basic idea behind graph coloring is to construct a graph representing the possible candidates for allocation to a register and then to use the graph to allocate registers. Roughly speaking, the problem is how to use a limited set of colors so that no two adjacent nodes in a dependency graph have the same color. The emphasis in the approach is to achieve 100% register allocation of active variables. The problem of coloring a graph in general can take exponential time as a function of the size of the graph (NP-complete). There are heuristic algorithms, however, that work well in practice, yielding close allocations that run in near-linear time.

Graph coloring works best when there are at least 16 (and preferably more) general-purpose registers available for global allocation for integer variables and additional registers for floating point. Unfortunately, graph coloring does not work very well when the number of registers is small because the heuristic algorithms for coloring the graph are likely to fail.

Impact of Optimizations on Performance

It is sometimes difficult to separate some of the simpler optimizations—local and processor-dependent optimizations—from transformations done in the code generator. Examples of typical optimizations are given in Figure A.20. The last column of Figure A.20 indicates the frequency with which the listed optimizing transforms were applied to the source program.

Figure A.21 shows the effect of various optimizations on instructions executed for two programs. In this case, optimized programs executed roughly 25%–90% fewer instructions than unoptimized programs. The figure illustrates the importance of looking at optimized code before suggesting new instruction set features, because a compiler might completely remove the instructions the architect was trying to improve.

The Impact of Compiler Technology on the Architect's Decisions

The interaction of compilers and high-level languages significantly affects how programs use an instruction set architecture. There are two important questions: how are variables allocated and addressed? How many registers are needed to allocate variables appropriately? To address these questions, we must look at the three separate areas in which current high-level languages allocate their data:

Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	*At or near the source level; processor-independent*	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., $A = X$) with X	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	2%
Processor-dependent	*Depends on processor knowledge*	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

Figure A.20 Major types of optimizations and examples in each class. These data tell us about the relative frequency of occurrence of various optimizations. The third column lists the static frequency with which some of the common optimizations are applied in a set of 12 small Fortran and Pascal programs. There are nine local and global optimizations done by the compiler included in the measurement. Six of these optimizations are covered in the figure, and the remaining three account for 18% of the total static occurrences. The abbreviation *N.M.* means that the number of occurrences of that optimization was not measured. Processor-dependent optimizations are usually done in a code generator, and none of those was measured in this experiment. The percentage is the portion of the static optimizations that are of the specified type. Data from Chow, F.C., 1983. A Portable Machine-Independent Global Optimizer—Design and Measurements (Ph.D. thesis). Stanford University, Palo Alto, CA (collected using the Stanford UCODE compiler).

- The *stack* is used to allocate local variables. The stack is grown or shrunk on procedure call or return, respectively. Objects on the stack are addressed relative to the stack pointer and are primarily scalars (single variables) rather than arrays. The stack is used for activation records, *not* as a stack for evaluating expressions. Hence, values are almost never pushed or popped on the stack.

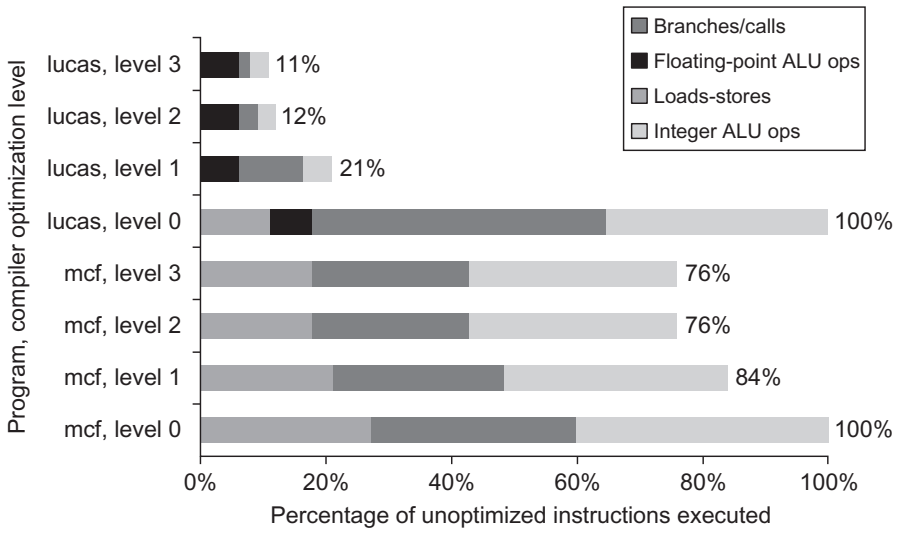


Figure A.21 Change in instruction count for the programs lucas and mcf from the SPEC2000 as compiler optimization levels vary. Level 0 is the same as unoptimized code. Level 1 includes local optimizations, code scheduling, and local register allocation. Level 2 includes global optimizations, loop transformations (software pipelining), and global register allocation. Level 3 adds procedure integration. These experiments were performed on Alpha compilers.

- The *global data area* is used to allocate statically declared objects, such as global variables and constants. A large percentage of these objects are arrays or other aggregate data structures.
- The *heap* is used to allocate dynamic objects that do not adhere to a stack discipline. Objects in the heap are accessed with pointers and are typically not scalars.

Register allocation is much more effective for stack-allocated objects than for global variables, and register allocation is essentially impossible for heap-allocated objects because they are accessed with pointers. Global variables and some stack variables are impossible to allocate because they are *aliased*—there are multiple ways to refer to the address of a variable, making it illegal to put it into a register. (Most heap variables are effectively aliased for today's compiler technology.)

For example, consider the following code sequence, where `&` returns the address of a variable and `*` dereferences a pointer:

```
p =&a  - gets address of a in p
a =...  - assigns to a directly
*p =...  - uses p to assign to a
...a...  - accesses a
```

The variable `a` could not be register allocated across the assignment to `*p` without generating incorrect code. Aliasing causes a substantial problem because it is